MINISTERUL EDUCAȚIEI

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

FRAMEWORK PENTRU CREAREA JOCURILOR 3D, DEZVOLTAT ÎN VULKAN API

LUCRARE DE LICENȚĂ

Absolvent: **George OPRUȚA**

Coordonator științific: **Șl. dr. ing. Constantin NANDRA**

2025

Cuprins

Capitolul 1 Introducere **1**

Capitolul 2 Obiectivele proiectului **3**

Capitolul 3 Studiu bibliografic **5**
3.1 Arhitectura motoarelor de jocuri . . . 6
3.1.1 **Modul de joc** . . . 6
3.1.2 **Dinamica jocului în timp real** . . . 6
3.1.3 **Gestionarea resurselor** . . . 6
3.1.4 **Motorul de randare 2D sau 3D** . . . 7
3.2 Framework-uri existente . . . 7

Capitolul 4 Analiză și fundamentare teoretică **8**
4.1 Cerințe funcționale . . . 8
4.1.1 Inițializarea și configurarea sistemului Vulkan . . . 8
4.1.2 Încarcarea obiectelor 3D din fișiere .obj . . . 8
4.1.3 Implementarea unei camere virtuale . . . 9
4.1.4 Aplicarea texturilor și iluminarea scenei . . . 9
4.1.5 Integrarea unui algoritm de mișcare pentru NPC-uri . . . 10
4.1.6 Crearea unui sistem de gestionare a nivelurilor . . . 10
4.2 Cerințe non-funcționale . . . 11
4.3 Cazuri de utilizare . . . 12
4.4 Soluția problemei și metodologiile folosite . . . 13
4.4.1 **Modulul de randare** . . . 13
4.4.2 **Game Loop - bucla principala a jocului** . . . 15
4.4.3 **Modulul de control al intrărilor** . . . 16
4.4.4 **Caracteristicile modului de joc** . . . 16
4.4.5 **Sistemul de propagare a sunetului** . . . 17
4.4.6 **Modelul Obiectelor** . . . 17

Capitolul 5 Proiectare de detaliu și implementare **19**
5.1 Arhitectura aplicației . . . 19
5.2 Sistemul de randare - Render System . . . 20
5.2.1 **coreV** . . . 20
5.2.2 **Shadere** . . . 30
5.3 Sistemul entităților - Entity System . . . 34
5.4 Construcția nivelului - Level Design . . . 36
5.5 Sistemul de vizualizare - Visualization System . . . 37
5.6 Sistemul de Control - Control System . . . 39
5.7 Sistemul de propagare a suntetului - Propagation System . . . 42
5.8 Bucla Jocului - Game Loop . . . 44
5.9 Rezultat - Output . . . 49

Capitolul 6 Testare și validare **50**
6.1 Testarea componentelor principale 50
6.2 Metrici de performanță 53
6.3 Specificațiile sistemului de testare 55

Capitolul 7 Manual de instalare și utilizare **56**
7.1 Pașii pentru instalarea aplicației 56
7.1.1 Instalarea dependințelor 56
7.1.2 Instalarea motorului de jocuri 57
7.1.3 Manual de utilizare 58

Capitolul 8 Concluzii **59**

Bibliografie **61**

Anexa A Codul de început **62**

Capitolul 1. Introducere

Odată cu avansul tehnologic, reprezentarea vizuală a informației folosită de sistemele de calcul în prelucrarea grafică 2D și 3D, împreună cu interacțiunea în timp real cu acestea, a devenit o sarcină mult prea complexă și solicitantă pentru procesoare obișnuite. Astfel, pentru a prelua o bună parte din volumul de muncă ce trebuia depus, s-a introdus conceptul de placă video. Rolul plăcii video este de a afișa pe ecran și de a reprezenta vizual informația care se află în memorie și procesor în timp real, precum și facilitarea interacțiunii programatorului cu aceasta într-un mod mult mai intuitiv, care să pună în lumină rezultatele obținute.

Printre primele astfel de echipamente hardware, care au primit numele de GPU (Graphics Processing Unit – Unități de Procesare Grafică), prin intermediul NVIDIA, a fost 3Dfx Voodoo 1, apărută în anul 1996. Doar în anul 1999, GeForce 256 își făcea apariția pe piața largă. La acea vreme, existau limitări în ceea ce privește posibilitatea de a fi programabile în scopul obținerii unei performanțe mai ridicate sau a modului în care informația se afișa pe ecran, existând doar posibilitatea de a configura câteva mici aspecte legate de performanță și modul în care rezultatele ajungeau pe ecran. Doar după câțiva ani s-a trecut spre oferirea flexibilității și libertății programatorului de a schimba modul în care datele erau manipulate de către unitatea grafică a calculatorului, deschizând astfel noi orizonturi în ceea ce privea cercetarea interacțiunii dintre calculator și om [1, pp. 1–29].

Una dintre ramuri care s-a format în urma acestor cercetări a fost cea a jocurilor video. Aici, programatorul are posibilitatea de a crea programe care să ofere divertisment utilizatorilor prin modul în care grafica își schimbă interpretarea pe ecran în funcție de intrările primite. Astfel, fiecare programator care dorea să creeze jocuri era nevoit să cunoască în profunzime modalitățile prin care sistemul de operare interacționa cu unitatea grafică și să construiască de la zero modul în care viziunea sa era interpretată.

Pe măsură ce tehnologia grafică avansa, codul pe care programatorul trebuia să îl scrie pentru realizarea jocului devenea din ce în ce mai complex. Totodată, fiecare idee pe care acesta dorea să o materializeze însemna scrierea, de cele mai multe ori, a aceluiași cod care să îndeplinească funcțiile de bază la nivel grafic, ceea ce în cele din urmă devia atenția de la ideea jocului propriu-zis. Datorită cererii în creștere a jocurilor pe piață și a presiunii timpului, s-a dorit automatizarea procesului de realizare a jocurilor prin introducerea a ceea ce azi se numesc game engine-uri (motoare de jocuri).

Motoarele de jocuri au fost concepute ca un intermediar între programator, sistemul de operare și placa video. Rolul principal pe care îl ocupă este de a oferi programatorului uneltele necesare pentru crearea jocurilor într-un mod eficient și cât mai ușor posibil, fără a pune dezvoltatorul în dificultatea de a interacționa în mod direct cu apelurile de sistem către placa video sau de a gestiona memoria, acesta concentrându-se asupra aspectelor legate de logică și aspectul produsului final. Cu alte cuvinte, motorul de jocuri oferă refolosirea, la scară largă, a uneltelor puse la dispoziție și împachetate într-un singur program, pentru dezvoltarea a nenumărate idei de jocuri.

Pe fondul acestei creșteri semnificative a pieței jocurilor video și a diferitelor genuri în care acestea se încadrează (FPS – first person shooter, joc de acțiune din perspectiva

jucătorului, puzzle, strategie, simulator, platformă, third-person – a treia persoană, lupte, curse, joc pe roluri) [2, pp. 13–27], anumiți dezvoltatori s-au concentrat pe realizarea motoarelor de jocuri care să ofere o multitudine de unelte, optimizate și ușor de folosit, cu scopul de a livra o varietate extinsă de genuri de jocuri provenite dintr-un singur program. Astfel de motoare mai cunoscute, care se găsesc în momentul actual pe piață și care sunt oferite gratuit publicului țintă, se enumeră: Unity [3], Unreal Engine [4], CryEngine [5].

Acestea vin în sprijinul programatorilor și oferă: o interfață grafică menită să afișeze în timp real stadiul în care se află jocul, o modalitate de a rula și testa mersul jocului, adăugarea printr-un singur click a mai multor resurse în joc, adăugarea de logică precum mișcarea obiectelor, interacțiunea între acestea, obiectivele jucătorului, interfața grafică a jocului în sine, limite ale spațiului destinat jucătorului, cu ajutorul fișierelor de tip script care pot fi programate de către utilizatori în IDE-uri, fie integrate în motorul de jocuri propriu-zis, fie externe (Integrated Development Environment – Mediu de Dezvoltare Integrat). Unele motoare de jocuri mai moderne oferă posibilitatea de visual scripting (scriptare vizuală), care abstractizează în totalitate și automatizează necesitatea de a mai scrie codul logicii, dezvoltatorul fiind nevoit doar să traseze modul prin care blocurile de bază (exemplu: obiectele jocului) interacționează [6]. De reținut este și faptul că, deși acestea sunt oferite gratuit, companiile care dețin drepturile de autor ale motoarelor precizate anterior percep o taxă dezvoltatorului pentru expunerea pe piață a jocului creat.

Însă această abordare generală aduce un compromis în zona optimizărilor care puteau fi aduse jocurilor, deoarece motorul oferă unelte gata implementate de către dezvoltatori într-un scop general și nu erau axate pe un anumit gen de joc, rezultând într-o dezvoltare mai îngreunată și câteodată chiar greu de optimizat. Din acest motiv, au apărut motoarele de jocuri special concepute pentru a aduce optimizările necesare, împreună cu îmbunătățirea procesului de implementare, adaptate genului specific al jocului și care nu au fost expuse publicului larg, ele rămânând a fi programe dezvoltate intern. Un astfel de exemplu de motor de jocuri este Frostbite Engine. Acest motor este deținut de către EA (Electronic Arts) și dezvoltat de către DICE, fiind folosit pentru randarea avansată de grafică foto-realistă, fizică dinamică, inteligență artificială pentru caracterele necontrolate de jucător și animație cinematică.

Printre primele folosiri notabile ale motorului Frostbite, care au avut un impact semnificativ pe piață, s-a remarcat dezvoltarea jocului FIFA 17, apărut în anul 2016, aducând o nouă perspectivă și îmbunătățire vizuală față de precedentele iterații ale acestei francize de jocuri din cadrul genului simulator de jocuri sportive [7, 8]. Astfel, chiar și în prezent, anumiți dezvoltatori investesc timp în crearea unui motor de jocuri intern, care să dețină strict uneltele necesare pentru dezvoltarea optimă a jocului.

Această lucrare dorește să prezinte o parte din ingineria software din spatele creării unui motor de jocuri, care să permită implementarea jocurilor din genul stealth prin intermediul unei aplicații software, pe care alți programatori să o poată folosi în acest scop. Jocul stealth reprezintă un subgen al jocurilor FPS, în care scopul principal este mișcarea personajului controlat de jucător spre obiectivul nivelului, fără a fi observat sau capturat de către caracterele necontrolate.

Capitolul 2. Obiectivele proiectului

Scopul prezentei lucrări este de a evidenția etapele implicate în proiectarea de la zero a unui framework(software reutilizabil pentru dezvoltarea de aplicații) pentru crearea jocurilor 3D gândit sa folosească tehnologiile dezvoltate de Kronos Group [9] prin intermediul API -ului Vulkan [10], astfel încât rezultatul acestei lucrări să ofere uneltele de bază necesare utilizatorilor din acest domeniu, care își propun să realizeze jocuri video care se încadrează în genul stealth. Principalul aspect pe care acest proiect îți dorește să îl detalieze este dezvoltarea unei aplicați software care să dispună de funcționalitățile esențiale pentru: afișarea grafici pe ecran, crearea scenelor și popularea acestora cu obiecte 3D, atribuirea de logică funcțională obiectelor respective, ulterior oferind posibilitatea scalabilități.

Pentru a îndeplini acest scop, lucrarea a fost concepută pe baza unui set de obiective clare care să ofere sprijin în realizarea aplicației software. Obiectivele sunt următoarele:

1. Proiectarea unui sistem de bază care să organizeze componentele unui motor de jocuri de tip stealth.

   Primul pas în dezvoltarea unui framework de jocuri de tip stealth este de a realiza arhitectura de bază care să integreze conceptele cheie care definesc un astfel de motor. Astfel, acesta trebuie sa cuprindă în primul rand un mod prin care se realizează dinamica jocului și anume bucla principala care servește ca un mecanism de coordonare intre acțiunile jucătorului și actualizarea corespunzătoare a logici.

   De asemenea, aceasta trebuie să asigure definirea unor entități care, în funcție de logica implementată, să își actualizeze starea. Totodată, framework-ul trebuie să includă caracteristicile de baza ale jocurilor de tip stealth precum: propagarea sunetului, detecția sursei acestuia, realizarea unor acțiuni în urma declanșării unor astfel de stimuli și căutarea unei căi spre sursa respectivă. Aceste caracteristici susțin modelarea ideei centrale a genului stealth: evitarea detectării prin simularea indirectă a consecințelor acțiunilor jucătorului.

2. Integrarea unui sistem de randare care face posibila afișarea conținutului 3D în mod eficient și flexibil.

   Al doilea pas presupune integrarea unui API grafic care să permită folosirea resurselor plăci grafice dedicate pentru afișarea pe ecran a obiectelor 3D prin intermediul unui pipeline grafic. Pentru eficientizarea acestui proces este necesar și un sistem capabil să încarce fișiere de tip `.obj`, care conțin descrierea geometrica a obiectelor în spațiu, pentru a putea reda structura acestora în scenă. Totodată, un alt aspect important îl reprezintă gestionarea texturilor, astfel încât obiectele sa poată fi reprezentate cu detaliile vizuale adecvate, precum și integrarea surselor de lumină, contribuind parțial la redarea unui nivel minim de realism vizual.

3. Gestionarea resurselor.

   Acest obiectiv urmărește abstractizarea componentelor framework-ului cu sco-

pul de a oferi control utilizatorului în gestionarea resurselor și posibilitatea de a extinde funcționalitățile de bază pe care aplicația le are. Astfel, se evita supra-încărcarea anumitor module ale acestuia prin separarea logica a componentelor în clase distincte, oferind o structură clară, ușor de gestionat și ușor de înțeles pentru utilizator capabilă de a fi extinsă.

4. Gestionarea intrărilor de la utilizator.

   Obiectivul constă în facilitarea interacțiunii dintre jucător și aplicație. Pentru a fi îndeplinit, se dorește implementarea unui sistem simplu care să permită preluarea comenzilor primite de la utilizator folosind periferice precum tastatura și mouse-ul, acestea fiind ulterior interpretate și integrate în logica jocului, având ca efect actualizarea stării acestuia.

Obiectivele enumerate anterior reflectă funcționalitățile pe care aplicația își dorește să le includă, acoperind aspectele referitoare la grafică, logica jocului și interacțiunea utilizatorului cu acesta.

Astfel, aplicația va servi ca un punct de plecare în dezvoltarea jocurilor de tip stealth și va oferi posibilitatea de a lucra direct cu unul dintre cele mai performante API-uri grafice ale erei moderne, Vulkan. Totodată, prin stilul de programare low-level impus, programatorul va fi expus unui mediu care presupune gestionarea eficientă a resurselor implicate în dezvoltarea motoarelor de jocuri.

Capitolul 3. Studiu bibliografic

În capitolul 1, secțiunea 3, intitulată "What is a game engine?" din cartea [2], autorul vorbește despre popularizarea conceptului de motor de jocuri prin intermediul jocului *DOOM* publicat de id Software, în perioada anilor 90. Arhitectura acestui joc era clar structurată și separată pe componente, ceea ce a adus un plus de valoare atunci când compania a început sa licențieze jocul și sa refolosească aceste unelte pe care le avea la dispoziție pentru a crea alte jocuri. De asemenea, studiourile independente și cu resurse limitate au început să modifice jocurile și arhitectura acestora prin folosirea unor seturi de instrumente gratuite, oferite de dezvoltatorii originali.

Spre finalul anilor '90, același autor relatează faptul că jocurile precum *Quake III Arena* si *Unreal* au fost gândite și proiectate de la început, astfel încât să permită schimbarea și refolosirea componentelor de baza într-un mod mult mai ușor față de celelalte jocuri. Acest lucru a fost făcut posibil datorita limbajelor de scripting, precum Quacke C de la id Software. Astfel, dezvoltatori mari au adus pe piață motoarele de jocuri pe care studiourile independente le pot folosi gratuit, local, dar sunt nevoite să achite prețul unei licențe asociate pentru a publica jocul și a obține la rândul lor venituri.

Asa cum s-a precizat în capitolul 1 al acestei lucrări, datorita multitudini de genuri de jocuri și a veniturilor adiționale generate de licență, multe companii au dezvoltat motoare de jocuri care oferea o multitudine de componente menite să acopere cât mai multe dintre cerințele dezvoltatorilor, adresându-se astfel unui public cât mai extins.

Cu toate că, la momentul actual dezvoltatori de jocuri au la dispoziție o serie de motoare de jocuri performante și complexe, spre exemplu *Unity*, *Unreal Engine 5*, *CryEngine*, aceștia aleg sa își dezvolte propriul motor de jocuri cu scopul de a deține control absolut asupra: arhitecturi interne a jocului, portabilității și a performantei referitor la platformele țintă. De asemenea, aceștia pot elimina funcțiile care nu au nici un rol în cadrul aplicației pe care o dezvolta și rămân astfel, independenți fata de părțile terțe.

Aceasta idee de a deține control absolut asupra arhitecturii aplicației și performanței acesteia reprezintă și punctul de interes al acestei lucrări. Astfel, pentru a dezvolta un motor de jocuri dedicat genului stealth, este necesar studiul mai multor direcții esențiale, fiecare contribuind la înțelegerea și construirea unei platforme flexibile și scalabile.

Pentru a putea selecta informațiile necesare scopului lucrări, următoarele puncte de interes stau la baza documentării și dezvoltării aplicației:

- Arhitectura motoarelor de jocuri
- API-ului Vulkan
- Cerințele specifice ale unui joc de tip stealth
- Simularea comportamentului NPC-urilor prin AI.
- Compararea idei proiectului cu framework-uri existente

3.1. Arhitectura motoarelor de jocuri

3.1.1. Modul de joc

Modul de joc presupune definirea unui set de reguli care determină interacțiunea dintre entitățile jocului, obiectivele pe care jucătorul trebuie sa le atingă, precum și abilitățile pe care acesta le are la dispoziție. Pentru a oferi un mediu în care aceste reguli să poată evolua, modul de joc mai are nevoie de o lume 2D sau 3D, ficțională, alcatuită dintr-o serie de elemente statice (teren, clădiri, drumuri) și dinamice(vehicule, personaje, lumini dinamice), care să îndrume jucătorul spre atingerea scopului, să îi ofere un punct de revenire atunci când nu reușește să își atingă obiectivul pentru a-și relua progresul pierdut și care aduce un aspect cinematic și plăcut vizibil pe plan secundar. În funcție de complexitatea și dimensiunea jocului, acesta lume poate fi împărțită în bucăți, reprezentând diferitele etape prin care jucătorul trebuie să avanseze. [Cap. 15, pp. 1015-1020][2]

Din punct de vedere arhitectural, aceste mecanici pe care jocul se bazează sunt rezultatul a mai multor componente tehnice ale motorului de jocuri: motorul de randare 2D sau 3D, sistemul de coliziune, simularea în timp real a dinamici jocului, gestionarea entităților, evenimentelor și a componentelor.

3.1.2. Dinamica jocului în timp real

Scopul dinamici jocului este de a aduce o componenta temporala în cadrul spațiului 2D sau 3D amintit mai sus. Aceasta componenta ajută la spațierea în timp a evenimentelor care se desfășoară în fața jucătorului și îi oferă posibilitatea de a reacționa și a lua decizi importante. Fără aceasta, jocul devine prea rapid sau imposibil de jucat.

Cu toate ca în capitolul 8 *The Game Loop and Real-Time Simulation* al cărții [2] autorul descrie în detaliu tehnic metodele de dezvoltare a mecanismului care simulează efectul temporal al realități din cadrul motoarelor de jocuri, ideea principală care stă la baza acestui sistem este aceeași cu cea a filmelor: schimbarea succesiva și în mod rapid a imaginilor statice pentru a produce iluzia mișcării și a interacțiunii. Astfel, această componentă reprezintă una dintre elementele fundamentale care stau la baza arhitecturii unui motor de jocuri, fără de care restul componentelor ar fi nesincronizate și incompatibile.

3.1.3. Gestionarea resurselor

Gestionarea resurselor reprezinta, în acest context, un termen general care cuprinde procesele implicate în încărcarea, organizarea, și utilizarea datelor externe de care jocul are nevoie pentru a funcționa. Aceste resurse includ, în primul rand, fișierele sursă ale codului, care sunt separate pe componente specifice care îndeplinesc sarcini bine definite în arhitectura motorului. Texturile și modelele 3D ale obiectelor sunt, de asemenea, separate pe categorii facilitând gestionarea și accesul ușor asupra acestora din interiorul aplicației.

Avantajul pe care aceasta organizare îl oferă dezvoltatorilor este scalabilitatea. Odată cu creșterea în complexitate a jocului, resursele sunt gestionate eficient prin clasificarea și încărcarea lor în funcție de contextul de utilizare. Un exemplu potrivit care susține această abordare îl reprezintă gestionarea diferitelor elemente ale mediului jocului și anume cele statice și dinamice. Elementele statice, precum terenul, clădirile și alte decoruri fixe, pot fi optimizate la momentul randării prin precalcularea iluminării și a

coliziunilor fără mai fi nevoie de actualizarea acestora în timp real. Astfel, elementele dinamice: personajele, efectele speciale, obiectele mobile beneficiază de o putere crescuta de computare pentru afișare datorită optimizărilor specificate anterior.[Cap. 7, pp. 481-523][2]

3.1.4. **Motorul de randare 2D sau 3D**

Una dintre ultimele componente menționate aici, care face parte din arhitectura unui motor de jocuri, este sistemul de randare 2D sau 3D. Acest subsistem are rolul de a genera imaginea vizibilă pe ecran prin afișarea obiectelor din lumea jocului, pe baza datelor furnizate de celelalte componente. Randarea este realizata de către placa video, iar comunicarea dintre aplicație (proces ce rulează pe CPU), datele aplicației și GPU se face prin intermediul unui API grafic precum *DirectX, OpenGL sau Vulkan.*

Motivul pentru care se folosește un astfel de API este de a abstractiza interacțiunea directă cu hardware-ul grafic, oferind un set de funcționalități standardizate care pot trimite comenzi către placa video. În lipsa acestuia, dezvoltatorul ar trebui să scrie cod specific pentru fiecare tip de placă video, ceea ce ar face dezvoltarea jocurilor imposibilă. De asemenea, acesta este privit ca un intermediar intre aplicație și hardware, oferind un limbaj comun prin care se pot descrie funcționalitățile dorite. Totodată, în funcție de nivelul de control și complexitate oferit, se disting două categorii de API-uri grafice: de nivel înalt precum *OpenGL* și *DirectX* sau de nivel mai jos apropiat de hardware precum *Vulkan.* Cele de nivel mai jos oferă o optimizare mult mai fina, dar necesita un efort mult mai mare în ceea ce privește programarea.[Cap 11][2].

Revenind la subiectul randări, mai este de precizat faptul că acesta urmărește un proces denumit "*Rendering Pipeline*"[Cap 11.2, pp. 668 - 697][2] care consta într-o serie de pași care operează cu datele primite ca intrare pentru a afișa în mod corect pe ecran imaginile dorite. Detalierea acestor pași este descrisa în capitolele 4 si 5.

3.2. Framework-uri existente

Pentru a oferi claritate în procesul de proiectare și structurare a motorului de jocuri, s-a analizat framework-ul 2D `LÖVE` ,love2d.org. Acest framework este scris în limbajul de programare Lua, fiind deschis publicului(open source). Framework-ul este portabil pe toate platformele cunoscute, iar utilizarea acestuia este intuitivă în scopul creării jocurilor 2D, fiind însoțită de un tutorial explicativ.

Cel mai popular joc creat cu acest framework este jocul `Balatro`, câștigătorul *The Game Awards 2024.* playbalatro.com.

Astfel, analiza acestui framework va fi folosita pentru a ghida organizarea și definirea funcțiilor prin care utilizatorul interacționează cu motorul de jocuri, facilitând dezvoltarea cat mai eficientă a jocurilor 3D.

Capitolul 4. Analiză și fundamentare teoretică

Pentru a explica principiile funcționale ale aplicației, este necesar ca, mai întâi, să fie detaliate cerințele funcționale și non-funcționale, pentru a avea o privire de ansamblu asupra componentelor care vor alcătui arhitectura logica a aplicației. De asemenea, se vor analiza și cazurile de utilizare relevante, cu scopul de a forma designul teoretic al aplicației.

4.1. Cerințe funcționale

4.1.1. Inițializarea și configurarea sistemului Vulkan

Conform definiției unui game engine, acesta introduce un nivel suplimentar de abstractizare între limbajul de nivel jos al hardware-ului și conceptele pe care programatorul își dorește să le implementeze rapid și eficient. Din acest motiv, este necesar ca aplicația sa ofere de la început o modalitate prin care programatorul sa fie scutit de configurarea detaliata a tuturor dependințelor necesare pentru ca Vulkan API să poată afișa grafică pe ecran.

Pe baza structurii generale descrise în capitolul *Overview* din [11], pași care pun la dispoziție toate funcționalitățile de care este nevoie sunt:

- Instantierea și selectarea unui dispozitiv fizic
- Dispozitivul logic și familia de cozi(*Queue families*)
- Suprafața ferestrei și Swap chain(*Crearea unei ferestre de afișare*)
- Image views și framebuffers
- Render passes
- Pipeline-ul grafic
- Pool pentru comenzi și buffere
- Bucla principala

4.1.2. Încarcarea obiectelor 3D din fișiere .obj

În matematica euclidiana, dar și în grafica pe calculator, o forma geometrica este definită de un număr finit de puncte în plan sau spațiu numite coordonate și definesc extremele (vârfurile) respectivei forme. Cunoscând multitudinea axiomelor care definesc planele, spatiile tridimensionale și relațiile dintre forme și coordonate, aceste reguli sunt de asemenea, aplicate și în contextul graficii pe calculator.

Dacă în geometria euclidiana, un triunghi are 3 vârfuri, iar dreptele conectează câte 2 vârfuri pentru a forma triunghiul, același fenomen stă la baza afișării unui triunghi pe ecran: sunt definite 3 vârfuri în spațiul memoriei, iar GPU-ul este cel care realizează interpolarea intre cele 3 pe suprafața ecranului folosind pixeli. Plecând de la aceasta figura geometrica simpla, prin înlănțuirea mai multor triunghiuri, de data aceasta în spațiul 3D, folosind coordonatele *X, Y, Z*, se pot realiza forme geometrice complexe care definesc obiecte din spațiul real.

Cu toate ca suprafața unui ecran este 2D, GPU-ul este capabil de a realiza calculele necesare astfel încât, obiectul 3D să fie proiectat pe acea suprafața și sa ofere iluzia tridimensionalități. Este de amintit și faptul că acel obiect poate fi colorat, texturat și

iluminat, astfel GPU-ul are nevoie de date adiționale pentru a reprezenta în mod corect obiectul.

O prima soluție, care nu oferă scalabilitate este de a defini manual toate vârfurile, coordonatele de textura, normalele de pe suprafețe pentru calculul iluminării și culoarea fiecărui punct. Soluția funcționează doar pentru forme simple cum ar fi un cub, sau o piramidă, dar dacă este vorba de un model al unei mașini detaliate, este vorba despre mii de astfel de vârfuri și caracteristici specifice, care ar lua un timp exagerat de mult pentru a introduce aceste date.

A doua soluție și cea care va fi implementata, este folosirea fișierelor de tip *.obj* care oferă un mod standardizat de a reprezenta forme geometrice complexe, incluzând și detaliile obiectului. Astfel, într-un fișier .obj se regăsește mulțimea tuturor componentelor unui obiect sub formă de text, care doar trebuie parsată și transmisă mai departe GPU-ului.

Pentru a face acest lucru posibil va fi nevoie de implementarea metodei care sa abstractizeze parsarea și trimiterea datelor către GPU.

Tabela 4.1: OBJ file keys [12]

Key	**Description**
#	Comment
v	Vertex
l	Line
f	Face
vt	Texture Coordinate
vn	Normal
g	Group
...	...

4.1.3. Implementarea unei camere virtuale

Pentru a transforma scena tridimensionala într-o imagine 2D care poate fi afișata pe ecran, în grafica 3D acesta operație este realizată prin intermediul unei camere virtuale, care simulează modelul de funcționare al unei camere de filmat reale. Fără o astfel de camera, nu ar exista nici un punct de vedere prin care scena sa poată fi *observată*.

Această cameră funcționează ca și un ochi virtual, descrisă de o poziție, o direcție de privire și "lentila" care determina cat de larg este câmpul vizual. Pin intermediul acestor 3 componente se definește ce parte a scenei este vizibilă și cum sunt privite obiectele în funcție de distanța față de cameră, simulând efectul de perspectiva. Figura 2.1 din [Cap. 2][1] oferă o imagine vizuala explicativă și este inclusă în acestă subsecțiune la 4.1.

Scopul motorului de jocuri este de a oferi implementarea camerei virtuale programatorului de la început, permițându-i acestuia de a schimba doar parametrii poziției, direcției și a câmpului vizual, cu posibilitatea doar de a instanția mai multe astfel de camere în scena virtuală, la nevoie.

4.1.4. Aplicarea texturilor și iluminarea scenei

Fișierele de tip .obj oferă doar date despre obiecte și nu aplică automat texturile pe obiecte. Astfel, texturile vin la pachet cu aceste fișiere și sunt imagini 2D simple,

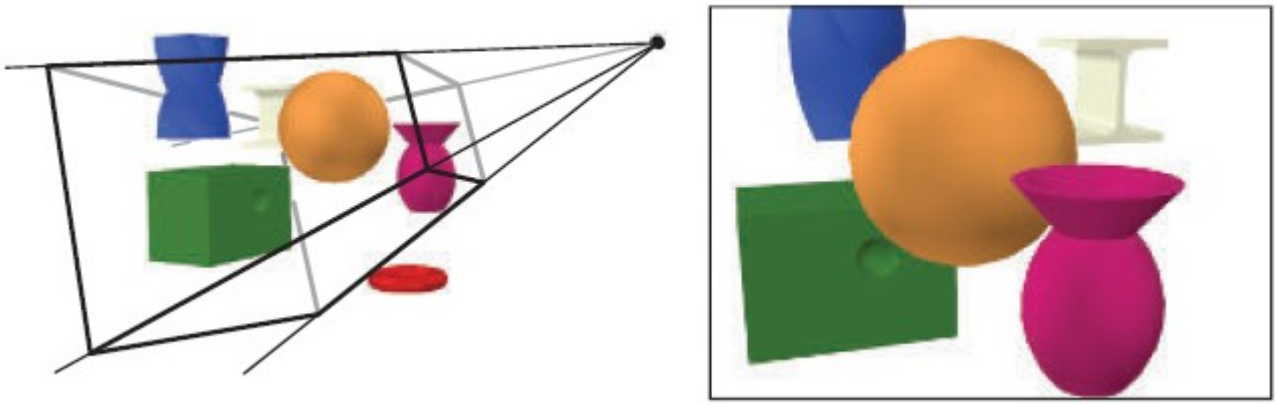


Figura 4.1: Model vizual al unei camere virtuale, preulat din [Cap. 2][1]

salvate ca .png sau .jpg. Pentru a putea textura un obiect 3D este necesar ca datele care descriu coordonatele de textură a obiectului să fie proiectate/ translatate pe imaginea 2D, astfel încât GPU-ul să acceseze conținutul imaginii și să interpoleze datele pe suprafețele obiectului. Această corelare dintre datele obiectului și accesarea imaginii din memorie este un proces care la rândul lui trebuie abstractizat în cardul motorului de jocuri.

Iluminarea scenei presupune modelarea diferitelor surse de lumină întâlnite în lumea reala și aduse în spațiul virtual al scenei pentru a interacționa cu obiectele, aducând o nota de realism. Bineînțeles, într-un motor de jocuri, pentru modelarea rapidă a iluminării în scenă, inginerul are nevoie de funcțiile care automatizează procesul de punere în scenă a diverselor tipuri de lumină.

4.1.5. Integrarea unui algoritm de mișcare pentru NPC-uri

Scopul pe care aceasta cerința funcțională îl are de îndeplinit, este de a implementa și a oferi caracterelor de tip NPC o modalitate prin acestea să poată interacționa cu mediul de joc. Așa cum este prezentat în capitolul 3, NPC-urile simulează comportamentul unei persoane. Astfel, acest caracter trebuie să fie capabil de a se mișca autonom în limitele scenei și sa poată lua decizii în funcție de anumiți stimuli pe care îi receptează din mediul exterior.

În cadrul contextului unui joc de tip stealth, un NPC trebuie să fie capabil să:

- Se deplaseze conform unor algoritmi de găsire a drumului.
- Efectueze o acțiune corespunzătoare stimulului pe care îl percepe din mediul exterior.

Elementele enumerate mai sus, reprezintă cerințele minime pentru realizarea acestei cerințe funcționale. Vor fi implementate și abordate în cadrul motorului de jocuri, fiind detaliate în capitolul 5.

4.1.6. Crearea unui sistem de gestionare a nivelurilor

Componentă care aparține modului de joc, nivelul reprezinta gruparea logica de obiecte, reguli și condiții care definesc o etapa pe care jucătorul trebuie să o parcurgă pentru a progresa spre scopul final.

Datorita complexității crescute a jocurilor moderne, îndeplinirea scopului final presupune o succesiune extinsa de acțiuni și evenimente, desfășurate adesea pe o perioada

lunga de timp, pe care jucătorul nu o poate parcurge întotdeauna într-o singură sesiune de joc. Pentru a facilita progresul și organizarea logica a jocului, pașii spre scopul final sunt fragmentați în obiective intermediare reprezentate de un nivel distinct.

În acest context un motor de jocuri are nevoie de un sistem dedicat pentru gestionarea nivelurilor, care sa permită descrierea, salvarea stării și eventual reutilizarea obiectelor și elementelor componente ale unui nivel. Acest sistem trebuie separat de celelalte module ale motorului și sa ofere scalabilitate pentru dezvoltarea eficienta a conținutului de joc.

4.2. Cerințe non-funcționale

Regăsite în capitolul 2, cerințele non-funcționale aduc un plus de stabilitate aplicației și contribuie la conturarea arhitecturi pe care aceasta se bazează.

- Performanta.
  Motorul de jocuri trebuie să asigure un timp de afișare constant și optim, astfel încât să poată rula în timp real fără întârzieri vizibile.
- Extensibilitate și scalabilitate.
  Se dorește ca motorul să permită adăugarea de noi funcționalități care ajuta la creșterea complexității modului de joc, într-un mod ușor de integrat, fără a fi nevoie de restructurări majore în arhitectura internă a motorului.
- Modularitate.
  Gruparea pe module a funcționalităților care compun motorul de jocuri, contribuind la organizarea clară a codului și permițând dezvoltarea, testarea și întreținerea independenta a fiecărei componente.
- Simplitatea utilizării.
  Deși aplicația nu va dispune de o interfață de utilizator(UI - User Interface), ci va afișa doar output-ul vizual generat de motor, este nevoie ca funcționalitățile interne să poată fi utilizate în mod intuitiv.

În urma analizei acestor cerințe funcționale și non-funcționale, se poate contura o viziune din ansamblu asupra arhitecturii logice a motorului de jocuri, precum și asupra provocărilor pe care le implica dezvoltarea acestuia. Chiar dacă unele cerințe, precum gestionarea ferestrei și intrărilor de la tastatură pot fi abordate și implementate prin aplicarea unor principi cunoscute, proiectarea sistemului de propagare a sunetului și mecanismul de căutare al sursei de proveniență a sunetului de către NPC reprezintă provocările care necesita găsirea unei soluții satisfăcătoare scopului lucrării.

4.3. Cazuri de utilizare

Pentru a oferi ajutor în conceperea arhitecturii logice a motorului de jocuri, analiza unor cazuri de utilizare poate oferi detalii referitoare la structurarea modulelor pentru a oferi o utilizare cat mai eficienta a aplicației.

În diagrama 4.2 au fost enumerate câteva modalități prin care dezvoltatorul poate interacționa cu mecanismele interne ale motorului de jocuri pentru obținerea unor rezultate noi.

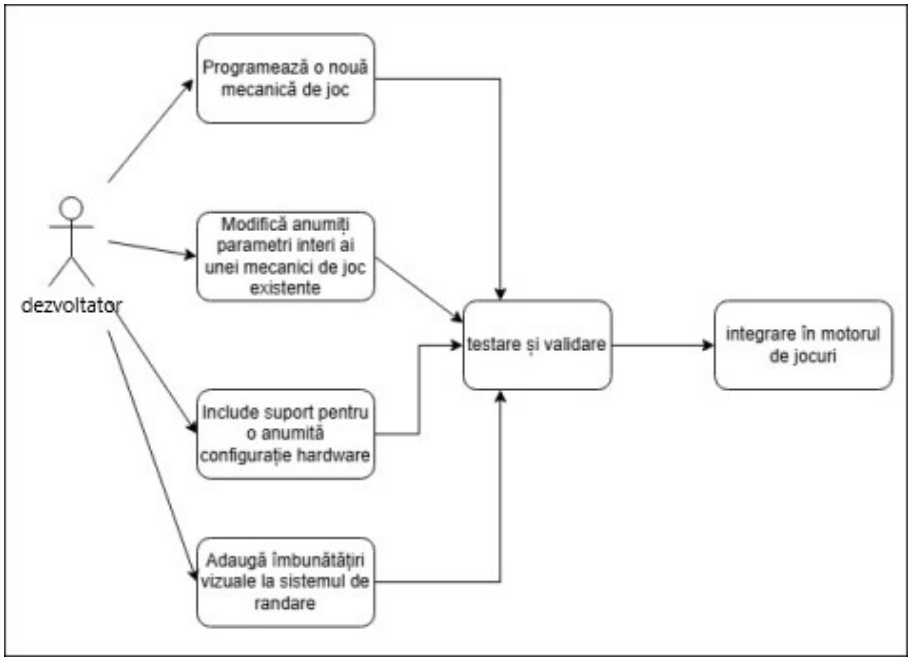


Figura 4.2: Diagrama cazurilor de utilizare

Unul dintre cazurile de utilizare care necesita o înțelegere cât mai complexa a motorului de joc este adăugarea îmbunătățirilor vizuale la sistemul de randare, precum adăugarea Skybox-ului.

Skybox-ul este un element vizual folosit în grafica 3D pentru a simula un fundal static infinit la fel ca și cerul sau peisajele în planul îndepărtat. Este alcătuit, de cele mai multe ori, de un cub texturat cu o hartă de mediu(textura care infățișează o vedere 360 a fundalului), care este poziționat în jurul scenei. Deoarece acest element se deplasează deodată cu camera virtuala, el oferă iluzia unei lumi infinite contribuind la veridicitatea realismului în scena.[Cap. 13][1]

Astfel, pentru ca un dezvoltator sa adauge aceasta funcționalitate trebuie mai întâi să definească poziția acestui cub în scena. Apoi este nevoit sa modifice modul în care sunt interpretate coordonatele de textura, astfel încât fiecare fata a cubului sa corespunda corect unei porțiuni din harta de mediu.

Pentru a simula infinitatea fundalului, cubul este mereu poziționat în centrul camerei și se deplasează odată cu aceasta , dar fără să fie afectat de perspectiva camerei, deoarece nu se mai realizează iluzia dorită.

Adițional în sistemul de randare trebuie modificat modul în care sunt procesate și afișate pe ecran obiectele scenei, întrucât acestea trebuie să fie afișate în ordinea adâncimi lor în scena, din punctul de vedere al camerei. Astfel, skybox-ul trebuie redat separat și fără a fi afectat de adâncimea relativa a scenei.

Pe baza analizei de mai sus, motorul trebuie sa fie modularizat astfel încât să permită integrarea noilor elemente grafice, dar și schimbarea modului de funcționare al camerei fără a schimba funcționalitățile și caracteristicile precedente.

4.4. Soluția problemei și metodologiile folosite

Conform celor prezentate în secțiunile precedente, pentru realizarea acestei aplicații s-a conceput o diagrama conceptuala a arhitecturi aplicației care prezintă la nivel teoretic componentele motorului de jocuri și relația dintre acestea. Aceasta diagrama va fi detaliata în cele ce urmează și va ajuta la implementarea aplicației finale în capitolul 5.

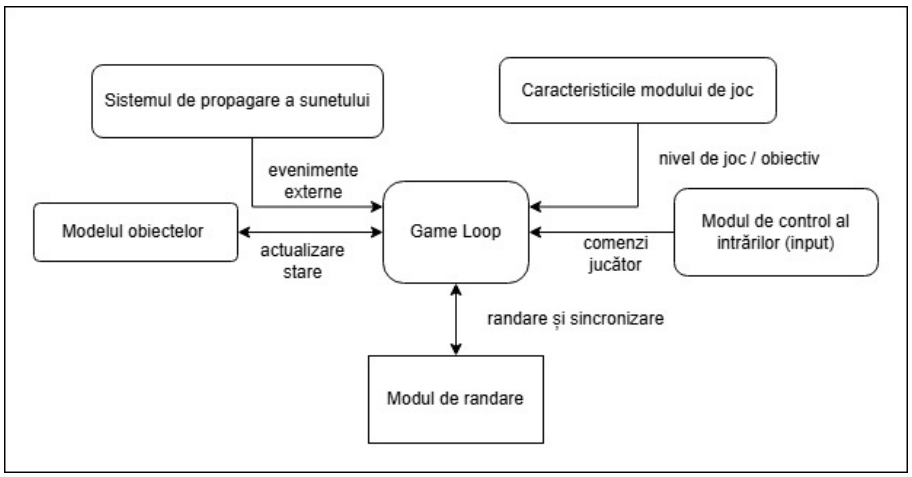


Figura 4.3: Diagrama conceptuala

4.4.1. Modulul de randare

Aceasta componenta, reprezentata în figura 4.3, este responsabilă pentru transformarea datelor logice din joc(pozițiile obiectelor, culorile prezente, ordinea obiectelor în scena etc.) în imagini vizibile pe ecran. Astfel, modulul de randare are scopul de a genera cardele grafice folosind interfața grafica Vulkan. Din punct de vedere teoretic se vor detalia conceptele și pașii pe care Vulkan API le folosește pentru a afișa grafica pe ecran și care au fost amintite pe scurt în 4.1.1 și se regăsesc în [11].

1. **Instanta și selectarea unui dispozitiv fizic**.
   Orice aplicație Vulkan începe prin instanțierea API -ului. Aceasta instanță este folosită pentru a descrie scopul aplicației și extensiile pe care aceasta le va folosi. Totodată, în acest prim pas, după crearea instanței se va interoga sistemul pentru găsirea unei sau mai multor componente hardware care oferă suport pentru folosirea și înțelegerea API ului Vulkan. De asemenea, se pot interoga și alte specificații hardware, precum mărimea VRAM-ului sau capabilitățile dispozitivului

2. **Dispozitivul logic și familia de cozi**
   După selectarea hardware-ului corespunzător este necesar crearea unui dispozitiv logic care va face legătura dintre aplicația logica și hardware. Astfel, prin intermediul dispozitivului logic se pot specifica funcționalitățile folosite, ce fel de tip-uri de date se folosesc(exemplu: float pe 64 biti), randare în mai multe ferestre de vizualizare(exemplu: afișarea simultan a mai multor fețe ale cubului din diferite

perspective).
Majoritatea operațiunilor care se execută prin API-ul Vulkan(comenzi de desenare, operați cu memoria) sunt executate asincron prin intermediul cozilor. Aceste cozi sunt alocate dintr-o familie de cozi, unde fiecare dintre acestea efectuează un set specific de opereațiuni. Disponibilitatea acestor familii de cozi poate fi un factor de selecție a dispozitivelor grafice fizice.

3. **Suprafața ferestrei și swap chain-ul**
   Termenul de fereastra este folosit pentru a descrie spațiul de pe ecran în care aplicația rulează și spațiul în care utilizatorul poate interacționa cu aceasta.
   În contextul Vulkan pentru a afișa grafica într-o astfel de fereastră sunt necesare două componente, provenite din extensii ale API-ului, care trebuie configurate: *suprafața ferestrei și swap chain-ul*.
   Fiind un API care nu depinde de platforma, comunicarea dintre managerul de ferestre al sistemului se face prin intermediul unei extensii standard numita WSI (Window System Interface - Interfata Ferestrei Sistemului). Astfel, suprafața ferestrei este o abstractizare independentă de platforma și are nevoie de o referință spre managerul de ferestre a sistemului pe care aplicația îl folosește.
   Swap chain este un termen specific Vulkan și scopul de bază al acestuia este să se asigure de faptul că imaginea care urmează să fie afișată este diferita de cea care se afla pe ecran. Acest concept este foarte important deoarece este cel care se asigură că pe ecran ajung doar imagini complete. De fiecare dată când se dorește afișarea unei imagini, acest ”lanț” este interogat pentru returnarea imaginii cerute.

4. **Image view si framebuffers**
   Acești termeni fac parte din vocabularul folosit de Vulkan. O componenta image view reprezintă o parte specifica a unei imagini care urmează sa fie folosita, iar framebuffer face referire la image view-urile care folosesc culori, adâncime și șabloane(stencils).

5. **Render pass - etapa de randare**
   Aceasta etapa este folosita in Vulkan API pentru a descrie tipul de imagini folosite în timpul operaților de randare, cum vor fi folosite, și cum ar trebui procesat conținutul lor.

6. **Pipeline-ul grafic**
   Termenul de pipeline grafic, în contextul graficii pe calculator, reprezinta pașii care prelucrează datele din memoria aplicației cu scopul da a fi afișate pe ecran, similar linilor de asamblare din fabrici.[1].
   Pipeline-ul grafic în Vulkan este folosit pentru a descrie starea configurabila a placi grafice și starea programabila a acesteia prin intermediul shaderelor.
   Un shader este acel program care se executa strict pe partea de GPU. Shaderul este programat folosind limbaje de programare similare cu C (exemplu: GLSL), însă Vulkan API interpretează acel cod folosind standardul SPIR-V(Standart Portable Intermediate Representation - Vulkan) care compilează codul shaderului scris în GLSL sau alte limbaje direct în bytecode, contribuind la portabilitatea generală a API-ului.[1].
   Una dintre cele mai reprezentative caracteristici ale Vulkan, comparativ cu alte API-uri grafice, este faptul că aproape toate configurațiile pipeline-ului grafic trebuie

setate din timp. Ca și consecință a acestui fapt, simpla schimbare a unui shader, sau schimbarea structurilor folosite, necesită reconstruirea întregului pipeline grafic. De asemenea, toate stările necesita descrieri explicite deoarece nu există stări de baza care sa înlocuiască valorile lipsă.
Avantajul pe care acesată abordare îl aduce este întâlnit la timpul de compilare, deoarece optimizările din acest stadiu sunt mult mai predictibile datorita configurărilor explicite.

7. **Pool-ul de comanda și bufferele de comandă**
   Asa cum s-a menționat anterior, majoritatea operațiunilor din Vulkan sunt transmise cozilor. Dar mai întâi de a fi trimise ele trebuie înregistrate în bufferele de comandă care la rândul lor sunt alocate din pool-ul de comandă asociat fiecărei familii de cozi.

8. **Bucla principala**
   În acest ultim pas are loc tranziția către următorul component al diagramei și anume bucla principală. Modulul de randare este responsabil cu preluarea datelor generate de procesarea evenimentelor și a logici de joc din cadrul buclei pricipale și afișarea acestora pe ecran.

Astfel, aceasta componenta are rolul de a crea imaginea vizuală, în funcție de starea actuală a buclei de joc(*Game Loop*), poziționarea obiectelor în scena.

4.4.2. Game Loop - bucla principala a jocului

Întru-un game engine, componenta centrală care este responsabilă cu preluarea și procesarea evenimentelor introduse de la mouse și tastatură, actualizarea logici jocului și sincronizarea datelor cu modulul de randare pentru afișarea acestora pe ecran se numește *game loop*.

Din punct de vedere teoretic, funcția principala pe care un game loop o are de îndeplinit este de a crea iluzia mișcării continue a elementelor în timp real, adesea fiind vorba despre o bucla infinita care se execută pe întreaga durată de viață a aplicației game engine-ului. Astfel, la fiecare trecere prin această buclă, datele componentelor asociate sunt actualizate, iar rezultatele primite sunt trimise către modulul de randare. Pentru a asigura sincronizarea între starea datelor și imaginea de pe ecran, game loop-ul așteaptă ca compunerea imaginii să fie încheiată cu succes, ca pe urmă să reia procesul descris. Termenul folosit care descrie sincronizarea și afișarea pe ecran are denumirea de cadru și în domeniul graficii pe calculator timpul în care se afișează cadrele pe ecran este măsurabil în cadre pe secunda(FPS - Frames Per Second), un indicator esențial al performantei grafice. Criteriul minim pe care game loop-ul trebuie sa îl îndeplinească pentru a fi catalogat drept optimizat și eficient este de a afișa 60 de cadre pe secunda. Acest număr provine în urma cercetărilor din domeniul percepției vizuale, care afirmă faptul că ochiul uman percepe informația vizuală în mod fluid la 60 de cadre pe secunda [1].

Bucla principala a jocului nu se limitează doar la performanța cu care aceasta afișează imaginile provenite de la modulul de randare, ci aceasta are rolul de mediator intre toate celelalte componente care fac parte din arhitectura motorului de joc. Astfel, este responsabilă de a inițializa relațiile dintre componente și de a crea contextele în care acestea se desfășoră.

Folosind această abordare, game loop-ul nu doar sincronizează fluxul de date între module, ci asigura o experienta de joc interactivă și realistă, bazată pe reacții în timp real. Fiecare componenta depinde de acest mecanism pentru a ști când și cum să acționeze, transformând bucla principala într-un centru de comandă a motorului de joc.

Astfel, arhitectura va fi modulara, dar și interdependenta, fiecare element find gestionat în cadrul buclei principale, permițând scalabilitate, reutilizare și control absolut asupra execuției aplicației.

4.4.3. **Modulul de control al intrărilor**

Scopul acestei componente este de a procesa evenimentele provenite de la periferice(mouse și tastatură) în urma intercepției acestora de către bucla principala, prin intermediul contextului ferestrei aplicației. Procesarea propriu-zisă nu se limitează doar la simpla detecție a apăsărilor de taste, ci introduce un nivel de abstractizare care definește acțiunile posibile ale entităților de joc și modul în care logica își schimbă starea.

Prin acesta abordare, comportamentele de bază precum deplasarea în scena a obiectelor controlate de jucător, rotirea camerei sau interacțiunea cu diferite obiecte, sunt descrise în funcție de tipul de acțiune din lumea reala(mergi înainte, sari, interacționează cu obiectul "X"), care reprezintă abstractizări asupra tastelor fizice. Astfel, se creează o separare clară intre input și comportamentul rezultat, care permite o flexibilitate și scalabilitate din punct de vedere a noilor acțiuni ce pot fi implementate ulterior. De asemenea, oferă motorului de jocuri posibilitatea de a se adapta și altor tipuri de control(exemplu: controller), deoarece adăugarea noilor comenzi nu influențează implementarea comportamentelor.

Pe lângă acțiunile ce pot fi declanșate de către utilizator, modulul gestionează și comportamentul entităților de tip NPC, entități care nu sunt controlate direct prin input, ci funcționează în mod autonom pe baza unor reguli bine definite. Astfel, acesta interpretează stimuli externei proveniți din mediul virtual, pe care ii transforma în acțiuni autonome pe care NPC-urile le executa, incluzând și aspectul AI pe care aceste caractere îl poartă. Dintre aceste acțiuni posibile care acoperă aspecte cheie ale unui joc stealth se enumera: deplasarea spre sursa sunetului, declanșarea terminării jocului atunci când intereacționează cu o entitate a jucătorului, urmărirea jucătorului.

Astfel, modulul de control al intrărilor devine un mecanism esențial atât pentru interacțiunea om-mașina, cât și pentru coordonarea comportamentelor dinamice ale NPC-urilor.

4.4.4. **Caracteristicile modului de joc**

Aceasta componentă realizează separarea descrierilor care definesc mediul virtual al jocului fata de restul componentelor motorului de jocuri.

Din punct de vedere teoretic se dorește ca această componentă să descrie elementele statice și dinamice care alcătuiesc nivelurile jocului, precum și numărul entităților și al obiectivelor care vor guverna scena și scopul principal pe care jucătorul trebuie sa îl atingă. Astfel, se detaliază cat mai succint cum va arata scena din punct de vedere static prin includerea obiectelor ce definesc clădiri, terenul pe care se vor deplasa entitățile și iluminarea scenei. De asemenea, sunt descrise și limitele scenei de care orice entitate nu poate sa treacă, proprietățile de coliziune ale obiectelor statice și structura cailor de acces ale entităților.

Datorită acestei abordări, dezvoltatorul de jocuri își poate defini mediul virtual fără a interacționa cu alte componente ale motorului de jocuri, concentrându-se doar pe

aspectul vizual pe care scena îl va avea și poziția generală pe care obiectele ce reprezintă entități o vor avea. Astfel, prin oferirea acestei flexibilități se face o separare clara între logica jocului și descrierea medului, ceea ce facilitează reutilizarea scenelor în contexte diferite.

De asemenea, aceasta compartimentare contribuie la dezvoltarea eficienta și modulara a jocului, deoarece permite ca nivelurile să fie construite, testate și ajustate independent de restul sistemului, fără a afecta celelalte module.

4.4.5. **Sistemul de propagare a sunetului**

Jocurile de tip stealth se bazează pe colectarea informațiilor necesare jucătorului pentru a avansa în nivel fără a oferi, la rândul său, informați pe care inamicii le pot folosi pentru a descoperi poziția sa. Una dintre modalitățile prin care jucătorul oferea tot timpul astfel de informații către mediul exterior este sunetul pe care îl produce atunci când se deplasează în interiorul nivelului. Astfel, simularea sunetului în spațiul 3D al jocurilor nu are doar un rol estetic, ci contribuie direct la crearea unei experiente de joc credibile și reprezintă o componentă esențiala în proiectarea unui game engine dedicat acestui scop.

În cadrul aplicației, acest sistem este proiectat folosind analogia cu lumea reală. Orice sunet are o origine în care acesta s-a produs. În funcție de nivelul de zgomot al sunetului acesta se poate propagă în aer pe distanțe diferite fața de originea acestuia folosind *Legea inversului pătrat al sunetului* descris de formula 4.1, unde I reprezinta intensitatea sunetului la distanta d.

$$I \propto \frac{1}{d^2} \tag{4.1}$$

Componenta va fi responsabilă de a cataloga, în funcție de intrările utilizatorului și interacțiunea acestuia cu mediul jocului, nivelul de intensitate al sunetului și distanta pe care acesta va fi propagat. Aceasta abordare va oferi dezvoltatorului posibilitatea de a crea situații în care jucătorul este nevoit sa analizeze tactic următoarea mișcare, întrucât orice acțiune va produce zgomot și, eventual, va alerta entitățile inamice.

Modelul teoretic folosit pentru propagarea sunetului este construit sub forma unei rețele de noduri așezate în spațiu, unde fiecare nod acționează ca sursă a sunetului atunci când acesta interacționează cu entitatea jucătorului. În momentul generării unui sunet, acesta se propaga de la nodul sursă către nodurile adiacente, atenuându-se în funcție de distanța și numărul de noduri prin care sunetul s-a propagat, oferind un punct de vedere vizual asupra simulării sunetului.

Astfel, separarea acestei componente față de celelalte module ale game engine-ului contribuie la menținerea arhitecturii modulare care oferă dezvoltatorului libertatea de a extinde, modifica sau înlocui comportamentul sistemului de propagare a sunetului.

4.4.6. **Modelul Obiectelor**

Cu ajutorul acestui motor de jocuri se urmărește crearea jocurilor de tip stealth cât mai complexe, fiind necesar ca dezvoltatorul să își poate defini în mod clar tipurile de obiecte și cum sunt acestea percepute în lumea virtuală. Cu toate că, într-un joc simplu se regăsete doar entitatea jucătorului, un inamic și obiectivul nivelului, pentru scalarea jocului este novoie de introducerea a mai multor tipuri de entități care trebuie organizate în mod eficient cu scopul de a spori performanța implementări. Astfel, dezvoltatorul poate opta pentru introducerea unor inamici ce simulează un comportament diferit fata

de cel de baza, ca de exemplu cele care dezvăluie în mod constant pentru câteva secunde poziția jucătorului tuturor entităților inamice după intersecția acesteia cu jucătorul sau îngreunează mobilitatea jucătorului prin intermediul unor forte odată ce acesta a fost observat. De asemenea, când se face trimiterea la termenul de obiect, conceptul nu este limitat doar la obiectele care reprezintă entități și elemente de decor statice, ci pot face referire și la obiectele pe care jucătorul le observa vizual dar care reprezintă cele care au nevoie de interacțiunea cu acesta pentru realizarea obiectivului din cardul nivelului.

Astfel, pentru a putea descrie un nivel de joc și crea un ecosistem în mediul de joc intre personajele care participa la acțiune, este necesar ca motorul de jocuri să ofere posibilitatea de a organiza și cataloga aceste obiecte în funcție de rolul pe care îl au și abilitățile de care sunt capabile, cu scopul de a simplifica procesul de definire a logicii jocului.

Pentru a face posibila aceasta organizare flexibilă și extensibila a obiectelor, în cadrul motorului de jocuri se vor utiliza concepte împrumutate atât din programarea orientata pe obiecte (OOP- Object Oriented Programming), cat și din arhitectura de tip Entity-Component-System (ECS). Astfel, prin aceasta abordare, fiecare obiect din joc va fi tratat ca o entitate de bază care nu reprezintă nici o categorie, căreia i se pot atașa doar componentele care descriu caracteristicile, comportamentul și funcționalitățile specifice rolului în cardul jocului.

Acest model modular ii va permite dezvoltatorului sa creeze tipuri diverse de obiecte fără a repeta cod sau a impune ierarhii rigide, oferind scalabilitate, reutilizare și optimizarea resurselor în cadrul jocului.

În urma descrierii soluției propuse și a analizei cazurilor de utilizare împreună cu cerințele funcționale și non-funcționale, capitolul 4 prezintă la nivel teoretic funcționalitățile pe care aplicația își dorește să le dezvolte în capitolul următor, oferind totodată o direcție clară pentru etapele de implementare și construind o platforma care poate servi drept punct de plecare în domeniul de dezvoltare a jocurilor pentru viitori utilizatori.

Capitolul 5. Proiectare de detaliu și implementare

Înainte de a începe procesul de detaliere al implementării trebuie făcute câteva precizări legate de complexitatea API-ului Vulkan și despre ipoteza temei lucrării de la care s-a pornit.

Datorită naturii low-level și a stilului de programare orientat pe obiecte folosind C++ 17, Vulkan folosește în abundența structurile (*structs*) pentru a transmite parametri funcțiilor. Astfel, programatorul este obligat să specifice explicit valoarea fiecărui parametru [API concepts pp. 14-15][11]. Această abordare presupune un nivel de cunoștințe extrem de ridicat despre arhitectura Vulkan și modul de operare de la bun început. În consecință curba de învățare și familiarizare cu API-ul este abruptă și poate lua un timp îndelungat pentru adaptare. Însă acest risc a fost asumat din timp și astfel s-au găsit resursele necesare pentru a avansa rapid în dezvoltare.

Plecând de la ipoteza prezentată pentru această lucrare și anume: proiectarea și implementarea unui motor de jocuri, pornind de la funcționalitățile minime de gestionare a ferestrei, gestionare a evenimentelor și trasarea de primitive grafice punând accent pe proiectarea claselor care să facă posibila crearea unui joc și adăugarea altor funcționalități, în primi pași ai dezvoltări a fost nevoie de parcurgerea unui mic tutorial explicativ al tuturor conceptelor folosite în Vulkan pentru crearea unui render grafic(aplicație care doar afișează scena statica pe ecran și permite mișcarea camerei) conform surselor [13] și [14], deoarece [11] oferă doar o aplicație structurată într-un singur fișier ca exemplu demonstrativ.

Tutorialul este dezvoltat pentru sistemul de operare Mac OS, fiind necesară portarea acestuia pe sistemul de operare MS Windows.

Rezultatul final al tutorialului, care se este deschis publicului, este folosit ca punct de plecare în dezvoltarea aplicației finale, fiind aduse modificările și restructurările necesare îndepliniri scopului propus de lucrare: motor de jocuri video pentru genul stealth.

5.1. Arhitectura aplicației

În urma considerentelor teoretice prezentate în capitolul anterior 4, împreună cu analiza cerințelor și a cazurilor de utilizare, a fost realizată diagrama arhitecturală a aplicației care oferă o vedere de ansamblu asupra structurii sistemului.

Rolul acesteia este de a evidenția componentele principale ale game engine-ului, relațiile dintre acestea și modul de funcționare al aplicației finale, astfel încât să contribuie la o înțelegere cât mai rapidă a structurii interne de către viitori dezvoltatori. De asemenea, arhitectura propusă oferă extensibilitate pentru adăugarea de noi funcționalități, datorită construcției interne modulare, care permite, totodată, un nivel bun de scalabilitate.

Aplicația finala va fi dezvoltată folosind limbajul de programare C++ 17, API-ul Vulkan și mediul de integrare pentru dezvoltare(IDE) Visual Studio. Pentru a susține implementarea și aspectele descrise mai sus, figura 5.1 oferă reprezentarea schematică a arhitecturii propuse.

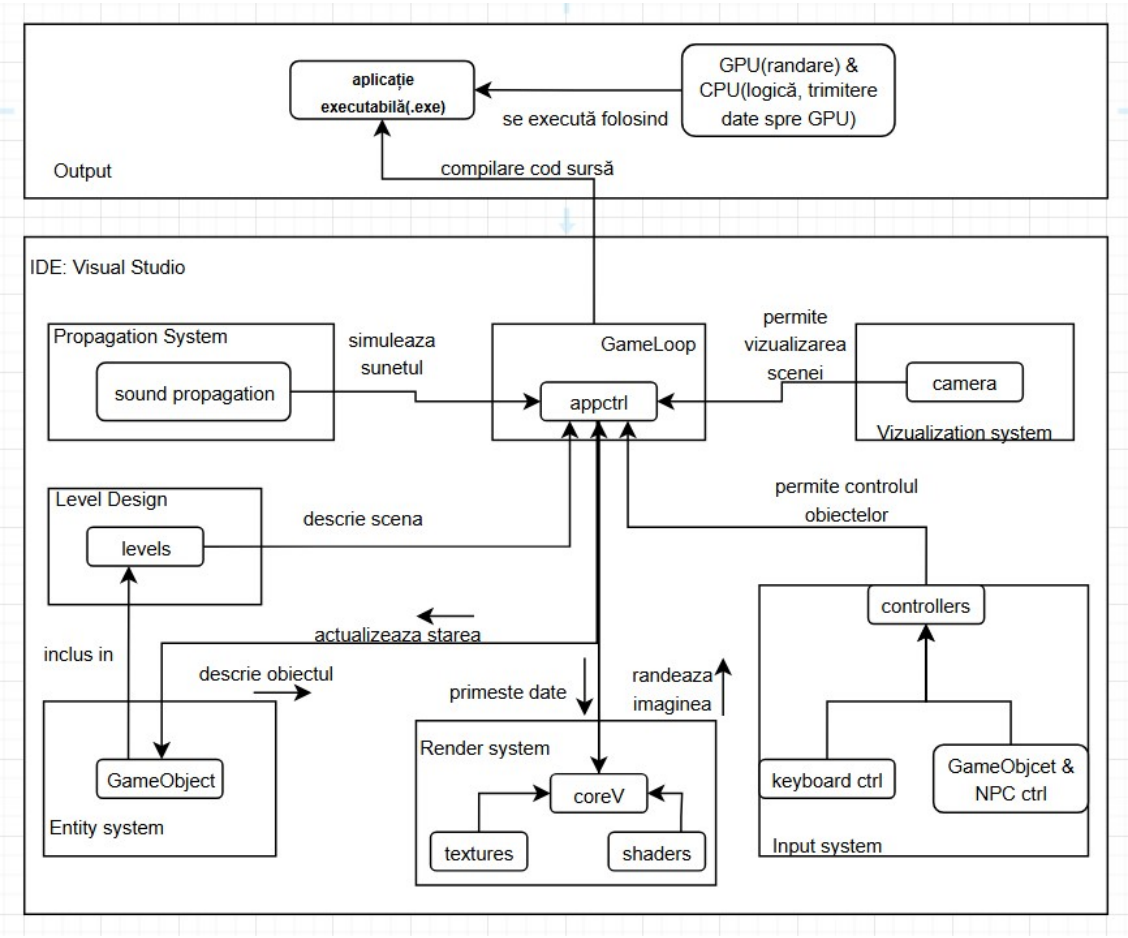


Figura 5.1: Diagrama arhitecturală

5.2. Sistemul de randare - Render System

Sistemul de randare este componenta care realizează imaginile grafice pe baza datelor primite de la componenta GameLoop 5.8. Este implementat după aplicația descrisă în prima parte a acestui capitol, însă în cadrul proiectului au fost efectuate restructurări.

5.2.1. coreV

Componenta internă a sistemului de randare, coreV (prescurtare de la core Vulkan - miezul Vulkan) reprezintă gruparea tuturor claselor care alcătuiesc dependințele necesare de care Vulkan API are nevoie pentru a furniza imaginile care vor fi afișate pe ecran și de a face trecerea de la datele logice la reprezentarea grafică a acestora.

Făcând o paralelă cu diagrama 4.3, coreV prezentat aici descrie implementarea propriu-zisă a modulului de randare. Aceasta componentă va fi detaliată în continuare, prin explicații relevante pentru înțelegerea funcționării sale.

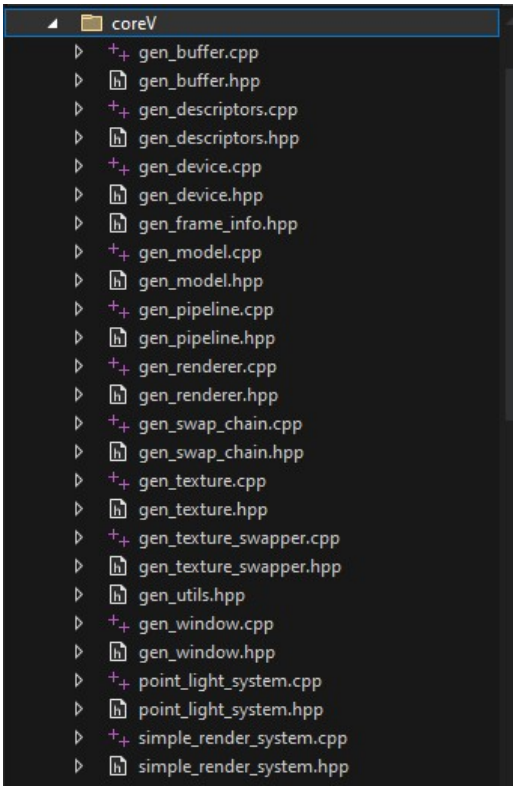


Figura 5.2: Pachetul coreV

Figura 5.2 prezintă structura pachetului coreV și multitudinea claselor pe care acesta le implementează, întrucât diagrama claselor devine complexă și greu de înțeles. În aceasta subsecțiune se vor detalia doar clasele mai importante, împreună cu diagramele de clasa a acestora.

Fundația sistemului de randare - Clasa GenDevice(gen_device.cpp\hpp)

Are rolul de a automatiza și organiza procesul complex de inițializare a contextului Vulkan. Prin această clasă se asigură alocarea corectă a resurselor Vulkan, cât și eliberarea lor din memorie în mod controlat la finalul ciclului de viață.

Astfel, oferă un strat de abstractizare prin intermediul clasei care implementează funcționalitățile ilustrate în diagrama de clasă din figura 5.3 și care sunt detaliate în cele ce urmează:

- Inițializarea instantei Vulkan - funcția `createInstance()`
  Se realizează prin intermediul structurii `VkInstance` în care se configurează informațiile despre aplicație, extensiile necesare(exemplu suportul pentru ferestre) și *validation layers*. De menționat este faptul că *validation layers*(straturi de validare) sunt mecanismul de debug pe care Vulkan API îl oferă dezvoltatorilor ce verifică fiecare parametru care interacționează cu funcțiile specifice API-ului și alocarea memoriei,

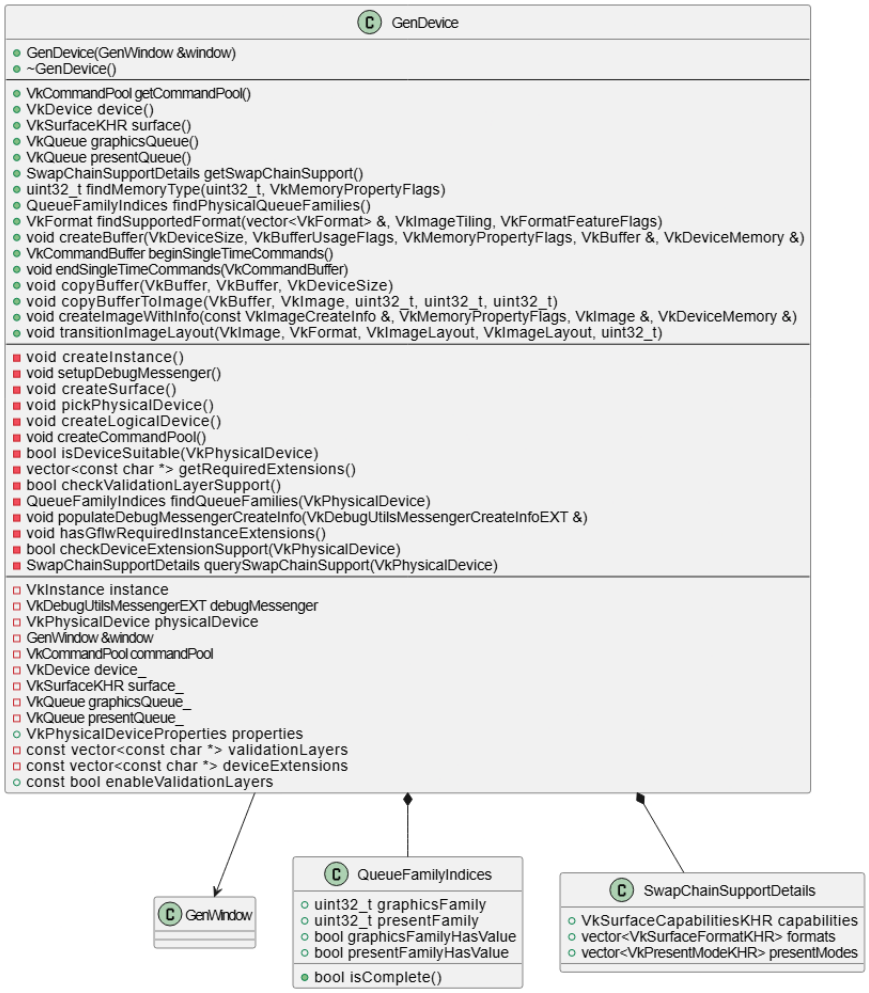


Figura 5.3: Diagrama clasei GenDevice.

urmărind să ofere mesaje detaliate în caz de eroare, fapt care simplifica procesul de depanare al aplicațiilor comparativ cu OpenGl și Direct3D[11].

- Activarea sistemului de validare
  Sistemul de validare prezentat mai sus este un sistem opțional pe care dezvoltatorul poate opta să îl folosească doar în modul de depanare, deoarece acesta consumă, în aplicația finală, mai multe resurse pentru verificarea tuturor parametrilor și urmărirea memoriei.
- Crearea suprafeței de afișare
  Pentru a realiza conexiunea dintre fereastra de afișare alocată de către sistem, funcția `createSurface()` se folosește de biblioteca GLFW pentru configurarea structurii `VkSurface` și face parte din clasa `GenWindow`.
- Selectarea dispozitivului fizic
  Prin intermediul funcției `pickPhysicalDevice()`, se face o parcurgere a listei de plăci grafice disponibile și se selectează conform unui criteriu ales placa video pe care motorul o va folosi (în funcție este aleasă placa care oferă suport complet pentru cerințele motorului)
- Crearea dispozitivului logic și obținerea cozilor
  Odată ce s-a selectat un GPU, clasa creează un dispozitiv logic prin care game engine-ul poate trimite comenzi către placa video, configurând structura `VkDevice`.

Ținând cont de faptul că orice comanda trebuie trimisa unei cozi, funcția realizează și crearea cozilor: una pentru comenzi grafice și una pentru prezentarea imaginilor pe ecran.

- Crearea unui pool de comenzi
  Acesta este utilizat pentru a aloca structuri `VkCmommandBuffer` pentru a înregistra comenzile grafice.
- Funcții auxiliare pentru resurse grafice
  Clasa `GenDevice` pune la dispoziție un set extins de funcții auxiliare care ajută la crearea și manipularea resurselor grafice.
  - `createBuffer()`, `copyBuffer()`: sunt utilizate pentru buffer-e de tip vertex, index sau staging.
  - `createImageWithInfo()`, `transitionImageLayout()`, `copyBufferToImage()`: sunt funcții care încarcă texturi sau care generează imagini în afara ecranului.
  - `findMemoryType()`, `findSupportedFormat()`: au rolul de a interoga caracteristicile dispozitivului pentru a selecta tipurile de memorie sau formatele compatibile.

Toate aceste operații descriu primele 3 etape care fac posibilă afișarea de imagini pe ecran, conform pașilor descriși în subsecțiunea 4.4.1.

Infrastructura Vulkan - Clasele GenRenderer și GenSwapChain

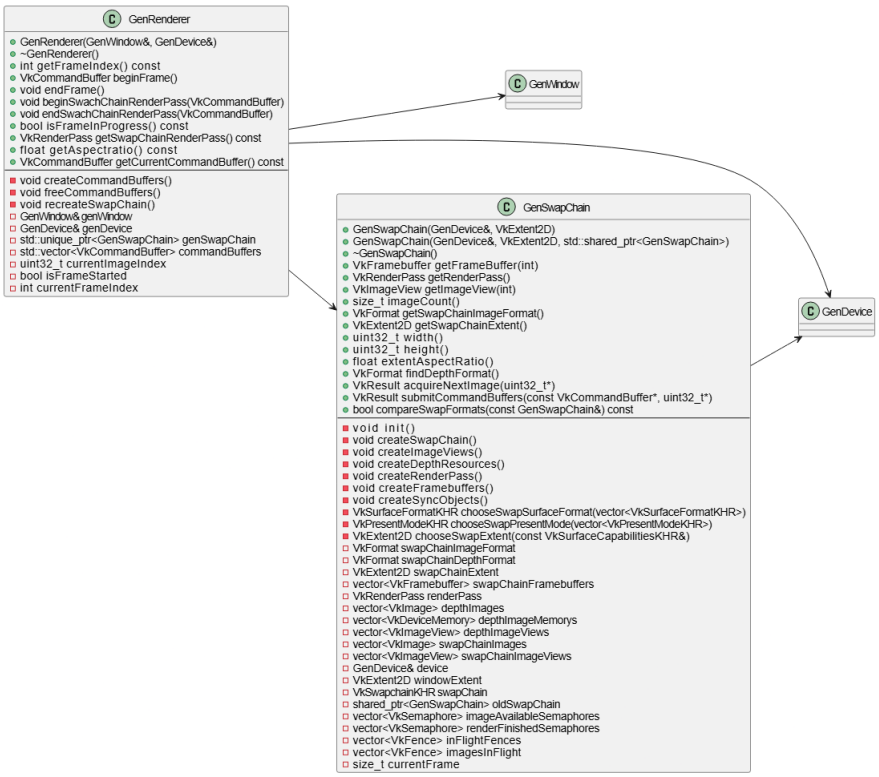


Figura 5.4: Diagrama claselor din infrastructura Vulkan.

Clasa `GenRenderer` este cea care gestionează procesul de creare al unui cadru utilizat în bucla principală. Astfel, oferă o abstractizare asupra funcțiilor care inițiază și încheie perioada în care pot fi înregistrate comenzile de randare corespunzătoare cadrului respectiv. De asemenea, clasele `GenSwapChain` și `Gendevice` interacționează direct cu

aceasta, deoarece comenzile de randare sunt trimise către GPU prin dispozitivul logic, iar datele despre imagine provin din swap chain.

Funcțiile pricipale implementate sunt:

- Crearea și refacerea SwapChain-ului - `recreateSwapChain()`
  Deoarece swap chain-ul deține imaginile ce urmează să fie afișate, o simplă redimensionare a ferestrei schimbă proprietățile imaginii, cel puțin în ceea ce privește înălțimea și lățimea. Astfel, configurația anterioară prin care erau create imaginile nu mai corespunde cu starea actuala a ferestrei aplicației.
  Pentru a transmite aceste schimbări ale caracteristicilor imaginii, această funcție are scopul de a recrea și actualiza configurația swapchain-ului. Totodată, funcția este apelata și la inițializare, din aceleași considerente.
- Începerea și încheierea unui cadru - `beginFrame(), endFrame()`
  În această etapă, clasa controlează momentul în care se obține o nouă imagine(= echivalent cu o pânză alba pentru pictura) din swapchain și se înregistrează comenzile de randare. Apoi, odată ce au fost înregistrate toate comenzile, sunt transmise către GPU la finalul acestei perioade.

Celelalte funcții implementate de această clasă sunt funcții utilitare și funcții care aduc un nivel de detaliu ridicat în privința modului de funcționare a API-ului Vulkan și care sunt regăsite în [11].

Pentru gestionarea elementului care permite afișarea imaginilor pe ecran, clasa `GenSwapChain` este responsabilă cu implementarea funcțiilor necesare. În contextul Vulkan, *swap chain-ul* este o coada de imagini care sunt randate de GPU și afișate de sistemul de ferestre. Astfel, este necesara implementarea funcțiilor care să facă posibila configurarea și utilizarea acestor imagini.

- Inițializarea swap chain-ului - `init()`
  Această funcție oferă o imagine de ansamblu asupra cerințelor necesare creări unui swap chain, prin gruparea mai multor pași, fiind *constructor-ul* clasei.
  - `createSwapChain()`: funcție care preia toate datele necesare, precum formatul suprafeței, modul de prezentare(exemplu V-Sync) pentru a configura structura `VkSwapChain` a sistemului.
  - `createImageViews()`: funcția are rolul de a seta câmpurile structuri `VkImage` pentru a obține "imaginea" peste care se va randa.
  - `createRenderPass()`: funcție complexă care configurează modul de utilizare a imaginilor, oferind detalii în privința culorilor folosite și a altor elemente.
  - `createDepthResources()`: se atașează și se creează imaginile de adâncime pentru buffer-ul Z. Imaginile de adâncime și buffer-ul Z sunt folosite pentru a determina ordinea în care obiectele din scena vor fi vizibile în funcție de eventuala suprapunere a acestora rezultata de perspectiva camerei.
  - `createFramebuffers()`: grupează din punct de vedere al memoriei toate structurile descrise mai sus pentru a asocia fiecare imagine cu toate caracteristicile sale.
  - `createSyncObjects()`: oferă mecanismele de sincronizare care controlează ordinea corecta de execuție intre CPU și GPU în cardul fiecărui frame. Deoarece în Vulkan randarea este asincronă, CPU-ul poate trimite comenzi către GPU, timp în care acesta încă procesează un cadru anterior. Aceste mecanisme previn accesarea simultană a aceleași resurse și asigură folosirea imagini la momentul potrivit.

Implementarea celor două clase oferă o infrastructură de bază pentru procesul de

randare în care cadrele randate de GPU sunt construite conform formatelor și dimensiunilor corespunzătoare ferestrei aplicației, iar afișarea lor se face în mod sincronizat.

Gestionarea resurselor în memorie

Clasele `GenDescriptorSetLayout`, `GenDescriptorPool` și `GenDescriptorWriter` gestionează modul în care datele precum matricile de transformare, luminile sau texturile sunt transmise către shadere. Aceste clase abstractizează concepte *low-level* (de nivel scăzut) din Vulkan, precum layout-uri ale seturilor de descriptor, alocarea acestora în pool-uri speciale și actualizarea corespunzătoare a datelor acestora. În figura 5.5 este prezentată diagrama de clase a acestora.

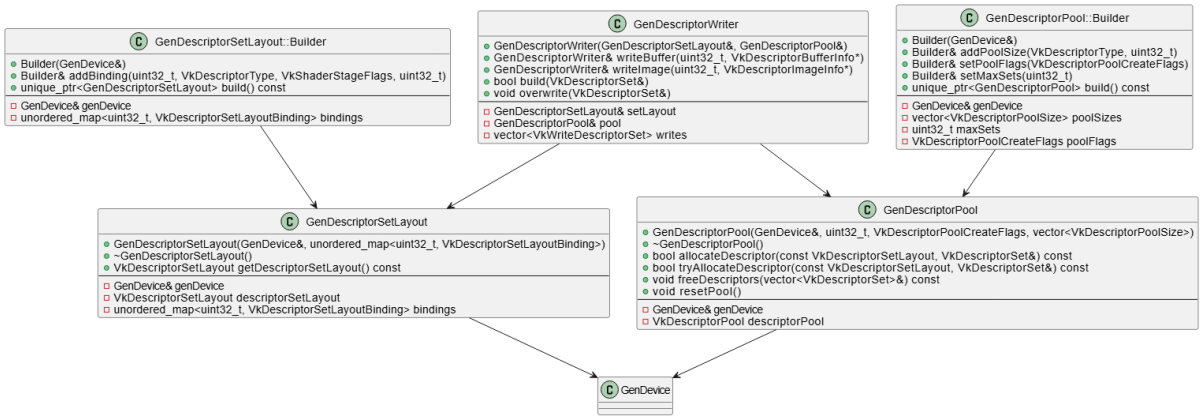


Figura 5.5: Diagrama de claselor pentru gestionarea memoriei.

Este important de menționat modul general în care descriptor pool-ul funcționează. API-ul Vulkan introduce un nou nivel de control explicit în privința transmiterii memoriei din bufferele uniforme către pipeline-ul grafic. Astfel fiecare parte distincta de memorie din bufferul uniform este grupată în seturi de descriptori. Pipeline-ul grafic va tine cont de aceste seturi și va cere aplicației așezarea specifica de memorie prin folosirea `VkDescriptorSetLayout`. Dar, și pentru alocarea acestor așezări(layouts) este nevoie de un obiect `VkDescriptorPool`. Acest proces complex oferă o performanță ridicata în contextul actualizării rapide a bufferelor uniforme și procesarea acestora de către shadere.

Tot în contextul gestionării resurselor fac parte și texturile. Clasele `GenTexture` și `TextureSwapper` reprezintă contribuția personală adusă sistemului de randare pentru aplicarea texturilor pe obiecte cu scopul de a îmbunătăți aspectul vizual, întrucât renderer-ul construit prin tutorial nu realiza acesta funcționalitate. Figura 5.6 ilustrează diagrama de clase pentru gestionarea texturilor.

Clasa `GenTexture` oferă funcțiile necesare pentru încărcarea din memoria sistemului a texturilor, transferul și pregătirea acestora pentru utilizare în pipeline-ul grafic Vulkan. Astfel, se automatizează procesul de alocare și distrugere a resurselor Vulkan asociate texturilor, având ca și referința pași descriși în capitolul *Texture mappping* din [11], adaptați la arhitectura aplicației.

In cele ce urmează sunt prezentate detaliile de implementare a funcțiilor componente acestei clase.

- `createTextureImage()`: în aceasta funcție, datele despre imagini(texturile 2D) sunt încărcate cu ajutorul librăriei `stb_image` utilizând funcția `stbi_load(...)`. Astfel sunt preluate caracteristicile precum lățimea, înălțimea imagini și canalele

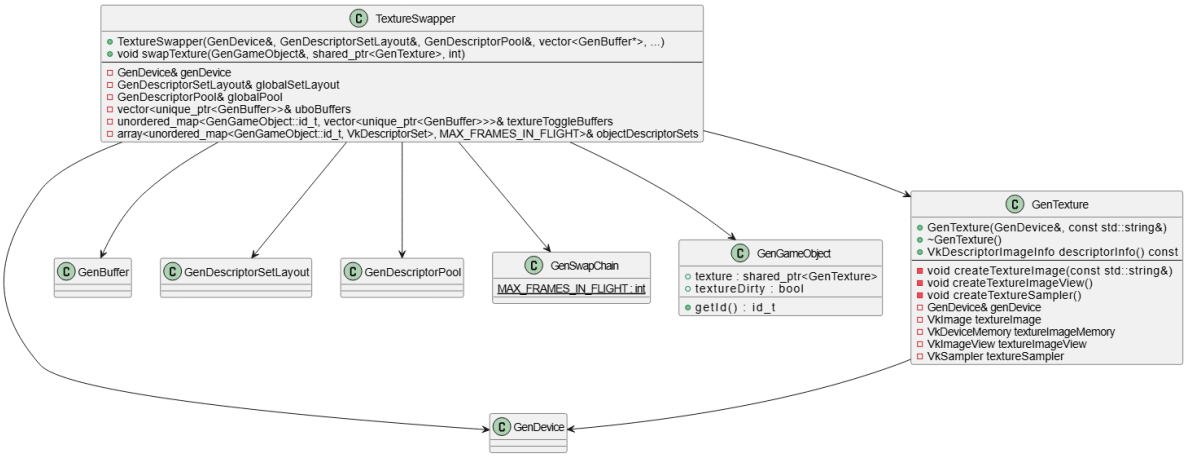


Figura 5.6: Diagrama claselor pentru gestionarea texturilor

de culoare. Folosind aceste date se creează buffer-ul de memorie care va transmite obiectul `VkImage` în pipeline-ul și memoria GPU-ului, urmând să fie populat cu structura de configurare Vulkan a unei imagini de acest tip.

- `createTextureSampler()`: Având în vedere faptul că accesarea texturii se realizează într-un sistem de coordonate normalizate între [0, 1] în cadrul shaderelor, textura își poate pierde calitatea prin scalarea efectuata, astfel conceptul de sampler pe care funcția îl implementează are scopul de a aplica filtrele necesare pentru creșterea calității texturi care ajunge la shadere. Funcția fiind un pas necesar în procesul de creare al texturilor.

Clasa `TextureSwapper` reprezintă un mecanism care extinde flexibilitatea sistemului de randare, care face posibilă schimbarea texturilor la runtime(în timp ce aplicația se execută în timp real) pentru orice obiect grafic.

- `swapTexture(...)`: funcția principală din aceasta clasa realizează schimbarea texturii obiectului în cadrul unui frame. Primul pas este de a atribui noua textura a obiectului și a marca faptul ca acesta are nevoie de reconstruirea `DescriptorSetului` asociat prin atributul `textureDirty = true`.
  Următori pași reprezintă preluarea bufferelor și a descriptor seturilor care conțin vechea textura a obiectului, transmise prin referință și actualizarea corespunzătoare a acestora.
  La final se indica faptul că textura a fost actualizată prin setarea atributului `textureDirty` înapoi la `false`.

Pipeline-ul grafic

Totalitatea operațiunilor pe care GPU-ul le efectuează pentru a colora pixeli de pe ecran, folosind datele punctelor care definesc obiectele, transformări matematice pentru proiectarea acestora pe ecran, interpolări pentru calculul culorilor reprezintă pipeline-ul grafic. Această ordine bine definită are la rândul ei stadii care pot fi programabile de către utilizator și stadii care sunt fixe și reprezinta funcționalități la nivel de hardware pe care GPU-ul le executa și sunt specifice acestuia. În figura 5.7 sunt ilustrate clasele prin care este implementat pipeline-ul.

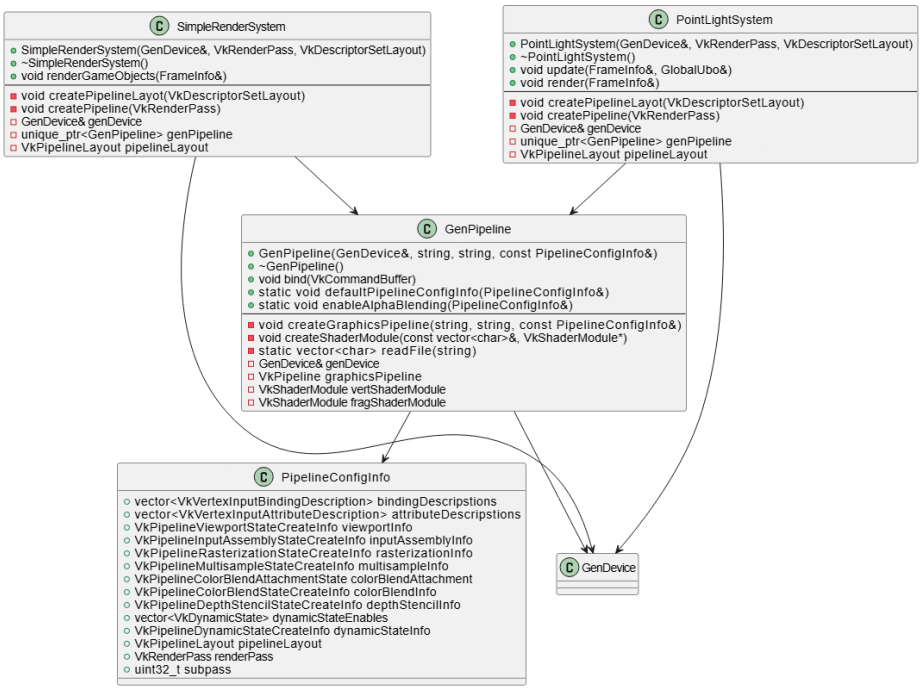


Figura 5.7: Diagrama claselor din pipeline-ul grafic.

Clasa `GenPipeline` este responsabilă pentru crearea pașilor care definesc secventa de operații pe care GPU-ul o are de urmat și integrarea operațiunilor programabile de către dezvoltator în cadrul acesteia.

Pentru a ilustra vizual conceptul pipeline-ului grafic se folosește figura 5.8 preluată din [11] în care se codifică prin culoarea verde stadiile fixe care permit doar schimbarea unor configurații specifice, iar prin culoarea portocalie sunt codificate, stadiile pe care dezvoltatorul le poate programa și reintegra în pipeline.

Astfel, funcțiile care se regăsesc în această clasă sunt responsabile pentru:

- Crearea modulelor shader și citirea codului shaderelor `readFile(...)` și `createShaderModule(...)`:
  Aceste funcții au rolul de a citi codul shaderelor compilat în cod binar SPIR-V și de a transforma aceste date în obiectele de tip `VkShaderModule` prin structurile de configurare specifice. Diagrama din figura 5.9 ilustrează modelul abstract al funcției.
- Crearea propriu-zisa a pipeline-ului grafic - `createGraphicsPipeline()`:
  In această funcție se automatizează configurarea fiecărui stadiu al pipeline-ului printr-o serie complexă de structuri de configurare specifice pipeline-ului Vulkan, aspecte ce contribuie la complexitatea generală a acestuia. Rezultatul funcției fiind

pipeline-ul dorit de către dezvoltator.

- Componente adiționale:
  Clasa oferă și funcția `defaultPipelineConfigInfo()` care setează, pentru stările importante ale pipeline-ului, valori implicite astfel, acoperind stadiile fixe. De asemenea, prin funcția `enableAplhaBlending(...)` care oferă posibilitatea de a reda transparenta în randare utilizând factorul alpha.
- Folosirea pipeline-ului:
  Pana în momentul de față pipeline-ul are pregătite toate stadiile, însa ultima componentă care rămâne de adăugat este cea care deține comenzile de randare. După ce acestea sunt înregistrate, pipeline-ul este activat și folosit de GPU.

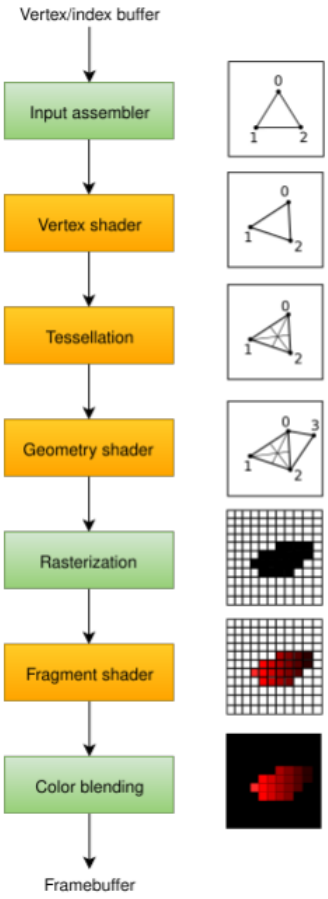


Figura 5.8: Imagine de ansamblu a pipeline-ului grafic preluata din [11]

Clasele `SimpleRenderSystem` și `PointLightSystem` sunt cele care se ocupă de procesul de randare a obiectelor 3D din scena respectiv randarea luminilor punctiforme, folosind pipeline-ul grafic dedicat fiecărui scop. Prin intermediul acestora se creează aceste pipeline-uri folosind funcțiile din clasa `GenPipeline` oferindu-le toate datele de care acestea au nevoie și de asemenea, sunt responsabile pentru definirea comenzilor de randare pe care pipeline-ul final le va folosi.

Procesul pe care ambele clase îl implementează este asemănător deoarece la baza și elementele de lumină sunt obiecte 3D pe care `SimpleRenderSystem` le-ar putea randa, însa luminile sunt coordonate de către shadere diferite ceea ce, în contextul Vulkan, necesită crearea unui pipeline dedicat.

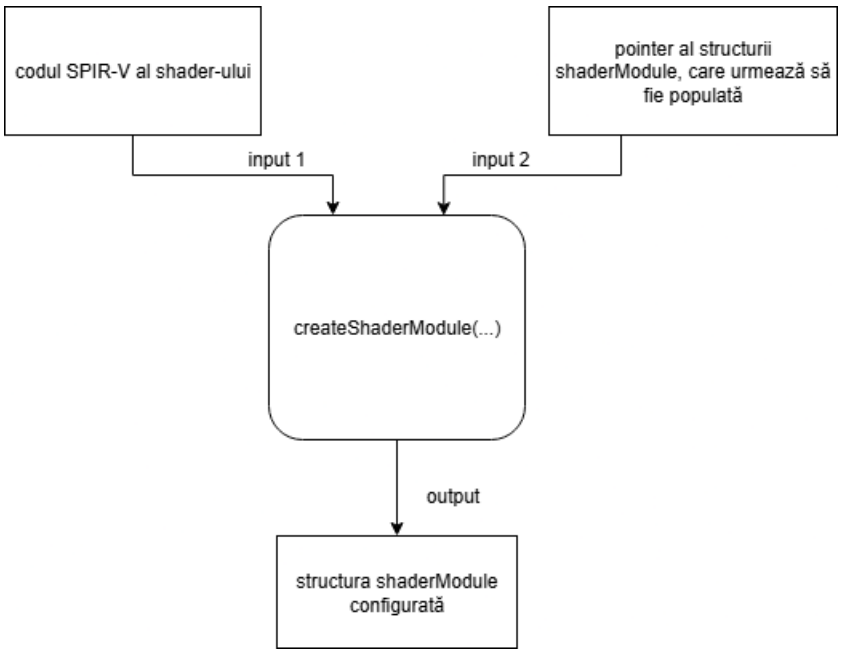


Figura 5.9: Model abstractizat al funcției `createShaderModule(...)`

Astfel, procesul este următorul:

- Se inițializează pipeline-ul și layout-ul datelor asociate obiectelor și se încarcă în pipeline shaderele corespunzătoare provenite din fișierele .spv.
- Se activează pipeline-ul creat în funcțiile de randare `renderGameObject()`, respectiv `render()`
- Se transmit datele specifice obiectelor spre shadere prin utilizarea descriptor set-urilor și a Push Constants. Pentru claritate, push constants sunt structuri care conțin date constante și sunt transmise către shadere într-un mod mult mai rapid și eficient, dar nu dispun de memorie foarte mare având un volum de date, ce poate fi transmis, limitat în comparație cu alte metode de transmitere a datelor în acest scop. [11].
- In final se executa randarea propriu-zisă.

Modelul geometric al obiectelor 3D

După cum s-a prezentat la nivel teoretic în capitolul 4, modelele obiectelor 3D sunt descrise în spatii 3D individuale prin intermediul fișierelor de tip `.obj`.

Pentru a clarifica câteva aspecte legate de acest mod de reprezentare, trebuie evidențiat faptul că într-o aplicație de randare, inclusiv un motor de jocuri, obiectele 3D sunt grupate astfel: coordonatele vârfurilor ce compun geometria obiectului(3 vrafuri - primul triunghi, alte 3 varfuri - al doilea triunghi, pana când se definește întreg modelul), culoarea fiecărui vârf, normala suprafeței definite de către triunghiurile care compun obiectul și coordonatele *UV* care descriu cum se proiectează textura pe obiect, toate sunt într-o zona continua de memorie, numita *vertexBuffer*, care este trimisa către *vertex shader*. De asemenea, pot apărea indici care specifica pentru fiecare vârf apartenenta la triunghiurile definitorii ale formei 3D cu scopul de a reduce redundanta în definirea vârfurilor. Trecerea prin zona de memorie și utilizarea datelor se face specificând poziția și aranjarea diferitelor tipuri de date din cadrul acesteia.

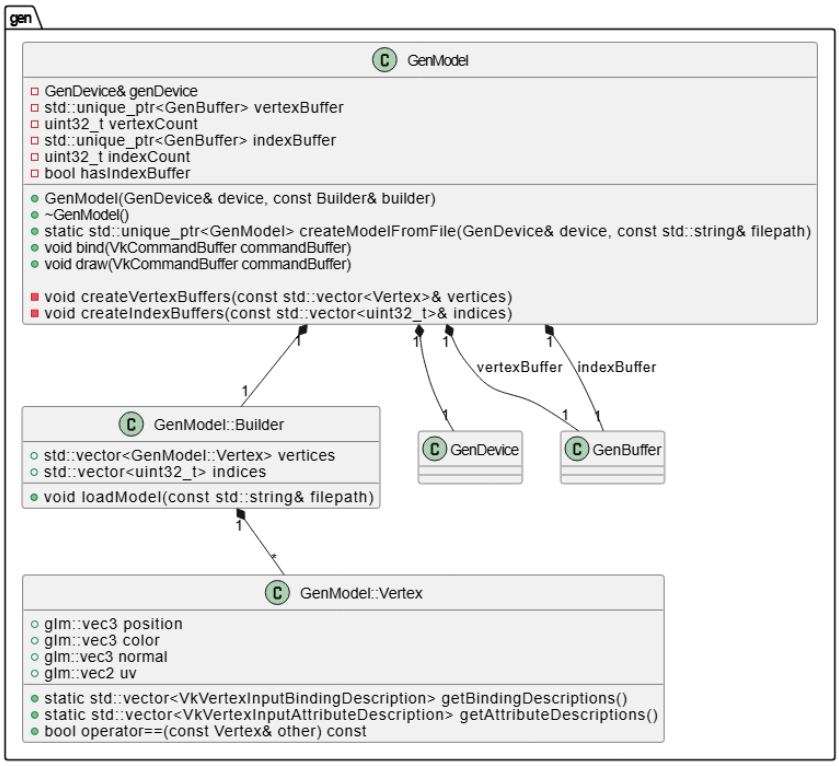


Figura 5.10: Diagrama de clase a modulului geometric al obiectelor 3D.

Astfel, figura 5.10 prezintă diagrama clasei `GenModel`, prin intermediul căreia sunt implementate funcțiile ce realizează procesul de grupare și transmitere mai departe a tuturor datelor prezentate mai sus și specifică modul de trasare al obiectelor în pipeline-ul grafic. Clasa se va folosi de biblioteca `tinyobjloader` tinyobjloader pentru a parsa datele din respectivele fișiere.

- Funcția `loadModel()` este responsabilă cu preluarea datelor din fișier și transformarea acestora într-un set unic de vârfuri împreuna cu indici de apartenență a acestora la triunghiurile care definesc suprafețele modelului. Chiar dacă obiectul nu dispune de indici corespunzători, iar acesta este definit folosind vârfuri duplicate, funcția automatizează procesul de eliminare al duplicatelor, motivul fiind optimizarea timpului de execuție de către GPU pentru calculul matricilor de transformare pentru fiecare vârf și reducerea volumului de date.
  Unicitatea vârfurilor este realizata prin introducerea unei noi definiri hash în cadrul `std::unordered_map` care face posibila definirea structurilor de tipul $< Vertex, uint32\_t >$.
- Metodele `bind(...)` și `draw(...)` realizează conexiunea dintre model și comenzile de trasare Vulkan. Astfel, `bind(...)` atașează vertex buffer-ul la buffer-ul de comanda, iar funcția `draw(...)` furnizează comanda de trasare propriu-zisă.

5.2.2. **Shadere**

Parte din sistemul de randare shader-ul este codul care rulează pe placa video și folosește datele precizate anterior. Exista două tipuri de astfel de programe, care îndeplinesc funcționalități diferite: *vertex shader* - manipulează coordonatele obiectelor și efectuează transformările matematice necesare pentru a pune în spațiul scenei aceste obiecte; *fragment shader* - realizează colorarea propriu-zisa a modelelor și se aplica pe toți

pixeli care fac parte din contextul ferestrei aplicației dându-le culoarea corespunzătoare. Output-ul de la codul vertex este folosit ca date de intrare pentru codul fragment.

Detaliile referitoare la așezarea în scena și spațiul scenei vor fi detaliate în secțiunea 5.5.

Sistemul de randare *coreV* se folosește de 4 shadere grupate în perechi de cate 2(vertex și fragment). `simple_shader.vert` și `simple_shader.frag` sunt folosite pentru obiectele obisnuite, iar `point_light.vert` și `point_light.frag` se folosesc pentru a reprezenta vizual obiectele de tip lumină.

Vertex Shader - simple_shader.vert

Datele de intrare pe care acest shader le folosește sunt descrise de sintaxa `layout(location = X) in`, specificând poziția fiecărui atribut în bufferul de date primit și reprezintă date referitoare la modelul obiectelor.

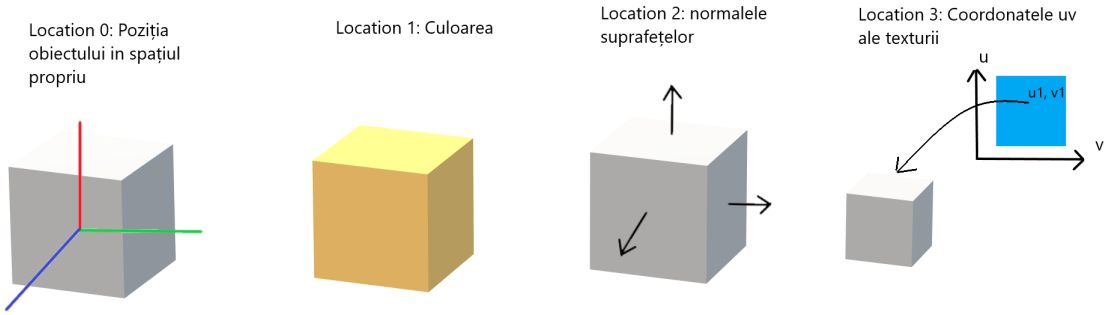


Figura 5.11: Datele de intrare ale simple_shader.vert

Datele globale care reprezintă definirea scenei, direcția de privire a camerei, matricea care proiectează obiectele pe planul ecranului, luminile din scena și matricile de transformare care trebuie aplicate obiectelor sunt transmise prin uniformele globale, structuri adiționale și push constants. Acestea sunt folosite pentru a aduce obiectele din spațiul propriu în spațiul scenei și al ecranului.

Astfel, prin funcția `main()` GPU-ul procesează fiecare vârf al modelului astfel:

1. Se aduce poziția vârfului din spațiul local în spațiul lumii prin `modelMatrix`.
2. Din spațiul lumii, folosind matricile `view` și `projection`, vârful se aduce în spațiul camerei care reprezintă poziția finala pe ecran.
3. Rezultatele calculelor sunt apoi trimise către shader-ul de fragmente prin variabilele declarate folosind cuvântul `out`.

După această etapă sunt pregătite toate datele necesare pentru efectuarea iluminării și texturării la nivel de pixel.

Fragment Shader - simple_shader.frag

Utilizând valorile interpolate intre vârfurile triunghiurilor, cele obținute ca rezultat în pasul 3. descris anterior , shader-ul de fragmente calculează culoarea finala a fiecărui pixel(fragment) primit ca intrare de la GPU în contextul ferestrei aplicației, ținând cont de iluminare și material(dacă este cazul).

Pentru alegerea culorii pe suprafața obiectului, shader-ul funcționează în doua moduri:

- Utilizează culoarea texturii dacă variabila `useTexture` este setata pe 1
- Folosește culoare transmisă de vertex shader în caz contrar.

Alegere controlata dinamic prin descriptor set-ul `TextureToggle`.

Modelul de iluminare folosit în shader este Blinn-Phong, un model clasic care aproximează reflexia luminii provenite de la o sursa punctiforma, precum un bec, oferind un aspect vizual realist.

Astfel, acest model este descris în cele ce urmează utilizând și parți relevante din codul fragment shader-ului:

- Lumina ambientală:
  Este componenta care oferă o lumina de baza constant cu rolul de a simula reflexia indirectă a lumini din mediul înconjurător.
- Lumina difuză: Redă intensitatea în culoare pe care suprafața o primește de la lumina și este influențata de unghiul dintre normala suprafeței obiectului și direcția spre camera virtuală.
- Lumina specular: Are rolul de a simula reflexia lucioasă a obiectelor de diferite materiale oferind realism vizual scenei.

În cadrul implementării, cu cat este mai mic unghiul dintre normala și vectorul `halfAnghle`, reflexia specular care apare pe obiect este mai intensă. Rolul exponentului din calculul final al luminii speculare determina cât de calară este pata respectivă. Reprezentarea vizuală a acestui model se regăsește în figura 5.12.

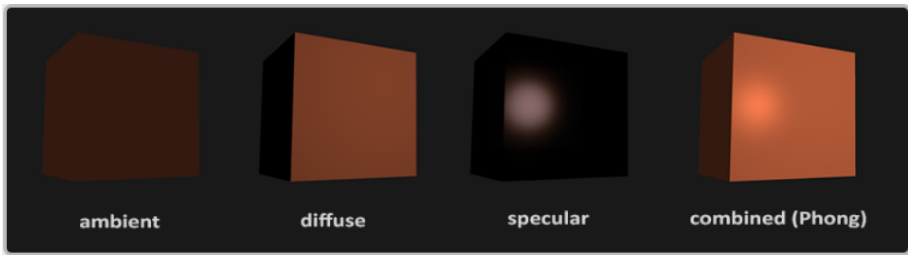


Figura 5.12: Componentele de lumină ale modelului Blinn-Phong. [15]

Culoarea finala a pixelului este obținuta prin înmulțirea culorii suprafeței sau a culorii texturii cu suma luminilor calculate de modelul de iluminare Blinn-Phong. Această culoare va fi transmisă către frameBuffer și va fi afișata pe ecran.

Următoarele două shadere sunt responsabile pentru vizualizarea punctelor de lumina punctiformă în scena folosind un disc 2D care este mereu îndreptat spre camera. Astfel, dezvoltatorul va avea mereu un punct de referință asupra poziționării surselor de lumina în scena, îmbunătățind procesul de așezare în scena a obiectelor.

Vertex Shader - point_light.vert

Discul 2D este format dintr-un pătrat care la rândul sau este format din 2 triunghiuri, pentru a oferi aspectul circular dorit, colțurile sunt rotunjite printr-un efect de atenuare folosind canalul alpha al culorilor.

Astfel, se definesc la începutul shader-ului triunghiurile care alcătuiesc pătratul prin specificarea vârfurilor componente. Figura 5.13 ilustrează coordonatele figurii.

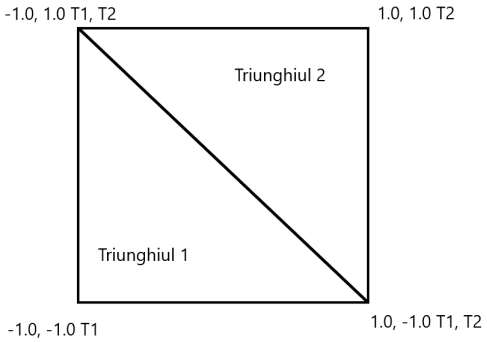


Figura 5.13: Vârfurile figurii geometrice care definesc lumina punctiformă.

Similar cu procesul de transmitere a datelor adiționale, precizat anterior în etapa de vertex, se transmite poziția din spațiul local al figurii, culoarea și raza discului prin push constants. Figura 5.14 exemplifică așezarea în memorie a acestor elemente.

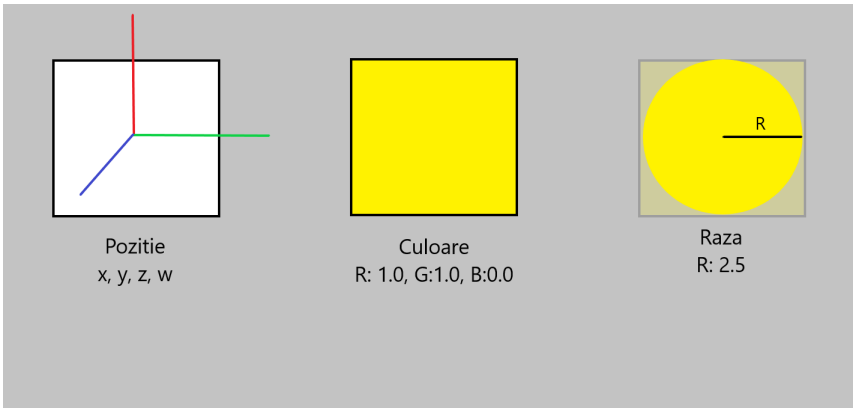


Figura 5.14: Reprezentarea memoriei Push Constant pentru lumina punctiformă.

Pentru a poziționa discul astfel încât să fie orientat mereu spre camera de vizualizare se extrag axele *Right* și *Up* ale camerei din matricea `view` cu scopul de a construi un plan perpendicular cu camera pe care va fi randat discul. Figura 5.15 ilustrează procesul descris.

Fragment Shader - point_shader.frag

În cadrul acestui shader se obține colorarea și definirea discului care reprezintă lumina punctiformă. Astfel, se elimină fragmente care nu aparțin discului. Apoi culoarea finala a pixelului este obținuta în funcție de factorul de transparentă influențat de distanta față de centru(cos(0) - în centru => aplha = 1.0; cos(π) = -1 => alpha = 0.0), dând discului aspectul vizual al unei aureole. Figura 5.16 ilustrează forma finală descrisă.

Componentele descrise în secțiunile 5.2.1 si 5.2.2 reprezintă implementarea sistemului de randare al motorului de jocuri și este cel care comunica direct cu Game Loop-ul 5.8 și indirect cu Sistemul de control5.6, Sistemul de Vizualizare5.5 și Sistemul Entităților5.3 pentru a afișa în timp real imaginile randate pe ecran.

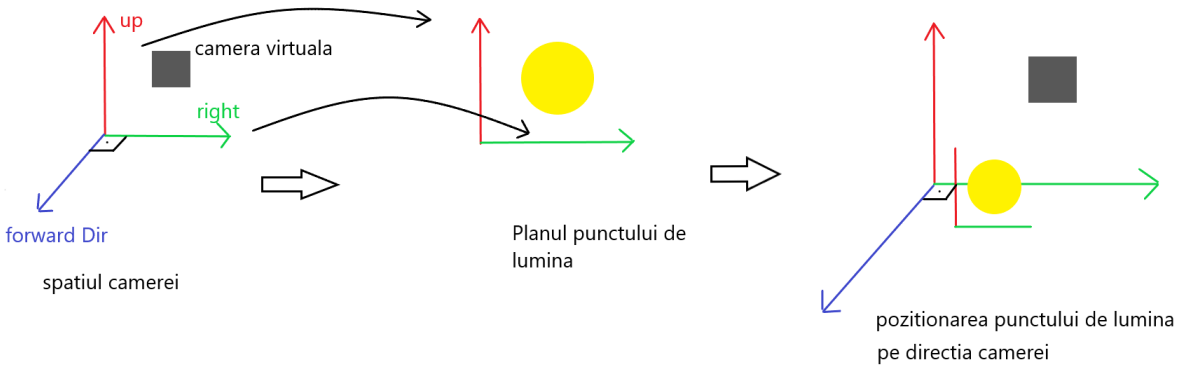


Figura 5.15: Pozitionarea punctului de lumina perpendicular cu planul camerei.

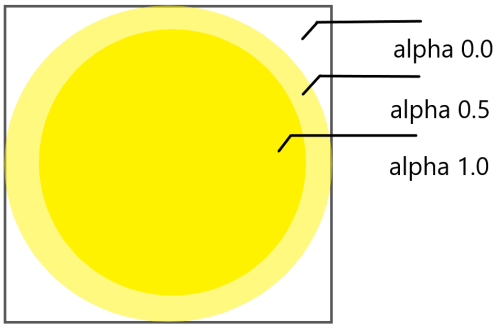


Figura 5.16: Aspectul final al unei lumini punctiforme.

5.3. Sistemul entităților - Entity System

Sistemul de randare 5.2 se concentrează exclusiv pe crearea imaginilor care urmează sa fie afișate pe ecran, astfel, clasa `GenModel` este doar responsabilă pentru gruparea și încapsularea datelor referitoare la modele 3D utilizate de GPU, dar nu oferă nici o funcționalitate logică sau comportamentală pe care aceste modele să le implementeze. Cu alte cuvinte, clasa `GenModel` descrie doar forma vizuala a unui obiect într-o scena statică, fără a descrie modul prin care acesta interacționează cu lumea virtuală.

Pentru a extinde modelul de date vizuale cu un set de funcționalități logice care sa ofere fiecărui obiect o identitate proprie și un rol bine definit, a fost introdusă clasa `GenGameObjcet`. Această clasă reprezintă, în esența, un obiect generic din scena și permite descrierea unei game variate de componente pe care obiectul le poate deține, în funcție de scopul sau logica sa în aplicație.

În implementarea clasei se folosesc concepte preluate din [Cap. 16 pp.1043-1047][2] pentru a reprezenta atributele obiectelor în mod eficient, evitând crearea ierarhiilor complexe și rigide care ar putea îngreuna extensibilitatea. În acest sens, au fost introduse componente pe care obiectul le poate sau nu însuși, în funcție de nevoile programatorului, și de asemenea obiectele create vor fi numerotate print-un id pentru a facilita accesul acestora în programarea logicii de joc, precum este ilustrat in diagrama 5.17.

Pentru atribuirea eficienta a fiecărei componente unui obiect din joc(*game object*) s-au folosit doua dintre tipurile de *smart pointers* disponibile în C++ 17 și anume:

- `std::shared_ptr`(*shared_ptr reference*) pentru partajarea resurselor reutilizabile

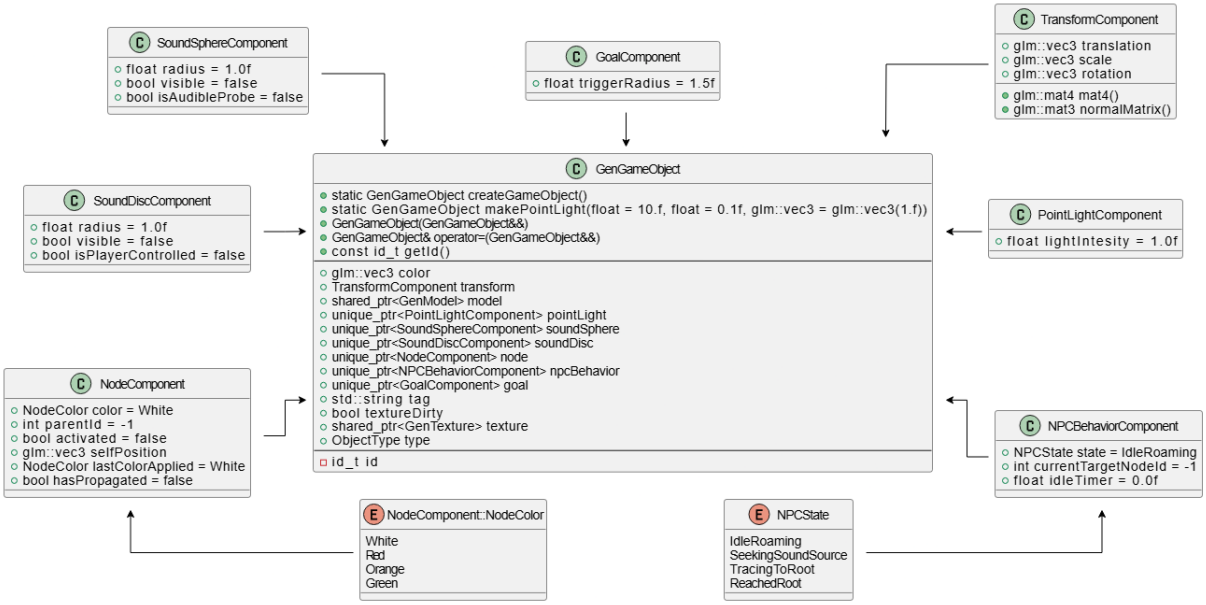


Figura 5.17: Diagrama de clase a sistemului entităților `GenGameObject`.

între obiecte cum ar fi modelele 3D.

- `std::unique_ptr`(*unique_ptr refference*) pentru componentele specifice fiecărui obiect, evitând copierea inutilă a acestor resurse în folosirea obiectelor.

In cele ce urmează se vor detalia aceste componente din implementarea clasei `GenGameObject`.

1. `std::shared_ptr<GenModel>model`: folosind clasa `GenModel`, această componentă este responsabilă pentru atribuirea unui model 3D (vaza, sfera, cub) game object-ului.
2. `TransformComponent`: este structura care încapsulează matricile de transformare alea obiectului.
3. `PointLightComponent`: componenta care diferențiază un obiect generic față de o lumină punctiformă a scenei. Această structură oferă atributul de intensitate al lumini respective.
4. `SoundDiscComponent`: definește zona circulara în planul suprafeței solului(XZ), în care un nod de sunet poate percepe sunetul provenit de la jucător, sau cea în care acesta poate acționa pentru a transmite sunetul spre alte obiecte care însușesc această componentă, simulând astfel propagarea sunetului. Programatorul poate modifica distanta zonei de acțiune prin schimbarea proprietății `radius`.
5. `NodeComponent`: este utilizată pentru a facilita crearea structurilor de tip arbore(`tree`), care definesc rețele logice reprezentative pentru zona de acoperire a unui sunet prin diverse noduri de sunet. Astfel, prin acesta componenta se oferă informații în privința:
   - poziției nodului în scena
   - id -ul nodului părinte de la care s-a propagat nodul
   - starea nodului: dacă este activ și dacă s-a propagat mai departe
   - culoarea logica prin care se oferă informația vizuală a tipului de sunet(zgomot puternic, zgomot normal, zgomot slab, nici un zgomot).

   Detaliile referitoare la modul de utilizare al acesteia se regăsesc în secțiunea 5.7.
6. `ObjcetType`: este componenta ce definește ce fel de obiect de joc reprezintă modelul

3D, fiind și componenta prin care se identifică categoria din care face parte.

7. `NPCBehaviorComponent`: prin aceasta componentă se descriu obiectele de tip NPC, care memorează nodul destinație la care trebuie sa ajungă în urma algoritmilor de căutare. De asemenea, pentru a reprezenta starea de căutare în care se afla acestea se utilizează `enum class NPCState`:
   - `IdleRoaming`: rătăcire aleatoare prin scena.
   - `SeekingSoundSource`: caută o sursă a sunetului.
   - `TracingToRoot`: urmărește activ sursa sunetului.
   - `ReachedRoot`: a ajuns la sursa sunetului.

   . Valorile din enumerației sunt folosite în algoritmii de căutare implementați și descriși în secțiunea 5.6.
8. `GoalComponent` Pentru a descrie complet un joc este nevoie de o condiție de câștig, cea prin care se oferă un scop jucătorului și face posibil progresul. Aceasta componenta se atribuie obiectelor care reprezintă obiectivul nivelului / jocului și este definită de un disc din interiorul căruia jucătorul poate interacționa cu acel obiect.

Pentru definirea completa a unei entități de joc se pot urma pașii:

- se creează modelul 3D al obiectului, folosind `GenModel::createModelFromFile(...)`
- se creează un obiect de joc: `GenGameObject::createGameObject();`
- se atribuie modelul creat
- se creează textura obiectului folosind constructorul din clasa `GenTexture` și pe urmă se atribuie la obiectul de joc
- se definesc matricile de transformare a obiectului: translație, rotație și scalare.
- se adaugă componenta care adaugă funcționalitatea specifica obiectului

Folosind aceste proprietăți motorul de jocuri face posibilă definirea unor scene complexe, în care poate fi integrate sisteme de reguli avansate.

5.4. Construcția nivelului - Level Design

Cu toate ca sistemul de entități5.3 oferă definirea unor entități cu proprietăți variate care interacționează în mod unic cu modul de joc, acestea trebuie așezate în scenă și configurate prin atribuirea componentelor specifice fiecăruia.

Pentru a accelera procesul de construire al lumii virtuale și popularea acesteia cu entitățile de joc corespunzătoare(`GenGameObjects`), modulul *Level Design* se ocupă de automatizarea procesului de creare și definire a nivelurilor de joc. Diagrama din figura 5.18 ilustrează interacțiunea dintre clasele pentru nivel și așezarea automata a nodurilor în scenă.

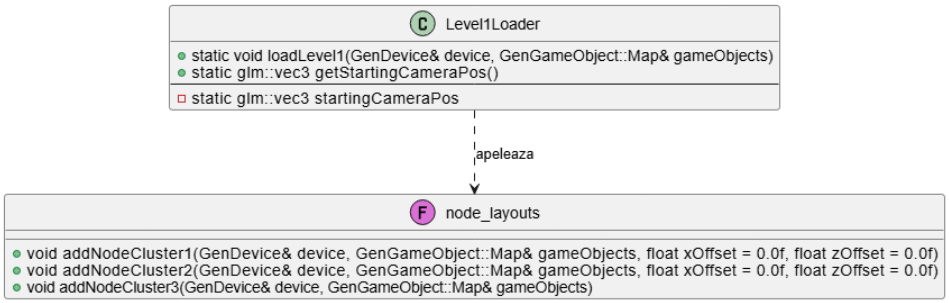


Figura 5.18: Diagrama clasei `Level1Loader` împreuna cu utilizarea funcției de așezare automată a nodurilor.

Astfel, programatorul poate include ulterior aceste niveluri în bucla principala a jocului, cu scopul de a testa în mod eficient design-ul fiecărui nivel. Totodată, se evita aglomerarea codului principal din game loop, deoarece nivelurile sunt abstractizate în fișiere separate.

In ceea ce privește așezarea nodurilor de sunet în scenă, modulul oferă câteva layout-uri predefinite, numite `NodeCluster`, cu scopul de a fi integrate automat în descrierea nivelului. Astfel, se evita duplicarea codului intre niveluri multiple, fiind necesar doar apelul funcției corespunzătoare layout-ului dorit.

5.5. Sistemul de vizualizare - Visualization System

Pana în momentul de fata, prin intermediul modulelor descrise, motorul de jocuri este capabil de a reprezenta scena virtuala împreuna cu entitățile care aparțin de aceasta și realizarea imaginilor pe baza datelor furnizate. Însa, sistemul care face posibilă proiectarea lumii virtuale pe ecran și transmiterea datelor care lipsesc din modulul de randare în acest sens, este sistemul de vizualizare.

Cu alte cuvinte, clasa `GenCamera` este responsabilă pentru definirea camerei virtuale și a planului ecranului, prin care este posibila vizualizarea scenei 3D utilizând proiecția ortografică sau proiecția perspectivă, fiind la baza un `GameObjcet` ce încapsulează funcționalitățile unei camere. Figura 5.19 reprezinta diagrama clasei și funcțiile pe care acesta le implementează.

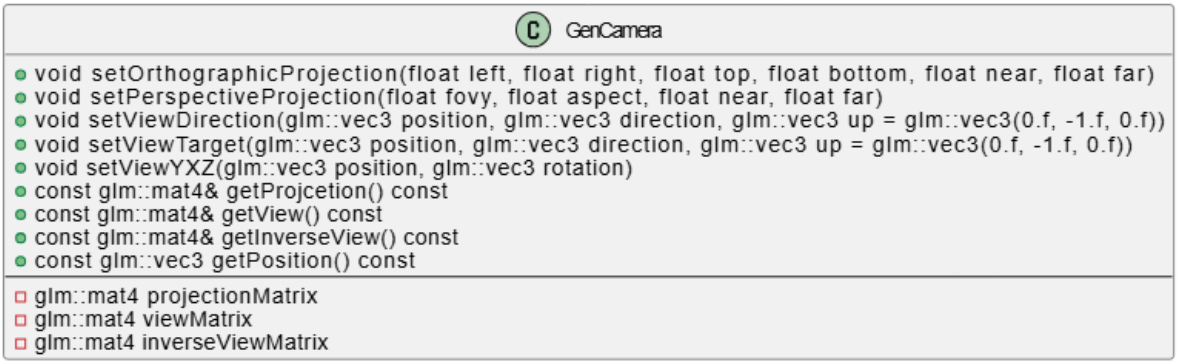


Figura 5.19: Diagrama classei `GenCamera`.

Pentru a oferi claritate, proiecția ortografică are proprietatea ca orice linie paralelă din cadrul scenei să rămână paralela, iar obiectele își mențin aceeași mărime indiferent de distanta acestora față de camera, fiind definită printr-un cub care cuprinde doar obiectele care vor apărea pe ecran.[Cap 4.7.1, p. 93][1].

Proiecția perspectiva este cea care simulează efectul real al perspectivei prin care ochiul uman percepe lumea din jur. Astfel, liniile paralele converg spre un singur punct la extremitatea lor. Definirea unei proiecții perspectivă are la baza un frustrum care definește:

- planul apropiat(near plane): toate obiectele care se află în fața acestui plan nu mai fac parte din zona vizibilă.
- planul depărtat(far plane): are rolul de a simula linia orizontului, astfel obiectele care se afla în spatele acestui plan nu mai sunt vizibile.
- fov(field of view - câmp vizual): este unghiul total exprimat în grade, sub care camera virtuală poate vizualiza scena.

Modelul proiecției perspectivei este ilustrat în figura de mai jos 5.20 preluată din [Cap. 4.7.2, p. 96][1] și reprezintă frustrum-ul descris împreună cu cubul unitate în care este transformat frustrum-ul pentru a mapa ecranul cu scopul de a simplifica calculele pe partea de GPU.

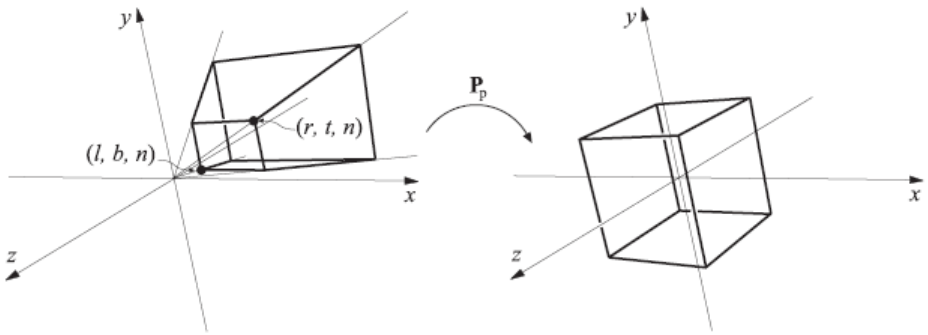


Figure 4.20. The matrix $\mathbf{P}_p$ transforms the view frustum into the unit cube, which is called the canonical view volume.

Figura 5.20: Modelul frustrum-ului

Astfel, clasa `GenCamera` implementează următoarele elemente definitorii ale camerei virtuale.

- Matricile caracteristice camerei virtuale sunt:
  - `projectionMatrix`: are rolul de a transforma coordonatele din spațiul camerei virtuale în planul ecranului.
  - `viewMatrix`: se ocupa cu transformarea coordonatelor obiectelor din spațiul virtual al lumii 3D în spațiul camerei(ceea ce observa camera).
  - `inverseViewMatrix`: este matricea prin care se transformă coordonatele din spațiul camerei înapoi în spațiul 3D al lumii virtuale, folosit în diverse calcule.
- Metodele `setOrthographicProjection(...)` și `setPerspectiveProjection(...)` sunt cele care realizează calculele necesare, astfel încât camera virtuala să vizualizeze lumea virtuală 3D conform definițiilor enunțate mai sus referitoare la acestea.
- Metoda `setViewDirection(...)` este metoda prin care se setează matricea de vizualizare utilizând 3 vectori prin care se poate defini aceasta și anume:
  - `w`: direcția în care se privește
  - `u`: vectorul spre dreapta al camerei
  - `v`: vectorul care reprezintă direcția în sus din poziția camerei.

  Astfel, prin aceasta metodă se poate crea o camera care privește într-o direcție generală, fără a fi nevoie specificarea unui punct anume la care să privească.
- Metoda `setViewTarget(...)`: este utilizată atunci când se dorește fixarea unui anumit punct spre care să privească camera.
- Metoda `setViewYXZ(...)`: este metoda prin care se definește camera pornind de la unghiurile Euler[Cap. Look Around][15]. Aceste unghiuri sunt adesea folosite pentru a descrie mișcarea unui avion, iar însemnătatea acestora este descrisa prin figura de mai jos 5.21 preluata din același capitol din [15].

Prin intermediul acestor metode, motorul de jocuri este acum capabil de a afișa pe ecran din perspectiva unei camere virtuale statice(nu poate fi deplasa în timp real) scena descrisă prin obiectele din spațiul virtual al lumii 3D.

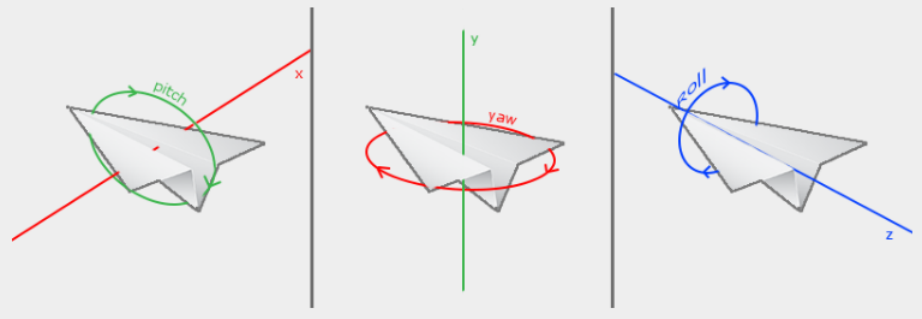


Figura 5.21: Reprezentare vizuala a unghiurilor Euler [15]

5.6. Sistemul de Control - Control System

Sistemul de control oferă motorului de jocuri funcționalitatea de a manipula poziția în scena, în timp real a obiectelor 3D(modele 3D, camera virtuala) prin intermediul intrărilor utilizatorului sau prin intermediul unor algoritmi care definesc un anumit comportament dinamic NPC-urilor.

Acest sistem este compus din trei clase a căror funcționalitate și implementare va fi detaliată cu scopul de a înțelege mecanismul care pune în mișcare lumea virtuala 3D. Diagrama de clase a acestora este prezentată în figura 5.22.

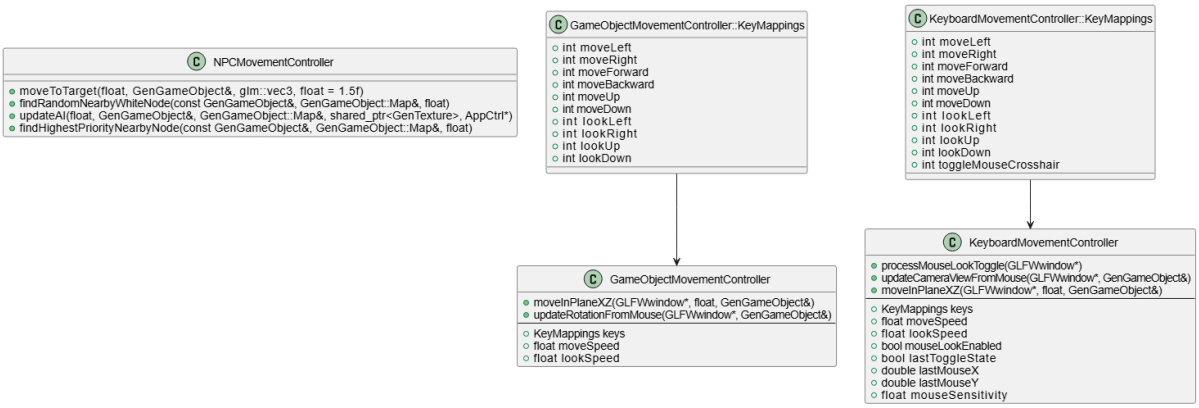


Figura 5.22: Diagrama de clase a claselor pentru control.

KeyboardMovementController

Aceasta clasa abstractizează procesul de traducere a input-urilor de la utilizator în transformări matriciale aplicate obiectelor din scenă, în special celor care necesită controlul direct, precum camera virtuala. Prin această clasă se gestionează deplasarea în planul solului(XZ), urcarea sau coborârea pe axa Y, cât și rotația în jurul axelor, oferind fluiditate în navigarea în scenă.

De precizat este faptul că lumea virtuală urmărește o logică inversă față de cea care este implementată folosind alte API-uri grafice. Prin urmare, lumea virtuala pe care motorul de jocuri o folosește este definita astfel:

1. Axa Y (verticala) : pentru a urca în altitudine componenta y trebuie să scadă(valori negative). Pentru a cobori, componenta y trebuie sa crească(valori pozitive)
2. Axa X (orizontala): valorile pozitive ale componentei x deplasează obiectul spre stânga, iar valorile negative deplasează obiectul spre dreapta.

3. Axa Z (adancime): pentru a îndepărta obiectul de ecran valorile componentei z trebuie să scadă, iar pentru a apropia obiectul acestea trebuie să crească.

Implementarea clasei este descrisă astfel:

- Structura `KeyMappings`: are rolul de a separa configurarea tastelor de logica de control propriu-zisa. Astfel, fiecare acțiune(deplasare înainte, rotire la dreapta) este asociată cu o denumire semantica explicită, ușor de înțeles și de utilizat în cod, abstractizând tasta fizică atribuită. Folosind aceasta structură, modificarea sau remaparea comenzilor este o operațiune flexibila care nu afectează logica de mișcare implementată.
  Controalele definite în această structură sunt:
  - Deplasare pe planul solului(XZ): tastele `I,K,J,L` (înainte, înapoi, stânga, dreapta).
  - Deplasarea pe axa verticala(Y): tastele `U,O`(sus, jos).
  - Rotirea camerei: săgețile de la tastatura și mouse-ul.
  - Activarea rotirii folosind mouse-ul: tasta `V`.
- Metoda `moveInPlaneXZ(...)`: această metodă definește logica de deplasare în spațiul 3D pe care un obiect din clasa `GenGameObject` o poate folosi și logica de rotație a acestuia, folosind doar săgețile de la tastatura. Astfel, orice acțiune pe care o face obiectul este raportata la timpul în care este activă aplicația, prin variabila `dt`, evitând dependenta de FPS-urile la care rulează jocul. Pentru a oferi o reprezentare a solului pentru obiectele afectate de gravitație, s-a limitat posibilitatea de a trece la valori pozitive pe axa verticala Y. Astfel, se obține coliziunea obiectelor cu solul.
- Metoda `processMouseLookToggle(...)`: prin aceasta funcție se verifică apăsarea tastei `V` pentru a activa sau dezactiva folosirea mouse-ului pentru a roti camera în scena. În modul de vizualizare al scenei folosind mouse-ul, cursorul dispare de pe ecran și camera poate fi operată mult mai ușor de utilizator.
- Metoda `updateCameraViewFromMouse(...)`: această metodă funcționează doar dacă modul de vizualizare al scenei folosind mouse-ul este activat. Rolul acesteia este de a calcula deplasamentul făcut de mișcarea mouse-ului fata de ultima poziție memorată a acestuia în urma activării modului respectiv. Calculele efectuate au rolul de a actualiza rotația pe axa Y(`yaw`) și rotația pe axa X(`pitch`), limitând rotația totala pe axa X prin funcția `clamp()` cu scopul de a împedică camera de a se da peste cap. Astfel, mișcarea camerei va fi asemănătoare cu cele folosite în jocurile FPS(First Person Shooter).

GameObjectMovementController

Această clasă preia unele aspecte din `KeyboardMovementController` și implementează doar funcționalitatea de deplasare în planul XZ a unui obiect din scena. Prin intermediul acestei clase, obiectul respectiv are posibilitatea de a fi controlat de către utilizator de la tastatura și simulează tipurile de mișcare ale unei persoane(înainte, stânga, dreapta, înapoi).

Astfel, în definirea clasei se folosește, la fel, o structura de tip `KeyMappings`, dar care abstractizează alte taste față de clasa precedentă. Tastele abstractizate de către implementarea clasei sunt:

- Tastele `W,A,S,D`(înainte, stânga, înapoi, dreapta) realizează deplasarea în planul XZ.

- Tastele `Q,Z`(sus, jos) realizează deplasarea pe axa verticala Y.
- Săgețile tastaturii sunt cele prin care se poate roti obiectul în jurul axelor, dar sunt rar folosite și menționate în cazul de față.

De asemenea, această clasă implementează funcția pentru deplasare în planul XZ numită `moveInPlaneXZ` similar cu cea care este definita în `KeyboardMovementController`

NPCMovementController

Această clasa este responsabila cu implementarea comportamentului folosit de către obiectele de tip NPC pentru a se deplasa în scena 3D. Este singura clasa din acest modul care nu folosește pentru input tastele, ci se bazează pe date referitoare la starea lumii virtuale în care NPC-urile există. Pentru claritate, NPC-urile au scopul de a traversa nodurile de sunet cu scopul de a găsi nodul cu intensitatea sunetului cel mai mare. Ierarhia intensităților este descrisa prin culoare de la liniște completă spre cel mai înalt zgomot: alb(liniște completă) < verde(sunet atenuat) < portocaliu(sunet normal) < roșu(sunet înalt). Detalierea completă a sistemului de sunet se regăsește în secțiunea următoare.

Pentru a simula mișcarea obiectul de tip NPC trebuie să fie capabil să: se deplaseze către o locație specifică, să caute printre nodurile din interiorul razei sale de acțiune următoarea destinație, în funcție de nivelul de sunet. Astfel, înainte de a oferi detaliile referitoare la implementare, diagrama din figura 5.23, exemplifica comportamentul NPC-ului.

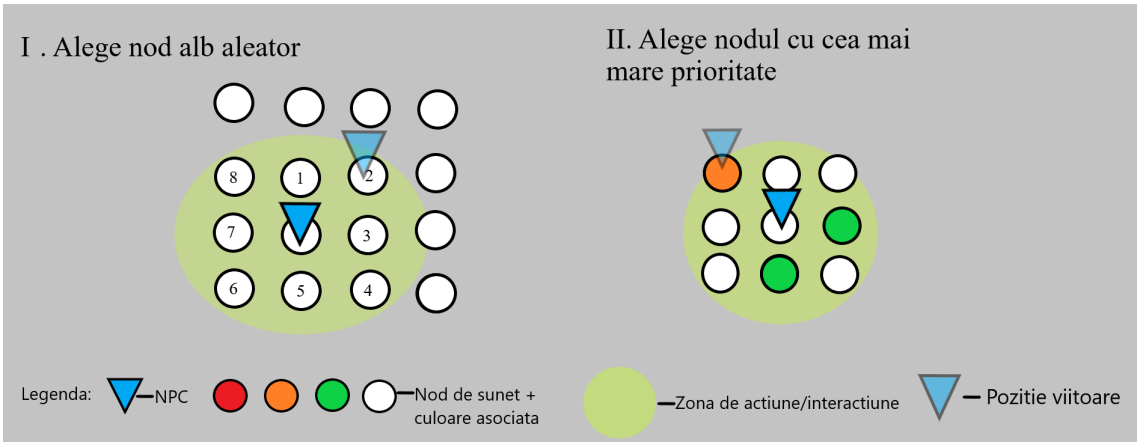


Figura 5.23: Comportamentul de deplasare al NPC-urilor

- Metoda `moveToTarget(...)`: realizează mișcarea obiectului NPC către o poziție țintă. Deplasarea se face pe planul XZ ignorând componenta axei Y. Calculează distanta dintre centrul obiectului(pe planul XZ) și destinația țintă. În cazul în care NPC-ul se afla lângă țintă, la o marja de eroare foarte mică, atunci deplasarea s-a realizat cu succes și NPC-ul este fixat în centrul țintei. Dacă distanta dintre cele doua este prea mare atunci se calculează distanta care trebuie parcursa și se aduna, folosind componenta de timp, la poziția curenta a NPC-ului, pana când se îndeplinește condiția de oprire descrisa anterior.
- Metoda `findRandomNearbyWhiteNode`: aceasta metoda este utilizata pentru a găsi în mod aleator următorul nod țintă la care NPC-ul trebuie sa ajungă, folosită pentru a simula deplasarea aleatorie în scena. Metoda filtrează toate obiectele de tip Node, iar apoi verifică care dintre acestea se afla în raza de acțiune a NPC-ului. Dacă sunt

găsite noduri cu o prioritate mai mare, funcția nu face nimic, deoarece este utilizata strict pentru deplasarea intre noduri vecine albe. Astfel, funcția găsește un nod alb aleator, pe care îl returnează cu scopul de a se folosi componenta sa de poziție în apelarea ulterioara a funcției `moveToTarget(...)`. Corespunde cazului *I* afișat în figura 5.23.

- Metoda `findHighestPriorityNearbyNode(...)`: implementarea acestei metode este folosită pentru a găsi nodul cu prioritatea cea mai mare în ierarhia de sunet din interiorul razei de acțiune a NPC-ului. Astfel, se filtrează toate nodurile din scenă care sunt în apropierea NPC-ului și se alege nodul cu prioritatea cea mai mare pentru a fi returnat de către funcție. Rezultatul funcție fiind folosit pentru a seta următoarea țintă la care NPC va trebui să ajungă.
- Metoda `updateAI(...)`: este funcția prin care se aplica metodele de deplasare ale NPC-ului în funcție starea lumii virtuale, actualizând în timp real poziția acestuia. Acesta ia decizii pe baza stărilor distincte de comportament, definite în structura `NPCState`.

  Implementarea acestei funcții urmează următoarele stări:

  1. `IdleRoaming`: verifica dacă există un nod cu prioritate mai mare în apropiere folosind funcția `findHighestPriorityNode(...)`. In cazul în care exista își schimbă starea în `SeekingSoundSource`. In caz contrar apelează funcția care caută nodurile albe din jur, `findRandomNearbyWhiteNode(...)`, urmând deplasarea spre nodul găsit.
  2. `SeekingSoundSource`: după identificarea unui nod cu prioritate mai mare, dar care nu este sursa sunetului, acesta se deplasează prin structura de tree găsita până ajunge la rădăcina, asigurându-se ca nodurile prin care trece redevin albe(mute), pentru a evita ciclurile în procesul de raționament. Odată ce a găsit un nod rădăcina, se schimba starea în `TracingRoot`.
  3. `TracingRoot`: se continuă deplasarea spre nodul rădăcina, repetând procesul de dezactivare. Când se dezactivează nodul părinte, acesta trece automat înapoi la starea `IdelRoaming`.

  Tot în cadrul funcției `updateAI(...)` se verifica dacă în raza de acțiune a NPC-ului se află un obiect de tipul `Player` cu scopul de a inița acțiunea de terminare a jocului care are ca rezultat resetarea nivelului de joc.

5.7. Sistemul de propagare a suntetului - Propagation System

Fiind un element de baza al jocurilor de tip stealth, sistemul de propagare a sunetului adaugă un strat suplimentar de complexitate în logica jocului. Scopul acestuia este de a aduce în lumea virtuală un model inspirat din comportamentul real al sunetului. De asemenea, realizează interacțiunea indirecta intre jucător și entitățile NPC. Deoarece spațiul virtual a jocului este definit folosind structuri de date și logica geometrica, propagarea sunetului trebuie să urmeze de asemenea aceasta construcție în cadrul aplicației.

Pentru implementarea propagării sunetului s-a proiectat următorul model:

1. Sunetul este produs doar de către jucător.
2. Sunetul produs este împărțit în patru categorii de intensitate: înaltă (codificata prin culoarea roșie), normală (codificata prin culoarea portocalie), scăzuta(codificata prin culoare verde) și liniște completă (codificata prin culoare alb).
3. Sunetele de intensitate normală și înalta sunt atenuate astfel:
   - Intensitate înaltă(rosu) > intensitate normală(portocaliu) > intensitate scă-

zuta(verde) > liniște completă(alb).
- Intensitate normală(portocaliu) > intensitate scăzuta (verde) > liniște completă(alb).

4. Pentru a se putea urmări sursa de la care provine sunetul se folosește o structura de date de tip `tree` în care sunetul activat de către jucător este nodul părinte, iar celelalte noduri din jurul acestuia reprezinta nodurile de atenuare a sunetului și vor fi noduri copil în structura de tree.

Figura de mai jos 5.24 oferă o reprezentare vizuala a modului de atenuare a sunetului în scena.

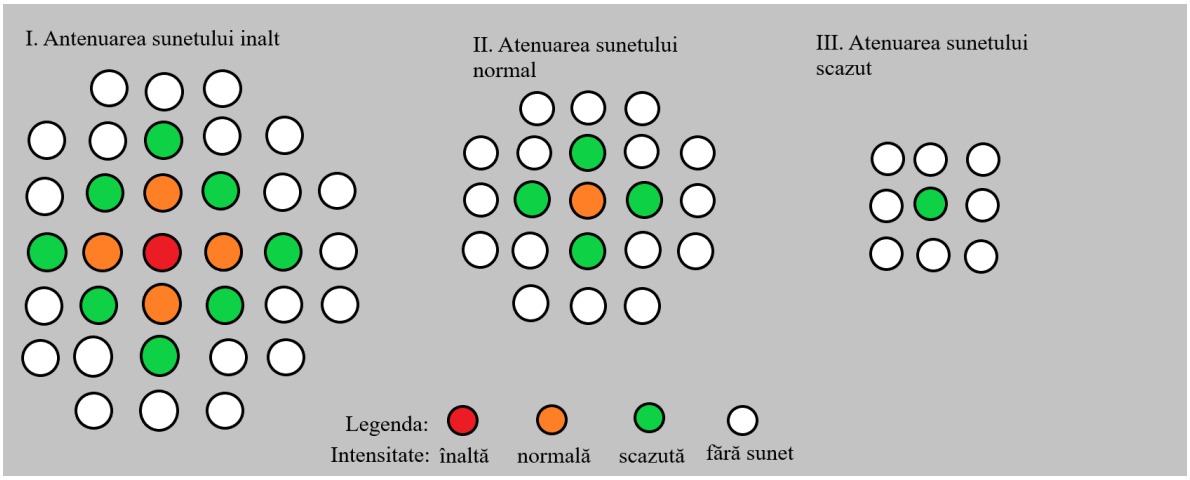


Figura 5.24: Modul de propagare a sunetului de diferite intensități

Clasa `NodePropagationSystem` este cea care implementează modelul de propagare a sunetului descris anterior.

- Metoda `findAdjacentNodes(...)` are scopul de a găsi mulțimea nodurilor vecine care se află în raza de influență a nodului sursă. Astfel, aceasta parcurge și filtrează toate obiectele de tip `Node` care sunt diferite de sursa. Apoi calculează distanța dintre aceste noduri și nodul sursă pentru a selecta doar acele noduri care se afla în raza de influenta menționata, adică distanța dintre nodul găsit și cel sursa este mai mica decât `maxDistance`.
- Metoda `propagateFromNode()` este apelată în momentul în care un nod este activat de către jucător cu scopul de a realiza atenuarea sunetului produs. Astfel, în prima parte a implementării se identifica nodurile vecine cu nodul activat folosind funcția `findAdjacentNodes(...)` pentru a crea "mediul de propagare". Procesul de propagare nu este unul arbitrat deoarece orice nod codificat de o culoare cu posibilitatea de atenuare nu poate afecta noduri care au deja aceeași culoare. Aceasta logica de tranziție controlează direcția și ritmul de răspândire, formând rețeaua de relații de tipul părinte-copil, esențială pentru construirea traseului de propagare a sunetului sursă.
  Structura de tip arbore a traseului de propagare este obținută prin salvarea în câmpul `parentId` a identificatorului nodului părinte de la care s-a primit semnalul, permițând astfel ca orice nod din rețea sa poată fi parcurs în sens invers pana la nodul rădăcina.
  Pentru a reflecta vizual starea în care se afla nodul în scena, fiecare culoare are asociata câte o textură distinctă și este aplicată modelului 3D al nodului doar dacă nu am mai fost aplicată recent. Controlul aplicării texturi este realizat printr-un

mecanism de cooldown care previne reactualizarea inutilă a texturilor în intervale prea scurte de timp pentru a evita artefactele vizuale.

Figura 5.25 reprezintă o parte vizuală a structurii de tree care se generează la apelarea funcției `propagateFromNode()`. De asemenea, se poate observa și traseul pe care NPC-urile îl urmează pentru a ajunge la atributul de poziție al nodului rădăcină.

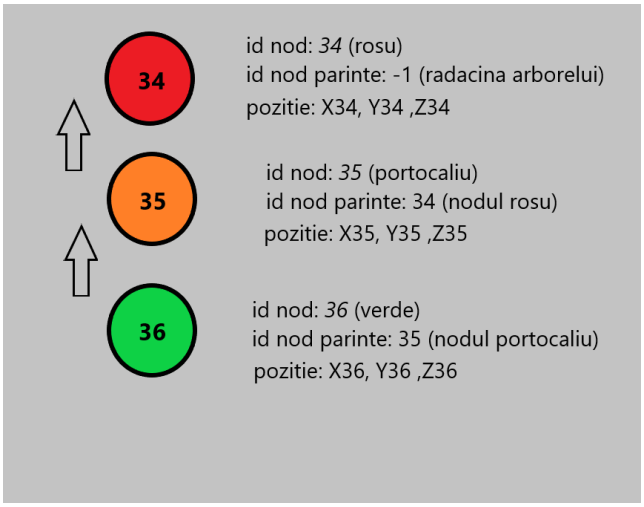


Figura 5.25: Vizualizare a structurii de tree

5.8. Bucla Jocului - Game Loop

Până în acest punct al lucrării s-au detaliat componentele individuale care stau la baza game engine-ului conform diagramei arhitecturale 5.1 prezentate în prima parte a capitolului. Totuși, pentru a crea un motor de jocuri complet funcțional, toate aceste componente trebuie să interacționeze pentru a facilita transferul de date, actualizarea stării interne și oferirea unui comportament vizual și logic coerent în timp real.

Pentru a oferi un punct central de control în aplicație, care să aibe rolul unui mediator între toate aceste componente și prin care programatorul sa-și poată defini propria logica de joc utilizând mecanismele puse la dispoziție, este necesară o clasa care să îndeplinească aceste cerințe. Clasa `AppCtrl` este componenta care unifica și organizează celelalte elemente astfel încât dezvoltarea unui joc sa fie posibilă. Diagrama de clasa detaliată a clasei `AppCtrl` se regăsește în figura 5.26. Deoarece clase utilizate de `AppCtrl` sunt componente externe auxiliare, acestea au fost reprezentate doar prin relații de asociere, fără a fi detaliate intern.

Înainte de a detalia aceasta clasa, trebuie precizat faptul ca în cadrul aplicației finalei, pentru a ilustra în mod clar și vizual utilizarea engine-ului, s-a dezvoltat un joc demonstrativ compus din doua niveluri. Ambele niveluri au același scop și anume traversarea încăperii în care se află jucătorul, fără a fi detectat de către NPC-uri, în scopul obținerii unui fragment dintr-un obiect mai mare care servește drept condiția de finalizare a jocului în urma coletării acestora.

Aceasta secțiune va detalia clasa `AppCtrl`, incluzând multitudinea funcționalităților pe care aceasta le implementează, precum și logica jocului regăsita în aplicația finală.

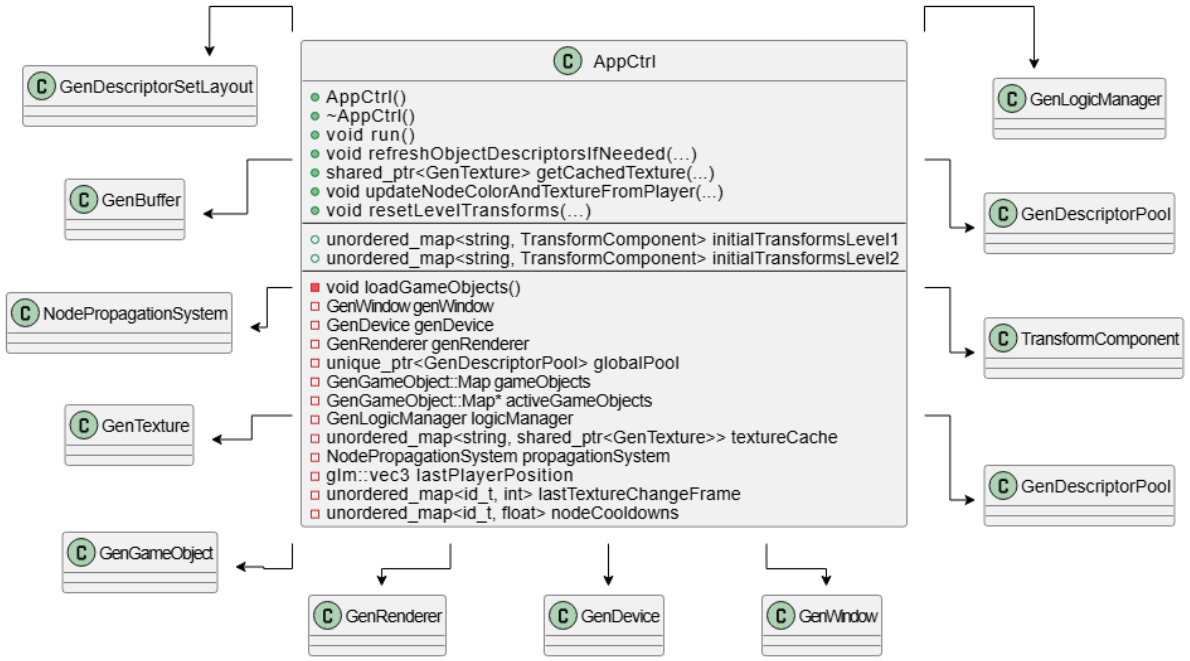


Figura 5.26: Diagrama clasei `AppCtrl`.

Constructorul `AppCtrl()`

Aplicația începe prin apelarea constructorului cu scopul de a inițializa sistemele esențiale ale aplicației, precum încărcarea nivelurilor și inițializarea descriptor pool-ului global.

Pentru a încărca nivelul definit prin intermediul componentei Level Design 5.4, se apelează funcția statica corespunzătoare fiecăruia, adăugând în referința către `gameObjects` obiectele definitorii acelui nivel. Deoarece jocul dispune de mecanismul de repetare a nivelului, prin intermediul câmpului `tag` al fiecărui `gameObject` se descrie apartenența obiectului la nivel. Astfel, se creează o structura de tip `map` în care sunt stocate elementele dinamice ce trebuie reașezate la resetarea nivelului, în poziția inițială. Tabela 5.1 reprezintă această structura de tip `map`, unde cheia este `tag-ul` , iar valoarea este referința către obiectul dinamic.

Tabela 5.1: Reprezentare în structura de tip map `initialTransformsLevel_`

Key	Value
Tag	&GameObject

De asemenea, o referință către toate aceste obiecte este memorata de variabila `avtiveGameObjects`, pentru a oferi claritate în filtrarea ulterioara a obiectelor pe nivel și în implementarea logici de joc.

Ultimul pas pe care constructorul îl efectuează este de a crea un pool global de descriptori, cu o capacitate suficient de mare pentru a susține toate obiectele din joc și texturile acestora.

Bucla principala `AppCtrl::run()`

Metoda `run()` reprezintă punctul de control al întregii aplicații, implementând bucla principala a motorului de jocuri. Această funcție gestionează inițializarea tuturor sistemelor logice și vizuale, realizează actualizările obiectelor în timp real și gestionează logica jocului, oferind suportul necesar pentru input, AI-ul NPC-urilor și resetarea nivelului.

Bucla principala implementata de aceasta funcție se va executa continuu pana la închiderea ferestrei, realizând un ciclu de joc descris de trei elemente: input, actualizare logica, randare.

Înainte de a se executa bucla principala, se configurează un set de resurse esențiale pentru fiecare cadru de randare:

- Se creează o textură albă care va fi folosită pentru orice obiect care nu are o textură atribuită prin variabila `fallbackTexture`. Pentru a putea fi folosită de către mai multe entități, textura creată se folosește de *smart pointer-ul* `shadred_ptr`.
- Se aloca buffer-ele de date uniforme pentru fiecare cadru cu scopul de a stoca datele de proiecție și de vedere ale camerei. Se utilizează metoda `map()` pentru a permite scrierea directa în memoria CPU-ului, având ca efect actualizarea mult mai ușoara a acestora în fiecare cadru fără a fi necesar un transfer intermediar.
- Se creează layout-ul descriptor-ilor globali care conțin informațiile utilizate de shadere. Datele sunt structurate astfel:
  1. un buffer UBO (binding 0)
  2. textura împreuna cu sampler-ul(binding 1),
  3. buffer-ul suplimentar care indică faptul că obiectul are textura(binding 2)

  Această structurare poate fi regăsita în codul shadere-lor descrise în subsecțiunea 5.2.2 și este comună tuturor obiectelor din scena, unificând modul de acces la resurse în cadrul pipeline-ului grafic.
- Se generează seturile de descriptori pentru fiecare obiect din scena, pentru fiecare cadru. Astfel, se verifica dacă obiectul are textură și se atribuie textura alba descrisa mai sus în cazul în care obiectul nu era texturat. Se creează din același considerente și buffer-ul separat care indică dacă textura este activă. Apoi se scriu elementele UBO, imagine, flag în setul de descriptori.
  Acest sistem este cel care asigură asignarea dinamică a texturilor și schimbarea acestora pe parcursul execuției, fără a fi necesara recompilarea sau reinițializarea.
- Se inițializează sistemele de randare `simpleRenderSystem`, `pointLightSystem` și clasa responsabilă pentru schimbarea propriu-zisa a texturilor `textureSwapper`.

Următorul pas după configurarea resurselor este definirea camerelor virtuale care să permită vizualizarea scenei.

Procesul de creare al camerelor este următorul:

- Se creează un obiect de tip `GenCamera` care reprezintă camera principală și va fi actualizata dinamic în timpul execuției.
- Se creează un obiect de tip `GenGameObject`, numit `viewerObject`, la care se va atașa camera principală pentru a permite controlul acesteia.
- Folosind un vector de obiecte de tip `GenGameObject`, numit `cameraStand` se definesc încă 3 poziții în scenă la care camera poate fi atașată.
- Pentru a comuta între camerele definite în vector, în cadrul buclei principale, se folosește un index numit `activeCameraIndex`.

După definirea camerelor, folosind aceeași logica cu vectorul de camere multiple, se definesc doi vectori de tipul `GenGameObject` pentru a stoca obiectele controlate de jucător(`ObjectType::Player`) și cele care pot fi controlate prin clasa `NPCMovementController`, adică NPC -urile (`ObjectType::NPC`).

Game Loop

În urma acestor inițializări, aplicația intră în bucla principală. Implementarea acesteia se face sub forma unei bucle `while` care se oprește odată cu închiderea ferestrei.

Aceasta bucla principala reprezintă ciclul de execuție a game engine-ului, realizându-se secvențial trei pași fundamentali pentru fiecare cadru:

1. procesarea input-ului
2. actualizarea logicii de joc împreuna cu starea obiectelor
3. randarea propriu-zisă a grafici pe ecran.

Contextul prin care această buclă cunoaște starea ferestrei și preia inputurile este prin intermediul funcțiilor de interogare a ferestrei ale librăriei GLFW.

La rularea aplicației, utilizatorul controlează una dintre camerele predefinite în vectorul `cameraStand`. Această cameră nu este legată de niciun obiect de tip `ObjectType::Player`, fiind utilizată pentru a observa scena în mod liber.

Pentru a schimba punctul de vedere din perspectiva altei camere, se apasă tasta `TAB`, fapt care declanșează codul de mai jos responsabil pentru această funcționalitate. Astfel, se incrementează indexul camerei și realizează decuplarea camerei de obiectul controlat de jucător(în cazul în care a fost folosita în acest sens), variabila care controlează acest comportament, `followPlayerCamera`, este setată pe `false`.

Atașarea unei camere la un obiect de tip `ObjectType::Player` se realizează prin intermediul tastei `C`. Apăsarea acestei taste are un rol dublu: permite comutarea intre obiectele controlare de către jucător și realizează atașarea camerei la aceste obiecte.

Astfel, prin fiecare apăsare a tastei `C`, indexul obiectului controlat , `activeObjectIndex`, este incrementat pentru a selecta următorul obiect din vectorul `controllabeObjecets`. Camera virtuală este apoi poziționată și fixata automat deasupra modelului obiectului selectat. De asemenea, variabila `followPlayerCamera` este setată la valoarea `true` pentru a împiedica alte mecanisme sa controleze camera virtuală.

Prin combinarea acestei logici cu mecanismul de control al mișcării obiectelor de către utilizator, se obține perspectiva specifică jocurilor de tip FPS(First-Person Shooter), în care utilizatorul observă scena din punctul de vedere a caracterului(obiectului) controlat.

Deși s-a vorbit despre controlul și selecția camerei, mai este de precizat faptul că prin tasta `V` se activează și se dezactivează controlul camerei folosind mouse-ul. Când acest mod este activ, cursorul dispare, iar mișcarea mouse-ului controlează direcția de privire a camerei, modificând rotația pe axa verticala (`pitch`) și orizontală (`yaw`). Activarea și dezactivarea acestui mod de control al camerei este similar cu un întrerupător, iar controlul acestei stări este realizat prin intermediul clasei `KeyboardMovementController`.

Dacă jucătorul rămâne blocat, sau eșuează sa atingă scopul final al jocului acesta are la dispoziție tasta `R` pentru a reseta așezarea inițială a scenei, fără a reîncărca întreaga aplicație, utilizând funcția `resetLevelTransforms(...)`. Resetarea este aplicata doar obiectelor care au tag-ul nivelului la care se afla jucătorul.

Tasta care ajuta jucătorul sa interacționeze cu obiectele de tip `ObjectType::Goal` este tasta `E`. Astfel, în implementarea mecanismului de îndeplinire a obiectivului se verifica

cat de aproape este un obiect de tip `Player` față de un obiect `Goal`, iar dacă jucătorul este suficient de aproape de acesta, la apăsarea tastei `E` se comuta pe obiectul controlat de către jucător din nivelul următor în mod automat.

Având în vedere faptul ca s-au detaliat inputurile auxiliare pe care utilizatorul le are la dispoziție, în plus fata de cele prezentate în subsecțiunea 5.6, în cele ce urmează este detaliata logica de joc propriu-zisa împreuna cu mecanismul de randare al cadrelor.

În cadrul motorului de jocuri, se simulează efectul gravitațional asupra obiectelor. Astfel, prin apelarea funcției `logicManager.update(...)`, se actualizează starea interna a obiectelor, folosind următorii pași:

- Se verifica tipul obiectului și se aplica forța de gravitație:
- Se detectează coliziunea cu solul și se previne trecerea sub nivelul acestuia
- Pentru fiecare obiect se verifică în planul XZ(solului), pe o anumita raza în jurul fiecărui obiect, dacă există suprapunere între discurile care descriu interacțiunea dintre obiecte. În cazul în care acestea se suprapun, utilizatorul este avertizat prin intermediul consolei despre cele doua obiecte intersectate.

În continuare, codul implementării realizează resetarea automată a culorii nodurilor după expirarea cooldown-ului. Astfel, pentru a reutiliza nodurile prin care jucătorul a trecut, și care nu au fost resetate de către NPC, acest mecanism parcurge obiectele din scena selectându-le pe cele de tip `Node`, verificând dacă perioada de cooldown a expirat. Dacă acestă condiție este îndeplinită se efectuează setarea atributelor nodului la valorile care definesc un nod de culoare alba: identificatorul părintelui este -1, culoare sa devine `NodeComponent::NodeColor::White;`, iar textura aplicată devine tot albă, asa cum este descris în porțiunea de cod prezentată mai jos.

Următoarea funcționalitate pe care bucla principala o implementează, este actualizarea stării NPC-ului. Astfel, se apelează funcția `updateAI(...)` descrisa în substituțiunea 5.23, fiind transmise ca parametrii starea actuală a lumii virtuale în care acest tip de caracter se află și obiectul texturii albe folosite de acesta la trecerea peste noduri colorate.

Controlul obiectelor de tip `Player`, în game loop, este realizat prin apelul funcției `moveInPlaneXZ(...)` din cadrul clasei `GameObjectMovementController`. Totodată, în paralel cu mișcarea obiectelor respective în scenă, se realizează propagarea sunetului provenit de la jucător.

Mecanismul de emitere a sunetului urmează următorii pași:

- Dacă obiectul controlat este de tipul `ObjectType::Player` se apelează funcția `updateNodeColorAndTextureFromPlayer(...)`.
  Aceasta funcție preia la rândul ei inputurile de la tastatura pentru a identifica intensitatea sunetului produs în funcție de combinația de taste primită. Dacă jucătorul apasă tasta `SHIFT` în combinație cu cele de mișcare, sunetul produs are intensitate înaltă, iar dacă se apasă în combinație cu acestea tastă `L CTRL`, sunetul are intensitate scăzuta. Simpla apăsare de taste emite un sunet de intensitate normala. După ce s-a identificat sunetul emis de către jucător se decide care nod de sunet este mai aproape de acesta pentru a reprezenta sursa de propagare a nodului.
- După ce a fost aleasă sursa de propagare a sunetului, pe baza câmpurilor setate obiectelor de tip `Node`, se apelează pe fiecare obiect de acest gen funcția `propagationSystem.propagateFromNode` cu scopul de a inițializa și efectua propagarea propriu-zisa a sunetului.

În momentul de față, bucla principală a jocului a efectuat toate operațiile de ac-

tualizare care s-au declanșat într-un cadru asupra datelor obiectelor. Singurul pas rămas este randarea scenei, folosind datele acumulate pana în acest punct.

Procesul de randare începe prin verificarea disponibilității buffer-ului de comanda în care se pot înregistra comenzile de desen. Dacă buffer-ul este valid, se pot înregistra comenzile de randare pentru cadrul curent.

Apoi se verifică dacă exista obiecte care și-au schimbat textura în timpul cadrului prin apelarea funcției `refreshObjectDescriptorsIfNeeded(...)`, deoarece acestea necesită actualizarea setului de descriptori. Fără aceasta funcție nu ar exista nicio garanție ca modificările aduse de actualizarea logici jocului asupra texturilor obiectelor, ajung sa fie reflectate pe ecran.

Configurarea structurii `FrameInfo` facilitează transferul de informații relevante între subsistemele de randare, reunind date precum buffer-ul de comandă, camera, seturile de descriptori și obiectele active.

Pentru afișarea corecta a obiectelor în raport cu camera activa, se configurează `GlobalUBO` (Global Unifirm Buffer Obejct), care conține date despre: matricea de proiecție, matricea view, și matricea inversa a view-ului.

În urma acestor configurări a datelor, Începe procesul de desenare al scenei utilizând funcția `genRenderer.beginSwachChainRenderPass(commandBuffer);`. Astfel, sunt apelate sistemele de randare `simpleRenderSystem` și `pointLightSystem`, care descriu modul în care trebuie desenate de către GPU, elemente corespunzătoare fiecăruia.

După completarea randării, procesul de randare este încheiat prin funcțiile `genRenderer.endSwachChainRenderPass(commandBuffer);` și `genRenderer.endFrame();`, semnalând faptul ca sa încheiat cu succes randarea unui cadru, urmând afișarea acestuia. Totodată se incrementează numărul cadrului curent.

În final, atunci când se închide fereastra, prin intermediul funcției `vkDeviceWaitIdle(genDevice.device();` se asigură că toate operațiile aflate în desfășurare pe GPU sunt finalizate înainte ca aplicația să distrugă toate resursele Vulkan folosite.

5.9. Rezultat - Output

După parcurgerea tuturor etapelor de procesare, coordonare și randare, rezultatul final al sistemului descris în diagrama arhitecturala 5.1 este o aplicație executabilă. Această aplicație reprezintă jocul 3D creat în urma folosiri motorului de jocuri implementat sub forma unui framework în care fiecare componenta din arhitectura sistemului contribuie direct la funcționarea corectă a întregului sistem.

Capitolul 6. Testare și validare

Pe baza detaliilor oferite în capitolele 4 și 5, aplicația practică a acestei lucrări reprezinta un joc stealth construit prin intermediul framework-ului 3D dezvoltat în Vulkan API. Acest framework funcționează similar unui game engine, fiind modularizat pentru a permite extensibilitate și scalabilitate.

6.1. Testarea componentelor principale

Pentru a testa funcționarea corecta a componentelor esențiale ale aplicației, s-a realizat un prototip al jocului final. Acest prototip are rolul unui mediu de testare, în care toate aceste componente au fost integrate fără elementele estetice finale, precum texturarea tuturor obiectelor și definirea completa a încăperilor din nivelul de joc. Mediul de testare creat este folosit pentru a verifica vizual funcționarea corecta a motorului de jocuri. Astfel, acesta cuprinde:

- Sistemul de propagare a sunetului
- Camerele virtuale
- Comportamentul NPC-urilor
- Definirea nivelurilor
- Obiecte controlate de utilizator
- Lumini punctiforme
- Tranziția intre niveluri

În prima fază a testării, s-au validat, cu ușurința, componentele legate de camera virtuală, lumina punctiformă și reprezentarea obiectelor în scena prin intermediul descrierii nivelurilor. Aceste componente pot fi observate în fereastra aplicației de la început, și cuprind un set restrâns de input-uri de la tastatura. Astfel, pe baza descrierii modului de funcționare a camerei virtuale și al controalelor folosite pentru deplasare, se poate observa scena în mod liber. Cele doua figuri de mai jos reprezinta câteva cadre surprinse din diverse unghiuri ale scenei.

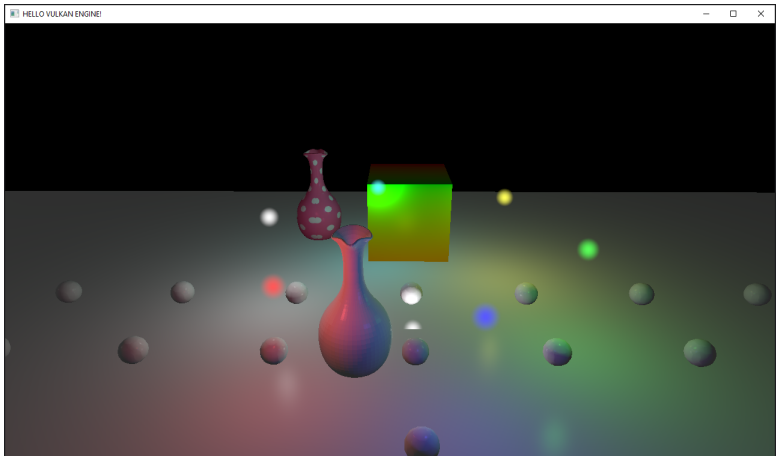


Figura 6.1: Validare camera virtuala, controlul camerei, incarcarea nivelurilor. Partea 1

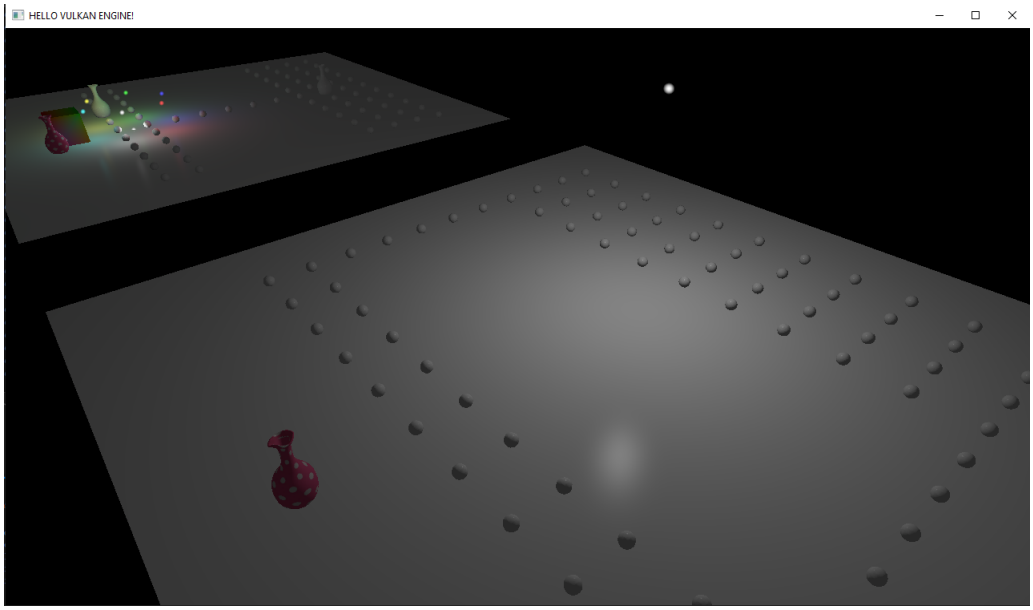


Figura 6.2: Validare camera virtuala, controlul camerei, încărcarea nivelurilor. Partea 2

Următoarele aspecte care s-au implementat în capitolul 5, și care necesită validare vizuală sunt controlul jucătorului, controlul NPC-urilor și propagarea sunetului.

Pentru controlul jucătorului, rezultatul așteptat este poziționarea fixa a camerei virtuale deasupra modelului controlat de acesta, iar combinația tastelor sa genereze culoarea corecta a nodului.

Pentru claritate figura de mai jos exemplifica conceptul de nod de sunet în cadrul aplicației, descris prin diagrama 5.24.

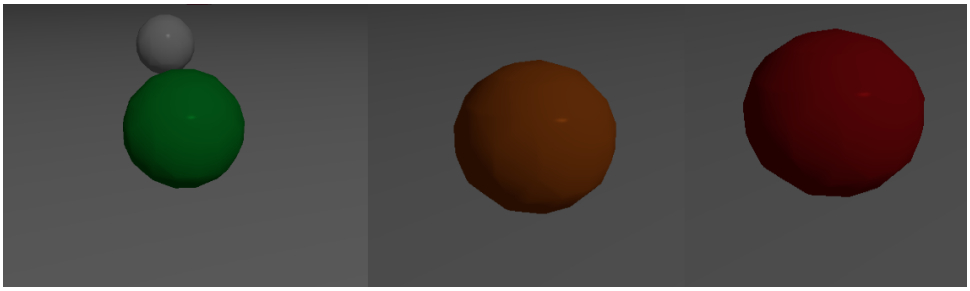


Figura 6.3: Reprezentare vizuală a nodurilor

În urma testelor efectuate pentru controlul jucătorului și a propagării sunetului, figurile următoare oferă veridicitatea funcționarii acestor componente.

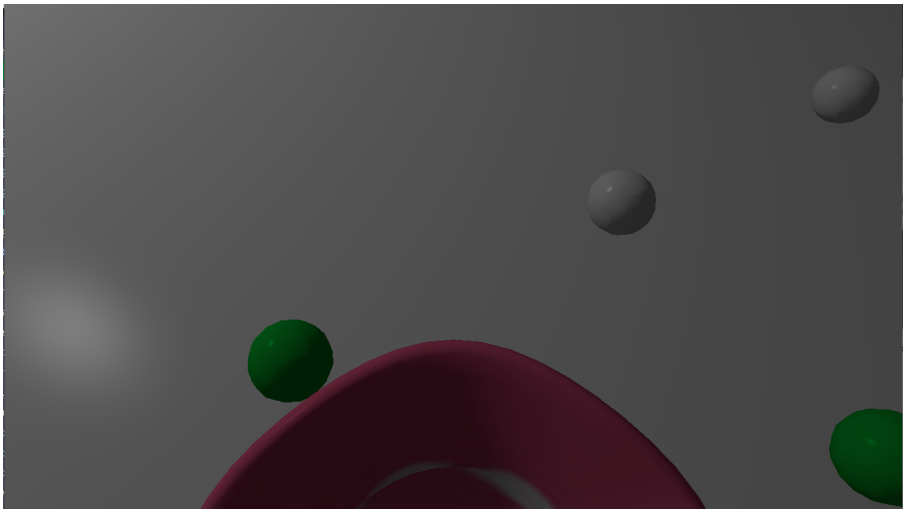


Figura 6.4: Controlul jucătorului & camera fixa pe model

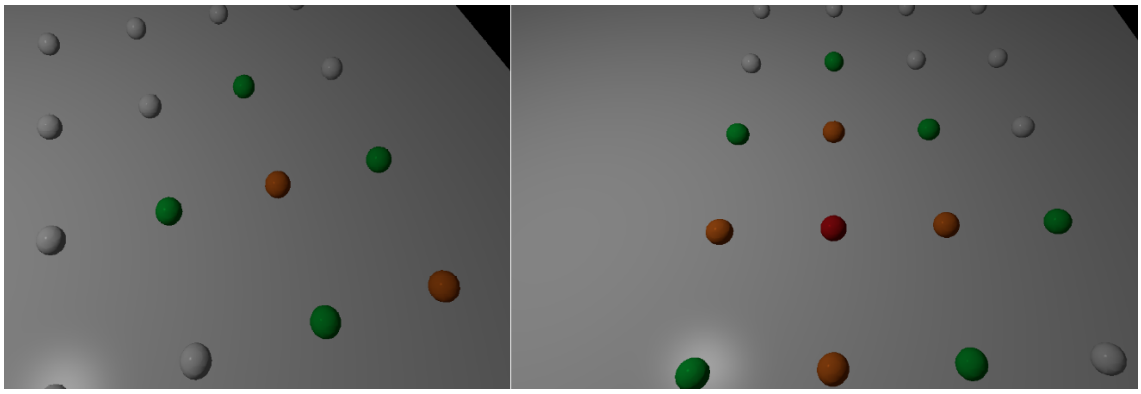


Figura 6.5: Propagarea sunetului

Componenta finală testată este comportamentul NPC-urilor. Astfel, NPC-ul trebuie să fie capabil de a traversa scena spre noduri aleatorii, iar la întâlnirea unui nod colorat de propagarea sunetului acesta trebuie să înainteze spre sursa sunetului în mod automat.

Astfel, următoarea figura exemplifica, prin panouri numerotate modul, de acțiune al NPC-ului conform algoritmului descris în secțiunea 5.23.

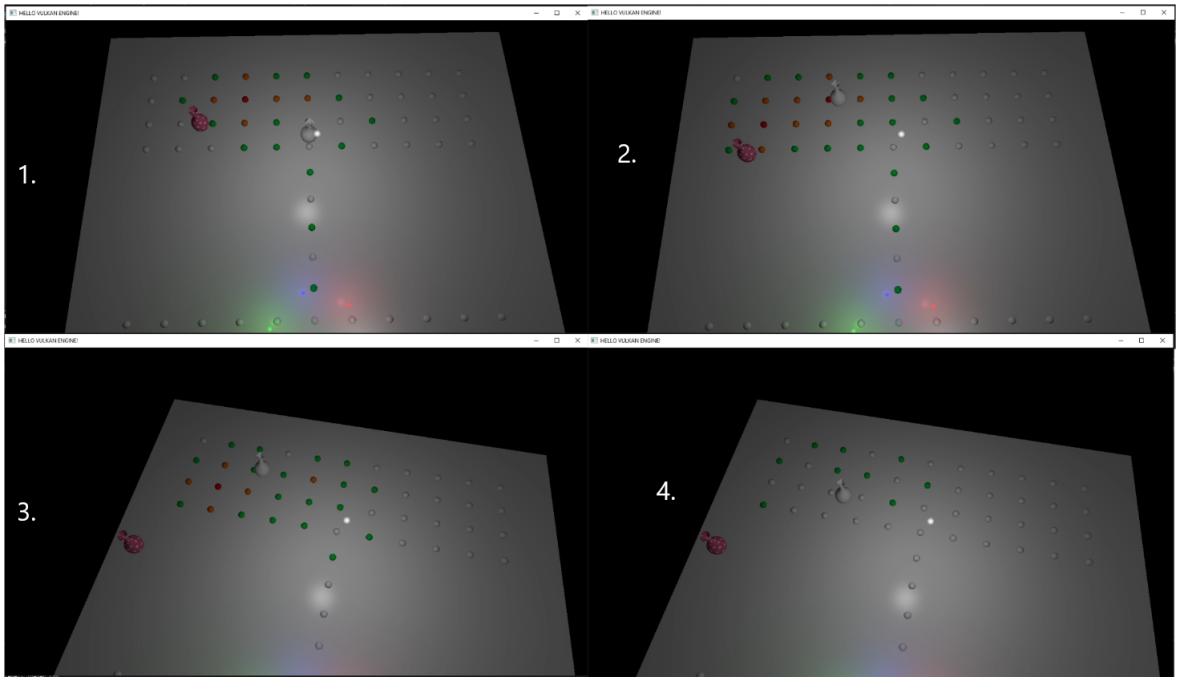


Figura 6.6: Deplasarea NPC-ului

În cadrul 1. se observa sunetul creat de către jucător pe care NPC-ul deja îl urmărește, îndreptându-se către nodul de culoare roșie. În al doilea cadru se observa cum NPC-ul a ajuns la acesta și cauta automat următorul nod de prioritate înaltă. În cadrul 3 se observa că se îndreaptă către nodul ales anterior, timp în care cooldown-ul resetează starea nodurilor. In ultimul cadru, NPC-ul revine la deplasarea aleatoare, găsind pe alocuri noduri verzi rămase din procesul de atenuare.

Prin aceasta modalitate de testare vizuală s-a dezvoltat aplicația.

6.2. Metrici de performanță

Pentru a testa nivelul de resurse folosite de către aplicație s-a folosit funcția de `Performance profiler` oferita de `Visual Studio 2022` pentru a vizualiza resursele utilizate de către: GPU, CPU, RAM, precum și numărul cadrelor pe secunda la care aplicația rulează.

Totodată în aceste teste s-a crescut, individual în medii de testare separate, numărul de noduri de propagare într-o scena și numărul de entități NPC care traversează scena. Numărul de noduri crește de la 100, la 500, la 1000. Iar numărau entităților NPC va crește de la 10, la 50, până la 100.

Rezultatele acestor teste sunt detaliate în cele ce urmează.

În codiți normale, precum cele prezente în prototip și în aplicația finala, sistemul rulează la o medie de 170 FPS-uri, utilizând doar 0.4% din GPU, 11% din CPU și aprox. 250 MB de memorie Ram așa cum este ilustrat de figura 6.7.

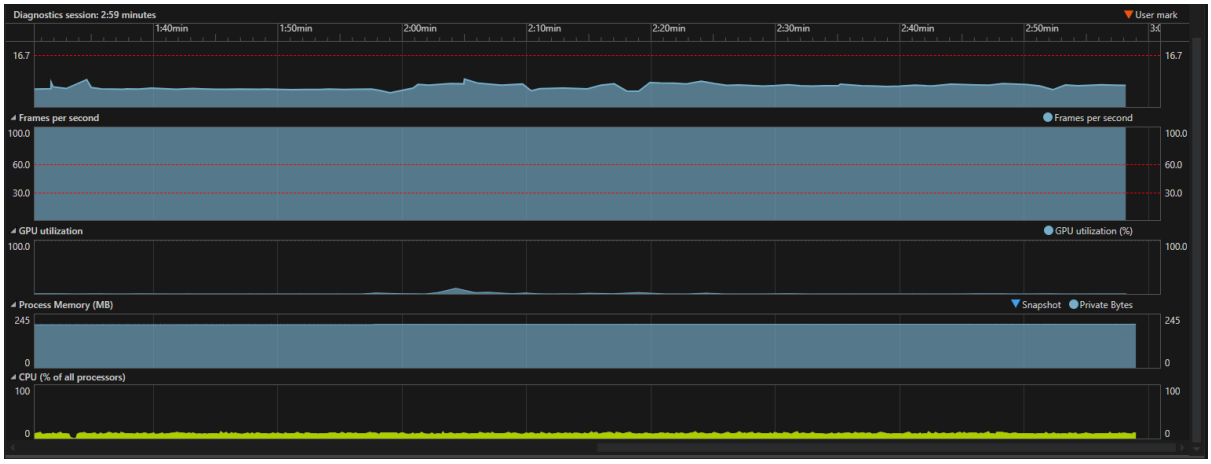
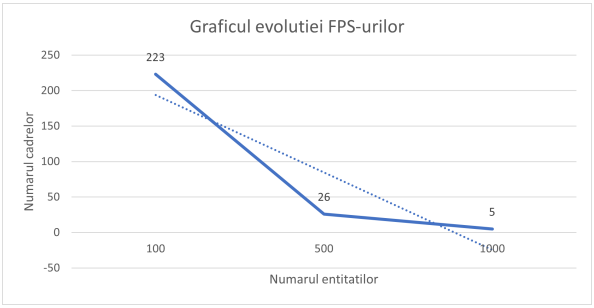


Figura 6.7: Metrica inițială a prototipului

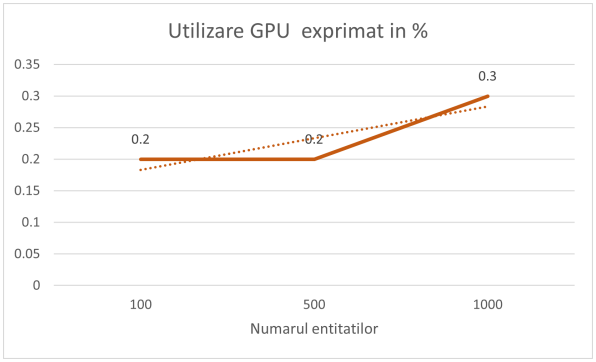
Performanta sistemului la un număr ridicat de noduri în scenă

Pentru a testa performanța, s-a creat o clasa care să servească acest scop și anume clasa `MemTestLoader`. Astfel, în cadrul `AppCtrl()` s-a inițializat scena conform cerințelor testului folosind noduri.

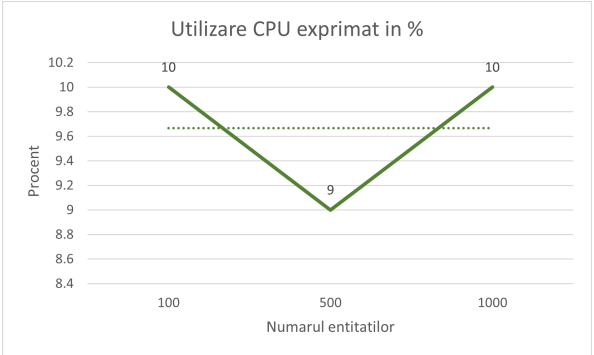
Graficele de mai jos indica faptul că la o creștere semnificativă de noduri în scenă, adică 1000, se observa o scădere considerabilă a cadrelor pe secundă, însa celelalte componente rămân stabile la modificarea parametrilor ilustrând performanța API-ului Vulkan. Astfel, optimizările trebuie aduse la design-ul pipeline-ului grafic pentru a crește rata de desenare a obiectelor.



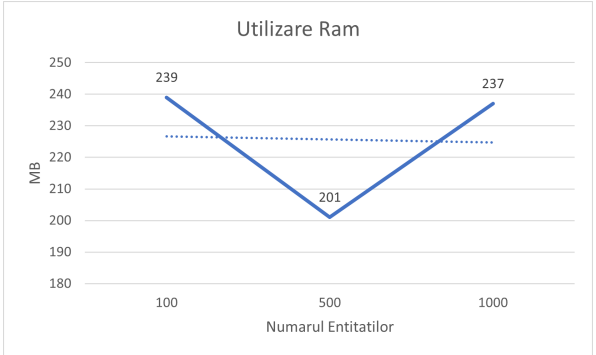
(a) Graficul cadrelor pe secunda



(b) Graficul utilizării GPU



(c) Graficul utilizării CPU

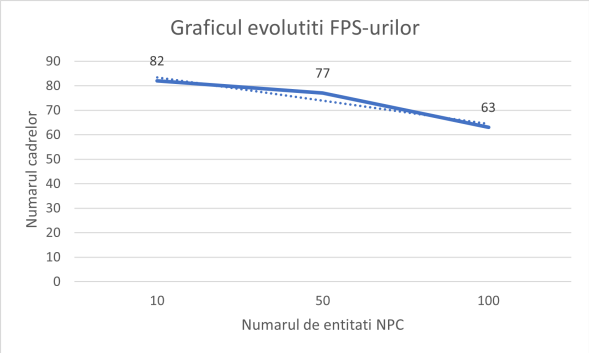


(d) Graficul utilizării RAM

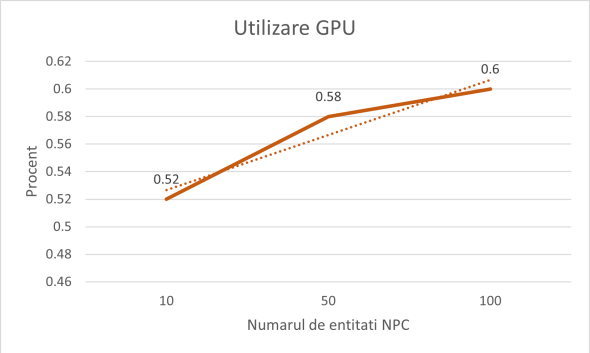
Performanta sistemului la un număr ridicat NPC-uri

Similar cu modul de testare a performanței sistemului la adăugarea unui număr ridicat de noduri, pentru a testa capabilitatea sistemului de a afișa pe ecran obiecte în mișcare, clasa de test va încărca în scena de la 10, la 50, pana la 100 de obiecte de tip `NPC` care se vor deplasa în mod aleator în scena într-un grid de 225 de noduri (15x15).

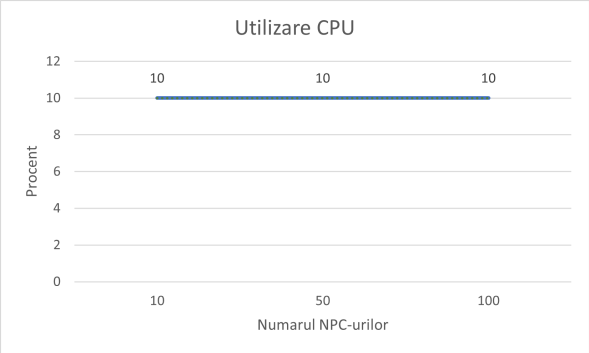
Graficele din figurile de mai jos reprezintă comportamentul sistemului la adăugarea mai multor entități NPC, care la runtime își modifica constant poziția.



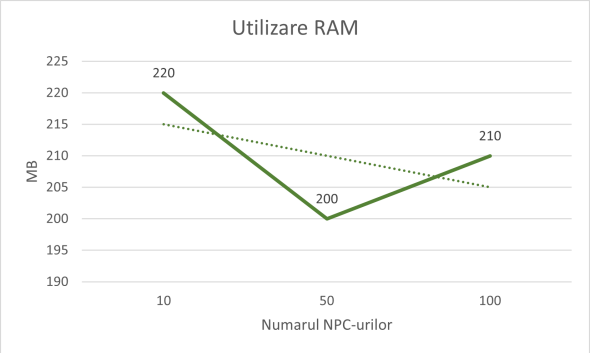
(a) Graficul cadrelor pe secunda



(b) Graficul utilizării GPU



(c) Graficul utilizării CPU



(d) Graficul utilizării RAM

Prin analiza graficelor se observă o ușoara stabilitate a cadrelor pe secunda, și o scădere mult mai liniara față de cea analizata anterior. De asemenea, faptul ca GPU lucrează constant cu obiecte în mișcare se observa o ușoara creștere în utilizarea întregii memorii a GPU-ului.

6.3. Specificațiile sistemului de testare

Testarea aplicației a fost efectuată pe un laptop cu sistem de operare Windows 10 64-bit, având următoarele componente hardware:

- Procesor: `AMD Ryzen 5 5600H`, 12 nuclee, 3.3 GHz cu `Radeon Graphics`
- Memorie Ram: 32 GB DDR4 - 3200 MHz CL22
- Placa video: `NVIDIA GeForce RTX 3060 Laptop GPU` cu memorie VRAM de 6GB.

Capitolul 7. Manual de instalare și utilizare

Pentru a utiliza motorul de jocuri implementat, sistemul utilizatorului trebuie să ruleze pe un sistem de operare Windows, cel puțin Windows 10(versiune pe 64 de biți). În plus, sistemul trebuie să îndeplinească următoarele cerințe minime software și hardware:

- Cerințe software:
  - Visual Studio 2022(cu suport pentru C++)
  - Standard C++ 17 activat în configurația proiectului
  - Vulkan SDK instalat
  - Biblioteci externe: GLFW, GLM, tinyobjloader și stb_image
- Cerințe hardware:
  - Procesor compatibil pe 64 de biți cu cel puțin 3.0 GHz
  - Placa video compatibila cu extensiile Vulkan utilizate de aplicație(recomandat: NVIDIA GeForce GTX 1070 cu 8 GB VRAM)
  - Memorie RAM: cel puțin 8 GB
  - Spațiul de stocare al aplicativei: ≈6 GB (care include SDK-ul Vulkan, librăriile externe, și soluția Visual Studio a aplicației)

7.1. Pașii pentru instalarea aplicației

7.1.1. Instalarea dependințelor

1. Se descarcă IDE-ul Visual Studio 2022 de pe site-ul visualstudio.microsoft.com/vs/. Visual Studio 2022 oferă suport complet pentru C++ 17.
2. Se descarcă installer-ul SDK-ul(software development kit) Vulkan de pe site-ul /vulkan.lunarg.com, folosind butonul cu logo-ul Windows. Se urmează pașii de instalare după rularea executabilului și se reține locația instalării.
3. Se descarcă următoarele librării adiționale și se grupează într-un director pentru a fi incluse în soluția proiectului:
   (a) GLFW: Este o librărie care gestionează crearea ferestrelor. Se descarcă binarele pe 64-bit de pe site-ul glfw.org/download.
   (b) GLM: Este o librărie folosită pentru operațiile de algebra liniară. Se descarcă de pe site-ul glm.mirror/.
   (c) tinyobjloader: Este un fișier header ca oferă posibilitatea de a citi și încarcă datele din fișierele `.obj`. Se descarcă doar fișierul `tiny_obj_loader.h` din GitHub-ul tinyobjloader.
   (d) stb_image: Este, de asemenea, un fișier header care oferă posibilitatea de a încărca în aplicație texturile ce vor fi folosite de obiecte. Se descarcă doar fișierul `stb_image.h` din GitHub-ul stb.
4. Se deschide Visual Studio 2022 și se creează un proiect gol(Empty project), setând tipul aplicației la *Console Application (.exe)*. Apoi se urmează următorii pași:
   (a) Se adaugă un fișier cu extensia `.cpp` numit `main` în care se inserează codul din Anexa A.
   (b) Din bara de meniu se selectează `Project -> NumeProiect Properties`.
   (c) În fereastra deschisă se setează în primul rând `Configuration: -> All`

`Configurations` din partea de sus a ferestrei. Apoi se navighează la `C++ -> General -> Additional Inlcude Directories` și se apasă pe butonul `<Edit...>`.

(d) Se adaugă calea spre directoarele în care se afla librăriile descărcate.
`C:\...\glm`,
`C:\... \glfw-3.2.1.bin.WIN64\include`
`C:\...\VulkanSDK \`*`versiune`*` \Include`
`C:\... \`*`nume director pentru tinyobjloader`*
`C:\... \`*`nume director pentru stb _image`*

(e) Se navighează la `Linker -> General -> Additional Library Directories` și se apasă pe butonul `<Edit...>`.

(f) Se adaugă locația fișierelor obiect pentru Vulkan și GLFW.
`C:\...\VulkanSDK \`*`versiune`*` \Lib`
`C:\... \glfw-3.2.1.bin.WIN64\lib-vc2015`

(g) Se navighează la `Linker -> Input -> Additional Dependencies` și se apasă pe butonul `<Edit...>`.

(h) Se adaugă pe rânduri separate următoarele nume ale fișierelor obiect:
`vulkan-1.lib`
`glfw3.lib`.

(i) Pasul final în această fereastră este setarea standardului limbajului C++ la `ISO C++ 17 Standart`, prin navigarea la `C/C++ -> Language -> C++ Language Standard`

5. Dacă pași enumerați mai sus au fost efectuați, rularea aplicației cu succes indica faptul ca toate dependințele au fost configurate cu succes.

Descrierea detaliata a acestor pași, însoțiți de suport vizual, se regăsește în capitolul *Development Envirorment*, secțiunea *Windows* din cartea [11].

7.1.2. Instalarea motorului de jocuri

Instalarea codului sursa a motorului de jocuri, în cadrul aceleași soluții Visual Studio descrisă mai sus, urmează următorii pași:

1. Din GitHub-ul OpGeorge/GameEngine se descarcă codul sursă sub forma de arhiva `.zip` sau se face o clona a repo-ului.
2. Având codul sursă descărcat acesta se copiază în întregime și se inserează doar în locul fișierului `main.cpp` din soluție, pentru a nu șterge directoarele și fișierele specifice unei soluții Visual Studio.
3. Deși codul sursă conține fișierele compilate `.spv` a shaderelor, în cazul în care utilizatorul dorește să modifice fișierele sursă ale shaderelor trebuie setat modul automat de compilare al acestora urmând pașii de mai jos:
   (a) Din bara de meniu se selectează `Project ->` *`NumeProiect`* `Properties`.
   (b) În fereastra deschisă se setează în primul rand `Configuration: -> All Configurations` din partea de sus a ferestrei.
   (c) Se navighează la `Build Events -> Pre-Build Event -> Command Line` și se apasă pe butonul `<Edit...>`.
   (d) Se adaugă următoarele linii de cod:
   `for %%f in (shaders.vert) do C:\VulkanSDK\`*`versiune`*`\Bin\glslc.exe %%f -o %%f.spv`
   `for %%f in (shaders.frag) do C:\VulkanSDK\`*`versiune`*`\Bin\glslc.exe %%f -o %%f.spv`

4. Se rulează aplicația care va porni motorul de jocuri.

7.1.3. Manual de utilizare

In cadrul ferestrei aplicației, interacțiunea cu motorul de jocuri se face folosind următoarele comenzi de la tastatură și mouse:

- Deplasarea camerei în mod liber în scena: `I,J,K,L,U,O`.
- Activare \Dezactivare folosirea mouse-ului pentru rotația camerei: `V`.
- Preluare control jucător, pornire joc: `C`
- Deplasare jucător: `W,A,S,D,Q,Z`.
- Resetare nivel: `R`
- Interacțiune cu obiectivul nivelului: `E`.

Capitolul 8. Concluzii

Lucrarea de față prezintă procesul de proiectare al unui framework pentru crearea jocurilor video 3D dedicate genului stealth, folosind API-ul Vulkan și limbajul de programare C++17. S-a pus accent pe eficiență, modularitate și extensibilitate, scalabilitatea fiind o cerința funcționala lăsată în plan secundar. Rezultatul final al lucrării este o aplicație funcțională care integrează elementele specifice modului de joc stealth precum: propagarea sunetului, implementarea unui AI simplificat pentru entitățile NPC, sistem de niveluri și logica de detecție.

În cadrul proiectului contribuția personală reprezinta extinderea semnificativa a funcționalităților de bază, oferite de tutorialul de la care s-a pornit. Cele mai importante elemente adăugate sunt:

- Definirea logicii avansate pentru modul de joc în componenta centrală a game engine-ului (Game Loop),
- Integrarea texturilor în pipeline-ul grafic pentru a permite schimbarea la runtime a texturilor obiectelor și a personaliza vizual obiectele în vederea creșterii nivelului de realism,
- Implementarea unui sistem de control al obiectelor pe care jucătorul le folosește,
- Implementarea unui sistem care integrează elementele care alcătuiesc AI-ul NPC-urilor în scopul controlării în scena a comportamentului acestora,
- Implementarea unui sistem care facilitează crearea entităților pe componente,
- Implementarea unui sistem de reprezentare și propagare a sunetului,
- Crearea unui framework extensibil, care oferă suport pentru integrarea de noi funcționalități.

Obiectivele stabilite în capitolul 2 al lucrări au fost atinse astfel:

- S-a realizat arhitectura motorului de jocuri stealth ce cuprinde bucla principala, entități configurabile, sistem de niveluri, deplasarea și controlul entităților NPC, mecanismul de propagare a sunetului și mecanismul de interpretare a input-urilor pentru obiectele controlate de utilizator.
- S-a realizat separarea logica în module a componentelor în scopul gestionarii resurselor.
- S-au definit input-uri clare pentru a facilita interacțiunea utilizatorului cu elementele camerei de vizualizare și controlul obiectelor folosite de acesta.
- Cerințele non-funcționale precum performanta, modularitatea și extensibilitatea au fost respectate. Acest fapt fiind susținut în capitolul 6.

Cerința non-funcțională cu privire la capacitatea de scalabilitate a framework-ului a fost atinsă parțial, deoarece procesul de optimizare al elementelor care influențează aceasta cerință poate deveni unul îndelungat care nu servește ca scop principal al aplicației finale.

Dezvoltările ulterioare pe care framework-ul le poate urma sunt:

- Integrarea unui sistem extins de coliziuni,
- Editor 3D pentru modelarea nivelurilor,
- Extinderea setului de algoritmi pe care entitățile NPC le implementează.

În concluzie, mini-motorul de jocuri video creat de la zero în aceasta lucrare, folosind API-ul Vulkan, reprezintă o bază solidă pentru dezvoltarea jocurilor 3D de tip stealth. Acesta este proiectat pentru a răspunde nevoilor dezvoltatorilor independenți, dar și pentru a servi ca instrument educational în formarea cunoștințelor viitorilor studenți în utilizarea acestui API. Codul acestui framework este deschis publicului, fiind open source (OpGeorge/GameEngine).

Bibliografie

[1] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards, *Real-time rendering*, 4th ed. CRC Press Taylor Francis Group, 2018.

[2] J. Gregory, *Game engine architecture*, 3rd ed. CRC Press Taylor Francis Group, 2019.

[3] A. Hussain, H. Shakeel, F. Hussain, N. Uddin, and T. L. Ghouri, "Unity game development engine: A technical survey," *Univ. Sindh J. Inf. Commun. Technol*, vol. 4, no. 2, pp. 73–81, 2020.

[4] V. H. Jokikokko, "The potential of unreal engine 5 in game development: Exploring the capabilities of the unreal engine," 2023.

[5] C. Developers, "Cryengie website." [Online]. Available: https://www.cryengine.com/showcase

[6] L. Bertolini, *Hands-On Game Development without Coding: Create 2D and 3D games with Visual Scripting in Unity*. Packt Publishing Ltd, 2018.

[7] Incredibuild, "Frotbite engine." [Online]. Available: https://www.incredibuild.com/glossary/frostbite-engine

[8] Medium, "Frostbite™ engine & fifa," 2020. [Online]. Available: https://medium.com/@urgen.nyc/frostbite-engine-fifa-1679174feda2

[9] K. Developers, "Kronosgroup." [Online]. Available: https://www.khronos.org/

[10] V. Developers, "Vulkan." [Online]. Available: https://www.vulkan.org/

[11] A. Overvoorde, *Vulkan Tutorial*. Kronos Group, 2023. [Online]. Available: https://vulkan-tutorial.com/

[12] K. McHenry and P. Bajcsy, "An overview of 3d data content, file formats and viewers," *National Center for Supercomputing Applications*, vol. 1205, no. 22, pp. 1–21, 2008.

[13] B. Galea, "Vulkan (c++) game engine tutorials." [Online]. Available: https://www.youtube.com/playlist?list=PL8327DO66nu9qYVKLDmdLW_84-yE4auCR

[14] ——, "littlevulkanengine." [Online]. Available: https://github.com/blurrypiano/littleVulkanEngine/tree/tut27

[15] J. de Vries, "Learnopengl." [Online]. Available: https://learnopengl.com/

Anexa A. Codul de început

```
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>

#define GLM_FORCE_RADIANS
#define GLM_FORCE_DEPTH_ZERO_TO_ONE
#include <glm/vec4.hpp>
#include <glm/mat4x4.hpp>

#include <iostream>

int main() {
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    GLFWwindow* window = glfwCreateWindow(800, 600,
    "Vulkan window", nullptr, nullptr);

    uint32_t extensionCount = 0;
    vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount, nullptr);

    std::cout << extensionCount << " extensions supported\n";

    glm::mat4 matrix;
    glm::vec4 vec;
    auto test = matrix * vec;

    while(!glfwWindowShouldClose(window)) {
        glfwPollEvents();
    }

    glfwDestroyWindow(window);

    glfwTerminate();

    return 0;
}
}
```