UNIVERSITATEA POLITEHNICA BUCUREȘTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

PROIECT DE DIPLOMĂ

Dezvoltarea unui motor de joc și a unei aplicații demonstrative pe baza acestuia cu API-ul grafic OpenGL

Alin Drăguț

Coordonator științific:
As. drd. Ing. Cristian Lambru

BUCUREŞTI

2020

CUPRINS

SINOPSIS

În acest proiect este prezentat un motor de joc pentru ușurarea dezvoltării jocurilor video, cu unelte de depanare și sisteme folositoare programatorilor de jocuri. Obiectivele dezvoltării motorului constau în facilitatea folosirii acestuia, performanța și fidelitatea grafică a jocurilor dezvoltate. Pe parcursul lucrării este prezentată modalitatea de implementare a motorului, împreună cu capabilitățile acestuia. Rezultatele sunt mulțumitoare, aplicația dezvoltată folosind motorul având un grad bun de performanță și de fidelitate grafică, rămânând ca ulterior să se dezvolte și mai multe instrumente care să simplifice munca utilizatorilor.

MULȚUMIRI

Aș dori să-i mulțumesc coordonatorului științific Cristian Lambru pentru materialele foarte folositoare și pentru ajutorul acordat în privința pașilor de dezvoltare și al arhitecturii aplicației.

De asemenea, aș dori sa-i mulțumesc colegei mele Teodora Lăbușcă care a dezvoltat aplicația de prezentare a motorului.

1 INTRODUCERE

În acest capitol se va trece în revistă relevanța acestui tip de proiect, viziunea și o scurtă prezentare a următoarelor capitole.

1.1 Context

Din moment ce jocurile video au devenit o industrie, la fel este și crearea motoarelor de joc pe care aceste jocuri se bazează, de obicei, crearea unui joc într-unul dintre motoarele de joc cu o cotă de piață mare implică și o monetizare a acestora, de obicei prin plătirea a unui procent din vânzările făcute de acel joc. Prin urmare, proiectul acesta va încerca implementarea proprie a unui astfel de motor de jocuri, procesul fiind unul interesant, atât din punct de vedere arhitectural cât și din punct de vedere al tehnologiilor și al tehnicilor grafice folosite.

1.2 Problema

Problema principală apărută o dată cu începerea dezvoltării jocurilor a fost faptul că acestea trebuiau construite de la zero de fiecare dată. Dat fiind și domeniul, în care apar noi algoritmi și noi metode de rezolvare a unor lucruri din punct de vedere grafic aproape în fiecare an, multe părți din jocuri nu puteau fi refolosite. Din cauza acestor probleme, timpul de dezvoltare era mare, productivitatea era scăzută și era nevoie de un pachet foarte mare de cunoștințe pentru a reuși dezvoltarea unui joc. Prin dezvoltarea unui motor de joc, se rezolvă multe probleme legate de rapiditatea de dezvoltare și reutilizabilitatea componentelor dezvoltate prin simpla arhitectură a acestora, arhitectura presupunând de multe ori dezvoltarea de componente care pot fi reutilizabile și schimbate oricând cu alte componente mai specifice sau diferite. De asemenea, acestea au în componență sisteme utile dezvoltării unui joc, precum: sistem de fizică, sistem de desenare, sistem audio, sistem de inteligență artificială etc.

1.3 Obiective

Obiectivele propuse de către acest proiect este de a oferi funcționalitățile utile unei dezvoltări rapide și eficiente a jocului prin arhitectura motorului de joc, printr-un sistem de fizică, de desenare, de control și prin unelte de debugging. De asemenea, un alt obiectiv propus al lucrării este de a avea o performanță și o fidelitate grafică bună.

1.4 Structura lucrării

Lucrarea va acoperi în capitolul următor o analiză a unor motoare de jocuri existente, urmărind arhitectura și diversele sisteme folosite. După analiza motoarelor existente, se va intra în tehnicile grafice folosite, pe lângă arhitectura folosită de către motor și alte tehnologii folosite de către acesta, existând un capitol care explică conceptele acestora și un capitol care explică implementarea propriu zisă. La sfârșit se vor prezenta câteva rezultate din motor și concluziile acestei lucrări.

2 STATE OF THE ART

Proiectul propus este format din 2 părți: o parte este reprezentată de către motorul de joc și o parte este reprezntată de aplicația (jocul) creată folosind acest motor.

Această lucrare se ocupă de partea motorului de joc, astfel că sunt analizate pe parcursul acestui capitol arhitectura, sistemele și anumite tehnici folosite de alte motoare de joc existente pe piață. Motoarele analizate sunt următoarele:

2.1 Unity

Game engine cross-platform care oferă utilizatorilor posibilități de creare de jocuri 2D și 3D prin intermediul plugin-urilor, interfețelor drag and drop și script-urilor scrise in C#.

2.1.1 Arhitectura

Unity folosește o paradigmă data-oriented în loc de cea object-oriented, având un sistem denumit ECS[1] pentru a reprezenta aplicația, format din 3 părți:

- Entități – reprezintă obiectele din aplicație
- Componente – datele asociate entităților (și principala diferență între designul data-oriented și cel object-oriented)
- Sisteme – logica prin care o entitate trece dintr-o stare în alta, pe baza datelor aflate în component

Entitățile nu conțin date sau logică, ci doar encapsulează un obiect din aplicație. Entitățile sunt reprezentate printr-un ID și sunt întreținute de către EntityManager, care menține o listă cu acestea și le organizează pentru performanță optimă.

Componentele mențin datele specifice entităților, o colecție de componente reprezintă un arhetip. EntityManager întreține arhetipurile, iar arhetipurile asemănătoare sunt grupate în blocuri de memorie denumite chunks.

Sistemele sunt descoperite automat și sunt instanțiate la runtime. Prin atribute de sistem se poate specifica grupul din care face parte sistemul respectiv, astfel ca Unity să știe cum să-l întrețină. Ordinea în care sistemele se execută este nespecificată, dar deterministă.

2.1.2 User Interface

Unity folosește 3 tipuri de UIToolkits[2] pentru creare de UI, fie în editor, fie în aplicație:

- UIElements - UI retained-mode similar HTML/CSS
- UnityUI – sistem UI simplu de folosit pentru dezvoltarea de UI pentru aplicații, bazat pe obiecte și componente pentru a personaliza interfața. UnityUI nu se poate folosi in editor.

[1] https://docs.unity3d.com/Packages/com.unity.entities@0.10/manual/index.html -
[2] https://docs.unity3d.com/Manual/UIToolkits.html

- IMGUI – UI immediate-mode, în principiu folosit de dezvoltatori pentru a crea interfețe de debug. Nu este cea mai bună alegere pentru UI-ul aplicației

1) UIElements

UIElements este un sistem de interfață utilizator inspirat foarte mult din HTML/CSS. Folosește fișiere de tip UXML pentru a defini logica interfeței și structura acesteia. Pentru a memora toate obiectele vizuale din interfață, UIElements folosește un graf numit Visual Tree care are ca noduri visual elements, încărcate fie manual, fie la încărcarea unui asset UXML. Aceste noduri pot avea noduri copil, iar rădăcina grafului este numit panel, și este nodul la care trebuiesc conectate celelalte noduri pentru ca acestea să fie desenate în scenă, ordinea desenării fiind de la rădăcină către frunze.

2) UnityUI

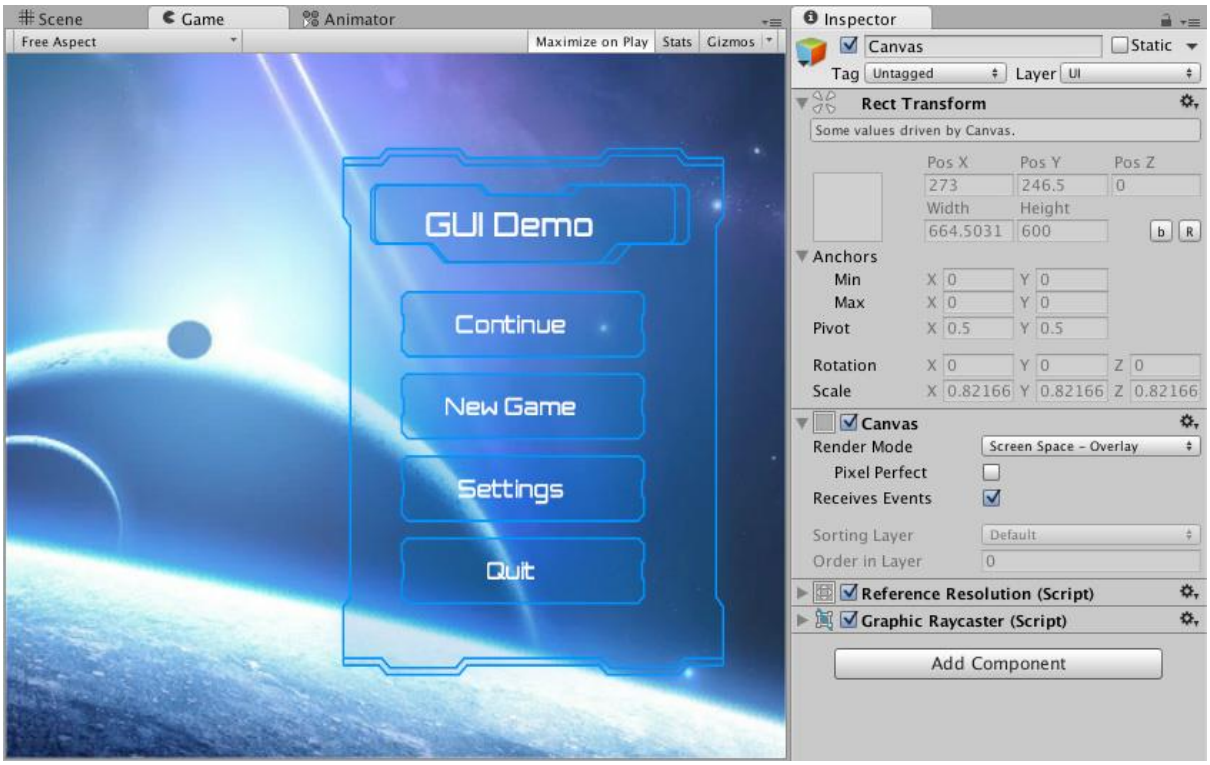

Figura 2.1: UI creat folosind UnityUI[3]

UnityUI este un sistem de interfață utilizator bazat pe GameObjects care folosește componente și Game View pentru a aranja, poziționa și stiliza interfețele. Pentru a putea crea astfel de interfețe trebuie definit un GameObject cu o componentă Canvas, care specifică zona în care vor fi elementele interfeței. Ordinea în care elementele sunt desenate este tot una ierarhică, fiind necesară o rearanjare (folosind mouse-ul) pentru ca aceasta să poata fi schimbată.

[3] https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/UICanvas.html

Există mai multe moduri de randare a interfeței și anume:

- Screen Space – Overlay (figura 2.1)

  Acest mod de randare afișează interfața direct în fața scenei, dacă este modificată dimensiunea ecranului, Canvas se va modifica în același mod.

- Screen Space – Camera

  Similar cu Screen Space – Overlay, doar că interfața este randată din punctul de vedere al unei alte camera, ceea ce inseamnă că setările camerei respective vor influența aparența acesteia.

- World Space

  Prin acest mod se poate specifica poziția, dimensiunea, rotația Canvas-ului pentru ca interfața să poată fi plasată direct in lume.

Pentru a poziționa elementele înăuntrul interfeței, există o componentă numita Rect Transform, care oferă facilitățile unui Transform convențional, doar că există și alte opțiuni precum pivotare, ancorare și modificare de dimensiuni (o scalare doar a dimensiunilor dreptunghiului, fără a fi afectate fonturi de text sau altele).

După ce se face poziționarea elementelor în interiorul Canvas-ului, se pot adăuga acestora fie componente vizuale (text, imagini), fie componente de interacțiune (butoane, slidere). Componentele de interacțiune se pot folosi și de sistemul de animație al Unity pentru a realiza diferite tranziții. IMGUI este un sistem de interfață utilizator bazat pe cod, destinat mai mult programatorilor pentru a crea interfețe simple de debug. Interfețele care folosesc IMGUI vor avea scrisă logica în metoda OnGUI. Principalul avantaj al IMGUI este rapiditatea de dezvoltare, trecând de la setarea caracteristicilor elementelor, poziția și modul de interacțiune al acestora de la celelalte sisteme de UI la doar scrierea a câtorva linii de cod.

3) IMGUI

IMGUI dispune de mai multe tipuri de Control, printre care:

- Butoane
- Slidere
- Toggle-uri
- Toolbar

Cu toate că IMGUI este dezvoltat în special pentru debugging, se pot personaliza multe aspecte din acesta folosind GUIStyles, într-o manieră care imită CSS.

2.1.3 Motorul de fizică

Unity folosește un motor de fizică propriu[4], care are rolul de a oferi componente care produc simulări fizice pentru utilizator, aceste componente fiind controlate prin scripturi.

Componente oferite:

1) Rigidbody

Componenta principală care activează comportamentul fizic pentru un GameObject.

Aceasta componentă "suprascrie" modul în care o componentă se mișcă, permițând aplicarea de forțe asupra obiectului și lăsând motorul de fizica să calculeze mișcarea. Prin urmare nu mai trebuie folosite scripturi care modifică direct poziția obiectului.

Dacă se dorește totuși ca un GameObject să aibă un Rigidbody fără ca mișcarea sa să fie controlată de către motorul de fizică, se poate activa proprietatea "IsKinematic" care permite ca mișcarea să fie controlată dintr-un script.

O optimizare adusă de către motor pentru a reduce numărul de calcule efectuate de procesor este de a trece obiectele care se mișcă cu o viteză mai mică decât un prag în modul de sleep, nemaifiind aplicate alte update-uri până când se interacționează cu o forță asupra obiectului "adormit". Totuși, din moment ce aceste forțe nu se pot aplica de către geometria statică, dacă se dorește ca podeaua să dispară la un moment dat, iar obiectul este în starea de "sleep", va rezulta într-un obiect care levitează. Pentru a rezolva astfel de cazuri, se poate "trezi" obiectul manual, dintr-un script.

2) Collider

Componenta care conferă un volum unui GameObject pentru detectarea de coliziuni.

Collider-ul este invizibil și trebuie doar să aproximeze forma mesh-ului unui GameObject. Există mai multe tipuri de collider-e:

- statice (care nu au un Rigidbody, adică nu se mișcă la coliziuni cu alte obiecte)
- dinamice (care au un Rigidbody) care la rândul lor sunt compound (care aproximează volumul obiectului prin forme simple) sau mesh (când este nevoie de geometria obiectului exactă pentru coliziuni, dar mult mai costisitoare din punctul de vedere al calculelor

La interacțiunea dintre collidere, suprafețele acestora trebuie să simuleze proprietățile materialelor pe care le reprezintă. Pentru a face acest lucru, se pot seta diferite proprietăți cum ar fi elasticitatea sau coeficientul de frecare folosind Physics Material.

[4] https://docs.unity3d.com/Manual/PhysicsSection.html

Când se petrece o interacțiune între collidere, motorul de fizică apelează anumite metode către orice script atașat obiectelor implicate, cum ar fi:

- OnCollisionEnter – semnalează primul cadru în care se detectează coliziunea
- OnCollisionStay – pentru cadrele în care este menținută coliziunea
- OnCollisionExit – semnalează terminarea coliziunii
- OnTriggerEnter, OnTriggerStay, OnTriggerExit – analog ca cele de mai sus, doar că un collider este setat ca trigger

Collider-ele interacționează diferit în funcție de setările acestora, tipurile principale fiind:

- Static Collider – folosit pentru geometria nivelului, nu au Rigidbody, iar coliziunile cu alte collidere nu le mișcă
- Rigidbody Collider – un GameObject cu un Collider și un Rigidbody (fără IsKinematic) atașat, pot să aibă coliziuni cu alte obiecte și sunt cele mai des folosite configurări
- Kinematic Rigidbody Collider – un GameObject cu un Collider și un Rigidbody Kinematic, ideal pentru obiecte care de obicei acționează ca geometrie statică, dar pentru care este nevoie de mișcare ocazional; ele aplică forțe altor obiecte la coliziune, spre deosebire de colliderele statice

3) Joint

Componentă care conectează rigid bodies, sau un rigid body de un punct fixat în spațiu. Cu ajutorul componentei se pot aplica forțe care mișcă obiectele atașate.

4) Character controller

Componentă folosită în principal pentru control first-person sau third-person atunci când nu are rost să se facă controlul cu Rigidbody. (mediu nerealist)

2.1.4 Umbre

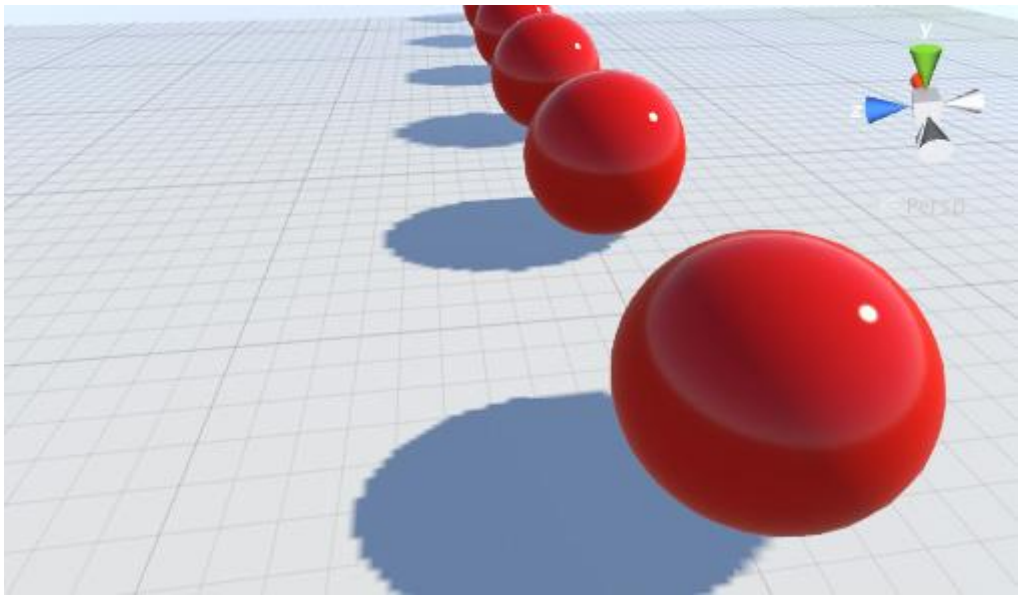

Figura 2.2: Cascade de umbre în Unity[5]

Umbrele în Unity[6] sunt create folosind tehnic shadow-mapping, tehnică în care este translatată poziția camerei la poziția sursei de lumină, de unde se randează scena folosind buffer-ul de adâncime într-o textură numită "shadow map". Acest shadow map este folosit ulterior în altă trecere de randare a scenei, de data aceasta din poziția normală a camerei, pentru a calcula ce vârfuri se află în umbră.

Această tehnică suferă de un fenomen numit perspective aliasing în cazul luminilor direcționale, în care umbrele obiectelor din apropierea volumului de vizualizare arată disproporționat.

Un mod de rezolvare a acestei probleme este creșterea rezoluției shadow map-ului, lucru care poate fi setat din opțiunile grafice, dar această rezolvare este destul de costisitoare ca performanță. Un alt mod de rezolvare al problemei folosit de Unity este folosirea unor cascade (figura 2.2), care împart frustum-ul camerei în mai multe zone (2 sau 5 pentru Unity), creând mai multe shadow map-uri, fiecare asignată unei zone.

Alt fenomen care poate să apară este "shadow acne", în care pixelii care ar trebui să fie exact la distanța specificată în shadow map sunt calculați ca fiind puțin mai îndepărtați, astfel pixelul respectiv se randează ca fiind în umbră, creând modele arbitrare de umbre pe suprafețe.

Pentru a preveni acest fenomen, Unity folosește un termen de bias care poate fi setat din interiorul programului și o tehnică numită "shadow pancaking", prin care se micșorează volumul de randare al luminii, sincronizând near plane cu frustum-ul camerei.

[5] https://docs.unity3d.com/Manual/shadow-cascades.html
[6] https://docs.unity3d.com/Manual/Shadows.html

2.2 Unreal Engine

Game engine dezvoltat de către Epic Games, cu prima apariție în anul 1998. Scopul inițial al motorului a fost pentru jocuri first-person shooter, dar a fost folosit cu succes pentru multe alte genuri de jocuri. Este scris în C++ și oferă un grad mare de portabilitate.

2.2.1 Arhitectura

Arhitectura Unreal Engine[7] este bazată pe obiecte.

Obiectul (UObject) este clasa de bază în ierarhie, fiecare Obiect din Unreal fiind un UObject.

Un Actor este o clasă care derivă din AActor (clasa de bază de gameplay). Un Actor poate fi văzut ca o entitate, foarte des folosind componente (care sunt niște obiecte specializate), pentru a conține valorile anumitor proprietăți sau pentru a defini anumite aspecte ale funcționalității.

Clasele de gameplay de bază includ funcționalități pentru reprezentarea jucătorilor, de asemenea pentru controlarea acestora cu AI. Mai există clase pentru afișare de display, de camere și pentru controlarea stării jocului – GameMode, GameState, PlayerState.

Reprezentarea jucătorilor, aliaților și inamicilor este făcută prin 2 clase:

- Pawn – un Actor care poate fi un agent; poate avea un Controller pentru mișcare/AI, poate interacționa cu jucătorii; un Pawn nu este neapărat controlat de către un jucător
- Character – un Pawn cu un CapsuleComponent pentru coliziune, CharacterMovementComponent pentru mișcare (plus replicare în cazul în care jocul este în rețea) și niște funcționalități de animație

Controlul unui Pawn se face printr-un Controller, care este responsabil pentru direcționarea acestuia. Acest Controller poate fi de 2 feluri:

- PlayerController – interfață între Pawn și jucător; oferă jucătorului posibilitatea să controleze Pawn-ul după bunul plac
- AIController – mișcarea Pawn-ului se face conform unui algoritm

Pentru a afișa informații către jucător exista:

- HUD – informație vizuala 2D direct pe ecran: gloanțe, viață etc.; fiecare PlayerController are câte un HUD
- Camera – PlayerCameraManger (camera în sine), fiecare PlayerController are câte una

Pentru a gestiona starea jocului, Unreal Engine dispune de:

[7] https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/index.html

- GameMode – reguli despre joc și condiții de câștig; ar trebui să existe doar pe server
- GameState – starea jocului; ar trebui să existe pe server, iar clienții să aibă acces la aceasta pentru actualizări
- PlayerState – informații despre jucători (scor, nivel etc.); există pentru fiecare client, se pot replica oricând de la un jucător la altul pentru ca jucătorii să fie sincronizați

2.2.2 UI

Unreal Engine folosește Unreal Motion Graphics UI Designer[8] pentru a crea elemente UI, pentru a crea meniuri, HUD-uri sau alte interfețe pe care dezvoltatorul vrea să le prezinte jucătorilor. La baza acestui sistem de UI sunt Widgets, care sunt funcționalități deja implementate pentru butoane, slidere, bare de progress etc. Pentru a construi o interfață se folosește un Widget Blueprint, cu 2 tab-uri pentru construcție în care trebuiesc definite aceste Widgets:

- Designer tab – schema vizuală a interfeței
- Graph tab – funcționalitățile pentru Widget-urile folosite

Interacțiunea cu Widget-urile se face prin intermediul evenimentelor, existând, de exemplu evenimente de OnClicked, OnPressed, OnReleased pentru un buton, logica găsindu-se în rutina de tratare a evenimentului.

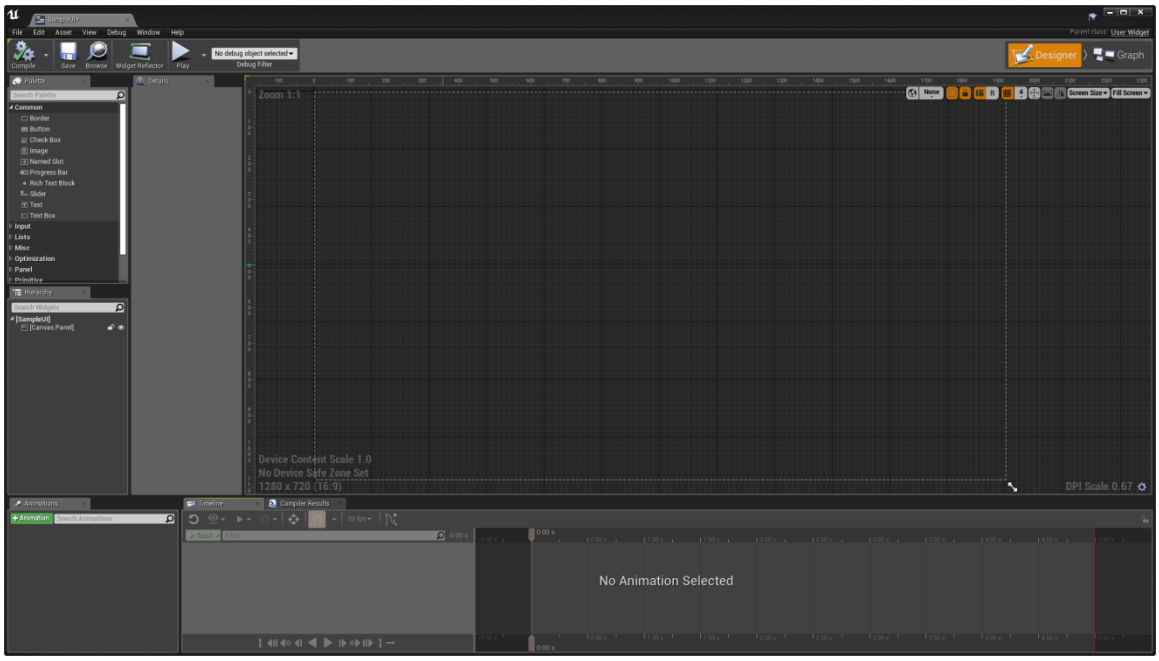

Figura 2.3: Sistemul de design UI în Unreal Engine[9]

[8] https://docs.unrealengine.com/en-US/Engine/UMG/index.html
[9] https://docs.unrealengine.com/en-US/Engine/UMG/HowTo/CreatingWidgets/index.html

2.2.3 Motorul de fizică

Unreal Engine se folosește de PhysX 3.3 de la Nvidia[10] pentru a rula simulările fizice și pentru a efectua toate calculele aferente coliziunilor.

Sistemul de coliziuni și de ray cast este reprezentat de Collision Responses, respectiv Trace Responses. Orice obiect care poate participa la coliziuni trebuie să aibă un Object Type și o serie de proprietăți care definesc cum va reacționa acesta la coliziunea cu alte obiecte. Prin aceste proprietăți, obiectele pot fi setate să ignore coliziuni, sau să fie complet afectate de acestea.

Trace Response funcționează la fel ca un Collision Response, doar că se referă la o rază efectuată în urma unui ray cast, fiecare obiect putând fie să interacționeze cu aceasta, fie să o ignore, în urma setărilor impuse.

Câteva interacțiuni importante din sistemul de fizică:

- Dacă doi Actori sunt setați pe Block coliziunea se va petrece, doar că, pentru ca evenimentul Event Hit să se transmită (folosit la Triggers, Blueprints etc.), este nevoie de activarea proprietății Simulation Generate Hit Events
- Dacă Actorii sunt setați pe Overlap, ei se vor ignora, iar evenimentele de coliziune nici nu vor fi generate dacă nu este activată proprietatea Generate Overlap Events.
- Pentru două obiecte simulate, ca acestea să se blocheze reciproc, trebuie să fie setate să blocheze Object Type-ul obiectului respectiv
- Pentru două obiecte simulate, dacă unul este setat să blocheze celălalt obiect, iar celălalt să se suprapună, suprapunerea se va petrece, dar nu și blocarea
- Evenimentele de suprapunere se pot întâmpla și în cazul în care obiectul este setat pe modul de blocare, iar acesta are o viteză foarte mare.
- Dacă un obiect este setat să ignore coliziunile, iar altul este setat să se suprapună cu alte obiecte, nu va exista un eveniment de Overlap

Pentru ca aceste simulări de coliziune să funcționeze, este nevoie de o formă care să reprezinte obiectul. În urma creșterii complexității obiectelor 3D, se folosesc forme simplificate ale acestora, pentru a scădea numărul de calcule efectuate. Clasa care se ocupă de acest proces în Unreal Engine este Physics Body (BodyInstance), care reprezintă forma simplificată a obiectului. Aceasta poate fi formată din cutii, sfere, capsule sau ”convex hull” (cel mai mic set convex care conține toate vârfurile obiectului).

[10] https://docs.unrealengine.com/en-US/Engine/Physics/index.html

2.2.4 Umbre

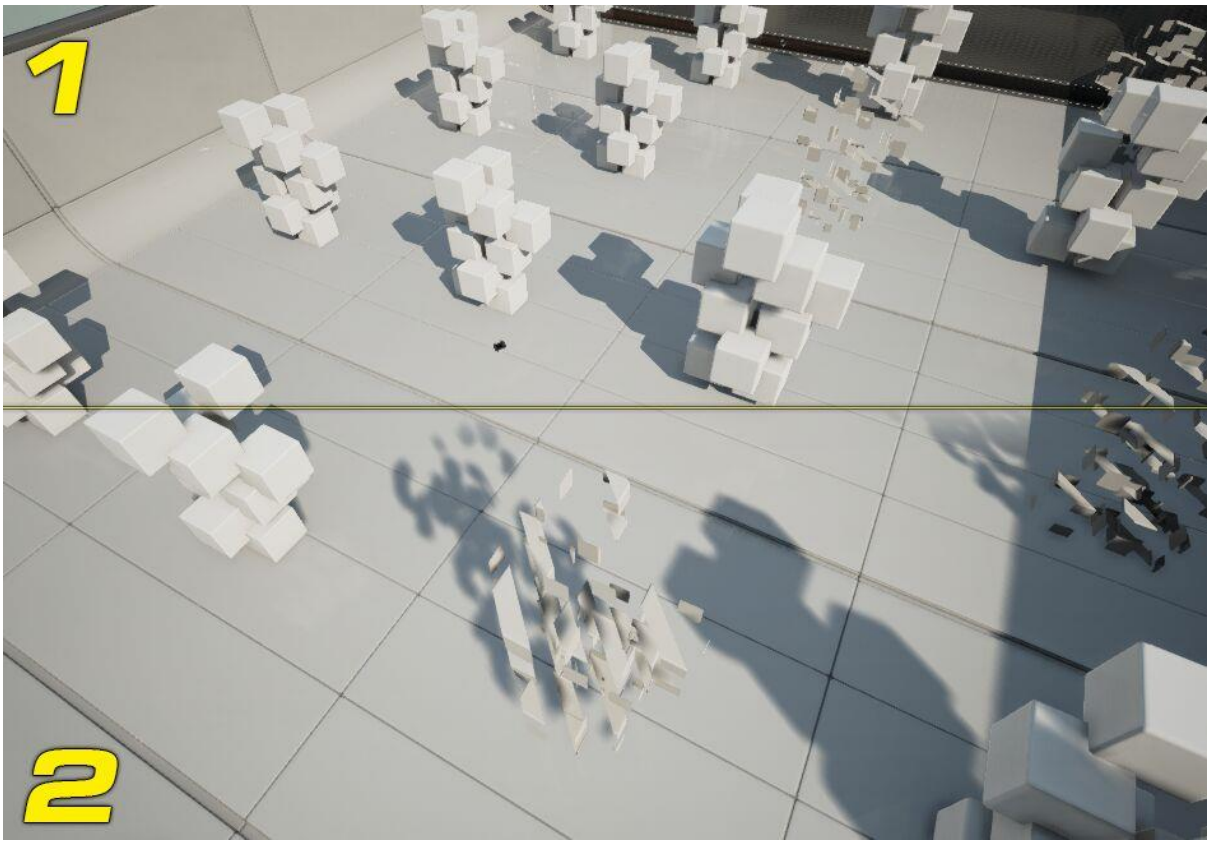

Figura 2.4: Umbre în Unreal Engine într-o scenă statică și într-o scenă dinamică[11]

Umbrele sunt o parte importantă din întreaga scenă, făcând ca obiectele să pară ancorate în lume, dându-le acestora un sens de adâncime și spațialitate. În Unreal Engine exista 4 tipuri de umbre[12]:

1) Static Lights:

   Luminile statice au lumină și umbră complet statică, care nu au nici un efect asupra obiectelor dinamice. Datorită acestui fapt, ele sunt foarte bune din punct de vedere computațional.

2) Directional Light Cascading Shadow Maps:

   La fel ca în Unity, se folosește tehnica de Cascading Shadow Maps pentru a randa umbrele obiectelor dinamice din toată scena, doar că, la o anumită distanță (care poate fi setată), se face tranziția în umbre statice, pentru a avea umbre dinamice doar în apropierea jucătorului.

[11] https://docs.unrealengine.com/en-US/Resources/ContentExamples/DynamicSceneShadows/index.html
[12] https://docs.unrealengine.com/en-US/Engine/Rendering/LightingAndShadows/Shadows/index.html

3) Stationary Light Shadows:

   Obiectele dinamice trebuie să integreze umbra statică ale scenei pe acestea. Acest lucru este realizat cu umbre ”Per Object”, care în esență sunt 2 umbre, una care se ocupă de randarea umbrelor statice pe obiect, iar alta care se ocupă de randarea umbrei obiectului în scenă. Acest lucru se face tot cu Shadow Map, aceasta folosindu-se de marginile obiectului. În cazul a multor obiecte dinamice în scenă, este mai eficientă folosirea unei Movable light.

4) Dynamic Shadows:

   Movable lights se folosește doar de umbre și lumină dinamica, astfel ca orice obiect din scenă va căpăta și va proiecta mai departe umbra dinamică de la această lumină. În medie, aceste lumini sunt cele mai scumpe din punct de vedere computațional.

2.3 CryEngine

Game engine dezvoltat de către compania Crytek, folosit pentru toate jocurile dezvoltate de aceștia (cum ar fi Far Cry). Este scris în C++, Lua și C# și rulează pe mai multe platforme.

2.3.1 Arhitectura

Arhitectura folosită de către motorul de joc CryEngine[13] este una bazată pe entități și componente, diminuând nevoia de a avea cod scris pentru a gestiona entitățile din scenă. De asemenea, sistemul este gândit în așa fel încât să fie intuitiv pentru dezvoltator.

Entitatea este gândită ca un container pentru componente, componente care conțin logica necesară jocului. Exemple de componente de bază:

- Mesh
- Lights
- Character Controllers

Cu aceste componente se pot crea prefabs, care pot fi amplasate în scenă.

Entitățile și componentele pot fi create cu Create Object Tool, putând crea noi entități și având posibilitatea selectării componentelor necesare entității, precum și edita anumiți parametrii specifici acestora.

Prefabs sunt o grupare de obiecte care pot fi amplasate în scenă. O dată amplasate, acestea devin instanțe, toate instanțele ale aceluiași prefab au aceleași proprietăți. (alterarea unui prefab are ca efect alterarea tuturor instanțelor acestui obiect)

[13] https://docs.cryengine.com/display/CEMANUAL/Entity+Components

Prefabs comunică prin evenimente cu ajutorul instrumentului Flow Graph. (utilitar propriu CryEngine pentru controlare de evenimente și logică în interiorul nivelelor)

Mai există un tip de obiect și anume Archetype Entity, care se bazează pe Entitatea normală, având posibilitatea de schimbare a parametrilor acestuia. Dacă valoarea unui parametru este schimbată, atunci toate instanțele arhetipului din scenă vor fi actualizate.

2.3.2 UI

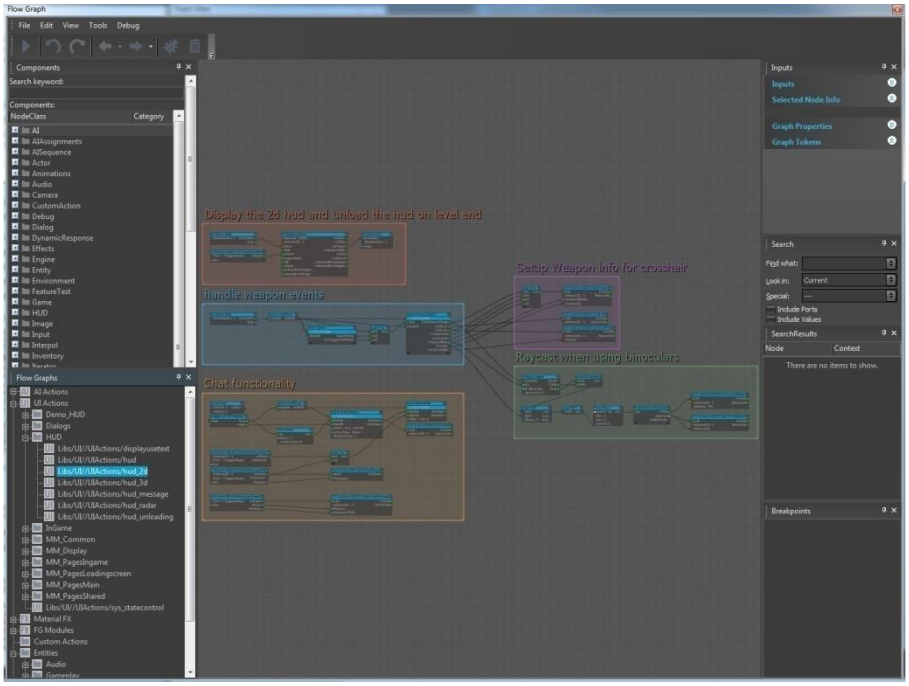

Figura 2.5: Crearea logicii de UI[14]

CryEngine folosește Scaleform[15] pentru sistemul de interfață utilizator. Acesta este un middleware care permite crearea de interfețe folosind suita Adobe Flash, fișierele rulate de către acestaa sunt folosite pentru a implementa meniuri, HUD etc.

Elementele principale ale sistemului de UI sunt:

- UI Elements – asset-uri Flash care reprezintă ceea ce se vede pe ecran și conțin și logică folosită de către Flow Graph sau accesibilă din cod C++; acestea reprezintă unitatea de bază peste care se construiește interfața
- UI Actions – folosite pentru a face legătura între logica UI și codul jocului
- UI Event System – simplifică legătura între logica UI, codul jocului și Flow Graph; trimite evenimente bidirecționale

[14] https://docs.cryengine.com/display/CEPROG/UI+Action
[15] https://docs.cryengine.com/display/CEPROG/User+Interface

CryEngine dispune de un sistem de localizare care permite localizare de text pentru interfața utilizator. Pe lângă localizare de cuvinte, este posibilă și selectarea unui alt font în funcție de limba utilizată.

2.3.3 Motorul de fizică

Simularea fizicii în CryEngine este realizată fie de către motorul de fizică propriu CryPhysics[16], fie de către PhysX de la Nvidia[17]. (la alegerea utilizatorului)

CryPhysics are propriul set de entități și geometrie, asociat cu obiectele/particulele din scenă. În mod implicit, motorul de fizică rulează pe un thread separat, cu un număr de ”worker threads” configurabil.

Geometria care poate fi folosită pentru a reprezenta o entitate în motorul de fizică poate fi creată prin apelul a uneia dintre următoarele 2 funcții prin intermediul GeometryManager:

- CreateMesh – creează geometrie pe baza informațiilor vârfurilor și indicii obiectului; geometria creată ia forma unui box sau a unui arbore de astfel de boxes
- CreatePrimitive – creează geometrie primitivă (cylinder, sphere, box etc.)

După crearea geometriei, ea poate fi înregistrată în motorul de fizică folosind funcția RegisterGeometry.

Entitățile fizice trebuie create prin intermediul funcției CreatePhysicalEntity.

Câteva tipuri de entități sunt:

- PE_STATIC – entitate statică, poate fi mișcată manual din afara lumii fizice
- PE_RIGID – rigid body; poate să nu fie simulat prin setarea masei infinită (0), el va interacționa totuși cu celelalte entități
- PE_ARTICULATED – structură de rigid bodies conectate prin joints (cel mai des folosită pentru efectul de ragdoll)

La fiecare cadru, motorul de fizică este notificat să ”pășească”, la finalul acestui ”pas” trimițând notificări sub formă de evenimente. Pentru a primi aceste evenimente, sistemele trebuie să se înregistreze către ”listeners”, utilizând un apel ”AddEventClient”.

Există opțiuni de profiling și de debug draw al entităților din motorul de fizică utilizând anumite comenzi în consolă. (p_draw_helpers, p_profile)

[16] https://docs.cryengine.com/display/CEPROG/CryPhysics
[17] https://docs.cryengine.com/display/CEPROG/Using+NVIDIA+PhysX+in+CRYENGINE

2.3.4 Umbre

Umbrele in CryEngine[18] se generează folosind tehnica shadow mapping, fiecare obiect proiectându-și umbra în scenă.

Deoarece umbrele sunt foarte costisitoare din punctul de vedere al performanței, CryEngine dispune de mai multe optimizări:

Figura 2.6: Umbre în CryEngine[19]

1) Cached Shadows:

   O metodă de optimizare pentru a reduce numărul de apelul de desenare al umbrelor, în mod direct crescând distanța maximă de randare a umbrelor. Această metodă se folosește de faptul că umbrele foarte distanțate sunt mult mai greu de ”apreciat” față de cele apropiate de cameră, astfel că acestea vor fi desenate o singură dată, următoarele cadre folosind cascadele de shadow map deja calculate anterior. Când camera se va apropia prea mult de aceste cascade, ele se vor actualiza.

2) Per Object Shadows:

   Cu această tehnică, un obiect are asignat propriul shadow map, rezultând într-o umbră de o mai mare calitate datorită numărului mai mare de texeli de umbră. În mod direct nu este o tehnică de optimizare, deoarece implică mai multe apeluri de desenare și un consum mai mare de memorie. Totuși, CryEngine poate

[18] https://docs.cryengine.com/display/SDKDOC2/Shadows
[19] https://docs.cryengine.com/display/SDKDOC2/Shadow+Proxies

optimiza acest tip de umbre, grupând materialele similare ale obiectelor și utilizând un singur apel de desenare pentru acestea.

3) Shadow Proxies:

   O altă metodă de a reduce costurile de performanță ale umbrelor prin crearea unei geometrii dedicată umbrei obiectului. Astfel se poate reduce drastic numărul de triunghiuri randate, fără a pierde din calitatea umbrei.

3 TEHNICI

În continuare, se va trece peste tehnicile folosite pentru a implementa anumite sisteme din motorul de joc:

3.1 Animație scheletală

Reprezintă una dintre modalitățile capabile să facă un caracter să se miște. Tehnica animațiilor scheletale (Luna, 2004) presupune reprezentarea unui caracter prin doua structuri (care pot fi create de către artiști în diferite aplicații 3D (cum ar fi Blender, Maya), și anume:

- Modelul – partea vizuală a acestuia, reprezentat de colecția de vârfuri, indici, texturi etc.
- Scheletul – o ierarhie logică care mapează în mod direct părți din model către ”oase”

Oasele sunt caracterizate printr-o transformare 3D compusă din translație, scalare și rotație și influențează în mod direct vertex-i din modelul caracterului. Din moment ce acestea sunt într-o ierarhie (mai specific într-un arbore, fiecare os având cel puțin un părinte), poziția finală a unui os poate fi influențată și de către transformările oaselor care se găsesc mai sus în ierarhie. Numărul de influențe este de regulă limitat din considerente de performanță, fiecare transformare care participă la calculul poziției finale va avea o pondere specifică. Acest lucru se poate face direct pe GPU, putând fi trimise, pentru fiecare vertex, o colecție cu transformările fiecărui os, alături de indicii oaselor care influențează transformarea acestuia și ponderile lor. Astfel, o animație poate fi reprezentată printr-o colecție de astfel de transformări 3D compuse, care duc oasele dintr-o poziție în alta. Aceste transformări sunt prezente în fișierul specific animației, sub numele de ”keyframes”, algoritmul fiind nevoit să interpoleze între acestea în funcție de timpul curent din animație pentru ca modelul să aibă o mișcare fluidă.

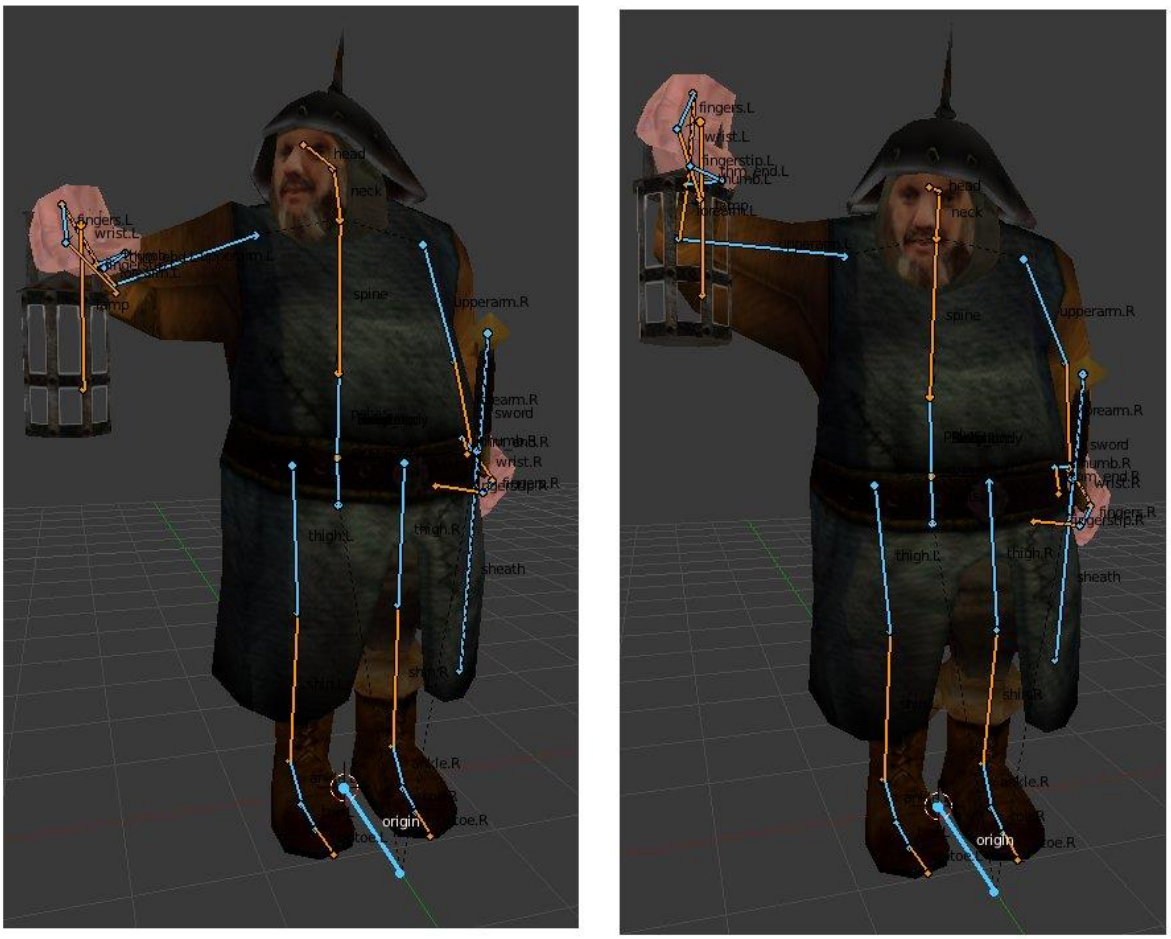

Figura 3.1: Un model și transformările oaselor la 2 cadre diferite (Luna, 2004)

La tranziția dintre doua animații, motorul de joc face blending între acestea, după o metodă proprie. Prin setarea unui timp de tranziție între acestea, motorul face blend prin considerarea următorului keyframe din prima animație ca fiind keyframe-ul interpolat la t = timpul de tranziție din a doua animație.

Desigur, aceasta tehnică (Luna, 2004) se poate folosi și pentru alte întrebuințări (în afară de animații), cum ar fi deformarea unor obiecte sau simularea unor fenomene fizice (în limita sistemului ierarhic de oase).

3.2 Directional Shadow Mapping

Directional Shadow Mapping (Vries, 2020) este una dintre tehnicile de randare a umbrelor prin verificarea fiecărui pixel dacă este în umbră sau nu.

Principiul funcționării algoritmului este randarea scenei de două ori: o dată din perspectiva luminii, iar a doua oară din perspectiva camerei.

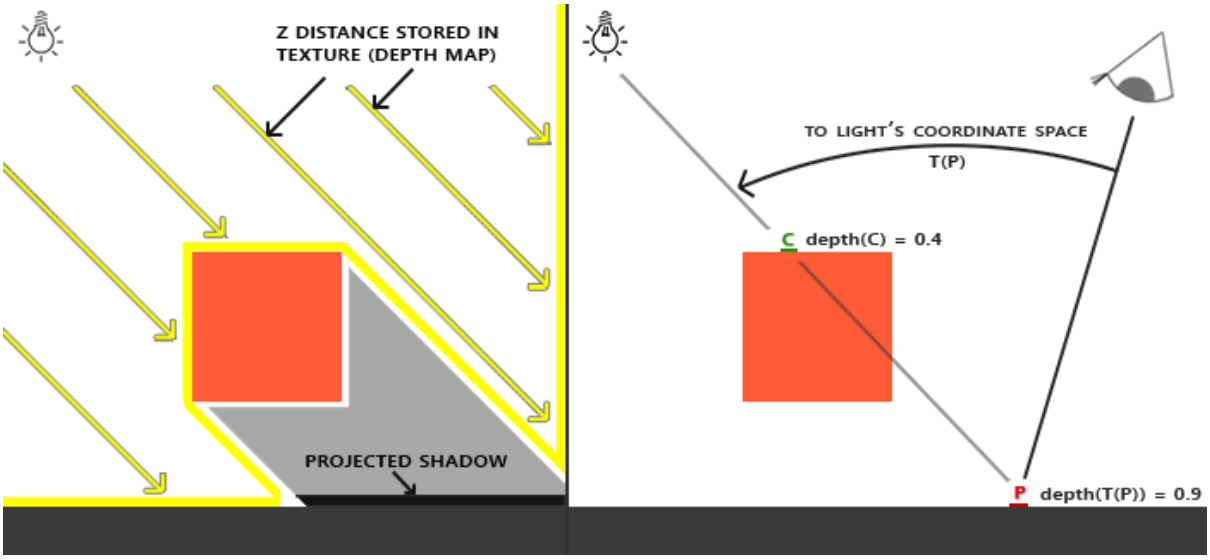

Figura 3.2: Cele două treceri de desenare ale algoritmului (Vries, Shadow-Mapping, 2020)

Prima trecere de randare (cea din poziția sursei de lumină), ilustrată în partea stângă a figurii 3.2, presupune verificarea fiecărui vertex dacă acesta se află în umbră sau nu. Acest lucru se poate face cu ajutorul algoritmului z-buffer, care va calcula componenta de adâncime a fiecărui vârf aflat în spațiul de coordonate normalizate. În urma calculelor, vârful cu cea mai mică valoare de adâncime va fi înserat în depth buffer (mapare directă de la coordonatele 2D ale vârfului către poziția din depth buffer). Informațiile din depth buffer vor fi stocate într-o textură 2D, care va fi folosită ulterior pentru a determina care pixeli sunt în umbră. De asemenea, deoarece lumina este direcțională, proiecția camerei va fi una ortografică, deoarece modelează perfect o lumină care are razele paralele și este amplasată la infinit.

A doua trecere de randare, ilustrată în partea dreaptă a figurii 3.2, presupune pe lângă transformarea în spațiul de decupare din perspectiva camerei, transformarea poziției fiecărui vertex în spațiul de coordonate al luminii. Acest lucru se poate face prin înmulțirea poziției 3D locale a vârfului cu matricea compusă din matricea de vizualizare și matricea de proiecție a camerei utilizată în trecerea precedentă. După acest pas, coordonatele fragmentului reprezentat de vârfurile transformate anterior vor avea valori în domeniul $[-1, 1]$. Aceste coordonate trebuie aduse în domeniul $[0, 1]$pentru a putea fi folosite asemeni coordonatelor de textură. După această transformare, se va folosi X-ul și Y-ul pentru a eșantiona din textura generată la prima trecere, rezultatul fiind coordonata z asociată celui mai apropiat fragment de sursa de lumină. De aici urmează doar o simplă comparație între valoarea eșantionată și adâncimea fragmentului, dacă adâncimea fragmentului este mai mare atunci este în umbră, altfel nu.

Figura 3.3: Aliasing (Vries, Shadow-Mapping, 2020)

Shadow Mapping (Vries, 2020) are totuși problemele ei, calitatea umbrei depinzând foarte mult de rezoluția folosită pentru textură. De asemenea, parametrii folosiți pentru proiecția ortografică influențează direct calitatea, putând apărea aliasing (figura 3.3). O metodă de rezolvare al aliasing-ului implementată de către motorul grafic este PCF (Percentage-closer Filtering) care implică verificarea texelilor vecini fiecărui fragment pentru care se dorește testarea dacă se află în umbră sau nu și să se facă media valorilor de umbră ale acestora.

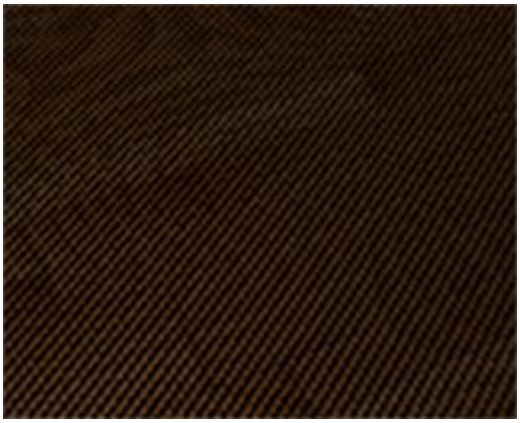

Figura 3.4: Shadow Acne (Vries, Shadow-Mapping, 2020)

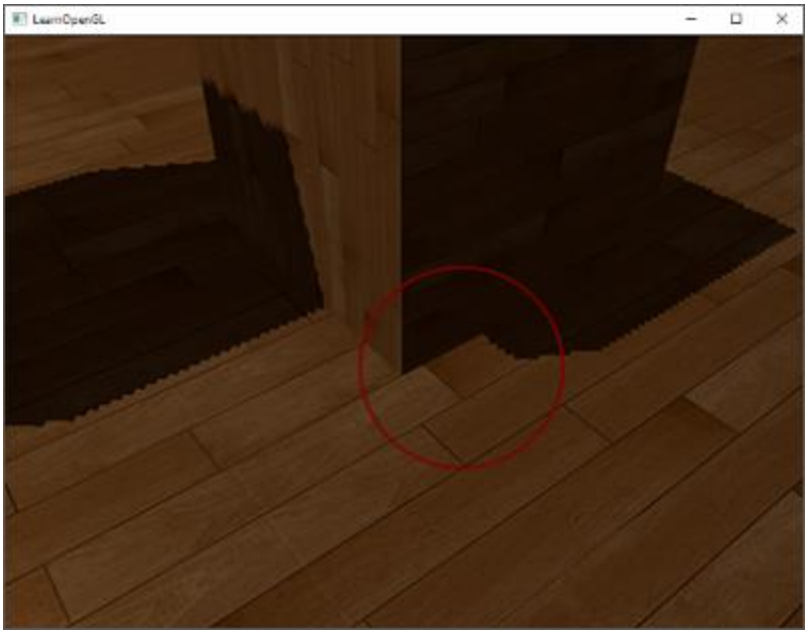

Figura 3.5: Peter Panning (Vries, Shadow-Mapping, 2020)

O altă problemă care poate apărea este shadow acne (figura 3.4), care creează modele arbitrare de umbre pe suprafețe, datorată rezoluției scăzute a texturii. O metodă de rezolvare este introducerea unei margini de eroare care va fi scăzută din adâncimea fragmentului care se dorește a fi testat. Dar și această abordare are problemele ei, în cazul setării unei valori prea mare de bias, va apărea fenomenul de peter panning (figura 3.5), în care umbra unui obiect prezintă un offset față de acesta. Pe lângă setarea corectă a factorului de bias, se poate încerca și activarea front culling înainte de trecerea de desenare a umbrelor pentru a rezolva acest fenomen.

O altă tehnică de vizualizare a umbrelor este Shadow Volume[20], prin care se transmite o rază dinspre sursa de lumină către fiecare vertex al obiectului, procedeu în urma căruia se crează un volum care reprezintă pixelii care sunt situați în umbră. Avantajul principal al acestei tehnici este calitatea umbrei, care nu depinde de mărimea texturii în cazul procedeului Shadow Mapping (Vries, 2020). Totuși, un dezavantaj al acestei metode este timpul în care se execută algoritmul, în cele mai multe cazuri tehnica de Shadow Mapping (Vries, 2020) fiind superioara. (și, de altfel, motivul pentru care s-a ales această metodă)

3.3 Omnidirectional Shadow Mapping

Tehnica omnidirectional shadow mapping (Vries, Point-Shadows, 2020) este foarte asemănătoare cu cea de Directional Shadow Mapping (Vries, 2020), diferența principală este sursa de lumină care este punctiformă. Datorită acestui lucru, este nevoie de 6 texturi pentru fiecare parte a luminii: față, spate, sus, jos, stânga, dreapta. Pentru a nu avea 6 texturi diferite, se creează un cubemap cu acestea.

[20] http://ogldev.atspace.co.uk/www/tutorial40/tutorial40.html

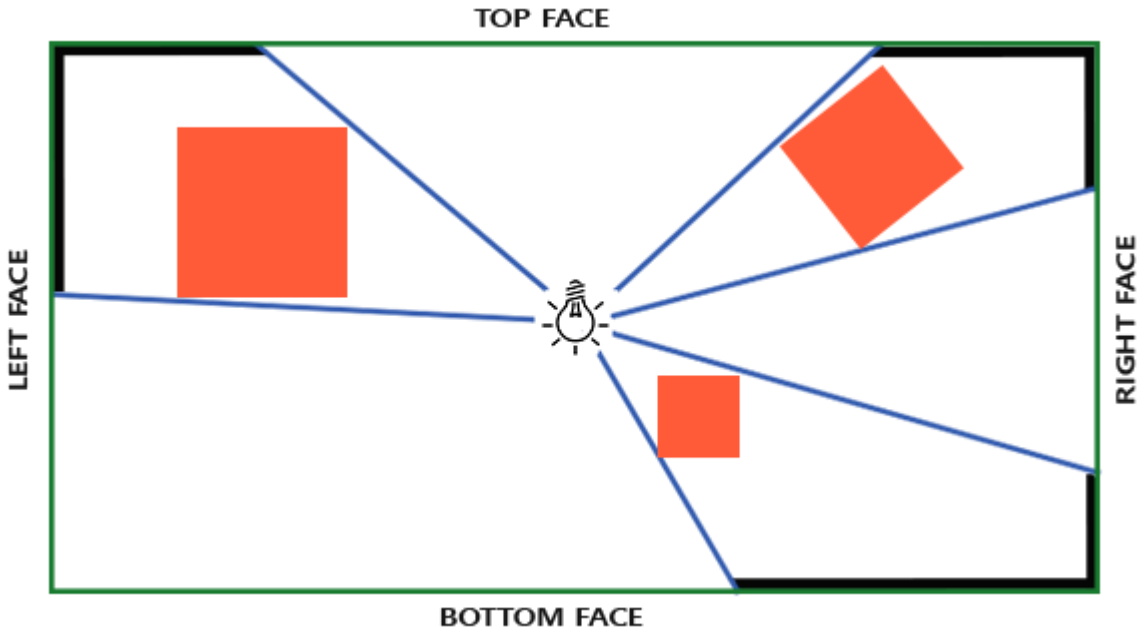

Figura 3.6: Principiul funcționării algoritmului (Vries, Point-Shadows, 2020)

În prima trecere de randare, se creează o cameră cu proiecția perspectivă cu field of view de 90°. Pentru a crea cele 6 fețe ale cubemap-ului, se folosesc 6 matrice de vizualizare, câte una pentru fața respectivă. Pentru restul trecerii, procedeul este aproximativ identic cu cel de la umbrele direcționale, produsul final fiind un cubemap cu fețele reprezentări ale celor 6 depth buffere.

În cea de-a doua trecere, trebuie pasat ca parametru către shader și poziția luminii folosită în trecerea anterioară. Cu aceasta se va construi vectorul către poziția fragmentului, vector care va fi folosit pentru eșantionarea texturii cubemap-ului, eșantionare care va întoarce adâncimea limită pentru ca fragmentul să fie în lumină. Mai departe se va compara lungimea vectorului construit anterior cu această limită, iar dacă lungimea este mai mică înseamnă că fragmentul este în umbră.

Problemele acestei abordări sunt aceleași ca la Directional Shadow Mapping (Vries, Shadow-Mapping, 2020), având aceleași rezolvări. O diferență ar fi la PCF, unde eșantionarea pixelilor vecini se va face și în adâncime.

3.4 Sistemul de componente

Pentru a ușura dezvoltarea jocurilor și de a evita supraîncărcarea claselor, motorul de joc folosește un sistem de componente care presupune decuplarea funcționalităților unui obiect în componente. Obiectul va avea mai multe astfel de componente, rolul său este de a le actualiza la fiecare cadru nou și de a le permite acestora să comunice între ele, pentru a evitata cuplarea strânsă între acestea. Ideea din spatele sistemului este de a face obiectele

flexibile prin metode care pot adăuga/scoate componente la runtime, putând schimba felul în care obiectul face anumite lucruri.

Obiectele sunt întreținute direct de către motor, acesta ocupându-se de actualizarea obiectelor (care la rândul lor actualizează componente), de adăugarea/ștergerea acestora și de trimiterea evenimentelor de input către componente.

Obiectele se creează în principal printr-un factory, care deține mai multe șabloane de configurări de obiecte folosite în joc, dar obiectele se pot crea și manual, iar componentele adăugate pe urmă.

