Universitatea POLITEHNICA Bucureşti
Facultatea de Automatică şi Calculatoare

ARHITECTURI ŞI TEHNICI DE PARALELIZARE PENTRU SERVERE DE SPAŢII VIRTUALE 3D MMO

ARCHITECTURES AND PARALLELIZATION TECHNIQUES FOR 3D MMO SERVERS

Doctorand:
Ing. Victor-Iosif Asavei

Conducător ştiinţific:
Prof.Dr.Ing. Florica Moldoveanu

Bucureşti 2011

CUPRINS

LISTA FIGURILOR

LISTA TABELELOR

1 INTRODUCERE

Jocurile de tip Massive Multiplayer Online (MMO) precum şi alte tipuri de spaţii virtuale cum ar fi muzee virtuale, expoziţii, etc, atrag un număr din ce în ce mai mare de utilizatori. Tehnologia cât şi numărul de aplicaţii de acest fel se află într-o continuă dezvoltare şi foarte probabil spaţiile virtuale 3D vor reprezenta următoarea generaţie de interfeţe de comunicare pe Internet, înlocuind sau integrând pe cele actuale (browsere de web, instant messengers, portaluri de socializare etc).

Aplicaţiile de tip MMO se desfăşoară online având ca suport o lume virtuală persistentă la care sunt conectaţi şi în cadrul căreia interacţionează simultan sute şi chiar mii de utilizatori. O lume **persistentă** este o lume virtuală online care continuă să 'funcţioneze' chiar şi atunci când utilizatorul se deconectează şi nu mai participă la acţiunea din lumea virtuală. Utilizatorul se poate conecta înapoi la orice moment de timp şi poate să continue de unde rămăsese în momentul deconectării.

Un utilizator al unei aplicaţii MMO îşi asumă rolul unui caracter fictiv şi apoi controlează acţiunile acestuia. Caracterul fictiv reprezintă 'avatarul' utilizatorului. Aplicaţiile de tip MMO oferă utilizatorilor posibilitatea îndeplinirii unor anumite sarcini cu obiective bine definite. Spre deosebire de aplicaţiile single-user, în aplicaţiile MMO utilizatorii pot îndeplini aceste sarcini fie într-un mod colaborativ împreună cu alţi utilizatori sau pot alege să o facă într-un mod competitiv concurând împotriva altor jucători.

Ca exemple de tipuri de spaţii virtuale 3D MMO putem menţiona sute de jocuri MMO, simulări sociale cum ar fi *Second Life*, aplicaţii de e-learning, edutainment (educatie + entertainment), simulatoare militare pentru antrenament , etc. Spaţiile virtuale au devenit foarte atractive şi din punct de vedere financiar, de exemplu MMO-ul *World of Warcraft ([WOW11])* având o cifra anuală de afaceri ce depăşeşte 1 miliard de dolari.

Din punct de vedere al proiectării, unul dintre obiectivele principale ale unei aplicaţii MMO 3D este acela de a crea un grad de imersiune cât mai ridicat pentru utilizatorii spaţiului virtual - prin calitatea simulării (grafica 3D moderna , sunete

stereo şi spaţiale, simulări realiste ale fizicii, etc.), dinamica interacţiunilor (timp real) şi atractivitatea scenariului.

Principalul atu al unei aplicaţii de tip MMO este însă dat de numărul mare de utilizatori pe care îl poate suporta simultan. Aceasta este principala caracteristică prin care se diferenţiază de aplicaţiile single-user / multiplayer care simulează o lume virtuală 3D.

1.1 MOTIVAŢIE ŞI OBIECTIVE

Principala problema a arhitecturilor curente pentru servere de spaţii virtuale 3D MMO este cea a scalabilităţii acestora, atunci când trebuie sa facă faţă unui număr mare de utilizatori simultan, numărul de operatii ce trebuie sa fie executate ajungând să fie extrem de ridicat în momentele de încărcare maximă. Aceste probleme de scalabilitate apar în mare masura datorita necesitatii de a mentine consistenţa lumii virtuale.

Marea majoritate a spaţiilor virtuale 3D MMO folosesc în prezent o arhitectură de tipul Client-Server. Deşi acest tip de arhitectură pune la dispoziţie funcţionalităţile necesare unei aplicaţii MMO o face cu un cost mare. Pentru a putea simula spaţii virtuale complexe, în cadrul arhitecturii Client-Server se folosesc un număr ridicat de echipamente pentru partea de server şi în ciuda numărului lor mare, clusterele de servere sunt limitate din punct de vedere computaţional iar traficul de reţea introdus de comunicarea cu clienţii nu este nici el de neglijat.

Costurile mari introduse de arhitectura Client-Server limitează practic scalabilitatea acesteia, fiind necesară împărţirea spaţiului virtual în mai multe versiuni distincte ale aceleiaşi lumi virtuale. Deşi aceasta soluţie permite un anumit grad de scalabilitate al aplicaţiilor MMO, ea limiteaza gradul de realism împiedicând interacţiunea unui număr mare de utilizatori.

De asemenea, pentru a obține un nivel de performanţă în prezenţa unui număr mare de utilizatori, producătorii spaţiilor virtuale MMO trebuie sa facă faţă unui cost financiar semnificativ pentru a menţine infrastructura sistemului.

Ţinând cont de aceste limitări, obiectivele principale ale lucrării de faţă sunt următoarele :

- Analiza soluţiilor tehnologice / arhitecturilor actuale folosite pentru redarea spaţiilor virtuale 3D atât la nivelul clientului cât şi la cel al serverului
- Identificarea operaţiilor ce pot fi optimizate, atât la nivelul clientului cât şi la cel al serverului, şi implementarea acestor optimizări folosind tehnici de paralelizare GPGPU
- Propunerea unei soluţii arhitecturale originale pentru a elimina sau reduce o parte din problemele curente de scalabilitate prin :
    - Introducerea în arhitectura serverelor a capabilităţilor de procesare a sarcinilor de calcul folosind tehnici GPGPU
    - Folosirea unui mecanism de alocare dinamică a resurselor care va ţine cont de o serie de factori asociaţi utilizatorilor spaţiului virtual şi operaţiilor executate de aceştia
- Implementarea şi testarea unui prototip funcţional al soluţiei arhitecturale propuse

1.2 PREZENTAREA PE SCURT A CONŢINUTULUI FIECĂRUI CAPITOL

1. SPAŢII VIRTUALE 3D MMO – CONCEPTE ŞI TEHNOLOGII

Acest capitol prezintă conceptele specifice spaţiilor virtuale 3D MMO precum şi tipurile de aplicaţii MMO existente în prezent pe piaţă. De asemenea sunt trecute în revistă pe scurt şi tehnologiile folosite de servere pentru a realiza comunicarea în reţea şi a asigura accesul controlat al utilizatorilor la spaţiul virtual.

Un aspect foarte important atunci când vorbim de simulările 3D, este acela al creării unui grad ridicat de imersiune pentru utilizatori în spaţiul virtual. Pentru a realiza acest lucru, aplicaţiile MMO utilizează pentru partea de client motoare grafice care folosesc tehnologii 3D moderne. În cadrul acestui capitol sunt prezentate şi o parte dintre aceste tehnologii.

2. ANALIZA SOLUŢIILOR ARHITECTURALE ACTUALE PENTRU SERVERE DE SPAŢII VIRTUALE 3D MMO

Acest capitol investighează soluţiile arhitecturale utilizate în prezent în implementarea de spaţii virtuale 3D MMO. Este prezentată în detaliu funcţionarea arhitecturii Client-Server care este cea mai folosită soluţie utilizată în acest moment de aplicaţiile MMO. De asemenea este trecut în revistă şi modul de funcţionare al arhitecturilor Peer-To-Peer şi hibride.

În acest capitol sunt identificate principalele avantaje dar şi lipsuri pentru fiecare din arhitecturile actuale, aceste neajunsuri apărând mai ales din cauza problemelor de scalabilitate ce sunt generate de numărul ridicat de utilizatori ce accesează în acelaşi timp spaţiul virtual.

3. TEHNICI INOVATIVE DE PARALELISM PENTRU SPAŢII VIRTUALE 3D MMO BAZATE PE GPGPU

Acest capitol prezintă la începutul său o descriere a tehnologiei GPGPU (General Purpose on Graphical Processing Units), de ce a fost aleasă această metodă pentru optimizarea operaţiilor efectuate în spaţiile virtuale 3D MMO şi cum se poate implementa paralelizarea operaţiilor folosind arhitectura CUDA.

În continuarea capitolului sunt identificate operaţiile ce pot fi optimizate la nivelul serverului cât şi al clientului, propunându-se pentru acestea soluţii originale de implementare ce folosesc paralelismul atât la nivel single-GPGPU cât şi la nivel multi-GPGPU.

Pentru fiecare dintre implementările realizate se prezintă rezultatele obţinute cât şi o analiză a acestora.

4. ARHITECTURĂ ORIGINALĂ PENTRU SERVERE 3D MMO UTILIZÂND PARALELISMUL GPGPU

În acest capitol se propun soluţii la nivel arhitectural pentru a reduce problemele de scalabilitate cu care se confruntă arhitecturile tradiţionale de servere de spaţii virtuale 3D MMO. Abordările propuse încearcă obţinerea unei scalabilităţi crescute prin introducerea de capabilităţi de procesare single şi multi GPGPU la nivelul

serverelor de spaţii virtuale 3D MMO cât şi a unui mecanism de alocare dinamică a resurselor necesare strâns legat de procesarea GPGPU.

În continuarea capitolului se prezintă o soluţie arhitecturală completă pentru a integra ideile originale propuse anterior în cadrul arhitecturilor serverelor de spaţii virtuale 3D MMO. Se descriu modulele prezente la nivelul arhitecturii precum şi modul lor de funcţionare pentru implementarea soluţiilor propuse.

5. TESTAREA SCALABILITĂŢII SOLUŢIEI

În acest capitol se prezintă rezultatele unor teste ce vizează scalabilitatea soluţiei arhitecturale prezentate în capitolul anterior. În acest scop a fost realizat un prototip funcţional al arhitecturii şi au fost implementate atât aplicaţia server cât şi aplicaţia client necesară testării.

Sunt prezentate detaliat specificaţiile funcţionale ale aplicaţiilor server şi client precum şi modul în care s-a utilizat aplicaţia client pentru a simula un număr mare de clienţi ce accesează spaţiul virtual.

Capitolul se încheie prin prezentarea şi analiza rezultatelor obţinute, care sunt foarte încurajatoare, reprezentând astfel o primă evaluare pozitivă a soluţiilor propuse.

6. CONCLUZII, CONTRIBUŢII ORIGINALE ŞI EVOLUŢII VIITOARE

În partea finală a lucrării se realizează o sinteză a activităţilor de cercetare, a rezultatelor obţinute şi se precizează direcţiile viitoare de aprofundare şi dezvoltare.

De asemenea, se evidenţiază contribuţiile originale ce au fost aduse de către autorul tezei.

2 SPAŢII VIRTUALE 3D MMO – CONCEPTE ŞI TEHNOLOGII

Majoritatea spaţiilor virtuale de tip MMO care există în prezent, sunt jocuri virtuale comerciale de tipul MMOG (massively multiplayer online game).

Aşa cum spune şi numele, aplicaţiile MMOG permit accesul simultan al unui număr mare de jucători, în prezent acest număr variind de la sute până la zeci de mii în cadrul aceleiaşi lumi virtuale. Conexiunea acestora se realizează prin intermediul Internetului, aceste spaţii virtuale fiind practic accesibile oricui în mod gratuit sau cu costuri foarte reduse.

Aceste lumi virtuale prezintă o tema ficţionala, puternic influenţată de fantezie, science-fiction, legende, istorie sau uneori cinematografie.

Felul în care aceasta temă este implementată în lumea virtuală variază de la simpla sursa de inspiraţie pentru denumiri sau aspect al elementelor virtuale, servind doar ca tematică, până la implementarea unui scenariu narativ, pe care utilizatorul îl urmează în timp, urmărind îndeplinirea unor obiective care ii permit avansarea în cadrul jocului.

2.1 CONCEPTE GENERALE

Următoarele concepte generale sunt cele mai des utilizate în cadrul jocurilor MMOG :

1) Avatar

Avatarul este reprezentarea utilizatorului în lumea virtuală, cel mai adesea sub forma unui personaj 3D sau 2D animat.

Avatarul este vizibil atât utilizatorului caruia ii aparţine cât şi celorlalţi utilizatori.

Utilizatorul are posibilitatea de a-şi controla propriul avatar într-un mod mai mult sau mai puţin avansat, elementele de baza prezente în majoritatea jocurilor fiind:

- mişcarea avatarului
- efectuarea unor acţiuni specifice, spre exemplu: saluturi, dans, etc.

- setarea stării emoţionale a avatarului: vesel, trist, atent, etc.

De asemenea, există posibilitatea de a particulariza reprezentarea avatarului, prin:

- setarea unor trăsături fizice: înălţime, forma corporală, facială, culoare păr, ochi etc.
- modificarea hainelor avatarului, alegerea vocii, etc.

2) Personaj

Foarte apropiat de conceptul de avatar este cel de ***personaj*** sau ***caracter***. Acesta reprezintă utilizatorul din punct de vedere al logicii lumii virtuale respective.

Un caracter este reprezentat prin avatar şi este caracterizat în general prin aspecte cum ar fi:

- clasa
- nivel: în general o valoare numerică indicând avansul caracterului pe o scara valorică
- abilităţi
- posesiuni : echipamente, bani (virtuali), etc
- apartenenţa la anumite grupuri, etc

În general, mare parte din timpul de joc este dedicată îmbunătăţirii nivelului, abilităţilor şi posesiunilor personajului.

3) Monştrii (Mobs)

Aceste lumi virtuale nu sunt populate doar de avatarele jucătorilor ci şi de alte categorii de entităţi, cum ar fi monştrii sau NPC (non player characters).

Termenul consacrat „mobs” se referă la entităţi cu care utilizatorul este oarecum obligat de logica jocului să le confrunte pentru a îndeplini anumite obiective sau obține anumite resurse. Spre exemplu, în momentul distrugerii unui monstru, un utilizator poate obține o anumită cantitate de aur sau obiecte virtuale pe care monstrul le „conține”. Distrugerea unor monştrii poate fi condiție de avansare în cadrul unor quest-uri, etc.

Monştrii au reprezentări dintre cele mai variate: animale, extratereştrii, personaje umanoide, etc.

De asemenea, monştrii pot avea niveluri, forţă şi abilităţi diferite, fiind mai greu sau mai uşor de distrus.

Acţiunile acestor entităţi sunt coordonate de către sistem, în general cu ajutorul unor elemente decizionale foarte simple.

4) NPC (Non-player characters)

NPC sunt entităţi ce populează lumea virtuală şi nu se află sub controlul utilizatorului.

NPC nu sunt în general inamici ce trebuie distruşi ci au un rol mai variat, spre exemplu ei putând fi folosiţi pentru:

implementarea anumitor dialoguri cu variante predefinite în cadrul quest-urilor sau pentru a asista utilizatorul
a oferi anumite produse sau servicii în cadrul jocului
însoţitori ce îl ajuta pe utilizator

5) Posesiuni virtuale

Fiecare avatar poate avea anumite posesiuni virtuale, asupra cărora în general are drepturi absolute sau limitate în cadrul jocului respectiv. Spre exemplu:

- arme
- echipamente speciale
- bani
- diverse resurse
- terenuri
- etc.

El poate utiliza, modifica, vinde sau distruge posesiunile sale.

Conceptul de posesiune virtuală a dus şi la crearea unor sisteme de schimb/tranzacționare a acestora, majoritatea jocurilor MMOG incluzând piețe virtuale sau sisteme de licitații („auction-house”), etc.

6) Quest-uri

Un *quest* este o suită de acţiuni pe care utilizatorul trebuie să le îndeplinească pentru a obține o recompensă în cadrul lumii virtuale.

Acţiunile pot fi eliminarea unui număr de monştri de un anumit tip, eliminarea unui monstru foarte puternic, ieşirea dintr-un labirint, cautarea şi găsirea unui obiect special, asamblarea unui obiect din componente, etc.

De regulă, pe parcursul quest-urilor, utilizatorul este ghidat de către un NPC, care ii explică ce are de făcut iar uneori îi acordă ajutor.

Quest-urile pot fi individuale, la nivel de echipă sau ghildă.

7) Grupuri de utilizatori

Utilizatorii se pot organiza în diverse tipuri de grupuri, pentru a colabora în cadrul jocului, spre exemplu în eliminarea monștrilor, în efectuarea unor quest-uri, competiţie împotriva altor utilizatori, etc.

Grupurile de jucători pot avea caracter :

- temporar (echipe/teams/party)
- persistent (ghilde)

În general, orice grup are un conducător, care decide admiterea în sau excluderea din echipă a unui membru.

8) Termeni (acronime) des întâlniţi în jocurile MMOG

AC – armor class : valoarea numerică a „armurii” unui obiect

Alt – un caracter atașat contului unui utilizator altul decât caracterul principal

AoE – Area of Effect : termen asociat unei „vrăji” care afectează un număr mare de monştrii aflaţi într-o zonă largă

AFK – Away from keyboard. Jucătorul nu se mai află în faţa calculatorului. De obicei sistemul de joc trece utilizatorul în aceasta stare după un timp de inactivitate al acestuia

Aggro / Threat – nivelul de amenințare pe care un jucător îl are faţă de un monstru din joc. De obicei într-un grup de mai mulți jucători , monstrul din joc va ataca jucătorul care are „Aggro”-ul cel mai ridicat. Exista mai multe modalităţi prin care acest nivel poate creşte pentru un jucător, variind de la o simplă apropiere de monstrul respectiv până la atacuri directe asupra acestuia.

Aggro Radius – zona din jurul unui monstru în care dacă un jucător intră va genera „Aggro”

Buff – O vrajă benefică care este executată asupra unui caracter din joc

Caster – tip de jucător care pentru atacuri apelează de obicei la vrăji

DOT – Damage over time. Tip de damage care nu afectează cu întreaga valoare un caracter din momentul aplicării ci în decursul unui interval de timp.

DPS – Damage per second. Valoare a atacului instant al unei arme sau a unei vrăji pe secundă. De exemplu dacă o armă are o valoare asociată atacului şi un timp de execuție , DPS-ul este raportul acestora.

De-buff – O vrajă cu efect negativ care este executată asupra unui caracter din joc

Grinding – rămânerea pentru un timp ridicat într-o anumită zonă din joc pentru a lupta cu acelaşi tip de monștri pentru a câștiga fie experienţă , fie resurse care se obțin în urma înfrângerii acestora

Guild – grupare permanentă de jucători , de obicei cu un număr mare al acestora

Health - Sănătatea asociată caracterului unui jucător. Atunci când aceasta este epuizată , caracterul este incapacitat.

Instanţă – o zonă din joc care se copiază pentru fiecare grup care intră în aceasta. Doar grupul care intră în „instanţa” respectivă o poate accesa , aceasta nefiind vizibilă celorlalţi utilizatori.

Loot – resursele care au fost obţinute în urma înfrângerii unui monstru

Mana – atribut necesar pentru a executa vrăji sau atacuri care au asociate ca cerinţă acest atribut. Atunci când aceasta este epuizată nu se mai pot executa vrăjile sau atacurile care necesită o anumita cantitate din acest atribut.

Newbie / Newb – un termen care de obicei desemnează un jucător nou în joc. Câteodată este folosită şi ca apelativ pentru un jucător care nu joaca foarte bine şi face greșeli în joc.

NPC - non player-controlled character. Un caracter care este controlat (parţial sau total) de către calculator.

Party – un grup de jucători care se asociază de regulă pentru îndeplinirea unor obiective comune.

Pet – un NPC care este controlat parţial de jucător

PK – Player killer. Un tip de jucător care de obicei are drept obiectiv înfrângerea celorlalţi jucători prezenți în joc.

PvE - Player Vs. Environment. Acest termen descrie competiția între jucătorii din sistem şi oponenții controlați de către acesta.

PvP – Player Vs. Player. Acest termen descrie competiția directă dintre jucătorii sistemului.

Raid – O grupare de jucători care are de obicei un număr mult mai mare decât un „Party". Aceasta grupare se realizează de obicei atunci când este necesară îndeplinirea unui obiectiv foarte dificil care nu ar putea fi realizat de un număr mic de jucători.

Tank – tip de jucător melee care preferă atacurile directe cu arme

2.2 CATEGORII DE APLICAŢII MMO

Din punct de vedere al jocului, există câteva categorii de tipuri de jocuri MMOG. Aceste categorii sunt următoarele :

- MMORPG: MMO Role-Playing Game – sunt cele mai faimoase şi răspândite tipuri de jocuri MMOG. În acest tip de joc utilizatorul este transpus în rolul unui personaj virtual pe care trebuie să îl dezvolte pe parcursul jocului.
- MMORTS: MMO Real-Time Strategy – acest tip de joc combină tipurile de joc RTS cu o lume virtuală persistentă. De obicei în acest tip de joc , utilizatorul este pus în rolul unui conducător al cărui rol este să conducă o armată în batalii şi să gestioneze resursele necesare pentru a-şi conduce spre victorie armata
- MMOFPS: MMO First-Person Shooter – în acest tip de joc este combinat conceptul de lume virtuală cu cel de FPS accentul punându-se pe luptele directe la scară largă între jucători
- MMOSG: MMO Sports Game – în acest tip de joc , utilizatorii concurează în anumite sporturi cu mare răspândire cum ar fi fotbal , golf, baschet , fotbal american , etc
- MMOMG: MMO Management Game – acest tip de joc nu necesită în general prezenţa utilizatorului în mod permanent în lumea virtuală. În mod uzual utilizatorul se loghează de câteva ori pe săptămână în cont , stabileşte ordinele şi acţiunile care trebuiesc fie îndeplinite de către unităţile care le

controlează şi strategia corespunzătoare pentru a câştiga. De obicei, acest tip de joc este jucat în interiorul unui browser

În continuare vor fi prezentate câteva exemple de jocuri MMOG care sunt foarte răspândite si au un număr ridicat de utilizatori.

World of Warcraft

URL: http://www.worldofwarcraft.com/

Producător: Blizzard Entertainment

Data de lansare: lansarea pentru public a avut loc în 2004
3 expansiuni aparute (The Burning Crusade , Wrath of The Lich King, Cataclysm)

Costuri: abonament lunar / gamecard (14.99 $ / luna)

Descriere:

World of Warcraft este poate cel mai cunoscut joc de tipul MMOG existent în prezent. Cu un număr de jucători activi ce depăşeşte 10 milioane, se poate spune că este de asemenea şi cel mai de succes MMOG din istoria acestui tip de joc. Jucătorul este introdus într-un univers fantasy şi este pus să aleagă între două facţiuni aflate în conflict (Alliance şi Horde). Odată aleasă apartenenţa la una din aceste facţiuni jucătorul este introdus într-un univers fantasy bazat pe jocurile din seria « Warcraft « a cărui poveste i se va dezvălui pe măsură ce avansează în nivel. Jucătorul poate sa aleagă să lupte contra facţiunii adverse în diverse activităţi de tip PVP , sa exploreze conţinutul PVE variat care există în joc fie prin îndeplinirea de questuri sau prin parcurgerea numeroaselor instanţe aflate în joc sau să se ocupe cu câteva dintre numeroasele profesii existente. Jocul excelează prin dimensiunea şi conţinutul foarte bogat oferit atât din punct de vedere al lumii care poate fi explorată cât şi al activităţilor pe care un jucător le poate îndeplini.

Figura 2-1: World of Warcraft

Facilităţi :

- Conţinut PVE
  - Quest-uri
  - Instanţe (5-man , 10-man , 25-man şi 40-man)
- Conţinut PVP
  - 2 facţiuni opuse Alliance şi Horde
  - Arena
  - Battlegrounds
- 10 Rase
  - Alliance : Draenei, Dwarves , Gnomes, Humans, Night Elves
  - Horde : Blood Elves, Orcs, Tauren , Trolls , Undead
- 10 Clase : Death Knight(Hero Class), Druid , Hunter , Mage , Paladin , Priest , Rogue , Shaman , Warlock , Warrior

- Sistem de profesii : Alchemy , Blacksmithing , Cooking , Enchanting , Engineering , First Aid , Fishing , Herbalism , Leatherworking , Mining , Skinning , Tailoring , Jewelcrafting , Inscription
- Sistem de health şi mana
- Sistem de talente pentru fiecare clasă
- Sistem de achievements
- Sistem de transport public şi mounts pentru jucători
- Posibilitate de alcătuire party de maxim 5 persoane
- Posibilitate de alcătuire raid de maxim 40 persoane
- Outdoor bosses
- Sistem de ghilde
- Action-House, Mail şi Bank în-game
- 4 continente (Azeroth , Kalimdor , Northrend , Outland)
- Evenimente in-game
- Content nou adăugat periodic
- Forumuri

Lord of the Rings Online

URL: http://www.lotro.com/

Producător: Turbine

Dată de lansare: lansarea pentru public a avut loc în 2007
2 expansiuni apărute (Mines of Moria, Siege of Mirkwood)

Costuri: free 2 play / microtranzacţii

Descriere:

Lord of the Rings Online este construit pe universul creat de J.R.R. Tolkien în cărţile « The Hobbit » şi « Lord of The Rings ». Jocul redă cu foarte mare fidelitate locaţiile din Middle-Earth îndrăgite de fanii seriei şi se bazează pe o grafică modernă şi foarte detaliată , acest lucru contribuind foarte mult la atragerea jucătorilor. Acestora li se ofera posibilitatea urmăririi şi implicării directe în povestea descrisă în cărţile « Lord of The Rings » prin îndeplinirea obiectivelor din lanţul de questuri « Epic » care este organizat sub forma de cărţi. De asemenea jucătorii vor descoperi detalii despre lumea « Middle-earth »-ului şi pot înfrunta multe dintre pericolele existente în universul « Lord of The Rings » prin îndeplinirea de questuri şi instanţe. Jocul excelează prin grafica oferită şi prin imersiunea ridicată a jucătorilor în universul virtual creat.

Figura 2-2: Lord of The Rings Online

Facilităţi :

- Conţinut PVE
  - Quest-uri
  - Instanţe (6-man , 12-man , 24-man)
- Conţinut PVP : deşi nu exista facţiuni opuse , jocul ofera posibilitatea de PVP prin implementarea mecanismului de „Monster Play” unde fiecare jucător poate să aleagă să impersoneze un monstru având posibilitatea să joace contra monştrilor controlaţi de alţi jucători într-o instanţă specială
- 4 Rase : Man , Elf , Dwarf, Hobbit
- 9 Clase : Burglar, Captain, Champion, Guardian, Hunter , Lore-master , Ministrel , Rune-keeper, Warden
- Sistem de vocaţii : în funcţie de vocaţia aleasă vor fi disponibile câte 3 profesii distincte :
  - Armorer : Prospector, Metalsmith, Tailor
  - Armsman : Prospector, Weaponsmith, Woodworker
  - Explorer : Forester, Prospector , Tailor
  - Historian : Farmer , Scholar , Weaponsmith

  - Tinker : Prospector , Jeweller , Cook
  - Woodsman : Forester , Farmer , Woodworker
  - Yeoman : Farmer , Tailor , Cook
- Sistem de morală (health) şi power (mana)
- Sistem de skills pentru fiecare clasă
- Sistem de individualizare a caracterului prin : deeds, traits , titles
- Sistem de muzică : odată cu nivelul 5, fiecare caracter poate să compună muzica proprie folosind instrumente specifice
- Sistem de transport public şi mounts pentru jucători
- Posibilitate de alcătuire party de maxim 6 persoane (fellowship)
- Posibilitate de alcătuire raid de maxim 24 persoane
- Action-House, Mail şi Bank in-game
- Sistem de ghilde (kinship)
- Posibilitatea de explorare a Middle-Earth (cu câteva locaţii bine cunoscute cum ar fi : Shire , Bree , Rivendell , Moria , Lorien)
- Evenimente in-game
- Conţinut nou adăugat periodic
- Forumuri

Warhammer Online

URL: http://www.warhammeronline.com/

Producător: Mythic Entertainment

Dată de lansare: lansarea pentru public a avut loc în 2008

Costuri: abonament lunar / gamecard (14.99 $ / luna)

Descriere:

Warhammer Online este un joc în care accentul se pune în principal pe gameplay-ul de tip PVP. Jocul este unul de tip Realm vs Realm în care două facțiuni se luptă pentru a cuceri realm-urile inamicului. În acest tip de joc , utilizatorii fac parte din armate aparținând unei facțiuni şi luptele între armate sunt date la scară largă cu participarea unui număr mare de jucători.

Jocul fiind axat în principal pe PVP, există un număr foarte mare de clase disponibile jucătorilor în funcţie şi de rasa aleasă, cu roluri bine definite : tank , melee şi ranged dps , healer.

Figura 2-3: WarHammer Online

Facilităţi :

- Conţinut PVE
  - Public Quest
- Conţinut PVP : Realm vs Realm prin :
  - Skirmish
  - Battlefields
  - Scenarii
- 2 facţiuni organizate sub forma a două armate :
  - Order
  - Destruction
- 6 Rase în funcţie de facţiune :
  - Order : Dwarfs , Empire (Humans) , High Elves
  - Destruction : Greenskins (Orcs , Goblins), Chaos (Humans), Dark Elves
- 24 de Clase (Cariere) în funcţie de rasă (câte 4 pentru fiecare rasa cu rolurile de Tank , Melee DPS, Ranged DPS şi Healer) :
  - Dwarfs : Ironbreaker , Slayer , Engineer, Runepriest

  - Empire : Knight of the Blazing Sun, Witch Hunter , Bright Wizard , Warrior Priest
  - High Elves : Swordmaster , White lion , Shadow Warrior , Archmage
  - Greenskins : Black Orc, Choppa , Goblin Squig Herder , Goblin Shaman
  - Chaos : Chosen , Marauder , Magus , Zealot
  - Dark Elves : Black Guard , Witch Elf , Sorceress/Sorcerer, Disciple of Khaine
- Sistem de health şi action points (mana)
- Collision detection avansat între personajele jucătorilor creând astfel noi mecanici de joc
- Sistem de skills pentru fiecare clasă bazat pe specializări
- Sistem de transport public şi mounts pentru jucători
- Posibilitate de alcătuire party de maxim 6 persoane
- Posibilitate de alcătuire raid (warbands) de maxim 24 persoane
- Action-House, Mail şi Bank în-game
- Conţinut nou adăugat periodic
- Forumuri

2.3 TEHNOLOGII CLIENT-SIDE

Aşa cum s-a precizat şi anterior, un aspect foarte important, atunci când vorbim de simulările 3D, este acela al imersiunii utilizatorului în cadrul lumii virtuale. Pentru a realiza acest lucru, trebuie sa existe pe partea de client o redare grafică care foloseşte tehnologii 3D moderne pentru a avea :

- o reprezentare foarte detaliată a lumii virtuale, acest lucru presupunând reprezentarea obiectelor din spaţiul virtual cu un număr ridicat de poligoane;
- efecte vizuale spectaculoase cum ar fi :
  - fum
  - explozii
  - umbre dinamice
  - texturi detaliate, etc

OpenGL şi Direct3D, principalele biblioteci dedicate dezvoltării de aplicaţii grafice 3D, permit crearea şi redarea de spaţii 3D în care un observator virtual poate naviga folosind mouse-ul sau tastatura. Scena este descrisa prin modelele 3D ale obiectelor, pozitionarea acestora şi a observatorului în scena, proprietăţile de material ale suprafeţelor obiectelor, pozitionarea şi proprietăţile surselor de lumina, texturile care trebuie aplicate pe suprafeţele obiectelor, şi altele.

2.3.1 BANDA GRAFICĂ (GRAPHICS PIPELINE)

Componenta grafica a oricărei lumi virtuale este probabil domeniul cel mai dinamic, aflat mereu în căutare de mai multa performanţă, atât la nivel de arhitecturi hardware şi software cât şi la nivel de algoritmi [PET97a] şi detalii de implementare.

Modelul scenei 3D, descris de o aplicaţie apelând funcţiile interfeţei de programare (API) OpenGL / Direct3D, poate fi compus din simple primitive grafice (linie, triunghi, poligon), primitive grafice conectate (set de triunghiuri conectate, set de poligoane conectate), obiecte 3D, cum ar fi sfera, şi suprafeţe de forma libera (suprafeţe Bezier, B-spline, Nurbs). Bibliotecile OpenGL şi Direct3D discretizează suprafeţele obiectelor 3D şi pe cele de forma libera aproximându-le printr-o reţea spaţiala conectată de faţete poligonale (patrulatere sau triunghiuri).

2.3.1.1 Banda grafică fixă

Transformarea descrierii scenei 3D într-o imagine afişată pe ecran este realizata automat de API-urile 3D, printr-un lanţ de prelucrări fix. Lanţul de prelucrări constă din etapele redate în figura următoare:

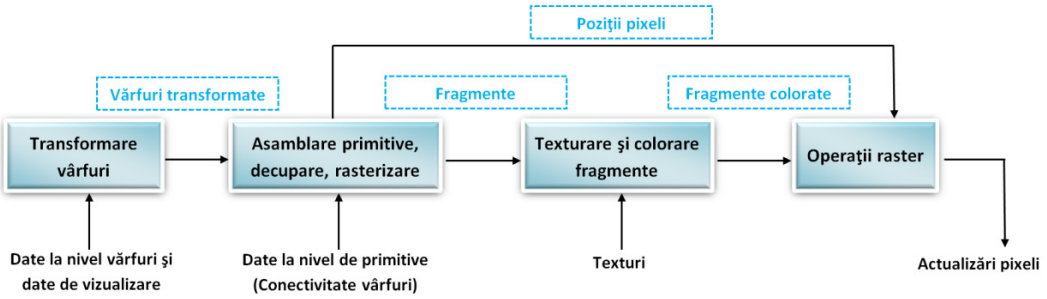

Figura 2-4: Banda grafică fixă (clasică)

Deoarece fiecare etapa primește intrări de la etapele anterioare şi transmite rezultate pentru etapele următoare, secvenţa de operaţii este asemănată cu o "bandă de asamblare" într-un proces de fabricație. Deoarece obiectul fabricat în acest caz este imaginea unei scene 3D, banda este denumită „banda grafică" („graphics pipeline").

Operațiile din **banda grafică fixa**, redată în figura de mai sus, sunt aceleaşi pentru toate primitivele unei scene 3D.

În cele ce urmeaza vom explica pe scurt aceste operaţii.

2.3.1.1.1 Transformarea vârfurilor

Operaţiile din aceasta etapă se efectuează asupra fiecărui vârf. Datele de ieşire ale etapei sunt folosite în etapa următoare a benzii grafice. Astfel, poziţia vârfului, care este specificată în spaţiul obiect, este transformată în spaţiul de proiecţie (spaţiul de decupare) pe baza matricei de modelare-vizualizare-proiecţie (specificata ca un parametru global de vizualizare). Dacă poziţia este singura dată asociată vârfului, atunci toţi pixelii primitivei din care face parte vor fi afişaţi în aceeaşi culoare, specificată ca o dată globală de vizualizare (iluminare constantă).

2.3.1.1.2 Asamblare primitive, decupare, rasterizare

În aceasta etapa vârfurile sunt conectate în primitive şi primitivele sunt supuse operaţiei de decupare la marginile volumului vizual canonic (view frustum clipping).

Din decupare rezultă noi vârfuri. Orice nou vârf este situat pe o latura a primitivei decupate. În funcţie de poziţia noului vârf pe latura decupată, se calculează prin interpolare datele asociate noului vârf: culoarea, coordonatele textură şi coordonata ceţii.

Poziţiile vârfurilor primitivelor rezultate dupa decupare sunt transformate din spaţiul de decupare în spaţiul ecran apoi primitivele sunt "rasterizate", adică descompuse în rastru de puncte corespunzător spaţiului discret al suprafeţei de afişare. Rezultatul rasterizării este un set de adrese de pixeli şi un set de "fragmente" de primitivă care trebuie sa fie afişate în pixeli.

În acest punct , trebuie făcuta foarte clar distincţie între termenul de "pixel" şi "fragment".

Un pixel reprezintă continutul framebuffer-ului la o locaţie specifică cum ar fi culoarea, adâncimea şi alte valori asociate cu locaţia respectivă.

Un fragment este un “potential pixel”. Dacă fragmentul trece testele de rasterizare (în etapa de Operaţii raster) el va actualiza pixelul caruia ii este asociat în framebuffer.

2.3.1.1.3 Texturare şi colorare fragmente

Fiecare fragment are asociată o adresă de pixel, o valoare de adâncime (corespunzătoare coordonatei Z a fragmentului, ce reprezintă distanţa sa faţă de observatorul virtual al scenei 3D) şi o culoare. De asemenea, fragmentului îi mai pot fi asociate unul sau mai multe seturi de coordonate textură şi coordonata de ceaţă. Coordonatele textură şi coordonata ceţii se obţin prin interpolarea valorilor atașate vârfurilor primitivei din care face parte fragmentul.

Modelul de colorare (shading)

OpenGL şi Direct3D permit aplicaţiilor specificarea modului de calcul al culorii fragmentelor: aceeași culoare pentru toate fragmentele unei primitive, preluată de la unul dintre vârfuri, sau calculul culorilor fragmentelor prin interpolarea culorilor asociate vârfurilor după etapa de transformare a vârfurilor (modelul Gouraud).

Texturare

Coordonatele textură permit calculul unei culori prin accesarea unei texturi (1, 2 sau 3-dimensionale). Culoarea textură se combină cu culoarea fragmentului folosind diverşi operatori. Transparenţa texturii poate fi controlată prin canalul alfa.

În cazul multi-texturării (fragmentul are asociate mai multe seturi de coordonate textură), culoarea fragmentului se combină succesiv cu culorile extrase din diferite texturi. Multi-texturarea permite obținerea de efecte care măresc realismul scenei sintetizate.

2.3.1.1.4 Operaţii raster

Operațiile raster se efectuează asupra fiecărui fragment înainte de actualizarea buffer-ului imagine (framebuffer). Ele pot fi specificate prin interfețele de programare OpenGL şi Direct3D. În această etapa sunt eliminate părţile nevizibile ale primitivelor care alcătuiesc scena 3D, pe baza testului de adâncime (algoritmul z-buffer). Alte operaţii raster sunt: testul de apartenenţă a pixelului la contextul OpenGL curent, testul de decupare la marginile unei ferestre din zona de afișare (scissor test), testul alfa, testul stencil, testul de combinare (blending) precum şi diverse operaţii logice între culoarea fragmentului şi culoarea existentă în buffer-ul imagine.

Culorile finale ale fragmentelor care "trec" de operațiile raster sunt afișate în pixelii corespunzători.

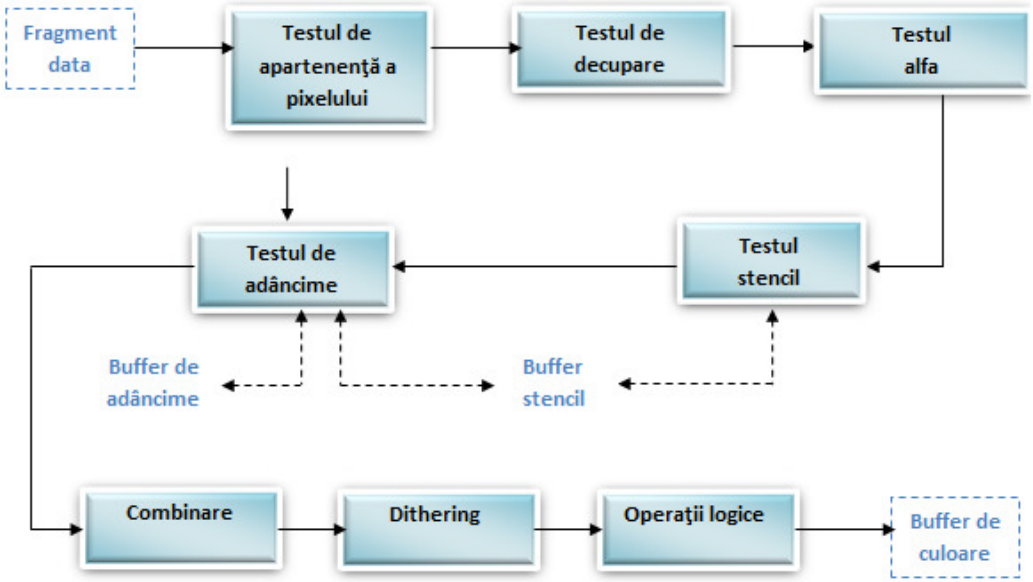

Figura 2-5: Operaţii raster

2.3.1.2 Banda grafică programabilă

Modul în care sunt implementate operațiile din banda grafică fixă depinde de suportul hardware oferit de unitatea (placa) grafică a calculatorului. Unitatea de Prelucrare Grafică poate implementa diferite nivele de accelerare grafică, de la un sistem raster capabil sa rasterizeze linii şi poligoane, până la un sistem complex de procesoare capabile sa transforme şi sa prelucreze date geometrice, cu un înalt grad de paralelism. Astfel, implementarea fiecărei comenzi OpenGL/Direct3D este împărțită în mod optim între CPU (Unitatea de Prelucrare Centrala a Calculatorului) şi GPU, în funcţie de capacitatea de prelucrare a GPU-ului.

Ultimele versiuni ale bibliotecilor OpenGL/Direct3D includ suportul necesar pentru implementarea conceptului de ***bandă grafică programabilă***. Programatorul are posibilitatea de a furniza programele care sunt executate la nivel de vârf, respectiv fragment, înlocuind prelucrările standard din banda grafică fixă. Aceste programe se numesc ***vertex shader*** şi ***pixel shader*** deoarece principala lor utilizare este în calculul culorii de afișare a pixelilor. Ele pot fi implementate în limbaje de nivel înalt: **Cg** (**C for Graphics**) `[FEK06]` oferit de NVIDIA şi **HLSL** (**High Level Shading Language**) dezvoltat de Microsoft pentru a fi utilizat împreună cu

Direct3D (de fapt, Cg şi HLSL reprezintă un acelaşi limbaj, rezultat din colaborarea dintre NVIDIA şi Microsoft), analog cu **GLSL** (**OpenGL Shading Language**).

Prin intermediul programelor vârf (vertex shader) şi pixel (pixel shader) programatorul poate implementa modalităţi diverse de transformare a vârfurilor, diferite de cele standard, de exemplu pentru realizarea animației, precum şi calcule de iluminare mai exacte la nivel de fragment. De exemplu, în loc să se interpoleze culorile vârfurilor pentru obținerea culorilor fragmentelor interioare, se poate calcula o culoare pentru fiecare fragment folosind normala interpolată (modelul Phong), sau o normală citită dintr-o „harta de normale", etc.

Primul model de programare a Unităţii de Prelucrare Grafice bazat pe vertex shader şi pixel shader ("shader model 1") a apărut odată cu generația a 4-a de GPU (incepand cu 2003 : NVIDIA GeForce FX, ATI Radeon 9700). Generația anterioară suporta programabilitatea numai la nivel de vârf. Penultimul model de programare a Unităţii de Prelucrare Grafică, «**shader model 4**» conține încă o etapă programabilă în banda grafică, implementată prin "**geometry shader**", care permite crearea de noi primitive în banda grafică. Acest model a fost introdus în noiembrie 2006 de Direct3D10 şi GLSL1.2 cu suport hardware GPU NVIDIA 8800GTX (2006) şi Radeon 2900XT (2007). Ultimul model «**shader model 5**», apărut in 2008, conţine etape programabile dedicate tehnicii de „tesselation" si pentru calcule GPGPU.

Unitățile de Prelucrare Grafică moderne, cu care sunt dotate tot mai multe calculatoare personale în prezent, pot executa integral operațiile din banda grafică. Execuția unei aplicaţii OpenGL/Direct3D pe un astfel de calculator este mult mai rapidă, redarea scenelor 3D cu texturi sofisticate, lumini şi umbre putând fi realizata în timp real. Aceasta deoarece CPU-ul este eliberat de operațiile necesare la nivelul fiecărui vârf şi pixel, fiind disponibil pentru alte operaţii ale aplicației. Totodată, execuția de către GPU a operațiilor din banda grafică este mult mai rapidă, atât datorită specializării procesoarelor pentru operațiile respective cât şi datorită paralelizării ridicate a prelucrărilor. Datorită specializării hardware şi paralelismului operațiilor la nivelul GPU, sinteza imaginilor foto-realiste în timp real a devenit posibilă. De asemenea, animația personajelor şi alte metode de animație pot beneficia din plin de puterea de calcul a GPU-ului.

