



UNIVERSITATEA DIN
BUCUREŞTI



FACULTATEA DE
MATEMATICĂ ȘI
INFORMATICA

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

**TEHNICI MODERNE DE
OPTIMIZARE ÎN GRAFICA PE
CALCULATOR UTILIZÂND
VULKAN**

Absolvent
Mihalache Sebastian-Ştefan

Coordonator științific
Prof. Dr. Stupariu Mihai-Sorin

București, iunie 2025

Rezumat

Procesarea unui volum mare de date este o operațiune computațional costisitoare, o provocare cu care se confruntă atât industria creativă, producția cinematografică sau dezvoltarea de jocuri video, cât și domeniile de cercetare științifică. În primul caz, dificultățile apar în crearea de efecte vizuale impresionante și realiste, iar în al doilea, în simularea unor fenomene fizice avansate.

Scopul acestui proiect este de a prezenta un sistem de particule real-time modern, performant, optimizat și scalabil, scris în C++ utilizând API-ul grafic Vulkan, care utilizează la maximum capacitatele plăcii video, explorând beneficiile unei arhitecturi în care datele necesare simulării sunt stocate și procesate integral pe GPU.

Soluția prezentată în această lucrare evidențiază avantajele concentrării sarcinilor de calcul direct pe GPU, punând în lumină modul în care această strategie poate contribui la îmbunătățirea performanței aplicațiilor grafice complexe real-time, unde viteza de procesare și eficiența utilizării resurselor sunt esențiale pentru a obține rezultate cât mai fezabile.

Abstract

Processing large amounts of data is a computationally intensive task, posing a significant challenge across various fields, including the creative industries, such as film production and video game development, as well as scientific research. In the former case, the primary difficulties arise in generating impressive and realistic visual effects, while in the latter, it stems from the simulation of advanced physical phenomena.

The aim of this project is to present a modern, high-performance, optimized, and scalable real-time particle system, written in C++ using the Vulkan graphics API, which fully leverages GPU capabilities by exploring the benefits of an architecture where all simulation data is stored and processed entirely on the GPU.

The solution presented in this paper highlights the advantages of focusing computational workloads directly on the GPU, highlighting how this approach can significantly improve the performance of complex real-time graphics applications, where processing speed and resource efficiency are essential for achieving the most feasible results.

Cuprins

1 Introducere	4
1.1 Contribuția proprie, obiective și motivații	4
1.2 Evoluția tehniciilor de procesare a particulelor	5
1.3 Domenii abordate	5
1.4 Structura lucrării	6
1.4.1 Infrastructura din spatele aplicației	6
1.4.2 Aplicația desktop	6
2 Infrastructura din spatele aplicației	7
2.1 Structura proiectului	7
2.2 Automation scripts	8
2.3 Premake5	8
2.4 GIT hooks	9
2.5 CI/CD Pipeline	9
3 Aplicația desktop	10
3.1 Simulator de particule	10
3.2 Vulkan API	11
3.3 Arhitectura aplicatiei	13
3.4 Buffers	15
3.5 Pipeline	16
4 Analiza performanței și concluzii	18
4.1 Performanță	18
4.2 Limitări	21
4.3 Direcții posibile pentru evoluția temei abordate	22
Bibliografie	24

Capitolul 1

Introducere

1.1 Contribuția proprie, obiective și motivații

În prezent, una dintre cele mai mari provocări în domeniul graficii pe calculator este crearea unor soluții eficiente pentru a afișa în timp real un volum mare de informații, fără a compromite calitatea vizuală, fluiditatea sau timpul de răspuns al sistemului software. În industria filmelor și a animațiilor 2D/3D, unde timpul de execuție nu este o problemă critică, procesul poate dura zeci de ore sau zile [1]. Efectele vizuale și simulările din astfel de produse folosesc unități de calcul extrem de puternice pentru a putea construi rezultate cât mai realist posibil, cu o înaltă fidelitate vizuală. Accentul este pus pe calitatea finală a imaginii, iar timpul de procesare nu reprezintă o constrângere majoră ce trebuie luată în calcul. Pe de cealaltă parte, în aplicațiile interactive care necesită *Real-Time Rendering*, cum ar fi jocurile video sau simulările, abordările de acest tip nu sunt deloc fezabile. Astfel de cazuri necesită o abordare diferită și mai atentă: se caută soluții care pot aproxima cu acuratețe rezultatele unor astfel de calcule costisitoare, menținând, în același timp, o performanță optimă, cu o rată de cadre pe secundă stabilă, pentru a putea oferi utilizatorilor o experiență cât mai fluidă și mai apropiată de realitate.

În cadrul acestei lucrări, contribuția mea principală constă în dezvoltarea unei aplicații care abordează această problemă, explorând o metodă de rendering accelerată, concentrându-se pe procesarea unui volum mare de date, menținând, în același timp, o experiență fluidă și realistă. Ca studiu de caz, am ales să implementez un sistem de particule modern, deoarece acestea sunt ideale pentru simularea și afișarea unui număr mare de entități dinamice, cum ar fi focul, fumul sau apa. Un astfel de sistem implică gestionarea a milioane de elemente care interacționează în mod continuu. Aplicația oferă un cadru și o platformă flexibilă pentru testarea tehniciilor de optimizare, axându-se pe utilizarea funcționalităților hardware-ului modern, în special pe proprietatea de a diviza și de a distribui pe cât este posibil munca computațională. În dezvoltarea acestui simulator, m-am bazat pe concepte validate în literatura de specialitate sau în aplicații grafice

anterioare, dar am adaptat și extins aceste idei pentru un API grafic modern, valorificând noile funcționalități și capabilități, cu accent pe paralelizare, minimizarea transferurilor CPU-GPU (unitatea centrală de procesare - unitatea de procesare grafică) și optimizarea utilizării memoriei.

Motivul alegerii acestui subiect derivă din dorința de a maximiza performanța și de a utiliza capacitatele plăcilor video la potențialul lor maxim. Tehnologia modernă a GPU-urilor este capabilă să efectueze milioane de operațiuni paralele simultan, iar acest proiect pune în evidență beneficiile folosirii acestei arhitecturi pentru simularea particulelor. Paralelizarea întregului proces de calcul computațional este crucială pentru a face față cerințelor aplicațiilor interactive de simulare.

1.2 Evoluția tehniciilor de procesare a particulelor

Printre primele implementări și definiții oficiale ale unui sistem de particule în grafica pe calculator au fost introduse de William T. Reeves [31], care lucrat la dezvoltarea unui sistem de particule complex cu un rendering secvențial pentru filmul *Star Trek II: The Wrath of Khan* din anul 1982. Cu tehnologia de la acea vreme, rezultatele obținute au fost spectaculoase, având o scenă cu 750.000 de particule. De-a lungul anilor au fost dezvoltate diferite metode de a face fezabil un sistem de particule și pentru *Real-Time Rendering*. O primă idee a fost implementarea simulării pe CPU, iar procesarea grafică pe GPU [26]. Odată cu evoluția plăcilor video au apărut diverse metode de a exploata arhitectura paralelă a acestora și a devenit posibilă o nouă abordare, și anume, mutarea ambelor procese, atât partea de simulare, cât și cea de rendering, direct pe GPU [30]. Una dintre tehnologiile moderne care au incorporat o strategie bazată pe rularea sistemului de particule direct pe GPU este *Unreal Engine*, un *Real-Time Rendering engine*, folosit în mare parte pentru dezvoltarea celor mai avansate jocuri, cu bugete de milioane de dolari, dar și pentru domenii precum industria auto, arhitectură, producție cinematografică, animație și simulări [2]. Această tehnică a fost introdusă sub numele de GPU Sprites [3], și datorită performanțelor pe care le oferă, a fost implementată la nivelul mai multor module și subsisteme [4]. Într-o comparație directă [34], un singur *GPU particle emitter* poate înlocui aproximativ 20 de *CPU particle emitters*. Totodată, acestă tehnică păstrează calitatea vizuală a efectelor, fără să aibă vreun impact negativ asupra rezultatului final, și este mult mai eficientă din punct de vedere al resurselor consumate.

1.3 Domenii abordate

Această lucrare va aborda concepte și tehnici din mai multe domenii relevante, printre care se numără:

- Grafică pe calculator, cu accent pe metode de optimizare și rendering pentru prelucrarea eficientă a unui volum mare de date folosind Vulkan.
- Sisteme de operare, analizând în special interacțiunea cu placa video, mecanismele de alocare a resurselor hardware, și gestionarea memoriei.
- Programare concurentă și multi-threading, utilizate pentru a îmbunătăți performanța întregului sistem care necesită calcule foarte costisitoare.
- Dezvoltarea de aplicații Desktop în C++, demonstrând aplicarea practică a concepțiilor prezentate prin implementarea unui sistem funcțional multi-platformă, compatibil atât cu Windows cât și cu Linux.

1.4 Structura lucrării

Lucrarea de față este împărțită în două capituloare principale:

- Infrastructura din spatele aplicației.
- Aplicația desktop.

1.4.1 Infrastructura din spatele aplicației

Secțiunea prezintă în detaliu infrastructura tehnică care susține dezvoltarea și funcționarea aplicației, punând accent pe automatizări pentru gestionarea întregului cod sursă și a procesului de livrare pentru produsul final destinat utilizatorilor.

1.4.2 Aplicația desktop

În această secțiune, este analizat tot procesul de dezvoltare a aplicației, abordând tehnologiile utilizate, arhitectura aleasă și soluțiile tehnice implementate pentru optimizarea performanței.

Capitolul 2

Infrastructura din spatele aplicației

2.1 Structura proiectului

Codul sursă al proiectului este organizat folosind sistemul de version control GIT [5] și poate fi direct accesat pe GitHub de către oricine la următorul link [6]. Dependețele proiectului sunt gestionate folosind git submodules, o modalitate prin care în repository-ul principal al proiectului sunt păstrate referințe la alte repositories. Astfel, atunci când este folosită comanda `git clone --recursive` pentru repository-ul principal al proiectului, se vor descărca și dependențele automat în directorul `vendor`. Dependențele proiectului și modul în care acestea sunt integrate în proiect sunt definite în fișierul `dependencies.lua`, responsabil de centralizarea informațiilor referitoare la directoarele de includere `IncludeDir`, directoarele bibliotecilor statice `LibraryDir`, precum și numele bibliotecilor necesare în funcție de platforma alesă.

Structura de organizare este definită la nivelul scriptului Premake5 [7], care are rolul de a genera configurațiile corespunzătoare pentru sistemul de build al platformei țintă, asigurând respectarea convențiilor de mai jos.

Întregul cod sursă al aplicației se regăsește în folderul `ParticleSystem/source` și este compus din fișiere `.cpp` și `.h`, în care este descris modul de operare al sistemului de particule. Implementările fișierelor care definesc funcționalitatea unui shader (unitate programabilă de tip shader rulată exclusiv pe placa video) se regăsesc în directorul `ParticleSystem/shaders` și sunt împărțite în trei tipuri: `.comp` pentru compute shaders; `.vert` pentru vertex shaders; `.frag` pentru fragment shaders.

Fișierele binare finale rezultate în urma procesului de compilare a codului sursă se vor regăsi în folderul `bin` urmat de subfolderul corespunzător celor două configurații disponibile ale aplicației, `Debug-windows-x86_64`, respectiv `Release-windows-x86_64`. Cele două subfoldere conțin la rândul lor un alt subfolder `shaders` în care se regăsesc toate shaders compilate care urmează să fie folosite, cele cinci benchmark-uri predefinite în folderul `benchmark`, alături de executabilul aplicației desktop ParticleSystem în formatul

corespunzător pentru Windows `.exe`, respectiv `.elf` pentru Linux; și versiunile compilate static ale bibliotecilor ImGui și GLFW, care au ca rezultat un fișier de tipul `.lib` pentru Windows, respectiv `.a` pentru Linux. Fișierele intermedii generate în timpul procesului de compilare a codului sursă se regăsesc în folderul `.bin-int` și respectă structura de subfoldere descrisă anterior. Aceste fișiere au extensia `.obj` pe Windows, respectiv `.o` pe Linux, și conțin codul mașină aferent fiecărui fișier sursă individual, dar care nu a fost încă legat într-un executabil final sau într-o bibliotecă.

2.2 Automation scripts

Pentru a asigura un mediu de dezvoltare eficient și portabil, am creat două script-uri `generate_project` care se ocupă automat de tot ce înseamnă setup-ul și dependențele necesare proiectului. Pe Windows este folosit un fișier cu formatul `.bat`, iar pe Linux este folosit un fișier cu formatul `.sh`. Cele două script-uri au ca scop executarea automată a următoarelor etape, fără a mai necesita intervenția utilizatorului: descărcarea, instalarea și configurarea Vulkan SDK [8] [9]; copierea directoarelor Vulkan `include` și `lib` în directorul principal al proiectului; copierea executabilului `glslc.exe` cu care se compilează toate fișierele sursă pentru shaders; configurarea git hooks; și într-un final, apelarea executabilului premake5.

2.3 Premake5

Premake5 este un instrument de dezvoltare software care asigură generarea fișierelor de configurare și build ale proiectului în funcție de platforma utilizată [7]. În spate sunt definite mai multe fișiere de configurare independente de platformă folosind limbajul de programare Lua [10]. În folderul `vendor/premake5`, se regăsesc două executabile de linie de comandă portabile premake5, unul pentru Windows, respectiv Linux. La apelarea acestor executabile, se vor folosi fișierele Lua definite pentru a genera pe Windows fișierele unui proiect care folosește Visual Studio 2022 [11], iar pe Linux fișierele unui proiect care folosește GNU Makefiles [12]. În generearea noilor fișiere, Premake5 asigură următoarele aspecte: configurarea arhitecturii pe x64; folosirea C++20; crearea configurațiilor de `Debug` și `Release`; includerea în proiect a fișierelor necesare care definesc funcționalitățile aplicației; legăturile de compilare statică pentru bibliotecile GLFW și ImGui, compilate local; definirea directivelor de preprocesare; adăugarea unui pas suplimentar înaintea tuturor etapelor de build, care identifică și compilează automat noile schimbări aduse unui fișier ce definește un shader.

2.4 GIT hooks

Există două git hooks, *post-checkout* și *post-merge*, configurate la nivelul proiectului, care vor asigura că de fiecare dată când sunt apelate comenzi de git care pot aduce modificări în structura proiectului, este rulat fișierul de setup *generate_project*. Astfel se vor regenera toate fișierele de tip build, reflectând mereu ultimele modificări aduse proiectului. În acest mod, automatizarea asigură constant integritatea proiectului.

2.5 CI/CD Pipeline

Pipeline-ul de CI/CD (Continuous Integration / Continuous Delivery) este construit cu ajutorul serviciilor GitHub Actions, care folosesc mașini GitHub-hosted cu Windows și Linux. Pe aceste sisteme este compilat tot codul dezvoltat, garantând în acest mod că noile modificări aduse la nivelul codului sursă sunt mereu compatibile cu ambele platforme. De asemenea, acest pipeline este responsabil și de distribuția aplicației atunci când este publicat un nou git tag. Produsul software final este publicat în secțiunea **Releases**, care se regăsește pe pagina web principală a proiectului [13]. Astfel, totul poate fi foarte ușor accesat de orice utilizator care își dorește să descarce și să încerce aplicația pe propriul calculator.

Există 2 fișiere separate în format *.yaml*, unul pentru Windows, și unul pentru Linux, care definesc modul și ordinea în care trebuie efectuate toate etapele din pipeline. Primul pas este cel în care se descarcă codul sursă al proiectului de pe GitHub, urmat de pasul de setup în care este rulat script-ul automat *generate_project*, responsabil pentru configurarea întregului proiect. Imediat după, vine pasul de build în care este folosit build engine-ul corespunzător platformei țintă, MSBuild pentru Windows [14], respectiv Make pentru Linux [12], pentru a crea executabilul final al aplicației. Etapele descrise mai sus sunt executate identic atât pentru versiunea Debug, cât și pentru versiunea *Release*, cu mențiunea că se schimbă doar valoarea parametrului *build-configuration*. Ultimul pas din pipeline este realizat doar pentru versiunea Release a aplicației și doar atunci când există un tag nou, ce semnalează faptul că există o nouă actualizare a aplicației ce trebuie distribuită. Pentru această etapă, este creat un folder în care sunt copiate toate fișierele de shader compileate, testele de benchmark și executabilul aplicației. Întregul folder este arhivat sub forma unui simplu *.zip* cu numele platformei concatenat cu noul tag, care este încărcat ulterior în secțiunea **Releases** [13].

Capitolul 3

Aplicația desktop

3.1 Simulator de particule

Aplicația desktop este un simulator interactiv de particule în care utilizatorul poate acționa asupra acestora prin definirea unui punct pe un plan 2D, care se comportă ca un centru gravitațional. Simulatorul oferă o reprezentare vizuală intuitivă a mișcării particulelor, permitând experimentarea cu diferite valori pentru a crea niște efecte vizuale cât mai spectaculoase. Punctul de atracție gravitațională este stabilit în funcție de poziția mouse-ului, iar forța gravitațională este activată doar atunci când utilizatorul apasă butonul stâng al mouse-ului. În momentul în care butonul nu mai este acționat, atracția punctului este dezactivată, iar particulele își continuă mișcarea sub influența inerției.

Pentru a facilita interacțiunea utilizatorului cu parametrii utilizati de simulator, dar și pentru a vedea rezultatele acestuia, aplicația integrează o interfață grafică intuitivă care permite monitorizarea și configurarea parametrilor. Interfața este compusă dintr-un meniu principal, reprezentat ca o bara orizontală în partea de sus a ferestrei, și încă două ecrane separate: *Settings* și *GPU Metrics*. Meniul principal este acționat folosind tasta ALT, iar pentru celalalte două ferestre se pot folosi shortcut-urile CTRL+E pentru *Settings*, respectiv CTRL+C pentru *GPU Metrics*. Fereastra *Settings*, oferă control asupra parametrilor fundamentali ai simulării. Utilizatorul poate regla numărul total de particule redată și schema de culori utilizată pentru vizualizarea lor. Există două tipuri de culori: o culoare statică, reprezentând particulele în repaus, cu viteza minimă sau nulă, și o culoare dinamică, reprezentând particulele în mișcare, cu viteza crescută. Fereastra *GPU Metrics*, oferă informații detaliate despre performanța aplicației. Acestea afișează următoarele metrii: numele plăcii video utilizate, rata de cadre pe secundă (FPS) alături de un grafic de evoluție, timpul de procesare pentru un singur cadru (Frame Time) măsurat în milisecunde vizualizat folosind un grafic corespunzător, precum și statistici legate de FPS: FPS mediu, FPS minim și FPS maxim. Toate aceste metrii sunt calculate pe baza ultimelor 4096 de cadre randate, oferind o perspectivă relevantă asupra performanței sis-

temului. Graficul pentru vizualizarea timpului de procesare pe cadru a fost adaptat după următorul articol [32]. Fiecare cadru din secvența graficului este reprezentat printr-un dreptunghi codificat vizual: culoarea indică performanța, lățimea este raportată la durata de timp a cadrului, iar înălțimea evidențiază diferențele relevante de timp între oricare două cadre succesive. Ferestrele destinate interacțiunii cu utilizatorul pot fi acționate separat, în afara contextului Vulkan de rendering. Toate funcționalitățile interfeței au fost construite cu ajutorul bibliotecii externe ImGui [15], care oferă mai multe instrumente pentru a dezvolta într-un mod ușor și eficient interfețe vizuale.

Pentru a evalua performanța aplicației într-un mod sistematic și ușor de reprodus, este implementat un sistem de benchmark automat. Acesta execută teste predefinite cu inputuri standardizate salvate sub forma unui fișier cu formatul `.json`, ceea ce va permite o analiză obiectivă a performanței pe diferite configurații hardware și sisteme de operare. În acest fișier sunt definite valorile corespunzătoare coordonatelor poziției mouse-ului și acționării butonului stâng al mouse-ului, împreună cu un timestamp cu ajutorul căruia este specificat momentul în care trebuie trimise evenimentele, astfel încât să fie recepționate de aplicație. Există cinci teste predefinite care pot fi efectuate folosind opțiunea de benchmark din meniul principal, dar pot fi modificate în orice moment pentru a acoperi alte scenarii de testare atât timp cât este respectat formatul prestabilit. Pentru a simplifica și pentru a respecta automat formatul necesar fișierului de intrare `.json`, a fost introdusă opțiunea de capta istoricul evenimentelor pe care le-a generat un utilizator. Acest instrument poate fi activat/oprit direct din meniul principal, secțiunea *Tools*, opțiunea *Capture Input*, sau prin folosirea shortcut-ului **CTRL+R**. Rezultatul va fi salvat în fișierul `auto-capture.json` în momentul în care este oprită înregistrarea. Valorile colectate pot fi modificate și utilizate ulterior pentru a crea un nou test de benchmark personalizat. Pentru a putea lucra cu formatul `.json` este integrată biblioteca externă header only, nlohmann [16] gândită pentru dezvoltarea aplicațiilor care doresc în implementarea lor o abordare C++ modernă.

3.2 Vulkan API

Vulkan este un API modern de randare și computație, dezvoltat de Khronos Group [17], care oferă control low-level asupra GPU-ului. Spre deosebire de API-uri mai vechi precum OpenGL, Vulkan este conceput pentru a minimiza overhead-ul driver-ului, ceea ce oferă o gestionare mai eficientă a resurselor hardware și permite totodată, optimizarea semnificativă a performanței, reducând latențele și îmbunătățind viteza de execuție. Filosofia din spatele API-ului Vulkan se bazează pe oferirea unui control aproape complet asupra GPU-ului, permitând dezvoltatorilor să obțină performanță maximă [29]. Totuși, acest nivel ridicat de control vine cu complexitate sporită, deoarece API-ul necesită gestionarea manuală a fiecărui aspect low-level ce ține de interacțiunea cu placa video, de

la modul în care sunt gestionate alocarea resurselor, până la sincronizarea dintre CPU și GPU. Dezvoltatorii trebuie să fie foarte expliciti la etapele de creare și configurare a pipeline-urilor grafice, ordinea comenziilor de rendering și la gestionarea memoriei tuturor buffers create.

Am optat pentru Vulkan datorită performanței superioare și a flexibilității pe care o oferă în comunicarea cu placa video. Printre avantajele cheie care au condus la alegerea acestui API se numără: suportul avansat pentru multi-threading; gestionarea explicită a memoriei; pipeline grafic personalizabil. În primul rând, suportul avansat pentru multi-threading permite distribuirea sarcinilor grafice și de computație pe mai multe fire de execuție, reducând astfel blocajele și maximizând utilizarea resurselor disponibile. Aplicația se folosește de acestă componentă prin execuția unui *compute shader*, ce are ca scop distribuirea sarcinilor pe mai multe nuclee și thread-uri ale plăcii video. În al doilea rând, gestionarea explicită a memoriei, permite programatorilor să controleze direct alocarea, eliberarea și reutilizarea spațiului în VRAM. Această abordare reduce overhead-ul asociat cu managementul automat al memoriei și oferă posibilitatea de a optimiza cu exactitate granularitatea transferurilor de date și alocarea resurselor, aspecte critice pentru minimizarea fragmentării și maximizarea coerentei cache-ului. Acest proces reduce operațiile de transfer redundante, permitând dezvoltatorilor să evite copierile inutile de date între memoria RAM și memoria VRAM, fapt ce duce la scăderi semnificative ale timpului de procesare și ale necesarului de lățime de bandă. Această idee este explorată în simulator prin folosirea de **Push Constants** și **Uniform Buffer Objects**. În același timp, această abordare permite reutilizarea zonelor de memorie pentru diferite resurse, cum ar fi reutilizarea unui buffer pentru multiple etape de procesare sau pentru obiecte temporare, maximizând astfel eficiența utilizării VRAM și reducând nevoie de alocări dinamice de memorie. Conceptul este implementat prin folosirea unui **Shader Storage Buffer Object**, un buffer prezent și utilizat atât în etapa definită de *compute shader*, cât și în cea de *vertex shader*. În plus, Vulkan permite crearea unui pipeline grafic personalizabil, oferind dezvoltatorilor control complet asupra etapelor de procesare grafică. Acest lucru elimină operațiile inutile din driver și permite optimizări avansate, cum ar fi utilizarea layout-urilor optimizate sau reducerea schimbărilor redundante de stare. Nu în ultimul rând, Vulkan oferă un control avansat al sincronizării între CPU și GPU, asigurând coerența datelor și prevenind conflictele de acces la resurse comune. Acest control este realizat prin mecanisme precum semafoare, bariere de memorie și fences, care permit coordonarea precisă a comenziilor și reducerea timpilor de așteptare.

Deși Vulkan este un API scris în limbajul C, am ales să folosesc C++ pentru dezvoltarea aplicației datorită combinației sale de performanță ridicată, control manual asupra memoriei și suportului avansat pentru programarea orientată pe obiecte. C++ oferă mecanisme puternice de gestionare a resurselor prin intermediul paradigmii RAII (Resource Acquisition Is Initialization) [33], un principiu fundamental care asigură că resursele

alocate sunt eliberate automat atunci când obiectele care le dețin ies din contextul de operare, evitând astfel *memory leaks* și alte erori legate de alocarea dinamică, frecvent întâlnite în limbajul C.

3.3 Arhitectura aplicației

Arhitectura aplicației se bazează pe o implementare modulară și eficientă a sistemului de particule, care se folosește la capacitate maximă de proprietățile plăcii video. Sistemul de particule este proiectat astfel încât fiecare particulă să fie complet independentă, cu un set unic de date, ceea ce permite paralelizarea tuturor calculelor necesare unei singure instanțe prin intermediul unui *compute shader*, responsabil pentru gestionarea și actualizarea proprietăților fiecărei particule, cum ar fi poziția și viteza, profitând în acest mod, de capacitățile de procesare paralelă ale arhitecturilor moderne de GPU.

Această abordare exploatează pe deplin arhitectura paralelă masivă [27] (Massively Parallel Architecture) a unităților de procesare grafică, care au fost concepute special pentru a executa mii de operații și calcule simultan. Spre deosebire de unitățile centrale de procesare optimizate pentru execuția secvențială, plăcile video integrează mii de nuclee destinate unor calcule simple, fiecare capabil să proceseze independent seturi de date. Acest ecosistem computațional specializat permite manipularea eficientă a volumelor mari de date omogene, prin distribuirea uniformă a sarcinii de calcul pe toate nucleele fizice disponibile. Astfel, fiecare particulă beneficiază de resurse dedicate de procesare, eliminând blocajele de performanță specifice arhitecturilor secvențiale. Un alt aspect care justifică alegerea acestei soluții este dat de faptul că între procesoarea oricărora două particule nu este necesar niciun mecanism de așteptare, nu există nicio formă de comunicare sau schimb de date între două instanțe de particule, fiecare având proprietățile calculate într-un mediu complet izolat, fără să țină cont de datele altor particule din sistem.

Pentru vizualizarea rezultatelor sistemului de particule, este necesară crearea unui context grafic și a unei ferestre în care placa de video să poată afișa conținutul. Această funcționalitate este implementată folosind biblioteca GLFW (Graphics Library Framework) [18], care oferă un API modular și eficient pentru gestionarea ferestrelor, contextului Vulkan, configurarea suprafețelor de redare și prelucrarea evenimentelor de intrare (tastatură, mouse, controller). O altă bibliotecă externă care a fost integrată este GLM (OpenGL Mathematics) [19], o bibliotecă C++ header only care oferă un set complet de tipuri și funcții matematice esențiale pentru dezvoltarea unei aplicații grafice, precum operații cu vectori, matrici, transformări geometrice (translație, rotație, scalare) și calcul de proiecții (ortografică, perspectivă).

Alegerea fizică a dispozitivului grafic optim reprezintă o etapă importantă în configurarea aplicației, deoarece performanța și stabilitatea sistemului de particule depind în mod direct de proprietățile hardware ale unității de procesare grafică. În majoritatea sistemelor

moderne, sunt disponibile două categorii de dispozitive grafice. În prima categorie se regăsesc plăcile video integrate, care sunt incorporate direct în procesorul central și utilizează memoria RAM, iar în a doua categorie sunt plăcile video dedicate, echipate cu memoria video proprie (VRAM), fiind special proiectate pentru sarcini de calcul intensiv. Această selecție joacă un rol esențial deoarece va avea cel mai mare impact în determinarea performanței și a capacitaților de calcul ale aplicației. Diferențele dintre o placă video integrată și una dedicată pot duce la diferențe uriașe în aplicațiile solicitante din punct de vedere grafic. Pentru a asigura compatibilitatea și performanța maximă, aplicația implementează un mecanism de selecție manuală a GPU-ului, care enumera toate dispozitivelor grafice disponibile în sistem și formează o ierarhie ce reflectă performanța acestora, fiind evaluate conform unui set de criterii tehnice predefinite. Vom alege device-ul cu cele mai multe și rapide nuclee disponibile pentru calculul paralel. De asemenea, este necesar să aibă abilitatea de a prezenta imaginile rezultate în urma întregului proces computațional. O altă cerință este suportul pentru extensii obligatorii de care are nevoie sistemul de particule, printre care se regăsesc *VK_KHR_swapchain*, *VK_KHR_storage_buffer_storage_class*. În contextul unui simulator de particule, care solicită o putere de calcul paralelă cât mai mare, utilizarea unei plăci video dedicate este preferată.

În arhitectura API-ului Vulkan, conceptele de queue families și queues joacă un rol crucial în gestionarea execuției operațiilor pe GPU. Acestea sunt în esență structuri care controlează fluxul de comenzi trimise de CPU către GPU. Fiecare GPU expune una sau mai multe Queue Families, fiecare specializată în anumite tipuri de operații. În contextul unui simulator de particule, aplicația necesită acces la două tipuri de cozi: *graphics queue* și *compute queue*, utilizate pentru execuția celor două pipeline-uri specializate. *Graphics Queue* este folosită pentru procesul de rendering al particulelor pe ecran, ceea ce include desenare, blending, rasterization, execuția de vertex și fragment shaders. *Compute Queue* este folosită pentru a executa operații în paralel, ceea ce include calcularea și actualizarea proprietăților particulelor. Această componentă a API-ului este extrem de eficientă în manipularea volumelor mari de date și în execuția rapidă a algoritmilor matematici necesari pentru simularea realistă a mișcării particulelor.

Swap chain (lanțul imaginilor pregătite pentru afișare) este mecanismul central în Vulkan pentru gestionarea imaginilor prezentate pe ecran. Acesta constă într-o colecție de buffers, de obicei două sau trei, care alternează între stările de afișare și rendering, asigurând o experiență fluidă. Pentru a îmbunătăți calitatea procesului de rendering, este utilizată o imagine temporară numită *Intermediary Image*, care are un nivel mai mare de multisampling, fiecare pixel este eșantionat de 8 ori. Tehnica *Multisampling Anti-Aliasing* [28] (MSAA) reduce artefactele de la marginile obiectelor desenate, îmbunătățind calitatea imaginii procesate. După ce imaginea intermediară este procesată, aceasta este transpusă în imaginea din swap chain, numită *Swap Chain Image*, utilizată pentru prezentarea finală în fereastra aplicației.

3.4 Buffers

Sistemul de particule dezvoltat în cadrul acestui proiect utilizează trei tipuri principale de buffere pentru a gestiona eficient datele între CPU și GPU: **Uniform Buffers**, **Push Constants** și **Shader Storage Buffer Object**. Alegerea acestor tipuri de buffere a fost ghidată de nevoiea de a maximiza performanța și de a minimiza latențele de transfer.

Uniform Buffers Objects sunt folosite pentru a stoca date care nu se schimbă frecvent. Aceste buffers sunt preferate într-un context în care este nevoie de o modalitate eficientă și simplă de a partaja date constante între CPU și GPU, fără a consuma excesiv lățimea de bandă a memoriei. Pentru a permite accesul CPU-ului la aceste date, obiectele buffer sunt create astfel încât să fie vizibile atât pentru procesor, cât și pentru placa video. Pentru aplicația implementată, am ales să folosesc acest tip de buffer pentru a stoca matricea de proiecție și culorile folosite în colorarea particulelor deoarece sunt valori actualizate foarte rar, posibil să rămână constante pe tot parcursul derulării simulatorului. Matricea de proiecție este actualizată doar când sunt schimbate dimensiunile ferestrei principale, iar cele două culori pot fi modificate doar la comanda utilizatorului.

```
1 struct PushConstants
2 {
3     uint32_t enabled;
4     float timestep;
5     glm::vec2 attractor;
6 }
```

alloare explicită, gestionare a memoriei și un set de *binding descriptors*, acestea sunt inserate direct în pipeline-ul grafic și transmise prin comenzi de rendering, eliminând overhead-ul asociat transferurilor de date. Conform documentației [20], Vulkan garantează cel puțin 128 bytes pentru capacitatea unui bloc de date tip **Push Constants**. O cerință importantă de care trebuie ținut cont în construcția unei structuri **Push Constants** este să respecte regulile formatului SPIR-V pentru offset și stride [21]. Dimensiunea primitelor **uint32_t** și **float** este de 4 bytes, prin urmare trebuie aliniate în memorie la un multiplu de 4 bytes, iar **glm::vec2** are 8 bytes, așa că trebuie aliniat la un multiplu de 8 bytes. O modalitate de a organiza datele necesare astfel încât se fie respectate regulile enunțate și să aibă cel mai compact mod de a reprezenta toate cele 3 variabile, este cel definit în structura **PushConstants**, ilustrată în Figura 3.1. Această structură păstrează datele contiguu în memorie, fără să aibă goluri de memorie pe post de offset, asigurând coerența cache-ului și acces eficient.

```
1 struct UniformBufferObject
2 {
3     glm::mat4 projection;
4     glm::vec4 staticColor;
5     glm::vec4 dynamicColor;
6 }
```

Push Constants sunt în esență un alt mecanism de transmitere a datelor, care oferă o metodă extrem de rapidă de a furniza valori mici și frecvent actualizate către shaders. Spre deosebire de **Uniform Buffer Objects**, care necesită

alloare explicită, gestionare a memoriei și un set de *binding descriptors*, acestea sunt inserate direct în pipeline-ul grafic și transmise prin comenzi de rendering, eliminând overhead-ul asociat transferurilor de date. Conform documentației [20], Vulkan garantează cel puțin 128 bytes pentru capacitatea unui bloc de date tip **Push Constants**. O cerință importantă de care trebuie ținut cont în construcția unei structuri **Push Constants** este să respecte regulile formatului SPIR-V pentru offset și stride [21]. Dimensiunea primitelor **uint32_t** și **float** este de 4 bytes, prin urmare trebuie aliniate în memorie la un multiplu de 4 bytes, iar **glm::vec2** are 8 bytes, așa că trebuie aliniat la un multiplu de 8 bytes. O modalitate de a organiza datele necesare astfel încât se fie respectate regulile enunțate și să aibă cel mai compact mod de a reprezenta toate cele 3 variabile, este cel definit în structura **PushConstants**, ilustrată în Figura 3.1. Această structură păstrează datele contiguu în memorie, fără să aibă goluri de memorie pe post de offset, asigurând coerența cache-ului și acces eficient.

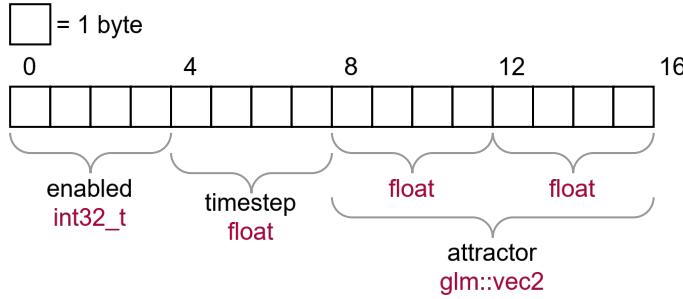


Figura 3.1: Reprezentarea în memorie pentru **Push Constants**

Un **Shader Storage Buffer Object**

(**SSBO**) este folosit pentru a gestiona volume mari de date care trebuie procesate exclusiv pe GPU. Acest tip de buffer este extrem de eficient în contextul unui simulator de particule deoarece permite stocarea directă a datelor în VRAM, ceea ce conferă plăcii video abilitatea de a efectua calculele instanțelor de particulă independent de CPU, reducând astfel latența asociată transferurilor de date. GPU-ul poate accesa aceste date la viteze mult mai mari decât ar fi posibil prin transferuri repetitive între CPU și GPU. În cazul sistemului de particule, SSBO-ul va fi definit folosind datele specifice unei particule, reprezentate prin structura **Particle**. Prima etapă în crearea datelor pentru acest buffer este generarea tututor valorilor de către CPU, urmată de transferul acestora către GPU. În esență, operația mută toate datele din memoria RAM în memoria VRAM. Odată transferate valorile inițiale ale particulelor de la CPU la GPU, placă video preia întreaga responsabilitate a actualizării lor. CPU-ul nu va mai avea niciun fel de acces, nu poate să citească sau să modifice în niciun fel datele din buffer, acesta devenind o resursă exclusivă a plăcii video. O caracteristică importantă a SSBO-urilor este că ele permit fiecărui fir de execuție al GPU-ului să acceseze independent datele particulelor, facilitând astfel paraleлизarea masivă a calculelor.

```

1  struct Particle
2  {
3      glm::vec2 position;
4      glm::vec2 velocity;
5  };

```

3.5 Pipeline

Codul implementează un flux de rendering în două etape distincte, exploatajând pe cât de mult este posibil capabilitățile plăcii video pentru a maximiza performanța. În prima etapă, este folosit pipeline-ul de calcul (compute pipeline), care se ocupă de actualizarea stării particulelor. Acest prim pipeline folosește un *compute shader*, cu ajutorul căruia procesează volume mari de date folosind mii de nuclee de calcul ale plăcii video, evitând astfel calculul lor pe CPU. Mutarea procesării datelor de simulare direct pe placă video aduce o îmbunătățire notabilă și performanțe foarte mari deoarece nu mai este necesară

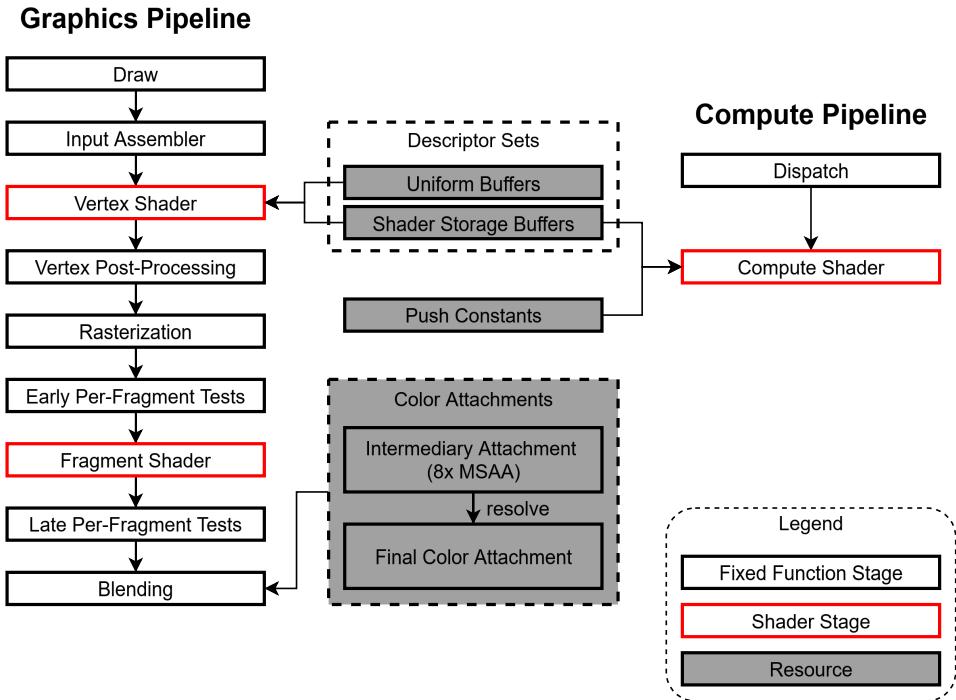


Figura 3.2: Vulkan Pipeline (adaptare după figura din documentație [22])

efectuarea mai multor transferuri costisitoare între memoria principală a CPU-ului și memoria GPU-ului. Astfel, toate datele rămân pe GPU, eliminând latențele asociate cu copierea datelor și permit o execuție paralelă în care sunt utilizate toate resursele hardware de care dispune placa video. În a doua etapă, este folosit pipeline-ul grafic (graphics pipeline), care realizează procesul de rendering a particulelor pe ecran. Acest pipeline este mai complex, incluzând mai mulți pași enumerați în Figura 3.2.

În *compute shader* sunt calculate viteza și poziția fiecărei particule, folosind datele din **Push Constants** referitoare la poziția și starea punctului de atracție, împreună cu ajutorul unui **Shader Storage Buffer Object** în care sunt stocate datele actuale despre particule. Pe baza informațiilor primite, fiecare thread va actualiza valorile unei particule, poziția și viteza. *Vertex shader* este responsabil de a lua datele din SSBO calculate în etapa anterioară, iar pe baza lor, împreună cu informațiile din **Uniform Buffer Object**, determină culoarea și poziția în planul 2D specific fiecărei particule. Nu în ultimul rând, *fragment shader* atribuie fiecărei particule, ce se află în spațiul de rendering al ferestrei principale, culoarea calculată în vertex shader.

Capitolul 4

Analiza performanței și concluzii

4.1 Performanță



Figura 4.1: Vizualizare Benchmark 1 - 8.388.608 particule

În acest capitol voi prezenta rezultatele obținute de sistemul de particule folosind diferite configurații hardware și sisteme de operare, Windows 10/11, respectiv Ubuntu 24.04.2. Evaluarile testelor realizate pot oferi o perspectivă interesantă asupra modului în care sistemele de operare gestionează resursele și rularea aceleiași aplicații. Pe lângă analiza directă dintre cele două sisteme de operare, voi introduce și comparații între diferite tipuri de configurații hardware pentru a putea înțelege scalabilitatea, performanța și adaptabilitatea sistemului de particule. Pentru a putea acoperi diferite scenarii de utilizare și rendering vom folosi două benchmark-uri. Benchmark 1 (Figura 4.1) simulează 8.388.608 de particule, în care majoritatea dintre ele sunt concentrate în poziția centrului

de atracție. Benchmark 2 (Figura 4.2) simulează 8.388.608 de particule, în care ele sunt cât mai răspândite, acoperind aproape întreaga suprafață de vizualizare.

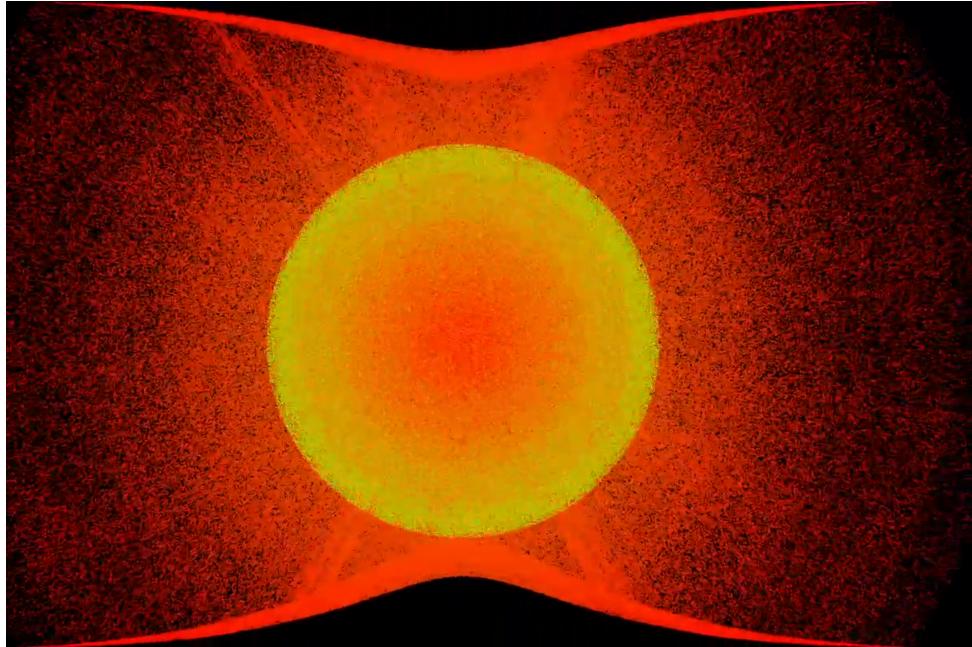


Figura 4.2: Vizualizare Benchmark 2 - 8.388.608 particule

Prima comparație constă în prezentarea rezultatelor celor două benchmark-uri rulate pe Windows 11 (Figura 4.3, Figura 4.5) și Ubuntu 24.04.2 (Figura 4.4, Figura 4.6), ambele având aceeași configurație hardware: procesor Intel i5-8400 4.0 GHz, 16 GB RAM DDR4 2400 MHz CL15, NVIDIA GTX 1060 6 GB GDDR5 cu Boost Clock 1759 MHz. În Benchmark 1 se poate observa că aplicația pe Windows a avut o performanță pe total puțin mai bună decât pe Ubuntu, FPS-ul fiind mai mare pe Windows, dar experiența pe Ubuntu este mult mai stabilă, neavând la fel de multe spike-uri (creșteri brusăte) de performanță ca Windows. Comportamentul descris este confirmat și prin valoările minime ale FPS-urilor, diferența fiind de 8 FPS în defavoarea versiunii de Windows. De asemenea, se poate observa că în momentul în care este procesată și afișată aceeași imagine, versiunea de Ubuntu are cu aproape 9 FPS mai mult decât cea de Windows. În Benchmark 2 se păstrează tendințele înregistrate în Benchmark 1, cu observația că pe Windows s-a atins un număr maxim de FPS cu mult peste valoarea de pe Ubuntu. Din nou, acest efect susține afirmația că Windows este capabil de a atinge performanțe mai bune ca Ubuntu, dar nu într-un mod constant.

A doua comparație constă tot în prezentarea rezultatelor celor două benchmark-uri rulate pe Windows 11 (Figura 4.7, Figura 4.9) și Ubuntu 24.04.2 (Figura 4.8, Figura 4.10), dar cu o configurație hardware foarte performantă: procesor AMD Ryzen 7 9800X3D 5.2 GHz, 32 GB RAM DDR5 6000 MHz CL30, NVIDIA RTX 5080 16 GB GDDR7 cu Boost Clock 2640 MHz. Se observă că tendințele din prima comparație se păstrează, versiunea de Windows oferă performanțe mai bune decât cea de Ubuntu, și sunt diminuate acele spike-

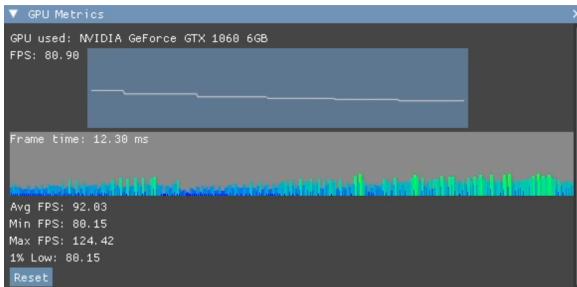


Figura 4.3: Windows Benchmark 1
GTX 1060 6GB

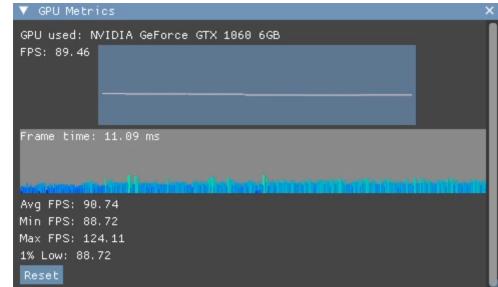


Figura 4.4: Ubuntu Benchmark 1
GTX 1060 6GB

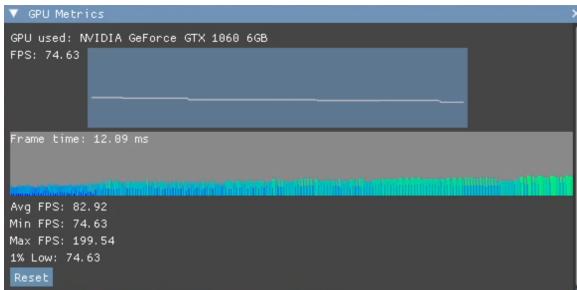


Figura 4.5: Windows Benchmark 2
GTX 1060 6GB

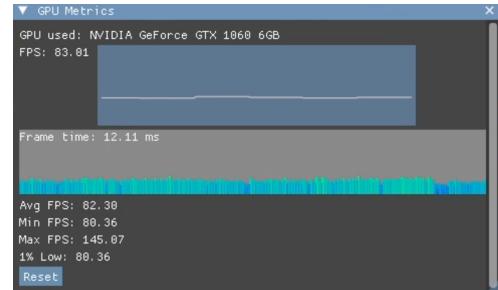


Figura 4.6: Ubuntu Benchmark 2
GTX 1060 6GB

uri de performanță prezente pe prima configurație hardware. Noua configurație oferă un suport mult mai stabil față de cea precedentă, fapt ce se poate datora noilor implementări atât la nivel hardware, cât și software. De asemenea, față de configurația anterioară, acest sistem oferă performanțe de până la trei ori mai mari. Este o diferență uriasă justificată în mare parte de avansul tehnologic al placilor video. GTX 1060 a apărut în anul 2016, iar RTX 5080 a apărut în anul 2025, o diferență de 9 ani însoțită de progres și invenții noi. Raportat la sistemul de particule dezvoltat, acest aspect nuantează că odată cu creșterea performanței, implementarea se va folosi de toate resursele disponibile pe care le poate oferi placa video. Cu trecerea timpului, aplicația va rămâne relevantă pentru că se adaptează odată cu hardware-ul, reflectând în mod direct performanța oricărui dispozitiv.

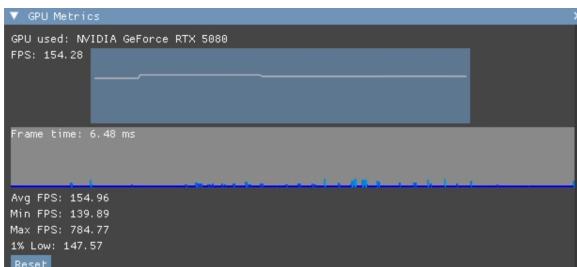


Figura 4.7: Windows Benchmark 1
RTX 5080 16 GB

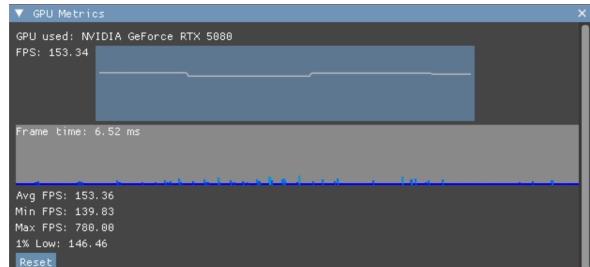


Figura 4.8: Ubuntu Benchmark 2
RTX 5080 16 GB

Ultima comparație constă în prezentarea rezultatelor unui singur benchmark cu un număr variat de particule, rulate pe Windows 10 (Figura 4.11, Figura 4.12), dar cu o

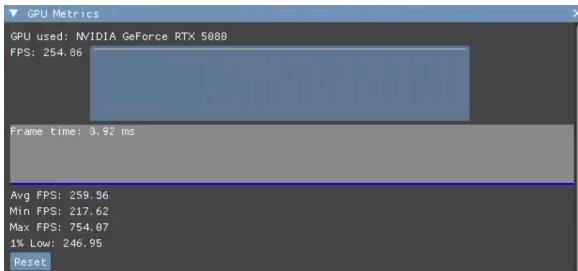


Figura 4.9: Windows Benchmark 2
RTX 5080 16 GB

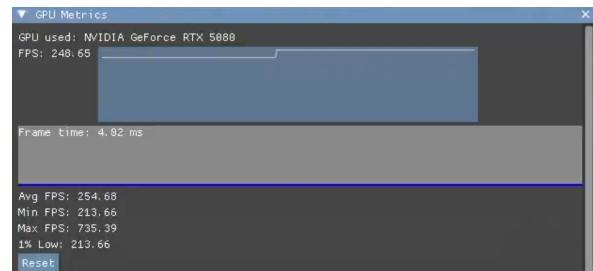


Figura 4.10: Ubuntu Benchmark 2
RTX 5080 16 GB

configurație hardware modestă: procesor AMD Ryzen 5 3500U 3.7 GHz, 8GB RAM DDR4 2400 MHz, AMD Radeon Vega 8. Aceste teste folosesc o placă video integrată, iar asta scade drastic performanța întregului sistem de particule. În Benchmark 2, cu 8.388.608 de particule, aplicația este aproape imposibil de utilizat, având un FPS mult sub pragul necesar pentru *Real-Time Rendering*. Împărțind numărul inițial de particule la 32, cu un total de 262.144, putem ajunge din nou la un FPS acceptabil, puțin sub obiectivul actual de 60 FPS, standardul minim pentru a obține o experiență fluidă în grafica pe calculator. O placă video integrată nu poate face față unui flux atât de mare de lucru și informații. Prin urmare, folosirea unei plăci video dedicate aduce într-adevăr îmbunătățiri semnificative de performanță față de una integrată.



Figura 4.11: Windows - 8.388.608 particule
AMD Radeon Vega 8

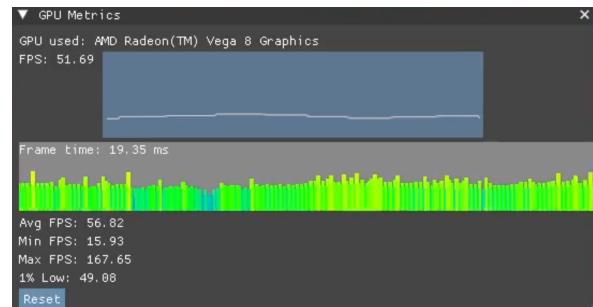


Figura 4.12: Windows - 262.144 particule
AMD Radeon Vega 8

Tabela 4.1 și Tabela 4.2 centralizează toate rezultatele testelor prezentate anterior.

4.2 Limitări

Una dintre principalele limitări ale sistemului de particule este nevoia unei plăci video dedicate sau integrate, suficient de nouă și actualizată pentru a oferi suport complet pentru API-ul Vulkan. În urma testelor efectuate, o altă limitare semnificativă este legată direct de capacitatea hardware-ului folosit. Cu cât acesta este mai capabil, cu atât sistemul poate fi scalat și extins, fără să compromită performanța sau calitatea vizuală. Această constrângere este comună în cazul aplicațiilor cu un nivel ridicat de procesare

computațională, unde creșterea complexității calculelor necesită resurse hardware proporțional mai puternice.

4.3 Directii posibile pentru evoluția temei abordate

Proiectul de față este un punct de plecare foarte bun pentru a crea un sistem de particule mult mai complex și mai spectaculos din punct de vedere grafic, capabil să simuleze cu un grad ridicat de realism diferite fenomene fizice și efecte vizuale. Se poate face tranziția la un sistem de particule în 3D și la simularea unor fenomene fizice complexe precum simularea materialelor textile, focului, fumului, lichidelor. Un alt obiectiv important ar fi integrarea unui sistem care să permită utilizatorilor să definească comportamentul particulelor prin shaders personalizabile. Acest lucru ar putea fi realizat prin intermediul unui editor integrat în aplicație, care să includă funcționalități de hot-reload pentru shaders, permitând compilarea acestora on-the-fly, fără a mai fi necesar recompilarea întregii aplicații pentru a reflecta noile modificări aduse fișierelor care definesc comportamentul pentru diferite shaders: compute, vertex, fragment. Un alt aspect interesant, care trebuie luat în calcul, este suportul pentru mai multe platforme. API-ul Vulkan permite integrarea aplicației și pe macOS sau iOS prin utilizarea MoltenVK [23], un subset al API-ului Vulkan peste framework-ul grafic Metal dezvoltat de Apple exclusiv doar pentru producție. O altă platformă care include suport pentru Vulkan și pentru care ar putea fi extinsă aplicația curentă, este Android. Nu în ultimul rând, un aspect crucial al aplicației este dat de alegerea API-ului grafic folosit în implementare. Orice API are puncte forte și arii în care excelează, în funcție de arhitectura hardware-ului și cerințele aplicației. Un alt API poate oferi avantaje în anumite contexte, cum ar fi optimizări specifice pentru un sistem de operare și componentele hardware din spate. De aceea, introducerea și implementarea altor API-uri grafice moderne, în special DirectX 12 Ultimate [24] și Metal [25], ar putea aduce noi perspective asupra tehniciilor de optimizare și arhitecturii folosite.

Configurație Hardware	Indicator	Benchmark 1		Benchmark 2	
		Windows 11	Ubuntu 24	Windows 11	Ubuntu 24
GTX 1060 6GB	AVG FPS	92.03	90.74	82.92	82.30
	MIN FPS	80.15	88.72	74.63	80.36
	MAX FPS	124.42	124.11	199.54	145.07
	1% Low FPS	80.15	88.72	74.63	80.36
RTX 5080 16GB	AVG FPS	154.96	153.36	259.36	254.68
	MIN FPS	139.89	139.83	217.62	213.66
	MAX FPS	784.77	780.00	754.07	735.39
	1% Low FPS	147.57	146.46	246.95	213.66

Tabela 4.1: Comparație performanță Benchmark 1 și 2

Configurație Hardware	Indicator	Particule	
		8.388.608	262.144
AMD Radeon Vega 8	AVG FPS	15.03	56.82
	MIN FPS	3.83	15.93
	MAX FPS	22.64	167.65
	1% Low FPS	3.83	49.08

Tabela 4.2: Comparație cu un număr diferit de particule

Bibliografie

- [1] Pixar Animation Studios - "Each frame in this scene took about 50 hours to render. Pixar has a huge "render farm," which is basically a supercomputer composed of 2000 machines, and 24,000 cores. This makes it one of the 25 largest supercomputers in the world. That said, with all that computing power, it still took two years to render Monster's University.", URL: <https://sciencebehindpixar.org/pipeline/rendering>.
- [2] Documentația oficială Unreal Engine, URL: <https://www.unrealengine.com/en-US>.
- [3] Documentația oficială Unreal Engine, URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/gpusprites-type-data?application_version=4.27.
- [4] Documentația oficială Unreal Engine, URL: <https://dev.epicgames.com/documentation/en-us/unreal-engine/how-to-create-a-gpu-sprite-effect-in-niagara-for-unreal-engine>.
- [5] Documentația oficială GIT, URL: <https://git-scm.com/>.
- [6] Pagina web a proiectului, URL: <https://github.com/sebimih13/Particle-System>.
- [7] Documentația oficială Premake5, URL: <https://premake.github.io/>.
- [8] Documentația oficială VulkanSDK Windows, URL: https://vulkan.lunarg.com/doc/sdk/latest/windows/getting_started.html.
- [9] Documentația oficială VulkanSDK Linux, URL: https://vulkan.lunarg.com/doc/sdk/latest/linux/getting_started.html.
- [10] Documentația oficială LUA, URL: <https://www.lua.org/>.
- [11] Documentația oficială Microsoft Visual Studio, URL: <https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2022>.
- [12] Documentația oficială GNU Make, URL: <https://www.gnu.org/software/make/manual/make.html>.

- [13] Pagina web de unde poate fi descărcată aplicația, URL: <https://github.com/sebimih13/Particle-System/releases>.
- [14] Documentația oficială Microsoft MSBuild, URL: <https://learn.microsoft.com/en-us/visualstudio/msbuild/msbuild?view=vs-2022>.
- [15] Documentația oficială ImGui, URL: <https://www.dearimgui.com/>.
- [16] Documentația oficială nlohmann JSON, URL: https://json.nlohmann.me/api/basic_json/.
- [17] Platforma web oficială a organizației Khronos Group, URL: <https://www.khronos.org>.
- [18] Documentația oficială GLFW, URL: <https://www.glfw.org/>.
- [19] Documentația oficială GLM, URL: <https://glm.g-truc.net/0.9.9/>.
- [20] Documentația oficială Vulkan, secțiunea *Limit Requirements* - Table 2. Required Limits, maxPushConstantsSize - Limit Type = min 128 (Vulkan Core) / 256 (Vulkan 1.4), URL: <https://docs.vulkan.org/spec/latest/chapters/limits.html#limits-minmax>.
- [21] Documentația oficială Vulkan, secțiunea *Offset and Stride Assignment* - "A scalar of size N has a scalar alignment of N", URL: <https://docs.vulkan.org/spec/latest/chapters/interfaces.html#interfaces-resources-layout>.
- [22] Documentația oficială Vulkan, URL: <https://docs.vulkan.org/spec/latest/chapters/pipelines.html>.
- [23] Documentația oficială MoltenVK, URL: <https://www.moltengl.com/docs/readme/moltenvk-readme-user-guide.html>.
- [24] Documentația oficială Microsoft DirectX 12 Ultimate, URL: <https://www.nvidia.com/en-us/geforce/technologies/directx-12-ultimate/>.
- [25] Documentația oficială Apple Metal, URL: <https://developer.apple.com/metal/>.
- [26] John van der Burg, „Building an Advanced Particle System”, în (Iun. 2000).
- [27] Michael J. Flynn și Kevin W. Rudd, „Parallel Architectures”, în (Mar. 1996), URL: <https://dl.acm.org/doi/pdf/10.1145/234313.234345>.
- [28] Jorge Jimenez, Diego Gutierrez, Jason Yang, Alexander Reshetov, Pete Demoreuille, Tobias Berghoff, Cedric Perthuis, Henry Yu, Morgan McGuire, Timothy Lottes, Hugh Malan, Emil Persson, Dmitry Andreev și Tiago Sousa, „Filtering Approaches for Real-Time Anti-Aliasing”, în *ACM SIGGRAPH Courses*, 2011, URL: <https://www.iryoku.com/aacourse/>.

- [29] Graham Sellers; with contributions from John Kessenich, *Vulkan Programming Guide: The Official Guide to Learning Vulkan*, Addison-Wesley Professional, pp. 21–22.
- [30] Lutz Latta, „Building a Million-Particle System”, în (Iul. 2004).
- [31] William T. Reeves, „Particle Systems A Technique for Modeling a Class of Fuzzy Objects”, în (Iul. 1983).
- [32] Adam Sawicki, „An Idea for Visualization of Frame Times”, în (Mai 2022), URL: https://asawicki.info/news_1758_an_idea_for_visualization_of_frame_times.
- [33] Bjarne Stroustrup, *The C++ Programming Language - Fourth Edition*, Secțiunea 13.3 *Resource Management*, Addison-Wesley, pp. 354–357.
- [34] Juan Zhang, „Implementation and Optimization of Particle Effects based on Unreal Engine 4”, în (2020), URL: <https://iopscience.iop.org/article/10.1088/1742-6596/1575/1/012187/pdf>.