Componentele conțin pe lângă logica necesară gameplay-ului și logica pentru ca obiectele să poată fi întreținute de diverse sisteme. În caz concret, există o componentă de bază care se ocupă de adăugarea automată a obiectelor în motorul de fizică și o componentă care se ocupă de transmiterea anumitor parametrii către shader și de desenarea obiectului. Pentru a crea noi componente, acestea trebuie să derive fie dintr-o componentă de bază, fie pot să derive dintr-o componentă nou creată anterior pentru a adăuga/suprascrie funcționalități.

Avantajul acestei abordări față de cea în care se scriu clase uriașe, cu foarte multe funcționalități, pentru un obiect, este productivitatea crescută, deoarece se poate lucra în paralel mult mai ușor pentru dezvoltarea componentelor diferite necesare unui obiect și, de asemenea, evitarea codului duplicat prin refolosirea componentelor. Un alt factor important este ușurința înțelegerii unui sistem nou atunci când acesta este decuplat, nemaifiind nevoia înțelegerii tuturor sistemelor care compun un obiect și modul în care acestea comunică. Un dezavantaj este overhead-ul indus de ”spargerea” claselor în componente, deoarece acestea trebuiesc actualizate separat. Totuși, overhead-ul nu este semnificativ decât în cazul a foarte multor obiecte cu foarte multe componente.

3.5 Coliziuni folosind motorul de fizică Bullet

Pentru sistemul de fizică, motorul de joc folosește Bullet Physics (Coumans, 2015), cu capabilități de simulare pentru detectarea coliziunilor, dinamica corpurilor rigide și deformabile. Dintre acestea, motorul de joc folosește detectarea coliziunilor, care implică și adăugarea de corpuri rigide în lumea simulată de acesta, doar că acestea au masa infinită și mișcarea nu este simulată de motorul de fizică, ci este executată de componente.

Metoda prin care se face detectarea coliziunilor este raycast, prin care se trimite o rază de la o anumită poziție către o altă poziție, iar dacă această rază intersectează un obiect, se va întoarce obiectul și o valoare cuprinsă în intervalul $[0, 1]$ care reprezintă fracțiunea din distanța dintre cele doua poziții.

Motorul de fizică este înfășurat într-o clasă care are metode de inițializare a acestuia, de a face un pas în simulare, de adăugare a unui obiect din scenă în lumea fizică și de mapare a acestei legături, două metode de raycast pentru distanța până la un obiect, respectiv pentru

obiectul lovit de rază și o metodă care, folosind maparea anterioară, face maparea inversă, de la obiectul din lumea fizică la obiectul din scenă.

Un caz de utilizare al sistemului de detectare de coliziuni util pentru motorul de joc este pentru tehnica de raypicking. Această metodă constă în transformarea din spațiul ecran a coordonatelor mouse-ului în spațiul lumii. Acest lucru se poate face prin trecerea din spațiul ecran în NDC prin împărțirea coordonatelor la dimensiunea ferestrei și trecerea din acest domeniu $[0, 1]$într-un domeniu cu valori cuprinse în intervalul [-1, 1]. După acest pas, folosind transformările inverse de proiecție și de vizualizare, se ajunge în spațiul lumii. Tot acest proces este inclus în pachetul matematic GLM folosit de către motor, prin metoda unProject. După ce este extrasă poziția în spațiul lumii, se calculează direcția către aceasta din perspectiva poziției camerei, se va face un raycast de la poziția camerei către direcția calculată anterior înmulțită cu un factor, iar rezultatul va fi obiectul (sau locația) care se intersectează cu raza.

3.6 Modelul de iluminare Phong

Modelul de iluminare pe care-l folosește motorul de joc la randare este modelul Phong (Phong, 1975). Acest model este unul empiric, un model de iluminare globală fiind prea costisitor din punct de vedere al performanței pentru a putea fi folosit într-un joc.

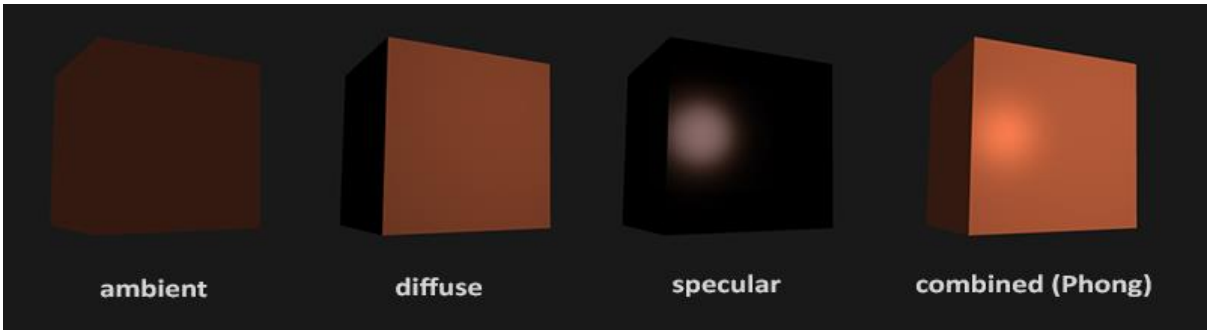

Figura 3.7: Cele 3 componente de lumină ale modelului Phong (Phong, 1975)

Componentele principale (prezentate vizual în figura 3.7) pentru acest model sunt:

- Lumina ambientală
- Lumina difuză
- Lumina specular

Lumina ambientală aproximează faptul că, în lumea reală, tot timpul va exista o sursă de lumină, fie ea si foarte îndepărtată, care îi va oferi obiectului puțină culoare, chiar și când acesta este în umbră. Este reprezentată de un factor constant k_a.

Lumina difuză este componenta care simulează impactul direcțional al luminii asupra obiectului. Această lumină este dată de măsura unghiului dintre normala suprafeței ($\widehat{N}$) și vectorul direcției luminii normalizat ($\widehat{D}$), rezultată din produs scalar, înmulțită cu factorul

constant k_d. De asemenea, dacă unghiul dintre cei doi vectori este mai mare de $\frac{\pi}{2}$, produsul scalar trebuie să devină 0.

Lumina specular se referă la proprietatea obiectului de a reflecta lumina. Lumina specular se folosește și ea de direcția luminii și de normala suprafeței, pe lângă direcția vizualizării ($\hat{V}$). Cu cât observatorul este mai apropiat de unghiul reflectat de suprafață ($\hat{R}$), cu atât intensitatea luminii va fi mai puternică. Astfel, calculul este dat de $k_s * \left(\hat{V} \cdot \hat{R}\right)^n$, unde k_s este constanta specular, iar n este exponentul materialului de reflexie. Din nou, având în vedere faptul că produsul scalar să devină 0 dacă unghiul este mai mare de $\frac{\pi}{2}$.

Dacă lumina este punctiformă și nu direcțională, componenta difuză și cea specular trebuie înmulțite cu un factor de atenuare $a = \frac{1}{(1+ d^2)}$, unde d reprezintă distanța de la sursa de lumină până la fragment.

Având toate aceste componente, lumina finală calculată de model (fiind incluse și umbrele) este:

$$L = k_a * L_a + umbră * \left(k_d * L_d * \left(\hat{D} \cdot \hat{N}\right) + k_s * L_s * \left(\hat{V} \cdot \hat{R}\right)^n\right)$$

$$+ a * \sum_{l \in lumini\ punctiforme} k_a * l_a + umbră * \left(k_d * l_d * \left(\widehat{D_l} \cdot \hat{N}\right) + k_s * l_s * \left(\hat{V} \cdot \widehat{R_l}\right)^n\right)$$

unde:

L_a, L_d, L_s = culoarea luminii ambientale, difuze respectiv speculare pentru lumina direcțională

l_a, l_d, l_s = culoarea luminii ambientale, difuze respectiv speculare pentru lumina punctiformă

3.7 Immediate mode GUI

Motorul se folosește de Dear ImGui (Dear ImGui) pentru sistemul de creare de interfețe utilizator. Paradigma Immediate mode GUI presupune că aplicația se ocupă de randarea elementelor de UI și că sistemul de UI trebuie notificat la fiecare nou cadru.

Integrarea acestei biblioteci într-un motor de joc scris în C++ se face foarte simplu, fiind menținute fișiere de configurare pentru diferite combinații de platforme, renderers și framework-uri pe repository-ul de github.

Dear ImGui (Dear ImGui) își creeaza propriile buffere de vertexi care pot fi randate oricând de către aplicație, într-un mod eficient, instanțiat și cu un număr mic de draw call-uri către GPU. Principalul scop al bibliotecilor de GUI immediate mode este de a oferi un model de creare de interfețe simplu de folosit, fără un overhead mare. Deoarece aceste biblioteci sunt create cu simplitate de dezvoltare în minte, unele funcționalități găsite în alte tipuri de sisteme de GUI pot să lipsească. Biblioteca poate fi folosită pentru:

- Afișare de text pe ecran
- Creare meniuri
- Creare layout-uri
- Afișare culori
- Modificare variabile la runtime prin slidere
- Etc.

Aceste funcționalități sunt disponibile prin simpla apelare a unor funcții statice de oriunde din program (după apelarea funcției ImGui::NewFrame), cum ar fi ImGui::SliderFloat sau ImGui::BeginText, sistemul de GUI ocupându-se de stocarea și aranjarea vertexilor pentru eficientizarea apelurilor de desenare, aplicația având acces la acestea după ce API-ul de UI este notificat că s-a terminat cadrul prin ImgUi::EndFrame. Datele necesare randării sunt accesate folosind ImGui::GetDrawData.

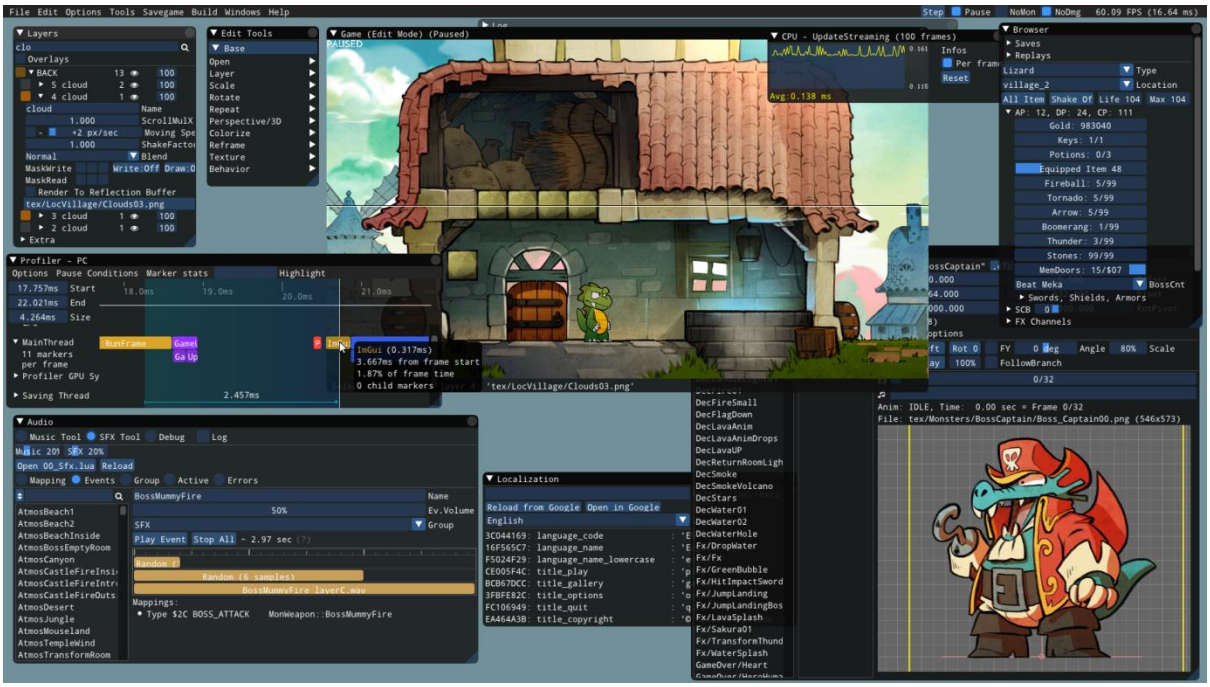

Figura 3.8: O interfață de debug a unui joc realizată folosind Dear ImGui (Dear ImGui)

4 IMPLEMENTARE

Pentru a implementa motorul de joc, a fost nevoie de găsirea unui limbaj de programare cu suport extins pentru:

- API pentru randare – cum procesul de randare eficientă este foarte complex, un API este necesar
- Programare orientată obiect – pentru dezvoltarea rapidă, design patterns, abstractizare, fiind nevoie de o comunicare eficientă între modulele dezvoltate
- Performanța – este nevoie de un limbaj performant, deoarece scopul motorului este de a fi cât mai eficient posibil
- API pentru tratare evenimente input – stă la baza unui joc, este necesar
- Bibliotecă pentru matematică – foarte utilă, operații matematice folosind matrici și vectori stau la baza jocurilor
- Bibliotecă pentru fizică – ar trebui să existe pentru majoritatea limbajelor
- Bibliotecă pentru UI – la fel ca la biblioteca de fizică

Având în vedere factorii de mai sus, limbajul de programare care se potrivește cel mai bine cerințelor este C++. De asemenea, framework-ul de la materia SPG (UPB, 2018) este scris în C++, prin urmare s-a decis dezvoltarea motorului bazat pe acest framework.

API-ul care este folosit pentru randare este OpenGL, alături de limbajul de shading GLSL și biblioteca GLFW care oferă abilitatea de creare a ferestrelor și de tratare a evenimentelor de input de la tastatură, joystick sau mouse.

Pentru matematică s-a folosit biblioteca GLM, care oferă un pachet complet pentru un software grafic.

Pentru biblioteca de fizică s-a folosit Bullet Physics (Coumans, 2015), o bibliotecă open source foarte performantă.

Biblioteca de UI folosită este Dear ImGui (Dear ImGui), un API immediate mode pentru dezvoltarea rapidă de tool-uri, interfețe simple și interfețe de debug.

În timpul dezvoltării s-a încercat decuplarea modulelor dezvoltate cât mai mult posibil, realizând comunicarea dintre acestea prin design pattern-uri cum ar fi: Singleton, Component, Observer.

4.1 Arhitectura aplicației

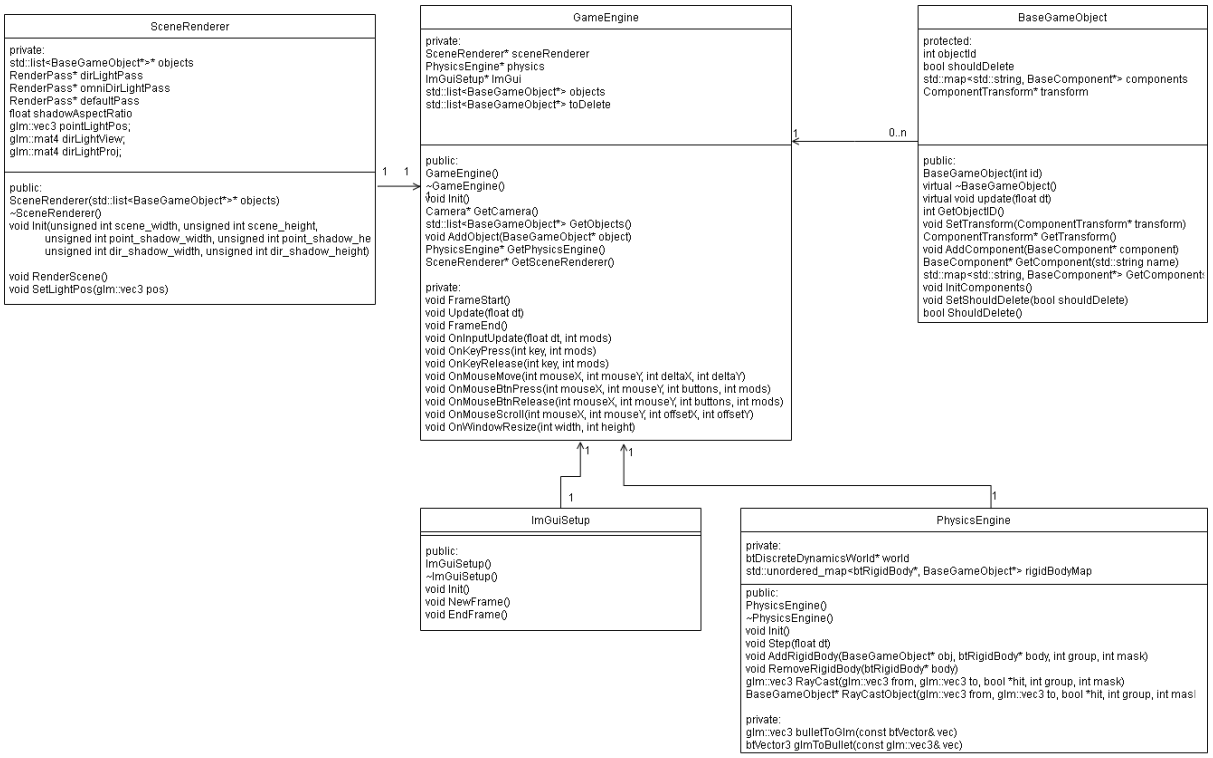

Figura 4.1: Diagramă de clase pentru arhitectura modulelor principale din aplicație

Aplicația este împărțită în 4 module conform figurii 4.1, mai departe se va intra în detaliu, explicând întrebuințările fiecăruia:

4.1.1 Modulul de control

Motorul rulează într-o buclă infinită, care se oprește doar la închiderea aplicației. La fiecare iterație din buclă, rolul modulului de control este de a actualiza celelalte module și obiectele din scenă. Din acest fapt, în această clasă se creează la începutul programului și se mențin instanțe către celelalte module. Obiectele din scenă sunt create tot de către acest modul, ele fiind salvate într-o listă, pentru eficientizarea procesului de ștergere din aceasta. (în cazul în care nu mai este nevoie de un obiect).

Din moment ce există referințe către celelalte sisteme, s-a creat o clasă folosind design pattern-ul Singleton cu rolul de a menține o referință către modulul de control. Având această referință, se poate accesa instanța unui sistem în mod static, și se pot adăuga noi obiecte pentru a fi întreținute din orice parte a codului. De asemenea, se menține o referință către cameră, pentru a fi accesată în mod facil în pasul de desenare.

O altă întrebuințare este de a trimite mai departe evenimentele de input primite de la GLFW către componentele fiecărui obiect din lista de obiecte.

4.1.2 Modulul de UI

API-ul de UI a fost înfășurat într-o clasă care are rolul de a inițializa și actualiza starea în care se află cele doua subsisteme care fac legătura între API și OpenGL, respectiv API și GLFW.

Pentru subsistemul care face legătura cu OpenGL, la inițializare, acesta necesită informații despre versiunea de OpenGL folosită, pentru a ști ce fel de setări sunt necesare și ce fel de apeluri de OpenGL sunt oferite în versiunea selectată. Când apare un nou cadru și se apelează funcția de NewFrame din cadrul subsistemului, se creează buffere pentru a ține vertexii transmiși de către ImGui și se leagă shader-ele care vor fi folosite, în funcție de versiunea de OpenGL selectată. La sfârșitul frame-ului, după ce este anunțat API-ul că este dorită randarea UI-ului, datele de desenare primite de la API sunt trimise către subsistem, care are rolul de a folosi buffer-ele și shader-ele pentru a le încărca și a le trimite către GPU.

Pentru subsistemul care face legătura cu GLFW, la inițializare, se trimite fereastra oferită de API-ul GLFW și se instalează callback-urile de mouse și de tastatură ca să se poată interacționa cu ferestrele oferite de ImGui. La un nou cadru, prin apelarea metodei NewFrame, se actualizează poziția cursorului și a gamepad-ului (dacă există) și se verifică dacă este vreun buton apăsat.

4.1.3 Modulul de fizică

API-ul de fizică (Coumans, 2015) a fost înfășurat și el într-o clasă pentru a avea acces mai facil la anumite metode oferite.

Principala problemă a motorului de fizică este că folosește o bibliotecă proprie de matematică, care are un alt format față de GLM. Prin urmare, au fost create metode care convertesc din GLM în formatul suportat de Bullet (Coumans, 2015).

Această clasă oferă metode pentru inițializarea sistemului de fizică (cu valori implcite), step în lumea de fizică, adăugare/ștergere de corpuri rigide și de raycast-ing.

Având în vedere sistemul de componente prezentat anterior, există o componentă numită RigidBodyComponent care are rolul de a adăuga automat un obiect în sistemul de fizică, folosind forma de box, cu poziția, orientarea și scalarea specifică obiectului. Toate aceste informații se abstractizează sub forma unei instanțe a clasei btRigidBody. Pe lângă aceste informații, se poate specifica și grupa, respectiv masca (ambele măști de biți), utilă pentru a separa obiectele create. Un caz concret de folosire este când se creează inamici și ziduri, și se dorește o funcționalitate diferită atunci când se face raycast, raza atingând fie inamicul, fie zidul.

La adăugarea unui rigid body în sistemul de fizică, se folosește un unordered map pentru a stoca referința către rigid body mapată la referința către obiect. Utilitatea unordered map este la apelul RayCastObject, când nu se dorește distanța până la rigid body-ul intersectat de rază, ci chiar obiectul din scenă. La ștergerea unui rigid body, se șterg toate informațiile despre acesta din lumea de fizică, după care se șterge intrarea din unordered map asociat cu acesta, astfel evitându-se memory leak-urile.

4.1.4 Modulul de desenare

Modulul de desenare este reprezentat printr-o clasă care are o referință la lista de obiecte întreținută de modulul de control și referințe către clase care moștenesc RenderPass, reprezentând o trecere de desenare a obiectelor, o metodă de desenare și o metoda de inițializare.

Inițializarea constă în schimbarea rezoluției a trecerilor de desenare cu valorile primite ca parametru. Rezoluțiile pot să se refere fie la rezoluția umbrelor în cazul trecerilor de desenare de adâncime, fie la rezoluția ferestrei pentru desenarea finală. Prin urmare, la un eveniment de schimbare a dimensiunilor ferestrei, trebuie reinițializat sistemul de desenare.

Scopul acestui modul este de a seta mediul specific trecerii de desenare curente (prin schimbarea parametrilor camerei de exemplu) și desenarea obiectelor pentru fiecare trecere. Pentru a desena obiectul, acesta oferă o metodă GetComponent, metodă care se folosește pentru a verifica ce tip de renderer folosește obiectul respectiv. Există doua tipuri implicite de astfel de componente: o componentă de randare pentru mesh (implementată deja în framework-ul SPG (UPB, 2018)) și una pentru modele cu animații. Ambele moștenesc componenta GraphicComponent, cu metoda render, care, în funcție de ce tip de renderer este, trimite parametrii necesari către shader, după care apelează metoda render din mesh/model, care știe să trimită informațiile de desenare către GPU.

4.2 Arhitectura sistemului de componente

Pentru a implementa sistemul de componente (figura 4.2), s-a pornit de la o componentă de bază (BaseComponent) și un obiect de bază (BaseGameObject). Scopul obiectului este de a actualiza componentele la fiecare nou cadru (obiectul este anunțat de modulul de control de acest lucru), iar rolul componentelor este de a avea logică de gameplay sau logică utilă sistemelor prezente în motor.

Obiectul de bază ține referințele către componentele sale într-o structură de map, pentru a avea acces rapid la acestea și pentru a păstra ordinea în care acestea au fost adăugate. El mai deține câmpuri pentru id, un câmp boolean care specifică dacă obiectul trebuie șters sau nu din lista de obiecte din modulul de control (acest câmp este verificat pentru fiecare obiect la fiecare cadru) și o referință la o clasă utilitară ComponentTransform, care deține informații precum poziția, orientarea, scalarea și matricea de model a obiectului. El are metode de Get și Set pentru toate câmpurile, și metode de AddComponent și GetComponent. Metoda de AddComponent presupune adăugarea unei componente în map, cheia acesteia fiind numele componentei (câmp care trebuie setat la crearea componentei prin metoda GetName) și setarea referinței obiectului stăpân al componentei către sine însuși. Metoda de GetComponent verifică existența numelui componentei în map, iar dacă acest nume există, întoarce componenta respectivă. Când obiectul trebuie eliberat, se va seta câmpul shouldDelete pe true (fie de către acesta, fie de către o componentă a sa sau o componentă a altui obiect), destructorul având grijă să elibereze fiecare componentă

din map-ul de componente. Cât timp componentele eliberează memoria alocată de ele în destructorul specific, nu vor exista memory leak-uri.

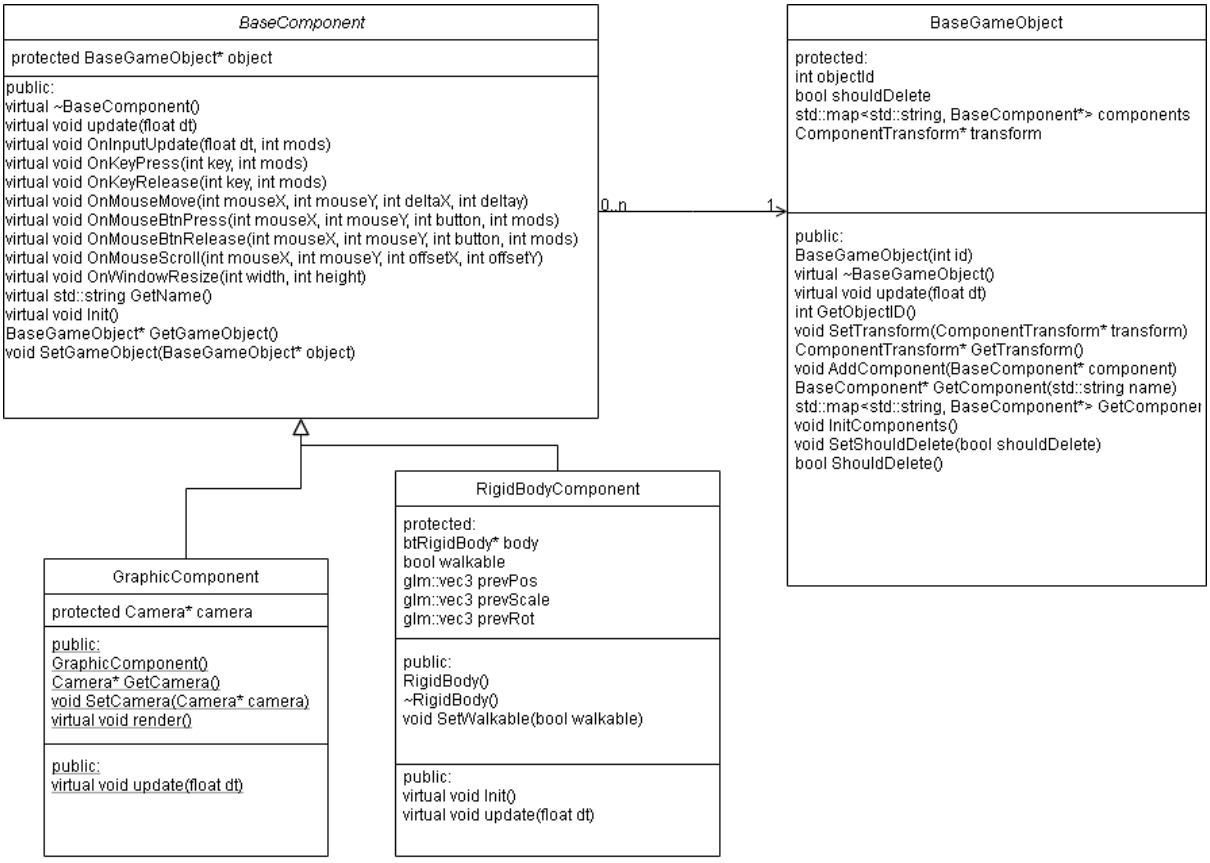

Figura 4.2: Diagramă de clase pentru arhitectura sistemului de componente

Componenta de bază deține o referință la obiectul care o actualizează, metode pentru tratarea evenimentelor de input, metodă de inițializare și doua metode care trebuie suprascrise de componentele moștenitoare: GetName și update. Metoda GetName trebuie suprascrisă cu numele componentei respective, pentru ca obiectul să poată revendica o componentă după nume. Metoda update trebuie suprascrisă cu logica componentei respective. Pentru a evita leak-urile de memorie, în destructor-ul specific fiecărei componente se va elibera toată memoria alocată.

Componente importante din acest sistem sunt componentele de render, care au rolul de a trimite informații către shader și de a desena mesh-a sau modelul obiectului. Există mai multe tipuri de astfel de componente, dar, ca implementare generală, acestea dețin, pe lângă câmpurile prezente în componenta de bază, și câmpuri pentru numele mesh-ei specific obiectului, numele shader-ului folosit pentru randare și o referință către camera folosită de motor.

Referința către cameră se poate lua de la modulul de control, prin clasa Singleton care menține referința către acesta.

Pentru a nu duplica instanțele de shader sau mesh pentru fiecare componentă de render în parte, mai ales dacă se folosește același shader sau mesh, s-a implementat un sistem de întreținere pentru shadere prin ShaderManager, iar pentru mesh-e sau modele și animația acestora prin MeshManager.

ShaderManager are rolul de a menține mapări între nume și instanțe de shader printr-un unordered map și de a fi accesat de oriunde. Prin urmare, clasele de Manager sunt implementări de Singleton. Metodele expuse sunt GetShader și AddShader. Metoda AddShader are ca parametrii numele shader-ului, numele căii unde se află shaderele, numele fișierului Vertex Shader, numele fișierului Fragment Shader și opțional numele Fișierului Geometry Shader (dacă este trimis ca parametru). Metoda verifică să nu existe vreun shader cu același nume în map, după care creează și leagă shader-ele și le adaugă în map, cheia fiind numele primit ca parametru al metodei. Metoda GetShader doar verifică existența numelui în map, dacă există este întoarsă instanța de shader.

MeshManager, la fel ca ShaderManager, implementează design pattern-ul Singleton și conține trei unordered maps, una pentru mesh-e, una pentru modele și una pentru animațiile modelelor. Metodele sunt AddMesh, AddInstancedMesh, AddModel, AddAnimation și Get pentru acestea.

Metodele de Get sunt asemănătoare cu metoda de GetShader specificată mai sus.

Metoda AddModel presupune specificarea numelui modelului, calea către fișier și numele fișierului. Folosind assimp (Assimp) se citește fișierul, se creează o instanță de Model și una de Bone (care va fi rădăcina scheletului), se încarcă modelul și scheletul, după care se realizează maparea.

AddAnimation are, pe lângă parametrii din AddModel, și numele modelulu. Logica este aproximativ aceeași, doar că se caută și modelul al cărui schelet va fi folosit pentru animație.

Metodele AddMesh și AddInstancedMesh sunt asemănătoare, ambele primind parametrii numele mesh-ei, locația acesteia, numele fișierului și diferiți parametrii de configurare. Metodele doar verifică existența mesh-ei în map, în caz că nu există, se încarcă informațiile despre fișierul de la calea specificată folosind biblioteca assimp (Assimp), după care este adăugată maparea nume și mesh.

Diferența dintre Mesh și InstancedMesh este faptul că InstancedMesh presupune crearea și randarea unui singur obiect pentru mai multe instanțe de mesh-e identice. Procesul implică folosirea funcției glDrawElementsInstancedBaseVertex din API-ul OpenGL pentru a desena toate instanțele obiectului într-un singur apel. Pentru a face acest lucru, este nevoie de un buffer pe GPU care să mențină matricea de Model specifică fiecărei instanțe care va fi desenată. Acest lucru impune stocarea matricelor de Model în clasa InstancedMesh într-un

vector și recalcularea buffer-ului de pe GPU cu informațiile din vector după adăugarea unei noi instanțe. (sau după adăugarea tuturor instanțelor)

4.3 Sistemul de animații scheletale

Animațiile scheletale (Luna, 2004) sunt reprezentate de o instanță a clasei Model (care deține informațiile de desenare și scheletul) și o instanță a clasei Animation (care deține informațiile legate de transformările oaselor modelului la diferite eșantioane de timp și metode de interpolare a acestor transformări în cazul în care timpul curent este între doua astfel de cadre). Se va începe explicația cu clasa Bone, deoarece atât Model cât și Animation au nevoie de structura scheletală.

Clasa Bone are în componență un nume, un ID, o matrice de transformare și un vector de referințe la alte instanțe ale clasei Bone, aceștia fiind ”copiii” în ierarhie. Numele este util deoarece animațiile, la fiecare keyframe, au o transformare specifică pentru numele unui os. Id-ul reprezintă index-ul osului în array-ul de transformare, și va fi folosit când se va calcula poziția finală a unui os într-un cadru.

Clasa Model se ocupă de încărcarea și menținerea scheletului și a informațiilor de desenare. Încărcările se realizează cu biblioteca assimp (Assimp), care va întoarce o scenă rezultată din citirea fișierului specificat, care cuprinde nodurile ierarhiei și un array de mesh-e. Pentru început, se creează o instanță a clasei Bone, care va fi primul nod din ierarhie. Mergând pe ierarhia din scena returnată anterior, putem construi scheletul, având acces în fiecare nod la informații precum numele osului, matricea acestuia de transformare, numărul și copiii acestuia. Totuși, deoarece assimp (Assimp) folosește alt standard matematic față de GLM, sunt definite proceduri de conversie. După ce ierarhia este construită, se vor citi informații despre vertex-i la nivelul fiecărei mesh-e, care se vor stoca în vectori. De asemenea, fiecare astfel de mesh-ă conține un array de oase, cu informații precum ID-ul acestora, numele și o matrice de transformare de offset, date care vor fi stocate într-un unordered map de oase, indexate după numele osului. Această matrice de offset are rolul de a transforma osul respectiv din spațiul local în spațiul relativ la părintele său, astfel încât oasele să fie conectate. De asemenea, pentru fiecare os, există un array cu vertex-ii și ponderea cu care acesta îi influențează. Aceste date se vor stoca într-un array de structuri, unde index-ul este reprezentat de ID-ul osului, structura având un câmp cu ID-ul vertexului și un câmp cu ponderea cu care acesta este influențat. S-a decis folosirea unei astfel de colecții deoarece aceste date trebuie trimise direct către GPU. După ce se face citirea tuturor mesh-elor din care este compus modelul, se vor citi texturile, tot pornind de la scena întoarsă anterior. Texturile vor fi citite într-un vector de Texture2D, folosind TextureManager (care au fost deja implementate în framework-ul de SPG (UPB, 2018)). După ce inițializarea este terminată, se creează buffere pe GPU pentru a putea trimite datele specifice vertex-ilor și ID-ul oaselor care le influențează împreună cu ponderea.

Clasa Animation face citirea în același mod ca Model, scena având un array de noduri de animații care va fi parcurs pentru a putea fi memorat. Fiecare nod este specific unui nume de os, și, de asemenea, fiecare nod are un număr de transformări de translație, rotație, scalare care trebuie memorate. S-au creat vectori de structuri pentru a memora transformările specifice keyframe-urilor, structuri în care sunt prezente transformarea specifică de scalare/translație/rotație și un câmp pentru timpul la care se întâmplă acea transformare.

La runtime, pentru a reda animația, este trimis către Model (la apelul funcției render) referința către animația care trebuie afișată și timpul la care se află aceasta (pe lângă alți parametri care trebuie transmiși către shader), se vizitează ierarhia de oase pornind de la osul rădăcină și se transmite recursiv către copiii nodului animația, timpul și transformarea părintelui. Transformarea aceasta presupune înmulțirea matricei de offset al osului curent cu matricea de Model compusă din scalarea, rotația și translatarea provenită din interpolarea transformărilor a două keyframe-uri alăturate (cele între care se găsește valoarea timpului curent din animație) și matricea transformării părintelui (în mod recursiv, dacă nu este rădăcină). Aceste transformări se memorează într-un vector de transformări, index-ul în vector fiind ID-ul osului. După calculul acestui vector, el este transmis către shader ca uniform. În shader (fragment de cod 4.1), se vor calcula pozițiile finale ale vertex-ilor, compunând transformările oaselor ponderate, (maxim 4 oase pot influența poziția unui vertex) transformarea compusă înmulțindu-se cu poziția locală a vertex-ilor.

```
mat4 BoneTransform = Bones[boneIDs[0]] * weights[0];
BoneTransform += Bones[boneIDs[1]] * weights[1];
BoneTransform += Bones[boneIDs[2]] * weights[2];
BoneTransform += Bones[boneIDs[3]] * weights[3];

vec4 PosL = BoneTransform * vec4(v_position, 1.0);
gl_Position = Projection * View * Model * PosL;
texture_coord = v_texture_coord;
```

Fragment de cod 4.1: Calcularea pozițiilor finale ale vertex-ilor în vertex shader

Pentru a face blend la schimbarea animațiilor, s-a creat o clasă AnimationInfo, care conține câmpuri cu referințe către clasele Model și Animation folosite de obiect și un câmp pentru timestamp. Când se va schimba animația obiectului (prin componenta de render), se va apela metoda SetAnimation din clasa AnimationInfo care are ca scop oprirea blend-ului animației trecute, setarea parametrului boolean isBlending al animației curente, setarea timestamp-ului de blend ca fiind timestamp-ul curent și setarea unui câmp care conține o referință către animația trecută (cea de la care se face blend). Fiecare animație având un timp de blend-ing, se va verifica la apelurile funcțiilor de interpolare (care se folosesc la calcularea transformărilor finale ale oaselor) dacă parametrul isBlending este adevărat și dacă diferența de timp între timestamp-ul curent și timestamp-ul de blend este mai mic decât timpul de blend-ing. Dacă acestea sunt adevărate, atunci funcțiile de interpolare vor

întoarce o interpolare liniară între transformările animației trecute la eșantionul de timp la care s-a făcut schimbarea de animații și transformările animației curente la eșantionul de timp egal cu timpul de blend-ing, factorul folosit la interpolare fiind fracțiunea din timpul de blend al diferenței de timp între timestamp-ul curent și timestamp-ul la care a început blending-ul. La fiecare cadru se verifică de către renderer dacă trebuie setat parametrul isBlending al animației curente ca fiind fals.

4.4 Sistemul de iluminare

Sistemul de iluminare este reprezentat de trei treceri de desenare, primele două având ca scop stocarea umbrelor obiectelor în texturi, iar a treia trecere având ca scop desenarea finală a obiectelor din scenă, influențate de lumini și umbre.

4.4.1 Trecerea de desenare a umbrelor direcționale

Această trecere este încapsulată într-o clasă numită ShadowMapPass, care are metode pentru inițializare, BindForWriting și BindForReading.

Metoda de inițializare stochează informațiile despre rezoluție, creează un FBO și o textură (alături de parametrii acesteia) care va fi folosită în trecerea finală. De asemeea, pentru FBO-ul creat, se dezactivează informația de culoare, fiind necesară doar informația de adâncime.

Metoda BindForWriting are ca scop setarea viewport-ului OpenGL la rezoluția trimisă în pasul de inițializare, golirea depth buffer-ului și setarea face culling-ului, pentru a evita fenomenul de peter panning. Metoda BindForReading primește ca parametru o unitate de texturare și va lega textura creată anterior la această unitate.

Sistemul de desenare se ocupă de apelarea acestor metode și de setarea parametrilor camerei, iar componenta de randare se ocupă de trimiterea informațiilor către GPU.

Din moment ce scopul acestei treceri este doar de a face o textură cu valorile depth buffer-ului, shader-ele folosite sunt simple. Vertex shader-ul doar va calcula poziția vertex-ilor în spațiul de coordonate al luminii, iar fragment shader-ul nu va face nimic. (depth buffer-ul va fi scris oricum)

După acest pas, va exista prima textură necesară desenării umbrelor, urmând trecerea de desenare pentru umbrele luminilor punctiforme (Vries, Point-Shadows, 2020).

4.4.2 Trecerea de desenare a umbrelor omnidirecționale

Clasa care se ocupă de această trecere este CubeShadowMapPass, cu metode identice clasei precedente, diferența principală fiind textura, care de data aceasta este un cubemap, necesitând parametrii de setare diferiți. De asemenea, trebuie trimise către shader poziția luminii, far plane-ul acesteia și 6 matrice de transformare din spațiul lume în spațiul de proiecție, pentru proiecția perspectivă cu FoV de 90° compusă cu matricele de vizualizare pentru fiecare din cele 6 direcții posibile. De toate acestea se va ocupa componenta de randare.

Vertex shader-ul folosit este unul simplu, care doar înmulțește poziția fiecărui vertex cu matricea de Model, calculele realizându-se în world space.

```
for(int i = 0; i < 6; i++)
{
    gl_Layer = i;
    for(int j = 0; j < 3; j++)
    {
        frag_pos = gl_in[j].gl_Position;
        gl_Position = view[i] * frag_pos;
        EmitVertex();
    }
    EndPrimitive();
}
```

Fragment de cod 4.2: Generare a 6 fragmente pentru fiecare față a cubemap-ului în geometry shader

Pentru a desena depth buffer-ul pe toate cele 6 fețe, se folosește un Geometry Shader (fragment de cod 4.2), care transformă fiecare primitvă în 6 primitive, schimbând gl_Layer-ul (fața din cubemap pe care se desenează) pentru a putea desena fiecare primitivă din perspectiva a celor 6 transformări.

```
gl_FragDepth = length(frag_pos.xyz - light_pos) / far;
```

Fragment de cod 4.3: Calculul adâncimii liniare a fragmentului

Fragment shader-ul (fragment de cod 4.3) folosit suprascrie gl_FragDepth, adâncimea fragmentului devenind distanța dintre acesta și poziția luminii, împărțite la far plane-ul camerei. Se folosește adâncimea liniară pentru a ușura calculele din trecerea finală de desenare.

4.4.3 Trecerea de desenare a luminii

Clasa care încapsulează această trecere este similară cu cele despre care s-a vorbit anterior, diferența fiind că nu mai există o metodă BindForReading (deoarece nu mai există o textură), iar în metoda BindForWriting se activează back culling-ul. (nemaifiind nevoie de face culling pentru peter panning)

Înainte de a desena obiectele, se vor lega la shader texturile pentru umbre apelând BindForWriting pe cele 2 render pass-uri anterioare și se vor mai trimite matricea de vizualizare și de proiecție a luminii direcționale, cât și poziția observatorului, poziția camerei folosită la generarea umbrelor omnidirecționale, far plane-ul acesteia și culoarea difuză a obiectului. (s-au considerat culorile diferitelor componente ale luminii din modelul Phong ca fiind egale cu culoarea difuză a obiectului, respectiv culoarea texturii)

```
world_position = (Model * vec4(v_position, 1.0)).xyz;
world_normal = mat3(transpose(inverse(Model))) * interior * v_normal;
tex_coords = v_texture_coord;

gl_Position = Projection * View * Model * vec4(v_position, 1.0);
light_space_position = Light_Proj * Light_View * Model * vec4(v_position, 1);
```

Fragment de cod 4.4: Calculul normalei, poziției în lume și a poziției în spațiul luminii

În vertex shader-ul acestei treceri (fragment de cod 4.4) se calculează normala și poziția în spațiul lumii a vertex-ilor, poziția obținută prin transformarea poziției locale cu matricea MVP și poziția în spațiul luminii direcționale (folosind matricea de vizualizare și proiecție a camerei ortografice care simulează lumina respectivă).

```
vec3 pos = (light_space_position.xyz / light_space_position.w) * 0.5 + 0.5;
pos.z = clamp(pos.z, 0, 1);
float bias = 0.0005;
pos.z -= bias;
float shadow = 0.0;
vec2 texelSize = 1.0 / textureSize(shadow_map, 0);
for (int i = -1; i <= 1; i++) {
	for (int j = -1; j <= 1; j++) {
		vec3 newPos = vec3(pos.xy + vec2(i, j) * texelSize, pos.z);
		shadow += texture(shadow_map, newPos, 0);
	}
}

return shadow / 9.0;
```

Fragment de cod 4.5: Calculul umbrei provenită de la lumina direcțională

În fragment shader, se calculează umbra direcțională (Vries, Shadow-Mapping, 2020) (fragment de cod 4.5), aducând adâncimea fragmentului în domeniul $[0, 1]$. După aceasta, se scade termenul de bias (pentru a evita shadow acne), și se face PCF, făcând media

valorilor de umbră al fragmentelor învecinate. De asemenea, pe lângă PCF-ul software, se folosește sampler2DShadow pentru textura de umbră, pentru a avea un PCF gratuit hardware.

```
vec3 dir = world_position - light_pos;
float bias = 0.05;
return length(dir) - bias > texture(shadow_cube, dir).r * far ? 0.0 : 1.0;
```

Fragment de cod 4.6: Calculul umbrei provenită de la lumina punctiformă

Umbra omnidirecțională (Vries, Point-Shadows, 2020) (fragment de cod 4.6) este mai ușor de calculat (deoarece s-a considerat adâncimea liniară), trebuie comparată distanța de la poziția în lume a vertex-ului la poziția luminii punctiforme cu valoarea de adâncime din textura de cubemap eșantionată de direcția de la locația luminii la poziția vertex-ului. De asemenea, se folosește și aici un termen de bias.

După ce s-au calculat umbrele, ele sunt folosite în calculul luminii (Phong, 1975) (conform capitolului 3.6), făcând un blend aditiv între lumina venită de la sursa direcțională cu ceavenită de la sursa punctiformă, cea din urmă fiind atenuată cu factorul de atenuare.

5 EVALUAREA REZULTATELOR

Metoda de evaluare a rezultatelor este de a observa din punct de vedere grafic și din punct de vedere al cadrelor pe secundă (fps) ce se întâmplă în aplicația dezvoltată folosind diferite configurații de scenă și de parametri, neexistând o altă implementare cu care se poate compara. De fiecare dată, rezoluția de desenare este 1920x1080, placa video fiind un Intel HD Graphics 4400, numărul de obiecte din scenă fiind aproximativ 1000, fiecare obiect având în jurul a 1000 vertex-i, textura umbrei direcționale fiind 2048x2048, iar cea a umbrei punctiforme fiind 512x512(x6 fețe). Astfel, urmează o serie de figuri, fiecare explicând configurația și numărul de cadre pe secundă atins:

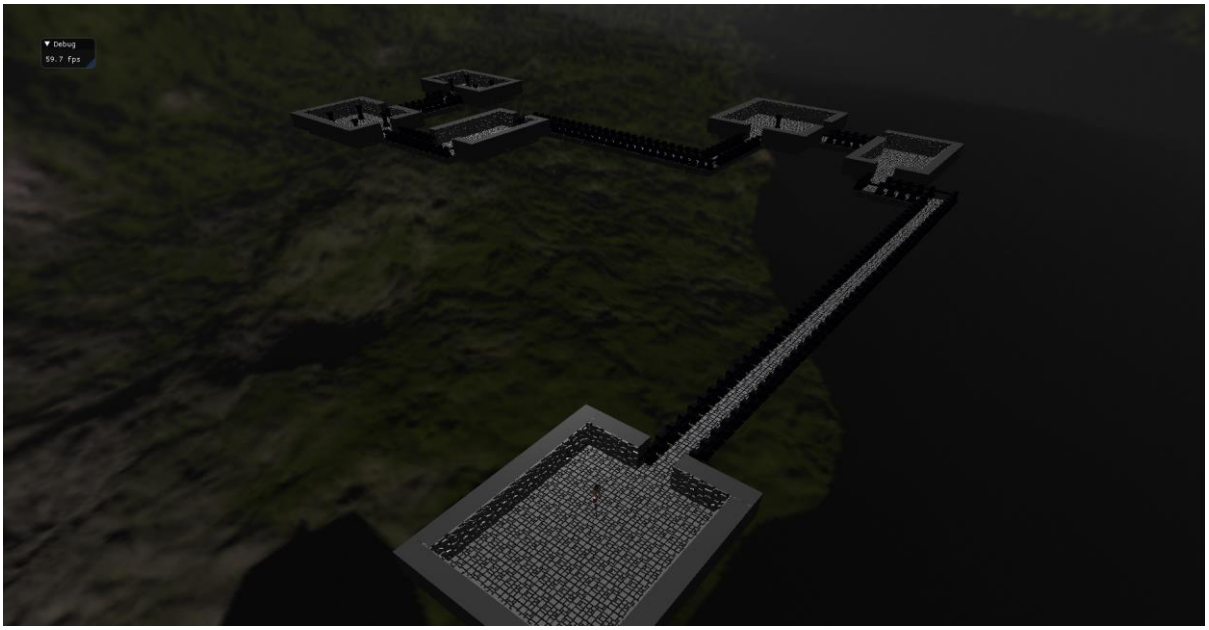

Figura 5.1: 60 fps, , valoare posibilă datorită instanțierii obiectelor pentru a nu fi nevoie de foarte multe apeluri de desenare; sunt desenate obiectele fără umbre, culoarea obiectelor este dată de lumina direcțională

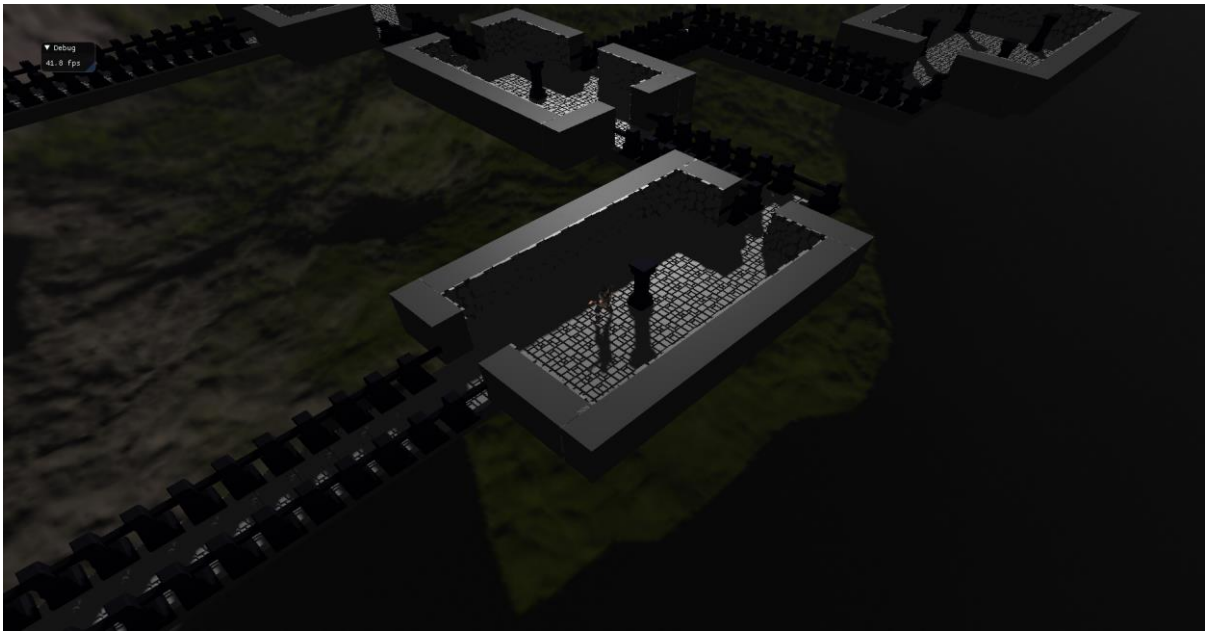

Figura 5.2: 42 fps, este activă doar lumina direcțională și umbrele create de aceasta

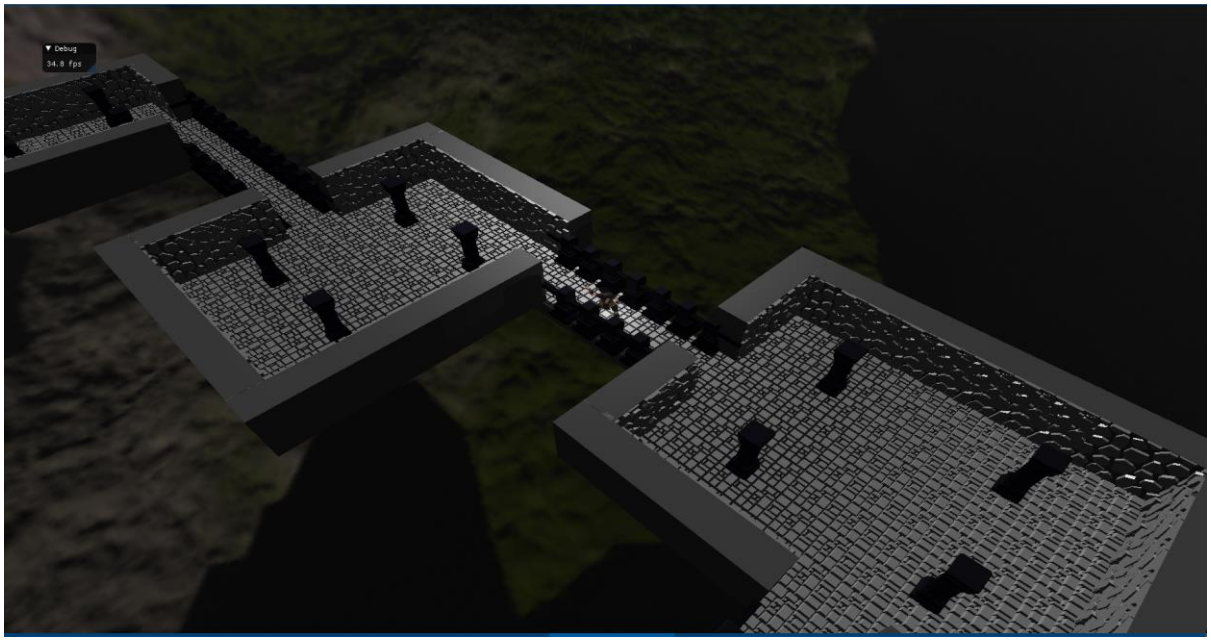

Figura 5.3: 35 fps, sunt active ambele tipuri de lumini dar se desenează doar umbra omnidirecțională (poziția caracterului fiind poziția luminii punctiforme), aceasta având un overhead mai mare de performanță față de umbra produsă de lumina direcțională

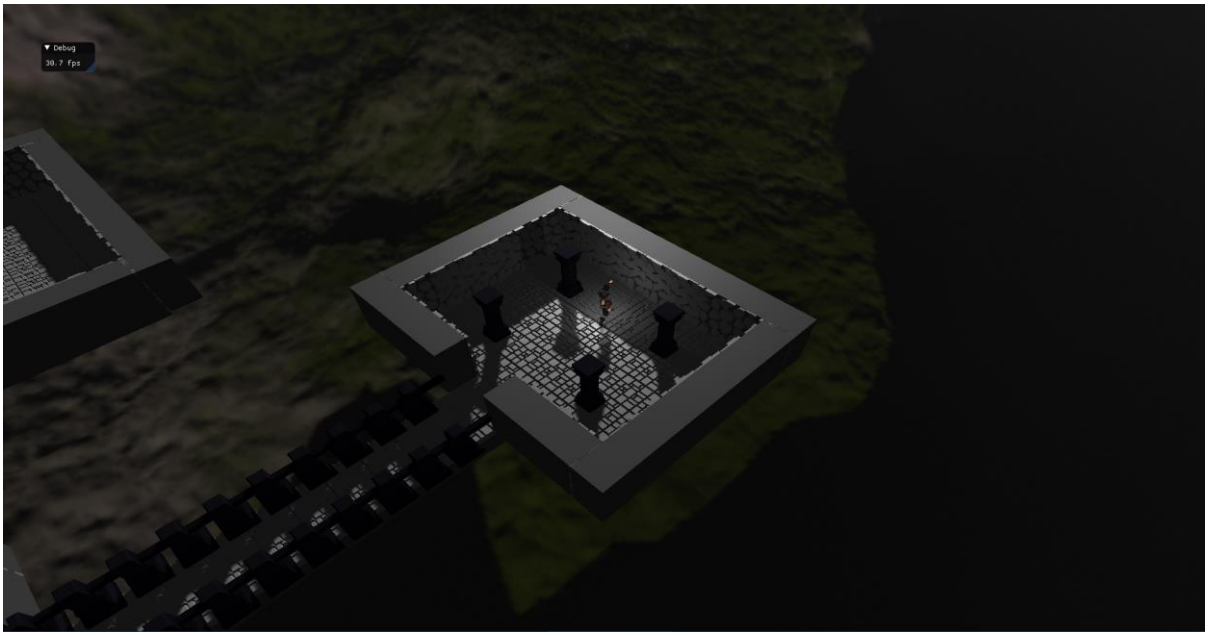

Figura 5.4: 30 fps în aplicație, sunt calculate ambele tipuri de lumini și umbrele provenite de la acestea

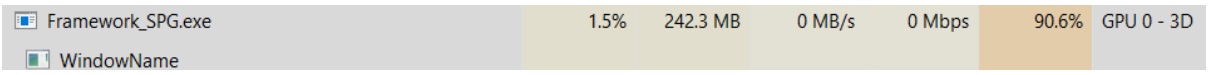

Figura 5.5: Aplicația este GPU-bound (90%), procesorul fiind folosit în proporție de 1.5%, memoria folosită fiind de 242MB

6 CONCLUZII

Proiectul a avut drept obiectiv crearea unui motor de joc pentru mărirea productivității dezvoltării acestora, făcând procesul de dezvoltare fie mai rapid, fie mai ușor, prin sisteme de întreținere, sisteme utile în mod general dezvoltării unui joc (cum ar fi sistemul de fizică) și unelte de debugging. De asemenea, s-a urmărit ca jocurile dezvoltate folosind aplicația să arate bine și să aibă performanță bună.

După realizarea analizei altor motoare de joc, s-a ales arhitectura și componentele necesare pentru atingerea scopurilor proiectului, după care s-a trecut la integrarea/implementarea acestora în aplicație.

Au existat câteva probleme întâmpinate de-a lungul dezvoltării, care au fost rezolvate între timp. Prima problemă întâmpinată a fost la integrarea motorului de fizică, deoarece au fost necesare înțelegerea și schimbarea anumitor parametri de compilare din cadrul proiectului pentru ca acesta să compileze. O altă problemă întâmpinată a fost la generarea hărții din aplicația dezvoltată, existând foarte multe obiecte separate pentru pereți, podea, ornamente etc. Acest fapt a dus la foarte multe apeluri de desenare care scădeau performanța aplicației drastic. Eventual s-a rezolvat această problemă prin crearea unui sistem care gestionează instanțele obiectelor asemănătoare, trimițând toate instanțele asemănătoare într-un singur apel către GPU.

Proiectul a fost interesant prin simplul fapt că unește foarte multe domenii software, fiind o aplicație destul de complexă de dezvoltat și de întreținut. Astfel, se pot învăța multe lucruri prin crearea unei asemenea aplicații.

6.1 Dezvoltări ulterioare

Motorul de joc este încă în stagiile de început, existând foarte multe opțiuni de dezvoltare din punctul curent.

Un neajuns al aplicației în stagiul curent este lipsa de interfețe de dezvoltare a jocului la runtime, prin care programatorul ar putea adăuga automat componente anumitor obiecte, ar putea observa starea anumitor module, de a seta anumiți parametri pentru anumite module sau obiecte etc., fără a mai fi nevoie de cod pentru a face acest lucru.

Pe lângă neajunsul aplicației menționat anterior, pe viitor se va căuta optimizarea anumitor module din motor, necesitând a se face o evaluare a performanțelor individuale ale acestora, pentru a face o prioritizare.

Pentru o fidelitate grafică mai bună, se poate implementa un sistem pentru particule, suport pentru PBR (physically-based rendering), sau trecerea într-un alt mod de rendering, mai specific deferred rendering, pentru a putea suporta multe surse de lumini pe ecran. Toate acestea ar trebui să existe la un moment dat, deoarece acest motor de joc ar trebui să aibă un suport general pentru multe tipuri de jocuri.

De asemenea, aplicația în mod curent nu poate rula folosind debugger-ul incorporat în Visual Studio, fiind necesare bibliotecile Bullet pentru debug (pe lângă schimbarea unui număr de parametri ca acestea să funcționeze).

Un alt sistem util pentru un joc ar fi un sistem audio, care va reda fișiere audio atunci când se întâmplă anumite acțiuni pe parcursul jocului, sau pentru a exista un sunet de fundal. Se poate găsi o bibliotecă care să realizeze acest lucru.

Un modul interesant care ar putea exista ar fi unul de multiplayer, momentan singurul mod de joc suportat este singleplayer. Acesta ar putea fi unul simplu pentru început, în care unul dintre jucători poate ține server-ul (cumva în paralel cu jocul), ceilalți jucători conectându-se la acesta și trimițând date specifice client-ului, pe care server-ul le va colecta și le va difuza înapoi jucătorilor pentru actualizarea stării jocului pe toate instanțele.

Totuși, deși motorul de joc este încă la început și în ciuda neajunsurilor din stagiul curent, consider că aplicația și-a atins scopul, instrumentația prezentă în mod curent facilitează dezvoltarea jocurilor, iar aplicațiile dezvoltate cu ajutorul motorului au o fidelitate grafică și o performanță bună.

7 BIBLIOGRAFIE

Assimp. (fără an). Preluat pe 6 22, 2020, de pe https://github.com/assimp/assimp

Coumans, E. (2015). *Bullet Physics.* Preluat pe Iunie 22, 2020, de pe https://bulletphysics.org: https://github.com/bulletphysics/bullet3/blob/master/docs/Bullet_User_Manual.pdf

Dear ImGui. (fără an). Preluat pe Iunie 22, 2020, de pe https://github.com/ocornut/imgui

Luna, F. (2004, February 20). *Skinned Mesh Character Animation with Direct3D 9.0c.* Preluat pe Iunie 22, 2020, de pe https://www.gamedevs.org/uploads/skinned-mesh-and-character-animation-with-directx9.pdf

Phong, B. T. (1975). *Illumination for Computer Generated Pictures.* Preluat pe Iunie 22, 2020, de pe https://users.cs.northwestern.edu/~ago820/cs395/Papers/Phong_1975.pdf

UPB. (2018). *SPG-Framework.* Preluat pe Iunie 22, 2020, de pe https://github.com/UPB-Graphics/SPG-Framework

Vries, J. d. (2020, Iunie). *Point-Shadows.* Preluat pe Iunie 22, 2020, de pe https://learnopengl.com/: https://learnopengl.com/Advanced-Lighting/Shadows/Point-Shadows

Vries, J. d. (2020, June). *Shadow-Mapping.* Preluat pe June 22, 2020, de pe https://learnopengl.com/: https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping