UNIVERSITATEA POLITEHNICA BUCUREȘTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

PROIECT DE DIPLOMĂ

Utilizarea GPGPU (Cuda/OpenCL) pentru arhitectura serverelor de spații virtuale 3D de tip MMO

Alin Drăguț

Coordonator științific:
Sl. Dr. Ing. Victor Asavei

BUCUREŞTI

2022

CUPRINS

SINOPSIS

Gestionarea spațiilor virtuale pentru jocurile de tip MMO presupune rezolvarea mai multor probleme, fiind nevoie de asigurarea unei aplicații fiabile, persistente, cu o latență mică și care nu permite jucătorilor să trișeze. Acest proiect propune o arhitectură clasică client-server, cu accent pe decuplarea componentelor și rularea de programe optimizate GPGPU pentru procesările care pot fi paralelizate. Rezultatele obținute sunt foarte bune, obținând un factor de accelerare de două ordine de magnitudine față de o implementare naivă GPGPU pentru procesarea calculelor de coliziune și pentru gestionarea zonelor de interes pentru utilizatori, putând rula simulări în timp real pentru sute de mii de obiecte și mii de jucători conectați.

MULȚUMIRI

Aș dori să-i mulțumesc coordonatorului științific Sl. Dr. Ing. Victor Asavei pentru discuțiile și pentru feedback-ul acordat de-a lungul dezvoltării proiectului.

1 INTRODUCERE

Jocurile MMO (Massively Multiplayer Online) atrag o populație foarte mare de utilizatori, aceste jocuri prezentând un grad mare de imersiune datorită spațiului virtual masiv și al aspectului social, cu mii de jucători conectați în mod concurent care pot interacționa între ei, fiind o categorie de jocuri care implică mult mai multe aspecte decât un joc singleplayer/multiplayer. Dezvoltarea unei astfel de aplicații presupune mai multe necesități, care pot fi redate din acronimul MMO:

- M (Massively) - presupune ca back-end-ul folosit de aplicație să fie unul scalabil, capabil să livreze conținut pentru mii de utilizatori conectați concurent
- M (Multiplayer) - utilizatorii conectați trebuie să rămână sincronizați, pentru a evita probleme de securitate sau de situații nedorite în timpul interacțiunii dintre jucători
- O (Online) - aplicația trebuie să asigure o latență cât mai bună, fenomenul de lag (întârzieri la actualizarea stării jocului) să fie pe cât de mult evitat pentru a nu cauza o experiență frustrantă pentru utilizatori. De asemenea, așteptările de la o astfel de aplicație sunt ca aceasta să fie disponibilă oricând, fiind nevoie de un grad mare de fiabilitate și de o stocare persistentă a lumii, astfel ca progresul jucătorilor să fie întotdeauna salvat. Un alt lucru important în cazul aplicațiilor online este securitatea, fiind nevoie de un canal securizat pentru comunicarea datelor confidențiale

Obiectivele propuse pentru acest proiect sunt de a folosi o arhitectură centralizată autoritară, algoritmi GPGPU (General Purpose Computing on Graphical Processing Units) pentru a rezolva problema scalabilității, dezvoltarea unui mecanism de gestionare a zonelor de interes ale utilizatorilor pentru a reduce din lățimea de bandă necesară pentru actualizarea stării jocului pentru mii de utilizatori conectați, de a asigura persistența datelor utilizatorilor și de a oferi un canal securizat de comunicare.

Următorul capitol va acoperi studiul aplicațiilor similare existente și studiul anumitor arhitecturi și algoritmi folosiți în domeniu. În continuarea studiului domeniului este relatat capitolul tehnologiilor folosite în proiect, continuând cu detalii despre implementare ale diferitelor sisteme din prototip, și, bineînțeles, un capitolul cu rezultatele obținute, unde se vor prezenta câteva rezultate comparative între timpii de procesare pentru simularea fizicii folosind un algoritm naiv care folosește doar un singur nucleu al CPU, un algoritm naiv care folosește programare GPGPU și un algoritm optimizat GPGPU. Pentru algoritmul optimizat GPGPU sunt prezentați in detaliu timpii de procesare pentru diferitele sisteme dezvoltate. La finalul proiectului este relatată concluzia, fiind adăugate și câteva idei pentru dezvoltări ulterioare.

2 STUDIUL DOMENIULUI

În capitolele următoare sunt relatate informații despre studiile efectuate din domeniu, fie la nivel de aplicații deja existente, fie la nivel de tehnici folosite.

2.1 Studiu aplicațiilor din domeniu

În acest capitol sunt relatate câteva aplicații din domeniu care au reprezentat un punct bun de început în gândirea arhitecturii prototipului. Aceste aplicații prezintă fie o arhitectură cu o scalabilitate foarte bună, fie implementează tehnici ingenioase care permit procesarea în timp real a unui număr foarte mare de jucători și interacțiuni între aceștia.

2.1.1 Eve Online

Eve Online[1] este un joc MMORPG plasat în spațiu, în care jucătorii pilotează nave și pot lua parte la diferite activități cum ar fi:

- Minat
- Meșteșugărit
- Comerț
- Explorare
- Lupte atât între jucători, cât și împotriva mediului

Acest joc este diferit față de alte jocuri MMORPG prin faptul că folosește o singură instanță care este responsabilă de starea lumii. Acest lucru este făcut posibil prin universul masiv, care conține mii de sisteme solare, ale căror peisaje sunt generate procedural. De asemenea, deoarece acțiunea se petrece în spațiu, există mult mai puține obiecte cu care jucătorul poate interacționa la orice moment de timp.

Centrul arhitecturii este o bază de date relațională care menține întreaga stare a jocului, acompaniată de un nivel de control foarte mare asupra eficienței metodelor folosite în baza de date, deoarece se caută o viteză de răspuns cât mai bună pentru jucători.

Arhitectura acestui joc permite simularea unui sistem solar pe un singur nucleu al procesorului, doar anumite procedee asincrone și partea de rețea fiind distribuite pe alte nuclee. Legătura dintre sistemele solare este reprezentată printr-o gaură de vierme, transferul jucătorului fiind făcut într-un mod ”seamless” (fără o perioadă în care jocul se blochează pentru a încărca un nivel). Există totuși probleme cu această abordare, fiind situații pe parcursul dezvoltării jocului în care anumite sisteme solare ofereau avantaje foarte mari în cazul comerțului, ele având multe legături cu alte sisteme solare, astfel acestea au devenit un fel de ”hub” de jucători. Acest lucru a dus la încărcări foarte mari pentru nucleul care rula simularea acelui sistem solar, crescând timpul de răspuns al serverului la acțiunile jucătorilor. Acest lucru a fost rezolvat prin metode de game design, căutându-se modalități de a distribui jucătorii pe mai multe sisteme solare. Totuși, tot mai

[1] www.eveonline.com

apar probleme de genul acesta, mai ales în bătălii în care participă mulți jucători - zonele care sunt contestate în mod regulat și zonele care sunt hub-uri de jucători sunt de obicei rulate pe servere mai puternice.

Totuși, Eve Online a dezvoltat un mecanism prin care încărcarea unui server în timpul unei situații care consumă multe resurse (de exemplu o bătălie în care participă mii de jucători) și anume dilatarea timpului[2]. Acest mecanism încetinește timpul, pentru a impune o limită asupra solicitării serverului – se poate vedea și în timpul jocului de către jucători, putând observa că anumite animații durează mai mult timp (de exemplu când se distruge o navă). Dilatarea timpului se face în mod dinamic, în funcție de cât de încărcat este serverul, în incremente mici pentru a nu fi foarte sesizabil.

Deși arhitectura cu o singură instanță folosită de Eve Online pare simplă la nivel înalt, există foarte multe provocări în legătura cu optimizarea I/O-ului bazei de date, a protocolului de comunicare, chiar și a game design-ului.

2.1.2 Pikko Server

O altă arhitectură care propune o soluție de a scala arhitectura client-server folosită adeseori în prezent pentru jocurile MMO. Problemele enunțate ale arhitecturilor clasice client-server sunt legate de faptul că acesta nu suportă o densitate mare a jucătorilor, sau zone de lupte de proporții mari, aceste lucruri fiind posibile doar prin diverse mecanisme cum ar fi instanțierea jucătorilor în mai multe zone, având o limită de câteva sute de jucători într-o zonă, prin dilatarea timpului (Eve Online) sau prin interzicerea luptelor între jucători în zone cu densitate mare – toate lucrurile acestea duc la o experiență plictisitoare.

Soluția oferită de către Pikko server[3] constă în adăugarea unui ”middleware” care este situat între clienți și servere, rolul acestuia este de a ruta traficul jucătorilor către servere specifice bazându-se pe poziția acestora. Algoritmul presupune vizualizarea lumii virtuale ca o lume compusă din antene (servere) și telefoane (jucători), diferența fiind că antenele se pot mișca prin lume.

Un mecanism de înmânare este implementat, folosit atunci când un jucător se îndepărtează de serverul care primea traficul acestuia, fiind acum necesar ca jucătorul să fie alocat altui server – acest lucru se va face prin serializarea stării jucătorului și transferul acesteia între servere.

Datele raportate susțin că arhitectura este capabilă să susțină 1000 de jucători conectați simultan, serverele trimițând 15 actualizări pe secundă.

2.1.3 Kiwano

Kiwano (Diaconu & Keller, 2013) este o arhitectură distribuită care permite interacțiunea a unui număr nelimitat de jucători (în limita resurselor). Arhitectura obține acest lucru printr-o împărțire dinamică eficientă a zonelor de interes ale jucătorilor, prin

[2] https://www.eveonline.com/news/view/introducing-time-dilation-tidi
[3] http://www.erlang-factory.com/upload/presentations/297/PikkoServerErlangconference.pdf

calculul triangulării Delaunay între pozițiile jucătorilor pentru a furniza un număr constant de vecini acestora, indiferent de densitatea sau distribuția jucătorilor.

Calculul și actualizarea grafului Delaunay se face într-o manieră distribuită, fiecare server din rețea având un nod din graf, găzduind un număr de jucători. Arhitectura are soluții pentru multiple provocări în recalcularea eficientă a grafului, de la adăugarea unui nou jucător până la adăugarea unui nou server pe baza procentului de încărcare a două servere învecinate.

Recalcularea grafului se face într-o manieră periodică, de 10 ori pe secundă, primind mesajele de actualizare a pozițiilor jucătorilor și procesându-le în grupuri.

Din nou, există un mecanism de înmânare atunci când în urma recalculării grafului un jucător trebuie procesat de alt server.

Arhitectura se bazează pe un API care este apelat de către clienți. În urma apelurilor API-ului, aceștia se vor conecta la un server reprezentativ, fiecare server reprezentativ ocupându-se de o anumită parte din utilizatorii conectați. Aceste servere reprezentative fac posibilă comunicarea între jucător și serverul care calculează nodul din graful Delaunay pentru jucător și vecinii acestuia.

Performanțele atinse de această arhitectură sunt foarte bune, putând simula interacțiuni între 22000 jucători cu actualizări de poziție la 0.5 secunde, pe 24 noduri de rețea, scalabilitatea fiind liniară după un anumit punct.

2.2 Studiul tehnicilor folosite în domeniu

În acest capitol sunt relatate cele mai des întâlnite tehnici folosite în domeniul jocurilor MMO și în domeniul procesării GPGPU.

2.2.1 Arhitectura Client-Server

Această arhitectură are ca scop distribuirea aplicației de tip client către jucători, asigurarea gestiunii și infrastructurii serverului rămâne în mâinile firmei care se ocupă de joc.

Este cel mai popular tip de arhitectură în rândul jocurilor MMO (Chambers, Feng, & Feng, 2006), în special datorită controlului asupra stării jocului care vine alături de o arhitectură centralizată - serverul se ocupă de actualizarea fiecărui jucător și a interacțiunilor acestora, controlând starea globală a jocului. Aplicația client poate interoga serverul pentru a-și actualiza starea. De asemenea, deoarece serverul este principala componentă, întreținerea și modificarea se poate face destul de ușor, simplitatea acestei arhitecturi oferind un mare avantaj față de alte arhitecturi disponibile.

Câteva din responsabilitățile serverului:

- Calcularea coliziunilor sau distanța între jucător și NPC-uri (Non-Playable Character) sau alți jucători
- Determinarea schimbărilor de stare ale jucătorilor sau ale lumii (de exemplu schimbarea poziției jucătorilor)
- Notificarea jucătorilor în legătură cu diverse lucruri (de exemplu dacă a primit un mesaj de la alt jucător sau dacă a crescut în nivel etc.)
- Stocarea în mod persistent a stării jucătorilor pentru ca aceștia să continue să existe în lumea virtuală
- Detectarea tentativelor de trișare ale jucătorilor

Un alt avantaj pe care îl are această arhitectură este persistența, starea jocului fiind deseori salvată de către server, astfel un jucător nu-și va pierde progresul nici după ce conexiunea către server a fost pierdută.

Totuși, există câteva dezavantaje destul de mari în cadrul acestei arhitecturi, cea mai importantă fiind problema scalabilității, un server având o capacitate de câteva mii de jucători în lume, iar în cazul în care aceștia se condensează într-o singură regiune poate apărea fenomenul de ”lag” (întârzieri la actualizarea stării jocului), serverul devenind suprasolicitat din punct de vedere computațional, el fiind responsabil de calculul actualizărilor pentru fiecare jucător. Un alt dezavantaj este costul menținerii infrastructurii, fiind nevoie de servere performante care să poată menține conexiuni cu mii de jucători conectați simultan fără a impune întârzieri la acțiunile acestora, pe lângă cerințele de lățime de bandă necesare pentru procesarea tuturor mesajelor primite și trimise de către server.

În cazul în care jocul MMO trebuie să suporte o bază largă de jucători, se poate folosi o abordare în care se folosesc mai multe servere. Astfel, lumea jocului se va împărți în mai multe regiuni, fiecare server controlând o regiune. În acest fel, se distribuie costul computațional print acest mecanism, arhitectura devenind scalabilă. Totuși, există și dezavantaje, jucătorii neputând comunica cu jucători din alte regiuni, dar existând mecanisme de transfer a informațiilor jucătorilor între servere pentru a combate acest lucru în anumite situații.

În principiu, în cazul în care este nevoie de scalabilitate și se folosește o arhitectură client-server cu mai multe servere, există mai multe tipuri de responsabilități specifice unui server, câteva exemple fiind:

- Un server poate copia întreaga lume și va fi situat în altă zonă geografică, jucătorul alegând cel mai apropiat server de locația sa din punct de vedere geografic pentru a avea cel mai bun timp de răspuns la acțiuni în timpul jocului
- Un server poate copia întreaga lume cu rol de redundanță - în general este util pentru a crește toleranța la defecte a arhitecturii, dar poate fi folosit și pentru a împărți un influx mare de jucători, jucătorii conectându-se la acest server atunci când serverul principal atinge o limită de jucători de exemplu

- Un server poate copia doar o parte din lume, având același scop ca serverul care copiază întreaga lume, diferența fiind în resursele consumate, această abordare fiind deseori mai eficientă în cazul în care jucătorii sunt condensați doar în anumite zone din lume
- Un server se poate ocupa de autentificarea jucătorilor și transferul acestora către servere de joc

Folosind mai multe servere, arhitectura poate deveni tolerantă la defecte, putând avea servere de backup care pot citi starea jocului din zona de stocare persistentă folosită și pot relua jocul de la momentul în care serverul principal a devenit neresponsiv.

Un factor care conferă un avantaj major acestei arhitecturi față de celelalte și unul din motivele pentru care această arhitectură este preferata companiilor de jocuri MMO este securitatea, factorul de trișare al jucătorilor fiind foarte diminuat datorită controlului centralizat. Jucătorii care respectă regulile jocului pot deveni frustrați din cauza jucătorilor care trișează - dacă situația iese de sub control și jocul devine nedrept, foarte mulți jucători vor părăsi jocul, iar alții vor recurge și ei la trișare. Acest lucru poate duce până la conceptul de ”joc mort”, în care populația activă de jucători este mult prea mică, ducând la oprirea întreținerii serverului de către compania de jocuri.

2.2.2 Arhitectura Peer-To-Peer

Factorii cei mai importanți care participă la succesul unui joc de tip MMO sunt scalabilitatea, timpul de răspuns la acțiunile posibile din timpul jocului și costurile de mentenanță - arhitectura jocului având un rol critic în felul în care aceste cerințe sunt îndeplinite.

Avantajul arhitecturii peer-to-peer (Yahyavi & Kemme, 2013) este faptul că oferă o scalabilitate dinamică, ridicată, la un cost foarte redus – fiecare jucător contribuind cu resursele sale. De asemenea, un timp foarte bun de răspuns la acțiunile jucătorilor poate fi realizat prin crearea eficientă de conexiuni directe între jucători, nemaifiind nevoia unui drum dus-întors între jucător și server, precum în cazul arhitecturii client-server.

Totuși, ca acest tip de arhitectură să funcționeze pentru jocuri de tip MMO, există anumite probleme complexe care trebuie să fie rezolvate. Față de arhitectura client-server, unde o singură entitate deține controlul asupra stării in care se află jocul pentru toți jucătorii, în arhitectura peer-to-peer acest lucru devine mult mai complex, jocul devenind distribuit, fiind nevoie de o soluție prin care jucătorii să se poată sincroniza cu starea globală a jocului. De asemenea, o altă problemă care apare în urma pierderii entității care menține controlul asupra jocului este securitatea – din moment ce orice nod din rețeaua distribuită poate avea control asupra unei părți din starea jocului, apare o vulnerabilitatea ridicată la trișat din partea jucătorilor.

Pe lângă abordarea pur peer-to-peer a arhitecturii, mai există și abordări hibride, în care se combină elemente din arhitectura peer-to-peer cu elemente din arhitectura client-server. Aici pot exista mai multe categorii, în funcție de ce rol va realiza serverul:

- Starea jocului poate fi menținută de către server, iar actualizările se fac în manieră distribuită - fiecare jucător trimite mesaje cu interacțiunile sale către server, acestea sunt procesate laolaltă, după care serverul trimite actualizările în manieră multicast către jucători.
- Starea jocului poate fi distribuită către jucători, aceștia fiind responsabili de executarea anumitor acțiuni, rolul server-ului fiind de a controla mesajele (actualizările) dintre aceștia, autentificarea lor, contorizează când jucătorii se conectează sau ies și alte acțiuni asemănătoare.
- Atât starea jocului cât și metoda de actualizare a stării dintre jucători este distribuită, rolul server-ului fiind de a păstra date importante cum ar fi informații despre autentificare pentru fiecare jucător, informații de plată etc. De asemenea, serverul se mai poate ocupa de autentificare, de contorizare a jucătorilor care se conectează sau care ies și de alte tipuri de interacțiuni de organizare a jucătorilor.

În mod general, o rețea peer-to-peer este creată în mod dinamic, noduri noi putând fi adăugate în rețea în orice moment. Pentru a asigura o scalabilitate ridicată, rețeaua este împărțită în subrețele, un nod cunoscând doar un număr limitat din totalitatea nodurilor existente în rețea. Există doua categorii de rețele peer-to-peer, în funcție de modul în care sunt construite conexiunile dintre noduri:

- Rețele peer-to-peer structurate, în care nodurile sunt așezate sub formă de graf prin aplicarea unui protocol. Acest mod de organizare permite schimbul de mesaje între oricare nod din rețea într-o manieră eficientă, prin folosirea unei tabele de dispersie distribuită (distribuită peste nodurile din rețea) pentru a face rutarea între acestea.
- Rețele peer-to-peer nestructurate, în care nodurile nu au o organizare anume, conexiunea dintre acestea făcându-se în manieră probabilistică: fie total aleatoriu, fie folosind anumite euristici pentru optimizarea legăturilor, cum ar fi conectarea nodurilor mai apropiate etc. Căutarea în astfel de rețele se face tot în manieră probabilistică, căutarea putând fi accelerată prin replicarea conținutului în mai multe noduri, sau prin mecanisme de inundare a mesajelor.

În cazul arhitecturii peer-to-peer pentru jocuri MMO, pentru rețelele structurate, cheile din tabela de dispersie distribuită pot fi ID-uri de obiecte (valoare hash a obiectului) și ID-uri de noduri. Fiecare nod va fi responsabil de actualizarea anumitor obiecte, mai exact de obiectele ale căror ID-uri sunt cele mai apropiate de ID-ul nodului respectiv. Astfel, când un nod oarecare din rețea va trebui să actualizeze un anumit obiect din lume, va folosi

pentru cheia de căutare în tabela de dispersie distribuită valoarea hash a obiectului, va afla nodul responsabil de obiectul respectiv și va putea trimite astfel actualizările.

În rețelele nestructurate, actualizările se fac pe baza detectării nodurilor vecine din zona de interes și trimiterea actualizărilor direct către aceștia. Acest lucru se poate face prin menținerea unei liste de noduri vecine în fiecare nod, aceasta fiind actualizată de fiecare dată când un jucător intră sau pleacă din zona de interes. Pentru ca acest lucru să funcționeze, fiecare jucător se bazează pe vecinii săi pentru actualizări: când un jucător trimite actualizări către un nod vecin din zona sa de interes, acest vecin redirecționează mesajul și către nodurile sale vecine, astfel actualizându-se lista de vecini pentru fiecare jucător.

De asemenea, există și abordări hibride, care pot combina rețelele structurate și cele nestructurate, având avantajele ambelor tipuri de rețele în schimbul overhead-ului de memorie, de lățime de bandă și de putere de procesare pentru a rula cele 2 sisteme în același timp.

Comunicarea între nodurile din arhitectura peer-to-peer poate fi făcută în două moduri:

- Nodul care deține un obiect trimite actualizările către nodurile interesate de acel obiect în mod direct. Avantajele acestei abordări sunt latența scăzută și modul simplu de implementare, dar dezavantajul principal fiind lățimea de bandă necesară în anumite cazuri, de exemplu dacă un jucător are un obiect care este de interes pentru mai mulți jucători, el va trebui să trimită actualizări către fiecare în parte. De asemenea, viteza de upload poate fi o problemă, furnizorii de servicii de internet de multe ori oferă o viteză asimetrica, în care viteza de upload este mai mică decât viteza de download. Există mecanisme prin care se poate optimiza comunicarea directă, care se ocupă fie de reducerea dimensiunii pachetelor, fie de distribuirea mesajelor care trebuie trimise către alte noduri din rețea care pot trimite mesajele mai departe, împărțind lățimea de bandă necesară către mai multe noduri.
- Crearea unui arbore multicast și trimiterea unui mesaj de la rădăcină către copii, lățimea de bandă necesară pentru un nod fiind mult mai mică decât la comunicarea directă, fiecare nod având doar câțiva copii. Dezavantajul acestei metode este o creștere a latenței, deoarece mesajul va trece prin mai multe noduri până va ajunge la destinație. Există optimizări și pentru această abordare: crearea mai multor astfel de arbori în mod dinamic în funcție de regiunile din joc, alegerea nodurilor copii astfel încât aceștia să fie cât mai apropiați din punct de vedere geografic pentru a scădea latența livrării pachetelor între nodurile intermediare și altele.

Gestionarea zonelor de interes pentru rețelele structurate se face de obicei în mod static, divizând lumea jocului în regiuni, un nod putând trimite actualizările prin multicast către toți jucătorii care se află în regiunea respectivă, iar de fiecare dată când un jucător va schimba regiunea în care se află, va exista un mecanism de preluare a informațiilor specifice acestuia către noua regiune. Dezavantajul acestei abordări este că un jucător va primi de multe ori mesaje fără interes pentru el - lucru care poate fi rezolvat prin împărțirea regiunilor în regiuni mai mici - totuși, și această abordare are o problemă, astfel că mecanismul de preluare se va executa de mai multe ori.

În rețelele nestructurate, gestionarea zonelor de interes se face în mod dinamic, acestea fiind actualizate prin schimbul de mesaje între nodurile învecinate. Există mai multe abordări aici, principala diferența fiind forma zonei de interes, care poate fi un simplu cerc sau o formă mai complexă, detecția nodurilor învecinate fiind diferită în funcție de forma folosită pentru zona de interes.

În arhitecturile peer-to-peer, fiecare entitate din joc este întreținută de unul dintre "peers", acesta având o copie primară a entității. Există mai multe modalități de distribuire a acestor copii și moduri diferite de a trimite actualizări către alte noduri interesate de copiile primare. De exemplu, un nod din rețeaua peer-to-peer poate avea rolul de coordonator într-o anumită regiune, acesta fiind responsabil de primirea cererilor de actualizare pentru obiecte și de trimiterea actualizărilor către alte noduri situate în aceeași regiune. În aceste tipuri de arhitectură, se caută împărțirea regiunilor în regiuni mici, astfel încât un nod coordonator să nu fie suprasolicitat.

Având în vedere consistența stării jocului, diferite tipuri de jocuri necesită nivele diferite de control al consistenței. Cu cât nivelul de consistență necesar este mai restrictiv, cu atât mecanismele necesare să asigure consistența sunt mai costisitoare. Adeseori, totuși, nivelul de consistență necesar este unul relaxat, pentru a nu afecta foarte mult timpul de răspuns la acțiunile jucătorilor. Datorită acestui lucru, se folosesc mecanisme de rezolvare a conflictelor, atunci când există inconsistențe între stările a două noduri.

Toleranța la defecte pentru aceste arhitecturi este un factor important, atunci când un nod se deconectează de la rețea în mod neașteptat pot dispărea informațiile referitoare la starea jocului care erau întreținute de nodul respectiv, pe lângă posibila stricare a structurilor de date folosite, cum ar fi arborii de multicast, sau tabelele de dispersie distribuite cu informații de rutare. Totuși, față de rețelele generice peer-to-peer, în cazul jocurilor MMO defectele sunt mai puțin destructive:

- Jucătorii sunt foarte răsfirați din punct de vedere geografic, astfel că defectarea unui număr mare de noduri în același timp este puțin probabilă
- Pentru unele tipuri de jocuri există pedepse în cazul în care jucătorul a ieșit din joc înainte de terminarea meciului, astfel că maniera în care nodurile părăsesc rețeaua este controlată
- Adeseori jocurile MMO nu caută o consistență strictă, astfel că există deja mecanisme pentru rezolvarea inconsistențelor

- Există mecanisme de tip ”heartbeat” care asigură reconstrucția structurilor de date folosite de rețea

Problema principală care poate apărea la deconectarea unui nod este pierderea copiilor primare ale obiectelor care erau întreținute de nodul respectiv. Astfel, pentru a rezolva această problemă, sunt implementate mecanisme de backup, prin care starea unui nod va fi replicată în mai multe noduri, în funcție de diverși parametri, sau chiar în mod dinamic, existând sisteme care se ocupă cu administrarea redundanței stării jocului.

Un alt mod prin care se poate crește nivelul toleranței la defecte a unui joc de tip MMO este prin persistența stării jocului. Există o multitudine de sisteme de fișiere distribuite care pot fi folosite pentru a stoca informațiile, atât pentru arhitecturile structurate, cât și pentru cele nestructurate, deși cele din urmă obțin persistența prin diferite metode de replicare a datelor, nu prin sisteme de fișiere distribuite tipice. Informațiile care sunt stocate depind de tipul jocului și de arhitectură, de multe ori fiind stocate statistici despre jucător de exemplu. În arhitecturile hibride se pot stoca mai multe informații despre starea jocului, de multe ori serverul fiind responsabil de persistența stării. Acest lucru este destul de simplu în cazul în care serverul se ocupă direct de starea jocului și acesta transmite actualizările către jucători. Dar lucrurile pot deveni mai complexe în cazul în care jucătorii dețin copiile primare ale entităților din joc, serverul fiind nevoit să adune actualizările de la jucători. Aici, în funcție de nivelul de persistență dorit, serverul poate cere actualizări în mod frecvent de la jucători și poate stoca informațiile de stare în mod persistent, crescând astfel toleranța la defecte a rețelei.

Jucătorii care trișează reprezintă cea mai mare problemă a sistemelor peer-to-peer, jocul devenind frustrant pentru jucătorii care respectă regulile. Există mai multe categorii de trișat, câteva dintre acestea fiind:

- Din moment ce într-o rețea distribuită fiecare jucător este responsabil cu actualizarea stării jocului, acesta poate trimite informații falsificate către server sau către alți jucători (în funcție de arhitectură) pentru ca acesta să fie avantajat. Tot aici pot intra și atacurile de rețea denial-of-service, prin care se caută o suprasolicitare a server-ului, astfel încât jocul să nu mai poată fi jucat din cauza nereceptivității severului la acțiunile jucătorilor.
- Nerespectarea regulilor jocului prin modificarea codului aplicației client, căutând avantaje precum modificarea atributelor caracterului cum ar fi viteza de deplasare. De asemenea, protocoalele de consistență folosite de către arhitectură pot fi abuzate prin trișare, trimițând actualizări diferite către jucători diferiți, ducând la o stare a jocului inconsistență.
- Citirea zonelor de memorie și a pachetelor transmise în rețea pentru a capta informații despre starea jocului la care utilizatorul nu ar trebui să aibă acces în mod normal – de obicei informații despre poziția celorlalți jucători.

Există metode de prevenție a trișatului, dar doar pentru anumite categorii în anumite situații, fie în mod proactiv prin reducerea sau eliminarea posibilității trișării sau în mod reactiv prin detectarea situațiilor în care jucătorul trișează. Astfel, în mod proactiv se poate face:

- Criptarea mesajelor pentru a elimina problema citirii pachetelor.
- Adăugarea de protocoale de verificare la transmiterea actualizărilor de la un jucător la altul – scopul aici este de a diminua sau chiar elimina anumite atacuri care aveau în vedere abuzul nivelului consistenței jocului
- Alte tehnici prin care informațiile jucătorilor sunt mai întâi transmise către noduri intermediare pentru a diminua trișarea având acces direct la informații precum viteza jucătorilor, poziția lor etc.

Ca metode de verificare a trișatului, de obicei se caută alegerea unui arbitru, care, în funcție de arhitectura folosită, poate fi fie un server care va fi responsabil de verificarea actualizărilor jucătorilor dintr-o anumită zona dacă au sens din punct de vedere al fizicii jocului sau nu, fie un alt jucător ales în mod aleatoriu pentru a reduce șansa ca acesta să fie un complice. În cazul în care arbitrul este un jucător aleatoriu, trebuie implementat și un mecanism de reputație, deoarece pot exista cazuri în care deși un jucător a fost detectat ca fiind trișor, acesta să fi fost detectat în mod eronat. Astfel, la detectări repetate ale aceluiași jucător, reputația acestuia scade, în cele din urmă acesta nu va mai avea voie să joace.

2.2.3 Programarea GPGPU

Programarea GPGPU se referă la utilizarea capabilităților plăcilor video pentru calcule, ele fiind folosite în mod normal pentru grafica pe calculator. Avantajul folosirii plăcii video pentru computare față de microprocesor este capabilitatea de procesare în mod paralel a datelor, conform arhitecturii lor.

Framework-urile cele mai folosite pentru GPGPU sunt CUDA (NVIDIA, 2022), dezvoltat de NVIDIA și OpenCL[4], dezvoltat de Khronos Group. O diferența între acestea este faptul că NVIDIA CUDA este un framework proprietar, având suport doar pentru plăcile video NVIDIA, pe când OpenCL este un framework open source, cu suport pentru diverse tipuri de hardware. CUDA include multe pachete științifice pentru multe tipuri de probleme, în cazul OpenCL neavând biblioteci standard, dar există multe biblioteci ”third-party” care se pot folosi pentru același scop.

Pentru programator, framework-urile de GPGPU oferă o nouă paradigmă de programare, cu care acesta trebuie să se familiarizeze, dar pentru început, bibliotecile care sunt disponibile ajută cu mult în procesul de folosire a acestora.

[4] https://www.khronos.org/opencl/

Paradigma programării pe GPU este restrictivă din mai multe puncte de vedere, scopul inițial al GPU-urilor fiind să facă procesări grafice, pe seturi de date care pot să fie procesate ușor în mod paralel, prin procesare în flux. În principiu, aceste framework-uri funcționează în felul următor:

- Identificarea device-urilor (GPU) disponibile
- Crearea contextului și a buffer-elor care vor fi folosite
- Copierea în memoria GPU a datelor folosite din memoria RAM
- Înștiințarea API-ului să folosească un anumit kernel (practic, o funcție care se va executa în paralel pe fiecare nucleu al plăcii video)
- Executarea kernel-ului
- Copierea rezultatului înapoi în memoria RAM

O altă diferență între CUDA și OpenCL este în legătura cu modul în care este compilat kernel-ul. Aici CUDA are un avantaj, deoarece suportul acestei platforme este doar pentru plăci video NVIDIA. Avantajul pe care îl are CUDA este în faptul că un kernel este compilat în cod PTX alături de program, care este un limbaj low-level, foarte apropiat de codul assembly specific dispozitivului NVIDIA care folosește kernel-ul. Fiind un limbaj low-level, în CUDA există un overhead foarte mic la compilarea run-time a kernel-ului, pe când în OpenCL kernel-ul este compilat direct din cod C, care are un overhead mai mare, fiind o discrepanță la timpul de rulare între cele două limbaje pentru probleme de dimensiuni mai mici. De asemenea, o precompilare a kernel-ului nu este posibilă în OpenCL, acesta fiind agnostic față de hardware-ul pe care rulează, pe când în CUDA este posibil acest lucru.

Ca limbaje de programare suportate, pe platforma CUDA se poate folosi C, C++ sau Fortran, în timp ce limbajul suportat de OpenCL este un limbaj de programare bazat pe C99, adaptat astfel încât să se potrivească modelului folosit de acesta.

S-au făcut studii (Demidov, Ahnert, Rupp, & Gottschling, 2013) în care s-au comparat vitezele de execuție a mai multor tipuri de probleme pe CPU și pe GPU, folosind CUDA și OpenCL. În principiu, pentru probleme mici, overhead-ul copierii datelor în memoria GPU este mai semnificativă față de viteza pe care o oferă procesarea paralelă, astfel că performanțele sunt în general mai bune folosind CPU. Totuși, pentru probleme mai mari, programele GPGPU oferă un factor de accelerare între 10-20 ori. Există o foarte mică diferență de performanță între cele două platforme și anume o diferență de câteva procente în favoarea CUDA, dar în practică acest lucru nu este un factor relevant la alegerea platformei GPGPU. Mai degrabă, API-ul și faptul că OpenCL suportă mai multe platforme ar fi factori mai adecvați de comparație.

2.2.4 Detecția coliziunilor în paradigma GPGPU

Problema detectării coliziunilor este o problemă fundamentală în simularea corpurilor dintr-o lume virtuală. Rezolvarea problemei folosind tehnici GPGPU duce la posibilitatea simulării în timp real a interacțiunilor unui număr mare de obiecte din scenă.

Problema detectării coliziunilor se rezumă la detectarea perechilor de obiecte care interacționează la un moment dat și aplicarea unei acțiuni specifice asupra acestora la momentul de timp următor. În modul cel mai simplu, rezolvarea acestei probleme se poate face prin testarea coliziunii dintre fiecare pereche de obiecte. Totuși, fiecare corp se ciocnește mai întâi cu corpurile cele mai apropiate de acesta, astfel putând dezvolta algoritmi mai eficienți, prin folosirea de structuri de date pentru partiționarea spațiului.

În practică, aplicațiile acestui gen de problemă sunt în simularea obiectelor rigide și deformabile, simularea hainelor, simularea interacțiunii dintre n corpuri etc., dorindu-se o viteză cât mai mare a simulării, cercetările curente axându-se foarte mult pe paralelizare. Totuși, problema detectării coliziunilor nu este ușor paralelizată, corpurile nefiind distribuite în mod egal.

Detectarea coliziunii se poate face în doua moduri:

- ”Discrete collision detection” (DCD) - detectarea coliziunii se face la poziția curentă a perechii de obiecte în fiecare cadru, existând posibilitatea de nedetectare a coliziunii în cazul în care un obiect are o viteză foarte mare – fenomenul se numește ”tunneling”
- ”Continuous collision detection” (CCD) - detectarea coliziunii se face prin calcularea punctului intermediar de coliziune dintre 2 obiecte

Mai departe, indiferent de modul de detectare a coliziunilor, există 2 etape:

- ”Broad phase” - în această fază se înlătură obiectele care nu vor interacționa, rezultatul fiind o listă cu obiectele care au potențial de coliziune
- ”Narrow phase” - în această fază se preia rezultatul etapei anterioare și se execută teste de coliziune exacte, costisitoare

Etapa broad phase se poate îndeplini în mai multe moduri, dar o metodă eficientă de a face acest lucru este prin folosirea unui algoritm de subdivizare a spațiului.

Există mai multe tipuri de structuri de date pentru partiționarea spațiului care pot fi folosite pentru etapa broad phase. Câteva exemple ar fi:

- Octrees
- Uniform grids
- Bounding Volume Hierarchy (BVH)
- K-d tree

O atenție mai mare în cercetările curente este pusă asupra structurilor de date BVH și uniform grids.

Există mai multe metode pentru construcția unui BVH, fie în manieră top-down sau bottom-up:

- Metodele top-down pornesc de la setul de obiecte de intrare și fac o împărțire a acestuia în două sau mai multe subset-uri în mod recursiv. Aceste metode sunt mai populare și mai ușor de implementat dar nu produc o calitate foarte bună a BVH-ului rezultat.
- Metodele bottom-up pornesc de la nodurile frunză și fac gruparea a două sau a mai multor astfel de noduri în noduri interne, algoritmul continuând în aceeași manieră până la crearea nodului rădăcină. Metodele bottom-up sunt mai dificil de implementat, dar de multe ori oferă o calitate a BVH-ului mult mai bună față de metodele top-down.

BVH poate fi folosit pentru scene dinamice, această structura suportă în mod natural actualizarea nodurilor din ierarhie. Totuși, calitatea BVH-ului se poate deteriora în urma actualizărilor, la un moment dat fiind necesară o reconstruire a ierarhiei.

Există metode de paralelizare GPGPU atât pentru construcția BVH-ului, pentru actualizări cât și pentru traversarea acestuia. O metodă foarte eficientă de construcție și parcurgere a BVH-ului folosind în întregime programarea GPGPU este cea dezvoltată de Karras et al. (Karras, Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees, 2012), care oferă un timp de construcție foarte rapid și o calitate foarte bună a BVH-ului construit.

Uniform grid-ul presupune subdivizarea spațială a lumii virtuale în celule de dimensiune egală, astfel încât o celulă din acest grid nu este mai mică decât volumul încadrator al celui mai mare obiect care participă la coliziune. Fiecare celulă din grid conține o listă cu obiectele a căror volum încadrator le intersectează, astfel că în spațiul 3D un obiect poate să fie în lista a cel mult 8 celule în același timp. Calculul de coliziune se va petrece doar între 2 obiecte care aparțin de aceeași celulă, în care cel puțin unul dintre obiecte își are situat centroidul.

Algoritmul de construcție și de verificare a coliziunilor între celule pentru grid-ul uniform poate fi paralelizat folosind GPGPU[5], existând anumite condiții de cursă care apar datorită procesării paralele, dar care pot fi soluționate.

Pentru simulări mai exacte, după ce au fost procesate coliziunile din etapa broad phase, se poate continua cu etapa narrow phase, care poate fi executată tot pe GPU, testând în mod exact coliziunea pentru toate obiectele suspecte de coliziune întoarse de etapa broad phase, putând folosi mai departe informațiile din coliziune pentru a aplica diferite forțe de fizică obiectelor, cum ar fi forța de impuls.

[5] **https://www.kalojanov.com/data/gpu_grid_construction.pdf**

3 TEHNOLOGII FOLOSITE

În acest capitol sunt relatate câteva informații în legătură cu tehnologiile principale folosite în prototip, alături de o detaliere scurtă a modului în care sunt folosite în aplicație.

3.1 Unity

Unity[6] este un motor de joc cross-platform folosit pentru a crea jocuri 2D și 3D, pe lângă alte tipuri de aplicații de grafică pe calculator.

Unity este o aplicație gândită să facă crearea de jocuri mai rapidă și mai ușoară, având integrat un editor vizual cu un UI foarte util prin care se pot vedea schimbările din starea jocului în timp real, menit să accelereze prototiparea jocurilor. Arhitectura motorului este bazată pe componente, obiectele din joc fiind compuse din mai multe astfel de componente, lucru care ajută la modularizare, refolosire și extensibilitate.

Acest motor de joc vine cu o gamă largă de sisteme utile folosite în mod comun de către jocuri. Câteva exemple de astfel de sisteme:

- Randare
- Fizică
- Input
- Animații
- Audio
- AI
- Networking
- Asset manager

Un dezvoltator poate crea noi componente prin script-uri C#, având acces la funcționalitățile de bază oferite de către bibliotecile standard din C#, pe lângă funcționalități extra oferite de către Unity, plus acces la alte componente deja definite.

Pe lângă capabilitățile de prototipare rapidă deduse din informațiile redate anterior, Unity a fost ales și din motive de experiență de dezvoltare folosind acest motor de joc, pentru a nu pierde prea mult timp cu învățarea unei tehnologii noi.

În acest prototip, Unity este folosit pentru aplicația client. Prototiparea s-a făcut în mod rapid, fiind necesară crearea a câtorva componente cu diferite funcționalități, cum ar fi:

- Simularea jucătorului și a acțiunilor acestuia
- Păstrarea obiectelor dinamice într-o structură de date care oferă acces rapid folosind ID-ul acestora
- Codificarea și decodificarea pachetelor folosite în protocolul de comunicare
- Comunicarea între client și server

[6] **https://unity.com/**

- Aplicarea actualizărilor primite de la server prin schimbarea pozițiilor obiectelor dinamice
- Dispunerea obiectelor statice în scenă
- Urmărirea jucătorului în manieră ”third-person” folosind camera principală

Folosind editorul vizual din Unity, se pot schimba diverși parametri din aceste componente în mod rapid, cum ar fi distanța camerei față de jucător, numărul de autentificări efectuate, numărul de obiecte statice din scenă și altele.

3.2 CUDA

CUDA (NVIDIA, 2022) este un SDK pentru C/C++ oferit de către NVIDIA, care are capabilitatea de a folosi un GPU CUDA pentru programare GPGPU.

Având în vedere puterea masivă de procesare paralelă a GPU-urilor, folosirea tehnologiilor GPGPU se pretează foarte bine pe probleme de dimensionalitate mare, care pot fi împărțite în probleme mai mici (astfel ca fiecare nucleu de pe GPU să rezolve o problemă de dimensiune mai mică în paralel, reducându-se astfel timpul de procesare semnificativ).

Modul de funcționare al framework-ului CUDA este următorul:

- Copierea datelor din memoria CPU către memoria GPU
- Rularea programului (kernel) pe GPU
- Copierea datelor din memoria GPU către memoria CPU

Rularea kernel-ului pe GPU se face în felul următor:

- Se creeaza thread-uri
- Thread-urile sunt grupate în blocks
- Block-urile sunt grupate într-un grid
- Kernel-ul este executat ca un grid, compus din block-uri de thread-uri
- Fiecare thread rulează kernel-ul, la rularea kernel-ului programatorul având acces la ID-ul thread-ului curent/block-ului curent pentru a putea controla împărțirea în subprobleme

CUDA vine la pachet cu un profiler care este foarte folositor în pasul de optimizare a rulării programului GPGPU, de multe ori acest pas fiind necesar pentru a avea o performanță bună, fiind mulți factori care pot contribui la acest lucru – mai ales din punct de vedere al utilizării eficiente a memoriei.

În prototip, CUDA este folosit pe server pentru a paraleliza calculul detectării coliziunilor între jucători, pentru aplicarea forțelor de fizică și pentru gestionarea zonelor de interes pentru jucători. De asemenea, biblioteca CUB (NVIDIA, 2022) este folosită pentru rularea anumitor algoritmi, această bibliotecă oferind kernel-uri optimizate în CUDA, cu performanțe state-of-the-art.

3.3 Berkeley Sockets

Este un API[7] generic de rețea furnizat de către sistemul de operare, folosit pentru comunicarea inter-proces.

Un ”socket” este o abstractizare pentru un terminal dintr-o rețea. În Linux, un socket este echivalent cu un file descriptor, fiind o interfață comună atât pentru input cât și pentru output către un flux de date.

API-ul oferă suport pentru conexiuni de tip UDP și de tip TCP, la crearea socket-ului. Există mai multe funcții în acest API, de tipul creare și închidere de sockets, de acceptare de conexiuni, de primire/trimitere de date și altele.

Există două moduri în care aceste socket-uri funcționează:

- Mod blocant, astfel că socket-ul blochează thread-ul din care a fost apelată funcția de trimitere sau de primire de date până se primesc (sau se trimit) un număr de octeți - este modul standard de funcționare
- Mod non-blocant, în care socket-ul nu blochează thread-ul, în cazul apelării funcției de primire de date returnează ce exista în momentul respectiv în buffer-ul de primire – pot exista condiții de cursă aici în cazul în care logica nu este bună

În cazul prototipului, se folosesc Berkeley Sockets atât pe server cât și pe client, astfel:

- Pe server, este folosit API-ul pentru sockets furnizat de către Windows în limbajul C++. API-ul este încapsulat într-o clasă pentru modularizare, iar înainte de pornirea socket-ului este setat port-ul pe care acesta o să primească pachete de la client. Protocolul folosit este UDP, iar modul de funcționare al socket-ului este non-blocant, operațiile de receive/send fiind apelate într-o anumită ordine, neexistând condiții de cursă între acestea
- Pe client, este folosit API-ul pentru sockets furnizat de către biblioteca standard din C#. Din nou, protocolul folosit este UDP, modul de funcționare fiind blocant, având un thread separat care se ocupă de primirea de actualizări de la server. Pe thread-ul principal se trimit mesaje cu acțiunile jucătorului către server în mod blocant, în fiecare frame

[7] https://inst.eecs.berkeley.edu/~ee122/sp07/Socket%20Programming.pdf

3.4 Transport Layer Security

Transport Layer Security (Rescorla, 2018) este un protocol foarte popular de securitate, folosit adeseori în comunicarea din Internet, fiind protocolul de securitate folosit de protocolul HTTPS.

Acest protocol funcționează peste protocolul TCP, partea de criptare fiind implementată folosind criptografie cu chei publice, iar, pentru a asigura integritatea mesajelor, se adaugă un cod care poate fi verificat la recepția mesajului, pachetul fiind alterat față de original în cazul în care verificarea eșuează.

În cazul prototipului, serverul generează un certificat care va fi folosit în pasul de autentificare al protocolului TLS, certificatul conținând informații despre server și o cheie publică care va fi folosită pentru criptare. După ce conexiunea între client și server a reușit, aceștia comunică pe baza unui cifru, mesajele fiind transmite peste un canal securizat.

3.5 MySQL

MySQL (MySQL, 2022) este unul dintre cele mai populare sisteme de gestiune ale bazelor de date relaționale, existând deja un nivel de experiență acumulată folosind această tehnologie.

MySQL oferă mai multe drivere în diverse limbaje de programare care permit comunicarea cu baza de date, driver-ul folosit în aplicația server fiind MySQL Connector (MySQL, 2022).

3.6 Limbajul C++, limbajul Python și limbajul C#

Cele trei limbaje folosite în prototip, toate având o gamă largă de biblioteci fie standard, fie ”third-party”, cu foarte multe funcționalități.

Limbajul folosit pe server este C++ din mai multe motive:

- Performanță foarte bună (factorul cel mai important)
- Compatibil cu CUDA (NVIDIA, 2022) (în mod nativ)
- Compatibil cu driver MySQL (MySQL, 2022)
- Experiența deja acumulată cu acest limbaj

Pe client se folosește limbajul C#, acesta fiind limbajul primar folosit de către motorul de joc, scopul fiind de extindere a funcționalităților de bază oferite de către acesta prin script-uri.

Python este un limbaj de programare foarte popular, cu multe funcționalități care permit o dezvoltare foarte rapidă de programe. Python este folosit în prototip pentru a implementa canalul de comunicare criptat între client și server.

4 IMPLEMENTARE

În capitolele următoare sunt prezentate detalii despre implementarea arhitecturii sau anumitor funcționalități ale prototipului.

4.1 Arhitectura

Arhitectura client-server este cea predominant folosită în industria jocurilor de tip MMO, spre deosebire de arhitecturile de tip peer-to-peer sau arhitecturile hibride. Principalul motiv pentru acest lucru este faptul că, având un server autoritar, problema trișării din partea jucătorilor este diminuată semnificativ – aceasta fiind o problemă foarte importantă în jocurile de acest gen, care poate duce la pierderi de capital foarte semnificative dacă situația nu este sub control. De asemenea, este o arhitectură destul de ușor de implementat, cu niște principii bine stabilite.

Având în vedere avantajele enumerate mai devreme (plus altele enumerate în capitolul specific studiului domeniului), s-a ales arhitectura client-server pentru acest prototip. În acest caz, serverul este autoritar, calculând întreaga stare a jocului și transmițând-o către clienți. Astfel, clienții trimit către aplicația server doar cereri sau intenții pentru acțiuni, doar cel din urmă ocupându-se de validarea și aplicarea acțiunilor.

4.2 Funcțiile aplicației server

Aplicația server are rolul de a primi acțiuni de la clienți, procesarea acestora, actualizarea stării jocului folosindu-se de aceste acțiuni și trimiterea stării înapoi către clienți. De asemenea, serverul se ocupă de plasarea și gestionarea NPC-urilor, care se mișcă în mod aleator pe hartă.

Înainte de a porni serverul, câteva setări trebuie să fie configurate din cod:

- Numărul de tick-uri pe secundă pentru server – un tick reprezintă o actualizare a stării jocului
- Port-ul pe care se primesc și se trimit pachete
- Viteza de bază cu care se mișcă jucătorii la fiecare actualizare
- Numărul maxim de jucători
- Numărul de NPC-uri și distanța dintre aceștia
- ID-ul hărții de joc
- Dimensiunea AABB-ului (Axis Aligned Bounding Box) folosit pentru gestionarea interesului

La pornirea aplicației, se creează un timestamp cu timpul curent, se creaza un nou socket pe port-ul specificat, setat ca acesta să funcționeze în mod non-blocant și se inițializează componenta de fizică și de stare a jocului, alocând/inițializând buffer-ele de memorie CPU/GPU și plasând NPC-urile pe hartă folosind setările configurate anterior. Mai departe, serverul intră într-o buclă infinită, la fiecare iterație a buclei acesta verifică dacă a

primit vreun pachet. În cazul în care a primit un pachet, acesta trece printr-o etapă de verificare și parsare, în care se verifică tipul pachetului și a informațiilor conținute de acesta. Există mai multe tipuri de pachete, după cum urmează:

- Pachet de autentificare, în care se primesc informații precum nume de utilizator, parolă și un UUID (Universal Unique Identifier), rolul serverului fiind de a interoga baza de date pentru a verifica corectitudinea informațiilor de autentificare, de a stoca informațiile despre poziția și tipul obiectului utilizatorului într-un array de entități, utilizatorul primind un ID de entitate. Mai departe urmează stocarea mapării între UUID-ul primit anterior cu acest ID de entitate, iar în cele din urmă trimiterea acestor informații înapoi către utilizator
- Pachet de acțiune, în care se primește acțiunea pe care utilizatorul vrea să o facă alături de UUID-ul acestuia. Server-ul, pe baza UUID-ului, întoarce entitatea specifică utilizatorului și modifică acțiunea curentă a acestuia, acțiunea fiind aplicată într-un pas ulterior al procesării, după calculele de fizică
- Pachet de începere al simulării trimis de către aplicația client, trimis după ce procesul de autentificare s-a încheiat, serverul putând începe simularea jocului

La fiecare iterație a serverului se verifică câte milisecunde au trecut de la timestamp-ul precedent - dacă numărul de milisecunde este mai mare decât perioada de timp setată pentru server (împărțind 1000 la numărul de tick-uri setat) și dacă a fost primit pachetul de începere a simulării, atunci se face un pas în simularea de fizică. Componenta de fizică are acces la starea jocului, atunci când se face un pas în simulare se transferă datele necesare pentru fizică (cum ar fi poziția obiectelor, dimensiunea acestora, tipul acestora etc.) din memoria CPU în memoria GPU, și se încep calculele de fizică, verificându-se dacă entități din scenă se suprapun, aplicând forțe de fizică precum impuls, gravitație și forța de frecare. După ce pozițiile finale au fost calculate, pozițiile vechi devin pozițiile actuale. Mai departe, se rulează pasul de gestionare a interesului pentru fiecare entitate corespunzătoare unui utilizator, un utilizator fiind interesat doar de actualizări cu entitățile dinamice apropiate de acesta. După acest pas sunt calculate deja pachetele de actualizare care trebuie trimise de către server către fiecare utilizator, rămas fiind doar pasul de trimitere a acestor pachete.

Pe server există verificări în legătura cu timpul de procesare pentru anumite activități (cum ar fi durata actualizării pozițiilor jucătorilor sau durata trimiterii pachetelor către client), acestea fiind afișate la ieșirea standard.

4.3 Funcțiile aplicației client

Aplicația client este un joc în Unity, aceasta a fost gândită mai mult ca o unealtă de vizualizare a stării jocului și ca o unealtă de testare a comunicării cu serverul, astfel că funcționalitățile acesteia nu sunt neapărat complexe.

În primul rând, componentele aplicației client au câțiva parametri care pot să fie schimbați din editorul din Unity înainte de rulare:

- IP-ul și port-ul serverului
- Numărul de autentificări efectuate, aplicația client având rol de a simula procesul de autentificare pentru un număr arbitrar de utilizatori
- Numărul de obiecte statice de pe hartă, existând 3 tipuri de obiecte suportate (copaci, cutii, pietre), fiecare având o valoare minimă și maximă de scalare și o valoare de offset. Alături de acestea, există și un parametru care setează seed-ul generatorului de valori aleatoare, pentru a avea rezultate persistente între rulări.
- Numărul de NPC-uri simulate pe server

La rularea aplicației, se creează un socket pentru a face legătura cu serverul, se crează un thread pentru primirea pachetelor de la server, se creează componentele principale (de comunicare și de gestionare a entităților), se alocă memorie pentru entitățile dinamice din simulare și se instanțiază obiectele statice în funcție de parametrii aleși. După acest pas, se trimit pachetele de autentificare către server, utilizatorii și parolele stocate pe server fiind sub forma ”user” urmat de un indice, respectiv ”password” urmat de un indice. Utilizatorul specific primei autentificări este jucătorul simulat de aplicația client, acesta fiind urmărit de către camera principală și a cărui mișcare poate fi controlată de la tastatură. După ce se trimit pachetele de autentificare, se mai trimite un pachet care semnalează serverului că poate începe simularea jocului.

În fiecare frame, aplicația client trimite către aplicația server UUID-ul și acțiunea jucătorului, acțiunea fiind codificată sub formă unei măști de 5 biți:

- Primul bit semnifică dacă este apăsată tasta săgeata sus (mișcare înainte)
- Al doilea bit semnifică dacă este apăsată tasta săgeata jos (mișcare înapoi)
- Al treilea bit semnifică dacă este apăsată tasta săgeata dreapta (mișcare în dreapta)
- Al patrulea bit semnifică dacă este apăsată tasta săgeata stânga (mișcare în stânga)
- Al cincilea bit semnifică dacă este apăsată tasta spațiu (săritură)

La primirea unui pachet de la server pe thread-ul de primire pachete, acesta trece printr-o funcție care verifică dimensiunea pachetului, dacă dimensiunea este cea bună atunci se decodifică informațiile din pachet sub forma unui vector 3D cu poziția jucătorului și o valoare întreagă care conține ID-ul și tipul obiectului acestuia. Informațiile sunt adăugate într-o structură și mai apoi adăugate într-o coadă. La fiecare cadru al aplicației coada este golită, iar poziția jucătorilor este actualizată, folosind ID-ul acestora ca index în structura de date de jucători.

Pentru a menține un FPS mulțumitor în aplicația client în cazurile de test (cu sute de mii de obiecte în scenă), s-a aplicat un culling mai agresiv, fiind folosit un far plane de 50.

O altă optimizare pentru FPS în aplicația client este dezactivarea obiectelor care nu primesc actualizări de la server pentru un număr configurabil de cadre.

4.4 Folosirea GPGPU

Obiectivul aplicației este de a fi scalabilă, astfel că se dorește folosirea GPGPU pentru problemele de dimensiune mare care pot fi paralelizate. O astfel de problemă este detectarea coliziunilor între entitățile dinamice, aplicarea forțelor de fizică acestora și gestiunea interesului pentru utilizatori.

4.4.1 Gestionarea memoriei

Aplicația server a fost gândită în așa fel încât să maximizeze numărul de transferuri de memorie GPU-GPU și să minimizeze numărul de transferuri de date de memorie CPU-GPU, cele din urmă având o viteză de transfer mult mai mică. De asemenea, s-a urmărit folosirea eficientă a cache-ului memoriei GPU în timpul procesărilor, astfel că s-a folosit un design data-oriented.

Pentru a mări viteza de transfer între memoria CPU și memoria GPU s-a folosit alocare de memorie pinned pentru memoria CPU, GPU-ul nemaiavând nevoie de ajutor de la CPU în acest caz, putând copia memoria folosind mecanisme DMA. Acest lucru a dublat viteza de transfer de memorie CPU-GPU.

4.4.2 Sistemul de fizică

Aplicația server are rolul simulării fizicii pentru toate entitățile din spațiul virtual, simularea fiind rulată doar pe server pentru a controla starea jocului în mod centralizat - reducând astfel tentativele și metodele de trișare disponibile pentru jucători în aplicațiile descentralizate.

Calculele de fizică se pretează foarte bine pe tehnicile GPGPU, existând un grad mare de paralelism de date, framework-ul folosit fiind CUDA (NVIDIA, 2022).

Forțele simulate în sistemul de simulare a fizicii sunt forța de frecare, forța de gravitație, viteza liniară și impulsul liniar. Pentru integrare se consideră integrare Verlet[8]. Pentru a putea aplica impulsul liniar trebuie să existe un pas de detectare a coliziunilor între obiectele din scenă și de rezolvare a acestora. În aplicația server există 3 tipuri de implementări pentru detectarea și rezolvarea coliziunilor: o implementare naivă pe CPU, o implementare naivă GPGPU și o implementare optimizată GPGPU. Implementarea optimizată constă în construirea unei structuri de date spațiale de accelerare având la bază volumele încadratoare ale obiectelor din scenă și de folosire a acesteia pentru a face interogări rapide de coliziune. Deoarece se folosesc volume încadratoare pentru a mări viteza testelor de coliziune, se împarte pasul de detecție a coliziunilor în 2 faze: broad phase și narrow phase. În broad phase se fac interogările de coliziune bazate pe volume

[8] https://www.physics.udel.edu/~bnikolic/teaching/phys660/numerical_ode/node5.html

încadratoare și se generează perechi de obiecte potențial în coliziune, pe când în narrow phase se fac teste de coliziune mai exacte și mai costisitoare pe aceste perechi de obiecte generate.

În capitolele următoare se va detalia implementarea celor două faze.

4.4.2.1 Broad phase

Structura de date folosită pentru broad phase este BVH, folosind accelerare GPGPU conform metodei dezvoltată de Kerras et al. (Karras, Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees, 2012).

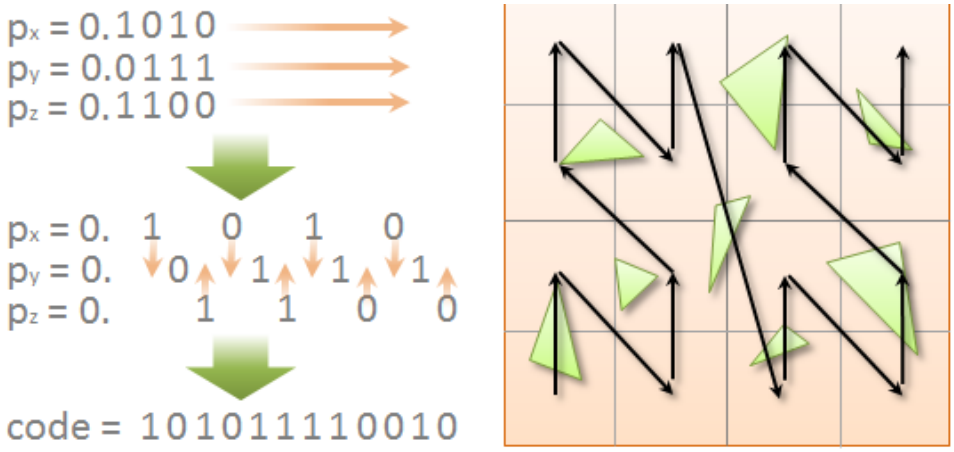

Figura 1 Calcularea și vizualizarea codurilor Morton (Karras, Thinking Parallel, Part III: Tree Construction on the GPU, 2012)

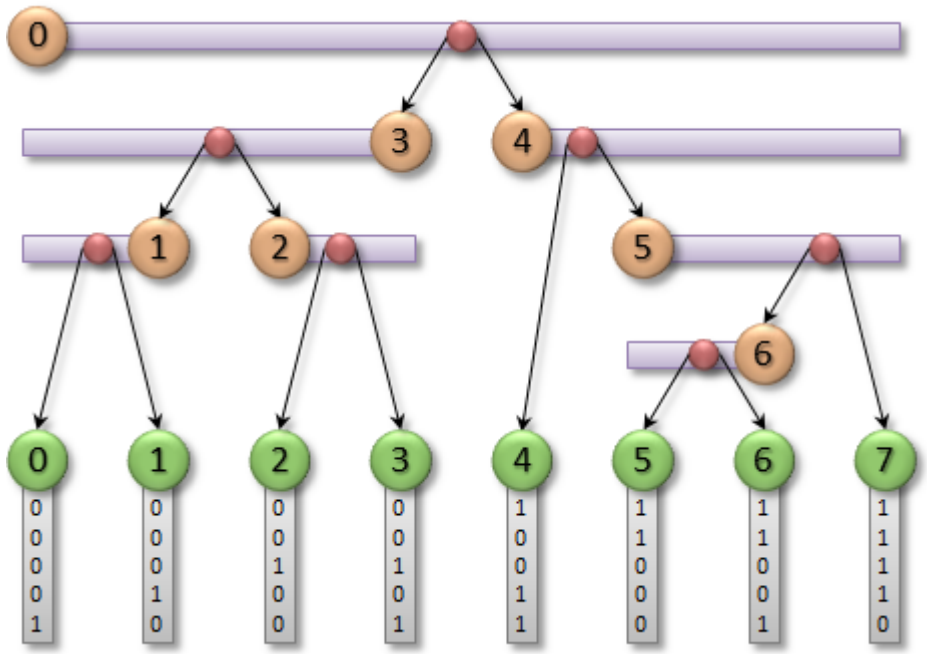

Figura 2 Schema de numerotare pentru nodurile interne (Karras, Thinking Parallel, Part III: Tree Construction on the GPU, 2012)

Algoritmul de construcție a BVH-ului folosind GPGPU presupune:

- Calcularea de coduri Morton pentru coordonatele centrelor volumelor încadratoare ale obiectelor din scenă. Codurile Morton mapează puncte dintr-un spațiu multidimensional într-o singură dimensiune, având proprietăți de păstrare a localității punctelor. Codurile Morton sunt calculate prin intercalarea biților din coordonatele obiectelor, folosind 32 biți, intercalând 10 biți din componentele X, Y și Z ale pozițiilor, luând în considerare doar partea întreagă a acestora, deci se folosesc doar valori întregi din intervalul [0, 1023]. Acest pas se poate paraleliza foarte ușor în manieră GPGPU. Un exemplu de calcul și de vizualizare al acestor coduri este prezentat în figura 1
- Sortarea codurilor Morton. Acest lucru se face în mod eficient folosind Radix Sort pe GPGPU, în cazul aplicației prototip fiind folosită implementarea de Radix Sort pe GPGPU din biblioteca CUB (NVIDIA, 2022). Codurile Morton sortate reprezintă ordinea nodurilor frunză ale BVH-ului.
- Generarea ierarhiei BVH-ului în paralel folosind o schemă de numerotare pentru nodurile interne care să ofere informații despre mulțimea de noduri copii pe care acestea le cuprind, fără a fi nevoie de informații adiționale - această schemă de numerotare permite generarea ierarhiei în paralel pentru fiecare nod, neexistând dependințe. Algoritmul este compus din 4 pași - în primul pas se calculează direcția mulțimii, iar pe baza direcției se extinde mulțimea cât se poate de mult. Urmează un pas de împărțire a acestei mulțimi, iar submulțimile rezultate în urma împărțirii sunt folosite pentru a determina nodurile copii. Pasul de extindere și de impărțire a mulțimii se pot face în mod eficient folosind căutare binară. Un exemplu al schemei de numerotare este prezentat în figura 2
- Popularea nodurilor interne, un nod fiind compus din:
  - O structură pentru AABB-ul nodului, compusă din punctul 3D minim și punctul 3D maxim al AABB-ului, folosind tipul float16[9] oferit de CUDA (NVIDIA, 2022)
  - 3 pointeri către părintele și copii din stânga și din dreapta ai nodului
  - Indexul și codul Morton al nodului ca valori întregi pe 32 biți
  - Codul Morton maxim atins traversând nodul copil din stânga, respectiv codul maxim atins traversând nodul copil din dreapta, ambele fiind valori întregi pe 32 biți
  - O codificare a tipului nodului, tipul putând fi fie nod intern, fie nod frunză

9 https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__INTRINSIC__HALF.html#group__CUDA__MATH__INTRINSIC__HALF

- Ultimul pas presupune calcularea volumelor încadratoare pentru nodurile interne. Acest lucru se poate face în paralel în manieră bottom-up, pornind de la nodurile frunză și calculând volumele încadratoare ale nodurilor interne părinte. Pentru a evita calculele duplicate și pentru a asigura corectitudinea se definește un set de variabile atomice pentru fiecare nod intern cu rolul de a opri execuția primului thread GPU care procesează nodul intern respectiv. Astfel, este eliminată munca duplicată și este asigurat faptul că ambii copii ai nodului intern au volumul încadrator deja calculat

Având în vedere faptul că majoritatea calculelor pot fi paralelizate, acest algoritm oferă performanțe foarte bune pentru construcția BVH-ului, de ordinul milisecundelor pentru zeci/sute de mii de obiecte. Acest lucru permite reconstrucția BVH-ului la fiecare pas al simulării fizicii, evitând problema deteriorării calității BVH-ului în cazul în care ar fi fost folosite metode de actualizare a nodurilor din ierarhie.

Traversarea BVH-ului este paralelizabilă pe GPU, lansând un kernel pentru fiecare nod frunză (echivalent cu fiecare obiect care a fost folosit pentru construcția BVH-ului). Acest kernel face următoarele procesări:

- Menține o stivă cu nodurile care urmează a fi procesate
- Pornind de la nodul rădăcină, se compară, în fiecare pas, codul Morton al nodului frunză cu codurile Morton maxime atinse din subarborii specifici copiilor nodului curent. În cazul în care codul Morton maxim atins dintr-un subarbore al unui nod copil este mai mare decât codul Morton al nodului frunză, se verifică intersecția volumului încadrator al nodului frunză cu volumul încadrator al nodului copil respectiv. Volumele încadratoare folosite în prototip sunt AABB-uri, deci se testează intersecția AABB-AABB, comparând suprapunerea proiecțiilor coordonatelor minime și maxime pe cele 3 axe principale. În cazul intersecției se stochează perechea de chei pentru a fi testate în narrow phase. Verificarea ID-urilor elimină generarea de self-collisions și de coliziuni duplicate. În prototip, se pot procesa maxim 26 de coliziuni pentru fiecare ID într-un pas de simulare
- Se scoate nodul curent din stivă și se adaugă în aceasta eventuale alte noduri interne care intersectează volumul nodului pentru care se rulează kernel-ul

Perechile de ID-uri de obiecte care colizionează vor fi procesate mai departe în narrow phase, pas în care se aplică și calculele de fizică.

4.4.2.2 Narrow phase

Cea de-a doua fază a sistemului de fizică. Primește perechile de ID-uri de obiecte care sunt potențial în coliziune și rulează un kernel pentru procesarea în detaliu a coliziunilor dintre acestea. Kernel-ul este rulat pentru fiecare astfel de pereche și are următoarele roluri:

- Se calculează intersecția între sfere, AABB-uri și capsule, structura care încapsulează tipul obiectului și informațiile despre forma acestuia conținând următoarele câmpuri:
  - Raza (informație necesară pentru sfere și capsule)
  - Jumătate din lungimea, lățimea și înălțimea obiectului (informație necesară pentru AABB)
  - Înălțimea (informație necesară pentru capsule)
- În caz de intersecție, se calculează normala coliziunii, adâncimea intersecției și viteza relativă a celor două obiecte folosind pozițiile curente și pozițiile de la pasul de simulare anterior. De asemenea, se verifică faptul că obiectele nu se mișcă în direcții diferite
- Se aplică o schimbare instantă în poziția finală sub formă de impuls. Pozițiile finale sunt actualizate în mod atomic.

După ce kernel-ul și-a terminat execuția, se rulează un kernel de aplicare a forțelor (considerând forța de gravitație, forța de frecare și accelerația dată de acțiunea obiectului respectiv) și de actualizare a pozițiilor obiectelor având în vedere și limitele extreme ale spațiului virtual.

Pentru a extrage informațiile precum normala coliziunii și adâncimea coliziunii, au fost dezvoltate funcții pentru următoarele cazuri de coliziune:

- Sferă - sferă
- Sferă - AABB
- Sferă - capsulă
- AABB - AABB
- AABB - capsulă
- Capsulă - capsulă

Au fost urmărite explicațiile din Game Physics Cookbook[10] și din Wicked Engine[11] pentru a extrage în mod corect informațiile coliziunilor.

[10] https://gamephysicscookbook.com/
[11] https://wickedengine.net/2020/04/26/capsule-collision-detection/

Funcția care calculează informațiile de coliziune pentru cazul sferă - sferă presupune mai întâi testul de intersecție sferă – sferă, acesta verificând dacă distanța între pozițiile celor două sfere este mai mică decât suma razelor sferelor. În caz afirmativ, normala coliziunii este vectorul normalizat al diferenței dintre cele două poziții, iar adâncimea coliziunii este dată de diferența între suma razelor sferelor și distanța dintre cele două poziții.

Pentru cazul sferă - AABB, se calculează poziția 3D a punctului din AABB situat cel mai aproape de poziția sferei. După ce poziția acestui punct a fost calculată, se măsoară distanța de la poziția sferei până la acest punct - în cazul în care distanța este mai mică decât raza sferei, înseamnă ca există coliziune între sferă și AABB. În cazul acesta, vectorul normal al coliziunii este un vector de la centrul sferei la poziția punctului calculată anterior. Adâncimea coliziunii dintre cele două corpuri este distanța dintre poziția punctului calculată anterior și poziția punctului de pe sferă în direcția vectorului normal al coliziunii.

Cazul sferă - capsulă este foarte similar cu cazul sferă - sferă, fiind nevoie de o funcție care proiectează poziția centrului sferei pe vectorul format din poziția de bază a capsulei și poziția de vârf a capsulei. După ce această poziție este calculată, se testează intersecția dintre sferă și o nouă sferă care are centrul în poziția proiectată pe capsulă și raza capsulei, iar în cazul în care există intersecție între cele două, se extrag informațiile de coliziune.

Testul de intersecție pentru cazul AABB – AABB a fost detaliat în capitolul 4.4.2.1. Acest test poate fi folosit pentru a extrage informațiile necesare pentru coliziune, vectorul normal al coliziunii fiind axa pe care s-a produs cea mai mică suprapunere, iar adâncimea coliziunii fiind chiar această suprapunere minimă.

Cazul AABB - capsulă este foarte similar cu cazul sferă - capsulă, combinat cu cazul sferă - AABB. Similar cu cazul sferă - capsulă, se calculează poziția proiectată a centrului AABB-ului pe capsulă. Această poziție, împreună cu raza capsulei, formează o sferă, care mai apoi este supusă testului sferă - AABB, fiind extrase informațiile despre coliziune.

Cazul capsulă - capsulă presupune extragerea sferelor relevante din cele două capsule folosind funcția de proiectare a pozițiilor definite anterior, prin proiectarea pozițiilor de bază și de vârf a fiecărei capsule pe capsula opusă și compararea distanțelor între acestea. După ce cele două sfere au fost extrase, se testează intersecția între cele două și se extrag informațiile de coliziune.

4.4.2.3 Gestionarea obiectelor statice și a terenului

Gestionarea obiectelor statice se face într-un mod similar cu gestionarea obiectelor dinamice – se alocă memorie pe GPU pentru pozițiile obiectelor și informațiile despre tipul/forma acestora și se creează un BVH în manieră similară ca în cazul obiectelor dinamice, diferența fiind în folosirea de coduri Morton pe 64 biți, înmulțind valorile componentelor X, Y și Z ale pozițiilor cu 1024 și folosind 21 biți pentru fiecare componentă. Simularea fizicii pentru aceste obiecte se face din nou în 2 faze – broad phase și narrow phase, foarte asemănător cu cazul obiectelor dinamice, existând doar câteva modificări:

- În cazul broad phase, se rulează un kernel care, pentru fiecare obiect dinamic, testează coliziunile între AABB-ul obiectului dinamic și AABB-urile obiectelor statice, adăugând potențialele coliziuni într-un vector cu ID-uri, putând fi adăugate un maxim de 26 coliziuni/ID-uri pentru fiecare obiect dinamic
- În cazul narrow phase, diferența constă în calculul impulsului, la obiectele dinamice magnitudinea impulsului fiind împărțită la 2, in cazul obiectelor statice fiind nevoie de întreaga valoare a impulsului, neputând aplica impuls obiectului static

Ordinea de aplicare pentru operațiile de fizică este:

- Broad phase static
- Broad phase dinamic
- Narrow phase static
- Narrow phase dinamic
- Aplicarea forțelor
- Verificarea constrângerilor în legătură cu limitele pozițiilor

Pentru gestionarea terenului se folosește o hartă de înălțime, în care sunt salvate valori în intervalul [0, 65535], harta având o rezoluție specifică, în cazul curent fiind folosită o hartă cu de rezoluție 1025x1025. Valoarea 0 semnifică o înălțime de 0, iar valoarea 65535 semnifică o înălțime maximă, specificată la încărcarea hărții, în cazul curent înălțimea maximă fiind 128, valorile fiind convertite din intervalul [0, 65535] în intervalul [0, 128], valorile finale fiind în virgulă mobilă.

După încărcarea hărții, se creează o textură 2D în memoria GPU în care se vor salva valorile convertite.

După pasul de simulare a fizicii și după ce toate forțele au fost aplicate asupra obiectelor (pozițiile acestora fiind schimbate), obiectele se pot afla în afara spațiului de joc. În acest caz, se rulează un kernel pentru fiecare obiect dinamic, în care se verifică dacă un obiect a ieșit din spațiul de joc, fiind definite limite inferioare pentru componentele X și Z ale pozițiilor, respectiv limite superioare pentru toate componentele. Limita inferioară pentru componenta Y depinde de componenta Y a terenului într-un anumit punct, astfel că se transformă componentele X și Z ale poziției unui obiect dinamic în coordonate uv, coordonate care mai apoi vor fi folosite pentru a face o eșantionare folosind bilinear filtering în textura 2D care reprezintă harta de înălțimi.

4.4.3 Gestionarea zonelor de interes

Fiecare jucător este interesat de actualizări pentru obiecte dinamice doar din proximitatea acestuia, această proximitate fiind denumită zonă de interes. În acest scop, a fost dezvoltat un sistem de gestionare a zonelor de interes pentru utilizatori, fiind folosit BVH-ul specific obiectelor dinamice deja calculat pentru a extrage obiectele aflate în proximitatea fiecărui jucător. Acest lucru se face prin definirea unui AABB centrat în poziția jucătorului, iar în funcție de parametrii server-ului care definesc AABB-ul să se extragă obiectele care intersectează volumul, evitând astfel trimiterea de N actualizări către client, unde N este numărul de obiecte simulate. După extragerea obiectelor de interes din proximitatea jucătorilor, se rulează un kernel care populează pachetele de actualizări care trebuie trimise de către server.

4.5 Protocolul de comunicare

În prototip, comunicarea între server și client se realizează cu ajutorul protocolului UDP. Motivele principale pentru care acest protocol a fost ales sunt următoarele:

- UDP este mult mai eficient decât TCP, pachetele au header-ul de dimensiune mai mică iar transmisia este mai rapidă, latența fiind un factor foarte important în această aplicație
- Se poate face multicast/broadcast cu ajutorul UDP
- Ordinea în care pachetele ajung la server nu este importantă în cazul aplicației
- Pierderea de pachete de actualizare a stării jocului trimise de către server este o problemă critică, serverul retrimițând un alt pachet de actualizare în scurt timp

Există mai multe tipuri de pachete:

- Un pachet de actualizare a stării unui jucător, definit în aplicația server, având un număr variabil de octeți, încapsulând actualizări pentru maxim 50 obiecte dinamice, pachetul fiind format dintr-un câmp de un octet care reprezintă numărul de obiecte dinamice din pachet și dintr-un câmp de maxim 50 de structuri de 10 octeți, o structură având următoarele câmpuri:
  - ID-ul și tipul obiectului încapsulate într-un câmp cu dimensiunea de 4 octeți, 30 biți fiind folosiți de ID, iar restul de 2 biți fiind folosiți de tip
  - Poziția 3D a obiectului cu dimensiunea de 6 octeți, conținând componenta X, Y și Z a poziției obiectului în format virgulă fixă, folosind 10 biți pentru partea întreagă și 6 biți pentru partea fracționară
- Un pachet de acțiune, definit în aplicația client, având 18 octeți și fiind compus dintr-un câmp de un octet care reprezintă ID-ul pachetului, un câmp de un octet care reprezintă acțiunea jucătorului și un câmp de 16 octeți care reprezintă UUID-ul

- Un pachet de autentificare, definit în aplicația client, având 97 octeți și fiind compus dintr-un câmp de un octet care reprezintă ID-ul pachetului, un câmp de 32 octeți care reprezintă numele de utilizator și un câmp de 64 octeți care reprezintă parola
- Un pachet care semnifică sfârșitul autentificărilor, astfel că serverul poate să înceapă simularea jocului. Are dimensiunea de 5 octeți, reprezentat de șirul ”READY”

Pentru canalul criptat de comunicare s-a folosit biblioteca SSL[12] din Python, care implementează TLS (Rescorla, 2018), pe lângă multe altele. Pe baza acestei biblioteci s-au dezvoltat din nou 2 aplicații: aplicația client TLS și aplicația server TLS, rolul acestora fiind de deschidere a canalului de comunicare criptat și îndeplinirea următoarei secvențe:

- Aplicația client construiește un pachet de autentificare, care conține ID-ul pachetului, numele de utilizator și parola. Acest mesaj este trimis prin mecanisme de IPC (Inter Process Comunication) către aplicația client TLS, care criptează pachetul și îl trimite către aplicația server TLS
- Aplicația server TLS primește pachetul de login, datele sunt decriptate, se generează și se adaugă un UUID (Universal Unique Identifier), iar datele împreună cu UUID-ul se trimit către aplicația server prin mecanisme de IPC
- Aplicația server primește datele de autentificare și UUID-ul. Datele de autentificare sunt verificare folosind procedura stocată CheckUserAndRetriveInfo, iar în cazul în care datele de autentificare sunt corecte, se adaugă în memorie un nou obiect dinamic specific utilizatorului (cu datele primite de la functia CheckUserAndRetrieveInfo) și se pastrează o mapare între UUID și ID-ul obiectului dinamic specific utilizatorului. Acest UUID trebuie să fie adăugat de către aplicația client în fiecare pachet trimis către aplicația server, astfel ca aplicația server să știe cărui utilizator să îi atribuie cererea
- Datele primite de la procedura stocată împreună cu UUID-ul se trimit prin IPC către aplicația server TLS, care trimite mai departe datele criptate către aplicația client TLS, aici fiind decriptate, iar în cele din urmă datele ajung la aplicația client, care păstrează UUID-ul pentru a putea fi adăugat pachetelor viitoare către aplicația server și folosește restul datelor primite pentru a crea un nou obiect și pentru a-l plasa corect în scenă

4.6 Baza de date

Baza de date folosită este una relațională, sistemul de gestiune a bazei de date fiind MySQL (MySQL, 2022).

Există multiple tabele adăugate în baza de date:

[12] **https://docs.python.org/3/library/ssl.html**

- Users
- Maps
- Users_game_info
- Static_objects

Tabela users conține informații despre utilizatori, sub formă de ID utilizator, nume de utilizator și parolă. Parola este salvată sub formă de hash, pentru a adăuga un strat de siguranță asupra parolelor utilizatorilor în cazul unei breșe de securitate.

Tabela maps conține informații despre hărțile din joc, sub formă de ID al hărții, calea pe disk către fișierul care reprezintă harta de înălțimi a terenului și înălțimea maximă a acestuia.

Tabela users_game_info conține informații despre starea din joc a utilizatorilor pe diferite hărți:

- O cheie primară compusă, formată din ID utilizator și ID al hărții
- Tipul obiectului jucătorului (sferă, cutie, capsulă)
- Poziția 3D a jucătorului pe hartă (3 valori în virgulă mobilă)

Tabela static_objects conține informații despre starea din joc a obiectelor statice. Are un format similar cu tabela users_game_info, doar că în loc de ID utilizator se folosește ID obiect.

În sistemul de gestiune a bazei de date au fost adăugate următoarele proceduri stocate:

- NewUser - populează tabela users cu o nouă intrare
- CheckUser - verifică validitatea datelor de autentificare trimise
- CheckUserAndRetrieveInfo - verifică validitatea datelor de autentificare și întoarce poziția și tipul obiectului utilizatorului de pe o anumită hartă
- NewUserGameInfo – adaugă o intrare în tabela users_game_info, specificând poziția și tipul obiectului utilizatorului pe o anumită hartă
- UpdateUserGameInfo – suprascrie poziția unui utilizator pe o anumită hartă
- GetUserGameInfo - întoarce poziția și tipul obiectului unui utilizator pe o anumită hartă
- NewMap - adaugă o intrare în tabela maps
- GetMapInfo - întoarce informațiile despre o anumită hartă
- NewStaticSphereObject/NewStaticBoxObject/NewStaticCapsuleObject - adaugă o intrare în tabela static_objects cu informațiile despre un obiect static pe o anumită hartă
- GetMapStaticObjects - întoarce toate obiectele statice de pe o anumită hartă

Pentru ca aplicația server să comunice cu baza de date, s-a dezvoltat un wrapper peste driver-ul MySQL Connector (MySQL, 2022), aplicația server apelând doar proceduri stocate din baza de date.

5 REZULTATE

Mașina pe care s-au rulat scenariile din acest capitol dispune de următoarea configurație hardware:

- CPU AMD Ryzen 9 5900X
- GPU Nvidia GTX 1660
- Placă de rețea on-board de 2.5 Gb/s

Există 3 scenarii de testare, în care sunt variate numărul de obiecte statice și numărul de obiecte dinamice (reprezentate de NPC și utilizatori). Obiectele statice sunt amplasate în mod aleator, fiind variate dimensiunile acestora. NPC-urile și utilizatorii sunt plasați inițial în mod uniform pe hartă, aceștia mișcându-se în mod aleator. Scenariile sunt următoarele:

- Scenariul 1
  - 3000 obiecte statice (5000 AABB-uri, 1000 capsule)
    - 1000 copaci
    - 1000 cutii
    - 1000 pietre
  - 6000 obiecte dinamice (2000 sfere, 2000 AABB-uri, 2000 capsule)
    - 5000 NPC
    - 1000 utilizatori
- Scenariul 2
  - 15000 obiecte statice (25000 AABB-uri, 5000 capsule)
    - 5000 copaci
    - 5000 cutii
    - 5000 pietre
  - 55000 obiecte dinamice (18334 sfere, 18333 AABB-uri, 18333 capsule)
    - 50000 NPC
    - 5000 utilizatori
- Scenariul 3
  - 99999 obiecte statice (166665 AABB-uri, 33333 capsule)
    - 33333 copaci
    - 33333 cutii
    - 33333 pietre
  - 260000 obiecte dinamice (86668 sfere, 86666 AABB-uri, 86666 capsule)
    - 250000 NPC
    - 10000 utilizatori

În primul rând, s-a comparat timpul de procesare al simulării fizicii cu obiecte statice și obiecte dinamice, pentru un algoritm naiv CPU, pentru un algoritm naiv GPGPU și pentru algoritmul GPGPU prezentat (optimizat). Timpii care au durat peste o secundă nu au mai fost adăugați. Rezultatele se pot vedea în tabelul 1.

Scenariul	CPU static (ms)	CPU dinamic (ms)	GPGPU static (ms)	GPGPU dinamic (ms)	GPGPU optimizat static (ms)	GPGPU optimizat dinamic (ms)
1	396.5	198.6	5.1	4.4	0.2	1.1
2	-	-	320.4	175.3	0.3	2.1
3	-	-	-	-	2.2	5.9

Tabelul 1 Timpul de execuție pentru calculele de fizică

Folosind algoritmul GPGPU optimizat pentru partea de fizică, s-a creat un nou set de rezultate folosind aceleași scenarii ca mai devreme, în care s-au măsurat timpii de execuție în diferite stagii ale aplicației server. Rezultatele se pot vedea în tabelul 2.

Scenariul	Actualizare structuri GPU (ms)	Construcție BVH dinamic (ms)	Broad phase static (ms)	Broad phase dinamic (ms)	Narrow phase static (ms)	Narrow phase dinamic (ms)	Aplicare forțe (ms)	Actualizare pachete (ms)	Trimitere pachete (ms)
1	0.1	0.8	0.1	0.2	0.1	0.1	0.1	1.1	1.2
2	0.2	0.7	0.2	1.2	0.1	0.2	0.1	1.8	7.1
3	0.4	1.4	1.6	3.9	0.6	0.6	0.5	10.5	19.3

Tabelul 2 Timpul de execuție pentru diferite stagii ale aplicației server

6 CONCLUZII

Prototipul dezvoltat și-a atins obiectivele propuse, programarea GPGPU oferind un factor de speed-up foarte mare, de până la 2 ordine de magnitudine, comparativ cu aceleași calcule care rulează pe CPU. Optimizările și structurile de date de accelerare folosite pentru accelerarea calculelor GPGPU au oferit din nou un speed-up de până la 2 ordine de magnitudine. Algoritmii folosiți lasă loc și de scalare verticală, putând adăuga mai multe GPU-uri și folosirea acestora în paralel. Prin urmare, folosirea GPGPU este esențială pentru gestionarea spațiilor virtuale masive, putând simula sute de mii (sau chiar milioane) de obiecte și zeci de mii de jucători în timp real, în cazul în care se folosesc GPU-uri de ultimă generație.

6.1 Dezvoltări ulterioare

O limitare a aplicației este în algoritmul de construcție a BVH-ului – algoritmul presupune chei (coduri Morton) unice. Există o soluție pentru această limitare, cheile putând fi extinse cu reprezentarea binară a index-urilor nodurilor, dar în cazul în care există foarte multe chei duplicate a fost observat faptul că algoritmul nu mai funcționează. De asemenea, în prototip se folosesc coduri Morton pe 32 biți pentru construcția BVH-ului obiectelor dinamice, care limitează poziția acestora în coordonate din domeniul [0, 1023], putând folosi coduri Morton pe 64 biți (sau coduri Morton de dimensiune arbitrară) pentru a putea reprezenta coordonatele într-un spațiu virtual mai mare. În cazul obiectelor statice s-au folosit deja coduri Morton pe 64 biți, deoarece în testarea cazurilor cu foarte multe obiecte, acestea erau foarte apropiate, ducând la nefuncționarea algoritmului de construcție a BVH-ului când erau folosite codurile pe 32 biți. Folosind coduri de dimensiune mai mare implică un timp mai mare de execuție al algoritmului de construcție al BVH-ului, dar diferența de timp ar trebui să fie neglijabilă.

O idee de dezvoltare ulterioară pentru a avea timpi și mai buni de execuție este de a folosi Uniform Grid ca structură de date de accelerare. Totuși, acest lucru implică anumite constrângeri asupra dimensiunilor obiectelor din joc.

Algoritmul pentru gestionarea zonelor de interes pentru utilizatori poate fi optimizat prin folosirea de memorie GPU shared, prin regândirea algoritmului sau/și prin rularea acestui pas înainte de broad phase și prin folosirea listei de coliziuni din acest pas în broad phase, fiind necesară doar o filtrare, nemaifiind nevoie de o parcurgere repetată a BVH-ului.

În biblioteca Winsock2[13] din Windows nu există funcționalități pentru trimiterea mai multor pachete UDP deodată, fără schimbare de context între user space și kernel space. Linux oferă această funcționalitate, existând o bună posibilitate de optimizare.

[13] https://docs.microsoft.com/en-us/windows/win32/api/winsock2/

7 BIBLIOGRAFIE

Chambers, C., Feng, W.-c., & Feng, W.-c. (2006). Towards public server MMOs. *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games - NetGames '06* (p. 3). New York, New York, USA: ACM Press.

Demidov, D., Ahnert, K., Rupp, K., & Gottschling, P. (2013). Programming CUDA and OpenCL: A Case Study Using Modern C++ Libraries. *SIAM Journal on Scientific Computing*, C453-C472.

Diaconu, R., & Keller, J. (2013). Kiwano: A scalable distributed infrastructure for virtual worlds. *2013 International Conference on High Performance Computing & Simulation (HPCS)* (pg. 664-667). IEEE.

Karras, T. (2012). Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. *Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics* (pg. 33-37). The Eurographics Association.

Karras, T. (2012, Dec 19). *Thinking Parallel, Part III: Tree Construction on the GPU*. Preluat pe Iun 21, 2022, de pe Nvidia: https://developer.nvidia.com/blog/thinking-parallel-part-iii-tree-construction-gpu/

MySQL. (2022). *MySQL*. Preluat pe Iun 21, 2022, de pe MySQL: https://www.mysql.com/

MySQL. (2022, Iun 3). *MySQL Connector*. Preluat pe Iun 21, 2022, de pe MySQL: https://dev.mysql.com/doc/connector-cpp/8.0/en/

NVIDIA. (2022). *CUB*. Preluat pe Iun 21, 2022, de pe NVLabs Github: https://nvlabs.github.io/cub/

NVIDIA. (2022). *CUDA*. Preluat pe Iun 21, 2022, de pe NVIDIA: https://developer.nvidia.com/cuda-zone

Rescorla, E. (2018). The Transport Layer Security (TLS) Protocol Version 1.3. RFC Editor. doi:10.17487/RFC8446

Yahyavi, A., & Kemme, B. (2013). Peer-to-peer architectures for massively multiplayer online games. *ACM Computing Surveys*, 1-51.