UNIUNEA EUROPEANĂ GUVERNUL ROMÂNIEI Fondul Social European POSDRU 2007-2013 Instrumente Structurale 2007-2013 MINISTERUL EDUCAȚIEI NAȚIONALE OIPOSDRU Universitatea Politehnica din Bucuresti

Proiect KNOWLEDGE - POSDRU/159/1.5/S/134398

Dezvoltarea resurselor umane din cercetarea doctorală și postdoctorală: motor al societății bazate pe cunoaștere

UNIVERSITATEA **POLITEHNICA** DIN BUCUREŞTI

Facultatea Automatică si Calculatoare

Departamentul de Calculatoare

Nr. Decizie Senat 237 din 02.09.2015

TEZĂ DE DOCTORAT

Redarea de scene masive 3D in timp real

Rendering massive 3D scenes in real-time

Autor: Ing. Alexandru Lucian Petrescu

Conducător de doctorat: Prof. dr. ing. Florica Moldoveanu

COMISIA DE DOCTORAT

Preşedinte	Prof. dr. ing. Adina Magda Florea	de la	Universitatea POLITEHNICA din Bucureşti
Conducător de doctorat	Prof. dr. ing. Florica Moldoveanu	de la	Universitatea POLITEHNICA din Bucureşti
Referent	Profesor dr. ing. Vasile Manta	de la	Universitatea Tehnică „Gheorghe Asachi” din Iași
Referent	Profesor dr. ing. Ștefan Pentiuc	de la	Universitatea „Ștefan cel Mare” din Suceava
Referent	Conferențiar dr. ing. Irina Mocanu	de la	Universitatea POLITEHNICA din Bucureşti

Bucureşti

RENDERING MASSIVE SCENES IN REAL-TIME

A Dissertation

Submitted to the Faculty of Automatic Control and Computers

of

University POLITEHNICA of Bucharest

by

Alexandru Lucian Petrescu

In Partial Fulfillment of the

Requirements for the Degree

Of

Doctor in Computer Science

September 2015

University POLITEHNICA of Bucharest

Romania

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am grateful for the help and guidance of my advisor, Prof. Dr. Florica Moldoveanu. Under her supervision I increased my understanding of computer graphics beyond the topics of real-time rendering, I had many interesting theoretical and practical research opportunities and I learned how to properly do research.

I am also grateful for the research insights, opportunities and guidance offered by Prof. Dr. Michael Wimmer and Prof. Dr. Werner Purgathofer during my internship at Vienna University of Technology.

I would like to thank my colleagues Dr. Ing. Alexandru Egner, Msc. Ing. Mihai Francu, Conf. Dr. Ing. Alin Moldoveanu and Dr. Ing. Anca Morar for many interesting research discussions, for their support and their useful suggestions. I am particularly grateful to my colleague, Dr. Ing. Victor Asavei, as he introduced me to computer graphics and his recommendations, help and counsel were extremely helpful during my doctoral studies.

Many thanks go to Horea Caramizaru, for the numerous discussions about algorithms, complexities and acceleration structures and for helping me improve my knowledge of how to match different computer science methods to practical rendering problems.

Special thanks go to Károly Zsolnai for introducing me to the world of global illumination, and for recommending me a large number of influential scientific works that shaped this field.

The 3D models used in the images generated for this thesis are freely available at the Stanford 3D scanning repository, the Google 3D Warehouse, AIM@Shape, the Utah 3D animation repository, Crytek and at the McGuire Graphics Data Archive.

I am most grateful to my family, for their complete support, understanding and love, without them this would have been impossible.

FINANCIAL ACKNOWLEDGEMENTS

The work has been funded by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Ministry of European Funds through the Financial Agreement POSDRU/159/1.5/S/134398."

ABSTRACT:

In this thesis a novel rendering pipeline is presented, which can handle a large number of light paths in real-time, using innovative techniques that decouple the rendering process into several modular and memory efficient components. Several contributions are presented and compared to state of the art algorithms, which are used or can be used in the context of real-time rendering. This original rendering pipeline can render both high quality images in real-time and correct, photorealistic images offline. The contributions of the thesis are categorized into geometry rendering methods and illumination methods.

The geometry rendering algorithms modify state of the art rasterization methods to better handle large scenes by decoupling non geometric bandwidth, originating from texture data, from geometric computations. This drastically reduces the bandwidth consumption and is implemented for both opaque and transparent objects, in novel algorithms named virtual deferred and virtual order independent transparency. These rendering methods are directly integrated into a virtual texturing system. Opaque rasterized objects are antialiasing with a novel antialiasing method for deferred rendering, which improves upon the state of the art sub geometric reconstruction antialiasing methods. Transparent objects are shaded and textured adaptively.

Moreover, all geometric rendering is culled with a novel hierarchical method, which culls objects for more than a single frame. The culling algorithm uses GPU task generation to walk the scene tree and hierarchical impostors to lessen geometric aliasing. Other geometric rendering contributions include a distribution based variant of occupancy maps, which adaptively samples the occupancy, measurement metrics for deferred and decoupled algorithms, a variant of the marching cubes algorithm designed for massive datasets and a new geometry selection method.

This thesis introduces both exact and approximative illumination methods. The approximative methods decouple the low and high frequency transport of light, handling each type differently. Conservative Inexact Voxelization is a new imperfect voxelization method, with a construction complexity much lower than the state of art methods, and benefits from deferred geometry information. This voxelization method is used to relax the visibility operator in the rendering equation, acting as a visibility determination structure for virtual lights based illumination. The visibility operator for tracing inside the introduced voxelization structure is perception adaptive, being almost accurate inside the visualization volume and coarse outside of it.

Novel algorithms are presented for rendering solutions which are only at the edge of interactivity as of today, but which will be used on consumer hardware for real-time rendering in the future. The correct rendering pipeline generates images with a bidirectional path tracer, which uses a novel type of importance sampling. The light flux importance sampling introduces a map, which can quickly link unproductive paths to light vertices generated by light tracing. It is a global sampling method which is much more exact than skeleton importance sampling, the only other global importance sampling method for path tracing. From a computational standpoint, light flux importance sampling is also cheaper.

Keywords: real-time rendering, decoupled rendering, bandwidth reduction, complex scenes, global illumination

LIST OF PUBLICATIONS AND RESEARCH PROJECTS:

Lucian Petrescu, Anca Morar, Florica Moldoveanu, Victor Asavei, "Real time reconstruction of volumes from very large datasets using CUDA", in the 15th International Conference on System Theory, Control, and Computing (ICSTCC), 14-16 October, pp 1-5, ISBN: 978-1-4577-1173-2, 2011

Anca Morar, Florica Moldoveanu, Victor Asavei, Lucian Petrescu, Alin Moldoveanu, Alexandru Egner, "GPGPU Based Non-photorealistic Rendering of Volume Data", in Control Engineering and Applied Informatics (CEAI), vol.15, no. 1, pp. 45-52, 2013

Lucian Petrescu, Moldoveanu Florica, Moldoveanu Alin, Morar Anca, Asavei Victor, "Efficient Picking Through Atomic Operations", 19th International Conference on Control Systems and Computer Science (CSCS), pp 66-70, ISBN: 978-1-4673-6140-8, 29-31 May 2013 2013

Lucian Petrescu, Moldoveanu Florica, Victor Asavei, Moldoveanu Alin, Oana Ferche, "A GPU Task Generator for Rendering" in ICSTCC 2014 - 18th International Conference On System Theory, Control and Computing, pp. 562-567, ISBN: 978-1-4799-4602-0, October, 2014.

Lucian Petrescu, Florica Moldoveanu, Victor Asavei, Alin Moldoveanu, "Analyzing Deferred Rendering Techniques", in Control Engineering and Applied Informatics (CEAI), accepted, to be published.

Lucian Petrescu, Florica Moldoveanu, Victor Asavei, Alin Moldoveanu, "Virtual deferred rendering", in 20th International Conference on Control Systems and Computer Science (CSCS), pp 373-378, ISBN 978-1-4799-1780-8, DOI 10.1109, 27-29 May, 2015.

Lucian Petrescu, Florica Moldoveanu, Victor Asavei, Alin Moldoveanu, "Guarded Order Independent Transparency", The Scientific Bulletin of University POLITEHNICA of Bucharest, Series C, Electrical Engineering and Computer Science, pp 3-14, Vol. 77, Iss. 1, ISSN 2286-3540, April, 2015.

Personalized implants for hip arthroplasty (SABIMAS, PNCDII-Joint Applied Research Projects, 2008-2011), http://se.cs.pub.ro/SABIMAS/

SYMBOLS AND ACRONYMS

A – area
bpp – bits per pixel
Bpp – bytes per pixel
d – diffuse reflection, in Heckbert notation
e – eye, camera, in Heckbert notation
E – irradiance
f_r – bidirectional reflectance distribution function
g – glossy reflection in Heckbert notation
l – light(s), in Heckbert notation
L – radiance
λ – wavelength
n – normal
Ω – hemisphere
o – object(s)
Φ – radiant flux or power
P – position
pdf – probability distribution function
ps – photo (pixel) sensor
px – pixel
Q – radiant energy
Ω – hemisphere
s – specular reflection, in Heckbert notation
spp – sample per pixel
t – time
x – position
ω – direction
A-Buffer – antialiased area averaged accumulation buffer
AABB – axis aligned bounding box
AGAA – aggregate G-buffer antialiasing
AGS – atomic geometry selection
AO – ambient occlusion
ASM – adaptive shadow maps
ASSM – adaptive soft shadow mapping
ASSAO – alchemy screen space ambient occlusion
AVSM – adaptive volumetric shadow maps
b-SAH – binned/bucketed surface area heuristic
BDPM – bidirectional photon mapping
BDPT – bidirectional path tracing
BDRT – bidirectional ray tracing
BIH – bounding interval hierarchy
BIR – bidirectional instant radiosity
BRDF – bidirectional reflectance distribution function
BRT – backwards (camera) ray tracing
BSDF – bidirectional scattering distribution function

BSP – binary space partitioning
BSSRDF – bidirectional scattering surface reflectance distribution function
BTDF – bidirectional transmission distribution function
BV – bounding volume
BVH – bounding volume hierarchy
C&PD – clipping and perspective divide
CHC – coherent hierarchical culling
CIV – conservative inexact voxelization
CPU – central processing unit
CRT – camera (backwards) ray tracing
CS – compute shader
CSAA – coverage sampling anti-aliasing (alternative EQAA)
CSM – cascaded | convolution shadow maps
CSSM – camera space shadow maps
DSRAA – decoupled Subpixel reconstruction antialiasing
DACRT – divide and conquer ray tracing
DEAA – distance to edge antialiasing
DLAA – directionally localized anti-aliasing
DOF – depth of field
DOM – deep opacity maps
DOP – discrete oriented polytopes
DRT – distributed/distribution/stochastic ray tracing
DR – decouple rendering
DS – domain shader
DDS – deep deferred shading
ERPT – energy redistribution path tracing
EQAA – enhanced quality antialiasing
ESC – early split culling (BVH)
EVH – early volume heuristic (BVH)
ESM – exponential shadow maps
EWA – elliptical weighted average
FFAO – far field ambient occlusion
FPS – frames per second
FOM – Fourier opacity map
FRT – forward (light) ray tracing
FS – fragment shader
FSAA – full screen anti-aliasing
FU – GPU functional unit
FXAA – fast approximate anti-aliasing
G-Buffer – geometry buffer
GBAA – geometry buffer anti-aliasing
GI – global illumination
GPU – graphics processing unit
GPGPU – general purpose computation on graphics processing units
GS – geometry shader
HBAO – horizon based ambient occlusion
HDR – high dynamic range
HIZC – hierarchical Z culling

HRBVH – hybrid rasterized bounding volume hierarchies
HRT – Heckbert ray tracing
HS – hull shader
IA – input assembler
IBL – image based lighting
IR – instant radiosity
IS – importance sampling
QBVH – quad bounding volume hierarchy
LEAN – linear efficient antialiased normal (mapping)
LDR – low dynamic range
LFIS – light flux/flow importance sampling
LFM – light flux map
LOD – level of detail
LSAO – line sweep ambient occlusion
LSM – logarithmic shadow maps
LRT – light (forward) ray tracing
LSPSM – light space perspective shadow maps
MB – motion blur
MBR – maximum bounding rectangles
MBVH – multiple bounding volume hierarchies
MC – Monte Carlo
MCMC – Markov Chain Monte Carlo
MCRT – Monte Carlo ray tracing
MIR – metropolis instant radiosity
MIS – multiple importance sampling
MLAA – morphological anti-aliasing
MSAA – multisample anti aliasing
MLT – Metropolis light tracing
MVAOIS – multi-view ambient occlusion with importance sampling
NFAA – normal filter anti-aliasing
NOHC – near optimal hierarchical culling
OIT – order independent transparency
OM – output merger
OOBB – object oriented bounding box
OSM – opacity shadow maps
PA – primitive assembler
PB – photon beams
PBR – point based rendering
PCSS – percentage closer soft shadow
PM – photon mapping
POM – parallax occlusion mapping
PPB – progressive photon beams
PPM – progressive photon mapping
PRT – packet ray tracing
PS – pixel shader
PSM – perspective shadow maps
PSSM – parallel split shadow maps
PSSMLT – primary sample space Metropolis light transport

PT – path tracing
PVS – potentially visible sets
PWAA – phone wire antialiasing
R – rasterizer
RBVH – rasterized bounding volume hierarchies
RC – (volume) ray casting
RCSM – relaxed cone step mapping
RGSM – Reconstructable geometry shadow mapping
RIS – resampled importance sampling
RR – Russian roulette
RSAA – resampling anti-aliasing
RSM – reflective shadow map
RTW – rectilinear texture warping
SAH – surface area heuristics (KD, BVH)
SAO – scalable ambient occlusion
SAT – summed area tables
SBAA – surface based anti-aliasing
SDSM – sample distribution shadow maps
SH – spherical harmonics
SIMD – single instruction multiple data
SIMT – single instruction multiple thread
SM – Shader Model
SMP – Streaming Multiprocessor
SMAA – sub pixel morphological antialiasing
SO – stream output
SPPM – stochastic progressive photon mapping
SPSM – sub-pixel shadow mapping
SRAA – Subpixel reconstruction anti-aliasing
SRAD – sub pixel reconstruction antialiasing for deferred rendering
SS – screen space
SSAA – supersampling antialiasing (FSAA)
SSAO – screen space ambient occlusion
SSBC – screen space bent cones
SSCT – screen space cone tracing
SSDO – screen space directional occlusion
SSDM – screen space displacement mapping
SSGI – screen space global illumination
SSLR – screen space local reflections
SSPCSS – screen space percentage closer soft shadows
SSPM – screen space photon mapping
SSRT – screen space ray tracing
SSS – sub surface scattering
SSSSAA – screen space super sampling antialiasing
T - tessellator
TCS – tessellation control shader
TES – tessellation evaluation shader
TF – transform feedback
TSM – trapezoidal shadow maps

TXAA – temporal antialiasing
VAL – virtual area lights
VBL – virtual beam light
VD – virtual deferred
VL – virtual light
VA-Buffer – virtual A-Buffer (VOIT)
VOIT – virtual order independent transparency (VA-Buffer)
VOSS – volumetric obscurance in screen space
VPL – virtual point light
VRC – volume ray casting
VRL – virtual ray light
VS – vertex shader
VSL – virtual spherical light
VSM – variance shadow maps
VT – virtual texturing
WRT – Whitted ray tracing

1. INTRODUCTION

1.1. Real-time Rendering

Computer graphics and vision is an established field in computer science. Its primary goal is the analysis, synthesis and manipulation of visual data, focusing on mathematical and computational aspects rather than aesthetics. It is used in a multitude of applications in areas such as medicine, surveillance, architectural design, recreational activities, video processing, and remote presence systems and data analysis.

The main objective of computer vision, also known as visualization, is to present predefined data in a manner which emphasizes properties of interest, such as: trends in data mining, tissue appearance in medical applications or tensions in structural components in engineering stress simulations. Computer graphics, commonly known as rendering, is primarily involved in the efficient generation of images without emphasizing content, with the ultimate goal of photorealism. Among its applications are special effects for movies, architectural visualizations and procedural generation.

Real-time rendering is a specialized topic in the field of computer graphics. It attempts to recreate the same visual results as general computer graphics methods, but does so prioritizing interactivity, with the final purpose of maintaining the illusion of continuous virtual reality. Humans visually perceive their surroundings as a sequence of images which are unconsciously reconstructed by the brain into motion [**Wat13**] [**Hum09**] . In order for an image based process to properly create a convincing illusion of reality, the number of generated frames per second has to be sufficiently high. Because of this, real-time rendering algorithms usually generate more than 30 frames per second (FPS). The FPS rate is generally higher, usually greater than the refresh rate of the output monitor. Compared to the long processing durations of time unbounded rendering, real-time rendering has to produce photorealistic images on a much lower computational budget, therefore algorithm design, optimization and hardware utilization maximization represent key subjects in this specialized topic. Real-time rendering has a large number of practical applications, especially in performance critical fields such: aviation and military simulators, video games or previsualisations.

Besides needing to render the scene a high number of times per second, real-time rendering must also be stable. If some of the rendered frames take much longer than the others to finish, then the illusion of continuous virtual reality is ruined, even if the average frame time stays within the expected budget.

Massive scenes, also named multi-scale scenes contain many objects of unevenly spatially distributed objects, of varying sizes and properties. Such scenes are very common in rendering because they depict regular types of vistas, like the natural or urban views rendered in Figure 1.

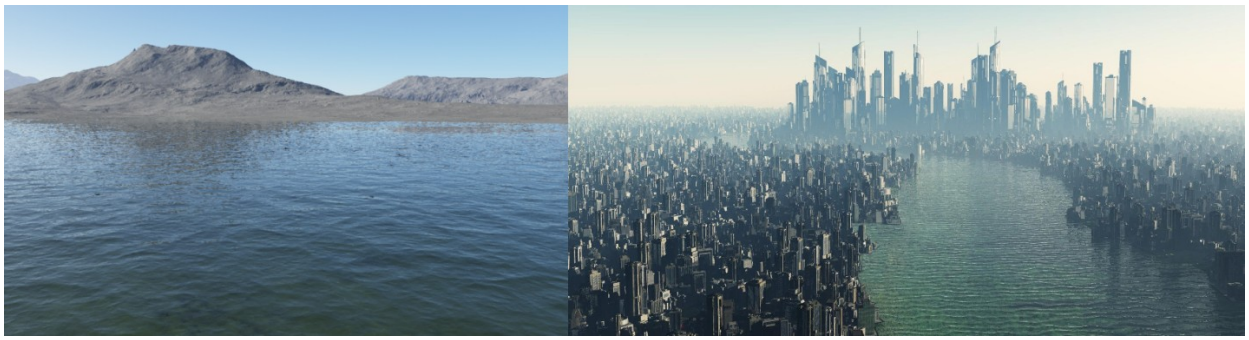

Figure 1 Massive scenes**.** These images show renderings of massive scenes, also called multi-scale scenes. These scenes contain a large number of unpredictably positioned objects. Rendering such scenes in real time is very difficult, especially if they contain a large number of dynamic objects. The purpose of this thesis is to introduce new algorithms, which are better suited to rendering such scenes in real-time. The image on the right was rendered with Blender **[**Ble15**]**.

Massive scenes contain a great variety of content, for example, the left image of Figure 1 contains water in the foreground and the mountains with all the crevices in the background. The water ripple details are measured in millimeters, while the mountains are measured in kilometers. This is a common example for the great disparity in the scale of the objects. Massive scenes are usually so large that the assets can't be loaded entirely into GPU memory. Out of core algorithms are required to constantly stream the assets pertaining to the objects and lights, which are implicated in the generation of the final image. This process is repeated per frame, independently of the chosen rendering pipeline. The streamed data can be represented in a compressed or not native format, thus it might need to be processed or reconstructed before rendering.

Because of the extreme size of the represented scenes, their storage and representation into memory is an important subject. In rendering, objects can be represented through polygons and textures, voxels, bounding trees or analytically. Data is frequently transformed into rendering compatible data, such as triangles, sparse voxel structures or intersection trees. Because many important rendering algorithms use special data formats, data that does not match a certain representation has to be converted. This process is extremely costly and is mostly performed outside real-time rendering. It can be performed interactively [**Che13**] [**Lau09**] or even in real-time [**Eis08**], but with steep computational costs which would leave little resources for performing effective rendering.

There are many types of base algorithms used in rendering, but few are truly applicable in real time rendering of multi-scale scenes. In general, rasterization is used for real time rendering of massive scenes. Hardware rasterization is used in all consumer graphic card rendering pipelines [**Seg15**]. It only solves the problem of camera and scene intersection, without covering lights-objects interactions, global illumination (GI), analytic anti-aliasing or analytic surface representation. Inspired by Reyes rendering [**Coo87**], micropolygon rendering [**Tat10**] [**Eis10**] is a special case of rasterization that adds analytic anti-aliasing and analytic surface representation capabilities to standard rasterization. Stochastic rasterization [**Mcg10**] enhances the basic rasterization pipeline with the abilities to handle motion and defocus blur. Both stochastic rasterization and micropolygon rendering dramatically increase the quality of rasterization rendering, but their costs are relatively steep for current consumer hardware.

Even if real-time rasterization based algorithms are vastly inferior from a quality standpoint when compared to time unbounded global illumination techniques, the computational

cost of complete global illumination algorithms is still too high for consumer hardware. Solutions such as stochastic ray tracing [**Bik07**], path tracing [**Bik13**] [**van11**], beam tracing [**Cra09**], photon mapping [**Mar13**], progressive light-cuts [**Dav12**] or other many-light global illumination solutions [**Dac14**] are getting closer to real-time, but they do so with severe performance requirements. They also have increased costs in scenes with dynamic objects, caused by the long update times in intersection acceleration structures.

The previously mentioned global illumination algorithms are separable. Programmable multisampled rasterization generates exactly the same lights-objects interactions as the set of first rays in ray tracing, the first beams of beam tracing or as the first set of path segments in path tracing. Because of this, the majority of high performance real-time rendering is performed at least partially with rasterization based techniques. The rasterization rendering stage is then followed by a global illumination stage, which interprets the results of the rasterization stage as a first step in the global illumination method.

This decoupling permits different sampling and filtering rates in these stages, thus each stage possesses different aliasing levels. If the entire rendering process, containing both the rasterization stage and the GI stage, is considered a sampling process, by using different sampling rates for the different stages the entire process is effectively importance sampled. The human eye can easily detect even very low differences in spatial representations, because contours and edges are extremely important in visual perception [**Sha73**], therefore the aliasing from the rasterization stage is much more important to mitigate than the aliasing from the global illumination stage. This perception difference is a significant advantage, because rasterization algorithms are significantly less demanding than GI algorithms from a computational standpoint. Rendering the global illumination stage at a lower sampling rate can be done without dramatically decreasing the perceptual quality of the rendered image.

1.2. Motivation and Objectives

The main objective of this thesis is to introduce a novel rasterization based rendering pipeline for massive scenes, which minimizes storage and bandwidth consumption, provides stable runtime performance and decouples the majority of the involved algorithms.

Because real-time rendering is a relatively novel research field and due to the tremendous growth in hardware capabilities in the last decade there are still many problems lacking acceptable answers and many algorithms that can be optimized or modified to efficiently run on the GPU. Even though the rendering equation is separable [**Kaj86**], rendering algorithms are usually monolithic and suffer from high coupling, therefore they can be hard to maintain and even harder to properly analyze. Finding new methods to decouple sub processes and sub stages of rendering methods can offer new insights into algorithm design and performance tradeoffs. **High coupling and high maintenance** costs represent one of the motivations of this thesis, and that is to find new ways to increase the degree of modularity between the many components involved in generating the final image in rendering.

Increasing the speed of the rendering process is not sufficient if the variance in frame duration is too large. The **frame rate stability** of the rendering is just as important as its speed. Performance spikes are frequently caused by long operations with irregular occurring patterns such as data streaming and conversion. Developing faster data reconstruction algorithms offers

the incentive of decreased frame duration variance, by minimizing the costs of data conversion into rendering compatible formats. Stability is one of the most important properties of real-time rendering. If the frame time variance is sufficiently high, then the entire purpose of real-time rendering is unsatisfied. In order to maximize stability, the causes of the performance fluctuations have to be easy to detect. Furthermore, the causes have to be sufficiently decoupled in order for such a measuring process to be applicable. Among the objectives of this thesis is the introduction of measuring metrics for the rasterization based rendering techniques.

Storage space, access efficiency and bandwidth consumption are in general the most performance critical aspects of real-time rendering. Since the processing power of GPUs is increasing at a rate higher than that of their memory, the importance of memory usage efficiency will continue to increase. Storage space is a serious problem especially with stochastic algorithms like those involved in OIT [**Jan10**] or stochastic rasterization [**Mcg10**], where a large number of samples is needed for the stochastic process to reach an acceptable error level. The same problem appears in global illumination algorithms [**Bik07**] [**van11**] and in many types of deep deferred rendering techniques [**Mar14**] [**Bar11**]. One objective of this thesis is to provide improved algorithms that minimize memory storage for opaque and transparent objects in rasterization rendering.

Rasterization is almost always involved in the real-time rendering process, because of its cache oriented primitive intersection acceleration structure, a bidimensional grid called raster. Because of its nature, simple rasterization rendering is very limited in producing all light-object-camera interactions, and usually represents only a part of an advanced rendering pipeline. Rendering opaque objects with rasterization can solve occlusion with the help of the z-buffer algorithm. The z-buffer algorithm has been improved since its introduction through [**Gre93**] [**Joh05**]. The A-buffer algorithm [**Car84**] was the first to consider globally handling multiple objects intersected per pixel, decoupling fragment ordering from fragment shading. Partially inspired by the A-buffer, the deferred rendering algorithms family [**Dee88**] [**Ols11**] [**Ols12**] is the dominant choice in rasterization based rendering of opaque objects. Deferred algorithms are so successful because they decouple light-object intersection processing from visibility determination, dramatically decreasing the computational costs of rendering.

Decoupled algorithms [**Rag11**] [**Lik12**] [**Bur13**] are advanced types of deferred techniques, which further decouple the sub-stages of the rendering. Virtual texturing [**Eph06**] improves the performance of rendering by streaming only the obligatory information for the rendered frame. It does this by decoupling direct texture access and filtering through a system of virtual paging, inspired from operating systems paging solutions. Although there has been a wealth of research on the topics of rasterization based opaque object rendering and stage separation, there are still opportunities to further improve **resource re-usage** and to **decouple the algorithms** used in state of the art rendering pipelines.

Real-time rendering is deployed to multiple platforms frequently, therefore hardware variation is usually expected. Because of this variation, not all deployment platforms offer the best performance with the same rendering algorithm. Furthermore, power variation on the same platform can have the same effect, especially in power critical environments. Measuring performance in real-time and consequently adapting the rendering path has proven to be a complicated task, because of the many different potential performance bottlenecks. There is a clear motivation for the definition of **advanced metrics** that would allow separate analysis of

these bottlenecks, and which would provide unbiased performance information on the rendering pipeline.

Rendering can be a considered a large sampling process, in which the objects and lights of the scene are intersected with rays, paths, beams or pixels. The final image that is outputted on the screen represents the reconstruction of the sampled virtual reality. As all other reconstructions in sampling processes, rendering suffers from aliasing.

Since the rendering process is generally composed of many sub-processes, there are many sources of aliasing, each with different solutions. Pure geometric anti-aliasing can be tackled through progressive representations, such as level of detail representations of the geometric assets and impostors, but this can easily lead to temporal aliasing, which happens during level of detail transitions. Pixel level anti-aliasing can be tackled by state of the art algorithms such as the ones presented in [**Jim11**]. Exact global solutions – applicable for the entire scene - exist, but only for voxel based rendering [**Cra09**]. Furthermore, voxel based solutions need very large amounts of memory for a low alias result. A **global geometric anti-aliasing** solution for primitive based assets would greatly decrease aliasing in rendering.

Correctly rendering **transparent objects in rasterization** is a difficult problem. The problem was first tackled in [**Por84**]. Exact methods [**Car84**] are very costly to implement in real time rasterization based rendering, therefore inexact algorithms [**Liu09**] [**Sin09**] [**Jan10**] represent the best solutions for this problem. Stochastic solutions [**End10**] [**Sal11**] have excellent results, but require a large number of samples to reach acceptable error levels or even specialized hardware [**Sal14**]. Solving this problem with lighter bandwidth consumption would greatly increase the performance of rasterization based real-time rendering of transparent objects.

Tree traversal on the GPU [**Lai10**] [**Ail12**] is used in many global illumination algorithms, but is generally restricted to GPGPU methods. Tree traversal has critical applications in occlusion culling [**Mat08**] [**Gut06**] [**Mat15**], but the current tree traversal methods do not maximize GPU capabilities. In rendering, tree traversal isn't as general as in other fields, therefore there is a lot of room left for the specialization of the tree structures. Providing at least partial **GPU tree traversal** techniques for rasterization based rendering would boost performance.

Global illumination (GI) for real-time rendering remains an unsolved problem. There are many algorithm classes that tackle the GI problem: photon based, many lights based, path based, ray based, cone based, ray-packet based, irradiance cache based and so on. Despite this multitude of solutions, their applications in real-time rendering are minor, therefore the need for fast, inexact, global illumination algorithms has not been satisfied. Porting and adapting scattering prone processes such as global illumination methods to GPGPU environments has proven to be a large area of research, with many opportunities for algorithm development.

Furthermore, this adaptation has led to the usage of tree data structures that trade balancing for spatial awareness. The area of real-time inexact global illumination is, at the moment, one of the most important and researched topics in rendering.

1.3. Contributions

This thesis introduces a new rasterization based rendering pipeline, emphasizing **decoupling** and memory usage efficiency. This rendering pipeline decouples draw submission from state submission, visibility determination from texture fetching, texture fetching from shading, and post processing from texture fetching. It does this while also supporting a large number of light paths. The presented pipeline contains many novel contributions, ranging from improvements over previous algorithms to completely distinct rendering methods. Compared to the state of the art rasterization based rendering pipelines, the presented pipeline is significantly more efficient in solving high bandwidth rendering problems, such as those appearing in the rendering of massive scenes.

A **hierarchical impostor** method is presented, which creates impostors for entire scene tree nodes. Depending on the camera position and orientation, these impostors are updated into a virtual texture based cache system. For performance considerations, high level nodes in the scene tree need to be pre-computed. The impostors contain depth, normals and color information, which can be used for high quality distant object rendering, in order to reduce geometric and texture aliasing, using the same principle as texture mipmaps. Like all other impostor techniques, this method drastically decreases processing costs, by greatly simplifying scene complexity.

Compared to state of the art methods, this technique is integrated with the virtual texturing streaming mechanism and it operates at scene level and not at an object level, by maintaining an impostor per scene node. An impostor is either streamed or re-computed, depending on a view threshold defined by the distance to the camera and the angle of view. Geometric aliasing is largely solved using this type of impostors, and, compared to level of detail systems, impostors have a superior filtering mechanism, and therefore the flickering effects are minimized. The proposed solution also does not have the large storage requirements often found in state of the art voxel based techniques.

The need for **task generation** during rasterization is tackled in this thesis with a special type of non-recursive task generator, which functions in parallel with the normal rasterization rendering process. By using the GPU hardware for geometry amplification to create geometry which isn't rasterized but acts as tasks, the proposed method enables the generation of many light rendering tasks, in parallel with the rasterization process.

This task generator is then particularized for culling, laying the foundation of **a novel culling algorithm**. This method uses the task generator to create tasks, which explore a scene hierarchy. At each level in the hierarchy the node is view frustum culled and if the node is visible, new tasks are generated for the children nodes. Compared to the state of the art method, the improved coherent hierarchical culling algorithm [**Mat08**], this method is less exact and explores more nodes in the hierarchy, but is completely autonomous from CPU control. The proposed method also utilizes the massive parallelism of modern GPUs. While the culling algorithm is not designed with occlusion impostors in mind, it can benefit from them, as it is easy to integrate with hierarchical-Z culling [**Zha97**] methods. Furthermore, the introduced culling algorithm uses a novel concept of culling objects for multiple frames. This is achieved by making the generally correct assumption that camera rotations are not instantaneous in 3D interactive applications. The camera orientation is then represented as a solid angle, and the maximum camera rotation speed is used to determine the smallest number of frames necessary for an object to reach the

view frustum. Thus, the new culling algorithm culls objects for a number of frames, which leads to lower overall computational effort.

Novel **measurement metrics** for deferred algorithms introduced in this thesis. Since the state of the art offers a very large number of deferred rendering variants, correct selection of the suitable method can sometimes be difficult, especially because real-time rendering software is usually deployed on a large number of varied platforms. The novel measurement metrics provide a clearer picture, by providing a means of comparison between the existing methods.

Virtual deferred (VD) is a new type of deferred algorithm, which combines the mechanics of single geometry pass deferred rendering with those of virtual texturing, in order to obtain the best measurement metrics for high bandwidth high complexity rendering problems. Because of this, virtual deferred is especially suitable for complex scenes. Virtual deferred uses virtual texturing to stream only the data required for rendering. Since the majority of rendering pipelines aimed at complex scenes have a streaming mechanism, the proposed method only seems to have an additional cost. The benefit of combining virtual texturing and deferred rendering into virtual deferred is that rendering with this algorithm minimizes the bandwidth cost for complex materials, while processing the geometry only once, improving on the poor bandwidth performance of single-geometry pass state of the art deferred algorithms. Furthermore, virtual deferred also lowers the allocated memory for the geometry buffer, decreasing storage requirements.

With virtual deferred, both the bandwidth consumption and geometry buffer memory costs scale better than with state of the art single-geometry pass methods. Because of virtual deferred, the introduced rendering pipeline completely decouples texture fetching and shading bandwidth from visibility determination. The shading stage does not use rasterization, it is a completely GPGPU process. All the texture fetching is performed in a cached mode, through compute shader work tiles. If this stage is followed by an optional post-processing stage, the kernel participating pixel neighbors can benefit from the shading results stored in the work tile cache.

Correctly rendering transparent objects is especially difficult for rasterization based real-time rendering because the process requires ordering the fragments on a per-pixel level instead of just approximating the opacity function. While approximation methods can produce acceptable results for some types of rendering situations, there are times when handling transparency exactly is required. **Virtual Order Independent Transparency** (VOIT) modifies the state of the art GPU A-Buffer algorithm. Like virtual deferred, VOIT uses virtual texturing and is aimed at decreasing the bandwidth and allocated memory for scenes with many high-bandwidth objects. Compared to the state of the art, VOIT scales better in scenes with complex materials and many lights.

Besides Virtual Order Independent Transparency, a fast **approximate order independent transparency** solution is provided, which modifies the state of the art occupancy maps method. This proposed technique improves occupancy maps with distributions, by keeping a depth distribution for each pixel. This depth distribution is used to adjust the depth occupancy map, creating more precision in the areas which contain objects on the depth axis. Because of this artificially increased resolution, the opacity function over depth is better approximated. In

contrast to the original occupancy maps algorithm, the proposed method is much better suited to non-uniform object distributions, such as those found in real-life scenes.

A new **selection algorithm** is introduced, which optimally solves the problem of object selection in real-time rendering. The introduced algorithm is able to handle hardware instancing, high depth complexity scenes and alpha culling, either for a single pixel or for an entire screen area. The algorithm can handle the transient geometry created by hardware tessellation, making it easy to work with displaced geometry. The method can also correctly select fuzzy objects such as particle systems and other transparent objects, based on their visibility. This property is achieved by correctly performing transparent rendering, and taking into account alpha occlusion.

Conservative Inexact Voxelization (CIV) is a new inexact voxelization algorithm, which lowers the complexity of the voxelization operation from the number of primitives in the scene to the number of objects. This imperfect result is obtained by quickly dicing the bounding boxes of the objects into cuboids, which are stored in a hierarchic voxel representation of the scene. A push-pull process is used to translate the opacity information across all layers of the hierarchic representation. Other already available data, such as directly visible geometric features stored in the depth buffer, or impostors, can be back projected inside the highest resolution level of the hierarchic representation.

Conservative Inexact Voxelization is an inexact voxelization method, aimed at providing approximate but conservative information about the geometrical nature of a scene. It offers sufficiently precise information for different visibility determination operations. This property is used to alter the rendering equation, changing the exact visibility operator to an inexact but conservative visibility operator. This operator is then used in a **modified instant radiosity** method, which creates a large number of virtual lights through random walks in the voxel representation of the scene, starting from the scene lights. After a sufficient number of virtual lights is generated, the scene is illuminated with them, in a process analogous to deferred algorithms.

Compared to the state of the art methods, this enables fast diffuse light transport without any precomputation or special cases for animated or moving objects. While this method only supports diffuse light transport, it can be used together with fast specular light transport algorithms such as screen space cone tracing.

A new antialiasing algorithm is introduced which is compatible with deferred rendering. **Decoupled sub-pixel reconstructed anti-aliasing** (DSRAA) decouples visibility determination, the bandwidth of attribute samples and shading. Compared to the state of the art methods, it consumes the minimal amount of bandwidth required for accurate, non-morphological sub-pixel geometry reconstruction. Because the presented algorithm is not morphological, it does not suffer from strong temporal artifacts and it does not need temporal resampling to adjust the final result. The algorithm is most similar to sub pixel reconstructed anti aliasing (SRAA). Compared to the state of the art method, which uses a bilateral filter based approach for sub-geometric filtering, the presented technique uses a more accurate neighbor matching method.

The path tracing algorithm family is barely at the edge of interactive rendering, and while it is still impractical for real-time rendering, due to the computational limits of consumer hardware, it has reached the point of relevancy. The thesis renders photorealistic images with a

bidirectional path tracer and introduces a contribution to the path tracing field: **light flux importance sampling** (LFIS). The thesis first presents a procedure, through which the Conservative Inexact Voxelization is used to amortize the cost of visibility determination operations, which are responsible for much of the rendering time in path tracing. Thus the presented technique decreases the overall cost of tracing rays. The second contribution is an innovative importance sampling mechanism, which is used to guide paths to high energy areas. By using these novel techniques, the final path traced image is obtained faster.

Lastly, a new **variation of the GPU marching cubes** algorithm is introduced. It serializes the processing of a large dataset, enabling the computation of the marching cubes algorithm on very large datasets, which would otherwise be impractical on consumer hardware. The original dataset is cut into chunks, which are reconstructed serially and are then stitched.

1.4. Thesis Overview

The thesis is divided into 5 chapters: Introduction, State of the Art, Geometry Processing, Illumination and Conclusions. This chapter, Introduction, describes the particular field of the thesis, real-time rendering of very large and complex scenes. It then describes the motivation of the thesis and the objectives which catalyzed the presented research. The chapter then shortly presents the original contributions of the thesis and finishes with the thesis overview, this subchapter.

The State of the Art chapter provides an ample and detailed description of the applicable or potentially applicable algorithms for real-time rendering. It starts with a short description of the evolution of GPUs and GPGPU computing, followed by a conceptual discussion of the choices and consequences of different types of data representations. Since the nature of this thesis implies a very large volume of rendering information, the chapter continues with the presentation of all the relevant acceleration structures used in rendering.

The State of the Art chapter also presents high level rendering concepts and insights into the nature of the rendering process, from basic elements of radiometry to high level design problems such as solving rendering as large sorting process of visibility determination operation. The chapter ends with seven sub-chapters, which discuss the largest rendering algorithm families in a concise and encompassing manner: rasterization, image (screen) space methods, Reyes, ray tracing, path tracing, photon mapping and many light methods.

The next two chapters, Geometry Processing and Illumination, present the research work and the resulting contributions. These chapters largely decouple geometry processing and illumination algorithms, in a manner similar to deferred algorithms. Furthermore, while the Geometry Processing chapter is based on rasterization rendering, the Illumination chapter is fully implementable as GPGPU. Through this decoupling, the thesis provides the framework for implementing a robust rendering pipeline.

The Geometry Processing part of the pipeline performs geometry operations such as direct visibility determination, culling, asset definition and streaming, opaque and transparent rasterization rendering or indirect rendering. In this chapter the rendering asset format is defined, which is based on instances and modifiers. The streaming mechanism used by the proposed rendering pipeline is described, and new variation of the GPU marching cubes algorithm is

presented. The hierarchical impostor cache contribution is also described here. The entire streaming mechanism is based on virtual texturing and virtual meshes.

The Geometry Processing Chapter continues with the description of a novel task generator, which is able to use the rasterization scheduler, and thus is able to function in parallel with rasterization algorithms. This task generator is then specialized for a new culling algorithm, which enables hierarchical culling for a very large number of objects, without CPU interference.

The chapter continues with the analysis of existing rendering algorithms for opaque objects, in the context of rasterization, for which novel measurement methods are provided. It then presents virtual deferred, a novel type of deferred algorithm. Compared to state of the art deferred and decoupled algorithms, the virtual deferred method is designed for complex materials, such as those found in large, varied scenes.

Virtual deferred principles are also adapted to transparent object rendering, giving rise to the virtual order independent transparency algorithm, which is a memory efficient variant of the A-Buffer. Besides virtual order independent transparency, the Geometry Processing chapter also presents a novel approximate transparency algorithm, which improves the state of the art occupancy maps with per pixel distributions. The chapter ends with a novel selection algorithm. Many of the outputs computed by algorithms in this chapter are used as inputs for the algorithms in the Illumination chapter. The relationship between the algorithms presented in both chapters is depicted in Figure 2.

The Illumination Chapter contains two rendering paths, one that performs approximate global illumination, shading and post processing and one that computes correct global illumination. The chapter starts with the approximate path, with a presentation of how dominant lights are selected and how shadow maps are used to represent the visibility of each object found in the frustum of each light. This process is accelerated with the culling algorithm from the Geometry Processing chapter.

A novel acceleration structure is then presented, the Conservative Inexact Voxelization algorithm, which can be used to quickly and conservatively query the scene geometry. This structure is used to relax the visibility operator in a modified instant radiosity framework, which is used to generate a large number of virtual lights.

The chapter then presents how these virtual lights are stored together with the scene lights in a secondary lights acceleration structure. The chapter continues with the shading stage, where data is cached in compute shader tiles, and illumination is evaluated with the dominant lights, by using their shadow maps, and with the secondary lights stored in the acceleration structure, by intersecting them with the modified G-buffer produced in the Geometry Processing chapter and then using shadow rays inside the conservative inexact voxelization representation to query the visibility for each light. Screen space cone tracing is then used to approximate the specular light transport.

The chapter also presents a novel decoupled sub pixel reconstructed anti-aliasing method, and ends with a short post processing module.

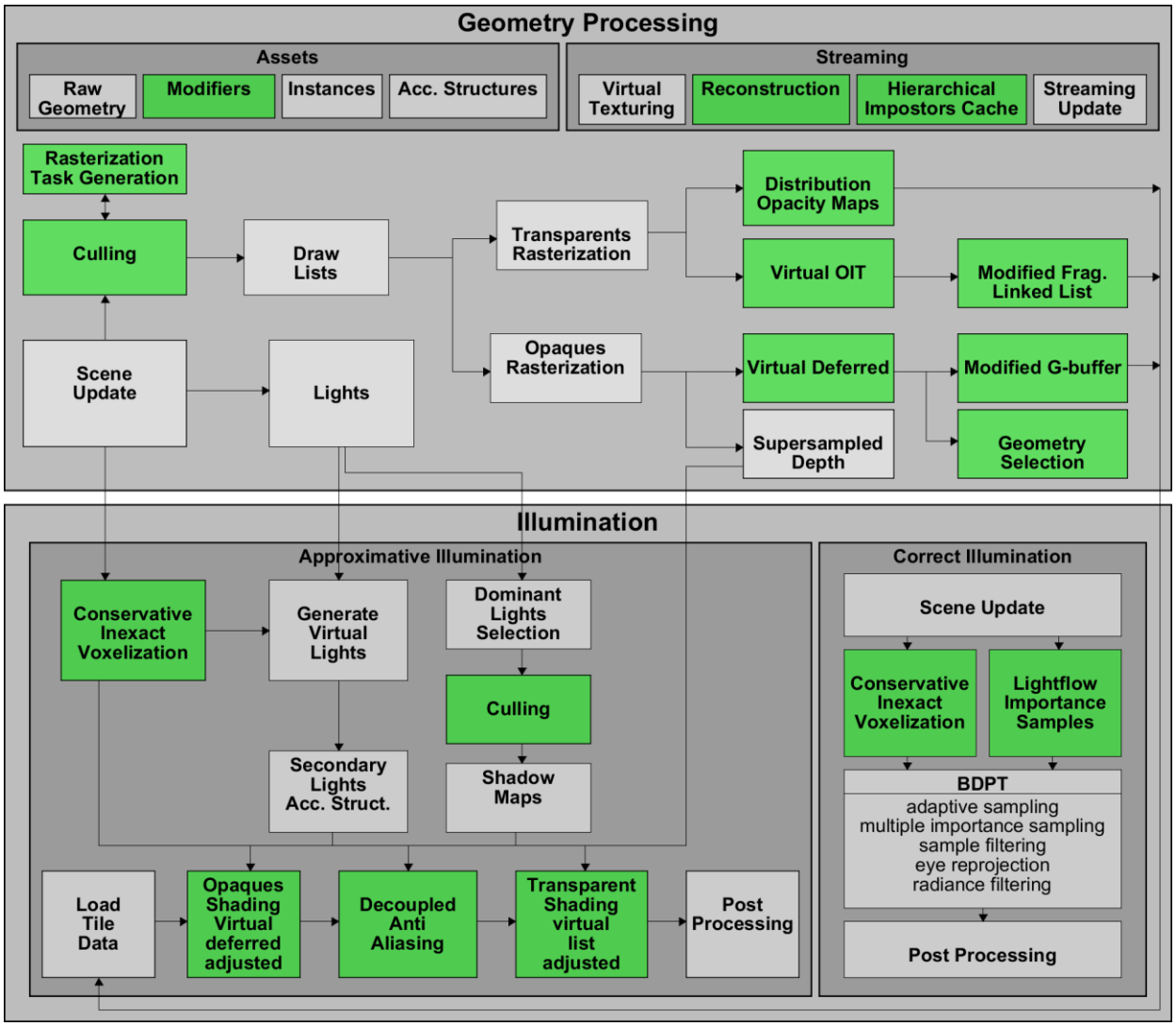

Figure 2 Proposed Rendering Architecture**.** Geometry Processing and Illumination are the two chapters of the thesis which describe the novel contributions, and they are also the two large stages of the proposed rendering pipeline. This image shows how the various algorithms described in this thesis are combined into a rendering pipeline. The contributions introduced by this thesis are colored in green. For architectural design reasons the Culling and Conservative Inexact Voxelization modules have been drawn twice. The proposed rendering pipeline decouples lighting and shading, which are implemented in the Illumination stage, from all the rasterization based algorithms and geometry streaming, which are implemented in the Geometry Processing stage. The Geometry processing stage includes contributions to opaque and transparent object rasterization, geometry selection or culling. The illumination stage contains two alternative rendering paths, one for approximate illumination and one for correct illumination. The approximate illumination path computes low frequency light transport and approximates high frequency light transport. It approximates the visibility operator through conservative inexact voxelization, a very fast imperfect voxelization method. The correct illumination path can be used to compute high quality visual reference results and it runs at very low frame rates on the GPU. It uses bidirectional path tracing, in which the Conservative Inexact Voxelization is used to approximately determine visibility before paying the cost for high quality visibility rays, amortizing their cost. The BDPT uses Light Flux Importance Sampling, which is a novel type of importance sampling that very quickly connects the light tracing and path tracing steps of the BDPT through high energy space exploration, indirectly importance sampling lights. The BDPT also uses state of the art algorithms such as adaptive sampling, multiple importance sampling, sample filtering, eye reprojection and radiance filtering.

The Illumination chapter also contains an alternative illumination path. While this alternative path can't run in real-time on current consumer hardware and its GPU implementation is at the edge of interactive rendering, it can be used to produce visual reference results.

The alternative path presents an adaptation of the bidirectional path tracing algorithm, which uses the Conservative Inexact Voxelization algorithm introduced for the approximate global illumination pipeline. The CIV is used to amortize the cost of visibility determination operations, which usually consume more than 90% of the rendering time. By first searching approximately and only then searching exactly, the bidirectional path tracer uses a visibility determination process with a lowered complexity. Because of this, the bidirectional path tracer produces visual results faster. The bidirectional path tracer uses a novel type of importance sampling, light flux importance sampling, which creates an imperfect map of the flux of light in the scene, similar to a flow map for fluids. The light flux map is then used to guide paths towards vertices produced by the light tracing stage of the bidirectional path tracer.

The thesis ends with the Conclusions chapter, where the impact of the newly introduced methods is reviewed.

2. STATE OF THE ART

2.1. GPU Evolution

Graphics Processing Units (GPU) are dedicated parallel processor architectures which were initially optimized for accelerating graphical computations. GPUs have evolved from the need to process an ever increasing number of vertices and fragments per frame, at least 30 frames per second. Because of this extremely large number of operations which lack an explicit order, GPUs evolved into massively parallel fully programmable architectures which excel at executing a large number of floating point operations [**McC11**]. Initially, each type of graphical computation used specialized hardware [**Cla82**], but modern GPUs have unified architectures [**NVI12**] [**Man13**], where the same computational resources are used for all types of operations. The only exception is with extremely specialized, performance critical operations, such as texture fetching, Z-buffer visibility determination or hardware tessellation.

The computational powers of GPUs have grown orders of magnitude higher than those of CPUs. This evolution prompted domains unrelated to rendering to start solving their computational problems on the GPU, giving rise to **general purpose computation on graphics processing units** (GPGPU) and greatly enhancing the field of computer graphics in the process. Real-time rendering transitioned from an exclusively rasterization based method to a composite process involving both rasterization and GPGPU.

The original GPUs were modeled as a set of stages in a hardware graphics pipeline. The hardware graphics pipeline transforms vertices from a tridimensional space defined by the user into pixels in a bidimensional space on the screen, a process called **rasterization.** Throughout the last 20 years this pipeline has evolved from the initial design into a long pipeline with many highly specialized, programmable stages.

These evolutions were made in steps, where new standards named **Shader Models** (SM) [**Seg15**] progressively supplemented the pipeline with new stages. The current Shader Model is 5 and it specifies 13 rasterization graphics pipeline stages, out of which 5 are programmable. It also specifies a Compute Shader (CS) stage outside of the rasterization pipeline that can be used for GPGPU. This thesis is written with the OpenGL nomenclature, but also offers the Direct3D names for completeness.

The hardware rasterization pipeline starts with the Input Assembler (IA), which takes the vertex data from memory and assembles with it tridimensional vertices. This is followed by the Vertex Shader (VS), which is a programmable stage where each vertex created by the IA is transformed. The transformed vertices are sent to the Primitive Assembler (PA) stage, where they are then combined with the topology, which is read from memory.

PA outputs primitives to the next, optional, superstage that handles hardware tessellation. It contains 4 sub-stages, out of which 2 are programmable. The Tessellation Control Shader (TCS), also named Hull Shader in Direct3D, is a completely programmable stage that takes the

primitives outputted by the first PA stage and computes tessellation factors. These tessellation factors are then used by the Tessellator (T), which creates new vertices and topology. These new vertices are then transformed in the Tessellation Evaluation Shader (TES), also named Domain Shader (DS) in Direct3D. The transformed vertices and the topology created by T are then assembled in a second Primitive Assembler (PA) stage.

The output from the last executed Primitive Assembler stage acts as input for the Geometry Shader (GS), which is another optional programmable stage, which processes each primitive. It is the only stage where primitive topology can be modified. The output from the GS can be optionally streamed into memory through the Transform Feedback (TF) stage, named Stream Output (SO) in Direct3D.

The output from the GS is then sent to the Clip and Perspective Divide (C&PD) stage which culls primitives completely outside the visualization volume. The primitives that pass this stage are rasterized into fragments by the Rasterizer (R). The vertex attributes are interpolated by the rasterizer in perspective correct manner [**Low02**].

The resulted fragments can be processed by a hierarchical variant of Z-buffer algorithm [**Gre93**] before or after the Fragment Shader (FS) stage, named Pixel Shader (PS) in Direct3D. Depending on several factors such as transparency rendering state settings, the Early Tests (ET) stage can discard fragments before they are shaded. The rasterization pipeline ends with the Output Merger (OM) stage where fragment tests are performed and the output is composited and written into the framebuffer.

A short overview is provided in Figure 3.

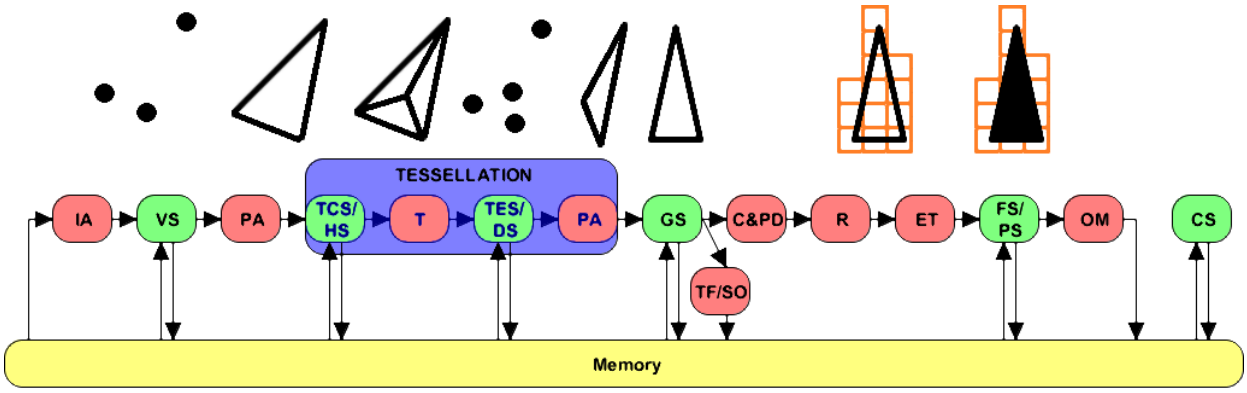

Figure 3 The hardware graphics pipeline in SM5**.** The programmable stages are colored in green. The Compute Shader (CS) can be used to perform GPGPU work. The hardware graphics pipeline begins by assembling the input geometry and topology in the Input Assembly (IA) stage and then processes each vertex in the Vertex Shader (VS). The processed vertices are then combined with the topology in the first Primitive Assembly (PA). The next four stages implement hardware tessellation. The Tessellation Control Shader (TCS), also named Hull Shader (HS) determines the hardware tessellation parameters, and then the Tessellator (T) creates the new vertices and their topology. The Tessellation Evaluation Shader (TES), also named Domain Shader (DS) transforms the newly creates vertices and the next Primitive Assembly (PA) stages links the new vertices with the tessellator created topology, which are then converted and transformed by the Geometry Shader (GS). The GS output can then be streamed into memory buffers with the Transform Feedback (TF), also named Stream Output(SO). The GS output can also continue in the Clipping and Perspective Divide (C&PD) stage, and then be rasterized into fragments by the Rasterizer (R). The fragments are tested in the Early Tests (ET) stage and are then processed by the Fragment Shader (FS), also named Pixel Shader (PS). The processed fragments are then outputted to the framebuffer in the Output Merger (OM) stage.

When GPUs transitioned to the **unified shader model**, every shader program began running on the same type of resource, a very lightweight GPU thread. GPUs are dedicated to efficiently solving massively parallel coherent computational loads, like fluid simulation or linear systems. Because of this, each GPU thread uses vectorized instructions. Further widening the data, GPUs use a special type of SIMD-like instructions, called SIMT, **single instruction multiple thread**, where each thread in a thread group executes the same instruction.

These threads are very cheap to create and destroy, have a small number of registers and have no heap in which to allocate local memory. **Functional Units** (FU) are the building block of hardware GPU architectures, and they contain one or more GPU threads. GPU cores are composed of many FUs along with a very lightweight thread scheduler and a small shared memory chip. Within a core, threads are grouped into warps (Nvidia), wavefronts (AMD-ATI) or groups (Intel) and they are run together on SIMT hardware. A Streaming Multiprocessor (SMP) contains many GPU cores, and the GPU has many SMPs. A simplified abstraction of GPU architecture is provided in Figure 4.

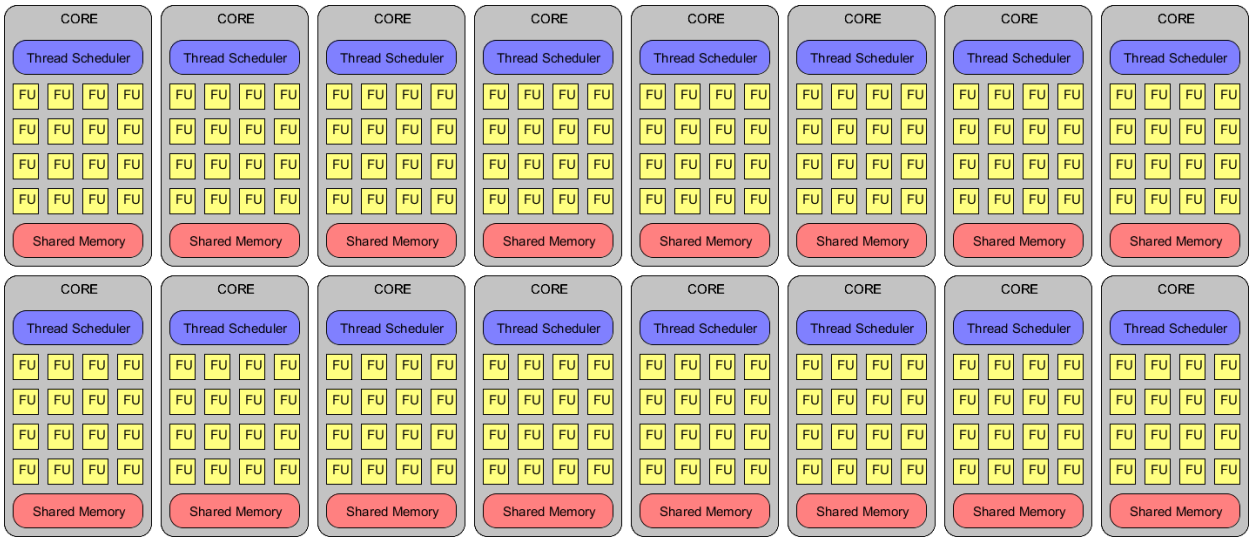

Figure 4 Abstract modern GPU Architecture**.** The GPU contains many cores, each composed of a large group of Functional Units (FU), which in turn contain one or more GPU threads. Each GPU core contains a thread scheduler and a small amount of shared memory.

Since GPU architectures are Single Instruction Multiple Thread they have very high latency when threads encounter long instructions, for example a texture fetch which needs 400-1000 cycles to complete, because all the threads in the active thread group have to wait after it. GPUs architecture attenuates these long waits by using latency hiding mechanisms. The most relevant latency hiding mechanism is lock step running, in which the same core is responsible for running more than a single work group of threads per core. When one group of threads encounters a long waiting instruction the core saves the state and switches to a new group, to not waste computational cycles. This concept is similar to the hyper-threading done by CPUs.

Because of the SIMT model and the limited amount of cache memory with which GPUs come, running branching code is very problematic, because all threads have to run both branches. When in-thread group branching occurs the code paths executed by the threads diverge. This is named code path divergence.

Even with latency hiding mechanisms like lock step execution and an increasing number of caching methods, latency is still limited by bandwidth. Therefore, **bandwidth and not computational power is usually the greatest bottleneck for GPUs**.

Dynamic parallelism is a recent development of GPGPU programming models [**NVI15**] [**KHR14**], where GPU tasks can generate other GPU tasks themselves. This is unfortunately limited to a small number of consumer graphics cards. Current and newly introduced drawing APIs [**NVI12**] [**Seg15**] [**Mic15**] [**AMD15**] [**Vul15**] are also evolving to a more general, stage-less and low level approach.

There have been initiatives [**Wal01**] [**Pur02**] to create hardware for other rendering algorithms besides rasterization, but data coherency is not favorable. Also, many core architectures exhibit better efficiency per watt [**NVI15**]. This is an important economic reason for the success of many core architectures.

2.2. Data Representation

2.2.1. Raw Data

Real-time rendering is a performance critical process, which works with a very large amount of data, therefore efficient data representation is vital. It is far from ideal to convert data from one format to another during rendering. Moreover, memory bandwidth and consumption are very often the performance bottleneck in many-core architectures; therefore the data representation of the assets used in real-time rendering has to exhibit excellent locality and size. Last but not least, data is streamed from the system memory into GPU memory, along with state information. When this takes place, it is usually accompanied by long synchronizations that lead to pipeline stalls. Thus, state transfers should be minimized.

The assets displayed in real-time are usually only representations (not scans) of real-life objects. Therefore, the assets represent a virtual, low resolution, sampled version of the real objects. This sampled data has to be stored in some sort of data structures. In rendering, asset data can have representations based on curves, triangles, voxels, functions or points, depending on the type of application. Curves are commonly used in modeling applications like CADs, points are used in applications that reconstruct topology from a sampled set. Functions are very computationally heavy representations and frequently real-life objects can only be represented by a large number of functions, therefore they are not the most practical representations and are used for exceptional cases. **Triangles and voxels** have seen the most success in real-time rendering, although **points** are used in some important algorithms, where high frequency information is not very important [**Rit08**] or completely unknown [**Kha06**].

The great difference between triangles and voxels is that triangles are a more analytical approach to data representation, with a much smaller representation error, and with a very low memory footprint. Voxels can represent real-life objects with the same error only by using a large number of samples, which causes them to be ineffective from a memory consumption standpoint. On the other hand voxels have excellent locality and have a regular topology which

makes it extremely easy to find neighbors, thus they are suited to a large number of mass parallel algorithms and architectures. A simple visual comparison is offered in Figure 5.

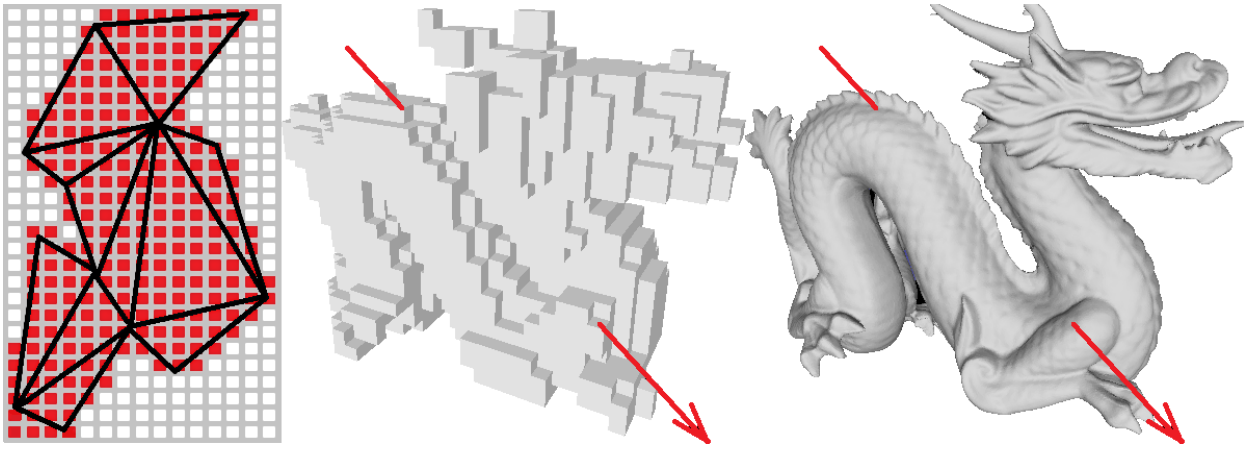

Figure 5 Triangles and voxels comparison**.** The triangles need much fewer space samples to accurately represent the same object. On the other hand voxels don't require graph-like structures for finding neighbors and are more cache friendly structures, thus they have a lot of applicability on SIMT architectures. Voxels also have an implicit LOD structure, while the computation of LODs is a difficult problem for triangular representations. In approximative intersection problems voxels can be just as good as triangles. In direct rendering problems the difference in the quality of the reconstruction of the original geometric signal is very large.

2.2.2. Raw Data Compression

Multi resolution scenes require large amounts of memory to represent a large number of objects and interaction media, and the majority of assets have to be compressed. The compression method must have very fast decompression complexity, while the compression complexity isn't considered critical, as the majority of assets go through non real-time pre-processing. Scene objects can be represented through meshes, a combination of meshes and textures or textures only.

Algorithm	BC1/ DXT1	BC2 /DXT3	BC3/DXT5	BC4	BC5	BC6	BC7
Data Type	RGB +1bit A	RGB+4bit A	RGBA	G	2 x G	RGB float	RGB(A)
Compression (byte/px)	0.5	1	1	0.5	1	1	1
Palette Size	4	4	4RGB+8A	8	8 per channel	8-16	4-16
Line Segments	1	1	1RGB+1A	1	1 per channel	1-2	1-3

Table 1 Texture Block Compression**.** State of the art Block Compression (BC) algorithms use different strategies to compress texture data. The compression rate varies drastically due to the variety of encoded texture data: intensity gradient (G), RGB format, RGBA format, RGB and binary alpha representation or high dynamic range (HDR) RGB.

Textures can be compressed with a large number of block algorithms. This is based on the assumption that colors vary slightly over small vicinities. Even if variable rate encoding produces the best results, real-time decompression considerations have made all block compression algorithms to be fixed rate encoded. DXTC/S3TC [**Iou99**] compression and **Block Compression** (BC) [**Khr15**] encode image information in 4x4 blocks. In each compression BC algorithms define a color vector between 2 endpoints, which represent the extents of the pixel values in color space. This color vector is named line segment. The original image pixel values are represented in a short binary representation over the line segment, which creates a small colorization palette. Greater palette sizes increase memory requirements and compression rate

but increase compression accuracy. BC6 and BC7 are complex algorithms and can use multiple segments and palettes. A comparison of BC algorithms is offered in Table 1.

Generalized Triangle Meshes were introduced in [**Dee95**] and can be compressed through lossless algorithms, but the decompression costs are extremely high for real-time rendering [**Hop96**]. Mesh simplification [**Coh98**] can create low resolution representations, thus the algorithm can be considered a lossy compression method.

Deferred Rendering [**Dee88**] is a very popular technique in real-time rendering, which stores frame data into a screen wide multi-layered buffer. While some of the stored data can be compressed in an application dependant manner through a decrease in the number of storing bits, normals need proper encoding. A short overview of existing methods is offered in [**Pra15**]. Normals can also be compressed with look up tables or as octahedron-normal vectors [**Mey10**].

In real-time rendering general compression is mostly applicable to streaming data. The majority of compression solutions are based on Lempel-Ziv (LZ) [**Ziv77**] inspired variants such as DEFLATE [**Kat91**] for general zip or LZ4 [**Col15**], which offers extremely fast compression and decompression. LZ4 has the option of running in constant space, which makes it ideal for fast decompression of large volumes of data.

2.2.3. Progressive Data Representations

Rendering is itself a sampling process, which takes objects from tridimensional coordinates and transforms them into projections of bidimensional coordinates on a grid. Independently of the rendering algorithm, the display process samples the geometry and color of the objects of the scene and constructs a view of them on the output monitor. Sampling distant objects can be a very costly process, therefore prefiltering is preferred in rendering, and therefore objects are generally pre-filtered in progressive data representations to minimize the computations which are required for their distant rendering.

Triangular based representations of objects are called meshes, and can have **levels of detail** (LOD) [**Hop96**] [**Hop99**], which are low sampled representations of the objects' geometry. The samples are taken at different distances and decrease the geometric detail with the increase in object distance, in order to decrease aliasing caused by undersampling detailed meshes. They are chosen for rendering depending on quality metrics like screen projection size or distance to the near visualization plane. Triangular meshes can also be imperfectly approximated into textures [**Los03**], which have their own LOD mechanism.

The images used as **textures** with the triangular based representations of meshes have mipmaps, which are hierarchical level of detail structures. While superior filtering methods like Summed Area Tables (SAT) or Elliptical Weighted Average (EWA) exist, the cheap bilinear, trilinear or anisotropic filtering methods implemented in hardware have provided sufficient quality for real-time rendering.

Voxels have **tridimensional mipmaps**, which provide superior quality metrics compared to the tridimensional LODs. From the standpoint of scene sampling the voxel representation is much more efficient because of its mipmaps, which represent the object with a small number of pre-filtered samples, effectively sampling entire parts of objects with few memory reads. On the other hand, the reconstruction of mesh LODs is a complicated process. Another important aspect

of voxel use in real-time rendering is the fact that voxelized geometry is usually implemented as tridimensional textures, for which GPUs have specialized filtering hardware, which leads to very efficient queries. Furthermore, voxel based simplified representations of the scene are resolution independent, making memory consumption and rendering times both lower and predictable. Because of these properties, **voxelization** has many uses in computer graphics and has been thoroughly explored as a fast and inexact object representation. Voxelization can be sparse [**Cra09**], conservative [**Cra14**], based on imperfect shadow maps [**Wym13**], or it can computed through rasterization as a set of slice/opacity/occupancy maps [**Eis06**] [**Eis08**].

Fractals are natural progressive representations and are easy, but expensive, to evaluate analytically. Fractals are often used to augment real data with quasi-random fine detail, named **procedural noise**. Procedural noise was first used in computer graphics as Perlin noise [**Per85**], which interpolates random samples across a multidimensional grid. This method was later refined into Simplex Noise [**Per01**], which uses a simplex instead of grid and thus decreases complexity from $O(2^N)$ to $O(N^2)$, where N is the number of dimensions. Techniques such as Wavelet noise [**Coo05**] and Gabor noise [**Lag09**] build upon the Perlin and Simplex noises by improving antialiasing in low frequency sampling patterns.

Triangular representations of geometry can also be approximated through **impostors**, which are lightweight memory wise and computationally, but are useful only for far away objects. Impostors can be used to great effect in scenes with controlled movement. There are many types of impostors.

Billboards [**Ger88**] are represented with a very small number of quads and can be aligned to always face the camera, but they can't successfully represent surface detail or parallax effects. Correct rotationally invariant impostors can't be obtained without some form of multi-view representation. **Billboard clouds** [**Dec03**] can be used to attenuate artifacts approximating an object through many billboards, instead of one. Heightmaps, bump maps and normal maps can be applied to billboards to increase surface detail. Billboards have also been used in the implementation of approximate volumetric effects. The **omni-directional relief impostors** [**And07**] uses a form of iterative parallax mapping with billboard clouds, in order to better represent surface detail.

True impostors [**Ris06**] introduces the idea of layered object representation to impostors. In this technique the original mesh is projected into a texture, where each channel holds a different depth sample, in a similar way to depth peeling. Therefore, a single texture can hold up to 4 different depth samples of a mesh, roughly approximating it. Then, the mesh is rendered as a simple impostor which performs a variant of ray-marching as shading. In the ray-marching stage the intersection between the view ray and the surface is much more accurate than intersecting a billboard cloud.

3-view impostors [**Har10**] uses distance fields to quickly traverse a coarse volumetric representation of the mesh. The volumetric representation is stored in 3 min-max maps, taken from the 3 canonical axis views. Each map stores the minimum and the maximum depth value per view, along with a maximum step with which the ray can be advanced, similar to relaxed cone stepping [**Pol07**]. Rendering with 3-view impostors is done by bounding the impostor within a box and sphere tracing it. The distance advanced between samples is decided using queries from each view.

Volumetric billboards [**Dec09**] are voxel structures, which can be traced with any form of ray-marching or tracing. Compared to the previously discussed impostors they can handle level of detail and scattering with much better results. Bundling of different impostors has been explored in [**Ume05**] and [**OHa02**], but only in the context of large homogenous systems such as flocks of birds of clouds.

Spheremaps and **cubemaps** are an important data structures in rendering because they are frequently used to sample or encode the environment of one or a group of objects. Because they are used for sampling, cubemaps offer a cheaper solution since they can be sampled uniformly, while the uniformly sampled spheremaps would suffer from distortion at the poles. Therefore, correct spheremaps would require extra computational effort to compute sample positions and minimize representation artifacts.

Polycubemaps [**Tar04**] are a surface approximation technique based on cubemaps, where a surface is parameterized to a set of cubemaps, similar to an axis aligned bounding box (AABB) tree, which can then be displaced to reconstruct the surface. The advantage of this technique is that it has an inherent level of detail system because all the parameterized displacement is written into a single displacement texture, therefore all texture LOD algorithms are applicable. Other advantages of this structure are that it can represent concave surfaces and that the displacement texture can be streamed like any other texture through virtual texturing mechanisms [**Eph06**].

Rasterized bounding volume hierarchies (RBVH) [**Nov12**] are impostor trees which inexactly describe the scene as bounding volume hierarchy of height fields, generalizing polycubemaps for rendering. RBVH represent geometry inexactly by finding sub-surfaces that are easy to express as parametric inside the geometry and then represent them as parameterized heightfields. When this is hard to achieve the RBVH can be transformed into a hybrid RBVH (HRBVH) which keeps the actual triangles and not heightfields at leaf level. A short comparison between complex impostors such as cubemaps, polycubemaps and RBVH is given in Figure 6.

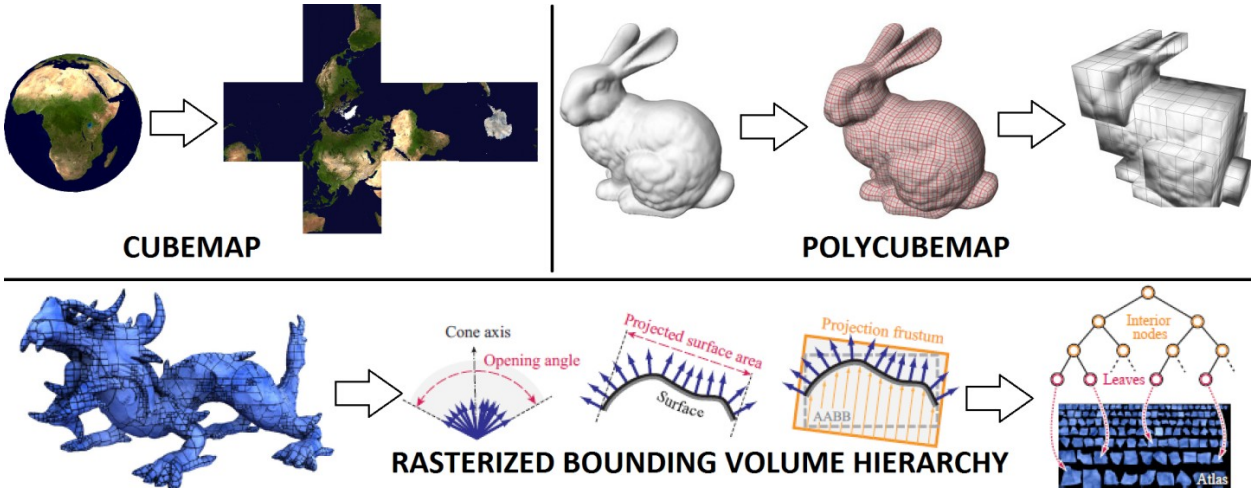

Figure 6 Complex Impostors**.** Cubemaps approximate an object by projecting it onto 6 planes. Polycubemaps take this concept further, by first creating a parametrization of the object onto an AABB tree structure and creating displacement maps for the AABB tree which can be used to inexactly reconstruct the original surface. Compared to cubemaps, polycubemaps can handle concave surfaces. Rasterized Bounding Volume Hierarchies (RBVH) map the surfaces on an object to heightfields, and create a tree hierarchy over them. RBVHs can be considered as a generalization of polycubemaps. Lower side of the image is from **[**Nov12**]**. Right upper side is from **[**Tar04**]**.

Comprehensive descriptions of massive scenes are done through storing all the information of a scene in hybrid hierarchies governed by scene managers. These representations are often application driven, and interoperability between such applications is difficult [**Ber11**].

Instance based scene definition is a relatively novel concept, which has only been tried with families of objects [**Bar08**] and not with entire general scenes.

Very detailed asset representations for distant or unimportant objects are not necessary, as the same visual result can be obtained with lower resolution variants of such assets. **Streaming** mechanisms such as virtual texturing [**Eph06**] and virtual meshes are used to load in memory only the most relevant data for rendering.

2.2.3.1. Subgeometric Data

An important part of scene description is that objects often exhibit minute detail. Scenes described entirely through voxels can be perfectly sampled through algorithms like [**Cra09**] but those that contain geometry information require a method to correctly represent subgeometric detail.

Displacement mapping is the simplest form of subgeometric rendering. The original mesh is tessellated to a new resolution and the newly created vertices are displaced with a **displacement map**. This technique is very easy to implement in SM5 rasterization pipelines. The disadvantage of this method is that tessellating geometry increases geometric aliasing, which can be very difficult to efficiently solve without oversampling like Reyes based renderers. Hard to sample surfaces that exhibit geometric detail not represented in geometry can be difficult to properly render, because sampling geometric detail over the frequency of the geometric representation is not a straightforward problem. Such surfaces are also named **wrinkled surfaces.**

Rendering wrinkled surfaces is performed through subgeometric algorithms, also known as surface algorithms, which transform information from the surface **tangent space** into tridimensional space and use it in shading. The tangents space computations can be bypassed with derivative maps [**Mik10**]. Various real-time algorithm have been used to adaptively tessellate surfaces with implicit displacement data, such as Phong Tessellation [**Bou081**], PN-Patches [**Vla01**], Gregory Patches [**Loo09**], or Semi-Uniform Adaptive Tessellation [**Dyk09**].

Microfacet based reflectance models [**Coo82**] simulate wrinkled surfaces by incorporating the surface geometric variations into the shading model. Similar to microfacet based reflectance models, **fractal** surfaces can be evaluated analytically, which solves the problem of sampling.

Wrinkled surfaces can be convincingly rendered with a small number of samples**. Bump mapping** [**Bli78**] treats wrinkled surfaces as a base geometric surface and a wrinkle function, which can be encoded in **heightmap**. The heightmap can then be evaluated multiple times in a vicinity of the surface sampling position. The multiple wrinkle function evaluations can be used to approximate the modified normal at the surface sampling point. **Normal mapping** [**Coh98**] is a variation of bump mapping, where the entire normal approximation algorithm is pre-processed, and its results are stored in a **normal map**, which is then queried during rendering. Compared to bump mapping, normal mapping requires less map sampling and has increased precision, but requires transforming the normal from the normal map space into tridimensional space.

Parallax mapping [**Kan01**], also known as offset mapping and virtual displacement mapping, modifies bump mapping to account for steep view angles, where the displacement is increased, simulating the effect of parallax.

Iterative wrinkle surface methods provide realistic, high quality renderings, but are significantly more expensive than the previously presented techniques. **Dynamic Parallax occlusion mapping** (POM) [**Bra04**], also known as iterative parallax mapping, relief mapping and interval mapping, is an iterative approach to wrinkled surface rendering. The algorithm uses mipmapped queries within a binary search over the ray that intersects the surface, to quickly and accurately find the intersection point. Because of the tracing nature of the algorithm a significant number of samples is sometimes required, but the computations are ameliorated because the process is projected in two dimensions, tracing the ray directly over the wrinkled surface heightmap. The algorithm was improved with **soft shadows**, efficiently implemented through adaptive sampling [**Tat06**]. Parallax mapping can be **offset limited** [**Pre06**] to mitigate the unpleasant "texture swimming" effects that take place at grazing view angles.

Relaxed cone stepping [**Pol07**] uses an expensive preprocessing step to compute the largest ray advance per pixel of the heightmap, making the tracing faster, but using a different type of map, named **cone step map**. A detailed analysis of wrinkled surface rendering is provided in [**Mik08**]. [**Nog12**] offers a comparative analysis.

Iterative wrinkle surface methods can be accelerated with the **secant surface intersection** technique, which finds a more accurate intersection point without using binary search. Instead, it performs iterative ray-segment intersections between the surface intersecting ray and the segment determined by the upper sampled bound and the lower sampled bound, as described in [**Ris07**].

Parallax mapping can be implemented in screen space, as shown in Screen space displacement mapping [**Lob08**] (SSDM). This technique projects the rendered normal onto the screen and then multiplies the projected normal with the displacement map value, creating a displacement vector. Then, a 3-4 level mipmap is created over the displaced vectors, over which each framebuffer pixel is displaced in an iterative process.

While the previously presented techniques were concerned with correct surface representation, correct surface shading also requires adequate **normal filtering**. The same problem of subgeometric detail is now applied to the normal and not the surface. The problems of geometric detail aliasing can easily be seen in high frequency lighting effects such as high specular reflection variation caused by incorrectly filtered normal maps. This effect is named shimmering or sparkling. Simple but ineffective or incorrect solutions are adaptive sampling, temporal coherent resampling, using tone-down functions to ease-out the effect over distance or even caching supersampled lighting in a virtual textured solution, basically enhanced lightmaps.

Linear Efficient Antialiased Normal (LEAN) [**Ola10**] Mapping is a filtering method which takes into account the problems of normal filtering. It permits normal map composition and offers correct results, but does so at a high storage cost, saving tangent space information per normal map. Cheap Linear Efficient Antialiased Normal (CLEAN) Mapping [**Bak11**] decreases the memory costs to half by sacrificing anisotropy support. A normal map specular aliasing solution is also explored in [**Tok04**].

2.2.4. Acceleration Structures

Independently of the data representation chosen and of the type of rendering algorithm, the very large numbers of objects found in rendering massive scenes bring about the need for faster space sampling. This is achieved through acceleration structures that subdivide space, which are used to query the scene and its objects in without accessing the raw data.

The rendered objects can be either static or dynamic in both form and topology. Static form and static topology objects are the easiest to render, because their acceleration structures can be precomputed. Objects with dynamic form or topology need to have their acceleration structures rebuild. If n represents the number of samples (triangles/voxels/points/etc) in an object than completely building an acceleration structure can't be performed in less than $O(n * \log n)$ complexity. Consequently, the acceleration structures for detailed objects with dynamic form or topology are extremely costly to maintain, especially in real-time rendering, where the computational budget is very tight.

Rendering acceleration structures work by either **partitioning the space that contains the objects** or by **partitioning the objects into sets**. While the obvious, most efficient, choice is a hybrid representation, sometimes either one of the previously presented approaches can be more efficient. Acceleration structures can also be classified by their **neighbor access efficiency**, since this aspect is very important in heavy neighbor sampling problems like simulations (cloth, fluids, etc) or scattering effects. Neighbor access efficiency isn't as important in real-time rendering as in these other computer graphics fields, because in general rendering samples space in predictable ray shaped patterns. Correct scattering effects are too costly to properly simulate in real-time rendering, therefore they are approximated with objects of uniform density, which can be better handled by implicit neighbor acceleration structures such as grids.

Another important aspect in acceleration structure design is construction direction: **bottom up**, **top down** or **hybrid clustering**. The temporal aspect of an acceleration structure can sometimes be very important, especially when it accelerates many dynamic objects. **Temporal coherence** friendly acceleration structures provide non-rigid support for short duration transitions, such as those found in object moving on a frame-to-frame basis.

The simplest acceleration structures used in rendering are **grids**. For example, the raster is a bidimensional hierarchic grid, and it is used as an implicit acceleration structure in many deferred algorithms [**Ols12**]. There are also specialized variants of grids like **perspective grids** [**Hun08**], which deform the grid. **Hierarchical grids** are largely used in many space partitioning problems. Grid representations are unreliable due to their finite precision of representation, and thus grid algorithms are in general forced to oversample. On the other hand, grids exhibit excellent memory coherency and can be used to easily access neighbors. **Hierarchical hash-grids** [**Sch09**] are hierarchies of grid-like structures that handle many objects of different sizes using hashed storage, keeping the grid $O(1)$ neighbor query and update complexities, while erasing the large memory requirements of grids, by storing only the buckets that contain data. **1.5D** and **2.5D grids**, also known as multilevel intervals and multilevel maps, are special types of grids, where one or more dimensions are normally sampled and the extra dimension is very sparsely sampled. This acceleration structure is often used in inexact intersection determination [**Har12**] or in the representation of multi-layer large objects such as vegetation or terrains.

Binary space partitioning into a hierarchical structure is done by adaptively subdividing space through arbitrary partitioning planes, which makes it very easy to surmount over empty space in grid-style space partitions. On the other hand such an object partitioning scheme is almost guaranteed to reference the same objects multiple times, increasing memory costs and especially memory management. **Binary Space Partitioning Trees** (BSP) [**Fuc80**] recursively subdivide the scene into convex sets of objects or triangles by using hyperplanes, and are then used with front-to-back or back-to-front rendering. Image space BSP is the projection of the BSP algorithm in 2D space.

Image space pyramids [**Had98**] subdivide space equally among a number of N predefined children. **Quadtrees** and **octrees** are popular hierarchical space subdivision structures, similar to image space pyramids. They subdivide space by partitioning each non-empty space area into 4 or 8 equal subspaces. Their sparse variants, the **sparse octree** [**Lai101**] and the **sparse quadtree** [**Sim12**] are often used to increase the efficiency of data storage, for example in [**Cra09**]. They achieve this by minimizing the storage for empty space. In contrast to KD-trees [**Moo91**], quadtrees and octrees do not subdivide optimally and thus they are dramatically less balanced space partitioning structures. On the other hand quadtrees and octrees can be implemented over grids, which offer quadtrees $O(1)$ neighbor access complexity.

Space partitioning schemes have problems in the accurate representation of small object which fall in unfavorable places for the space partitioning structure. For example, if an object or primitive falls at the border between two space partitioning nodes, even if it is very small in size it has to be stored either in the parent node or in both child nodes. Loose space partitioning can be used, in which the structure nodes have their collision bounding size doubled, which guarantees that any object will be stored in a node corresponding to its size. **Loose, fuzzy or dynamic octrees and quadtrees** are octrees and quadtrees that implement this principle. While this method greatly improves space partitioning management for dynamic scenes, it comes with increased processing costs caused by overlapping computations.

Kd-trees [**Ben75**] [**Moo91**] are very similar to BSP trees in concept, but their partitioning planes are always perpendicular to one of the canonical k-dimensional axes. In contrast to octrees and quadtrees, the Kd-trees do not subdivide space in equal subspaces, but try to subdivide space in order to maintain a balanced tree. The implicit Kd-tree is a variant of Kd-tree which is defined over a bidimensional grid. Min-max Kd-tree is another variant of Kd-tree, where each node in the tree contains the minimum and maximum extents of its children. Quadtrees and Octrees can be considered particular types of multiple plane splitted KD-trees. Kd-trees are usually constructed with a space partitioning heuristic. **Surface area heuristic** (SAH) [**Wal06**], is a space partitioning heuristic for Kd-trees, which utilizes a greedy heuristic function, in order to determine where to position the splitting hyperplane. The **binned SAH** (b-SAH) [**Dan10**] uses a binning method to sample potential KD-tree splitting planes, decreasing the time to construct a SAH KD-tree.

Partitioning object lists is a different approach to acceleration structure design. It combines clustering and top-down or bottom-up design to create object lists, which are then incorporated in a hierarchical structure. Since each object is referenced at most once, memory management is predictable and simpler. Furthermore, this dual approach permits optimizations for both high-level nodes, which are expected to contain general bounding data about the scene, and for the low-level nodes, which will normally contain objects. Because partitioning object

lists bundle objects or primitives together, they and are not as rigid as Kd-trees and grids, and they are favored for dynamic environments.

The **bounding volumes** (BV) for the low-level clustered nodes can either be axis aligned bounding boxes (AABB), object oriented bounding boxes (OOBB), discrete oriented polytopes (DOP), maximum bounding rectangles (MBR), convex hulls, spheres or capsules. In rendering, the AABBs are the most popular type of bounding structure.

Object trees are generalizations of balanced binary-trees. B-tree nodes are trees that have multiple children. Object R-trees cluster groups of elements into k-dimensional rectangles, which represent the minimum bounding rectangle. Object Interval Trees is an ordered data structure that holds intervals, making it easy to query all intervals that overlap a certain point.

Bounding volume hierarchies (BVH) are tree structures that wrap elements into bounding volumes, as shown in Figure 7. These volumes represent the trees' leafs, which are then recursively wrapped to create the rest of the tree. BVH trees have a large number of applications in rendering intersection problems such as culling and in tracing global illumination algorithms. Like Kd-trees, BVHs can be optimized with SAH [**Wal072**], but they still suffer from overlapping which makes them slightly inferior to KD-trees in the traversal of large scenes. **Early Split Culling** (ESC) [**Ern07**] relaxes the requirement that each primitive must be only once referenced by a BVH node and produces smaller bounding boxes by splitting all inconvenient primitives, which lead to faster BVH traversal. Compared to ESC, the **Edge Volume Heuristic** (EVH) [**Dam081**] splits only very expensive primitives, and provides more stable and less optimized results compared to ESC. The **SBVH** [**Sti09**] [**Pop09**] uses spatial splits on primitives in order to minimize the overlapping between children nodes, but it remains an object list partitioning spatial subdivision structure because it splits space only at a per object level. It can be considered as a BVH built with a SAH function that penalizes overlapping, as described in [**Pop09**], and it has superior results compared to ESC and EVH.

There are many simple variants of the BVH such as the popular AABB-BVH, known as an AABB tree or **bounding box tree** or hierarchical box trees, which use AABBs as BVs. Another simple variant is the quad-BVH (**QBVH**) [**Dam08**] which uses four BVHs bundled together and thus can easily be optimized for CPU SIMD. An AVX variant friendly of the BVH tree is a generalization of QBVH, the Multi Bounding Volume Hierarchies (**MBVH**) [**Tsa09**], which performs multiple intersection tests per tree node. MBVHs can be optimized for weak coherency problems by bundling different BVHs with a space partitioning heuristic. An interesting comparison between BVH and KD-trees for many-core architectures is given in [**Vin14**].

Spatial KD-trees, also named **SKD-trees** [**Ooi87**], are similar to KD-trees but have 2 splitting planes per node in order to guarantee that each object is referenced just once, which can be either overlapping or disjoint. Because this is done through a splitting axis and a single value, which defines the two splitting planes relative to the parent bounding, the SKD-tree is more memory efficient than normal KD-trees. On the other hand, the overlapping caused by nodes makes SKD-trees inferior to KD-trees in traversal efficiency. Because of its properties the SKD-tree is not a spatial subdivision structure but an object subdivision structure. The SKD-tree is also called a **BoxTree** [**Zac02**]. There is also a bitwise compressed variant of SKD-trees named Bounding Interval Hierarchy [**Wäc06**].

The **H-tree** [**Hav06**] is a hybrid object subdivision structure that improves the empty space handling of the SKD-tree. The H-tree has both SKD-tree nodes and BV nodes, and during its construction always chooses the type of node based the projected better bounding. BV nodes are often used in a H-tree to accelerate empty space traversal. The AH-tree is a variant of the H-tree that can be constructed in $O(NloglogN)$ for similar sized objects, with a worst case complexity of $O(NlogN)$ for general objects.

Bounding Interval Hierarchies (BIH) [**Wäc06**] is a low bandwidth bounding volume hierarchy. Similar to SKD-trees, BIH stores two splitting planes per node, therefore instead of storing the BV (e.g. AABB) for each child, like a normal BVH would, the BIH stores only two bounding boxes through shrewd bitwise codification. The nature of the BIH node and the clipping axis are encoded in binary. The nodes of a BIH contain three integer numbers. The first one contains the node type and clip axis encoding. The second and third integers store the clipping planes displacements, in the case of a normal node, and pointers to the children, in case of a leaf node. This structure makes the traversal of a BIH to be identical to that of a SKD-tree. Because the clipping axis and the clipping value can determine 3 subvolumes, the extra case of empty space has to be considered when intersecting the BIH with a ray, therefore, like SKD-trees the BIH theoretically has a slightly inferior traversal efficiency compared to that of the KD-tree. Conversely, due to its very small memory footprint, the BIH traversal is very close to that of KD-trees. BVHs can be constructed fast on the GPU [**Lau09**], which improves their usefulness for dynamic objects. Bottom up and top down construction of BVH is described in [**Wal071**].

Both space partitioning and object partitioning trees can be used as acceleration structures for **tree traversal**, which can be done with or without a stack. GPU implementations work better with stackless traversal, as memory allocation and cache coherency aren't GPU strengths. [**Fol05**] and [**Pop07**] present stackless traversal strategies for the kd-tree space partitioning structure. Object partitioning tree traversal has also seen a lot of research recently: trail-restart traversal [**Lai10**], parent links [**Hap11**], multi-BVH restart-less bitmask traversal [**Afr14**] and ray-stream traversal [**Bar14**]. Stackless traversal is deeper analyzed in Chapter 4.2.1. Tree traversal without acceleration structures [**Mor11**] [**Kel11**] [**Nab13**] [**Afr12**] is a recent trend in ray tracing, in which the traversal dynamically constructs the acceleration structure during rendering.

Tree traversal of **dynamic objects** is more complicated. While static objects need not have their spatial subdivision structures reconstructed, dynamic objects need proper handling. Dynamic structured objects, such as those perfectly defined by any kind of tree can be handled easily in ray intersection cases by inversely transforming the ray and then performing the intersection. While this method increases the ray-structure intersection cost, it does not force the reconstruction of the nodes. Unstructured dynamic objects, which can't be perfectly represented through trees, are best handled in a separate subdivision acceleration structure. This tree can either be reconstructed per frame, or just partially, on demand, like in the method described in [**Wal03**].

The **AH-tree** [**Hav06**] is an improvement over the H-tree, which decreases construction complexity from $O(NlogN)$ to $O(NloglogN)$ for limited size objects, while retaining $O(NlogN)$ for worst case scenarios. On the other hand, the AH-tree is inferior to the H-tree in traversal efficiency.

Min-max variants of standard acceleration structures can be used as interval trees, which can be used in any heavy sampling process to quickly approximate space. Dictionaries and

spatial hash structures are often used in the construction of spatial subdivision structures, but they are rarely used in the rendering process. Other acceleration structures such as Voronoi diagrams, Layered Depth Images [**Sha98**] or Thick Layered Depth Images [**Rad14**] which are useful for fast neighbor determination and many screen space collision algorithms have rarely been used in real-time rendering. A comparison between acceleration structures commonly used in rendering is presented in Figure 7.

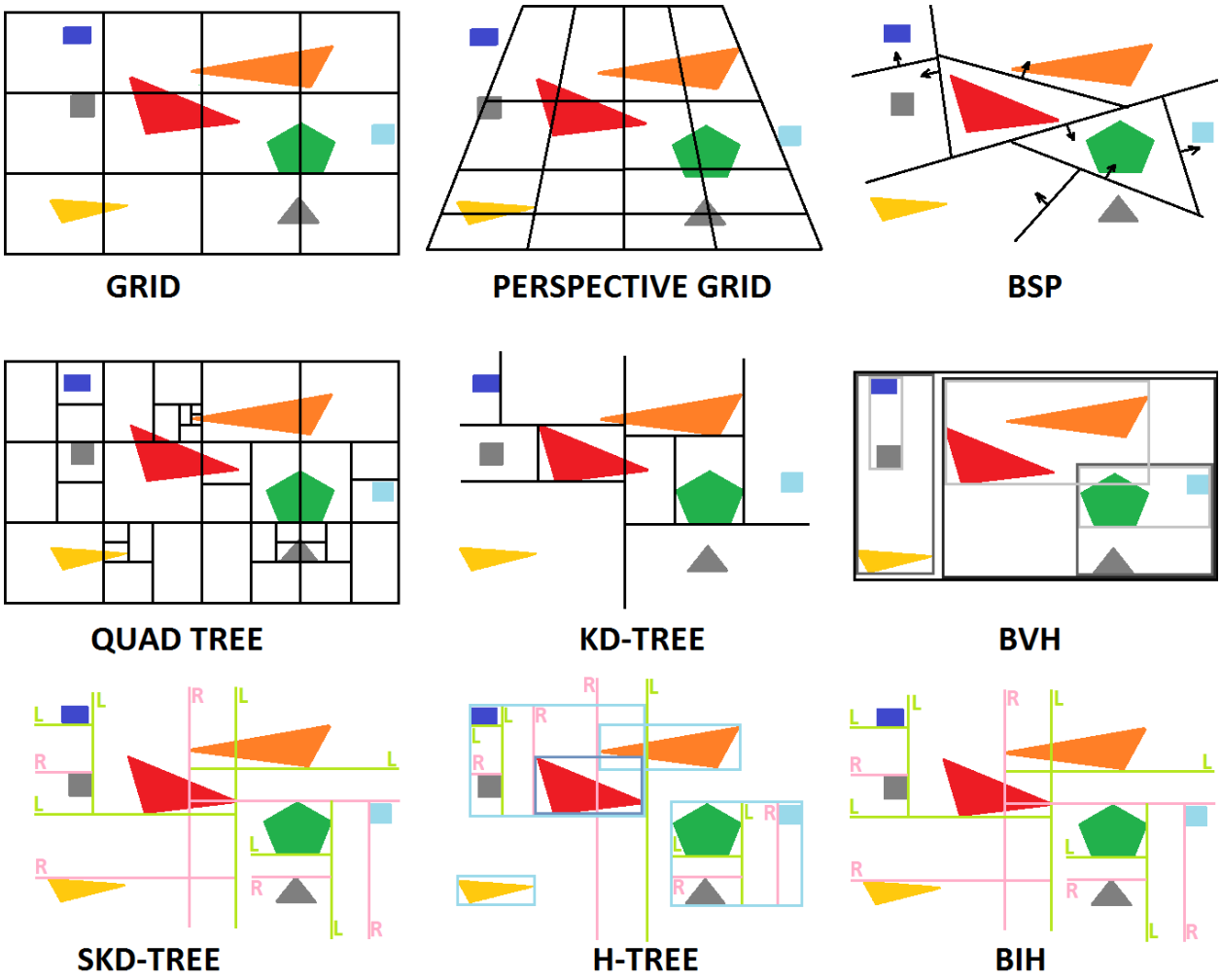

Figure 7 Acceleration structures**.** In rendering, space sampling is accelerated, usually with one of the depicted acceleration structures. Acceleration structures can be classified into two types: space partitioners and objects list partitioners. Space partitioners subdivide space efficiently, but suffer from increased memory usage because the objects are referenced more than once. Grids are the simplest and most inefficient type of acceleration structure. Perspective grids importance sample nearby space and can are especially important in planetary and terrain rendering. Binary Space Partitioning (BSP) trees divide by general splitting planes. Quadtrees and octrees divide space in 4 or 8 equal parts, but only when necessary. KD-trees are BSPs limited to axis parallel splitting planes. Object list partitioners guarantee that each object is referenced only once, therefore have lower memory requirements, but suffer from overlapping and less efficient traversal. BVHs bundle objects together in a hierarchic tree, but suffer from overlapping. SKD-trees, also called BoxTrees, are KD-trees with two parallel splitting planes per node, which also differ from KD-trees by being object list based, because they guarantee that each object is referenced only once. H-trees are a hybrid between SKD-trees and BVH, using either bounding volumes or SKD nodes, whichever offers better space culling per node. Bounding Interval Hierarchies (BIH) is a SKD-tree which benefits from shrewd bit compression.

2.3. Rendering Concepts

Rendering algorithms are just models that simulate light transport. While some rendering algorithms are more physically correct than other, no rendering algorithm is perfectly physically correct. The major reason for this is that many light transport phenomena are very hard to perceive for human beings, therefore investing a large budget of computational resources in to rendering them would be inefficient from a visual standpoint. Another reason is that perceptible light transport can depend on energy transport specific to effects usually not simulated in rendering, such as heat glowing or photosynthesis.

Drawing is performed by sampling the scene through a rendering algorithm, which creates the final rendered image by accumulating light on the photo receptors of the **virtual camera**. The virtual camera is described as a collection of virtual **photo sensors** which capture light coming from the scene. These camera sensor-light interactions are then sampled and filtered to determine the pixel colors. The scene can be sampled through a number of different light transportation means, which include projections, rays, photons or paths, which are used to transport light from the light sources to the camera.

These means can roughly or exactly simulate light-object direct and indirect interactions. The interaction between light and an object is defined through a material, which describes how the surface absorbs, transmits, reflects or emits light. The fundamental science used in rendering is **radiometry**, which studies optical radiation. Compared to photometry, it studies electromagnetic radiation beyond the human visible spectrum, which is represented by wavelengths between 360nm and 830nm.

In an idealized light transportation problem in rendering, energy is transferred as wave from a given light source to the objects of the scene. The energy, called **Radiant Energy**, is measured in Joules (J) and defines the amount of light produced by a surface in give amount of time. Since in rendering the purpose is to synthesize images and not to measure the scene energy, a more useful measuring instrument is Radiant Power, also named **Radiant Flux**, Φ. Radiant flux measures the flow of radiant energy transferred through a surface in a unit of time, thus it can be written as :

$$\Phi = \frac{dQ}{dt}$$

Radiant Power is measured in Joules per second ($\frac{J}{s}$), equivalent to Watts (W). **Irradiance** measures how much Radiant Energy is coming to, passing through or emerging from a surface, per unit of surface, per unit of time. It is measured in Watts per square meter ($\frac{W}{m^2}$), and can be defined as:

$$E = \frac{d\Phi}{dA} = \frac{\partial^2 Q}{\partial t\, \partial A}$$

Radiant Intensity, or Intensity, represents the angular density of Radiant Flux per unit solid, angle, measured in Watts per steradian ($\frac{W}{sr}$), defined as:

$$I = \frac{d\Phi}{d\omega} = \frac{\partial^2 Q}{\partial t\, \partial \omega}$$

Radiance is the area and angular density of radiant flux per unit projected area, per unit solid angle passing through, received at or emitted from a specified direction, on a point, on a surface. It is measured in Watts per square meter per steradian ($\frac{W}{m^2 sr}$) and defined as:

$$L = \frac{\partial^2 \Phi}{\partial \omega \, \partial^{\perp} A} = \frac{\partial^3 Q}{\partial t \, \partial \omega \, \partial^{\perp} A}$$

Since radiance is defined as a per direction function, it does not vary with distance. Radiance coming at a surface is called incident radiance, while radiance leaving a surface is called exitant radiance.

Reflection between incident radiance and a surface is handled through **bidirectional reflectance distribution functions** (BRDF). The BRDF can be defined as:

$$f_r(x, \omega_i, \omega_o) = \frac{dL_r(x, \omega_r)}{dE_i(x, \omega_i)} = \frac{dL_r(x, \omega_r)}{L_i(x, \omega_i) cos\theta_i d\omega_i}$$

In order to be physically realistic, a BRDF must be positive, must obey Helmholtz reversion-reciprocity principle and must conserve energy. Positivity means that a BRDF has to reflect some of the incoming radiance. The Helmholtz reverse-reciprocity principle states that if the direction of propagating light is reversed the same optical rules apply. Energy conservation states that a surface can't reflect more than what it can receive.

Similar to the BRDF, the bidirectional transmittance distribution function (BTDF), models the transmission of light through a surface. The bidirectional scattering surface reflectance distribution function (BSSRDF) generalizes the BRDF and models the reflections that take place when light interacts with a scattering reflection only material. The bidirectional scattering distribution function (BSDF) is a further generalization of BTFD and BSSRDF.

Such functions are not analytical, but can be approximated through **lobes**, which need to be measured with special instruments [**Nga05**]. For real-time rendering, these lobes can usually be simplified to mirror reflection, glossy reflection and diffuse reflection, and have been approximated by a large number of analytical and approximate models such as: Lambert [**Edw03**], Torrance-Sparrow [**Tor67**] Blinn and Blinn-Phong [**Bli77**], Cook-Torrance [**Coo82**], Minnaert [**Min41**], Ward [**War92**], Schlick [**Sch98**], Oren-Nayar [**Ore94**], Heidrich-Seidel [**Hei98**], Ashikmin [**Ash00**], Kelemen-Kalos [**Kel01**], GGX [**Wal07**], Wrap [**Slo11**] or GTR [**Bur12**]. A survey of existing distributions and how to combine them is presented in [**Sch11**].

The rendering process can be described as a recursive light transport equation, known as the **rendering equation**, as defined by Kayija in [**Kaj86**]:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\Omega} f_r(x, \omega_i, \omega_o) L_i(x, \omega_i)(\omega_i \cdot n) d\omega_i$$

In this equation the result $L_o(x, \omega_o)$ represents the outgoing radiance in $\omega_o$direction. It is the sum between the emitted radiance in that direction, $L_e(x, \omega_o)$, and the incoming radiance on the entire hemisphere Ω, $L_i(x, \omega_i)$, weakened by the incident angle, $\omega_i \cdot n$, and then reflected by the bidirectional reflectance distribution function, $f_r(x, \omega_i, \omega_o)$.

While traveling through non-void media light suffers from scattering events, because the light's photons interact with the particles from the media. In such media the light can be absorbed, emitted, in-scattered or out-scattered. The equation for non-homogenous media light propagation can be written as a sum of volumetric emission, absorbed in-scattering, out-scattering (extinction) and absorption:

$$L_o(x_o,\omega) = L_{ve}(x_i \to x_o,\omega) + \int_{x_i}^{x_o} \{\sigma_s(x,\omega) \int p(x,-\omega \to \omega') L_i(x,\omega') d\omega' \, (e^{-\int_x^{x_o} \sigma_t(x,x+t\omega)\, dt} + e^{-\int_x^{x_o} \sigma_a(x,x+t\omega)\, dt})\}\, dx + L_i(x_i,\omega)\, (\, e^{-\int_{x_i}^{x_o} \sigma_t(x,x+t\omega)\, dt} + e^{-\int_{x_i}^{x_o} \sigma_a(x,x+t\omega)\, dt})$$

The in-scattering and out-scattering are based on phase functions, which represent the probability distribution for scattering based on solid angle. They are similar to BSDF function, but are defined for scattering media. Some of the most used distributions are isotropic and Henyey-Greenstein [**Hen41**].

$$p_{isotropic} = \frac{1}{4\,\pi}$$

$$p_{Henyey-Greenstein} = \frac{1}{4\pi} \frac{1-g^2}{\left(1+g^2-2g(cos\theta)\right)^{1.5}}, \quad g - assymetry\ parameter$$

The Henyey-Greenstein is an anisotropic distribution and is accurately approximated by Schlick [**Sch93**]:

$$p_{Schlick} = \frac{1}{4\pi} \frac{1-k^2}{(1-kcos\theta)^2}, k = 1.55g - .55g^3$$

The light transports simulated in rendering are very diverse, and in order to be sampled efficiently they need specific sampling strategies.

The **Heckbert notation** [**Hec90**] considers the entire path traversed by the photons, from the emitting light to the camera receptor, and it is used to easily differentiate between different types light transfers. It uses regular expressions for path labeling, thus all paths can be written like $l(s|d)^*e$, where l represents a light, s a specular reflection, d a diffuse reflection and e the eye (camera).

For example le describes a path between a light and the eye while $lsse$ describes a path between the eye and a light with two specular reflections, such as the paths that produce caustics.

The glossy reflection g was added later, which represents a specular-like diffusion, which reflects light in a lobe, instead perfectly. Thus, a path can also be described as $l(s|d|g)^*e$.

2.4. Rendering algorithms

A rendering algorithm simulates light transport by determining the interactions between lights, objects and the virtual camera photo sensors, represented as pixels. These rendering interactions can be simplified to simple, **direct interactions**. This follows the principle of global illumination, where all the objects light the other objects. The fundamental operation in determining the validity of direct interactions is **visibility determination**. This concept was first explored as a visibility determination problem [**Sut74**], where the visibility determination operation is considered the most basic and most important operation for rendering algorithms.

Thus, rendering interaction can be expressed as an unordered set of visibility determination operations, and efficiently simulating the interactions is equivalent to efficiently computing the generated visibility operations, which is a **visibility determination sorting** problem. This in turn can be described as a searching problem. This aspect of rendering is thoroughly presented in [**Hav14**]. Consequently, correctly sorting and then solving the generated visibility operations is the most efficient method to simulate rendering interactions, as highlighted in [**Sut74**] and [**Hav14**] : "Sorting and searching usually takes more than 90% of the rendering time". Different rendering problems are presented as searching problems in Table 2.

Rendering Problem	Query Domain	Search Space	Answers Domain
Ray Shooting	rays	objects	(intersection) points
Hidden Surface Removal	rays	objects	points
Visibility Culling	rays	objects	objects
Photon Maps	points	points	points
Ray Maps	points	rays	rays
Irradiance Caching	points	spheres	spheres
Path Tracing	paths	objects	(intersection) points

Table 2 Rendering as sorting**.** Different rendering problems are depicted as search problems. Ray shooting algorithms like ray tracing search the scene objects with rays and produce intersection points. Hidden surface removal methods search visible points on objects by ray queries. Visibility culling uses rays to search the object space for visible objects. Photon maps use k-NN point queries into the point photon space, to determine which points affect a point directly visible from the camera. Ray maps use point queries into a ray-indexing structure to determine rays which are followed by photons. Irradiance caching uses points to search in the sphere space to determine which spheres are cached. Path tracing searches the object space with paths, determining intersection points.

This sorting process is actually a **scene sampling process**, where the scene objects, lights and camera sensors are sought to be sampled in the most coherent order. These different types of coherencies can be observed in different fundamental rendering problems: scanline (e.g. raster, texturing), frame (e.g. temporal reprojection), object (e.g. back face culling, clipping culling), depth (e.g. z-buffer, DFS tracing), list-based (e.g. BFS tracing, raster), face and edge coherence (e.g. raster, BVH) or area coherence (e.g. photons, texturing). Rendering algorithms try to capitalize on most of these coherencies to improve operations ordering and early decisions (e.g. early-z, use of acceleration structures). Since different lighting effects are caused by different phenomena, the majority of rendering algorithms use multiple coherencies, in effect **multiple importance sampling visibility operations**. Therefore, rendering algorithms are **interactions samplers**, not solvers, even though they use the results of previous interactions when sampling for other interactions. The interactions are simulated in a physically plausible manner through the use of BRDFs, BDTFs BSSRDFs or BSDFs, which are beyond the scope of rendering algorithms.

Rendering algorithms are differentiated by the type of light paths they can simulate. All rendering algorithms can handle direct light paths, which contain at most a single light-object interaction. These algorithms are named **direct illumination** algorithms. Another name for them is **local illumination** algorithms, because they require only local information in the rendering process. The best rendering quality is found in **global illumination** algorithms, which handle all types of light paths. Because such algorithms use scene wide information and light paths with many light-object interactions, they are also called **indirect illumination** algorithms.

The ultimate goal of rendering algorithms is to efficiently create the necessary objects-lights interactions required for generating photorealistic images. In some cases, rendering techniques can combine aspects from different rendering algorithm families, and can be equivalent from a result standpoint. Rasterization with customizable sampling points per pixel is equivalent to ray casting, with the exception that ray-casting also provides the intersections in a "front to back" order. Rasterization can provide the exact same order or produce the same stochastically expected result through the use of Order Independent Transparency (OIT) algorithms like [**Bar11**] [**End10**]. Even if the viewport is non-planar there are methods which make rasterization equivalent to ray casting [**Dav121**].

Ray tracing samples space by shooting rays, path tracing samples space by using paths, cone and beam tracing sample space by shooting volume rays and photon mapping samples space by shooting and storing photons. Each of these space sampling strategies is aimed at solving the same problem, that of global illumination. Some of the space sampling strategies aim to optimize for the first intersection (rasterization), diffuse reflection (many lights methods), specular light transport (photon mapping), or for maximizing lights-objects interactions (path tracing). Because of these different aims, some rendering algorithms perform better than others, depending on the scene and the degree of the correctness of the transport of light.

This space sampling process was first formally represented in the rendering equation, introduced by Kayija in [**Kaj86**]. It formalized the rendering problem as an order dependent, interaction determination and evaluation process.

The most important properties of the rendering equation are that it is **recursive** and **separable**. Because it is separable, different algorithms can be used to evaluate its different parts, for example rasterization can be used for the emissive and direct light reflection components, while a more cache unfriendly but more efficient space sampling algorithm like path tracing can be used to compute the rest of the equation. This is the best approach to obtain good performance, because each stage of the rendering equation is handled by the most efficient algorithm for it.

A rendering algorithm is said to be **physically correct** if it generates the lights-objects interactions necessary for the generation of a photorealistic image. Therefore physical correctness in rendering is not in a physics connotation but in a perception one.

A rendering algorithm is called **consistent** if, with enough sampling, it produces a correct expected result. A rendering algorithm is called **unbiased** if it does not introduce any regular error in the radiance approximation. Biased algorithms do not necessarily produce wrong results, as they can converge with enough sampling to the correct result, if they are consistent. On the other hand biased algorithms introduce an error called bias, which can be perceived as a blur or as a loss of high frequency visual information. This is usually done to reduce the variance found in sampling hard to sample rendering problems, like caustics from point light sources.

One of the most important aspects of algorithm design is data handling. Most **direct rendering** algorithms use preprocessed data structures, which are rarely modified during rendering. The most common data structures that suffer modifications are tree structures like BVH, and even if their reconstruction has seen improvements in on-GPU generation [**Lau09**], they are still rarely used because of the tight computational budgets in real time rendering. Likewise, **indirect rendering**, where the rendered data is converted from one format to another, is rarely used in real-time rendering because of its costs.

On a more fundamental level, **data traversal patterns** are by far the most important data-related performance topic in rendering algorithms. Because all rendering algorithms are sampling processes over the scene space and because GPU bandwidth is almost always the main bottleneck, the most coherent sampling process is expected to have the best performance. This is a major reason for the wide-spread adoption of the rasterization rendering algorithm.

Rendering algorithms can also be distinguished by design. **Top-down** approaches integrate the most relevant majority of visual effects and need no other additional techniques to correctly synthesize the rendered images. **Bottom up** approaches need additional algorithms to provide good final results, but are much more scalable in computational cost.

Based on how they implement and sort visibility determination operations, through various space sampling strategies, the many rendering techniques in computer graphics can be separated into algorithm families: rasterization, screen space techniques, Reyes, ray casting, ray tracing, path tracing, photon mapping and many light methods. While there is room for variation for the space sampling strategy used by a rendering family, the basic principles are usually the same. Some of these space sampling strategies are presented in Figure 8.

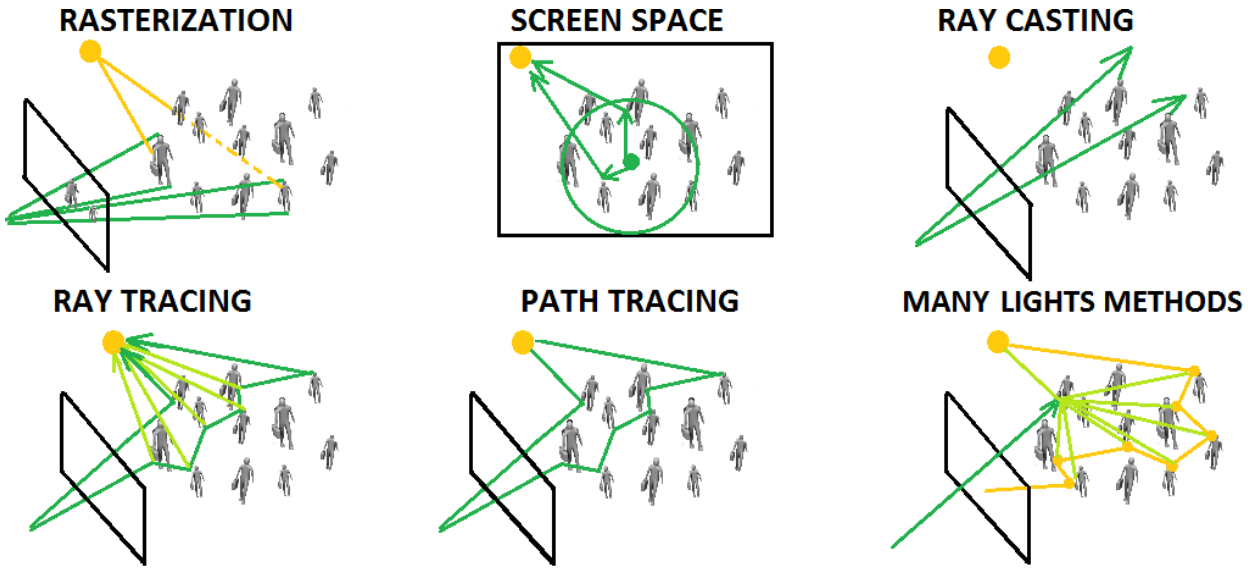

Figure 8 Rendering algorithms**.** Rendering algorithms provide the interactions between objects, lights and screen, by using different sampling strategies, which exploit different visibility determination coherencies. Rasterization projects space onto the screen, solving visibility determination operations with a bidimensional grid. Screen space methods work in 2D, where neighbor sampling and mipmaps can be used to hierarchically determine approximate visibility. Ray casting uses rays to determine visibility of objects on the screen. Ray tracing uses rays recursively, sampling space beyond the first screen-object interaction. Path tracing considers the entire transport of light as paths, which leads to many importance sampling opportunities. Many light and photon mapping methods transport light throughout the scene by creating many small lights, named virtual lights (VL), which simulate light transport and are then used as direct illumination sources.

Rasterization algorithms intersect each rendered primitive by projecting it onto the pixels of the screen. Their results can be used by screen space techniques to produce low quality approximations of global illuminations. Reyes [**Coo87**] rendering takes each primitive, splits it into smaller primitives which are then diced to micropolygons close to the size of a pixel. The micropolygon vertices are shaded and then sampled to create the final color for each pixel.

Ray casting [**App68**] generates rays, going from the camera through each of the pixels of the screen, which are used to intersect the scene. The final pixel color is obtained through the accumulation of the color contributions from all the intersections. Ray casting is usually employed in rendering transparent objects and scientific visualizations [**Wei06**]. Ray tracing [**Whi79**] algorithms start like ray casting, with rays starting from the camera and going through the pixels of the screen, but differ from ray casting because they can generate new rays when the old intersect objects. Path tracing [**Kaj86**] takes the concept of ray tracing further, by linking the new rays with the old rays into paths.

Photon mapping [**Jen96**] decouples lights-objects and camera-objects interactions by shooting a large number of photons from each light and simulating the light transport with them. After the photons are transported they are saved into a scene wide photon storage structure, a final gather stage is used in which rays shot from the camera through each pixel intersect the objects of the scene. At each intersection the lighting equation is evaluated, using the photons stored in that area.

Many lights based methods, also known as virtual lights (VL), [**Kel97**] approximate correct global illumination by generating virtual lights at each light-object interaction. The virtual lights are then checked for potential interactions with the objects of the scene, recursively spawning new virtual lights. All virtual lights are used for direct illumination in a final gathering stage.

2.5. Rasterization

Of all rendering algorithms, rasterization simulates the smallest subset of light paths, sampling only $l(d|s|g)^{?}e$ paths. On the other hand, it samples these paths with excellent data locality. Because of this, rasterization based renderers need to employ a large number of additional techniques to produce quality images, but, if implemented correctly, a rasterization based renderer can produce photorealistic results. The complexity of managing all the additional techniques makes rasterization renderers scalable, albeit difficult to maintain solutions. Thus, rasterization can be considered a **bottom-up approach** to rendering, compared to ray-tracing or path-tracing algorithms, which are top-down approaches.

The scene sampling process is based on a primitive-pixel intersection operator, which loads each primitive in memory and intersects it in parallel with many pixels. Therefore, rasterization is a SIMD friendly algorithm, which can easily be implemented in graphics drivers, and this makes it the best method for determining direct visibility. While culling algorithms give it the same visibility determination complexity of $\log num\_primitives$, identical to other rendering algorithms like ray tracing or path tracing, rasterization exhibits the best general performance in rendering visible objects. The only unfavorable performance case happens when micropolygons are rasterized, because of low numbers of potential fragments, generating rasterization tasks with few threads. Level of detail algorithms can be used to mitigate the

number of sub-pixel sized primitives, by rendering primitives of a suitable size. The **data locality** in rasterization permits the algorithm to efficiently access memory for a large number of pixels or sub-pixels, making hardware anti-aliasing practical.

Rasterization is also fully compatible with tile-based architectures, thus it is favored in energy deprived environments. Since the raster is a grid structure, tiles can easily be integrated within current hierarchical **early rejection** algorithms [**Gre93**] [**Joh05**] used in modern rasterization. The negative side of rasterization is that it generates only a small number of light-object interactions, making it impossible to implement correct global illumination without further complexity. Rasterization provides the direct interactions between camera and objects without order, making it difficult to represent basic phenomena such as transparency or translucency.

2.5.1. Visibility and Occlusion Culling

Because rasterization projects objects onto the screen it provides potential object-screen interaction **without order** and without bounds, therefore extra work is usually performed for occluded objects and for objects outside the visualization volume. Since rasterization uses the **Z-buffer** algorithm for opaque objects, it uses an implicit form of culling called depth rejection. Depth rejection ensures that each new fragment generated by a uniform distribution of objects has statistical chance of $\frac{1}{K+1}$ to not be culled, where K is the number of previously existing fragments. For a large number N of fragments per pixel the number of fragments that will pass the culling test grows like the harmonic series to $1 + \sum_{i=1}^{N} \frac{1}{i} < \ln(N) + 1$ [**Coz09**]. This complexity is further lowered with hierarchical depth rejection [**Gre93**] and guarantees the speed of rasterization, since the majority of computational costs is in fragment processing. Hardware vendors have brought many efficiency improvements to the Z-buffer algorithm like early rejection, double speed depth-only rendering or depth compression [**Coz09**].

Thus, the rasterization Z-buffer algorithm has a **linear geometry processing complexity** and a **(sub)logarithmic fragment processing complexity**, both of which have close to optimal data access coherency. Even with this large number of optimizations, the Z-buffer algorithm can suffer from floating point precision artifacts, which can be countered with **multiple depth frusta** [**Coz09**], judicious minimum triangle separation and LODs.

Culling algorithms cheaply determine visibility, with the purpose of minimizing this extra effort. Culling is itself a sampling process, as it samples the entire scene to find the best set of potentially visible objects which will be then sent to be rendered on the monitor. Because of occlusion, an exact match can't be obtained without interleaving the culling algorithm with rendering [**Mat15**].Static culling methods like potentially visible sets (PVS) [**Dur99**], portals and anti-portals [**Dur99**] can be precomputed, or dynamically computed by amortizing the computation on multiple frames, and have all seen wide adoption.

Hierarchical occlusion determination algorithms work by temporal information and visibility queries. Coherent Hierarchical Culling (CHC) [**Bit04**] uses information from the previous frames to test only a limited number of queries per frame. The algorithm was further refined in Near Optimal Hierarchical Culling (NOHC) [**Gut06**] and in CHC++ [**Mat08**] with statistical methods that bundle queries in order to minimize their costs. It was also adapted for ray tracing in CHC++RT [**Mat15**]. Culling algorithms have been particularized for special rasterization problems like shadow rendering in [**Llo04**] [**Bit11**].

Fast **inexact** culling algorithms are usually based on easy to obtain rasterization information like the depth buffer and other common screen space data. Hierarchical Z Visibility [**Gre93**] is integrated in the rasterization hardware and it has also been adapted in a programmable version where occlusion determination is based on testing the visibility of extended screen space projected AABBs against the mipmaps of the depth buffer from the current or previous frame.

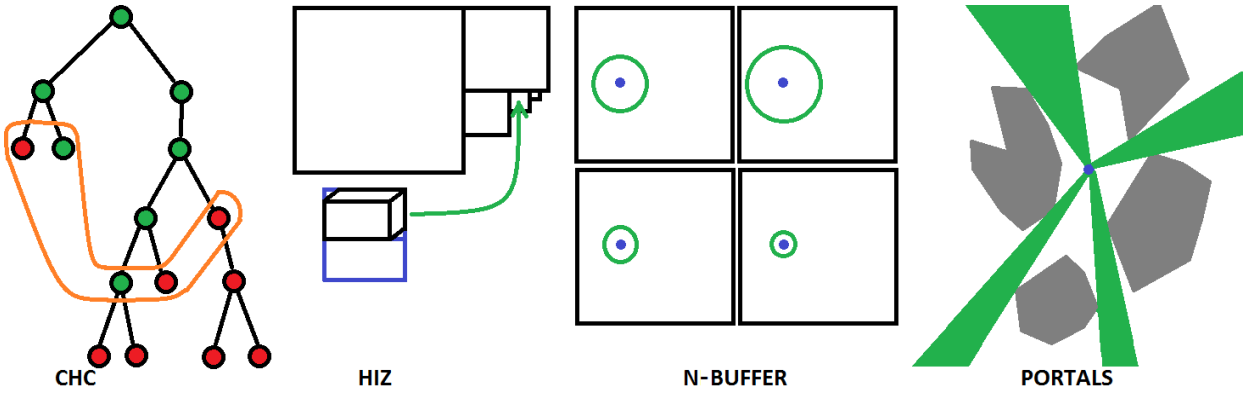

Figure 9 Culling algorithms**.** Coherent Hierarchical Culling (CHC) variants are exact and test for visibility only what might vary from the previous frame. Hierarchical Z (HIZ) methods extend AABBs in screen space and Z-test the visibility in a hierarchical depth buffer. Compared to HIZ, N-buffer stores exponentially increasing maximum depth vicinities, therefore it switches the HIZ hierarchy of maps into a map of hierarchies. Portals use multiple view frustums to reduce the visible space.

The concept of using the projections of **occluders** in a 2D occlusion process has also been tackled in Visibility Culling using Hierarchical Occlusion Maps [**Zha97**], where groups of low resolution LODs of occluders generate occlusion maps, which are then mipmapped and used like additional Z-buffers for visibility determination. Using multiple Z-buffers has also been explored in Triple Depth Culling [**Mar11**], where the depth maps are filled in alternating geometry passes. Instead of storing a hierarchy of maps **N-buffers** [**Déc05**] introduces a map of hierarchies, where each level of the N-buffer contains the maximum depth for an exponentially increasing vicinity. Hardware tessellation culling [**Nie12**] is a relatively novel field, currently with no significant applications in real-time rendering. [**Bar12**] contains a review of image space culling algorithms. There have also been culling hardware proposals [**Has07**], but as of now, none are officially used in consumer hardware. Some culling approaches are showed in Figure 9.

2.5.2. Geometric Antialiasing

Compared to algorithms that use rays or paths to synthesize the final image, rasterization does not have the option of analytic sampling per pixel. Therefore rasterization antialiasing has seen a large number of methods that try to minimize **geometric aliasing**, which is caused by the fixed sampling rate and pattern of the projected geometry. Super sampling antialiasing (SSAA), also called full scene anti-aliasing (FSAA) linearly increases the number of samples taken from the raster in order to create pixels. Multisampling antialiasing (MSAA) also does this, but it only oversampling the visibility attributes of the geometry.

Fast Approximate antialiasing (FXAA) [**Lot09**] finds all edges in a post processing stage, and then blurs them, to non-uniformly reduce high frequency detail and the aliasing caused by it. Morphological antialiasing (MLAA) [**Jim11**] finds all edges in a post process, and then tries to

match sub-edge parts to hardcoded cases, which can then be anti aliased efficiently. MLAA is refined in Subpixel Morphological antialiasing (SMAA) [**Jim12**] with better diagonal lines support and a more accurate pattern matching mechanism. SMAA uses a precomputed texture to correctly handle diagonal patterns and a velocity-weighted temporal reprojection [**Neh07**] mechanism. The temporal reprojection is a case of amortized supersampling [**Yan09**], which is used to increase the number of samples for moving objects, which are very difficult to antialias.

Subpixel reconstruction antialiasing (SRAA) [**Cha11**] is targeted at deferred renderers, and combines sub-pixel visibility through supersampled G-buffers with single-pixel shading. Subpixel reconstruction antialiasing (RSAA) [**Res12**] stores the results of geometric sampling as binary results in a mask, which are then resampled with optimally precomputed coefficients for each combination that can be stored in the mask. Aggregate G-buffer antialiasing (AGAA) [**Cra15**] decouples geometry sampling rate from shading sampling rate by aggregating all geometric contributions into an averaged set of surface descriptors (mean albedo, mean specular, mean roughness). The averaging is done through distance functions.

Impostor [**Ris06**] [**And07**] [**Har10**] based algorithms can be rendered instead of real far away objects. This can drastically lower the geometric complexity of the objects and thus reduce geometric aliasing.

2.5.3. Direct Illumination

While culling algorithms lower the geometry processing complexity in rasterization, the shading complexity is usually the main performance bottleneck. A low, or even constant shading complexity can be achieved by **deferred rendering** [**Dee88**] [**Sai90**] algorithms, which are a special group of rasterization based-rendering algorithms. They compute only the relevant interactions between lights and objects, guaranteeing constant shading complexity. Without deferred algorithms, rasterization can't make thr difference between potential and relevant light-object interactions, and it is forced to evaluate almost all possible combinations, greatly wasting computational resources in a $O(l \cdot o)$ complexity problem, where l is the number of lights and o is the number of objects. Deferred rendering uses explicit or implicit acceleration structures to compute only the relevant light-object interactions, lowering the computational complexity to $O(o + l)$. It does so by using screen-wide acceleration structures, which consume a large amount of GPU memory, named **geometry buffers**, or G-buffers.

Deferred rendering algorithms can be either **single-geometry** pass or **multi-geometry** pass [**Lee09**]. In single geometry pass algorithms, deferred rendering algorithms process the objects of the scene once, while in multi-geometry pass variants they process the objects many times. There are also hybrid variants [**van13**] which try to balance geometry processing costs and fragment processing costs.

Deep deferred shading (DDS) [**Mar14**], multisampled deferred [**Thi09**] and adaptive super-sampling deferred rendering [**Hol13**] come in single-pass and multi-pass variants and try to mitigate the large memory consumption problem anti-aliasing deferred rendering algorithms.

Interleaved sampling [**Kel011**] can be applied to deferred rendering [**Seg06**] in order to minimize the number of shading operations. The initial geometry buffer is subdivided into a number of smaller buffers. Each of the small buffers contains interleaved information from the original G-buffer. Then, the lights are distributed over the sub-buffers and shading is performed

normally. Finally, a discontinuity sensitive buffer is used to correctly filter all the small lighting buffers into a final full-resolution lighting buffer, which is then applied to the color buffer to create the final image. The same technique was adapted for transparent objects in inferred rendering.

Light indexed deferred rendering [**Tre09**] and list based light indexed deferred rendering [**Lau12**] solve the problem of low contribution light-object intersection by storing all light indices in a list and testing each interaction before evaluation. Tile based deferred rendering [**Ols11**], cluster based deferred rendering [**Ols12**] and Forward+ deferred rendering [**Har12**] take the concept of storing light index lists and improve upon it with **hierarchic** data structures like tiles, clusters, or pseudo-clusters defined through depth masks.

Stream compaction for deferred rendering [**Hob09**] is an algorithm used to minimize the GPU code path divergence in setups with many materials. Deferred++ [**Bur13**] is a tile-based deferred rendering variant, where primitive indices and light indices are stored in tile-based lists. The shading process loads the objects and lights for each tile, minimizing bandwidth and memory requirements. In general, deferred rendering suffers from large memory costs and an inability to represent global illumination effects and correct transparency. An abstraction of deferred rendering is presented in Figure 10.

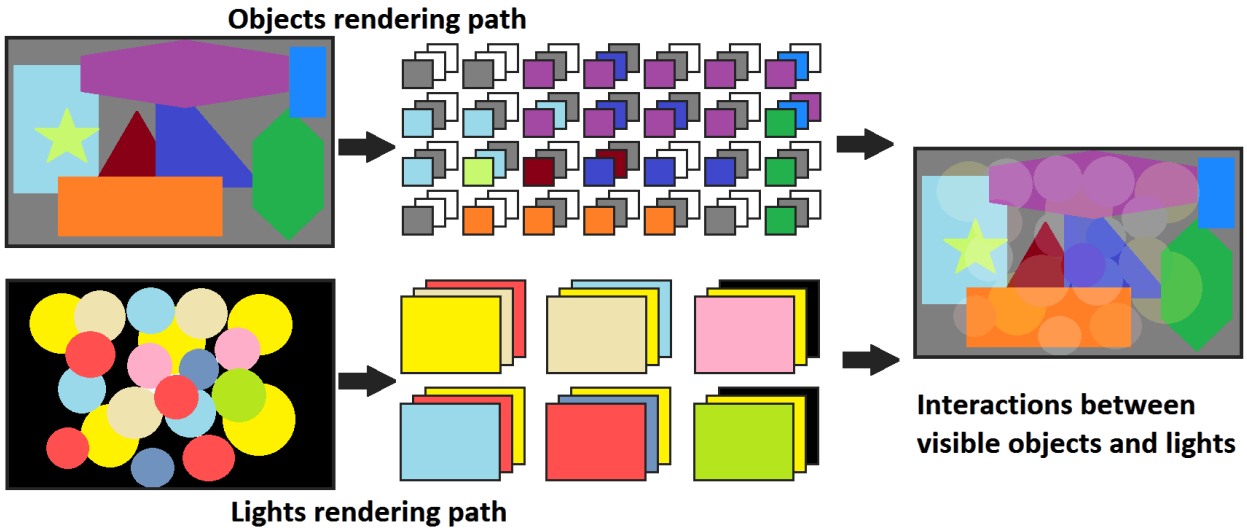

Figure 10 Abstract deferred rendering**.** Deferred rendering algorithms decouple visibility determination and shading. They first determine the visible object(s) for each pixel. Afterwards they store the lights in an acceleration structure (here tiles). In the end the algorithm intersects each pixel's visible object(s) with the relevant lights, providing all relevant light-object interactions.

Decoupled rendering (DR) [**Rag11**] takes deferred concepts further by completely decoupling visibility samples and shading samples, through the use of a memoization cache. It creates a many-to-one relationship between visibility samples and shading samples, which dramatically improves efficiency in evaluating effects that require multiple shading samples per fragment such as depth of field or motion blur. For example, a moving surface from a primitive might be rasterized over a different number of pixels in a single time frame. This is very common since no frame is instantaneous and objects will move during the frame time, creating the visual effect of motion blur. The memoization cache is very expensive to implement, because it has large memory requirements and it has to synchronize a lot of GPU data. Furthermore, the algorithm requires special hardware in order to work at maximum efficiency. This technique

was adapted to deferred rendering in decoupled deferred rendering [**Lik12**], where a modified memoization cache is used as compact geometry buffer. Similar to decoupled deferred rendering, sort based deferred rendering for decoupled [**Cla13**] uses Morton-order encoded approximations of primitives instead of a modified memoization cache. The principle of decoupled sampling is described in Figure 11.

Figure 11 Decoupled sampling**.** The same texture data is needed for multiple pixels during the period of a single frame, a common occurrence in scenes with fast moving objects. In order to correctly implement motion blur and not pay excessive bandwidth costs, decoupled sampling checks visibility samples separately from shading data and employs a single shading sample per pixel model. When it is first encountered, the shading sample is mapped in a memoization cache, and then referenced from the cache, in a process which applies dynamic programming principles to bandwidth rendering problems. Because the memoization cache is stored in cache memory, the access costs and bandwidth are greatly reduced. Source: **[**Rag11**]**.

Volumetric lighting takes into account scattering effects caused by non-vacuum media. While it can be considered a form of indirect illumination, it is caused by direct illumination, and therefore it should not be considered as a form of GI. Volumetric lighting is especially useful in effects such as fog, light shafts (also known as god rays or crepuscular rays) among other effects, where it is usually combined with **procedural noise** in order to increase medium irregularity. Pure **voxel** based methods can be used to accurately implement volumetric lighting such as the algorithm presented in [**Cra09**]. On the other hand, such effects require a very large number of samples, which can quickly amount to an excessive consumption of bandwidth.

In a rasterization context, volumetric lighting can be cheaply approximated with geometric information from the lights, where a product between the viewing vector and normal vector can be used to decrease lighting near edges [**Cha15**]. Volumetric lighting can also be approximated in **screen space**, as a post process [**Mit071**]. **Impostors** have also been used in inexact representations of scattering processes, where the volumetric object is represented as an impostor cloud. Such representations lack convincing quality, because impostors do not interact with the environment.

Volumetric lighting can be correctly implemented with shadow maps, by projecting the ray onto the volumetric light shadow map and **marching** the shadow map [**Tot09**]. This accumulates volumetric effects like absorption, emission, in-scattering and out-scattering by using the visibility data from the shadow map. **Interleaved Sampling** [**Kel011**] can be used to lower the complexity of marching a ray. It is based on the idea that neighboring pixels have similarly marched camera-surface rays; therefore, pixels in that vicinity can use samples from their neighbors. This is easily implementable in compute shaders or as post processing passes because the resulting scattering convolution kernel is separable. [**Eng10**] observers that scattering varies mostly in depth discontinuities and along **epipolar** lines and lowers the

complexity of ray marched scattering by interpolating epipolar samples. These samples are taken by marching rays from the screen space projected light position to epipolar positions on the edge of the screen space frustum. Extra attention is given to depth discontinuities. **Epipolar rectification** [**Fus00**] can be used in combination with a 1D min-max map to quickly evaluate volumetric lighting effects, as presented in [**Che11**]

A different implementation of scattering effects can be done with **Polygonal Light Volumes** [**Bil10**]. This method takes the shadow maps from scattering lights and displaces a pre-defined grid over them, quickly creating accurate shadow volumes.

Subsurface Scattering (SSS) defines the light transport in special media like skin or marble, where photons enter and suffer multiple interior reflections before leaving the surface at a point different from the point of entry. Effects such as translucency or backlight lighting are caused by SSS. From a rendering standpoint, the surface-light interactions in SSS materials are different from normal materials. The interactions are described by BSSRDFs, which are generalized BRDFs with superior dimensionality. Because of the higher dimensionality SSS has long been considered a high cost effect, and has been only lately brought to a reasonable rendering complexity. BSSRDFs can be implemented with a light diffusion **convolution kernel**, which composes the surface with the light diffusion. For isotropic media, this convolution kernel is almost always a sum of Gaussian kernels.

The light diffusion that takes place in SSS was first tackled in real-time in [**Bor03**], where instead of evaluating the convolution kernel, it was approximated in **texture space**, in the form of many blurred color texture maps, which were then composited through a simple weighted sum. Another algorithm [**dEo07**] uses Translucency Shadow Maps which are then blurred and summed, approximating the sum of Gaussians in real-time instead of using texture space for preprocessing like [**Bor03**]. Preprocessing can be used to approximate the thickness of a geometric object, in a manner similar to measuring an internal ambient occlusion. This precomputed thickness term was used to provide convincing results in [**Bri11**].

A novel approach towards SSS was offered in [**Jim09**], in which a screen space convolution kernel is used to approximate the real surface kernel. A depth-discontinuity aware filter is used to efficiently approximate the surface in screen space, in order to locally filter only fragments generated by the same objects. This method was extended to be separable for a sum of Gaussians in [**Jim15**].

2.5.4. Shadows

Shadows can be implemented through proxy geometry as **shadow volumes** [**Cro77**], through rasterized images as **shadow maps** [**Wil78**] or through hybrid types of rendering algorithms like ray tracing over rasterized data. Because shadow maps are implemented through rasterization, they benefit from its data locality advantages, making shadow maps more efficient compared to other solutions. Therefore, shadow maps have seen a wide adoption in shadow rendering solutions for real-time applications, even if they are notoriously difficult to efficiently implement, since they are a sampling process heavily dependent on the configuration of objects and lights. Shadow maps have been thoroughly researched, which led to a large variety of existing techniques.

Perspective aliasing is due to the non-linear mapping between shadow map pixels and camera pixels. **Precision warping methods,** like perspective shadow maps (PSM) [**Sta02**], light space perspective shadow maps(LSPSM) [**Wim04**], trapezoidal shadow maps (TSM) [**Mar04**], and camera space shadow maps (CSSM) [**Kol12**] and logarithmic shadow maps (LSM) [**Llo06**] warp the light visualization frustum, increasing precision near the shadow caster or near the camera. They are a simple form of adaptive sampling, in that they don't treat shadow map discontinuities or hard to sample cases in an analytical manner, they just warp the shadow map resolution for a good general case.

Partitioned shadow maps, include cascaded shadow maps (CSM) [**Dim07**], parallel-split shadow maps (PSSM) [**Zha06**] and sample distribution shadow maps (SDSM) [**Lau11**], and separate the light visualization frustum into multiple frusta, distributing shadow map resolution to better match the camera view. Partitioning techniques are very similar to precision warping methods; they just use multiple maps instead of warping, and are thus less exposed to the large number of problems that plague perspective warping shadow maps. On the other hand, multiple frusta shadow maps variants require manual tuning and special filtering between partitions.

Perspective aliasing can correctly only be solved through correct **adaptive sampling.** Adaptive Shadow Maps (ASM) [**Fer01**] store shadow samples in a hierarchic grid. The hierarchic grid is then sampled and filtered during rendering. ASM are memory intensive structures and require multiple rendering passes to completely fill. High Quality Adaptive Soft Shadow Mapping (ASSM) [**Gue07**] is an adaptive sampling method that first computes a normal shadow map, which is then used to create a hierarchical shadow map. The hierarchical map stores minimum and maximum values of the normal shadow map in hierarchies represented with mipmaps. It is then used together with the contours of occluders to determine the difficult occlusion cases which take place at contours and depth discontinuities. These difficult occlusion cases are then importance filtered, which greatly increases precision.

Rectilinear texture warping (RTW) [**Ros12**] augments shadow mapping by storing additional maps which describe space warping on both shadow map axes. While the method can produce results close to ray tracing, the bias of the distortion is not addressed. Moreover the method requires highly tessellated scenes. RTW works by computing the importance of each shadow map pixel and then by choosing the maximum importance value for each row and column, which are stored in max maps. The max maps are then warped into warping maps, which effectively provides adaptive sampling. The shadow map is used by projecting all vertices onto the conventional shadow map, and then using the warping maps to warp the vertex onto new coordinates.

Shadow maps can also be used in **analytical reconstruction** methods, which offer superior results because they provide sub-pixel level accuracy. Reconstructable geometry shadow maps (RGSM) [**Dai08**] offer an alternative storage for the shadow map, by encoding the closest visible triangle instead of only its depth. This technique addresses both projective and perspective aliasing but has problems in working with dense geometry and requires large amounts of memory. Subpixel Shadow Mapping (SPSM) follows the geometry storage concept from RGSM but uses a compressed representation for the shadow map, storing the vertices of the closest triangle in each pixel of the encoded shadow map. It also stores depth derivates. Instead of performing a simple comparison using the shadow map, SPSM performs ray-triangle intersection for the shadows, and applies an analytical form of vicinity sampling called silhouette recovery. Both RGSM and SPSM are not suited for scenes with high geometric complexity,

which would store more than one triangle per shadow map pixel. This would lead to a new type of aliasing, caused by a shadow map pixel stores an aliased triangle.

Volumetric shadows are very hard to represent because they need more than one sample per pixel. Deep shadow maps [**Lok00**], multiple depth shadow maps [**Pag04**], deep opacity maps (DOM) [**Yuk08**], adaptive volumetric shadow maps [**Sal10**], opacity shadow maps (OSM) [**Kim01**] and Fourier opacity maps (FOM) [**Jan10**] keep more than one sample per pixel in the shadow map and are therefore able to represent volumetric effects. Adaptive Volumetric Shadow Maps (AVSM) [**Sal101**] applies the idea of adaptive storing of signals [**Sal11**] to shadow maps.

Epipolar rectification [**Fus00**] can be used in combination with a 1D min-max map to quickly evaluate volumetric shadows, as presented in [**Che11**]. The algorithm creates depth maps from light and camera views and then uses epipolar rectification to rectify the images. It then uses a 1D min-max to create prefix sums for height field intersection, which are then used in a render pass to quickly determine the amount of scattering. The algorithm is explained in Figure 12.

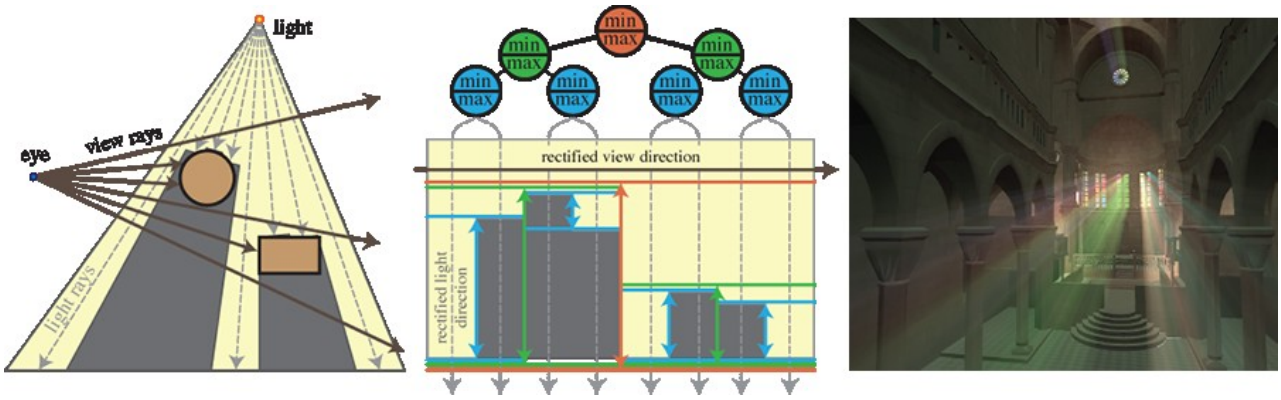

Figure 12 Volumetric shadow maps**.** Volumetric shadows can be implemented by taking depth images from light and camera views, which are then rectified through epipolar rectification. A binary tree of the shadow map depths is stored as a min-max tree, in order to approximate the occlusion for each rectified camera view direction ray. The min-max tree is traversed during rendering to evaluate the light scattering. Image source: **[**Che11**]**.

Shadow map **filtering** and sampling has also been thoroughly researched. Percentage-closer soft shadows (PCSS) [**Ran05**], fast percentage closer shadow maps using temporal coherence [**Sch13**] and **screen space** percentage-closer soft shadows (SSPCSS) [**Bag10**] introduce efficient methods to shadow map filtering through simple kernels. Perception based shadow filtering can be used to approximate the penumbra, by altering the kernel size based on blocker depth and light size [**Ran05**]. Stochastic filtering methods such as exponential shadow maps (ESM) [**Ann08**] and variance shadow maps (VSM) [**Don06**] further decrease the number of necessary samples for correct filtering. Convolution shadow maps [**Ann07**] permit arbitrary convolution filters over shadow maps, through prefiltered mipmaps.

Ray traced shadow maps [**Sto15**] can be implemented with conservative rasterization by storing triangle indices instead of depth in a different type of deep shadow map. In the shadow map pass the indices are stored with conservative rasterization, to guarantee each potentially intersected primitive is saved. In the rendering pass, the modified deep shadow map is ray marched and ray-primitive intersection tests are performed for each primitive index read in the marching sampling points.

Shadow maps techniques are usually **combined**, because many of the presented techniques only improve shadow maps for certain aspects: cascades and warping techniques importance sample the shadow map, variance and exponential methods improve depth comparisons, percentage closer filtering and screen space filtering improve shadow map samples filtering performance. Cascaded deep ray-traced shadow maps are an example of such a combined technique.

Shadow maps suffer from performance problems because each light needs at least one shadow map, point lights usually needing more [**Osm06**] [**Ger07**]. Imperfect shadow maps [**Rit08**] introduced a real-time method for shadowing many lights, using depth image reconstruction to approximate many low-resolution shadow maps. Efficient virtual shadow maps for many lights [**Ols14**] can handle a large number of dynamic lights by using a cluster acceleration structure to determine the shadow casters and receivers for each light, and then by writing the result in an array of shadow cubemaps.

2.5.5. Transparency

Rasterization provides the interactions between camera and objects without order, therefore transparent and translucent effects, which depend on interaction order, are not easy to implement within it. Rendering of transparents in a rasterization context was first considered in [**Por84**], as a series of fuzzy object compositions, which were mixed with an over operator. There are many approximation methods, but they can't offer correct results. Alpha to coverage [**Tar10**] is a stochastic method which determines the expected color in transparent rendering by representing the alpha as a number of samples which pass or fail, similar to hardware multisampling.

Other methods [**Mes07**] [**McG13**] separate the **order independent** term from the **order dependent** term. **Depth peeling** and dual depth-peeling [**Bav08**] use clipping planes to divide the scene into multiple layers, which can then be used to correctly accumulate transparent fragments, at the extra cost of greatly increasing geometry processing.

Correct transparent rendering within rasterization is achieved through **A-Buffer variants** like [**Car84**] [**Bar11**] [**Mau12**] which employ sorting strategies and consume large amounts of memory. Other correct results can be obtained through **stochastic** methods [**End10**], which use many samples per pixel to compute the accumulated expected value, and through adaptive methods [**Sal11**] [**Sal14**], which store high-fidelity approximations of combinations of multiple fragments, but require special hardware.

Occupancy maps [**Sin09**] are a special type of transparency algorithm, because they can excellently approximate depth distributions through the use of per-pixel depth masks, like a bitwise depth peeling. Their only problem is that they can't properly handle multiple objects per pixel. **Fourier opacity maps** [**Jan10**] take the depth distribution approximation concept further, by measuring the depth distribution in Fourier space and approximating it with a small number of components. Fourier opacity maps can't handle high frequency detail.

Refraction effects are usually handled in rasterization by multi-pass techniques, which use information from previous passes to convincingly fake these effects. Correct global illumination effects like reflections, refractions and shadows can be added to rasterization with other rendering passes, or other algorithms. Screen space reflections, cubemap reflections and

impostor based techniques can be used for reflections and refractions, but, for a high quality rendering, a superior algorithm is required.

2.5.6. Motion

Objects in motion are harder to render correctly, compared to static objects. In real life, when a photograph is taken, the camera shutter exposure time is responsible for storing light reflected by the photographed objects. When these objects move fast, they reflect light on a large number of photoreceptors inside the camera and produce effects such as motion blur (MB). The camera is also a lens based light sink, therefore the focal distance for which the camera is set controls the level of precision with which light is captured. Objects that are very far from the focal point suffer from lack of precision and blurring, in an effect named depth of field (DOF).

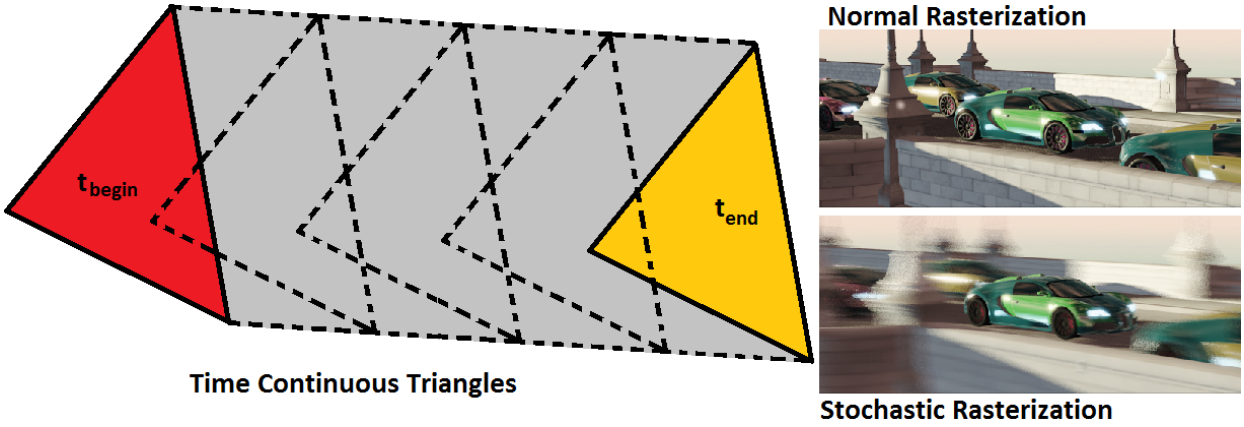

Figure 13 Stochastic rasterization**.** It enables motion blur effects by temporally sampling the primitives. Instead of rasterizing many primitives and suffering from aliasing, Time Continuous Triangles (TCT) is used, which is a type of primitive which bounds all the temporal samples of a triangle, akin to a temporal convex hull. Source: **[**Mcg10**]**.

Stochastic rasterization [**Mcg10**] [**Ake07**] is a variant of rasterization in which frames are not considered instantaneous. During the time required to render a single frame, the rasterized projections of dynamic objects can travel a very small distance in screen space coordinates. This effect is a close approximation to the real life motion blur. But in order to represent such an effect a very large number of temporal samples are required, which can easily lead to very long processing durations. Stochastic rasterization is able to represent this effect by using compact temporal geometric representations, which approximate multiple time samples for each rendered primitive. These continuous temporal primitives, called **time continuous triangles**, as presented in Figure 13. While time continuous triangles reduce computation, stochastic rasterization still has very high computational costs, and it is an interactive but not a real-time technique.

Motion can also be efficiently implemented with **post processing** techniques. In [**Gue14**] motion blur is implemented through a tile-based approach, storing the largest, dominant velocity in a screen space region. Then the motion blurring is done in the direction of the local velocity, which is obtained by interpolating the dominant velocities from the closest tiles. Tile boundary discontinuities are treated by stochastically sampling the local velocity for pixels which lie at the boundary of a tile. **Depth of Field** techniques have a lot in common with motion blur, because they are based on the principles of quickly mixing multiple samples in a vicinity. The effect of the aperture shape, known in photography as “Bokeh” varies from geometric at small aperture to

circular at high aperture. Bokeh can be rendered through point-splatting techniques or impostors [**Sou13**] or more realistically through separable gather filters [**McI12**]

2.6. Approximated and screen space methods

Correct global illumination is extremely expensive to compute, thus many methods have been developed to inaccurately approximate it. One of the simplest and less accurate methods to approximate global illumination is to **precompute incoming radiance** for every object in the scene. This can be inexactly represented through an environment map, where each entry encodes the radiance for a direction relative to the object. Such a map is usually called a light probe or an environment probe, and was researched first as a texture based technique [**Bli76**] and then as **cubemap** [**Bjo04**]. The major disadvantage of such maps is that they fail to represent correct radiance for high frequency concave geometry. They are also incompatible with dynamic geometry. **Parallax-projected** cubemaps, or box-projected cubemaps, warp and combine the precomputed radiance stored in cubemaps to better sample transitioning areas between precomputed environments [**Lag12**].

Even with this approximation, correct evaluation of the incoming radiance for the primitives of the objects requires a costly integration process, which requires heavy sampling in the environment map. **Spherical harmonics** (SH) **lighting** [**Ram01**] [**Now12**] [**Gre03**] separates the illumination equation into a sum represented in the spherical harmonics bases, akin to Fourier analysis. The first 4 bases are plotted in Figure 14. Spherical harmonics lighting solves the problem of sampling large environments at the cost of losing high frequency detail and it requires a large pre-computation overhead.

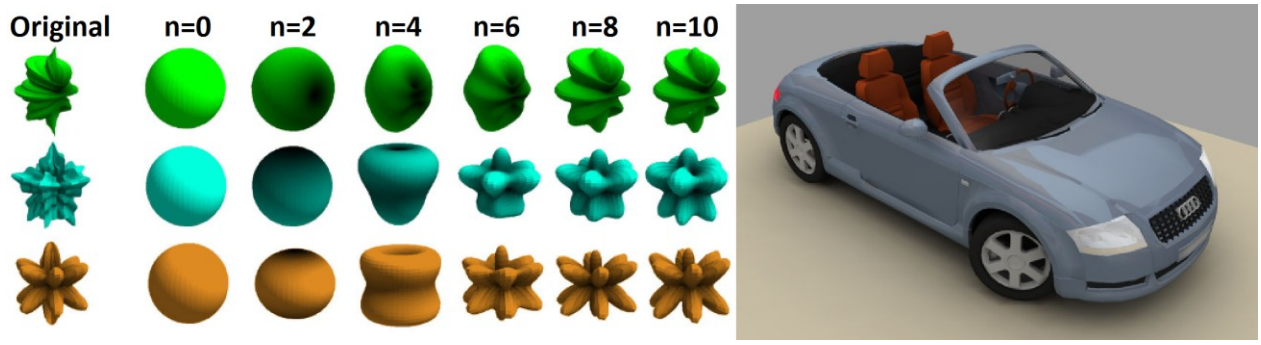

Figure 14 Spherical Harmonics**.** The image shows the how general functions can be represented in a compressed format through spherical harmonics. Light fields are functions that represent the irradiance of objects surroundings. Light fields can be encoded with spherical harmonics, and then they can be quickly reconstructed to perform environment illumination, like in the right side of the image. Source: **[**Gre03**]**.

While this method only samples $l(d|g)^?e$ light paths, used with rasterization it can sample $l(d|g)(d|s|g)^?e$ paths, and therefore can augment rasterization with one bounce diffuse or glossy global illumination. This method has been used in many real-time applications but usually with static scenes. The cost of pre-computing is usually amortized by doing this work across multiple frames. On the other hand this approach introduces a noticeable delay. Furthermore, this method does not include multiple light bounces, making its result far from the ground truth in many less than ideal cases. Spherical harmonics are also used as a form of compression for many light fields in other rendering algorithms.

Light Propagation Volumes (LPV) [**Kap09**] use a tridimensional grid to transport light in the scene. The scene is voxelized as a pre-process in a normal, sparse or cascaded grid, each element of the grid representing the estimated opacity of the scene in that vicinity. Light is injected into the grid and then propagated in iterative steps. Because it is a grid propagation process, LPV can be used for volumetric lighting. Since it uses opacity and not real geometry, there is a big difference between the reference rendering and the one obtained with LPV. Furthermore, since opacity does not represent the high frequency geometry data, LPV is only suitable for diffuse global illumination, and suffers from light bleeding artifacts. LPV and rasterization can simulate some of the $ld^*(d|s|g)^?e$ paths in the rendered scene.

Precompute Radiance Transfer and Local Radiance Transfer have been used together in a method named **Deferred Radiance Transfer Volumes** [**Gil12**]**.** This method starts by placing probes in the scene and storing the probe configuration in a tridimensional grid acceleration structure stored in a volume texture. At runtime the algorithm relights the probes, by amortizing the computational cost over time and storing the new results into the volume texture. The effective shading just loads the adequate lighting data to evaluate the illumination, and it is completely decoupled from the computational process.

2.6.1. Screen Space Ambient Occlusion

Screen space ambient occlusion algorithms attempt to approximate light occlusion caused by neighboring objects, creating a darkening effect by approximating ambient occlusion [**Zhu98**].

Ambient occlusion (AO) approximates the occlusion caused by ld^* paths, and together with rasterization and many light methods it can represent $ld^*(d|s)e$ paths. AO is defined as a visibility function which measures the direct ambient light that can reach a position, coarsely approximating global light transfer:

$$AO(x,n) = \frac{1}{\pi}\int_{\Omega} \varrho\big(d(x,\omega)\big)\max(0, \mathrm{n}\cdot\omega)$$

where d is the distance to the nearest occluder from point x in the ω direction, and ϱ is the fall-off function. While the ambient occlusion term is a correct approximation for occlusion in a direct lighting setup, it is based on a **coarse approximation** of global light transfer and global occlusion, and is therefore not physically correct.

Although screen space information is not sufficient for correct occlusion determination, screen space ambient occlusion algorithms have proven to be practical and very fast image enhancers, even if they are just an approximation of AO, which is a coarse approximation itself. They can be **considered as perception enhancing** algorithms.

Screen Space Ambient Occlusion (SSAO) [**Mit07**] is the first algorithm to approximate ambient occlusion in screen space. It uses the depth buffer of the scene as an approximation of the existing geometry, working with it as if it were a heightfield. Each pixel's screen space vicinity is sampled for occlusion. If the sample is found under the heightfield it is considered occluded, otherwise it is visible. The results are then averaged and a part of the $\omega \cdot n$ in the rendering equation is thus cheaply approximated. While the technique was far from exact, it paved the way for other screen space occlusion techniques.

Horizon based Ambient Occlusion (HBAO) [**Bav081**] traces rays on the screen space heightfield to find the angle of free horizon, an idea similar to relaxed cone step mapping (RCSM) [**Dru06**]. These angles are then averaged to produce the occlusion factor, albeit at a steep sampling cost.

Volumetric obscurance (VOSS) [**Loo10**] is a variant of screen space occlusion algorithm where the occlusion factor is computed through measuring the visible volume in the pixel vicinity.

The **Alchemy Screen Space Obscurance Algorithm** (ASSOA) [**McG11**] creates samples on a disk and then projects them on the pixel vicinity. The $\omega \cdot n$ term is evaluated for all the projected occlusion samples and the final obscurance is obtained through averaging. **Scalable Ambient Obscurance** (SAO) [**Mcg12**] improves on ASSOA by sampling in a hierarchical depth buffer, greatly improving cache efficiency.

Multi-view Ambient Occlusion with Importance Sampling (MVAOIS) [**Var13**] uses a weighting scheme to sample the screen space and available shadow maps in order to solve AO problems where SS geometry information is not sufficient.

Line-Sweep Ambient Obscurance (LSAO) [**Tim13**] uses line scans with a stack mechanism to determine the greatest occluder, and it is further refined in Far-Field Ambient Occlusion (FFAO) [**Tim131**], where a high quality approximation of obscurance is computed through prefix sums of line scan results, which significantly amortize the complexity of the sampling process. Sampling strategies for screen space occlusion algorithms can be observed in Figure 15.

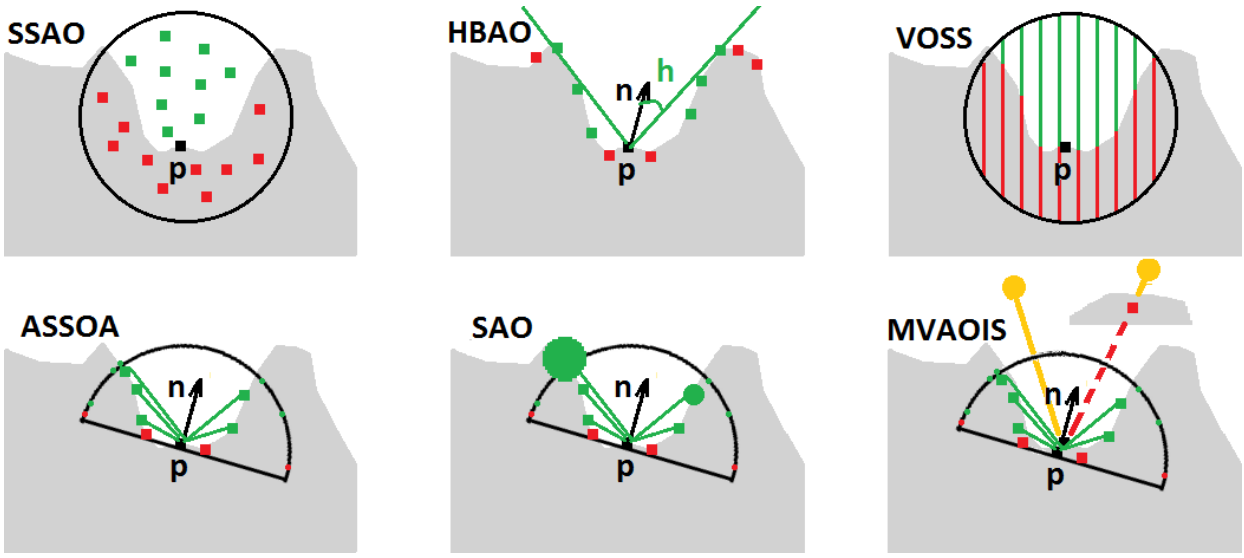

Figure 15 SSAO sampling**.** The image shows space sampling strategies used by state of the art screen space ambient occlusion algorithms: random (SSAO), horizon angle based (HBAO), volumetric (VOSS), $\boldsymbol{\omega \cdot n}$ based (ASSOA), scalable sample based (SOA), or shadow map augmented (MVAOIS).

2.6.2. Image Based Lighting

Image Based Lighting (IBL) is a family of techniques that use image space algorithms to approximate tridimensional illumination effects. IBL trades correct evaluation of light-object interactions for the coherent evaluation of the approximations of these interactions. Because IBL methods are cheap and coherent they are well suited for real-time applications.

Screen space global illumination (SSGI) methods are a special type of GI algorithms because they use only screen space information. Because they work with incomplete information they are inherently flawed, but they can be effective in representing some effects like front specular reflections and occlusion. Furthermore, their execution speed is very fast compared to other GI solutions, because they run on data with excellent locality, in image space. At their core they are image processing algorithms, but they run on images with enhanced information, like depth, material type and so on. They can also benefit from temporal reprojection and coherence methods [**Sch10**].

Screen space directional occlusion (SSDO) [**Rit09**] approximates direct lighting in a small vicinity of a pixel by correctly checking occlusion for each light and not using a visibility approximation for all sources like AO. SSDO also computes local indirect radiance transfers by doing a single indirect bounce between the tested samples. The algorithm can also be used for screen space shadows, albeit it can only work for special, local, geometry cases.

Bent normals [**Lan02**] can be used in highly occluded vicinities to better model the interaction between light and the occluded surface. They are basically a form of importance sampling the illumination at the geometry surface level, by bending the normal to a direction that maximizes received light. Screen space **bent cones** (SSBC) [**Rit11**] uses bent cones, an improved concept of bent normals, which also take the variance of the unoccluded direction into account. Lighting is then evaluated by sampling the bent cont for visibility for each light.

Screen space can also be used as a ray tracing and marching space, albeit without complete tridimensional information screen space tracing is reduced to a very small subset of the scene paths: $l(s)^*e$ for the normal method, $l(d|g)^*e$ for the distributed variants and , $l(d|g|s)^*e$ for hybrids. Together with rasterization, screen space tracing can sample $ls^*(s|d|g)^?e$ paths, or even $l(s|d|g)^*e$ paths, if the tracing is hybrid. Compared to ambient occlusion, ray tracing does not approximate direct light visibility but actually measures it, so screen space ray tracing approximates a correct light transport. Furthermore, screen space tracing can have access to shadow results from rasterization and to other data stored in G-buffers.

Image space gathering [**Rob09**] uses a parameter search and gather process to determine samples which could be ray traced in screen space, to determine reflection and to smooth shadows.

In [**Sol10**] **screen space ray tracing** (SSRT) is augmented with mipmapped buffers of the screen space to accelerate ray diffuse light sampling. Because the screen space sampling is done through mipmaps, a very large number of samples can be filtered instantaneously; therefore the 2D ray intersection costs are dramatically lowered. Furthermore, the same filtering can be used to approximate large screen space areas and coarsely approximate diffuse reflections, making it able to sample a small number of $l(s|d|g)^*e$ scene paths together with rasterization.

This technique is further refined in **Screen Space Cone Tracing** (SSCT) [**Her14**] [**Ulu14**] by cone tracing in screen space over a mipmaps hierarchy of the depth buffer. Basically, SSCT is a 2D adaptation of [**Cra09**].

Screen space local reflections (SSLR) [**Mcg14**] analyzes screen space ray tracing from ray accuracy standpoint, pointing out that screen space does not contain sufficient information to correctly define rays for all possible tracing cases. It increases the accuracy of rays with a digital differential analyzer line rasterization algorithm modified for perspective-correct interpolation.

Screen space Photon Mapping (SSPM) [**McG09**] coarsely approximates the shooting and gathering stages from photon mapping by doing them in screen space as photon volumes, using the same concept of screen space volumes used in SSCT.

The obvious problem with screen space global illumination methods is that they work on approximated data, reconstructing tridimensional environments from bidimensional representations. Some methods use multiple image spaces to better approximate the scene geometry. **Deep Geometry Buffers** [**Mar14**] can be used to store layers of parallel screen space representations, similar to the depth peeling [**Bav08**].

More correct hybrid approaches combine other types of rendering with image based lighting, like [**Gan14**]. In [**Gan14**] many G-buffers are used which represent image space representations for all objects and lights not present on the screen. The screen is ray traced over a BVH hierarchy and the rays interact with the IBL representations just as they would do with normal geometry. Such deep/many IBL methods provide more geometric information and make tridimensional reconstructions better, albeit at a significant cost in used memory. The tridimensional reconstructions greatly increase the number of light paths that can be simulated.

Some of the most relevant state of the art Screen Space Global Illumination methods are presented in Figure 16.

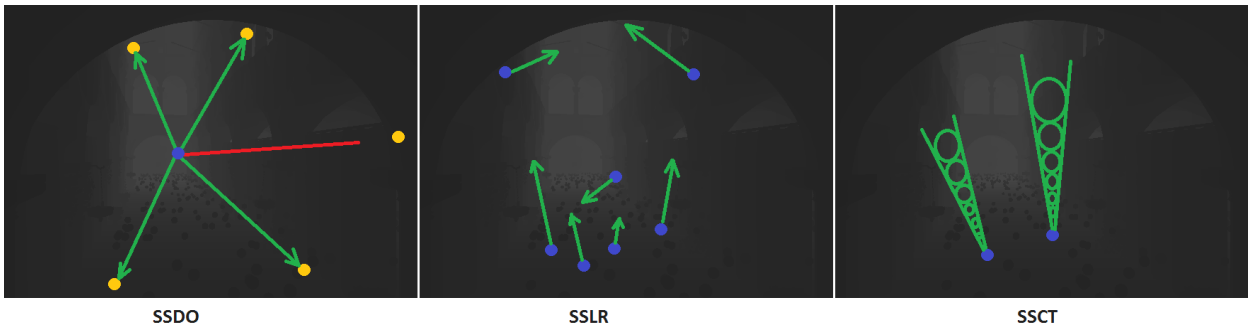

Figure 16 SSGI**.** Screen Space Directional Occlusion (SSDO) searches each direction to a light in screen space for occluders. The image presents an unbounded version of SSDO, the algorithm usually works on a screen space vicinity. Screen Space Local Reflection (SSLR) continues the ray defined by the sampling point and the camera into its projected direction in screen space. Screen Space Cone Tracing (SSCT) differs from SSLR by tracing cones over a hierarchical depth map. Because of this it can be used to represent glossy specular light transport.

2.7. Reyes

The Reyes [**Coo87**] rendering algorithm was created as an alternative for rasterization that could easily handle highly detailed models, such as those used in offline rendering. Such models would create **extremely small primitives** which would then produce extreme aliasing in rasterization, because of the loss of data caused by the z-tests. Moreover, any multisampling technique would need a very large number of samples to correctly filter the many resulting fragments. Furthermore, Reyes is also aimed at better integrating parametric surfaces and displacement mapping in the rendering process. Because of these traits, the Reyes algorithm has been very popular in the movie industry, where very detailed meshes are the norm. An industry standard that implements Reyes is Pixar's Renderman [**PIX15**].

The Reyes algorithm can be considered a special case of rasterization, where a complex importance sampling mechanism is used to sample the difficult case of very small polygons, called in Reyes micropolygons. Therefore, Reyes shares many common traits with rasterization, such as being a bottom-up approach, not being physically correct, not implementing global illumination and representing only $l(d|s|g)^{?}e$ without other rendering methods.

The Reyes algorithm has five stages: bounding, splitting, dicing, shading and sampling. In the **bounding** stage the original primitive is bounded by a convex hull which is then projected in screen space. This is done to approximate the size of the primitive. In the **splitting** stage the bounded primitive is divided into new, smaller primitives. The bounding and splitting phases are performed iteratively, until the newly divided primitives are smaller than a certain threshold.

When the newly generated primitives pass the splitting threshold, they enter the **dice** stage, where they are subdivided into pixel-sized polygons named micropolygons. This operation is named dicing. After the micropolygons are created they enter the **shading** stage, where only their vertices are illuminated and shaded.

The final stage of the Reyes algorithm is the **sampling** stage. In it each pixel accumulates samples from the suitable lit and shaded micropolygon vertices. The final color of the pixel is computed through a weighted average. Because of this sampling process, images rendered with Reyes suffer from very low geometric aliasing. The entire pipeline is described in Figure 17.

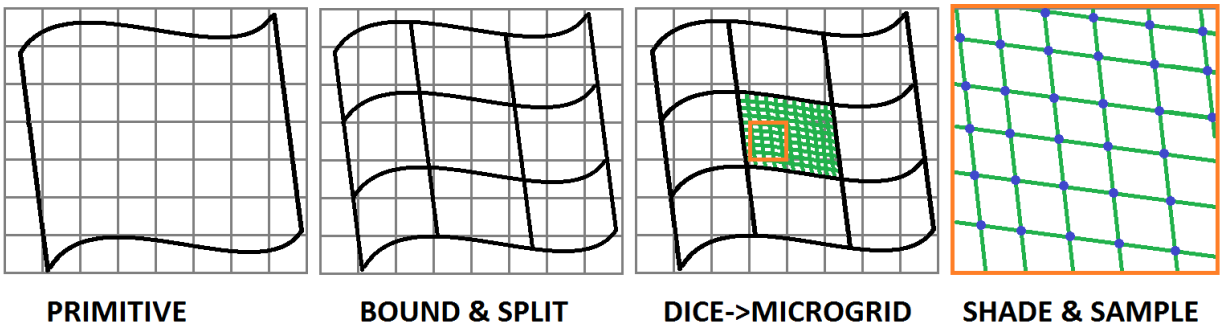

Figure 17 Reyes**.** The algorithm takes each primitive and iteratively bounds it in screen space and then splits it until a certain size in pixels is reached for each splitted primitive. Then, each splitted primitive is diced into a microgrid, which generates many polygons smaller than a pixel and which can be displacement mapped. In the final stages, the algorithm shades each vertex from the micropolygons, and obtains the final colors by averaging the samples found in each pixel.

The Reyes algorithm has also been adapted to modern hardware as either a rasterization process, augmented with the tessellation capabilities of Shader Model 5 [**Tat10**] or as a pure GPGPU technique [**Pat08**].

2.8. Ray tracing

Ray tracing takes a different space sampling approach than projection based algorithms like rasterization and Reyes. Instead of projecting primitives, ray tracing intersects them with rays. All algorithm variants and refinements trace the rays through the scene with the help of acceleration structures such as BVH or Kd-tree. While ray tracing is much better at finding relevant light-object interactions between objects and lights than rasterization, unsophisticated ray tracing is not based on physically correct principles, because it considers all reflections as specular. Ray tracing produces sufficiently convincing results for **photorealistic** rendering of simple objects, but it isn't suitable for the representation of caustics, subsurface scattering, and fluorescence and other advanced rendering effects or light paths. Ray tracing is a **top-down** rendering solution. Ray tracing needs a large number of samples for smooth rendering and has significant difficulties in sampling point lights.

The first ray tracing method was proposed in [**App68**], where the camera spawned rays that intersected the scene without suffering from reflection effects. This algorithm is nowadays called **ray casting** (RC), or **volume ray casting** (VRC) if the rays can pass through transparent primitives. It can represent $l(d|s|g)^?e$ paths, and consequently has the same results as multisample rasterization, the only difference being that ray casting sorts the camera-objects interactions while rasterization uses the Z-buffer.

The early variants of **recursive ray tracing** continued the ray casting process, by reflecting the rays against the surface of each intersected primitive. The original rays are named primary rays, while the rest of the generated rays are named secondary rays. The reflections in recursive ray tracing are specular, the light suffers no diffusion, therefore the paths sampled are ls^*e. **Forward ray tracing** (FRT), also known **as light ray tracing** (LRT), traces rays from the light and onto the scene. The camera acts as a ray collector, sampling the rays that are reflected from the scene towards it. Because of this, a large number of rays from the scene lights are necessary, therefore forward ray tracing is a very inefficient algorithm. Because it achieves results through brute force, bare FRT is extremely inefficient.

Backwards ray tracing (BRT), also known as **camera ray tracing** (CRT), traces rays from the camera to the scene. Its core idea is to simulate the reversed trajectories of photons in the scene, but it does so by tracing rays reflected in specular fashion, therefore it can only trace ls^*e paths. It can also be considered an early form of importance sampling, as it eliminates the simulation of rays which will not interact with the camera, and thus have no visual contribution. While it is a much more efficient variant of ray tracing, it has problems sampling effects caused by light concentration, such as caustics.

Both LRT and BRT consider lights as points and all reflections as perfectly specular, therefore they are not physically correct and can't handle fuzzy phenomena. Their use as rendering algorithms is severely deficient for photorealistic rendering, but they are useful as scene sampling strategies, which are often used to importance sample interactions between lights and objects in advanced rendering algorithms. **Shooting** and **gathering** are recognized terms in

computer graphics nomenclature for sampling the scene from the lights and sampling the scene from the camera. LRT and BRT are many times used together in advanced ray tracing, path tracing, photon mapping and many lights algorithms.

Whitted ray tracing (WRT) [**Whi79**] is considered the classic form of the recursive ray tracing algorithm. Instead of just reflecting the ray at any intersection, Whitted ray tracing generates three new rays: the reflected ray, the refracted ray and one or more shadow rays, effectively forming a tree of secondary rays per primary ray. The reflected ray behaves identically to the reflected rays in forward or backwards ray tracing. The refracted ray is generated when the intersected surface can transmit light, and the shadow rays are traced towards each light in the scene. Because of the extra rays generated at ray-surface interaction, Whitted ray tracing needs much fewer overall rays to sample the scene, as compared to simple LRT and BRT, because the shadow rays are much more efficient in sampling direct light contribution. While Whitted ray tracing samples the light-objects interactions much better than simple recursive ray tracing, sampling $l(s)^*(d|s|g)^?e$ paths. It still can't support global diffuse reflections, soft shadowing or other more advanced rendering effects.

Heckbert ray tracing (HRT) [**Hec90**] is a type of hybrid ray tracing. It separates surface interaction into diffuse and specular types. The specular component is evaluated with backwards ray tracing, while the diffuse component is stored in adaptive radiosity textures, which can be sampled by all BRT rays, therefore it can sample $l(s|d)^*(d|s|g)^?e$ paths. The adaptive radiosity textures are computed with a forward ray tracing based method, before BRT is used to synthesize the final image.

[**Vea95**] further analyzes ray tracing as a light-object interaction sampling process, and proposes an importance sampling technique which connects the ray trees created by backwards ray tracing with the trees created by forward ray tracing. The resulted algorithm, **bidirectional ray tracing** (BDRT), can sample light concentration effects like forward ray tracing and needs a reduced number of camera rays, similar to backwards ray tracing. This importance sampling approach is very similar to the approach used in bidirectional path tracing [**Laf96**].

Distributed ray tracing (DRT) [**Coo84**], also named distribution ray tracing, uses multiple rays over multiple sampling spaces (lens, spatial, temporal, BRDF) to augment ray tracing. DRT can handle fuzzy phenomena such as soft shadows, motion blur, depth of field, antialiasing or diffuse reflections. It does so by averaging multiple reflection, refraction and shadow rays, which can sample spatial interactions, material interaction and motion during the rendered frame much better than single rays. It can thus sample $l(s|d|g)^*e$ paths. Because it is essentially an application of Monte Carlo principles to ray tracing, the algorithm is also called stochastic ray tracing.

Similar to DRT, **Monte Carlo Ray Tracing** (MCRT) [**Dut93**] is a ray tracing algorithm that uses stochastic principles. MCRT is a type of light ray tracing which uses multiple rays per ray-surface intersection to accumulate radiance in the camera pixels. While distributed ray tracing traces ray from the camera to the scene, MCRT traces rays from the light to scene, and, on each ray-surface intersection it chooses a random camera pixel. The algorithm traces a ray between the intersection point and the chosen pixel, and if the ray is unoccluded, it computes the radiance leaving the intersection point on the constructed ray and it accumulates this radiance in the chosen pixel. Compared to light ray tracing, MCRT importance samples the potential ray paths by stochastically following only the ray paths that will have a visual contribution. This

algorithm stems from the same sampling concepts that are represented as light paths in path tracing [**Kaj86**].

In order for ray tracing algorithms to correctly render textured objects, the textured surface is usually super sampled in the image plane and then filtered. **Differential ray tracing** [**Ige99**], uses the length of the traced ray to approximate the derivative of the ray with respect to the intersected surface. In doing so, differential ray tracing super samples the texture locally, in image space. Therefore the sampling process can be precomputed with the help of texture mipmaps, drastically lowering the number of samples taken during rendering.

Ray tracing can be accelerated through various approaches: faster intersection tests with performant acceleration structures, tighter bounds given by preprocessing, fewer rays by early terminating rays and generating them adaptively and by using generalized rays such as cone rays, beam rays, sphere rays or pencil rays.

Ray tracing can be accelerated through **adaptive progressive refinement** methods [**Pai89**], in which the rendered image is created progressively. This has the advantage of producing rough visual results faster, which can be valuable when rendering durations are very long. The final synthesized image is still indistinguishable from a normally ray traced image, because the samples used in the adaptive refinement ray tracing process are chosen to produce the same expected result.

While the primary rays in ray tracing sample space in a coherent manner, the majority of the traced rays are of the secondary type. Secondary rays sample space in a camera and scene dependent manner, therefore they exhibit a high degree of data **incoherency** which can't be improved without extensive preprocessing.

Packet ray tracing (PRT) [**Bou07**] [**Ove08**] is based on the idea of combining groups of similar rays into a packet, also called a **bundle**. The packet is then intersected with the acceleration tree, with the benefits of loading an acceleration structure node per packet and not per ray. When the divergence of rays inside a packet reaches a certain threshold the packet is deconstructed and the original rays are reordered into smaller packets. Large packets are especially efficient for nodes high in the tree hierarchy, because packets intersecting these nodes have very low divergence.

A different approach to coherent tracing of secondary rays is to **globally reorder the rays** [**Pha97**] [**Nav07**] [**Ail10**]. The acceleration structure is partitioned into **treelets**, which consist of a small number of acceleration structure nodes bundled together. Each time rays from a packet intersects a treelet, the rays are added to that queue of that treelet. When the ray queue for a treelet has grown to a sufficient threshold, all the rays in it are intersected with the treelet. Thus, at the cost of one incoherent intersection test per packet, N coherent test per packet are done, where N is the number of nodes in the treelet.

Another strategy for packet ray tracing is to use **hybrid or wide acceleration structures** such as MBVH. The MBVH needs to load more data per ray-node intersection test because each MBVH node contains the data of multiple BVH nodes. The advance of this method is that the MBVH traversal has more coherent access patterns. The best results are usually obtained through hybrid strategies, where the acceleration structure is wide at the bottom and the top nodes are intersected with large packets.

The traversal can be further refined with **ray reordering** [**Bou08**], which analyzes the results of the intersection tests at packet level. If the results show a large degree of divergence the original packet is destroyed and the rays are recovered and stored in a cache. When the cache grows sufficiently large, the recovered rays are packaged into newly packets, with the idea that the worst coherence for the new packets will be superior to that of the old dismantled ones.

The fundamental idea behind packet ray tracing is to sample a larger amount of space in a coherent manner. The same idea is differently exploited by **Beam tracing** [**Hec84**] and **Cone tracing** [**Ama84**], which are ray tracing variants in which space sampling is done through beams (pyramid frusta) or cones. Beams and cones are rays with volume and are much more efficient in sampling easy to intersect forms. Cone tracing has been successfully adapted [**Cra09**] to modern hardware and is a viable solution for real-time applications, albeit one in which the memory requirements are extremely steep, and which suffers from serious limitations when rendering dynamic, interactive or skeletally animated objects. Furthermore, the adaptation is based on a voxel representation and it thus suffers from data representation inefficiency, therefore the large memory requirements are necessary. Beams can be processed through paraxial approximation theory to form **pencils**, which can be used for tracing, as described in [**Shi87**].

Sphere tracing [**Har96**] uses ray-marched spheres on the ray to determine intersecting objects in the vicinity of the ray. From a spacing sampling strategy it transforms the space volume tested for intersections by the cone tracing algorithm into a set of smaller, spherical volumes, which test approximately the same overall space. The advantage of sphere tracing is that testing sphere intersection is much cheaper than cone intersection, but the large number of spheres involved makes it less efficient than modern cone tracing.

Divide and Conquer Ray Tracing [**Mor11**] [**Kel11**], also named **Incoherent Ray Tracing**, is a variant of the ray tracing algorithm that uses no preprocessed acceleration structure. Instead, this algorithm partitions the scene primitives during traversal, making it ideal for dynamic scenes. The traversal function first subdivides the primitives into sets, usually with a fast spatial function. It then traverses the scene by testing the intersecting rays against the bounding volume of each set of triangles. The rays that intersect the bounding volume along with the enclosed primitives are then processed in a new instance of the function, usually in a depth-first-search way to increase data access coherency.

The efficient CPU vectorization of this algorithm is explored in [**Afr12**]. The algorithm is further improved in [**Nab13**] with the help of **sampling rays**, which are traced before space partitioning. The sampling rays are intersected against all the available triangles, and the results are used to statistically determine the best space partitioning distribution that can be used to split the triangles into multiple sets. This technique is basically a lower complexity SAH-like implementation for the memory less, implicit acceleration structure created during DACRT's traversal. While this technique does not dynamically allocate the scene acceleration structure, it still uses a large amount of stack storage, therefore, its GPU implementation might be memory limited, compared to the CPU one.

Space sampling strategies for different ray tracing algorithms are presented in Figure 18.

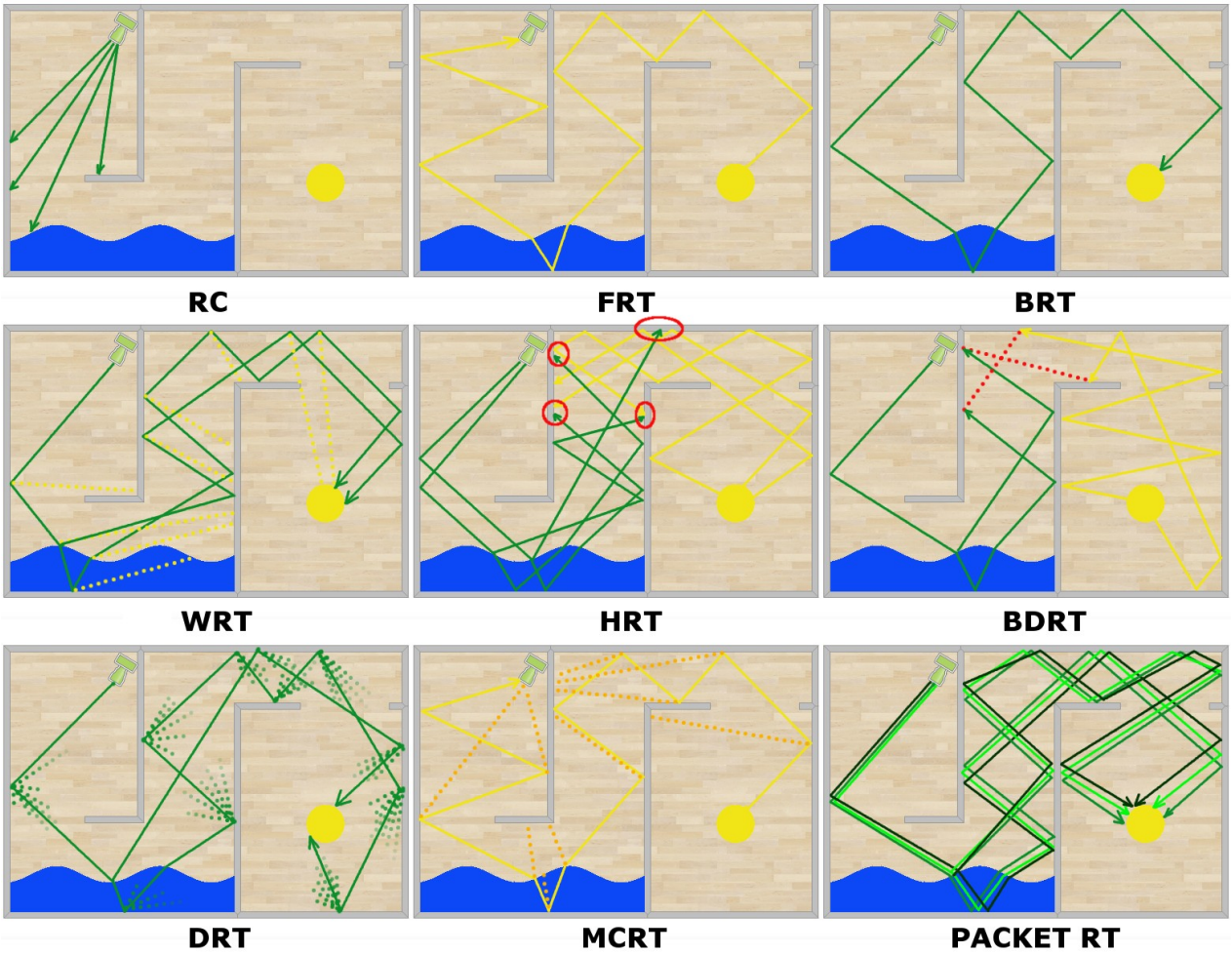

Figure 18 Ray tracing**.** Variants of ray tracing use different space sampling strategies. Ray casting (RC) casts rays from the camera to the scene objects without recursion, thus stopping at the first intersection. Forward Ray Tracing (FRT) shoots rays from the camera towards the scene, the rays are traced through multiple ray-surface interactions and the camera acts as a radiance collector. Backwards Ray Tracing (BRT) casts rays from the camera and traces the rays until they interact with lights. Whitted Ray Tracing (WRT) is a type of BRT that importance samples the scene lights at each ray-object interaction while also adding reflection and refraction rays. Heckbert Ray Tracing (HRT) uses FRT on the scene together with a radiance storing structure, which is then sampled by a BRT pass. Bidirectional Ray Tracing (BDRT) connects rays shot by the lights with rays casted by the camera. Distributed ray tracing (DRT) spawns many rays at each object-ray interaction. Monte Carlo Ray Tracing (MCRT) is a modified Forward Ray tracing algorithm that shoots rays from each light-object interaction towards the camera. Packet Ray Tracing (PRT) creates packets from many similarly directed rays and traces them together, obtaining better data access coherency. PRT can be improved with different strategies like global reordering, hybrid acceleration structures or local ray reordering.

While ray tracing is a valid solution for photorealistic rendering, it needs a large number of samples per pixel to obtain an acceptable image quality, making full resolution ray tracing solution prohibitive for real-time massive applications. Ray tracing difficultly samples highly specular paths like $LSSDE$, and is therefore unsuited for rendering caustics. Ray tracing can be anti-aliased if implemented over a deferred renderer [**Chi12**], by using the same mechanics as sub-pixel reconstruction anti aliasing [**Cha11**], but filtering rays instead fragments.

2.9. Path tracing

2.9.1. Essentials

Path tracing (PT) [**Kaj86**] is a rendering family which has a lot in common with ray tracing. Path tracing uses a space sampling strategy similar to ray tracing, therefore it can also produce **photorealistic** results and is a **top down** rendering method. Instead of sampling space through a tree of reflected and refracted rays, path tracing considers a **single path** through that tree, and uses this path as a sampling mechanism. In path tracing, each path is composed of many **path segments**, which are determined through visibility operations implemented with traced rays. The path segments link light-object-camera interactions. Path tracing uses **paths as a sampling space**, which can be particularly productive with the correct importance sampling mechanisms. Path tracing is a backwards algorithm, tracing paths from the camera and into the scene.

Path tracing is constructed under three principles: the principle of **global illumination**, the principle of **equivalence** and the principle of **direction**. The principle of global illumination states that all the objects in the scene will contribute at least a modicum of illumination to all the other objects. The principle of equivalence states that there is no difference between illumination coming from lights and illumination coming from other scene surfaces. The principle of direction states that illumination coming from surfaces must scatter in a direction that is some function of the incoming illumination. Thus, path tracing does not need direct illumination calculations and direct shadow rays, but solves the illumination as a large integral, **the rendering equation**, which respects the three aforementioned principles. In the original form, as presented in [**Kaj86**]:

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \int_\Omega f_r(x,\omega_i,\omega_o) L_i(x,\omega_i)(\omega_i \cdot n) d\omega_i$$

The rendering equation can be written in multidimensional form, as it used nowadays, which can handle different wavelength surface interactions, motion blur and depth of field and takes into account pixel photo sensors:

$$L_o(x,\omega_o,\lambda,t) = \int_{Pixel} W(ps) \left[L_e(x,\omega_o,\lambda,t) + \int_\Omega \int_T \int_\Lambda f_r(x,\omega_i,\omega_o,\lambda,t)\, L_i(x,\omega_i,\lambda,t)(\omega_i \cdot n) d\omega_i d\lambda dt \right] dps$$

Where $W(sz)$ is the weight of the photo sensor and $\int_{Pixel} W(ps) dps = 1$. The equation can be further expanded to account for volumetric light transfer, where it becomes volumetric path tracing.

2.9.2. Multidimensional integrals and Path Tracing

Exactly solving such an iterative multidimensional equation presents an impractically large computational effort, therefore stochastic methods are used, as in other computational heavy fields such as heat transfer or computational economics. **Monte Carlo** (MC) integration is a very popular integration method, especially for many dimensional problems, because it transforms local sampling on each dimension into global sampling, greatly lowering the computational complexity. The result of Monte Carlo integration is an **expected value**, which rapidly converges to the correctly computed value with a sufficient number of samples. The difference between the correctly computed result and the Monte Carlo approximated result is

measured through **variance.** A Monte Carlo integration result is said to be **converged** when the variance is lower than a required threshold, but the process converges at relatively slow $\frac{1}{\sqrt{N}}$ rate, where N is the number of samples**.** For a uniformly sampled k-dimensional domain D, the expected value can be computed through Monte Carlo integration with N samples, like this:

$$E[f(x)] = \int_D f(x)p(x)dx = \int_{D1}\int_{D2}\int_{D3} \dots \int_{Dk} f(x_1, x_2, \dots x_k)p(x_1, x_2, \dots x_k)dx_1dx_2 \dots dx_k =$$

$$= \frac{D_1D_2 \dots D_k}{N}\sum_{s=0}^{N} f(x_1, x_2, \dots x_k)$$

Since variance is all the stands between an acceptable and an unacceptable result, **minimizing variance** maximizes the computational effort. Since the majority of computationally heavy problems is not pure random in nature, judiciously choosing samples for Monte Carlo resumes to finding the best problem specific sample positions, a procedure named **importance sampling**. Importance sampling does not uniformly sample the problem space, but uses mechanisms such as probability distribution functions to better guide sampling, modifying the integration process to:

$$E[f(x)] = \frac{1}{N}\sum_{s=0}^{N} \frac{f(x_1, x_2, \dots x_k)}{p(x_1, x_2, \dots x_k)}$$

With sampling in mind, the rendering equation can be written in a simple, operator form, which underlines its recursive nature:

$$L = L_e + M \circ L = L_e + M \circ (L_e + M \circ (L_e + M \circ (L_e + \dots = \sum_{l=0}^{\infty} M^l \circ L_e$$

The *K* operator is a matrix that contains all the object-object reflection interactions. The final result is a Neumann expansion. The sum is thus evaluated through averaging many paths through the matrix powers. The paths are **Monte Carlo integrated Markov chains** (MCMC), since each path is represented through a set of interactions, where each of them depends only on the previous. The paths are random walks, and a sufficient number of them can approximate the entire path space.

In a rendering meaning, path tracing solves global illumination by averaging many scene paths, which act as **global samples, instead of local samples** as used in other rendering algorithms. These global samples are Markov chains because each light-object interaction depends only on the previous interaction. Monte Carlo importance sampling is performed by taking into account the type of light-object interaction. Path tracing rapidly computes an average of many Markov Chains, with which it can reliably approximate the scene sampling space. The rendering equation can thus be written in a new recursive form:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \frac{f_r(x, \omega_i, \omega_o)L_i(x, \omega_i)(\omega_i \cdot n)}{pdf(x, \omega_i)}$$

$pdf(x, \omega_i)$ is the probability distribution function for sampling that path segment and it depends on the surface, type of path, type of light if directly lit, type of contact and contact angle.

There are various strategies for creating these pdf functions, many of which are described in the next section.

With this recursive form, path tracing is easy to implement and it also permits many importance sampling functions, which can be further tailored by surface properties and light positions. Many samples per pixel are averaged at the end to complete the Monte Carlo integration process. Without a **termination condition** a path would be infinite, therefore paths have a maximum length after which they end. Another method of early termination is the **Russian Roulette** (RR) [**Pha10**], which stochastically terminates paths. Russian Roulette uses a termination probability q. If the path is not terminated, the path is further explored with a probability of $1 - q$, and the integrand is over-evaluated with a weight of $\frac{1}{1-q}$, to effectively account for the skipped samples.

Because there are other optics principles not respected by original [**Kaj86**] path tracing, the original algorithm has problems sampling such effects. The original algorithm samples space through rays, which work under the principle of direction. Because of this, surface radiance accumulation is hard to sample, therefore representing sharp caustics with path tracing is inefficient. Because the original algorithm respects the principle of direction, subsurface scattering isn't supported, along with iridescence, chromatic aberration and fluorescence. Furthermore, similar to ray tracing, path tracing has problems with hard to sample lights like point lights. All of these problems are relaxed with improved versions of the path tracing, which provide better sampling for such difficult cases.

2.9.3. Improved Sampling

The quality of path traced image depends only on the level of convergence, and thus on variance. Generating better samples to reduce variance can be done in various modes: geometric aliasing is handled through pixel sampling, direct lighting can be importance sampled and multiple importance sampling, and so on.

Pseudorandom low-discrepancy sequences can greatly reduce geometric alias. Sequences such as stratified, Sobol, van der Corput, Halton or Hammersly guarantee lower variance then pure random sampling. Some of these sequences are presented in [**Pha10**]. Using such sequences instead of pure random sequences leads to quasi Monte Carlo integration [**Szi00**].

Adaptive sampling is a different kind of strategy, where additional samples are provided in cases of need. A simple strategy [**Pha10**] is to test the differences between the pixel samples and to progressively generate more samples until the contrast is under a certain threshold. A more advanced algorithm is presented in [**Dam09**]. The algorithm takes a global hierarchical approach to adaptive sampling by considering the path traced image as a hierarchy of converged blocks of varying size: from the size of the entire image to a small tile. The method starts with the entire image and adaptively generates new samples for each block until it is converged. When a block is converged the process repeats itself for all its sub-blocks.

Importance sampling (IS) is used to direct the random walk towards areas in the scene with more radiance potential, in order to speed up the rendering convergence. This can be implemented through direct lighting, especially for small or point lights. It can also be used to modify the probability distribution function for each light-object interaction, depending on the

type of interaction, for example specular interactions can sample very elongated lobes. Superior algorithms like [**Vea97**] [**Leh13**] use different types of importance sampling to efficiently resample entire path spaces.

Evaluating direct lighting can be improved with splitting [**Pha10**]. At a pixel sample level this distributes the entire samples among visibility samples and lighting samples, importance sampling direct illumination. While this can be useful in unbalanced scenes, correct distribution of the entire sample set requires significant preprocessing.

Naively combining more importance sampling mechanisms is difficult, because more often than not this will lead to an increase in variance, hampering and not helping the convergence rate. For example, a light-object reflection can be sampled both by the light BRDF and for direct lighting. If the BRDF is highly specular, than sampling by it is a better strategy. On the other hand if the light is very small, directly sampling it is a better strategy. Generally, importance sampling with multiple strategies is the superior approach. **Multiple Importance Sampling** (MIS) [**Vea97**] is a multi sampling model under which more sampling strategies can be used together without increasing overall variance. [**Vea97**] proposed two heuristic functions: balance and power. **Resampled Importance Sampling** (RIS) [**Tal05**] can be used to further reduce variance. Volumetric sampling can also be improved with equiangular sampling [**Kul12**] and Woodcock tracking.

Bidirectional path tracing (BDPT) [**Laf93**] uses multiple space sampling strategies, tracing paths from both the lights and the camera, which are then used as sub-paths in a fused path. The paths are linked in a manner that preserves the Markov Chain property of detailed balance. A spatial hierarchical hash grid [**Sch09**] can be used to accelerate the process of linking sub-paths, as it is used in a Vertex Connection Merging (VCM) [**Geo12**]. Path tracing and Bidirectional Path Tracing can both be implemented through rasterization, like in [**Tok12**].

2.9.4. Path Space Algorithms

Metropolis Light Tracing (MLT) [**Vea97**] applies the Metropolis-Hastings sampling method to path tracing and bidirectional path tracing. Compared to the previous algorithms, MLT considers the entire path as a set of samples. New paths are created by proposing mutations to the existing path, based on a **mutation strategy**. If the mutation is accepted a new path is created and sampled. Because of this strategy, MLT introduces a start-up bias and a correlation between samples, which is not found in classical Monte Carlo integration. Because correlation between samples increases variance and consequently convergence time, the mutation strategy has to keep the correlation low

While both MLT and (bidirectional) path tracing start from a global space sampling strategy, MLT evolves the paths in local space. MLT can first globally explore all the paths and then locally explore the promising path changes, it can quickly find and sample the most relevant paths in the scene. Moreover, when an important path is found, MLT also samples paths that are similar with it, increasing the chance of finding other important paths. Compared to standard path tracing and bidirectional path tracing, this makes MLT very efficient in difficult to sample scenes, like those containing caustics, small geometry holes or many glossy surfaces.

The mutation strategy is the most difficult part of the MLT algorithm, because it has to account for very different light paths which are sampled through different strategies. [**Vea97**]

recognized this problem and proposed a mutation strategies for many light path types. On the other hand [**Vea97**] did not provide a method to determine which mutation strategy to use, making the algorithm dependent on scene dependent parameter tuning.

Primary Sample Space MLT (PSSMLT) [**Kel02**] executes the mutations in a space of uniformly distributed numbers called primary sample space. This is based on the idea that any path generation is a mapping between the primary sample space and the path space, therefore the primary sample space is an uneven distribution which allocates more volume to more important regions, which in turn generates more samples in important path space regions. Therefore PSSMLT creates small mutations on paths that show a lot of potential by collecting significant amounts of radiance and large mutations to unproductive paths, based on the idea that the unproductive path is exploring a path space with less radiance potential.

Energy Redistribution Path Tracing (ERPT) [**Cli05**] combines the advantages of path tracing and Metropolis [**Vea97**], enabling path tracing to sample important paths. Path tracing generates completely new paths for each of the pixel samples, without using information from the previously evaluated samples, thus not using already available information. ERPT defines an energy flow in path space, where each path node holds a small amount of energy. Then, ERPT seeks to generate new paths that would sample areas that contain high energy. It basically redistributes energy from high energy nodes to nodes in newly generated paths that explore the same vicinity, and would thus have a high potential for energy accumulation. ERPT defines a mutation strategy based on the energy of the nodes. Consequently, new paths do not completely create different paths but mutate already existing paths, based on the energy of the nodes. This enables ERPT to quickly sample the space of light transport paths, without introducing a start-up bias like MLT.

Manifold Exploration [**Jak12**] is a special path sampling mechanism created to explore difficult specular paths. It does this by using available geometric information to define a manifold space on one or multiple path segments, which can then be explored. It permits a productive exploration of complicated specular paths like $LSSSSSSSE$. The algorithm can be applied to both path tracing (MEPT) and MLT (MEMLT).

Gradient Domain MLT [**Leh13**] improves difficult path exploration by reasoning that the most difficult to sample general paths lie at edges in the final image. The algorithm defines the gradient domain based on the edges determined from a coarse rendering of the scene. It then sends additional pixel samples in the pixels contained by this domain.

Multiplexed MLT [**Hac14**] combines the Markov Chain light transport path generation process with multiple importance sampling (MIS) principles. Thus the Markov Chain as light path does not only explore the space of light transport paths, but it also explores different sampling strategies for each path.

2.9.5. Accelerated Tracing

Even with the large number of optimizations, path tracing variants are still extremely heavy computational algorithms, because of the extremely large light transport path spaces that they sample.

Because of the vastness of the explored path space the most efficient way to sample various contributions is to importance sample for each of them. In contrast with bidirectional sampling, importance sampling, multiple importance sampling and resampling importance sampling, which seek to better sample space near paths, **relaxation techniques** seek to ease connecting subpaths. **Vertex Connection Merging** (VCM) [**Geo12**] relaxes the constraints of path connection by seeking for connectable paths in a vicinity. This method actually reformulates photon mapping in Veach's [**Vea98**] path framework, enabling photon mapping and path tracing to coexist in a single mathematical framework. Therefore the method benefits from the photon mapping capability to sample SDS paths, while still functioning like a PT method with regards to variance and convergence. Therefore VCM converges in $\frac{1}{\sqrt{N}}$ instead $\frac{1}{\sqrt[3]{N}}$ like SPPM. At first the lights emit photons which are traced with photon mapping algorithms and are deposited in the hierarchical hash-grid [**Sch09**]. The algorithm tries to connect each photon to the eye through either direct connection or through merging the photon, expressed as a light vertex, with a camera vertex in the proximity. The operations of connection and merging are linked with MIS.

Path space regularization [**Kap13**] introduces a path framework in which multiple sub-paths can be linked through interaction mollification, essentially following the same idea as VCM, but without using photon mapping.

Sorted Deferred Shading for Path tracing [**Eis13**] does not evaluate the ray batches used for visibility determination operations in path tracing until they are sorted in coherent batches, similar to the packet techniques [**Bou07**] from ray tracing. The novel aspect of sorted deferred shading for PT is the fact that it sorts ray hit point before shading, deferring the shading. Because the number of ray hits is very large this method **incoherently streams and then sorts ray-batches by material in order to coherently stream textures**. As production rendering textures occupy much more storage space than ray hits, the tradeoff brings with itself significantly lowered streaming costs.

There has been a lot of interest recently for **interactive path tracing**, but so far no algorithm can lower the variance fast enough on consumer hardware, although there are some promising implementations, such as the Brigade renderer [**Bik13**]. **Eye reprojection** [**Hen11**] can be used to provide additional radiance to the current image. It works by taking each interaction along the traced path and projecting the interaction contact point on the image film, thus reprojecting towards the eye. Each eye reprojection adds to the image film a reflection of the surface contact incoming radiance, effectively obtaining very cheap samples, which increase convergence rate. **Streaming path tracing** [**van11**] is a GPU specialized PT variant, which uses the potential equation along with a recursive form of MIS. It functions by generating a large number of paths and then bundling them into batches, which run on GPU streaming processors. Since streaming processors are very performance sensitive to incoherency, the batches are kept coherent through **stream compaction**, which eliminates the finished paths from the batches, and then compacts the remaining paths into new coherent batches.

Instant Bidirectional Path Tracing [**Bog13**] uses two different GPU passes for light path tracing and normal path tracing. It then creates paths using the previously computed subpaths. The resulting paths can be explicit, if they are connected, or implicit, if a light path touches the camera or a normal path touches a light.

Noise removal techniques try to lower the variance by using either unbiased or biased means. The simplest and biased solution is to use a form of blurring augmented with geometric information, like a bilateral filter. An unbiased solution is to filter only light that went through a path with more **consecutive diffuse reflections** [**Jen95**]. A pair of consecutive diffuse interactions produces a large amount of diffusion, and filtering this result would not create visible bias. Other filtering methods include **temporal light field reconstruction** [**Leh11**], where a costly reconstruction process is used, and **radiance filtering** [**Sch12**]. In contrast to image filtering techniques, radiance filtering uses radiance samples from neighboring pixels. The algorithm stores the last intersected surface for each radiance sample. When filtering, it re-reflects the radiance by the BSDF of the last intersected surface, towards the center of the filtering kernel (the filtered pixel). By doing this, extra free contributing samples are obtained, which improves the convergence rate of the rendering.

Noise removal techniques decouple scene complexity from light transport complexity. A different type of noise removal algorithm is based on the idea that the primary hit points, those that are directly visible to the camera, are in general illuminated by the same radiance coming from the path that would continue from that vicinity. Therefore, a single path is completely traced per pixel, while a large number of direct rays are shot in the vicinity of the pixel. Each direct ray is then connected to the traced path, with regards to connection angle and direct visibility, basically reconstructing uncomputed paths.

A relatively novel method for path tracing acceleration is using scene **skeletonizations as importance sampling** [**Bir12**] [**Cha13**], which uses scene geometry skeletons to aid path tracing explore complicated scenes, where traditional algorithms find it difficult to sample important light paths. The algorithm uses a coarse tridimensional voxelization of the scene, which is then inverted and skeletonized. The skeleton is then used to compute importance vectors, which will direct light transport to the most efficient light transport space in the entire scene. The lights of the scene and the camera are also given importance in the skeleton creation. A skeleton sample point contains information about the distance and direction that is expected to be the most productive to explore, for a path arriving at the vicinity of the skeleton sample point. Thus, scenes with complicated light transport setups, like those containing slightly opened doors, long corridors or holes, become much easier to explore. Compared to the local importance sampling solutions found in Monte Carlo path tracing, skeleton based importance can be seen as a global importance sampling method. Instead of looking for good mutations which explore important light paths, skeleton based path tracing looks for efficient connections between the most important features of the scene – linked together by the skeleton – and the rest of the objects.

The skeleton contains additional information, like the distance and direction of the closest surface, and this information can be used to improve path tracing by shooting in hard to sample areas, like two room which are only connected by a barely open door.

Some of the most relevant path tracing methods are presented in Figure 19.

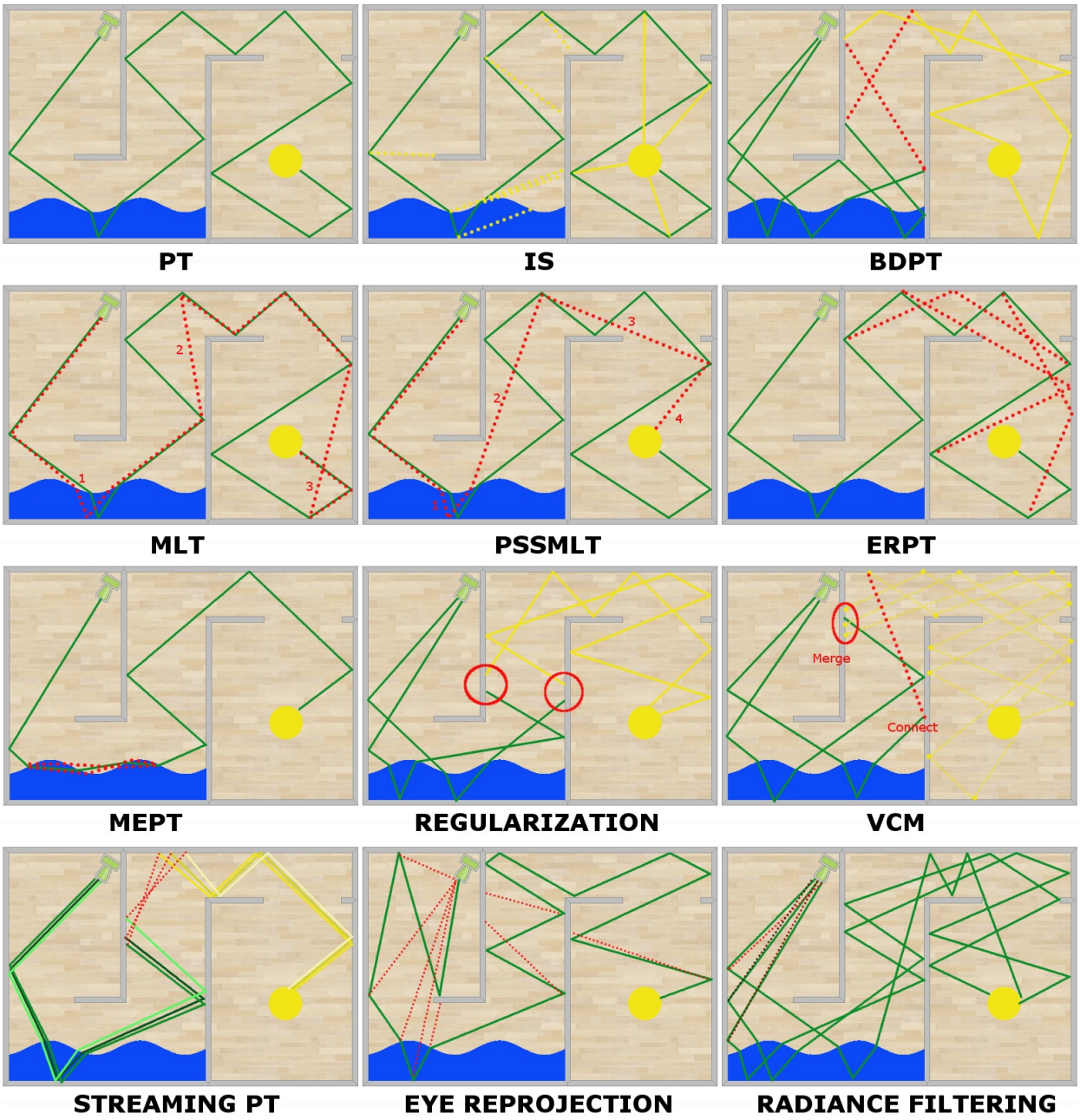

Figure 19 Path tracing**.** Path tracing (PT) explores the object-lights interactions by following many paths through the scene. Importance Sampling (IS) can be used to improve the light path space exploration, for example by directly sampling lights at each contact. Bidirectional PT (BDPT) is a form of importance sampling that links paths from the lights and from the camera, making difficult paths easier to sample. Metropolis Light Transport (MLT) mutates previous productive paths with different strategies. Primary Space Sampling MLT (PSSMLT) improves on MLT by using large mutations, when a path is unproductive, while keeping the small mutations to explore productive paths. Energy Redistribution PT (ERPT) mutates the energy-rich nodes of previously traced paths. Manifold Exploration PT (MEPT) can efficiently explore specular subpaths. Regularization mollifies subpath connections, enabling the linkage of close light and camera path nodes. Vertex Connection Merging (VCM) unifies path tracing and photon mapping. It first shoots photons, which are stored in a spatial acceleration structure. In a second pass, it traces paths, which are linked to the photons by either merging or connection. Streaming PT groups paths into groups, which are compacted when the group divergence reaches a threshold. Eye reprojection projects each path node onto the camera, creating free radiance samples. Radiance filtering uses information from neighbors.

2.10. Photon mapping

Photon mapping (PM) [**Jen96**] algorithms are based on the idea of light diffusion, approximating light transport through photon transportation. Photon Mapping considers photons as small packets of energy, which are described by position, direction and power. Compared to path tracing [**Kaj86**] the algorithm is **biased**, in that many different renderings will not converge, but it is also **consistent**, meaning that more photons will increase accuracy. More advanced variants of photon mapping, which combine shooting photons with measuring procedures relax the biasing [**Hac09**]. Photon mapping has been used to render **photorealistic** images and is a **top down** approach. Photon mapping trades the noisy artifacts found in path tracing for bias induced structured artifacts.

Compared to ray tracing, photon mapping methods were created to better sample complicated light paths like SDS (reflexion of caustics) which would otherwise require a very large number of samples to render properly with PT and RT algorithms. While photon mapping can be considered a many lights approach to rendering, it differs from many lights methods. The difference is that instead of spawning new direct illumination approximate virtual lights, photon mapping actually traces photons. Compared to many lights methods, photon mapping a number of photons is orders of magnitude larger than the number virtual lights, therefore the algorithm degrades gracefully by blurring radiance and not by eliminating lights.

Photon mapping [**Jen96**] works in two steps. In the first step a photon map is created by **tracing (shooting) photons** from the scene lights, which are then stored in an acceleration structure. This acceleration structure can be either surface based, storing photons on the surfaces of the objects of the scene, or volumetric, storing photons in space partitioning structure like a kd-tree. The latter type is used when the simulation of volumetric effects is intended. In the second step, called **final gather**, the camera-objects interactions are determined through an algorithm like ray tracing or rasterization and the illumination is solved using the light transported in the first step. Usually there is more than a single photon map, one for diffuse surface interaction and one for pure specular interactions. A study on the acceleration structure and reconstruction filters that can be used to implement photon mapping in real-time is given in [**Mar13**]**.** Photon maps can be extended to **ray maps** [**Hav05**], which index the rays that are followed by the photons which would normally be stored in the photon maps.

Bidirectional Photon Mapping (BDPM) [**Vor11**] combines path tracing and photon mapping. It first uses a photon map to store the photons traced with photon mapping. It then uses a path tracing algorithm to determine the surface interactions which bring radiance to the camera. For each vertex in each path the photon map is queried for photons and radiance is accumulated at the vertex. This algorithm is extremely similar to VCM [**Geo12**].

Progressive photon mapping (PPM) [**Hac08**] reverses the order of PM passes. In the first stage it does a ray tracing pass, to determine all the surfaces which are visible to the camera, named surface hits. These camera-object interactions are stored in a list. In the second stage PPM uses a progressive method to collect photons. Over many passes photons are shot from the lights and they are accumulated on the surfaces visible from the camera. A special metric is used to determine the radius of the surface hit. With each new photon pass, each surface hit is updated with new photons and has its surface reduced, based on a progressive radiance estimate.

The key idea behind PPM is that it **averages the results of many photon maps** in order to ensure that the surface hit local radiance converges to the correct result, making PPM unbiased. Another very important aspect of PPM is that it erases the enormous memory requirements needed to implement PM with convincing results. Only the key photons, those that directly impact the visual result, are stored. Furthermore, progressive photon mapping passes permit tracing a relatively low number of photons per pass, making streaming not necessary. PPM can be improved with adaptive sampling, as done in Adaptive Progressive Photon Mapping [**Kap131**] or through blue noise based reconstruction, as in [**Spe13**].

Stochastic progressive photon mapping (SPPM) [**Hac09**] improves PPM by computing the radiance in a region, and not in a single point as PPM. This makes SPPM useful for region effects such as depth of field and antialiasing. Instead of using only progressive photon passes like PPM, SPPM uses a progressive cycle, in which distributed ray tracing and photon tracing passes follow each other. The distributed ray tracing step generates new randomly distributed surface hits that can be seen from the camera. The photon passes work exactly like in PPM. By doing this SPPM ensures that the region effects are uniformly sampled, instead of just accumulating radiance for the single surface hit samples generated by one ray tracing pass.

Volumetric effects can be rendered with photon mapping algorithms by using a volume (volumetric) photon map [**Jen98**]. This map stores only the photons that interact with participating media. Implementing volumetric effects with volume photon maps is computationally expensive, because the acceleration structure that stores the photons has to be queried many times, in order to evaluate in-scattering done on the traced distance. **Ranged queries** in the photon acceleration structure are used, in order to find all the relevant photons for the marched distance. If the distance is sampled in small steps, there will be overlapping in the sampling of the photon acceleration structure, and the same photons will be found multiple times. If the distance is sample in large steps, some important photons will be missed.

Photon beams (PB) [**Jar08**] permit finding all the photons that influence a marched distance over a ray in a single query. Photon beams are not cylindrical, but have an adaptive form, modeled by the photons that interact with the marched ray. Implementing photon beams starts with a variant of photon mapping algorithm that stores all the photons in a balanced kd-tree. Then, each photon stored in this acceleration structure is given a radius, effectively creating photon disks. A **second acceleration structure** is created over the photon disks. This second acceleration structure is an object partitioning structure like a BVH, compared to the space (and points) partitioning structure used for photons. Photon beam tracing then marches a ray by querying photon disks in the second acceleration structure, through cheap intersection tests. Thus, instead of using many ranged queries in a photon acceleration structure, the photon beam tracing uses only intersection tests with the photon disks, greatly lowering the search complexity. The cost of creating the second acceleration structure is minor, as it is performed only once for the entire scene. The performance improvement offered by lower search complexity impacts rendering time much more, due to the very large number of search operations.

Progressive photon beams (PPB) [**Jar11**] improves photon beams, like PPM improves PM. By averaging many photon beam passes, each with a progressively decreasing photon disk size, PPM guarantees convergence, and is therefore unbiased.

Photon mapping algorithms are presented in Figure 20.

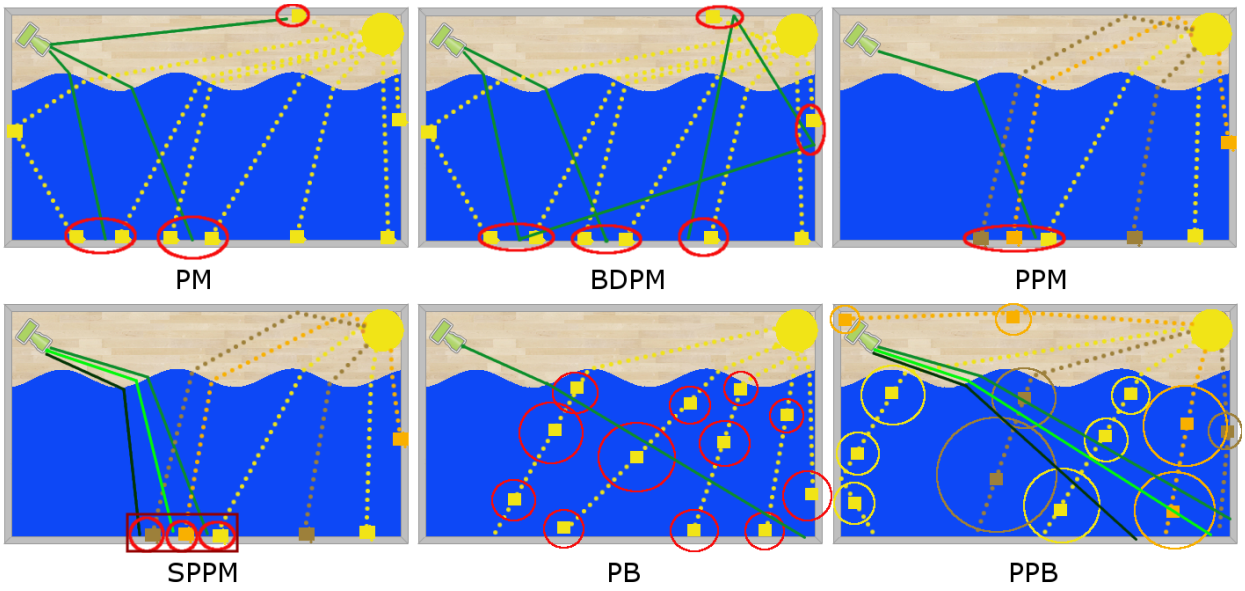

Figure 20 Photon Mapping**.** Photon mapping algorithms use photons to simulate light transport in the scene. Photon mapping (PM) traces photons from the light sources and stores then in a balanced kd-tree, called photon map. In a second pass the final image is synthesized by using a ray tracing pass, which illuminates with the radiance stored in the photons queried from the kd-tree. Bidirectional Photon Mapping (BDPM) combines normal photon mapping with path tracing. The radiance collected by the path is obtained by querying the photon map in the vicinity of each path node. Progressive Photon Mapping (PPM) reduces the large memory costs of photon mapping by tracing the photons in many progressive passes around camera-surface hits, which shrink with each pass. Stochastic Progressive Photon Mapping (SPPM) makes PPM unbiased, by using a cycle of photon mapping and distributed ray tracing algorithms. On each iteration photon mapping is performed like in PPM, but the radiance is used for illumination by a different random ray, generated by DRT. Photon beams (PB) decreases the large number of samples needed to perform volumetric photon mapping, by creating an additional acceleration structure which stores photons as objects with size, called photon disks. Progressive Photon Beams (PPB) computes radiance by intersecting random DRT generated camera paths with progressively smaller beams, obtained by shrinking the photon disks from PB.

2.11. Many lights methods

A different approach to solving global illumination can be implemented through approximated radiance transport. While photon mapping transports a very large number of tiny radiance quantities through photons, many lights methods transport a relatively small number of large radiance quantities. The key idea behind many lights is that global illumination can be approximated with direct illumination from many small lights, created at the contact of light from with object surfaces. These lights are called **virtual lights** [**Kel97**]. While the resulting method is **biased**, it can be extended to handle diffuse reflections [**Dac05**] and many shadow maps [**Rit08**] techniques can be used to augment the final rendering, but it can't solve specular reflections and difficult caustics paths like SDS. Advanced many lights methods mollify the bias introduced by virtual lights [**Nov11**] and improve the distribution and scalability of the virtual lights. Similar to rasterization, many light methods are more of a **bottom up** approach to rendering, and, like rasterization, they are also one of the most active fields of research for real-time rendering. Many light methods can be categorized into virtual lights **generation**, **illumination** with virtual lights and **scalability** improvements, as presented in [**Dac14**].

Like photon mapping methods, many light methods trade the noisy variance related artifacts found in ray tracing and path tracing for structured artifacts. Compared to photon mapping they do not model light transport as shooting photons from the light but by spawning new lights at light-surface interaction. While photon mapping considers photons as incoming light, many light methods consider virtual lights as outgoing light.

Many light methods have a common history with **radiosity**, which is a finite element method, in which all the surfaces are diced into surface patches, which are linked by a geometric term. The geometric term, also called a **form factor**, approximates the inter-visibility operation and BRDF reflection between to surface patches and it was originally defined for Lambertian [**Edw03**] surfaces. With this form factor radiosity is globally exchanged between all the surface patches of the scene, effectively simulating diffuse light transport. The hemicube generalizes the form factor, allowing arbitrary scenes with internal occlusion. Because solving all the light transports between surface patches is a large computational task, radiosity is usually implemented over low resolution meshes, and the results are generally cached. Because of this, radiosity isn't well suited for dynamic environments or for use as a single global illumination solution for photorealistic images.

Instant radiosity (IR) [**Kel97**] introduced the concept of (**many**) **virtual point lights** (VPL), as direct illumination sources spawned by simulated light transport. The method's name implies that this is a variant of radiosity, based on the idea that virtual lights transport radiance between them in a similar manner to the inter surface patch transports in radiosity. In reality, virtual lights are generated with random walks, and thus can be considered elements in light paths, where one end of the path is a real light and all the other nodes are only linked to their parents and children. Thus, instant radiosity differs fundamentally from the finite element approach to global illumination, because virtual lights are only linked to their parents and their children, and not with virtual lights from other light paths. Therefore, instant radiosity creates a path of virtual lights, where light transport has nothing in common with the original radiosity algorihtm. Furthermore, this can be extended through other methods to virtual light trees.

Instant radiosity has much more in common with photon mapping, because virtual lights are better conceptually perceived as supersized photons. Instead of shooting photons, instant radiosity shoots large bundles of photons, which generate **virtual lights**, which in term shoot large bundles of photons, which recursively generate other new virtual lights.

2.11.1. Generating Virtual Lights

The efficient generation of virtual lights can lead to a large decrease in rendering time. Only the virtual lights that interact with the objects which are visible from the camera have any visual contribution. The task of efficiently generating virtual lights is to determine the best virtual lights that lead to the most interactions with directly visible objects.

A naïve generation of virtual lights can be done by performing **random walks** from the scene lights, as it was performed in [**Kel97**]. This strategy explores the entire scene and will generate a new virtual light at each light-surface interaction, constructing many light paths. Another variant is to use a **distributed ray tracing** to generate a tree of virtual lights.

Creating many virtual lights can be performed in a single rasterization pass, by using **Reflective Shadow Maps** (RSM) [**Dac05**]. This method uses a buffer similar to the G-buffer

[**Ols11**] in deferred rendering, but at a much lower resolution. The buffer stores information about the world position, world normal and the outgoing unshaded color. Furthermore, each entry in this buffer generates a virtual point light, which uses the position, direction and color provided by the entry in the RSM. The process can be repeated for each light but without a bandwidth reduction technique like [**Rit08**] the rasterization costs greatly increase.

A relatively simple way to generate better VPLs is to use Russian Roulette on the VPLs generated by instant radiosity, based on the approximate impact on the entire screen, as done in the **rejection of unimportant VPLs** method [**Geo10**]. The method can be initialized with a small number of pilot VPLs. The disadvantage of the created VPL distribution is that it is not good for interreflections.

Bidirectional Instant Radiosity (BIR) [**Seg061**] uses a bidirectional approach towards VPL generation, connecting subpaths from the camera and from the scene lights, in a manner similar to bidirectional path tracing [**Laf93**]. The advantage of this method is that it rejects all the lights which are not in the second place in the light paths, coming from the camera. Therefore, the generated VPLs will all generate visible indirect illumination.

Metropolis Instant Radiosity (MIR) [**Seg07**] is a modification of Bidirectional Instant Radiosity, which first generates naïve light paths, as they are generated by the random walks in [**Kel97**]. It then mutates the paths with the Metropolis-Hastings algorithm, in order to direct them towards the camera. A large proportion of the mutations change the position of the VPL second from the camera. For illumination, the algorithm considers only the VPLs in the generated light paths, which are either first or second from the camera, similar to BIR. By doing so, MIR generates only VPLs that are directly responsible for camera visible indirect illumination. Furthermore, by using Metropolis-Hastings, this VPL generation scheme is guaranteed to efficiently transport light even in hard to sample scenes, mutating paths like MLT [**Vea97**].

Local Virtual Lights [**Dav10**] compensates the specular light transport, as it is not accurately modeled by virtual light methods. It splits light transport into global and local components. For the local specular component it generates local virtual lights, which are used to better approximate glossy reflections. VPL generation strategies are presented in Figure 21. Another method which improves specular light transport in many light methods is based on **rich VPLs** [**Sim15**].

In order to lower the computational effort in dynamic scenes temporal and spatial **caching** can be used, which are based on the idea of either spatial or temporal coherence between frames. The simplest form of caching can be implemented through Irradiance Caching [**War88**], but this stores only the integrated radiance at a surface patch, and has no angle information, therefore it can't be used in scenes with specular features. Radiance caching [**Kři07**] stores angle information and permits glossy light transport, but does so at greatly increased memory costs. These caching strategies can be translated to work with virtual lights. Instant Caching [**Deb09**] reutilizes computations from virtual lights, based on spatial and temporal coherence. Importance Caching [**Geo121**] implements a variation of multiple importance sampling in order to determine the cached light information that will be used in illumination evaluation.

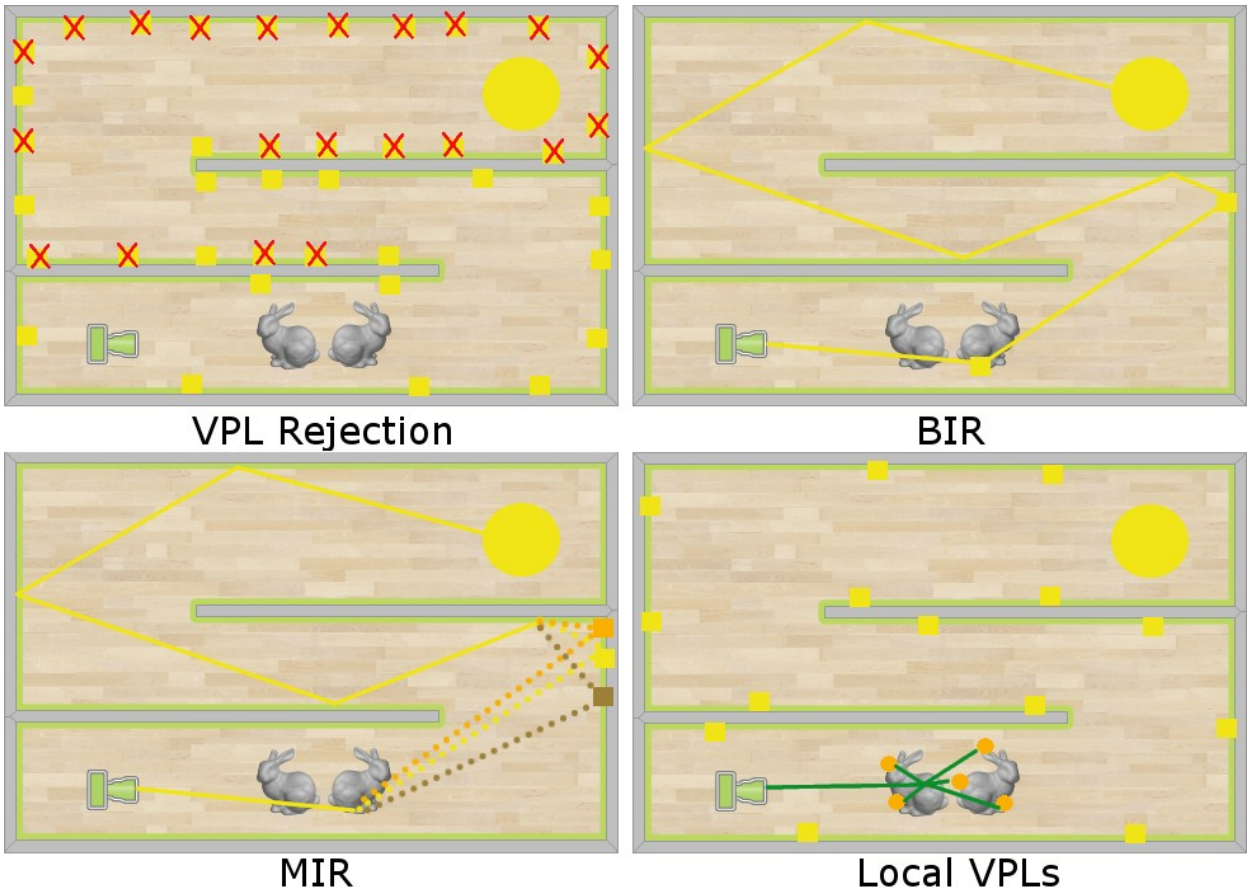

Figure 21 Virtual Light Generation**.** There are many strategies for the generation of virtual lights. Virtual point Light (VPL) rejection approximates the entire screen as a single pixel and computes the potential energy for each random walk-generated VPL, rejecting VPLs with low potential contributions. Bidirectional Instant Radiosity (BIR) connects subpaths from the light and from the camera, generating VPLs at the 1st and 2nd vertices from the camera, thus ensuring that all the generated VPLs are guaranteed to impact the visual result. Metropolis Instant Radiosity (MIR) improves BIR by mutating the BIR paths in the vicinity of the 2nd vertex from the camera, generating more important VPLs. Local VPLs can be used in combination with any of the previous strategies. Local VPLs is aimed at improving specular light transport and it does this by generating many VPLs local to the 1st vertices from the camera, which are directly visible from the surfaces directly visible to the camera. It then computes the total illumination as a sum of diffuse illumination computed with global VPLs and a specular component computed with the local VPLs.

2.11.2.Illumination with Virtual Lights

Lighting with virtual lights follows the same general principles as the rendering equation, where light is transported through a random walk, only that the many light random walk does not transport tiny radiance quantities but large quantities, which then spawn virtual lights. The equation for virtual light illumination is:

$$L(x_1 \to x_0) = L_e(x_1 \to x_0) + \int_A f_r(x_2 \to x_0) G(x_1, x_2) V(x_1, x_2) L(x_2 \to x_1) dA(x_2), where$$

$$G(x_1, x_2) = \frac{\cos\theta_1 \cos\theta_2}{\|x_1 - x_2\|^2}$$

This equation can be written in a recursive form, given the fact that VPL energy is transferred over random walk generated paths, thus the radiance can be approximated with:

$$L(x_1 \to x_0) \approx L_e(x_1 \to x_0) + \sum_i^N f_r\left(x_2^i \to x_0\right) \frac{\cos\theta_1 \cos\theta_2^i}{\left\|x_1 - x_2^i\right\|^2} V\left(x_1, x_2^i\right) f_r\left(x_2^i \to x_1\right) \Phi_{\mathrm{i}}$$

Where $f_r\left(x_2^i \to x_1\right)\Phi_{\mathrm{i}}$ represents the flux reflected to the current VPL on the specified direction, from the previously walked VPLs in the VPL light path. A great problem with this equation is that the geometry term $G(x_1, x_2)$ is a source of singularities, which manifest themselves as structured artifacts of maximum radiance. These singularities are sometimes called radiance splotches, or radiance stains in rendering.

There are two major approaches in limiting geometry term induced illumination singularities: **bounding the geometry term** and the **redistribution of the VPL energy** over an area or volume. Bounding the geometry term is basically a clamp operation, and is therefore easy to implement. On the other hand bounding the geometry term implies a loss of radiance, which manifests itself as synthesized images with less energy, thus darkened results. In order to compensate for the lost radiance a residual light transport term is defined as:

$$T_r L(x_1 \to x_0) = \int_A \max(G(x_1, x_2) - bound, 0)\, V(x_1, x_2) L(x_2 \to x_1) dA(x_2)$$

The entire energy of a path is the sum between the normal clamped radiance and the residual radiance. **Bias compensation** was introduced in [**Kol04**]. This method stochastically tests if the energy computed at a shading point is under-estimated, by randomly shooting a ray in the bounding vicinity. If the ray hits a surface, lighting is evaluated with nearby VPLs and the energy is compensated with the resulting radiance. But the compensating radiance is itself computed with clamping, therefore the method repeats the process of randomly shooting a ray, until it does not hit a surface. The method statistically recovers all the missing radiance, but is very expensive and can degenerate to path tracing.

The bias compensation method is improved in **Local Virtual Lights** [**Dav10**], where the random ray shooting is not a recursive process, and it is shot only once. If there is a point of contact inside the bounding vicinity, then that point is itself transformed into a virtual light, called local virtual light. The local virtual light receives radiance from all the nearby existing virtual lights, called global lights, and is then used in the illumination of not only the shading point (pixel), but in the illumination of an entire vicinity around the shading point. Thus, the algorithm is much faster than [**Kol04**], but it is also less correct as it represents an approximation.

Screen space bias compensation [**Nov11**] is a fast method to compute the bias compensation term. It works as a post processing filter, which computes the bias in an iterative manner, as a sum of residuals. The method uses only the information available in the vicinity of the shading point. The method also uses the observation that there rarely is occlusion in this vicinity. The method is further refined for participating media in [**Eng12**].

While the bias compensation methods try to recover lost energy due to bounding, redistribution of energy methods spatially spread the radiance, in order to avoid singularities.

Virtual spherical lights (VSL) [**Hašš09**] distribute the energy of the virtual point light over an entire sphere, basically inflating the VPL. Thus, the light is not transferred from the VPL to the shading point on a point-to-point basis, but is evaluated through an integral over the solid angle subtended by the VSL. This method correctly computes the radiance in the shading point, without losing energy, but by the evaluation of the integral for every shading point makes it computationally heavy.

Virtual Ray Lights (VRL) [**Nov121**] improves illumination with virtual lights in **participating media** by using rays instead of points. Instead of storing the entire radiance in a single VPL, VRLs distribute the radiance over an entire ray. [**Nov121**] provides importance sampling mechanisms, in order to simplify computations. VRLs are further improved in Progressive **Virtual Beam Lights** (VBL) [**Nov122**], where the rays have volume. This leads to improved scattering, as each VBL distributes energy over an entire volume.

Virtual Area Lights (VAL) [**Don09**] [**Pru12**] cluster virtual lights, in order to minimize the number of shadow maps that have to be computed. They are usually represented with easy to sample forms, such as disks. Virtual light types are presented in Figure 22.

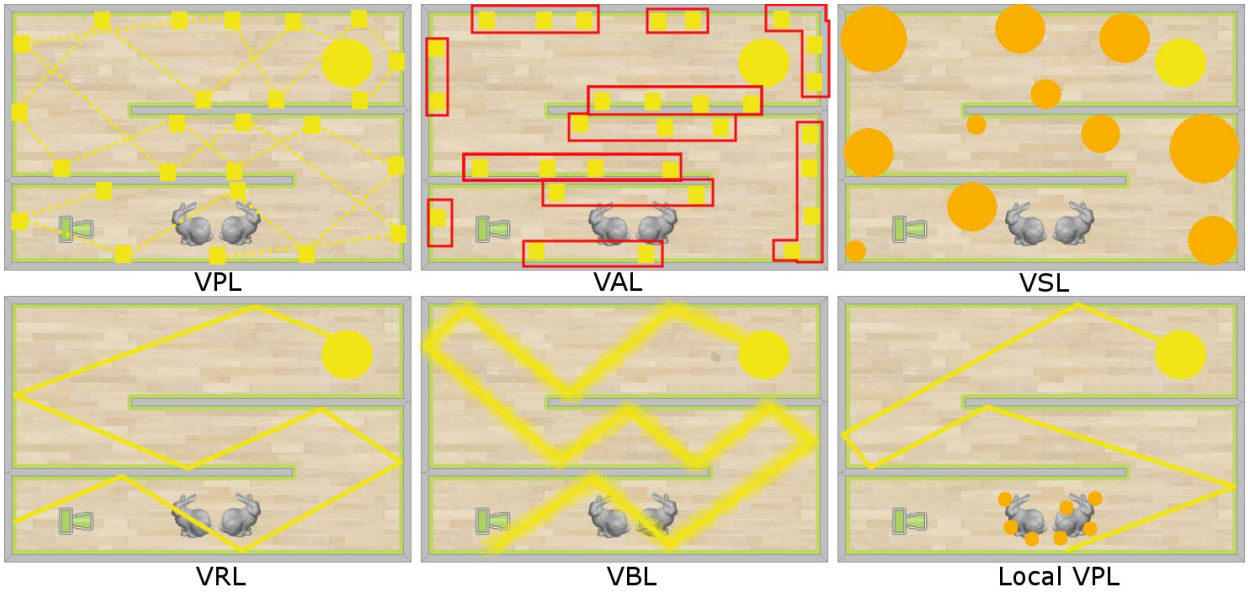

Figure 22 Virtual Light Types**.** Correctly illuminating with virtual lights requires a very large number of them, hence the many lights name. Virtual Point Lights (VPL) are the simplest type of virtual light, but are also the most problematic, suffering from singularity and scalability, as effects of sampling space with points. For this reason, there are many types of non-point virtual lights. Virtual Area Lights (VAL) are obtained by clustering many close VPLs, and usually take an easy to integrate form like a disk. Virtual Spherical Lights (VSL) inflate VPLs, distributing their energy over an entire volume, making them more expensive to evaluate but also not singularity prone. Virtual ray lights (VRL) are virtual lights that improve scalability in participating media rendering, by distributing the virtual light radiance over an entire path segment. Virtual Beam Lights (VBL) are extensions to VRLs, which provide additional radiance distribution, over the entire volume of each beam. Local VPLs are generated near specular objects, to better approximate glossy light transfer.

Shadowing virtual lights requires very fast visibility determination operations, because of the large number of virtual lights generated with instant radiosity algorithms. Because of this, full geometry processing for the entire scene is usually too expensive, either rasterization for normal shadow mapping or visibility determination through rays. Instead, approximate visibility

determination is computed through either **point based** methods or through **level of detail** methods.

Point based methods solve the problem of visibility by approximating geometric information through points. As points are not surfaces, the bandwidth processing is greatly reduced. **Imperfect Shadow Maps** (ISM) [**Rit08**] are shadow maps constructed over sparse scene geometry, represented through points clouds. After the ISM is filled with rasterized points, it uses a push-pull process in order to fill out the rest of the empty entries in the shadow map, which is implemented with mipmaps. ISMs can be implemented with a hierarchical point based rendering system like Qsplat [**Rus00**], or the one implemented in [**Rit091**].

Level of detail [**Hol11**] based approaches decrease the number of computations for the many shadow maps needed by many light methods. Another method to implement a large number of shadow maps is through virtual clustered shadow maps [**Ols141**]. A different form of level of detail can be implemented by tracing rays through an approximate voxelization of the scene.

2.11.3.Scalability

Rendering with a large number of many lights is a difficult computational endeavor, thus research in this area has proposed a large number of solutions. The purpose of this research was to lower the illumination complexity from a linear cost in generated VPLs.

A relatively simple solution is to distribute the lighting effort over regions, and then to compose the illumination results. This can be achieved through interleaved sampling [**Kel011**], in a manner similar to interleaved deferred [**Seg06**]. The same principle was also the basis for Incremental Instant Radiosity [**Lai07**].

Lightcuts [**Wal05**] converts all the existing lights to VPLs. It then takes all the generated VPLs and constructs an acceleration structure over them, dividing them into a tree of clusters. Each cluster is approximated by a cut, which is a set of nodes which partitions the lights into clusters. This partitioning is based on the insurance that each cluster remains under a certain error threshold. Thus, this method guarantees that the important lights will always be used in the illumination process. Multidimensional Lightcuts [**Wal061**] improves the Lightcuts method by taking into account pixel level effects such as depth of field, motion blur and participating media. It does this by creating a separate tree structure defined through cuts for sampling points inside a pixel, called gather points. Similar to Lightcuts, Multidimensional Lightcuts works only with VPLs. Bidirectional principles are applied to Lightcuts in [**Wal12**], improving glossy reflection and subsurface scattering support. Progressive Lightcuts [**Dav12**] provides better clamping and drastically reduces the memory costs of the original algorithm

Matrix row-column sampling [**Hašš07**] is an alternative to lightcuts. It interprets the general problem of illumination with many lights as a virtual light pixel global intersection test, which can be written in matrix form, where a row for each pixel and a column for each virtual light. Therefore, the problem of illumination reduces to the problem of computing the sum of contributions, on each column. The algorithm is based on the observation that the resulting matrix is very structured and is close to low-rank, thus only a small subset of elements can be computed to approximate the final result, therefore computation will be performed only for a small number of columns. In order to accelerate the computation the algorithm makes shrewd

use of shadow mapping. Each column-row intersection is equivalent to determining the visibility of many pixels by a single light. Likewise, each row-column intersection is equivalent to determining the visibility of a all the lights from a pixel. Both of these one-to-many visibility problems can be efficiently solved through shadow mapping. The key of the algorithm is in **how the columns are selected** for computation. The algorithm randomly selects a small number of rows in the matrix and creates a new matrix with reduced columns, which basically reduces the size of the illuminated image, similar to mipmapping, and results in a disproportionately wide matrix. The reduced columns are then partitioned into clusters, based on the norm of the multi-dimensional vector, which represents each reduced column. In the end, a column and its weight are chosen from each cluster, based on the multidimensional vector norm metric. The entire un-reduced columns are computed, and the entire matrix is thus approximated through the weighted composition of a small number of columns.

Other clustering strategies are based on grouping many VPLs into congregated lights, like virtual area lights [**Don09**] [**Pru12**].

3. GEOMETRY PROCESSING

This chapter describes a part of the proposed rendering pipeline which exclusively deals with direct visibility determination, computed with rasterization. The modules and algorithms presented in this chapter are succinctly depicted in Figure 23; the green modules represent thesis contributions.

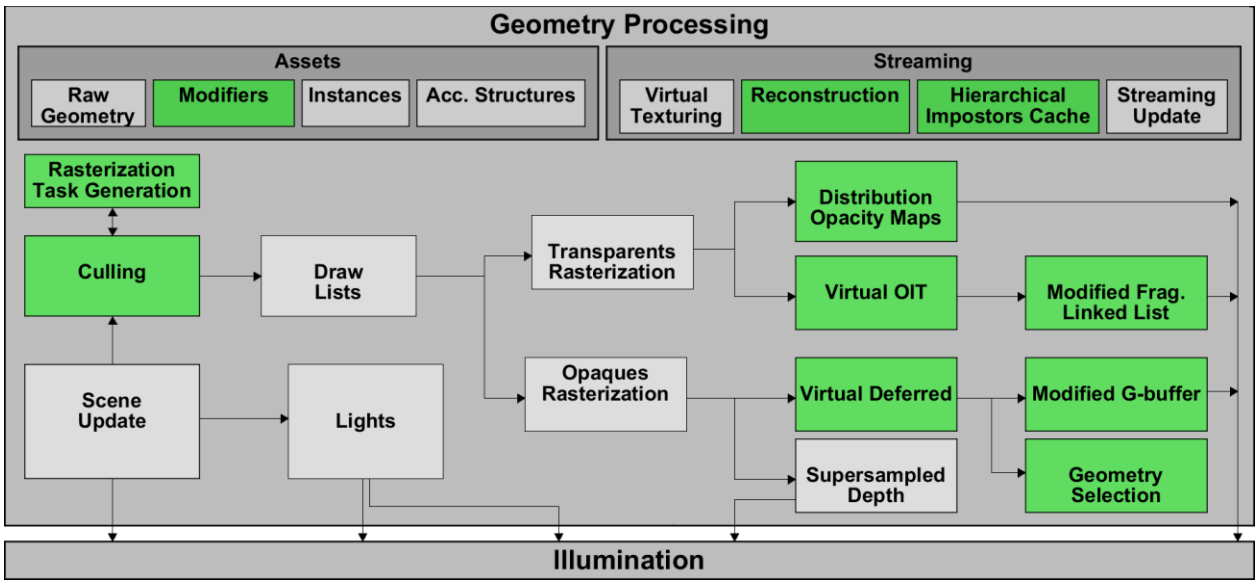

Figure 23 Geometry Processing Overview**.** The geometry processing pipeline includes asset definitions, streaming mechanisms and rendering paths for opaque and transparent objects. The outputs of this pipeline are then used by the Illumination pipeline. The green modules contain thesis contributions: modifier based assets, an altered marching cubes algorithm for very large datasets, scene-wide hierarchical impostors, task generation within the rasterization scheduler and a culling mechanism that uses it, distribution opacity/occupancy maps and virtual order independent transparency for approximate and exact transparent objects rasterization, virtual deferred for opaque rasterization and a novel geometry selection algorithm.

The geometry processing part of the proposed pipeline contains asset definitions, which are based on the concept of modifiers and instances, which help in the succinct description of very large scenes which use common rendering data. The asset definition also contains the pre-computed acceleration structures needed for rendering such as bounding volume hierarchy trees or sub-geometric detail maps.

A comprehensive streaming mechanism is presented, based on virtual texturing, virtual meshes and a novel hierarchical impostors cache method, which is implemented through virtual texturing. The hierarchical impostors cache method creates impostors for entire scene nodes and this makes it very useful in rendering effort scaling and as an anti aliasing solution. The streaming modules contain a novel **adaptation of the marching cubes GPGPU algorithm**, which is able to handle very large datasets. The streamed data is introduced in the geometry processing part of the rendering pipeline through the scene update module.

The geometry processing algorithms described in this chapter use the streamed data to produce outputs for the illumination stage. The active scene lights are extracted from the scene update module. This chapter presents a **task generation** method that works within the rasterization scheduler. This method is then modified, and used as a component of an innovative **hierarchical culling** algorithm, which neither suffer from CPU dependence nor requires special geometry impostors. The culling module also employs a new approach to culling, by using multiple frame tests, which cull objects over a number of frames. Because of this, the presented culling method is less computationally expensive than the state of the art.

Draw lists contain the objects which are not culled by the culling module. The geometry processing pipeline contains several rasterization based rendering paths, which are followed by the geometry of the objects contained by the draw lists. The rendering paths apply standard state of the art rendering techniques such as displacement mapping, sub-geometric rendering or geometric anti-aliasing.

Virtual deferred is a new opaque rendering deferred method. In contrast with the state of the art methods it not only decouples the visibility determination and shading components, but also minimizes allocated memory and bandwidth consumption, by guaranteeing minimal non-geometric bandwidth and material-independent allocated memory costs. While virtual deferred contains a decoupled shading part, only the geometry processing side of the algorithm is presented in this chapter, the other part is presented in the Illumination chapter. Virtual Deferred can be minimally modified to render antialiased images, by using the decoupled sub pixel reconstructed antialiasing method presented in the Illumination chapter.

Virtual OIT applies the principles used in virtual deferred to the problem of transparent geometry rendering. Compared to state of the art methods it can compute exact order independent transparency while only using a fraction of the allocated memory and bandwidth. As with virtual deferred, the shading side of the algorithm is not presented in this chapter, but in the Illumination chapter.

Distribution opacity maps modify the state of the art occupancy maps with per-pixel distributions, making them much more adaptable to real-life scene depth distributions. Because of this increased depth precision, distribution opacity maps are more memory efficient than occupancy maps in approximating order independent transparency. The downside of this method is that it has to reprocess the entire geometry in order to compute shading.

The pipeline also contains a novel **geometry selection** algorithm. Compared to the state of the art methods, the introduced algorithm handles all types of selection cases, such as multiple objects per pixel, area selections or even occlusion from fuzzy objects.

The geometry processing pipeline creates the inputs for the Illumination pipeline. The modified linked lists, produced by the Virtual OIT algorithm, and the modified G-Buffer, produced by the virtual deferred algorithm, are used together with the scene lights and the bounding boxes of the scene to compute approximate global illumination. The Illumination pipeline can also handle correct global illumination, but this requires extra data structures such as bounding interval hierarchies, which are needed for the acceleration of rays used for correct visibility determination.

3.1. Asset Definition

Massive scenes contain a very large amount of detail and a very large number of objects, but these objects generally use common assets. The necessary properties for realistic rendering can be divided into positional and structural properties such as transformation trees, animations and morphing and aspect properties such as BxDFs, sub-geometric information and so on. Usually, the positional and structural properties of an object contain its structure, the position and orientation of an object, either in absolute or in relative coordinates, the skeletal animation properties, the morphing animation and the trajectory properties. The structure of the objects is either defined through triangles or through voxels. These properties also define how light is scattered by the object, how the object moves and whether the form of the object is explicitly or implicitly defined. Explicit forms are defined as meshes or as voxel hierarchies, while implicit forms are obtained indirectly from voxel hierarchies or analytic definitions.

The aspect properties define whether and how the object emits light, the type of light absorption and transmission of light, usually quantified in as a BRDF, BSDF, BTDF, BSSRDF or a combination of the above. This information is usually encoded in maps, which are linked to objects through texturing. Common maps used in rendering are color maps, which describe the diffuse response on the surface of objects, emissive maps, used for the representation of light emission, ambient occlusion maps, used to represent sub-geometric visibility, specular power, specular color maps and gloss maps, used to measure the response to specular light interaction, thickness maps and alpha occlusion, used to measure transmission through object, derivative maps, normal maps and displacement maps, used to cheaply represent sub-geometric detail.

In order to construct a state of the art rendering pipeline all the assets have to be easily accessible and combinable by the GPU. While this is straightforward for aspect properties, usually bundled as materials, it is not as easy for structural and positional properties. The positional and structural properties are traditionally streamed from the CPU, but, for extremely large scenes with a very large number of objects, this introduces a data transfer bottleneck, especially when the fast combination of such properties is wanted.

This fast combination necessity is addressed in this thesis. All scene objects are defined as a combination of base, unique, objects, and are identified through **instances** particularized through spatial and aspectual **modifiers**. The objects are defined geometrically through triangles, as voxels are more suited for the approximation of structures than for their correct and detailed rendering, due to the very large implied memory requirements. Because of this instance-modifier mechanism, many other geometry influencing aspects of rendering can easily be integrated, such as morphing and skeletal animation. A minor novel aspect of scene representation is introduced in this thesis, GPU object **trajectories**, which are used to completely represent the dynamics of a scene on the GPU. A trajectory is defined through trajectory points, which are stored as object modifiers, and expressed as a set of values describing next position, next orientation, duration and next trajectory point, which can all be stored in 8 bytes. Because the majority of modifiers can be expressed within the same space, this enables a renderer to use a pool allocator over the modifiers, making their streaming and management easier. The same pool allocator strategy can also be applied over the virtual meshes and virtual textures, further simplifying streaming.

Geometry rendering can be performed without CPU control through indirect rendering. The algorithms presented in this thesis use a scheme based on a hierarchy of level of details, backed by a hierarchy based impostor tree.

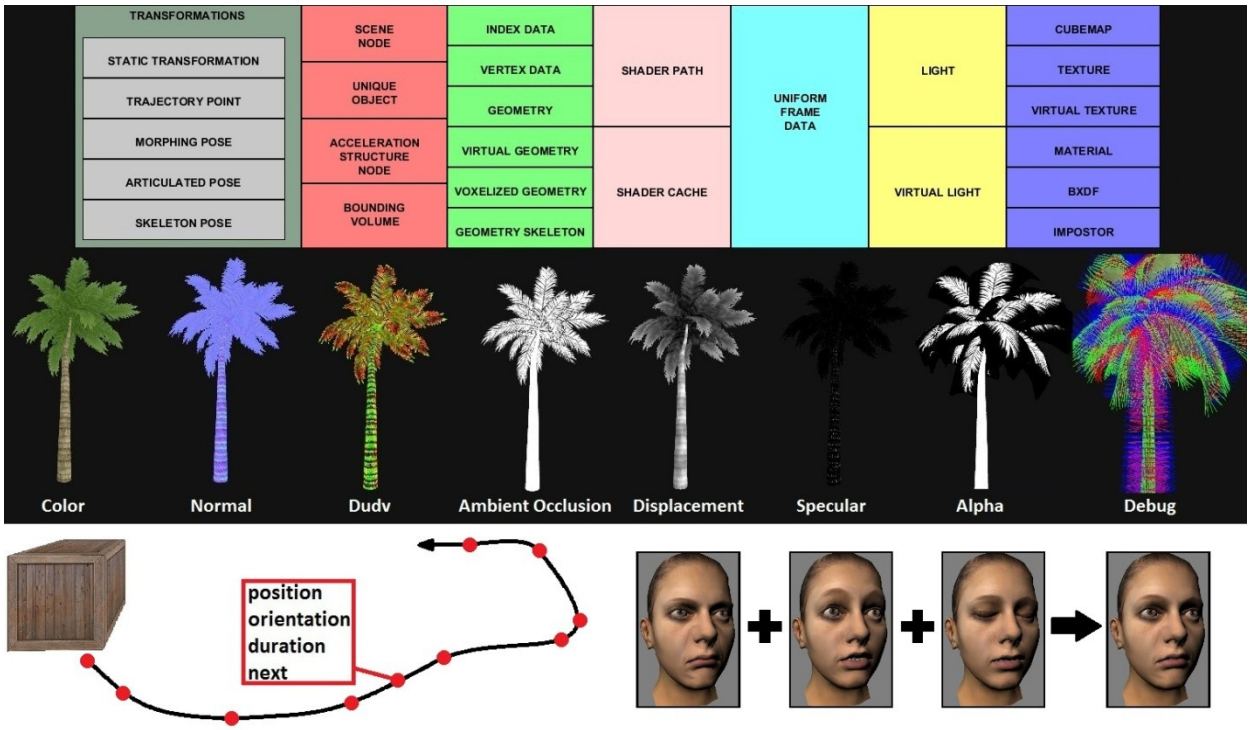

Figure 24 Rendering assets**.** This image depicts various rendering assets that are used in the rendering process. The upper side of the image contains different types of modifiers and raw assets which can be combined to completely describe a rendering object. The central side of the image depicts various aspect modifiers as maps. The lower side of the image presents two utilizations of modifiers: trajectories and morphing.

Because of the large amount of renderable data, hierarchic acceleration structures are necessary. Bounding Volume hierarchies are most suitable for real-time rendering since they are among the easiest to balance, without duplicating geometry. While low-level sub-trees can be computed on demand, large sub-trees require pre-processing. Because of this, rendering applications maintain more than one scene tree, generally one for static objects, one for lights and one for dynamic objects.

3.2. Streaming

In order to perform all the required rendering operations, all the necessary modifiers need be streamed. All map-based information can be easily controlled through a memory paging system like virtual texturing. This includes texture maps, cubemaps and impostors, which are all cut into template sizes and stored and queried through the virtual texturing mechanism. Geometric information can use the geometric correspondent to virtual texturing, called virtual meshes, where meshes and their level of details are cut into template sizes and drawn indirectly.

Streaming is a highly expensive process, since hard disk reads are very expensive operations. This thesis uses zip and LZ4 [**Col15**] in order to compress the assets offline and then be able to quickly in-place decompress them in real-time. Texture compression algorithms such as block compression methods [**Iou99**] are employed by consumer hardware, and they reduce bandwidth without perceptible artifacts.

3.2.1. Virtual Data

Virtual data is based on the principles of paging, which is a memory management system that breaks down data into blocks of the same size, called pages, with the size of a page usually being a power of 2. These pages are then linked to page frames, which are directly mapped to physical memory. Virtual meshes and virtual texturing are streaming mechanisms which break meshes and textures into page-size elements. The page-sized elements are then streamed and queried using a simple paging system, such as the one depicted in Figure 25.

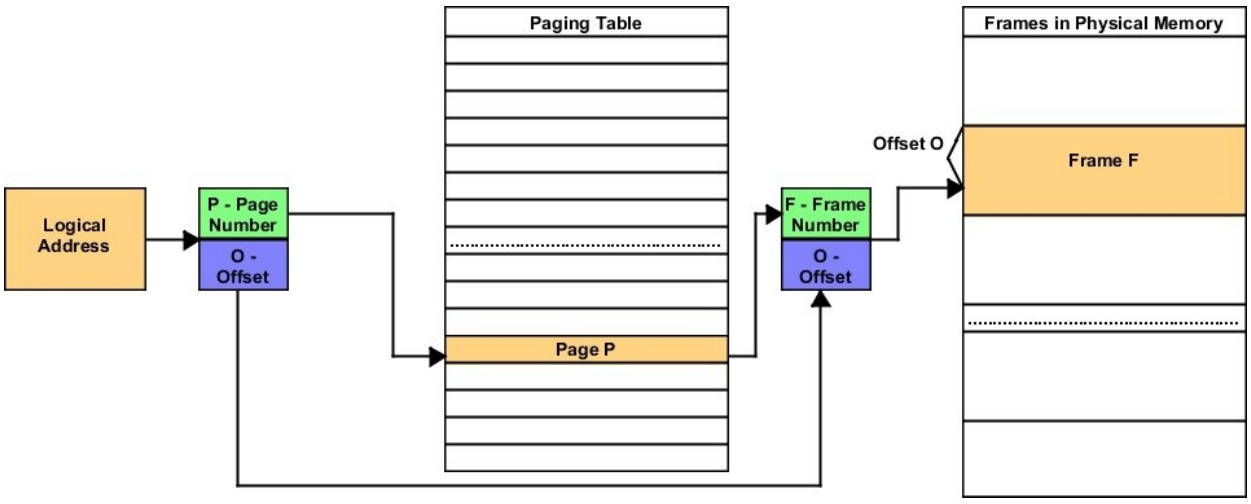

Figure 25 Paging System**.** The logical address contains a page number and a page offset. The page number is queried in the Paging Table, which maps the page number to a frame mapped to real physical memory. The original offset is then used together with the frame number to exactly determine the physical address at which the data is stored.

Virtual data applications use page replacement algorithms to maintain the most useful pages. There are different strategies such as demand paging, in which a page is loaded exactly after request, anticipatory paging, in which the cache preloads pages based on certain metrics, free page queue, in which a list of all the free cache pages is held and used when a cache fault is encountered, page stealing, in which pages that haven't been recently used are added to a free page queue, and pre-cleaning, which guarantees cache data coherency.

Virtual texturing applies the virtual data principle to textures and their mipmap levels. Texturing is not a straightforward process, because it involves filtering operations. In the case of trilinear anisotropic filtering, the consumer hardware texturing method selects the two most relevant mipmaps, which are filtered with bilinear anisotropic samples. The obtained results are then filtered linearly, through interpolation based on the distance between mipmaps. Therefore virtual texturing has to account for this peculiar sampling pattern.

An abstract virtual texturing system is presented in Figure 26.

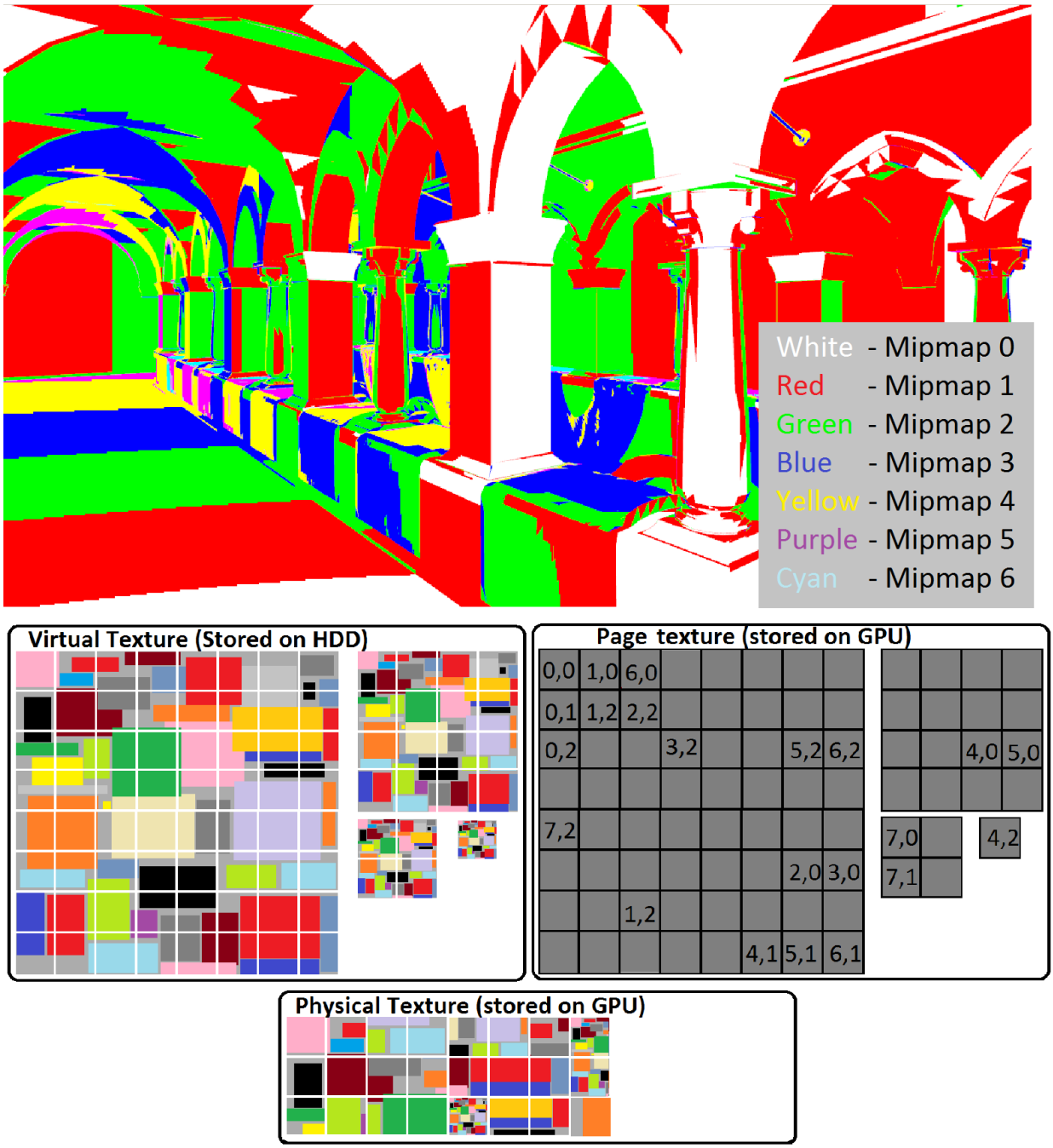

Figure 26 Virtual texturing**.** Virtual texturing is a page system based mechanism which enables the use of extremely large textures. In the top part of the image the colors encode the mipmap levels used from the rendered textures. It is obvious that loading all the assets at their full resolution would only waste GPU bandwidth. The same rendering results can be obtained by just loading the required mipmap levels. The same principle applies to all textures and pieces of textures. Virtual texturing is explained in the lower part of the image. Virtual texturing partitions the textures and their mipmaps into small memory pages, which are streamed on the GPU in a texture named physical texture. A virtual texture access is redirected through a page texture, which contains the mapping from the memory page to the physical texture, where the texture access is finally solved.

Virtual texturing applications usually border the pages used by their paging systems, in order to prevent artifacts generated by wrong sample usage. The border size can be relaxed by a relatively recent consumer hardware innovation, hardware virtual texturing, which simplifies the

management and addressing of virtual texture data. There are many types of variations of virtual texturing, which typically change the paging system, for example with a particular streaming order [**Tai09**], with a different address translation mechanism [**Gar08**], or with lightmaps baking [**Mit08**].

Virtual systems are not sufficient for the large amounts of texture data used in the real-time rendering of massive scenes. Block compression is usually utilized together with virtual texturing, to minimize the bandwidth of the rendering application.

Virtual meshes are the application of the virtual data principle to geometric meshes. Each mesh is partitioned into sub-meshes, whose number of vertices can't surpass a certain threshold. This process is applied to all meshes and their level of details. Depending on the rendering configuration the virtual meshes mechanism streams the pages of the required level of details and rendering is performed with the virtual meshes.

There are many benefits to virtual data in rendering: uncomplicated streaming and data management, low memory and bandwidth consumption, predictable rendering performance and frame rate stability.

3.2.2. Indirect Rendering

Indirect rendering is the rendering process in which the assets are not stored in a directly renderable state, and thus they must suffer a transformation into a renderable format. Indirect rendering is to not be confused with indirect drawing, sometimes also called indirect rendering, which is the process where the rendering hardware generates future rendering commands in a command buffer, which is then executed, without CPU interference.

In general, indirect rendering is used in real-time rendering for point clouds or voxel representations, which are at first reconstructed into triangle meshes and then rendered normally. This process is named surface reconstruction, and it is the inverse operation of pointification, for point clouds, and voxelization, for volume representations. The **surface reconstruction** problem has been thoroughly studied, with four general approaches: explicit reconstruction, local implicit reconstruction, global implicit reconstruction and isosurface extraction and regularization. Explicit reconstruction is based on adaptive triangulations of set of points, and it has strong guarantees but is sensitive to noise, often needing external pre-processing or human control. Local implicit reconstruction approximates the surface by using a low-frequency approximation functions locally, making this method resilient to noise. Global implicit reconstruction is a class of methods that first tries to match the entire sample set with a high level low frequency function. Isosurface extraction and regularization extracts the tridimensional contour from a dataset by locally sampling and reconstructing the surface. A comprehensive state of the art is provided in [**Cuc09**].

Isosurface extraction is particularly important in rendering and in scientific visualization because it is easily parallelizable and it is very robust to noise when provided with a large number of samples. The greatest weakness of isosurface extraction algorithms are the large memory requirements. This situation is common with very large and precise datasets, as are those used in medical visualization, such as the one presented in Figure 27.

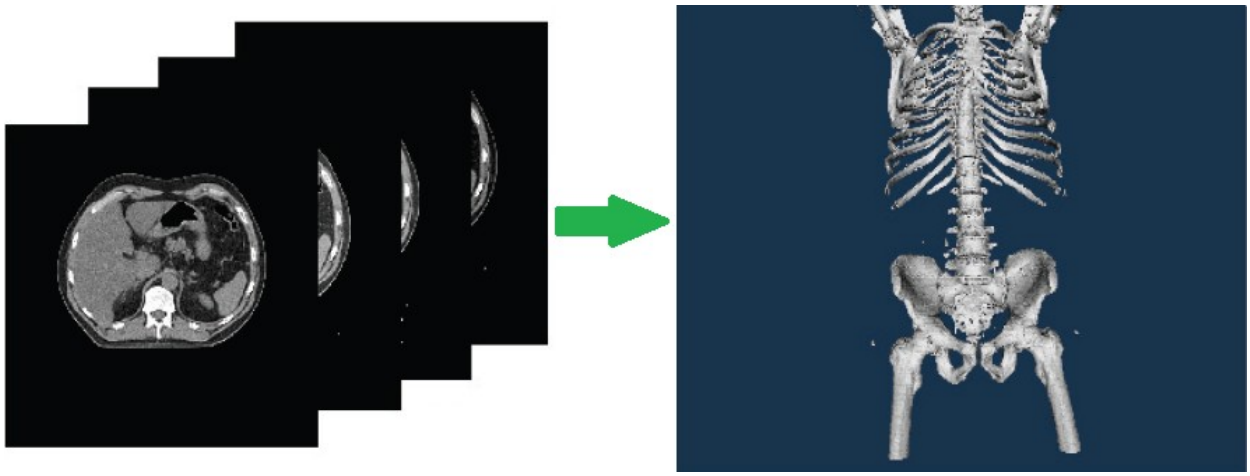

Figure 27 Surface Reconstruction**.** This image depicts the input and the output of a surface reconstruction algorithm. In the left side of the image, the input is represented by a large number of slices, which in this case describe the scanned body of patient. The output of the reconstruction algorithm is the rendered surface on the right.

A growing need of medical application is responsivity, with new applications putting more value into interactivity. Interactive dataset tridimensional reconstruction has many uses in medicine and the CPU marching cubes [**Lor87**], a popular solution to surface reconstruction, has been adapted for GPU usage.

One problem of this adaptation is that it is limited to the total amount of GPU memory available to the graphics card. It is common for datasets in excess of 1024x1024x1024 to be needed in interactive reconstruction, which represents 52 gigabytes of effective GPU data, without counting the space needed to store the reconstructed surface. This amount of memory makes GPGPU Marching cubes impractical on consumer hardware.

A novel **adaptation of the GPGPU marching cubes** algorithm is presented, which minimizes memory consumption, making it possible to reconstruct very large datasets in real-time. The presented algorithm is based on the "Real time reconstruction of volumes from very large datasets using CUDA" article [**Pet11**]. The proposed algorithm divides the data volume into maximum capacity sub-volumes, also called chunks, which are serially reconstructed on the GPU. This method requires less memory than the standards GPGPU marching cubes and can also cull entire chunks from sparse volumes, making the global reconstruction cost cheaper. Another useful property is that only the relevant parts of the dataset have to be reconstructed, therefore when only a part of the volume is changing, the cost of reconstruction is adaptive instead of constant.

The size of the chunks can be easily modified, but care must be taken to avoid inefficient chunk overlapping. In the proposed scheme, the overlap is that of one 1 pixel at the boundaries of the partitioning axis. This is done to reconstruct the surface in a watertight manner. The presented algorithm has two stages: the preprocessing stage and the reconstruction stage. The preprocessing stage analyzes the volume, partitions it into chunks and decides whether a chunk contains useful information. The reconstruction stage of the algorithm reconstructs the chunk and synchronizes the outputted sub-mesh into the global mesh.

The algorithm is described in the following pseudocode:

```
(ONCE) PREPROCESSING STEP (dataset)
edgetable texture, triangletable texture ← create textures to store the Marching Cubes tables on the GPU
bestoccupancy← 0
chunksize← 0
FOR size in 4-max chunksize range
        occupancy ← compute GPU occupancy based on GPU RAM and dataset size
        IF occupancy > bestoccupancy
                chunksize ← size
                bestoccupancy← occupancy
FOR chunk in dataset
        culled ← true
        FOR voxel in chunk
                IF voxel is set
                        culled← true
                        BREAK
        IF culled
                mark chunk as unimportant
RECONSTRUCTION STEP (dataset)
geometrylist ← Ø
FOR chunk in dataset
        IF chunk not unimportant
                reconstruct ← false
                FOR voxel in chunk
                        IF neighborhood of voxel close to isovalue
                                reconstruct ← true
                IF reconstruct
                        geometry ← run GPGPU Marching cubes on chunk
                        geometrylist ←geometrylist ∪ geometry
FOR geometry in geometrylist
        render geometry
```

The chunked reconstruction method is also depicted in Figure 28.

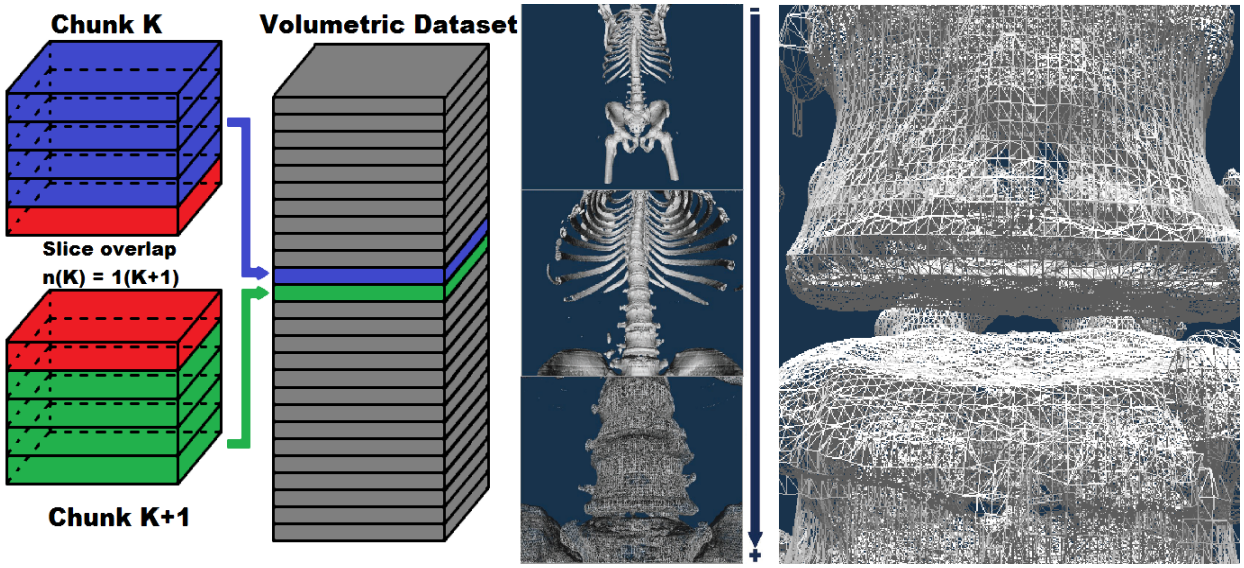

Figure 28 Chunked Marching Cubes**.** Very large volumetric datasets can be partitioned into chunks, which can be then serially reconstructed on the GPU. In order to guarantee a watertight extracted surface, the first and last slices of a chunk overlap with neighbor slices. The exact reconstruction results of a very detailed volumetric dataset are provided on the right part of the figure.

The implementation of the presented algorithm was tested on a video card with 1.5 GB of GPU RAM. Various sizes have been tested for the chunks, in order to compare memory usage, and thus determine the optimal chunk size. The memory occupancy obtained with different chunk sizes is presented in Table 3. Because the occupancy can be queried in real-time, it can be used as a metric to quickly determine the best chunk size for the reconstructed volume.

Chunk Size (number of slices)	RAM (MB)	Memory Usage
32	1495	99.6
16	1440	96
8	1432	95.4
4	>1500	>100

Table 3 Chunked Marching Cubes Memory Usage**.** This table presents the memory usage obtained during dataset reconstruction with different sizes, using the Chunked Marching Cubes algorithm. The table shows that there is an optimal chunk size and that chunk overlapping can become counterproductive for very small chunks.

This algorithm has been used in a medical application, 3D for Medicine, as part of the European Project SABIMAS, PNCDII-Joint Applied Research Projects, 2008-2011), http://se.cs.pub.ro/SABIMAS/ [**SAB15**]. The program is designed to help doctors personalize implants for hip arthroplasty, based on tomography results stored in Digital Imaging and Communications in Medicine (DICOM) [**NEM15**] format.

A screenshot from this software is presented in Figure 29. This method has also been used in non-photorealistic rendering pipelines, such as the one described in "GPGPU Based Non-photorealistic Rendering of Volume Data" [**Mor13**].

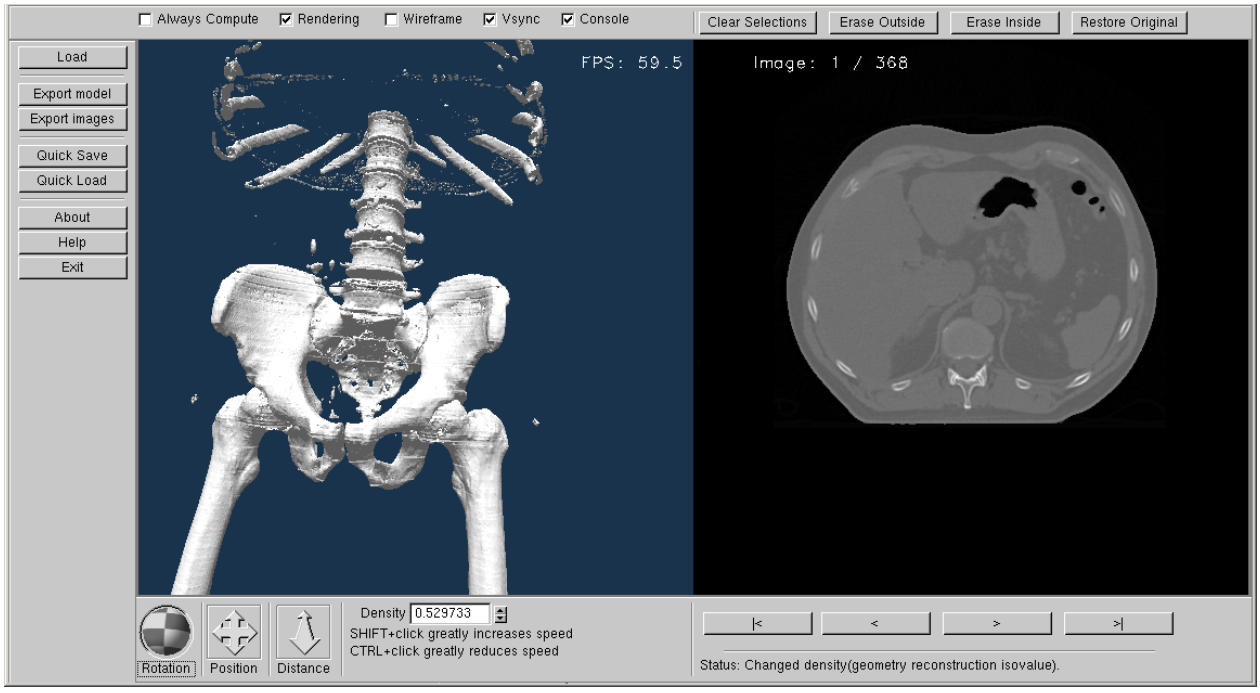

Figure 29 3D for Medicine**.** This screenshot is taken from the 3D for Medicine software, which can inspect and reconstruct very large DICOM datasets in real-time. The software can alter the dataset in real time, to eliminate scanning artifacts. The reconstruction algorithm is based on chunked marching cubes.

3.2.3. Hierarchical Impostors

Impostors represent a partial solution to two problems of real-time rendering: scalability and anti-aliasing. Rendering with impostors drastically lowers the geometry processing complexity of a scene, and thus significantly decreases rendering time. Impostors are over-sampled representations of objects stored as images, and therefore do not suffer from geometric reconstruction artifact, but are filtered like textures. Furthermore, because impostors are stored as textures they are very easy to integrate into a streaming system like virtual texturing.

Although the state of the art contains a large number of impostor techniques, such as billboards [**Ger88**], billboard clouds [**Dec03**], omni directional billboards [**And07**]**,** true impostors [**Ris06**]**,** 3-view impostors [**Har10**] or volumetric impostors [**Dec09**]**,** none of these methods extends beyond a single object. The closest methods to the proposed hierarchical impostors method are [**Ume05**] and [**OHa02**], in which several objects are bundled in the same impostor, but this is done in the context of homogenous super-objects such as clouds.

The newly presented method, **hierarchical imposters**, differs from the state of the art through the integration within a **virtual texturing** streaming mechanism, the ability to be rendered with parallax effects and through the fact that it explicitly applies to any type of object group. Each hierarchical impostor represents a group of objects and contains depth, normals and color information, which can be used for high quality distant object rendering. When the view distance or viewing angles change too much from the currently stored impostor, the impostor contents are updated with the new view and distance. Expected views and distances can also be used.

This algorithm is especially designed for very large scenes, where not all the scene assets are stored in GPU memory. The entire scene is stored offline in an acceleration structure. Depending on the camera position the scene nodes mapped high in the acceleration structure which are further than a certain threshold distance will have their impostors precomputed and loaded. Then impostors will be streamed in and out for all the other visible objects, which are greater than a certain threshold. When the size of a scene node is smaller than a certain screen space number of pixels threshold all the children of the scene node will not be rendered. Instead, the impostor of the scene node failing the threshold test will be rendered.

Impostors are updated depending on angle of view and distance of view. Because of this, a secondary acceleration structure can be used for dynamic objects, which can be update on a per frame basis. The complexity of this operation can be amortized by making the update over a number of frames.

Because hierarchical impostors group many distant scene nodes into a single renderable entity, which can remain valid for a large number of frames, they greatly decrease the rendering complexity of the scene. Because the only geometry rendered for an impostor is a simple billboard the geometry cost is dramatically decreased, while the shading cost is maintained close to constant per frame even without deferred algorithms. Perhaps the most important feature of impostors is that they minimize geometric aliasing, transforming geometric aliasing into texture aliasing, which is far easier to handle because of the automated texture filtering available on all consumer hardware. The loss of detail can be almost completely prevented with parallax sub-geometric rendering algorithms. A hierarchic impostor is presented in Figure 30.

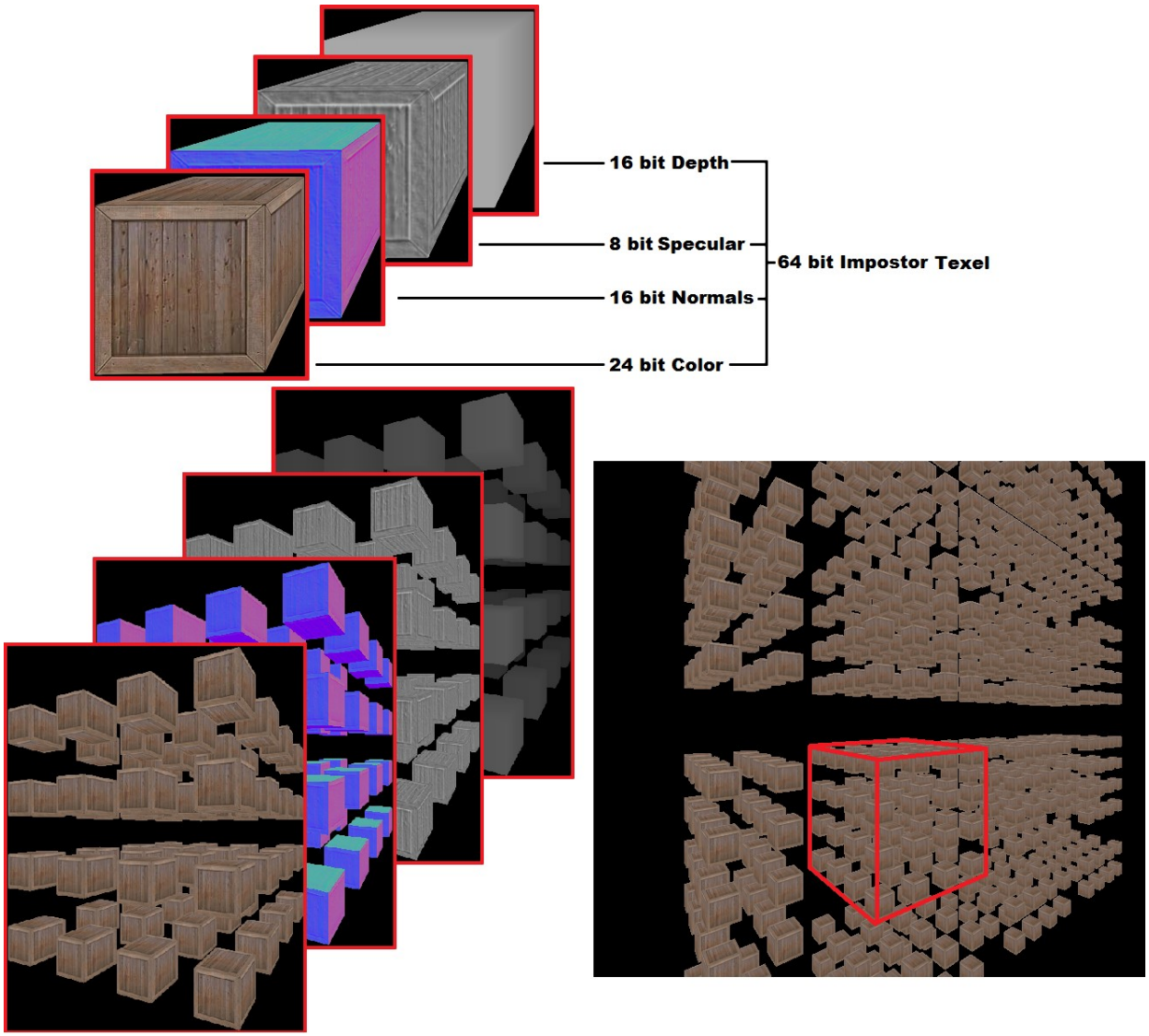

Figure 30 Hierarchical Impostors**.** An object impostor is presented in the superior part of the image. It contains data about color, normals, specular response and sub-geometric depth displacement data. As presented in the lower part of the image, the impostor concept can be applied to large groups of objects. The maps that contain all the impostor data can be managed and streamed through a virtual texturing system.

The presented impostor can also be augmented with emission and transparency properties, at the cost of more memory. Before the effective rendering the process of impostor determination can be integrated with culling and other scene management operations. The proposed method renders the hierarchical impostors like true impostors. Each impostor is sent to the graphics pipeline as a point which is expanded to a textured screen space aligned quad, a billboard. Then, the billboard is adjusted to the correct depth. For each fragment generated through billboard rasterization is intersected with a linear search, in a process often times called marching, in order to quickly determine the approximate location of the camera surface intersection. A secondary binary/secant search is used to accurately determine the camera surface intersection point. The map information found by using the texture coordinates resulted from the marching process is used for shading the pixel.

The pseudocode for hierarchical impostors is the following:

```
CACHED (node, camera, scene)
distance, angle, size ← compute distance and size based on node and camera
FOR element in impostorcache
        elementdistance , elementangle, elementsize ← load data for element
        IF elementdistance ≈ distance and elementangle ≈ angle and elementsize ≈ size
                RETURN element
RETURN null

CREATE (node, camera, scene)
cached ←  CACHED(node, camera, scene)
IF cached not null
        RETURN cached
ELSE
        IF impostorcache is out of storage space
                RETURN null
        impostorcamera, distance, angle, size ← compute distance and size based on node and camera
        poweroftwo← log size
        create impostor with resolution 2 poweroftwo
        FOR scenenode in scene
                IF scenenode visible by impostorcamera
                        nodedistance, nodeangle, nodesize ← compute distance and size based on scenenode
                                                            and impostorcamera
                        scenenodeimpostor← CACHED(scenenode, impostorcamera, scene)
                        IF scenenodeimpostor not null
                                impostorview ←render scenenodeimpostor
                                impostorview←depth, transparency, emission, color, displacement, specular
                        ELSE
                                render node into impostorview
                        impostorview ←depth, transparency, emission, color, displacement, specular
impostordepth, impostorangle ← impostor depth and angle of view, as viewed from camera
impostorcache ← impostorcache ∪ impostor

PRE-RENDERING (scene, camera)
renderqueue ←  traverse scene from camera
impostorrenderqueue← ∅
FOR node in renderqueue
        nodedistance, nodeangle ←  compute distance and angle of node  from camera
        IF nodedistance > threshold AND nodeangle < threshold
                cached ← CACHED(node, camera, scene)
                IF cached not null
                        renderqueue ← renderqueue∪ cached
                ELSE
                        newimpostor ← CREATE(node, camera, scene)
                        IF newimpostor not null
                                renderqueue ← renderqueue∪ newimpostor
LINEARSEARCH (displacementmap, entrypoint, entryray)
depth, preventrypoint ← depth of entrypoint
mapdepth ← read displacementmap at entrypoint
WHILE  depth > 0
        preventrypoint← entrypoint
        depth, entrypoint← entrypoint + entryray
        mapdepth ← read displacementmap at entrypoint
        IF mapdepth > depth
                RETURN entrypoint, preventrypoint
RETURN null, null
```

```
BINARYSEARCH (displacementmap, inf, sup, entryray)
error ← threshold
WHILE  error < threshold
        midpoint← (inf + sup)/2
        middepth ← read displacementmap at midpoint
        midpoint← adjust for middepth
        best ← closest to midpoint between inf and sup
        IF inf = best
                sup = (inf+sup)/2
        ELSE
                inf = (inf+sup)/2
        error ← distance to midpoint from best
RETURN best

RENDERING (renderqueue, camera)
FOR impostor in renderqueue
        send a point to the vertex shader
        billboard ← draw and expand point to a billboard in the geometry shader
        rasterizedfragments ← rasterize the billboard into fragments
        FOR fragment in rasterizedfragments
                displacementmap ← get displacement map of impostor
                depth ← get impostor depth
                cameraray ← ray from camera to pixel
                entrypoint ← coordinates of displacement map entry point, adjust to depth
                entryray ← transform cameraray to local depth adjusted space,
                inf, sup ← LINEARSEARCH displacementmap, entrypoint
                IF nextpoint, prevpoint not null
                        intersection ← BINARYSEARCH displacementmap, inf, sup, entryray
                        fragmentcolor ← shade intersection
                        fragmentcolor ← background color
        OUTPUT fragmentcolor
```

The process of rendering with hierarchical impostors is displayed in Figure 31.

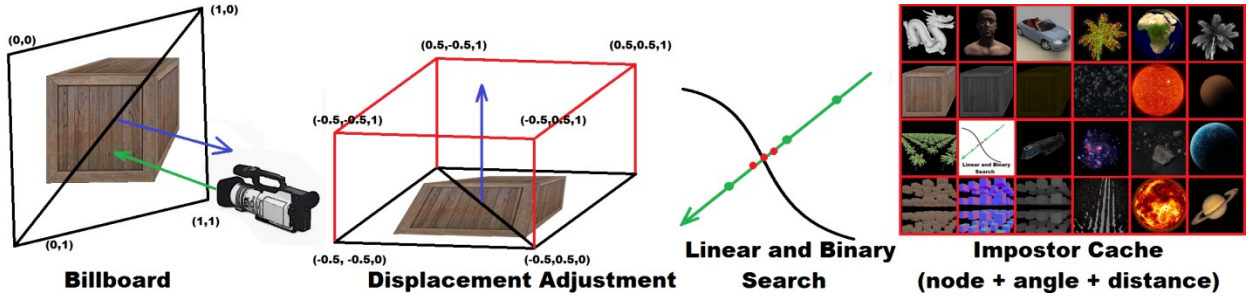

Figure 31 Rendering Hierarchical Impostors**.** The first step is to create a billboard, mapped with the impostor textures. In the second step, the billboard is displaced with the creation depth of the impostor. In the third step the surface-camera intersection point is accurately determined through linear search followed by binary search. All the impostors are saved in the impostor cache, uniquely identified by the tuple of scene node, view angle and view distance.

The proposed hierarchical impostor technique differs from the state of the art through its virtual texture integration, and through its explicit construction over many scene nodes. Because of the hierarchical nature of the proposed method it decreases the complexity of the rendering process from $O(objects)$ to $O(\log(objects))$.

3.3. Task Generation for Rasterization

Task scheduling on the GPU has been an important area of research in the last years [**Gro13**] [**Gup12**] [**Jog13**] [**Tze12**] [**Mem12**] because it maximizes GPU efficiency by minimizing CPU control. The additional GPU tasks, also called dynamic tasks, and the process through which they are created is called dynamic parallelism. The advantage of this technique is that it brings a significant simplification in solving massive parallel problems by offering superior performance and software design opportunities. Driver developers have also been offering this functionality for GPGPU programming languages like CUDA 5 [**NVI15**], OpenCL 2.0 [**KHR14**] or Mantle [**AMD15**].

These task generation and scheduling efforts are from a GPGPU perspective since the standard rasterization graphics pipeline is expected to schedule and dispatch the threads on which the specialized programs named shaders run. The task creators vary in complexity from small task generators, which only generate GPU work to full blown GPU task schedulers, which manage the parallelization of recursively generated tasks.

A special GPU task scheduler governs these shader invocations, but it is impossible to be directly controlled, therefore it is impossible to add general tasks to this scheduler manually. Current directly controllable GPGPU task scheduling solutions such as [**Gup12**] [**Jog13**] [**Tze12**] [**Mem12**] do not concern themselves with offering control over the GPU rasterization task scheduler, thereby making GPU rasterization programs unable to generate additional task without CPU control. There are many situations in which rasterization based GPU programs need to generate additional tasks, for example in the case of hierarchical culling, extreme tessellation, angle of view dependent rendering or for the evaluation of complex materials. When these programs generate computationally heavy GPU threads, the created work can't be parallelized without CPU control. In this section a novel **task generator that works within the hardware rasterization scheduler** is presented, which is based on the "A GPU task generator for rendering" article [**Pet14**].

The presented method is based on the idea of using the **geometry amplification capabilities of the hardware rasterization pipeline to generate GPGPU-like tasks**. This is achieved through the tessellation control, tessellation evaluation, geometry and fragment shaders, by using SM5 instructions.

The original primitives are sent to a modified rendering pipeline. The tessellation and geometry stages are either added, if they were missing, or modified to fit the task generator needs. In the tessellation control shader the computational effort of the primitive is approximated and it is divided into task groups.

The task groups are evaluated in the tessellation evaluation shader, where other finer grained task groups are created, through the geometry shader invocation mechanism.

In the geometry shader, each fine grained task group is analyzed and a large number of tasks are generated for each group.

In the fragment shader these tasks are executed like GPGPU-like threads.

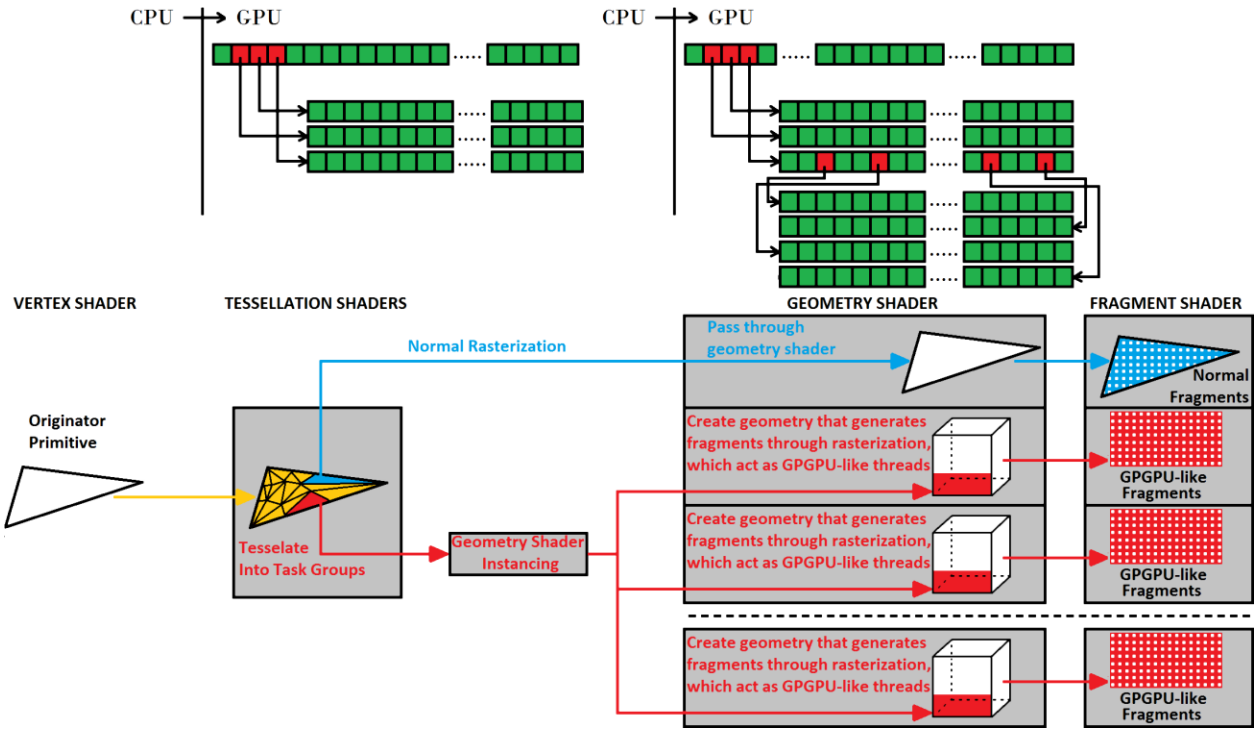

Figure 32 Task Generation on the GPU**.** The upper part of the image presents the difference between simple task generation and recursive task generation. Simple task generation can generate tasks, even recursively, but the recursion level is hard capped. Recursive task generation can generate tasks with no limitation. In the second part of the image a novel GPU task generator is presented. It can create additional tasks for originator primitives at several points in the hardware rasterization pipeline, using the hardware geometry amplification capabilities as task generators. Despite the fact that the presented pipeline can generate tasks at several points, it is still not a recursive generator, as the number of generation points is limited.

A small area of GPU memory that is used for keeping basic task information, such as thread indexes or basic numeric result codes is utilized as support for thread communication. This memory area can be written to and read from any stage of the rasterization pipeline by using SM5. The originator primitive first enters the tessellation control shader.

This originator primitive is first evaluated with an algorithm dependent metric, in order to approximate its computational effort. If the metric suggests parallelization is necessary, the original primitive set $count_t + 1$ the number of primitives that will be created by the tessellator. These $count_t + 1$ primitives contain the original primitive and $count_t$ task groups.

The original primitive is sent to normal rasterization and follows a normal rendering pipeline with a pass-through geometry shader, which generates normal fragments, which are shaded as the primitive was supposed to be shaded. The other $count_t$ primitives are instancing with the geometry shader instancing mechanism, reaching a total number of $count_t * instancing$ number of instanced primitives in the geometry shader. Each instanced geometry shader primitive then uses another algorithm dependent parallelization metric, determining $count_{g,i}$ for each ith invocation of the geometry shader. $count_{g,i}$ represents the number of working threads that will be spawned by each geometry shader invocation. These threads are created by drawing a billboard in normalized device coordinates with the following width and height.

$$width = \frac{2\ x\ count_{g,i}}{screenwidth\ x\ perthread\ x\ screenheight}$$

$$height = \frac{2}{screenwidth} - 1$$

The billboard is then rasterized and spawns $count_{g,i}$ fragments, which will be computed like GPGPU-like threads with the help of SM 5 instructions. The total number of spawned threads is $t_t x\ instancing\ x\ \sum_i count_{g,i}$. The GPGPU like fragments then take a completely different code path than the rendering fragments. Each GPGPU-like thread can read one or more tasks from the memory area in which they were written by the geometry shader. Thus, these special invocations of the fragment shader solve tasks, write the results back into GPU memory and do not write to the framebuffer. Thus, uneven rendering efforts are effectively parallelized in a manner that is both easily implementable and extremely efficient while working within the rendering pipeline. The pseudocode for the presented method is the following:

```
(ONCE) PREPROCESS
allocate sharedmemory for task communication
TESSELLATION CONTROL SHADER (originator primitives)
FOR primitive in originator primitives
    approxcost ← use algorithm dependent metric to approximate primitive rendering cost
    IF approxcost > threshold
        tessellationfactors, primitives ← set to generate count_t + 1 primitives count_t
        primitives represent task groups
    primitive is sent down the pipeline to normal rasterization
TESSELLATION EVALUATION SHADER (processedprimitives)
FOR primitive in processedprimitives
    IF processedprimitive = original primitive
        send processedprimitive to the normal rasterization path
    ELSE
        create taskgroup
        sharedmemory← write taskgroup
GEOMETRY SHADER (instances, primitives)
FOR primitive in primitives
    IF primitive is originator primitive
        fragments ← render primitive normally
    WHILE instances>0                                          //INSTANCING
        width ← 2 x count_{g,i} / (screenwidth x perthread x screenheight)
        height ← 2 / screenwidth − 1
        taskinfo ← sharedmemory
        generatedtaskgroups ←generate task groups for taskinfo
        sharedmemory ← generatedtaskgroups
        gpgpufragments ← rasterize screen space billboard with width and height
        instances ← instances - 1
FRAGMENT SHADER (fragments)
FOR fragment in fragments
    IF fragment rendered normally
        render normally
    ELSE
        taskinfo ← sharedmemory
        result ← solve task
        sharedmemory←result
```

This method can be extend to a fully extensible task generator by reading back the fragment tasks to the CPU and drawing a primitive linked to the fragment task. The primitives could follow the same task generation pipeline, without the normal rendering path. However, this task generator method was designed for rendering tasks which need to generate task trees with a large number of tasks but with a low height, for which this method is ideal.

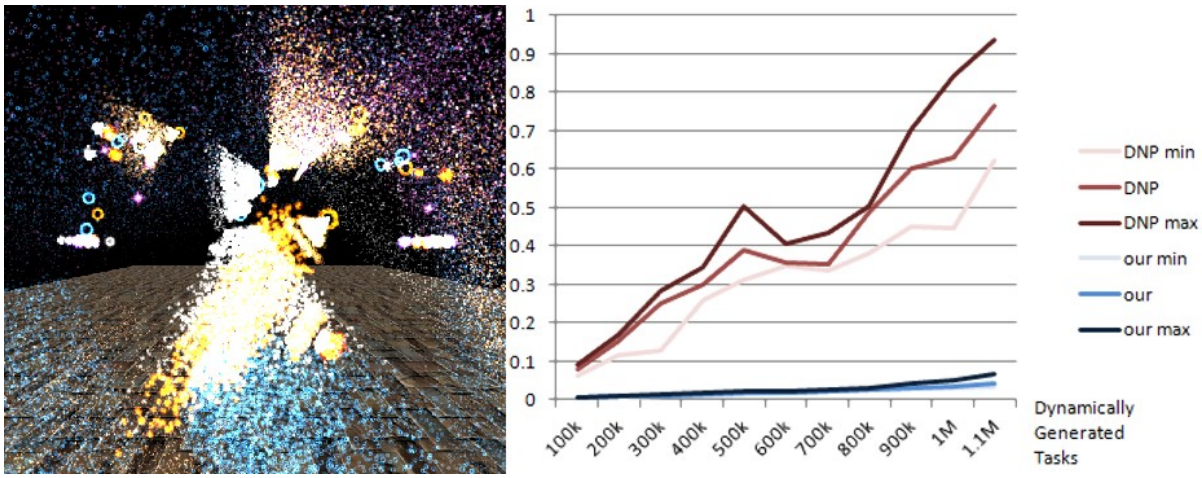

Figure 33 Task Generation Results**.** The left side of the image presents a simple rendering scenario, where many particle systems are drawn, and each particle system can in turn generate other particle systems. With Dynamically Not Parallel (DNP) methods the rendering time rate of growth is much larger than with the presented method. This is caused by the DNP inability to parallelize the computing effort, leading to a small number of computationally heavy threads. The presented method efficiently parallelizes the rendering effort.

The presented method efficiently parallelizes uneven rendering efforts, as presented in Figure 33 and Table 4. This task generator for rasterization rendering is fully compatible with the entire consumer hardware rasterization based rendering pipeline, working through the GPU rasterization scheduler. It is able to generate new tasks efficiently, without requiring preprocessing. It also doesn't monopolize the hardware resources compared to existing GPU task schedulers and generators, which usually control the entire pipeline.

Rendering method/ Tasks	100k	200k	300k	400k	500k	600k	700k	800k	900k	1M	1.1M
DNP min	0.06	0.11	0.12	0.25	0.31	0.34	0.33	0.38	0.45	0.44	0.62
DNP	0.07	0.15	0.25	0.30	0.38	0.35	0.35	0.48	0.60	0.63	0.76
DNP max	0.0	0.16	0.28	0.34	0.50	0.40	0.43	0.50	0.70	0.84	0.93
TaskGen min	0.00	0.00	0.01	0.01	0.01	0.01	0.02	0.02	0.02	0.02	0.03
TaskGen	0.00	0.00	0.01	0.01	0.01	0.01	0.02	0.02	0.03	0.03	0.04
TaskGen max	0.00	0.00	0.01	0.01	0.02	0.02	0.02	0.03	0.04	0.04	0.06

Table 4 Task Generator for Rendering Results**.** This table presents the memory usage obtained during dataset reconstruction with different sizes, using the Chunked Marching Cubes algorithm. The table shows that there is an optimal chunk size and that chunk overlapping can become counterproductive for very small chunks.

The task generator was not designed for full recursion, but it can be configured to be run as a fully recursive task generator. The reason behind this choice is that in rendering, the dynamic tasks are rarely required to create deep tree tasks themselves. Without the obvious exceptions like ray and path tracing, which are usually handled in specialized pipelines, the potential dynamic rendering tasks for which a task generator would be useful in a rasterization context are usually leafs, or close to leafs in the task graph.

3.4. Hierarchical GPU Culling

Massive scenes produce extremely large amounts of processed data. This is especially important from a geometric standpoint in rasterization, since the camera-surface intersections are obtained without order. Because of this, performant culling algorithms are vital to a high quality rendering pipeline. There are various types of culling, such as view frustum culling, which culls objects outside the visual volume, occlusion culling, which culls object objects occluded by other objects inside the visual volume, or distance culling, which culls objects if their projection is smaller than a given threshold. Because of the large number of operations involved in culling, such algorithms almost always use acceleration structures to decrease operation complexity.

State of the art methods [**Bit04**] [**Gut06**] [**Mat08**] [**Mat15**] [**Zha97**] [**Mar11**] [**Déc05**] [**Bar12**] either use geometry impostors or are not hierarchic or are dependent on costly synchronization operations. Because of this there is an opportunity for improvement in this area. Hierarchical GPU Culling is a **hierarchic view frustum culling** method, which runs entirely on the GPU, without CPU interference and without precomputation. Hierarchical GPU Culling is not limited to geometric impostors but can benefit from their presence. The algorithm is based on the previously presented GPU task generation mechanism and introduces multiple frames culling, where objects are culled for many frames.

The most relevant state of the art methods used today in culling are CHC++ [**Mat08**] and Hierarchical Occlusion Maps [**Zha97**]. While CHC++ can be adapted to an integral GPU algorithm with hardware occlusion query buffers or through the use of a stack [**Mat15**], it is still limited by the synchronization time introduced by waiting for rendering batches to finish.

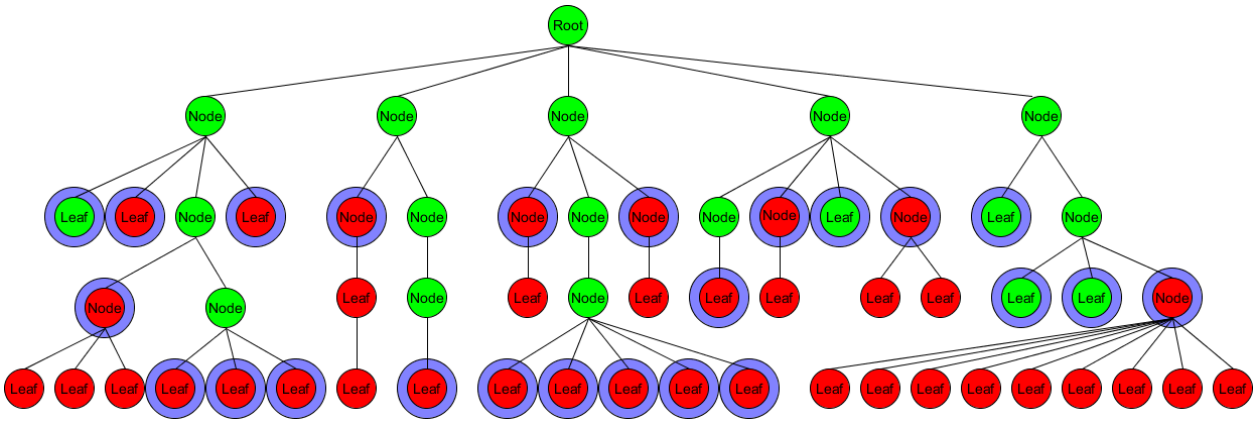

Figure 34 Coherent Hierarchical Culling**.** The algorithm works by rendering all the objects which were visible during the previous frame and then renders all their children recursively, until children are culled. In the image the nodes which were visible during the previous frame are colored in green and the nodes which were not are colored in red. The nodes which are tested for visibility during the current thread are encased in blue disks. The algorithm determines visibility through view frustum culling tests and hardware occlusion queries. CHC has been improved with CHC++ which uses statistics and multi-queries to decrease the number of occlusion queries. The weakness of CHC is that in order to render the children of a node, the node in cause has to be labeled visible, and therefore has to be rendered and occlusion queried. A CPU-GPU synchronization and wait event can appear because of this behavior. This behavior can be attenuated with a GPU implementation of CHC++.

The CHC algorithm processes the scene tree hierarchically, rendering only what needs to be rendered. Because of this the algorithm works correctly without pre-processing. On the other hand the nature of the algorithm makes it to translate poorly to GPUs, since there are many synchronization after wait events, which are not compatible to many core computing. CHC++

has problems especially with complex depth distribution scenes such as forests, where it is hard to define a visibility hierarchy. CHC++ is presented in Figure 34.

CHC++ is prone to frame rate fluctuations produced by an uneven per-frame number of synchronization and wait events and it can sometimes perform considerable extra work by rendering without testing the objects which were visible during the previous frame. Furthermore, CHC++ can't handle alpha accumulation and culls objects only for the current frame. Scenes with dynamic objects are particularly difficult for CHC algorithms because of the large changes between frames.

Hierarchical Occlusion Maps has been adapted into Hierarchical Depth Culling [**Rak15**], which is an integral GPU algorithm. Hierarchical Depth Culling uses the depth buffer as the original occlusion map, creating the occluder hierarchy based on it, and then culling against this hierarchy. Compared to Hierarchical Occlusion Maps, Hierarchical Depth Culling performs the visibility test on all the pixels on which an object is projected by approximating the tested objects with bidimensional bounding boxes which are then tested against a single pixel from one of the mipmaps of the depth buffer. It is an approximate algorithm in the absence of a strong visibility constraint such as geometric impostors. Such impostors are precomputable for a large majority of objects, but they are impossible to apply to high geometric frequency objects such as trees and fences. Another problem of Hierarchical Depth Culling is that it cannot reach maximum efficiency without front to back sorting. Because all the culling is performed relative to the previous frame buffer, Hierarchical Depth Culling is prone to a number of temporal artifacts. The algorithm is presented in Figure 35.

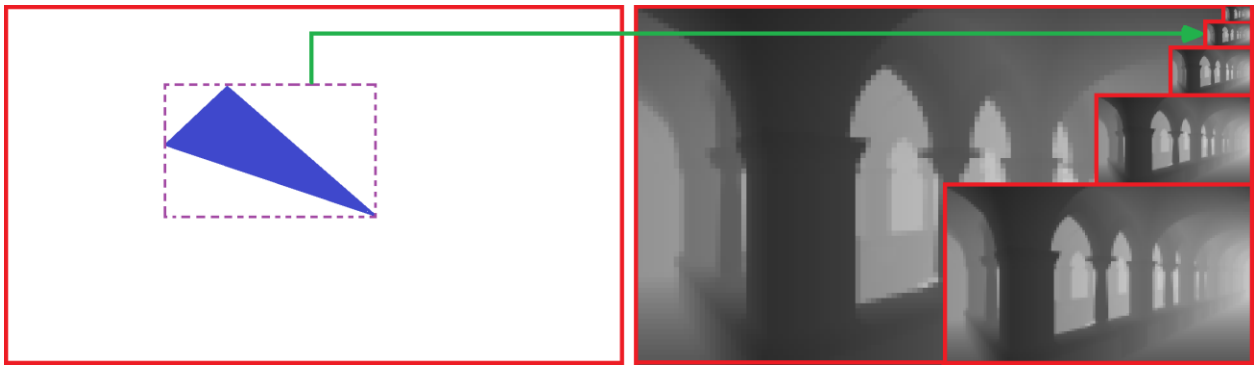

Figure 35 Hierarchical Depth Culling**.** The left side of the image presents a blue object that is currently tested for depth. The right side of the image presents the depth buffer obtained from the previous frame, along with all its mipmaps. Hierarchical Depth Culling tests the visibility of the blue object by extending it to a screen space bidimensional bounding box, and then culling against a single pixel from the best fitting mipmap of the previous frame depth buffer, as shown on the right side of the image. The best fitting mipmap is chosen based on the size of the extended bidimensional bounding box of the tested object.

In this subchapter a new hierarchical culling algorithm is introduced, which handles occlusion through hierarchical depth culling, when impostor geometry is available and view frustum culling through a hierarchical GPU algorithm, based on the task generator for rasterization presented in Chapter 3.3. The presented algorithm is integrated with the hierarchical impostors system presented in Chapter 3.2 and uses a multi-frame culling scheme which can greatly decrease computational costs. The culling method writes the visible objects in a buffer, which is rendered with indirect rendering. This method can also be implemented with dynamic parallelism.

The culling algorithm is based on task generation, generating new tasks for each child of scene nodes which were evaluated as visible. If the height of the resulted tree is higher than the number of task generation points supported, than the algorithm is run in several runs or the superior part of the tree, which represents a very small percent of the culling effort, is computed on the CPU. The latter case is desirable because it guarantees the creation of sufficient work for the GPU. The algorithm can be implemented in GPGPU fashion with the help of **dynamic parallelism**, which is easier to implement, therefore the **rasterization** variant of the algorithm is presented. The presented culling method requires no preprocessing and does not maintain data structures which require synchronization, such as the priority queue from the coherent hierarchical culling algorithms. The interaction with the CPU is minimal, the CPU is only used to generate a sufficient amount of work for the GPU, after which the algorithm is CPU interference free. The algorithm is depicted in Figure 36.

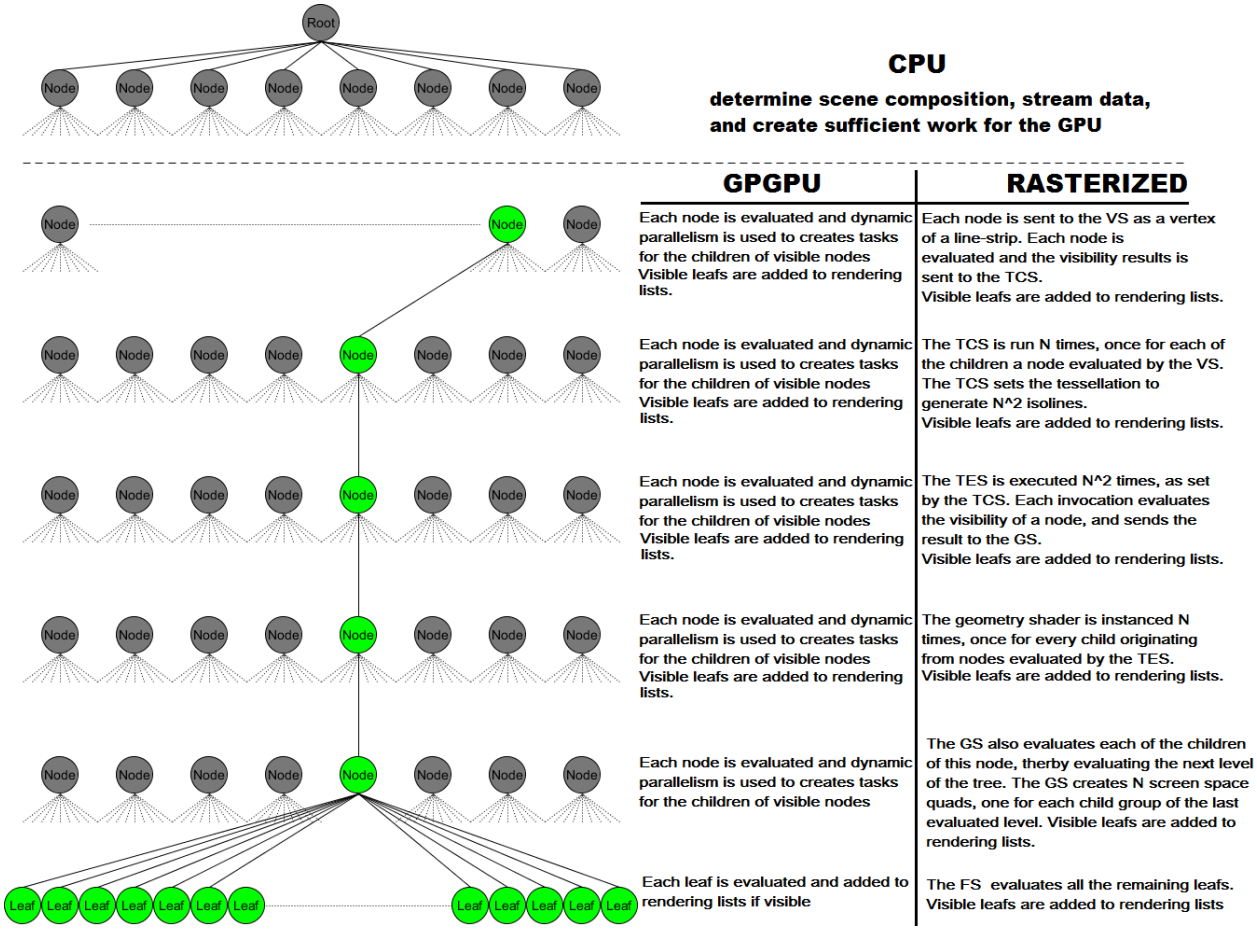

Figure 36 Hierarchical GPU Culling**.** The upper part of the image presents that some CPU work is required to create sufficient work to elicit GPU culling. The left side of the image displays a path taken through a scene tree. The right side of the image presents both GPGPU and Rasterization paths that can be used by the presented algorithm. The rasterization path uses principles from the task generator presented in Chapter 3.3.

The preprocessing required by the presented technique is minimal, as only a scene tree which guarantees a small number of children is required. The algorithm has been tested with a scene tree with structure of maximum $N = 8$ children per scene node.

The culling algorithm starts by allocating a large index array, where the indices of the visible scene nodes will be stored. The method then starts to generate a sufficient amount of

work for the GPU, by computing view frustum culling and occlusion culling, depending on the usage of Hierarchical Depth Culling.

After sufficient GPU work has been generated, each scene node that is to be tested for visibility is sent to the graphics pipeline as a vertex of a line strip. The Vertex Shader (VS) evaluates each scene node sent and outputs the visibility result to the tessellation control shader. If the vertex shader processes a scene node which is both visible and a leaf, the scene node id is added to visible nodes list. The Tessellation Control Shader (TCS) evaluates all N children of the scene node which was sent to it by the VS, saving the visible leafs to the visible nodes list. The TCS sets the Tessellator to generate vertices for N^2 isolines. The Tessellation Evaluation Shader (TES) is executed once for each of the generated vertices. Each invocation of the TES evaluates the visibility of a scene node, sends the visibility results to the Geometry Shader (GS) along with the primitive and saves the visible leaf into the visible nodes lists. The geometry shader is set to perform hardware instance N times, and each instance of the GS computes the visibility of one of the children of the scene node which was processed in the TES and whose visibility result was sent to the GS. If visible leaf nodes are found, they are added to the visible node list. The GS then evaluates the visibility of all the children of the node that was sent to the GS, effectively evaluating the next level of the scene tree. If visible leaf nodes are found, they are added to the visible node list. The GS then creates geometry for N billboards, following the same principles as the task generator from Chapter 3.3. Each of the N billboards generates a large number fragments which represent leaf tasks, which are sent to the Fragment Shader for evaluation.

The visible nodes list can be drawn without CPU control, through the use of indirect rendering. The culling algorithm can also be made aware of impostors, by maintaining two node lists, one for visible nodes which will be rendered geometrically and one for visible nodes which will be rendered with impostors. The Hierarchical Impostors presented in Chapter 3.2. can be tightly integrated with the presented culling solution, each GPU thread can compute the geometry level of detail or impostor for each object, besides performing view frustum culling.

Occlusion Culling can be handled through the integration of a Hierarchical Depth Culling mechanism, but this would make the culling solution inexact and prone to temporal artifacts.

Multiple frame culling is the final contribution of the presented culling algorithm. Instead of just culling objects for the current frame, the analysis shows that many objects are culled for relatively long durations. While this is difficult to prove efficiently for occluded objects, it is relatively easy to do for view frustum culled objects. As presented in Figure 37, objects can be trivially culled by just analyzing the speed of the camera.

If the camera direction is quantified as a solid angle and if it is considered that in real-time applications the camera has hard caps on its orientation change speed, then the entire scene can be partitioned. The partition is based on the idea that the camera would reach the partition area only after at least a number of frames at maximum speed, which is usually much less than the normal application speed. If an object is stationary inside a partition that is four maximum speed frames away from the camera, then it is guaranteed to be culled for the next four frames. The same conclusion can be taken for an object whose trajectory can be quickly evaluated on the GPU, using a method similar to that described in 3.1, and which is guaranteed to remain in such a partition.

The only exception happens when the camera is instantly oriented or moved, which is naturally handled with a reset of the entire multiple frame culling mechanism. For normal camera speed movements the multiple frame culling mechanism can lead to a large decrease in computational effort, because it drastically increases the number of operations. Furthermore, since the camera suffers reorientations constantly, the set of objects culled for more than one frame varies in time, therefore the starting set of five frames culled objects, as depicted in Figure 37, will not all reach reevaluation concomitantly. Therefore, the multiple frame culling computations are amortized over different frames. The amortization can further be sophisticated with an implicit checking order over the culled frames.

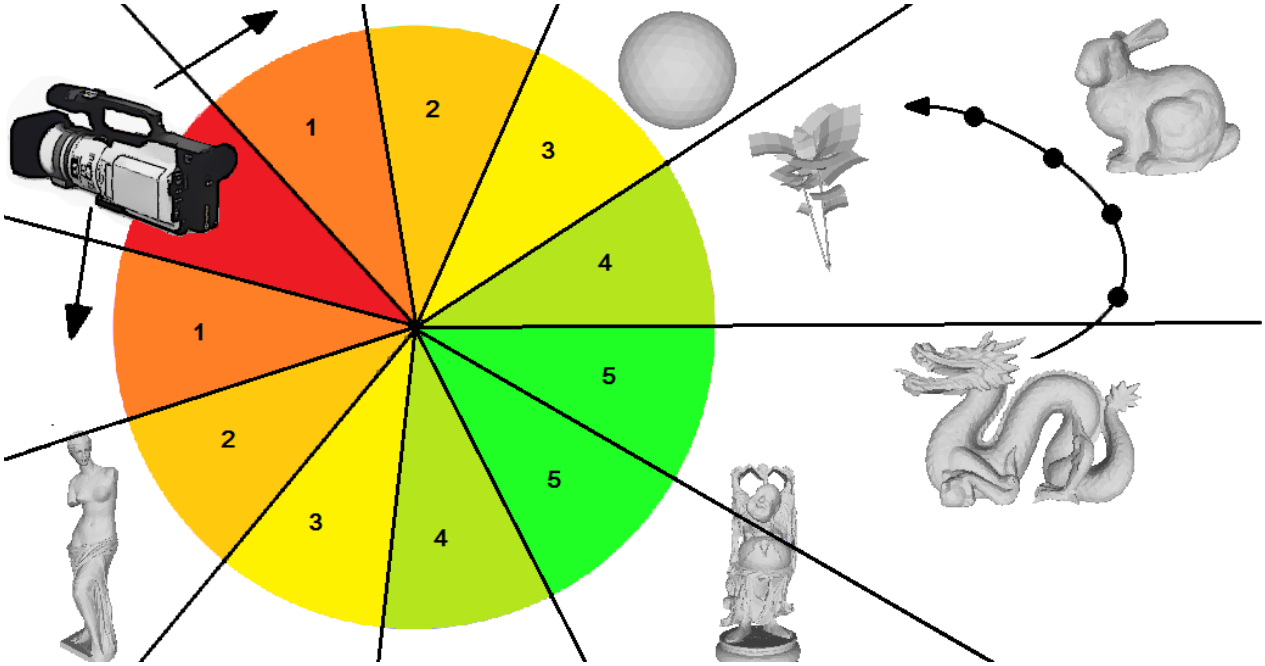

Figure 37 Multi frame culling**.** The image shows that the entire scene can be partitioned into radial zones, based on the smallest number of frames it would take the camera to reach them, while rotating at maximum speed. This optimization can lead to large computational cost decrease in many common rendering scenarios.

Storing the number of culled frames does not have to be performed inside a large precision number, because the maximum speed for camera in real-time applications is relatively large. The solution of an additional byte for each scene node, one bit to distinguish between leaf nodes and other scene nodes, four bits to store the number of frames culled, one bit to determine if the object is transparent or not and one bit to determine if the object is a light or not. With each passing frame the previously culled object will be have their count recomputed or decreased.

Since the algorithm is constructed with an object partitioning scheme in mind, like BVH, it is sufficient to decrease the multi frame culling value for an invisible parent and to update the child number of culled frames the next time it is walked, by comparing the multi frame culling value with that of its parent. If the value is greater, then the child has not been reached for a sufficient duration that its number of culled frames is guaranteed to have expired.

The pseudocode for the culling algorithm is shown on the next page.

```
CPU (root, camera)
initialize indexarray← 0
initialize numframesarray←0
childlist ← root
WHILE size of childlist < threshold
        list ← childlist
        childlist ← Ø
        FOR node in list
                IF node visible
                        FOR childnode of node
                                childlist ← childlist ∪ childnode
        IF dynamic parallelism available
                indexarray ← DYNAMIC (childlist)
        ELSE
                FOR node in childlist
                        indexarray ← indexarray ∪ VERTEX (node)
        renderqueue ←use indexarray to generate rendering queue
        draw renderqueue

EVALUATE(node, camera)
aabb ←axis aligned bounding box of node
visible, camerazone ← compute View Frustum Culling with aabb
IF visible
        indexarray ← set visible in index array
ELSE
        numframes ← get the number of frames from camera to camerazone
        numframesentry← numframesarray, node
        IF numframes >numframesentry
                numframesarray←set numframes
IF occlusion culling available
        COMPUTE occlusion culling
RETURN visible

DYNAMIC (nodes, camera)
FOR node in nodes
        visible ← EVALUATE(node, camera)
IF visible
        children ← get node children
        DYNAMIC(children, camera)

VS (node, camera)
visible ← EVALUATE(node, camera)
IF visible
        children ← get node children
        FOR childnode in children
                TCS(childnode, camera)
TCS (node, camera)
visible ← EVALUATE(node, camera)
IF visible
        children ← get node children
        FOR childnode in children
                visible ← EVALUATE(childnode, camera)
                IF visible
                        grandchildren← get childnode children
                        FOR grandchildnode in grandchildren
                                TCS(grandchildnode, camera)
Set TES to run for total number of grandchildren
```

```
TES (node, camera)
visible ← EVALUATE(node, camera)
IF visible
        children ← get node children
        FOR childnode in children
                visible ← EVALUATE(childnode, camera)
                IF visible
                        grandchildren← get childnode children
                        FOR grandchildnode in grandchildren
                                GS(grandchildnode, camera)

GS (node, camera)
instances←scene tree max number of child nodes
FOR instance in instances
        ichild ← ith child of node
        visible← EVALUATE ichild
        IF visible
                FOR child of ichild
                        visible ← EVALUATE child
                        grandchildren , numgrandchildren ←number of children of child
                        width ← 2 x numgrandchildren / screenwidth x perthread x screenheight
                        height ← 2 / screenwidth – 1
                        billboard ← create a billboard in NDC with width and height
                        fragments← RASTERIZE(billboards)
                        FOR fragment in fragments
                                grandchild ← pop grandchildren
                                FS(grandchild, camera)
FS (node, camera)
        EVALUATE (node, camera)
```

If the algorithm is used in combination with hierarchical occlusion culling, a minor optimization is to conservatively approximate opacity per pixel, and cull the object against the opacity occlusion. On the other hand this optimization can only be implemented with occlusion impostors.

Compared to the state of the art, the presented Hierarchical GPU Culling method runs on the GPU **without CPU interference**, without synchronization and wait mechanisms. It runs hierarchically and while it does not explicitly solve occlusion culling it can be tweaked to use Hierarchical Depth Culling. The presented algorithm uses a multiple frame culling mechanism, which can cull objects for more than one frame, lowering computational costs. The culling algorithm is integrated with the hierarchical impostor method presented in Chapter 3.2 and can also be implemented with dynamic parallelism.

The presented technique draws without CPU interference, by using indirect rendering.

The algorithm has been tested on very large scenes with different object distributions, such as the ones presented in Figure 38, with results culling results slightly inferior to those of CHC++ but with superior running time due to lack of CPU synchronization and to the better fit of the algorithm to the GPU many core architecture.

Figure 38 Hierarchical GPU Culling Results**.** The presented hierarchical GPU Culling algorithm has been tested on large scenes with different object distributions, such as the ones displayed in the image above. The view on the left is especially difficult to cull, because it is extremely incoherent spatially. Scenes with a large number of objects benefit greatly from multiple frames culled objects.

3.5. Opaque Rasterization

Opaque rasterization represents the process through which opaque objects are rasterized. The majority of rendered objects in real-time scenarios are opaque; therefore this area of rasterization based rendering has been thoroughly examined. Yet, with new opportunities offered by increasingly performant consumer hardware raise new opaque rasterization challenges, especially in bandwidth and memory usage.

In massive scenes opaque rasterization is usually performed with multiple depth frusta [**Coz09**], because the depth precision isn't stored in a linear format but in one directly proportional to the inverse of depth.

As the precision is not stored linearly and the resolution of the depth buffer is biased towards the near clipping plane and not towards the far plane, fragments from objects that are rasterized close to the far plane can suffer from an effect called z-fighting, where the z-Buffer algorithm returns incorrect results due to lack of precision. Because of this, even 32 bits of resolution are not sufficient for increasingly complex scenes.

The depth precision problem is illustrated in Figure 39. [**Ree15**] offers an in-depth discussion on this topic. A common solution to this problem is to modify the resolution of the depth buffer through multiple depth frustums [**Coz09**], or through modify the distribution of the depth samples with a logarithmic depth buffer [**Coz09**]. On the other hand a logarithmic depth buffer writes depth explicitly and this prevents critical rasterization optimizations [**Gre93**].

An efficient approximate solution to depth precision is displayed in Figure 39. The depth range can be reversed, from the normal 0 to 1 to 1 to 0, which has the effect of bringing the depth sampling closer to a uniform sampling distribution. This method is named the semi logarithmic depth buffer [**Ree15**].

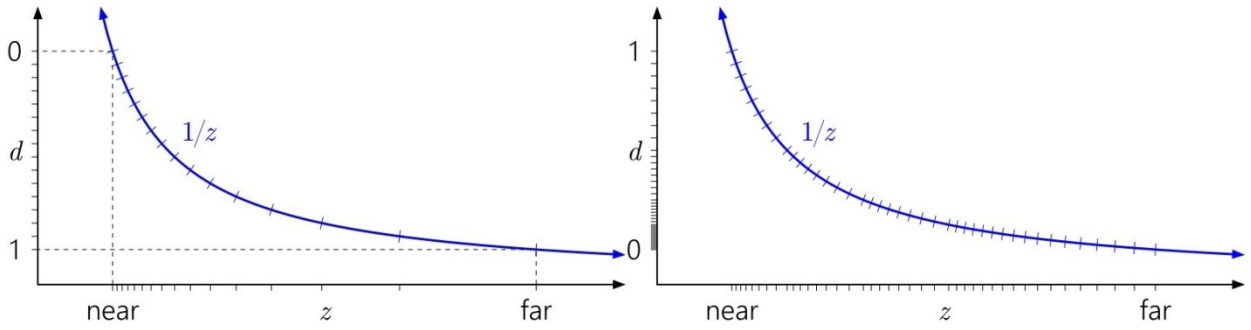

Figure 39 Depth Precision**.** The image on the left shows a normal depth buffer, and it shows that depth does not vary linearly, as it varies with 1/z. It shows that the hardware depth buffer allocates more precision in the proximity of the near clipping plane, and that precision decreases rapidly as the depth values approach the far clipping plane. This can adversely affect the stored depth values and lead to the incorrect Z-Buffer results, which lead to the z-fighting effect. A solution to the depth precision problem is presented on in the right side of the image, in which the depth range is reversed. This is called the quasi logarithmic depth buffer, and it samples with a close to uniform distribution. Images from **[**Ree15**]**.

Wrinkled surface rendering is a subfield of rendering which studies the detailed rendering of low frequency geometric meshes, which are augmented with visual detail stored in textured maps. There are two major directions for wrinkled surface rendering: mesh tessellation, which reconstructs geometry at a geometric level and mesh mapping, which reconstructs the geometry only in aspect.

Mesh tessellation [**Bou081**] [**Loo09**] [**Dyk09**] creates a large amount of vertices which are then displaced with displacement mapping, and which can exactly reconstruct a low frequency mesh. The weakness of this approach in the context of rasterization is in the resulting rendering alias, because a large number of created vertices is projected on a small number of pixels. This large number of vertices has to be heavily multisampled, in order for the geometric signal to be properly reconstructed; otherwise the pixel will show aliasing. Furthermore, the computational cost can be extremely high in exceedingly tessellated scenes. For this approach to be productive, the tessellation level has to adapt to the projected surface size, a method named adaptive tessellation.

In the context of rasterization rendering, mesh mapping [**Bli78**] [**Coh98**] [**Kan01**] [**Bra04**] [**Tat06**] [**Pre06**] [**Pol07**] is a more efficient alternative, because it works at pixel level. This approach is less computationally expensive and produces better visual results than extreme tessellation. On the other hand mesh mapping isn't as expressive as judicious tessellation, and usually mesh tessellation and mesh mapping are both applied, tessellation for the large sub-polygonal details and mapping for pixel-level effects.

The presented pipeline uses a combination of adaptive tessellation with Gregory Patches [**Loo09**] and adaptive use of either parallax iterative mapping [**Pre06**] or normal mapping [**Coh98**], depending on distance and angle of view. This combination is easy to tweak for different surfaces and minimizes aliasing without entailing an excessive computational cost. Screen space parallax mapping [**Lob08**] can also be used, but it is prone to reconstruction artifacts, especially for small on-screen contributors. The wrinkled surface rendering path is described in Figure 40.

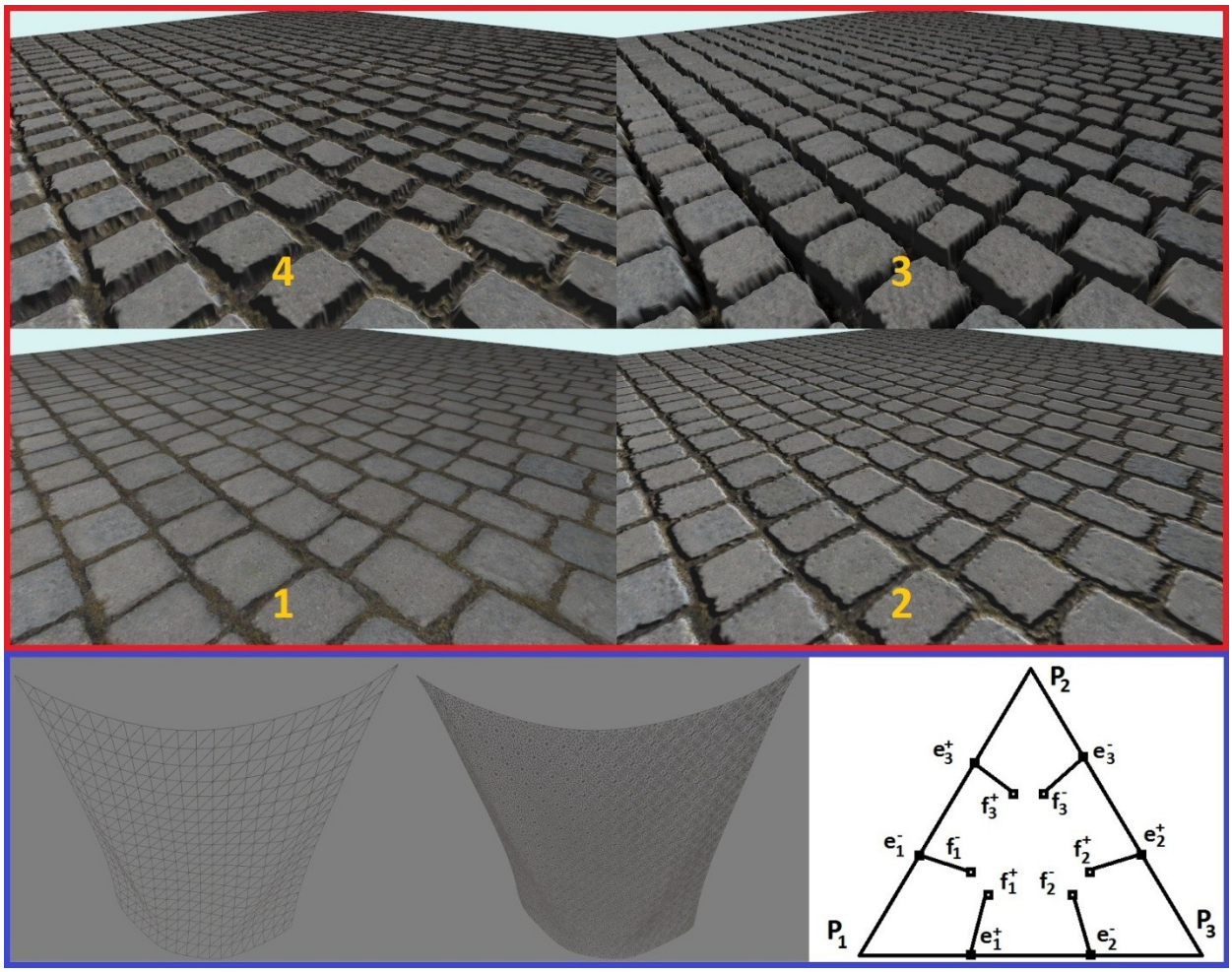

Figure 40 Wrinkled Surface Rendering**.** Tessellation is described in the lower part of the image, where each triangle is adaptive tessellated and displaced using either displacement map information or Gregory Patches, which is displayed in the right lower corner and is used if no displacement information is available. The superior part of the image displays several mesh mapping render paths: no wrinkled surface, normal mapped, iterative parallax mapped, offset limited iterative parallax mapped with soft shadows.

Wrinkled surface rendering requires working in the tangent space, which either means asset preprocessing or tangent space reconstruction during rendering [**Sch06**] [**Sch15**]. In the case of asset pre-processing this can be performed geometrically, by storing the tangent as a per-vertex attribute or through derivative maps (dudv), such as the ones depicted in Image 24. Tangent space reconstruction is performed analytically with screen space derivatives [**Sch15**], which are computed using the depth data available from the geometry buffer.

This is the pseudocode for the wrinkled surface rendering path:

```
TCS (primitive, camera)
ssaabb ← projected primitive screen space bidimensional bounding box
N ← number of pixels in ssabbb
normal ← normal of primitive
center ← center of primitive
cameraray ← ray from camera to center
A ← angle between cameraray and normal
IF N > threshold Tn
        IF A < threshold TA
```

```
                generatedvertices, tessellationfactors ← set tessellation factors to tessellate primitive
                controlpoints ← ∅
                IF displacementmap available
                        FOR vertex in primitive
                                controlpoints ← controlpoints ∪ vertex
                ELSE
                        cornerpoints ← gregory corner control points (P1, P2, P3)
                        edgepoints   ← gregory edge control points (e0-, e0+, e1-, e1+, e2-, e2+)
                        facepoints   ← gregory face control points (f0-, f0+, f1-, f1+, f2-, f2+)
                        specialpoints ← gregory special control points (F1, F2, F3)
                        controlpoints ← cornerpoints ∪ edgepoints ∪ facepoints ∪ specialpoints
                FOR vertex in generatedvertices
                        TES ( controlpoints)
ELSE
        FOR vertex in primitive
                TES (vertex)

TES (controlpoints)
barycentric ← obtain barycentric coordinates from tesselator
IF displacementmap available
        position, normal← compute position and normal with barycentric and controlpoints
        displacement ← read from displacementmap
        displacednormal← read from normalmap, use normal to compute tangent space
        displacedposition ← displace position with displacement
ELSE
        displacednormal, displacedposition ← controlpoints
RASTERIZE vertex with displacementnormal and displacementposition

FS (position, normal)
camera ← get rasterization camera
IF tangent not available
        derivatives ←compute derivatives required for tangent space reconstruction
        tangentspace ← reconstruct tangent space with derivatives
IF not alpha culled
        fragmentdistance ← distance from camera to position
        IF fragmentdistance < high quality threshold
                fragmentcolor ← parallax iterative mapped with secant, soft shadows , textured
        IF fragmentdistance < high quality threshold
                fragmentcolor ←parallax iterative mapped with secant, textured
        IF fragmentdistance < medium quality threshold
                fragmentcolor ←normal mapping, textured
        ELSE
                fragmentcolor ←textured
        OUTPUT fragmentcolor
```

Opaque surface rendering usually represents the largest computational effort in rasterization rendering, therefore the computational cost of this process has been analyzed in depth. Deferred and decoupled algorithms separate the geometry processing and shading operations, in an effort to minimize the surface-light-camera interactions. The presented pipeline is based on the same principle. The problems with deferred and decoupled algorithms appear when they are analyzed more thoroughly. In general, these algorithms suffer from high bandwidth, high geometry or high storage costs.

3.5.1. Analyzing Deferred Rendering

In opaque rasterization rendering process each camera-surface interaction is computed, and, through the z-buffer algorithm, the closest camera-surface interaction is determined and displayed on screen. This can lead to a large amount of unneeded computations, such as shading occluded fragments, which can be avoided by using deferred methods, as shown in Figure 41.

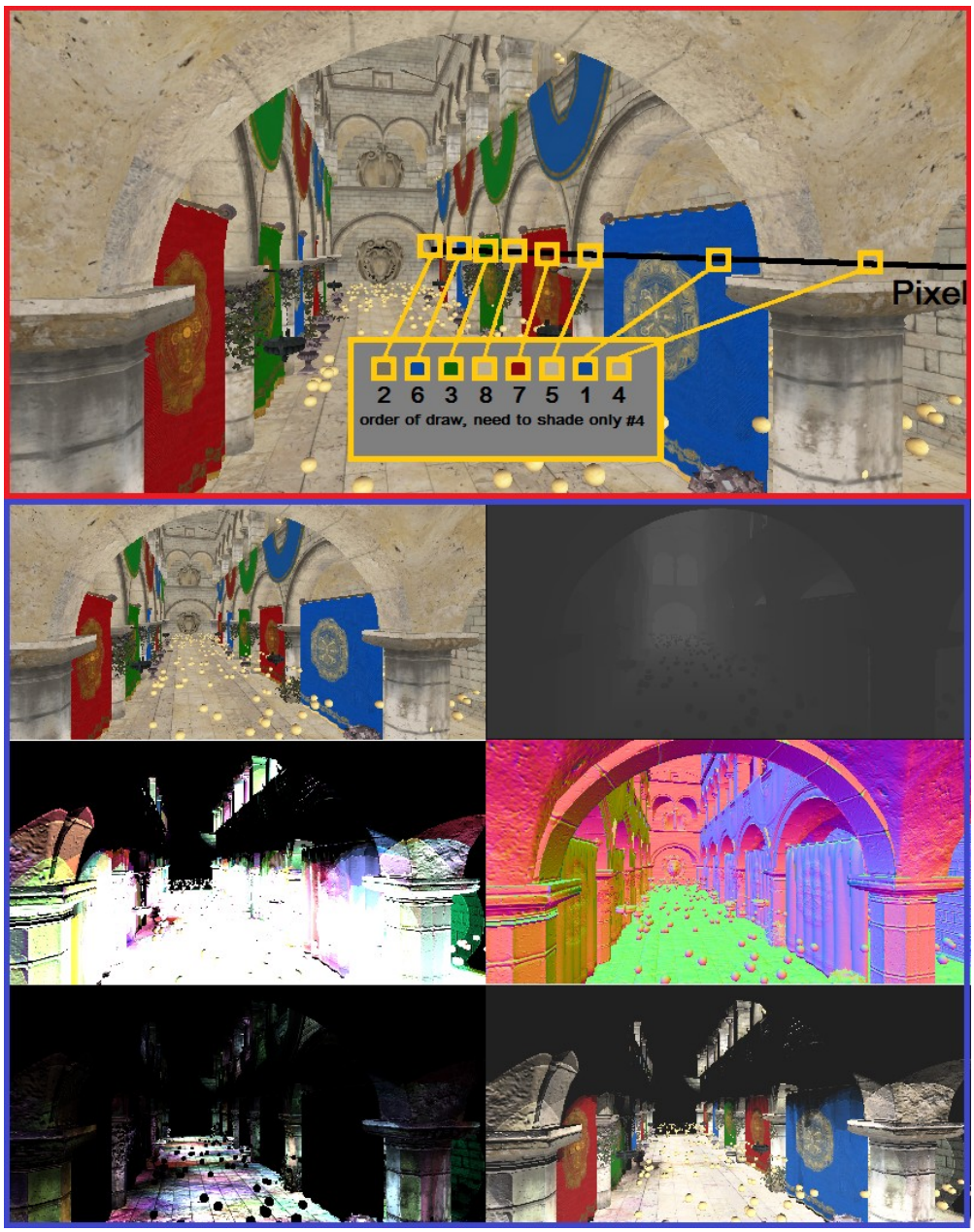

Figure 41 Why deferred**.** The upper part of the above image presents the surface-camera interactions which take place over a Pixel. A large number of per-pixel interactions can take place in complex scene, thus shading occluded interactions would be a very large and useless computation effort. Deferred rendering is widely used in real-time rendering applications because it does not shade occluded fragments, shading only the fragments visible on screen. It does this by storing shading information (depth, normals, colors, roughness, etc) in large screen sized buffers, which are then lit and shaded, as presented in the lower part of the image.

Deferred rendering decreases the number of shaded surface-camera interaction by shading only the interactions which are visible on the screen. Because of this, deferred rendering drastically lowers the complexity of opaque object rasterization rendering from $O(lights * objects)$ to $O(lights + objects)$. This complexity reduction is one the main reasons, for which deferred rendering is extremely popular in real-time rendering applications.

The deferred rendering idea has been further advanced with deferred decoupled rendering [**Lik12**], which completely decouples shading samples from fragments. While the idea of deferred rendering is simple, the implementation details have created difficult to solve problems such as correct anti-aliasing integration, transparent object integration, bandwidth and shading decoupling, light-surface acceleration and so on. Because of these problems, there are a large variety of methods which take different approaches to opaque rasterization, but these methods do not have taxonomy and lack more in-depth metrics to improve their comparison.

Deferred methods can be classified by the number of geometry passes: single geometry pass and multiple geometry pass. Deferred methods can also be classified by the surface-light interaction method which can either be implicit, explicit or decoupled. A large part of the deferred algorithms can be adapted to either be single or multiple geometry pass methods. Single pass methods have lower geometry processing costs, while multiple pass methods usually have lower bandwidth.

The implicit methods [**Lee09**] [**van13**] [**Thi09**] [**Mar14**] [**Hol13**] [**Seg06**] accelerate intersection of lights and objects through the raster grid structure, which acts as an implicit bidimensional associative array, in which the objects are binned, and in which the objects intersecting the lights are queried during the rasterization of lights, in the lighting stage.

Explicit methods [**Tre09**] [**Lau12**] [**Ols12**] [**Ols11**] [**Har12**] [**Hob09**] [**Bur13**] use acceleration structures instead of the raster grid. These include bitwise lists, per-pixel lists, tiles, 2.5D tiles and clusters. These explicit acceleration structures appear to not be hierarchic but it is only an appearance. When the structures are filled through rasterization of all lights, the raster hierarchically [**Gre93**] intersects them (as fragments) with the screen, and consequently with the acceleration structure. Thus, explicit structures are still created hierarchically.

Decoupled methods [**Lik12**] [**Cla13**] run rendering stages at distinct sampling rates, where the samples are linked through many-to-one or many-to-many mappings in addition to other acceleration structures. The disadvantage of the current state of the art decoupled methods is that they are still very expensive for the consumer hardware.

This thesis separates bandwidth consumption, processing cost, storage (memory) cost, and state cost, in order to ease the comparison between deferred methods. The different performance metrics can also be used to switch between deferred algorithms during rendering, depending on the situation, in the same spirit as hybrid deferred rendering [**van13**].

A comparison of state of the art deferred algorithm is provided in Table 5 and Table 6. While the presented comparison does not provide the measurement equations for the proposed metrics, the equations can be found together with the original analysis in the "Analyzing Deferred Rendering Techniques" article [**Pet15**].

Algorithm\Criterion	Light-Object Intersection Acceleration	Transparency Support	Hardware MSAA Support	Light Data Access Pattern	Shading Data Access Pattern
Forward	implicit	yes	yes	random	random
Depth Pre Pass	implicit	no	yes	random	random
Deferred	implicit	no	no	sequential	random
Deep Deferred	implicit	partial	depends	sequential	random
Light Pre Pass	implicit	yes	yes	sequential	random
Deferred Transparency	implicit	partial	no	sequential	random
Light Indexed Deferred	explicit	no	no	random	sequential
Light Indexed Forward	explicit	yes	yes	random	sequential
List Light Indexed Deferred	explicit	no	no	random	sequential
List Light Indexed Forward	explicit	yes	yes	random	sequential
Tiled Deferred	explicit	no	no	random	sequential
Tiled Forward	explicit	yes	yes	random	sequential
Forward+	explicit	yes	yes	random	sequential
Clustered Deferred	explicit	no	no	random	sequential
Clustered Forward	explicit	yes	yes	random	sequential
Deferred++	decoupled	yes	yes	random	sequential
Deferred Decoupled Sampling	decoupled	yes	no	random	sequential
Sort based deferred	decoupled	yes	yes	random	sequential

Table 5 Deferred Algorithms Comparison - I**.** The table provides a basic comparison of deferred rendering algorithms with respect to data access patterns and functionality.

Algorithm\Criterion	Decouples texture sampling	Decouples vertex attributes	GPU Commands Cost	Processing Cost	Allocated Memory Cost	Bandwidth Cost
Forward	no	no	very high	very high	very low	very high
Depth Pre Pass	no	no	very high	high	low	low
Deferred	no	no	low	low	high	high
Deep Deferred	no	no	low	high	very high	very high
Light Pre Pass	no	no	high	high	low	low
Deferred Transparency	no	no	low	high	very high	very high
Light Indexed Deferred	no	no	low	low	high	high
Light Indexed Forward	no	no	high	high	high	low
List Light Indexed Deferred	no	no	low	low	high	high
List Light Indexed Forward	no	no	high	high	high	low
Tiled Deferred	no	no	low	low	high	high
Tiled Forward	no	no	high	high	low	low
Forward+	no	no	high	high	low	low
Clustered Deferred	no	no	low	low	high	high
Clustered Forward	no	no	high	high	low	low
Deferred++	yes	no	low	high	low	low
Deferred Decoupled Sampling	yes	no	high	high	high	low
Sort based deferred	yes	yes	low	high	high	low

Table 6 Deferred Algorithms Comparison - II**.** The table provides a comparison of deferred rendering algorithms, by showing the approximate costs in different metrics.

The only state of the art algorithms that completely decouple texture sampling from visibility determination are all very costly from a processing standpoint because they either use expensive GPU synchronization to implement a cache, or completely reconstruct the geometry with all the attributes per fragment. In the next subchapter a novel deferred algorithm is introduced, called virtual deferred, which decouples texture sampling from visibility determination, while still having processing costs comparable to light pre pass deferred rendering.

3.5.2. Virtual Deferred

A common problem with state of the art algorithm is that they either decouple shading bandwidth from visibility determination and perform a multiple geometry passes or couple shading bandwidth with visibility determination and perform a single geometry pass. No deferred technique decouples visibility determination from texture and shading bandwidth and renders the geometry only once. This thesis introduces virtual deferred, a new deferred algorithm, which is able to perform this decoupling in a single geometry pass, based on the article "Virtual Deferred Rendering" [**Pet151**].

The introduced algorithm is a combination between virtual data methods and deferred algorithms, using the virtual texturing mechanism to s**tore only critical geometric and texturing data** in a small modified geometry buffer, **consuming texture bandwidth only when it affects the geometry rendering process**, for example for alpha culled or displacement mapped objects. In doing so virtual deferred guarantees **complete decoupling** between visibility determination methods, shading bandwidth, illumination and shading. Virtual deferred rendering offers these properties without a complex and hard to implement memoization cache, such as the one used in decoupled rendering [**Rag11**] [**Lik12**].

Virtual rendering is based on the idea of using virtual texturing to maximize deferring opportunities. Instead of storing texture values inside the G-buffer, the presented method stores texture coordinates and their derivatives. Thus, instead of consuming bandwidth for each occluded primitive for which textures are read, saved into the geometry buffer and then overwritten, virtual deferred saves only the absolute minimal texturing information. Depending on the scene configuration the texture coordinates and their derivates can be packed, as it is shown in Figure 42.

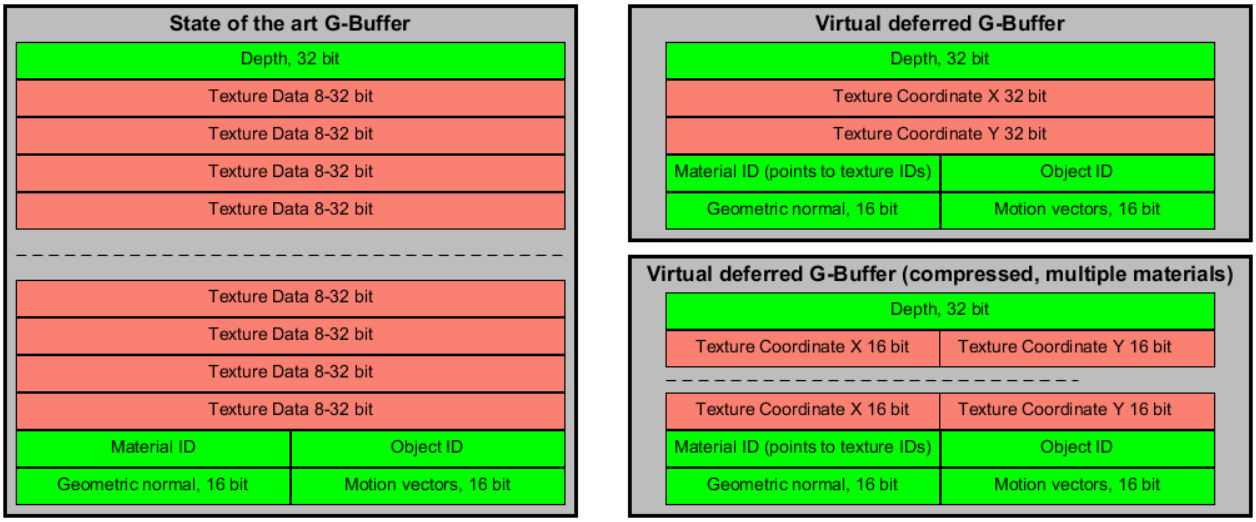

Figure 42 Virtual deferred G-Buffer**.** The image highlights the difference between state of the art geometry buffers and the one used in virtual deferred. Compared to state of the art geometry buffers, virtual deferred stores only critical texturing information, the texture coordinates and their derivatives. Because of this, when the deferred renderer is used in scenes with complex materials, the geometry buffer size scales better for virtual deferred than for the state of the art methods. The virtual deferred G-Buffer can also be further decreased in size through texture coordinate compression, if the scene configuration supports it.

The illumination stage of virtual deferred is identical to the illumination stage of clustered deferred, where the lights are first clustered into an acceleration structure and are then intersected with the G-buffer. In the shading stage, instead of loading the texture data stored in the G-buffer, virtual deferred reads the virtual texture whose coordinates were saved into the G-buffer. The texture fetches are then performed with the loaded texture coordinates and derivatives. Because virtual deferred is based on virtual texturing, it also has the benefit of being extremely easy to integrate into a virtual texturing system.

Virtual deferred can be combined with decoupled sub pixel reconstructed antialiasing (DSRAA), an antialiasing algorithm presented in the Post Processing sub-chapter, in the Illumination Chapter. The illumination stage of virtual deferred is presented in detail in the Illumination chapter. The pseudocode for virtual deferred is:

```
(ONCE) PREPROCESS
IF using multiple frusta
        frusta ← subdivide the visual volume frustum into multiple frusta
        N ← number of frusta
        objects[N], lights[N], lightgrid[N], depthbuffer[N], Gbuffer[N], visibilitybuffer[N] ← ∅,init
        FOR object in objects
                FOR frustum in frusta
                        i← frustum number
                        IF object in frustum
                                IF object visible
                                        objects[i] ← objects[i] ∪ object
        FOR light in lights
                FOR frustum in frusta
                        i← frustum number
                        IF light in frustum
                        lights [i]← lights[i]∪ object
        SORT frusta in front to back order
ELSE
        frusta ← frustum

GEOMETRY STAGE (objects)
FOR frustum in frusta
        IF using multiple frusta and frustum not first          //MULTIPLE FRUSTA ZBUFFER
                prevfrustum← get previous frustum from frusta
                prevdepthbuffer ← get depth buffer of prevfrustum
                depthbuffer ← frustum depth buffer
                FOR pixel in screenpixels
                        prevdepth← prevdepthbuffer value for pixel
                        IF prevdepth < far distance of prevfrustum
                                depth ← near distance of frustum
        i ← frustum index
        FOR object in frustumobjects[i]
                IF using DSRAA
                        fragments, visibilitysamples ← render object, generate fragments
                        visibilitybuffer[i]← visibilitysamples(depth, normal optional)
                ELSE
                        fragments ← render object, generate fragments
                FOR fragment in fragments
                        IF fragment is visible
                                depthbuffer[i] ← update
                        Gbuffer[i] ← store fragment data into virtual deferred gbuffer
```

```
LIGHTS STAGE(lights)
FOR frustum in frusta
        i ← frustum index
        FOR light in lights[i]
                depthbuffer[i] ← depth buffer of frustum, created in GEOMETRY STAGE
                fragments ← render light, generate fragments, cull using depthbuffer[i]
                FOR fragment in fragments
                        IF fragment is visible
                                cluster ← determine location in lightgrid[i]
                                cluster ← cluster∪ light

VIRTUAL TEXTURE STAGE
streamlist ← Ø
FOR frustum in frusta
        i ← frustum index
        FOR pixel in Gbuffer[i]
                IF pixel depth < frustum max depth
                        texderivatives ← load texture derivatives from pixel in Gbuffer[i]
                        material ← material of pixel
                        FOR texture in material
                                texmipmap ← determine mipmap level of texture with texderivatives
                                IF texmipmap not in virtualtexture
                                streamlist ← streamlist ∪ texmipmap
IF streamlist not Ø
        COMPACT streamlist
        WHILE streamlist not Ø
                mipmap ← pop streamlist
                send streaming command to virtual texture to stream mipmap
```

The shading stage for the virtual deferred algorithm is discussed in chapter 4.1.4, in the illumination chapter, as it is has more to do with light transport than with geometric computations. It is described here with an emphasis on the geometric constraints given by the multiple frusta, in the 4.1.4 chapter the emphasis is put on light transport and texture fetching.

```
SHADING STAGE (fragment)
FOR frustum in frusta
        i ← frustum index
        FOR pixel in Gbuffer[i]
                visible ← true
                IF i>0
                        prevdepth ← depthbuffer[i-1]
                        IF prevdepth < max depth for previous frustum
                                visible←false
                IF visible
                        pixeldata ← load non texturing pixel data
                        texturedata← load texturing data through  virtual texturing
                        diffuse ← compute low frequency illumination with lights[i] and other scene lights
                        specular ← compute high frequency illumination with lights[i] and other scene lights
                        pixelcolor ← diffuse, specular, pixeldata, texturedata
                        OUTPUT pixelcolor
```

From a complexity standpoint, virtual deferred lowers the storage and texture bandwidth complexity, as it is shown in Table 7 and Figure 43.

Algorithm\Criterion	Single Geometry Pass Deferred Rendering	Multiple Geometry Pass Deferred Rendering	Decoupled Deferred Rendering	Virtual Deferred
Geometry Processing	1x	2x	1x	1x
State changes	1x	2x	1x	1x
Texture bandwidth	T*TS*D	T*TS	T*TS	T*TS
Geometry buffer storage	Geometric Information + T*TS	Geometric Information	Geometric Information + memoization	Geometric Information + 2

Table 7 Virtual deferred and the state of the art**.** The table compares virtual deferred with state of the art deferred rendering families: single geometry pass deferred, multiple geometry pass deferred and decoupled deferred rendering. Virtual deferred combines the most desirable properties from each of these algorithm families.

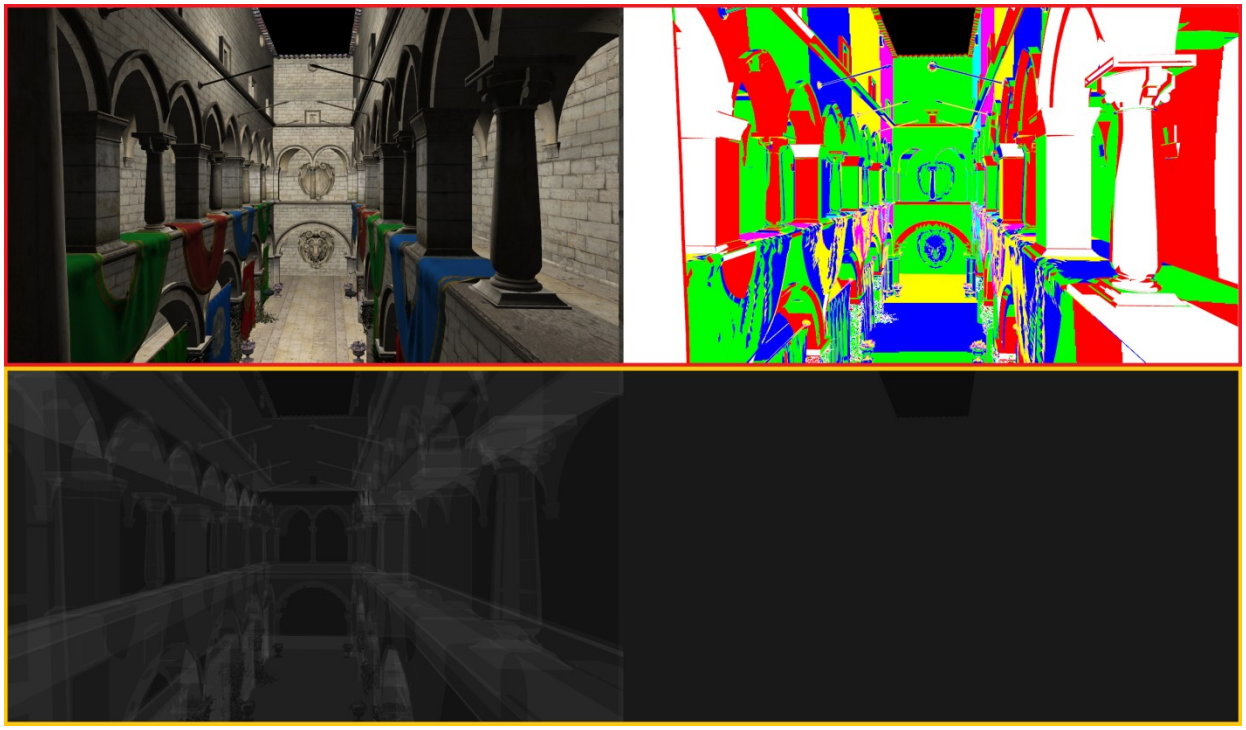

Figure 43 Virtual deferred Results**.** The upper part of the image shows the rendered scene and the mipmaps levels for the rendered textured. Virtual deferred renders the opaque scene objects into a modified G-Buffer, which stores only texture coordinates and their derivatives as texturing data, without storing effective texture data. The textures data is then loaded in the shading pass, through the texture coordinates and the object id. The lower part of the image shows the results of virtual deferred (right), compared to a standard single pass deferred rendered (on the left). The images in the lower part encode bandwidth consumption, darker is better.

The virtual deferred algorithm can be improved with a screen space optimization, in which the texture coordinates derivatives are computed through neighbor differentiating, like the ddx and ddy instructions in consumer hardware. This optimization further lowers the storage and bandwidth consumption of the presented algorithm.

Virtual deferred is not without problems, the biggest of which is multiple material support. Since texel fetching is performed through virtual texturing, it is impossible to texture a multiple textured material without storing multiple texture coordinates and their derivatives. This is easy to observe in a scene which is lit through precomputed lightmaps. Therefore, care must be taken to minimize the number of such problems, especially in systems with heavily varied

texturing systems. On the other hand such systems are slowly becoming obsolete as more work is performed during rendering and less is precomputed.

Even so, virtual deferred performs better than the state of the art single pass deferred methods, as can be seen in Figures 44 and 45. An analysis of the storage consumption of virtual deferred is offered in Figure 44.

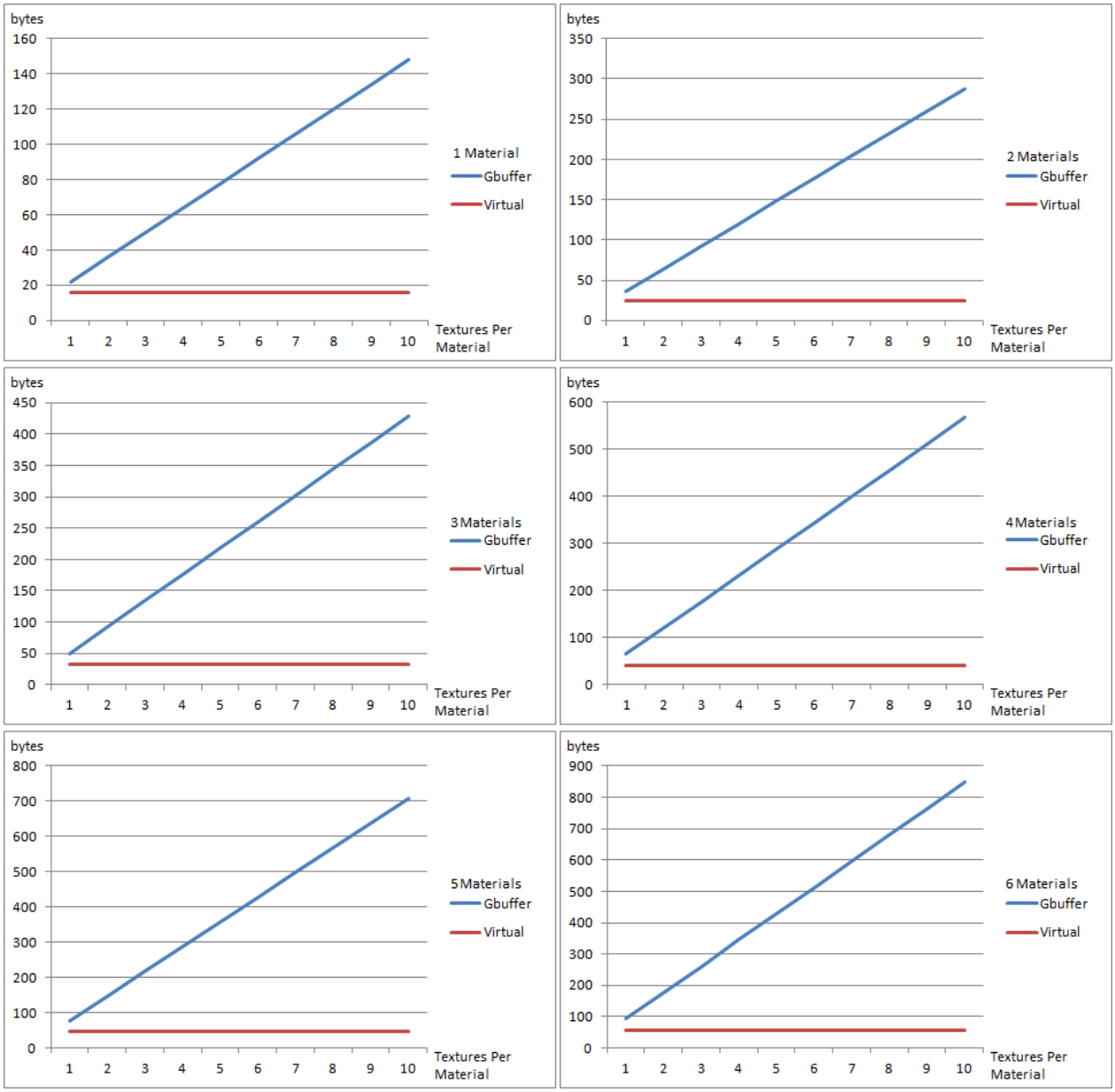

Figure 44 Virtual deferred Per Pixel Storage Analysis**.** The image depicts the storage costs difference between a single geometry pass deferred G-buffer, colored in blue, and the virtual deferred G-Buffer, colored in red. As long as all the texture in the same material use the same coordinates, a vastly common texturing setup, the virtual deferred algorithm will store less bytes pixel than the state of the art single geometry pass deferred. The difference between methods grows with the number of materials, as standard G-Buffers have a larger storage complexity than that of the virtual deferred G-Buffer. Multiple geometry pass deferred algorithms are not depicted in this image as their large geometry processing cost makes them less suitable for massive scenes.

The bandwidth consumption of virtual deferred is compared to the of the geometry buffer in Figure 45. It has to be noted that the bandwidth increase in virtual deferred is caused by the geometric data processed at a fragment level, as the algorithm loads only the shading data that is used for visible objects. If in Figure 45 the geometric bandwidth cost would not be considered, and the comparison would be only between the G-buffer and virtual buffer texture fetching bandwidth, the virtual buffer would have a constant cost in the number of different texture mappings per object, which is normally just one.

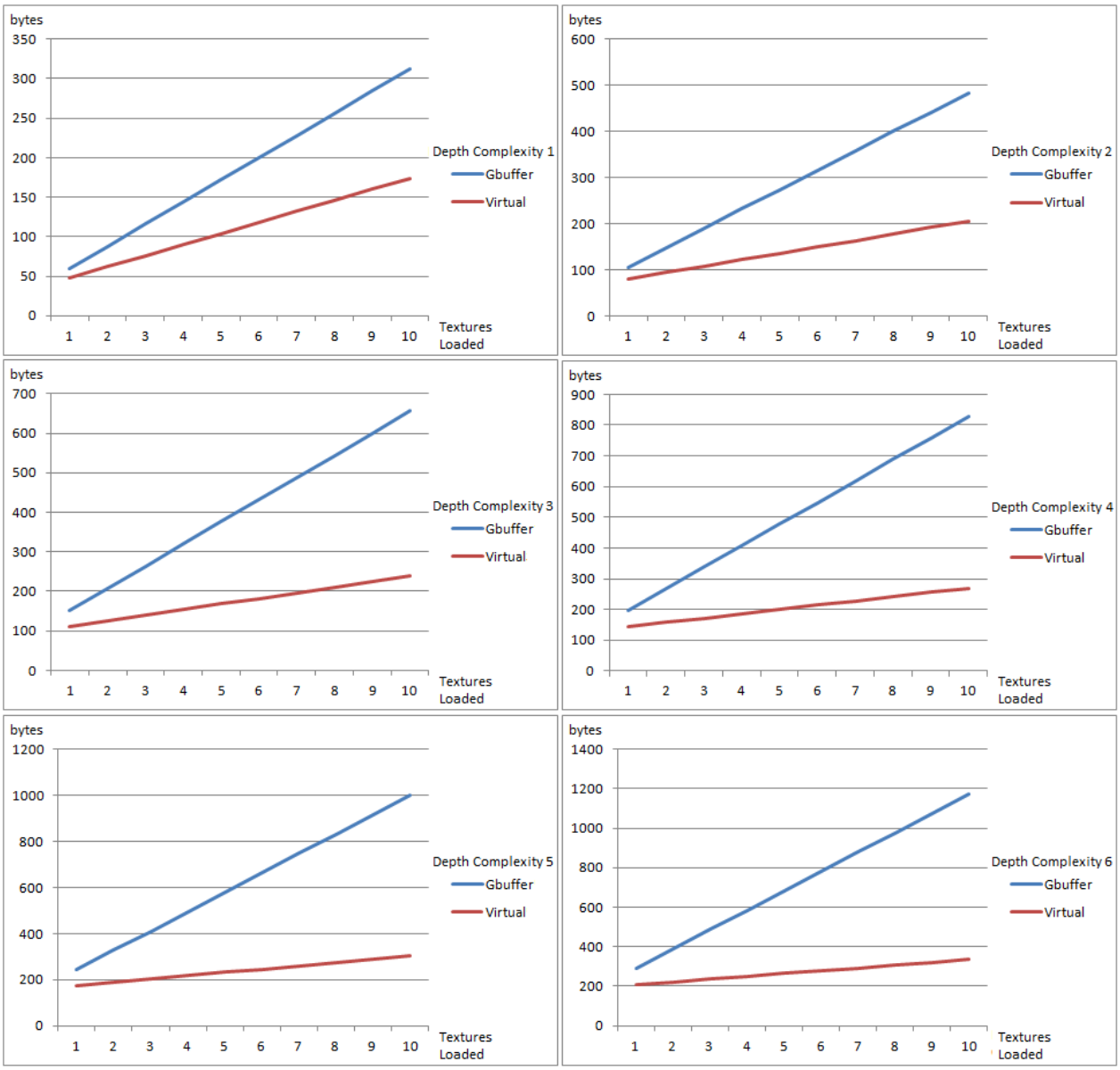

Figure 45 Virtual deferred Per Pixel Bandwidth Analysis**.** The image depicts a comparison in per pixel bandwidth consumption between virtual deferred rendering, colored in red, and single geometry pass deferred rendering, colored in blue. The measurements are directly performed in texture fetches per pixel, disregarding the number of materials, as that would further favorize virtual deferred. The difference between methods grows with the depth complexity, as single geometry pass deferred algorithms will consume more bandwidth on occluded pixels.

3.6. Transparent Rasterization

The rasterization of transparent objects has similarities and dissimilarities with the rasterization of opaque objects. The triangle projection process, the wrinkled surface methods and the material shading code are either identical or very similar. On the other hand the visibility determination process is different, caused by the lack of order in surface-camera interactions in rasterization.

Transparent object rendering is usually performed after opaque object rendering, in order to not pay the rendering costs for transparent objects that are occluded by opaque objects. Transparent rasterization uses the same displacement and parallax algorithms described in chapter 3.5.

The problem of transparent rasterization is very difficult in rasterization because transparency is implemented through the composition operator as defined by [**Por84**], but the rasterization process processes the surface-camera interaction in an unordered manner. If F_α is the fragment opacity, F_c is the fragment color, C_c is the composited color and B_c is the background color, then the composition operator is defined as in the following equation:

$$C_c = F_\alpha * F_c + (1 - F_\alpha) * B_c$$

The equation can be written in front-to-back [**Had06**] compositing order as:

$$\begin{cases} C_c = B_\alpha * (F_\alpha * F_c) + B_c \\ C_\alpha = (1 - F_\alpha) * B_\alpha \end{cases}$$

The equation can be also be written recursively as:

$$C_c = \sum_{i=0}^{Fragments} \left(\prod_{k=i+1}^{Fragments} (1 - F_{k_\alpha}) * F_{i_\alpha} * F_{i_c} \right) + \prod_{i=0}^{Fragments} (1 - F_{i_\alpha}) * B_c$$

Therefore the existing transparent rasterization rendering techniques either pay a very large rendering cost and render the image exactly [**Tar10**] [**Bav08**] [**Car84**] [**Bar11**] [**Mau12**] [**End10**] [**Sal11**] [**Sal14**], or resort to approximated methods that either approximate the composition operator [**Mes07**] [**McG13**] or completely redefine it [**Sin09**] [**Jan10**].

In this sub-chapter two new transparent object rasterization algorithms are presented. One of them is an exact order independent transparency rasterization algorithm which modifies the state of the art GPU implemented A-Buffer method [**Bar11**], applying virtual data principles to decrease bandwidth and storage consumption. The introduced method, virtual order independent transparency, distinguishes itself from other state of art algorithms by being a order independent transparency algorithm that shades and consumes bandwidth adaptively, stopping when the visual contribution decreases under a quality threshold.

The other one is an approximate method which enhances the state of the art occupancy maps with distributions, which artificially increase the occupancy map resolution and adapt to the depth configuration of the fragments which were rasterized on the pixel.

3.6.1. Virtual Order Independent Transparency

One of the most successful exact transparent rasterization solutions is the GPU implemented A-Buffer method [**Bar11**] [**Mau12**], albeit with extremely large storage and bandwidth costs. The A-Buffer method can be transformed into a high quality approximation method through the usage either stochastical storage [**Sal11**], or through guards, like in the "Guarded Order Independent Transparency" article [**Pet152**], but it then becomes prone to temporal artifacts caused by different approximations in consecutive frames which produce aliasing. Therefore, in real-time precision sensitive problems the original algorithm performs best.

While there are several types of high quality and low quality approximation methods, some based on the A-Buffer algorithm, the necessity for exact algorithms is easily shown through results comparison, as displayed in Figure 46.

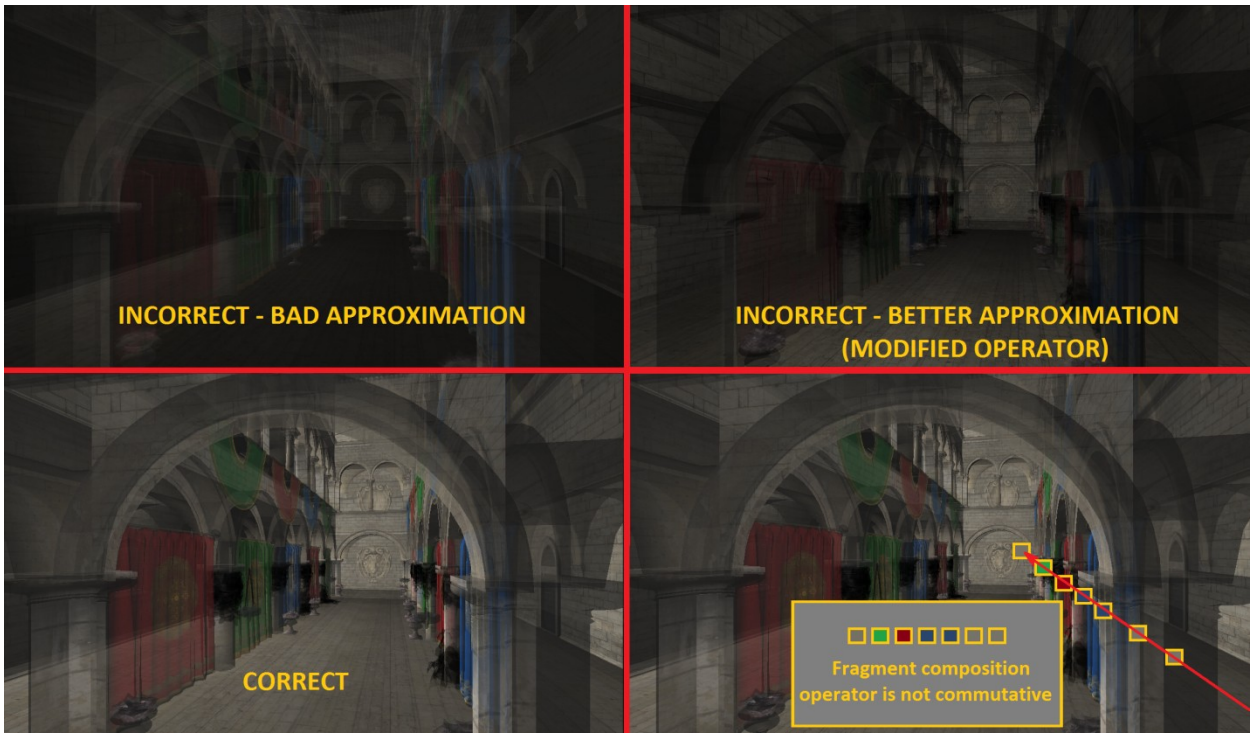

Figure 46 Order Independent Transparency**.** Transparent rasterization algorithms need to correctly handle the problem of surface-camera intersection order, as rasterization generates the intersection without order and the fragment composition operator used in transparency is not commutative. This requires either order approximation algorithms, which are cheap to compute but produce low quality results, or the correct per-pixel sorting of all the generated intersections. As it can be observed in the above figure, the difference in quality between a bad approximation and a better approximation algorithm is generally measured by the increased morphologic perception of the rendered scene. The difference in quality between a correct and approximated result can be drastic, as shown in the image.

A variation of the A-buffer algorithm is presented in this thesis. The **Virtual Order Independent Transparency A-Buffer** (VOIT or VA-Buffer) **decreases the excessive storage and bandwidth costs** of the state of the art A-Buffer algorithm. It uses the same virtual data principles which were applied to deferred rendering in the previous sub-chapter.

The A-Buffer is required to shade all the fragments generated through rasterization, before storing the color results in the A-Buffer list nodes. In comparison, VOIT decouples shading computation and bandwidth from the list construction, therefore VOIT adapts to the rendering situation of each pixel, consuming only what is necessary. After the per-pixel list is sorted, VOIT performs front to back composition and loads the texturing data for each node. When VOIT determines that the alpha channel is completely occluded it stops the rendering process, therefore it does not load bandwidth for nodes which are occluded, as would the A-Buffer algorithm. Because of this, VOIT is also superior from a software design perspective, as it is a decoupled solution to transparent object rasterization.

VOIT is visually presented in Figure 47.

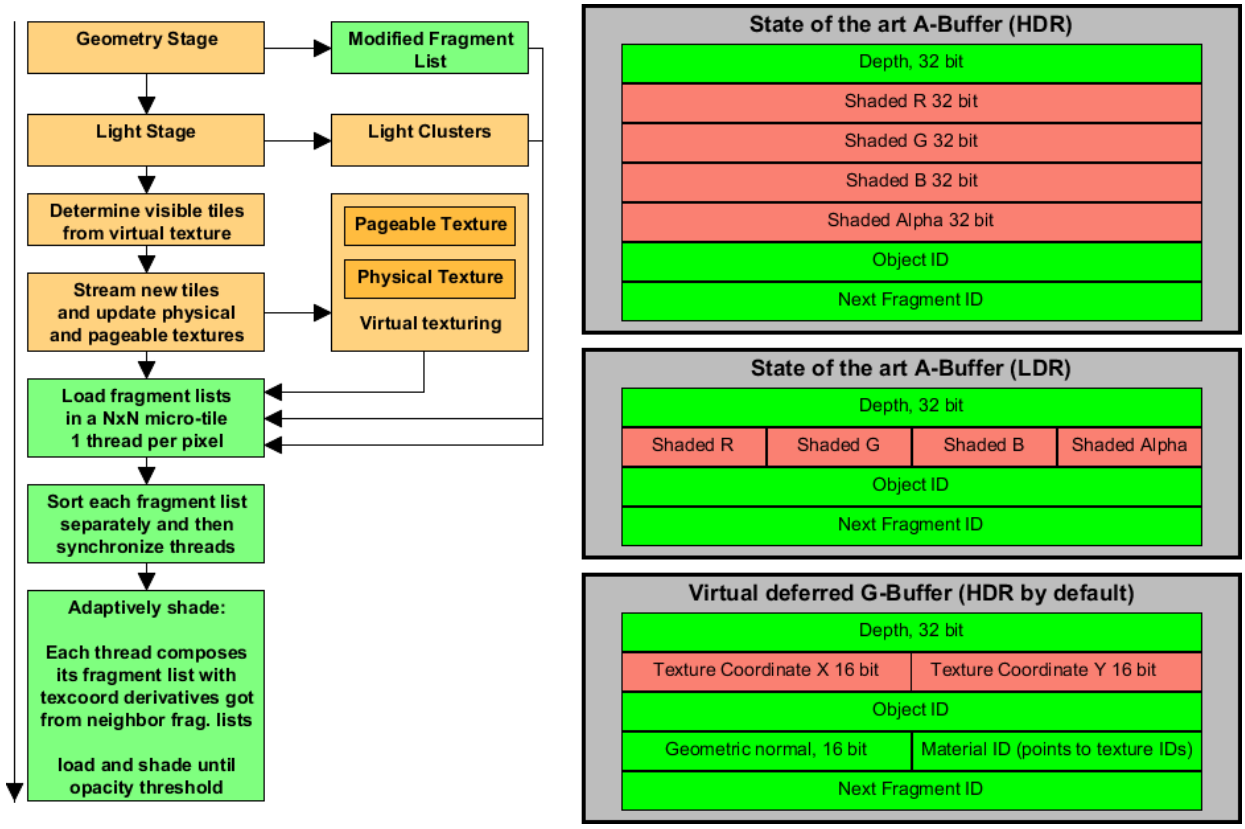

Figure 47 Virtual Order Independent Transparency - algorithm**.** The image presents the application of virtual data principles to the GPU A-Buffer algorithm, named Virtual Order Independent Transparency, or VA-Buffer. Compared to the state of the art A-Buffer the presented algorithm is designed to work with virtual texturing. The stages of the method are displayed on the left side of the image, the rasterization stages in orange and the shading stages in green. On the right side of the image VOIT is compared to A-Buffer. Instead of storing fragments with shaded color, as in a state of the art A-Buffer node list, the VOIT stores only texture coordinates. In the shading phase, the texture coordinates are used to reconstruct the texture coordinate derivatives, which are then used to perform the texel fetches. VOIT has the advantage of adaptively shading the fragments, stopping early if sufficient alpha occlusion is computed, and paying the texture bandwidth for exactly what it shades instead of paying it for all the fragments, as A-Buffer does. Compared to the state of the art A-Buffer nodes, the VA-buffer consumes slightly more storage than a low dynamic range node, but significantly less storage than a high dynamic range node. In this thesis the micro tile size is 2x2 pixels, the minimum size required to correctly reconstruct texture coordinate derivatives.

The introduced algorithm works on multiple stages, as depicted in Figure 47. The first stages work through rasterization, in which the geometry is rasterized and stored in the VA-Buffer specific node list.

Compared to the standard A-Buffer, **VOIT does not store the shaded colors** but the texture coordinates in compressed format. Because of this, the VOIT has better storage efficiency for high quality shading, where high dynamic range rendering is used. The next stage determines the visible textures and the required mipmap levels and sends streaming commands to the virtual texturing system. In order to prevent walking the entire fragment lists twice, once for shading and once for checking if the right texture mipmaps are loaded, VOIT uses a mipmap buffer, which stores the streamed state of all the scene textures, represented in binary. With each rasterized fragment, VOIT checks if the texture mipmaps that will be read are streamed. If the texture mipmaps are not streamed, the algorithm flags the texture mipmaps for streaming.

The shading part of the VOIT algorithm is presented in the Illumination chapter, the pseudocode for the rasterization based stages is:

```
(ONCE) PREPROCESS
IF lighting
        lightgrid ← allocate space for lights grid acceleration structure
nodebuffer ← allocate space for fragment list nodes
headbuffer ← allocate space for fragment head pointers
mipmapcachebuffer ←allocate space for mipmap streaming information (binary)
mipmapframebuffer ←allocate space for mipmap streaming information (binary)
FOR pixel in screenpixels
        set pixel head pointer to null

GEOMETRY STAGE (sceneobjects)
objects ← sceneobjects which are not culled
fragmentcounter ← 0
mipmapbufferframe← 0
FOR object in objects
        fragments  ← Ø
        FOR primitive in object
                fragments ← fragments  ∪ fragments from rasterized primitive
        FOR fragment in fragments
                atomically increase fragmentcounter
                node ← fragment depth, texture coordinates, other data
                store node in fragmentcounter position in the nodebuffer
                set node pointer to next element to null
                atomically swap head with node next pointer
                textures ← fragment textures
                FOR texture in textures
                        texturesderivatives ← texture coordinates
                        texturemipmap ← determine texture mipmap with texturesderivatives
                        texturemipmapstate ← read texture mipmap state from mip mipmapcachebuffer
                        IF texturemipmapstate unloaded
                                mipmapframebuffer← set texturemipmap has to be streamed
```

Exact order independent transparency is not mandatorily used with illumination, for example the geometry is rarely lit in real-time rendering performed for scientific visualization. This is valid because order independent algorithms are usually employed to render object design scenes in CADs. Therefore the following stage is optional.

```
LIGHTS STAGE(scenelights)
IF lighting
        FOR light in scenelights
                depth ← depth buffer from VIRTUAL DEFERRED GEOMETRY STAGE
                fragments ← render light, generate fragments, cull using depth
                FOR visible fragment in fragments
                        cluster ← determine location in lightgrid
                        cluster ← cluster∪ light
VIRTUAL TEXTURE STAGE (allscenemipmaps)
streamlist ← Ø
FOR mipmap in allscenemipmaps
        framestate ← state of mipmap in mipmapframebuffer
        cachestate ← state of mipmap in mipmapcachebuffer
        IF framestate AND NOT cachestate
                streamlist ← streamlist ∪ mipmap
FOR mipmap in streamlist
        stream mipmap from disk
        update virtual texture
        mipmapcachebuffer ← set mipmap
```

In order independent transparency problems texture streaming determination can't be determined like in deferred rendering algorithms, just by querying the geometry buffer. Querying all nodes would generate a very large cost in bandwidth, making the algorithm counterproductive. Streaming can be elegantly implemented with two **binary state buffers**, one which holds the state of the texture mipmaps for the current frame and one which holds the state of the texture mipmaps for the entire virtual texturing cache. The maps are queried and filled by each fragment during the geometry stage of the algorithm and are compared once in the texture stage, as shown in the pseudocode. The shading stage of the virtual order independent transparency algorithm is discussed here succinctly; it is presented in full detail in chapter 4.1.6.

```
SHADING STAGE
FOR microtile(2x2) in image
        allocate microtilecache
        FOR pixel in microtile
                list ← load the per pixel list into microtilecache
                FOR node in list
                        node ← owner by pixel
                        allocate extra space for texture coordinate derivatives
                sort list by depth
        synchronize microtile
        FOR pixel in microtile
                FOR node in list
                        derivatives ←  reconstruct derivatives from neighbors texture coordinates
                synchronize
                pixelocclusion ← 0
                pixelcolor← 0
                WHILE pixelocclusion < threshold
                        node ←  next in list
                        IF lighting
                                illuminate with lights from lightgrid
                        nodecolor, nodealpha←  load textures  and shade
                        pixelcolor, pixelocclusion  ←  compose with nodecolor and nodealpha
```

VOIT can benefit from the same optimizations as the A-Buffer, thus it can be implemented with compute indices in a geometry pass like in the transparency linked list offset array method [**Kno12**] or with micro-pages [**Cra10**]. Both methods improve data coherency. The offset array method uses a geometry pre-pass to compute the fragment list offsets for all the generated fragments, thus VA-buffer can be implemented with exact lists per pixel instead of a large list, guaranteeing data locality. The micro-pages method requires the implementation of a critical section and stores fragments in large nodes, called pages. Because of this the number of GPU links between the large nodes is much smaller than the normal number of links and data cache coherency is increased.

In high quality rendering scenarios, where high dynamic range rendering is used, VOIT also has superior storage and bandwidth consumption than the A-Buffer, as it is depicted in Figure 48.

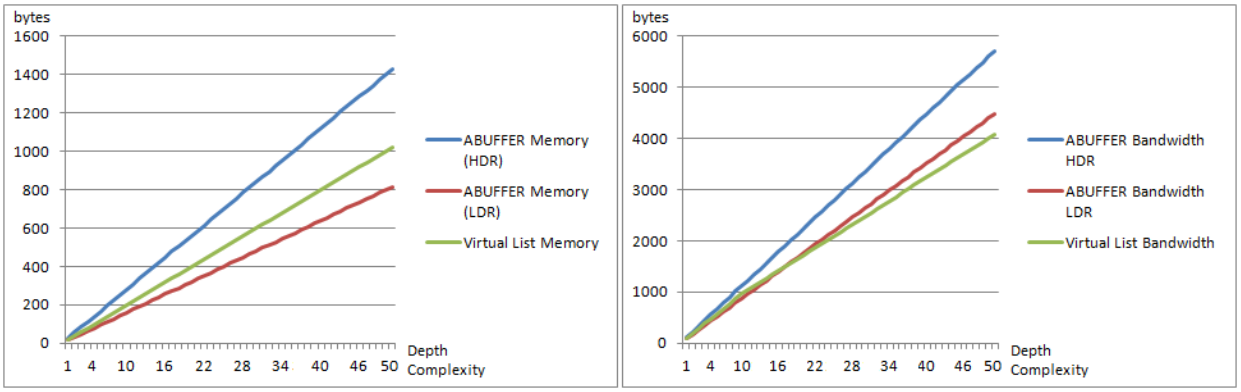

Figure 48 Virtual Order Independent Transparency – state of the art comparison**.** The image shows the bandwidth and storage usage of the state of the art A-Buffer and the proposed VOIT List method. The A-Buffer variants are displayed for both low dynamic range rendering (LDR) and high dynamic range rendering (HDR). While the presented method has a slightly worse storage consumption than the state of the art LDR A-Buffer it consumes the least bandwidth.

The presented virtual order independent transparency algorithm completely decouples shading from geometry processing while processing the geometry only once. It has comparable and results to the LDR variant of the A-Buffer, while offering HDR rendering quality. Furthermore, the shading computation scales better than the A-Buffer because VOIT adapts to the pixel opacity distribution.

A weakness of the presented algorithm is that objects must be textured with a single texture mapping, as with other virtual data methods. The algorithm gracefully resorts to a simple colorless A-Buffer in the case of real-time scientific visualization, where objects are rarely textured, and the scene object components each have colors which encode their usefulness. In this case the method only stores either the object id or the material id, as this information is sufficient to determine the color in the reconstruction. Texture coordinates and their derivatives are not computed, as they are not needed.

The presented algorithm processes and loads textures adaptively, only paying for operations which are guaranteed to have a visual impact in the final image. Because of this, VOIT is a bandwidth and cost efficient solution for detailed visualizations in real-time.

A real-time scientific visualization is displayed in Figure 49, in which the importance of order independent transparency is emphasized.

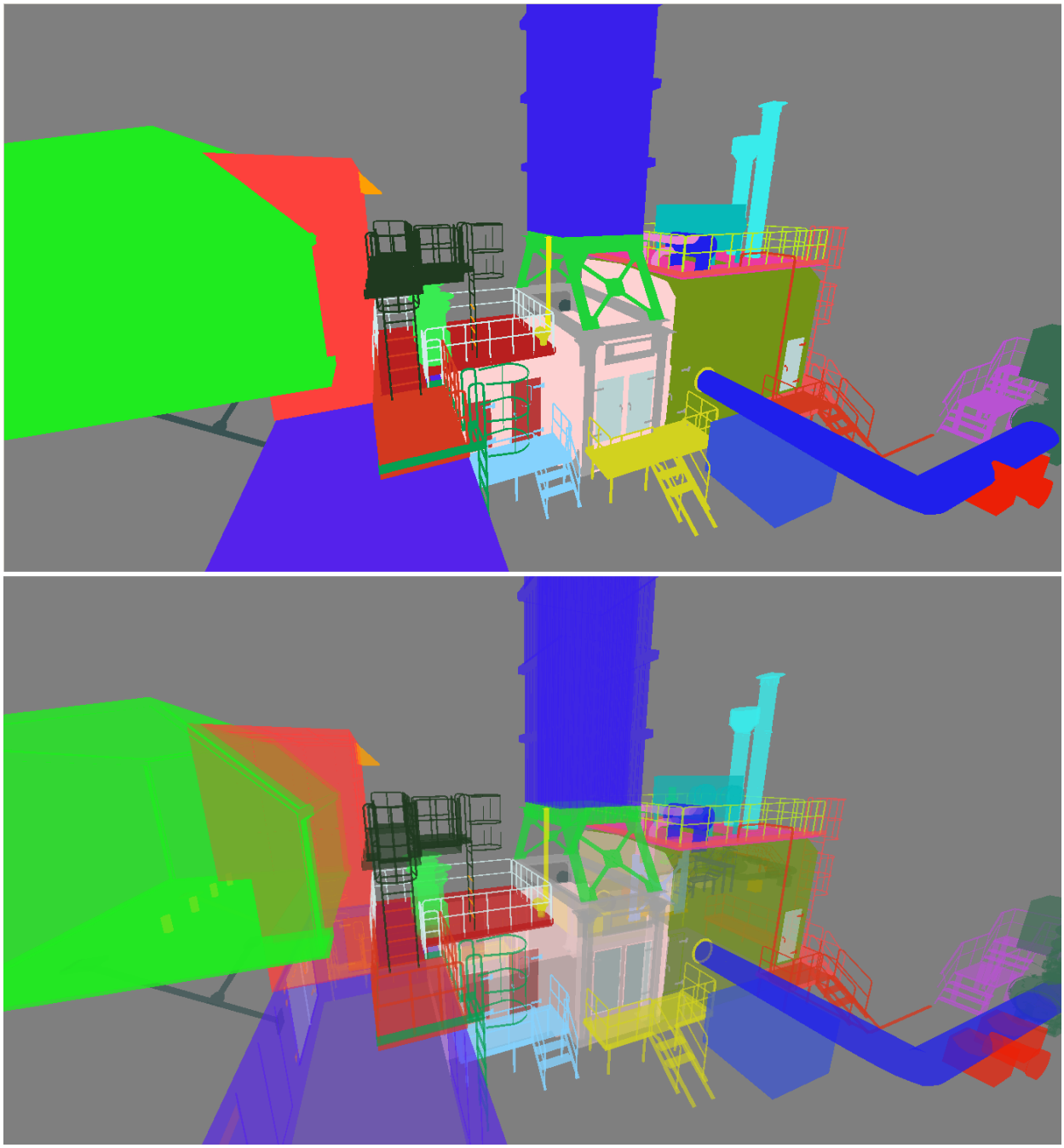

Figure 49 Virtual Order Independent Transparency – applications**.** Order independent transparency is particularly useful in scientific visualization. The image above shows the difference in visualization quality of a technical scene. The upper part of the image is normally shaded, while the lower part uses correct order independent transparency rendering. The second image is superior from a CAD point of view, as it permits more accurate design, and a richer perception of the objects with which the scene is modeled.

3.6.2. Distribution Occupancy Maps

The state of the art approximated solutions for transparent object rasterization are not good enough to render complex, non-uniform objects. Because such methods either approximate or redefine the composition operator they are either too inexact or too rigid for complex rendering scenarios. In general these methods are used to render low-frequency geometry, such as fuzzy objects, because they are able to perform this rendering operation with acceptable visual results and at a fraction of the cost that comes with exact solutions.

The most relevant methods that can produce high quality visual results are occupancy maps and Fourier opacity maps. Both these methods approximate the depth distribution per pixel, and use this distribution to redefine the composition operator. In the case of Fourier opacity maps, the algorithm approximates the depth distribution by analyzing opacity as signal varying on the depth axis. The signal is Fourier transformed and the Fourier coefficients are used to approximate opacity in a second and final geometry pass. In the case of occupancy maps there is an additional assumption that opacity is a constant α, with which the composition operation is changed into an opacity function over depth, which is commutative. As given by [**Sin09**] the composition equation changes to:

$$C_c = \sum_{i=0}^{Fragments} (1-\alpha)^i * \alpha * C_{Fragments-1-i} + (1-\alpha)^{Fragments} * B_c$$

This assumption can be used in real-time applications where there are many transparent objects that have to be rendered, but the rendering quality can be less accurate and all the rendered elements have similar opacity. These properties are common fuzzy, transparent objects such as smoke, clouds or liquids.

Based on the same assumption this thesis introduces an improved variant of the occupancy maps, distribution occupancy maps. The method uses per-pixel **distributions in order to make occupancy maps adaptive**, by altering the occupancy bits resolution. Instead of using a uniform distribution for all pixels, several simple uniform and multi-pole Gaussian distributions are used. Because of the altered distribution of samples, the resolution is artificially increased and the quality of opacity measured with occupancy maps increases.

Distributed occupancy maps use an additional buffer, the distribution buffer, which stores occupancy in a uniform depth distribution, like a very low resolution occupancy map. The algorithm stores this additional buffer for the current and the previous frame. The bits of this distribution buffer are used to partition the full resolution occupancy map adaptively, based on the depth distribution of the pixel fragments. Each partition of the full resolution occupancy map uses a different, local, sampling strategy. The used sampling strategies are based on distributions: uniform, multi-pole uniform, Gaussian, multi-pole Gaussian, sigmoid and multi-pole sigmoid distributions. For an 8 bit buffer the sampling offsets of these distributions can be easily precomputed and stored in small table, which resides GPU memory.

The basic idea of the distribution occupancy maps is presented in Figure 50.

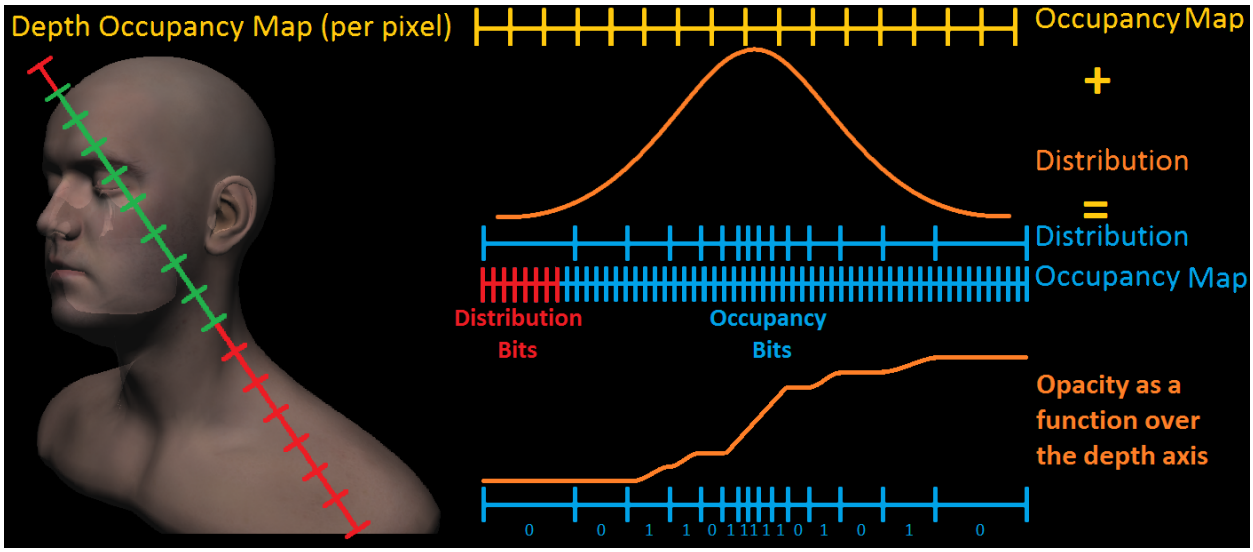

Figure 50 Distributed occupancy maps**.** Distributed occupancy maps enhance the state of the art occupancy maps by adapting the occupancy map sampling process to the distribution of rasterized fragments on the depth axis. The transparent composition is performed with the commutative opacity operator, which queries a function over the depth axis of the fragment, as shown at the bottom of the image. Compared to occupancy maps, distributed occupancy maps are more adaptable and use the samples more efficiently, therefore they produce better looking results in non-uniform depth distributions while demoting to simple occupancy maps in the case of a uniform depth distribution..

Before rendering, the algorithm allocates four buffers: two depth distribution buffers, one for the current and one for the previous frame, the occupancy buffer and an offset map. The occupancy buffer stores the occupancy on the depth axis of the pixel. The occupancy samples are controlled by the depth distribution from the previous frame, through the offsets read from the offset map, which stores data in the [0, 1] interval.

The two depth distribution buffers store the depth distribution bits for the current and previous frame, because the presented algorithm uses one of the depth distribution buffers to describe the current frame but uses the previous frame distribution buffer to query the offsets at which it stores occupancy. A depth distribution buffer can be considered as a minuscule uniform occupancy map, whose bits can be considered to represent an index. The index can be used in the offset map to query the exact offset needed for the occupancy samples.

The offset map stores the offsets for the occupancy sampling points, for each depth distribution. The offset map is divided into zones, one for each depth distribution area. Each zone contains the offset for each occupancy sample, transforming the uniform distribution into a non-uniform one.

The algorithm runs in two passes, the geometry pass and the shading pass, and it is a coupled algorithm, which couples shading, visibility determination and texture fetching.

In the geometry pass the geometry is normally rasterized in order to obtain the fragment depth. The fragment depth is used together with the previous frame depth distribution to properly insert into the occupancy buffer. The current depth distribution buffer is used as a very low resolution uniform occupancy map.

In the rendering pass the geometry is rendered once more and the fragment colors are computed based on the occupancy samples from the previous geometry pass, using the depth distribution buffer from the previous frame. T

The outputted fragments are blended additively. The pseudocode for the proposed algorithm is the following:

```
(ONCE) PREPROCESS
allocate the two depthdistributionbuffer, with resolution N_distrib (e.g. N_distrib = 8 bits)
allocate the occupancybuffer, with resolution N_occupancy (e.g. N_occupancy = 128 bits)
allocate an occupancyoffsetbuffer, with size = 2^N_distrib * N_occupancy * 4
allocate offsettmap, with sampling offsets for all depth distribution permutations
FOR permutation of 2^N_distrib bits of depthdistributionbuffer
        clusterid[N_distrib] ← 0
        FOR set bit in permutation
                i ← bit index
                clusterid[i] ← i
        change ← true
        WHILE change
                change← false
                FOR set bit in permutation
                        localchange ← true
                        traversalbit← bit
                        WHILE localchange
                                localchange← false
                                leftbit ← left neighbor bit of traversalbit
                                IF leftbit set AND clusterid [leftbit] != clusterid[bit]
                                        clusterid[leftbit] ← clusterid[bit]
                                        localchange , traversalbit ← true, leftbit
                        traversalbit← bit
                        WHILE localchange
                                localchange← false
                                rightbit ← right neighbor bit of traversalbit
                                IF rightbit set AND clusterid [rightbit] != clusterid[bit]
                                        clusterid[rightbit] ← clusterid[bit]
                                        localchange , traversalbit← true, rightbit
        N, index← 0
        WHILE index < N_distrib
                IF clusterid[i] > 0
                        N ← N+1
                        WHILE clusterid[i+1] = clusterid[i]
                        N, i ← N+1, i+1
                        N ← N+1
        bounds[2N], resolution[N], distribution[N]← remaining unique clusters in clusterid
        FOR cluster in uniquecluster
                k ← number of unique cluster
                resolution[k]← N_occupancy / (bounds[2*k] − bounds[2*k+1]) * N_distrib
                IF bounds[2*k+1]-bounds[2*k] =1
                        distribution[k] ← Gaussian
                ELSE
                        cluster margins bits add sigmoid distributions over a single bit
                        all inner margin bits add one uniform distribution
        offsettmap ← compute offsets for all the occupancy samples
```

GEOMETRY PASS (objects)
fragments ← rasterize **primitives** from **objects**
FOR fragment over **pixel** in **fragments**
    distribution ← previous frame **depthdistributionbuffer,** for **pixel**
    uniformoccupancysample ← determine uniform occupancy sample
    occupancysample ← **uniformoccupancysample, distribution**
    occupancybuffer ← set **occupancysample**
    depthbit ← determine bit occupied by **fragment** in **depthdistributionbuffer**
    depthdistributionbuffer← set **depthbit**

SHADING PASS (objects)
set outputmerge to additive
fragments ← rasterize **primitives** from **objects**
FOR fragment over **pixel** in **fragments**
    distribution ← previous frame **depthdistributionbuffer,** for **pixel**
    uniformoccupancysample ← determine uniform occupancy sample
    occupancysample ← **uniformoccupancysample, distribution**
    opacity← compute opacity for the number of set samples in the **occupancymap**, before **occupancysample**
    fragmentcolor ← shade with **opacity**
    OUTPUT fragmentcolor
OUTPUTMERGE final **pixelcolor**
previous **depthdistributionbuffer** ← current **depthdistributionbuffer**
curent **current depth distribution buffer** ← 0

As all temporal coherent algorithms, distribution occupancy maps suffer from temporal artifacts. The algorithm is designed for low frequency geometry, where temporal artifacts are imperceptible. An example of a rendering with distribution occupancy maps is given in Figure 51.

Figure 51 Distribution Occupancy Maps Results**.** This approximated order independent transparency algorithm works best with low frequency geometry such as clouds, or smoke. Compared to the state of the art occupancy maps, the presented algorithm adapts its sampling to the depth configuration of each pixel, therefore it obtains the same results as occupancy maps for uniform depth distributions and better results for non-uniform depth distributions.

3.7. Atomic Geometry Selection

The following section is based on the "Efficient picking through atomic operations" article [**Pet13**]. It presents a selection algorithm, atomic geometry selection (AGS), which is capable of selecting any type of renderable geometry.

Selection rendering, also named picking, is the process through which a single entity or a list of entities is selected from a scene. The subject of picking is both a rendering and a collision detection problem, with the majority of research being on optimizing the ray-scene intersection problem.

Several algorithms that solve the selection problem exist in the context of rasterization, but all of them lack several of the features of this proposed solution while, with one exception, all being much more expensive in terms of computational time. The introduced method is able to correctly select **not only primitives but also any type of objects** that may appear on the screen at a fragment level including hardware instanced, alpha culled, hardware tessellated, hardware animated and fuzzy objects. The proposed technique has **optimal memory requirements** and offers the opportunity to select at micro polygon level and is not limited to the first contact, offering the full intersection list per ray if required to do so.

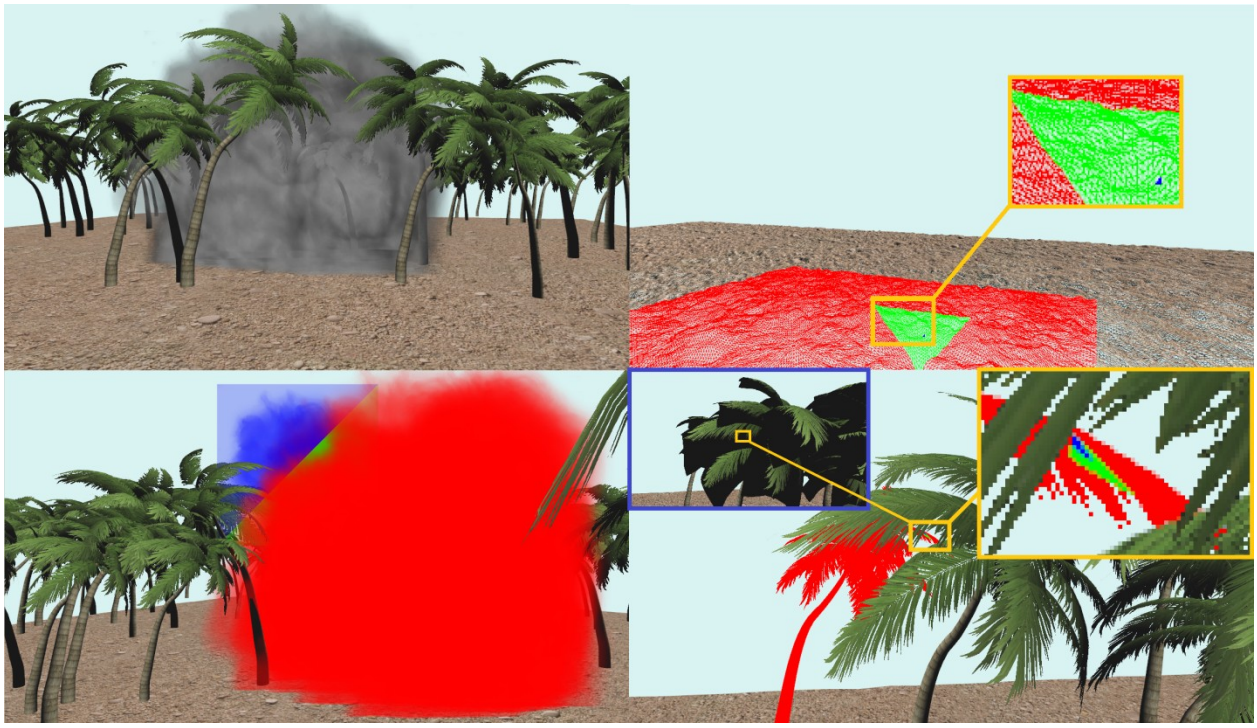

Figure 52 Object Selection**.** In the left upper corner a rendering scene is presented, which contains many difficult to select objects, such as the hardware instanced hardware tessellated ground, the alpha-culled palm trees and the fuzzy geometry of the smoke. The presented method can select all intersected objects rasterized over a fragment, but compared to all state of the art algorithms, it can account for alpha occlusion, thus it does not return the palm trees behind the smoke. Wrinkled surface geometry selection is presented in the right upper corner. In the left lower corner selection is performed on multiple billboards, returning a list of all the intersected surfaces. In the right lower corner alpha culled selection is presented. The tree support geometry is shown in the highlighted blue rectangle.

The proposed technique offers further unique opportunities such as flexible fuzzy object selection and **takes into account opacity accumulation** from transparent objects, in the case of

multiple transparent objects per fragment. Some of the most interesting selection cases handled the presented algorithm are shown in Figure 52.

The selection problem has been solved through geometry intersection [**Las03**], micro-raster rasterization [**Nei93**], color picking buffer methods [**Wri10**], geometry stream out [**Wri10**] and atomic selection [**Ric12**].

The geometry intersection method requires ray-scene intersection, which can solve multiple intersections per fragment but has the intersection complexity of $O(n \log n)$, where n is the number of primitives in the scene. This method does not work with fragment level effects such as alpha culling. The geometry intersection can also be implemented as depth picking over a deferred geometry buffer, as the depth stored in this buffer uniquely identifies the closest camera-geometry intersection, which lowers the intersection cost to $O(1)$ but enforces depth storage and returns only the closest surface-camera intersection, as there is no depth stored for occluded fragments.

Micro-raster rasterization renders the entire scene into a minuscule raster, which has a high geometry processing cost, because the scene has to be rendered multiple times. The atomic selection method can be considered as a one pixel micro-raster variant, which does not process the geometry multiple times. On the other hand the method does not handle multiple intersections per fragment.

Color picking buffer methods work by encoding the object id into a color representation. They use an additional screen output buffer, in which this encoding color is stored, and which can be used to uniquely determine the first camera-surface intersection point for each pixel. As the geometry intersection method, the color picking method can't handle multiple intersections per fragment.

The geometry stream out algorithm uses a buffer which is filled from the geometry shader in the stream out/transform feedback hardware rasterization stage. It can handle multiple intersections per fragment, but it does not work at fragment level even though it is a conservative method. Furthermore, the method can't handle fragment level effects such as alpha culling.

The main contribution of the presented selection algorithm is that it uses a atomically synchronized selection area buffer in which all the fragments that are rasterized over it store surface intersection points. The entire list is then sorted over the depth axis, in a process similar to that used in the GPU A-Buffer algorithm [**Bar11**].

The entire surface-camera intersection point saving process is performed at fragment shader level, thus the presented method can accurately detect pixel level visibility, which is very important for wrinkled surfaces and alpha culled geometry. Compared to the state of the art methods the presented method does not allocate more storage than what it needs, and processes only the relevant surface-camera intersection points without rendering the scene geometry multiple times. The algorithm also gracefully handles any geometry altering methods such as tessellated displacement mapping or instancing.

This selection and accumulation process is displayed in Figure 53.

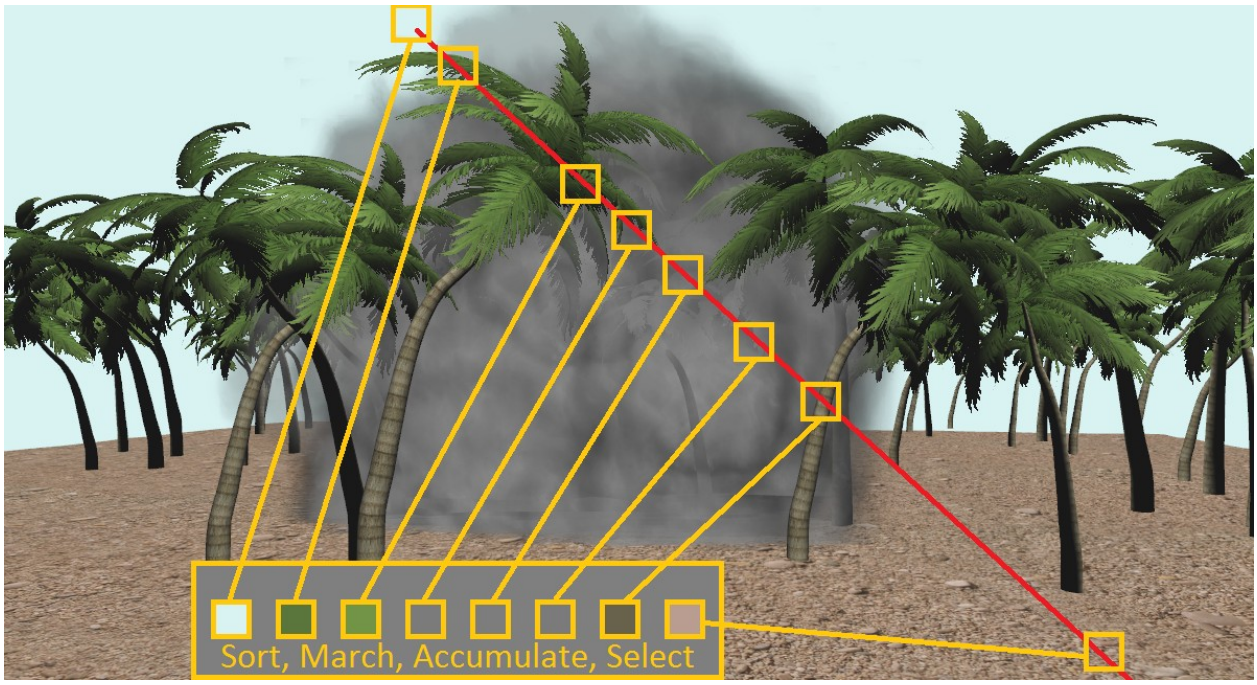

Figure 53 Atomic Geometry Selection Marching**.** The atomic geometry selection algorithm stores camera-surface intersection points per selection zone, which can be either a single pixel or a large area. The intersection points are stored in linked list, synchronized through atomic operations. The linked list is finally depth sorted and walked. The walk computes the accumulated opacity and ends as soon as the alpha channel is completely occluded.

The presented algorithm works on a selection zone, which can either be a single pixel or large number of pixels in a rectangle area. The algorithm starts at the vertex shader level, and sends further down the pipeline the object id, provided by the application and the vertex id and the instance id, which are freely provided by the rendering pipeline.

If the rendered geometry has tessellation stages, each micro polygon created with hardware tessellation receives a unique id based on a simple spatial hash, which is valid only for at primitive level. The spatial hash can be any sufficiently rare hash function that uses the tessellation barycentric coordinates (u, v, w) generated by the tessellator unit, for example:

$$hash(u, v, w) = a * u + b * v + c * w, where\ a \gg b \gg c$$

This unique id is then used to uniquely identify transient geometry, which is never permanently stored. Therefore the presented selection algorithm can select and work with sub-primitive geometry. The method uses a geometry shader to determine the rendered primitive id. The geometry shader sends the primitive id to the fragment shader, along with the vertex id, instance id and object id received from the vertex shader and, optionally, the unique id generated in the tessellation stage.

The fragment shader then uses all the ids to store the intersection point, along with depth and opacity of the fragment. The intersections points are stored in a buffer, named the picking buffer, which is per-selection zone linked list. Access to the linked list is synchronized through atomic operations.

The selection method ends with either a CPU or a compute shader stage, which sorts by depth and then walks the stored intersection points. Front to back composition is performed only to determine opacity. The walk ends as soon as the alpha channel reaches total opacity, thus the algorithm does not return occluded camera-surface interaction. This is the pseudocode for the atomic geometry selection algorithm:

```
INTEGRATED IN RASTERIZATION (selectionzone, objects)
FOR pixel in selectionsonze pixels
        count ← 0
        selectionlist← 0
FOR primitive in objects
        IF hardware tessellation used
                TCS ← ObjectId, InstanceId, VertexID←VS
                TES ← data ← TCS
                barycentric ← TES
                GS ← TransientID ← hash(barycentric)
        ELSE
                GS ← ObjectID, InstanceID, VertexID←VS
        FS ← PrimitiveID ← GS
        IF geometry shader instanced
                FS ← GSInstanceID ← GS
        IF fragment ⊂ selectionszone
                depth, opacity ← fragment depth
                data ← ObjectId, InstanceId, VertexID, TransientID, PrimitiveID, GSInstanceID, opacity
                IF fragment opaque
                        entry, entrydepth ← first entry in selectionlist
                        IF entry empty OR entrydepth > depth
                                save data to first entry in selectionlist
                ELSE
                        save data to count+1 index in the selectionlist
                        count ← count + 1

CPU/GPGPU (selectionzone, objects)
selected← Ø
FOR pixel in selectionzone
        opaqueentry ← first entry in selectionlist
        IF opaqueentry not empty
                selected ← selected ∪ opaqueentry
        ELSE
                list ← selectionlist, count for pixel
                sort list after depth
                fragmentopacity ← 0
                WHILE fragmentopacity < threshold
                        node ← next in list
                        opacity ← node opacity
                        fragmentopacity←accumulate opacity in front to back order
                        selected ← selected ∪ node
```

The selection algorithm is compared to the state of the art methods in Table 8. The algorithm combines all the strong features of the state of the art methods while still adding more useful properties such as micro primitive selection, fuzzy object selection, alpha occlusion awareness or zone selection.

Selection algorithm	Selection Algorithm Properties							
	Correct Picking	Alpha Occlusion Aware	Selects all per pixel intersections	Storage 10 fragments @1080p	Geometry passes	Hw. Instan-cing	Hw. Tessel-lation	Fuzzy Selection
Ray casted Geometry Intersection	no	no	yes	not applicable	not applicable	no	no	no
Depth Buffer Geometry Intersection	yes	no	no	66.35 mb	1	yes	no	no
Micro-Raster	yes	no	yes	320 b	2	yes	no	no
Color Selection	yes	no	no	49.76 mb	1	yes	no	no
Atomic Selection	yes	no	no	320 b	1	yes	no	no
Transform feedback	no	no	yes	320 b	1	yes	yes	no
AGS (this algorithm)	yes	yes	yes	320 b	1	yes	yes	yes

Table 8 Comparison of selection algorithms**.** The atomic geometry selection (AGS) algorithm combines the best aspects out of the state of the art methods, while also enabling transient geometry and alpha occlusion selection.

Because of the novel selection opportunities offered by the presented selection algorithm, it can be efficiently used in the wrinkled surface asset baking process, where it can displace the transient geometry of a tessellated mesh, which would then be directly saved in displacement maps. This is presented in Figure 54.

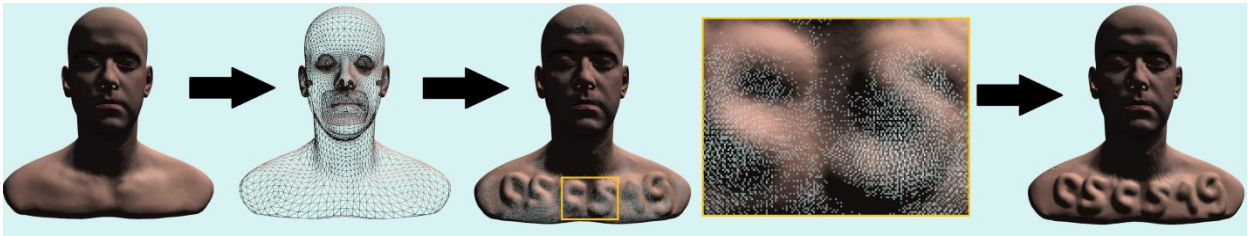

Figure 54 Other uses of selection**.** The selection algorithm can be used in the baking process of displacement maps. It can be used to load low resolution real-time assets, dramatically increase their primitive count dynamically through hardware rasterization and perform vertex displacement manually or with an intelligent tool. When the geometric displacement process is finished, the displaced transient geometry is saved directly into a displacement map. Thus, displacement map editing baking can be performed directly on the real-time assets.

4. ILLUMINATION

This chapter describes the second part of the proposed rendering pipeline, which computes global illumination and shading. The modules and algorithms presented in this chapter are succinctly depicted in Figure 55; the green modules represent thesis contributions.

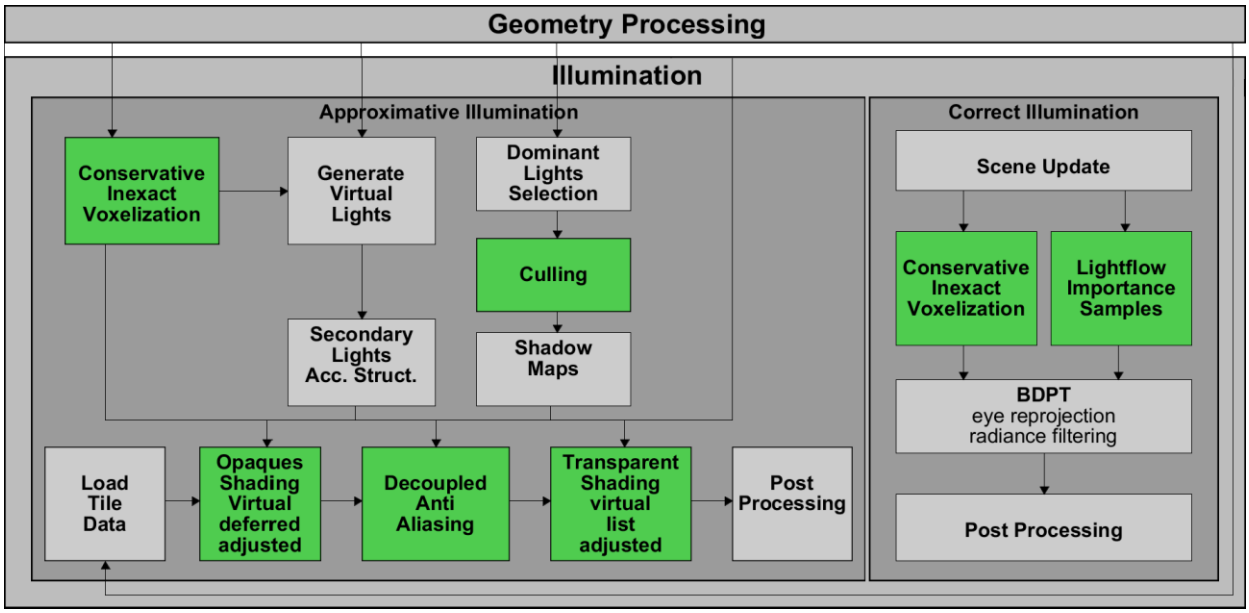

Figure 55 Illumination Overview**.** The chapter is subdivided into two large modules: Approximative Illumination and Correct Illumination. Approximative Illumination solves the global illumination problem through a combination of several rendering algorithms, using shadow mapping, virtual lights and screen space cone tracing, effectively decoupling light transport. The Approximative Illumination module contains many contributions. Conservative Inexact Voxelization is a fast voxelization algorithm designed for many/virtual light generation and shadowing. The shading of opaque objects is handled through the shading part of virtual deferred, a novel deferred algorithm which decouples texture bandwidth from visibility determination. Antialiasing for deferred rendering is done with an improved reconstruction method for the sub-pixel reconstructed antialiasing. The shading of transparent objects is computed with the shading part of virtual order independent transparency, a new bandwidth decoupled adaptive order independent transparency algorithm. In contrast to the Approximative Illumination module, the Correct Illumination module solves the illumination problem with a monolithic path tracing framework, accurately simulating light transport. The module renders images with a bidirectional path tracing algorithm which uses amortized visibility which lowers to cost of tracing rays. The bidirectional path tracer adapts the conservative inexact voxelization algorithm to path tracing, using it to trace fast conservative inexact rays, which can be then used to prevent the tracing of exact rays. The algorithm also uses a new type of importance sampling, Light Flux Importance Sampling, which generates a map of all the flux of light of the scene. This map is then used to quickly guide paths to vertices from the light tracing pass of the bidirectional path tracer.

The chapter is subdivided into two large modules, which tackle the global illumination problem with different approaches: the Approximative illumination module runs in real-time and the Correct Illumination module runs offline. It can also run at a very low number of frames per second, at the edge of interactivity, if computed on the GPU. Except for the shadow mapping algorithm used in the Approximative Illumination module the entire Illumination pipeline is not dependent on rasterization.

The Approximative Illumination module tackles global illumination from a real-time standpoint, using approximated visibility operators for light propagation. The module uses the inputs created by the Geometry Processing stage, such as the virtual deferred geometry buffer and the virtual order independent transparency per-pixel lists. The exact connection between the Illumination pipeline and the Geometry processing pipeline is shown in Figure 2. The illumination process is decoupled into multiple algorithms and rendering paths, each of which handles specific light paths. While the visual results are pleasing and close to photorealism a large number of light paths, the Approximative Illumination module can't simulate difficult types of light paths, for example light paths with multiple highly specular surface interactions.

The Approximative Illumination module approximates the potential impact of the scene lights received from the geometry processing pipeline and selects a few of them, which are considered dominant lights. Shadow maps are rendered for each of the dominant lights, maximizing the quality of the visibility operator for easy to perceive illumination. Each shadow map rendering uses the hierarchical culling algorithm presented in Chapter 3.4. The rest of the illumination is decoupled between low frequency (diffuse) and high frequency (specular).

A new light transport method is presented, which accelerates low frequency lighting by approximating the visibility operator through conservative inexact voxelization (CIV). CIV is an object-level acceleration structure which can be quickly created and used to determine a fairly accurate geometric composition of the scene. This geometric representation can be then used to trace visibility determination rays for the random walks used by the virtual light generation system, which generate virtual lights. The virtual lights are stored into a cluster acceleration structure. The CIV structure is then used to trace the visibility determination rays for the secondary lights.

Specular light transport is notoriously difficult to simulate correctly in real-time. Correct specular transport requires visibility determination mechanisms which work at high frequency, such as rays, paths or photons. These high frequency operations generate incoherent memory walks which greatly slow the rendering process, especially for streaming many-core architectures such as GPUs. While there are solutions for high quality approximations for specular transport in real time [**Cra14**], they require total data control, which in turn leads to resolution problems since volumetric solutions can't express detail as efficiently as analytical solutions, leading to impractical storage requirements. Hybrid solutions [**Mar14**] [**Her14**] hold the most innovation promise since they reconstruct high frequency signals from cheap to obtain or already existing data, without needing further geometry processing, which is extremely expensive in large real-time dynamic scenes.

Shading is performed in a tiled manner and there are separate shading phases for opaque and transparent objects. For each shading phase, each micro tile 2x2 pixels loads data for a small number of GPU threads.

In the opaque shading phase the tiles load the modified G-buffers created by the virtual deferred algorithm, a novel deferred method presented in the Geometry Processing pipeline, chapter 3.5.2. The modified G-buffers are then intersected with the clustered lights and shading is performed over texture data read through virtual texturing.

In the transparent shading phase the tiles load the modified A-buffer lists created by the virtual order independent transparency algorithm, a new transparent rasterization method presented in the Geometry Processing pipeline, chapter 3.6.1. The modified A-buffer lists are then intersected with the clustered lights and composited. The composition is performed front to back and it ends as soon as the alpha channel is completely occluded, thus, the shading is computed adaptively. The texture data read for shading is also adaptively loaded through virtual texturing, paying the bandwidth cost only for contributing modified A-Buffer nodes.

The module contains a novel, decoupled antialiasing algorithm, specialized for deferred renderers. The algorithm is named decoupled sub-pixel reconstruction antialiasing (DSRAA) and is an improvement over the state of the art sub-pixel reconstructed antialiasing (SRAA) [**Cha11**]. It requires only a minor integration step in the geometry processing pipeline, where it generates multiple depth samples per pixel, for which the depth test is run. It can optionally multisample normals for an even higher quality reconstruction. The method integrates in the geometry processing pipeline without affecting other The reconstruction phase of the algorithm uses neighbor matching, in order to more accurately filter the unshaded depth samples.

The Approximative Illumination module ends with a short post processing phase, which is shared with the Correct Illumination module.

The Correct Illumination module tackles global illumination from a correctness standpoint. The module uses special acceleration structures, an exact light transport mechanism and only uses approximations to accelerate the sampling processes. In contrast to the Approximated Illumination module, the correct illumination module is monolithic and coupled, solving the illumination problem in a single, consistent mode. The visual results are exact; the precision is limited only by the number of used samples.

The Correct Illumination module can run on both CPU and GPU, but it is not a real-time module, and it is only presented as a visual reference generator. The presented algorithms will run in real-time with better consumer hardware.

Since the module performs exact tracing it needs a different type of acceleration structure than the one used in the real-time path, therefore a Bounding Interval Hierarchy has to be maintained over the scene geometry. The cost of tracing rays is amortized with a modified conservative inexact voxelization structure. The module renders images with the bidirectional path tracing algorithm, with different types of importance sampling, including the novel scene-wide light flux sampling algorithm. Compared to the state of the art, the light flux algorithm approximates light flux over the entire scene and it used to guide unproductive paths to light vertices produces by light paths. In essence, light flux is a global sampling mechanism which is faster, more efficient from a storage standpoint and easier to implement than other high energy importance sampling mechanism such as Metropolis [**Vea97**], Energy Redistribution [**Cli05**] or Skeleton importance [**Bir12**].

The Correct Illumination module ends with a short post processing phase, which is shared with the Approximative Illumination module.

4.1. Approximate Illumination Stage

The approximate illumination stage is a perception based rendering approach to global illumination which seeks to render photorealistic images without any preprocessing. It uses different approximative visibility determination operators for indirect diffuse light transport and indirect specular light transport and accurate visibility operators for direct light transport, maximizing the efficiency of easy to perceive light transport. The greatest weakness of the approximate illumination pipeline is the light transport over complicated specular light paths, which is a common theme in real-time render, as these paths need accurate visibility operators which can't be implemented without ray tracing, path tracing or photon mapping, which are far from being acceptable real-time rendering solutions.

The presented method is based on the many lights rendering paradigm. While the paradigm supports accurate specular transport [**Sim15**], it does so at a cost which rivals ray tracing, path tracing and photon mapping. Specular light transport has been solved through sparse voxel cone tracing [**Cra09**], but it only works properly on extremely high resolution voxel representations which require extensive pre-processing, and such constraints are impractical for real-time rendering of massive dynamic scenes. Thus, the maximization of the most important rendering features from a perception standpoint [**Wat13**] [**Sha73**] indicates that hybrid solutions offer the best results in real-time.

The rendering solution combines rasterization through the shadow maps used the for direct visibility of dominant lights, many light methods for the secondary lights and the generated virtual lights, approximate ray tracing for the shadows of secondary lights and approximate cone tracing for indirect specular transport. Much of the approximated visibility operations are implemented through a novel acceleration structure named Conservative Inexact Voxelization (CIV), which is created from the bounding boxes of objects instead of the geometry of the objects and is thus extremely fast to compute. The many lights are generated with random walks inside the CIV. The shadows for secondary lights and the virtual lights are computed by tracing inside the CIV. The specular light transport is screen space cone tracing and CIV is used to augment the algorithm in many of its failure cases.

The approximate rendering pipeline first computes light transport for dominant lights, with state of the art shadow map techniques. Then the conservative inexact voxelization is performed for the scene objects. Secondary light sources are used to transport light, through the CIV visibility operators. Through ray traced random walks the pipeline generates thousands of VPLs.

4.1.1. Light transport for Dominant Lights

The dominant lights are solved with shadow mapping because it benefits from the coherency of rasterization and thus provides the highest quality visibility operator that can be used for real-time rendering. While rays can also be used for the visibility determination operator, they are inherently incoherent and the cache access penalties decrease the rendering speed too much. Tracing into a coherent structure like down in sparse voxel cone tracing [**Cra09**] needs a very detailed acceleration structure, which makes the storage costs impractical for high quality rendering. The voxel space structure is also a bad fit for dynamic objects, which means that the structure has to be recomputed per frame. Because of these reasons shadow mapping is the most cost effective high quality direct visibility determination process. There are many approaches to

shadow mapping: precision warping methods [**Sta02**] [**Wim04**] [**Mar04**] [**Kol12**] [**Llo06**] [**Ros12**], partitioning [**Dim07**] [**Zha06**] [**Lau11**], adaptive sampling [**Fer01**] [**Gue07**], analytical reconstruction [**Dai08**] [**Ros12**], volumetric [**Lok00**] [**Pag04**] [**Yuk08**] [**Sal10**] [**Kim01**] [**Jan10**] [**Sal101**] [**Che11**], temporal [**Sch13**], ray traced [**Sto15**] or distance field based [**Zho05**].

The presented rendering pipeline uses cascaded ray-trace shadows, which are filtered in a screen space pass [**Bag10**] with altered kernel size as introduced in [**Ran05**]. The solution renders the scene from the light point of view and instead of storing depths, it stores the indices of the primitive from which the depth is normally stored in classic shadow mapping [**Wil78**].

In the rendering pass the primitives in neighborhood around the pixel are streamed and filtered, based on the depth comparison test from the reconstructed primitive position and the current fragment position. This test is performed in a vicinity, with a kernel identical to that defined [**Ran05**], and the results are then combined into a weighted average. This enables high quality shadow generation and requires the same amount of samples. While the technique can suffer from aliasing, this can be mitigated through rendering with level of detail geometry or with and hierarchical impostors. Aliasing can also be mitigated through multisampled ray traced shadow maps, which can reconstruct the depth more accurately. The algorithm is shown in Figure 56.

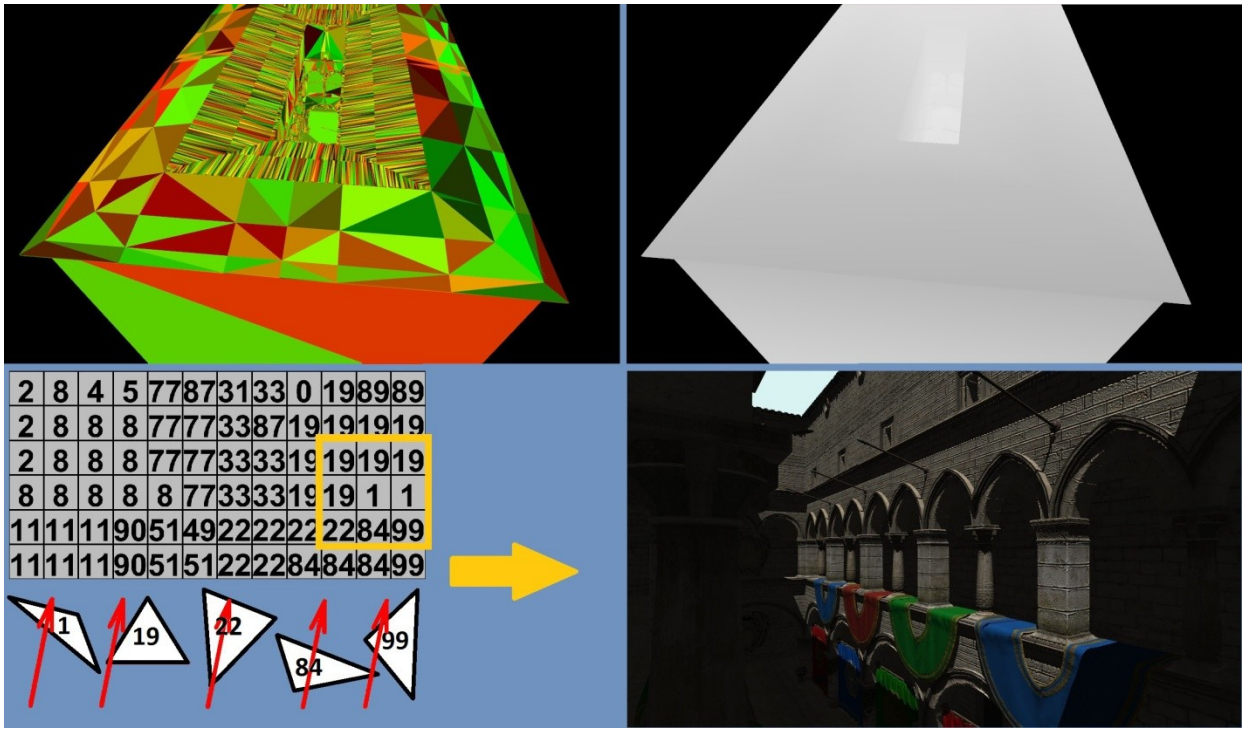

Figure 56 Ray traced shadow maps**.** The shadow maps do not store depth, but the primitive ID of the primitive whose depth would normally reside in the shadow map, as shown in the upper left corner of the image. The colors are generated from a hash of the primitive id. The lower left corner of the image shows the reconstruction process. Each primitive is then loaded into memory and the depth is computed analytically for each primitive, through camera ray – primitive intersection. The analytically reconstructed depth is also filtered with PCSS and it shows little alias, as can be seen in the final result, shown in the lower right corner and in the reconstructed light depth view, shown in the upper right corner.

4.1.2. Conservative Inexact Voxelization

Inexact representation of objects is very useful in rendering because it enables fast approximative representations of scene data. This principle has already been explored many times in global illumination, because it decreases the cost of the visibility determination operation, which in turn speeds up rendering. Inexact object representation has been done with point based rendering [**Rit08**], point based reconstructions and convex hulls, screen space voxelization [**Eis08**], imperfect voxelized shadow maps and volumes [**Wym13**], dense point clouds as imperfect volumes [**Mav11**] and sparse voxelization [**Cra09**]. The majority of these methods are or can be considered to be voxelization variants, being either bidimensional voxelizations such as the imperfect shadow maps method [**Rit08**] or tridimensional voxelizations. In the case of the bidimensional algorithm it has to be performed per light.

The problem with these methods is that the cost of voxelization is still very high, and the voxelization has to be computed for each frame in dynamic scenes. These reasons make voxelization based methods either barely interactive in the context of a large rendering pipeline or totally dependent on heavy preprocessing, which quickly becomes useless for dynamic environments.

In this thesis a new voxelization method is presented, which differs from the state of the art methods in several ways: it is a much a less exact but conservative representation and it is extremely cheap to compute.

This method, named Conservative Inexact Voxelization (CIV), is intended to be used with virtual lights methods, as presented in the 4.1.3 Chapter, where the inexact representation of scene geometry can be used in the transfer of low frequency (diffuse) light. There CIV is used to relax the cost of the visibility operator, speeding up the demanding task of virtual light generation.

CIV takes a different approach to voxelization, being a top to bottom algorithm. This choice is responsible for the speed of the algorithm as it guarantees an extremely **low complexity**, $O(num\ objects)$, compared to the state of the art complexities of $O(num\ primitives)$ or $O(num\ vertices)$. The method also uses all the existing scene geometry data usually found in real-time deferred rendering, **back projecting** information from the geometry buffer into the inexact voxelization, in order to maximize the precision for the visualization frustum. CIV uses hierarchical impostors, such as the ones presented in Chapter 3.2.3, but it can be adjusted to work without them.

The main idea of CIV is create a high resolution hierarchical representation of the scene geometry, highly accurate for data inside the visualization volume and inexact for the rest of the data. This is stored as a mipmapped tridimensional texture.

The algorithm runs in multiple stages: culled object determination, culled object voxelization, depth buffer reprojection and mipmap population. In the culled object determination stage, the algorithm runs hierarchically through the objects of the scene, a step which can be integrated with the culling method presented in Chapter 3.4. The algorithm walks the scene tree and inexactly determines which objects or scene nodes are to be voxelized, terminating the walk based on the distance from the camera. Only the scene nodes and objects outside of the visualization volume are voxelized in this step. This can be determined by

hierarchical depth buffer culling. The results can be stored in a draw list, like the ones used for transparent or opaque rasterization in the culling chapter.

In the culled object voxelization stage the algorithm expands each element to its axis aligned bounding box saves a single entry in the CIV texture, in the mipmap which is closest in size to the size of the axis aligned bounding box. Thus, the entire bounding box is added to the CIV texture highest resolution mipmap that encases it. In the case of highly elongated objects, in which the object sizes are unevenly long, the voxelization algorithm uses a geometry shader to dice the original bounding box into multiple smaller boxes, which are then saved into the CIV mipmaps. The impostors of important objects can have their depths projected on the boundary of the CIV texture. The CIV texture can represent the existence of objects in binary, thus even a large resolution CIV texture requires only little storage.

In the depth buffer reprojection stage, the depth buffer is used as a geometry information source, and all the depth buffer entries are back projected inside the CIV texture, in the highest resolution mipmap.

The mipmap population stages use a tridimensional push-pull process in which the mipmaps are populated based on the already stored information. This produces accurate results inside and in the vicinity of the visualization volume, and inexact results further away, making CIV an ideal solution for low frequency light transport through approximative ray tracing of shadows and virtual light random walks. The algorithm is visually presented in Figure 57.

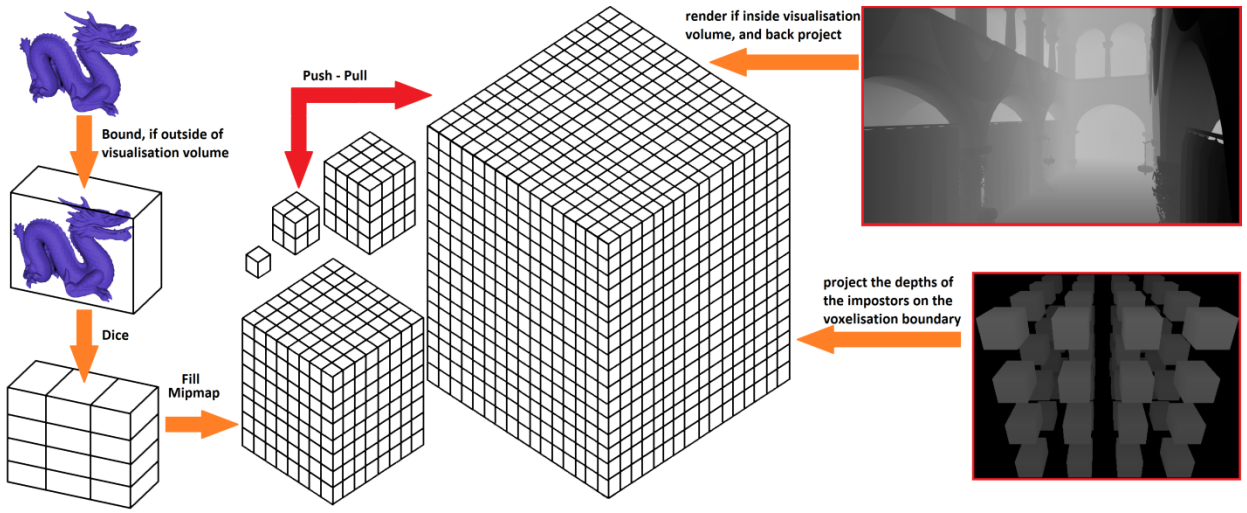

Figure 57 Conservative Inexact Voxelization**.** This image shows the construction of a Conservative Inexact Voxelization (CIV), an inexact representation of scene geometry. The method uses the depth buffer data to guarantee a highly accurate representation for the geometry inside the visualization volume while using a fast inexact bounding box dice and voxelize process for objects outside of the visualization volume. Each bounding box is diced if it has uneven sizes. Each of the diced bounding boxes is then saved as a single entry, in the highest resolution mipmap which completely encases it. Because of this CIV construction is very fast, when compared to the state of the art methods. CIV can also make use of hierarchical impostors.

The CIV texture is then used as a ray tracing visibility determination support, which can use ray tracing over voxels optimizations such as empty space skipping. The pseudocode for CIV construction is the following:

```
PREPROCESS
allocate texture with mipmaps for CIV
reset all data before each rendered frame

CULLED OBJECT DETERMINATION
currentnode ← root
walkqueue  ← root
voxelizationqueue ← Ø
WHILE walkqueue  not empty
        node ← pop walkqueue
        IF node inside the visualization volume (cull node bounding box against hierarchic depth buffer)
                CONTINUE
        ELSE
                distance ← distance between camera and node
                IF distance > threshold
                        voxelizationqueue ← voxelizationqueue ∪ node
                ELSE
                        children ← children of node
                        walkqueue ←  walkqueue ∪ children

CULLED OBJECT VOXELIZATION
voxelizationqueue← CULLED OBJECT DETERMINATION
WHILE voxelizationqueue not empty
        node ← pop voxelizationqueue
        aabb ←  node axis aligned bounding box
        minlength, maxlength ←  min(aabb.x, aabb.y, aabb.z)
        IF minlength << maxlength
                boxes ←  dice aabb into multiple even boxes
                FOR box in boxes
                        CIVmipmap, cluster ← compute which mipmap entry best encases box
                        flag cluster inside the CIVmipmap as occupied

DEPTH BUFFER REPROJECTION
depthbuffer ←  G-buffer
FOR pixel in screenpixels
        depth ←  sample depthbuffer at pixel
        position ←  reconstruct position from depth
        cluster ←  determine cluster in CIV mipmap 0
        flag cluster of the CIV texture mipmap 0 as occupied

MIPMAP POPULATION
mipmap= max CIV mipmap level
WHILE mipmap > 0
        FOR pixel in mipmap
                IF pixel is occupied
                        children ←  four pixels from mipmap-1 which are encased by pixel
                        IF any child in children is occupied
                                set the pixel to occupied
mipmap =0
WHILE mipmap < max CIV mimpap level
        FOR pixel in mipmaplevel+1
                children ←  four pixels from mipmap which are encased by this pixel
                IF any child in children is occupied
                        set the current pixel to occupied
```

The CIV texture storage can be modified to suit the needs of the renderer. If the renderer requires only inexact visibility for diffuse illumination than the CIV stores a single bit per pixel,

which is set when geometry is present. In this case a detailed 512x512x512 resolution would only require 16MB of GPU storage. If the renderer is used in for scattering effects, then the storage can be modified to a one byte per pixel structure, which holds one bit for transparent/slid geometry presence, one bit to differentiate between solids and transparents and 6 bits for either opacity or solid normal direction. In this case a detailed 512x512x512 resolution would only require 134MB of GPU storage.

If the renderer is used in more complex scenes, which contain scattering and indirect high frequency illumination, then the CIV storage can be modified to a 2 byte per pixel structure, which holds a bit for geometry presences, a bit to differentiate between solids and transparents and 14 bits which are either used for color (12 bits) and transparency (2bits) or for participating media transparency (5bit) and participating media color (9 bit). In this case a detailed 512x512x512 resolution would only require 268MB of GPU storage. In the case of specular lighting, tracing is also performed in screen space if the screen space resolution is superior to that of the CIV. This can be also applied to diffuse lighting, but it is not mandatory from an artifact suppression point of view.

Storage Type / Storage Requirements	128x128x128	256x256x256	512x512x512	1024x1024x1024
Simple (1bpp)	262.14 KB	2.09 MB	16.77 MB	134.21 MB
With transparents (1Bpp)	2.09 MB	16.77 MB	134.21 MB	1.07 GB
With transparents and colors (2Bpp)	4.18 MB	33.54 MB	268.42 MB	2.14 GB

Table 9 Conservative Inexact Voxelization Storage Requirements**.** Conservative Inexact Voxelization can either store the state of geometry in a single flag per pixel, store opacity information for transparents at one byte per pixel or store specular lighting information at two bytes per pixel. The storage structure can be modified depending on the needs of the renderer.

CIV consumes varying amounts of storage, depending on the rendering strategy, as shown in Table 10, but it compares positively with the state of the art methods, as shown in Table 10.

Rendering Algorithm \ Comparison Criterion	Full scene data	Conservative	Complexity	Ray Tracing Support	Scattering Support
Point Cloud Geometry Reconstruction	yes	no	O(vertices)	no	no
Point Cloud Convex Hull	yes	yes	O(vertices)	no	no
Imperfect Voxelized Shadow Maps	yes	yes	O(lights*vertices)	no	partial
Screen Space Voxelization	no	yes	O(primitives)	partial	partial
Imperfect Volumes	yes	no	O(primitives)	yes	yes
Sparse Octrees	yes	yes	O(primitives)	yes	yes
Conservative Inexact Voxelization (this algorithm)	yes	yes	O(objects)	yes	yes

Table 10 CIV and the state of the art**.** Conservative Inexact Voxelization is can be constructed much faster than all the other state of the art voxelizations, because it works directly at object level. Despite being an inexact representation, CIV is conservative, therefore it can be used to query the approximative structure of the scene.

The CIV algorithm can be optimized through many mechanisms. The construction cost can be paid once for state objects and amortized over multiple frames for the dynamic objects. This can also be used in combination with a temporal reprojection mechanism. Sparse storage methods can be applied to CIV to lower the storage requirements. Filtering over multiple reads into the CIV texture can be used to antialias traced rays.

The key property of conservative inexact voxelization is that if it is used as an acceleration structure for tracing rays, it offers a **varying visibility operator, accurate inside the visualization volume and coarse outside it**. This property is very useful for real-time light transport algorithms because it adapts to the perceptually important areas of the scene. It can be used for both low frequency and local high frequency light transport, and it is used chapters 4.1.3.1 and 4.1.3.2. The property it is presented in Figure 58.

$$L(x_1 \to x_0) = L_e(x_1 \to x_0) + \int_A f_r(x_2 \to x_0) G(x_1, x_2) V(x_1, x_2) L(x_2 \to x_1) dA(x_2)$$

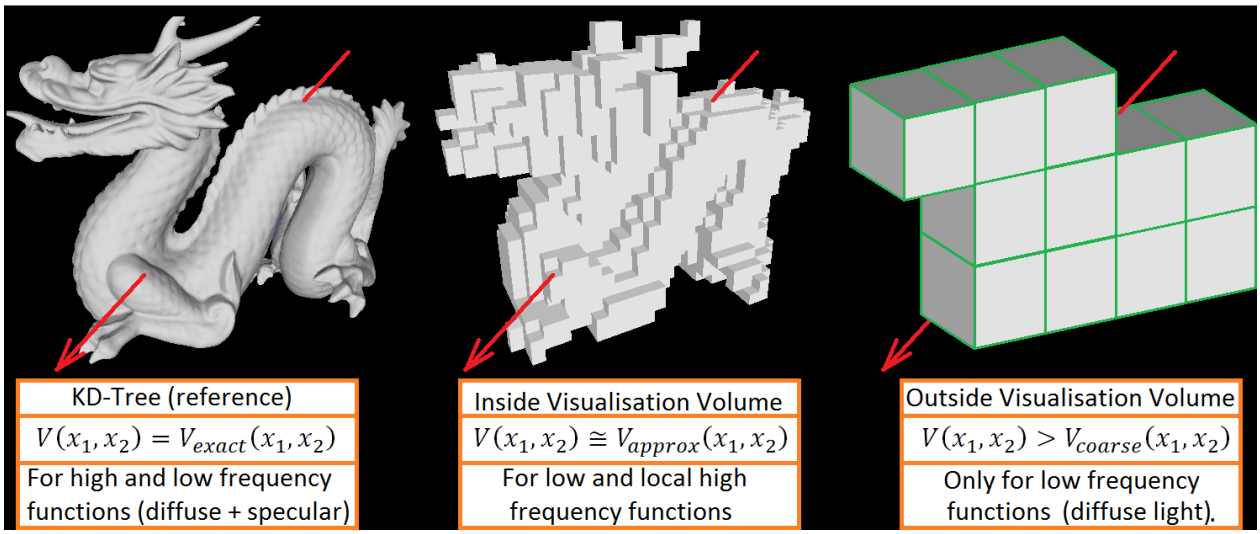

Figure 58 Varying Visibility Operators With Conservative Inexact Voxelization**.** This image shows how the conservative inexact voxelization algorithm can be used as a ray tracing acceleration structure. Because tracing inside the CIV is significantly more coherent than tracing inside a normal ray-tracing acceleration structure such as a kd-tree, the cost of visibility operations is much lower, while the quality is comparable inside the visualization volume, the area most perceptually important. Because CIV is also conservative, the tracing is conservative and no light transport can be done through geometry.

The presented voxelization method does not provide exact visibility, but it provides a fast conservative approximation of it. This property is sufficient for some light transport algorithms such as many light methods or secondary light shadowing. For real-time rendering purposes the presented structure provides a much faster generation method while still maintaining raster resolution accuracy inside the visualization volume. The presented conservative inexact voxelization is used in a modified instant radiosity algorithm in the next chapter.

The algorithm is also used to improve the fail cases of the screen space cone tracing algorithm, a high frequency light transport algorithm which works with approximated geometric information.

4.1.3. Light transport for Secondary Lights

Light transport for secondary lights is a notoriously difficult real-time rendering problem, for which a large number of solutions has been proposed. The difficulty to solve this is given by the nature of the simulated transport, which is expressed in the following equation, after [**Kaj86**]:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_r(x, \omega_i, \omega_o) L_i(x, \omega_i)(\omega_i \cdot n) d\omega_i$$

The bulk of the rendering time is usually dominated by visibility determination operations [**Hav14**] and the equation is by definition recursive, which quickly leads to incoherent visibility determination tests. But incoherent data accesses lead to the most costly operations in computer science, cache misses. And if the visibility queries are forced to be coherent through rasterization algorithms, than this leads to large number of passes over the entire scene as rasterization does not support recursion, as do path/ray/photon tracing. Furthermore, path/ray/photon tracing require acceleration structures, which have to be recomputed per frame, which further complicate rendering.

The problem is a long standing one and many approximations have been proposed. One of these approximations families, the many light algorithms, use few visibility operations to transfer large amounts of light, an approach compatible with the needs of real-time rendering. While this is practical with only a relatively small number of light transfers, the visual quality downgrade is acceptable in the context of real-time applications. Furthermore, this class of algorithms is inefficient for high frequency light transport.

This chapter uses a decoupled light transport solution, using different algorithms for low frequency and high frequency light transport. For low frequency (diffuse) light transport a novel method is presented. For high frequency (specular) light transport, a modified version of screen space cone tracing is used, which has its failure cases augmented with data from the low frequency solution. The two light transports algorithms have their results combined, to obtain the total light transport.

4.1.3.1. Light transport for Low Frequency Light

The many lights methods are based on instant radiosity [**Kel97**], which uses random walks inside the scene to transfer large amounts of light at once, which spawn many virtual lights, giving the many/virtual lights method its name. But in order to transfer these large amounts of light, the algorithm requires recursive visibility operations, which are impossible to implement efficiently through rasterization, therefore in the presented variant they are performed with rays. This problem makes instant radiosity especially difficult to implement for scenes with poor light transport, where the random walks need to trace more rays.

In this chapter a novel variant of instant radiosity is presented, in which the random walks are performed using ray tracing over a conservative inexact voxelization of the scene geometry. Because of this, the rendering equation used by this method has a different visibility operator than the exactly traced normal operator. But the introduced operator is both conservative and coherent, therefore it is sufficient for virtual light transport, as this operator is not used in the illumination computations directly. The modified visibility operator concept is shown in Figure

57. The visibility operator is used for both tracing the random walks which spawn the virtual lights, as displayed in Figure 58, and for tracing the shadows for the virtual lights.

The virtual lights are generated with a random walk process. A small number of samples are generated for each scene light. Distant lights are clustered together in hierarchic impostors, which then have their light emission sampled. The same procedure is applied to skyboxes, area and volume lights and impostor lights. The samples are generated from Halton sequences, which converge faster in $O(N^{-\frac{n+1}{2n}})$, compared to using random number series $O\left(N^{-\frac{1}{2}}\right)$, where n is the number of sampled dimensions and N is the number of samples. Care must be taken to drop the first few entries every other entry in the series, to minimize sample correlation and variance.

Each sample is traced, in a process similar to light tracing. Mutation strategies like [**Seg07**] [**Seg061**] can be optionally used. Each vertex of the path is treated as a potential virtual point light (VPL).

Depending on the type of CIV storage, the presented variation of instant radiosity can correctly perform color bleeding or not. For a CIV structure that does stores color (color bleeding is not available for the 1bit storing mode), the energy received and assigned to each virtual point light is given by the following equation (based on [**Vea95**]):

$$L(x_0 \to x_n) \approx L_e(x_0 \to x_1) \cdot \prod_{k=1}^{n} (brdf_k(x_{k-1} \to x_{k+1}) \cdot |\cos \theta_k|)$$

Where x_0 is the sample position on the light, x_1 is the next vertex in the path, x_n is the last vertex in the path, n is the length of the path, $brdf_k$ is the reflectance function and θ_k is the angle between the incident light direction and the surface normal. If a Lambertian reflectance function is used, as in the original instant radiosity paper [**Kel97**], then the equation becomes identical to that given by Veach and Guibas, with the $\frac{K_{d(x_k)}}{\pi}$ term [**Vea95**].

Scattering is also considered, if the used CIV storage supports it, changing the VPL energy equation to:

$$L(x_0 \to x_n) \approx L_e(x_0 \to x_1) \cdot \prod_{k=1}^{n} \left(brdf_k(x_{k-1} \to x_{k+1}) \cdot |\cos \theta_k| \cdot \sum_{j=0}^{N} p(j \to j+1) L_j \right)$$

Where N is the number of scattering events between x_{k-1} and x_k , p is the scattering probability, L_j is the incoming radiance to the j-th scattering event.

Not all the generated VPLs are saved, the rejection criteria for unimportant samples [**Geo10**] is that a sample must be either directly visible from the camera, or be directly linked with VPL that is directly visible from the camera.

The VPL generation process is presented in Figure 59.

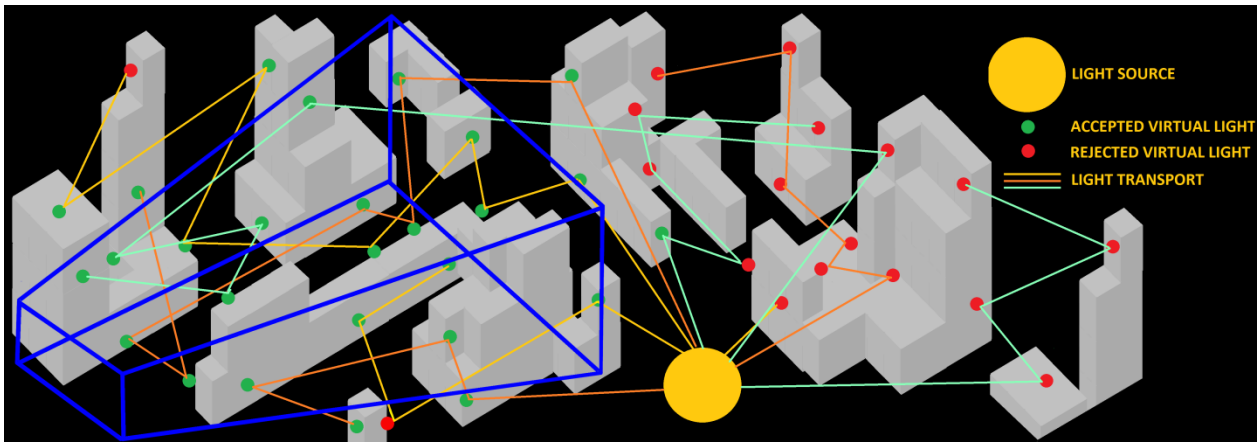

Figure 59 Virtual lights generation**.** Virtual lights are generated though random walk process, in which rays are traced in the conservative inexact voxelization acceleration structure. Each vertex of a random light path is a potential virtual point light. The samples are rejected based on their impact in visualization volume, shown in blue: the red samples are rejected, while the green samples are accepted.

The pseudocode for the modified instant radiosity is the following:

```
PREPROCESS
allocate texture with mipmaps for CIV
compute CIV

VPL GENERATION
scenelights ← scene lights
lightsqueue ← scene lights and light impostors
WHILE lightsqueue not empty
        node ← pop lightqueue
        samples ← generate n samples with Halton sequence, with position and radiance
        FOR sample in samples
                walkvpl ← Ø
                WHILE max recursion depth not reached
                        trace until surfacehit
                        FOR event in scattering events
                                compute inscatter and outscatter
                                compute surface interaction
                                vpl ← generate potential virtual point light
                                walkvpl ← walkvpl ∪ vpl
                WHILE walkvpl not empty
                        vpl← pop walkvpl
                        IF vpl succeeds rejection sampling
                                scenelights ← scenelights ∪ vpl
```

The presented instant radiosity variation can benefit from established scalability methods for many light algorithms such as the amortization of virtual light generation over multiple frames for both reduced computational effort and flickering prevention. While lightcuts [**Wal05**] [**Wal061**] [**Wal12**] [**Dav12**] and light grouping [**Don09**] [**Pru12**] are established methods for many lights storage, this thesis uses a clustered structure like one of the acceleration structures presented in Figure 60 to better integrate into the deferred inspired pipeline. All the secondary lights, VPLs and scene lights, are stored in a cluster.

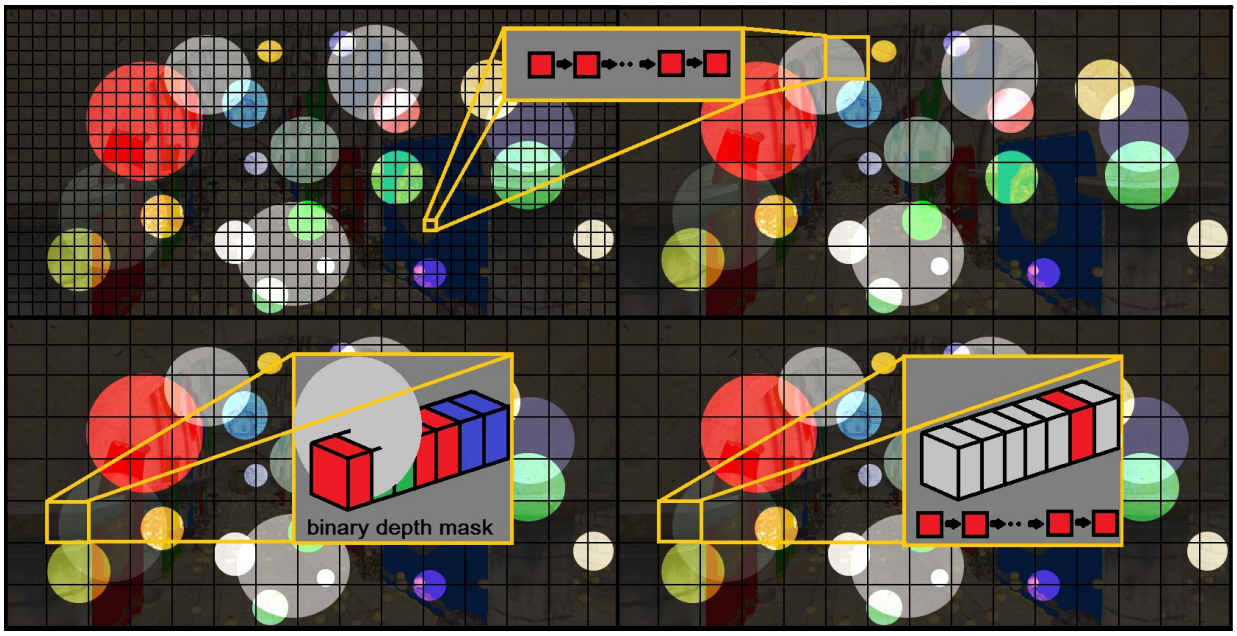

Figure 60 Scene lights acceleration structures**.** Secondary lights like small scene lights and generated virtual lights can be stored in a variety of methods which are easy to integrate with a deferred pipeline: per pixel linked lists (left top corner), tiles (right top corner), 2.5D tiles which uses a binary mask for light rejection (left bottom corner) and clusters (right bottom corner).

Rendering with virtual lights is performed by querying the light storage acceleration structure and solving all surface – light interactions. For the illumination with virtual lights the following equation is used (based on [**Vea95**]):

$$L(x \to p) = brdf_p(x \to p) \cdot V(x \to p) \cdot \frac{\cos\theta_r \cos\theta_i}{\|x - p\|^2} \cdot L(x' \to x)$$

Where x is the VPL used for illumination, p is the point being illumination by x, x' is the previous vertex to x in the random walk, θ_r is the angle between the VPL surface normal and the incident ray from x' , θ_i is the angle between the illuminated point normal and the incident ray from x and the visibility factor $V(x \to p)$ is traced over CIV, as shown in Figure 61.

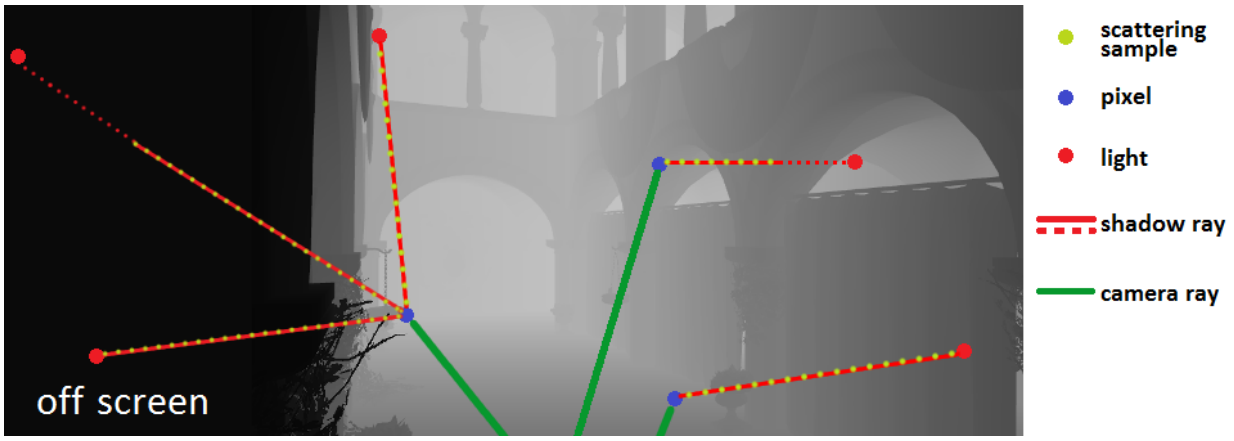

Figure 61 Secondary lights visibility**.** For each secondary light one or more shadow rays are traced over the conservative inexact voxelization acceleration structure. Scattering samples are also taken.

Illumination with virtual lights is not artifact free, as artifacts appear as structured illuminated hotspots, in contrast to the noise found in path tracing. This can be seen in Figure 61. These artifacts are caused by the singularities generated by the $\frac{1}{\|x-p\|^2}$ (part of the $\frac{\cos\theta_r \cos\theta_i}{\|x-p\|^2}$ form factor) term in illumination equation. A common solution to this problem is to clamp the contribution of each light [**Kol04**]. Another possibility is to store indirect diffuse illumination in a screen space buffer and filter it in a manner similar to SSPCSS [**Bag10**]. A scene illuminated with this method is presented in Figure 62.

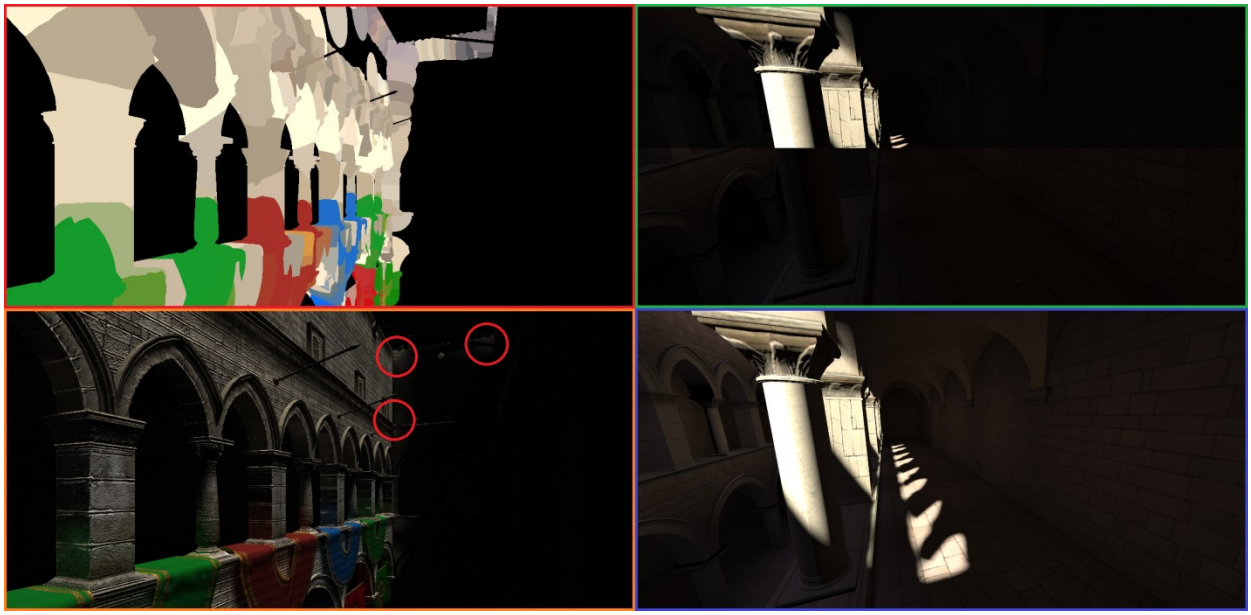

Figure 62 Illumination with virtual lights**.** The upper left corner of the image presents the intersection between virtual lights and the scene geometry. The lower left corner presents the results of lighting with these virtual lights, by using a color enabled conservative inexact voxelization acceleration structure. Artifacts can be seen at the top of the Sponza building, as structured illuminated hotspots, caused by an un-clamped form factor. Color bleeding can also be observed, especially on the columns. In the right upper corner the image is divided: the top shows the direct illumination and the bottom shows the indirect illumination computed with many virtual lights, using a colorless CIV. The combined result of both direct and indirect illumination is presented in the right bottom corner of the image.

4.1.3.2. Light transport for High Frequency Light

High frequency light transport is a difficult problem for the real-time rendering of dynamic scenes, and because of this, it is handled in a decoupled manner in this thesis, since many-light methods struggle to efficiently transport specular light [**Dav10**] [**Sim15**].

While high quality specular light transport is possible in real-time static scenes with heavy preprocessing [**Cra09**], the state of the art algorithm requires per-frame high resolution voxelization in order to work with dynamic scenes. Furthermore, the storage costs are extremely expensive.

The specular light transport method used in this thesis is based on screen space cone tracing [**Her14**] [**Ulu14**], which is an approximated variant of cone tracing. It can be executed before or after the diffuse light transport,

Screen space cone tracing generates one or more cones per pixel, which are then traced over the screen space. The direction of the reflected cone is given by the reconstructed surface – cone interaction. The size of the cone is based on the traced distance. In order to minimize the number of texture reads, the screen buffers are mipmapped and each cone surface intersection is performed at the fitting mipmap level, as determined by the cone size.

The screen space cone tracing algorithm has many fail cases, some of which are presented in Figure 63: lack of information due to depth overlap, tracing outside of the screen space or tracing through unknown space behind the screen space.

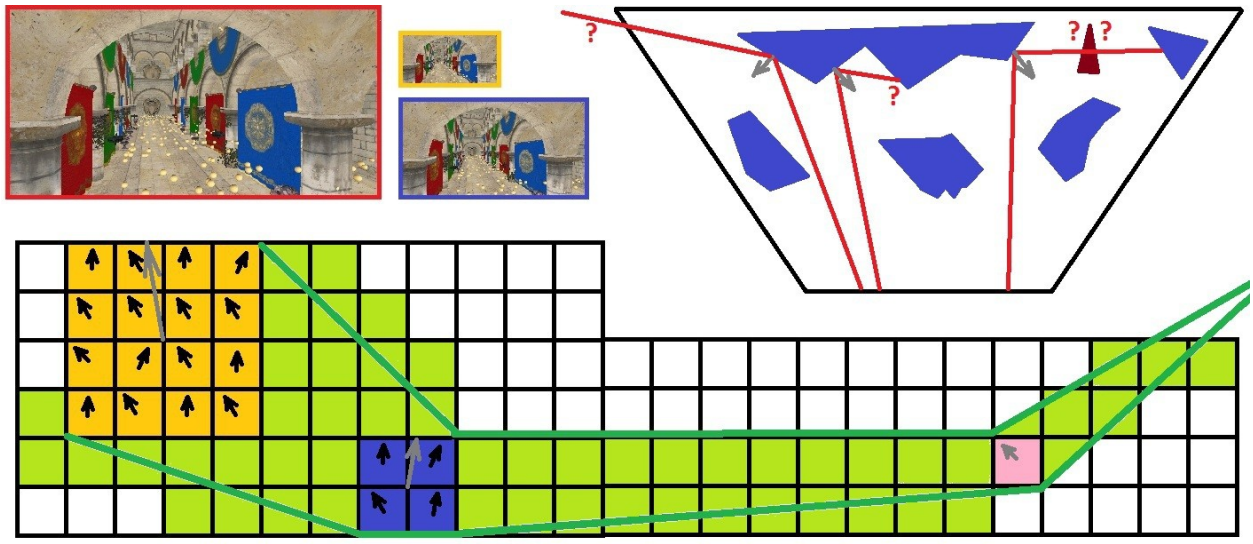

Figure 63 Screen space cone tracing**.** The image presents screen space cone tracing, an approximative tracing algorithm that works in screen space. Each pixel traces cones over the mipmapped screen buffers, reading from the fitting mipmap based on cone size. This is depicted in the three surface-ray interactions colored in pink (mipmap 0), blue (mipmap 1) and yellow (mipmap 2). The gray normals represent the averaged normal over the impact surface. The fail cases of this approximative algorithm are depicted in the upper right corner of the image: rays can be traced outside of the screen space, they can intersect surfaces behind the screen space, or they can pass through areas lacking information. All fail cases are directly caused by lack of geometric information outside the visualization volume. In this thesis, the conservative inexact voxelization is used to augment the rays which are traced outside the visualization volume, which greatly ameliorates the results of the fail cases.

The conservative inexact voxelization (CIV) acceleration structure used for low frequency light transport is used to augment screen space cone tracing in the failure cases, which can greatly lower visual artifacts, even if CIV is not designed for high frequency light transport. The CIV structure has to be one with color information enabled, which increases the storage costs drastically, as highlighted in Table 10.

Because the CIV resolution is usually significantly lower than the resolution of the screen space buffers, the cones are traced over both CIV and screen space, in order to maximize the accuracy of the resulting visibility determination operators. Scattering in participating media can also be implemented using CIV. Furthermore, cone tracing on the screen space and CIV is almost equivalent to cone tracing over an octree, if the voxelization for the CIV method is set to produce high quality results and the scene objects are cut into multiple other objects, but then the complexity of the method would be approximately equal to the one used in [**Cra14**] as the number of objects would soon be roughly the same as the number of primitives.

4.1.4. Opaque shading

This subchapter discusses opaque shading operations, in which lights and virtual lights generated and clustered in the previous subchapters are used to shade the opaque objects represented in the geometry buffers stored by the deferred opaque rasterization. The algorithm is the decoupled shading stage of the Virtual Deferred algorithm, using the virtual data mechanism as described in the 3.5.2 Chapter.

The method has two shading stages. In the first shading stage, the texture data is fetched and shading is performed with direct and indirect diffuse light transport and direct specular transport. In the second shading stage indirect high frequency light transport is performed, based on the results of the first shading phase.

The method starts by loading the Virtual Deferred G-Buffer data into work tiles, one GPGPU thread per pixel, in a GPGPU render pass. This is done in order to determine basic object properties such as which is the visible object which is being shaded in the pixel.

After loading the virtual G-Buffer data the algorithm has two alternate rendering paths, depending on the storage of texture coordinate derivatives. The more storage intensive bandwidth stores the texture coordinates derivatives per pixel. The more computationally intensive variant does not store the texture coordinates derivatives. Since proper texturing requires texture coordinates and their derivatives, the former rendering path is included in the latter rendering path.

The latter rendering path uses a texture derivative reconstruction process, which differentiates the texture coordinates stored in the pixel with the texture coordinates stored in neighboring pixels, if the neighboring pixels are of the same object id and material id. Thus, texture coordinate derivatives are reconstructed in the same manner they are initially generated by the hardware rasterization pipeline, through neighbor fragment texture coordinate differentiation. The algorithm then uses the texture coordinate derivatives to determine the mipmap levels required to properly perform the pixel texture fetches. This is done on a material basis. The shading method can include geometry reconstruction elements such as screen space tangent reconstruction through the use of normals and screen space position derivatives.

After the texture fetches have been performed and have been locally stored within the work group, the algorithm tackles illumination. It first solves dominant lights illumination through the use of the visibility operator provided by shadow maps techniques. The secondary scene lights are solved by using the visibility operator approximate in the conservative imperfect voxelization algorithm. Thus, the interaction of each secondary or virtual light with a pixel is determined through the tracing of one or a few rays. The illumination is then performed normally.

If multiple geometry frusta are used in the geometry pass the shading algorithm is not run multiple times, because the virtual geometry buffers produced by the multiple frusta are accumulated into a single coherent geometry buffer. A potential optimization is to perform multiple material paths, in order to minimize work tile divergence. On the other hand, if the algorithm is used in a rendering system where an asset format can be imposed, a proven and flexible reflectance function such as Cook Torrance [**Coo82**], is a good choice as it implicitly minimizes work tile thread divergence.

The pseudocode for the low frequency illumination and shading stage of the virtual deferred algorithm is:

```
LOW FREQUENCY ILLUMINATION AND SHADING STAGE
FOR tile in screenpixels
    allocate tilecache
    FOR pixel in tile
        FOR material in materials                          // TEXTURE FETCHING
        IF the texture derivatives are not stored in the virtual G-Buffer
            neighborlist ← Ø
            texderivatives←0
            FOR neighbor of pixel in 2x2 region
                IF pixel and neighbor share object id and material id
                    neighborlist ← neighborlist ∪ neighbor
                IF neighborlist not empty
                    avgderivative ← 0
                    FOR neighbor in neighborlist
                        difference ← compute texcoord difference between
                                     neighbor and pixel
                        avgderivative ← accumulate difference
                    texderivatives ← compute derivatives with avgderivative
                ELSE
                    Set derivatives ← 1 (highest mipmap)
            mipmap ← determine mipmap with texture coordinates, object ID, mat ID
            inf, sup ← use texderivatives to find the inf and sup texture mipmaps
            IF inf not stored
                inf ← closest stored mipmap level
            IF sup not stored
                sup ← closest stored mipmap level
            physicalcoordinates ← texture coordinates, inf, sup
            texturedata ← sample virtual texture with physicalcoordinates
            tilecache ← texturedata
    SYNCHRONIZE TILE
    dominantlights ← load dominant lights in tile        //ILLUMINATION PSEUDOCODE
    FOR pixel in tile
        pixelcolor ← 0
        FOR light in dominantlights
            IF light intersects pixel
                visibility← query dominant light shadow map
                FOR material in materials
                    shadedcolor ← lighting&shading← visibility, tilecache
                    pixelcolor. ← pixelcolor + shadedcolor
    secondarylights ← load the frustum secondary lights in tile
    SYNCHRONIZE TILE
    FOR pixel in tile
        FOR light in secondarylights
            IF light intersects pixel
                cone ← light, pixel
                visibility ← TRACE CIV(0, cone)
                FOR material in materials
                    shadedcolor ← lighting and shading using visibility and tilecache data
                    pixelcolor ← pixelcolor + shadedcolor
        OUTPUT pixelcolor
```

The second shading pipeline computes indirect approximative specular light transport. This is an optional stage and it should be disabled if a scene does not contain perceptible high frequency specular transport. The second shading stage is computed in screen space, based on the shading results of the first shading stage and because of this, the visibility operator used to transfer specular light between the surfaces is very poorly approximated and will not produce exact results. Because of this the illumination pipeline uses screen space cone tracing instead of screen space ray tracing, because cone tracing includes additional filtering which decreases the occurrence of artifacts.

The algorithm can trace color into the Conservative Inexact Voxelization structure, if the colors and normals are saved, which augmented by the screen space cone tracing makes the entire method work like a fast, low quality voxel cone tracing algorithm. The color/normal variant of the CIV is also a support for scattering, as it is explained in Chapter 4.1.2. While CIV is not created for high frequency visibility determination, it can be used to produce acceptable results from a perception standpoint. More importantly using CIV helps alleviate the many visual artifacts that appear in screen space cone tracing, by providing geometric information outside the visualization volume.

The pseudocode for the second shading pipeline of the virtual deferred algorithm is:

```
PREPROCESS
SS ← Create a mipmap with the depth, normal and the color outputted from the first shading pass
CIV ← compute inexact voxelization
IF using CIV AND CIV support normal and color
        CIV← CIV ∪ project each normal and color from ssmipmaps into CIV

TRACE CIV(recursiondepth, cone)
mipmaplevel ← max mipmap level (coarsest) of CIV
scatter ← 0
surfacehit ← null, trace cone over the mipmaplevel
IF surfacehit
        WHILE no exact surfacehit
        IF mipmaplevel > minimum mipmap level (given by cone angle and traced distance)
                surfacehit, scatter ← trace until approximative surfacehit and accumulate scattering
                mipmaplevel ← mipmaplevel – 1
        ELSE
                surfacehit, scatter ← trace until exact surfacehit and accumulate scattering
IF surfacehit
        radiance ← 0
        normal ← normal at surfacehit in CIV (either filtered or directly stored)
        cluster ← get cluster from lightgrid in vicinity of surfacehit
        IF surfacehit reflective
                IF recursiondepth < max recursion depth    //SAMPLE SURFACES
                        reflectedcone = reflect(cone, normal, distance traced, material)
                        incomingradiance← TRACECIV(recursiondepth+1, reflectedcone)
                        scatteredincomingradiance← compute scattering with scatter, lightradiance
                        reflectedradiance ← scatteredincomingradiance, normal
                        radiance ← radiance + reflectedradiance
        RETURN radiance
ELSE
        RETURN 0
```

```
TRACE SCREEN (recursiondepth, cone)
mipmaplevel ← max mipmap level (coarsest) of SS
surfacehit ← trace cone over the highest mipmaplevel
WHILE exact surfacehit not found AND inside of screen space
        IF mipmaplevel > minimum mipmap level (given by cone and traced distance)
                surfacehit ← trace until approximative surfacehit or outside of screen space
                mipmaplevel ← mipmaplevel - 1
        ELSE
                surfacehit ← trace until exact surface hit or outside of screen space
radiance ← 0
IF surfacehit found
        IF the surfacehit reflective AND recursiondepth < max recursion level
                position, normal, color, material ← read data from the mipmaplevel at surfacehit
                reflectedcone ← reflect(cone, normal, angle based on material)
                incomingradiance ← TRACE SCREEN (recursiondepth +1, reflectedcone)
                reflectedradiance ← incomingradiance, normal, material
                radiance ← radiance + reflectedradiance
ELSE
        IF using CIV
                radiance ← TRACE CIV (recursiondepth, cone)
RETURN radiance

LOW FREQUENCY ILLUMINATION AND SHADING STAGE
FOR tile in screen
        tilecache ← load data from virtual deferred G-buffer (depth, normals, colors, material id)
        SYNCHRONIZE tile
        FOR pixel in tile
                diffusecolor ← LOW FREQUENCY ILLUMINATION AND SHADING STAGE
                depth, normal, material ← read from G-buffer, pixel
                position ← reconstruct position from depth, camera
                cameraray ← ray from camera to position
                coneangle ← compute cone angle based on material, normal
                cone← cameraray, angle based on material
                indirectradiance←0
                IF using a sufficiently high resolution colored CIV
                        indirectradiance← TRACE CIV ( 0, cone)
                ELSE
                        indirectradiance←TRACE SCREEN ( 0, cone)
                specularcolor ← indirectradiance, normal, material
                pixelcolor ← diffusecolor + specularcolor
                OUTPUT pixelcolor
```

The presented shading pipeline perfectly integrates with the Virtual Deferred algorithm, and can be used as a self standing algorithm, outside of the rendering framework proposed in this thesis.

The shading pipeline is continued with the decoupled sub-pixel reconstructed antialiasing (DSRAA) method, a decoupled antialiasing algorithm especially designed for solving antialiasing in deferred pipelines. DSRAA can be used to tackle the aliasing inherent to the low sample reconstruction processes which take place in deferred algorithms. DACRT can also be used to counter the micro aliasing produced by texture coordinate derivative reconstruction, especially if the texture coordinates are stored in compressed format.

4.1.5. Decoupled sub pixel reconstructed anti-aliasing

In rendering there are many sources of aliasing such as geometric aliasing, texture aliasing or shading aliasing. Geometric aliasing is usually the easiest to perceive [**Hum09**] and also the most difficult to handle, especially in a deferred rendering context, therefore geometric antialiasing is one of the largest forms of aliasing in rasterization rendering. Deferred rendering is in particular prone to aliasing artifacts as storing the information for the entire set of shading samples quickly leads to excessive bandwidth and storage costs. Anti-aliasing methods can be categorized into a small number of algorithm families: sampled antialiasing, morphological antialiasing, temporal antialiasing, analytical antialiasing and hybrid sampled antialiasing.

Like in any digital signal processing problem, an increased number of samples leads to a better, more accurate, reconstructed result. In the case of rasterization rendering the original signal is represented by the analytically rasterized geometry, which is then reconstructed through samples multiple geometry samples per pixel. Sampled antialiasing methods keep multiple shading samples per pixel, which increase the resolution of the rasterization process. Sampled antialiasing methods either fully shade each of the samples of the pixels or use different rates in order to perform visibility determination at the maximum sample rate and shade at a reduced rate. Because shading is much more costly than basic visibility determination in rasterization, a reduced shading rate is almost always employed. Supersampling antialiasing (SSAA) [**Jim11**], Multisampling Antialiasing (MSAA) [**Jim11**], Coverage antialiasing (CSAA) [**Jim11**], enhanced quality antialiasing (EQAA) [**Jim11**] and deferred MSAA are all examples of techniques based on sampling. Because sampled techniques need to store data per sample they are extremely costly to use with deferred rendering.

Morphological antialiasing takes a different approach to antialiasing. Instead of trying to reconstruct the original signal, in this case geometry, morphological antialiasing methods try to find known patterns in the un-filtered result and then use precomputed filtering solutions for these cases. Because of this, morphological antialiasing is not an exact reconstruction method, and while algorithms from this category can produce visually antialiased results, they suffer from temporal instability. Fast Approximate antialiasing (FXAA) [**Lot09**], Morphological antialiasing (MLAA) [**Jim11**], Sub-pixel morphological antialiasing (SMAA) [**Jim12**] are all examples of solutions and processes which tackle aliasing from a morphological perspective. A variant of morphological solution are edge antialiasing solutions such as Directionally Localized antialiasing (DLAA) [**And11**], normal filter antialiasing (NFAA) and screen space SSAA [**Uni11**].

Analytical antialiasing methods solve aliasing through analytical reconstruction. They are rarely used in real-time rendering, because they are very expensive. Therefore, they are only used for specific scenes. Geometry Buffer antialiasing (GBAA) [**Jim11**] , Distance to edge antialiasing (DEAA) [**Jim11**] and phone wire antialiasing (PWAA) [**Per15**] are all analytical antialiasing methods.

Temporal solutions aim to increase the number of samples through the reuse of samples from previous frames, basically amortizing the sampling cost. These methods are usually combined with other antialiasing methods, leading to algorithms such as temporal antialiasing (TXAA) [**Yan09**] and Subpixel morphological antialiasing (SMAA) [**Jim12**].

Hybrid sampled algorithms completely decouple the sampling rate of visibility determination operations from the sampling rate of shading. These methods generally use multiple stages, in which sampling data is either aggregated or decoupled, only to be resampled and resolved in post-processing stage. Aggregate geometry buffer antialiasing (AGAA) [**Cra15**], surface based antialiasing (SBAA) [**Sal12**], Subpixel reconstruction antialiasing (SRAA) [**Cha11**] and resampling antialiasing (RSAA) [**Res12**] are examples of hybrid sampling antialiasing techniques. Hybrid methods have seen the most development in recent period, but they still do not decouple bandwidth usage.

While supersampled methods are excessively costly from a storage and bandwidth standpoint and morphological algorithm require temporal antialiasing methods to prevent frame to frame artifacts, hybrid methods take the best from both approaches, sampling only critically important data and then reconstructing the samples.

The thesis presents an improvement over hybrid methods, which completely **decouples shading bandwidth from visibility determination** from the antialiasing point of view, further lowering the bandwidth and storage costs of antialiasing methods while. The presented method, decoupled sub pixel reconstructed antialiasing, shades only once per pixel and uses the shaded results to reconstruct the other samples, at a sub-pixel level. The algorithm then uses the reconstructed samples in a standard resolve process. The method is similar to SRAA, but the **reconstruction method is based on shaded sample matching** and not on direct filtering. Furthermore, the reconstruction method is much easier to implement and less computationally expensive than SRAA.

Decoupled sub pixel reconstructed antialiasing is easily incorporated into any deferred rendering pipeline. The presented method has two stages. In the first stage, named the sampling stage, the method is integrated into any deferred renderer, such as the virtual deferred method presented in Chapter 3.5.2, which is slightly modified to sample visibility determination at a large rate. Therefore, the deferred renderer saves depth and performs the z-Buffer algorithm at many sub-pixel samples but saves all the other bandwidth-heavy information in a single sample. With this system, the increased bandwidth is comparable to that of a low bandwidth hybrid antialiasing algorithm such as SRAA. Optionally, normals can also be saved along with depths, as they can be used to increase the accuracy of the method, albeit increasing the bandwidth cost.

The second stage, named the reconstruction stage, of the algorithm works as a post process, in which both the shaded samples generated by the deferred renderer and the unshaded visibility determination samples are loaded in a tiled format. The unshaded samples are first linked to the pixel shading sample, based on a depth, and optionally normal, distance metric, such as SADP [**Res12**]. Then, the unshaded samples linked to the pixel samples have their color set to that of the pixel shaded sample, and are now considered “shaded”.

In the next step the remaining unshaded samples are linked to the shading samples from the neighbors of the processes pixel, again based on a metric of depth and optionally normal distances. When a sample is linked to more than one neighbor, each of the N neighbors is given a weight, based on the following metric:

$$weight_i = \frac{distance(sample, neighbor_i)}{\sum_{i=0}^{N} distance(sample, neighbor_i)}$$

Each unshaded sample is then resolved based on the linked neighbors:

$$Sample_{color} = \sum_{i=0}^{N} Neighbor_{i_{color}} * weight_i$$

If unshaded samples remain they are set to the background color. The final pixel color is obtained by resolving all the samples:

$$Pixel_{color} = \frac{\sum_{i=0}^{N} Sample_{i_{color}}}{N}$$

The algorithm is visually presented in Figure 64.

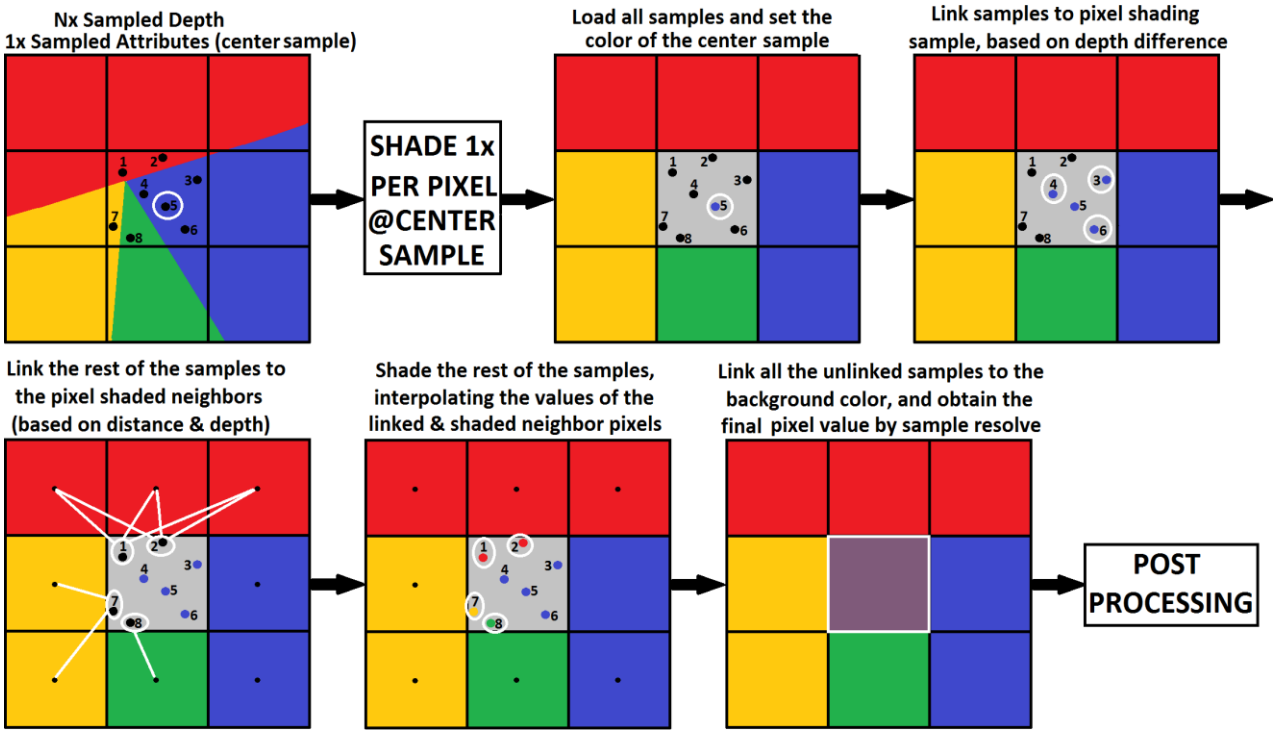

Figure 64 Decoupled sub-pixel reconstructed antialiasing**.** The antialiasing algorithm is designed for deferred renderers, and it has two stages. In the first stage the method slightly modifies a deferred renderer to sample all the bandwidth heavy attributes once per pixel, but to keep many visibility determination samples. After this, the deferred renderer shades each pixel with a single, central sample. The second stage of the algorithm has multiple steps. In the first steps all the unshaded samples are loaded per tile, in the second step the unshaded samples which are similar in depth to the shaded samples are linked to the shaded sample. The rest of the unshaded samples are linked to the neighbor pixels and colored through the interpolation of the colors of the linked neighbors. The final color is obtained by resolving all the sub-pixel samples.

The presented method computes the minimum bandwidth required to correctly perform sub pixel visibility determination while still working within a single geometry pass deferred renderer. Compared to the state of the art SRAA the bandwidth is equal and the reconstruction mechanism is easier to implement and less computationally expensive. Compared to other deferred oriented antialiasing algorithms such as AGAA, the method is material aware, and does not filter pre-shaded data or geometry.

This is the pseudocode for the decoupled sub-pixel reconstructed antialiasing method:

```
(ONCE) PREPROCESS
select a pixel sampling strategy and a centralsample for shading
allocate multisampled msdepthbuffer
IF using normals
        allocate multisampled msnormalbuffer

SAMPLING STAGE (DEFERRED RENDERER INTEGRATION)
FOR pixel in screenpixels
        FOR sample in pixel
                depth ← compute sample depth
                msdepthbuffer ← store sample depth
                IF using normals
                        normal ← compute sample normal
                        msnormalbuffer ← store sample normal
                normal deferred renderer
                attributes saved once per pixel, in centralsample
```

Between the sampling stage and the reconstruction stage the once per pixel sampled attributes of the geometry buffer are used for shading and illumination, thus shading is performed exactly once per pixel. The reconstruction stage takes place after the shading stage:

```
RECONSTRUCTION STAGE (TILED)
FOR tile in screenpixels
        FOR pixel in screenpixels
                tilecache ← load the centralsample from the deferred renderer results, with color
        SYNCHRONIZE tile
        pixelcolor←0
        FOR pixel in screenpixels
                centralsample ← load the shaded central sample from tilecache
                samples← load all the unshaded depth (and optionally normal) samples
                samplecolors[num samples]←0
                sampleshaded[num samples] ←false
                FOR sample in samples
                        distance ← distance metric sample to centralsample, use depth (optionally normals)
                        IF distance < threshold
                                samplecolors [sample] ← centralsample color
                                sampleshaded[sample] ← true
                FOR sample in samples
                        IF sampledshaded[sample] = false
                                linked ← ∅
                                FOR neighborpixel in neighbor pixels
                                        distance ← dist. metric sample to centralsample of neighborpixel
                                        IF distance < threshold
                                                linked ← linked ∪ (neighborpixel, distance)
                                        IF linked empty
                                                samplecolors [sample] ← background
                                        ELSE
                                                FOR neighbor in linked
                                                        weight ← neighbor distance
                                                        samplecolors [sample] ← average linked neighbors
                        pixelcolor ← pixelcolor + samplecolors [sample] * 1/(num samples)
                OUTPUT pixelcolor
```

A comparison of the introduced algorithm, along with various antialiasing techniques is given in Table 11.

Algorithm\Quantity per pixel	Depth Samples	Coverage Samples	Geometry Data	Shading Data	Storage Requirements	Bandwidth
No antialiasing	1	0	1	1	1	1
Multisampling Antialiasing (MSAA)	+++	+++	1	1	+	+
Coverage Sampling (CSAA/EQAA)	+++	+++++	1	1	+	+
Supersampling Antialiasing (SSAA)	+++++	0	+++++	+++++	+++++	+++++
Deferred MSAA	+++++	0	+++++	+++++	+++++	+++++
Fast Approximate Antialiasing (FXAA	1	0	1	1	1	+
Morphological Antialiasing (MLAA)	1	0	1	1	1	+
Subpixel Morphological Antialiasing (SMAA)	+	0	+	+	+	+++
Directionally Localized Antialiasing (DLAA)	1	0	1	1	1	+
Geometry Buffer Antialiasing (GBAA)	1	0	+++++	1	+	+
Distance to Edge Antialiasing (DEAA)	1	0	+++	1	+	+
Phone Wire Antialiasing (PWAA)	1	0	+++	1	+	+
Temporal Antialiasing (TXAA)	1	0	1	1	+++	+++
Aggregate G-Buffer Antialiasing (AGAA)	1	+	1	1	1	+
Surface Based Antialiasing (SBAA)	+++	+++	+++	1	+++	+++
Subpixel Reconstruction Antialiasing (SRAA)	+++	0	1/+++	1	+/+++	+/+++
Resampling Antialiasing (RSAA)	+++	+++	+++	1	+	+
Decoupled Subpixel Reconstructed Antialiasing (DSRAA)	+++	0	1/+++	1	+/+++	+/+++

Table 11 DSRAA and antialiasing algorithms**.** The table compares the presented algorithm in terms of sampling rates per pixel. Legend: 0 – no samples or not used, 1 - a single sample, + – a small number of samples or a small amount, +++ – many samples or a large amount, +++++ – a very high number of samples/amount .

In comparison to the state of the art SRAA algorithm, the presented method uses a better reconstruction stage, in which exact links are created between each of the unshaded visibility samples and either the shaded central sample of the pixel or the shaded central samples of the neighboring pixels. Because of this, instead of approximating each unshaded sample contribution with a bilateral filter, each unshaded sample is first linked to the existing shading samples, reconstructed, and only then used in the pixel resolve. Thus, this method produces results closer to the correct value that would be obtained if full supersampling would be applied. Figure 65 presents a comparison between DSRAA 8x and MSAA 8x.

Figure 65 Decoupled sub-pixel reconstructed antialiasing Results**.** This image presents the difference between DSRAA 8x and MSAA 8x. The visual results are extremely similar.

4.1.6. Transparent Shading

This subchapter discusses transparent shading operations. The approximated distribution occupancy maps method is a coupled solution, and it is discussed as a geometry method, in chapter 3.6.2. The subchapter presents various shading methods for the objects represented in the virtual a-buffer nodes, stored by the VOIT algorithm. The first presented method shades in the classical A-Buffer style, but with adaptive texture loading and shading. The second presented method is an adaptive texture loading and shading limited ray tracing method. Compared to the Virtual deferred shading stage which was presented in the previous sub-chapter, the shading stage for the virtual order independent transparency (VOIT) algorithm (also named virtual a-buffer) is a more intricate process.

If the texture coordinate derivatives are not stored at the virtual order independent transparency node level, they need to be reconstructed, and VOIT uses a micro tile phase in which it reconstructs the texture coordinate derivatives. The algorithm loads the fragment node lists into micro-tiles, which are smaller than the tiles used in virtual deferred because of the hardware memory per tile limitations, usually limited to just 2x2 pixel groups. The 2x2 limit is the minimum required to reconstruct texture coordinate derivatives, which are needed for proper texturing. After the fragment nodes are loaded in micro tiles, they are sorted by their depth, therefore multiple lists are sorted together. Thus, each pixel first sorts its own list and then one of the pixels does a merge step, like in the merge sort algorithm. This per-pixel sort operation can be implemented through in-place quicksort, or any other fast in-place sorting algorithm. Each pixel thread walks the sorted micro tile list, reconstructing the texture coordinates only for the nodes which it owns. For each owned micro tile list node each pixel search the vicinity of the micro tile list node in the micro tile list, for neighbors which have the same object id and material id. Based on the found neighbors the texture coordinates are differentiated in screen space and the texture coordinate derivatives are obtained.

If the texture coordinate derivatives are stored, than the virtual order independent transparency algorithm does not need to reconstruct them. The algorithm only loads the nodes of each pixel in local memory, where it sorts them.

After the texture coordinate derivatives are available, the algorithm walks the depth sorted nodes in front to back order, loading the texture data and shading and illuminating each walked node. The walk is adaptive, stopping as soon as the alpha channel reaches an opacity threshold. The front to back composition is done using the following equations:

$$\begin{cases} C_c = B_\alpha * (F_\alpha * F_c) + B_c \\ C_\alpha = (1 - F_\alpha) * B_\alpha \end{cases}$$

In the above equation F_α is the fragment opacity, F_c is the fragment color, C_c is the composited color and B_c is the background color. Because of the front to back composition strategy, the process stops as soon as the alpha channel is occluded. This can drastically reduce both shading computations and texture bandwidth. In order to determine the fragment color and opacity, VOIT uses the same virtual texturing backed mechanism as virtual deferred: the texture coordinate derivatives are used to determine the mipmap levels required to properly perform the pixel texture fetches, and the mipmaps are then sampled and interpolated.

The pseudocode for the shading pipeline of the virtual order independent transparency algorithm is:

```
SHADING STAGE
IF lighting
        lightgrid← scene lights
IF VOIT doesn't store texture coordinate derivatives
        FOR microtile(2x2) in screenpixels
                microtilecache ← allocate space for node lists per micro tile, each microtile node stores parent
                FOR pixel in microtile
                        microtilecache, list← load the pixel fragment list into the microtile storage cache
                        microtilecache,list← sort the list in place, in the microtilecache
                SYNCHRONIZE microtile
                mergedlist ← one pixel in microtile merges the lists
                SYNCHRONIZE microtile
                FOR pixel in pixels
                        finished ← false
                        WHILE not finished
                                node ← next node in mergedlist, owned by pixel
                                avgderiv ← Ø
                                FOR neighbornode in mergedlist
                                        IF neighbors (distance+ material metric)
                                                coord ← texture coordinates of node
                                                ncoord ← texture coordinates of neighbornode
                                                texturederivatives ← differentiate coord, ncoord
                                                avgderiv ← avgderiv ∪ texturederivatives
                                derivatives← reconstruct avgderiv
                                node ← store derivatives
ELSE
        FOR pixel in screenpixels
                locallist ← load the pixel fragment list into the local storage cache
                locallist ← sort the list in place, in the local storage cache
FOR pixel in screenpixels
        pixelcolor, pixelalpha ← 0
        IF VOIT doesn't store texture coordinate derivatives
                list ← mergedlist in microtile
        ELSE
                list ← locallist
        WHILE pixelalpha < threshold
                node ← get next fragment owned by the pixel, in front to back order, from list
                texture ← texture coordinates, object ID, material ID, like in standard virtual texturing
                inf, sup ← use texderivatives to find the inf and sup texture mipmaps
                IF inf not stored
                        inf ← closest stored mipmap level
                IF sup not stored
                        sup ← closest stored mipmap level
                physicalcoordinates ← texture coordinates, inf, sup
                texturedata ← sample virtual texture with physicalcoordinates
                IF using lighting
                        determine visibility through shadow maps or ray tracing over CIV
                        perform illumination with intersected scene lights
                pixelcolor, ← the fragment with the already walked fragments,front to back order
                pixelalpha ← compute the new fragment alpha occlusion in front to back order
OUTPUT pixelcolor
```

The presented shading method is more efficient than the standard classic a-buffer method because virtual order independent transparency has adaptive bandwidth consumption and adaptive shading, performing only the texture fetches and shading computations which have an impact in the final visual result. Thus, virtual order independent transparency decouples shading, bandwidth consumption and geometry processing for transparent object rasterization.

The presented method can be further visually improved through **better support for specular light transport** in order independent transparency, albeit at a steep bandwidth penalty. High quality high frequency light transports for transparent objects inside the visualization volume can be achieved through a sparse volume representation of the rendered transparent objects in the scene, in which rays can be traced.

Because virtual order independent transparency is an accurate rendering method, the visibility approximation operator can't be applied here. Instead of a conservative inexact voxelization an indexed cluster grid is used, which can accurately detect ray-surface intersection. On the other hand, if slight approximations are permitted, the visual results can be augmented with inaccurate data from a transparents enabled CIV, in similar fashion to the extension done in the high frequency light transport for opaque objects, as shown in chapter 4.1.4.

The visual volume is guaranteed to lack occluders, as the fragments generated by the VOIT algorithm are generated after the depth buffer is populated with opaque fragments. If a ray is reflected or refracted outside of the visualization volume it can either be traced inside the scene with CIV, or just be discarded.

If the rendering process is concerned with occlusion from opaque data, the conservative inexact voxelization is evaluated in clusters, which have the same size as the clusters used by the index cluster grid. These are used to

The method runs in two stages: the link stage, which links the linked list nodes to the index cluster grid, and the shading stage in which the index cluster grid is traced and the referenced nodes are adaptively shaded. Each index cluster from the index cluster grid holds a linked list with pointers to all the virtual order transparency nodes which are spatially located inside the cluster.

The shading stage spawns for each pixel reflection and refraction rays, for the closest camera-surface interaction. These rays are then traced inside the index cluster grid and new refraction and reflection rays are spawned for each contact. If a ray intersects an index cluster and the cluster is not empty, the ray is tested for intersection with each node linked to the cluster. Since the index cluster grid is a very sparse volume representation, this does not lead to excessive computation tests.

The linked list nodes are shaded on demand, the first time a node is reference by a ray, it has its textures fetched through the virtual data method and it is shaded. Instead of reconstructing the texture derivatives by walking multiple lists in micro-tiles this variant of virtual order independent transparency queries the nearby clustered nodes in order to obtain local texture coordinate differences. The texture derivatives are then obtained, the texel fetches performed and the color for the node is computed. The color is then written in the space of the texture coordinates, in a single atomic operation.

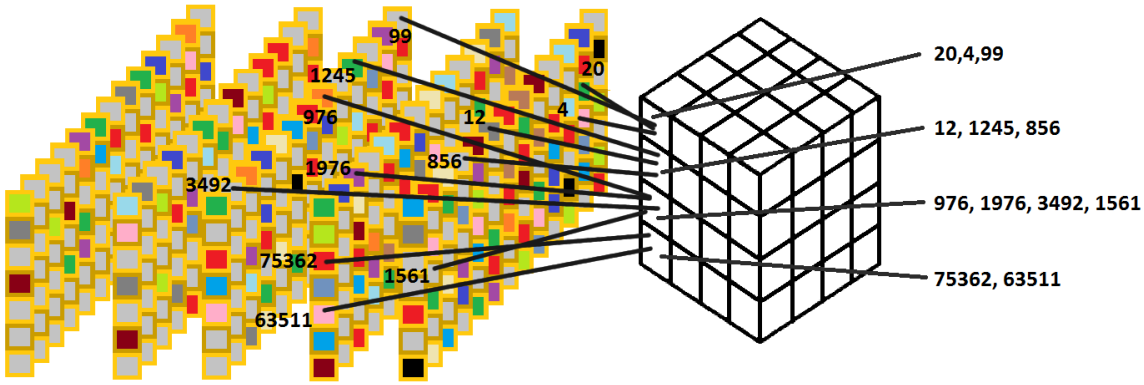

Figure 66 Virtual order independent transparency with a cluster grid**.** A cluster grid can be used together with virtual order independent transparency, which enables refractions and reflections with ray tracing over the fragments saved in the clusters. The fragments are mapped to the cluster grid, which stores a list of the mapped fragments per cluster. Because of this the visibility determination operator is accurate over this sparse voxelization. The algorithm runs over the objects visible in screen space, therefore occlusion from opaque objects is not considered, since transparent objects are rendered after the depth buffer is populated with opaque objects.

This variation of the VOIT algorithm is presented in Figure 66. The method still shades and reads texture data adaptively, only now it follows rays. The new shading pseudocode is:

```
PREPROCESSING
indexgrid ← allocate an index grid of clusters, which hold pointers to VOIT nodes
FOR cluster in indexgrid
        clusternodes ← Ø

LINK STAGE (integrated in GEOMETRY STAGE)
indexgrid ← clear
FOR pixel in pixels
        list ← load the pixel fragment list into local shader memory
        FOR node in list
                position ←use node depth to reconstruct the position of the stored fragment
                cluster ← use position to determine in which cluster in indexgrid would the node reside
                index← fragment index inside the nodebuffer, flag index as unshaded
                clusternodes ← clusternodes ∪ index

SHADING STAGE
FOR pixel in pixels
        list ← load the pixel fragment list into local shader memory
        IF list empty
                OUTPUT background color
        node ←first node in list, front to back order
        position ←depth from node
        cameraray ← create ray from camera to position
        normal ← the normal of the node
        refractedray ← refract cameraray against normal
        reflectdray ← reflect cameraray against normal
        cluster ←indexgrid cluster, closest to node
        incomingrefractedcolor ← TRACE(refractedray, cluster)
        incomingreflectedcolor ← TRACE(reflectdray, cluster)
        refractedcolor ← incomingrefractedcolor, normal, material
        reflectedcolor ← incomingreflectedcolor, normal, material
RETURN refractedcolor + reflectedcolor
```

```
TRACE(ray, cluster)
trace starting from cluster, over the indexgrid until a hit with non empty cluster is detected
IF no hit
        RETURN 0
ELSE
        cluster ←indexgrid cluster for closest to hit
        intersectednodes← Ø
        FOR index in cluster
                node ←LOAD(cluster,index)
                position ← reconstruct fragment position, from depth
                nodegeometry ←sphere, with radius of pixel size, centered at position
                IF ray intersects nodegeometry
                        intersectednodes ← intersectednodes ∪ node
        IF intersectednodes not emptry
                sort intersectednodes, over ray
        ELSE
                TRACE(ray, cluster)
        node ← first node in intersectednodes
        normal ← the normal of the node
        refractedray ← refract ray against normal
        reflectdray ← reflect ray against normal
        incomingrefractedcolor ← TRACE(refractedray, cluster)
        incomingreflectedcolor ← TRACE(reflectdray, cluster)
        refractedcolor ← incomingrefractedcolor, normal, material
        reflectedcolor ← incomingreflectedcolor, normal, material
        RETURN refractedcolor + reflectedcolor

LOAD(cluster, index)
IF  index flagged as unshaded
        IF VOIT variation is not storing texture coordinate derivatives
                finished ← false
                        WHILE not finished
                                node ← next node in mergedlist, owned by pixel
                                avgderiv ← Ø
                                FOR neighbornode in mergedlist
                                        IF neighbors (distance+ material  metric)
                                                coord ← texture coordinates of node
                                                ncoord ← texture coordinates of neighbornode
                                                texturederivatives ←  differentiate coord, ncoord
                                                avgderiv ← avgderiv ∪  texturederivatives
                                derivatives← reconstruct avgderiv
        texture ←  texture coordinates, object ID, material ID, like in standard virtual texturing
        inf, sup ←  use texderivatives to find the inf and sup texture mipmaps
        IF inf not stored
                inf ←  closest stored mipmap level
        IF sup not stored
                sup ←  closest stored mipmap level
        physicalcoordinates ←  texture coordinates, inf, sup
        texturedata ←  sample virtual texture with physicalcoordinates
        IF using lighting
                visibility ←  determine visibility through shadow maps or ray tracing over CIV
                color ←  perform illumination with intersected scene lights
        STORE color in place of texture coordinates, flag index as shaded
RETURN position,normal,color ←  node ←  index
```

4.2. Correct Illumination

The correct illumination module is not directly concerned with real-time rendering, as it presents a correct path traced rendering solution. While it is not a real-time solution, the correct illumination pipeline runs interactively on off the shelf hardware, and which will become real-time with more performant hardware.

The correct illumination modules uses ray tracing acceleration structures in order to accelerate the operation of tracing rays, which dominate the rendering time for tracing algorithms [**Hav14**]. While the module uses certain approximations to speed up the rendering process, the light transport is performed exactly, in the limits of photorealistic computer rendering. The algorithms used in this module can run both on the CPU and on the GPU.

The module rendering process contains three large stages: construction and management of ray tracing acceleration structures, light flux importance sampling and bidirectional path tracing.

In the construction and management of ray tracing acceleration structures stage Bounding Interval Hierarchies (BIH) are used for the exact tracing of rays. A variant of conservative inexact voxelization can be used as a secondary acceleration structure, with which rays are quickly tested for surface-cluster interaction before being fully traced. Thus by using two acceleration structures the complexity of tracing a ray is amortized.

In the correct illumination module images are rendered with a modified bidirectional path tracing algorithm which traces rays with amortized complexity. The algorithm uses a novel importance sampling mechanism, named Light Flux Importance Sampling (LFIS), which approximates the flux of light in the scene, and uses this approximation to guide unproductive paths to the vertices of light paths. Compared to the state of the art methods [**Vea97**] [**Cli05**] [**Bir12**] Light Flux Importance Sampling is faster and stores significantly less memory.

The bidirectional path tracer uses both light flux and amortized visibility to quickly produce images, with an algorithm that can run on both CPU and GPU.

4.2.1. Acceleration structures

In tracing rendering algorithms the rendering time is dominated by the visibility determination operations [**Hav14**]. Because of this the acceleration data structures used as support for tracing have to be implemented with the utmost care for performance.

The acceleration structures create scene geometry trees based on either space partitioning or object partitioning, a choice which stems from the fundamental acceleration structure question: to partition the space that contains the objects or to partition the objects into sets. Hybrid structures combine space partitioning with object partitioning. While the 2.2.4 chapter from the state of the art discusses all the acceleration structure topics relevant to rendering, a short comparison is given here, as background for the usage of the Bounding Interval Hierarchy (BIH).

Space partitioning acceleration structures cluster the scene objects based on space subdivision. Rendering space partitioning structures include grids, perspective grids [**Hun08**], hierarchical grids, hierarchical hash-grids [**Sch09**], 1.5D and 2.5D grids [**Har12**], binary space

partitioning trees [**Fuc80**], image space pyramids [**Had98**], quad/octrees, sparse quad/octrees [**Lai101**] [**Sim12**] and kd-trees[**Ben75**] [**Moo91**].

The kd-tree structure is commonly used in rendering due to its performance and advanced partitioning metrics such as the surface area heuristic [**Wal06**], and the binned surface area heuristic [**Dan10**]. The greatest problem of this class of structures it that primitives are partitioned along with space, making the scene geometry trees artificially large and data access coherency suffers. Also the scene geometry trees need to be reconstructed for dynamic geometry. Inexact space partitions, like the one done with grids, suffer from high storage requirements, because they use coherently but inefficiently.

Object partitioning acceleration structures cluster the scene objects based on their shape and proximity, and usually work with bounding volumes. Rendering object partitioning structures include object trees, B-trees, R-tree, sphere trees, AABB trees, bounding volume hierarchies, many partitioning metrics [**Wal072**] [**Ern07**] [**Dam081**] and variants [**Sti09**] [**Pop09**] [**Dam08**] [**Tsa09**], spatial kd-trees [**Ooi87**] [**Zac02**], h-trees and ah-trees [**Hav06**] and bounding interval hierarchies [**Wäc06**]. Object partitioning acceleration structures suffer from overlapping of sibling nodes.

In this thesis the bounding interval hierarchy (BIH) variant of the spatial kd-trees is used as the tracing acceleration structure, a choice motivated by several useful properties of BIH. The BIH is constructed through object partitioning, subdividing objects into potentially overlapping nodes, with two splitting planes. The splitting planes are parallel to one of the dominant axes.

BIH has very fast tracing performance compared to other object partitioning acceleration structures, because it stores its children in an implicit order, thus the tree traversal can quickly access the child closest to the ray origin, similar to how a kd-tree is traced. Furthermore BIH traversal adapts to empty space, as shown in Figure 67.

BIH has the lowest memory footprint out of all the object partitioning algorithms, because it stores only critical data, bitwise compressed, and the majority of the data is determined implicitly (during traversal, from parent splitting planes). It stores the axis and the leaf information in a compressed binary format, cheaply reconstructing each node bounding box on traversal. The total memory cost for a BIH node is only 12 bytes, which is much less than the node cost in other object partitioning structures. This information is depicted in Figure 67.

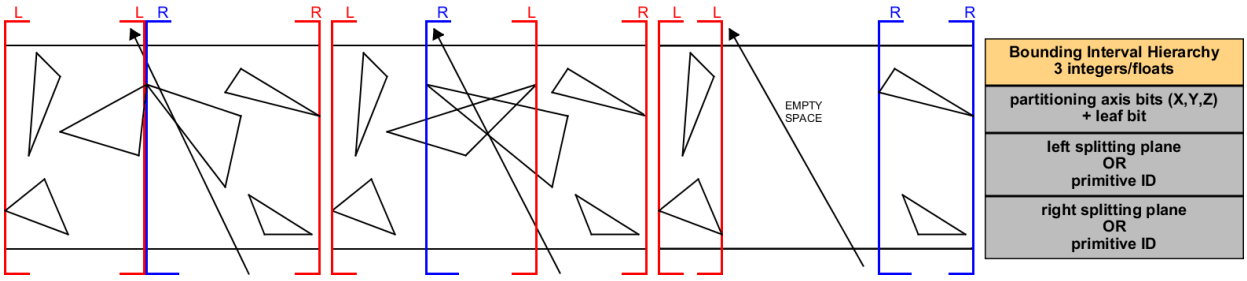

Figure 67 Bounding Interval Hierarchy Traversal**.** Bounding Interval Hierarchy (BIH) leaf nodes store pointers to the scene geometry primitives and the other BIH nodes use two splitting planes to partition the enclosed objects. The splitting planes are parallel to one of the dominant axes, as shown in right side of the image. This splitting strategy results in 3 object partitioning cases: one in which a node is perfectly partitioned, one in which the nodes overlap, and one in which empty space remains between the child nodes. This last case is used to accelerate tracing by skipping the empty space.

Because BIH doesn't need to load object bounding boxes on space traversal, as it computes them implicitly, it exhibits excellent bandwidth consumption and data coherency. Tracing the acceleration structure can be done either trough a traversal which uses a stack, or through a stackless traversal. Stackless traversals are particularly useful on the GPU, since the storage and bandwidth consumption for stack maintenance are very expensive, therefore stackless traversal consume significantly less storage and bandwidth per ray.

Stackless kd-tree traversal techniques [**Fol05**] [**Pop07**] are not suitable for object partitioning schemes because object partitioning nodes overlap [**Lai10**]. [**Lai10**] uses fixed sized buffer for local storage and a bitwise trail mask to represent the current ray tree traversal. Because the local storage for ray traversal is limited, ray traversal is performed through trail restarts, in which the bitmask trail is used to quickly guide the ray through the already traversed path. On the other hand the trail restarts make this method visit more than twice the nodes a stack based traversal would visit. [**Hap11**] uses parent links for each node but this increases storage requirements and needs to re-evaluate traversal order for revisited nodes. [**Afr14**] extends bitwise traversal to MBVH and makes traversal restart-less. Ray stream traversal bundles many rays in a stream and traverses them coherently [**Bar14**]. Acceleration structure-less traversals are a recent development [**Mor11**] [**Kel11**] [**Nab13**] [**Afr12**], but the extra costs during tracing make them inapplicable to interactive tracing.

The BIH tracing pseudocode is based on MBVH2 from [**Afr14**], which uses the while-while kernel introduced by Aila in [**Ail09**] [**Ail12**]. The method uses a 32bit or 64bit bitwise stack, called the bitstack, which stores 0 when the sibling of the current node doesn't need to be traversed and 1 when the sibling has to be traversed. The stack push and pop operators are implemented through binary shifting. The pseudocode for ray traversal is:

```
INTERSECT(ray)
    node ← root
    bitstack ← 0
    WHILE true
        IF node is inner                                    // NODE INTERSECTION
            intersect ray with node children
            IF any child is intersected
                bitstack ← bitstack ≪ 1
            IF a single child was intersected
                node ← that child
            ELSE
                node ← nearest child
                bitmask ← bitmask ∨ 1
            CONTINUE
        ELSE                                                // LEAF INTERSECTION
            intersect ray with leaf primitives
            shorten ray if intersection found
        WHILE bitstack ∧ 1 = 0                              // BACKTRACK
            IF bitstack = 0
                RETURN surface intersection data
            node ← parent of node
            bitstack ← bitstack ≫ 1
        node ← sibling of node
        bitstack ← bitstack ⊕ 1
```

4.2.2. Amortized Visibility Determination

The cost of ray-based visibility determination operations can be lowered through coherent ray-packets [**Bou07**] [**Bou08**] [**Ove08**], which can't be used by path tracing, and through imperfect geometry representations like voxelizations, which make the tracing operation approximated and not analytical. Tracing over voxel representations can greatly decrease the quality of the renderings, if the voxel representations have insufficient resolution.

The idea of amortized visibility is to **combine two acceleration structures**, one exact (analytical) and one approximative (voxel based) and to trace costly exact rays only when the approximative rays shows an apparent clear path between traced points.

Approximative rays can never determine whether a surface-ray interaction takes place, but they can conservatively query the potential ray surface interactions along an already established ray. Thus, amortized visibility can never be used to create new path segments, as this can only be accurately handled analytically. On the other hand amortized visibility can be used for visibility determination queries on already available potential path segments. An example of such usage is to test the visibility between already established entities such as: rays connecting path vertices with light vertices in bidirectional path tracing, rays connecting directly sampled lights with surfaces, and so on.

In such cases amortized visibility is used to ascertain if two points are probably visible or not. If the probability is high enough an exact visibility determination ray is traced over the bounding interval hierarchy, as described in Chapter 4.2.1. The exact ray receives the probable conservative intersection events from the approximative ray, and partitions the ray into multiple segments which are traced together. Therefore, during the single BIH traversal for all the ray segments, not all the BIH nodes that would normally be evaluated have to be visited, only those intersecting the ray segments.

Therefore, in the worst case cost of the amortized visibility for ray tracing, the cost for a ray traced with both approximated and exact tracing methods is approximately equal to the cost of ray fully traced with exact tracing. But the total cost for tracing all the rays in the path tracing algorithm with this amortized technique is lower than the total cost with only exact tracing, because the approximative rays quickly filter out statistically improbable connections.

The correct illumination pipeline can use any voxelization algorithm to create the imperfect representation of the scene geometry. One option is to use the conservative inexact voxelization algorithm (CIV) presented in Chapter 4.1.2.

Because a binary scene geometry representation consumes little memory (134MB for a detailed 1024^3 representation) and it is generated very fast in $(objects)$, with CIV, or $O(primitives)$, with the state of the art, the benefits outweigh the costs.

Tracing with approximative and exact rays is presented in Figure 68 along with a visualization of a BIH structure for a small scene, in which the objects are colored in dark gray. The Bounding Interval Hierarchy is represented with the same color encodings as the one used in Figure 67, red for left splitting planes and blue for right splitting planes. The depth of the BIH nodes is encoded in shades of green, darker being closer to leafs.

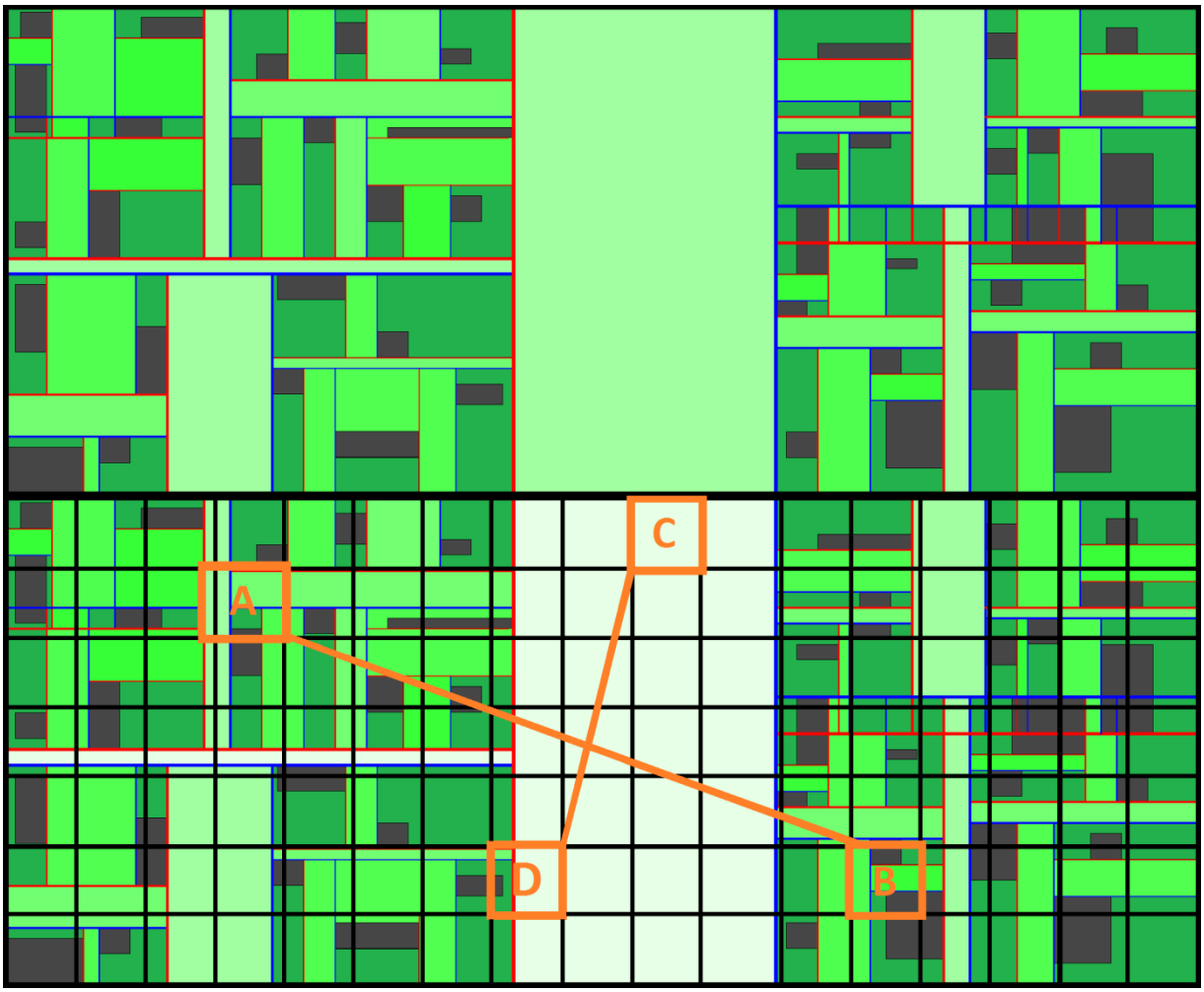

Figure 68 Amortized Rays in 2D**.** In the upper part of the image the Bounding Interval Hierarchy (BIH) for a small scene is displayed. The red lines depict the left partitioning plane and blue lines depict the right partitioning plane, keeping the color coding from Figure 67, in which BIH is presented. Objects are represented with gray. The depth of the nodes is represented with increasingly dark shades of green. The lower part of the image shows approximated tracing over the same scene, using a voxel representation for the scene. This is used to determine if there exists the potential for a clear path between points, for which an exact ray is then traced. If the potential is considered low, as would be for A → B then the costly exact ray would not be traced. If the potential is high, like for C → D, an exact ray is traced, which uses intersection data from the approximated ray, in order to test the intersection with less BIH nodes.

While amortized sampling shouldn't be used on all traced rays, as previously described, when used it can be considered as a form rejection importance sampling, because it pays the costs of the expensive analytic ray-scene intersection test only for the rays which are very likely to collect radiance. Amortized tracing can also be used to approximately compute expensive effects such as participating media.

The voxel representation used for amortized tracing can also be used to compute skeleton importance sampling. On the other hand, Light Flux importance sampling is global importance sampling method which is both faster and more reliable than skeleton importance sampling.

4.2.3. Light Flux Importance Sampling

The real problem with tracing rays is not necessarily their cost, which can only be minimized to a certain extent but the sheer number of rays generated during any path/ray/photon based algorithm. Decreasing requires more sophistication than raw intersection test efficiency and it is done through importance sampling. Importance sampling generates visibility determination rays which are much more likely to explore relevant surface and light interactions, and therefore to accumulate radiance and to be contribute to the final visual result. Radiance accumulation is very importance since path tracing requires many samples to converge, and the noise is easily observable in under sampled images, such as the ones found in real-time rendering, and shown in Figure 71, in the 4.2.4 subchapter.

Importance sampling strategies can be performed at many levels, but a simple taxonomy can be observed: local sampling methods, path sampling methods and global sampling methods. Local sampling methods can be combined through multiple importance sampling [**Vea97**]. Local importance sampling methods include general variance decreasing strategies such as pseudorandom low discrepancy series sampling [**Pha10**], adaptive sampling [**Dam09**], which can be used for a variety of things such as pixel sample generation, BSDF probability distribution function importance sampling, direct lighting [**Pha10**], resampled importance sampling [**Tal05**], which uses existing samples, volumetric sampling [**Kul12**], which can be used to sample rays for scattering events. Eye reprojection [**Hen11**] and radiance filtering [**Sch12**] and radiance filtering can also be considered local sampling methods.

Path space importance sampling methods include metropolis light transport [**Vea97**], primary space metropolis light transport [**Kel02**], energy redistribution [**Cli05**], manifold exploration [**Jak12**], gradient domain metropolis [**Leh13**] and multiplexed metropolis [**Hac14**]. All these methods use different mutation strategies on the entire path, to generate new radiance rich mutated paths. Vertex connection merging [**Geo12**] and path space regularization [**Kap13**] can also be considered importance sampling methods, as they try to maximize the connectivity of subpaths generated with bidirectional path tracing [**Laf93**]. Light field reconstruction [**Leh11**] can also be considered a path space importance sampling method as it reconstructs paths from light fields, basically importance sampling the path space of previously traced paths. The purpose of path space algorithms is to find productive paths, even in hard to sample light transport situations. Path space algorithms can inherently sample local events (surface interaction, scattering, etc.).

Global importance sampling are based on skeleton importance sampling methods [**Bir12**] [**Cha13**], which globally explore the scene to find potentially radiance rich areas. The strategy of this importance sampling family is to find the most productive empty space, which can then be easily linked to the lights and camera. The productive empty space is usually near the skeletonization of the empty space in the scene. Because the strategy is global, it is much more efficient in transporting light in extremely difficult scenes like those containing holes or barely opened doors. Bidirectional Importance [**Laf93**] sampling can be itself considered a global importance sampling method, because it samples paths from both camera pixels and scene lights and links the generated subpaths into full light transport paths.

Light flux importance sampling (LFIS), also named light flow importance sampling, is a **new global importance sampling**, designed to be used with bidirectional path tracing. The idea is **inspired by flow maps**, which are maps that describe the flux of fluids. The light flux map is

extremely similar as a concept, as it describes the flux of light in the scene. Light flux importance sampling differs from the state of the art by creating a light flux map (LFM) which can be queried to quickly determine a source of light in any area of the scene. The source of light can be any scene light or any light vertex generated by light tracing.

The light flux map is implemented as a tridimensional grid. It is populated by generating a small number of samples for each light and generating light paths for these samples. The light paths segments are voxelized, inside the tridimensional grid of the light flux map. Each cluster over which a light path is rasterized holds a reference to the scene light or light vertex from which the light originated. If a cluster is already populated, the radiance coming from the light vertex source of the newly voxelized path segment is compared to the one coming from the stored light vertex, and the vertex with the greatest radiance is kept. The light flux map thus contains sparse information about the scene light transport. A **tridimensional push-pull** process, akin to the one used in Conservative Inexact Voxelization in chapter 4.1.2 is used populate the entire map, guaranteeing a light connection for each cluster of the light flux map. A bit flag is kept for the approximated entries.

Thus, the light flux map enables indirect light importance sampling, by being able to link any light vertex spawned by random walks originating from the light. The light sources are modified to store sufficient data in order to evaluate the radiance for the entire path from which they originated. For each camera path vertex generated inside the light flux cluster, a linkage is generated to one of the referenced light vertices.

In the light tracing pass of the bidirectional path tracing algorithm, the light flux map entries which were flagged as approximated are updated when a light is found inside the flagged entry, either as a light segment or as a vertex. After a true, un-approximated, reference is stored in an entry, the entry is flagged as final and will not be sampled anymore. Thus, while the light flux map starts as a raw approximation of the scene it converges to an exact solution. Therefore, light flux enables a very fast connection between the path tracing vertices and light tracing vertices generated in bidirectional path tracing, which brings BDPT closer to real-time rendering.

Compared to path importance sampled path tracing algorithms, such as metropolis light transport [**Vea97**], primary space metropolis light transport [**Kel02**], energy redistribution [**Cli05**], manifold exploration [**Jak12**], gradient domain metropolis [**Leh13**] and multiplexed metropolis [**Hac14**], the light flux method is much faster in producing paths that contribute to the visual results, and it does not need any complex, storage and computational intensive mutation process. Light flux has minuscule storage costs as compared to vertex connection merging [**Geo12**] and path space regularization [**Kap13**]. Compared to the other global importance sampling method, skeleton importance sampling [**Bir12**] [**Cha13**], the light flux map is much more exact because it directly samples light paths, and it adapts to the illumination conditions of the scene. Skeleton importance sampling samples the scene based on the supposition that high energy will be concentrated in the empty areas of the scene, but this estimate isn't always accurate and it often leads to an oversampling of the empty space of the scene.

A weakness of the presented method is that the map can generate bad light linkage when it is extremely under sampled, like right after the sampling stage, such as linking to a light that is directly occluded. The number of such events is extremely small compared to the number of useful connections, and the map converges rapidly. The light flow sampling algorithm is displayed in Figure 69.

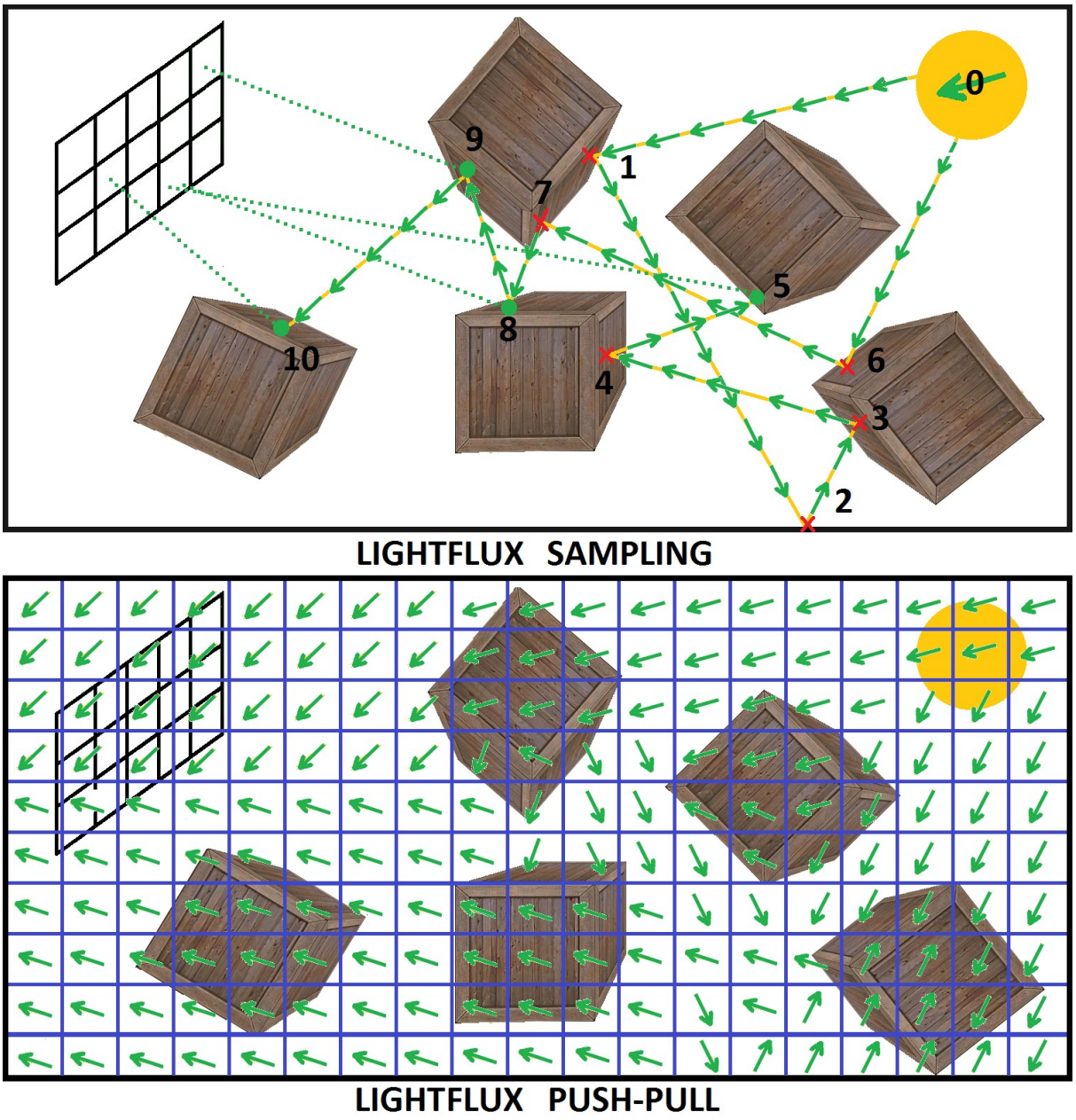

Figure 69 Light flux importance sampling**.** This algorithm stores light flux information inside a tridimensional grid, where each entry references the light vertex bringing the most radiance, here shown as the direction from it. Initially the algorithm starts with a seeding process, in which a small number of light paths is generated for each light and the generated light path segments have their flux information stored in the flux map, as depicted in the upper image. The stored data is then interpolated in a tridimensional push-pull process, generating an approximated flux of light for all entries, as depicted in the lower image. The upper image also presents an optimization of the light flux algorithm, which can be used to better generate light paths in the light tracing pass of the BDPT. Each light stores the direction of the seeding sample which generated the most light vertices which were visible from the camera. The difference between the best direction seeding light path and the rest of the seeding light paths is shown in the surface contacts: the best direction light path has green surface contacts while the other light path has red surface contacts. The direction of the best light path is then importance sampled during light tracing.

The pseudocode for the light flux map creation is the following:

```
PREREQUISTE
create voxelization for approximated tracing, can use CIV
allocate lightfluxmap and lightfluxmipmaps

LIGHTFLUX SEEDING (scenelights)
Reinitialize lightflux,  flag all clusters to empty
scenelights ← Ø
FOR light in scene IF light is considered relevant (radiance, distance to camera)
        scenelights ← scenelights + light
FOR light in scenelights
    samples ← generate a number of samples with pseudo random series (Halton/Sobol/vdCorput)
    bestdirection ← direction of first sample
    maxssprojections← 0
    FOR sample in samples
        lightvertex ← sample
        numssprojections, recursion depth ← 0
        ray ← sample position, sample direction
        WHILE recursiondepth < Threshold
            surfacehit ← trace until next surface contact
            clusters ← clusters from lightfluxmap which were traced over
            FOR cluster in clusters
                IF cluster empty
                    cluster ← radiance and reference to previous lightvertex
                    cluster ← flag contents to exact
                ELSE
                    storedradiance ← radiance of stored in cluster
                    radiance ← previous lightvertex radiance
                    IF radiance > stored radiance
                        cluster ← radiance and reference to previous lightvertex
            lightvertex ← generate new light vertex, at surfacehit
            ray ← ray, surfacehit data
            projection ← randomly project  lightvertex to screen space
            IF projection is unoccluded
                numssprojections ← numssprojections+1
        IF numssprojections > maxssprojections
            bestdirection ← current sample direction
            maxssprojections ← numssprojections
RETURN lightfluxmap (sparse)

LIGHTFLUX PUSHPULL (lightfluxmap)
mipmaplvl ← 0
WHILE mipmaplvl < (highest lightflux mipmaplvl-1)
    FOR cluster (texel) in mipmaplvl+1
        lightvertex← light vertex with most radiance among the four children from mipmaplvl
        cluster←  lightvertex
    mipmaplvl← mipmaplvl+1
mipmaplvl← highest lightflux mipmaplvl
WHILE mipmaplvl > 1
    FOR cluster(texel) in mipmaplvl
        FOR childcluster (texel in mipmaplvl-1) of cluster IF childcluster not exact
            childcluster ← store the lightvertex from cluster
mipmaplvl← mipmaplvl-1
RETURN lightfluxmap (approximated)
```

```
LIGHT FLUX ADDITION DURING LIGHT TRACING
FOR lightpath in paths generated by light tracing
    IF using only light vertices
        extrasamples ← vertices in lightpath
    ELSE
        extrasamples ← vertices in lightpath, generate vertices over lightpath segments
    clusters ← lightflux clusters intersected by extrasamples
    FOR cluster in clusters IF cluster not exact
        cluster ← radiance and reference to previous lightvertex
        cluster ← flag contents to exact
```

Light flux importance sampling is compared to the state of the art importance sampling methods in Table 12. Light flux importance sampling can be combined with other importance sampling methods through multiple importance sampling [**Vea97**].

Importance Sampling Method & Sampling Space(s)	Efficient with bad starting samples	Computation Complexity	Required Storage	Requires Vertex Mutations	Efficient Specular Path Exploration	Explicit knowledge of light data
adaptive sampling (local/general)	low	low	low	no	no	no
BSDF sampling (local)	no	low	none	no	yes	no
direct lighting (local)	no	low	none	no	no	yes
resampled importance sampling (local/existing samples)	no	low	low	no	no	no
volumetric sampling (local, path)	no	medium	low	no	no	no
eye reprojection (local, camera)	no	low	low	no	no	no
radiance filtering (local, camera, path)	no	low	medium	no	no	no
temporal light field reconstruction (local, camera, path, temporal)	no	high	medium	yes	no	no
metropolis light transport (local, path)	medium	high	high	yes	medium	no
primary space metropolis transport (local, path)	high	high	high	yes	medium	no
energy redistribution (local, path)	high	medium	high	yes	medium	no
manifold exploration (local, path, manifold)	medium	high	high	yes	v. high	no
gradient domain metropolis (local, path, gradients)	high	high	high	yes	high	no
multiplexed metropolis (local, path)	high	high	high	yes	v. high	no
vertex connection merging (local, path)	high	high	high	no	high	no
path space regularization (local, path)	high	high	high	no	high	no
Bidirectional (local, path, global)	low	medium	high	no	no	yes
Skeleton (local, global)	medium	low	medium	no	no	no
Light Flux - this algorithm (local, path, global)	high	low	medium	no	no	yes

Table 12 Path tracing sampling strategies**.** The table compares the Light Flux with other importance sampling algorithms or processes. Light Flux importance sampling exhibits desirable properties for fast light transport, with the exception of the exploration of difficult specular paths, which need special importance sampling mechanisms.

4.2.4. Bidirectional Path Tracing

Bidirectional path tracing (BDPT) is the path tracing variant with the highest quality that does not need heavy path mutation in order to transport light, and is therefore sufficiently compatible with many core architectures in order to obtain low interactivity levels. The bidirectional path tracing algorithm runs in two stages: the light tracing stage and the path tracing stage. The light tracing stage samples the lights and generates light paths through the scene, directly sampling the lights. Usually, in BDPT the path tracing stage samples the camera and generates camera paths through the scene. If the GPU implementation of the current pipeline is used, the path tracing stage can use the geometry-buffer data for the first camera path vertex, which introduces a tiny amount of bias but speeds up rendering and eases the integration between the geometry processing chapter and the correct illumination pipeline.

The generated light and camera paths are then combined, creating full light transport paths, from the scene lights to the camera. Both the lights and the camera pixels are sampled with pseudorandom low discrepancy series like Halton series [**Pha10**] and are stochastically terminated with Russian Roulette [**Pha10**]. The connection between the two stages is done with an acceleration structure, like a hierarchical cluster grid or a spatial hierarchical hash gird [**Sch09**], as used in [**Geo12**].

The bidirectional path tracing algorithm can be integrated tightly with the previously presented Light Flow importance sampling mechanism. As depicted in Figure 69, the seeding stage of the light flow map can be used to approximately determine the most productive sampling direction per light, by using the direction of the sample that generated the path with the most screen space visible light vertices. The computed direction can be then used as a minor importance sampling mechanism, which generates extra light paths around the determined productive direction.

The presented correct illumination pipeline is based on a bidirectional path tracer that uses light flux importance sampling to explicitly and indirectly sample the scene lights and bidirectional scattering distribution function (BSDF) importance sampling, which creates samples based on probability distribution of the BSDF. These sampling mechanisms are combined with MIS [**Vea97**], as they would otherwise decrease local variance but increase global variance. The combination function is the power heuristic introduced by [**Vea97**]:

$$L_{MIS} = \sum_{i=1}^{m} \frac{1}{n_i} \sum_{j=1}^{n} w_i(X_{i,j}) \cdot \frac{f(X_{i,j})}{p_i(X_{i,j})} \qquad \sum_{j=1}^{n} n_i = N \qquad w_i(X_{i,j}) = \frac{[n_i \cdot p_i(x)]^{\beta}}{\sum_{k=1}^{n} [n_k \cdot p_k(x)]^{\beta}}$$

Where m is the number of importance sampling strategies, n_i is the number of samples for each sampling strategies, $f(X_{i,j})$ is the estimated function, $p_i(X_{i,j})$ is the sampling probability, and $w_i(X_{i,j})$ is the power heuristic function. If $\beta = 1$, the resulting weighting function is called the balance heuristic.

Adaptive importance sampling, eye reprojection importance sampling and radiance filtering are used to importance sample data at the camera level. The BDPT algorithm is visually presented in Figure 70, with the most relevant sampling methods.

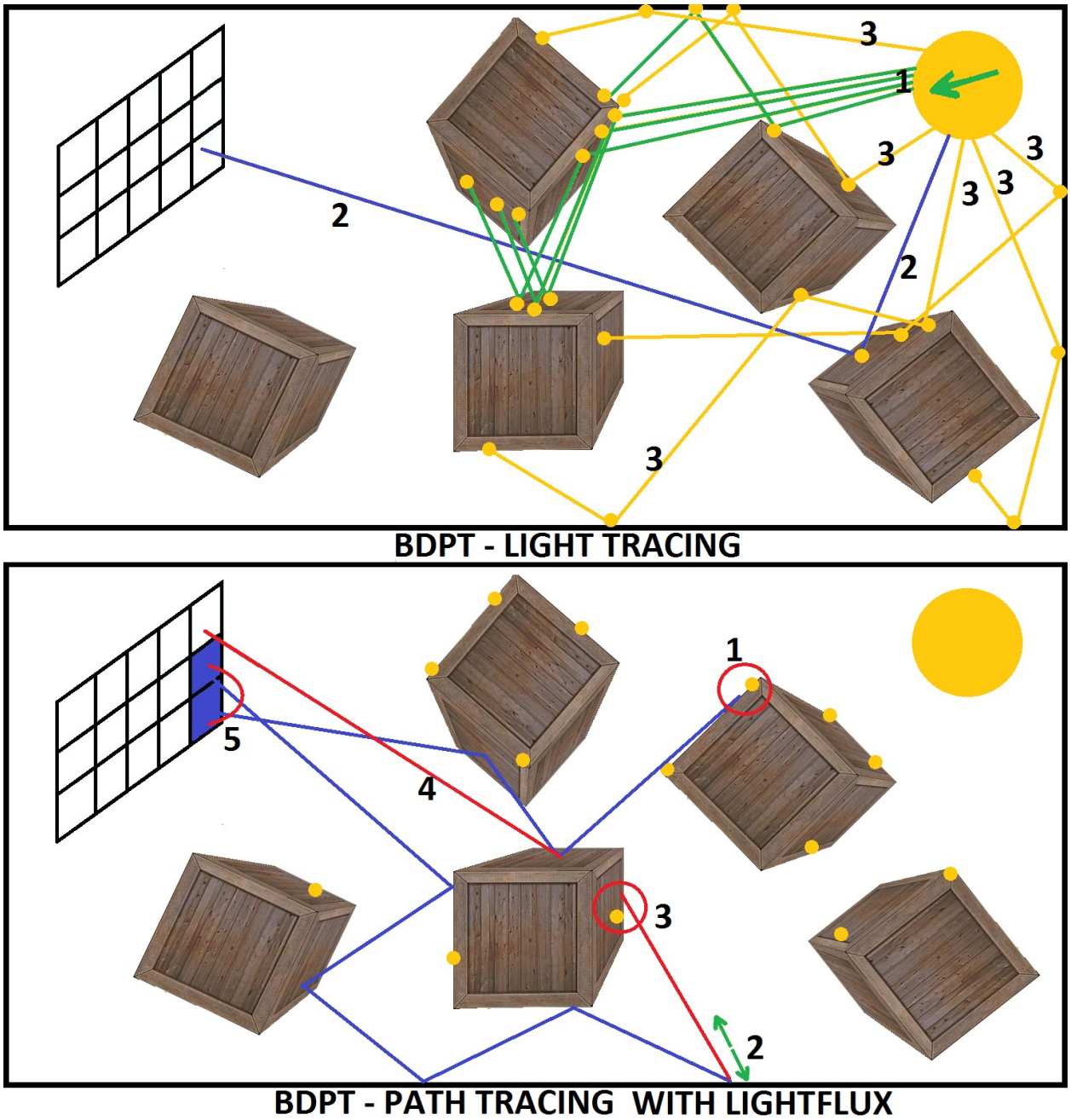

Figure 70 Bidirectional path tracing with light flux**.** In the upper part of the image, the light tracing pass of BDPT is displayed, which stores light vertices in an acceleration structure, shown in yellow. The light tracing algorithm sometimes generates lucky samples that directly light the screen, as shown in blue. The light tracing pass is importance sampled with the most productive light direction found in the light flux map construction step, shown in green. The bottom part of the image displays the path tracing pass of BDPT, which is explores the scene through camera paths (blue), which collect radiance. Different radiance collection algorithms are used, shown above with numbers 1,2,3,4,5. 1 and 3 are direct connection between a vertex path and a light path. 2 is the light flux importance sampling which is used on unproductive paths to quickly find light connections (3). 4 shows eye reprojection, in which the vertices of the traced camera paths are projected on the screen, along with their collected radiance. 5 displays radiance filtering, in which the pixels directly sample the first camera path vertices from neighboring paths.

While not used in this thesis, the quality of the connection between light tracing and path tracing can be further improved with connection streams, like those used in streaming path tracing [**van11**], a modified sampling strategy such as the one presented in [**Bog13**] or through path regeneration [**Nov10**]. Path mollification [**Kap13**] and vertex connection merging [**Geo12**] can be used for offline rendering.

The presented bidirectional path tracer can use two strategies for lightflux map extra sampling. The first strategy is to fill the approximated lighflux map entries in the vicinity of light vertices generated in the light tracing stage. The second, more exact but much more expensive strategy is to fill all the approximated lightflux map entries which enter in contact with light paths from the light tracing phase. This can be achieved through tridimensional segment rasterization over the lightflux map, and will produce a high quality lightflux map. The pseudocode for the light flux enabled light tracing BDPT algorithm is the following:

```
BDPT LIGHT TRACING
hashgrid ← allocate spatial hierarchical hash grid, hashgrid ← Ø
scenelights ← all scene lights
lightfluxmap ← LIGHTFLOW SEEDING with scenelights                    //LIGHTFLUX SEEDING
lightfluxmap ← LIGHTFLOW PUSHPULL lightflowmap
WHILE scenelights not empty                                          //GENERATE SAMPLES
        light ← pop scenelights
        samples ← generate samples for light
        lightflow_direction_samples ← generates samples for light using lightflow light direction
        samples ← samples ∪ lightflow_direction_samples
        WHILE samples not empty                                      // LIGHT TRACING
                sample ← pop samples
                pathlength ← 0
                path ← sample
                previousvertex ← sample
                ray ← create ray with light sample
                WHILE pathlength < max path level                    // MAX LENGTH
                        surface_intersection_data ← INTERSECT ray
                        vertex ← create light vertex with surface_intersection_data
                        store vertex in hashgrid
                        path ← path ∪ vertex
                        pathlength ← pathlength + 1
                        bsdfsample ← sample BSDF
                        ray ← create ray with bsdfsample
                        IF lightfluxmap is sampled during light tracing   // LIGHTFLUX SAMPLES
                                IF sampled with vertices
                                        cluster ← lightflux cluster for vertex
                                        IF cluster approximated
                                                cluster ← previous vertex, flag as not approximated
                                ELSE IF sampled with segments (very expensive!)
                                        segment ← vertex ⋈ previousvertex
                                        emptyclusters ← INTERSECT segment with lightflux
                                                approximated clusters
                                        WHILE emptyclusters not empty
                                                cluster ← pop emptyclusters
                                                cluster ← previous vertex, flag as not approximated
                        previousvertex ← vertex
                        IF random (0,1)< extinction probability      //RUSSIAN ROULETTE
                                BREAK
```

The camera path tracing stage of the BDPT algorithm samples the scene like a normal path tracer, but also looks to connect each path vertex to a light vertex generated by the light tracing pass, in order to created complete light transport paths. This connection is done through explicit paths, in which a camera path vertex directly connects, or is in a very close vicinity of a light vertex, or through implicit paths, which use connectors suggested by the light flux map.

Russian Roulette (RR) [**Pha10**] is used to stochastically terminate long paths, and it is an unbiased estimator since each path that survives the Russian roulette is weighted in order to compensate in expected value, as done in this equation:

$$E[PATH_{rr}] = p_{exctinction} \cdot 0 + (1 - p_{exctinction}) * \frac{E[PATH]}{(1 - p_{exctinction})} = E[PATH]$$

Adaptive importance sampling can be used to generate more camera paths, especially for paths starting in pixels prone to aliasing. These pixels can be determined with gradients like in [**Leh13**], which are extremely cheap to obtain if the camera path tracer is implemented starting from the deferred G-buffer. Another adaptive algorithm is based on the hierarchical automatic stopping condition presented in [**Dam09**], in which the screen is subdivided into tiles, over which path tracing phases are performed. After each path tracing phase the newly created tile image is compared to the existing tile image and if the error is under a very small threshold the tile is terminated. Otherwise, if the error is small enough the tile is split into multiple tiles, which then follow the same hierarchic process. If the error is large, the tile is not split, the radiance from the current path tracing pass is accumulated and a new path tracing pass over the tile is initiated. This technique is adapted to the correct illumination rendering pipeline, but instead of a hierarchic process which requires CPU synchronization a simpler micro-tile only structure is used, where samples are generated for each micro-tile until convergence is obtained. The error metric used by [**Dam09**] is adapted to:

$$error_{tile} = \frac{1}{N} \cdot \sum_{k=0}^{N} \frac{|I_k^r - A_k^r| + |I_k^g - A_k^g| + |I_k^b - A_k^b|}{\sqrt{I_k^r + I_k^g + I_k^b}}$$

Where N is the total number of pixels in the tile, I_k^r, I_k^g and I_k^r are the values for the red, green and blue channels which were determined in the last path tracing pass over the tile and A_k^r, A_k^g and A_k^b are the already accumulated values for the red, green and blue channels.

Eye reprojection is a cheap performance improver, which uses the existing camera paths bring additional radiance to the camera, by projecting each camera path vertex onto the screen and transporting radiance. In this thesis eye reprojection is used only at micro tile level, in order for this algorithm to be compatible with adaptive sampling and to avoid GPU races. Radiance filtering follows the same principle of sample re-usage as eye reprojection, but instead of projecting camera path vertices onto the screen it performs micro path mutations on the paths created by the neighbor camera path tracing samples. The neighbor paths are mutated to terminate on the current sample and not on the neighbor sample, re-using the radiance transport computed by the neighbor samples. Both sampling strategies are displayed in Figure 70. Sub pixel reconstruction for ray-tracing can be used if the BDPT is implemented with a deferred renderer [**Chi12**].

The pseudocode for the BDPT camera path tracing stage is:

```
BDPT PATH TRACING
scenelights← scene lights
lightfluxmap, hashgrid← BDPT LIGHT TRACING
microtiles ←separate the screen into microtiles
FOR tile in microtiles                                          //ADAPTIVE SAMPLE
        converged ← false
        sampler ← create Halton series low discrepancy sampler from camera and pixel
        FOR pixel in microtile
                pixelaccumulatedcolor ← 0
        WHILE not converged
                samples ← generate camera samples with sampler in microtile
                projectedsamples← Ø
                FOR sample in samples                           //TRACING
                        pathlength ← 0
                        path ← sample
                        ray ← create ray with sample
                        samplecolor ← 0
                        previousvertex ← sample
                        contribution ← 1
                        WHILE pathlength < max path level       //MAX LENGTH
                                surface_intersection_data ← INTERSECT ray
                                vertex ← create light vertex with surface_intersection_data
                                path ← path ∪ vertex
                                pathlength ← pathlength + 1
                                entry ← hashgrid entry for vertex
                                IF entry not empty              //EXPLICIT LINK
                                        connected ← false
                                        index ← random index from 0 to entry size
                                        WHILE not connected
                                                lightvertex ← select light vertex at index
                                                lightray ← ray from vertex to lightvertex
                                                IF no INTERSECT lightray
                                                        samplecolor ← contribution, vertex, lightvertex
                                                        BREAK
                                                ELSE
                                                        index←index+1
                                normal, bsdf, mis ← surface_intersection_data
                                IF lightflux importance sampling        //INDIRECT LINK
                                        lightfluxcluster ← get lightflux cluster        //MIS LIGHTFLUX
                                        light← get light from lightfluxcluster
                                        lightray ← ray from vertex to lightvertex
                                        IF no INTERSECT lightray
                                                Lightfluxestimator ← MIS
                                                samplecolor ← contribution, vertex, lightvertex
                                                BREAK
                                IF direct light importance sample       //MIS DIRECT LIGHT
                                        light← pick a light from scenelights
                                        lightray ← ray from vertex to sampled point on light
                                        IF no INTERSECT lightray
                                                lightestimator ← use ray, normal, mis
                                                directcolor ← evaluate path & lightvertex
                                                samplecolor←samplecolor + lightestimator *directcolor
                                IF bsdf importance sampling             //MIS BSDF
                                        bsdfestimator, ray ← use ray, normal, mis to sample bsdf
                                        contribution ←  contribution · bsdfestimator
                                IF eye reprojection && samplecolor>0    //EYE REPROJECTION
                                        camerasample ← sample camera in microtile
```

```
                projectedvertex ← project previousvertex on camerasample
                projectedray ← create ray from previousvertex to projectedvertex
                IF no INTERSECT projectedray
                        projectedcolor ← projectedray, previousvertex, vertex
                        projectedsamples ← projectedsamples ∪ projectedcolor
            IF random(0,1) < extinction probability      //RUSSIAN ROULETTE
                BREAK
            ELSE
                contribution ← contribution · 1/(1−random)
            previousvertex ← vertex
        END WHILE
END FOR

FOR sample in samples                                    //RADIANCE FILTERING
        FOR neighborsample in samples
            path← path of neighborsample
            firstvertex←first vertex in path
            secondvertex←first vertex in path
            projectedray ← create ray from sample to firstvertex
            IF no INTERSECT projectedray
                projectedcolor←projectedray, firstvertex, secondvertex
                projectedsamples ← projectedsamples ∪ projectedcolor
pixeltotalcolor ← 0
pixeltotalweight ← 0

FOR pixel in microtile                                   //SAMPLE FILTERING
        FOR sample in microtile
            sampleweight ← determine sample weight, based on screen space distance
            pixeltotalweight← pixeltotalweight + sampleweight
            pixeltotalcolor ← pixeltotalcolor + samplecolor
        IF pixeltotalweight > 0
            pixelcolor ← pixeltotalcolor / pixeltotalweight
        ELSE
            pixelcolor ← 0
        IF using deferred renderer                       //SRAART
            extrasamples← sub pixel reconstruction antialiasing
            pixelcolor← average with extrasamples
        pixelerror ← compute with error between pixelcolor and accumulatedcolor
microtileerror← 0
numpixels← number of pixels in microtile

FOR pixel in microtile                   //ADAPTIVE SAMPLING CONVERGENCE
        pixelerror ← error or pixel
        microtileerror ← microtileerror  + pixelerror
microtileerror← microtileerror/ numpixels
IF microtileerror < convergence threshold
        converged←true
ELSE                                     //ADAPTIVE SAMPLING ACCUMULATION
        FOR pixel in microtile
            FOR sample in pixel
                accumulate samplecolor in pixelaccumulatedcolor
```

The BDPT algorithm with the presented sampling algorithm creates the images in Figure 71 interactively. The figure also shows that the light flux importance sampling method speeds up light transport, especially for difficult to sample global light paths.

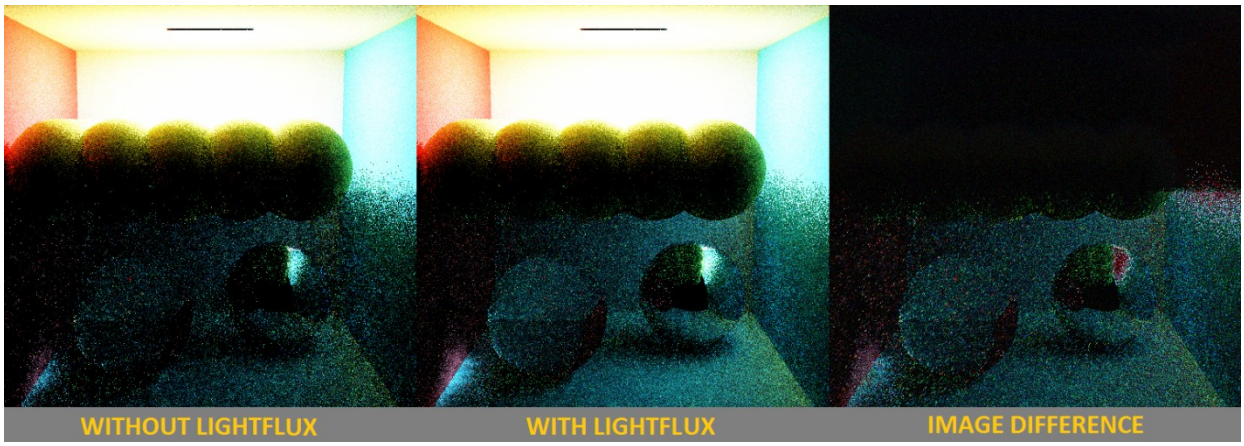

Figure 71 Interactive BDPT with light flux importance sampling**.** The presented bidirectional path tracer produces the above images interactively, on consumer hardware GPUs. The images show that light flux sampling is very productive for all light paths, as it generates light-camera paths very fast which ensure faster image convergence.

Figure 72 presents results obtained with rendering times outside interactivity.

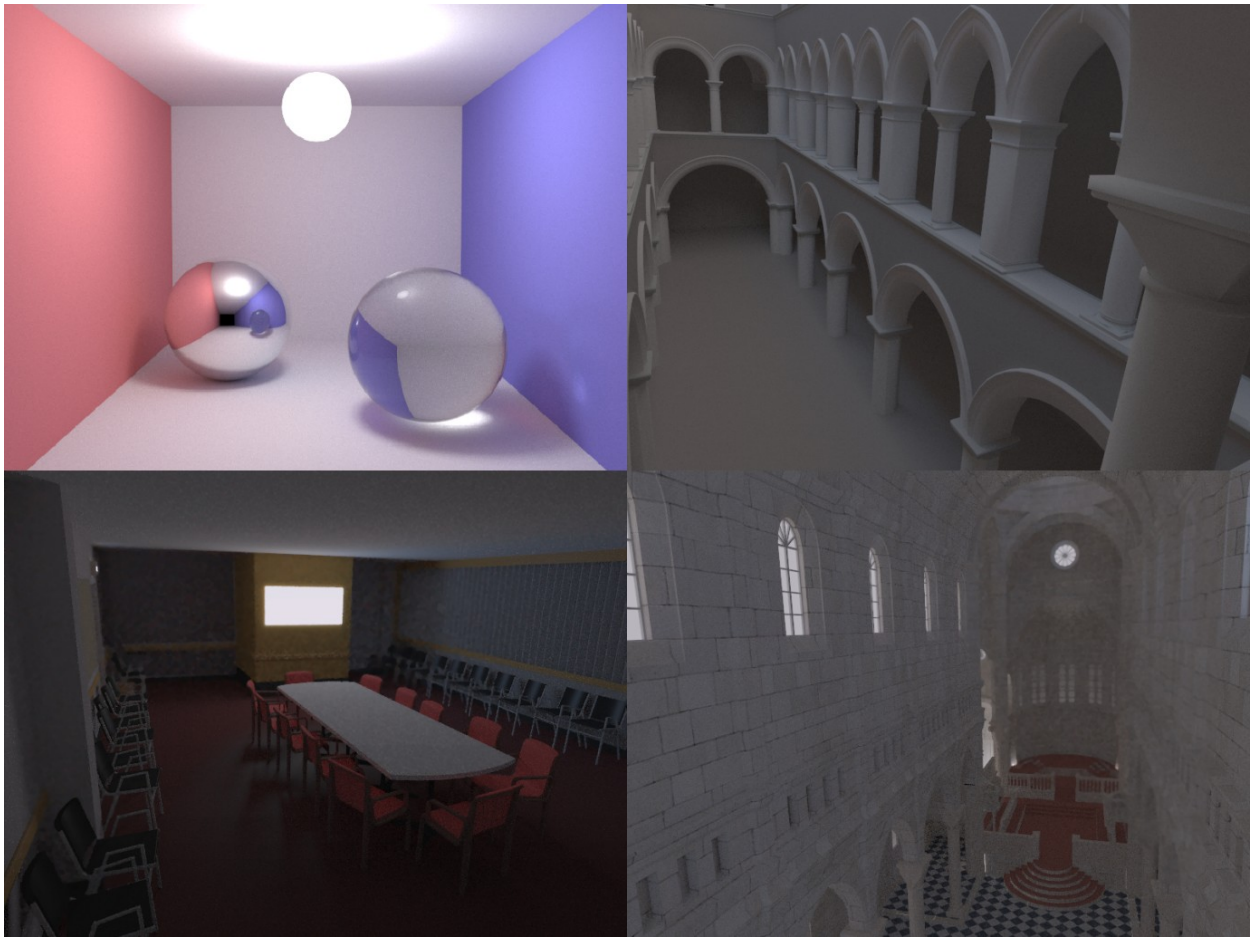

Figure 72 Offline bidirectional path tracing**.** The presented correct illumination can be used in offline mode to produce photorealistic results. Classic global illumination rendering scenes are shown in the figure.

Due to the very low memory usage and the fast connection between light vertices and path vertices, the light flow importance sampled bidirectional path tracing is an algorithm

designed for the GPU, which can be used to produce interactive previsualisations for offline rendering.

Fast and accurate previsualisations are critical in CAD modeling, because the final renderings use extremely large sample counts and are thus too expensive for interactive modeling.

4.3. Post Processing

Post processing is performed independently of the illumination rendering pipeline, and both the approximate illumination pipeline and the correct illumination pipeline can be post processed. The post processing stage implements many filters, which improve the visual result of the rendering by either enhancing it perceptually or by augmented the rendered image. Some of the implemented filters are shown in Figure 73.

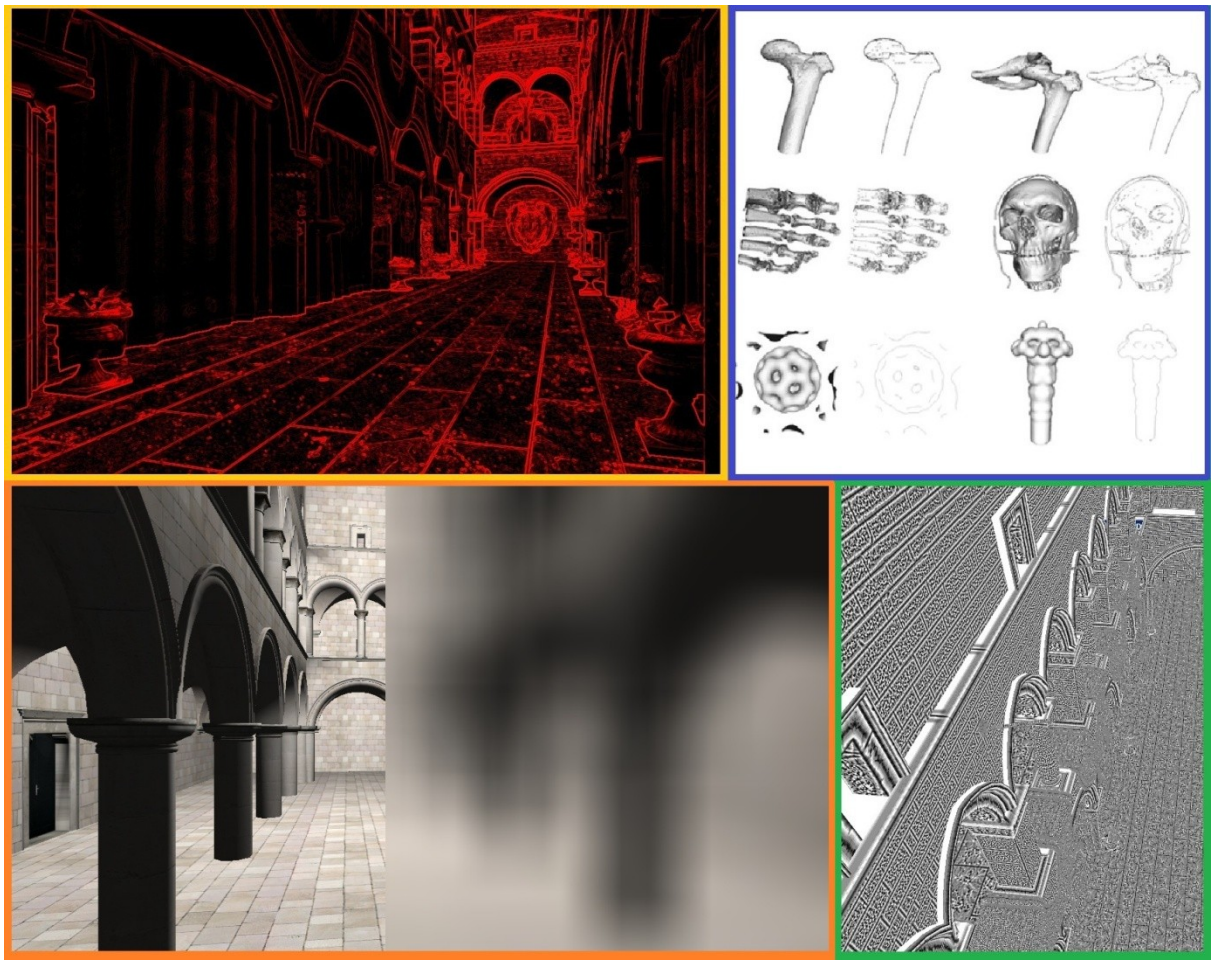

Figure 73 Post processing**.** This image shows various post processing algorithms such as edge detection in the upper left corner, perception enhancing in the upper right corner, micro ambient occlusion in the right lower corner. Because the pipeline is GPGPU computed, it can benefit from explicit caching methods, which enable the implementation of very wide kernels, such as the blur kernel used in the lower left corner.

The post processing stage works as a completely GPGPU stage, which uses outputs produced by illumination pipeline, such as depth, normals or color. These values are read into

tiles, which act as small caches, and which greatly speed up data access, because all the filters are implemented as kernels over some vicinity of image space.

The filters implemented in the post processing module are tone mapping, high dynamic range rendering, gamma correction, depth of filed, edge detection, motion blur and perception enhancing. They can be categorized into two types: perception enhancing filters and rendering enhancing filters. The rendering enhancing filters are particularly important, because they improve the approximated illumination visual results with motion blur and bokeh depth of field, as described in [**McI12**] and [**Gue14**].

A notable omission from this sub-chapter is the decoupled sub pixel reconstruction anti-aliasing algorithm, which is a post processing algorithm seamlessly integrated in the deferred-based in the approximated rendering illumination pipeline.

5. CONCLUSIONS

This thesis has introduced a modular rendering pipeline based on several novel real-time and interactive rendering techniques. The presented real-time algorithms are based on the principles of decoupling and bandwidth reduction, and can handle a large number of light paths. The correct rendering path runs at the edge of interactivity. The contributions are presented in detail in chapters 3 and 4.

The most important contributions to real-time rendering which are presented in this thesis are the virtual rasterization and antialiasing algorithms for opaque and transparent objects, the hierarchical culling algorithm, the conservative inexact voxelization method and its applications to approximative low and high frequency light transport. Other contributions include a variant of the marching cubes algorithm specialized for large datasets, a hierarchical impostor system using virtual texturing, a small rasterization task generator, measurement metrics for deferred algorithms, an approximative order independent transparency algorithm and a geometry selection method.

The correct rendering path introduces light flux sampling, a novel global importance sampling method, which adapts to scene light transport faster than the other global importance sampling methods. The correct rendering path also amortizes the cost of some of the visibility determination rays, by using the conservative inexact voxelization algorithm.

The rest of the chapter continues with a short description for each introduced algorithm and with an overview of the potential research directions that continue the ideas discussed in this thesis.

5.1. Summary of Contributions

In this small sub-chapter each contribution is shortly described.

The indirect rendering solution used in this thesis is based on a serialization method for the marching cubes algorithm, which enables running the algorithm on the GPU, for very large datasets. The algorithm is described in chapter 3.2.1, and also published in [**Pet11**].

A hierarchical impostor method is presented in chapter 3.2.3, which creates impostors for entire scene tree nodes. Compared to state of the art methods, this technique operates at scene level and it is integrated within a virtual texturing based streaming system while also being parallax mapping aware. It is then used for scene rendering scalability and anti-aliasing.

This thesis introduces a GPU task generator, which, compared to all the other state of the art task generators and schedulers, is capable of working within the rasterization thread scheduler. Because of this, the task generator can be used to augment the performance of rasterization-based rendering algorithms. This task generator is described in chapter 3.3, and published in [**Pet14**]. It is then applied to a hierarchical view frustum culling algorithm in chapter 3.4. The task generator is used to generate and solve culling tasks which appear during the culling scene tree traversal.

The multi frame culling algorithm culls objects hierarchically, over multiple frames without CPU control besides the initialization synchronization, and without any pre-processing like occlusion impostors. On the other hand if such impostors are available, it can be integrated with hierarchical depth occlusion to benefit from. The algorithm can be also be integrated with other hierarchic or non-hierarchic algorithms, such as hierarchical depth occlusion, though it can benefit from the latter if it is available. The algorithm can also be implemented using dynamic parallelism. Moreover, the method introduces a multi-frame culling mechanism based on the solid angle obtained from the camera orientation. By culling objects for multiple frames, the algorithm significantly lowers computational costs.

Novel measurement metrics for deferred algorithms are provided in this thesis, in chapter 3.5.1, and they are used to compare all the relevant deferred and decoupled algorithms. They are also published in [**Pet15**]. By using these methods a developer can easily select the most suitable deferred algorithm, depending on the constraints of the rendering problem and on the deployment hardware strengths and weaknesses.

This thesis introduces virtual rasterization rendering methods for both opaque and transparent objects. Both algorithms use virtual texturing to completely decouple texture fetching from geometry processing.

The opaque objects are rasterized with virtual deferred (VD), presented in Chapter 3.5.2. The shading part is presented in Chapter 4.1.4. Virtual deferred is a novel type of deferred algorithm, which combines the benefits of single geometry pass deferred rendering with the useful properties of virtual texturing and multi-pass deferred algorithms. In doing so, virtual deferred possesses the best deferred metrics for bandwidth, shading and geometry processing, while also being easy to incorporate in a streaming solution. The method was published in [**Pet151**].

Decoupled sub pixel reconstructed anti-aliasing (DSRAA) is a new method which is inspired by the sub geometric reconstruction anti-aliasing (SRAA), improving it by performing the sub-geometric reconstruction as a sample matching method. This antialiasing method does not introduce further storage and bandwidth costs and has cheaper reconstruction costs than SRAA. It is also used in combination with virtual deferred, to render antialiased rasterized opaque objects, as discussed in chapter 4.1.5.

Virtual deferred is also adapted for transparent object rendering. The thesis introduces Virtual A-Buffer (VA-Buffer), also named Virtual Order Independent Transparency (VOIT), in Chapter 3.6.1. The shading part is offered in Chapter 4.1.6. VOIT modifies the GPU variant of the A-Buffer algorithm with virtual deferred principles, lowering the bandwidth and storage requirements. By relaxing the greatest constraint of the A-Buffer algorithm, VOIT is able to handle scenes with increased material complexity. Moreover VOIT shades adaptively, based on the opacity of each pixel, thus it shades and textures only the nodes which are guaranteed to have a visual impact in the final image. Furthermore, VOIT scales much better than A-Buffer when high quality results are needed and can be used to compute antialiasing without storing multiple samples per fragment. The method is also adapted for specular light transfer.

Another novel order independent transparency algorithm is also introduced, which modifies occupancy maps with depth distributions, adaptively increasing the depth resolution for each pixel. The method is named distribution occupancy maps and it is presented in chapter 3.6.2.

Each pixel stores a small resolution depth occupancy map, which is used to approximate the depth distribution over the pixel. Using this depth distribution, the full resolution occupancy map has its sampling points modified, to better adapt to the fragments rendered over the pixel. While this technique is not applicable to sharp geometric features, because of the approximated per-pixel opacity, it can still produce very good results for fuzzy objects, such as participating media and particle systems.

A selection algorithm is presented in chapter 3.7, which integrates seamlessly in any rasterization process. The selection algorithm improves upon state of the art methods, being able to select anything renderable, include alpha culling, the instances of instanced geometry, transient geometry generated with hardware tessellation, or alpha occlusion. The method was published in [**Pet13**].

Conservative Inexact Voxelization (CIV) is a novel type of imperfect voxelization, which exchanges correctness for speed. It is introduced in chapter 4.1.2. Compared to the state of the art voxelization solutions CIV has a drastically lowered complexity. Instead of working at a triangle level it works directly on objects, which are approximated, diced and stored in a hierarchic voxel representation. A push-pull process is then applied to the hierarchic voxel representation, which quickly updates the content for all levels of the hierarchy. The CIV structure is designed for tracing approximative rays, such as the ones used in approximative global illumination or stochastical collision rays. CIV integrates seamlessly into deferred pipelines as the depth samples from the depth buffer can be back projected to create extra geometric information for the objects inside the visualization frustum.

The CIV method is then used to relax the visibility operator in the rendering equation, acting as a visibility determination structure for virtual light generation. Because the CIV contains more data closer to the visualization volume, it permits tracing with an adaptive visibility operator, which is highly accurate inside the visualization volume and coarse outside it. Virtual lights are generating with random walks through the CIV, starting from the scene lights, like in any instant radiosity variant. The illumination with virtual lights is then performed with a deferred lighting algorithm. CIV is also used to trace local shadow. Compared to the state of the art methods, this enables fast diffuse light transport without any precomputation or special cases for animated or moving objects, as discussed in Chapter 4.1.3.1.

CIV can also be used in conjunction with approximated screen space specular light transport algorithms, to augment the approximated methods in their many fail cases. This is presented in Chapter 4.1.3.2.

This thesis also provides algorithms for rendering methods which are only at the edge of interactivity as of today, but will be used on consumer hardware for real-time rendering in the future. Photorealistic images are rendered with a bidirectional path tracer, which amortizes the cost of visibility determination operations through the imperfect but conservative visibility computed with CIV. Because CIV is also used to approximate the visibility operator for the real-time global illumination solution presented in this thesis, the two rendering paths - correct and approximate - are easily interchangeable.

The correct illumination solution introduces a new type of importance sampling, light flux sampling, which quickly approximates the flow of light in the scene and uses this approximation to quickly create productive paths. Compared to other state of the art methods, it

does not need to use a metropolis process or scene-wide geometry skeletonization to generate high-energy paths. Furthermore, for scenes with normal light transport, it creates paths much faster than the other sampling algorithms. Light flux can be considered an indirect light importance sampling mechanism, as it indirectly samples lights with a connection probability comparable to direct importance sampling.

Light flux importance sampling and amortized visibility are then used in a bidirectional path tracer to create photorealistic images. The bidirectional path tracer also employs state of the art algorithms such as adaptive sampling per tile, multiple importance sampling, radiance filtering, eye reprojection sampling, Russian roulette extinction, sample filtering and explicit connection testing through a spatial hash grid.

5.2. Conceptual Contributions

The decoupled algorithms presented in this thesis completely separate visibility determination, texture fetching and shading, in the context of rasterization rendering. This is performed without storing a very large number of samples or needing to synchronize a dynamic programming solution like in [**Rag11**]. Decoupled solutions ease aliasing analysis, frame rate stability and analysis and software development.

Conservative inexact voxelization is a unique type of voxelization because it works in $O(objects)$ instead of working in $O(primitives)$, as do the other state of the art algorithms. Approximative light transport over the presented conservative inexact voxelization data structure is a reliable solution for real-time transport of low frequency light, offering a true real-time solution for many lights shadowing. The rendering algorithms presented in this thesis use conservative inexact voxelization to solve low frequency light transport in real-time, but, like all other rendering solutions, including voxel cone tracing [**Cra09**], fail to produce high quality specular light transport, especially caustics, leaving place for improvement. The adaptation of the conservative inexact voxelization acceleration structure to screen space high frequency light transport mechanics offers perceptually pleasing specular transport, through the augmentation of the screen space algorithm with information needed for its fail cases. This is done without the harsh requirements of sparse octrees: streaming, storage and bandwidth consumption catalyzed be resolution requirements, and expensive voxelization.

The concept of perception influenced degradation of the visibility operator in the rendering equation, as introduced by conservative inexact voxelization, can lead to comparable results with the current [**Cra09**] [**Mar14**] [**Gan14**] or emerging [**Bik07**] [**Bik13**] interactive high frequency light transport solutions, but with only a fraction of the computational costs. This is especially valuable for storage and bandwidth poor hardware.

From a non real-time standpoint, the novel light flux importance sampling method has shown that links between paths can be constructed efficiently without much computation, and that the light distribution in the scene can be progressively approximated and harnessed for more efficient transportation algorithms.

REFERENCES

[Wat13] A. B. Watson, "High Frame Rates and Human Vision: A View Through the Window of Visibility", vol. 122, no. 2, pp. 18-32, doi: 10.5594/j18266 , Mar. 2013.

[Eis10] C. Eisenacher and C. Loop, "Data-parallel Micropolygon Rasterization", *Eurographics*, 2010.

[Kan01] T. Kaneko, T. Takahei, M. Inami, N. Kawakami, Y. Yanagida, T. Maeda, and S. Tachi, "Detailed Shape Representation with Parallax Mapping", in *Proceedings of ICAT*, pp. 205-208, 2001.

[Bra04] Z. Brawley and N. Tatarchuk, "Parallax Occlusion Mapping: Self-Shadowing, Perspective-Correct Bump Mapping Using Reverse Height Map Tracing", in *ShaderX 3*, 2004.

[Tat06] N. Tatarchuk, "Practical parallax occlusion mapping with approximate soft shadows for detailed surface rendering", in *ACM SIGGRAPH 2006 Courses* , pp. 81-112, 2006.

[Pre06] M. Premecz, "Iterative parallax mapping with slope information", in *Central European Seminar on Computer Graphics*, 2006.

[Mik08] M. Mikkelsen, "Simulation of Wrinkled Surfaces Revisited," 2008.

[Nog12] G. Nogrega de Lima, H. Batagelo, and P. Gois, "Displacement Mapping Techniques : Analysis and Comparisons", in *SIBGRAPI - Workshop of Undergraduate Works*, pp. 59-64, 2012.

[Ris07] E. Risser, M. Shah, and S. Pattanaik, "Faster relief mapping using the secant method", *Journal of Graphics Tools*, vol. 12, no. 3, 2007.

[Lob08] R. Lobel, "SSDM: Screen Space Displacement Mapping," 2008.

[Ola10] M. Olano and D. Baker, "LEAN mapping", in *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* , pp. 181-188, 2010.

[Bak11] D. Baker, "Spectacular Specular -LEAN and CLEAN specular highlights", in *GDC*, 2011.

[Mcg10] M. Mcguirre, E. Enderton, P. Shirley, and D. Luebke, "Real-Time Stochastic Rasterization on Conventional GPU Architectures", *Proceedings of High Performance Graphics*, pp. 173-182, Jun. 2010.

[Tok04] M. Toksvig, "Mipmapping Normal Maps," NVIDIA, 2004.

[Hun08] W. Hunt and W. Mark, "Ray-Specialized Acceleration Structures for Ray Tracing", *IEEE/EG Symposium on Interactive Ray Tracing* , 2008.

[Sch09] F. Schornbaum, "Hierarchical Hash Grids for Coarse Collision Detection", PhD Thesis, University of Erlangen-Nurnbergm, Nuremberg, 2009.

[Har12] T. Harada, "A 2.5D culling for Forward+", *SIGGRAPH Asia 2012 Technical Briefs* , 2012.

[Fuc80] H. Fuchs, Z. Kedem, and B. Naylor, "On visible surface generation by a priori tree structures", *SIGGRAPH '80 Proceedings of the 7th annual conference on Computer graphics and interactive techniques* , pp. 124-133, 1980.

[Had98] M. Hadwiger and A. Varga, "Visibility Culling," 1998.

[Lai101] S. Laine and T. Karras, "Efficient sparse voxel octrees", *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* , pp. 55-63, 2010.

[Sim12] I. Simecek, D. Langr, and C. Tvrdik, "Minimal Quadtree Format for Compression of Sparse Matrices Storage ", *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pp. 359-364, Sep. 2012.

[Moo91] A. Moore, "An introductory tutorial on kd-trees," 1991.

[Ben75] J. L. Bentley, "Multidimensional binary search trees used for associative searching", *Communications of the ACM*, vol. 18, no. 9, p. 509, 1975.

[Bik07] J. Bikker, "Real-time Ray Tracing through the Eyes of a Game Developer", *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*, pp. 1-10, DOI 10.1109/RT.2007.4342584, Sep. 2007.

[Wal06] I. Wald and V. Havran, "On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N)", *IEEE Syposiom of Interactive Ray Tracing*, pp. 61-69, Sep. 2006.

[Dan10] P. Danilewski, S. Popov, and P. Slusallek, "Binned SAH Kd-Tree Construction on a GPU", in *High Performance Graphics*, 2010.

[Wal072] I. Wald, "On fast Construction of SAH-based Bounding Volume Hierarchies", *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pp. 33-40, 2007.

[Ern07] M. Ernst and G. Greiner, "Early Split Clipping for Bounding Volume Hierarchies", in *IEEE Symposium on Interactive Ray Tracing*, pp. 73-78, 2007.

[Dam081] H. Dammertz and A. Keller, "The edge volume heuristic - robust triangle subdivision for improved BVH performance", in *IEEE Symposium on Interactive Ray Tracing*, pp. 155-158, 2008.

[Sti09] M. Stich, H. Friedrich, and A. Dietrich, "Spatial splits in bounding volume hierarchies", *Proceedings of the Conference on High Performance Graphics*, pp. 7-13, 2009.

[Pop09] S. Popov, I. Georgiev, R. Dimov, and P. Slussallek, "Object partitioning considered harmful: space subdivision for BVHs", in *Proceedings of the Conference on High Performance Graphics*, pp. 15-22, 2009.

[Dam08] H. Dammertz, J. Hanika, and A. Keller, "Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays", *Proceedings of the Nineteenth Eurographics conference on Rendering*, pp. 1225-1233, 2008.

[Tsa09] J. Tsakok, "Faster incoherent rays: Multi-BVH ray stream tracing", in *Proceedings of the Conference on High Performance Graphics*, pp. 151-158, 2009.

[Vin14] M. Vinkler, V. Havran, and J. Bittner, "Bounding volume hierarchies versus kd-trees on contemporary many-core architectures", in *Proceedings of the 30th Spring Conference on Computer Graphics* , pp. 29-36, 2014.

[Bik13] J. Bikker and J. van Schneidel, "The Brigade Renderer: A Path Tracer for Real-Time Games", *International Journal of Computer Games Technology*, vol. vol. 2013, p. 14, Article ID 578269, doi:10.1155/2013/578269, Sep. 2013.

[Ooi87] B. Ooi, "Spatial kd-Tree: A Data Structure for Geographic Database", *Informatik-Fachberichte*, vol. 136, 1987.

[Zac02] G. Zachmann, "Minimal Hierarchical Collision Detection", in *"Proc. ACM*

Symposium on Virtual Reality Software and Technology, Hong Kong, pp. 121-128, 2002.

[Wäc06] C. Wächter and A. Keller, "Instant ray tracing: the bounding interval hierarchy", *Proceedings of the 17th Eurographics conference on Rendering Techniques* , pp. 139-149, 2006.

[Hav06] V. Havran, R. Herzog, and P. Slusalek, "On the Fast Construction of Spatial Hierarchies", *IEEE Symposium of Interactive Ray Tracing*, pp. 71-80, 2006.

[Wal071] I. Wald, S. Boulos, and P. Shirley, "Ray tracing deformable scenes using dynamic bounding volume hierarchies", *ACM Transactions on Graphics (TOG)*, vol. 26, no. 1, Jan. 2007.

[Fol05] T. Foley and J. Sugerman, "KD-tree acceleration structures for a GPU raytracer", in *Proceeding HWWS '05 Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* , pp. 15-22, 2005.

[Pop07] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek, "Stackless KD-Tree Traversal for High Performance GPU Ray Tracing", *Computer Graphics Forum*, vol. 26, no. 3, p. 415–424, Sep. 2007.

[Hap11] M. Hapala, T. Davidovič, I. Wald, V. Havran, and P. Slusallek, "Efficient stack-less BVH traversal for ray tracing", in *Proceedings of the 27th Spring Conference on Computer Graphics* , pp. 7-12, 2011.

[Afr14] A. Afra and L. Szirmay-Kalos, "Stackless Multi-BVH Traversal for CPU, MIC and GPU Ray Tracing", in *Computer Graphics Forum*, pp. 129-140, 2014.

[Bar14] R. Barringer and T. Akenine-Möller, "Dynamic ray stream traversal", *Journal ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH*, vol. 33, no. 4, Jul. 2014.

[van11] D. van Antwerpen, "Improving SIMD efficiency for parallel Monte Carlo light transport on the GPU", in *High Performance Graphics*, New York, USA, pp. 41-50, 2011.

[Mor11] B. Mora, "Naive ray-tracing: A divide-and-conquer approach", *ACM Transactions on Graphics (TOG)*, vol. 30, no. 5, Oct. 2011.

[Kel11] A. Keller and C. Waechter, "Efficient Ray Tracing without Auxiliary Acceleration Data Structure," NVIDIA, 2011.

[Nab13] K. Nabata, K. Iwasaki, Y. Dobashi, and T. Nishita, "Efficient divide-and-conquer ray tracing using ray sampling", in *Proceedings of the 5th High-Performance Graphics Conference* , pp. 129-135, 2013.

[Afr12] A. Afra, "Incoherent ray tracing without acceleration structures", in *Proc. of Eurographics Short Paper*, pp. 97-100, 2012.

[Wal03] I. Wald, C. Benthin, and P. Slusallek, "Distributed interactive ray tracing of dynamic scenes ", *IEEE Symposion of Parallel and Large-Data Visualization and Graphics*, pp. 77-85, Oct. 2003.

[Sha98] J. Shade, S. Goertler, L. He, and R. Szeliski, "Layered depth images", *Proceedings of the 25th annual conference on Computer graphics and interactive techniques, SIGGRAPH*, pp. 231-242, Aug. 1998.

[Rad14] M. Radwan, S. Ohrhallinger, and M. Wimmer, "Efficient Collision Detection While Rendering Dynamic Point Clouds", in *Proceedings of the 2014 Graphics Interface*

Conference, pp. 25-33, 2014.

[Nga05] A. Ngan, F. Durand, and W. Matusik, "Experimental analysis of BRDF models", in *Proceedings of the Sixteenth Eurographics conference on Rendering Techniques* , pp. 117-126, 2005.

[Edw03] A. Edward, *Interactive Computer Graphics: A Top-Down Approach Using OpenGL*, 3rd ed. Addison-Wesley, 2003.

[Tor67] K. Torrance and E. Sparrow, "Theory for Off-Specular Reflection from Roughened Surfaces", *Journal of Optical Society of America*, vol. 57, p. 1105–1114, 1967.

[Cra09] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "GigaVoxels : Ray-Guided Streaming for Efficient and Detailed Voxel Rendering", in *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, Boston, MA, USA, 2009.

[Bli77] J. Blinn, "Models of light reflection for computer synthesized pictures", in *Proceedings of the 4th annual conference on Computer graphics and interactive techniques* , pp. 192-198, 1977.

[Min41] M. Minnaert, "The reciprocity principle in lunar photometry", *The reciprocity principle in lunar photometry*, vol. 93, pp. 403-410, May 1941.

[War92] G. Ward, "Measuring and modeling anisotropic reflection", in *Proceedings of the 19th annual conference on Computer graphics and interactive techniques* , pp. 265-272, 1992.

[Sch98] C. Schlick, "An Inexpensive BRDF Model for Physically-based Rendering", in *Computer Graphics Forum*, 1998.

[Ore94] M. Oren and S. Nayar, "Generalization of Lambert's reflectance model", in *Proceedings of the 21st annual conference on Computer graphics and interactive techniques* , pp. 239-246, 1994.

[Hei98] W. Heidrich and H.-P. Seidel, "Efficient Rendering of Anisotropic Surfaces Using Computer Graphics Hardware", in *Image and Multi-dimensional Digital Signal Processing Workshop (IMDSP)*, 1998.

[Ash00] M. Ashikmin, S. Premože, and P. Shirley, "A microfacet-based BRDF generator", in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 65-74, 2000.

[Kel01] C. Kelemen and L. Kalos, "A microfaced based coupled specular-matte brdf model with importance sampling", in *Eurographics Short Presentations*, 2001.

[Wal07] B. Walter, S. Marschner, H. Li, and K. Torrance, "Microfacet models for refraction through rough surfaces", in *Proceedings of the 18th Eurographics conference on Rendering Techniques* , pp. 195-206, 2007.

[Slo11] P.-P. Sloan, D. Nowrouzezahrai, and H. Yuan, "Wrap Shading", *Journal of Graphics, GPU, and Game Tools*, vol. 15, no. 4, 2011.

[Mar13] M. Mara, D. Luebke, and M. McGuirre, "Toward Practical Real-Time Photon Mapping : Efficient GPU Density Estimation", *Interactive 3D Graphics and Games 2013*, Mar. 2013.

[Bur12] B. Burley, "Physically-Based Shading at Disney," Walt Disney Animation Studios, 2012.

[Sch11] K. Schwenk, "A Survey of Shading Models for Real-time Rendering," 2011.

[Hen41] L. G. Henyey and J. L. Greenstein, "Diffuse radiation in the galaxy", *Astrophysical*

Journal, vol. 93, pp. 70-83, 1941.

[Sch93] S. C. Schlick, B. Le Saëc, and P. Blasi, "A Rendering Algorithm for Discrete Volume Density Objects", *Computer Graphics Forum (Eurographics)*, vol. 12, no. 3, pp. 201-210, 1993.

[Hec90] P. Heckbert, "Adaptive radiosity textures for bidirectional ray tracing", in *Proceedings of the 17th annual conference on Computer graphics and interactive techniques* , pp. 145-154, 1990.

[Sut74] I. Sutherland, R. Sproull, and R. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms", *ACM Computing Surveys (CSUR)*, vol. 6, no. 1, pp. 1-55, Mar. 1974.

[Hav14] V. Havran and J. Bittner, "Efficient Sorting and Searching in Rendering Algorithms", in *Eurographics - Tutorials*, 2014.

[Dav121] T. Davidovič, T. Engelhardt, I. Georgiev, P. Slusallek, and C. Dachsbacher, "3D rasterization: a bridge between rasterization and ray casting", in *Proceedings of Graphics Interface 2012*, Toronto, pp. 201-208, 2012.

[App68] A. Appel, "Some techniques for shading machine renderings of solids", in *AFIPS '68 (Spring) Proceedings of the April 30--May 2, 1968, spring joint computer conference* , New York, pp. 37-45, 1968.

[Wei06] D. Weiskopf, "Volume Ray Casting", in *GPU-Based Interactive Visualization Techniques*. Springer Science & Business Media., 2006, p. 21.

[Dav12] T. Davidovič, I. Georgiev, and P. Slusallek, "Progressive Lightcuts for GPU", in *Proceeding SIGGRAPH '12 ACM SIGGRAPH 2012 Talks*, New York, USA, 2012.

[Whi79] T. Whitted, "An improved illumination model for shaded display", *ACM SIGGRAPH Computer Graphics*, vol. 13, no. 2, p. 14, Aug. 1979.

[Jen96] H. W. Jensen, "Global illumination using photon maps", in *Proceedings of the eurographics workshop on Rendering techniques '96* , London, pp. 21-30, 1996.

[Kel97] A. Keller, "Instant radiosity", in *SIGGRAPH '97 Proceedings of the 24th annual conference on Computer graphics and interactive techniques* , New York, pp. 49-56, 1997.

[Coz09] P. Cozzi, "Z-Buffer Optimizations," Analytical Graphics, Inc, 2009.

[Dur99] F. Durand, "3D Visibility: analytical study and applications," Université Joseph Fourier, Grenoble I, 1999.

[Bit04] J. Bittner, M. Wimmer, H. Piringer, and W. Purgathofer, "Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful", *Computer Graphics Forum*, vol. 23, no. 3, pp. 615-624, Sep. 2004.

[Llo04] B. Lloyd, J. Wendt, N. Govindaraju, and D. Manocha, "CC shadow volumes", in *EGSR'04 Proceedings of the Fifteenth Eurographics conference on Rendering Techniques* , pp. 197-205, 2004.

[Bit11] J. Bittner, O. Mattausch, A. Silvennoinen, and M. Wimmer, "Shadow caster culling for efficient shadow mapping", in *Symposium on Interactive 3D Graphics and Games* , pp. 81-88, 2011.

[Mar11] J. Marvie, P. Gautron, and G. Sourimant, "Triple depth culling", in *ACM SIGGRAPH Talks* , 2011.

[Déc05] X. Décoret, "N-Buffers for efficient depth map query", *Computer Graphics Forum*,

vol. 24, no. 3, Dec. 2005.

[Dac14] C. Dachsbacher, J. Křivánek, M. Hašan, A. Arbree, B. Walter, and J. Novák, "Scalable Realistic Rendering with Many-Light Methods", *Computer Graphics Forum*, vol. 33, no. 1, pp. 88-104, DOI: 10.1111/cgf.12256, Feb. 2014.

[Nie12] M. Niessner and L. Charles, "Patch-based Occlusion Culling for Hardware Tessellation", *Computer Graphics International*, vol. 2, 2012.

[Bar12] L. Barbagallo, M. Leone, M. Banquiero, D. Agromayor, and A. Bursztyn, "Techniques for an Image Space Occlusion Culling Engine," 2012.

[Has07] J. Hasselgren and T. Akenine-Möller, "PCU: the programmable culling unit", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH*, vol. 26, no. 3, Jul. 2007.

[Lot09] Lottes, Timothy; NVIDIA, "FXAA", 2009.

[Jim12] J. Jimenez, J. Echevarria, D. Gutierrez, and T. Sousa, "SMAA: Enhanced Morphological Antialiasing", *Computer Graphics Forum*, vol. 31, no. 2, 2012.

[Neh07] D. Nehab, P. Sander, J. Lawrence, N. Tatarchuk, and J. Isidoro, "Accelerating real-time shading with reverse reprojection caching", *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pp. 25-35, 2007.

[Yan09] L. Yang, D. Nehab, P. Sander, P. Sitthiamon, J. Lawrence, and H. Hoppe, "Amortized supersampling", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia*, vol. 28, no. 5, Dec. 2009.

[Cha11] M. Chajdas, M. Mcguirre, and D. Luebke, "Subpixel Reconstruction Antialiasing", *Interactive 3D Graphics and Games*, 2011.

[Res12] Reshetov, Alexander; Intel Labs, "Reducing Aliasing Artifacts through Resampling", *High Performance Graphics*, 2012.

[Cra15] C. Crassin, M. McGuirre, K. Fatahalian, and A. Lefohn, "Aggregate G-Buffer Anti-aliasing", *Proceedings of the ACM Symposium on Interactive 3D Graphics and Games*, Feb. 2015.

[Sha73] R. M. Shapley and D. J. Tolhurst, "Edge detectors in human vision", *The Journal of Physiology*, vol. 229, no. 1, pp. 165-183, PMCID: PMC1350218, Feb. 1973.

[Sai90] T. Saito and T. Takahashi, "Comprehensible rendering of 3-D shapes", in *Proceedings of the 17th annual conference on Computer graphics and interactive techniques SIGGRAPH*, pp. 197-206, 1990.

[Lee09] M. Lee, "Prelighting in Resistance 2", 2009.

[van13] M. van de Hoef, "Hybrid Deferred Rendering," 2013.

[Thi09] N. Thibieroz, "Deferred Shading with Multisampling Anti-Aliasing in DirectX 10", in *ShaderX 7*, 2009.

[Hol13] M. Holländer, T. Boubekeur, and E. Eisemann, "Adaptive Supersampling for Deferred Anti-Aliasing", *Journal of Computer Graphics Techniques*, vol. 2, no. 1, 2013.

[Kel011] A. Keller, E. Keller, and W. Heidrich, "Interleaved Sampling", in *Rendering Techniques (Proc. 12th Eurographics Workshop on Rendering)*, pp. 269-276, 2001.

[Seg06] B. Segovia, J. C. Iehl, R. Mitanchey, and B. Péroche, "Non-interleaved deferred shading of interleaved sample patterns", *Proceedings of the 21st ACM*

SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, pp. 53-60, 2006.

[Tre09] D. Treblico, "Light Indexed Deferred Rendering", in *ShaderX 7*, 2009.

[Lau12] A. Lauritzen, "Intersecting Lights", in , 2012.

[Hob09] J. Hoberock, V. Lu, Y. Jia, and J. Hart, "Stream Compaction for Deferred Shading", *Proceedings of the Conference on High Performance Graphics 2009* , pp. 173-180, 2009.

[Hum09] L. Humes, T. Busey, J. Craig, and D. Kewly-Port, "The effects of age on sensory thresholds and temporal gap detection in hearing, vision, and touch", *Atten Percept Psychophys*, vol. 71, no. 4, pp. 860-871, doi: 10.3758/APP.71.4.860, May 2009.

[Kaj86] J. T. Kajiya, "The Rendering Equation", *ACM SIGGRAPH Computer Graphics*, vol. 20, no. 4, pp. 143-150, Aug. 1986.

[Cla13] P. Clarberg, R. Toth, and J. Munkberg, "A sort based deferred shading architecture for decoupled sampling", *ACM Transactions of Graphics (Proceedings of SIGGRAPH 2013)*, vol. 32, no. 4, pp. 141-151, Jul. 2013.

[Cha15] J. Chapman. (2015, Apr.) http://john-chapman-graphics.blogspot.com. [Online]. http://john-chapman-graphics.blogspot.de/2013/01/good-enough-volumetrics-for-spotlights.html

[Mit071] K. Mitchell, "Volumetric Light Scattering as a Post-Process", in *GPU Gems 3*. Addison-Wesley Professional, 2007, ch. 13.

[Tot09] B. Toth and T. Umenhoffer, "Real-time Volumetric Lighting in Participating Media", in *Eurographics Short Papers*, 2009.

[Eng10] T. Engelhardt and C. Dachsbacher, "Epipolar sampling for shadows and crepuscular rays in participating media with single scattering", in *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* , pp. 119-125, 2010.

[Fus00] A. Fusiello, E. Trucco, and A. Verri, "A compact algorithm for rectification of stereo pairs", *Machine Vision and Applications*, vol. 12, pp. 16-22, 2000.

[Che11] J. Chen, I. Baran, F. Durand, and W. Jarosz, "Real-Time Volumetric Shadows using 1D Min-Max Mipmaps", in *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2011.

[Bil10] M. Billeter, E. Sintorn, and U. Assarsson, "Real Time Volumetric Shadows using Polygonal Light Volumes", in *Proceedings of the Conference on High Performance Graphics*, Saarbrucken, Germany, pp. 39--45, 2010.

[Bor03] G. Borshukov and J. P. Lewis, "Realistic human face rendering for "The Matrix Reloaded"", in *ACM SIGGRAPH 2003 Sketches & Applications* , pp. 1-1, 2003.

[dEo07] E. d'Eon, D. Luebke, and E. Enderton, "Efficient rendering of human skin", in *Proceedings of the 18th Eurographics conference on Rendering Techniques* , pp. 147-157, 2007.

[Jan10] J. Jansen and L. Bavoil, "Fourier opacity mapping", *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* , pp. 165-172, 2010.

[Bri11] C. Brisebois and M. Bouchard, "Approximating Translucency for a Fast, Cheap and Convincing Subsurface Scattering Look", in *Game Developers Conference*, 2011.

[Jim09] J. Jimenez, V. Sundstedt, and D. Gutierrez, "Screen-space perceptual rendering of human skin", *ACM Transactions on Applied Perception (TAP)*, vol. 6, no. 4, Sep. 2009.

[Jim15] J. Jimenez, K. Zsolnai, A. Jarabo, C. Freude, T. Auzinger, X.-C. Wu, J. von der Pahlen, M. Wimmer, and D. Gutierrez, "Separable Subsurface Scattering", *Computer Graphics Forum*, 2015.

[Cro77] F. Crow, "Shadow algorithms for computer graphics", in *SIGGRAPH '77 Proceedings of the 4th annual conference on Computer graphics and interactive techniques* , pp. 242-248, 1977.

[Wil78] L. Williams, "Casting curved shadows on curved surfaces", in *SIGGRAPH '78 Proceedings of the 5th annual conference on Computer graphics and interactive techniques* , pp. 270-274, 1978.

[Sta02] M. Stamminger and G. Drettakis, "Perspective Shadow Maps", in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques* , pp. 557-562, 2002.

[Wim04] M. Wimmer, D. Scherzer, and W. Purgathofer, "Light space perspective shadow maps", in *Proceedings of the Fifteenth Eurographics conference on Rendering Techniques* , pp. 143-151, 2004.

[Mar04] T. Martin and T.-S. Tan, "Anti-aliasing and continuity with trapezoidal shadow maps", in *Proceedings of the Fifteenth Eurographics conference on Rendering Techniques* , pp. 153-160, 2004.

[Kol12] I. Kolic and Z. Mihajlovic, "Camera space shadow maps for large virtual environments", in *Virtual Reality*, pp. 289-299, 2012.

[Llo06] B. Lloyd, N. Govindaraju, D. Tuft, S. Molnar, and D. Manocha, "Practical logarithmic shadow maps", in *ACM SIGGRAPH 2006 Sketches* , 2006.

[Mar14] M. Mara, M. McGuire, D. Nowrouzezahrai, and D. Luebke, "Fast Global Illumination Approximations on Deep G-Buffers", pp. 1-16, Jun. 2014.

[Dim07] R. Dimitrov, "Cascaded Shadow maps," NVIDIA, 2007.

[Zha06] F. Zhang, H. Sun, L. Xu, and L. K. Lun, "Parallel-split shadow maps for large-scale virtual environments", in *Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, pp. 311-318, 2006.

[Lau11] A. Lauritzen, M. Salvi, and A. Lefohn, "Sample distribution shadow maps", in *Symposium on Interactive 3D Graphics and Games* , pp. 97-102, 2011.

[Fer01] R. Fernando, S. Fernandez, K. Bala, and D. Greenberg, "Adaptive shadow maps", in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 387-390, 2001.

[Gue07] G. Guennebaud, L. Barthe, and M. Paulin, "High-Quality Adaptive Soft Shadow Mapping", *Computer Graphics Forum*, vol. 26, no. 3, pp. 525-533, Sep. 2007.

[Ros12] P. Rosen, "Rectilinear texture warping for fast adaptive shadow mapping", in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* , pp. 151-158, 2012.

[Dai08] Q. Dai, B. Yang, and J. Feng, "Reconstructable geometry shadow maps", in *Proceedings of the 2008 symposium on Interactive 3D graphics and games* , 2008.

[Lok00] T. Lokovic and E. Veach, "Deep shadow maps", in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 385-392, 2000.

[Pag04] C. A. Pagot, J. Comba, and M. Neto, "Multiple-Depth Shadow Maps", in *SIBGRAPI '04 Proceedings of the Computer Graphics and Image Processing*, pp. 308-315, 2004.

[Yuk08] C. Yuksel and J. Keyser, "Deep Opacity Maps", *Computer Graphics Forum*, vol. 27, no. 2, 2008.

[Bar11] P. Barta, B. Kovács, L. Szécsi, and L. Szirmay-kalos, "Order Independent Transparency with Per-Pixel Linked Lists", in *CESCG*, 2011.

[Sal10] M. Salvi, K. Vidimče, A. Lauritzen, and A. Lefohn, "Adaptive volumetric shadow maps", in *1289-1296* , p. Proceedingsofthe21stEurograhicsconferenceonRendering, 2010.

[Kim01] T.-y. Kim and U. Neumann, "Opacity Shadow Maps", in *Rendering Techniques*, pp. 177-182, 2001.

[Sal101] M. Salvi, K. Vidimce, A. Lauritzen, and A. Lefohn, "Adaptive volumetric shadow maps", *Proceedings of the 21st Eurographics conference on Rendering* , pp. 1289-1296, 2010.

[Ran05] F. Randima, "Percentage-closer soft shadows", *ACM SIGGRAPH 2005 Sketches* , 2005.

[Sch13] M. Schwärzler, C. Luksch, D. Scherzer, and M. Wimmer, "Fast percentage closer soft shadows using temporal coherence", in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* , pp. 79-86, 2013.

[Bag10] M. Bagher, J. Kautz, N. Holzschuch, and C. Soler, "Screen-space Percentage-Closer Soft Shadows", in *ACM SIGGRAPH 2010 Posters* , 2010.

[Ann08] T. Annen, T. Mertens, H.-P. Seidel, E. Flerackers, and J. Kautz, "Exponential shadow maps", in *Proceedings of Graphics Interface 2008* , pp. 155-161, 2008.

[Don06] W. Donnelly and A. Lauritzen, "Variance shadow maps", in *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pp. 161-165, 2006.

[Ann07] T. Annen, T. Mertens, P. Bekaert, H. P. Seidel, and J. Kautz, "Convolution shadow maps", in *Proceedings of the 18th Eurographics conference on Rendering Techniques* , pp. 51-60, 2007.

[Sto15] Story, Jon; NVIDIA, "Hybrid Ray Traced Shadows", in *Game Developers Conference*, 2015.

[Gre93] N. Greene, M. Kaas, and G. Miller, "Hierarchical Z-buffer visibility", *SIGGRAPH '93 Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pp. 231-238, ISBN:0-89791-601-8, doi: 10.1145/166117.166147, Aug. 1993.

[Osm06] B. Osman, M. Bukowski, and C. McEvoy, "Practical implementation of dual paraboloid shadow maps", in *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames* , pp. 103-106, 2006.

[Ger07] P. Gerasimov, "Omnidirectional Shadow Mapping", in *GPU Gems*, 2007.

[Ols14] O. Olsson, E. Sintorn, V. Kämpe, M. Bieleter, and U. Assarsson, "Efficient virtual shadow maps for many lights", in *Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* , pp. 87-96, 2014.

[Tar10] S. Tariq and C. Yuksel, "Transparency and Antialiasing", in *SIGGRAPH 2010 Courses*, 2010.

[Mes07] H. Meshkin, "Sort Independent Alpha Blending", in , 2007.

[McG13] M. McGuirre and L. Bavoil, "Weighted Blended Order-Independent Transparency", *Journal of Computer Graphics Techniques*, vol. 2, no. 2, 2013.

[Bav08] L. Bavoil and K. Myers, "Order Independent Transparency with Dual-Depth Peeling," NVIDIA, 2008.

[Mau12] M. Maule, J. Comba, R. Torchelsen, and R. Bastos, "Memory-Efficient Order-Independent Transparency with Dynamic Fragment Buffer", in *Proceedings of the 2012 25th SIBGRAPI Conference on Graphics, Patterns and Images* , pp. 134-141, 2012.

[Ake07] T. Akenine-Möller, J. Munkberg, and J. Hasselgren, "Stochastic rasterization using time-continuous triangles", in *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pp. 7-16, 2007.

[Gue14] J.-P. Guertin, M. McGuirre, and D. Nowrouzezahrai, "A Fast and Stable Feature-Aware Motion Blur Filter", *High Performance Graphics*, vol. 33, no. 2, 2014.

[Joh05] G. Johnson, J. Lee, C. Burns, and W. Mark, "The Irregular Z-Buffer:Hardware Acceleration for Irregular Data Structures", *ACM Transactions on Graphics (TOG)*, vol. 24, no. 4, pp. 1462-1482, Oct. 2005.

[Sou13] T. Sousa, "Cryengine 3 Graphic Gems", in *Siggraph Talks*, 2013.

[McI12] L. McIntosh, B. E. Riecke, and S. DiPaola, "Efficiently Simulating the Bokeh of Polygonal Apertures in a Post-Process Depth of Field Shader", *Computer Graphics Forum*, vol. 31, no. 6, pp. 1810-1822, Sep. 2012.

[Bli76] J. Blinn and M. Newell, "Texture and Reflection in Computer Generated Images", in *Communications of the ACM*, 1976.

[Bjo04] K. Bjorke, "Image-based lighting", in *GPU Gems*, 2004, pp. 307-322.

[Lag12] S. Lagarde and A. Zanuttini, "Local image-based lighting with parallax-corrected cubemaps", in *ACM SIGGRAPH 2012 Talks*, 2012.

[Ram01] R. Ramamoorthi and P. Hanrahan, "An efficient representation for irradiance environment maps", in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* , p. Hanrahan, 2001.

[Now12] D. Nowrouzezahrai, P. Simari, and E. Fiume, "Sparse Zonal Harmonic Factorization for Efficient SH Rotation", *ACM Transactions on Graphics*, 2012.

[Gre03] R. Green, "Spherical Harmonic Lighting: The gritty details," Sony Computer Entertainment America, 2003.

[Kap09] A. Kaplanyan, "Light Propagation Volumes in CryEngine 3," Crytek, 2009.

[Gil12] M. Gilabert and N. Stefanov, "Deferred Radiance Transfer Volumes", in *Game Developers Conference*, 2012.

[Car84] L. Carpenter, "The A-buffer, an antialiased hidden surface method", *ACM SIGGRAPH Computer Graphics*, vol. 18, no. 3, pp. 103-108, Jul. 1984.

[Zhu98] S. Zhukov, A. Iones, and G. Kronin, "An ambient light illumination model", in *Rendering Techniques*, pp. 45-55, 1998.

[Mit07] M. Mittring, "Finding next gen: CryEngine 2", in *ACM SIGGRAPH 2007 courses*, pp. 97-121, 2007.

[Bav081] L. Bavoil, M. Sainz, and R. Dimitrov, "Image-space horizon-based ambient occlusion", in *ACM SIGGRAPH 2008 talks* , 2008.

[Dru06] J. Drummer, "Cone step mapping," 2006.

[Loo10] B. Loos and P. P. Sloan, "Volumetric obscurance", in *Proceedings of the 2010 ACM*

SIGGRAPH symposium on Interactive 3D Graphics and Games , pp. 151-156, 2010.

[McG11] M. McGuirre, B. Osman, M. Bukowski, and P. Hennessy, "The alchemy screen-space ambient obscurance algorithm", in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pp. 25-32, 2011.

[Mcg12] M. Mcguirre, M. Mara, and D. Leubke, "Scalable ambient obscurance", in *Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics* , Aire-la-Ville, pp. 97-103, 2012.

[Var13] K. Vardis, G. Papaioannou, and A. Gaitatzes, "Multi-view ambient occlusion with importance sampling", in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* , pp. 111-118, 2013.

[Tim13] V. Timonen, "Line-sweep ambient obscurance", in *Proceedings of the Eurographics Symposium on Rendering* , pp. 97-105, 2013.

[Tim131] V. Timonen, "Screen-space far-field ambient obscurance", in *Proceedings of the 5th High-Performance Graphics Conference* , pp. 33-43, 2013.

[Dee88] M. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt, "The triangle processor and normal vector shader: a VLSI system for high performance graphics", *SIGGRAPH '88 Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pp. 21-30, ISBN:0-89791-275-6, Jul. 1988.

[Sch10] D. Scherzer, L. Yang, and O. Mattausch, "Exploiting temporal coherence in real-time rendering", in *ACM SIGGRAPH ASIA 2010 Courses* , 2010.

[Rit09] T. Ritschel, T. Grosch, and H.-P. Seidel, "Approximating dynamic global illumination in image space", in *Proceedings of the 2009 symposium on Interactive 3D graphics and games* , pp. 75-82, 2009.

[Lan02] H. Landis, "Production-ready global illumination", in *SIGGRAPH Course*, pp. 87-102, 2002.

[Rit11] T. Ritschel, O. Klehm, E. Eisemann, and H.-P. Seidel, "Bent Normals and Cones in Screen Space", in *Proceedings of the Vision, Modeling, and Visualization Workshop*, Berlin, pp. 177-182, 2011.

[Rob09] A. Robinson and P. Shirley, "Image space gathering", in *Proceedings of the Conference on High Performance Graphics*, pp. 91-98, 2009.

[Sol10] C. Soler, O. Hoel, and F. Rochet, "A deferred shading pipeline for real-time indirect illumination", in *ACM SIGGRAPH Talks* , 2010.

[Her14] L. Hermanns and T. Franke, "Screen Space Cone Tracing for Glossy Reflections", in *SIGGRAPH '14*, pp. 102-103, 2014.

[Ulu14] Y. Uludag, "Hi-Z Screen-Space Cone-Traced Reflections", in *GPU Pro 5*, W. Engel, Ed. CRC Press, 2014, p. 149–192.

[Mcg14] M. Mcguirre and M. Mara, "Efficient GPU Screen-Space Ray Tracing", *Journal of Computer Graphics Techniques*, vol. 3, no. 4, 2014.

[McG09] M. McGuirre and D. Luebke, "Hardware-accelerated global illumination by image space photon mapping", in *Proceedings of the Conference on High Performance Graphics* , pp. 77-89, 2009.

[Ols11] O. Olsson and U. Assarson, "Tiled Shading", *Journal of Graphics, GPU, and Game Tools*, vol. 15, no. 4, pp. 235-251, doi = 10.1080/2151237X.2011.621761, 2011.

[Gan14] P. Ganestam and M. Doggett, "Real-time multiply recursive reflections and

refractions using hybrid rendering", *The Visual Computer*, Sep. 2014.

[PIX15] PIXAR. (2015, Apr.) https://renderman.pixar.com. [Online]. https://renderman.pixar.com/resources/current/RenderMan/home.html

[Pat08] A. Patney and J. Owens, "Real-time Reyes-style adaptive surface subdivision", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia 2008 TOG*, vol. 27, no. 5, Dec. 2008.

[Vea95] E. Veach and L. Guibas, "Optimally combining sampling techniques for Monte Carlo rendering", in *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* , pp. 419-428, 1995.

[Laf96] E. Lafortune, "Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering ," 1996.

[Coo84] R. Cook, T. Porter, and L. Carpenter, "Distributed ray tracing", in *Proceedings of the 11th annual conference on Computer graphics and interactive techniques* , pp. 137-145, 1984.

[Dut93] P. Dutre, E. Lafortune, and Y. D. Willems, "Monte carlo light tracing with direct computation of pixel intensities", in *Compugraphics*, pp. 128-137, 1993.

[Ige99] H. Igehy, "Tracing ray differentials", in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* , pp. 179-186, 1999.

[Pai89] J. Painter and K. Sloan, "Antialiased ray tracing by adaptive progressive refinement", in *Proceedings of the 16th annual conference on Computer graphics and interactive techniques* , pp. 281-288, 1989.

[Bou07] S. Boulos, D. Edwards, D. Lacewell, J. Kniss, J. Kautz, P. Shirley, and I. Wald, "Packet-based whitted and distribution ray tracing", in *Proceedings of Graphics Interface*, pp. 177-184, 2007.

[Ols12] O. Olsson, M. Billeter, and U. Assarsson, "Clustered Deferred and Forward Shading", *HPG '12: Proceedings of the Conference on High Performance Graphics 2012*, 2012.

[Ove08] R. Overbeck, R. Ramamoorthi, and W. R. Mark, "Large ray packets for real-time Whitted ray tracing", in *IEEE Symposion on Interactive Ray Tracing*, pp. 41-48, 2008.

[Pha97] M. Pharr, C. Kolb, R. Gerschbein, and P. Hanrahan, "Rendering complex scenes with memory-coherent ray tracing", in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* , pp. 101-108, 1997.

[Nav07] P. Navratil, D. Fussell, C. Lin, and W. Mark, "Dynamic Ray Scheduling to Improve Ray Coherence and Bandwidth Utilization", in *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pp. 95-104, 2007.

[Ail10] T. Aila and T. Karras, "Architecture considerations for tracing incoherent rays", in *Proceedings of the Conference on High Performance Graphics* , pp. 113-122, 2010.

[Bou08] S. Boulos, I. Wald, and C. Benthin, "Adaptive ray packet reordering", in *IEEE Symposium on Interactive Ray Tracing*, pp. 131-138, 2008.

[Hec84] P. Heckbert and P. Hanrahan, "Beam tracing polygonal objects", in *Proceedings of the 11th annual conference on Computer graphics and interactive techniques* , pp. 119-127, 1984.

[Ama84] J. Amanatides, "Ray tracing with cones", in *Proceedings of the 11th annual conference on Computer graphics and interactive techniques* , pp. 129-135, 1984.

[Shi87] M. Shinya, T. Takahashi, and S. Naito, " Principles and applications of pencil

tracing", *ACM SIGGRAPH Computer Graphics*, vol. 21, no. 4, pp. 45-54, Jul. 1987.

[Har96] J. Hart, "Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces", *The Visual Computer*, vol. 12, no. 10, pp. 527-545, Dec. 1996.

[Chi12] Y. F. Chiu, Y. C. Chen, C. F. Chang, and R. R. Lee, "Subpixel Reconstruction Anti-aliasing for Ray Tracing", *Journal of WSCG*, vol. 20, no. 3, pp. 171-178, 2012.

[Ble15] Blender Foundation. (2015, Aug.) Blender. [Online]. https://www.blender.org/

[Rag11] J. Ragan-Kelley, J. Lehtinen, J. Chen, M. Doggett, and F. Durand, "Decoupled sampling for graphics pipelines", *ACM Transactions on Graphics (TOG)*, vol. 30, no. 3, Article No. 17, doi: 10.1145/1966394.1966396, May 2011.

[Pha10] M. Pharr and G. Humphreys, *Physically Based Rendering*. 2010.

[Szi00] L. Szirmay-Kalos, "Monte-Carlo Methods on Global Illumination," Institute of Computer Graphics, Vienna University of Technology, 2000.

[Dam09] H. Dammertz, J. Hanika, A. Keller, and H. Lensch, "A Hierarchical Automatic Stopping Condition for Monte Carlo Global Illumination", in *Proceedings of WSCG*, pp. 159-164, 2009.

[Vea97] E. Veach and L. Guibas, "Metropolis light transport", in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* , pp. 65-76, 1997.

[Leh13] J. Lehtinen, T. Karras, S. Laine, M. Aittala, F. Durand, and T. Aila, "Gradient-domain metropolis light transport", *ACM Transactions on Graphics (TOG) - SIGGRAPH Conference Proceedings*, vol. 32, no. 4, Jul. 2013.

[Tal05] J. Talbot, D. Cline, and P. Egbert, "Importance resampling for global illumination", in *Rendering Techniques , Eurographics Symposium on Rendering*, pp. 139-146, 2005.

[Kul12] C. Kulla and M. Fajardo, "Importance Sampling Techniques for Path Tracing in Participating Media", *Computer Graphics Forum*, vol. 31, no. 4, pp. 1519-1528, Jun. 2012.

[Laf93] E. Lafortune and Y. Willems, "Bi-Directional Path Tracing", in *Proceedings of Compugraphics*, pp. 145-153, 1993.

[Geo12] I. Georgiev, J. Křivánek, T. Davidovič, and P. Slusallek, "Light transport simulation with vertex connection and merging", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia* , vol. 31, no. 6, Nov. 2012.

[Tok12] Y. Tokuyoshi and S. Ogaki, "Real-time bidirectional path tracing via rasterization", *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* , pp. 183-190, 2012.

[Lik12] G. Liktor and C. Dachsbacher, "Decoupled deferred shading for hardware rasterization", *Proceeding I3D '12 Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 143-150, ISBN: 978-1-4503-1194-6, doi: 10.1145/2159616.2159640, 2012.

[Kel02] C. Kelemen, L. Szirmay-Kalos, L. Antal, and G. Csonka, "Simple and Robust Mutation Strategy for Metropolis Light", *Computer Graphics Forum*, vol. 21, no. 3, pp. 531-540, 2002.

[Cli05] D. Cline, J. Talbot, and P. Egbert, "Energy redistribution path tracing", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH*, vol. 24, no. 3, pp. 1186-1195, Jul. 2005.

[Jak12] W. Jakob and S. Marschner, "Manifold exploration: a Markov Chain Monte Carlo technique for rendering scenes with difficult specular transport", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH*, vol. 31, no. 4, Jul. 2012.

[Hac14] T. Hachisuka, A. Kaplanyan, and C. Dachsbacher, "Multiplexed metropolis light transport", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH*, vol. 33, no. 4, Jul. 2014.

[Vea98] E. Veach and L. Guibas, "Robust monte carlo methods for light transport simulation", PhD Thesis, Stanford University, Stanford, CA, 1998.

[Kap13] A. Kaplanyan and C. Dachsbacher, "Path Space Regularization for Holistic and Robust Light Transport", *Computer Graphics Forum (Proc. of Eurographics)*, vol. 32, no. 2, pp. 63-72, 2013.

[Eis13] C. Eisenacher, G. Nichols, A. Selle, and B. Burley, "Sorted Deferred Shading for Production Path Tracing", *Computer Graphics Forum*, vol. 32, no. 4, Jul. 2013.

[Hen11] N. Henrich, J. Baerz, T. Grosch, and S. Müller, "Accelerating Path Tracing by Eye Path Reprojection", *International Congress on Graphics and Virtual Reality*, 2011.

[Bog13] D. Bogolepov, D. Ulyanov, D. Sopin, and V. Turlapov, "GPU-Optimized Bi-Directional Path Tracing", in *International Conferences in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2013.

[Jen95] H. W. Jensen and N. J. Christensen, "Optimizing Path Tracing using Noise Reduction Filters", *WSCG*, pp. 134-142, 1995.

[Bur13] C. Burns and W. Hunt, "The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading", *Journal of Computer Graphics Techniques*, vol. 2, no. 2, pp. 55-69, Aug. 2013.

[Leh11] J. Lehtinen, T. Aila, J. Chen, S. Laine, and F. Durand, "Temporal light field reconstruction for rendering distribution effects", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH*, vol. 30, no. 4, 2011.

[Sch12] K. Schwenk and T. Drevensek, "Radiance filtering for interactive path tracing", *ACM SIGGRAPH 2012 Posters* , 2012.

[Bir12] J. Biri and J. Chaussard, "Skeleton based importance sampling for path tracing", in *Eurographics Short Papers*, 2012.

[Cha13] J. Chaussard, L. Noel, V. Biri, and M. Couprie, "A 3D curvilinear skeletonization algorithm with application to path tracing", *Discrete Geometry for Computer Imagery Lecture Notes in Computer Science*, vol. 7749, pp. 119-130, 2013.

[Hac09] T. Hachisuka and H. W. Jensen, "Stochastic progressive photon mapping", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia*, vol. 28, no. 5, Dec. 2009.

[Hav05] V. Havran, J. Bittner, R. Herzog, and H.-P. Seidel, "Ray Maps for Global Illumination", in *Eurographics Symposium on Rendering*, pp. 43-54, 2005.

[Vor11] J. Vorba, "Bidirectional Photon Mapping", in *CESCG*, 2011.

[Hac08] T. Hachisuka, S. Ogaki, and H. W. Jensen, "Progressive photon mapping", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia*, vol. 27, no. 5, Dec. 2008.

[Kap131] A. Kaplanyan and C. Dachsbacher, "Adaptive progressive photon mapping", *ACM Transactions on Graphics (TOG)*, vol. 32, no. 2, Apr. 2013.

[Spe13] B. Spencer and M. Jones, "Progressive photon relaxation", *ACM Transactions on Graphics (TOG)*, vol. 32, no. 1, Jan. 2013.

[Eph06] A. Ephanov and C. Coleman, " Virtual texture: A large area raster resource for the gpu", *The Interservice/Industry Training, Simulation and Education Conference (I/ITSEC)*, 2006.

[Jen98] H. W. Jensen and P. Christensen, "Efficient simulation of light transport in scenes with participating media using photon maps", in *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* , pp. 311-320, 1998.

[Jar08] W. Jarosz, M. Zwicker, and H. W. Jensen, "The Beam Radiance Estimate for Volumetric Photon Mapping", *Computer Graphics Forum (Proceedings of Eurographics)*, vol. 27, no. 2, pp. 557--566, Apr. 2008.

[Jar11] W. Jarosz, D. Nowrouzezahrai, R. Thomas, P.-P. Sloan, and M. Zwicker, "Progressive Photon Beams", *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, vol. 30, no. 6, Dec. 2011.

[Dac05] C. Dachsbacher and M. Stamminger, "Reflective shadow maps", in *Proceedings of the 2005 symposium on Interactive 3D graphics and games* , pp. 203-231, 2005.

[Nov11] J. Novak, T. Engelhardt, and C. Dachsbacher, "Screen-space bias compensation for interactive high-quality global illumination with virtual point lights", in *Symposium on Interactive 3D Graphics and Games* , pp. 119-124, 2011.

[Geo10] I. Georgiev and P. Slusallek, "Simple and Robust Iterative Importance Sampling of Virtual Point Lights", in *Eurographics Short Papers*, 2010.

[Seg061] B. Segovia, J. C. Iehl, R. Mitanchey, and B. Peroche, "Bidirectional Instant Radiosity", in *Proceedings of the 17th Eurographics Workshop on Rendering*, pp. 389-398, 2006.

[Seg07] B. Segovia, J. C. Iehl, and B. Peroche, "Metropolis Instant Radiosity", *Computer Graphics Forum*, vol. 26, no. 3, p. 425–434, Sep. 2007.

[Dav10] T. Davidovič, J. Křivánek, M. Hašan, P. Slusallek, and K. Bala, "Combining global and local virtual lights for detailed glossy illumination", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia*, vol. 29, no. 6, Dec. 2010.

[Sim15] F. Simon, J. Hanika, and C. Dachsbacher, "Rich-VPLs for Improving the Versatility of Many-Light Methods", *Computer Graphics Forum (Proceedings of Eurographics)*, vol. 34, no. 2, pp. 575-584, May 2015.

[Jim11] J. Jimenez, et al., "Filtering Approaches for Real-Time Anti-Aliasing", *ACM SIGGRAPH Courses*, 2011.

[War88] G. Ward, F. Rubinstein, and R. Clear, "A ray tracing solution for diffuse interreflection", *ACM SIGGRAPH Computer Graphics* , vol. 22, no. 4, pp. 85-92, Aug. 1988.

[Kři07] J. Křivánek, "Extension to glossy surfaces: radiance caching", *ACM SIGGRAPH 2007 courses* , 2007.

[Deb09] K. Debattista, P. Dubla, F. Banterle, L. P. Santos, and A. Chalmers, "Instant Caching for Interactive Global Illumination", *Computer Graphics Forum*, vol. 28, no. 8, p. 2216–2228, Dec. 2009.

[Geo121] I. Georgiev, J. Křivánek, S. Popov, and P. Slusallek, "Importance Caching for Complex Illumination", *Computer Graphics Forum*, vol. 31, no. 2, part 3, pp. 701-

710, May 2012.

[Kol04] T. Kollig and A. Keller, "Illumination in the Presence of Weak Singularities", *Monte Carlo and Quasi-Monte Carlo Methods*, pp. 245-257, 2004.

[Eng12] T. Engelhardt, J. Novak, T. Schmidt, and C. Dachsbacher, "Approximate Bias Compensation for Rendering Scenes with Heterogeneous Participating Media", *Computer Graphics Forum (Proceedings of Pacific Graphics 2012)*, vol. 31, no. 7, pp. 2145-2154, 2012.

[Haš09] M. Hašan, J. Křivánek, B. Walter, and K. Bala, "Virtual spherical lights for many-light rendering of glossy scenes", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia 2009*, vol. 28, no. 5, Dec. 2009.

[Nov121] J. Novak, D. Nowrouzezahrai, C. Dachsbacher, and W. Jarosz, "Virtual ray lights for rendering scenes with participating media", *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2012)*, vol. 31, no. 4, pp. 60-71, Jul. 2012.

[Nov122] J. Novak, D. Nowrouzezahrai, C. Dachsbacher, and J. Wojciech, "Progressive Virtual Beam Lights", *Computer Graphics Forum (Proceedings of EGSR 2012)*, vol. 31, no. 4, pp. 1407-1413, Jun. 2012.

[Don09] Z. Dong, T. Grosch, T. Ritschel, J. Kautz, and H.-P. Seidel, "Real-time Indirect Illumination with Clustered Visibility", in *Vision, Modeling, and Visualization Workshop*, 2009.

[Por84] T. Porter and T. Duff, "Compositing Digital Images", *SIGGRAPH '84 Proceedings of the 11th annual conference on Computer graphics and interactive techniques* , pp. 253-259, 1984.

[Pru12] R. Prutkin, A. Kaplanyan, and C. Dachsbacher, "Reflective Shadow Map Clustering for Real-Time Global Illumination", in *Eurographics Short Papers*, p. 2012.

[Rus00] S. Rusinkiewicz and M. Levoy, "QSplat: a multiresolution point rendering system for large meshes", in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 343-352, 2000.

[Rit091] T. Ritschel, T. Engelhardt, T. Grosch, H.-P. Seidel, J. Kautz, and C. Dachsbacher, "Micro-Rendering for Scalable, Parallel Final Gathering", *ACM Trans. Graph. (Proc. SIGGRAPH Asia 2009)*, vol. 28, no. 5, 2009.

[Hol11] M. Hollander, T. Ritschel, E. Eisemann, and T. Boubekeur, "ManyLoDs: Parallel Many-View Level-of-Detail Selection for Real-Time Global Illumination", *Computer Graphics Forum*, vol. 30, no. 4, pp. 1233-1240, Aug. 2011.

[Ols141] O. Olsson, E. Sintorn, V. Kaumlpe, M. Billeter, and U. Assarsson, "Implementing Efficient Virtual Shadow Maps for Many Lights", in *ACM SIGGRAPH 2014 Talks*, 2014.

[Lai07] S. Laine, H. Saransaari, J. Kontkanen, J. Lehtinen, and T. Aila, "Incremental instant radiosity for real-time indirect illumination", in *Proceedings of the 18th Eurographics conference on Rendering Techniques* , pp. 277-286, 2007.

[Wal05] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D. Greenberg, "Lightcuts: a scalable approach to illumination", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2005*, vol. 24, no. 3, pp. 1098-1107, Jul. 2005.

[Wal061] B. Walter, A. Arbree, K. Bala, and D. Greenberg, "Multidimensional lightcuts", *ACM*

Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH, vol. 25, no. 3, Jul. 2006.

[Wal12] B. Walter, P. Khungurn, and K. Bala, "Bidirectional lightcuts", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH*, vol. 31, no. 4, Jul. 2012.

[Hašš07] M. Hašan, F. Pellacini, and K. Bala, "Matrix row-column sampling for the many-light problem", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2007*, vol. 26, no. 3, Jul. 2007.

[Liu09] F. Liu, M. C. Huang, X. H. Liu, and E. H. Wu, "Efficient depth peeling via bucket sort", *Proceedings of the Conference on High Performance Graphics*, pp. 51-57, 2009.

[Tai09] J. Taibo, A. Seoane, and L. Hernández, "Dynamic Virtual Textures", *The 17-th International Conference on Computer Graphics, Visualization and Computer Vision '2009 (WSCG 2009)* , vol. 17, ISSN 1213-6972, 2009.

[Gar08] B. Garney, "Clipmapping on on SM1.1 and Higher", in *Game Programming Gems 7*. Charles River Media, 2008, p. 413–422.

[Mit08] M. Mittring and C. GMBH, "Advanced virtual texture", in *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pp. 23-51, 2008.

[Cuc09] G. Cuccuru, E. Gobbetti, F. Marton, R. Pajarola, and R. Pintus, "Fast low-memory streaming MLS reconstruction of point-sampled surfaces", *Graphics Interface*, pp. 15-22, May 2009.

[Lor87] W. E. Lorensen and H. E. Cline, "Marching Cubes : A high resolution 3D surface reconstruction algorithm", in *SIGGRAPH, Proceedings of the annual conference on Computer Graphics and Interactive Techniques*, 1987.

[Pet11] L. Petrescu, A. Morar, F. Moldoveanu, and V. Asavei, "Real time reconstruction of volumes from very large datasets using CUDA", in *15th International Conference on System Theory, Control, and Computing (ICSTCC)*, pp. 1-5, 2011.

[SAB15] SABIMAS. (2015, Aug.) Personalized implants for hip arthroplasty, (SABIMAS, PNCDII-Joint Applied Research Projects, 2008-2011). [Online]. http://se.cs.pub.ro/SABIMAS/

[NEM15] NEMA. (2015, Aug.) http://medical.nema.org/Dicom/about-DICOM.html. [Online]. http://medical.nema.org/dicom/geninfo/Brochure.pdf

[Mor13] A. Morar, F. Moldoveanu, V. Asavei, L. Petrescu, A. Moldoveanu, and A. Egner, "GPGPU Based Non-photorealistic Rendering of Volume Data", *Control Engineering and Applied Informatics (CEAI)*, vol. 15, no. 1, pp. 45-52, 2013.

[Gro13] M. Grossman, S. Chatterjee, A. Sbîrlea, and V. Sarkar, "Dynamic Task Parallelism with a GPU Work-Stealing Runtime System", *Languages and Compilers for Parallel Computing*, vol. 7146, pp. 203-217, 2013.

[Sin09] E. Sintorn and U. Assarsson, "Hair self shadowing and transparency depth ordering using occupancy maps", *Proceedings of the 2009 symposium on Interactive 3D graphics and games* , pp. 67-74, 2009.

[Gup12] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style for GPU programming for GPGPU workloads", *Innovative Parallel Computing*, pp. 1-14, May 2012.

[Jog13] A. Jog, O. Kayiran, N. Chidambaram, A. Mishra, M. Kandemir, O. Mutlu, R. Iyer,

and C. Das, "OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance", *ACM SIGARCH Computer Architecture News – ASPLOS 13*, vol. 41, no. 1, pp. 395-406, Mar. 2013.

[Tze12] S. Tzeng, B. Lloyd, and J. Owens, "A GPU task-parallel model with dependency resolution", *Computer Journal*, vol. 45, no. 8, pp. 34-41, Aug. 2012.

[Mem12] R. Membarth, J. H. Lupp, F. Hanning, J. Teich, M. Korner, and W. Eckert, "Dynamic task scheduling and resource management for GPU accelerators in medical imaging", in *Proceedings of the 25th International Conference on Architecture Computing Systems*, pp. 147-159, 2012.

[Pet14] L. Petrescu, F. Moldoveanu, V. Asavei, A. Moldoveanu, and O. Ferche, "A GPU Task Generator for Rendering", in *ICSTCC 2014 - 18th International Conference On System Theory, Control and Computing*, pp. 562-567, 2014.

[Rak15] D. Rakos. (2015, Aug.) http://rastergrid.com. [Online]. http://rastergrid.com/blog/2010/10/hierarchical-z-map-based-occlusion-culling/

[Ree15] Reed, Nathan; NVIDIA. (2015, Aug.) https://developer.nvidia.com. [Online]. https://developer.nvidia.com/content/depth-precision-visualized

[Sch06] C. Schüler, "Normal Mapping without Precomputed Tangents", in *ShaderX 5*, 2006.

[Sch15] C. Schüler. (2015, Aug.) http://www.thetenthplanet.de. [Online]. http://www.thetenthplanet.de/archives/1180

[Pet15] L. Petrescu, F. Moldoveanu, V. Asavei, and A. Moldoveanu, "Analyzing Deferred Rendering Techniques", *Control Engineering and Applied Informatics (CEAI)*, accepted, to be published, 2015.

[End10] E. Enderton, E. Sintorn, P. Shirley, and D. Luebke, "Stochastic transparency", *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* , pp. 157-164, 2010.

[Pet151] L. Petrescu, F. Moldoveanu, V. Asavei, and A. Moldoveanu, "Virtual deferred rendering", in *20th International Conference on Control Systems and Computer Science (CSCS)*, pp. 373-378, 2015.

[Had06] M. Hadwiger, A. Kratz, C. Sigg, and K. Buhler, "Gpu-accelerated deep shadow maps for direct volume", in *Proceedings of the 21st ACM SIGGRAPH/Eurographics symposium on Graphics hardware*, New York, pp. 49-52, 2006.

[Pet152] L. Petrescu, F. Moldoveanu, V. Asavei, and A. Moldoveanu, "Guarded Order Independent Transparency", *Scientific Bulletin of University POLITEHNICA of Bucharest, Series C, Electrical Engineering and Computer Science*, vol. 77, no. 1, pp. 3-14, Apr. 2015.

[Kno12] P. Knowles, G. Leach, and F. Zambetta, "Efficient Layered Fragment Buffer Techniques", in *OpenGL Insights*. CRC Press, 2012, ch. 20.

[Cra10] C. Crassin. (2010, Jul.) http://blog.icare3d.org. [Online]. http://blog.icare3d.org/2010/07/opengl-40-abuffer-v20-linked-lists-of.html

[Pet13] L. Petrescu, F. Moldoveanu, A. Moldoveanu, A. Morar, and V. Asavei, "Efficient Picking Through Atomic Operations", in *19th International Conference on Control Systems and Computer Science (CSCS)*, pp. 66-70, 2013.

[Las03] M. Lastra, J. Ravelles, R. Montes, and P. Cano, "A formal framework approach for ray-scene intersection test improvement", in *WCSG Posters Preceedings*, pp. 3-7,

2003.

[Nei93] J. Neider, T. Davis, and M. Woo, *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. Addison-Wesley, 1993.

[Wri10] R. Wright, N. Haemel, G. Sellers, and B. Lipchak, *OpenGL Superbible: Comprehensive Tutorial and Reference*. Addison-Wesley Profesional, 2010.

[Ric12] C. Riccio, "OpenGL 4.2 Review," 2012.

[Sal11] M. Salvi, J. Montgomerrey, and A. Lefohn, "Adaptive transparency", *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* , pp. 119-126, 2011.

[Zho05] K. Zhou, Y. Hu, S. Lin, B. Gao, and H.-Y. Shum, "Precomputed shadow fields for dynamic scenes", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2005*, vol. 24, no. 3, pp. 1196-1201, Jul. 2005.

[Mav11] P. Mavridis and G. Papaioannou, "Global Illumination Using Imperfect Volumes", in *Proceedings of the International Conference on Computer Graphics Theory and Applications (GRAPP)*, 2011.

[And11] D. Andreev, "Anti-Aliasing from a different perspective", in *Game Developers Conference*, 2011.

[Uni11] Unity Technologies. (2011) Unity Reference Manual.. [Online]. http://unity3d.com/support/documentation/Components

[Per15] E. Persson. (2015, Aug.) http://www.humus.name. [Online]. http://www.humus.name/index.php?page=3D&ID=89

[Sal12] M. Salvi and K. Vidimce, "Surface Based Anti-Aliasing", in *ACM SIGGRAPH Symposium on Interactive 3D Rendering and Games*, 2012.

[Ail09] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on GPUs", in *Proceedings of the Conference on High Performance Graphics*, pp. 145-149, 2009.

[Nov10] J. Novak, V. Havran, and C. Daschbacher, "Path Regeneration for Interactive Path Tracing", in *Eurographics (Short Papers)*, pp. 61-64, 2010.

[Res09] Reshetov, Alexander; Intel Labs, "Morphological Antialiasing", 2009.

[Kap14] A. Kaplanyan, J. Hanika, and C. Dachsbacher, "The Natural-Constraint Representation of the Path Space for Efficient Light Transport Simulation", *ACM Transactions on Graphics (Proc. SIGGRAPH)*, vol. 33, no. 4, 2014.

[Che13] J. Chen, D. Bautenbach, and S. Izadi, "Scalable Realtime Volumetric Surface Reconstruction", *ACM Transactions of Graphics*, vol. 32, no. 4, pp. 1-16, Jul. 2013.

[Sal14] M. Salvi and K. Vaidyanathan, "Multi-layer alpha blending", *Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* , pp. 151-158, 2014.

[Dav122] T. Davidovič, I. Georgiev, and P. Slusallek, "Progressive Lightcuts for GPU", in *ACM SIGGRAPH Talks* , 2012.

[Lai10] S. Laine, "Restart Trail for Stackless BVH Traversal", *Proceedings of High-Performance Graphics 2010*, 2010.

[Ail12] T. Aila, S. Laine, and T. Karras, "Understanding the Efficiency of Ray Traversal on GPUs – Kepler and Fermi Addendum", *High-Performance Graphics*, Jun. 2012.

[Mat08] O. Mattausch, J. Bittner, and M. Wimmer, "CHC++: Coherent Hierarchical Culling

Revisited", *Computer Graphics Forum (Proceedings Eurographics 2008)*, vol. 27, no. 2, pp. 221-230, Apr. 2008.

[Gut06] M. Guthe, A. Balzas, and R. Klein, "Near Optimal Hierarchical Culling: Performance Driven Use of Hardware Occlusion Queries", *Eurographics Symposium on Rendering 2006*, 2006.

[Mat15] O. Mattausch, J. Bittner, A. Jaspe Villanueva, E. Gobbetti, M. Wimmer, and R. Pajarola, "CHC+RT: Coherent Hierarchical Culling for Ray Tracing", *Computer Graphics Forum*, vol. 34, no. 2, 2015.

[Zha97] H. Zhang, D. Manchoa, T. Hudson, and K. Hoff, "Visibility culling using hierarchical occlusion maps", in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* , pp. 77-88, 1997.

[McC11] C. McClanahan, "History and Evolution of GPU Architecture," 2011.

[Cla82] J. Clark, "The Geometry Engine: A VLSI Geometry System for Graphics", *ACM SIGGRAPH Computer Graphics*, vol. 16, no. 3, pp. 127-133, 1982.

[NVI12] NVIDIA Corporation, "NVIDIA's Next Generation CUDA Compute Architecture Kepler GK110," NVIDIA, 2012.

[Lau09] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH Construction on GPUs", *Proceedings of EUROGRAPHICS '09*, vol. 28, no. 2, 2009.

[Man13] M. Mantor and M. Houston, "Amd Graphics Core Next Whitepaper," AMD, 2013.

[Low02] K.-L. Low, "Perspective-Correct Interpolation," 2002.

[NVI15] NVIDIA, "CUDA Runtime API version 6.5," 2015.

[KHR14] KHRONOS; Howes, Lee; Munshi, Aftab;, "The OpenCL Specification version 2.0," 2014.

[Mic15] Microsoft. (2015, Aug.) DirectX Blog. [Online]. http://blogs.msdn.com/b/directx/

[AMD15] AMD. (2015, Aug.) Mantle. [Online]. http://www.amd.com/en-us/innovations/software-technologies/mantle

[Vul15] Khronos. (2015, Aug.) Vulkan. [Online]. https://www.khronos.org/vulkan

[Wal01] I. Wald and P. Slussallek, "State of the Art in Interactive Ray Tracing", in *Eurographics*, 2001.

[Pur02] T. Purcell, Y. Buck, W. Mark, and P. Hanrahan, "Ray Tracing on Programmable Graphics Hardware", *ACM Transactions on Graphics*, vol. 21, no. 3, pp. 703-712, Jul. 2002.

[Rit08] T. Ritschel, T. Grosch, M. H. Kim, H. P. Seidel, C. Dachsbacher, and J. Kautz, "Imperfect shadow maps for efficient computation of indirect illumination", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia 2008*, vol. 27, no. 5, Dec. 2008.

[Eis08] E. Eisemann and X. Décoret, "Single-pass GPU solid voxelization for real-time applications", *GI '08 Proceedings of Graphics Interface 2008*, pp. 73-80, ISBN: 978-1-56881-423-0 , 2008.

[Kha06] M. Khazdan, M. Bolitho, and H. Hoppe, "Poisson surface reconstruction", in *Proceedings of the fourth Eurographics symposium on Geometry processing* , pp. 61-70, 2006.

[Iou99] K. Iourcha, N. Krishna, and H. Zhou, "System and method for fixed-rate block-based

image compression with inferred pixel values," US Patent 5,956,431, Sep. 21, 1999.

[Khr15] Khronos. (2015, Apr.) ARB_texture_compression_bptc. [Online]. https://www.opengl.org/registry/specs/ARB/texture_compression_bptc.txt

[Dee95] M. Deering, "Geometry compression", *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* , pp. 13-20, 1995.

[Hop96] H. Hoppe, "Progressive meshes", in *ACM SIGGRAPH Proceedings*, pp. 99-108, 1996.

[Coh98] J. Cohen, M. Olano, and D. Manocha, "Appearance-preserving simplification", in *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* , pp. 115-122, 1998.

[Pra15] A. Pranckevičius. (2015, Apr.) http://aras-p.info/. [Online]. http://aras-p.info/texts/CompactNormalStorage.html

[Mey10] Q. Meyer, J. Süßmuth, G. Sußner, M. Stamminger, and G. Greiner, "On floating-point normal vectors", *Proceedings of the 21st Eurographics conference on Rendering* , pp. 1405-1409, 2010.

[Ziv77] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression", *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337-343, May 1977.

[Kat91] P. Katz, "String searcher, and compressor using same ," US Patent US Patent 5,051,745, Sep. 24, 1991.

[Seg15] M. Segal, K. Akeley, C. Frazier, J. Leech, and P. Brown, *The OpenGL Graphics System : a Specification version 4.5*. USA: The Khronos Group Inc, 2015.

[Col15] Y. Collet. (2015, Apr.) https://code.google.com/p/lz4/. [Online]. https://code.google.com/p/lz4/

[Hop99] H. Hoppe, "New quadric metric for simplifying meshes with appearance attributes", in *EEE Visualization*, pp. 59-66, 1999.

[Los03] F. Lossaso, H. Hoppe, S. Schaefer, and J. Warren, "Smooth geometry images", *Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing* , pp. 138-145, 2003.

[Cra14] C. Crassin and S. Green, "Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer", in *OpenGL Insights*, 2014, ch. 22.

[Wym13] C. Wyman and Z. Dai, "Imperfect voxelized shadow volumes", in *Proceedings of ACM SIGGRAPH Talks* , p. 18, 2013.

[Eis06] E. Eisemann and X. Décoret, "Fast scene voxelization and applications", in *Proceeding I3D '06 Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pp. 71-78, 2006.

[Per85] K. Perlin, "An image synthesizer", in *Proceedings of the 12th annual conference on Computer graphics and interactive techniques* , pp. 287-296, 1985.

[Per01] K. Perlin, "Noise Hardware", in *Real-Time Shading SIGGRAPH Course Notes* , 2001.

[Coo05] R. Cook and T. DeRose, "Wavelet noise", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH*, vol. 24, no. 3, pp. 803-811, Jul. 2005.

[Lag09] A. Lagae, S. Lefebvre, G. Drettakis, and P. Dutré, "Procedural noise using sparse Gabor convolution", *Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH*, vol. 28, no. 3, p. 54, Aug. 2009.

[Coo87] R. L. Cook, L. Carpenter, and E. Catmull, "The Reyes Image Rendering

Architecture", *Computer Graphics*, vol. 21, no. 4, ACM-0-89791-227-6/87/007/0095, Jul. 1987.

[Ger88] S. Gernot, "Image-based object representation by layered impostors", *Proceedings of the ACM symposium on Virtual reality software and technology*, pp. 99-104, 1988.

[Dec03] X. Decoret, F. Sillion, F. Durand, and J. Dorsey, "Billboard clouds for extreme model simplification", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH*, vol. 22, no. 3, pp. 689-696, Jul. 2003.

[And07] C. Andujar, J. Boo, P. Brunet, M. Fairen, I. Navazo, P. Vazquez, and A. Vinacua, "Omni-directional Relief Impostors", *Computer Graphics Forum, Preceedings of Eurographics*, vol. 26, no. 3, 2007.

[Ris06] E. Risser, "True Impostors", in *ACM SIGGRAPH 2006 Research posters*, 2006.

[Har10] A. Hardy and J. Venter, "3-view impostors", *Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa* , pp. 129-138, 2010.

[Pol07] F. Policarpo and M. Oliveira, "Relaxed Cone Stepping for Relief Mapping", in *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley Professional, 2007, ch. 18, pp. 409-428.

[Dec09] P. Decaudin and F. Neyret, "Volumetric Billboards", *Computer Graphics Forum*, vol. 28, no. 8, pp. 2079-2089, 2009.

[Ume05] T. Umenhoffer and L. Szirmay-Kalos, "Real-Time Rendering of Cloudy Natural Phenomena with Hierarchical Depth Impostors", in *EUROGRAPHICS Short Papers*, 2005.

[OHa02] N. O'Hara, "Hierarchical Impostors for the Flocking Algorithm in 3D", *Computer Graphics Forum*, vol. 21, no. 4, pp. 723-731, Nov. 2002.

[Tar04] M. Tarini, K. Hormann, P. Cignoni, and C. Montani, "PolyCube-Maps", in *ACM SIGGRAPH*, pp. 853-860, 2004.

[Tat10] Tatarinov, Andrei; NVIDIA;, "Reyes using DirectX 11", *Proceeding of SIGGRAPH '10, ACM SIGGRAPH 2010 Talks*, ISBN: 978-1-4503-0394-1 , doi: 10.1145/1837026.1837071, Aug. 2010.

[Nov12] J. Novák and C. Dachsbacher, "Rasterized Bounding Volume Hierarchies", *Computer Graphics Forum, Proc. of Eurographics*, vol. 31, no. 2, pp. 403-412, 2012.

[Ber11] R. Berthelot, J. Royan, T. Duval, and B. Arnaldi, "Scene graph adapter: an efficient architecture to improve interoperability between 3D formats and 3D applications engines", *Proceedings of the 16th International Conference on 3D Web Technology*, pp. 21-29, 2011.

[Bar08] J. Barczak, N. Tatarchuk, and C. Oat, "GPU-based Scene Management for Rendering Large Crowds", in *SIGGRAPH Asia Talks*, 2008.

[Mik10] M. Mikkelsen, "Bump Mapping Unparametrized Surfaces on the GPU," 2010.

[Bou081] T. Boubekeur and M. Alexa, "Phong Tessellation", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia*, vol. 27, no. 5, Dec. 2008.

[Vla01] A. Vlachos, J. Peters, C. Boyd, and J. Mitchell, "Curved PN triangles", in *Proceedings of the Symposium on Interactive 3D graphics* , pp. 159-166, 2001.

[Loo09] C. Loop, S. Schaefer, T. Ni, and I. Castanao, "Approximating subdivision surfaces with Gregory patches for hardware tessellation", *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH Asia*, vol. 28, no. 5, Dec. 2009.

[Dyk09] C. Dyken, M. Reimars, and J. Seland, "Semi-uniform Adaptive Patch Tessellation", *Computer Graphics Forum*, vol. 28, no. 8, Sep. 2009.

[Coo82] R. Cook and K. E. Torrance, "A Reflectance Model for Computer Graphics", *ACM Transactions on Graphics (TOG)* , vol. 1, no. 1, pp. 7-24, Jan. 1982.

[Bli78] J. Blinn, "Simulation of wrinkled surfaces", in *Proceedings of the 5th annual conference on Computer graphics and interactive techniques* , pp. 286-292, 1978.