Figura următoare reda schematic etapele de prelucrare din "banda grafică programabilă", conform modelului de programare a Unităţii de Prelucrare Grafica 4 ("shader model 4") [FAT08].

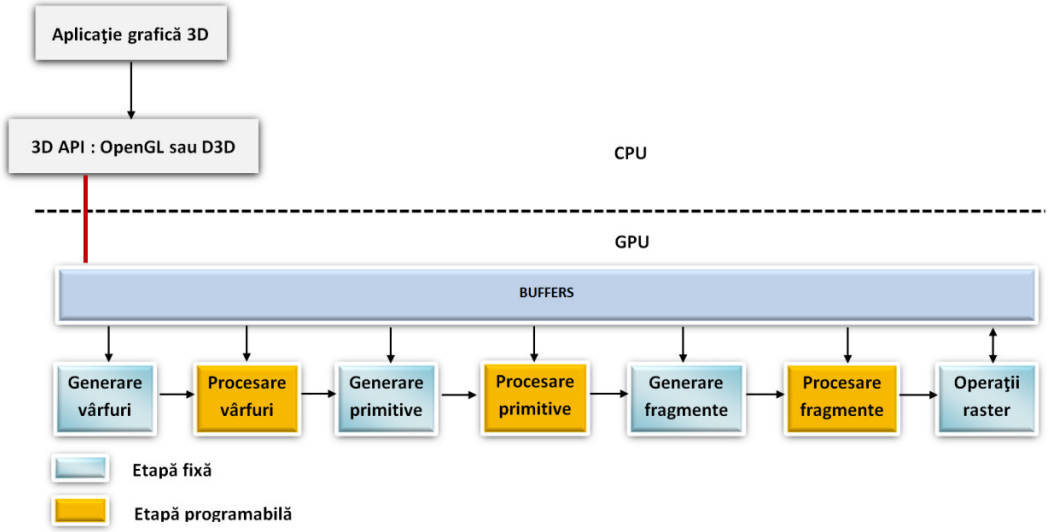

Figura 2-6: Banda grafică programabila

Banda grafică programabilă conține 7 etape, dintre care 4 sunt fixe şi 3 sunt programabile.

2.3.1.2.1 Generare vârfuri

Fiecare primitivă grafică este definită într-o aplicație OpenGL/Direct3D printr-un set de vârfuri. Pornind de la aceste vârfuri, în această etapă, se creează un şir (stream) de înregistrări "vârf" care se transmit mai departe în banda grafică. Fiecare înregistrare conține coordonatele unui vârf, împreună cu alte date asociate vârfului (culoare, normală, coordonate textură). Acestea sunt intrările pentru "vertex shader", care este executat în paralel, pentru diferite înregistrări vârf, de mai multe procesoare ale GPU.

2.3.1.2.2 Prelucrare vârfuri

Această etapă este programabilă şi poate fi implementată printr-un program vârf ("vertex shader"). Atunci când este activ un program vârf, el este apelat pentru fiecare vârf produs în etapa anterioară. Principala operație realizată într-un program vârf este transformarea coordonatelor vârfului în spaţiul de proiecție şi calculul

culorii vârfului. Aici însă pot fi programate şi alte transformări asupra vârfului, de exemplu pentru animaţia personajelor (“skinning”), “morphing”, simularea valurilor, etc. În afara datelor primite din banda grafică, programul vârf utilizează şi date din buffer-ele globale ale memoriei GPU, de exemplu, poziţia observatorului, matricea de proiecţie, poziţia sursei de lumină şi alte date specifice aplicaţiei.

Ieşirea programului vârf este o înregistrare “vârf de ieşire”, conţinând coordonatele transformate ale vârfului, culoarea sa, una sau mai multe coordonate textură.

La un moment dat, exista un singur program vârf, care se executa asupra tuturor vârfurilor primitivelor.

2.3.1.2.3 Generare primitive

Această etapă nu este programabilă. Scopul sau este de a asambla vârfurile în primitive, folosind topologia fiecărei primitive definită în aplicaţie. Este generat un şir ordonat de înregistrări “primitivă”, rezultate prin concatenarea înregistrărilor vârf produse de programul vârf.

2.3.1.2.4 Prelucrare primitive

Este o etapă programabilă, ce poate fi implementată printr-un program “primitivă” (“primitive shader” sau “geometry shader”). Acest program acţionează asupra unei singure primitive (primitivă de intrare), care poate fi : un singur vârf, o linie (2 vârfuri) sau un triunghi. De asemenea, datele de intrare pot fi vârfurile primitivelor adiacente cu primitiva de intrare: două vârfuri pentru o linie (extremităţile segmentelor adiacente), trei vârfuri pentru un triunghi (vârfurile suplimentare ale triunghiurilor adiacente). Programul poate produce un nou şir ordonat de primitive : şir de vârfuri (PointStream), şir de linii (LineStream) sau şir de triunghiuri (TriangleStream).

Atunci când este activ un program “primitivă”, el este apelat o dată pentru fiecare primitivă generată în etapa anterioară.

Exemple de algoritmi care pot fi implementaţi în programul primitivă sunt: sisteme de particule dinamice, generare de “sprite”-uri, generarea volumului de umbră al primitivei, generarea terenurilor, s.a.

2.3.1.2.5 Generare fragmente

În această etapă este efectuată decuparea primitivelor la marginile volumului vizual şi apoi rasterizarea lor, rezultatul fiind un şir de înregistrări “fragment”, care sunt

transmise procesoarelor de prelucrare a fragmentelor. Fiecare înregistrare fragment conține poziția pixelului în care va fi afișat fragmentul, coordonata z a fragmentului şi valorile rezultate din interpolarea datelor atașate vârfurilor (culori, coordonate textură).

2.3.1.2.6 Prelucrare fragmente

Prelucrarea din această etapă poate fi programată printr-un program "fragment" şi are drept scop calculul culorii şi al opacității unui fragment. În acest scop pot fi folosite texturi 1D, 2D sau 3D, se pot simula efecte de reflexie şi refracție a luminii de către suprafețele obiectelor din scena 3D.

2.3.1.2.7 Operaţii raster

În această etapă sunt eliminate fragmentele obturate de suprafețe aflate mai aproape de observator (algoritmul z-buffer), se combină culoarea fragmentului şi culoarea curentă a pixelului folosind opacitatea şi diverşi operatori raster.

Programele "shader" folosesc, nu numai datele primite prin şirurile de înregistrări ale benzii grafice, ci şi date memorate în buffere globale existente în memoria GPU (memoria video). Aceste buffere sunt inițializate cu datele necesare programelor "shader", furnizate de programul de aplicație.

2.3.2 ORGANIZAREA ŞI REPREZENTAREA SCENEI 3D

Alegerea unei structuri de date adecvate pentru reprezentarea scenei este extrem de importantă pentru optimizarea operațiilor necesare redării sale. Astfel, una dintre optimizările cele mai importante este excluderea din banda grafică a obiectelor care nu sunt vizibile în imagine. Într-o aplicație OpenGL această excludere este realizată automat la nivelul fiecărei primitive geometrice, în banda grafică, prin operația de decupare („culling") la marginile volumului vizual definit de programator. O metodă mai eficientă ar fi excluderea la nivel de obiect şi nu de primitive în care a fost descompus obiectul. Dar şi mai eficient este să se poată exclude un întreg grup de obiecte nevizibile. Evident, între obiectele grupului trebuie sa existe o relație spațială sau logică. Ideea de a grupa obiectele este utilă şi în privința transformărilor. De multe ori, aceeași transformare se aplică tuturor primitivelor în care este descompus obiectul atunci când este transmis benzii de redare.

Deziderateleor menționate mai sus au condus la ideea managementului scenei 3D la un nivel superior celui oferit de OpenGL sau Direct3D.

2.3.2.1 Managementul ierarhic al scenei

Structura ierarhică utilizată de regulă pentru reprezentarea scenei este de tip **arbore**, în care fiecare nod are un părinte şi orice număr de copii, iar fiecare copil are un singur părinte. O structura mai generala este de tip **graf aciclic orientat (DAG),** care permite reprezentarea partajării obiectelor (un nod poate avea mai mulţi părinţi). Structura ierarhică folosită pentru reprezentarea scenei este cunoscută sub numele de „**graful scenei**" (**scene-graph**).

Nodurile frunză conţin primitivele care sunt transmise benzii grafice dar şi alte date care au un efect final asupra scenei redate în imagine, de exemplu lumini sau sunet. Celelalte noduri au scop de grupare, de aceea sunt denumite **noduri grup.**

Nodurile arborelui conţin **Informaţii spaţiale** şi **informaţii semantice** ca: transformări, volume încadratoare, Informaţii de redare şi de animaţie. Transformările sunt utilizate pentru poziţionarea, orientarea şi scalarea obiectelor din ierarhie. Volumele încadratoare sunt utilizate pentru testul de vizibilitate (apartenenţă la volumul vizual) şi în detecţia coliziunilor (testele de intersecţie). Informaţiile de redare ataşate unui nod pot fi, de exemplu, o sursă de lumină care luminează obiectele din subarborele nodului, nivelul de ceaţă, texturile folosite la redare şi altele.

Exemple de tipuri de noduri în graful unei scene sunt: "cub", "sfera", "text", "teren", "skyBox", "arbore octal", "mesh", "sursă de lumină", "transformare", "cameră", "sistem de particule", etc. Indiferent de tipul sau, fiecărui nod ii este asociata o poziţie în spaţiul scenei.

Modificările care apar în scena 3D pe parcursul execuţiei se traduc în modificări ale atributelor nodurilor arborelui. O schimbare a unui nod afectează subarborele a cărui rădăcină este. Modificarea trebuie transmisă tuturor nodurilor subarborelui. Astfel, schimbarea poziţiei unui obiect reprezentat printr-un nod intern are ca efect schimbarea transformărilor care afectează nodurile descendente în graful scenei, precum şi schimbarea volumului încadrator al fiecărui nod.

2.3.2.2 Reprezentări bazate pe sortare spaţială

Reprezentarea ierarhică a scenei permite evitarea transmiterii în banda grafică a unui număr mare de obiecte nevizibile. Totuşi, un număr mare de primitive care sunt incluse în volumul vizual dar se suprapun în imagine sunt rasterizate, iar

problema vizibilității este rezolvată la nivel de pixel, prin testul de adâncime (coordonatele z, în algoritmul z-buffer).

Ideea de bază a sortării spațiale a obiectelor scenei este de a împiedica, pe cât posibil, "scrierea" de mai multe ori a unui pixel (în buffer-ul imagine) în procesul de formare a imaginii unei scene 3D. Termenul "depth complexity" se referă la numărul de scrieri al unui pixel în acest proces. Se dorește ca aceasta complexitate sa fie 1. Cu cât este mai mare cu atât timpul necesar formării imaginii este mai mare.

2.3.2.2.1 Arborele BSP (Binary Space Partitioning)

Arborele BSP este o metodă frecvent folosită pentru sortarea spațială a obiectelor scenei 3D. Arborele BSP, prin care se reprezintă scena 3D, se obține prin partiționarea binară, recursivă, a scenei, folosind plane de partiționare.

Reprezentarea este independentă de poziția observatorului, de aceea este convenabilă în aplicaţiile în care scena nu se modifică pe parcursul execuției aplicației, arborele fiind construit într-o etapa de pre-procesare.

Construirea arborelui

De regulă, planele de partiționare sunt planele poligoanelor în care a fost descompusă scena 3D. Construirea scenei poate începe cu oricare dintre poligoane sau cu unul ales astfel încât numărul de divizări de poligoane la construirea arborelui să fie redus. Nodul rădăcină reprezintă întreaga scenă. Lui i se atașează poligonul de start. Faţă de planul acestui poligon (planul de partiționare), celelalte poligoane sunt clasificate în două liste: lista poligoanelor „din faţă" şi lista poligoanelor „din spate". Clasificarea se face pe baza orientării normalei la planul poligonului (poligoanele sunt contururi orientate). Astfel, poligoanele „din faţă" se află în semispațiul din faţa planului de partiționare (semispațiul spre care este orientată normala la planul poligonului), iar poligoanele „din spate" sunt conținute în semispațiul spate. Poligoanele intersectate de planul de partiționare sunt divizate, rezultând alte poligoane care se introduc în lista poligoanelor „din faţă" sau „din spate" în funcţie de poziţionarea lor faţă de planul de partiționare. În arborele BSP, poligoanele „din faţă" se vor afla în subarborele „faţă" (sau „stânga") al nodului rădăcină iar cele „din spate" în subarborele „spate" (sau „dreapta") al nodului. Functia de construire a arborelui se apelează recursiv pentru cele două liste de poligoane, ieşirea din recursivitate având loc atunci când lista de poligoane de intrare este vidă. Fiecărui nod al arborelui îi este atașat un poligon al scenei.

Afişarea „din spate în faţă”

Afişarea scenei reprezentate prin arbore BSP este, de regula, de tip „din spate în faţă”, ţinând cont de poziţia observatorului. Funcţia de afişare este apelată iniţial pentru nodul rădăcină apoi, recursiv, pentru toate celelalte noduri, până la nodurile frunză. Astfel, dacă observatorul se află în semispaţiul „spate” al planului (poligonului) nodului curent şi planul intersectează volumul vizual, atunci se vor afişa mai întâi poligoanele din subarborele „faţă”, apoi poligonul din nod şi la sfârşit poligoanele din subarborele „spate”. Dacă planul nodului curent nu intersectează volumul vizual, atunci subarborele „faţă” al nodului nu intersectează volumul vizual (este invizibil) şi nu este procesat, fiind eliminat complet din banda grafică, la fel ca poligonul din nodul curent. Dacă observatorul se află în semispaţiul „faţă” al nodului curent, ordinea de afişare este inversă.

Afişarea „din faţă în spate”

Afişarea „din spate în faţă” asigură redarea corectă a scenei 3D dar numărul de suprascrieri ale pixelilor în procesul de redare poate fi semnificativ. Din această cauză algoritmul poate sa nu fie suficient de rapid pentru redarea în timp real.

Arborele BSP permite afişarea scenei „din faţă în spate”, adică începând cu poligoanele cele mai apropiate de observator. Odată scris un pixel, el nu mai trebuie rescris în procesul de rasterizare a altor primitive, datorită corectitudinii sortării. În acest scop se folosesc diferite metode care folosesc măşti la nivel de pixel.

2.3.3 TEXTURARE

Utilizarea modelelor de iluminare pentru redarea suprafeţelor obiectelor 3D nu este suficientă pentru obţinerea de imagini realiste. Suprafeţele reale nu sunt perfect netede şi lucioase, efect produs prin iluminare şi colorare. Ele au adesea detalii care, folosind numai iluminarea, pot fi redate numai prin rafinarea modelului lor geometric (creşterea numărului de poligoane din reprezentarea lor). Astfel de detalii pot fi : neregularităţi(rugozităţi), pete, zgârieturi, etc. Problema redării detaliilor este rezolvată prin utilizarea texturilor, nefiind necesară creşterea complexităţii modelului geometric.

În utilizarea normală, termenul de “textură” se referă la calităţile suprafeţei unui obiect, din punctul de vedere al rugozităţii acesteia. În sinteza imaginilor pe

calculator însă, o textură reprezintă o colecţie de culori pentru pixeli, memorată în general sub forma unei imagini, utilizată pentru a simula din punct de vedere vizual textura unui obiect.

Texturile utilizate sunt, în general, imagini bidimensionale care codifică diverse informaţii despre suprafaţa corespunzătoare. În cazul cel mai simplu, acestea sunt utilizate pentru a memora culoarea suprafeţei în diverse puncte. De exemplu, o imagine care redă textura lemnului poate fi utilizată pentru a simula suprafețele obiectelor cu textură lemnoasă. În acelaşi mod pot fi simulate extrem de multe obiecte, cum ar fi, straturile de rocă din stânci, marmura de pe podele, etc.

Prin utilizarea mai multor texturi în paralel pentru redarea aceleiaşi suprafețe, fiecare în alt scop, se pot obține efecte mult mai complexe. De exemplu, este posibilă modificarea coeficientului de reflexie a luminii pe suprafaţa unui obiect care este parţial mat, parţial reflectant. Sau, este posibilă alterarea normalelor la suprafaţă pentru a simula obiectele cu suprafeţe rugoase fără a fi necesară adaugarea detaliilor prin poligoane suplimentare.

Cele mai multe texturi sunt reprezentate de fapt de matrici care conţin în celule culori, codificate RGB (Red, Green, Blue). Fiecare astfel de valoare poartă numele de element al texturii, sau **texel.** Texelii sunt adresati într-un spatiu parametric bi-dimensional, (u,v), cu 0<= u,v <= 1, numit **spaţiul textura.**

Atunci când este necesară aplicarea unei texturi pe un obiect din spaţiul 3D, adresele texelilor trebuie transformate în spaţiul obiectului, după care pot fi transformate în coordonatele ecranului. În mod uzual însă, această mapare se efectuează invers, adică pentru fiecare pixel de pe ecran este calculată o coordonata textura, (u,v), care se foloseşte la calculul unei culori din matricea textură. Metoda se numeste **mapare inversă.** Deoarece o pereche (u,v) nu poate selecta în mod unic un element al matricei textura, culoarea extrasă din textură se calculează printr-o **metoda de filtrare a texturii**. Filtrarea este în general realizata printr-o mediere biliniară între texeli vecini.

Maparea inversă presupune stabilirea unei corespondenţe între punctele suprafeței 3D texturate şi spaţiul parametric al texturii. Această corespondenţă este definită matematic numai pentru anumite tipuri de suprafețe, care au o reprezentare parametrică: sfera, cilindrul, planul, suprafețele bicubice. Funcțiile matematice folosite în acest scop se numesc **funcţii de mapare standard**. Orice suprafață 3D este descompusa de sistemul grafic, în vederea redării sale, într-o plasă poligonală. Astfel, texturarea se aplică primitivelor geometrice în care a fost descompusă

suprafaţa, triunghiuri sau poligoane, la momentul rasterizării lor. Problema mapării inverse apare deci pentru fragmentele acestor primitive.

Aplicarea texturilor pe suprafeţele 3D este aproximata astfel:

- se calculează o adresă textură, (u,v), pentru fiecare vârf al reţelei poligonale care aproximează suprafaţa; pentru aceasta se aplica ***maparea în doi pasi ;***
- pentru fiecare fragment al unui poligon, se calculează o adresa textura, (uf, vf), prin interpolarea adreselor textură asociate vârfurilor sale. Pentru rezultate corecte este necesară efectuarea unei interpolări perspectivă. interpolarea liniară este mai rapidă dar poate produce defecte ;
- în cazul utilizării mai multor texturi la redarea unui poligon, fiecărui vârf îi sunt asociate un acelaşi număr de perechi de coordonate textură (ui,vi), 1<= i <= n, pentru accesarea celor n texturi; n este în general limitat la 8.

Utilizarea coordonatelor textură

Cu toate ca spaţiul parametric al texturii este adresat prin valori (u,v) cuprinse între 0 şi 1, valorile (u,v) asociate vârfurilor pot fi în afara acestui domeniu. În procesul de rasterizare ele sunt interpolate pe suprafaţa primitivei, rezultând coordonate (u,v) asociate fragmentelor. Acestea trebuie transformate în valori 0<= u',v'<= 1.

Cele două moduri standard de transformare a coordonatelor textură în domeniul [0,1] sunt denumite "clamping" şi "wrapping (sau "repeating"). Ele permit utilizarea eficientă a texturilor şi obţinerea de efecte interesante.

Filtrarea texturii

O textură este definită pe o latice discretă de puncte în timp ce coordonatele textură sunt valori reale. În general, o pereche (u,v) nu corespunde unui texel al texturii, ci selecteaza o pereche (i', j'), unde i' şi j' sunt numere reale . Metoda de calcul a culorii dintr-o textură, corespunzătoare unei perechi (u,v), se numeste "filtrarea texturii".

Sunt două moduri standard de filtrare:

- Se consideră culoarea texelului cel mai apropiat în spaţiul laticei (texelul cu indicii (i, j) = ([i' + 0.5], [j'+0.5]), unde [x] reprezintă cel mai mare întreg mai mic sau egal cu x.
- Filtrarea liniara : se efectuează o interpolare biliniară între culorile celor mai apropiaţi texeli în spaţiul laticei.

Fiecare dintre cele două metode de filtrare are avantaje şi dezavantaje. Primul tip de filtru este mai rapid dar poate produce defecte în imagine, în funcţie de textură.

Dacă se preferă calitatea, se va alege filtrarea liniară, pe seama scăderii vitezei de generare a imaginii. Un efect secundar al filtrarii liniare este de începețoșare a liniilor.

Transparenţa şi opacitate

Imaginea textură poate avea o componentă adiţională, numită "canalul alfa", prin care se controlează transparenţa sau opacitatea texturii aplicate. Fiecare texel este în acest caz reprezentat prin 4 componente: (R,G,B,A), unde A reprezintă opacitatea. Dacă A = 1, atunci texelul este complet opac. Dacă A=0, texelul este complet transparent. Pentru valori ale lui A cuprinse între zero şi unu, culoarea texelului modulează culoarea fragmentului, determinată pe baza geometriei.

Interfețele de programare OpenGL şi Direct3D permit specificarea modului de combinare între culoarea unui fragment, determinată pe baza geometriei primitivei, şi culoarea rezultată din filtrarea texturii (texturilor), folosind operatori care se aplică culorilor (RGB) şi operatori care se aplică opacităților (A).

2.3.3.1 Environment mapping (maparea mediului înconjurător pe suprafețe)

Redarea mediului înconjurător pe suprafețele obiectelor se poate aproxima printr-o metodă specială de aplicare a texturilor, cunoscută sub numele de "Environment mapping". Mediul înconjurător este codificat într-o imagine, numită "harta mediului înconjurător"(environment map). Aceasta imagine se aplică sub forma unei texturi, pe suprafața unei sfere sau a unui cub, în centrul căreia / căruia se află obiectul. Dacă obiectul este reflectant (suprafața sa reflectă specular lumina incidentă) atunci harta textură este reflectată de suprafața obiectului. Dacă obiectul este transparent, atunci harta textură este redată pe suprafața obiectului prin fenomenul de refracție. În ambele cazuri, imaginea redată pe suprafața obiectului este dependentă de poziția observatorului, raza reflectată/refractată fiind calculată în funcţie de poziția observatorului şi normala la suprafață.

2.3.3.2 Simularea detaliilor fizice ale suprafețelor

Dupa cum am arătat mai înainte, calculul reflexiei luminii într-un punct al unei suprafețe utilizează normala la suprafața.

În cazul unei suprafețe netede, normala nu variază brusc în punctele sale vecine. Să ne imaginam însa un perete alcatuit din cărămizi, sau suprafața unei stânci. În astfel de cazuri, exista variații bruşte ale normalei în puncte vecine ale suprafeţei. Pentru redarea exactă a unor astfel de suprafețe, folosind numai modelul de iluminare, este

necesar ca suprafaţa sa fie modelată geometric prin foarte multe poligoane. Astfel, fiecare cărămidă şi fiecare spaţiu dintre cărămizi ar trebui modelate prin poligoane. Modelul geometric devine şi mai complicat dacă vrem sa redăm o suprafaţă rugoasă, cum ar fi aceea a unei pietre.

2.3.3.2.1 Bump mapping

În loc de a modela geometric detaliile fizice ale unei suprafeţe, putem să le simulăm, înlocuind, în calculul luminii reflectate, normala la suprafaţă fără detalii (de exemplu, peretele neted, fără cărămizi) cu o normală care ar corespunde suprafeţei cu detalii (peretele cu cărămizi). Ideea a apartinut lui Blinn([BLI78]), care a şi introdus metoda cunoscută sub numele de "Bump mapping". El a modelat matematic suprafaţa cu detalii fizice folosind o funcţie de perturbare definită în fiecare punct al suprafeţei fără detalii.

Astfel, dacă suprafaţa este definită parametric printr-o ecuaţie Q(u,v), atunci normala la suprafaţa într-un punct (u,v) este dată de produsul vectorial dintre vectorii tangenţi la suprafaţă în punctul (u,v):

$$N = Q^u \times Q^v$$

unde Q^u şi Q^v sunt derivatele parţiale în raport cu u şi v. Suprafaţa perturbată este definită astfel:

$$Q'(u,v) = Q(u,v) + P(u,v)\frac{\overline{N}(u,v)}{|\overline{N}|}$$

unde P(u,v) este funcţia de perturbare. Normala la suprafaţa perturbată este:

$$N' = Q'^{u} \times Q'^{v} = N + D$$

Funcţia de perturbare poate fi orice funcţie derivabilă, f(u,v). Pentru texturile fără o definiţie matematică, Blinn a sugerat utilizarea unui tablou bi-dimensional, indexat prin parametrii u şi v. Valorile intermediare se obţin prin interpolare bi-liniară iar derivatele parţiale prin diferenţe finite.

Metoda se integrează bine într-un model de iluminare la nivel de fragment care foloseşte normala obţinută prin interpolarea normalelor din vârfurile primitivei geometrice.

2.3.3.2.2 Hărţi de normale (Normal maps)

Ideea lui Blinn a fost extinsă, pe de o parte pentru a se uşura definirea detaliilor fizice ale suprafeţelor, pe de alta parte pentru obţinerea vitezei necesare sintezei imaginilor în timp real. Astfel, în locul funcţiei de perturbare a normalei se foloseşte o textură din care se iau normalele utilizate în calculul de iluminare la nivel de pixel. O astfel de textura se numeşte «harta de normale».

Un vector normală este un vector 3D orientat de la suprafaţa înspre exterior. În cazul în care iluminarea este realizată în spaţiul obiect, normalele stocate în textură corespunzătoare trebuie să fie reprezentate tot în spaţiul obiect. Avantajul evident al acestei abordări este ca în momentul când sunt efectuate calculele de iluminare la nivel de pixel, normala poate fi citita din textura şi utilizată direct, eventual după o renormalizare.

Această tehnică este folosită în mod uzual pentru simplificarea geometriei suprafeţelor astfel încât sa fie posibilă redarea lor în timp real. Astfel, un model poligonal extrem de complex poate fi înlocuit cu un model compus dintr-un număr rezonabil de poligoane şi o hartă textură de normale care să îi adauge acestuia detaliile care s-au pierdut în urma simplificării.

Simplificarea obiectelor poate fi realizată în orice aplicaţie de modelare 3D, utilizând un instrument corespunzător. În general, aceasta reducere se face prin concatenarea poligoanelor vecine care au unghiul dintre planele lor sub un anumit prag, şi prin concatenarea punctelor aflate la distanţe relative sub un alt prag specificat. Generarea texturii care memorează normalele se face pe baza diferenţelor între obiectul redus şi obiectul original, utilizând trasarea de raze pentru a deduce normalele peste obiectul redus.

2.3.4 SUNET

Pe măsură ce posibilităţile dispozitivelor de redare a sunetelor pe calculator şi API-urile acestora s-au îmbunătăţit, sunetul 3D a început sa aibă un rol din ce în ce mai important în simularea lumilor virtuale.

Crearea unei experienţe cât mai imersive pentru utilizator în lumea virtuală nu înseamnă numai o reprezentare vizuală cât mai reală ci trebuie ţinut cont şi de elementele de sunet ale mediului virtual.

Implementate corect, sunetele ambientale, muzica şi alte efecte audio pot duce simularea virtuală la un cu totul alt nivel.

2.3.4.1 Efecte audio 3D

Deşi poziţionarea în spaţiul 3D este un factor important pentru modul în care utilizatorul percepe sunetele, aceasta singură nu este suficientă. Elementele mediului înconjurător pot modifica sunetul emis de o sursă înainte de a fi perceput de un personaj virtual existent în spaţiul 3D. Principalele efecte acustice produse de interacţiunea sunetului cu mediul sunt:

- reverberaţia
- obstrucţia
- ocluzia

Reverberaţia este fenomenul reflectării multiple a sunetului atunci când acesta întâlneşte o suprafaţă.

Un factor de care trebuie ţinut cont în simularea reverberaţiei este materialul suprafeţei de care se "loveşte" unda sonoră, în funcţie de acesta absorbţia va fi mai mare sau mai mică rezultând o reverberaţie mai slabă sau mai puternică.

De asemenea, reverberaţia este influenţată şi de dimensiunea şi forma incintei în care se propagă sunetul. Astfel, o incintă mai mare va face ca timpii de reverberaţie sa fie mai lungi iar dacă avem suprafeţe neregulate, cu multe muchii, acestea vor determina creşterea consistenţei reverberaţiei.

Reverberaţia ca fenomen acustic ne înconjoară în permanenţa în realitate şi oferă foarte multe informaţii despre mediul înconjurător. De aceea, prezenţa acestui efect în simularea lumii virtuale contribuie semnificativ la realismul acesteia.

De asemenea, reverberaţia intervine şi în cazul obstrucţiei acustice.

Obstrucţia este fenomenul ce se produce atunci când între utilizator şi sursa care emite sunetul se află poziţionat direct un obstacol. Dacă se doreşte ca simularea să fie cât mai apropiată de realitate, trebuie implementată şi obstrucţia sunetelor. În acest caz, undele sonore nu mai ajung direct la ascultător, dar totuşi el le poate percepe într-o formă modificată, datorita reverberaţiei.

Fenomenul de reverberaţie, care permite reflectarea sunetului, face posibil ca, prin reflexii multiple, acesta sa ocolească obstacolul şi, deşi a pierdut din intensitate, să ajungă la utilizator.

Pe lângă aceste două fenomene, într-un spatiu real poate sa apară şi fenomenul de ocluzie acustică. Sa luam situaţia de mai sus, în care sursa de sunet este despărțită de utilizator printr-un obstacol, numai că de aceasta dată obstacolul izolează perfect sunetul de utilizator, aceştia aflându-se în două incinte diferite, neexistând nici o posibilitate ca sunetul sa "ocolească" obstacolul (ca exemplu, putem considera două camere separate de un zid).

Astfel, dacă obstacolul este suficient de gros şi izolant (de exemplu, piatra) sunetul nu va ajunge deloc la utilizator. Pe de altă parte, dacă obstacolul este alcătuit din lemn şi este subţire sunetul va trece prin acesta atenuat, în funcţie de grosimea obstacolului.

Fenomenul de atenuare a sunetului la trecerea printr-o suprafața de separaţie a două incinte sau a două medii diferite se numeşte ocluzie.

Toate aceste fenomene sunt elemente importante în modelarea cât mai reala a lumilor virtuale. Pentru a obține aceste efecte, au fost dezvoltate diferite API-uri pentru efecte audio, cum ar fi Microsoft DirectSound şi DirectSound3D , Aureal A3D şi Creative Labs EAX.

Mai nou, Creative Labs au dezvoltat OpenAL, care este echivalentul pentru implementarea sunetului al OpenGL-ului.

2.3.4.2 Formate Audio

Exista numeroase formate pentru fişierele ce conțin informaţiile audio. În continuare sune enumerate cateva dintre cele mai uzuale formate folosite în prezent.

WAV

Acesta este probabil formatul cel mai documentat folosit în prezent. Formatul .WAV este comparabil cu formatul .BMP pentru imagini în sensul ca amândouă au o calitate foarte bună a informaţiei pe care o conțin dar acest lucru este direct proporţional cu dimensiunea pe care o ocupă pe disc.

Deoarece aceste dimensiuni pot ajunge destul de mari este recomandat ca acest format sa fie folosit pentru sunete şi efecte audio de scurtă durată.

MP3

Este unul dintre cele mai populare formate audio. Acest format beneficiază de o tehnologie de compresie foarte bună cu pierderi minime de calitate. Un alt avantaj al acestui format ar fi ca este bine documentat, cu multe exemple online.

Acest format este recomandat a fi folosit pentru dialog şi muzica ambientală.

OGG

Acest format a început recent sa capteze din ce în ce mai multă atenţie. Deşi încă nu este la fel de popular ca formatul MP3 , recuperează rapid terenul pierdut. Faţă de MP3 are avantajul ca se obţine pentru sunete o compresie şi o calitate mai bună.

2.3.5 ANIMAŢIA PERSONAJELOR

În cadrul acestui sub-capitol sunt enumerate şi prezentate pe scurt principalele metode de modelare şi animare a personajelor (“characters”).

2.3.5.1 Modelarea personajelor

Metodele de modelare a personajelor au devenit din ce în ce mai complexe şi reuşesc sa creeze modele foarte apropiate de personajele reale, dar nu întotdeauna modelul cel mai realist este şi cel mai bun în orice aplicaţie. Alegerea uneia dintre metodele de modelare depinde de necesităţile aplicaţiei.

2.3.5.1.1 Plasa poligonală

Acesta a fost primul mod de reprezentare a personajelor, direct inspirat din tehnicile de animaţie folosite în plan. Personajul este reprezentat printr-o plasă poligonală 3D. Animaţia personajului constă în deformarea plasei, fiecărui cadru imagine corespunzându-i o “poză” (ipostază) a personajului. Numărul de vârfuri ale plasei şi conectivitatea lor nu se modifică de la o poză la alta, ci doar poziţiile vârfurilor.

2.3.5.1.2 Reprezentari bazate pe schelet

Scheletul unui personaj 3D animat este similar scheletului uman. Acesta constă din oase şi articulaţii şi poate fi folosit ca un mecanism de control pentru deformarea straturilor de deasupra scheletului: muşchi şi piele. Scheletul este definit ca o ierarhie de oase sau articulaţii. El este reprezentat printr-un arbore ale cărui noduri corespund oaselor sau articulaţiilor.

Unele aplicaţii care permit definirea de personaje reprezintă scheletul prin oase, altele prin articulaţii. Fiecare os/articulaţie are asociată o matrice de transformare (rotaţie şi translaţie) care defineşte mişcarea articulaţiei/osului. Transformarea totală a unui nod al ierarhiei schelet este produsul dintre transformarea nodului părinte şi propria transformare. Animaţia scheletului constă în aplicarea de transformări, care se modifică în timp, asupra nodurilor.

Deoarece modelarea unui schelet real este destul de dificilă, în practică se foloseşte un schelet simplificat, cu câteva zeci de oase şi articulaţii, cu unul, două sau trei grade de libertate, cărora uneori le sunt ignorate limitele.

Un avantaj al reprezentării prin schelet este faptul ca permite animatorului să controleze doar acele caracteristici ale modelului care se mişca independent. Animaţia este mult mai simplu de definit, rezumându-se la mişcarea oaselor, în loc de mişcarea fiecărui vârf (în cazul reprezentării prin plasă poligonală).

2.3.5.2 Animaţia personajelor

Există mai multe posibilităţi de a realiza animaţia personajelor iar în majoritatea cazurilor alegerea unei metode de animaţie este puternic dependentă de necesităţile aplicaţiei ţintă.

2.3.5.2.1 Interpolarea între cadre cheie

Metoda constă în realizarea de către animatori a unor schiţe pentru personajul ce se doreşte a fi animat. Schiţele reprezintă imagini ale personajului la anumite momente importante. Ele se numesc “cadre cheie”. Apoi, se completează animaţia cu cadre intermediare între cadrele cheie succesive.

Sunt utilizate în acest scop diferite metode de interpolare. Pentru interpolarea poziţiilor se folosesc de regulă interpolarea liniara şi cea pătratică. Pentru interpolarea orientării (între rotaţiile corespunzătoare din două cadre cheie) se foloseşte interpolarea sferică, folosind reprezentarea rotaţiilor prin quaternioni (Slerp - interpolarea sferică liniară şi Squad - interpolarea sferică cubică).

Interpolarea între cadre cheie a fost folosită la inceput pentru animaţia reprezentării prin plasă poligonală, caz în care se aplică vârfurilor plasei. Ulterior ea a fost asociată şi altor tehnici de animaţie, de exemplu, celor bazate pe animaţia scheletului, caz în care cadrele cheie reprezintă ipostaze ale scheletului iar interpolarea se foloseşte pentru generarea matricilor de transformare ale oaselor.

2.3.5.2.2 Interpolarea formei

Interpolarea formei (Shape Interpolation) este o tehnică folosită în special în animaţia facială. Se defineşte un set de forme "cheie". Coordonatele punctelor de pe suprafaţa formei sunt calculate ca o combinaţie liniară a coordonatelor punctelor din formele cheie, pe baza unor ponderi asociate punctelor:

$$S = \sum_{k} w_k \bullet S_k$$

O variaţie a acestei tehnici foloseşte o formă ca bază, celelalte fiind specificate doar ca diferenţe faţă de această bază:

$$S = S_0 + \sum_{k} w_k \bullet (S_k - S_0)$$

Ponderile cu care se combină aceste date variază în timp.

Metoda are o aplicabilitate limitată, deoarece se utilizează pentru combinare forme rigide.

2.3.5.2.3 Forward Kinematics

Este o metodă folosită pentru specificarea animaţiei unei structuri articulate, cum ar fi scheletul unui personaj. Metoda presupune specificarea explicită de către animator a mişcărilor fiecărei părţi a structurii articulate. Animatorul trebuie să definească mişcările obiectelor rigide din structură (oasele) începând cu nodul rădăcină al ierarhiei, coborând în ierarhie până la fiecare terminaţie a structurii.

2.3.5.2.4 Inverse Kinematics

În această metodă animatorul trebuie să specifice numai poziţia (poziţiile) în care trebuie să ajungă terminaţia (sau terminaţiile) structurii. Prin cinematica inversă se deduc mişcările tuturor părţilor structurii articulate astfel încât să se obţină "poza" dorită.

Problema este că pe măsura ce structurile devin din ce în ce mai complexe, soluţia cinematicii inverse devine din ce în ce mai greu de găsit. Există o rezolvare analitică a poblemei doar pentru lanţuri foarte scurte. În cazul unei ierarhii precum cea a scheletului personajelor umanoide, soluţia nu poate fi găsită decât folosind tehnici iterative. Din acest motiv metoda este lentă.

2.3.5.2.5 Inverse Kinetics

Înrudită cu cinematica inversă, permite ierarhizarea scopurilor pe care trebuie să le atingă personajul de animat şi specificarea unui criteriu în funcţie de care trebuie optimizată soluţia. Criteriul este exprimat ca o funcţie de starea personajului. În plus faţă de cinematica inversă, această tehnică ţine cont de distribuţia masei, obţinând astfel rezultate mai bune, mai repede. Deşi are aceeaşi complexitate, folosind metode asemănătoare, cinetica inversă converge mai repede în cazul încercării de a poziţiona centrul de masă.

Influenţa fiecărei articulaţii asupra posturii căutate este estimată în funcţie de masa pe care o suportă în starea curenta a sistemului. Rotaţia oaselor în articulaţii, este pusă în legătură cu translaţia unui end-effector la fel ca în cazul cinematicii inverse.

2.3.5.2.6 Forward Dynamics

Metodele de cinematica ignoră o mare parte din influenţele mediului. Nu tin cont de masa obiectului de animat sau de forţele care-i sunt aplicate.

Această metodă calculează poziţiile folosind distribuţia masei personajului şi forţele care se aplică. În cazul jocurilor se pot folosi legile mecanicii Newtoniene. Ştiind forţele, putem calcula acceleraţia, care apoi este integrată şi se calculează poziţia. Deşi robotica a dezvoltat algoritmi eficienţi pentru structuri cu multe grade de libertate, metoda este greu de folosit. Forţele nu sunt nişte parametri sugestivi pentru un animator. Controlul unui personaj astfel simulat este foarte dificil.

2.3.5.2.7 Inverse Dynamics

În aceasta metodă, pornind de la poziţia ce se doreşte a fi obţinută se încearcă aflarea forţelor care trebuie aplicate.

Rezultatele obţinute sunt in general aproximative. Dinamica inversă face anumite presupuneri: că nu există forţe de frecare la articulaţii, că oasele sunt rigide şi masa lor e concentrată în centrul de masă şi nu există frecare cu aerul. Are aplicaţii în biomecanică. Încearcă să afle forţele produse de muşchi în timpul mişcării şi în acest scop presupune ca muşchii nu-şi anulează reciproc acţiunile.

2.3.5.2.8 Motion Capture (captura mişcării)

"Captura mişcării" are drept scop înregistrarea mişcărilor unui personaj real, cu ajutorul unor echipamente, pentru a fi apoi folosită în animaţia personajelor

sintetizate. Informația capturată variază de la poziția diverselor părţi ale corpului până la deformări ale feţei sau muşchilor.

Există mai multe tipuri de echipamente de captură a mişcării: magnetice, optice, electro-mecanice, opto-mecanice ce folosesc pentru înregistrarea mişcării senzori şi markeri ce sunt plasaţi pe corp.

Animaţiile produse prin captura mişcării nu sunt decât varianta moderna a animației bazată pe cadre cheie (cadrele cheie nu sunt desenate ci inregistrate) şi ele pot servi ca date de foarte bună calitate şi care pot fi obtinute rapid, pentru alte metode.

2.3.5.2.9 Motion Warping (deformarea mişcării)

Mişcarea poate fi văzută ca un set de curbe care descriu valorile parametrilor modelului animat în funcție de timp. Motion Warping pleacă de la convingerea că se pot modifica anumite traiectorii, pentru a respecta unele constrângeri, fără a afecta credibilitatea animației. Principala aplicație a fost modificarea datelor provenite din captura mişcării pentru a putea fi folosite în situaţii specifice. Metoda păstrează intr-adevar aspectul natural al mişcării, dar curbele sunt dificil de înţeles şi editat.

2.3.5.2.10 Motion Blending (combinarea mişcărilor)

«Motion Blending» a pornit de la ideea ca se pot combina mai multe animaţii pentru a obține o animație diferită. De exemplu, se poate combina o animație care reprezintă un om mergând furios, cu o animație a aceluiaşi om care aleargă, pentru a obține un om furios care aleargă.

Se pot aplica anumite constrângeri fizice. De exemplu, înălţimea la care ajunge mâna în cazul unei lovituri.

Metodele de combinare a mişcărilor par a fi promiţătoare pentru sistemele de realitate virtuală. Deşi nu respectă în totalitate legile fizicii, ele produc în general mişcări care arată natural, iar unele dintre ele pot fi controlate cu parametri de nivel înalt. Astfel de parametrii pot fi folosiţi pentru modelarea relaţiei agenţilor cu mediul virtual.

2.3.6 SISTEME DE PARTICULE

Termenul de **sistem de particule** se referă la o tehnică folosită în grafica pe calculator, prin care se pot simula anumite fenomene sau obiecte neregulate cum sunt: focul, exploziile, fumul, apa, scânteile, frunzele, nori, ceaţă, zăpada, praful sau efecte vizuale abstracte cum ar fi o urmă strălucitoare, etc. Astfel de obiecte sau fenomene sunt greu de reprezentat şi redat în imagini prin tehnici convenţionale.

De regulă, poziţionarea sistemului de particule în spaţiul 3D şi dinamica acestuia sunt controlate de către un **emiţător**.

Emiţătorului îi este ataşat un set de parametri ce caracterizează comportamentul particulelor: rata de generare a acestora (câte particule sunt generate într-o unitate de timp) , vectorul iniţial al vitezei particulelor (în ce direcţie se vor mişca particulele când acestea vor fi emise) , durata de viaţă a particulelor, variaţia culorii particulelor în timpul vieţii lor , etc.

Atributele unei particule individuale sunt : poziția, viteza, mărimea, culoarea, transparenţa, forma, durata de viaţă.

În mod normal, toți acesti parametrii nu au valori regulate, acestea putând varia de la o particulă la alta, fiind aleatorii între anumite limite.

O comportare caracteristică pentru un sistem de particule poate fi împărțită în două etape:

- etapa de actualizare a stării sistemului
- etapa de redare a sistemului (crearea imaginii curente a sistemului)

În timpul etapei de actualizare sunt generate noi particule şi se actualizează starea celor existente în sistem. Fiecare nouă particulă este creată într-o poziție 3D specifică, bazată pe poziția emiţătorului şi pe zona de generare specificată, iar atributele ei (viteza, culoarea) sunt inițializate în funcţie de parametrii emiţătorului.

Particulele existente în sistem a căror durata de viaţă a expirat sunt eliminate din simulare.

Starea celorlalte este modificată pe baza unui mecanism de simulare, care poate să fie foarte simplu, constând în calculul noii poziţii folosind viteza de deplasare, sau unul complicat în care se fac calcule ale traiectoriei particulei corecte din punct de vedere fizic, care iau în considerare acţiunea unor forţe externe (cum ar fi de exemplu gravitaţia).

În mod normal, tot în această etapă ar trebui verificată detecţia coliziunilor particulelor cu obiectele 3D din sistem (sau din spaţiul virtual), astfel încât particulele sa ricoşeze la întâlnirea acestora. Totuşi verificarea detecţiei coliziunilor în cazul sistemelor de particule este rar utilizată deoarece este foarte costisitoare din punct de vedere computaţional.

