Санкт-Петербургский государственный университет

МОСЯГИН Олег Сергеевич

Выпускная квалификационная работа

Реализация движка для симуляции физики частиц на GPU

Уровень образования: бакалавриат

Направление *02.03.03 «Математическое обеспечение и администрирование информационных систем»*

Основная образовательная программа *СВ.5006.2019 «Математическое обеспечение и администрирование информационных систем»*

Научный руководитель:
доцент кафедры системного программирования, к.т.н. Ю.В. Литвинов

Рецензент:
разработчик графики ООО «Сабер Интерактив СГС» В.А. Овчинников

Санкт-Петербург
2023

Saint Petersburg State University

Oleg Mosiagin

Bachelor's Thesis

Particles simulation engine implementation on GPU

Education level: bachelor

Speciality *02.03.03 "Software and Administration of Information Systems"*

Programme *CB.5006.2019 "Software and Administration of Information Systems"*

Scientific supervisor:
C.Sc. Docent Yurii Litvinov

Reviewer:
rendering programmer at "Saber Interactive SGS LLC" V.A. Ovchinnikov

Saint Petersburg
2023

Оглавление

Введение

В настоящее время компьютеры зачастую используются для расчетов различных симуляций физических эффектов. Ядерная физика не стала исключением. Вычисления на компьютерах помогают получать различные теоретические или даже практические результаты. Например, при помощи компьютерной симуляции частиц можно рассчитывать, какие показания будут на детекторе этих частиц, расположенном у подножия горы. С помощью такого моделирования можно предсказывать показания детектора при тех или иных изменениях внутри горы. Таким образом, можно наблюдать изменения в горных породах, имея только детекторы элементарных частиц.

В Московском физико-техническом институте разрабатываются программные решения для симуляции поведения элементарных частиц. Как правило, такие решения основаны на численном интегрировании методами Монте-Карло. Для симуляций со сложной геометрией (например, при расчете распределений частиц вблизи горы) прослеживается путь множества частиц, обычно рассчитывая пересечения лучей, сгенерированных по частицам, на каждый шаг симуляции. Для подобного рода симуляций можно использовать различные формы представления сцены. Например, библиотека TURTLE [1] позволяет загрузить карту высот и искать пересечения лучей с геометрией, представленной в такой форме, из кода на CPU. Но иногда таких возможностей бывает недостаточно и требуется больше гибкости. Зачастую, в таком случае данная задача решается представлением геометрии для симуляции в виде набора треугольников с требуемой детализацией.

Учитывая то, что задача симуляции множества частиц хорошо параллелизируема, для данной задачи имеет смысл использовать видеокарты. Современные графические процессоры [27], состоят из множества потоковых процессоров (рисунок 1[1]), имеющих несколько блоков выполнения, исполняющих одновременно только одинаковые операции

[1] https://developer-blogs.nvidia.com/wp-content/uploads/2018/09/image11.jpg (дата обращения: 09.12.2022)

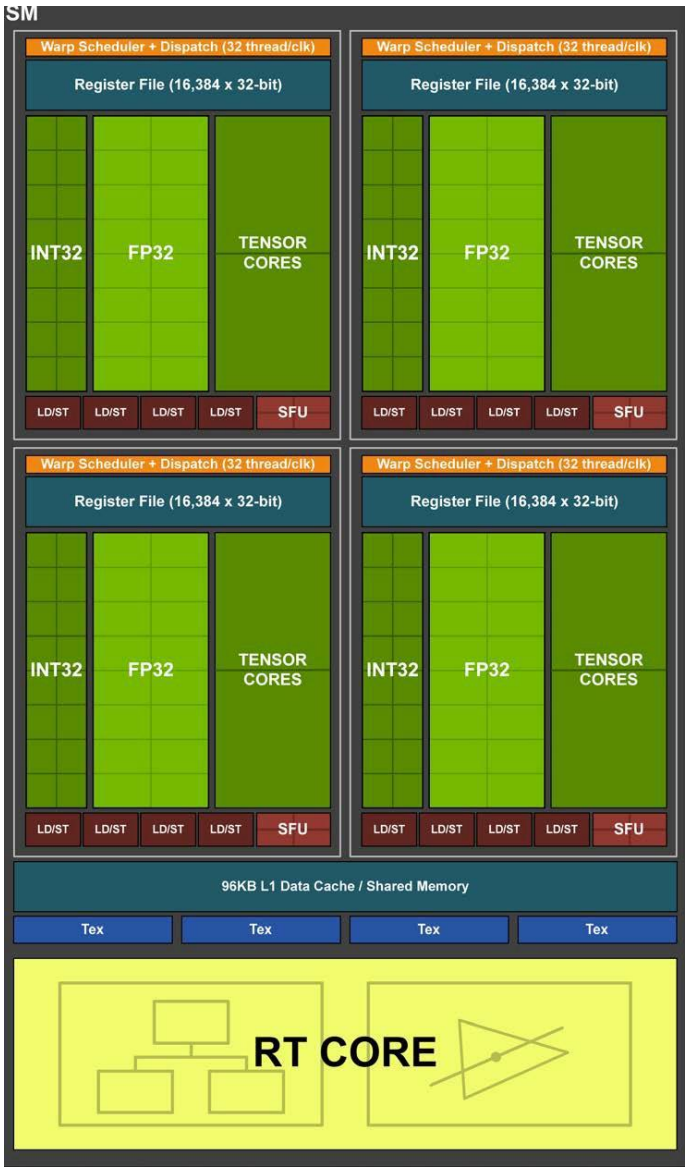

Рис. 1: Архитектура потокового процессора Turing.

на всех потоках, запущенных на этом блоке. Также GPU имеют большую пропускную способность памяти, как правило, с большой задержкой, которая скрывается планировщиком, снимающим с блоков выполнения потоки, ожидающие выполнения операции с видеопамятью. Часто современные видеокарты имеют аппаратное ускорение пересечений лучей с геометрией. Для запуска программируемой работы на видеокарте пишется шейдер (программа, выполняемая на видеокарте с разными входными данными). Как правило, вычислительная работа на GPU разбивается программистом на множество рабочих групп (групп выполнений шейдера, которые распределяются по вычислитель-

ным блокам драйвером видеокарты).

По описанным выше причинам, для разработки алгоритмов симуляции элементарных частиц будет полезен фреймворк, позволяющий задавать геометрию в формате набора треугольников и предоставляющий интерфейс для пересечения лучей с заданной сценой. При этом данное решение должно позволять реализовывать новые алгоритмы, в отличие от множества существующих решений (например, MCGPU [9]), уже имеющих внутри себя алгоритмы, основанные на методах Монте-Карло. Работа направлена на разработку движка, позволяющего получать доступ к вычислительным мощностям современных видеокарт, при этом инкапсулируя внутри себя часть сложности работы с API для взаимодействия с видеокартой. Тему дипломной работы предложил Ролан Гринис, научный сотрудник МФТИ.

1. Постановка задачи

Целью работы является реализация движка для ускорения расчетов физики частиц на современных видеокартах с аппаратным ускорением трассировки лучей. Для достижения этой цели были поставлены следующие задачи.

1. Провести обзор существующих работ по симуляции физики частиц на видеокартах.
2. Разработать архитектуру движка симуляции частиц.
3. Реализовать фреймворк для разработки алгоритмов симуляций ядерной физики.
4. Провести апробацию полученного решения.

2. Обзор

2.1. Работы по симуляции элементарных частиц на видеокартах

Для того, чтобы рассмотреть существующие работы по симуляции физики частиц с использованием современных видеокарт, проводится обзор. Дополнительной целью обзора является выделение полезных для такого рода симуляций приемов.

По запросу «monte carlo particle transport on gpu» в Google Scholar было найдено несколько статей, посвященных программным решениям для симуляций физики частиц.

Статья «Exploiting Hardware-Accelerated Ray Tracing for Monte Carlo Particle Transport with OpenMC» [16] описывает добавление в библиотеку OpenMC [28] поддержки аппаратного ускорения трассировки лучей. OpenMC предназначена для симуляций нейтронов и фотонов. Авторы статьи утверждают, что в результате работы они получили значительное ускорение расчетов. В работе было получено примерно 33-кратное ускорение на геометрии из треугольников, при переносе вычислений с центрального процессора AMD Ryzen 7 2700 на видеокарту NVIDIA RTX 2080 Ti. Работа ориентирована на взаимодействие с библиотекой OptiX [26].

Статья «Evaluation of Single-Node Performance of Parallel Algorithms for Multigroup Monte Carlo Particle Transport Methods» [18] рассказывает про различные подходы и методы оптимизации симуляции частиц. В частности, рассказывает про технику замены отброшенных частиц на новые с целью повышения загрузки GPU [2]. Реализация в данной статье опирается на метод событий, заключающийся в выделении таких событий, как пересечение поверхности или продвижение частицы, и последующей обработки очередей событий. При переносе такого подхода на видеокарты можно группировать вызовы вычислений по типам событий и сокращать таким образом разнообразность инструкций, выполняемых в рамках одного блока выполнения видеокарты. Так как,

видеокарты обычно не имеют возможности выполнять на одном вычислительном блоке разные инструкции, данный подход может давать прирост производительности.

Статья «Modeling parameterized geometry in GPU-based Monte Carlo particle transport simulation for radiotherapy» [25] рассказывает о симуляции взаимодействия частиц с геометрией, представленной параметрически заданными поверхностями. Данный подход по заверениям автора позволил повысить производительность по сравнению с аналогичной симуляцией с вокселизированной геометрий.

В тексте «LLNL Monte Carlo Transport Research Efforts for Advanced Computing Architectures» [23] описывается реализация прокси-приложения для быстрого прототипирования алгоритмов симуляции частиц. Реализация полагается на CUDA [5] для использования ресурсов графического процессора. Данная реализация воплощает подход к симуляции частиц, основанный на событиях. В целях оптимизации перед операциями с частицами они копируются в разделяемую память [4] с меньшей задержкой и большей пропускной способностью, диапазон которой доступен каждой рабочей группе. Более того, такой подход может ускорить чтение данных из глобальной памяти благодаря оптимальному доступу, достигаемому путем чтения потоками одной рабочей группы памяти из одной кэш линии (доступ к одной кэш линии потоками одной рабочей группы обычно проводится за одну транзакцию памяти). Константы событий располагаются в константной памяти, имеющей кэш в каждом потоковом процессоре.

Таким образом, в результате обзора стало понятно, что перенос такого рода симуляций на современные видеокарты имеет смысл с точки зрения ускорения вычислений. Еще большее значение с точки зрения производительности имеют работы по ускорению расчетов для физики частиц на видеокартах с аппаратным ускорением трассировки лучей. В результате обзора были подчеркнуты техники для повышения производительности симуляций.

2.2. Выбор API для работы с видеокартами

Рассмотрим Vulkan [20], Direct3D 12 [10] и OptiX [26], предоставляющие интерфейс для работы с аппаратным ускорением трассировки лучей.

OptiX предоставляет API, близкий к CUDA [5]. Данный интерфейс доступен только на видеокартах от NVIDIA. Вследствии того, что интерфейс направлен на вычисления общего назначения, он упрощен по сравнению с графическими API, так как имеет меньшую область применения.

Direct3D 12 – это низкоуровневый графический API для программирования под Windows, основное назначение которого заключается в построении кадров компьютерных игр. Интерфейс имеет поддержку трассировки лучей в виде модуля DXR [12]. Дизайн Direct3D 12 направлен на уменьшение неявных операций внутри драйвера и многопоточное программирование. Доступны широкие возможности для отладки взаимодействия с Direct3D 12 такими инструментами, как PIX [15] или RenderDoc [29] (это отладчики работы с графическими API, позволяющие записывать работу на GPU через обращения к API и отлаживать вычисления, производимые на видеокарте). PIX имеет поддержку отладки функциональности DXR [11] (например, в PIX можно просматривать содержимое иерархических структур, в которые нужно собирать геометрию для вызовов конвейера трассировки лучей).

Vulkan – кросс-платформенный графический интерфейс, имеющий много общего с DirectX 12. Для Vulkan существуют расширения, добавляющие поддержку аппаратно-ускоренной трассировки лучей. Данный интерфейс имеет возможность захвата в RenderDoc, однако поддержки данных расширений в отладчике нет [17].

Вследствии более широкой поддержки видеокарт и лучших возможностей отладки, по сравнению с Vulkan, был выбран Direct3D 12. Именно в этом графическом API достигается компромисс между доступностью на различных платформах и удобством отладки и разработки (данное удобство распространяется также и на использование библио-

теки).

2.3. Direct3D 12

2.3.1. Логическое устройство

Основные операции в Direct3D [13] выполняются при помощи объекта логического устройства. API спроектирован с упором на минимальное количество накладных расходов, поэтому по умолчанию правильность обращения не проверяется. Для того, чтобы включить дополнительные проверки времени выполнения, позволяющие обнаружить многие ошибки обращения к графическому интерфейсу, можно перед созданием устройства включить проверочные слои.

2.3.2. Работа с командами графического процессора

При помощи логического устройства можно создать очередь команд. Это объект, отвечающий за передачу команд на выполнение видеокарте. При создании можно указать приоритет очереди. Команды отправляются на выполнение в очередь в виде предварительно записанных списков команд. Для выделения списков команд необходим распределитель команд, который можно создать вызовом метода логического устройства. При этом доступ к спискам команд должен быть внешне синхронизирован, однако можно создать условия для параллельной записи нескольких списков команд.

2.3.3. Состояние конвейера

Поскольку компиляция шейдеров из байткода в машинный код конкретной видеокарты и соотнесения параметров вычислительного или графического конвейера может быть сложной операцией, Direct3D 12 позволяет создавать объекты состояния конвейера, собирающие в себе множество параметров выполнения кода на видеокарте. Также создание данных объектов можно кэшировать.

2.3.4. Ресурсы

Для создания ресурсов необходимо устройство. В Direct3D 12 ресурсами являются такие объекты, как буферы или изображения. При работе с ресурсами они должны опираться на кучи памяти. Кучи памяти аналогичны выделениям памяти. При этом создание кучи на каждый ресурс может иметь серьезные накладные расходы [8]. Поэтому хорошей практикой является расположение нескольких ресурсов в одной куче и использование связанных с кучей ресурсов только для больших ресурсов.

2.3.5. Доступ к ресурсам из шейдеров

При создании состояния конвейера указывается объект корневой подписи, показывающий как располагаются ресурсы для привязки к шейдерам. В подписи можно указать несколько ресурсов, доступ к которым будет выполняться напрямую, и таблиц привязок ресурсов (куч дескрипторов), обеспечивающих двухуровневую иерархию обращения. Куча дескрипторов, привязываемая к корневой подписи, должна быть помечена флагом, обеспечивающим видимость в шейдерах. Можно также создавать кучи дескрипторов без доступа из шейдеров. Дескрипторы можно создавать из ресурсов, выделив место в куче дескрипторов. После создания можно копировать дескрипторы из одной кучи в другую. При этом копирование из видимой в шейдере кучи обычно имеет больше накладных расходов.

2.3.6. Барьеры

При записи команд необходимо вставлять барьеры ресурсов. Барьеры используются для указания зависимостей между командами для ресурсов. Также барьерами указываются состояния, в которых должен находиться ресурс во время тех или иных операций.

Например, чтобы сначала записать данные в буфер, а затем прочитать, можно использовать два вызова вычислений на видеокарте. Для этого можно записать команды запуска вычислений (с

помощью метода Dispatch [21] класса ID3D12GraphicsCommandList, обеспечивающего интерфейс для записи команд) в список команд. В первом использовать для записи буфер в состоянии D3D12_RESOURCE_STATE_UNORDERED_ACCESS [7], предназначенном для доступа для чтения и записи в произвольном порядке. Во втором можно читать данные из этого буфера в состоянии D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE [7] для доступа ресурса из шейдера только для чтения. Между этими вызовами вычислений должен быть осуществлен перевод ресурса из одного состояния в другое. Для этого можно записать барьер в список команд методом ID3D12GraphicsCommandList::ResourceBarrier [22].

2.3.7. Ускоряющие структуры

Для оптимизации трассировки лучей в DXR введены иерархические структуры, в которые собирается сцена. Существует разделение на ускоряющие структуры нижнего уровня и высокого уровня. Структуры низкого уровня содержат в себе иерархию геометрии [3], а структуры высокого уровня объединяют структуры нижнего уровня в единый шейдерный ресурс.

3. Реализация

В рамках работы реализовывался фреймворк для упрощения работы с видеокартой. Основной целью данного решения было предоставить удобный интерфейс для запуска вычислений на видеокартах с возможностью использования аппаратного ускорения трассировки лучей. Несмотря на то, что Direct3D 12 и был выбран в качестве основного интерфейса для работы с видеокартой, было бы полезно оставить в рамках реализации удобную возможность для дальнейшего добавления поддержки Vulkan для доступности использования фреймворка на большем количестве платформ.

Фреймворк состоит из двух частей. Одна реализована на C# и отвечает за генерацию кода объявления ресурсов шейдера и создание файлов с описанием этих ресурсов. Вторая часть написана на C++ и предоставляет программный интерфейс для работы с видеокартой и реализации алгоритмов, использующих многочисленные пересечения лучей с геометрией из треугольников. Например, симуляций процессов переноса энергии движением элементарных частиц.

3.1. Архитектура

В рамках работы разработана архитектура решения, упрощающего разработку алгоритмов симуляции частиц на GPU. Диаграма компонентов представлена на рисунке 2.

Слой «Direct3D 12 layer implementation» нужен для абстрагирования от конкретного графического программного интерфейса. Остальные части решения взаимодействуют исключительно с интерфейсами из компоненты «Graphics API abstraction layer». В дальнейшем, благодаря такому разделению, при необходимости можно будет добавить поддержку Vulkan, создав компоненту «Vulkan layer implementation», реализовав в ней интерфейсы слоя абстракции.

Компонента «Bindings code generator» отвечает за генерацию кода, отвечающего за привязку ресурсов шейдеров. Также данная компонента решения генерирует элементы кода, такие как объявление структур

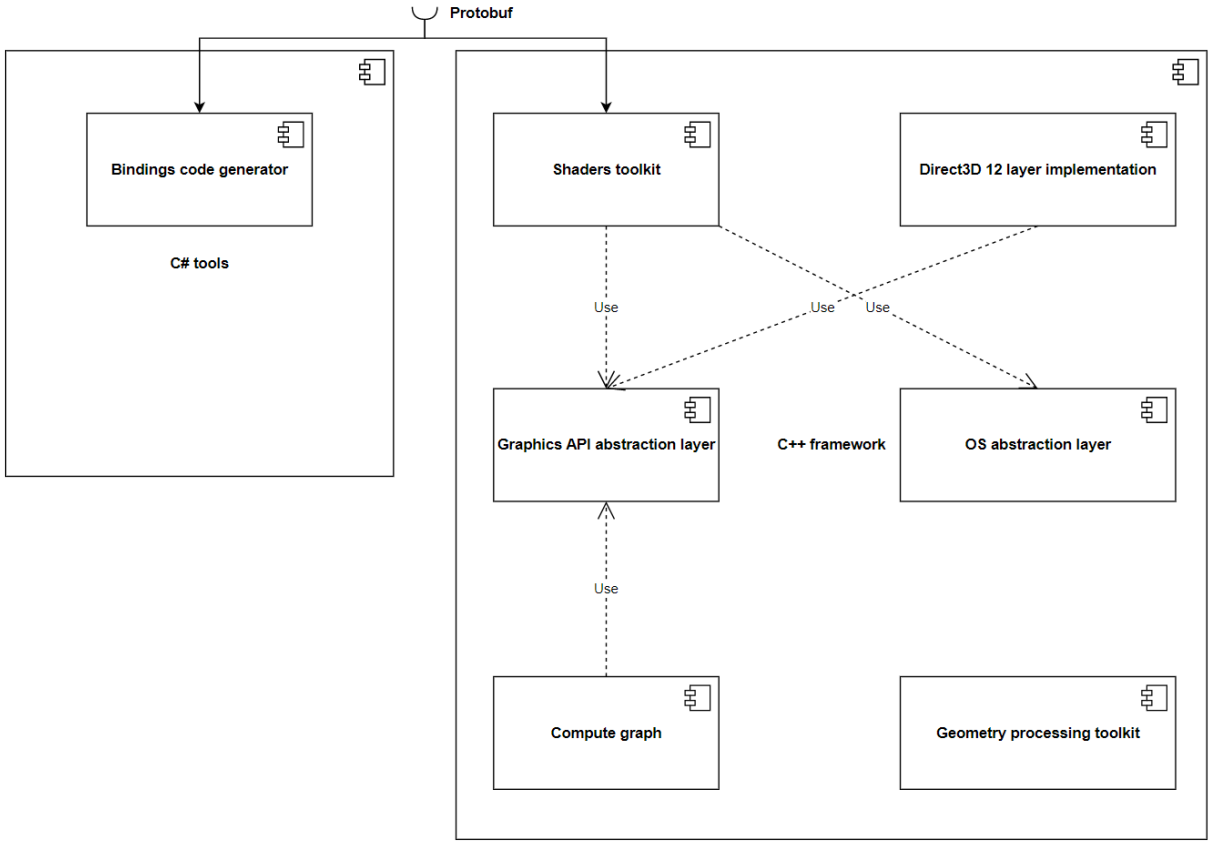

Рис. 2: Диаграмма компонентов.

или констант, помогающие избежать дублирования кода. Сгенерированный код привязки ресурсов использует слой абстракции от графического API.

Основная функциональность фреймворка находится в компоненте «Compute graph» [8]. Данная компонента отвечает за диспетчеризацию пользовательского кода, задаваемого в виде событий с зависимостями. При помощи компоненты «Dependencies manager» она анализирует зависимости событий и расставляет барьеры.

3.2. Слой абстракции от графического API

Был реализован слой абстракции от графического API Direct3D 12. Слой упрощает работу с графическим интерфейсом. При разработке внимание уделялось возможности добавления поддержки Vulkan.

Так, например, интерфейс привязки ресурсов в слое сделан близким

образом с Vulkan. Можно создавать из пользовательского кода объекты взгляда (в англ. view) на ресурс и записывать их в пулы дескрипторов. При запросе записи дескрипторов происходит копирование в видимые шейдеру кучи из куч, недоступных шейдеру. А создание и распределение дескрипторов в кучах, недоступных шейдеру, происходит в слое неявно.

Для ожидания работы видеокарты на процессоре внутри слоя эмулируется поведение забора (в англ. fence), примитива синхронизации из Vulkan. При отправке команд в очередь на выполнение можно передать примитив и после ожидать с его помощью выполнения работы. Такой подход не только хорошо подходит для переноса на Vulkan, но и более удобен для пользователя, чем подход из DirectX, основанный на 64-битных отметках выполнения.

Для распределения используется библиотека D3D12 Memory Allocator [6]. Библиотека создает большие кучи памяти и распределяет их память между ресурсами.

Для работы с аппаратным ускорением трассировки лучей реализована поддержка встроенного пересечения, позволяющего обращаться к ускоряющим структурам из любого шейдера [14].

3.3. Генератор кода для взаимодействия с ресурсами шейдеров

Код, предназначенный для выполнения на GPU, пользователи фреймворка смогут писать на языке программирования шейдеров HLSL [19]. Для согласованности кода для работы с ресурсами шейдеров на C++ и на HLSL реализуется генератор кода. Генератор кода загружает файл специального формата, который описывает ресурсы, используемые конвейером шейдеров. Формат поддерживает объявления привязок ресурсов и объявления структур и перечислений. В качестве результата генерируется код, который может быть подключен в шейдеры для работы с ресурсами. Также создается файл, хранящий структуру ресурсов конвейера шейдеров. Данный файл предназначен

для загрузки со стороны кода, исполняемого на центральном процессоре, который будет привязывать данные.

Ниже приведен пример файла, принимаемого в качестве входных данных этого компонента.

```
struct Type1
{
    float field1;
};

struct Type2
{
    int field1;
    Type1 field2;
};

UAV_BUFFER<Type2> readWriteBuf;
UAV_TEXTURE<float4> readWriteTex;
SRV_BUFFER<float4> readBuffer;
SRV_TEXTURE<uint2> readTexture;
TLAS tlas;

enum Enum1
{
    ELEMENT1,
    ELEMENT2,
};
```

В результате работы генератора могут быть получены два файла. Первый файл в бинарном формате с описанием ресурсов шейдера для использования в C++ коде. Второй файл для включения в шейдеры на HLSL, в котором будут содержатся объявления структур, перечислений и ресурсов шейдера в формате, поддерживаемом Direct3D 12.

3.4. Граф зависимостей ресурсов

Для управления состояниями ресурсов в современных графических API нужно расставлять барьеры, переводящие ресурсы из одного состояния в другое или указывающие драйверу видеокарты на зависимость данных для предотвращения состояния гонки данных. Основываясь на этих указаниях, драйвер может сбрасывать кеш памяти или даже менять расположение (англ. layout) ресурса.

Для расстановки барьеров можно выделить основные методы [8].

- Ручная расстановка. Может достигаться большая эффективность, однако процесс написания кода с вручную расставленными барьерами является сложным. Такой подход имеет большой риск допуска ошибки, которая приведет к неопределенному поведению.

- Перевод ресурса в «базовое» состояние после каждой операции и перевод из него перед любой операцией. Такой подход может приводить к проблемам с производительностью из-за лишних операций.

- Сохранение состояний вместе с ресурсом. Такой подход является рабочим, однако может страдать из-за отсутствия гибкости при параллельной записи команд. Также в данном подходе трудно достичь группировки барьеров, позволяющей ускорить выполнение команд.

- Анализ состояний ресурсов через граф вычислительных узлов с зависимостями. Данный подход позволяет реализовать группировку барьеров и достичь высокой скорости исполнения команд.

В данной работе реализован подход с графом вычислительных узлов с группировкой барьеров с возможностью ручного выставления некоторых барьеров.

3.5. Библиотека для работы с шейдерами

В рамках фреймворка реализован набор инструментов для работы с шейдерами. Данный компонент отвечает за загрузку байткода шейдеров, создание конвейеров и привязку ресурсов. Для привязки ресурсов загружается файл, сгенерированный компонентой «Bindings code generator», и заполняется множество дескрипторов, соответствующих нужным ресурсам. «Shaders toolkit» также отвечает за хранение множеств дескрипторов, чтобы они могли использоваться из очередей видеокарты.

3.6. Блок работы с геометрией

Для упрощения работы с картами высот и создания геометрии по ним создан компонент «Geometry processing toolkit». Таким образом, с помощью реализованного фреймворка можно загружать карты высот, строить геометрию по ним и собирать ускоряющие структуры для работы с аппаратно ускоренной трассировкой лучей. При этом построение ускоряющих структур упрощено с учетом задачи. В симуляциях физических процессов редко используется меняющаяся геометрия. Direct3D 12 позволяет создавать ускоряющие структуры, разбитые на множество частей, имеющих возможность изменения своего расположения. В реализации фреймворка данная возможность не используется для простоты интерфейса, а ускоряющая структура строится из одной части.

3.7. API фреймворка

Для работы с фреймворком нужно будет создавать объекты классов вычислительных узлов, реализующих интерфейс, требующий наличие методов записи команд и формирования списка зависимостей узла. Данные объекты необходимо будет отправлять во фреймворк и после вызывать метод, планирующий работу, анализирующий зависимости, записывающий список команд и отправляющий на выполнение в очередь команд.

Для привязки буферов и изображений нужны объекты взаимодействия с ресурсами, которые создаются методом класса устройства. Сами ресурсы создаются распределителем ресурсов из слоя абстракции от графического API. Перед вызовом вычислений нужно привязать необходимые шейдеру ресурсы вызовами методов класса шейдера, который генерируется из файла с объявлением ресурсов шейдера.

В методе записи команд вычислительного узла будет необходимо добавить необходимые команды в принимаемый методом объект для их записи. Объект для записи команд из слоя совместимости с графическим API содержит методы, соответствующие командам, записываемым в список команд Direct3D 12.

Для того, чтобы создать вычислительный узел, нужно объявить класс с методами GetDependencies (должен возвращать список необходимых зависимостей узла с требуемыми состояниями) и WriteCommands (должен записывать команды для видеокарты через специальный класс). Например, так может выглядеть метод WriteCommands.

```
void WriteCommands(core::CommandBundle* commandBundle) override
{
    // Установка выходного изображения ресурсом шейдера
    shader->SetShaderImageUav(''renderTarget'', *imageUav);
    // Установка сцены ресурсом для чтения
    shader->SetShaderTlas(''scene'', *tlasSrv);
    // Установка текущего шейдера и заполнение таблиц ресурсов
    shader->ConfigurePipeline(*commandBundle);
    // Запуск вычислений
    commandBundle->Dispatch(
        window.GetWidth(), window.GetHeight(), 1);
}
```

Апробация

Для апробации решения была проведена презентация данной технологии на семинаре в МФТИ. Лаборатория методики ядерно-физического эксперимента [24], проводящая данный семинар, занимается исследованиями в области ядерной физики и разработкой компьютерных симуляций физических процессов данного типа.

В конце презентации был проведен опрос для оценки удобства и применимости данной технологии.

В результате апробации были подчеркнуты следующие достоинства библиотеки.

- Удобная загрузка шейдеров и привязка ресурсов к ним.
- Автоматическая расстановка барьеров.
- Удобная загрузка карт высот и использование сцены на GPU.

Также были выявлены недостатки.

- Сложность копирования данных между CPU и GPU.
- Неудобная установка.

Общая оценка системы, полученная в результате опроса, представлена в таблице 1. По каждому критерию оценка проводилась по шкале от 1 до 10.

Критерий	Средний результат
Простота использования	6.86
Достаточность возможностей	9.57

Таблица 1: Оценка системы, полученные во время апробации

Заключение

На данный момент достигнуты следующие задачи.

1. Проведен обзор существующих работ по симуляции физики частиц на видеокартах.
2. Разработана архитектура движка симуляции частиц.
3. Реализован фреймворк для разработки алгоритмов симуляций ядерной физики.
4. Проведена апробация реализованного решения.

Исходный код открыт и доступен в GitHub [2].

[2] C++ часть: https://github.com/F5DXWsqPme/RenderGraph (дата обращения: 14.12.2022)
C# часть: https://github.com/F5DXWsqPme/ShadersCodegen (дата обращения: 07.05.2023)

Список литературы

[1] [1904.03435] TURTLE: A C library for an optimistic stepping through a topography. — URL: https://arxiv.org/abs/1904.03435 (дата обращения: 2022-12-09).

[2] Achieved Occupancy. — URL: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm (дата обращения: 2022-09-18).

[3] Akenine-Moller T. Haines E. Hoffman N. Real-Time Rendering, Fourth Edition. — A K Peters/CRC Press, 2018. — P. 837–821. — ISBN: 9781351816144.

[4] CUDA C++ Programming Guide. — URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-memory-accesses (дата обращения: 2022-12-10).

[5] CUDA Zone - Library of Resources | NVIDIA Developer. — URL: https://developer.nvidia.com/cuda-zone (дата обращения: 2022-09-18).

[6] D3D12 Memory Allocator - GPUOpen. — URL: https://gpuopen.com/d3d12-memory-allocator (дата обращения: 2022-12-11).

[7] D3D12_RESOURCE_STATES - Win32 apps | