



ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ

**Факултет Компютърни Системи и Технологии
Катедра Компютърни Системи**

маг. инж. Георги Георгиев Георгиев

**ПЛАТФОРМЕНО НЕЗАВИСИМИ ПРИЛОЖЕНИЯ С
ИЗПОЛЗВАНЕ НА GPU И МАШИННО ОБУЧЕНИЕ**

А В Т О Р Е Ф Е Р А Т

на дисертация за придобиване на образователна и научна степен
"ДОКТОР"

Област: 5. Технически науки

Професионално направление: 5.3. Комуникационна и компютърна техника

Научна специалност: Системно програмиране

Научни ръководители:
Проф. д-р Милена Лазарова-Мицева
Проф. д-р Даниела Гоцева

СОФИЯ, 2026 г.

Дисертационният труд е обсъден и насочен за защита от Катедрения съвет на катедра „Компютърни системи“ към Факултет Компютърни системи и технологии на ТУ-София на редовно заседание, проведено на 22.06.2026 г..

Публичната защита на дисертационния труд ще се състои на __. __.2026 г. от __, __ часа в Конферентната зала на БИЦ на Технически университет – София на открито заседание на научното жури, определено със заповед № ОЖ-.... / __. __.2026 г. на Ректора на ТУ-София в състав:

1. Проф./доц. д-р/д.т.н. _____ – председател
2. Проф./доц. д-р/д.т.н. _____ – научен секретар
3. Проф./доц. д-р/д.т.н. _____
4. Проф./доц. д-р/д.т.н. _____
5. Проф./доц. д-р/д.т.н. _____

Рецензенти:

1. Проф./доц. д-р/д.т.н. _____
2. Проф./доц. д-р/д.т.н. _____

Материалите по защитата са на разположение на интересуващите се в канцеларията на Факултет Компютърни системи и технологии на ТУ-София, блок №, кабинет № _____.

Дисертантът е задочен докторант към катедра „Компютърни системи“ на факултет Компютърни системи и технологии. Изследванията по дисертационната разработка са направени от автора, като някои от тях са подкрепени от научноизследователски проекти.

Автор: маг. инж. Георги Георгиев

Заглавие: *“Платформено независими приложения с използване на GPU и машинно обучение”*

Тираж: 30 броя

Отпечатано в ИПК на Технически университет – София

I. ОБЩА ХАРАКТЕРИСТИКА НА ДИСЕРТАЦИОННИЯ ТРУД

Актуалност на проблема

Графичните процесори (GPU) претърпяват фундаментална промяна в своята роля през последното десетилетие. От специализиран хардуер за рендиране на изображения с приложения в областите на видеоигрите, виртуалната реалност, киното и инженерното проектиране, тяхната паралелна изчислителна архитектура позволява високопроизводителни имплементации на алгоритми с общо предназначение, често надвишавайки възможностите на централните процесори. Парадигмата GPGPU позволява използване на графични техники като проследяване на лъчи в реално време, която създава много по-точни симулации на взаимодействията на светлината с обекти в триизмерно пространство спрямо традиционната растеризация. Хилядите изчислителни ядра на даден чип също са в основата на динамичният растеж и развитие на машинното обучение и изкуствения интелект, които освен в сферите на обработка на т. нар. големи данни, езикови модели и компютърно зрение, имат приложение и в компютърната графика. Еволюцията на графичните процесори е свързана със софтуерните интерфейси за тяхното управление. В резултат на исторически обособените различия, GPU-базираните приложения често остават обвързани към определена хардуерна или софтуерна среда, или пък изискват усвояването на множество програмни интерфейси и внедряване на сложни абстрактни слоеве в програмната архитектура. Това ограничава възможността за пълноценно оползотворяване на все по-актуалните паралелни изчислителни устройства, особено при малки екипи от разработчици или научни изследователи.

Цел на дисертационния труд, основни задачи и методи за изследване

Този дисертационен труд цели да изследва възможностите за създаване на унифициран и платформено независим софтуер за създаване на приложения с използване на GPU в областта на компютърната графика и интегрирането им с машинно обучение.

Задачи:

1. Създаването на платформено независима архитектура, позволяваща разработването на приложения с използване на GPU, без съществено ограничаване на производителността спрямо платформено специфичните решения.
2. Разработване на единен програмен модел за графични и изчислителни задачи намалявайки сложността при управлението на ресурсите и интеграцията между отделните компоненти на приложението, като същевременно се постига преносимост между различни хардуерни и софтуерни платформи.
3. Изследване и прилагане на подходящи оптимизационни алгоритми за проследяването на светлинни лъчи в реално време с цел постигане на производителност, подходяща за интерактивни приложения, без съществено влошаване на качеството на визуализацията.
4. Изследване на методи с използване на машинно обучение за компенсиране на ограниченията, произтичащи от изчислителната сложност на фотореалистичното рендиране за ефективно скалиране към високи разделителни способности и генериране на междинни кадри при запазване на приемливо визуално качество.
5. Комбинирането на платформено независим графичен конвейер, оптимизирано проследяване на лъчи и техники от машинното обучение за изграждането на практически приложими системи за графика в реално време, които използват

ефективно възможностите на съвременните графични процесори независимо от конкретната хардуерна платформа.

Научна новост

Научната новост се изразява в създаването на концептуален модел за изграждане на платформено независима система за рендиране на изображения в реално време, който се основава на унифициран програмен модел за графични и изчислителни операции, и предложен функционално-императивен подход за разработване на високопроизводителни графични приложения в реално време, стъпващ на безопасно управление на ресурсите, модулност и ефективно използване на съвременните многоядрени архитектури.

Практическа приложимост

Изграден е прототип на платформено независима графична система за рендиране в реално време чрез проследяване на пътя на лъчите (path tracing). Проведените с него тестове показват, че производителността е еднаква или напълно съпоставима при изпълнение върху Windows и Linux. По експериментален път е потвърдено, че системата постига сравнима производителност на различни операционни системи и графични среди, без да разчита на платформено-специфични реализации.

Практическата стойност на тези резултати се изразява във възможността за създаване на високопроизводителни приложения с реалистично осветление в реално време – например кросплатформени игри, интерактивни архитектурни и инженерни визуализации, симулации за филмовата индустрия и научни среди, които работят с еднаква ефективност на Windows и Linux без нужда от отделни оптимизации за всяка платформа.

Апробация

Дисертационният труд е докладван и обсъден на катедрен съвет на катедра “Компютърни системи” при ТУ – София. Части от темата са докладвани на:

- XI-та Международна Научна Конференция „Компютърни науки“ – COMSCI 2023
- 60-та Международна научна конференция по информационни, комуникационни и енергийни системи и технологии ICEST 2025
- XIII-та Международна Научна Конференция „Компютърни науки“ – COMSCI 2025

Публикации

Основни постижения и резултати от дисертационния труд са публикувани в 4 научни статии, една от които самостоятелна. Четирите публикации са индексирани в Scopus, едната от тях е индексирана и в WoS.

Структура и обем на дисертационния труд

Дисертационният труд е в обем от **135** страници, като включва увод, **4** глави за решаване на формулираните основни задачи, списък на основните приноси, списък на публикациите по дисертацията и използвана литература. Цитирани са общо **108** литературни източници, като всички са на латиница. Работата включва общо **63** фигури и **4** таблици. Номерата на фигурите и таблиците в автореферата съответстват на тези в дисертационния труд.

II. СЪДЪРЖАНИЕ НА ДИСЕРТАЦИОННИЯ ТРУД

УВОД

Развитието на графичните процесори е тясно обвързано със софтуерните интерфейси, които ги управляват. Поради историческата им дивергенция, приложенията с използване на GPU често са обвързани към конкретна хардуерна или софтуерна платформа или налагат изучаване на множество приложно програмни интерфейси и въвеждането на сложни абстракции в архитектурата на програмата. Това възпрепятства използването на пълния потенциал на нарастващите по релевантност паралелни изчислителни устройства, особено за по-малки екипи от разработчици или научно изследователи.

Обект на изследването в дисертационния труд са приложения с използване на графични процесори за генериране на изображения в реално време и изпълнение на алгоритми, базирани на машинно обучение.

Предмет на изследването в дисертационния труд са методите, архитектурните решения и алгоритмите за изграждане на платформено независими приложения, използващи графични процесори, включително организацията на графичните и изчислителните процеси чрез единен програмен модел, оптимизацията на алгоритми за проследяване на светлинни лъчи в реално време и прилагането на методи на машинното обучение за скалиране към високи разделителни способности и генериране на кадри.

ГЛАВА 1. СЪВРЕМЕННО СЪСТОЯНИЕ НА ИНТЕРФЕЙСИ ЗА РАБОТА С ГРАФИЧНИ ПРОЦЕСОРИ

В първа глава на дисертационния труд се съпоставят текущите версии на приложно програмните интерфейси за компютърна графика и изчисления с общ характер, като се разглежда съвременната архитектура, трудностите при постигане на максимално използване на изчислителен капацитет и разликите при различните платформи.

1.1 Основни принципи на програмирането за GPU архитектури

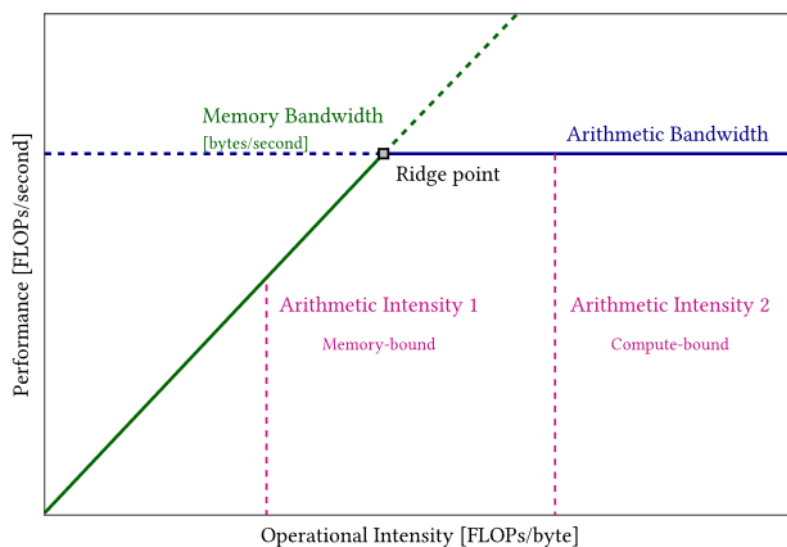
Графичните процесори са оптимизирани за висока пропускателна способност чрез масивна паралелност за разлика от CPU, където развитието се проявява в минимизирането на времето за изпълнение на единична инструкция чрез увеличаване на тактовата честота. Хилядите изчислителни ядра се обединяват в изпълнение на дадена задача по модела Single-Instruction Multiple-Threads (SIMT). Наблюдават се разлики в използваната терминология в зависимост от конкретната архитектура на, но общият принцип на изпълняване на еднаква инструкция едновременно е общовалиден. Принципът на запълване на заетост на нишките е ключов за постигане на добра производителност.

Освен изчислителни ядра, високопроизводителните графични процесори разполагат с вградена памет, за да се избегне ненужна комуникация със системната памет през шината PCIe (Peripheral Component Interconnect Express). Като общ принцип при програмирането за GPU може да се отбележи, че каквато и да било комуникация през PCIe е желателно да бъде минимизирана.

Структурата на вградената памет в GPU е отново зависима от конкретния производител, но някои компоненти са универсални и наподобяват паметта на централните процесори. Обикновено са включени регистри (най-бързата памет, всяка нишка разполага със собствен), споделена локална памет / L1 (относително много бърза, споделена за всеки блок), L2 кеш (по-голям размер, споделена за всеки блок) и глобална памет.

Глобалната памет е с голям капацитет, но обикновено не е интегрирана в самия чип, което води до значителен недостатък: макар и с висока пропускателна способност, латентността е относително голяма. Латентността при достъп до паметта в общия случай може да бъде маскирана чрез многоредово изпълнение. Съответно моделът на

програмиране е throughput-ориентиран, целта е ALU ядрата и бързата кеш памет да бъдат максимално ангажирани, за да бъде избегнато забавяне при изчакване на данни от паметта (т.нар. memory bottleneck). Алгоритмите за GPU трябва да бъдат моделирани с цел постигане на максимална заетост (occupancy), като заетостта се дефинира като съотношението между активните thread warps на даден блок към теоретично възможните. От друга страна е възможно и пренасищане на изчислителния капацитет, регистрите и споделената памет при изпълнение на твърде много warps за даден блок. Балансът между двата вида насищане е показан на фиг. 2, където Memory Bandwidth представлява максималната пропускателна способност на паметта, а Arithmetic Bandwidth максималния капацитет за изчислителни операции. Изчислителни ядра, които попадат отляво на седловата точка са ограничени по отношение на памет, а отдясно – по изчислителен капацитет:



Фиг. 2 Модел на отношението между изчислително насищане и теоретично постижима производителност

Друга особеност при GPU програмирането е планирането на изпълнението. Модерните графични процесори разполагат с динамични диспечери на задачите (schedulers) аналогични на тези за CPU при операционните системи с многопроцесни възможности. Целта им е избиране на най-подходящи нишки или warps. Конкретните имплементации са платформено зависими, но общият принцип е характерен за всички GPU. Нишка може да бъде избрана само ако всички данни са налични и налице са нужните изпълнителни единици, като добра практика е да се поддържа достатъчно голям брой warps в състояние в готовност за изпълнение, за да не се намалява заетостта на диспечера.

При оптимизация на алгоритми в това отношение е необходимо да се спомене дивергенцията при изпълнение. При условно разклоняване на нишките в даден warp (if branch), нишките поемат по различни пътеки на изпълнение на изчислителната логика. Графичният процесор изпълнява всички разклонения, но нишките, които не се намират в активния път, биват временно „изключвани“. Това води до голям спад в пропускателната способност, тъй като нишките фундаментално са обвързани в даден warp и няма възможност диспечерът да зададе задача на само част от тях преди всичките да са приключили. Тук се наблюдава значителна разлика спрямо централните процесори, които обикновено разполагат с хардуерно имплементиран модул за предвиждане на разклонено изпълнение, но дори в случаите, когато правилният път не бъде избран, цената по отношение изгубено време за изпълнение е много по-ниска при CPU поради по-високата тактова честота. Т.нар. branch miss при централните процесори също не води до същата

загуба на пропускателна способност, тъй като е на нивото на една нишка, докато при GPU тя е ниво warp от нишки. Вероятно най-фундаменталната разлика в моделирането на паралелен код за централен и графичен процесор е именно избягването на разклонения.

1.2 Разлики в архитектурите на производителите

Гореописаните характеристики на GPU програмирането са общовалидни за почти всички съвременни хардуерни устройства на основата на SIMT архитектурата. Въпреки приликите, за постигане на добри резултати при имплементация на платформени независими GPU приложения е необходимо да се отчетат и различията от гледна точка на налични модули за хардуерно ускорение на дадени операции и приложно програмните интерфейси. Целта е да бъдат установени общи характеристики, които могат да бъдат използвани на всички платформи за постигане на максимална производителност и при възможност да не се налага писането на платформено зависим код. Текущите производители на графични процесори са NVIDIA, AMD, Intel и Apple.

Обобщение на характеристиките на графичните процесори на различните производители е показано в табл. 1:

Производител	Ядра/SM	Ray/Tensor	Памет	API
NVIDIA	128 SM	RT/Tensor	GDDR7 дискретна	CUDA, Vulkan, OpenCL
AMD	64 CU	RA/CDNA	GDDR6 дискретна	Vulkan, OpenCL
Intel	Без групиране в SM	RTU/XMX	GDDR6 дискретна	Vulkan, OpenCL
Apple	32 шейдър	Metal RT/Compute	LPDDR5 обединена	Metal, Vulkan (MoltenVK)

Табл. 1 Сравнение на характеристиките на текущи GPU

По отношение на разработка на платформени независими приложения за модерни графични процесори могат да се направят следните изводи: Специализирани ядра за изчисления с общ характер и хардуерни ускорители за проследяване на лъчи са налични за всички платформи. По-разпространената структура на паметта е дискретна, следователно използването на глобална памет трябва да бъде съобразено с повишено време за комуникация спрямо интегрирана в чипа. Наблюдават се разлики в наличните имплементации на приложно програмни интерфейси.

1.3 Приложно програмни интерфейси за компютърна графика

Особеностите на отделните интерфейси се изразяват в нивото на достъп до хардуера, конструирането на графичния конвейер и многоплатформената поддръжка от драйвърите.

Direct3D осигурява достъп на ниско ниво до графичния процесор, което позволява на разработчиците прецизно настройване и оптимизиране на производителността при рендиране. Интерфейсът предоставя механизми за управление на ресурси като буфери, текстури и шейдъри, както и средства за контрол върху използването на паметта и ефективността на графичния конвейер. Конвейерът на Direct3D поддържа многонишкова обработка, което позволява паралелизация на задачите по рендиране и максимално използване на възможностите на съвременните многоядрени централни процесори. Ръчното управление на голяма част от ресурсите и възможността за програмна промяна на поведението на части от конвейера са характерни за графичните интерфейси от ниско ниво и представляват основната разлика между D3D12 и предходния D3D11. Това елиминира допълнителните разходи от конвертиране на ниво драйвър (т.нар. driver

overhead) и води до теоретично подобряване на производителността. Основен недостатък на Direct3D обаче е липсата на поддръжка от операционни системи различни от Windows, което налага използване на слоеве за трансляция.

OpenGL е платформено независим приложно програмен интерфейс с архитектура от сравнително по-високо ниво спрямо Direct3D и Vulkan. Графичният му конвейер е значително по-прост. Поради възрастта си (първа версия от 1992г.) интерфейсът не е съобразен с многоядрени процесорни архитектури. Всички ресурси в OpenGL се управляват на принципа на краен автомат (state machine).

Vulkan е проектиран с оглед на междуплатформена съвместимост и по-подходяща архитектура за съвременен хардуер спрямо OpenGL, като поддържа широк спектър от операционни системи, включително Windows, Linux, macOS (чрез MoltenVK) и Android. Програмният модел е изключително експлицитен, дори до по-голяма от степен от Direct3D 12.

1.4 Приложно-програмни интерфейси за изчисления с общ характер

Наред с приложно-програмните интерфейси за компютърна графика, развитието на графичните процесори като платформи за изчисления с общ характер (General-Purpose Computing on Graphics Processing Units, GPGPU) води до появата на специализирани програмни модели, предназначени за използване на масивния паралелизъм на съвременните GPU архитектури. Най-разпространените решения в тази област са CUDA и OpenCL, които предлагат сходни концепции за организация на паралелното изпълнение, но се различават съществено по отношение на степента на платформена независимост, нивото на абстракция и достъпа до специфични хардуерни възможности.

CUDA (Compute Unified Device Architecture) е програмна платформа и приложно-програмен интерфейс, разработени от NVIDIA през 2007 г. Основната цел на CUDA е предоставяне на унифициран модел за разработване на високопроизводителни приложения върху графичните процесори на производителя. Основният недостатък на CUDA произтича именно от нейната обвързаност с конкретен производител. Приложенията, разработени чрез CUDA, могат да бъдат изпълнявани единствено върху графични процесори на NVIDIA. Това затруднява разработката на преносими решения и ограничава приложимостта на създадения софтуер в хетерогенни изчислителни среди.

Като алтернатива на CUDA през 2009 г. консорциумът Khronos Group представя OpenCL (Open Computing Language) – отворен стандарт за паралелни изчисления върху хетерогенни системи. OpenCL е проектиран с идеята да осигури единен интерфейс за използване на различни типове изчислителни устройства, включително графични процесори, централни процесори, цифрови сигнални процесори и системи с програмируема логика. OpenCL дефинира абстрактен модел на платформата, който разделя хост устройството от изчислителните устройства. Взаимодействието между тях се осъществява чрез контексти, командни опашки и буфери в паметта. Създаването и управлението на тези обекти е експлицитно и в много отношения наподобява философията на Vulkan, при която контролът върху ресурсите се предоставя на разработчика. Това увеличава сложността на кода, но същевременно позволява адаптиране към широк набор от хардуерни конфигурации.

Съпоставката между CUDA и OpenCL показва фундаменталния компромис между максимална производителност и платформена независимост. CUDA предоставя по-зряла екосистема, богата документация и възможност за използване на всички специфични характеристики на графичните процесори на NVIDIA. OpenCL, от своя страна, предлага по-голяма преносимост и възможност за разработване на приложения, предназначени за хетерогенни изчислителни среди. Въпреки това, както OpenCL, така и CUDA са проектирани предимно като интерфейси за изчисления с общ характер и не предоставят естествено обединение между графичния и изчислителния конвейер.

В настоящия дисертационен труд основно изискване е разработването на платформено независима система за графика в реално време, която да интегрира в единна

архитектура както традиционното растеризиране, така и изчислителни алгоритми с общ характер, включително техники за проследяване на лъчи и последваща обработка на изображенията. Поради тази причина използването на CUDA не е подходящо, тъй като би ограничило приложимостта на разработената система единствено до графични процесори на NVIDIA. От друга страна, въпреки своята платформена независимост, OpenCL представлява отделен изчислителен стек, чието интегриране с графичните приложно-програмни интерфейси е свързано с допълнителна сложност при споделянето на ресурси, синхронизацията и управлението на паметта.

Съвременните версии на Vulkan предоставят възможност за изпълнение на изчислителни задачи чрез Vulkan Compute, използвайки същия механизъм за управление на устройства, памет, командни буфери и синхронизация, който се използва и при графичното рендиране. Това позволява изграждането на единна архитектура, в рамките на която графичните и изчислителните операции използват общ модел на ресурсите и могат да бъдат комбинирани без необходимост от междинни слоеве или допълнителни програмни интерфейси. Наличието на широка поддръжка от съвременните графични процесори и възможността за използване на разширения за хардуерно ускорено проследяване на лъчи превръщат Vulkan Compute в подходящ избор за реализиране на платформено независима система за графика в реално време.

1.5 Изводи към първа глава

Въз основа на извършения анализ на съвременните графични архитектури и приложно-програмни интерфейси могат да бъдат формулирани следните изводи:

1. Съвременните графични процесори използват сходни принципи на организация, базирани на SIMT модела на изпълнение, йерархична структура на паметта и масивен паралелизъм, независимо от различията в терминологията и конкретните имплементации при отделните производители.

2. Производителността на GPU алгоритмите зависи в значителна степен от ефективното използване на наличните изчислителни ресурси чрез поддържане на висока заетост на изпълнителните единици, минимизиране на достъпа до глобална памет и ограничаване на дивергенцията при условните разклонения.

3. Независимо от архитектурните различия между NVIDIA, AMD, Intel и Apple, всички съвременни графични процесори предоставят програмируеми изчислителни ядра и специализирани модули за ускоряване на определени операции, което позволява дефиниране на общи принципи за разработване на платформено независими алгоритми.

4. OpenGL не удовлетворява изискванията на съвременните високопроизводителни графични приложения поради ограниченията на своя модел за управление на състоянието, недостатъчната пригодност за многонишково използване и липсата на поддръжка на актуални хардуерни възможности, включително хардуерно ускорено проследяване на лъчи.

5. Direct3D 12 предоставя необходимото ниво на контрол върху ресурсите и висока производителност, но обвързаността му с платформите на Microsoft го прави неподходящ като основа за реализиране на действително платформено независима система.

6. CUDA предлага зряла екосистема и възможност за използване на всички специфични хардуерни възможности на графичните процесори на NVIDIA, но зависимостта от един производител ограничава приложимостта ѝ при разработване на преносими решения.

7. OpenCL осигурява по-висока степен на хардуерна независимост, но използването му като отделен изчислителен стек води до допълнителна сложност при интеграцията с графичните програмни интерфейси, споделянето на ресурси и синхронизацията между графични и изчислителни задачи.

8. Vulkan съчетава ниско ниво на достъп до хардуера, широка поддръжка от различни производители и операционни системи и възможност за използване както на графични, така и на изчислителни операции в рамките на единен програмен модел.

9. Vulkan Compute предоставя общ механизъм за управление на паметта, командните буфери и синхронизацията между графичния и изчислителния конвейер, което позволява реализиране на комплексни алгоритми без необходимост от използване на допълнителни програмни интерфейси или слоеве за транслация.

10. Vulkan и неговите изчислителни възможности чрез Vulkan Compute представляват най-подходящата технологична основа за разработването на платформено независима система за графика в реално време.

ГЛАВА 2. КОНЦЕПТУАЛЕН МОДЕЛ И АРХИТЕКТУРА НА ПЛАТФОРМЕНО НЕЗАВИСИМА СИСТЕМА ЗА ГРАФИКА В РЕАЛНО ВРЕМЕ

В тази глава се разглежда модел за проектиране графични приложения с използване на проследяване на лъчи чрез приложно програмния интерфейс Vulkan. Съображения са платформена независимост, производителност, надеждност, паралелизъм и намаляване на сложността при поддръжка. С тази цел следва да бъдат съпоставени програмни езици и парадигми за създаване на софтуер. Тъй като инструкции към графичния процесор се управляват от CPU, архитектурата е зависима от тази част на софтуера.

2.1 Сравнителен анализ на програмни езици за създаване на графични приложения

Програмните езици могат да бъдат категоризирани по множество начини. С оглед на горе поставените изисквания категорията, която значително редуцира броя кандидат езици е производителност. В това отношение езиците могат да се разделят на такива с ръчно управление на паметта и на такива с автоматизирано. Управлението на паметта има отношение и към надеждността, тъй като съществува голям набор от известни уязвимости породени от грешки при ръчно управление. Те обаче не са гарантирани, в частност се наблюдават и при езиците с автоматична колекция на ресурси, и съществуват техники за избягването им, които следва да бъдат разгледани.

Някои парадигми за сигурно системно програмиране (MISRA, CERT, JSF) забраняват динамично заделяне на памет след инициализация на процеса. Това обаче не е приложима стратегия при графичното програмиране в общия случай поради ограничения в дискретната памет на GPU. Налага се ресурси за отделните сцени като геометрични примитиви, текстури, индекси и т.н. да бъдат динамично зареждани от файлове и предавани през PCIe на глобалната памет. Управлението на Vulkan обекти като буфери, устройства и синхронизиращи примитиви също по дизайн е динамично и се случва по време на изпълнение, а данните в тях се зареждат от страна на оперативната памет посредством CPU. Избягването на динамично разделяне в тези случаи е неизбежно, което води до разглеждането на стратегии за управление на паметта при различните програмни езици.

2.2 Езици с автоматично управление на паметта

Езиците с Garbage Collection (GC) разчитат на т.нар. managed heap – блок последователна памет за динамично заделяне на ресурси. Паметта не се управлява ръчно, а на принципа на броенето на референции. Стековите променливи се унищожават щом излязат от обхват (scope). Аналогично на това обхватът за всеки обект в heap паметта се дефинира от броя референции сочещи към него. Когато този брой стане нула, GC освобождава паметта.

Колекторът изпълнява алгоритъм известен като mark-and-sweep. Идентифицират се кои обекти все още са референцирани (live) и кои се намират във вече недостъпни участъци от кода (garbage). Неизползваната памет се маркира като свободна. Блоковете свободна памет се обединяват при връщането им в менажирания heap с цел избягване на фрагментация на паметта. Фрагментацията е нежелан феномен, при който свободни участъци от паметта биват прекъснати от заети. Той се наблюдава както при GC езиците, така и при тези с ръчно управление на паметта. При програмните езици с ръчно управление на памет са налице стратегии за избягване на фрагментация, но при

менажираните езици това зависи изцяло от имплементацията на GC. Алгоритмите за обединяване на блокове в тази среда се случват в runtime и съответно отнемат от общата производителност.

Освен структурирането на програмата около избягване на фрагментация на паметта, при използване на езици с GC има и ситуации, в които се налага ръчно управление на ресурси извън managed heap. От значение тук за графичните приложения са файлови дескриптори и GPU буфери. Те не могат да бъдат управлявани от GC без модификации по кода на съответния runtime, тъй като той отговаря единствено за системната памет, а не за дискретната видеопамет.

Достига се до изводът, че автоматизиран Garbage Collection не е подходящо решение за управление на системните и графичните ресурси в контекста на високопроизводителни GPU приложения.

2.3 Езици с ръчно управление на паметта

В сферата на графичния и общоизчислителен софтуер с използване на графични процесори системният език C++ се е утвърдил като де факто стандарт. Наблюдава се диспропорционално използване на C++ при разработка на софтуер за 3D визуализация, компютърни игри и машинно обучение спрямо други системни езици. Това налага по-подробен и задълбочен анализ на съвременния C++ стандарт с цел да бъде установено, дали съвместяването му с интерфейса Vulkan е най-подходящото решение за създаване на платформено независими GPU приложения.

Съвременният стандарт не следва единен модел, а често предлага множество припокриващи се механизми, изпълняващи сходни функции по различни начини. Всеки подход има различни характеристики за производителност, поддръжка, семантика и четимост. Не съществува стандартен подход, което принуждава непрекъснато вземане на решения за приет в рамките на проект начин за имплементиране дори на привидно прости конструкции. Проблемът не е просто наличието на стари практики, а съществуването на множество частично припокриващи се абстракции, всяка със свои частни случаи, конвенции и нива на съвместимост. В резултат правилното използване на езика изисква познания на историческите му „неправилни, но все още валидни“ аспекти. Това се наблюдава при по-сложни проекти като графичните приложения, където често се налага интеграция с legacy библиотеки за управление на графичен контекст или събития, като техните интерфейси не винаги следват съвременни конвенции.

Една от най-съществените практически слабости на C++ произтича не толкова от самия език, колкото от фрагментираната екосистема около него. За разлика от езици като Rust, Go или C#, които предоставят сравнително стандартизирана среда за разработка, стандартът на C++ не дефинира единен компилатор, build система, мениджър за зависимостите, или Application Binary Interface (ABI). В резултат на това създаването и поддръжката на развойна среда на даден проект само по себе си се превръща в инженерно предизвикателство. Няма установен стандарт и изборът има дългосрочни последици върху поддръжка, платформена зависимост и цена за бъдеща разработка.

Освен съображения при процеса на компилация, възникват въпроси относно сигурност и надеждност при използване на C++. Езикът предоставя директен достъп и контрол на ниско ниво върху паметта, което е сред основните му предимства. Същевременно тези възможности пораждаат рискове от некоректно използване. Стандартът допуска множество категории известни грешки при работа с паметта:

- „висящи“ указатели
- използване на освободена памет
- двойно освобождаване
- анулиране на итератори
- атаки чрез препълване на буфери
- състезания при паралелизъм.

Следователно съвременният модел на C++ за управление на притежание – RAII не е изцяло подходящ за високопроизводителни паралелни приложения. Това е от ключово значение при използването на приложно програмни интерфейси за GPU, тъй като, както е споменато по-горе, съвременни имплементации като Vulkan изискват експлицитно ръчно управление от страна на CPU кода. Езиците с GC също не предоставят адекватно решение на поставения проблем.

2.4 Програмен език Rust

Един от най-обещаващите заместители на широко използвания за програмиране на графични процесори C++ е системният език Rust. Rust е проектиран с цел да замести именно C++ чрез по-съвременен дизайн и липса на недостатъци от старо историческо развитие. Също така някои от функционалностите му са потенциално изключително подходящи за разрешаване на уникалните проблеми свързани с използването на Vulkan. Следва да бъде разгледана хипотезата, че разработката на платформено независими приложения с използване на GPU е възможна с Rust.

На първо място е логично да бъде разгледана основната парадигма за програмиране и структуриране на архитектура. Предпочитания стил може да бъде описан като императивно-функционален. Императивното програмиране е един от най-старите начини за структуриране на код, като се поддържа практически от всички системни езици включително C++. При него обаче се наблюдават голяма част от вече описаните проблеми по отношение на сигурност, надеждност и управление на ресурси (особено в многонишков контекст). По-голям интерес представляват функционалните възможности на езика Rust.

Функционалното програмиране (ФП) представлява програмна парадигма, при която изчисленията се моделират чрез оценяване на функции, а не чрез последователна модификация на изменяемо (mutable) състояние. В основата на подхода стои идеята, че програмите следва да бъдат изградени от малки, композируеми и предвидими функции, които трансформират данни без да променят външно състояние. Исторически ФП произлиза от lambda calculus, разработен от Алонсо Чърч през 30-те години на XX век, който представлява математически модел за изчисления чрез прилагане на функции. Съвременните функционални езици като Haskell, OCaml и F# изграждат значителна част от своята философия за дизайн върху концепции като неизменяемост на данните, „чисти“ (pure) функции, декларативно програмиране и композиция на функции. Изброените езици обаче разчитат на garbage collection и не са подходящ избор за високопроизводителни приложения. Разглеждането на функционалното програмиране в контекста на Rust все пак представлява интерес за текущата цел, тъй като е език без GC и разчита на компилатор, който може да оптимизира някои от недостатъците свързани с FP.

При функционалния подход се използва трансформация на стойностите. Вместо да описва как точно да бъде изпълнена операцията, кодът описва какъв резултат се търси. Основно съображение е производителността на този стил. Функционалното програмиране исторически е свързано със слаба производителност породена от честото последователно извикване на функции (т.нар. function chaining). Rust използва компилационен бекенд LLVM, който е платформено независим и има възможност за генериране на машинен код за други платформи чрез крос компилация. По-важната му характеристика в случая е, че може да оптимизира стандартни функционални операции на езика към платформено специфичен код с пълноценно използване на всички регистри и избягване на ненужни преходи към функции.

Предвид потенциално добрите performance характеристики на Rust, следва да бъде разгледан като кандидат за заместване на C++ в сферата на графичните приложения и по-конкретно, да бъде направена оценка на интеграцията му с Vulkan.

Езиковите конструкции произхождащи от функционалната парадигма с приложение в разработката на графични приложения са:

- неизменяемост (immutability) по подразбиране – силно подходяща за избягване на известни проблеми като състезания при паралелизъм; по дефиниция са невъзможни при неизменяем state

- pure функции / функции от по-висок ред – по-интуитивен стил за описание използваните математически алгоритми в компютърната графика

- композиция на функции – особено приложима техника при конструкция на структури предлагани по подразбиране от интерфейса Vulkan чрез използване на софтуерен шаблон Builder; непълната им инициализация или подаване на неочаквани аргументи често водят до трудни за диагностициране грешки при изпълнение, това се избягва чрез compile-time рестрикции

- софтуерен шаблон Pattern matching – задължава строго дефинирани проверки за всички очаквани състояния и обработката им чрез monads; по-добра производителност спрямо обработка на изключения, тъй като се случва по време на компилация

2.5 Предложена архитектура

Предложената глобална архитектура е модул базирана, като модулите се дефинират чрез имплементация на свойства (traits) за постигане на полиморфизъм.

```
App::new()  
    .add_systems(Startup, setup_system)  
    .add_systems(Update, (movement,  
collision).chain())  
    .add_systems(FixedUpdate, physics_system)
```

Фиг. 38 Псевдо код на използване на модули чрез композиция на функции

Архитектурата за управление на ресурсите е базирана на Entity Component System (ECS) – стандартна практика при разработка на графични приложения. Приносът на езика Rust е borrow системата, която позволява определяне на жизнения цикъл на ресурсите по време на компилация, като се забранява повече от една изменяема референция или съвместно съществуване на изменяема и неизменяема референция. В случаи, когато метод или структура съхраняват две референции от различни източници, се задължава експлицитно определяне на споделен жизнен цикъл (lifetime в Rust). По този начин се гарантира, че едната референция няма да стане невалидна докато другата се използва от притежателя. Дефинират се следните модули:

- Entity – идентификатор за групиране на компоненти
- Component – съдържа данните на отделните компоненти, структурират се като структури от масиви вместо масиви от структури с цел по-добро използване на кеш и SIMD регистри
- System – функции (closures), които определят операциите за извършване върху компонентите
- Resource – управляват заделените GPU буфери и дескриптори

За избягване на ръчно управление на графичните ресурси на Vulkan се използва псевдо garbage collector структура базирана на вградения borrow checker. Жизненият цикъл на всички ресурси е известен по време на компилация и чрез групирането им се позволява множествоно заделяне едновременно, което редуцира броят системни повиквания и комуникации през PCIe.

Шейдър програмите също са имплементирани в предимно функционален стил на езика Rust, като се допуска императивната парадигма в секции, където тя има по-добри характеристики на производителност. Те се компилират до междинния език Standard Portable Intermediate Representation (SPIR-V), който е отворен и платформено независим стандарт за генериране на инструкции за GPU чрез Vulkan. Това позволява цялата система да използва единен език, система за компилация и архитектура.

2.6 Резултати към втора глава

С цел извършване на предварителна оценка на производителността на разработената графична система са проведени тестове с рендиране на две сцени – опростена с няколко светлинни източника и сложна сцена с голям брой полигони от порядъка на 5 милиона. За оценка на платформената независимост тестовите среди са:

- Microsoft Windows 11 Enterprise Edition
- Archlinux с версия на ядрото 7.0.12 и разпределител на задачите Burst-Oriented Response Enhancer (BORE) с цел по-добра латентност

На таблица 2 са дадени осреднени времена за генериране на един кадър при изпълнение в рамките на 10мин. На фиг. 40 са представени генерираните изображения, като по-простата сцена е дадена вляво, а сложната – вдясно.

Platform	Average frame time simple scene	Average frame time complex scene
Windows	10.11 ms	66.5 ms
Linux (X11 and Wayland)	10.11 ms	65.6 ms

Табл. 2 Резултати от рендиране по алгоритъм на Уитед



Фиг. 40 Генерирани изображения по алгоритъм на Уитед

2.7 Изводи към втора глава

Въз основа на извършения сравнителен анализ между програмните езици и проведените предварителни тестове се достига до следните заключения:

1. Използването на езици с garbage collector не е подходящо за високопроизводителни приложения поради недостатъци на метода на системно ниво. При използване на графични интерфейси на ниско ниво GC не автоматизира процеса по управление на GPU ресурси без необходимост от допълнителен специфичен код.

2. Традиционно използваният системен език C++ проявява значими недостатъци при разработка на платформено независим софтуер с използване на външни библиотеки поради историческото си развитие и зависимост от поддръжка на обратна съвместимост. Сред най-изявените са тромаво управление на динамични ресурси, силно фрагментирана build система и липса на единна конвенция.

3. Езикът Rust е подходящ кандидат за заместване на C++ в посочения контекст, като стандартизираната крос компилация е силно предимство за постигане на платформена независимост. Функционалната парадигма е добре съвместима с математическото изразяване на графични алгоритми и интеграция с експлицитния програмен модел на Vulkan.

4. Съвременните графични процесори позволяват изпълнение на алгоритми за проследяване на лъчи в реално време, но са необходими оптимизации при изпълнение на SIMD архитектури за постигане на по-добри резултати при по-сложни сцени

ГЛАВА 3. Оптимизационни алгоритми за проследяване на лъчи

В тази глава на дисертационния труд се разглеждат оптимизационни алгоритми за проследяване на светлинни лъчи в реално време чрез графичен процесор.

3.1 Проследяване на светлинни пътища

Проследяването на светлинни пътища представлява разновидност на алгоритмите за проследяване на лъчи, която решава уравнението за рендирането (1). Методът е без отклонение и акумулативен, което означава, че с натрупване на броя извадки за пиксел получената оценка за осветеност се стреми към физически коректната стойност за взаимодействия като т.нар. меки сенки, каустики, цветово просмукване (colour bleeding), отражения и пречупване.

Алгоритмите за генериране на фотореалистични изображения се стремят да симулират известното уравнение за рендирането:

$$(1) \quad L_o(\omega_o) = L_e(\omega_o) + \int L_i(\omega_i) * f(\omega_i, \omega_o) * \cos(\theta_i) d\omega_i$$

Трудността при симулирането му произтича от факта, че величината L присъства и от двете страни на уравнението. Освен това L е функция на пространствената позиция и посоката, което означава, че стойностите в различните членове на уравнението в общия случай не съвпадат.

Уравнението може да бъде опростено като се счита, че разликата между L_o и L_i е несъществена за решението.

$$(7) \quad L(x_1 \rightarrow x_0) = L_e(x_1 \rightarrow x_0) + \sum_{n=1}^{\infty} P(n)$$

Тази форма е използваната в компютърната графика за симулация на светлина с алгоритъма за проследяване на пътища. Излъчването от дадена точка към друга представлява сумата от емисията на повърхността L_e и допълнителния принос на n брой отражения $P(n)$.

За да бъде възможно пресмятането на $P(n)$ за стойности на n от 1 до ∞ с хардуерни изчислителни устройства, е необходимо да бъдат решени следните казуси:

- как да бъде генериран път със специфичен брой отражения
- как да се пресметне приноса от безкрайно много отражения

3.2 Монте Карло интеграция

Методът на Монте Карло интеграцията дава задоволително решение на непрекъснати интеграли с безкрайни граници като използва случайно оценяване на извадки – Multiple importance sampling (MIS). Основата на алгоритъма е вземане на краен брой N независими случайни извадки от цялата област за интеграция. На тяхна основа може да бъде направена числена оценка на интеграла I , без да се налага пълното му пресмятане:

$$(8) \quad I = \int_{\Omega} f(x) dx$$

Първа стъпка е генериране на независимите извадки $x_0, x_1, \dots, x_n$ на базата на функцията за плътност на вероятностите $p(x)$. След това се прави оценка на интеграла:

$$(9) \quad F_N = \frac{1}{N} \sum_{i=1}^N \frac{f(x_i)}{p(x_i)}$$

Оценката направена по този начин е без отклонение, т.е. математическото очакване на оценителя съвпада точно със стойността на интеграла, тъй като извадките принадлежат на областта му. При увеличаване на броя извадки вероятността за значително отклонение клони към нула. Следователно оценителят клони към стойността на интеграла.

Следните свойства правят Монте Карло методът подходящ за решаване на интегрални задачи в компютърната графика:

- методът е сравнително прост, достатъчно е да се генерират краен брой извадки и да се изчислят стойностите на интеграла в съответните точки
- оценяването на многоизмерни интеграли представлява пряко разширение върху едномерния частен случай – единствено е необходимо извадките да се вземат от пълната област на интеграла
- скоростта на сближаване към сходимост с точната стойност е $O(N^{-1/2})$
- методът не е чувствителен към прекъснатостта на интеграла – запазва същата скорост на сходимост и при прекъснати интеграли

3.3 Монте Карло с извадки по важност

За имплементация на Монте Карло е необходимо да бъде определена функция за плътност на вероятностите (PDF - Probability density function). Съгласно свойствата на алгоритъма всяка функция би дала правилен резултат с достатъчно извадки. Най-простата, която може да бъде разгледана, е равномерно събиране на извадки върху цялата област. Това решение няма да бъде задоволително, тъй като не гарантира, че стойността клони към реалното решение. За пример, нека бъде разгледано решението с Монте Карло на интеграла $f(x)$ (14) с $p(x)$ (15):

$$I = \int_0^1 f(x) dx$$

(13)

$$f(x) = \begin{cases} 0.01 & : x \in [0, 0.01] \\ 1.01 & : x \in [0.01, 1] \end{cases}$$

(14)

$$p(x) = \begin{cases} 99.01 & : x \in [0, 0.01] \\ 0.01 & : x \in [0.01, 1] \end{cases}$$

(15)

Методът Монте Карло оценява интеграла напълно коректно при използване на дадената вероятностна плътност. Оценката се сближава към точната стойност с увеличаване на броя извадки, но скоростта на сходимост е изключително неблагоприятна.

Основната идея на претегляне на извадките по важност (importance sampling) е да се вземат повече извадки в онези области на дефиниционната област, които имат по-голям принос към крайната стойност на интеграла. Идеалният случай е споменатият по-горе, но той почти никога не е реализируем на практика. Въпреки това, дори намирането на вероятностна плътност, която има форма, близка до тази на интеграла, може значително да подобри ефективността, тъй като концентрира извадките там, където приносът е по-голям.

Този подход представлява практичен метод за намаляване на дисперсията, без да се въвежда значителен допълнителен изчислителен разход. Това е ключово за имплементиране на Монте Карло върху физически хардуер.

3.4 Монте Карло с множествено претегляне на извадки

Претеглянето на важност на извадките не е универсално решение. В редица ситуации при симулация на светлина е необходимо да се използват различни вероятностни плътности в зависимост от конкретния случай.

Обобщено правило на практика е да се генерират извадки съгласно вероятностна плътност, пропорционална на доминиращия член в уравнението на рендирането. Макар в горния пример да е възможно да се избира между извадково оценяване по BRDF и по светлинния източник в зависимост от повърхността и осветлението, в повечето случаи предварително не е ясно кой фактор е доминиращ - и именно това представлява основния проблем.

Вместо да се търси една „перфектна“ вероятностна плътност, методът за оценка чрез множествени извадки (Multiple Importance Sampling, MIS) следва различен подход: той комбинира резултатите, получени чрез различни вероятностни плътности, като ги претегля по подходящ начин, така че да се намали дисперсията без въвеждане на съществен допълнителен шум.

Нека p_i извадки са взети за $p_i(x)$ функции за плътност на вероятностите, където n е броят на функциите. Оценителят по метода MIS е:

$$F_{mis} = \sum_{i=1}^n \frac{1}{n_i} \sum_{j=1}^{n_i} w_i(X_{i,j}) \frac{f(X_{i,j})}{p_i(X_{i,j})} \quad (16)$$

Следните условия трябва да бъдат изпълнени, за да не се внася отклонение в оценката:

$$\begin{aligned} & \bullet \sum_{i=1}^n w_i(x) = 1 \text{ if } f(x) \neq 0 \\ & \bullet w_i(x) = 0 \text{ if } p_i(x) = 0 \end{aligned} \quad (17)$$

3.5 Евристични методи за подобряване на MIS

Ключов фактор за ефективността на MIS е намирането на подходящи функции за теглата. Една от най-често използваните стратегии е т.нар. балансиран евристичен метод. Теглата се дефинират по следния начин:

$$w_i(x) = \frac{n_i p_i(x)}{\sum_k n_k p_k(x)} \quad (20)$$

Оценителя изглежда така след заместване:

$$F_{mis} = \sum_{i=1}^n \sum_{j=1}^{n_i} \frac{f(X_{i,j})}{\sum_k n_k p_k(x)} \quad (21)$$

Дисперсията намалява при използването на този оценител. От уравнението следва, че оценката е добра, стига поне една от използваните функции за плътност на вероятностите да наподобява интегрираната функция. Това се наблюдава дори в частен случай в две плътности функции в MIS.

Достига се до изводът, че алгоритъмът Монте Карло в комбинация с MIS е подходящ за начално разрешаване на проблемите с изчисляване на опростеното уравнение за рендирането.

3.6 Проследяване на пътища с Монте Карло

Следва да бъде преразгледан проблемът с изчисляване $P(n)$ за стойности на n от 1 до ∞ в контекста на Монте Карло. Необходимо е да се генерират пътища с точно n отражения, за да се симулира съответният член от интеграла. При зададен брой n възниква въпросът как да се конструира път с точно толкова взаимодействия.

Основният проблем е, че в почти всички случаи не е на разположение информация за разпределението на излъчването в сцената. Следователно не може да се конструира pdf, която точно да съответства на интегрираната функция. Начален подход е подбиране на функции, които емпирично дават добри резултати за определен тип повърхности. Например оценяване на извадки по посоката чрез тежести определени от косинус (cosine-weighted pdf) дава добри резултати за дифузни повърхности. При силно отразяващи повърхности (идеално огледало) разпределението е равномерно и отговаря на закона на отраженията. Последната точка задължително лежи на повърхността на светлинен източник. В случая вземане на извадки според повърхности дава по-добри резултати от извадки по brdf, тъй като вероятността е по-голяма да бъде „улучен“ светлинен източник. Следователно най-задоволителни резултати се получават при комбинация от различни функции за плътност на вероятностите чрез използване на MIS.

Остава да бъде решен проблемът с генериране на приносът от безкраен брой отражения. Използваният механизъм е методът руска рулетка. Нека решението на уравнението за рендирането е сумата от приноса на всички пътища $P(n)$, където n е броят отражения. Приема се, че при увеличаване на n приносът на даден път намалява поради редуциране на интензитета на излъчването. Съществуват изключения при силно отразяващи повърхности, но в общия случай може да се допусне, че твърдението е валидно за повечето сцени. Тогава вероятността да не бъде въведено отклонение при терминиране на даден път след определен брой отражения може да бъде дефинирана така:

$$P'(n) = \begin{cases} \frac{P(n)}{T} & : t < T \\ 0 & : t \geq T \end{cases} \quad (25)$$

T е вероятността за продължаване на пътя, а t е равномерно разпределена случайна величина в интервала $[0, 1)$. Съответно пътят се прекратява с вероятност $1-T$ и продължава с вероятност T . Методът на руската рулетка гарантира, че вероятността да се достигне път с произволен брой отражения клонящ към безкрайност е ненулев. Същевременно в общия случай повечето светлинни пътища прекратяват след сравнително малък брой взаимодействия. По този начин не се внася отклонение, но изчислителният процес е практически приложим.

3.7 Sampling-Importance Resampling

За подобрене производителността на евристичните методи за определяне на подходяща функция на плътност на вероятностите следва да бъдат разгледани алгоритми, които могат да работят с по-малко налични данни. SIR (Sampling-Importance Resampling) е метод за генериране на извадки по зададена вероятностна плътност (PDF). За разлика от класическите техники, SIR изисква значително по-малко информация за самата целева плътност. Този метод обаче не генерира директно извадки от целевата PDF, а я апроксимира.

Нека $\hat{p}(x)$ е функция, от която да бъдат направени извадки. Стъпките на алгоритъма са както следва:

1. генерират се M извадки от по-проста и по-лесна за вземане на извадки функция $p(x)$ наречена предложена функция (proposed PDF)

2. на всяка извадка се присвоява тегло x по формула: (27) $w(x) = \hat{p}(x)/p(x)$

3. от множеството M се избира една извадка (целева), като вероятността на избор е пропорционална на теглото

Алгоритъмът използва няколко разновидности PDF според тяхното предназначение.

Предложената функция (Proposal PDF) е функцията, от която се правят първоначалните извадки. Както е споменато по-горе, тя трябва да бъде сравнително проста, за да не бъде изчислително скъп процесът на вземане на извадки. В частния

случай на рендирането на кадри с проследяване на пътища в реално време е важно да се отбележи, че дори такава функция да не е налична – тук първоначалните извадки се правят чрез изстрелване на лъчи, което е скъпа операция – алгоритъмът предразполага първоначалните извадки да бъдат разпределени във времето (между генерирани кадри) за постигане на по-добра производителност. Това води до шум в първоначално генерираните кадри, който може да бъде минимизиран чрез прилагане на филтър.

Целевата функция (Target PDF) е търсената функция. Отново в частния случай на проследяване на пътища обикновено тази функция не е известна за дадена сцена. Най-често целевата функция се дефинира като нормализирана функция на самата функция на интеграла. Тъй като теглата в SIR са пропорционални на отношението $(28) \hat{p}(x)/p(x)$, нормализиращият множител на целевата функция няма влияние върху алгоритъма. Следователно не е необходимо целевата функция да бъде нормализирана. Това е съществено предимство на SIR - възможността да се използва ненормализирана функция.

Крайната PDF, от която се вземат финалните извадки апроксимира целевата. При брой извадки $M=1$ на практика извадките се вземат от предложената функция, при M клонящо към безкрайност крайната функция клони към целевата. Обикновено достатъчно добра сходимост се постига при сравнително малки стойности на M .

Недостатъците на методът в текущата му форма са:

- SIR е неаналитичен; от дадени извадки M , предложена и целева PDF не може да бъде изведена аналитична форма на крайната PDF функция, от която се вземат крайните извадки
- от горепосоченото следва, че ако се разгледа произволна точка от друго разпределение, не може да бъде изчислена вероятността тя да бъде генерирана чрез SIR
- в частни случаи при някои по-сложни PDF се изисква голям брой извадки за получаване на задоволителна апроксимация

3.8 Resampled Importance Sampling

RIS (Resampled Importance Sampling) е техника, при която SIR се използва за апроксимация на пълен интеграл, докато SIR е фокусиран единствено върху генерирането на извадки.

Стъпките на алгоритъмът са:

1. избира се предложена PDF $p(x)$, която е сравнително опростена и предразполага към вземане на извадки без много изчислителен ресурс
2. генерират се M предложени извадки x_i
3. избира се целева PDF $\hat{p}(x)$, която е пропорционална на $f(x)$
4. на всяка извадка се присвоява тегло (31); теглото е модифицирано спрямо SIR като множителя $1/M$ е включен в дефиницията; както е споменато SIR позволява скалиране с произволна положителна константа
5. от множеството M се избират N нови извадки u_i с възстановяване, като вероятността на избор е пропорционална на теглата $w(x_i)$; в контекста възстановяване означава, че една и съща предложена извадка може да бъде избрана повече от веднъж, което ускорява процеса по вземане на извадки на база на теглата
6. чрез извадките N се прави оценка на апроксимацията на интеграла, като се добавя коефициент за корекция на отклонението (32)

Като при всички методи за проследяване на светлинни пътища, едно от ключовите условия е да не се внася отклонение при оценката на интеграла, което би довело до шум и стойности, които не са физически коректни. Използваният коефициент е:

$$(32) \quad \frac{\sum_{i=1}^M w_i(x_i)}{N}$$

Достига се до междинен алгоритъм за оценяване на интеграла, който комбинира вземане на извадки от ненормализирана функция чрез SIR и оценяване на Монте Карло интеграла без внасяне на отклонение чрез RIS.

3.9 Комбинация на RIS и MIS

При работа със симулация на светлинни взаимодействия вземане на извадки от една функция на плътност на вероятностите често не е достатъчно, поради различните дифузионни и поглътящи свойства на повърхностите в дадена сцена. Това е посочено при обсъждане на MIS. Следва да бъде разгледано как това може да бъде интегрирано с техниката RIS, като тук са възможни два подхода.

Първият метод е техниката MIS да бъде приложена на ниво предложена функция. При генериране на извадки в RIS няма изискване всички те да произхождат от една и съща предложена PDF. Алгоритъмът може да работи с множество функции $p_i(x)$. Изискване е да са изпълнени упоменатите условия за работа с MIS, а именно:

- сумата на теглата на дадена извадка да бъде единица
- теглата да бъдат ненулеви само там, където съответната PDF е ненулева

Обобщената формула за теглата на всяка предложена функция може да бъде представена като:

$$w_i(x) = m_i(x) \frac{\hat{p}(x)}{p_i(x)} \quad (35)$$

Където $p_i(x)$ е i -тата предложена функция, $\hat{p}(x)$ е общата целева функция, $m_i(x)$ е тегло определено по MIS (например чрез балансиран евристичен метод). Вижда се, че предходното използвано тегло в RIS представлява частен случай на обобщената форма, където всички предложени функции съвпадат, теглото е равномерно и броят извадки за всяка PDF е еднакъв. Тъй като $m_i(x)$ представлява MIS тегло, то задължително трябва да удовлетворява условието да не се внася отклонение на MIS. В противен случай комбинирането на различни предложени PDF в RIS може да доведе до грешки в крайния оценител. При използване на множество предложени функции в рамките на RIS самият механизъм на ресемплирането остава валиден, но коректността зависи от правилното дефиниране на теглата в рамките на MIS.

Комбинирането на MIS и RIS позволява:

- да се използват различни стратегии за генериране на предложена PDF в зависимост от ситуацията (например при проследяване на пътища – по BRDF, по светлинни източници и т.н.)
- да се комбинират техните предимства на етапа на изграждане на резервоара от извадки
- да се прилагат допълнителни техники като ресампелиране или повторно вземане на извадки

3.10 Weighted Reservoir Sampling

Претеглено оценяване на извадки с използване на резервоар (Weighted Reservoir Sampling - WRS) е алгоритъм, който позволява избирането на една извадка от поток (stream) без предварително да е известно колко извадки ще бъдат налични. Броят на наличните извадки може да се променя динамично при изпълнението без да се нарушава предварително зададените вероятности. Това е полезно при втората стъпка от алгоритъма SIR, която изисква вземане на извадка от множество предложени, като вероятността дадена такава да бъде избрана е пропорционална на нейното зададено тегло. Времевата сложност на тази стъпка теоретично е $O(\lg(N))$, но предпроцесните процедури са ограничени до сложност $O(N)$ по отношение на време и пространство в паметта. WRS дава значително подобрение, като при този подход паметта е константна $O(1)$. Това е от изключителна важност при рендиране в реално време чрез графичен процесор с физически ограничено количество дискретна памет, защото позволява поддържане на малки по размер резервоари за всеки пиксел.

За реализация на алгоритъма се изисква дефиниране на структура данни резервоар (reservoir) съдържаща поне следните две полета:

- текуща избрана извадка
- обща натрупана тежест

Възможен е празен резервоар без референция към текуща извадка и обща изчислена тежест равна на нула. Може да се отбележи, че това са изискванията към базовия WRS. В приложение за проследяване на пътища се използва поне едно допълнително поле за брой обработени извадки.

Всяка новопостъпила извадка от потока може да бъде приета, като това заменя текущата избрана, или да бъде отхвърлена. Генерира се случайно число със стойност от нула до едно. Дефинира се праг:

$$(37) \quad threshold = W_{new} / (W_{new} + W_{total})$$

Ако случайната стойност е по-висока от прага, извадката се отхвърля. Ако тя е приета, новата извадка измества текущата (ако има такава). Независимо от решението, натрупаната тежест се актуализира.

Първите няколко извадки с нулево тегло могат да бъдат игнорирани. Те не оказват влияние на процеса. При първата ненулева извадка прагът е равен на 1.0, поради което случайното число никога няма да бъде по-голямо от него и новата извадка със сигурност ще бъде избрана. Това съответства на очакването, че ако има само една извадка, вероятността тя да бъде избрана е 1. Както е посочено по-горе, структурата на резервоара изисква единствено два елемента: обща натрупана тежест и избраната извадка. Всяка нова извадка също притежава тегло. Новите извадки от потока могат да бъдат интерпретирани като частен случай на резервоар, където избраната извадка е самата себе си, а общото тегло е равно на индивидуалното. От тази гледна точка отделни извадки вече не съществуват – всичко може да бъде разгледано като резервоар. Алгоритъмът WRS тогава просто комбинира текущия резервоар с нов входящ резервоар, съдържащ една извадка. Възниква въпросът дали WRS може да приема като вход резервоари, генерирани от други изпълнения на WRS. Нека се предположи, че до момента са обработени N извадки. Алгоритъмът получава като следващ вход резервоар, който е обработил M извадки. В този входящ резервоар всяка от обработените M извадки у_k има вероятност да бъде избрана:

$$(41) \quad p(y_k) = \frac{w(y_k)}{\sum_{i=1}^M w(y_i)}$$

Когато този резервоар бъде обработен от текущия WRS, вероятността дадена извадка у_k да бъде крайно избрана става:

$$(42) \quad p(y_k) = \frac{w(y_k)}{\sum_{i=1}^M w(y_i)} * \frac{\sum_{i=1}^M w(y_i)}{\sum_{i=1}^N w(x_i) + \sum_{i=1}^M w(y_i)} = \frac{w(y_k)}{\sum_{i=1}^N w(x_i) + \sum_{i=1}^M w(y_i)}$$

Всяка от предварително обработените M извадки получава точно правилната вероятност, еднакво на това всички N+M извадки да са обработени едновременно. Макар процесът да е извършен на два етапа, резултатът е еквивалентен на едноетапно изпълнение. Може да се заключи, че WRS може да се изпълнява рекурсивно на няколко нива, без това да влияе върху коректността на разпределението на вероятностите. Процесът не е ограничен до две стъпки - той може да бъде композиран многократно, като във всеки случай се запазва правилната пропорционалност спрямо теглата на всички участващи извадки.

Достигнатите заключение за предимствата на този метод са:

- не се изисква предварително знание за броя на извадките, работи се с динамичен поток от входни данни
- изискванията за памет са константни, алгоритъмът съхранява само определен брой стойности

- всеки нов вход може да бъде обработен за константно време, дори когато той представлява резервоар, който вече е обработил милиони извадки
- алгоритъмът е силно подходящ за паралелизация; ако се разполага с източник с огромен брой извадки, той може да бъде разделен на няколко части и всяка да бъде обработена паралелно; редът на обработката е без значение

3.11 Reservoir Spatio-Temporal Importance Resampling

Гореописаните алгоритми са предпоставка за използвания в съвременното проследяване на светлинни пътища с използване на GPU. Алгоритъмът ReSTIR DI е насочен към решаване на проблема с големия брой източници на светлина („many-light problem“). Това е първоначална стъпка при симулиране на осветеност на по-сложни и реалистични сцени.

Алгоритъмът рендерира 2D изображения (на базата на 3D сцена). Достатъчно е ефективен, за да бъде използван в реално време, което е целта. Всеки пиксел пази информация за собствен резервоар и акумулира в него при генерирането на нови кадри. Процесът на работа е както следва:

За всеки пиксел на екрана алгоритъмът първоначално генерира множество начални светлинни проби (в оригиналната имплементация $M=32$). Тези проби се разглеждат като предложени извадки, които впоследствие се използват в алгоритъма RIS.

Първо се избира източник на светлина на случаен принцип, като вероятността за избор е пропорционална на неговата излъчвателна мощност. Частен случай е, че ако в сцената присъства HDRI небе (skybox), по подразбиране му се задава вероятност 25%. Тъй като вероятностите са пропорционални на теглата на източниците, а не зависят от конкретната точка на засенчване, може да се използва класическият метод с кумулативна таблица и бинарно търсене за избор на източник. Това е изчислително евтин метод. След като бъде избран конкретен източник, се прилага съответен метод за генериране на случайна светлинна проба върху него, в зависимост от това дали става въпрос за HDRI осветление или източник от тип излъчваща площ.

Всички предложени проби впоследствие се обработват в рамките на RIS чрез резервоарно вземане на извадки на базата на WRS. Този процес е силно подходящ за паралелизация, което го прави идеален за обработка от графичен процесор. Като целева функция PDF се използва приносът на светлината без отчитане на засенчване. Математически това се изразява като:

$$(43) \quad L(w_i, w_o) f_r(w_o, w_i) \cos(\theta_i)$$

Това е компонент от уравнението за рендирането, като на този етап на изчислението не е известно дали съответният светлинен източник е видим от точката на засенчване или не. На този етап може по избор да бъде приложена фаза на повторно използване на видимостта. Тя се въвежда поради факта, че дори при неограничен брой предложени извадки резултатът може да остане шумен вследствие на разликата между използваната целева функция и реалното уравнение на осветяване; по-конкретно поради липсата на функцията за видимост.

За включване на проверка за видимост на този етап може да се използва интегралната част от пълното уравнение на рендирането като целева функция. Това води до значително намаляване на шума, породен от проблеми с видимостта. Ако се приложи тази вариация, алгоритъмът трябва също така да извършва проверка за видимост при обединяване на целеви извадки от съседни пиксели на по-късен етап, за да се въвежда отклонение.

Внасянето на тази фаза може да не е подходящо за устройства с ограничен изчислителен капацитет, тъй като може да доведе до значително намаляване на производителността поради увеличения брой лъчи, изстрелвани за всеки пиксел.

Втори етап на алгоритъма е преизползване (reuse) във времева дименсия, т.е. на базата на информация от предишно генерирани кадри. За всеки кадър се извлича информация за движението в т.нар. вектор на движението. На базата на векторите на

движението се изчислява съответстващата позиция в предходния кадър и се взима резервоара, принадлежащ на съответния пиксел от историята на рендирането. Това е форма на кеширане.

Трети етап е изпълняване на алгоритъма итеративно в няколко стъпки. При всяка итерация се избира случаен съседен пиксел и се комбинира неговия резервоар (или единична извадка третирана като резервоар) с текущия. Тук се наблюдава изключително бързо нарастване на броя индиректно посетени извадки от предложената функция. При разглеждане на частен случай за първия генериран кадър всеки пиксел би имал $(k+1)*M$ извадки. При множество итерации теоретичният максимум достига $(k+1)n*M$, където n е броят итерации и k е броят на съседите.

С натрупване на кадри процесът се повтаря и за всеки пиксел се акумулират извадки, чийто брой понякога достига порядък от милиони. Чрез използване на RIS сред тях се определя „печеливша“ извадка т.е. с най-голям принос. На този етап все още не е взето предвид засенчването и тук то се изчислява чрез проследяване на единствен лъч, което е сравнително изчислително евтино.

3.12 Резултати към трета глава

Направено е сравнение с имплементация на същите оптимизационни алгоритми с използване на интерфейс DirectX 12, за да бъде сравнена производителността му с Vulkan. На фиг. 49 е представен примерен кадър от рендиране, а в табл. 3 е дадена производителността при различни разделителни способности.



Фиг. 49 Кадър от рендиране на сцена Sponza с ReSTIR

Resolution	Sponza scene FPS			Lights scene FPS		
	Vulkan average	Vulkan lowest	D3D12 average	Vulkan average	Vulkan lowest	D3D12 average
1080p	58.52	56.50	57.89	48.47	48.42	48.42
1440p	38.32	38.30	38.32	28.71	26.23	25.89

Табл. 3 Кадри за секунда с използване на алгоритъм ReSTIR

3.13 Изводи към трета глава

Вследствие на проведените тестове се достига до следните изводи:

1. Алгоритъмът за оптимизация при проследяване на пътища ReSTIR може да бъде имплементиран платформено независимо чрез подходяща имплементация, която не е фиксирана към архитектурата на графичните процесори на Nvidia

2. При правилно прилагане на оптимизационни алгоритми, техниката проследяване на светлинни пътища е приложима в реално време за постигане на максимално

реалистични светлинни взаимодействия дори при сравнително високи опреснителни честоти

3. Езикът Rust използван като описателен език за математически трансформации чрез функционалната парадигма в рамките на шейдър програми за GPU има капацитета да реализира сложни оптимизационни алгоритми с висока степен на производителност

4. Vulkan като платформено независима алтернатива на DirectX 12 не отстъпва по отношение на използване на изчислителния капацитет на съвременни графични процесори в реалистични приложни сценарии

5. Наблюдава се един съществен недостатък на алгоритъма ReSTIR – при много високи резолюции се достига до изчерпване на видео паметта поради голямото количество кеширани стойности за повторно използване, което води до невъзможност за рендиране на сцените

ГЛАВА 4. МЕТОДОЛОГИЯ ЗА СКАЛИРАНЕ КЪМ ВИСОКИ РЕЗОЛЮЦИИ И ГЕНЕРИРАНЕ НА КАДРИ С ИЗПОЛЗВАНЕ НА МАШИННО ОБУЧЕНИЕ

Целта на тази глава на дисертационния труд е да анализира методите и архитектурите за AI базирано скалиране към високи резолюции и генериране на кадри при рендиране в реално време, тяхната интеграция с GPU графичните конвейери и ролята им в платформено независими графични системи.

4.1 Основи на скалиране към високи резолюции и генериране на кадри с използване на машинно обучение

Супер-резолюцията представлява процес на реконструкция на изображение с висока резолюция от изображение с по-ниска резолюция. В графичните системи това позволява рендиране при по-ниска разделителна способност и последващо увеличаване до целевата резолюция, което намалява натоварването върху GPU.

Задачата за супер-резолюция се разглежда като обратна реконструкция на високорезолюционно изображение от ограничено наблюдение:

$$(44) \quad I_{HR} = F(I_{LR}; \theta)$$

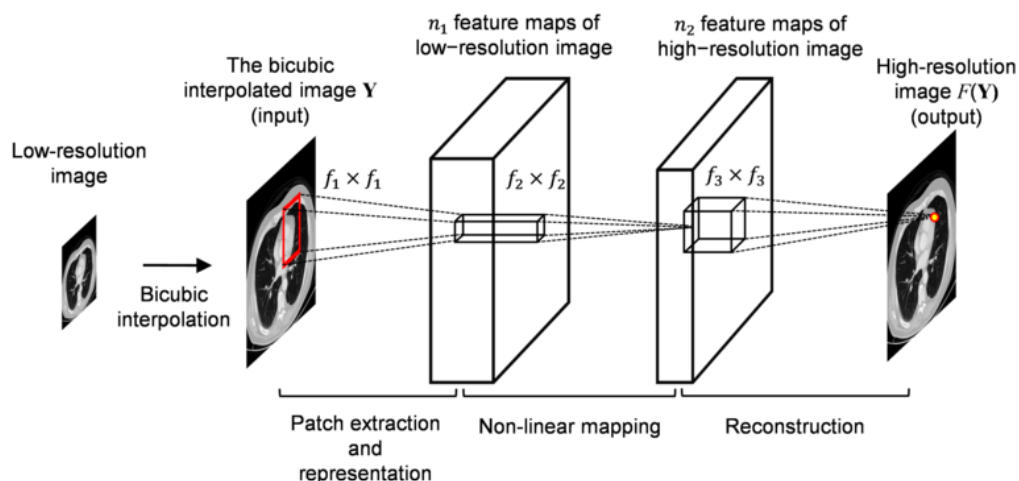
където I_{LR} е нискорезолюционно изображение, I_{HR} е реконструирано високорезолюционно изображение, F е невронна мрежа с параметри θ .

Задачата е слабо определена, тъй като множество високорезолюционни изображения могат да съответстват на едно нискорезолюционно изображение. Невронните мрежи решават този проблем чрез статистическо обучение върху големи набори от изображения и извличане на статистически зависимости между ниско- и високочестотни структури.

Класическите методи за мащабиране като интерполация по най-близък съсед, билинейна и бикубична интерполация реконструират пикселни стойности чрез изчисляване на стойностите на липсващите пиксели чрез локални математически функции, базирани на съседни пиксели и използване на локални полиноми без реконструкция на детайли. Основното ограничение на класическите методи е невъзможността за възстановяване на високочестотни детайли и текстури, тъй като не се използва информация извън локалната околност. В резултат на това скалираните изображения често изглеждат размазани и с ниска рязкост.

С развитието на дълбокото обучение скалирането към висока резолюция претърпява значителна трансформация. Невронните мрежи позволяват автоматично извличане на признаци и моделиране на сложни зависимости между ниско- и високорезолюционни изображения. За разлика от класическите методи, които използват фиксирани математически модели, дълбоките мрежи се обучават върху големи набори от данни и могат да реконструират статистически вероятни детайли, които не са налични във входното изображение.

Типичната архитектура на невронна мрежа за скалиране към висока резолюция включва слоеве за извличане на признаци, реконструкция на детайли, слой за мащабиране.



Фиг. 51 Архитектура на невронна мрежа за супер-резолюция

Генерирането на междинни кадри представлява задача за интерполация във времето, при която се синтезира изображение между два последователни кадъра чрез моделиране на движението на сцената. В реално време рендирането честотата на кадрите е ограничена от изчислителната сложност на сцената. Генерирането на кадри използва ML за синтез на междинни кадри между реално рендирани кадри, увеличавайки ефективната честота.

Съвременните ML-подпомогнати рендеринг системи комбинират пространствена супер-резолюция и времева интерполация в единен модел като пространствена супер-резолюция, времева интерполация и ML денойзинг. Тези архитектури използват информация от предходни кадри и текущото изображение с ниска резолюция за реконструкция на високорезолюционен резултат, който е времево консистентен и позволяват едновременно намаляване на пространствената и времевата сложност на рендиране, като се запазва визуалното качество.

4.2 Платформена независимост на AI-подпомогнатото рендиране

Платформено независимото рендиране със скалиране към високи резолюции и генериране на кадри изисква архитектурен подход, при който процесът на увеличаване на резолюцията и генерирането на кадри се реализира чрез хардуерно абстрактен изчислителен слой, който позволява изпълнение върху различни графични платформи. В този модел скалирането и генерирането на кадри се разглежда като обща изчислителна операция върху изображения и тензори, независима от конкретната реализация на GPU.

Интеграцията на AI-подпомогнато рендиране чрез скалиране към високи резолюции и генериране на кадри в платформено независими графични системи се базира на няколко ключови архитектурни принципа.

Първият принцип е абстракция на GPU изчисленията. Скалирането и генерирането на кадри се реализира чрез универсални паралелни операции (конволюция, филтриране, трансформации), които могат да бъдат изпълнявани чрез compute shaders или общи GPU kernels. Това позволява алгоритмите да бъдат независими от конкретната графична архитектура.

Вторият принцип е използване на унифицирани представяния на невронни модели. Невронните мрежи за скалиране и генериране на кадри могат да бъдат представени чрез хардуерно независими формати на графи на изчисление, което позволява тяхното изпълнение върху различни GPU чрез адаптиран backend слой.

Третият принцип е отделяне на рендеринг и реконструкция. Рендирането при ниска резолюция остава част от графичния конвейър, докато скалирането и генерирането на кадри се реализира като независим изчислителен модул. GPU-базирани ML операции могат да бъдат реализирани чрез compute shaders, CUDA/OpenCL, Vulkan compute. Това

позволява замяна или адаптиране на метода за скалиране и генериране на кадри без промяна на рендеринг логиката.

В платформено независимата архитектура процесът на рендиране и скалиране се организира в три основни етапа. Първият етап е рендиране при по-ниска резолюция чрез стандартния графичен pipeline. Вторият етап включва извличане на допълнителни буфери, като дълбочина и вектори на движение, които подпомагат реконструкцията. Третият етап е платформено независимото скалиране чрез ML или аналитичен метод, изпълняван върху абстрактен GPU слой. Този модел позволява изпълнение върху различни графични API и операционни системи без промяна на алгоритмите за реконструкция.

4.3 Индустириални реализации на AI-базирано скалиране

В съвременните графични системи за рендиране в реално време AI-базираното скалиране към високи резолюции се реализира чрез индустириални технологии, интегрирани директно в графичните драйвери и GPU архитектури. Най-значимите представители на този клас технологии са Intel® Xe Super Sampling (XeSS) и NVIDIA Deep Learning Super Sampling (DLSS), които използват дълбоки невронни мрежи за реконструкция на високорезолюционни изображения от нискорезолюционно рендирани кадри с използване на пространствено-времева информация. В контекста на платформено независимите графични системи XeSS демонстрира как невронна супер-резолюция може да бъде реализирана като универсален модул, интегриран в графичния pipeline без зависимост от конкретна GPU архитектура. За разлика от XeSS, DLSS е силно интегриран с хардуерната архитектура на NVIDIA и изисква тензорни ядра за оптимална производителност. Поради това технологията е по-малко платформено независима, но предлага максимално качество и ефективност върху поддържаните GPU архитектури. Макар и обещаващо решение за платформено независим upscaling метод, невъзможността за интегриране към съществуващ софтуер за рендиране без достъп до изходния код представлява проблем при интеграция на FSR 3 с legacy проекти или такива със затворен код, тъй като се нуждае от вътрешна информация за състоянието на процеса по рендиране. FSR 1 има нужда единствено от framebuffer, за да извършва скалиране, но дава значително по-слаби резултати и не разполага с техника за генериране на кадри.

Достига се до извода, че съществуват платформено независими решения за мащабиране при рендиране в реално време, но липсват лесни за интегриране и модификация модели с използване на машинно обучение за генериране на кадри с цел.

4.4 Обучаване на модел чрез Vulkan Compute

Обучаването на невронен модел с използване чрез Vulkan има силно предимство, когато система за рендиране с графичния интерфейс вече е налична. Не се налага допълнителен setup на среда, код за инициализация на GPU ресурси или използване на други програмни езици. Единствената разлика спрямо графично приложение е използването на чисто изчислителни GPGPU Compute шейдъри, като това практически означава единствено смяна на тип. Процесът може да се разгледа на няколко етапа:

1. Генериране на данни – за разлика от голяма част от ML проектите, генерирането на dataset за upscaling е сравнително лесно; основната стъпка е генериране на идентични кадри с ниска и висока резолюция, за да може моделът да бъде обучен върху разликите между тях; това е тривиално при наличие на функционалности за рендиране; допълнителни данни включват вектори за движение, буфери за дълбочина, нормали и цвят; отново те са директно достъпни

2. Създаване на невронната архитектура – на входно ниво се приемат гореописаните данни както и последния изход от мрежата за темпорална информация; използва се конволюция на ниво суб-пиксел за извършване на мащабирането;

3. Дефиниране на многокомпонентна функция на загубата - след деформиране на текущия изход към следващия кадър чрез векторите на движение от основната истина, се наказва разликата спрямо следващия изход; намалява се тежестта на загубите в области,

където деформацията е ненадеждна (на база валидност на векторите на движение или разлики в дълбочината)

4. Цикъл на обучение с последователни пакети от кадри – обучава се върху пакети от кадри (от порядъка на 5-10 в серия) като се преизчислява загубата и се обновяват теглата

5. Оптимизиране за изпълнение в реално време – извършва се прунинг и квантуване на модела с цел подобряване на скоростта на изпълнение; целевата скорост е под милисекунда при приложение в реално време

Алгоритъмът отново може да бъде реализиран на Rust и транслиран към SPIR-V. Ключова особеност при работа с Vulkan Compute, е че средата все още не предлага специализирани инструменти за машинно обучение, а служи единствено за изчисления с общ характер. За постигане на добра производителност при процеса на обучение, най-важна е оптимизацията на скоростта на матричното умножение $M \times N \times K$.

4.5 Резултати към четвърта глава

За оценка на създадения модел е направена съпоставка с FSR 3, който най-близко наподобява наличните функционалности и характеристики за платформена независимост. Тестът се състои от скалиране към 4k от входна резолюция 1080 при сценарии, където генерирането на native 4k кадри не дава задоволителна производителност. Отбелязва се, че тъй като моделите се изпълняват от графичния процесор заедно с рендирането, се наблюдава известно забавяне в сравнение със случая без мащабиране.

Резолюция	Средни кадри за секунда
4k native	17.6
1080 native	60.5
FSR	48.5
Vulkan Compute Upscaled	49.7
VCU + FXAA	49.2

Табл. 4 Кадри за секунда при различните техники за мащабиране



Фиг. 62 Сравнение в близък план на Native 4k, Native 1080 и VC + FXAA

ЗАКЛЮЧЕНИЕ

Възможно е създаване на високопроизводителна платформа за рендиране и използване на машинно обучение, при която няма съществени разлики при използване на различни операционни системи и производители на графични карти.

Функционалният подход на програмиране не внася измерима изчислителна цена спрямо процедурното при условието, че се комбинира с компилиран език и набор от компилационни инструменти, които притежават способността да транслират

декларативни инструкции към такива, които са ефективни за съответната архитектура. Разрешава се отклоняване от чисто функционалното програмиране в критични за производителността участъци от някои алгоритми.

Използването на Vulkan е за предпочитане пред OpenGL за създаване на платформено независим софтуер за GPU при условие, че е изграден слой абстракция с достатъчно висока кохезия и ниско ниво на зависимост между компонентите.

Алгоритмите ReSTIR DI и ReSTIR GI могат да бъдат адаптирани за платформено независими приложения и са подходящи за използване при проследяване на светлинни пътища върху силно паралелни архитектури.

Мащабирането към по-високи резолюции с използване на изчислителни методи, методи с машинно обучение или хибридни подходи са ефективен начин за редуциране на необходимата изчислителна мощност и дискретна памет при приложенията за рендиране в реално време, когато необходимия ресурс за изпълнението им е подходящо балансиран.

НАУЧНО-ПРИЛОЖНИ И ПРИЛОЖНИ ПРИНОСИ

Научни приноси:

- Разработен е концептуален модел за изграждане на платформено независима система за рендиране на изображения в реално време основан на единен програмен модел за графични и изчислителни операции чрез Vulkan;
- Предложен е функционално-императивен подход при разработването на високопроизводителни графични приложения в реално време, който се базира на безопасно управление на ресурсите, модулност и ефективно използване на съвременните многоядрени архитектури.

Научно-приложни приноси:

- Предложена е архитектура на платформено независим rendering engine за рендиране на изображения в реално време с проследяване на лъчи, реализирана чрез Vulkan и езика Rust, която позволява използването на единен модел за управление на графичните и изчислителните ресурси върху различни операционни системи без необходимост от междинни слоеве и платформено специфични промени;
- Разработен е подход за интегриране на модифицирани и адаптирани алгоритми за директно и глобално осветление ReSTIR DI и ReSTIR GI в платформено независима среда за рендиране на изображения в реално време с path tracing, който позволява преносимо рендиране в реално време и използване на хардуерно ускорено проследяване на лъчи върху различни GPU архитектури;
- Предложен е подход за използване на машинно обучение при рендиране на изображения чрез AI-базирано мащабиране на изображения към високи резолюции и генериране на междинни кадри в реално време с цел намаляване на изчислителното натоварване в графичния конвейер при запазване на визуалното качество на рендираните изображения.

Приложни приноси:

- Разработена е методика за експериментално изследване и сравнителна оценка на производителността на платформено независими графични приложения при различни операционни системи, графични среди и графични интерфейси;
- Реализиран е прототип на платформено независим rendering engine за рендиране на изображения в реално време с проследяване на лъчи с path tracing, използващ Vulkan и Rust, с който са проведени експериментални тестове, доказващи еднаква или еквивалентна производителност при изпълнение върху Windows и Linux среди;
- Експериментално е доказана възможността за постигане на съпоставима производителност при използване на разработената система върху различни операционни системи и графични среди без използване на платформено специфични реализации.

СПИСЪК НА ПУБЛИКАЦИИТЕ ПО ДИСЕРТАЦИОННИЯ ТРУД

1. Georgi G. Georgiev, Milena Lazarova, "Programming Interfaces for Cross-platform Image Rendering and Deep Learning GPGPU," 2023 International Scientific Conference on Computer Science (COMSCI), Sozopol, Bulgaria, 2023, <https://doi.org/10.1109/COMSCI59259.2023.10315835> (индексирана в Scopus)
2. Georgi G. Georgiev, Milena Lazarova, "Cross-Platform Functional Programming Real-Time Path Tracing Rendering Engine," 2025 60th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST), Ohrid, North Macedonia, 2025, <https://doi.org/10.1109/ICEST66328.2025.11098413> (индексирана в Scopus и Wos)
3. Georgi G. Georgiev, "Cross-Platform Rust Based Implementation of GPU-Accelerated Importance Resampling Path Traced Illumination," 2025 13th International Scientific Conference on Computer Science (COMSCI), Sozopol, Bulgaria, 2025, doi: 10.1109/COMSCI67172.2025.11225132 (индексирана в Scopus)
4. Georgi G. Georgiev, Daniela Gotseva, "AI Upscaling and Frame Generation for Real Time Image Rendering," Proc. of 2026 14th International Scientific Conference Computer Science (COMSCI), 13-15 Sept. 2026, Sozopol, Bulgaria (приета за публикуване)

SUMMARY

Dissertation title: Platform independent applications using GPU and machine learning

Author: Eng. Georgi Georgiev, MSc

Graphics processing units (GPUs) have undergone a fundamental shift in their role over the last decade. From specialized hardware for image rendering with applications in video games, virtual reality, cinema, and engineering design, their parallel computational architecture enables high-performance implementations of general-purpose algorithms, often surpassing the capabilities of central processors. The GPGPU paradigm enables the use of graphics techniques such as real-time ray tracing, which creates far more accurate simulations of light interactions with objects in three-dimensional space compared to traditional rasterization. The thousands of computational cores on a given chip are also at the heart of the dynamic growth and development of machine learning and artificial intelligence, which, in addition to the fields of so-called big data processing, language models, and computer vision, also find application in computer graphics. The evolution of graphics processors is tied to the software interfaces that control them. As a result of historically distinct divergences, GPU-based applications often remain bound to a specific hardware or software environment, or require the mastery of multiple application programming interfaces and the integration of complex abstraction layers into the program architecture. This limits the ability to fully harness the increasingly relevant parallel computing devices, especially for small developer teams or scientific researchers.

This dissertation aims to explore the possibilities for creating unified and platform-independent software for developing applications that use GPUs in the field of computer graphics and their integration with machine learning. The following tasks have been performed to achieve this aim:

Creating a platform-independent architecture that enables the development of GPU-accelerated applications without significantly limiting performance compared to platform-specific solutions.

Developing a unified programming model for graphics and compute tasks, reducing the complexity of resource management and integration between individual application components, while achieving portability across different hardware and software platforms.

Researching and applying suitable optimization algorithms for real-time ray tracing in order to achieve performance suitable for interactive applications, without substantially degrading visualization quality.

Investigating methods using machine learning to compensate for the limitations arising from the computational complexity of photorealistic rendering, for effective scaling to high resolutions and generating intermediate frames while preserving acceptable visual quality.

Combining a platform-independent graphics pipeline, optimized ray tracing, and machine learning techniques to build practically applicable real-time graphics systems that efficiently utilize the capabilities of modern GPUs regardless of the specific hardware platform.