Un sistem de particule are propriile reguli, care sunt aplicate fiecărei particule în parte. Uzual aceste reguli conţin şi o metoda de interpolare a valorilor atributelor particulei de-a lungului vieţii sale. De exemplu, multe sisteme fac ca particulele să dispară uşor prin interpolarea valorii alfa (opacitatea) de-a lungul vieţii particulei ("fading out").

Dupa ce etapa de actualizare s-a incheiat, fiecare particulă este redată în imagine, de obicei sub forma unui billboard. Aceasta nu este însă singura modalitate de a reprezenta o particulă : se pot folosi reprezentări de la cea mai simplă, un singur punct, până la reprezentări de suprafeţe (mesh-uri).

2.3.7 REDĂRI SPECIALE

2.3.7.1 Redarea vegetaţiei

Absenta redării cu mare acurateţe a vegetaţiei în aplicaţiile de timp real, cum ar fi şi simulările virtuale, poate fi atribuită în mod direct cantităţii imense de informaţie geometrică necesară pentru a modela corespunzător vegetaţia.

Iarba şi alte forme de vegetaţie naturală sunt frecvent întâlnite în simulările spaţiilor exterioare. Deşi, de exemplu, forma unui singur fir de iarba nu este complexă şi aceasta poate fi reprezentată cu ajutorul câtorva poligoane, firele de iarbă nu apar într-un număr mic, acestea întinzându-se de obicei pe suprafeţe foarte mari. Astfel, reprezentarea şi iluminarea vegetaţiei devin greu de realizat din punct de vedere practic.

Mai mult, vegetaţia poate avea o variaţie structurală foarte mare ceea ce duce la apariţia de proprietăţi caracteristice de reflexie a luminii.

Redarea vegetaţiei prin modelarea sa geometrică este posibila dar nu este o metodă viabilă în practică datorită cantităţii de informaţie foarte mare care necesită putere computaţională ridicată.

O soluţie constă în simularea vegetaţiei folosind texturi.

Utilizarea de texturi statice produce rezultate destul de apropiate de realitate, dar exista două probleme importante în utilizarea acestei soluţii, care scad din realismul simulării :

- texturile folosite fiind statice, nu se modifică în funcţie de poziţia camerei şi de iluminare; aceasta înseamnă că, de exemplu, un petic de iarba va arăta la fel din orice poziţie este privit, umbrele vor rămâne fixe, spoturile speculare nu se vor modifica, etc; astfel, natura statică a texturii scade foarte mult realismul reprezentării grafice a vegetaţiei;
- texturile fiind imagini bidimensionale, sunt mapate pe un poligon plat astfel încât realismul 3D al mediului scade simţitor; nu se pot obţine siluete şi efectul interacţiunii vegetaţiei cu alte obiecte din mediul înconjurător dispare.

O altă metodă, care reprezintă o evoluţie a folosirii texturilor statice, este folosirea de billboard-uri. Transparenţa texturii aplicate pe suprafaţa billboard-ului face posibilă reprezentarea de obiecte complexe, cum ar fi plantele şi copacii. Metoda însa nu permite realizarea unei iluminări realiste.

Alte abordări se bazează pe utilizarea de texturi volumetrice ([DEN04]).

Totodată, evoluţia hardware-ului grafic a creat o nouă provocare în privinţa redării vegetaţiei folosind modele geometrice şi iluminare realistă([CDN04]).

În practica se foloseşte o schemă hibridă, bazată pe un model cu trei nivele:

- pentru vegetaţia care se află aproape de poziţia camerei se foloseşte o reprezentare geometrică detaliată;
- pentru vegetaţia aflată la distanţă medie se folosesc texturi volumetrice;
- pentru vegetaţia aflata la distanţă mare, pentru care oricum detaliile nu s-ar putea distinge, se foloseşte o schemă de texturare statică.

Sinteza imaginilor realiste a terenurilor acoperite cu vegetaţie este şi în prezent o provocare şi o problema importantă în grafica pe calculator.

2.3.7.2 Redarea apei

Simularea apei constituie un subiect asupra căruia se fac cercetări de un timp îndelungat. Simularea realistă a fluidelor s-a dovedit a fi foarte dificilă datorită complexităţii inerente a proceselor fizice ce intervin în comportarea acestora.

Cele mai realiste soluţii folosesc un sistem de tip ray-tracing şi de ecuaţii diferenţiale complexe pentru a aproxima comportamentul optic şi dinamica

mişcării apei. Din nefericire aceste abordări de o acurateţe fizică mare pot necesita multe ore de procesare numai pentru un singur cadru.

O dată cu dezvoltarea continuă a hardware-ului programabil pentru grafica 3D, mulţi algoritmi care înainte erau folosiţi doar pentru procesare off-line sunt acum viabili pentru folosirea în timp real.

Ca de obicei, atunci când se implementează efecte complexe în contexte interactive, trebuie ajuns la un compromis între calitatea reprezentării şi performanţa computaţională necesară pentru a face această reprezentare. Astfel, foarte multe proprietăţi vizuale ale apei pot fi aproximate prin tehnici eficiente, care deşi nu sunt corecte din punct de vedere fizic par convingătoare pentru ochiul uman.

La fel ca majoritatea efectelor fizice, simularea apei este o combinaţie a doi paşi distincţi :

- **Generarea fizică** a apei, denumită şi modelul undei, simulează mişcarea apei sub influenţa unor forţe externe şi interne; acest model este cel care face ca apa să se mişte, să îşi schimbe forma şi să interacţioneze cu mediul înconjurător. Din punct de vedere matematic, sunt posibile mai multe abordări. Cele mai simple aplică o perturbare procedurală, cel mai adesea sub forma unei perturbări Perlin, simulând o mişcare ce pare aleatoare. Metodele care încearcă să reproducă mişcarea reală a apei folosesc ecuaţii diferenţiale, aproximarea legilor fizice ale hidrostaticii şi hidrodinamicii şi permit interacţiunea dintre utilizator şi apă.
- **Vizualizarea** apei pe baza datelor calculate la pasul anterior. Pentru un realism cât mai ridicat trebuie să se ţină iarăşi cont de legile fizicii : o suprafaţă de apă are interacţiuni complexe cu lumina din mediul înconjurător şi modelarea acestor proprietăţi optice este foarte importantă pentru obținerea unui efect vizual cât mai real.

Ambii paşi pot beneficia de puterea de calcul a GPU, dar mai ales al doilea. Simularea optica a suprafeței apei necesită operaţii complexe de calcul la nivel de pixel. Hardware-ul modern cu posibilităţi de programabilitate atât la nivel de vârf cât şi la nivel de pixel este potrivit pentru această sarcină.

2.4 TEHNOLOGII SERVER-SIDE

În proiectarea protocolului de comunicaţie dintre server şi utilizatori, într-o aplicaţie spaţiu virtual care se execută în cadrul unei reţele pe mai multe calculatoare, se poate opta pentru abordarea "peer-to-peer" sau abordarea "client/server".

2.4.1 ABORDAREA PEER-TO-PEER

În abordarea peer-to-peer, două sau mai multe calculatoare comunică între ele, fiecare rulând o copie a aplicației şi schimbând informaţii despre starea aplicației.

Aceasta abordare, deşi conduce la un trafic redus pe reţea, implică alte probleme cum ar fi, de exemplu, tratarea traficului pierdut. Aceasta este mult mai importantă în momentul în care ruleaza mai mult de o copie a aplicației, deoarece nu se mai ştie cine are datele corecte despre starea aplicației şi cine nu. Diferenţele între stările aplicației la diferiţi utilizatori pot crea probleme, deoarece toate copiile aplicației trebuie să se sincronizeze între ele.

În plus, fiecare instanţă a aplicației trebuie sa aştepte intrările de la celelalte instanţe înainte de a putea simula următorul cadru. Doom 3 [http://www.doom3.com/] a fost unul dintre jocurile care a folosit această abordare şi existau momente în care jocul "îngheța" în aşteptarea stării de la maşinile celorlalţi jucători.

2.4.2 ABORDAREA CLIENT/SERVER

În abordarea client/server, în sistem exista doi actori :

- maşina server
- clienţii

Maşina server este cea care rulează aplicaţia şi transmite către toți clienţii starea curenta a aplicației, pe baza căreia fiecare client isi actualizează imaginea spaţiului virtual. Clienţii afișează scena spaţiului virtual aşa cum le-a fost transmisă de către server şi redau sunetele pe care server-ul "le spune" să le redea.

Efectiv, clienţii se comporta ca nişte terminale care transmit intrările către server şi îl lasă pe acesta să se ocupe de aproape tot. În practică, acest mod de împărţire a

responsabilităţilor nu este un lucru chiar aşa de rau cum pare la prima vedere, ocazional server-ul putând sa ceară unui client să indeplinească şi alte funcționalități ce ar putea sa cadă în seama sa.

Pot exista foarte multe variații în abordarea client/server însă ideile de bază sunt cele prezentate în continuare.

2.4.2.1 Probleme ce trebuie tratate în realizarea proiectării sistemului client/server

În proiectarea unui protocol multi-utilizator, pot apărea următoarele probleme, de care trebuie să se ţină cont :

- tehnologia folosită : TCP sau UDP
- încărcarea pachetelor
- frecvenţa pachetelor
- ordinea pachetelor şi pachetele lipsă (în cazul UDP)
- clienţi care încearcă să păcălească sistemul

Tehnologia folosită: TCP sau UDP

UDP (User Datagram protocol) şi TCP (Transmission Control Protocol) sunt două protocoale ce funcţionează la nivelul Transport al modelului OSI.

Diferenţa dintre UDP şi TCP este aceea ca TCP garantează livrarea pachetelor la destinaţie în ordinea în care au fost transmise pe când UDP nu garantează livrarea lor şi cu atât mai puţin ordinea în care acestea sosesc.

Deşi la o prima vedere TCP pare sa fie superior fata de UDP, avantajele oferite de protocolul TCP au un cost: overhead-ul (timpul suplimentar consumat cu stabilirea conexiunii) şi latenţa.

Astfel, în protocolul TCP, expeditorul (ca sa fie sigur ca pachetele ajung la destinaţie) aşteaptă să primească un ACK (Acknowledgement) din partea destinatarului pentru fiecare pachet pe care îl trimite. Dacă nu primește nici un ACK atunci încetează să mai trimită pachete noi şi le retrimite pe cele presupuse pierdute, continuând sa facă acest lucru până când destinaţia răspunde.

Problema este întârzierea dintre momentul în care expeditorul încetează să mai trimită pachete noi şi momentul în care pachetul ajunge într-adevăr la destinaţie.

Această întârziere poate ajunge uneori la nivel de secunde, ceea ce nu ar fi un motiv de îngrijorare dacă s-ar descărca o pagina sau un fişier, însă în cazul unui

pachet dintr-un joc apar probleme, deoarece aceasta împiedică actualizarea stării jocului la destinaţie.

2.4.2.2 Încărcarea pachetelor

Trebuie acordată o atenţie foarte mare la selectarea datelor transmise, astfel încât să fie transmise numai acele date care sunt strict necesare. Cu cât este mai mare pachetul trimis, cu atât se încarcă mai mult transmisia ajungându-se astfel la latenţe mari dacă conexiunea folosită nu are o lăţime de bandă suficient de mare.

Trebuie ţinut cont şi de faptul că un pachet ce se poate trimite are o dimensiune maximă numită MTU (maximal transmission unit). De obicei, în cazul protocoalelor multiplayer se defineşte o dimensiune maximă a pachetului iar atunci când se depăşeşte această dimensiune, datele se împart în mai multe pachete, transmiţându-se mai întâi datele care trebuie sa ajungă cât mai repede.

O altă tehnică prin care se poate reduce încărcarea pachetelor este eliminarea mesajelor text. Dacă serverul trimite foarte multe mesaje/comenzi text prestabilite este logic ca acestea sa fie pre-încărcate la client şi acesta sa primească de la server doar identificatorul mesajului în loc de mesaj. Această tehnică poate reduce traficul de mesaje în mod considerabil.

Trimiterea doar a obiectelor/datelor care au relevanţă pentru scena vizualizată de utilizator este de asemenea indicată. Clientul nu are nevoie sa cunoască informaţii despre acţiuni şi evenimente care nu se află în raza sa vizuală sau auditivă. Oricum acestea nu vor fi redate sau auzite de client. Acest lucru ar putea ca în cadrul unei scene exterioare sau al unui simulator în spatiu sa nu fie tocmai indicat, însa şi aici se pot lua decizii de implementare care să compenseze, cum ar fi, de exemplu, implementarea nivelului de detaliu.

Nu este necesar ca la client sa fie transmise rezultatele execuţiei efectelor speciale, acest lucru consumând lăţimea de bandă atât a serverului cât şi a clientului. Este de ajuns, de exemplu, ca serverul sa spună “aici se intampla o explozie” şi clientul face restul afisand efectul corespunzător pe display.

O altă tehnică folosită este *compresia bazată pe modificare*. Ideea este să se transmită numai ceea ce s-a modificat între două cadre succesive. Se poate păstra o copie a ceea ce s-a transmis ultima oara, bineînţeles la nivel de obiect, şi pentru cadrul curent să se transmită numai modificările.

Ca ultimă idee, se pot implementa tehnici de compresie a datelor din interiorul pachetelor trimise pentru a micşora dimensiunea pachetelor, însă de cele mai multe ori timpul consumat atât pe server cât şi pe client pentru compresie/decompresie este mai mare decât trimiterea unui pachet necompresat.

2.4.2.3 Frecvenţa pachetelor

Deşi motorul grafic este în stare să redea scena la un număr mare de cadre pe secundă, dacă infrastructura reţelei nu poate sa susţină o rată de trimitere/sosire a pachetelor suficient de mare, acest lucru nu este un avantaj.

De exemplu, serverul de *Quake* functioneaza la 10 cadre pe secunda, transmiţând datele la această rată. Funcţionează la 10 cadre pe secundă din două motive:

- din cauza dimensiunii datelor pe care le procesează pentru fiecare client
- deoarece este o rată a pachetelor care poată fi susţinută uşor

2.4.2.4 Ordinea pachetelor şi pachetele lipsă

Modul de abordare este de a trata un simptom şi de a-l ignora pe celălalt. Dacă pachetele sunt numerotate, clientul va şti când a primit un pachet ce nu a sosit în ordinea corectă. Cea mai simplă soluţie este să îl ignore şi să il trateze ca pe un pachet pierdut.

Dacă la client se păstrează o copie a ultimului pachet primit de la server, se poate compara pachetul ce soseşte cu acesta pentru a vedea dacă un obiect a fost eliminat din scenă. Dacă acest lucru s-a întâmplat, se poate elimina şi de către client obiectul imediat sau se poate stoca într-un buffer pentru a mai verifica cu încă câteva pachete ce vor sosi. Avantajul acestei tehnici este că nu va fi necesar niciodată să se trimită de către server clientului comanda de “remove” deoarece lipsa unui obiect din pachet garantează eliminarea acestuia şi de la client. Chiar dacă au existat pachete lipsă, acest lucru nu contează deoarece până la urma clientul va primi un pachet şi va vedea ca obiectul nu mai exista !

Tot aici se poate discuta despre mecanismul de predicţie a acţiunilor clientului. În cazul în care apar pachete lipsă, atât din partea clientului cât şi a serverului, în timpul până la primirea unui pachet corect, clientul trebuie să execute o acţiune pentru a nu da impresia de “îngheţare”. Se poate prezice până la un anumit punct ce se va întâmpla în următoarele cadre la client (acesta poate avansa, sau executa o săritură, etc) şi să se afişeze pe ecran efectele. Desigur acest mecanism

funcţionează la nivel de fracţiuni de secundă şi în cel mai bun caz la nivel de câteva secunde, dar de obicei este de ajuns pentru ca sistemul de trimitere/primire a pachetelor să îşi revină şi clientul să primească din nou pachete de la server, moment în care clientul poate să corecteze acţiunile care au fost prevăzute greşit.

În cel mai bun caz, acţiunile au fost prevăzute corect şi clientul nici nu va şti că s-au pierdut date, în cazul mediu se fac mici corecţii imperceptibile, iar în cel mai rau caz starea clientului s-a modificat foarte mult din cauza lipsei pachetelor de la server.

2.4.2.5 Clienţi care încearcă să păcălească sistemul

Există câteva modalităţi de a trata această problemă, dar nu exista o soluţie care să funcţioneze în toate cazurile, găsindu-se mereu noi modalităţi de a ocoli orice protecţie.

Nu este plăcut pentru nimeni sa joace un joc în care cineva este invincibil datorită faptului că trişează. Acest lucru se poate produce în multe feluri, de exemplu modificând clientul să nu arate zidurile, adăugând lumini sau elemente ce pot scoate în evidenţă poziția celorlalţi jucători foarte uşor, modificând ţinta astfel încât de fiecare dată să nimerească adeversarul cu precizie absolută.

Toate aceste lucruri sunt modificari ce se întâmplă în partea de client a jocului şi când sunt facute așa cum trebuie sunt aproape de neobservat de către server.

S-ar putea, de exemplu, verifica precizia fiecărui jucător şi eliminarea celor care au o precizie care depășește un anumit prag. În acest fel se pot pierde jucători foarte buni dar este puţin probabil ca cineva poate avea o precizie de peste 80% (*[KES05]*) tot timpul.

Integritatea clientului este aspectul important aici. Soluţia adoptată de jocurile Quake1 şi Quake3, aceea a unei maşini virtuale unde în locul încărcării jocului de către client acesta “construieşte” şi “compilează” jocul în urma instrucţiunilor primite de la server este un start bun.

Tot ceea ce poate face dezvoltatorul de jocuri / spaţii virtuale este să facă sarcina celui care vrea să păcălească sistemul cât mai dificilă.

2.5 CONCLUZII

Principala caracteristică a aplicaţiilor MMO este dată de faptul că oferă posibilitatea unui număr ridicat de utilizatori să se întâlnească şi să interacţioneze în cadrul aceleiaşi lumi virtuale. În funcţie de scara aplicaţiei MMO, acest număr poate să varieze de la câteva sute ajungând până chiar la zeci de mii de utilizatori ce accesează simultan spaţiul virtual. Pentru a crea un grad de realism ridicat utilizatorilor ce folosesc spaţiul virtual, aplicaţiile MMO trebuie să ofere pe partea de client o redare 3D cât mai bogată din punct de vedere vizual şi pentru a realiza acest lucru sunt utilizate motoare grafice 3D în timp real moderne.

În acest capitol au fost prezentate categoriile de aplicaţii MMO cele mai des întâlnite pe piaţă şi au fost detaliate conceptele specifice folosite de acestea. De asemenea au fost trecute în revistă tehnologiile 3D majore care sunt folosite de motoarele grafice 3D în timp real.

Noţiunile descrise în acest capitol au fost prezentate şi într-o carte la care autorul lucrării de faţă a fost co-autor *([MMA09a]*).

3 ANALIZA SOLUŢIILOR ARHITECTURALE ACTUALE PENTRU SERVERE DE SPAŢII VIRTUALE 3D MMO

3.1 FUNCŢIONALITĂŢI DE BAZĂ

O arhitectură a unei aplicaţii MMO trebuie să ofere următoarele funcţionalităţi de bază:

- Autentificare utilizatori
- Gestionarea drepturilor de acces ale utilizatorilor în lumea virtuală
- Consistenţa lumii virtuale
- Gestiunea ordinii evenimentelor
- Propagarea evenimentelor către entităţile spaţiului virtual
- Stocarea securizată a caracterelor şi posesiunilor acestora
- Planificarea operațiilor computaţionale
- Efectuarea calculelor de actualizare a stării lumii virtuale
- Timpi de raspuns scăzuţi
- Securitatea lumii virtuale pentru a preveni eventualele încercări de a trișa

Se pot astfel identifica în cadrul unei arhitecturi generice următoarele componente fundamentale:

1) Componenta de **Autentificare** : este responsabilă cu controlul accesului la spaţiul virtual
2) Componenta de **Comunicare** : este responsabilă cu schimbul de mesaje şi gestiunea evenimentelor. Principalele sarcini ale acestei componente sunt de a menţine ordinea corectă a evenimentelor la nivelul intregului spaţiu

virtual, de a menţine o latenţă cât mai scăzută şi de a asigura mecanisme de securitate pentru a preveni încercările de trişare.

3) Componenta de **Stocare** : asigură stocarea persistentă a datelor lumii virtuale pe termen lung. Datele ce se pot stoca pot fi împărțite la rândul lor în două categorii : permanente şi temporare

4) Componenta **Computaţională** : este responsabilă cu planificarea şi execuția operațiilor ce ţin de logica spaţiului virtual. De asemenea, tot în cadrul acestei componente este necesară implementarea de mecanisme de securitate pentru a asigura corectitudinea operațiilor executate de utilizatori.

3.2 ZONARE

Cum s-a menționat şi anterior, un aspect important de luat în considerație atunci când se proiectează arhitectura este faptul ca aplicaţiile MMO trebuie să simuleze lumi virtuale de întindere foarte mare şi acest lucru face necesară existenţa unui număr ridicat de servere, uneori de ordinul sutelor, pentru a găzdui în mod complet mediul virtual. Astfel, a apărut necesitatea dezvoltării unor metode de a împărţi spaţiul virtual în mai multe subspații distincte (*[GLI08]*).

În prezent, exista două abordări distincte atunci când vorbim de împărţirea spaţiului virtual în subspații.

3.2.1 ZONARE NETRANSPARENTĂ UTILIZATORULUI

Cea mai simpla din punct de vedere al implementării este cea a împărţirii stricte pe zone care practic partiţionează întreaga lume virtuală în diferite părţi. Aceste părţi sunt suficient de mici pentru a putea fi gestionate de un singur server.
Granițele acestor zone sunt conectate între ele din punct de vedere logic şi un utilizator se va conecta la serverul corespunzător atunci când trece dintr-o zonă în alta. Dezavantajul evident al acestei abordări este dat de faptul că există un timp de încărcare atunci când se schimba zonele, acesta ducând la scăderea realismului simulării (de obicei, atunci când este vorba de un MMO dinamic, utilizatorii pot traversa un număr mare de zone într-un interval de timp relativ scurt pentru a îndeplini diverse obiective în cadrul spaţiului virtual).

3.2.2 ZONARE TRANSPARENTĂ UTILIZATORULUI

Cea de-a doua abordare îşi propune crearea unei simulări "seamless" prin care utilizatorul să nu îşi dea seama de existenţa zonelor şi a graniţelor acestora. Acest lucru va crea utilizatorului senzaţia că se află într-o lume 'continuă' care funcționează fără 'întreruperi'.

Deşi nu este vizibilă explicit utilizatorului, separarea logica în zone a lumii virtuale există. La fel ca la abordarea precedentă, zonele importante care se învecinează sunt gestionate de servere diferite. În exemplul din figura de mai jos, atunci când utilizatorul iese din zona gestionată de serverul zonei 1, va exista o zonă de tranziție care este sincronizată cu zonele adiacente. Preluarea utilizatorului de către serverul zonei 2 nu se va efectua imediat ci doar atunci când acesta va traversa complet zona de tranziție.

Dezavantajul acestei abordări este complexitatea ridicată din punct de vedere al programării precum şi posibilitatea apariției unui trafic crescut atunci când există multe evenimente în zonele de tranziție.

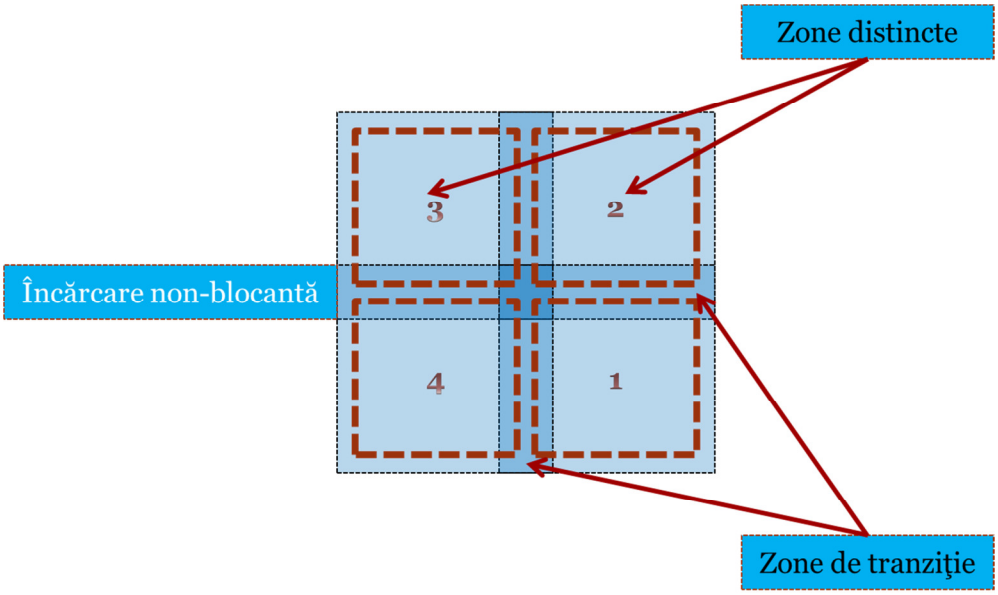

Figura 3-1: Zonare fără „întreruperi" a lumii virtuale

3.3 FOLOSIREA DE INSTANŢE

Într-o aplicaţie MMO, o instanţă reprezintă o zonă specială care îşi creează câte o copie a ei (instanţele sunt create şi gestionate de obicei în cadrul arhitecturii spaţiului virtual de către grupuri de servere dedicate acestora), de unde şi denumirea de „instanţă”, pentru un anumit grup de utilizatori care doresc să o acceseze. Astfel, acel grup de utilizatori accesează practic acea copie obţinându-se astfel o izolare faţă de ceilalţi utilizatori ai spaţiului virtual deoarece conţinutul instanţei este vizibil numai în contextul grupului de utilizatori care a accesat-o.
De obicei, exista servere dedicate pentru instanţe, acestea reprezentând zone mai deosebite ale spaţiului virtual, al căror design nu ar permite accesul unui grup numeros de utilizatori simultan.

Câteva dintre avantajele folosirii de instanţe ar fi :

- Reducerea operațiilor ce trebuie executate de serverele care simulează spaţiul virtual global, acestea fiind transferate serverelor pentru zonele instanțelor.

- Reducerea traficului de reţea al utilizatorilor, acesta fiind practic izolat în cadrul instanței în care se află, obținându-se astfel un timp de răspuns scăzut pentru aceştia.

- Posibilitatea proiectării unor experienţe focalizate, având ca ţintă numai un anumit număr de utilizatori

- Eliminarea competiției, care poate crea uneori mari probleme, asupra unor anumite resurse / obiecte datorită limitării numărului de utilizatori care pot accesa o instanţă.

Cu toate aceste avantaje însa este de observat ca există şi un dezavantaj al folosirii de instanţe şi anume cel al scăderii realismului simulării spaţiului virtual datorită izolării utilizatorilor din instanţă faţă de restul spaţiului virtual.

3.4 ARHITECTURI TRADIȚIONALE CLIENT-SERVER

3.4.1 DESCRIERE GENERALĂ

Marea majoritate a aplicaţiilor de tip MMO folosesc o arhitectură de tipul Client-Server. Clienţii accesează spaţiul virtual prin intermediul unui *server de conectare* care apoi ii direcţionează către un *shard server*.

Shard-urile reprezintă versiuni independente ale aceleiaşi lumi virtuale şi se folosesc pentru a îmbunătăţi scalabilitatea aplicației. *Shard-urile* nu sunt sincronizate şi utilizatorii de pe un s*hard* nu pot interacţiona cu utilizatorii de pe alt *shard*.

Deşi această soluţie permite un anumit grad de scalabilitate al aplicaţiilor MMO, ea limitează gradul de realism împiedicând interacţiunea unui număr mare de utilizatori.

Unele aplicaţii MMO, cum ar fi, de exemplu, *Eve Online ([EVE11])* încearcă găsirea de noi soluţii, care să nu folosească *shard-uri* şi să permită interacţiunea **tuturor** utilizatorilor lumii virtuale. Aceste încercări sunt inca în faza incipientă şi momentan depind foarte mult de particularităţile lumii virtuale.

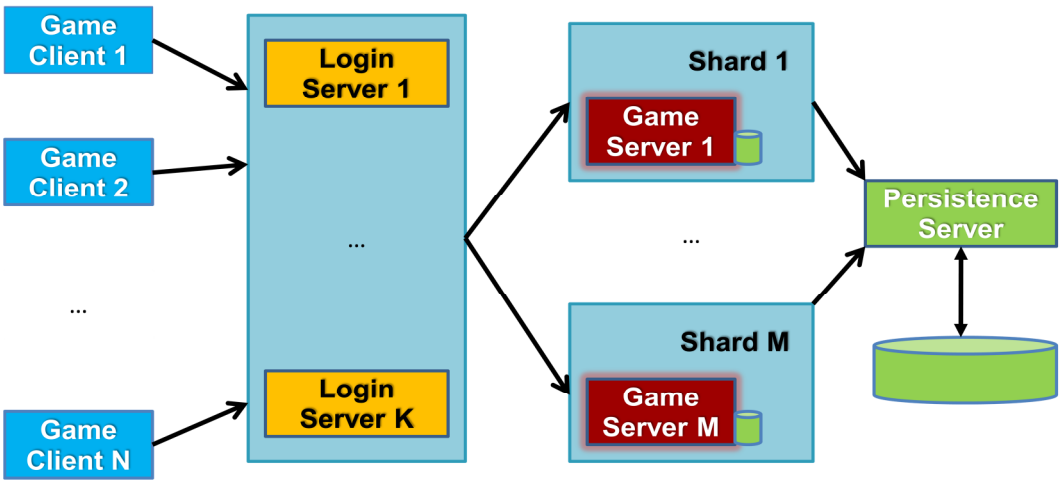

Figura 3-2: Arhitectura clasică MMO client-server

Arhitectura client-server este în general preferată deoarece un sistem centralizat oferă în mod natural următoarele avantaje:

- consistenţa lumii virtuale
- securitate ridicată
- evitarea problemelor de sincronizare
- implementare mai simplă
- model de afaceri viabil

3.4.2 INTERACŢIUNEA DINTRE APLICAŢIILE CLIENT ŞI SERVER

Serverul şi clientul au ambele o arhitectură bazată pe nivele, nivelul cel mai de jos ocupându-se de comunicarea prin Internet folosind protocoale de comunicatie standard. Din motive de securitate, următorul nivel este unul care ascunde formatul intern al datelor transmise folosind mecanisme de criptare/decriptare.

Ultimul nivel este cel aplicație şi el reprezintă nivelul arhitectural propriu-zis al spaţiului virtual 3D MMO, fiind cel mai complex nivel al aplicației.

Pentru a face transparentă nivelului aplicație complexitatea nivelului de criptare, se folosește un nivel intermediar de mesaje între acestea două.

Din punct de vedere conceptual, aplicaţia client interacţionează cu serverul într-o manieră specifică șablonului de proiectare *publisher / subscriber*. Datorită naturii aplicaţiilor MMO, serverul trebuie să comunice cu un număr ridicat de clienţi, astfel pentru a minimiza traficul, serverul nu ii anunța pe aceştia imediat ce are loc o schimbare. În schimb, clienţii se înregistrează ca *subscribers* la server, astfel că serverul va anunța modificarile doar clienţilor înregistraţi pentru notificare.

Avantajul acestei abordări este că serverul nu trebuie să memoreze informaţii detaliate referitoare la stările clienţilor. Astfel, se simplifică logica aplicației şi creşte performanţa serverului.

Pentru a minimiza traficul de reţea şi a folosi tehnici de *load balancing*, unele componente ale aplicaţiei vor fi replicate la nivel de client (de exemplu, elementele de grafică în timp real) iar altele vor fi păstrate doar pe server (de exemplu, informaţii de autentificare ale utilizatorilor, logica spaţiului virtual,etc).

3.4.3 COMPONENTELE PRINCIPALE ALE APLICAŢIEI SERVER

Aplicaţia server conține mai multe componente:

- Componenta de logică a spaţiului virtual
    - Modulul de inteligenţă artificială
    - Modulul de gestiune a entităţilor spaţiului virtual
    - **Modulul de arbitrare**
- Componenta de reţea
    - Modulul de reţea
    - Modulul de criptare/decriptare
    - Modulul de mesagerie
- Componenta de stocare a lumii virtuale
    - Modulul reprezentării lumii virtuale
    - Modului bazei de date al lumii virtuale
- Componenta de gestiune a utilizatorilor
    - Modulul de autentificare
    - Modulul bazei de date de utilizatori

Modulul de arbitrare este componenta centrală a serverului, fiind responsabilă cu transmiterea de mesaje între diferitele module ale acestuia. Acesta gestionează ceasul principal (*tick*) al spaţiului virtual 3D MMO şi transmite mesaje utilizatorilor care se află în lumea virtuală. Modulul arbitru comunică direct cu componenta care gestionează lumea virtuală, cu cea care gestionează utilizatorii, cu modulele de gestiune a entităţilor şi de inteligenta artificială şi cu componenta de reţea.

Componenta de stocare a lumii virtuale conţine modulul reprezentării lumii virtuale şi modulul bazei de date corespunzătoare. Baza de date a lumii virtuale stochează informaţii despre toate obiectele care sunt prezente în întreg spaţiul virtual 3D. Modulul reprezentării lumii virtuale memorează un subset al informaţiilor din baza de date a lumii virtuale, pentru a permite accesul rapid al modulului arbitru la acestea. Aceste informaţii sunt memorate într-un cache, fiind evitate astfel apelurile repetate la baza de date care sunt consumatoare de timp şi afectează performanţa sistemului.

Componenta de gestiune a utilizatorilor conţine modulul de autentificare al utilizatorilor şi baza de date a acestora. Modulul de autentificare decide pe baza

unor reguli şi a unor informaţii din baza de date a utilizatorilor dacă un anumit utilizator se poate conecta la server. De asemenea, tot la nivelul acestei componente se stabilesc rolurile şi drepturile de acces pe care utilizatorii le au în cadrul lumii virtuale. Baza de date a utilizatorilor deține informaţiile utilizatorilor precum: user-name, parola, nume real sau informaţii despre personajul care reprezintă utilizatorul în cadrul lumii virtuale.

Modulul de gestiune al grupurilor spaţiului virtual stochează informaţii despre diferitele grupuri de care poate aparține un anumit utilizator. Acest modul permite trimiterea de mesaje de către arbitru către un întreg grup de utilizatori. Utilizatorii sunt reprezentaţi în lista de notificări menținută în acest modul.

Modulul de inteligenţă artificială este responsabil cu gestiunea personajelor care nu sunt sub controlul direct al utilizatorilor, aceste personaje fiind denumite generic NPC (non player characters).

Componenta de reţea conține modulele de reţea, de mesaje şi de criptare/decriptare. Modulul de mesaje se ocupă cu crearea şi transmiterea mesajelor. În momentul în care modulul arbitru decide trimiterea unui mesaj către utilizatorii din lista sa de notificări, acesta este compus de către modulul de mesaje, dupa care este transmis modulului de criptare/decriptare. Modulul de criptare/decriptare comunică direct cu modulul reţea. De asemenea, modulul de mesaje are şi rolul de a primi mesaje decriptate şi de a le translata în apeluri ale modulului arbitru. Modulul de mesaje servește scopului de furnizare a unei interfeţe uniforme pentru server şi client şi realizează totodată minimizarea numărului de pachete care sunt trimise prin reţea. Modulul reţea este responsabil cu iniţierea, acceptarea sau distrugerea de conexiuni.

Clientul înglobează componentele lumii virtuale care sunt stocate pe calculatoarele utilizatorilor. Pentru a obține performanţă maxima, este nevoie să se aloce clienţilor o cantitate cât mai mare din volumul de lucru al întregului sistem.

3.4.4 COMPONENTELE PRINCIPALE ALE APLICAŢIEI CLIENT

Aplicaţia client este alcătuita din următoarele componente principale :

- Componenta grafică
    - Modulul grafic
    - Modulul de vizualizare

  - Modulul de gestiune a intrărilor utilizatorilor
- Componenta de reţea
  - Modulul de mesaje
  - Modulul de criptare/decriptare
  - Modulul de reţea
- Componenta de control
  - **Modulul de arbitrare**
  - Modulul de comenzi
  - Modulul de chat

Șablonul folosit în arhitectura aplicației client este model-view-controller, fiind foarte răspândit în aplicaţiile care conțin interfeţe cu utilizatorul. Această arhitectură nu respectă în totalitate modelul initial MVC, fiind eliminată legatura dintre model şi view pentru a obține o viteză mărita.

Modulul arbitru este analog celui prezent în arhitectura serverului. Acesta este responsabil cu luarea unor decizii locale, spre deosebire de omologul său de pe server care se ocupă cu luarea unor decizii globale. Modulul arbitru este responsabil cu transmiterea mesajelor între diferitele module ale clientului şi cu gestionarea ceasului local al clientului.

Componenta grafică conține modulele de grafica, de vizualizare şi de gestiune a intrărilor. Modulul de vizualizare este apelat de către arbitru în vederea folosirii funcţionalităţilor modulului grafic. Modulul grafic conține toate apelurile grafice dependente de platforma sau de API şi se ocupă cu redarea 3D a spaţiului virtual pe ecran. Modulul de vizualizare efectuează şi translatarea intrărilor primite de la utilizator în mesaje care sa poată fi înțelese de arbitru. Modulul de gestiune a intrărilor preia intrările de la utilizator şi le transmite modulului de vizualizare, care determină semnificația acestora şi transmite mesajele arbitrului. Acest modul abstractizează intrările provenite de la utilizator indiferent dacă acestea sunt transmise prin intermediul unei tastaturi, a unui mouse sau a altui dispozitiv de intrare.

Componenta de control conține modulele de comandă şi de chat. Modulul de comandă gestionează anumite comenzi ale utilizatorului care nu au legătură directă

cu interacţiunea personajului direct cu lumea virtuală. Astfel de comenzi pot fi cele de verificare a timpului pe server şi pe client. Modulul de chat este responsabil cu gestiunea comenzilor de chat prin formarea unui mesaj care este trimis arbitrului, acesta urmând sa il comunice la rândul lui serverului pentru distribuție.

Componenta de reţea conține modulele de mesaje, de criptare/decriptare şi de reţea. Modulul de mesaje creează mesaje care vor fi trimise serverului pentru distribuție şi translatează mesajele trimise de către server clientului. Modulul de criptare/decriptare efectuează criptarea/decriptarea mesajelor trimise, respectiv primite de client. Modulul de reţea conectează aplicaţia client la reţeaua fizică şi gestionează primirea şi trimiterea de pachete.

3.4.5 LIMITĂRI ŞI INCONVENIENTE

Arhitectura Client-Server distribuită pretinde că generează mai puţin trafic în reţea şi evită congestionarea serverului. Totuşi, au existat studii care afirmă că traficul în reţea al unui sistem centralizat este aproximativ acelaşi ca şi cel în cazul unui sistem distribuit. Acest rezultat poate să fie justificat prin topologia reţelei şi aplicarea diferitelor tehnici de optimizare specifice arhitecturii client-server.

O versiune la scară redusă a unui MMO poate să fie alcătuită dintr-un singur server şi mai mulţi clienţi conectaţi la server. Aceasta implementare limitează numărul de jucători ce pot accesa lumea virtuală precum şi dimensiunile acesteia. Pentru o versiune mai complexă şi mai imersivă a spaţiului virtual este necesar ca partea de server sa fie implementată folosind un număr mare de maşini fizice.

Deşi arhitecturile de tip Client-Server pun la dispoziţie funcţionalităţile necesare unei aplicaţii MMO, o fac însă cu un cost ridicat. Din nefericire, natura puternic centralizata a arhitecturii Client-Server introduce o sugrumare din punct de vedere al performanţei. În ciuda numărului lor, care poate fi foarte mare, clusterele de servere sunt limitate din punct de vedere computaţional şi traficul de reţea este concentrat practic în echipamentele din data center.

Costurile mari introduse de arhitectura Client-Server limitează practic scalabilitatea (*[GUP09]*) acesteia, fiind necesara împărţirea spaţiului virtual în mai multe instanţe / sharduri. Cu un număr relativ mic de servere care trebuie sa facă faţă unui volum mare de sarcini, o singură defecțiune poate sa provoace căderea întregului sistem. De asemenea, pentru a obține un nivel de performanţă care să facă faţă unui

număr mare de utilizatori, producătorii spaţiilor virtuale MMO se confruntă cu un cost financiar semnificativ pentru a menţine infrastructura sistemului funcţională.

3.5 ARHITECTURI P2P

3.5.1 DESCRIERE GENERALĂ

Aplicaţiile MMO la scară redusă, de cele mai multe ori, folosesc abordări P2P. Folosirea modelului Peer-to-Peer oferă anumite avantaje într-un mod natural atrăgând astfel cercetari în domeniu în ceea ce privește această abordare pentru aplicaţiile MMO (*[MIL10]*).
Aplicaţiile la scara redusă, în mod uzual, partajează între participanți încărcarea simulării unui mediu virtual. Astfel, abordarea P2P permite utilizatorilor noi să ofere sistemului resurse suplimentare astfel încât acesta sa facă faţă încărcării suplimentare pe care aceştia o introduc, sistemul reușind astfel să scaleze corespunzător.

Un alt avantaj al acestei abordări este că dacă una dintre resurse se defectează, ceilalţi participanți pot prelua atribuțiile acesteia oferind astfel sistemului robustețe. Traficul de reţea se desfăşoară în funcţie de necesităţile participanților permițând partajarea încărcării între utilizatorii implicaţi în mod direct.

Toate aceste caracteristici oferă de asemenea reducerea costurilor pentru proprietarul spaţiului virtual care practic reduce cheltuielile cu mententanţa spaţiului virtual.

Principala provocare în adaptarea unei lumi virtuale 3D MMO la o arhitectură peer-to-peer o constituie realizarea funcţionalităţilor serverelor centralizate după modelul distribuit.

Astfel, se impune menţinerea consistenţei unui spaţiu virtual partajat de un număr foarte mare de utilizatori fără intervenția unui server. Un concept important este acela al gestiunii interesului (*[BEZ08]*) care susţine că un utilizator singular nu trebuie sa aiba cunoștințe despre ceea ce se întâmplă în lumea virtuală atâta timp cât nu îl afectează şi că orice avatar are o viteza de mişcare şi o capacitate de detecție limitată. Viziunea utilizatorului asupra lumii virtuale se rezumă la un

anumit domeniu de interes, acesta fiind notificat numai în privința evenimentelor care se petrec în domeniul acestuia de interes.

Schemele de gestiune a interesului pot fi clasificate folosind :

- modele spațiale
- modele de regiuni de tip publish / subscribe
- modele de comunicaţie hibride

Modelul spațial mai este numit şi modelul "aura-nimbus" unde aura semnifică granițele care limitează prezenţa unui obiect în spaţiu, iar nimbus presupune nivelul de conștientizare reciprocă a două obiecte. Fiecare obiect ar trebui să comunice cu obiectele care se află în *nimbusul* său în vederea pregătirii interacţiunilor viitoare. Avantajul modelului spațial este acela că numai mesajele necesare sunt transmise între entităţi. Totuşi este nevoie ca entităţile să schimbe informaţii privitoare la actualizarea poziţiilor pentru a determina dacă domeniile acestora de interes se intersectează.

Determinarea de către un utilizator a vecinilor săi se realizează prin menţinerea de către acesta a unei diagrame interne, construită pe baza poziției celorlalţi utilizatori. Deşi se reduce numărul mesajelor schimbate, se creşte efortul computaţional impus de menţinerea diagramei interne.

Modelul regiunilor bazate pe tehnica publish / subcribe presupune împărţirea spaţiului virtual în regiuni statice. Responsabilitatea mecanismului de gestiune a interesului este aceea de a determina regiunile care intersectează domeniul de interes al utilizatorului şi de a forma o regiune de subscriere pentru evenimentele relevante pentru reuniunea regiunilor intersectate.

Avantajul acestui model îl constituie faptul că este mai simplu şi mai ieftin de calculat regiunea de subscriere a unui utilizator, decât calculul coliziunilor domeniului de interes al acestuia. Canalul de emitere al unei regiuni se poate mapa foarte uşor peste un grup de utilizatori, astfel încât evenimentele lumii virtuale sunt transmise eficient.

Modelul de comunicare hibrid este o mixtură între modelul spațial şi cel bazat pe regiuni. Lumea virtuală este împărțită în regiuni şi este desemnată o super-entitate

(care rulează pe unul sau mai multe calculatoare ale utilizatorilor din regiunea respectivă) care să se ocupe de fiecare regiune. Atunci când un utilizator este pe cale să intre într-o anumită regiune, acestuia i se comunica identitatea super-entității care gestionează regiunea curenta, căreia acesta trebuie să îi transmită actualizările în ceea ce privește poziția sa.

Utilizatorii ale caror domenii de interes sunt pe cale să se intersecteze sunt notificați de către super-entitatea care gestionează regiunea în care se află, în vederea realizării unei conexiuni peer-to-peer între aceştia. Acest model profita de avantajele modelelor spațial şi bazat pe regiuni, fiind mai uşor de implementat decât modelul spațial pur.

În comparație cu arhitectura Client-Server, arhitectura Peer-to-Peer adopta o serie de măsuri care reprezintă soluţii pentru problemele întâlnite în utilizarea arhitecturii Client-Server:

- interschimbarea mesajelor se realizează direct între utilizatori, fără a mai fi nevoie de trecerea printr-un server central
- se pot folosi resursele disponibile pe maşinile utilizatorilor în ceea ce privește puterea computaţională, capacitatea de stocare a informaţiilor şi lăţimea de bandă
- descentralizarea creşte robustețea sistemului deoarece se elimină punctul unic de defectare, a cărui cădere ar fi condus la încetarea funcționării întregului sistem

3.5.2 LIMITĂRI ŞI INCONVENIENTE

Deşi arhitectura Peer-to-Peer atunci când este folosită pentru implementarea serverelor de spaţii virtuale 3D MMO oferă în mod natural avantajele descrise mai sus, ridica în acelaşi timp şi o serie de probleme care trebuie rezolvate :

- riscul de securitate (*[WIE06]*) care apare odată cu oferirea puterii de decizie aplicației ce rulează la utilizator, acesta putând modifica aplicaţia pentru a obține avantaje care nu îi sunt cuvenite;
- menţinerea unei lumi virtuale persistente care să aibă aceleaşi caracteristici pentru toți utilizatorii care fac parte din ea;
- ordonarea evenimentelor şi propagarea acestora către toți utilizatorii.

Atunci când se foloseşte pentru schema de gestiune a interesului modelul spaţial, dezavantajele sunt date de dificultatea de determinare a dimensiunii optime a unei regiuni astfel încât utilizatorii să poată schimba mesaje între ei înainte să treacă în alte regiuni. Totuşi, folosirea unor regiuni mari conduce la primirea de către utilizatori de mesaje nerelevante. De asemenea, o altă deficienţă a modelului este aceea ca aceste regiuni nu vor funcționa bine atunci când obiectele sunt inegal distribuite.

De asemenea, folosirea modelului de comunicaţie hibrid poate necesita un volum mare de calcule şi mesaje interschimbate pentru super-entitatea care gestionează o regiune aglomerată, fiind necesare tehnici dinamice care să distribuie sarcinile unei mulțimi de super-entităţi, fiecare având în gestiune o sub-regiune. O super-entitate este un potenţial punct de defectare, fiind nevoie de pregătirea unor super-entităţi de rezervă care să preia controlul în cazul defectării celei curente.

3.6 ARHITECTURI HIBRIDE

Atât arhitectura client-server cât şi cea peer-to-peer prezintă unele avantaje şi dezavantaje. Exista unele studii (*[CHE06]*) care propun o arhitectură hibridă compusă din următoarele elemente:

- un server central
- mai mulți clienți selecționați ce sunt legați între ei şi cu serverul într-un mod peer-to-peer
- clienţi obişnuiţi care se conectează folosind mecanismul client-server

În funcţie de anumiţi parametri ce sunt calculaţi folosind diverşi factori (latentă, trafic generat, operaţii executate, etc) modul de procesare al clienţilor poate să fie schimbat dinamic între cele două abordări (client-server respectiv peer-to-peer).

Deşi arhitectura peer-to-peer reduce semnificativ traficul de date de la serverul central, ea este vulnerabilă la posibile defectări care pot duce la inconsistente în lumea virtuală. De aceea, ar fi indicată o arhitectură hibrida în care un client devine server local atunci când are suficienta putere computaţională, spaţiu de stocare şi lăţime de bandă. Pentru a trata defectările, serverul central are o copie a stării

jocului din fiecare server de joc local.

Serverul central se ocupă doar de schimbări de stare precum mutarea într-o nouă regiune, completarea unui quest sau atacarea unui jucător. Majoritatea operațiilor sunt operaţii poziționale ce presupun mişcarea jucătorului prin lumea virtuală (fără trecerea în alte regiuni). Aceste operaţii pot fi executate de către serverele locale (clienţii selecționați).

Astfel, scade traficul de date îndreptat către server, şi lăţimea de banda câştigată poate să fie folosită pentru mărirea numărului de utilizatori. Această arhitectură scalează mai bine decât arhitectura tradiţională client-server folosind şi unele din caracteristicile specifice abordării peer-to-peer.

3.7 GRID / CLOUD COMPUTING

Un alt mod de abordare ce ar putea fi viabil, mai ales datorită ultimelor dezvoltări tehnologice apărute în domeniu cât şi faptului ca unele cerinţe, cum ar fi lăţimea de bandă, devin din ce în ce mai accesibile utilizatorilor, ar fi cel care foloseşte pentru procesare Grid / Cloud Computing.

Astfel, partea de putere de calcul necesară arhitecturilor serverelor de spaţii virtuale 3D MMO ar putea fi obţinută din platformele de grid / cloud computing ce sunt disponibile pe piață.

În practică însă, există provocări ce trebuie depăşite pentru a se obţine o soluţie funcţională pentru o arhitectură de spaţiu virtual 3D MMO integrată cu grid / cloud computing şi din această cauză în acest moment există doar cercetări în această direcţie fără a fi disponibile şi soluţii funcţionale.

Câteva dintre provocările integrării arhitecturilor de spaţii virtuale 3D MMO cu soluţii de grid / cloud computing ar fi :

- complexitatea modificărilor ce trebuie efectuate la nivelul arhitecturilor actuale de spaţii virtuale 3D MMO pentru a folosi grid / cloud computing
- latenţa încă destul de ridicată pentru a putea folosi în timp real grid / cloud computing pentru un volum foarte mare de operaţii
- toleranţa la defecte

- securitatea accesului şi secretul datelor folosite în cadrul spaţiului virtual

La nivel de aplicaţii client, în ultimul an au fost lansate 2 platforme, bazate pe tehnologie de streaming, care oferă servicii de acces online la diverse jocuri clienţilor prin Internet : Gaikai (*[GAI11]*) şi OnLive (*[ONL10]*).
Aceste platforme sunt încă în fază incipientă, numărul de aplicaţii puse la dispoziţie este mic şi sunt foarte dependente de lăţimea de bandă a utilizatorilor care le accesează.

3.8 CONCLUZII

Arhitectura cea mai des întâlnită folosită de aplicaţiile MMO este în prezent cea Client-Server. Aceasta pune la dispoziţie functionalitatile necesare unei aplicaţii MMO, însa o face cu un cost ridicat. Avantajul major al unei soluţii cu servere centralizate este că oferă un grad mare de control asupra sistemului, care face posibila asigurarea autentificării utilizatorilor, persistenţei şi securităţii spaţiului virtual.

Din nefericire, natura puternic centralizată a arhitecturii Client-Server introduce o sugrumare din punct de vedere al performanţei. În ciuda numărului lor, care poate fi foarte mare, clusterele de servere sunt limitate din punct de vedere computaţional şi traficul de reţea este concentrat practic în echipamentele din data center.

Costurile mari introduse de arhitectura Client-Server limitează practic scalabilitatea acesteia, fiind necesara împărţirea spaţiului virtual în mai multe instanţe / sharduri. De asemenea, pentru a obţine un nivel de performanţă care să facă faţă unui număr mare de utilizatori, producătorii spaţiilor virtuale MMO se confruntă cu un cost financiar semnificativ pentru a menţine infrastructura sistemului.

Arhitecturile Peer-To-Peer, chiar dacă rezolvă unele dintre neajunsurile arhitecturilor Client-Server, au la rândul lor câteva lipsuri semnificative care le fac de nefolosit la scara largă în aplicaţiile MMO moderne.

Cel mai important neajuns al arhitecturilor Peer-To-Peer este acela că nu au nici un mecanism prin care să asigure persistenţa stării lumii virtuale. Atunci când un utilizator se deconectează de la spaţiul virtual, resursele puse la dispoziţie dispar, inclusiv datele pe care aplicaţia acestuia le gestiona.

Faptul ca nu exista un mecanism de autoritate centralizat face foarte dificilă actualizarea spaţiului virtual de către producători şi introduce posibile găuri de securitate.

De asemenea, deşi participanţii pot pune la dispoziţie resurse suplimentare sistemului, acest lucru se realizează într-un mod eterogen, fiecare maşina având propriile limite computaţionale. Fără un mecanism simplu de distribuire a încărcării, o concentrare mare de activităţi asupra unui peer poate sa consume foarte repede resursele acestuia.

Ţinând astfel cont de lipsurile şi inconvenientele pe care arhitecturile actuale le prezintă , este necesară explorarea de noi soluţii privind arhitecturile pentru servere de spaţii virtuale 3D MMO şi de asemenea găsirea unor modalităţi de optimizare a operaţiilor efectuate în cadrul simulării unui lumi virtuale pentru a elimina, sau măcar reduce, o parte din problemele curente cu care se confruntă aplicaţiileMMO.

4 TEHNICI INOVATIVE DE PARALELISM PENTRU SPAŢII VIRTUALE 3D MMO BAZATE PE GPGPU

GPGPU (General Purpose on Graphical Processing Units) reprezintă o metodă prin care unitatea de procesare grafică poate fi utilizată pentru a executa calcule/programe de uz general. Acest lucru a devenit posibil odată cu apariția etapelor programabile ale GPU precum şi datorită existenţei bibliotecilor şi instrumentelor de dezvoltare puse la dispoziţie de producătorii de unităţi de procesare grafică.

4.1 DE CE GPGPU

Unitatea de Prelucrare Grafică (GPU) modernă a devenit un hardware foarte versatil în domeniul arhitecturilor de calcul paralele multi-core (*[BRE08], [CHE08]*). Aceste arhitecturi, care includ GPU şi CPU multi-core de la AMD şi Intel, procesoarele CELL , procesoarele SUN UltraSparc, se diferențiază faţă de arhitectura clasică CPU prin următorul fapt : favorizarea operațiilor ce se pot executa în paralel asupra unei cantităţi mari de date faţă de execuția operațiilor *single-task* de latenţă scăzută.

Type	Processor	Cores/Chip	ALUs/Core
GPU	AMD Radeon HD 5870	20	80
	NVIDIA GeForce GTX 480	15	32
CPU	Intel XEON Westmere	6	2
	CELL	8	4

Figura 4-1: Arhitecturi GPU şi CPU moderne

Deşi există diferenţe de implementare între diverşii producători, toate GPU-rile moderne încearcă să menţină o eficienţă ridicată prin folosirea de arhitecturi multi-core, care folosesc atât multithreading hardware cât şi procesare SIMD – Single Instruction Multiple Data. Aceste tehnici nu sunt unice pentru GPU însă în comparaţie cu CPU, GPU duc la extrem aceste arhitecturi.

De exemplu, GPU NVIDIA GeForce GTX590 are 1024 de procesoare scalare care operează la 1,2 GHz. Acestea sunt organizate în grupuri de 32 şi au un *peak-rate* (vârf de performanţă) de 2,5TFLOPS. În comparaţie, un procesor high-end de la Intel , XEON Westmere care operează la o frecvenţă de 3.3GHz conţine 6 core-uri şi are un *peak-rate* de 146GFLOPS.

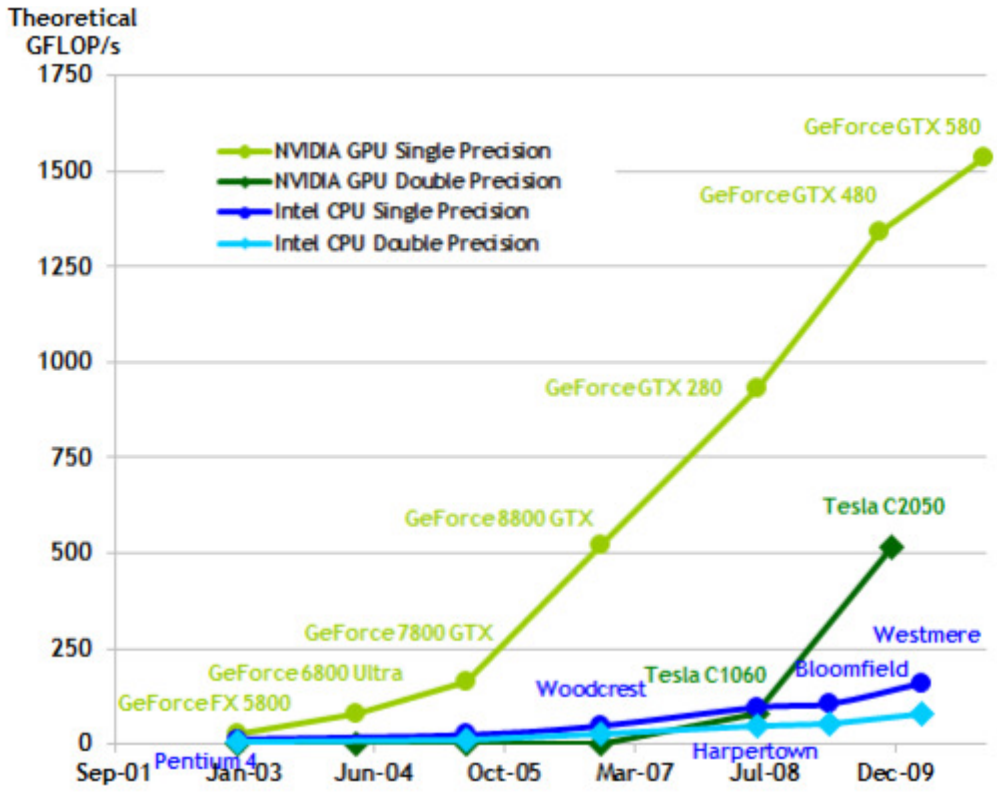

Figura 4-2: Comparaţie între NVidia GPU şi Intel CPU
Sursa : NVidia CUDA SDK 4.0

4.1.1 ABSTRACTIZĂRI CUDA

CUDA (Compute Unified Device Architecture) este o arhitectură hardware şi software pentru realizarea şi gestionarea calculelor pe GPU, fără a fi nevoie de maparea la un API specializat pentru grafică.
În programarea realizată cu ajutorul CUDA, GPU este văzut ca un co-procesor la CPU principal, numit gazdă. Cu alte cuvinte, porţiuni care sunt intensiv computaţionale şi prezintă paralelism de date, din aplicaţia ce rulează pe gazdă, sunt încărcate pe dispozitiv (GPU).
Această porțiune din aplicație este organizată sub forma unei funcţii care este executată simultan de un număr mare de fire de execuție. Funcția destinată rulării pe GPU este compilată în instrucțiuni specifice dispozitivului GPU, rezultând un program numit *kernel*.

CUDA furnizează 3 abstracții principale:

- ierarhie de grupuri de fire de execuție,
- spaţii de memorie
- sincronizarea prin bariere

Aceste abstractizări ușurează comunicarea şi sincronizarea între firele de execuție care reprezintă principalele unități computaţionale. Astfel, programatorul este îndrumat să își partiționeze problema în subprobleme de granularitate mare, care la rândul lor sunt divizate în subprobleme de granularitate mai mică ce pot fi rezolvate **independent** şi **cooperativ**.

1. **Ierarhie de grupuri de fire de execuție.**

Un grup de fire de execuție care cooperează prin partajarea eficientă a datelor şi își pot sincroniza execuția poartă numele **de bloc de fire de execuție**. Fiecare fir de execuție dintr-un bloc este identificat printr-un număr unic la nivelul blocului. Un bloc mai poate fi definit şi ca un tablou de fire de execuție cu 2 sau 3 dimensiuni. Dimensiunea unui bloc este limitată, însă se pot defini mai multe blocuri de aceeași dimensiune, care sunt **grupate într-un grid**.
Deoarece toate firele de execuție dintr-un grid executa aceeași funcţie kernel, pentru a se deosebi între ele şi a identifica porțiunile datelor pe care trebuie să le prelucreze, se folosește un mecanism de coordonate unic, ce permite identificarea

exacta a fiecărui fir de execuție în cadrul gridului (**blockId**) şi în cadrul blocului (**threadId**).

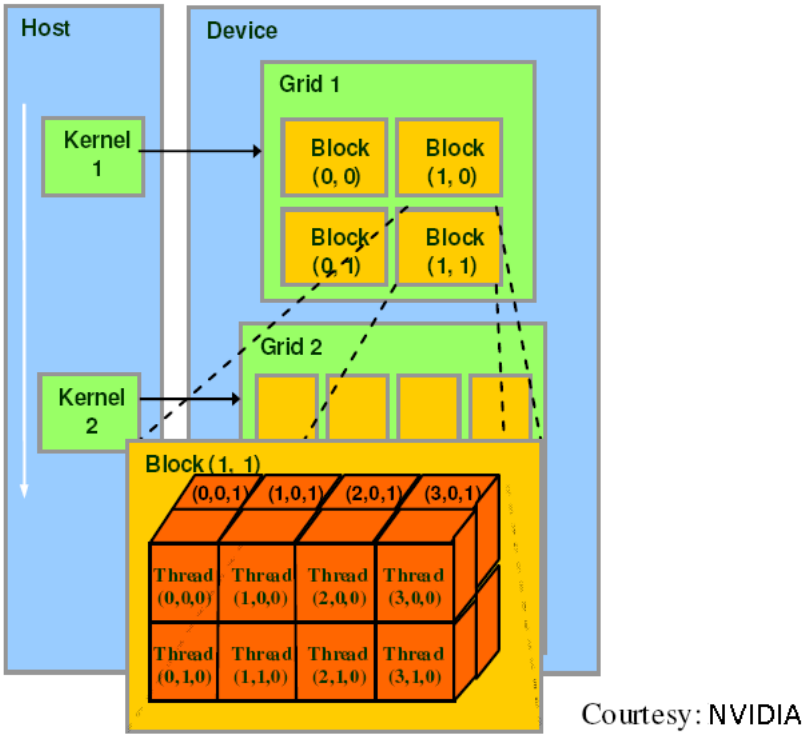

Figura 4-3: Ierarhie grupuri fire de execuţie

Firele de execuție din blocuri diferite nu pot însă comunica între ele şi nu se pot sincroniza. Dimensiunea şi forma blocurilor şi grid-ului fac parte din configurația de execuție a kernelului şi nu se pot modifica pe parcursul execuției.

Blocurile de fire de execuție trebuie să se execute independent. Este necesar să existe posibilitatea ca ele să se execute în orice ordine, în paralel sau în serie. Independența permite blocurilor de fire de execuție sa fie planificate în orice ordine şi executate pe oricâte procesoare, permițând programatorului să scrie cod scalabil. Numărul de blocuri de fire de execuție dintr-un grid este dictat în mod obişnuit de dimensiunea datelor care trebuie procesate şi nu de numărul de procesoare din sistem, număr care poate fi cu mult depăşit.

Un grid de blocuri este executat prin rularea unuia sau a mai multor blocuri pe fiecare multiprocesor folosind *time-slicing*. Fiecare bloc este împărţit în grupuri SIMD de fire de execuție numite ***warp-uri***, fiecare având acelaşi număr de fire de

execuţie, numit *dimensiunea warp-ului*. Un warp este executat pe un multiprocesor dupa modelul SIMD. Un planificator de fire de execuţie comută de la un warp la altul pentru a maximiza folosirea resurselor computaţionale ale multiprocesorului.

Modul în care un bloc este împărţit în warp-uri este mereu acelaşi: fiecare warp conține fire de execuţie consecutive cu threadId-uri crescătoare, primul warp incepand cu firul de execuţie cu threadId-ul 0. Un bloc este procesat de un singur multiprocesor, de aceea spaţiul de memorie partajată rezida în memoria shared on-chip a multiprocesorului, rezultând astfel viteze mari de acces.

În cazul în care numărul de fire de execuţie din bloc este mai mare decât numărul de registre ale multiprocesorului, kernelul nu va putea fi executat şi nu va fi lansat. Mai multe blocuri pot fi procesate de acelaşi multiprocesor concurent prin alocarea registrelor şi a memoriei partajate între blocuri

2. Spaţii de memorie

A două abstractizare majoră folosită de CUDA se referă la diferitele tipuri de memorie și gradul lor de accesibilitate. Ele diferă prin strategia de acces precum și prin vizibilitatea în cadrul ierarhiei de fire de execuție. Următoarele tipuri de memorie sunt vizibile pentru fiecare fir de execuţie la nivel de grid :

- **regiştri** :
    - cea mai rapidă memorie;
    - accesibilă doar în interiorul firului de execuție; acelaşi *scope* ca firul de execuție.
- **memoria partajată** :
    - împărțită în module de memorie de dimensiune egală numite *bancuri de memorie,* fiecare banc memorând o variabilă de 32 de biți; conflictele la nivel de bloc apar atunci când sunt accesate date din cadrul aceluiaşi bloc, caz în care hardware-ul serializează accesul la date forțând firele de execuţie să aştepte până când cererile de acces la memorie au fost rezolvate;

  - în cazul în care toate firele de execuţie citesc de la aceeaşi adresă de memorie partajată, se evită serializarea prin utilizarea unui mecanism de broadcast;
  - la fel de rapidă ca regiştrii atunci când nu exista conflicte între bancurile memoriei partajate;
  - vizibila de către toate firele de execuţie din interiorul blocului; acelaşi scope ca al blocului.

- **memoria globală** :
  - memoria globală şi memoria locală nu beneficiază de mecanism de caching, deci fiecare acces la memoria globală sau locală generează un acces explicit la memorie; un acces la memoria locală sau la memoria globală impune o latenţă suplimentară de 400-600 cicli de ceas şi astfel memoria globală este de aproximativ 150 de ori mai lentă decât memoria partajată;
  - dispozitivul este capabil sa citească 32, 64 sau 128 de biţi într-o singura instrucţiune însă datele trebuie sa fie aliniate corespunzător; altfel, compilatorul va genera mai multe instrucţiuni de load;
  - operaţiile de accesare a memoriei globale, din interiorul firelor de execuţie ale unui warp, ar trebui aranjate astfel încât să fie reunite într-o singură instrucţiune load;
  - accesibila atât din gazdă cât şi din dispozitiv; are scope-ul aplicaţiei.

- **memoria locală** :
  - nu dispune de un dispozitiv hardware dedicat, fiind alocată de către compilator în cadrul memoriei globale, având aceleasi performanţe ca aceasta; variabilele unui fir de execuţie pot fi plasate în memoria locală atunci când nu există un număr mare de variabile registru sau când datele ar ocupa prea mult spaţiu de memorie dacă ar fi plasate în regiştri;
  - considerabil mai lentă decât regiştrii şi memoria partajată;

- accesibilă doar în interiorul firului de execuție; acelaşi scope ca firul de execuție;
- spaţiile de memorie locală şi globală sunt regiuni read-write din memoria dispozitivului şi nu dispun de cache.

- **memoria constantă**
  - poate fi doar citita din interiorul unui kernel şi este optimizată pentru cazul în care toate firele de execuţie citesc din aceeași locaţie de memorie;
  - memoria constantă poate fi scrisă doar de către gazdă prin intermediul funcţiei **cudaMemcpyToSymbol** şi este persistentă pe durata execuției kernelurilor din cadrul aceleiaşi aplicaţii; se pot memora 64 de KB de date în cache-ul constant şi exista 8 KB de cache pentru fiecare multiprocesor;
  - durata accesului la date variază de la un ciclu de ceas în cazul unui cache hit până la câteva sute de cicluri de ceas, depinzând de localitatea memoriei.

- **memoria de texturi**
  - oferă capabilităţi de caching a memoriei globale;
  - reprezintă o modalitate de interacţiune cu capabilitățile de afișare ale GPU, având rolul de a accelera procesul de mapare şi deformare a unei texturi 2D pe un model poligonal 3D; de asemenea, reprezintă o modalitate mai simplă de accesare a memoriei globale, deoarece fiecare unitate de texturi are o memorie internă care stochează date din memoria globală;
  - utilizarea memoriei de texturi ofera o performanţă sporita în cazul în care firele de execuţie ale unui *warp* citesc de la adrese de memorie apropiate; o scădere de performanţă poate apărea doar atunci când se înregistrează un *cache miss* care impune citirea de date din memoria globală;
  - utilizarea memoriei de texturi poate avea următoarele beneficii :

- datele împachetate pot fi difuzate în variabile diferite într-o singura operație
- intregii pe 8 şi 16 biți pot fi convertiți în numere în virgula mobila pe 32 de biți cu valori între 0.0 şi 1.0 sau -1.0 şi 1.0 de către unitățile de texturare prin operaţii de interpolare liniară, bi-liniară şi tri-liniară care sunt realizate direct de către hardware dedicat

- utilizarea memoriei de texturi pe post de cache este uşor de folosit deoarece aceasta este optimizată pentru o localitate spațiala 2D şi poate oferi o creştere de performanţă când firele de execuţie ale unui warp accesează zone aflate în locaţii apropiate din interiorul texturii; pentru a putea folosi memoria de texturi este nevoie să se mapeze o textură la porțiunea de memorie care se doreşte a fi folosită;
- tipuri de memorie cu care poate fi asociată o textură :
  - memorie liniara : alocata cu cudaMalloc ; permite scrierea în memoria globală
  - tablouri CUDA : alocate cu cudaMallocArray sau cudaMalloc3D; nu este permisă scrierea în tablouri din interiorul kernelului
  - memorie 2D pitch linear : alocată cu cudaMallocPitch; permite scrierea în memoria globală;
- pașii care trebuie urmați pentru folosirea memoriei de texturi în CUDA sunt :
  - pe gazdă (CPU) :
    - alocare/obținere memorie (memorie liniara, pitch linear sau tablou CUDA)
    - creare obiect referinţă la textură
    - mapare referinţă textură la memorie
    - eliberare referinţă textură şi resurse

- pe dispozitiv (GPU) :
  - *fetch* utilizând referinţa la textură
  - se foloseşte tex1Dfetch pentru memorie liniara şi tex1D, tex2D, tex3D pentru tablouri şi *memorie pitch linear*

Se pot trage următoarele concluzii referitoare la utilizarea diverselor tipuri de memorie în cadrul arhitecturii CUDA :

- spaţiul de memorie local şi global sunt implementate ca regiuni read-write ale memoriei dispozitivului şi nu beneficiază de caching, ceea ce înseamnă că fiecare acces la memoria globală sau fizică generează un acces fizic
- pentru un multiprocesor, o instrucţiune de acces la memorie în cadrul unui warp durează 4 cicluri de ceas, în vreme ce un acces la memoria locală sau globală are o latenţă de 400-600 de cicluri de ceas
- cuvântul cheie *__device__* specifica o variabila care se află în memoria globală şi care nu poate fi accesată de către codul rulat pe gazdă; pentru a mari viteza de execuţie se încearcă evitarea accesurilor la memoria locală sau globala şi se încurajează utilizarea la maximum a variabilelor registru, *__shared__* sau *__constant__*
- pentru accesarea memoriei de texturi fiecare multiprocesor foloseşte o unitate de textură

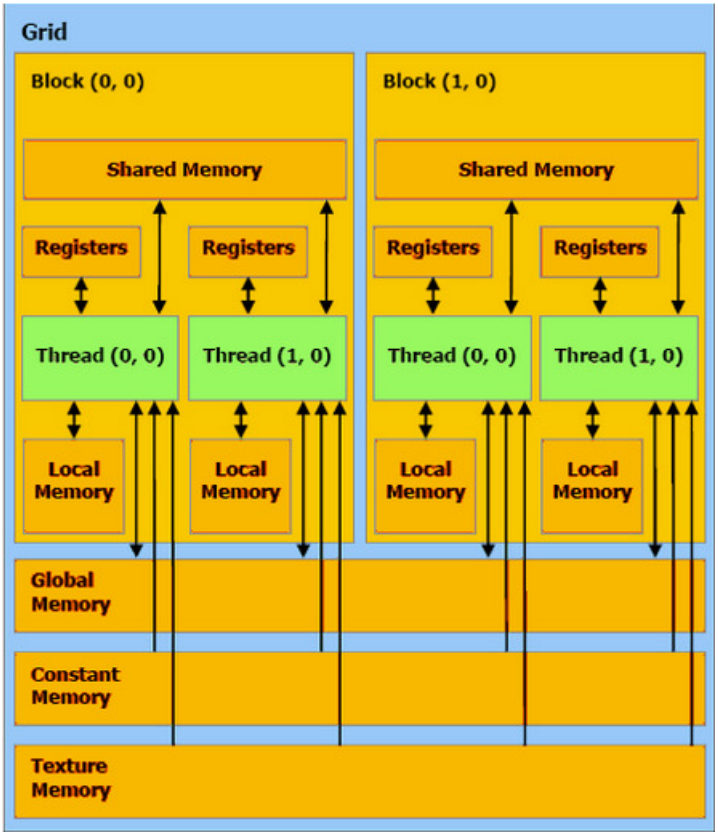

Figura 4-4: Spaţii de memorie

3. Sincronizarea prin bariere

CUDA pune la dispoziţie un mecanism special de sincronizare şi anume prin plasarea unor bariere. Astfel, toate firele de execuție dintr-un bloc vor fi blocate până când ultimul dintre ele atinge bariera. Costul asociat este foarte scăzut, nefiind întârzieri semnificative la apelarea instrucțiunii de sincronizare. Sincronizarea firelor de execuție în afara unui bloc nu este posibilă.

4.1.2 MODELUL DE EXECUŢIE CUDA

Arhitectura procesoarelor grafice care suportă CUDA constă într-un set de multiprocesoare cu memorie partajată on-chip. Aceste multiprocesoare suporta mai multe fire de execuție şi poartă numele de Streaming Multiprocessors (SM). Când un program gazdă invocă un grid de kernel-uri, blocurile grid-ului sunt distribuite către multiprocesoare. Firele de execuție dintr-un bloc se execută concurent pe un

multiprocesor. Când firele de execuție din blocuri se termină, noi blocuri sunt lansate pe multiprocesoarele rămase libere. Chipsetul 580GTX de la NVidia, care este construit pe cea mai nouă arhitectură de GPU-uri denumită Fermi, conține 16 astfel de multiprocesoare.

Figura 4-5: Arhitectura Fermi

Un multiprocesor conține 32 de procesoare scalare (SP), două unități speciale pentru funcții, o unitate de instrucțiuni cu mai multe fire de execuție, şi memorie partajată on-chip. Multiprocesoarele creează, gestionează şi execută firele de execuție concurente în hardware fără penalizări date de planificare. Bariera pentru sincronizare este implementată printr-o singură instrucțiune cea ce face sincronizarea foarte rapidă. Astfel se permite paralelism de granularitate fină, descompunându-se problema prin atribuirea unui fir de execuție fiecărui element de date.

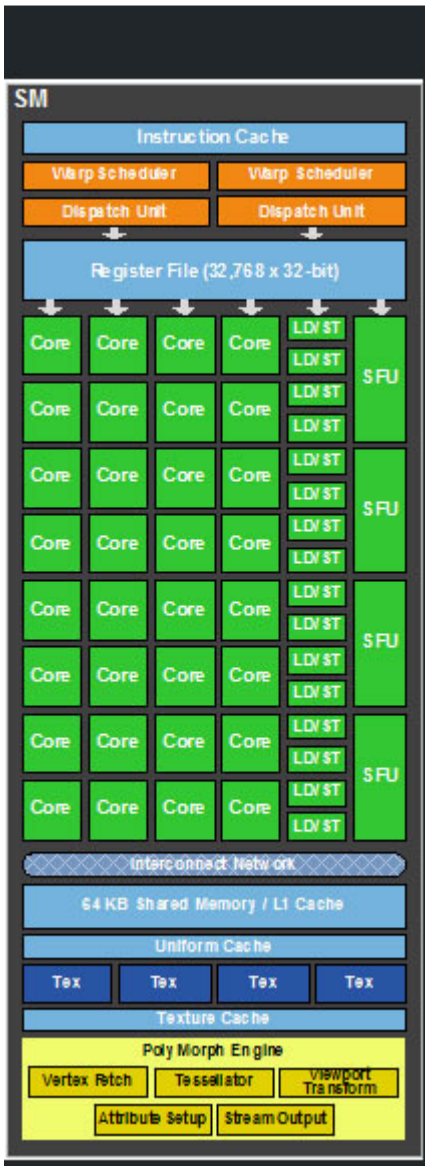

Figura 4-6: Conţinut SM din arhitectura Fermi

Multiprocesorul mapează fiecare fir de execuţie la un procesor scalar, astfel execuţia se va face independent folosindu-se adrese de instrucţiuni şi registre separate. Două sau mai multe blocuri pot rula pe acelaşi multiprocesor. Dacă un bloc de fire de execuţie este blocat în aşteptarea accesului la memoria globală, de exemplu, un alt bloc poate fi rulat.

Numărul de blocuri pe care un multiprocesor le poate procesa la un moment dat depinde de câte registre pe fir de execuţie şi câtă memorie partajată pe bloc sunt

necesare pentru un kernel, deoarece registrele multiprocesorului şi memoria partajată sunt resurse folosite de către toate firele de execuţie ale blocurilor active. Pentru gestionarea numărului mare de fire de execuţie multiprocesoarele folosesc o nouă arhitectură numită SIMT (single-instruction, multiple-thread).
Unitatea SIMT din multiprocesor, creează, gestionează, planifică şi execută firele de execuţie în grupuri de câte 32, numite warps. Firele de execuţie individuale ce compun un warp, pornesc împreună la aceeaşi adresă de program, dar sunt libere să se ramifice şi să se execute independent.

La fiecare moment la care se emite o instrucţiune, unitatea SIMT selectează un warp care este gata să se execute şi emite următoarea instrucţiune către firele de execuţie active ale warp-ului. Un warp execută o instrucţiune comună la fiecare moment de timp, deci eficienţa maximă este atinsă când toate cele 32 de fire de execuţie cad de acord asupra căii lor de execuţie.

Dacă firele de execuţie dintr-un warp diverg din cauza unor ramuri condiţionale dependente de date, warp-ul execută serial fiecare cale luată, dezactivând firele de execuţie care nu sunt pe acea cale şi când toate căile se termină, firele de execuţie converg înapoi pe aceeaşi cale de execuţie. Divergenţa ramurilor are loc doar în cadrul unui warp. Warp-urile diferite se execută independent, indiferent dacă execută căi comune sau disjuncte.

4.1.3 ELEMENTE DE LIMBAJ CUDA

Programarea în CUDA presupune izolarea unei portiuni dintr-o aplicaţie care se execută de multe ori pe date diferite, în mod independent, într-o funcţie care va fi rulata pe GPU. Aceasta funcţie poartă denumirea de *kernel* şi este compilată în setul de instrucţiuni specific device-ului, dupa care este transmisă acestuia spre execuţie.

O funcţie kernel este o funcţie care este apelată de pe gazdă şi este executată pe GPU de către mai multe fire de execuţie simultan.

Apelul unei funcţii kernel conţine numele acesteia, configuraţia de execuţie separată de “<<<” şi “>>>” şi parametrii funcţiei între paranteze rotunde.

Configuratia de execuție conține numărul de blocuri şi numărul de fire de execuţie din fiecare bloc.

La definiția funcţiei kernel se prefixează antetul acesteia cu cuvântul cheie *__global__* care specifica o funcţie kernel care se execută pe GPU, dar care poate fi apelată numai de pe sistemul gazdă. De asemenea, o funcţie kernel trebuie să întoarcă intotdeauna valori de tipul void.

Atât gazda cât şi dispozitivul au propria memorie DRAM numita memoria gazdei, respectiv memoria dispozitivului. Pentru a putea prelucra datele pe GPU este necesar transferul acestora de pe CPU pe GPU, acest lucru fiind realizat prin intermediul **cudaMemcpy**. Însa, înainte de a transfera datele pe GPU este nevoie să se aloce memorie pentru acestea folosind **cudaMalloc**.

De asemenea, transferul rezultatelor pe CPU se realizează tot prin intermediul cudaMemcpy, tipul de transfer al datelor (cudaMemcpyHostToHost, cudaMemcpyHostToDevice,cudaMemcpyDeviceToHost,cudaMemcpyDeviceToDevice) fiind specificat printr-un parametru.

CUDA oferă posibilitatea folosirii limbajelor de programare de nivel înalt cum ar fi C/C++ pentru a dezvolta aplicaţii care beneficiază de avantajul unui nivel ridicat de performanță şi scalabilitate pe care arhitecturile GPU le oferă. Scopul extensiei limbajului C oferită de CUDA este de a furniza acomodarea ușoară pentru utilizatorii familiari cu limbajul de programare C. Există şi unele limitări pentru codul ce va fi rulat pe dispozitiv, nefiind suportată recursivitatea şi nici pointerii la funcţii, dar există suport pentru șabloane, referințe și supraîncărcarea funcţiilor și a operatorilor asemănătoare cu cele din C++.

Extensiile de limbaj se pot grupa în patru categorii:

- Atribute pentru tipuri de funcţii pentru a se specifica dacă funcția se execută pe gazdă sau pe dispozitiv, şi dacă ea poate fi apelată de către gazdă sau numai de către dispozitiv. Exemple: __device__, __global__, __host__.
- Atribute pentru tipul variabilelor pentru a specifica locația de memorie unde va fi stocată o variabilă. Exemple: __device__ (spaţiul de memorie global), __constant__ (spaţiul de memorie constant), __shared__ (spaţiul de memorie partajată).

- Variabile predefinite care specifică dimensiunile blocului şi grid-ului precum și indicii firelor de execuție și ai blocurilor. Exemple: blockIdx, gridDim, threadIdx, blockDim, warpSize.

Exemplu pentru calculul identificatorului firului de execuție în cadrul grid-ului şi blocului :

```
__global__ void actiune_fir de execuţie(float* a_d, float* b_d,float *r_d,int N)
{
        // Identificarea poziției exacte a firului de execuţie
        int x,y;
        x = (blockIdx.x * blockDim.x + threadIdx.x);
        y = (blockIdx.y * blockDim.y + threadIdx.y);
        ...
}
```

Folosind atributul __global__ funcția de mai sus se va executa pe dispozitiv și va fi apelată din programul gazdă. Pentru calcularea identificatorului se folosesc variabilele predefinite blockDim, blockIdx şi threadIdx. Fiecare fișier sursă care conține aceste extensii trebuie compilat cu un compilator special oferit de CUDA și anume *nvcc*. Modul de operare al *nvcc* constă în separarea codului pentru dispozitiv, de codul rulat pe gazdă și compilarea codului pentru dispozitiv într-o formă de limbaj de asamblare (cod ptx) sau în formă binară (obiect cubin).
Aplicaţiile pot să ignore codul generat pentru gazdă şi să încarce și să execute codul ptx sau obiectul cubin pe dispozitiv, folosind CUDA driver API. Pentru codul rulat pe dispozitiv, compilatorul nu suportă opțiunii specifice C++ cum ar fi clasele, moștenirea, sau declararea variabilelor în blocuri de bază.

4.1.4 INTEGRARE CUDA CU BUFFEROBJECTS OPENGL

4.1.4.1 PBO (Pixel Buffer Object)

CUDA se poate folosi în combinaţie cu OpenGL prin utilizarea PBO (Pixel Buffer Object) pentru a manipula bufferul de imagine. Procesările sunt realizate în cadrul CUDA pixel cu pixel şi sunt redate pe ecran cu ajutorul OpenGL.

De asemenea, CUDA poate genera mesh-uri 3D care pot fi redate prin utilizarea VBO-urilor (Vertex Buffer Objects) OpenGL sub forma de suprafețe colorate, imagini wireframe sau mulțimi de puncte 3D.

CUDA mapează buffere-le OpenGL într-un spaţiu de memorie propriu prin intermediul funcţiei **cudaGLMapBufferObject**. OpenGL nu trebuie să opereze modificări asupra niciunui buffer care este mapat în spaţiul de memorie CUDA. Buffer-ul poate fi accesat prin intermediul unui pointer şi se pot efectua modificări din interiorul kernelului, lucrul cu acesta realizându-se la nivel de pixel.

Pentru a putea mapa buffer-ul OpenGL în spaţiul de memorie CUDA şi a putea efectua modificări asupra acestuia prin intermediul kernelului trebuie parcurşi următorii pasi:

- se creează o fereastră
- se creează un context OpenGL
- se setează viewport-ul şi sistemul de coordonate OpenGL
- se generează unul sau mai multe buffere OpenGL care vor fi partajate cu CUDA
- se înregistrează buffer-ele în CUDA

Crearea unui buffer se realizează prin intermediul funcţiei **glGenBuffers**, iar legarea acestora se realizează prin **glBindBuffer**. Alocarea memoriei se face prin apelul **glBufferData**, iar înregistrarea în CUDA se face prin apelul funcţiei **cudaGLRegisterBufferObject**.

Pentru a manipula bufferul de imagine dintr-o aplicație CUDA este necesară parcurgerea următorilor paşi:

- se alocă un buffer OpenGL cu dimensiunea egală cu a bufferului de imagine
- se alocă o textură OpenGL cu dimensiunea egală cu a bufferului de imagine
- se mapează buffer-ul OpenGL în memoria CUDA
- se scrie imaginea din CUDA în buffer-ul OpenGL mapat
- se demapează buffer-ul OpenGL
- se atașează textura buffer-ului OpenGL
- se desenează un quad care specifică coordonatele texturii în fiecare colţ
- se interschimbă bufferele pentru a se desena imaginea pe ecran

4.1.4.2 VBO (Vertex Buffer Object)

VBO (Vertex Buffer Object) se foloseşte pentru manipularea de obiecte 3D în CUDA şi redarea acestora folosind OpenGL sub forma unei mulțimi de puncte, imagini wireframe sau suprafețe.

Pentru a crea şi afişa obiectele 3D folosind VBO-uri, sunt necesare tablouri diferite pentru reținerea vârfurilor şi culorilor. Utilizarea VBO-urilor furnizează performanţe mai bune atunci când buffer-ele OpenGL sunt mapate în spaţiul de memorie CUDA.

Pentru utilizarea VBO trebuie parcurși următorii pasi:
- crearea unei ferestre OpenGL
- setare coordonate de vizualizare
- selectare dispozitiv CUDA
- creare buffere VBO pentru vârfuri şi culori
- înregistrare buffere VBO

4.1.5 EVOLUŢIE CAPABILITĂȚI CUDA

Primele versiuni de CUDA aveau următoarele caracteristici:
- integrare CUDA în driver-ul GPU
- API asincron pentru copieri de memorie şi lansări de kerneluri în execuție
- API pentru interogarea statusului apelurilor CUDA
- suport pentru depanarea programelor folosind gdb
- suport pentru Visual Studio 2008
- compilare Just-in-time (JIT) pentru aplicaţii care generează în mod dinamic kerneluri CUDA
- suport pentru template-uri C++ în kernelurile CUDA

Toolkit-ul 2.2 oferă faţă de versiunile anterioare următoarele caracteristici:
- Visual Profiler pentru GPU : CUDA Visual Profiler este un instrument grafic care activează analiza performanţelor kernelurilor care rulează pe GPU. Acesta dispune de instrumente de analiză pentru tranzacţiile cu

memoria, oferind dezvoltatorilor posibilitatea de a analiza timpul consumat pentru acest tip de operaţii

- îmbunătăţirea interoperabilității cu OpenGL - performanţă îmbunătăţită pentru aplicaţiile OpenGL care sunt executate pe GPU-uri profesionale din seria Quadro prin implementarea calculelor folosind CUDA
- zero-copy – permite funcţiilor CUDA să scrie şi să citească direct în/din memoria sistemului, reducându-se dimensiunea şi frecvenţa cu care datele sunt transmise între CPU şi GPU
- Pinned Shared Sysmem – aplicaţiile multi-GPU pot avea o performanţă crescută datorită posibilităţii de a accesa aceleaşi date din memoria sistemului de către mai multe GPU-uri
- memcopy asincron pe Vista – permite copierea datelor asincron, lucru care era disponibil pe alte platforme, dar nu şi pe Vista
- debugger hardware pentru GPU – CUDA GDB debugger pentru Linux oferă toate caracteristicile necesare pentru a depana direct pe GPU, inclusiv setarea de breakpointuri, inspectarea de variabile sau a stării.
- modul device exclusiv - este prezentă o opțiune de folosire exclusivă a GPU care garanteaza ca toata puterea de procesare şi memoria GPU este alocată aplicației respective
- schimbă paradigma în care datele erau copiate pe GPU, erau prelucrate pe GPU dupa care erau transferate pe host cu o noua paradigma în care se permite maparea memoriei gazdei în memoria device-ului prin intermediul funcţiei cudaMallocHost

CUDA 3.0 aduce ca elemente de noutate fata de versiunile anterioare următoarele caracteristici:

- interoperabilitate ridicată între CUDA şi API-urile 3D OpenGL şi DirectX
- CUDA-GDB dispune de un dispozitiv de verificare a memoriei în vederea adreselor nealiniate
- moștenirea claselor şi template-urilor C++
- sporirea capabilităților OpenCL
- **suport pentru arhitectura Fermi**
- permite atât programarea folosind API-ul de runtime cât şi cel de driver, care este mai greu de programat şi presupune scrierea unei cantităţi mai mare de cod, dar care ofera un nivel mai bun de control şi este independent de limbaj

- bibliotecile toolkit-ului CUDA suporta versionarea, foarte importantă pentru dezvoltarea de cod care să ruleze pe o varietate de sisteme fara a fi necesara recompilarea; acest lucru se realizează prin inspectarea versiunii driver-ului şi oferirea caracteristicilor necesare în funcţie de acestea.
- conține o versiune separată pentru CUDA C runtime pentru debugging în modul de emulare
- oferă o performanţă îmbunătăţită pentru programele compilate în modul de debugging deoarece registrele nu mai sunt forțate a fi reținute în memoria locală

CUDA 4.0 aduce ca elemente de noutate faţă de versiunile anterioare următoarele caracteristici:

- suport mult mai bun pentru aplicaţii multi-GPGPU :
    - posibilitatea de partaja accesul la GPU-uri din contextul mai multor fire de execuţie independente
    - utilizarea tuturor GPU-urilor din sistem în mod concurent din cadrul unui singur fir de execuţie
    - **acces Peer-To-Peer** : posibilitatea de comunicare directă între GPU-urile din sistem

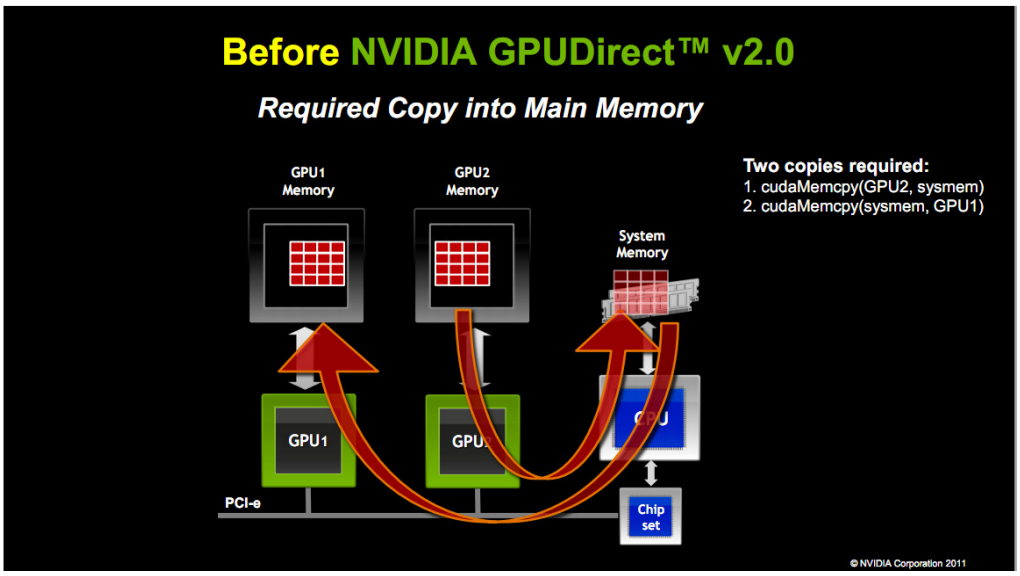

Figura 4-7: Comunicare inter-GPU înainte de CUDA 4.0.
Sursa: NVidia 2011

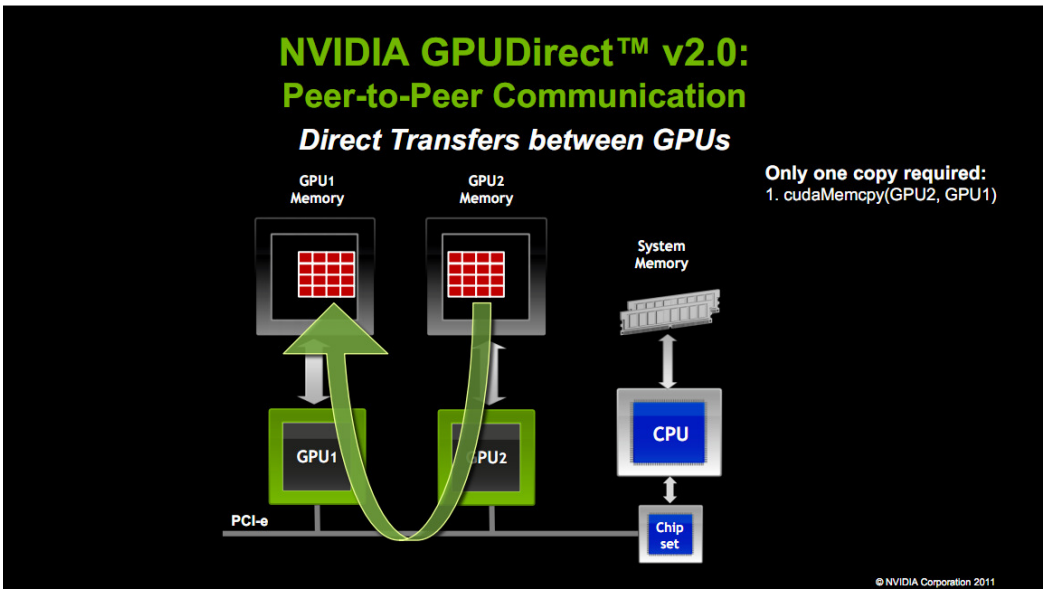

Figura 4-8: Comunicaţie inter-GPU Peer-To-Peer în CUDA 4.0.
Sursa: NVidia 2011

- **spaţiu de memorie virtual unificat** : existenţă unui singur spaţiu de adresare (CPU şi GPU) pentru o utilizare mai facilă a resurselor de memorie

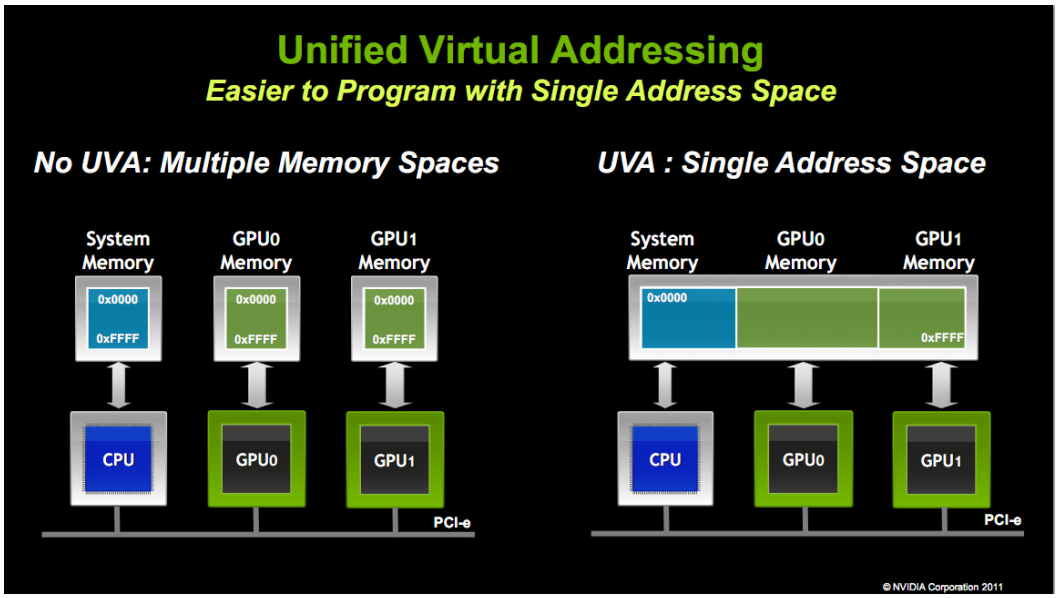

Figura 4-9: Spaţiu de adresare unificat în CUDA 4.0.
Sursa: NVidia 2011

- no-copy pinning pentru memoria sistem : o alternativă mai rapidă pentru cudaMallocHost
- suport pentru operatorii C++ new şi delete
- suport pentru funcţii virtuale
- punerea la dispoziţie a unei biblioteci cu funcţii dedicate pentru procesarea de imagini

4.2 UTILIZAREA GPGPU PENTRU REDAREA 3D CLIENT-SIDE

Aşa cum s-a menţionat şi la începutul lucrării, unul dintre obiectivele principale ale unei aplicaţii MMO 3D pe partea de client, este acela de a crea un grad de imersiune cât mai ridicat pentru utilizatorii spaţiului virtual. Aceste aplicaţii utilizează pe partea de client un motor grafic 3D modern care foloseşte multe din tehnologiile prezentate anterior pentru a crea un mediu virtual extrem de interactiv şi realist.

RayTracing reprezintă o metodă de redare, care utilizată în cadrul simulărilor 3D, poate să ofere un grad de realism net superior comparativ cu tehnologiile client-side prezentate anterior care utilizează ca metodă de redare banda grafică.

Până de curând, utilizarea RayTracing ca metodă de redare în timp real nu era posibilă deoarece necesita un efort computaţional ridicat şi hardware-ul existent nu era suficient de rapid pentru a obţine rezultate în timp real.

În continuare se va prezenta o descriere a algoritmului RayTracing, o soluţie pentru a executa algoritmul RayTracing folosind procesare GPU şi multi-GPGPU precum şi evaluarea rezultatelor obţinute.

4.2.1 SIMULAREA FOTO-REALISTĂ ÎN TIMP REAL A SCENELOR 3D FOLOSIND RAYTRACING

Comparativ cu metodele de redare clasice, prin utilizarea algoritmului de RayTracing se obţin, în mod natural, efecte dificil de implementat folosind metodele clasice :

- umbre

- reflexii
- refracţii

Toate aceste efecte sunt însă obţinute cu un cost computaţional ridicat şi acesta este principalul motiv pentru care RayTracing nu este folosit la scară largă în simulările 3D computerizate ce trebuie sa ruleze în timp real. Deşi RayTracing-ul este intensiv computaţional, existenţa independenţei între calculele efectuate în cadrul algoritmului îl fac un candidat optim pentru paralelizare.

4.2.1.1 Descrierea algoritmului RayTracing

RayTracing foloseşte pentru obținerea imaginii 2D finale care va fi afişată pe ecran, un set de raze care pornesc din punctul din care se privește scena. Razele vor trece prin fiecare pixel al planului virtual al ecranului urmând să intersecteze obiectele din scena 3D. Aceste raze se numesc raze primare iar procedeul prin care ele sunt folosite se numește raycasting reprezentând primă etapă a algoritmului RayTracing.

Imaginea 2D din planul virtual al ecranului corespunde unei ferestre de proiecție, iar pentru determinarea direcțiilor razelor, este necesară calcularea pozițiilor pixelilor în spaţiul scenei, lucru care se realizează printr-o transformare de coordonate.

După ce se calculează direcțiile pentru fiecare rază, urmează etapa a doua a algoritmului : intersecţia acestora cu primitivele din scenă. Astfel, pentru fiecare rază primară, se determină primitiva cea mai apropiată (care este intersectată prima), calculându-se poziția punctului de intersecție.

După ce s-a obţinut punctul de intersecție, se determină vizibilitatea față de sursele de lumină, pentru a se afla dacă punctul este sau nu umbrit. Pentru aceasta, se generează pentru fiecare sursă de lumină câte o rază ce are ca origine punctul de intersecție şi are direcţia către sursa de iluminare. Aceste raze se numesc raze de umbră şi sunt intersectate cu primitivele din scenă pentru a determina dacă există obstacole între punctul de intersecție şi sursele de lumină. La prima intersecție găsită, căutarea este întreruptă deoarece lumina de la sursă va fi blocată şi nu va mai ajunge la punctul de pe obiect şi astfel sursa de lumină către care este îndreptată raza de umbră curentă nu mai este luată în considerare la calcularea culorii punctului.

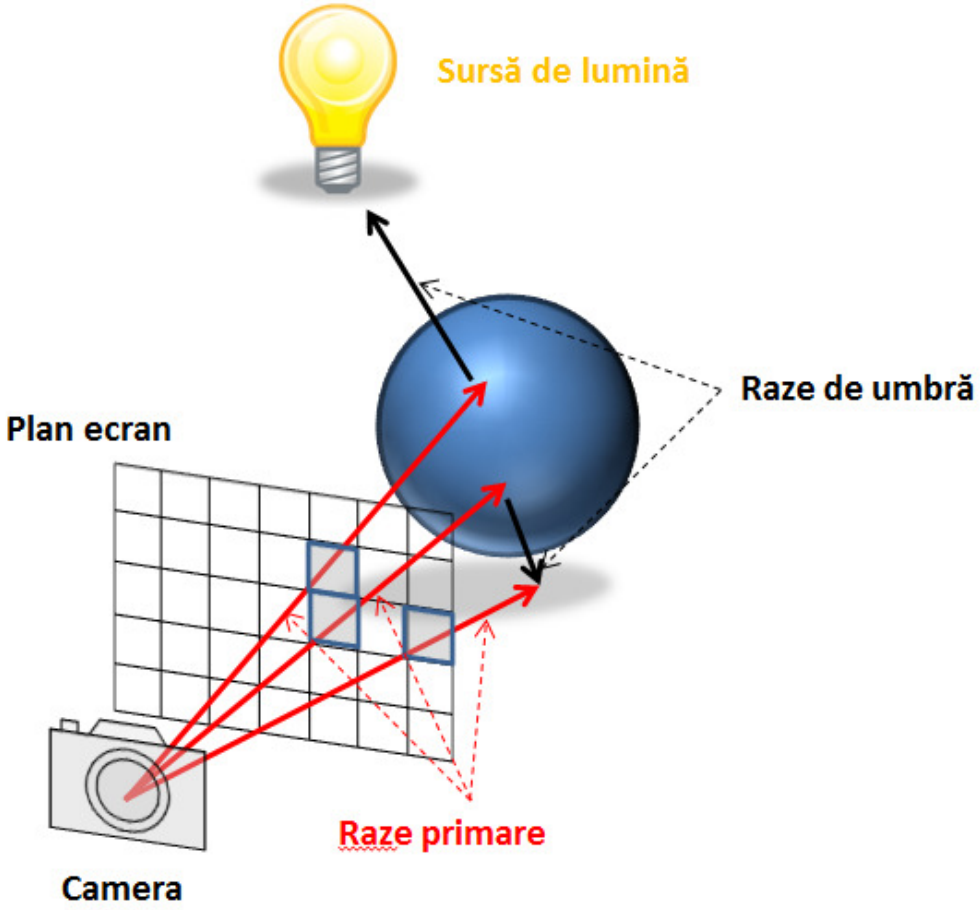

Figura 4-10: Algoritm RayTracing

Culorile pixelilor corespunzători punctelor de intersecţie se determină folosind calcule ce implementează un model de iluminare, la calculul culorii finale contribuind şi rezultatele obținute din razele de reflexie și refracție. După ce se efectuează toate aceste operaţii pentru fiecare pixel al imaginii din planul virtual al ecranului, fiecare pixel va avea atribuită o culoare iar imaginea generată poate fi afișată.

Datorită faptului că operaţiile executate pentru fiecare pixel pot fi executate independent unele față de celelalte, există un potenţial ridicat de paralelism pentru algoritm fiind posibilă executarea concurentă a calculelor pentru determinarea culorii finale a fiecărui pixel al imaginii.

Pseudo-codul pentru algoritm este următorul *[MOL96]* :

```
Pentru fiecare pixel al imaginii
{
      Construieşte rază de la observator prin pixel
      Iniţializează CelMaiAPropiatT cu infinit şi
      CelMaiApropiatObiect cu NULL

      Pentru fiecare obiect al scenei
      {
            Dacă Există Intersecţie(rază,obiect)
            {
                  Dacă t intersecţie < CelMaiApropiatT
                  {
                        CelMaiAPropiatT = intersecţie
                        CelMaiApropiatObiect = obiect curent
                  }
            }
      }
      Dacă CelMaiApropiatObiect este NULL
      {
            culoare pixel = culoarea de fundal
      }
      Altfel
      {
            Construieşte raze de umbră către toate sursele
            de lumină din scenă

            Dacă intersecţia nu este în umbră
            {
                  Dacă suprafaţa generează reflexie
                  {
                        Construieşte raza de reflexie
                        Reapelează algoritm pentru noua rază
                  }

                  Dacă suprafaţa este transparentă
                  {
                        Construieşte raza de refracţie
                        Reapelează algoritm pentru noua rază
                  }
                  culoare pixel = culoare calculată folosind
                  model de iluminare
            }
            Altfel
            {
```

```
                        culoare pixel = culoare calculată folosind
                        model de iluminare şi faptul ca pixelul
                        curent este în umbră
                }
        }
}
```

Se observă în pseudocodul de mai sus, faptul ca algoritmul este unul recursiv (prin reapelarea algoritmului pentru razele de reflexie si refracţie), pentru fiecare rază primară fiind generate alte raze, denumite raze secundare, care vor contribui la culoarea finală a pixelului. Astfel, în practică se foloseşte o adâncime maximă până la care se apelează algoritmul (numărul maxim de reflexii/refracţii secundare).

Razele de reflexie şi refracţie nu sunt generate decât dacă obiectul intersectat are un material care permite reflexia respectiv refracţia (grad de transparenţă). Etapa cea mai importantă şi costisitoare este cea a testelor de intersecţie deoarece conţine calcule cu vectori.

Razele generate au următoarea formă parametrizată:

$$\vec{\mathbf{r}}(\mathbf{t}) = \vec{\mathbf{o}} + \mathbf{t}\vec{\mathbf{d}}$$

unde :

- $\vec{o}$: originea razei
- $\vec{d}$: direcţia razei
- t : distanţa parcursă de raza.

În pseudocodul prezentat, t reprezintă acest parametru şi are valoarea distanţei pe care o rază trebuie să o parcurgă până la punctul de intersecţie.

4.2.1.2 Calculul culorii pixelului folosind un model de iluminare

Exista mai multe modele de iluminare (Phong *[PHO75]*, Gouraud, Blinn *[BLI77]*, Lambert, etc) care pot fi folosite pentru a determina modul în care lumina afectează obiectele din scenă.

Pentru a obţine astfel culoarea unui obiect iluminat vom avea următoarele componente :

- Componenta emisivă

- Componenta ambientală
- Componenta difuză
- Componenta specular

Contribuția fiecărei dintre aceste componente este calculată ca o combinaţie dintre proprietățile de material ale obiectului (factorul de strălucire, culoarea materialului) şi proprietățile sursei de lumină (culoarea sursei de lumină, poziția sursei de lumină).

Astfel pentru culoarea finală asociată obiectului vom avea :

culoareObiect = emisiva + ambientala + difuza + speculara

Componenta emisivă

Aceasta reprezintă lumina emisă de un obiect şi nu ţine cont de nici o sursă de lumină. Dacă un obiect care are o anumită culoare emisiva s-ar afla într-o scenă complet întunecată atunci el ar aparea exact cu această culoare.

O utilizare des întâlnită pentru componenta emisivă este aceea de a simula strălucirea unui obiect.

Avem astfel :

emisiva = Ke

Ke – culoarea emisivă a materialului

Componenta ambientală

Aceasta reprezintă lumina care a fost reflectată de către obiectele din scena de atât de multe ori încât pare să vină de peste tot.

Astfel, lumina ambientală nu vine dintr-o direcţie anume, apărând ca venind de fapt din toate direcțiile. Din aceasta cauză, componenta ambientală este independentă de poziția sursei de lumină.

Componenta ambientală depinde de culoarea de material ambientală a obiectului şi de culoarea ambientală a luminii.

Similar componentei emisive, componenta ambientală este o constantă (se poate extinde modelul atribuind fiecărei lumini din scena o culoare ambientală proprie).

Avem astfel :

*ambientala = Ka * culoareaAmbientalaGlobala*

Ka – culoarea de material ambientală a obiectului

culoareaAmbientalaGlobala – culoarea ambientală a luminii

Componenta difuză

Aceasta reprezintă lumina reflectată de suprafața obiectului în mod egal în toate direcțiile.

Cantitatea de lumina reflectată este proporțională cu unghiul de incidenţă al razei de lumina cu suprafața obiectului.

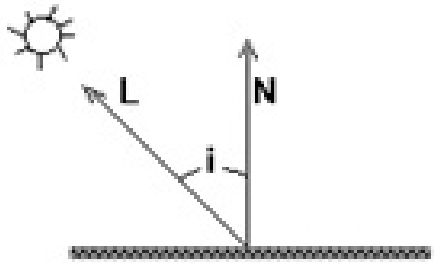

Figura 4-11: Calcul componentă difuză

Avem astfel :

*difuza = Kd * culoareLumina * max (N · L,0)*

Kd - culoarea de material difuză a obiectului

culoareLumina – culoarea difuză a luminii

N – normala la suprafață (normalizată)

L – vectorul direcției luminii incidente (normalizat)

max (N · L,0) – produsul scalar N · L ce reprezintă măsura unghiului dintre aceşti 2 vectori; astfel, dacă i este mai mare decât π/2 valoarea produsului scalar va fi mai mică decât 0 acest lucru însemnând că suprafața nu primește lumina (sursa de lumină se află în spatele suprafeței) şi de aici şi formula care asigură faptul că în acest caz suprafața nu primește lumina difuză

Componenta speculară

Un reflector perfect, de exemplu o oglindă, reflectă lumina numai într-o singură directie R, care este simetrică cu L faţă de normala la suprafață. Deci numai un observator situat exact pe directia respectivă va percepe raza reflectată.

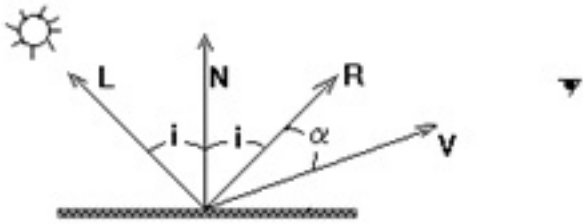

Figura 4-12: Calcul componentă specular

Componenta specular reprezintă lumina reflectată de suprafaţa obiectului numai în jurul acestei directii R.
În modelul Phong se aproximează scăderea rapidă a intensităţii luminii reflectate atunci când α creşte prin $\cos(\alpha)^n$, unde n este exponentul de reflexie specular al materialului (shininess).

O altă formulare a modelului Phong se bazează pe vectorul median, notat cu H. El face unghiuri egale cu L şi cu V. Dacă suprafaţa ar fi orientată astfel încât normala sa să aibă direcţia lui H, atunci observatorul ar percepe lumina specular maximă (deoarece ar fi pe direcţia razei reflectate specular).

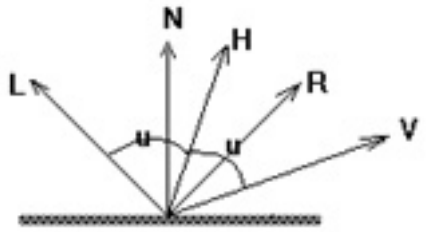

Figura 4-13: Calcul componentă specular folosind vectorul median

Termenul care exprimă reflexia specular este în acest caz :

$(Nu \cdot Hu)^n$ *unde* $Hu = (Lu+Vu)$ *(normalizat)*

Atunci când sursa de lumină şi observatorul sunt la infinit, utilizarea termenului Nu · Hu este avantajoasă deoarece Hu este constant.

După cum se observă, faţă de celelalte 3 componente, componenta specular depinde şi de poziţia observatorului. Dacă observatorul nu se află într-o poziţie unde poate vedea razele reflectate atunci nu va vedea reflexie specular pentru zona respectivă. De asemenea nu va vedea reflexie specular dacă lumina se află în spatele suprafeţei.

Ţinând cont de toate acestea avem pentru componenta speculară următoarea formulă :

$speculara = Ks * culoareLumina * primesteLumina * (max(N \cdot H,0))^n$

Ks - culoarea de material speculară a obiectului

culoareLumina – culoarea speculară a luminii

N – normala la suprafață (normalizată)

L – vectorul direcției luminii incidente (normalizat)

H – vectorul median (normalizat)

primesteLumina – 1 dacă N · L este mai mare decât 0, 0 în caz contrar

Atenuarea intensității luminii

Atunci când sursa de lumină punctiformă este suficient de îndepărtată de obiectele scenei vizualizate, vectorul L este acelaşi în orice punct. Sursa de lumină este numită în acest caz direcţională. Aplicând modelul pentru vizualizarea a două suprafeţe paralele construite din acelaşi material, se va obţine o aceeaşi intensitate (unghiul dintre L şi normală este acelaşi pentru cele două suprafeţe). Dacă proiecţiile suprafeţelor se suprapun în imagine, atunci ele nu se vor distinge. Aceasta deoarece în model nu se ţine cont de faptul că intensitatea luminii descreşte proporţional cu inversul pătratului distanţei de la sursa de lumină la obiect. Deci, obiectele mai îndepărtate de sursă sunt mai slab luminate. O posibilă corecţie a modelului, care poate fi aplicată pentru surse poziționale (la distanţă finită de scena) este:

*culoareObiect = emisiva + ambientala +factorAtenuare * (difuza + speculara)*

factorAtenuare = $1/d^2$ este o funcţie de atenuare; d este distanţa de la sursă la punctul de pe suprafață considerat.

Corecţia nu satisface cazurile în care sursa este foarte îndepărtată; de asemenea, dacă sursa este la distanţă foarte mică de scenă, intensităţile obţinute pentru două suprafeţe cu acelaşi unghi i, între L şi N, vor fi mult diferite.

O aproximare mai bună este următoarea:

factorAtenuare = $1/(Kc + Kl*d + Kq*d^2)$

unde Kc este factorul de atenuare constant, Kl este factorul de atenuare liniar şi Kq este factorul de atenuare pătratic

Lumina spot

Un efect interesant este acela de a face lumina sa aiba un efect de spot, astfel încât lumina sa afecteze doar obiectele ce se află în interiorul unui con de lumină (similar cu efectul produs de exemplu de o lampă de birou).

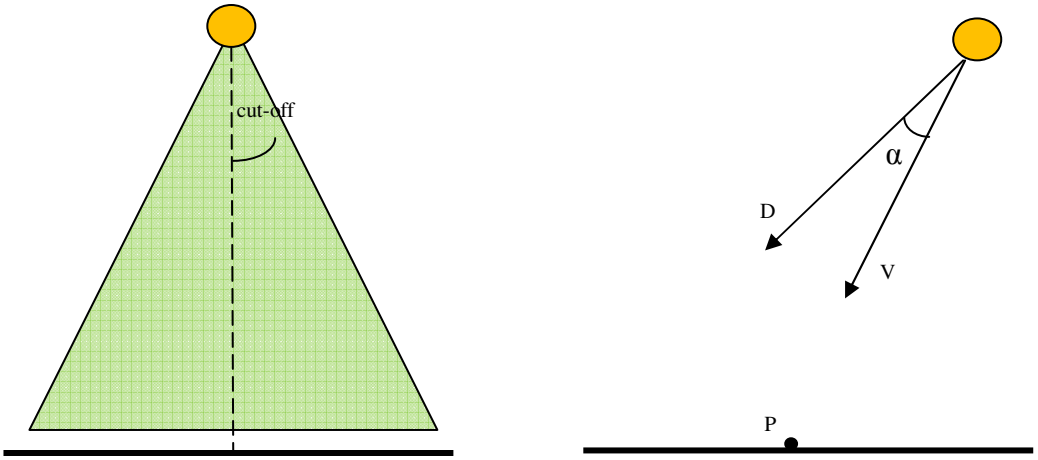

Figura 4-14: Lumină de tip spot

Pentru a crea o astfel de lumina este nevoie de următorii parametrii :

- poziția luminii
- direcţia spotului (D)
- unghiul de cut-off al spotului (controleaza “deschiderea” conului de lumina dupa cum se vede din figura de mai sus)
- poziția punctului care se dorește a fi iluminat (P)

Astfel din aceste Informaţii se poate deduce şi V care reprezintă vectorul directiei de la lumina la punctul P.

Pentru a determina dacă punctul P se află sau nu în conul de lumina este de ajuns sa determinam dacă este îndeplinita condiția :

$cos(\alpha) \geq cos(cut\text{-}off)$, unde $\cos(\alpha) = V \cdot D$

4.2.1.3 Testele de intersecţie

Obiectele din scena 3D care sunt folosite în implementarea algoritmului sunt descrise analitic. În continuare sunt prezentate ecuațiile prin care sunt descrise aceste obiecte precum şi rezultatele intersecţiei lor cu razele de lumină.

4.2.1.3.1 Sfera

Fie o sferă de centru $\mathbf{c} = (c_x, c_y, c_z)$ şi rază r ce este descrisă prin ecuația implicită:

$$(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 - r^2 = 0$$

Această ecuație se poate scrie simplificat în formă vectorială:

$$(\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) - r^2 = 0$$

În această ecuație $\mathbf{p} = (x, y, z)$ este un punct de pe sferă. Pentru a află intersecția cu raza, se consideră $\mathbf{p}$ aparținând razei descrisă de ecuația parametrică $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$. Se obține astfel:

$$(\mathbf{o} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{o} + t\mathbf{d} - \mathbf{c}) - r^2 = 0$$

Dezvoltând, rezultă ecuația pătratică:

$$(\mathbf{d} \cdot \mathbf{d})t^2 + [2(\mathbf{o} - \mathbf{c}) \cdot \mathbf{d}]t + (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2 = 0$$

cu delta $\Delta = 2\sqrt{\left(((\mathbf{o} - \mathbf{c}) \cdot \mathbf{d})^2 - (\mathbf{d} \cdot \mathbf{d})((\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2)\right)}$

Din valoarea lui delta se poate află câte intersecții sunt între rază şi sferă :

- delta > 0, există 2 intersecții
- delta = 0 există o singură intersecție
- delta < 0 raza nu se intersectează cu sfera.

Un caz special este atunci când raza intersectează sfera în două locuri, punctul de intersecție corespunzând celui mai mic t pozitiv.

$$t = \frac{-(\mathbf{o} - \mathbf{c}) \cdot \mathbf{d} \pm \sqrt{(\Delta)}}{\mathbf{d} \cdot \mathbf{d}}$$

Normala la sferă în punctul de intersecție este dată de gradientul ecuației vectoriale

$$\mathbf{N} = 2(\mathbf{p} - \mathbf{c})$$

4.2.1.3.2 Planul

Fie un plan versorul normalei $\mathbf{n} = (A, B, C)$ aflat la distanța D față de origine. Planul poate fi descris astfel prin ecuația implicită

$Ax + By + Cz + D = 0$

Ecuația se poate scrie simplificat în formă vectorială:

$\mathbf{n} \cdot \mathbf{p} + D = 0$

unde $\mathbf{p} = (x, y, z)$ este un punct din plan.

Pentru a determina intersecția cu raza, se consideră $\mathbf{p}$ aparținând razei descrisă de ecuația parametrică $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$.

Înlocuind, se obține:

$\mathbf{n} \cdot (\mathbf{o} + t\mathbf{d}) + D = 0$

Se observă că este o ecuație liniară din care se poate scoate parametrul t

$$t = -\frac{D + \mathbf{n} \cdot \mathbf{o}}{\mathbf{n} \cdot \mathbf{d}}$$

Dacă numitorul este 0, atunci raza este paralelă la plan, iar dacă atât numitorul cât și numărătorul sunt 0 atunci raza este inclusă în plan, ecuația fiind satisfăcută pentru orice t.

De interes este doar cazul în care numitorul este pozitiv, punctul de intersecție fiind dat de parametrul t pozitiv. Normala la plan în punctul de intersecție este dată de gradientul ecuației implicite şi valoarea este în acest caz chiar $\mathbf{n}$.

4.2.1.3.3 Triunghiul

Un triunghi este definit de trei puncte $\mathbf{V_1}$, $\mathbf{V_2}$ și $\mathbf{V_3}$, care dacă nu sunt coliniare, definesc un plan. Cel mai facil mod pentru descrierea acestui plan este folosind coordonatele baricentrice.

$\mathbf{p}(u, v) = (1 - u - v)\mathbf{V_0} + u\mathbf{V_1} + v\mathbf{V_2}$

Dacă se îndeplinesc condițiile $0 \le u \le 1$ și $0 \le v \le 1$ atunci punctul $\mathbf{p}(u, v)$ aparține triunghiului.

Pentru a află intersecția cu raza, se consideră $\mathbf{p}$ aparținând razei descrisă de ecuația parametrică $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$. Înlocuind, se obține:

$\mathbf{o} + t\mathbf{d} = (1 - u - v)\mathbf{V_0} + u\mathbf{V_1} + v\mathbf{V_2}$

Rearanjând termenii se obține sistemul liniar de ecuații:

$$\begin{bmatrix} -\mathbf{d} & \mathbf{V}_1 - \mathbf{V}_0 & \mathbf{V}_2 - \mathbf{V}_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \mathbf{o} - \mathbf{V}_0$$

Rezolvând acest sistem se obţin valorile pentru t, u şi v. Se fac notaţiile $\mathbf{E}_1 = \mathbf{V}_1 - \mathbf{V}_0$, $\mathbf{E}_2 = \mathbf{V}_2 - \mathbf{V}_0$ şi $\mathbf{T} = \mathbf{o} - \mathbf{V}_0$. Soluţia ecuaţiei anterioare se poate găsi cu regula lui Cramer:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\left|-\mathbf{d}, \mathbf{E}_1, \mathbf{E}_2\right|} \begin{bmatrix} \left|-\mathbf{T}, \mathbf{E}_1, \mathbf{E}_2\right| \\ \left|-\mathbf{d}, \mathbf{T}, \mathbf{E}_2\right| \\ \left|-\mathbf{d}, \mathbf{E}_1, \mathbf{T}\right| \end{bmatrix} = \frac{1}{(\mathbf{d} \times \mathbf{E}_2) \cdot \mathbf{E}_1} \begin{bmatrix} (\mathbf{T} \times \mathbf{E}_1) \cdot \mathbf{E}_2 \\ (\mathbf{d} \times \mathbf{E}_2) \cdot \mathbf{T} \\ (\mathbf{T} \times \mathbf{E}_1) \cdot \mathbf{d} \end{bmatrix}$$

Se consideră intersecţie doar dacă t obţinut este pozitiv. Normala în punctul de intersecţie este dată de interpolarea normalelor la vârfuri $\mathbf{N}_1$, $\mathbf{N}_2$, $\mathbf{N}_3$ în funcţie de coordonatele (u,v)ale punctului de intersecţie.

$$\mathbf{N} = (1 - u - v)\mathbf{N}_1 + u\mathbf{N}_2 + v\mathbf{N}_3$$

4.2.2 RAYTRACING FOLOSIND TEHNICI GPGPU

În ultimii ani, interesul pentru a oferi soluţii de redare ce folosesc RayTracing a înregistrat o creştere semnificativă atât în industria producătorilor de hardware (GPU şi CPU) cât şi în cea a dezvoltatorilor de jocuri. Există astfel în prezent modificări ale motoarelor grafice din unele jocuri apărute pentru a folosi RayTracing (*[QUA04], [QUA06] , [QWR08], [WOL10]),* programe demonstrative de la producătorii de GPU (*[NVI09a]*), încercări de a implementa motoare grafice ce folosesc RayTracing de la dezvoltatorii de jocuri (*[CAR11]*), etc.

Soluţiile/implementările curente de RayTracing pe GPU se concentrează în principal pe următoarele aspecte :

- Optimizarea algoritmului RayTracing folosind diverse tehnici de partiţionare spaţială pentru reprezentarea scenei 3D folosind arbori K-D (*[FOL05]*), BVH (Volume Încadratoare Ierarhice),etc;
- Execuţia algoritmului RayTracing pe GPU folosind banda grafică; acest lucru presupune de obicei scrierea de *shadere* (vertex shader, geometry shader şi fragment shader);

- Utilizarea de abordări hibride, unde părți ale algoritmului sunt executate pe GPU folosind shadere;
- Abordări distribuite ce executa algoritmul RayTracing pe mai multe maşini folosind un framework specializat (*[WAL04]).*

Spre deosebire de soluţiile prezentate mai sus, soluţia descrisă în continuare îşi propune să execute algoritmul RayTracing în totalitate pe procesoarele scalare ale GPU utilizând o implementare GPGPU, fiind astfel o abordare ce se diferenţiază de utilizarea standard a GPU, care foloseşte banda grafică cu programe de tip shader.

Având în vedere că soluţia propusă se concentrează pe implementarea algoritmului direct pe procesoarele scalare ale GPU am decis sa fie folosită o varianta a algoritmului RayTracing care este intensiv computaţională, fără alte optimizări suplimentare. În acest fel, se pot compara rezultatele obţinute de variantele GPGPU (single şi multi GPGPU) şi de implementarea doar pe CPU.

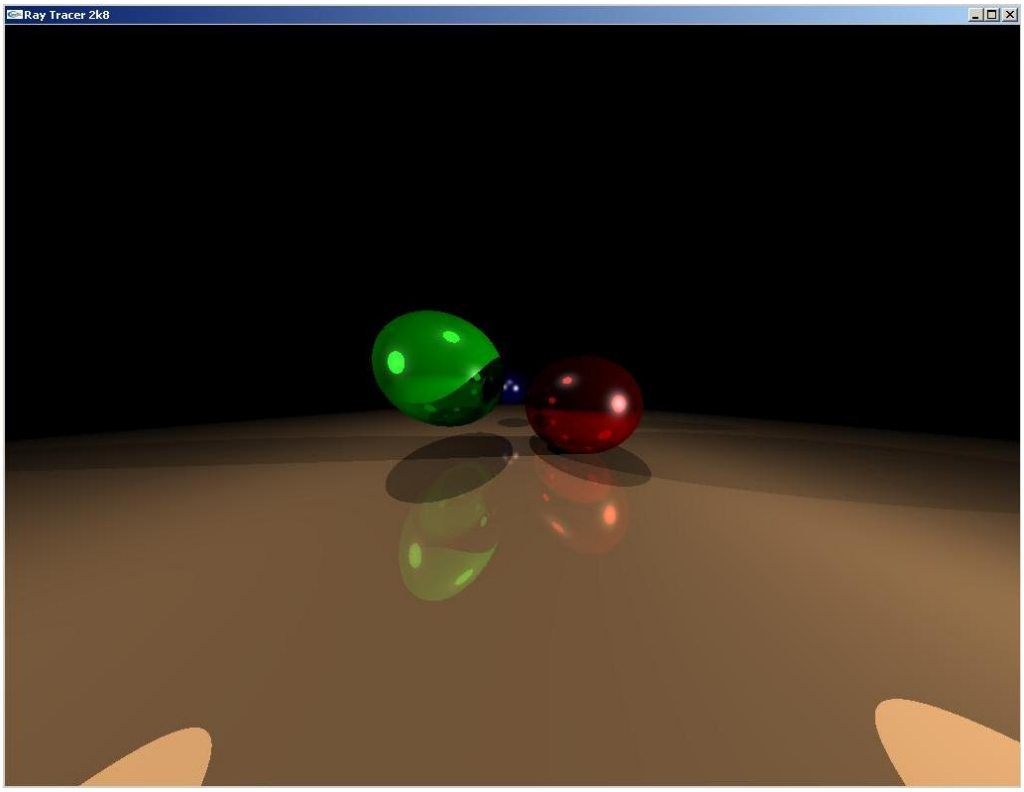

Figura 4-15: Scena redată folosind RayTracing

4.2.3 ADAPTARE ALGORITM RAYTRACING PENTRU GPGPU

Algoritmul pentru împărţirea calculelor RayTracing pe firele de execuţie CUDA este următorul :

- Datele iniţiale ale scenei sunt făcute disponibile în memoria globală a GPU;
- Setul de raze este determinat de o matrice de pixeli ce are dimensiunea ecranului (Figura 3-16);
- Matricea ecran va fi la rândul ei împărţită în sub-matrici ce au dimensiuni fixe (8,16 sau 32);
- Fiecare sub-matrice va fi procesată de un bloc de fire de execuţie unde fiecare fir de execuţie va fi responsabil de calculele executate pentru un element al sub-matricei (un pixel şi raza care trebuie sa treacă prin el);
- Pentru implementarea multi-GPGPU, imaginea se va împărţi la numărul de GPU, fiecare GPU având asignată o porţiune din imagine (de exemplu, dacă se folosesc două GPU, GPU0 va procesa jumătatea superioară a ecranului şi GPU1 va procesa jumătatea inferioară a ecranului)

```
GPUWorker gpu0(0);
GPUWorker gpu1(1);

gpu0.callAsync(bind(launch_thread,pixels,
device,dimGrid,dimBlock));

gpu1.callAsync(bind(launch_thread,pixels,
device,dimGrid,dimBlock));
```

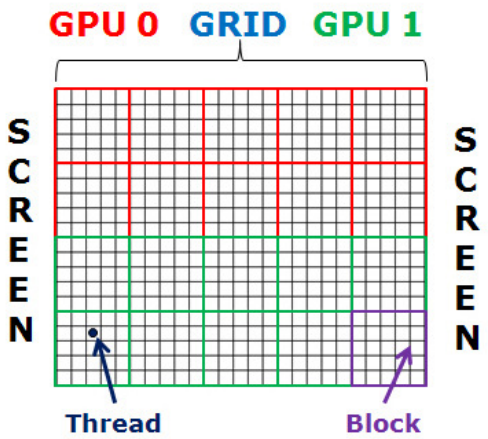

Figura 4-16: Împărţirea pe GPU / blocuri SIMD a sarcinilor

Pentru implementarea *single-GPGPU* pseudocodul algoritmului este următorul: :

```
DimensiuneBlocFireExecuţie = (32,32)
DimensiuneGridFireExecuţie =
(LăţimeRezoluţieScenă /  dimensiuneXbloc,
ÎnălţimeRezoluţieScenă / dimensiuneYbloc)

Copiază date în memoria globală GPU de pe maşina gazdă

Lansează în paralel pe GPU funcţie kernel(dimensiune
bloc,dimensiune grid)

Sincronizează prin barieră terminarea firelor de execuţie

Copiază date din memoria globală GPU pe maşina gazdă
```

Funcţia kernel ce se execută pe fiecare fir de execuţie de pe GPU este :

```
X = IdBlockX * DimensiuneBlocX + IdFirDeExecuţieX
Y = IdBlockY * DimensiuneBlocY + IdFirDeExecuţieY

IndiceMemorieImagine = X * LăţimeRezoluţieScenă + Y

RayTracing(pixel ecran identificat de IndiceMemorieImagine)
```

Pentru implementarea *multi-GPGPU* pseudocodul algoritmului se modifică astfel :

```
DimensiuneBlocFireExecuţie = (32,32)
DimensiuneGridFireExecuţie =
(LăţimeRezoluţieScenă/dimensiuneXbloc/nrGPU,
ÎnălţimeRezoluţieScenă/dimensiuneYbloc/nrGPU)

Pentru fiecare GPU
{
   Copiază date în memoria globală GPU de pe maşina gazdă

   Lansează în paralel pe GPU funcţie kernel(dimensiune
   bloc,dimensiune grid, idGPU)

   Sincronizează prin barieră terminarea firelor de execuţie

   Copiază date din memoria globală GPU pe maşina gazdă
```

```
}
```

Funcţia kernel ce se execută pe fiecare fir de execuţie de pe GPU este :

```
X = IdBlockX * DimensiuneBlockX + IdFirDeExecuţieX
Y = IdBlockY * DimensiuneBlockY + IdFirDeExecuţieY

Offset = ((LăţimeRezoluţieScenă*ÎnălţimeRezoluţieScenă)/nrGPU)
                              * idGPU
IndiceMemorieImagine = X * LăţimeRezoluţieScenă + Y + Offset
RayTracing(pixel ecran identificat de Indice)
```

4.2.4 ANALIZĂ TIMP DE EXECUŢIE

Timpul de execuţie al algoritmului RayTracing în implementarea pe CPU este următorul :

*T(W*H*NrObiecte*(NivelRazeSecundare+1))*,

unde,

W – lăţimea în pixeli a rezoluţiei scenei

H – înălţimea în pixeli a rezoluţiei scenei

NrObiecte – numărul de obiecte din scenă

NivelRazeSecundare – numărul maxim admis de ricoşee ce pot genera raze secundare în urma unor reflexii sau refracţii

Timpul de execuţie al algoritmului RayTracing în implementarea *single-GPGPU* este următorul:

*T(((W*H /(nrSM * nrSP))*(NrObiecte*(NivelRazeSecundare+1)))*

+ constantaOverhead,

unde,

W – lăţimea în pixeli a rezoluţiei scenei

H – înălţimea în pixeli a rezoluţiei scenei

NrObiecte – numărul de obiecte din scenă

NivelRazeSecundare – numărul de ricoşee ce pot genera raze secundare în urma unor reflexii sau refracţii

nrSM – numărul de multiprocesoare de pe GPU

nrSP – numărul de procesoare scalare de pe fiecare multiprocesor

constantaOverhead – constantă introdusă de operaţiile de transfer de memorie şi sincronizare

Timpul de execuţie al algoritmului RayTracing în implementarea *multi-GPGPU* este următorul:

*T(((W*H /(nrSM*nrSP*nrGPU))*(NrObiecte*(NivelRazeSecundare+1)))*

*+ nrGPU*constantaOverhead*,

unde,

W – lăţimea în pixeli a rezoluţiei scenei

H – înălţimea în pixeli a rezoluţiei scenei

NrObiecte – numărul de obiecte din scenă

NivelRazeSecundare – numărul de ricoşee ce pot genera raze secundare în urma unor reflexii sau refracţii

nrSM – numărul de multiprocesoare de pe GPU

nrSP – numărul de procesoare scalare de pe fiecare multiprocesor

nrGPU – numărul de GPU-uri disponibile

constantaOverhead – constantă introdusă de operaţiile de transfer de memorie şi sincronizare

4.2.5 EVALUAREA REZULTATELOR

Pentru scena de test au fost folosite :

- O rezoluţie de 1024x768
- Iluminare difuză şi specular ă de la 3 surse de lumină
- Generarea de umbre (care este obţinută natural de RayTracing)
- Generarea de reflexii, cu o limită de maxim 3 raze secundare ce pot rezulta în urma reflexiilor.

Pentru implementarea pe CPU (Single Core – 1 fir de execuţie şi Quad Core – 4 fire de execuţie) a fost utilizat un CPU Intel Core2Quad Q6600 şi algoritmul RayTracing a fost rulat folosind 1 fir de execuţie, respectiv 4 fire de execuţie.

Pentru implementările GPGPU (single şi multi) au fost folosite 2 GPU-uri NVidia 8800 Ultra montate în acelaşi calculator şi toolkit-ul CUDA 2.2.

Au fost testate toate cele 3 implementări şi rezultatele obţinute sunt cele prezentate în continuare.

	FPS Min	FPS Max	FPS Mediu
CPU Single Core 1 fir de execuţie	1	3	2.083
CPU Quad Core 4 fire de execuţie	5	7	5.983
NVidia 8800Ultra 128 procesoare folosind GPGPU	8	10	8.967
2xNVidia 8800Ultra 256 procesoare folosind GPGPU	16	18	16.980

Tabel 4-1: Rezultate implementare RayTracing folosind GPGPU

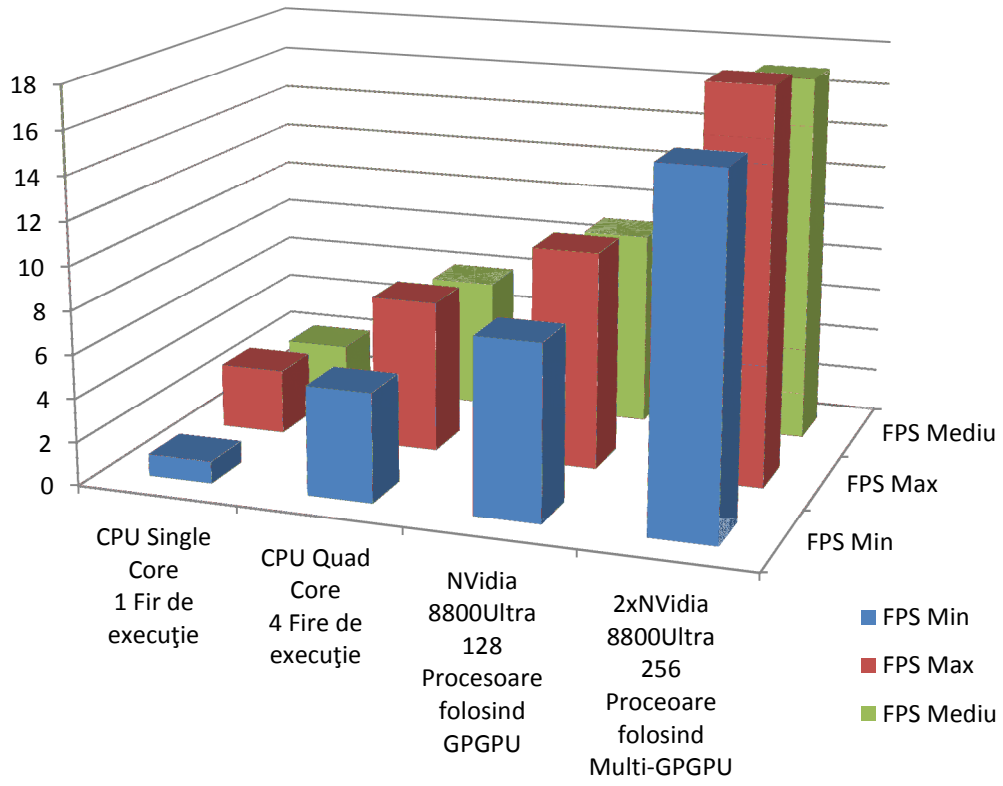

Figura 4-17: Rezultate rulare algoritm RayTracing

Rezultatele obţinute demonstrează faptul că algoritmul RayTracing scalează foarte bine atunci când este executat paralel şi ca se obţine un spor de performanţă prin implementarea GPGPU şi Multi-GPGPU.

4.3 UTILIZAREA GPGPU ÎN CADRUL SERVERELOR MMO

4.3.1 IDENTIFICARE OPERAŢII CE POT FI PARALELIZATE

4.3.1.1 Descriere generală

Cantitatea de informaţii pe care o aplicaţie trebuie să le proceseze este în continuă creştere. Aplicaţiile MMO monitorizează şi proceseaza un număr de peste 1 milion (*[LAK09]*) de operaţii pe secundă în perioadele de maximă încărcare.
Este clar ca scalabilitatea în ceea ce priveşte numărul de evenimente procesate este o cerinţă importantă a aplicaţiilor MMO.
Primele generaţii de sisteme de procesare a evenimentelor în cadrul aplicaţiilor MMO au fost de tip centralizat, însa de curând au apărut soluţii ce încearcă să obţină scalabilitatea folosind o abordare distribuită.
Partiționarea proceselor se dovedește însă a fi destul de dificilă datorită numeroaselor dependențe ce pot aparea între procese cât şi între resursele ce le procesează. De obicei, partiționarea se face într-un mod vertical, existând un singur set de partiționare ce folosește un mecanism de distribuire centralizat.

4.3.1.2 Logica spaţiului virtual

Nucleul unei aplicaţii MMO este reprezentat de o buclă internă în cadrul simulării, care se execută sincronizată cu un interval de timp ce corespunde unui anumit cu un frame-rate (de obicei 30 sau 60 de FPS - cadre pe secundă).
Din punct de vedere conceptual, starea lumii virtuale a unei aplicaţii MMO poate fi văzută sub forma unei tabele de date ce conține obiectele spaţiului virtual incluzând atât utilizatorii precum şi obiectele cu care aceştia interacţionează la un anumit moment de timp.
Pentru a face fata cerinţelor stricte de simulare în timp real, starea activă a spaţiului virtual este de obicei păstrată în memoria serverelor centrale ce asigură funcţionarea acestuia. Discurile fizice sunt folosite de obicei pentru a asigura persistenţa sau pentru a stoca informaţii auxiliare.

Pe parcursul fiecărei iteraţii (*tick*) a buclei principale a simulării, porţiuni ale stării sunt actualizate în concordanţă cu logica simulării spaţiului virtual. Aceste actualizări pot fi declanşate de acţiuni ale utilizatorilor, expirarea unui anumit interval de timp sau alte evenimente.
Este posibil ca o acţiune singulară a unui utilizator să declanşeze actualizări fizice multiple ale stării curente. De exemplu, o comandă de deplasare la nivelul utilizatorului se poate translata într-o serie de mici actualizări ale poziţiei acestuia care se desfăşoară pe parcursul a mai multor *tick-uri*.
Vor fi enumerate în continuare câteva dintre principalele operaţii ce se execută la nivelul logicii simulării spaţiului virtual şi care se execută de obicei folosind un mecanism distribuit, fiind astfel susceptibile la posibile optimizari.

4.3.1.3 Fizica spaţiului virtual

Pentru a asigura consistenţa şi corectitudinea funcţionării spaţiului virtual, aplicaţiile MMO trebuie să execute, la nivelul serverului, un număr foarte mare de operaţii pentru a simula o fizică realistă a spaţiului virtual (detecţia coliziunilor, forţe de frecare, forţa de gravitaţie, etc).
Acestea reprezintă gama de operaţii care sunt cele mai frecvent efectuate de către serverele spaţiului virtual şi astfel sunt principalele consumatoare de timp de calcul.
Spaţiul virtual trebuie să asigure corectitudinea interacţiunilor fizice între entităţile care îl populează şi astfel trebuie să execute foarte multe calcule de detecţie a coliziunilor. De obicei, la fiecare deplasare / actualizare a poziţiei unui caracter în spaţiul virtual trebuie asigurat faptul că acesta are voie să ajungă la poziţia respectivă şi că nu intră, de exemplu, în zidul unei clădiri.
Deoarece calculele de detecţie a coliziunilor sunt foarte costisitoare, în practică se limitează numărul entităţilor între care se efectuează aceste calcule (caracterele cu terenul, caracterele cu anumite cladiri, etc) şi sunt folosite de obicei modele de aproximare ale obiectelor asupra cărora se execută calculele.

4.3.2 IMPLEMENTAREA SIMULARILOR FIZICII FOLOSIND GPGPU

Aşa cum s-a precizat şi anterior, operaţiile cele mai costisitoare din punct de vedere al timpului de calcul care sunt executate de către serverele spaţiilor virtuale 3D MMO sunt cele de detecţie a coliziunilor. În practică, se utilizează pentru detecţia coliziunilor metode ce iau în consideraţie :

- fie poligoanele din care sunt alcătuite obiectele 3D
- fie o aproximare a obiectelor 3D folosind volume încadratoare

In mod uzual se folosește cea de-a doua metodă, deoarece prin simplificarea suprafeţelor pentru care trebuie facute calculele de coliziune se obţine un timp mai mic de execuţie.

În continuare se propune o modalitate de implementare a acestor tipuri de operaţii folosind tehnici GPGPU.

4.3.2.1 Volume încadratoare

În grafica pe calculator , un volum încadrator pentru un set de obiecte este un volum închis care conține toate obiectele mulţimii respective. Acesta este folosit pentru a îmbunătăţi eficienta operațiilor geometrice asupra unor forme complexe, folosind forme geometrice mai simple. Astfel de volume mai simple permit folosirea unor metode mai eficiente de testare a coliziunilor.

Volumele încadratoare cele mai folosite sunt:

- Sfere
- Elipsoizi
- Cilindri
- Paralelipipede

Sunt folosite următoarele metode de detecție a coliziunilor folosind volume încadratoare:

A) AABB

AABB (Axis Aligned Bounding Box) este un paralelipiped încadrator ale carui feţe sunt paralele cu axele sistemului de coordonate.

Pentru a determina dacă două AABB se intersectează, se testează separat pentru fiecare dintre axele x, y, z. Acest test separat pentru fiecare din cele 3 axe va determina coliziunea a două AABB chiar şi în cazul în care unul dintre ele îl conține pe celălalt.

Cele două AABB a căror intersecție este verificată, sunt specificate prin coordonatele centrului C şi un vector E, care reprezintă distanta de la centru la

fiecare faţă a paralelipipedului, rezultând 3 valori, corespunzătoare feţelor paralele cu axele x, y , z .

Se calculează pentru fiecare axă distanţa coordonatelor centrelor celor două AABB, obținându-se vectorul T.

T = AABB1.C – AABB2.C

Se compară distanţele corespunzătoare celor 3 axe cu suma distanţelor D ale celor 2 AABB. Astfel cele două AABB se intersectează atunci când:

|T.x| <= AABB1.E.x + AABB2.E.x &&

|T.y| <= AABB1.E.y + AABB2.E.y &&

|T.x| <= AABB1.E.z + AABB2.E.z

B) OBB

OBB (Oriented Bounding Box) sunt folosite pentru a încadra obiecte care sunt rotite şi pentru care folosirea AABB conduce la determinarea unui volum încadrator prea mare, detecția coliziunii nefiind foarte precisă.

Un OBB este specificat prin centrul acestuia C,o mulțime de axe ortonormale A0, A1, A2 şi extensiile a0, a1, a2 (a0 > 0, a1 > 0, a2 > 0).

Un avantaj al OBB constă în faptul ca acestea pot fi rotite odată cu obiectele, lucru care necesită însa reținerea orientării volumului.

Algoritmul folosit este foarte util deoarece încearcă minimizarea timpului necesar determinării intersecţiei a două obiecte.

Algoritmul foloseşte faptul că două poliedre convexe care nu se intersectează pot fi separate printr-un plan care, fie este paralel cu o faţă a unuia dintre poliedre, fie conține câte o muchie a fiecăruia dintre cele două poliedre.

Astfel, este necesar şi suficient pentru a determina dacă două poliedre se intersectează sau nu, să se examineze intersecția proiecțiilor poliedrelor pe linii perpendiculare pe planurile de separaţie.

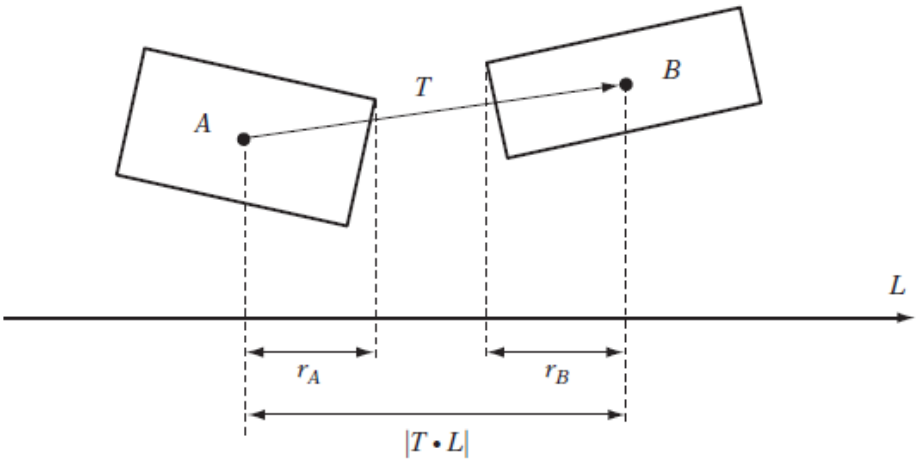

Figura 4-18: Intersecţie OBB

Două OBB nu se intersectează atunci când suma extensiilor proiectate este mai mică decât distanţa dintre proiecţiile centrelor:

$$|T \cdot L| > r_A + r_B$$

Pentru a determina corect starea de intersecţie, este necesar să se facă verificări pentru un număr total de 15 axe:

- 3 pentru axele de referinţă ale primului OBB
- 3 pentru axele de referinţă al celui de-al doilea OBB
- 9 pentru axele perpendiculare pe planul format de oricare două din cele 6 axe de referinţă ale celor 2 OBB

Dacă exista cel puţin o axă pentru care nu se realizează intersecţia, atunci obiectele nu se intersectează.

Numărul de operaţii poate fi redus prin exprimarea lui B în funcţie de coordonatele lui A. Pentru aceasta se utilizeaza vectorul t de translaţie de la A la B şi matricea de rotaţie R care aduce B în orientarea coordonatelor lui A.

C) Bounding-Sphere Collision Detection

O modalitate destul de simplă de încadrare a unui obiect într-un volum o constituie determinarea unei sfere care să conţină obiectul respectiv.

Determinarea coliziunii a două sfere se poate face prin calculul distanţei dintre centrele celor două sfere şi compararea rezultatului cu suma razelor celor două sfere.

Astfel, dacă suma celor două raze este mai mică decât distanţa calculată între centre, sferele nu se vor intersecta.

Dacă avem sferele reprezentate prin centrele C1(x1, y1, z1) şi C2 (x2, y2, z2) şi razele r1 şi r2:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

,iar pentru intersecţie este de ajuns să se compare d cu (r1 + r2)

4.3.2.2 Forţe implicate în simulările fizice

Numărul de forţe simulate în aplicaţiile grafice depind de gradul de realism pe care simularea dorește sa il obţină. Cu cât aplicaţia ţine cont de mai multe forţe din mecanică, cu atât aceasta se apropie mai mult de realitate. Cele mai importante forţe de care trebuie să se ţină cont sunt :

A) Forţa de frecare

Forţa de frecare reprezintă rezistenţa venită din partea mediului la mişcarea obiectului. Aceasta poate fi indusă de aer, un fluid sau de frecarea cu alte obiecte din scenă (la coliziuni). Forţa de frecare decelerează mişcarea corpului cu un anumit factor, proporţional cu un indice de frecare corespunzător materialului obiectului.

Forţa de frecare respectă formula:

$$F = \mu \cdot N$$

unde N este forţa de apăsare normală, iar μ este coeficientul de frecare

Exista 2 forme ale forţei de frecare:

- frecare statică
- frecare cinetică

Frecarea statică apare atunci când cele 2 obiecte nu se află în mişcare relativă, deoarece forţa de mişcare este mai mare decât forţa de tracţiune.

Frecarea cinetică apare atunci când cele 2 obiecte se află în mişcare relativă. Coeficientul de frecare este în acest caz notat μk şi este în general mai mic decât coeficientul de frecare statică.

B) Forţa de gravitaţie

Prin folosirea acestei forţe se obţine efectul de "cădere" al obiectelor lăsate liber în spaţiu. Constă în acceleraţia obiectului (de obicei) în direcţia negativă a axei verticale.

Forţa de gravitaţie respectă formula:

$$G = g \cdot m$$

unde m este masa obiectului şi g este constanta de accelerație gravitaționala, aproximativ egală cu 9.8

Un corp având iniţial viteza 0, lăsat liber, va accelera cu aproximativ $9.8\,m/s^2$

În cazul general, viteza corpului la timpul t, v(t) este:

$$v(t) = v(0) + g \cdot t$$

Distanţa parcursă de corp până la timpul t, d(t) este:

$$d(t) = v(0) \cdot t + g/2 \cdot t^2$$

Pentru a obține efecte realiste, forţa de atracție gravitațională trebuie folosită împreună cu forţa de frecare.

4.3.2.3 Algoritmi

Au fost implementate următoarele calcule de fizică atât pe GPU cât şi pe CPU (pentru comparaţie) :

- Detecţia coliziunilor între obiecte
- Procesarea forţelor de ciocnire în urma coliziunilor
- Calcule pentru forţa de gravitaţie

De asemenea, au fost folosite următoarele tipuri de obiecte asupra cărora au fost efectuate calculele de coliziune :

- Cuburi
- Sfere

- Cilindri

Aceste tipuri de obiecte reprezintă aproximările cele mai folosite în practică ca volume încadratoare a obiectelor 3D complexe pentru efectuarea calculelor de coliziune.

Algoritmul pentru lansarea calculelor de coliziune pe firele de execuţie CUDA este următorul :

- Datele iniţiale ale scenei sunt făcute disponibile în memoria globală a GPU-ului
- Setul de sarcini de coliziune este determinat de un vector unidimensional de dimensiune N*N (unde N este numărul de obiecte din scenă) deoarece se simulează coliziuni N-la-N între obiecte
- Acest vector se împarte în sub-vectori de dimensiune fixă ce au fiecare dimensiunea 32
- Fiecare sub-vector va fi procesat de un bloc de fire de execuţie unde fiecare fir de execuţie va fi responsabil de calculele executate pentru un element al sub-vectorului (obiectul identificat de firul de execuţie va fi testat pentru coliziuni cu toate celelalte obiecte ale scenei)
- Pentru implementarea *multi-GPGPU*, vectorul va fi împărţit la numărul de GPU-uri obţinându-se astfel pentru fiecare GPU o porţiune de care acesta este responsabil (de exemplu dacă se folosesc două GPU-uri, GPU0 va procesa prima jumătate a vectorului şi GPU1 va procesa cea de-a doua jumătate a vectorului)

Pentru implementarea *single-GPGPU* pseudocodul algoritmului este următorul :

```
Dimensiune bloc fire de execuţie = (32,1)
Dimensiune grid fire de execuţie =
( numarObiecteScena / dimensiuneXbloc,  1)

Copiază date în memoria globală GPU de pe maşina gazdă

Lansează în paralel pe GPU funcţie kernel(dimensiune
bloc,dimensiune grid)

Sincronizează prin barieră terminarea firelor de execuţie

Copiază date din memoria globală GPU pe maşina gazdă
```

Funcţia kernel ce se execută pe fiecare fir de execuţie de pe GPU este :

```
IndexObiect = IdBlocX * DimensiuneBlocX + IdFirDeExecuţieX

Pentru fiecare obiectJ din scena
    CalculeazăIntersectie(  obiectDinScena[IndexObiect],
                            obiectDinScena[obiectJ] )
```

Pentru implementarea *multi-GPGPU* pseudocodul algoritmului se modifică astfel :

```
Dimensiune bloc fire de execuţie = (32 , 1)
NrObiectePerGPU = numarObiecteScena / nrGPU
Dimensiune grid fire de execuţie =
( NrObiectePerGPU / dimensiuneXbloc ,  1)

Pentru fiecare GPU
{
   Copiază date în memoria globală GPU de pe maşina gazdă

   Lansează în paralel pe GPU funcţie kernel(dimensiune
   bloc,dimensiune grid, idGPU)

   Sincronizează prin barieră terminarea firelor de execuţie

   Copiază date din memoria globală GPU pe maşina gazdă
}
```

Funcţia kernel ce se execută pe fiecare fir de execuţie de pe GPU este :

```
Offset = NrObiectePerGPU * idGPU
IndexObiect = IdBlocX * DimensiuneBlocX + IdFirDeExecuţieX +
Offset

Pentru fiecare obiectJ din scena
   CalculeazăIntersectie( obiectDinScena[IndexObiect],
                          obiectDinScena[obiectJ] )
```

4.3.3 ANALIZĂ TIMP DE EXECUŢIE

Timpul de execuţie pentru efectuarea calculelor de coliziune în implementarea pe CPU este următorul :

*T(NrObiecte * NrObiecte)*,

unde,

NrObiecte – numărul de obiecte din scenă

Timpul de execuţie pentru efectuarea calculelor de coliziune în implementarea single GPGPU este următorul :

*T(((NrObiecte * NrObiecte) / (nrSM * nrSP))*

+ constantaOverhead,

unde,

NrObiecte – numărul de obiecte din scenă

nrSM – numărul de multiprocesoare de pe GPU

nrSP – numărul de procesoare scalare de pe fiecare multiprocesor

constantaOverhead – constantă introdusă de operaţiile de transfer de memorie şi sincronizare

Timpul de execuţie pentru efectuarea calculelor de coliziune în implementarea multi GPGPU este următorul :

*T(((NrObiecte * NrObiecte) / (nrSM * nrSP * nrGPU))*

*+ nrGPU*constantaOverhead*,

unde,

NrObiecte – numărul de obiecte din scenă

nrSM – numărul de multiprocesoare de pe GPU

nrSP – numărul de procesoare scalare de pe fiecare multiprocesor

nrGPU – numărul de GPU-uri disponibile

constantaOverhead – constantă introdusă de operaţiile de transfer de memorie şi sincronizare

Testele au fost efectuate pe următoarele 3 implementări :

- CPU : Intel Core2Quad Q6600
- Single GPGPU : 1 core al unui GPU GTX590
- Multi GPGPU : 4 core-uri GPU (2 GPU GTX590)

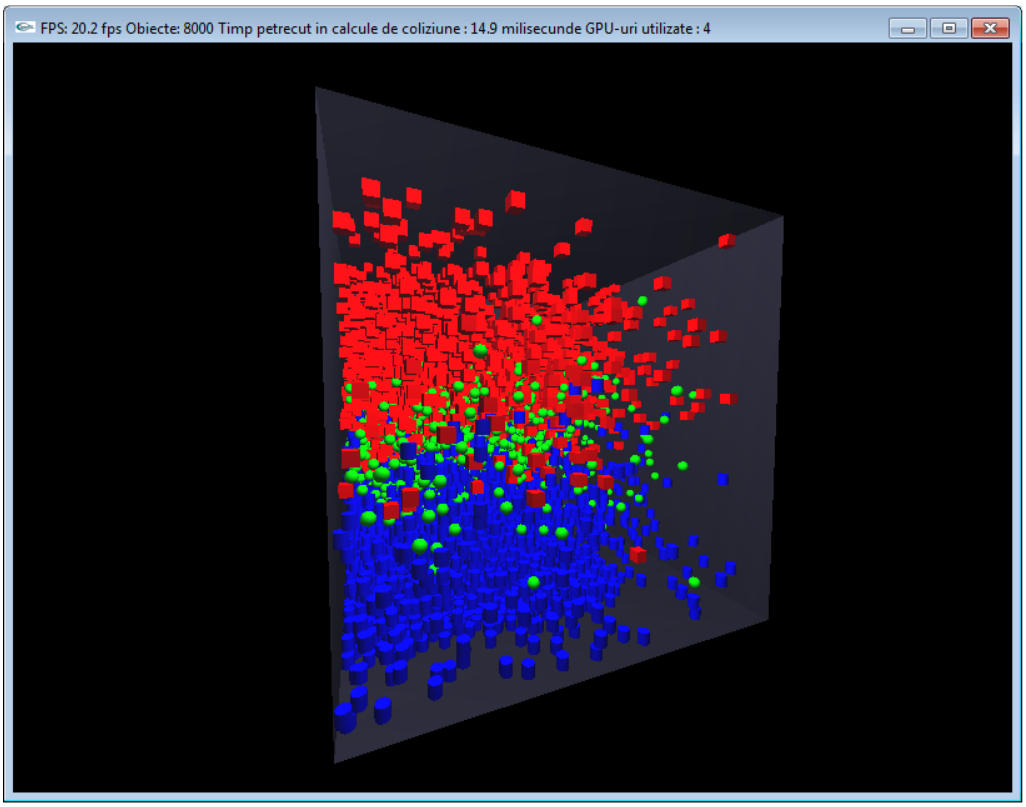

Figura 4-19: Simularea coliziunii a 8000 de obiecte folosind 4 core-uri GPU

Rezultatele obţinute sunt cele din tabelele şi graficele de mai jos.

	850 de obiecte		1750 de obiecte		4000 de obiecte	
	FPS	Timp calcule fizică / iteraţie	FPS	Timp calcule fizică / iteraţie	FPS	Timp calcule fizică / iteraţie
1 CPU Quad Core Q6600	130	16ms	24.7	67ms	2.9	332ms
Single GPGPU 1 core GPU GTX590	202	2ms	98	3.8ms	44.6	7ms
Multi GPGPU 4 cores GPU GTX590	200	2.3ms	96	4ms	44	7.2ms

Tabel 4-2: Rezultate implementare calcule fizică – Partea I

	8000 de obiecte		16000 de obiecte		32000 de obiecte	
	FPS	Timp calcule fizică / iteraţie	FPS	Timp calcule fizică / iteraţie	FPS	Timp calcule fizică / iteraţie
1 CPU Quad Core Q6600	0.7	1314ms	N/A	N/A	N/A	N/A
Single GPGPU 1 core GPU GTX590	21.1	18ms	8	79ms	2.4	310ms
Multi GPGPU 4 cores GPU GTX590	22.4	15ms	11.6	30ms	5.4	80ms

Tabel 4-3: Rezutate implementare calcule fizică – Partea a II-a

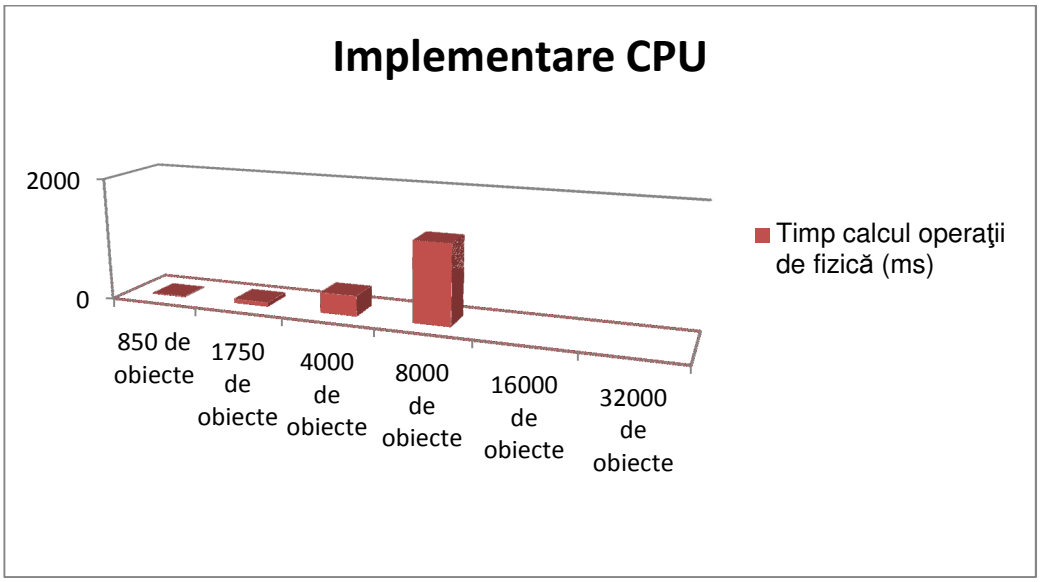

Figura 4-20: Calcule de fizică pe CPU – Timpi de calcul

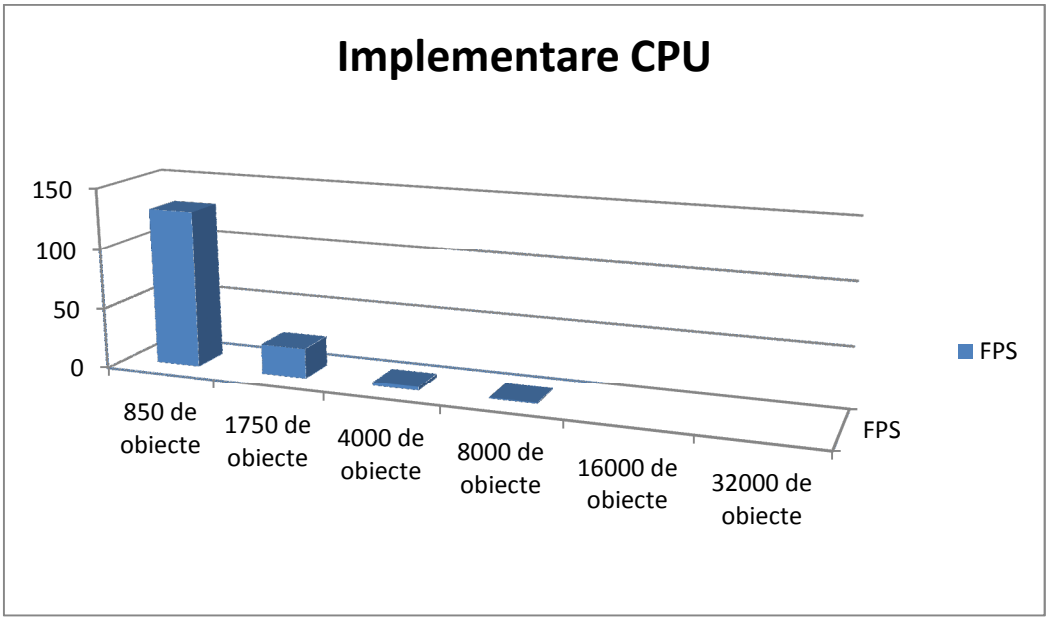

Figura 4-21: Calcule de fizică pe CPU – FPS

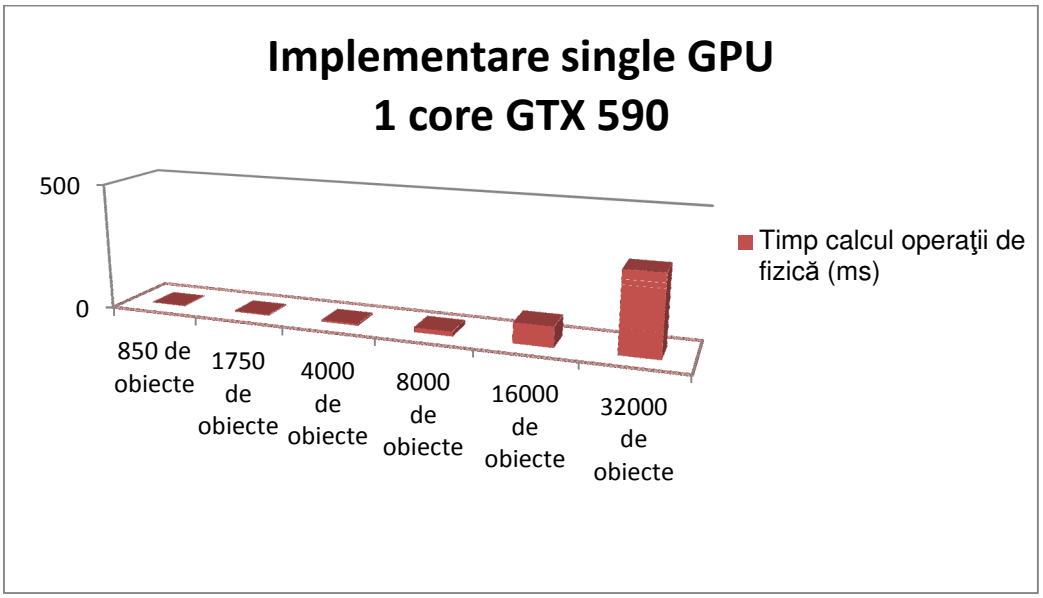

Figura 4-22: Calcule de fizică pe 1 GPU – Timpi de calcul

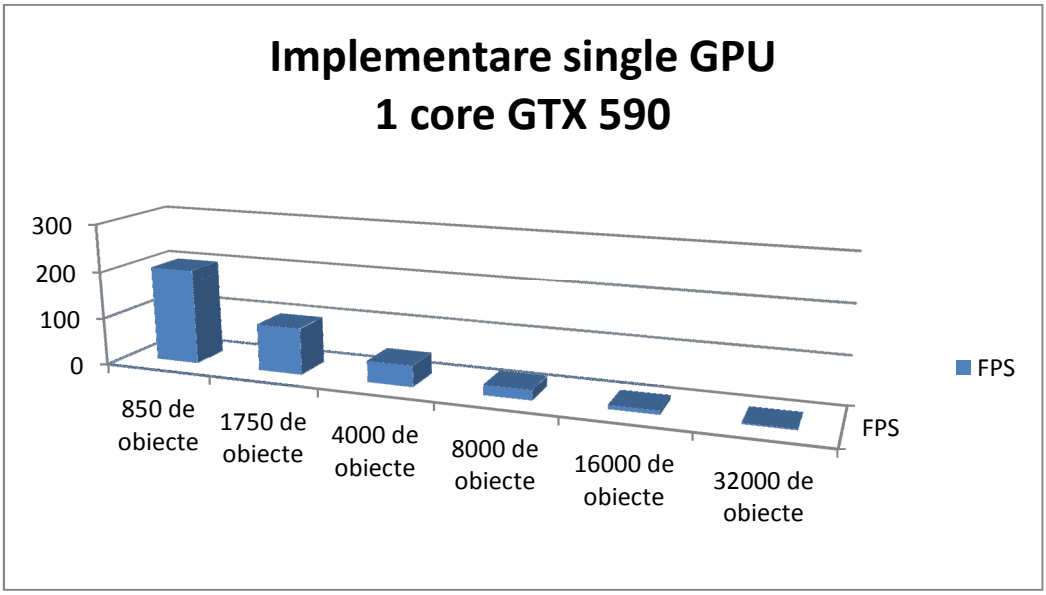

Figura 4-23: Calcule de fizică pe 1 GPU – FPS

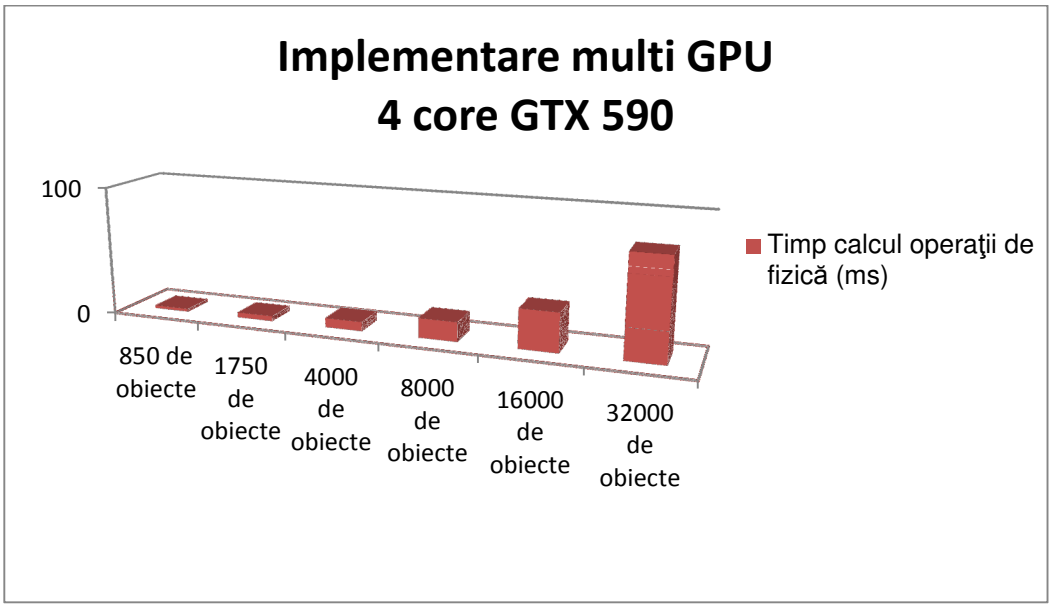

Figura 4-24: Calcule de fizică pe 4 GPU – Timpi de calcul

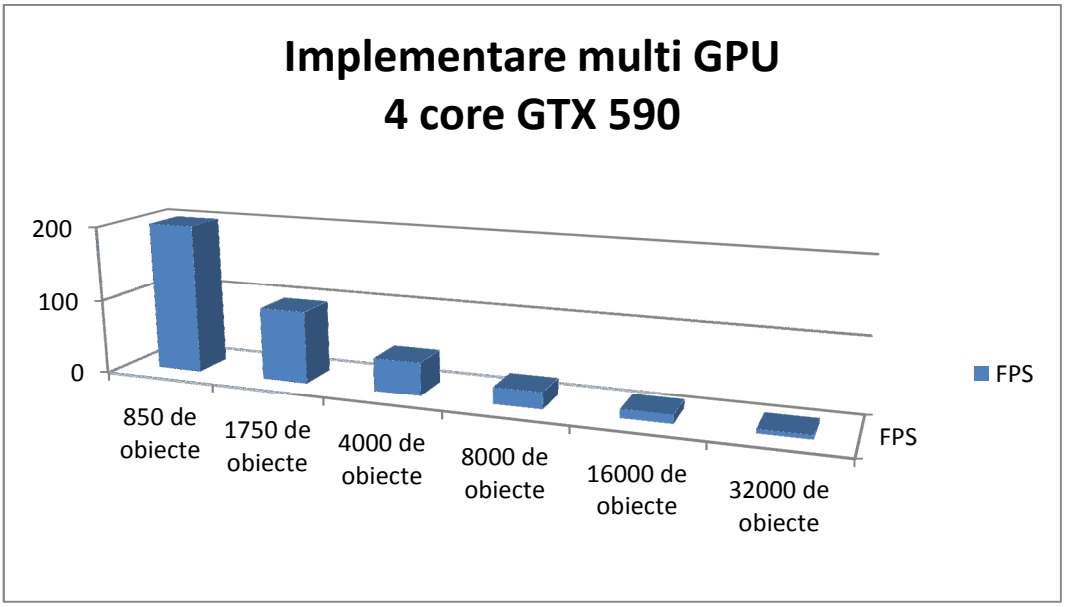

Figura 4-25: Calcule de fizică pe 4 GPU – FPS

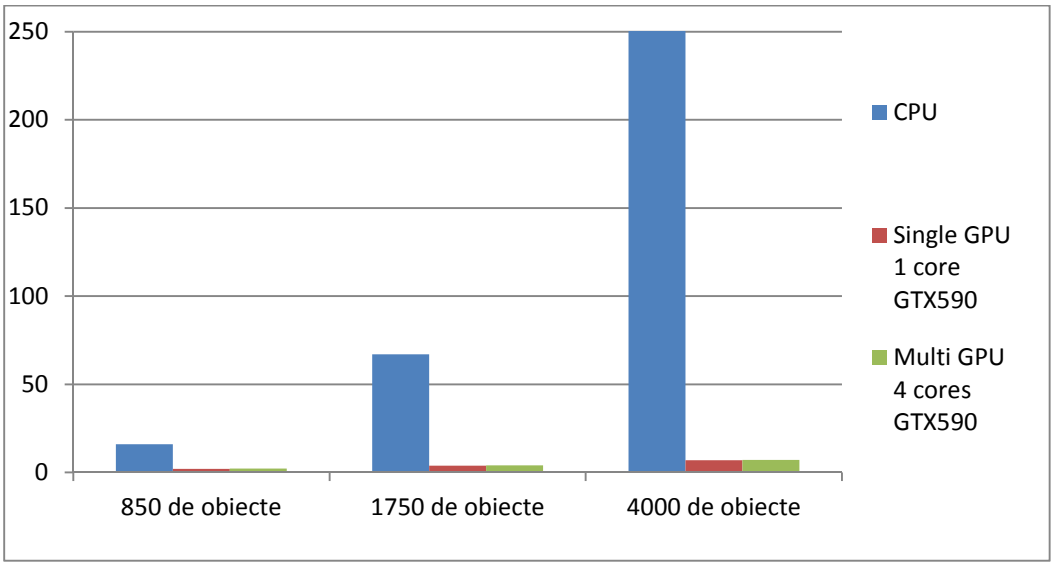

Figura 4-26: Comparaţie timpi de calcul implementări operaţii de fizică executate în spaţiile virtuale – Partea I

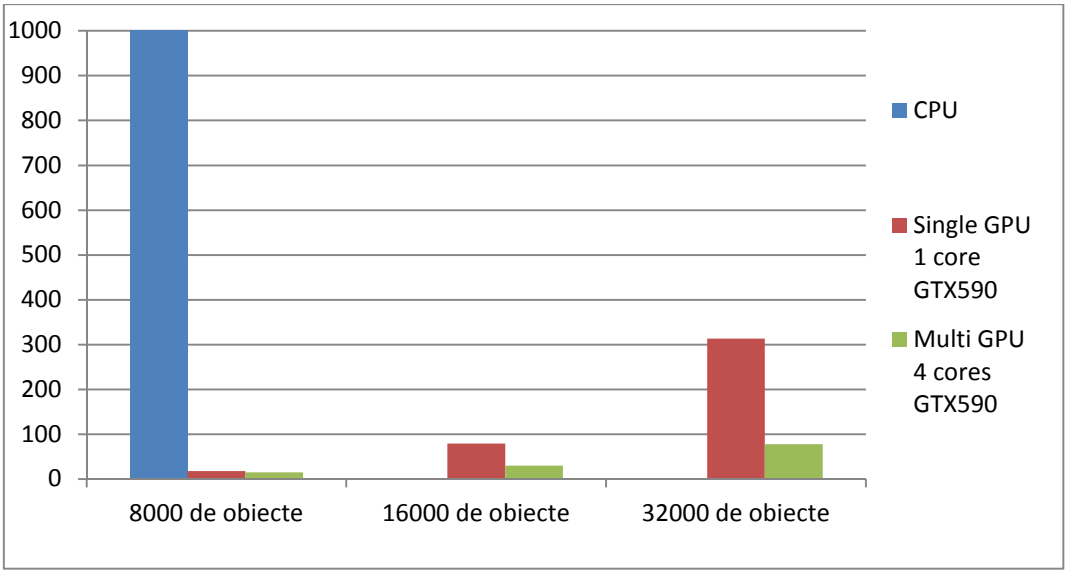

Figura 4-27: Comparaţie timpi de calcul implementări operaţii de fizică executate în spaţiile virtuale – Partea a II-a

Din analiza rezultatelor de mai sus se pot trage următoarele concluzii :

- Pe implementarea CPU se poate observa faptul că timpul necesar pentru calculele de fizică creşte rapid pe măsură ce creşte şi numărul de obiecte ce sunt implicate în simulare; pentru 4000 de obiecte în scenă, CPU deja nu mai face faţă, costul de timp fiind peste 300 de ms / iteraţie; de asemenea pentru 16000, respectiv 32000 de obiecte în scenă nu au putut fi obţinute rezultate deoarece numărul de obiecte este prea mare pentru a putea fi procesat într-un timp rezonabil de CPU; având în vedere rezultatele de mai sus se poate deduce faptul că procesarea pe CPU a calculelor de fizică nu este scalabilă cu numărul de obiecte din scenă.

- Implementările GPGPU (single şi multi) scalează bine o dată cu creşterea numărului de obiecte din scenă, comparativ cu implementarea pe CPU realizându-se sporuri de performanţă semnificative; astfel pentru 1750 de obiecte în scenă se obţine un *speedup* (spor de performanţă) de aproximativ 17x, pentru 4000 de obiecte în scenă se obţine un speedup de aproximativ 47x iar pentru 8000 de obiecte în scenă se obţine un speedup de aproximativ 90x; de asemenea, se poate observa faptul că inclusiv atunci când există un număr foarte mare de obiecte în scenă (16000 respectiv 32000) timpii obţinuţi sunt acceptabili.

- Între implementarea single GPU (care foloseşte un singur core GPU GTX590) şi implementarea multi GPU (care foloseşte 4 core GPU GTX590) diferenţele de spor de performanţă sunt mici pentru un număr relativ mic de obiecte din scenă, mai mult, implementarea single GPU fiind chiar un pic mai performantă decât cea multi GPU; acest lucru se întâmplă din cauza overhead-ului din implementarea multi GPU ce apare în principal datorită operaţiilor de sincronizare şi de transfer de memorie (gazdă-dispozitiv GPU şi dispozitiv GPU-gazdă) ce trebuie efectuate pentru toate dispozitivele GPU din sistem; adevăratul spor de performanţă se vede atunci când numărul de obiecte din scenă creşte semnificativ, astfel pentru 16000 de obiecte în scenă avem un speedup de 2.66x pentru implementarea multi GPU faţă de cea single GPU, iar pentru 32000 de obiecte din scenă avem un speedup de 3.87x pentru implementarea multi GPU faţă de cea single GPU; se poate observa că pentru un număr mare de obiecte, implementarea multi GPU ajunge aproape de *speedup-ul* maxim teoretic (4x) faţă de cea *single-GPU*

- FPS-ul obţinut este puternic dependent de capabilităţile de afişare grafică ale GPU-ului; astfel, pentru un număr mare de obiecte (mai mare de 4000) nu mai este direct dependent de timpii pentru calculele de fizică ci doar de numărul mare de obiecte ce trebuie afişate.

4.4 CONCLUZII

GPU modern a devenit un hardware foarte avansat în domeniul arhitecturilor de calcul paralele many-core şi se diferenţiază faţă de arhitectura clasică CPU prin utilizarea la scară largă a multithreading-ului hardware şi a procesării SIMD – Single Instruction Multiple Data. În acest fel, se acordă prioritate operaţiilor ce se pot executa în paralel asupra unor cantităţi mari de date faţă de execuţia operaţiilor *single task* de latenţă scăzută.

GPGPU (General Purpose on Graphical Processing Units) reprezintă o metodă prin care unitatea de procesare grafică poate fi utilizată pentru a executa calcule/programe de uz general. Acest lucru a devenit posibil odată cu apariţia etapelor programabile ale GPU precum şi datorită existenţei bibliotecilor şi instrumentelor de dezvoltare puse la dispoziţie de producătorii de unităţi de procesare grafică.

Evoluţia puterii de calcul a GPU moderne a înregistrat în ultima perioadă creşteri spectaculoase şi sunt demne de menţionat următoarele aspecte :

- viteza de dezvoltare a arhitecturilor GPU este una ridicată, de obicei la fiecare 12-18 luni fiind disponibile îmbunătăţiri spre deosebire de ciclul de dezvoltare al CPU unde modificările la nivel arhitectural au loc la aproximativ 2 ani;
- soluţiile curente high-end de *procesoare* GPU deja au o putere de calcul de câteva zeci de ori mai mare decât cea a CPU atunci când sunt folosite pentru calcule asupra datelor ce se pretează la procesare SIMD.

Pentru a crea un grad de imersiune cât mai ridicat pentru utilizatorii spaţiului virtual, la nivelul aplicaţiei client se poate folosi ca metodă de redare algoritmul RayTracing.

Până de curând, utilizarea RayTracing ca metodă de redare în timp real nu era viabilă deoarece costul computaţional al metodei este unul ridicat iar hardware-ul existent nu era suficient de rapid pentru a obţine rezultate în timp real.

O data cu lansarea la finalul anului 2006 a arhitecturilor unificate pentru GPU, a apărut posibilitatea pentru o gamă largă de operaţii de uz general sa poată fi implementate folosind tehnici GPGPU.

În acest capitol au fost propuse soluţii pentru a executa algoritmul RayTracing folosind procesare *single GPGPU* şi *multi-GPGPU* şi au fost prezentate rezultatele obţinute. Este important de menţionat că la timpul realizării cercetării (2008 şi 2009) încercările de exploatare a paralelismului de tip GPGPU pentru implementarea algoritmului RayTracing se aflau într-o stare incipientă, astfel încât cercetarea realizată în acel moment a fost una de actualitate.

Soluţiile şi rezultatele obţinute au fost publicate într-o serie de articole (*[AMM09a] , [AMM09c]*).

La nivelul serverelor de spaţii virtuale 3D MMO se execută un număr foarte mare de operaţii pentru a simula o fizică realistă a spaţiului virtual (detecţia coliziunilor, forţe de frecare, forţa de gravitaţie, etc).

Acestea reprezintă gama de operaţii care sunt cele mai frecvent efectuate de către serverele spaţiului virtual şi astfel sunt principalele consumatoare de timp de calcul.

În acest capitol a fost propusă o soluţie de paralelizare folosind procesare single GPGPU şi multi GPGPU a calculelor de detecţie a coliziunilor ce sunt executate în mod obişnuit în cadrul spaţiilor virtuale.

Rezultatele obţinute au scos în evidenţă următoarele aspecte majore :

- implementarea folosind CPU nu mai face faţă în timp real pentru un număr relativ mediu de obiecte existente în scenă; se poate deduce astfel că procesarea pe CPU a calculelor de fizică nu este scalabilă cu numărul de obiecte din scenă;

- implementarea ce foloseşte *single GPGPU* obţine sporuri de performanţă de ordinul zecilor faţă de CPU;
- implementarea ce foloseşte *multi GPGPU* obţine sporuri de performanţă de ordinul sutelor faţă de CPU;
- diferenţele de spor de performanţă între implementarea *single GPU* şi implementarea *multi GPU* sunt mici pentru un număr relativ mic de obiecte din scenă, datorită overhead-ului introdus de operaţiile de sincronizare şi de transfer de memorie (gazdă-dispozitiv şi dispozitiv-gazdă) ce trebuie efectuate pentru a gestiona mai multe GPU în sistem; adevăratul spor de performanţă se vede când numărul de obiecte din scenă creşte semnificativ, sporul de performanţă crescând simţitor.

O parte din ideile utilizate pentru paralelizarea operaţiilor de detecţie a coliziunilor prin tehnici GPGPU au fost prezentate într-o serie de articole ((*[AMM10a]* , *[AMM11]*).

5 ARHITECTURĂ ORIGINALĂ PENTRU SERVERE 3D MMO UTILIZÂND PARALELISMUL GPGPU

Una dintre principalele probleme cu care se confruntă dezvoltatorii de spaţii virtuale 3D MMO este cea a încărcării foarte mare la care trebuie sa facă faţă lumea virtuală. Aceasta încărcare este dată atât de numărul de sarcini, care este direct proporţional cu numărul de utilizatori on-line, cât şi de complexitatea acestor sarcini care depinde de particularităţile lumii virtuale simulate.

5.1 NECESITATEA MODIFICĂRII ARHITECTURII CLASICE

Deşi anumite cerinţe necesare pentru a accesa o aplicaţie MMO în condiţii optime, cum ar fi o lăţime de bandă mare, latenţă scăzută, devin din ce în ce mai accesibile utilizatorului de rând, acest lucru nu este suficient pentru a rezolva problemele de scalabilitate cu care se confruntă aplicaţiile MMO în incercarea lor de a suporta cât mai mulţi utilizatori.

Aceste probleme de scalabilitate apar în mare măsură datorită necesităţii de a menţine consistenţa lumii virtuale. Încălcarea cerinţelor de consistenţă poate să ducă, de exemplu, la apariţia de artefacte vizuale (de exemplu, intrarea unui utilizator într-un zid al unei clădiri) care nu au consecinţe pe termen lung, dar pot cu uşurinţă să apară probleme mult mai serioase cum ar fi, de exemplu, pierderea sau duplicarea de obiecte în urma unei tranzacţii între utilizatorii spaţiului virtual. Astfel, pe lângă scăderea realismului, pot să apară probleme importante din punct de vedere al securităţii.

În mod tradiţional, marea majoritate a aplicaţiilor MMO sunt implementate folosind o arhitectură de tip client-server care oferă următoarele avantaje:

- Control centralizat
- Securitate crescuta
- Implementare relativ simplă

Deşi această arhitectură, care este foarte răspândită, este potrivită pentru multe tipuri de aplicaţii distribuite, introduce şi o serie de dezavantaje cele mai importante fiind :

1) **Scalabilitatea** – Performanţa sistemului central de servere care simulează lumea virtuală reprezintă o sugrumare ajungându-se astfel la o limită a numarului total de utilizatori care pot accesa simultan lumea virtuală.

2) **Redundanţa** – Pentru a asigura faptul ca un server este suficient de puternic astfel încât sa facă faţă momentelor de utilizare maximă, este necesară asigurarea unui grad ridicat de redundanţă hardware pentru echipamentul respectiv

3) **Fiabilitatea –** Acest tip de arhitectură nu are o fiabilitate ridicată deoarece serverele reprezintă posibile puncte de cădere

4) **Costul** – În general, dezvoltarea unei aplicaţii MMO durează între 2-3 ani şi costul de dezvoltare pentru a lansa o aplicație MMO de calitate medie este unul ridicat. Un alt aspect de luat în considerație este faptul ca aplicaţiile MMO trebuie sa simuleze lumi virtuale de întindere foarte mare şi acest lucru face necesară existenţa unui număr ridicat de servere, uneori de ordinul sutelor, pentru a găzdui în mod complet mediul virtual. Se poate ajunge astfel ca dupa lansarea unei aplicaţii MMO, costurile de întreținere să ajunga chiar şi până la 80% din totalul încasărilor.

Ţinând astfel cont de aspectele menționate anterior, apare necesitatea explorării de noi soluţii privind arhitecturile pentru servere de spaţii virtuale 3D MMO şi de asemenea de a găsi modalităţi de optimizare a operațiilor efectuate în cadrul simulării unui lumi virtuale pentru a elimina sau măcar reduce o parte din problemele curente cu care se confruntă aplicaţiile MMO.

5.2 PREZENTAREA MODIFICĂRILOR

După cum s-a menţionat, principala problemă a arhitecturilor curente pentru servere de spaţii virtuale 3D MMO este cea a scalabilităţii acestora atunci când trebuie sa facă faţă la un număr mare de utilizatori simultan deoarece aceştia generează un volum ridicat de operaţii ce trebuie executate pe partea de server.

Modificările propuse în continuare pentru a fi aplicate în cadrul arhitecturilor serverelor spaţiilor virtuale încearcă să adreseze această problemă din două direcţii:

- introducerea în arhitectura serverelor a capabilităţilor de procesare a sarcinilor de calcul folosind tehnici GPGPU;
- propunerea unui mecanism original de alocare a resurselor necesare execuţiei calculelor ţinându-se cont de o serie de factori ce pot influenţa alocarea; deoarece această alocare va fi strâns legată de tehnica de zonare ce este prezentă în cadrul tuturor tipurilor de arhitecturi, ne vom referi în continuare la acest mecanism ca reprezentând modificări aduse „operaţiilor de zonare" prezente în cadrul arhitecturilor serverelor de spaţii virtuale 3D MMO.

Cele două direcţii abordate sunt strâns legate între ele, modificările propuse la mecanismul de alocare al resurselor fiind făcute plecându-se de la premiza că în cadrul arhitecturii vor exista capabilităţi de procesare masiv paralelă folosindu-se tehnici GPGPU.

5.3 DETALIEREA OPERAŢIILOR DE ZONARE

Zonarea este unul dintre mecanismele foarte importante ale unei arhitecturi de spaţii virtuale MMO. La nivelul mecanismului de zonare se execută foarte multe operaţii ce tin de acomodarea utilizatorilor în aria geografică a spaţiului virtual deservită de o anumită zonă logică.

De asemenea, un număr important de operaţii se execută pentru a asigura persistenţa spaţiului virtual, fiind necesară o sincronizare perfectă între zonele adiacente / cu zonele de tranziţie, existând astfel un flux crescut de informaţii.

De exemplu, când un utilizator trece dintr-o zona în alta care este adiacentă, este de dorit ca acest lucru să se facă într-un mod transparent pentru acesta şi chiar dacă operațiile de transfer ce se efectuează între cele două zone sunt numeroase, acestea sa nu fie vizibile într-un anumit fel utilizatorului (ecrane de încărcare, timpi de răspuns , etc).

Aşa cum s-a detaliat în capitolul 3, se folosesc în practică două abordări pentru a face zonarea unui spaţiu virtual de o întindere geografică mare :

- cea a zonării simple, unde trecerea utilizatorului dintr-o zonă în alta este vizibilă acestuia (ecran de încărcare, obiectele dintr-o zonă nu sunt vizibile direct în zonele vecine, etc) realismul simulării având astfel de suferit;
- cea care realizează împărțirea pe zone logice într-un mod transparent pentru utilizator, prin utilizarea unor zone suplimentare speciale, denumite zone de tranziţie, obținându-se astfel o simulare „seamless", fără întreruperi cauzate de trecerea dintr-o zonă în alta, a spaţiului virtual.

Indiferent de varianta de zonare folosită, există următorul dezavantaj prezent în implementările actuale : determinarea zonelor şi a resurselor alocate pentru acestea nu se face dinamic, existând o configuraţie fixă sau care ţine cont de foarte puţini parametri, aceste setări fiind stabilite încă de la începutul intrării în execuţie a părţii de server a aplicaţiei MMO.

Astfel, indiferent de necesarul de resurse pentru o anumită zonă, există la nivel global o singură alocare bine stabilită pentru acea zonă.

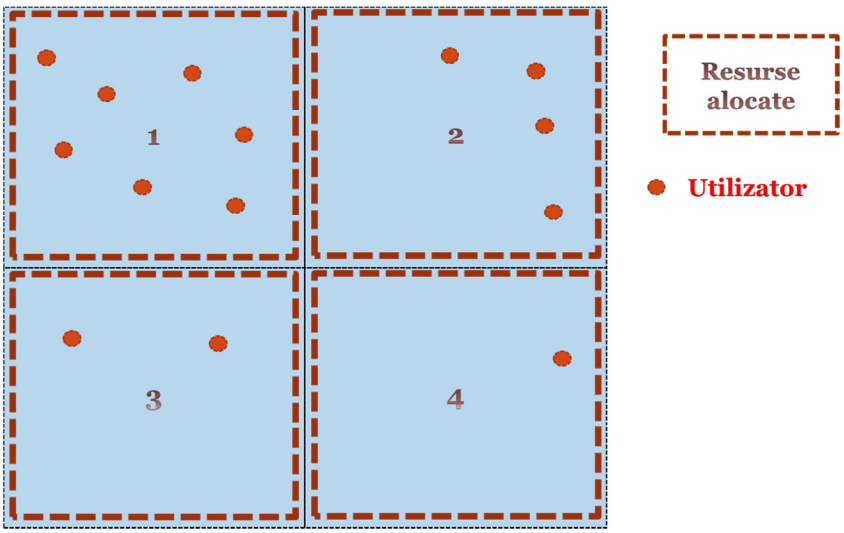

Figura 5-1: Alocare statică a resurselor

Modificările prezentate în continuare pentru operaţiile de zonare vizează optimizarea alocării resurselor, ţinându-se cont de următorii 3 factori :

- poziţionarea spaţială a utilizatorilor care sunt on-line în cadrul spaţiului;
- un cost de „greutate a operaţiilor efectuate” asociat fiecărui utilizator care este on-line în cadrul spaţiului virtual;
- tipurile de entităţi pentru care trebuie realizate calcule.

5.3.1 OPTIMIZARE CE ŢINE CONT DE POZIŢIONAREA SPAŢIALĂ ACTUALĂ A UTILIZATORILOR

O primă optimizare este dată de faptul că la alocarea resurselor se va ţine cont de poziţionarea spaţială a utilizatorilor din cadrul spaţiului virtual.

Astfel, dacă într-o zona geografică se află la un moment dat un anumit număr de utilizatori, alocarea resurselor pentru acea zonă poate fi ajustată dinamic (crescută sau scăzută) în funcţie de numărul acestora.

În acest fel, nu se vor aloca resurse în mod inutil pentru zone unde nu există un număr mare de calcule de executat (din cauza lipsei sau a numărului mic de utilizatori ce pot genera operaţii de procesat), evitându-se astfel fenomenul de „înfometare” iar resursele disponibile vor putea fi utilizate în alte zone unde este nevoie de resurse suplimentare.

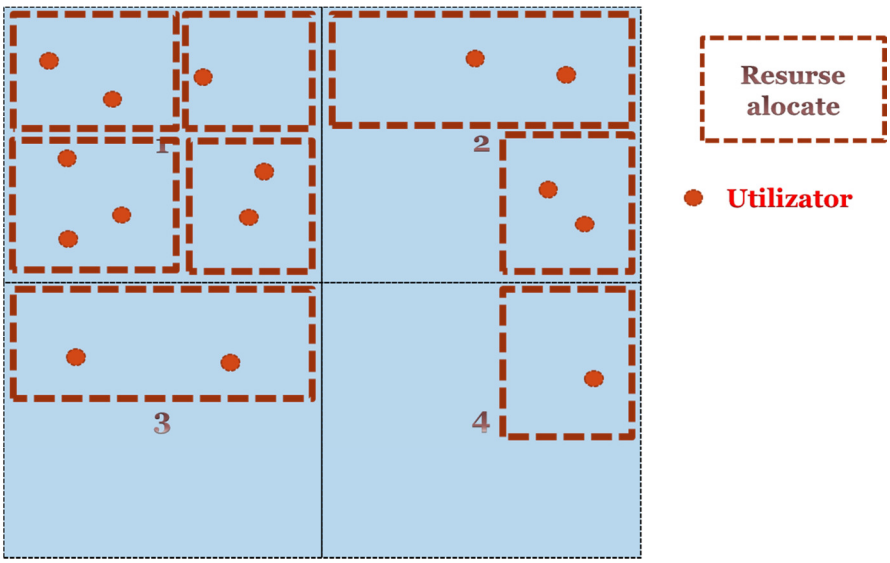

Figura 5-2: Alocare dinamică a resurselor ce ţine cont de poziţionarea spaţială a utilizatorilor

5.3.2 OPTIMIZARE CE ŢINE CONT DE COSTUL OPERAŢIILOR EFECTUATE DE FIECARE UTILIZATOR

O altă optimizare propusă ţine cont de tipul de operaţii efectuate de un utilizator şi de frecvenţa cu care acesta generează aceste operaţii.

De exemplu, dacă un utilizator nu se mişcă foarte des în spaţiul virtual (este plecat din faţa tastaturii, discută pe chat cu alţi utilizatori, etc) el nu va genera operaţii ce sunt intensiv computaţionale (cum ar fi cele de detecţie a coliziunilor). Cazul opus celui prezentat anterior este acela în care utilizatorul execută foarte multe acţiuni ce presupun interacţiuni cu spaţiul virtual (se află în tipul unui *quest*, lupte, etc) şi atunci operaţiile ce vor trebui executate de către server pentru gestiunea acestuia vor avea un cost computaţional mare.

Prin execuţia de operaţii de *profiling* în timp asupra acţiunilor unui utilizator din spaţiul virtual, se poate astfel determina un cost de „greutate a operaţiilor efectuate" asociat acestuia şi în funcţie de acest cost se poate schimba categoria din care face parte utilizatorul (utilizator care generează operaţii care nu sunt intensiv computaţionale, utilizator care generează operaţii care sunt intensiv computaţionale, etc).

În funcţie de apartenenţa la diversele categorii existente se poate face o alocare a resurselor la un nivel mult mai fin, astfel încât să existe o mai bună utilizare a acestora.

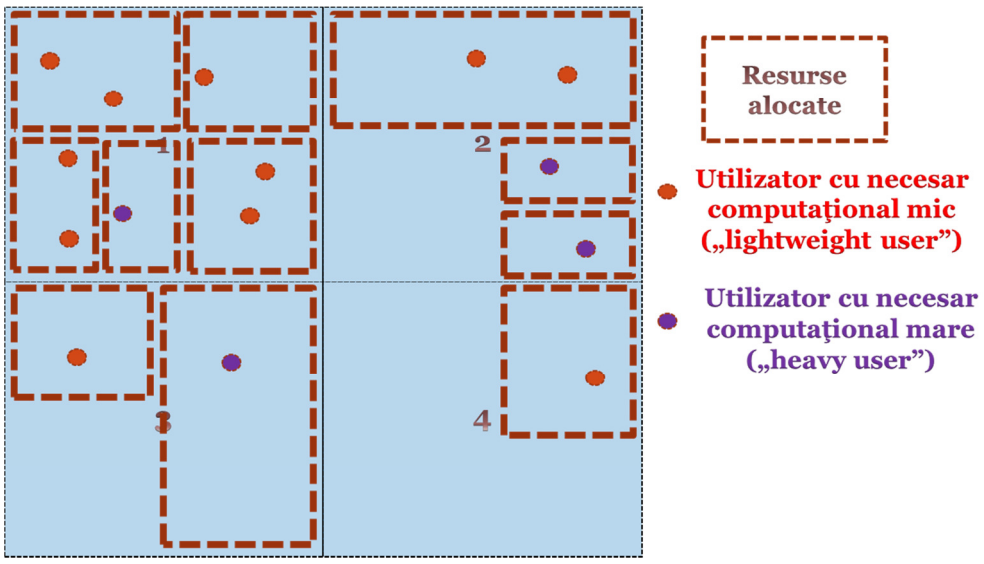

Figura 5-3: Alocare dinamică a resurselor ce ţine cont de un cost asociat utilizatorilor

5.3.3 OPTIMIZARE CE ŢINE CONT DE TIPUL ENTITĂŢILOR PENTRU CARE SE EFECTUEAZĂ CALCULE

Ultima optimizare propusă vizează o alocare a resurselor care sa ţină cont şi de tipul de entitate pentru care se vor efectua calcule. Mai exact, se va face o diferenţiere între următoarele tipuri de entităţi ce populează spaţiul virtual :

- utilizatori umani
- utilizatori controlaţi de server : NPC (non player characters)

Într-un spaţiu virtual 3D MMO, cantitatea totală de NPC-uri raportată la numărul de utilizatori umani poate fi mai mare de zeci, chiar sute de ori.

De asemenea, NPC-urile fiind controlate de către modulul de inteligenţă artificială a serverului, au un alt comportament decât cel al unui utilizator normal astfel şi operaţiile generate de acestea sunt diferite faţă de cele ale unui utilizator uman.

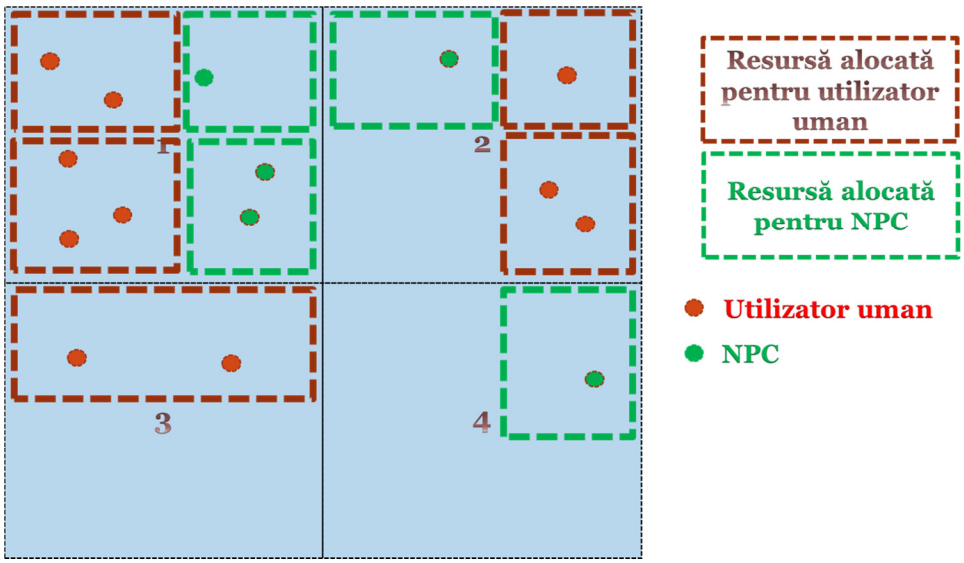

Figura 5-4: Alocare dinamică a resurselor ce ţine cont de tipul entităţilor

5.4 ABSTRACTIZAREA MATEMATICĂ A OPTIMIZĂRILOR PROPUSE

Pentru a putea folosi programatic în cadrul soluţiei arhitecturale optimizările propuse mai sus, se va folosi pentru determinarea unui cost asociat fiecărui task computaţional declanşat de către un utilizator, următoarea funcţie de cost :

*Cost(task,utilizator) = PozitionareSpatiala(task,utilizator) * pondereSpatiala +*
*ProfilingOperatii(task,utilizator) * pondereOperatii +*
*TipEntitate(task,utilizator) * pondereTipEntitate*

unde,

PozitionareSpatiala(task,utilizator), ProfilingOperatii(task,utilizator) şi *TipEntitate(task,utilizator)* sunt funcţii asociate optimizărilor prezentate anterior şi vor întoarce un cost determinat de acestea

iar,

pondereSpatiala, pondereOperatii şi *pondereTipEntitate* sunt ponderi asociate costurilor determinate de cele 3 funcţii care vor determina contribuţia acestora la costul total al task-ului

Pseudocodul simplificat pentru funcţia *PozitionareSpatiala(task,utilizator)* este următorul :

```
zonaUtilizator = determinaZonaGeografica(Utilizator)

numarUtilizatoriZona =
         determinaNumarUtilizatoriZonali(zonaUtilizator)

gradIncarcare =
   determinaGradIncarcareSpatial(zona, numarUtilizatoriZona)

Întoarce gradIncarcare
```

Funcţia *determinaGradIncarcare*, va întoarce valori asociate unor plaje discrete care sunt calculate în raport cu anumiţi parametri ai spaţiului virtual. De exemplu, ţinând cont de numărul total al utilizatorilor din spaţiul virtual şi de valorile primite ca parametrii ai funcţiei (zona şi numărul de utilizatori prezenţi în aceasta) va întoarce valori asociate următoarelor grade de încărcare a zonei :

- încărcare mică
- încărcare medie
- încărcare mare

Pseudocodul simplificat pentru funcţia *ProfilingOperatii (task,utilizator)* este următorul :

```
numarOperatiiUtilizator =
   profiligNumarOperatii (Utilizator, interval de timp)

tipOperatiiUtilizator =
    profiligTipOperatii (Utilizator, interval de timp)

gradIncarcare =
    determinaGradIncarcareOperatii
               (     Utilizator,
                     profiligNumarOperatii,
                     profiligTipOperatii )

Întoarce gradIncarcare
```

Funcţia *determinaGradIncarcareOperatii*, va întoarce valori asociate unor plaje discrete care sunt calculate în raport cu operaţiile si tipul acestora executate de către utilizator într-un interval de timp. De exemplu, va întoarce valori asociate următoarelor categorii de utilizatori :

- utilizator cu necesar computaţional redus
- utilizator cu necesar computaţional mediu
- utilizator cu necesar computaţional ridicat

Pseudocodul simplificat pentru funcţia *TipEntitate (task,utilizator)* este următorul :

```
tipEntitate =
   determinaTipEntitate (Utilizator)

gradIncarcare =
    determinaGradIncarcareEntitate(Utilizator,tipEntitate)

Întoarce gradIncarcare
```

Funcţia *determinaGradIncarcareEntitate*, va întoarce valori asociate tipului de entitate (uman sau NPC) a utilizatorului care a declanşat taskul. Astfel, pentru o

entitate de tip NPC, aceste valori pot depinde de o serie de parametri : numărul acestora comparativ cu cel al utilizatorilor umani, comportamentul acestora, etc.

Toţi parametrii folosiţi pentru determinarea diferitelor grade de încărcarea vor putea fi ajustaţi în cadrul implementării soluţiei arhitecturale pentru a realiza o calibrare cât mai fin posibilă a funcţiei globale de cost asociate unui task computaţional.

5.5 PREZENTAREA SOLUTIEI ARHITECTURALE COMPLETE

Pentru a putea introduce posibilitatea de execuţie de operaţii GPGPU în cadrul arhitecturilor pentru servere de spaţii virtuale 3D MMO a fost necesară introducerea/modificarea şi implementarea următoarelor componente :

- componenta de alocare a clienţilor şi resurselor
- componenta pentru creare / planificare sarcini :
    - global la nivel de server
    - specializat la nivel de GPU
        - *single GPGPU*
        - *multi GPGPU*
- componenta pentru procesarea şi propagarea rezultatului operaţiilor CUDA

După cum se poate observa, modificările pentru a introduce capabilităţi GPGPU sunt în legătură strânsă cu modalitatea de alocare a clienţilor şi resurselor prezentată anterior. Totodată, modificările aduse arhitecturii sunt gândite astfel încât sa fie posibilă scalarea la nivel multi GPU.

Soluţia prezentată în continuare are drept scop obținerea unui nivel ridicat de scalabilitate, atât vertical cât şi orizontal, precum şi reducerea limitărilor arhitecturilor tradiţionale pentru servere de spaţii virtuale 3D MMO prin implementarea unei arhitecturi cu următoarele caracteristici importante :

- descarcă operaţiile intensiv computaţionale din sarcina CPU şi le implementează ca operaţii GPGPU;

- este proiectată sa fie masiv paralelă prin suportul planificării operaţiilor la nivel de multi GPGPU;
- foloseşte un model care este condus de evenimente şi sarcini, fiind centrat în principal pe locaţia, tipul şi acţiunile executate de entităţile utilizator în cadrul spaţiului virtual, deoarece ei sunt consumatorii puterii de calcul la nivel de server şi nu datele spaţiului virtual în sine.

Pentru implementarea operaţiilor GPGPU a fost folosit hardware grafic de la NVidia şi toolkit-ul de dezvoltare CUDA (*[CUD11]*). CUDA este o platformă hardware şi software pentru executarea operaţiilor de uz general, care în mod normal sunt executate de CPU, direct pe GPU fără a fi nevoie să se utilizeze un API grafic.

Următoarele operaţii vor fi implementate ca operaţii GPGPU :

- Operaţii pentru simularea fizicii spaţiului virtual

Pentru a face posibilă execuţia de operaţii GPGPU la nivelul arhitecturii serverelor de spaţii virtuale 3D MMO, următoarele module au fost implementate şi utilizate la nivelul serverului :

- Modul de alocare clienţi şi resurse
- Modul de planificare sarcini (care are două componente distincte)
- Modul procesare CUDA

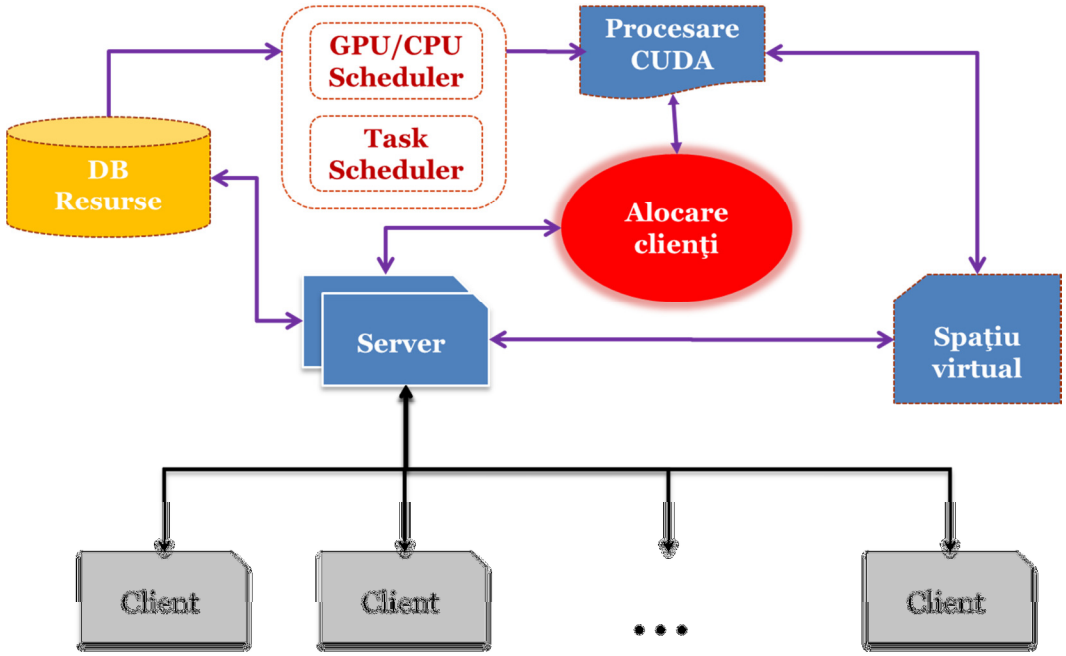

Figura 5-5: Soluţie arhitecturală cu posibilităţi de procesare GPGPU

5.5.1 MODULUL ALOCARE CLIENŢI ŞI RESURSE

Acest modul este responsabil cu alocarea clienţilor şi a resurselor necesare executării calculelor la nivel de server.

Pentru realizarea alocării se folosesc următoarele criterii :

1. Poziţionarea spaţială în cadrul geografiei spaţiului virtual a utilizatorilor care sunt on-line la momentul respectiv de timp. Astfel, dacă într-o anumită zonă logică sunt prezenţi mai mulţi utilizatori comparativ cu o altă zonă a spaţiului virtual, celei dintâi îi vor fi alocate mai multe resurse de calcul din cele disponibile în sistem pentru a putea face astfel faţă la numărul mai mare de operaţii pe care aceasta le are de efectuat.
2. Tipul şi numărul de operaţii executate de utilizatorii spaţiului virtual. Astfel, alocarea procesării sarcinilor de calcul asociate clienţilor este realizată luându-se în consideraţie un cost de „greutate a operaţiilor efectuate” asociat fiecărui utilizator care este on-line în cadrul spaţiului virtual. În felul acesta, utilizatorii care generează un număr crescut de operaţii vor putea

beneficia de mai multe resurse pentru a se asigura faptul că sistemul va face faţă în timp real la acţiunile lor.

3. Tipul de utilizator pentru care trebuie realizate operaţii de calcul. Se face astfel o alocare diferită pentru utilizatorii de tip uman (care vor avea prioritate) şi utilizatorii de tip NPC (non player characters)

Acest modul este responsabil de implementarea şi utilizarea funcţiei globale a costului asociat unui task computaţional, prezentate în paragraful 5.4. Astfel, se va aloca fiecărui task un cost ce va fi folosit mai departe în modulul de planificare a sarcinilor.

5.5.2 MODULUL PLANIFICARE SARCINI

Acest modul este responsabil cu crearea şi planificarea sarcinilor ce vor fi executate la nivelul serverelor spaţiilor virtuale 3D MMO.

În arhitectura propusă, modulul de planificare a sarcinilor are două componente majore :

- componenta de creare a sarcinilor la nivel global (de sistem)
- componenta de planificare a sarcinilor la nivel de GPU / CPU

5.5.2.1 Creare sarcini la nivel de sistem

Această componentă realizează crearea de sarcini de calcul asociate operaţiilor executate în cadrul spaţiului virtual. Astfel, la fiecare iteraţie (*tick*) a buclei principale a simulării, această componentă procesează acţiunile entităţilor spaţiului virtual, creează sarcinile ce rezultă în urma procesării şi le inserează într-o coadă de sarcini globală.

Tot această componentă este responsabilă şi cu asigurarea menţinerii ratei interne cu care funcţionează spaţiul virtual. De obicei, aceasta are o valoare ce asigura un FPS de 30 sau 60.

De asemenea, tot această componentă realizează trimiterea sarcinilor pentru a fi planificate la componenta de planificare.

5.5.2.2 Planificare sarcini la nivel de GPU / CPU

Această componentă este responsabilă cu planificarea şi trimiterea spre execuţia propriu-zisă a sarcinilor la CPU sau la GPU.

Pentru implementarea GPGPU, folosind costul asociat fiecărui task, acest modul realizează planificarea sarcinilor ce vor fi executate pe multiprocesoarele GPU din sistem astfel :

- determină numărul de resurse de calcul (GPU) din sistem;
- în funcţie de *tick-ul* intern si numărul de resurse de calcul, determină numărul maxim de sarcini ce pot fi executate la fiecare iteraţie a buclei principale a simulării spaţiului virtual;
- în funcţie de numărul de sarcini ce vor fi executate, determină dimensiunea blocurilor de fire de execuţie si a grid-ului ce le conţine;
- extrage din coada de sarcini globală acele sarcini ce vor fi planificate în *tick-ul* curent, în funcţie de costul asociat acestora;
- creează vectorul de sarcini de executat folosind sarcinile extrase la pasul anterior şi costul asociat acestora.

Pseudocodul simplificat pentru planificarea sarcinilor pentru calculul GPGPU este următorul :

```
nrGPU = determinaNumarResurseDeCalculDisponibile()

nrMaximSarcini = determinaSarciniDeExecutat
                (nrGPU,totalSarcini,tick intern)

dimBloc = determinaDimensiuneBlocFireDeExecutie()

nrSarciniPerGPU = nrMaximSarcini / nrGPU

dimensiune grid fire de execuţie =
(nrSarciniPerGPU / dimensiuneXbloc ,  1)

sarcini = extrageSarciniInFunctieDeCost(listaGlobalaSarcini)

vectorSarcini = creeazaVectoSarcini(sarcini, nrMaximSarcini)
```

Un lucru important care a fost luat în consideraţie la proiectarea acestei componente a fost faptul ca aceasta să poată funcţiona având la dispoziţie mai multe resurse hardware pentru care să poată planifica sarcinile.

Astfel, s-a implementat fără mari dificultăţi şi soluţia arhitecturală ce foloseşte procesare *multi-GPGPU*. În acest fel, se obţine o scalabilitate transparentă în momentul în care în sistem sunt disponibile / introduse mai multe GPU.

5.5.3 MODULUL DE PROCESARE CUDA

Acest modul primeşte sarcinile GPGPU de la modulul de planificare şi este responsabil cu execuţia acestora direct pe procesoarele plăcii grafice.

La nivelul acestui modul sunt realizate următoarele operaţii :

- implementarea funcţiilor kernel ce vor fi executate de firele de execuţie ale procesoarelor scalare ale GPU-ului
- transferul de memorie de la dispozitivul gazdă la GPU
- după ce sarcinile au fost executate pe GPU, se realizează transferul în sens invers al memoriei (ce conţine rezultatele execuţiilor) de la GPU la gazdă
- operaţiile de sincronizare necesare pentru asigurarea faptului ca toate firele de execuţie ale GPU-ului şi-au terminat procesările

Pseudocodul simplificat pentru procesarea sarcinilor pentru calculul GPGPU este următorul :

```
Pentru fiecare GPU
{
   Copiază sarcini în memoria globală GPU de pe maşina gazdă
   Copiază date spaţiu virtual în memoria globală
   GPU de pe maşina gazdă

   Lansează în paralel pe GPU funcţie kernel(dimensiune
   bloc,dimensiune grid, idGPU)

   Sincronizează prin barieră terminarea firelor de execuţie

   Copiază rezultate sarcini din memoria globală GPU pe maşina
   gazdă
   Copiază date spaţiu virtual din memoria globală GPU pe
   maşina gazdă
}
```

Tot acest modul este responsabil şi cu propagarea rezultatelor execuţiilor către nivelul ce implementează logica spaţiului virtual.

Folosind prototipul de arhitectură ce conţine modulele prezentate mai sus s-au obţinut rezultate încurajatoare în cadrul unor simulări. Acestea au permis interactiunea în timp real a unui număr mare de utilizatori folosind soluţii *single GPGPU* şi *multi GPGPU*, bazate pe hardware grafic de la NVidia (GTX590).

Rezultatele obţinute precum şi o analiză a acestora vor fi prezentate în capitolul următor.

5.6 CONCLUZII

Creşterea în popularitate a aplicaţiilor MMO a dus la atragerea unui număr din ce în ce mai mare de utilizatori ce le accesează, acest fapt ducând la o încărcare foarte mare la care trebuie sa facă faţă lumea virtuală în momentele de utilizare maximă. Aceasta încărcare este data atât de numărul de sarcini, care este direct proporţional cu numărul de utilizatori on-line, cât şi de complexitatea acestor sarcini care depinde de particularităţile spaţiului virtual.

Problemele principale cu care se confruntă arhitecturile curente sunt :

- de scalabilitate
- de fiabilitate
- de redundanţă
- de cost

Modificările propuse în prima parte a acestui capitol pentru a fi aplicate în cadrul arhitecturilor serverelor spaţiilor virtuale încearcă să adreseze aceste probleme prin propunerea a două abordări originale :

- procesarea sarcinilor de calcul intensive la nivelul arhitecturilor folosind tehnici GPGPU;
- folosirea unui mecanism de alocare dinamică a resurselor necesare execuţiei calculelor şi care este centrat pe utilizatorii spaţiului virtual, deoarece aceştia generează necesarul computaţional şi nu datele lumii virtuale în sine.

În partea a două a capitolului este propusă o soluţie arhitecturală completă şi sunt prezentate modulele necesare integrării în arhitectura serverelor de spaţii virtuale 3D MMO a abordărilor prezentate în prima parte a capitolului.

Soluţiile propuse şi arhitectura ce le foloseşte au fost prezentate într-o serie de lucrări ştiinţifice ([AMB10], [MMA09b], [MMA09c]).

6 TESTAREA SCALABILITĂŢII SOLUŢIEI

Pentru a realiza testarea de scalabilitate a fost realizat un prototip funcţional ce implementează funcţionarea unui spaţiu virtual 3D MMO bazat pe soluţia arhitecturală propusă în capitolul anterior.

Astfel, au fost implementate module atât pentru partea de server cât şi pentru cea de client.

6.1 SPECIFICAŢII DE FUNCŢIONARE SERVER

La nivelul arhitecturii serverului au fost implementate modulele descrise în capitolul anterior :

1. **Modulul de alocare a resurselor**
2. **Modulul de creare şi planificare a sarcinilor**
3. **Modulul de procesare a sarcinilor**

Pentru a putea face o comparaţie între rezultate, toate cele 3 module au fost implementate astfel încât să funcţioneze folosind procesare :

- CPU
- Single GPGPU
- Multi GPGPU

Serverul pune la dispoziţie următoarele funcţionalităţi către clienţii care accesează spaţiul virtual:

- Permitere conectare clienţi
- Autentificare clienţi
- Trimitere evenimente către clienţi :
    - Actualizarea poziţiei acestora
    - Actualizări legate de setările lumii virtuale (de exemplu, cu cât se poate deplasa la fiecare iteraţie în lumea virtuală)

  - Actualizarea altor utilizatori conectaţi în spaţiul virtual şi care se află în aceeași zonă cu utilizatorul care a declanşat evenimentul de actualizare
  - Actualizări legate de stări ale lumii virtuale
  - Actualizări privind mesaje trimise de utilizatori între ei
- **Verificare corectitudine deplasare utilizator în spaţiul virtual**
  - Utilizatorii se vor deplasa corect respectând următoarele restricţii :
    - Nu vor putea să „intre” într-un alt utilizator
    - Nu vor putea să „intre” într-un alt obiect din spaţiul virtual
    - Nu vor putea să „zboare”
    - Nu vor putea să ajungă sub teren
    - Nu se vor putea deplasa decât cu o viteza prestabilita; cu alte cuvinte dacă un utilizator va incerca să se deplaseze de la coordonatele curente la nişte coordonate unde pentru a ajunge depăşeşte viteza de deplasare impusă de spaţiul virtual, mişcarea nu va fi considerata validă

Pentru simularea internă a logicii spaţiului virtual la nivel de server au fost implementate următoarele funcţionalităţi:

- Bucla principală a simulării spaţiului virtual :
  - Nucleul unei aplicaţii MMO este reprezentat de o bucla de simulare care se executa sincronizat cu un *frame-rate*; pe parcursul fiecărei iteraţii (*tick*) al buclei de simulare, porţiunii ale stării sunt actualizate în concordanta cu logica simulării spaţiului virtual; în implementarea de faţă a fost folosit un tick intern de 30 de milisecunde, echivalent cu un framerate de aproximativ 30 de cadre pe secundă;
- Descrierea geografiei lumii virtuale :
  - S-a folosit un teren simplu ce a fost generat dintr-o hartă de înălţimi. Aceasta geografie este „partajată” cu aplicaţia client, aceeași reprezentare fiind disponibilă fiecărui utilizator ce va accesa spaţiul virtual.

Serverul pune la dispoziţie către clienţi următoarele funcţii de API (Application programming interface) pentru comunicare :

- *Login()* : Conectare şi autentificare la spaţiul virtual

- *GetPosition()* : se obţine poziţia clientului în spaţiul virtual; această poziţie va fi folosită atunci când utilizatorul accesează de fiecare dată spaţiul virtual pentru a fi „poziţionat” unde trebuie, deoarece el poate părăsi spaţiul virtual când se află în locuri diferite ale acestuia (serverul va stoca şi pe disc poziţiile utilizatorilor astfel încât să poată să le furnizeze şi dacă acesta este oprit / pornit din nou);
- *SetPosition()* : setează pozitii pentru clienţi ; aceasta funcţie se va folosi în mod special de către server pentru a trimite unui utilizator al spaţiului virtual, poziţiile celorlalţi utilizatori (fie când aceştia se conectează, se deplasează, etc);
- *GetParameters()* : se întorc parametrii spaţiului virtual (unul dintre ei foarte important, de care va trebui să se tina cont, este viteza de deplasare a utilizatorilor);
- *Move()* : utilizatorul incearca deplasarea la o anumita poziţie a spaţiului virtual; i se va da voie sau nu (partea de decizie şi de logică trebuie să fie implementată pe server pentru a preveni încercările de trişare ale clienţilor).

6.2 SPECIFICAŢII DE FUNCŢIONARE CLIENT

Modulul client implementează următoarele funcţionalităţi importante :
- interacţiunea cu utilizatorul;
- comunicarea cu serverul folosind interfata de programare (API) pusă la dispoziţie de acesta;
- redarea 3D a spaţiului virtual.

Modulul client oferă clienţilor ce accesează spaţiul virtual următoarele:
- Conectare/Deconectare la spaţiul virtual;
- Autentificare prin username/parolă;
- Interacţiune cu utilizatorul
    - Deplasarea acestuia în spaţiul virtual
    - Comunicare prin chat cu ceilalţi utilizatori
- Clientul funcţionează în următoarele 2 moduri
    - Vizualizarea 3D a spaţiului virtual :
        - Vizualizare a celorlalţi utilizatori din spaţiul virtual
        - Vizualizare a evenimentelor din spaţiul virtual (deplasarea celorlalţi utilizatori)

- Mod consolă :
  - În acest mod nu este necesară vizualizarea 3D a spaţiului virtual; se vor primi în continuare toate evenimentele din spaţiul virtual şi se vor afişa informaţii legate de acestea în consolă; în continuare utilizatorul se va putea deplasa în spaţiul virtual fie folosind taste direcţionale, fie prin apelul direct din consolă al funcţiei de API *Move();*
  - acest mod este necesar pentru a putea simula cu uşurinţă conectarea unui număr ridicat de utilizatori la spaţiul virtual; astfel, se poate porni câte un proces/fir de execuţie pentru fiecare client simulat, iar acesta poate primi un set de comenzi *Move()* pentru a avea un comportament în spaţiul virtual.

6.3 COMUNICAREA ÎNTRE CLIENT ŞI SERVER

Pentru a asigura posibilitatea procesării simultane a mai multor clienţi, serverul utilizează o implementare bazată pe WinSockets. Astfel, aplicaţia server conţine un fir de execuţie pentru tratarea comunicaţiei cu clienţii şi un alt fir de execuţie pentru simularea lumii virtuale.

La pornirea unui client, acesta trimite serverului numele contului şi parola asociate şi, în caz ca sunt acceptate datele de autentificare, clientul primeşte de la server poziţia şi orientarea în lumea virtuală, precum şi un identificator folosit în protocolul de comunicare.

Lumea virtuală este împărţită pe zone (în prototip, harta lumii virtuale are dimensiunea de 256x256 de unităţi şi este împărţită în 16x16 zone). O regiune asociată unei zone reprezintă acea zonă şi cele 8 zone înconjurătoare.

Comunicarea client-server este bazată pe evenimente. Astfel, clientul trimite mesaje serverului doar atunci când se realizează o schimbare de stare (execuţia unei comenzi de mişcare din modul consolă, apăsarea sau ridicarea unei taste de mişcare din modul grafic, trimiterea unui mesaj de chat, ieşirea din aplicaţie).

Aceste evenimente au o arie de vizibilitate pentru client doar la nivelul regiunii în care se află. Astfel, la primirea unui mesaj ce indică declanşarea unui eveniment, serverul îl va propaga doar la clienţii ce se află în regiunea clientului ce a generat evenimentul.

Atât clientul cât şi serverul realizează o simulare a lumii virtuale cu datele care le au la dispoziţie. De asemenea, testele de detecţie de coliziune se realizează numai pe server.

Atunci când se detectează o coliziune, serverul trimite mesaje pentru a anunța acest eveniment tuturor clienţilor din regiunea în care s-a petrecut coliziunea. Acest mesaj conține pozițiile clienţilor (implicati în coliziune) dinainte de realizarea coliziunii.

În plus, serverul trimite şi mesaje periodice tuturor clienţilor înregistraţi în sistem, cu stările celorlalţi clienţi (poziție, orientare, stare de mişcare, destinaţie) din aceeaşi regiune. Acestea sunt folosite pentru sincronizarea clienţilor cu serverul. Prin acestea se elimină şi necesitatea de a trimite mesaje cu starea clienţilor dintr-o nouă regiune atunci când clientul se mută în altă zonă.

Clientul conține şi un timer asociat fiecărui client despre care el ştie că se află în regiunea sa. Timer-ul asociat unui client este resetat atunci când se primește un mesaj ce face referire la acel client. Dacă un timer expiră, atunci se consideră ca acel client asociat timer-ului a părăsit regiunea (altfel s-ar fi primit informaţii despre el de la server prin mesajele periodice) şi nu mai este nevoie să se păstreze informaţiile despre el.

6.4 ALOCAREA RESURSELOR PENTRU CALCULUL COLIZIUNILOR

Pentru a realiza calculul de coliziuni eficient se folosește o repartizare a clienţilor pe zone. În server se formează o reprezentare zonală a acestora, fiecare conţinând toți clienţii din zona asociată. Aceasta repartizare nu trebuie să se facă la fiecare iterație a buclei de simulare a lumii virtuale, deoarece clienţii nu pot sa străbată foarte repede o zonă.

Testele de coliziune sunt astfel limitate la nivelul unei zone şi zonele adiacente (pentru a trata şi cazul în care un avatar se află chiar pe marginea unei zone). În cadrul unui zone, testul de coliziune este de tip N-la-N, testându-se toate avatarele între ele.

Pentru modulul de detecţie a coliziunilor pe GPU se foloseşte un grid în care fiecare bloc este asociat unei zone. În cadrul fiecărui bloc exista mai multe fire de execuţie care realizează testele de coliziune de tip N-la-N, fiecare fir de execuţie testând un anumit număr de avatare.

De asemenea, a fost implementat şi mecanismul de alocare prezentat în capitolul anterior care ţine cont de poziţionarea spaţială a utilizatorilor. Astfel, dacă un bloc de fire de execuţie asociat unei zone nu are nimic de procesat (practic zona asociată nu conţine utilizatori) acesta va fi realocat pentru a procesa teste de coliziune dintr-o zonă ce conţine un număr mare de utilizatori, preluând astfel din efortul computaţional necesar blocului care era asociat la zona ţintă.

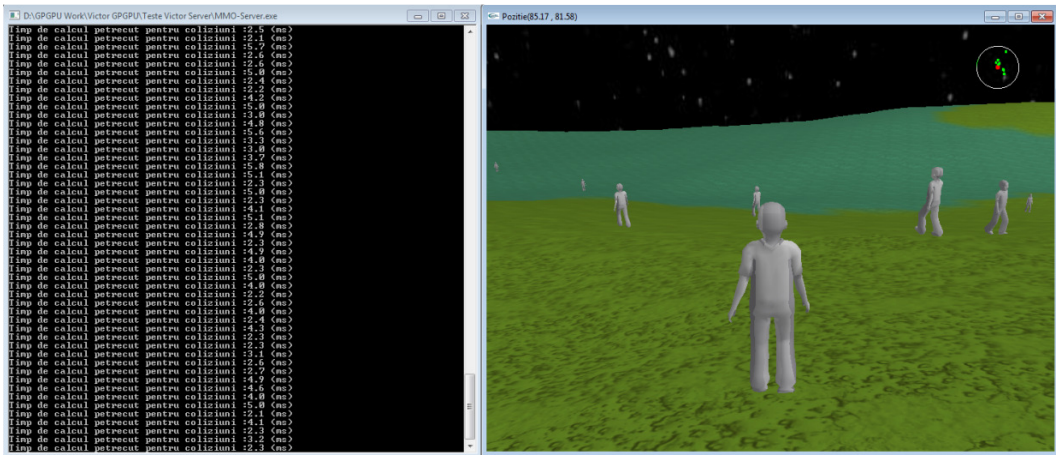

Figura 6-1: Prototip spaţiu virtual ce implementează arhitectura propusă

6.5 REZULTATE

Pentru a simula încărcarea spaţiului virtual cu un număr mare de clienţi a fost folosită aplicaţia client în modul de funcţionare consolă. Astfel, au fost pornite

pentru simularea clienţilor mai multe instanţe ale acesteia, fiecare primind un fişier ce conţine comenzi de deplasare la poziţii aleatoare din spaţiul virtual cu scopul de a simula un comportament pentru fiecare client simulat.

Au fost realizate teste separate pentru următoarele numere de clienţi ce au fost simulaţi că accesează spaţiul virtual :

- 100 de clienţi
- 1000 de clienţi
- 4000 de clienţi
- 8000 de clienţi

Pentru efectuarea calculelor s-au testat toate cele 3 abordări :

- CPU : Intel Core2Quad Q6600
- Single GPGPU : 1 core al unui GPU GTX590
- Multi GPGPU : 4 core-uri GPU (2 GPU GTX590)

S-a avut în vedere timpul total necesar pentru calculul coliziunilor la fiecare iteraţie a buclei principale a spaţiului virtual. *Tick-ul* intern folosit în implementarea serverului este de 30 de milisecunde, deci practic orice depăşire a acestui prag conduce la imposibilitatea procesării în timp real pe partea de server a buclei principale de simulare, apărând întârzieri vizibile la nivelul aplicaţiei client.

Rezultatele sunt prezentate în continuare.

	100 de clienţi	1000 de clienţi	4000 de clienţi	8000 de clienţi
	Timp calcule fizică / iteraţie	Timp calcule fizică / iteraţie	Timp calcule fizică / iteraţie	Timp calcule fizică / iteraţie
1 CPU Quad Core Q6600	4.7ms	46ms	185ms	365ms
Single GPGPU 1 core GPU GTX590	2ms	7ms	23ms	42ms
Multi GPGPU 4 cores GPU GTX590	3ms	6.8ms	9ms	11ms

Tabel 6-1: Rezultate prototip funcţional al arhitecturii propuse

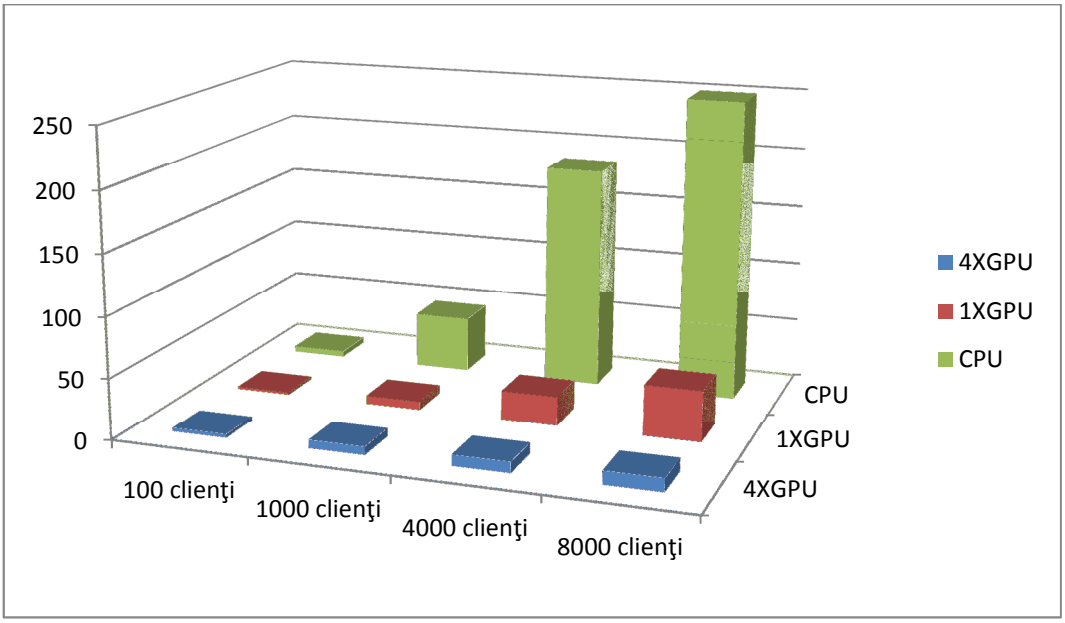

Figura 6-2: Comparaţie intre timpii de execuţie ai prototipului funcţional pentru cele trei implementari

Din rezultatele obţinute se pot astfel observa următoarele :

- pentru 1000 de clienţi, implementarea pe CPU deja nu mai este în stare să facă faţă în timp real ratei interne la care funcţionează spaţiul virtual (30 de milisecunde);
- implementarea *single-GPU* are un speedup fata de cea CPU ce creşte o dată cu numărul de utilizatori şi de abia când numărul acestora atinge 8000 nu mai este în stare sa facă faţă la procesări în timp real;
- implementarea *multi-GPGPU*, datorită overhead-ului rezultat din operaţiile de transfer de memorie şi sincronizare, obţine sporul de performanţă aşteptat faţă de cea *single-GPGPU* doar atunci când numărul de utilizatori este unul ridicat; astfel, chiar şi pentru un număr de 8000 de utilizatori este capabilă să proceseze volumul de operaţii în timp real.

6.6 CONCLUZII

În partea finală a tezei de doctorat s-a tratat testarea de scalabilitate a soluţiilor şi arhitecturilor propuse, pentru a avea rezultate concrete de analizat.

În acest scop, a fost implementat un prototip funcţional al arhitecturii propuse, ce foloseşte :

- procesarea GPGPU a sarcinilor aferente operaţiilor de detecţie a coliziunilor;
- alocarea dinamică a resurselor GPU în funcţie de poziţionarea spaţială a utilizatorilor în cadrul lumii virtuale.

În acest capitol au fost prezentate specificaţiile de funcţionare pentru aplicaţia server şi client precum şi modul în care s-a simulat un număr mare de clienţi ce accesează spaţiul virtual.

Rezultatele obţinute au confirmat validitatea şi viabilitatea soluţiei propuse.

7 CONCLUZII, CONTRIBUŢII ORIGINALE ŞI EVOLUŢII VIITOARE

Popularitatea din ce în ce mai mare a spaţiilor virtuale 3D MMO a condus inevitabil şi spre cresterea semnificativă a numărului utilizatorilor care accesează astfel de spaţii. Acest lucru a condus la următoarele două consecinţe majore :

- pentru a crea o experienţă cât mai bogată şi realistă pentru utilizatorii care accesează spaţiile virtuale, aplicaţiile MMO trebuie sa poată oferi la nivelul aplicaţiei client o redare 3D cât mai fidelă şi realistă din punct de vedere grafic;
- a împins spre extrem încărcarea cu care se confruntă arhitectura acestor aplicaţii la nivel de server, uneori aceasta nemaifăcând faţă numărului foarte mare de utilizatori.

În prezent, arhitecturile de tip Client-Server pun la dispoziţie funcţionalităţile necesare unei aplicaţii MMO, însa o fac cu un cost ridicat. Avantajul unei soluţii cu servere centralizate este dat de faptul că oferă un grad ridicat de control asupra sistemului, lucru care face posibilă asigurarea autentificării utilizatorilor, asigurarea persistenţei şi securităţii spaţiului virtual. Astfel un cluster de servere performante poate oferi o putere computaţională semnificativă iar centrele de date moderne pot face faţă unui trafic mare de date.

Din nefericire, natura puternic centralizată a arhitecturii Client-Server introduce o sugrumare din punct de vedere al performanţei. În ciuda numărului lor care poate fi foarte mare, clusterele de servere sunt de la un anumit punct limitate din punct de vedere computaţional iar traficul de reţea este concentrat practic în echipamentele din centrele de date.

Costurile mari introduse de arhitectura Client-Server limitează practic scalabilitatea acesteia, fiind necesară împărţirea spaţiului virtual în mai multe instanţe / *sharduri* (versiuni independente ale aceleiaşi lumi virtuale). De asemenea, deloc de neglijat este faptul că producătorii spaţiilor virtuale 3D MMO trebuie să facă faţă unui cost financiar semnificativ pentru a menţine funcţională infrastructura sistemului.

Din nefericire, arhitecturile Peer-To-Peer chiar dacă rezolvă câteva dintre neajunsurile arhitecturilor Client-Server, au la rândul lor câteva lipsuri semnificative care le fac de nefolosit la scară largă în aplicaţiile MMO moderne.

Cel mai important neajuns este acele că nu oferă un mecanism viabil prin care să asigure persistenţa stării lumii virtuale. Atunci când un utilizator se deconectează de la spaţiul virtual, resursele puse la dispoziţie dispar, inclusiv datele pe care aplicaţia acestuia le gestiona.

Faptul că nu există un mecanism de autoritate centralizat face foarte dificilă actualizarea spaţiului virtual de către producători şi introduce posibile găuri de securitate.

De asemenea, deşi participanţii pot pune la dispoziţie resurse suplimentare sistemului, acest lucru se realizează într-un mod eterogen, fiecare maşina având propriile limite computaţionale. Fără un mecanism simplu de distribuire a încărcării, o concentrare mare de activităţi asupra unui peer poate să consume foarte rapid resursele acestuia.

Ţinând cont de cele spuse mai sus, a fost realizată cercetarea posibilităţilor de îmbunătăţire a tehnologiilor pentru spaţii virtuale MMO în următoarele direcţii majore:

- la nivelul redării 3D oferite utilizatorilor:
    - posibilitatea creşterii realismului redării spaţiului virtual 3D prin folosirea algoritmului RayTracing implementat utilizând tehnici de paralelizare GPGPU
- la nivelul serverelor de spaţii virtuale 3D MMO :
    - optimizarea operațiilor efectuate în cadrul spaţiilor virtuale 3D MMO prin folosirea de tehnici GPGPU
    - modificări în arhitectura curentă a serverelor de spaţii virtuale 3D MMO

În continuare se va prezenta un rezumat al contribuţiilor originale așa cum reies din capitolele anterioare.

7.1 CONTRIBUŢII ORIGINALE

- **ANALIZA CONCEPTELOR, TEHNOLOGIILOR ŞI ARHITECTURILOR CURENTE FOLOSITE DE APLICAŢIILE MMO**

A fost efectuată o amplă analiză a conceptelor şi tehnologiilor folosite pentru realizarea aplicaţiilor MMO atât la nivel de client cât şi de server.

De asemenea, au fost analizate arhitecturile curente folosite în implementarea spaţiilor virtuale 3D MMO, punându-se în evidenţă atât avantajele utilizării acestora cât şi lipsurile cu care se confruntă.

Această analiză, a dus la stabilirea unui punct de vedere propriu şi original asupra tehnologiilor folosite în domeniu, stând la baza cercetărilor ulterioare şi a soluţiilor propuse.

- **UTILIZAREA TEHNICILOR GPGPU PENTRU REDAREA 3D LA CLIENT**

Pentru a crea un grad de imersiune cât mai ridicat (prin creşterea realismului vizualizării lumii virtuale 3D) pentru utilizatorii spaţiului MMO, la nivelul aplicaţiei client se poate folosi ca metodă de redare algoritmul RayTracing.

Până de curând, utilizarea RayTracing ca metodă de redare în timp real nu era viabilă deoarece costul computaţional al metodei este unul ridicat iar hardware-ul existent nu era suficient de rapid pentru a obţine rezultate în timp real.

Odată cu lansarea la sfârşitul anului 2006 a arhitecturilor unificate pentru GPU, a apărut posibilitatea ca o gamă largă de operaţii de uz general sa poată fi implementate folosind tehnici GPGPU.

Au fost astfel propuse soluţii pentru a executa algoritmul RayTracing folosind procesare *single-GPGPU* şi *multi-GPGPU*. Solutiile propuse au fost implementate şi au fost prezentate rezultatele obţinute.

Este important de menţionat că la timpul realizării cercetării (2008 şi 2009) încercările de exploatare a paralelismului de tip GPGPU pentru implementarea algoritmului RayTracing se aflau într-o stare incipientă astfel încât cercetarea realizată în acel moment a fost una de actualitate.

- **UTILIZAREA TEHNICILOR GPGPU ÎN CADRUL SERVERELOR MMO**

La nivelul serverelor de spaţii virtuale 3D MMO se execută un număr foarte mare de operaţii pentru a simula o fizică realistă a spaţiului virtual (detecţia coliziunilor, forţe de frecare, forţa de gravitaţie, etc).

Acestea reprezintă gama de operaţii care sunt cele mai frecvent efectuate de către serverele spaţiului virtual şi astfel sunt principalele consumatoare de timp de calcul.

A fost propusă o soluţie de paralelizare folosind procesare *single-GPGPU* şi *multi-GPGPU* a calculelor de detecţie a coliziunilor ce sunt executate în mod obişnuit în cadrul spaţiilor virtuale. Solutia a fost şi implementată.

Rezultatele obţinute în urma implementării soluţiei au confirmat validitatea ei şi au scos în evidenţă următoarele aspecte majore :

- implementarea folosind CPU nu mai face faţă în timp real pentru un număr relativ mediu de obiecte existente în scenă; se poate deduce astfel că procesarea pe CPU a calculelor de fizică nu este scalabilă cu numărul de obiecte din scenă;

- implementarea ce foloseşte single-GPGPU obţine sporuri de performanţă de ordinul zecilor faţă de CPU;

- implementarea ce foloseşte *multi-GPGPU* obţine sporuri de performanţă de ordinul sutelor faţă de CPU.

- **PROPUNEREA UNUI MECANISM DE ALOCARE DINAMICĂ A RESURSELOR, CENTRAT PE UTILIZATORII SPAŢIULUI VIRTUAL**

A fost propusă folosirea unui mecanism de alocare dinamică a resurselor necesare execuţiei calculelor, care este centrat pe utilizatorii spaţiului virtual deoarece aceştia generează necesarul computaţional şi nu datele lumii virtuale în sine.

Soluţia propusă vizează optimizarea alocării resurselor ţinându-se cont de următorii 3 factori :

- poziţionarea spaţială a utilizatorilor care sunt on-line în cadrul spaţiului virtual;
- un cost de „greutate a operaţiilor efectuate” asociat fiecărui utilizator care este on-line în cadrul spaţiului virtual;
- tipurile de entităţi pentru care trebuie realizate calcule.

- **SOLUŢIE ARHITECTURALĂ COMPLETĂ CU POSIBILITATEA EXPLOATĂRII PARALELISMULUI DE TIP GPGPU**

A fost propusă o soluţie arhitecturală completă ce conţine modulele necesare integrării în arhitectura serverelor de spaţii virtuale 3D MMO a soluţiilor de optimizare prezentate anterior :

- capabilităţi de procesare a sarcinilor de calcul folosind tehnici GPGPU;
- folosirea mecanismului de alocare a resurselor centrat pe utilizatori.

- **IMPLEMENTAREA UNUI PROTOTIP FUNCŢIONAL PENTRU ARHITECTURA PROPUSĂ ŞI TESTAREA DE SCALABILITATE A ACESTEIA**

A fost implementat un prototip funcţional al arhitecturii propuse, ce foloseşte:

- procesare GPGPU a sarcinilor aferente operaţiilor de detecţie a coliziunilor;
- alocarea dinamică a resurselor GPU în funcţie de poziţionarea spaţiala a acestora în cadrul lumii virtuale.

A fost implementată aplicaţia server, aplicaţia client precum şi un mecanism de simulare a unui număr mare de clienţi ce accesează spaţiul virtual pentru a se putea realiza testele de scalabilitate.

Rezultatele obţinute au confirmat validitatea şi viabilitatea soluţiei arhitecturale propuse.

7.2 EVOLUŢII VIITOARE

În viitor se poate continua cercetarea în următoarele direcţii, pentru a îmbunătăți / rafina soluţiile prezentate în această lucrare :

- implementarea la nivelul prototipului funcţional şi a celorlalţi factori propuşi a fi folosiţi de mecanismul de alocare :
    - tipul şi frecvenţa operaţiilor executate de utilizatorii spaţiului virtual
    - tipurile de entităţi care generează sarcini de calcul
- ameliorarea overhead-ului introdus de operaţiile de transfer de memorie şi sincronizare efectuate atât pentru implementările *single-GPGPU* cât şi *multi-GPGPU*; se pot cerceta soluţii de optimizare a acestor operaţii consumatoare de timp folosind noile capabilităţi de adresare / transfer a memoriei disponibile în arhitectura 4.0 CUDA ce a apărut în Mai 2011 :
    - posibilitatea de partaja accesul la GPU din contextul mai multor fire de execuție independente;
    - utilizarea tuturor GPU din sistem în mod concurent din cadrul unui singur fir de execuţie;
    - **spaţiu de memorie virtual unificat** : existenţa unui singur spatiu de adresare (CPU şi GPU) pentru o utilizare mai facilă a resurselor de memorie;
    - **acces Peer-To-Peer** : posibilitatea de comunicare directă între Unităţile de Prelucrare Grafica (GPU) din sistem.

8 LISTA LUCRĂRILOR ŞI ACTIVITĂŢILOR ŞTIINŢIFICE ALE AUTORULUI

8.1 LUCRĂRI LEGATE DE TEMATICA TEZEI

8.1.1 CĂRŢI

[MMA09a] Alin Moldoveanu, Florica Moldoveanu, **Victor Asavei**, Costin Boiangiu

Realitatea Virtuală

Editura Matrix Rom, ISBN:978-973-755-488-8, 216 pg., 2009

8.1.2 ARTICOLE

[AMM09a] **Victor Asavei**, Florica Moldoveanu, Alin Moldoveanu

Ray Tracing as GPGPU

CSCS 17 - The 17th International Conference On Control Systems And Computer Science, 26-29 May 2009, Bucharest

[AMM09b] ***Victor Asavei***, *Vlad-Valentin Ionita, Florica Moldoveanu, Alin Moldoveanu*

Real-Time Parallel Volume Rendering using Nvidia Cuda for Medical Imaging

1483-1485, Annals of DAAAM for 2009 & Proceedings of the 20th International DAAAM Symposium, ISBN 978-3-901509-70-4, ISSN 1726-9679, pp 742, Editor B[ranko] Katalinic, Published by DAAAM International, Vienna, Austria 2009 **(Proceeedings ISI)**

[AMM09c] ***Victor Asavei***, *Florica Moldoveanu, Alin Moldoveanu, Costin-Anton Boiangiu, Ruxandra Marasescu*

Ray Tracing as Multi GPGPU

1203-1205,Annals of DAAAM for 2009 & Proceedings of the 20th International DAAAM Symposium, ISBN 978-3-901509-70-4, ISSN 1726-9679, pp 602, Editor B[ranko] Katalinic, Published by DAAAM International, Vienna,Austria 2009 **(Proceeedings ISI)**

[AMM10a] ***Victor Asavei***, *Alin Moldoveanu, Florica Moldoveanu, Anca Morar, Alexandru Egner*

GPGPU for Cheaper 3D MMO Servers

International Conference on Telecommunications and Informatics : Session Information Science and Applications, Catania, Italia, 29-31 Mai 2010 **(Proceeedings ISI)**

[AMB10] ***Asavei, V.;*** *Moldoveanu, A. D. B.; Moldoveanu, F.; Boiangiu, C. A.;Morar, A. - A. & Egner, A. I.*

Innovative 3D MMO Servers Architectures Based on GPGPU

Annals of DAAAM for 2010 & Proceedings of the 21st International DAAAM Symposium, ISBN 978-3-901509-73-5, ISSN 1726-9679, pp 0325, Editor B. Katalinic, Published by DAAAM International, Vienna, Austria 2010 **(Proceeedings ISI)**

[AMM11] ***Victor Asavei****, Florica Moldoveanu, Alin Moldoveanu, Alexandru Egner, Anca Morar*

Multi GPGPU Optimizations for 3D MMO Virtual Spaces

CSCS 18 - The 18th International Conference On Control Systems And Computer Science, 24-27 May 2011, Bucharest

[MMA08a] *Alin Moldoveanu, Florica Moldoveanu, Alexandru Soceanu,* ***Victor Asavei***

A 3D Virtual Museum

Scientific Bulletin of UPB, Series C, vol. 70, No.3, ISSN 1454-234x, 2008

[MMA08b] *Alin Moldoveanu, Florica Moldoveanu,* ***Victor Asavei***

Méga Musée Virtuel

Conférence Internationale Francophone d'Automatique (CIFA) , ISBN: 978-2-915913-24-8 , Bucarest, 3-5 Septembre 2008

[MMA09b] *Alin Moldoveanu, Florica Moldoveanu,* ***Victor Asavei***

Highly Scalable Server Architecture for Massive Multi-player 3D Virtual Spaces

"Recent Advances în Applied Mathematics and Computaţional and Information Sciences", vol II, Proceedings of the 15th American Conference on Applied Mathematics and Proceedings of the International Conference on Computaţional and Information Science 2009, Houston, USA, April 30-May 2, 2009, ISSN: 1790-5117, ISBN: 978-960-474-071-0

[MMA09c] *Alin Moldoveanu, Florica Moldoveanu,* ***Victor Asavei***

More scalability at lower costs – Server Architecture for Massive Multi-player 3D Virtual Spaces powered by GPGPU

International Journal of Computers and Communication, Issue 2, Volume 1, 2009, ISSN: 2074-1294, published by University Press, London, UK, 2009

[MMA09d] *Alin Moldoveanu, Florica Moldoveanu,* ***Victor Asavei***

Méga Musée Virtuel

Automatique Avancee et Informatique Appliquee, Editura Academiei Romane, 2009

[MMA11] *Alin Dragos Bogdan Moldoveanu, Florica Moldoveanu,* ***Victor Asavei****, Alexandru Egner, Anca Morar*

From HTML to 3DMMO - a Roadmap Full of Challanges

CSCS 18 - The 18th International Conference On Control Systems And Computer Science, 24-27 May 2011, Bucharest

[LMM11] *Lucian Petrescu, Anca Morar, Florica Moldoveanu and* ***Victor Asavei***

Real Time Reconstruction of Volumes from Very Large Datasets Using CUDA

International Conference on System Theory, Control and Computing, 14-16 October, Sinaia, Romania (accepted as a full paper and for publication în the conference proceedings)

8.2 ALTE LUCRĂRI PUBLICATE ÎN PERIOADA DESFĂŞURĂRII DOCTORATULUI

[MMM10a] *Anca Morar, Florica Moldoveanu, Alin Moldoveanu,* ***Victor Asavei****, Alexandru Egner*

Computer Assisted Analysis of Orthopedic Radiographic Images

International Conference on Signal Processing : Session Image Processing, Catania, Italia, 29-31 Mai 2010 **(Proceeedings ISI)**

[MMM10b] *Anca Morar, Florica Moldoveanu, Alin Moldoveanu,* ***Victor Asavei****, Alexandru Egner*

Medical Image Processing in Hip Arthroplasty

WSEAS TRANSACTIONS on SIGNAL PROCESSING, Volume 6, Print ISSN: 1790-5052, E-ISSN: 2224-3488, 2010

[MMM10c] *Anca Morar, Florica Moldoveanu, Alin Moldoveanu,* ***Victor Asavei***

Computer Assisted Analysis of 2D/3D Medical Images

Annals of DAAAM for 2010 & Proceedings of the 21st International DAAAM Symposium, ISBN 978-3-901509-73-5, ISSN 1726-9679, Editor B. Katalinic, Published by DAAAM International, Vienna, Austria 2010 **(Proceeedings ISI)**

[EMM10] Egner*, A. I.; Moldoveanu, F.; Moldoveanu, A. D. B.;* ***Asavei, V****.; Morar, A. - A.; & Boiangiu, C. A*.

Testing the interoperability of HL7-based applications using TTCN-3

Annals of DAAAM for 2010 & Proceedings of the 21st International DAAAM Symposium, ISBN 978-3-901509-73-5, ISSN 1726-9679, Editor B. Katalinic,Published by DAAAM International, Vienna, Austria 2010 **(Proceeedings ISI)**

[MAM10] *Moldoveanu, A. D. B.; **Asavei, V**.; Moldoveanu, F; Morar, A. - A. & Egner, A. I. .; Boiangiu, C. A.*

Turning Nearshoring into a Success – Managing Technical Background Differences
Annals of DAAAM for 2010 & Proceedings of the 21st International DAAAM Symposium, ISBN 978-3-901509-73-5, ISSN 1726-9679, Editor B. Katalinic, Published by DAAAM International, Vienna, Austria 2010 **(Proceeedings ISI)**

[BPM10] *Boiangiu, C. A.; Petrescu, S.B. ; Moldoveanu, A. D. B.; .; **Asavei, V.**; Bucur, I.*

Systematic Printing Space Recognition
Annals of DAAAM for 2010 & Proceedings of the 21st International DAAAM Symposium, Editor B. Katalinic,Published by DAAAM International, Vienna, Austria 2010 **(Proceeedings ISI)**

[APM11] ***Victor Asavei**, Alexandru-Lucian Petrescu, Florica Moldoveanu*

A Fractal Approach to Terrain Generation and Rendering
CSCS 18 - The 18th International Conference On Control Systems And Computer Science, 24-27 May 2011, Bucharest

[EMM11] *Alexandru Egner, Florica Moldoveanu, Alin Moldoveanu, Victor Asavei, Anca Morar*

Automated generation of TTCN-3 type set used for testing of healthcare applications
CSCS 18 - The 18th International Conference On Control Systems And Computer Science, 24-27 May 2011, Bucharest

[MMM11] *Anca Morar, Florica Moldoveanu, Alin Moldoveanu, Victor Asavei, Alexandru Egner*

Computer Assisted Insertion of Prostheses Based on Medical Images
CSCS 18 - The 18th International Conference On Control Systems And Computer Science, 24-27 May 2011, Bucharest

8.3 ALTE ACTIVITĂŢI ŞTIINŢIFICE REALIZATE ÎN PERIOADA DESFĂŞURĂRII DOCTORATULUI

Proiect de cercetare PNCDII- Parteneriate: "**Sisteme incorporate tip neuroproteză pentru recuperarea persoanelor cu handicap neuromotor**" (**SINPHA**), membru în echipa de cercetare (2007-2010)

Proiect de cercetare EUREKA: "**Testarea fiabilităţii sistemelor medicale**" **ReTeMes**, membru în echipa de cercetare (2008-2010)

Proiect de cercetare Leonardo da Vinci : "**Strategic Planning for Sustainable Clustering of Collaborative SMEs**" (**SMECluster**), membru în echipa de cercetare (2008-2010)

Proiect de cercetare Leonardo da Vinci : "**Establishment of Sustainable Collaborative SME Networks**" (**SMENet**), membru în echipa de cercetare (2008-2010)

Proiect de cercetare PNCDII- Parteneriate: "**Sistem informatic avansat, bazat pe imagistică medicală, pentru producerea implanturilor personalizate dedicate artroplastiei de şold"** (**SABIMAS**), membru în echipa de cercetare (2008-2011)

Proiect de cercetare EuroStars Inovare: "**Risk Detection in Laboratory Information Systems"** (**RELIS**), membru în echipa de cercetare (2010-prezent)

Proiect de cercetare EuroStars Inovare: "**Enterprise Unified Guideline Engine"** (**EUGEN**), membru în echipa de cercetare (2010-prezent)

Proiect de cercetare EuroStars Inovare: "**Visualization of Patient Data for easy management of care processes"** (VISUAL-D), membru în echipa de cercetare (2011-prezent)

Proiect de cercetare EUREKA: "**Identificarea riscurilor medicale operaţionale şi detecţia fraudelor"** (MORIS FD), membru în echipa de cercetare (2011-prezent)

Organizare Workshop „Graphics and Virtual Reality Summer Workshop" 2008,2009,2010,2011 în cadrul Facultăţii de Automatică şi Calculatoare

9 BIBLIOGRAFIE

[MIL10] John L. Miller, John Crowcroft
The near-term feasibility of P2P MMOG's
NetGames '10 Proceedings of the 9th Annual Workshop on Network and Systems Support for Games

[LAK09] Geetika T. Lakshmanan, Yuri G. Rabinovich, Opher Etzion
A stratified approach for supporting high throughput event processing applications
Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, 2009

[MAR09] Alma Martinez G., Hector Orozco A., Felix Ramos C., Mario Siller
A Peer-To-Peer Architecture for Real-Time Distributed Visualization of 3DCollaborative Virtual Environments
2009 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications

[CHE08] Che S. , Others
A Performance Study of General Purpose Applications on Graphics Processors
Journal of Parallel and Distributed Computing, Volume 68, Issue 10, Pages 1370-1380, 2008

[NVI09a] NVidia Corporation
NVIDIA OptiX RayTracing Engine,
SIGGRAPH 2009, NEW ORLEANS—Aug. 4, 2009

[CAR11] John Carmack,
GPU Race, Intel Graphics, Ray Tracing, Voxels and more!,
HTTP://WWW.PCPER.COM/REVIEWS/EDITORIAL/JOHN-CARMACK-INTERVIEW-GPU-RACE-INTEL-GRAPHICS-RAY-TRACING-VOXELS-AND-MORE

[BRE08] Breitbart J.
Case studies on GPU usage and data structure deşign
Dept. of Computer Science and Electrical Engineering, Universitat Kassel, 2008

[FOL05] Foley T. and Sugerman J.
Kd-tree acceleration structures for a GPU Raytracer
ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, ACM Press, Pages 15-22

[CHE06] Alvin Chen and Richard R. Muntz
Peer Clustering: A Hybrid Approach to Distributed Virtual Environments

The 5th Workshop on Network & System Support for Games 2006 — NETGAMES 2006

[LAV02] Antonia Lucinelma Pessoa Albuquerque, Luiz Velho
Togetherness through virtual worlds: How real can be that presence?
În Fifth Annual International Workshop on Presence 2002, pages 435–447, 2002

[GAR03] Maia Garau
The Impact of Avatar Fidelity on Social Interaction în Virtual Environments
2003

[PET97a] S. Petrescu, Zoea Racoviţă, G. Hera, F. Moldoveanu, Alin Moldoveanu
Algorithm for logic operation on connected closed contours CSCS 11,
Proceedings of the 11th Int. Conference on Control Systems and Computer Science, vol. II, pg. 260-264, 1997

[WAL04] Wald, Ingo
Real Time Ray-Tracing and Interactive Global Illumination
2004

[MOL96] Moldoveanu, Florica & Alţii
Grafica pe calculator
Ed. Teora, Bucureşti, 1996

[BLINN] Blinn, James
Models of light reflection for computer synthesized pictures
SIGGRAPH '77 Proceedings of the 4th annual conference on Computer graphics and interactive techniques

[PHONG] Phong, B.T.
Illumination for computer generated pictures
Communications of ACM 18

[QUA04] Quake3 RayTraced
HTTP://WWW.Q3RT.DE

[QUA06] Quake4 RayTraced
HTTP://WWW.Q4RT.DE

[QWR08] Quake Wars RayTraced
HTTP://WWW.QWRT.DE

[WOL10] Wolfenstein RayTraced
HTTP://WWW.WOLFRT.DE

[GAI11] Gaikai
HTTP://WWW.GAIKAI.COM

[ONL10] OnLive
HTTP://WWW.ONLIVE.COM

[EVE11] Eve OnLine
HTTP://WWW.EVEONLINE.COM

[WOW11] World of Warcraft
HTTP://WWW.WOW-EUROPE.COM

[CUD11] CUDA Programming Guide
HTTP://WWW.NVIDIA.COM/OBJECT/CUDA_DEVELOP.HTML

[FEK06] Fernando Randima, Kilgard Mark
The Cg Tutorial-The Definitive Guide to Programmable Real-Time Graphics
Addison-Wesley, 2006

[FAT08] Kayvon Fatahalian and Mike Houston
A Closer look at GPUs
Communication of the ACM, vol 51, no 10, 2008

[WAO82] M. W. Walker, D. Orin
Efficient dynamic computer simulation of robotic mechanisms
ASME Journal of Dynamic Systems, Measurement, and Control, 1982

[FEA87] R. Featherstone
Robot Dynamics Algorithms
Kluwer Academic Publishers, 1987

[BMT96] Ronan Boulic, Ramon Mas, Daniel Thalmann
A robust approach for the control of the center of mass with inverse kinetics
Computers & Graphics 20(5): 693-701,1996

[PER02] K Perlin
Generation based on motion blending
Proceedings of ACM SIGGRAPH Symposium on Computer Animation, 2002]

[WIH97] D J Wiley, J K Hahn
Interpolation synthesis for articulated figure motion

Proceedings of Virtual Reality Annual International Symposium, 1997

[GUR96] Shang Guo, James Robergé

A high-level control mechanism for human locomotion based on parametric frame space interpolation

Proceedings of the 6th EuroGraphics Workshop on Animation and Simulation, 1996

[RCB98] Charles Rose, Michael F. Cohen, Bobby Bodenheimer

Verbs and Adverbs: Multidimensional Motion Interpolation

IEEE COMPUTER GRAPHICS AND APPLICATIONS, Volume 18 Issue 5, 1998

[DEN04] Philippe Decaudin, Fabrice Neyret

Rendering Forest Scenes în Real-Time

RENDERING TECHNIQUES '04 ,EUROGRAPHICS SYMPOSIUM ON RENDERING, 2004

[CDN04] Florent Cohen, Philippe Decaudin, Fabrice Neyret

GPU-Based Lighting and Shadowing of Complex Natural Scenes

SIGGRAPH Conf., 2004

[CHA06] Chris Chambers, Wu-chang Feng, Wu-chi Feng

Towards public server MMOs

Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games, 2006

[KES05] J. Kesselman

Server Architectures for Massively Multiplayer Online Games

În Session TS-1084, Javaone conference, SUN, 2005

[WIE06] A. Wierzbicki

Trust Enforcement în Peer-to-peer Massive Multi-player Online Games

Proc. Grid computing, high-performAnce and Distributed Applications (GADA'06), Springer, LNCS 4276 (2006), 1163-1180

[GUP09] Nitin Gupta and others

Scalability for Virtual Worlds

Proceedings ICDE '09 Proceedings of the 2009 IEEE International Conference on Data Engineering

[BEZ08] Carlos Eduardo B. Bezerra, F´abio R. Cecin, Cl´audio F. R. Geyer

A3: a Novel Interest Management Algorithm for Distributed Simulations of MMOGs

12th 2008 IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications

[GLI08] Frank Glinka, Alexander Ploss, Sergei Gorlatch, and JensM¨ uller-Iden
High-level development of multiserver online games
International Journal of Computer Games Technology - Networking for Computer Games, Volume 2008, January 2008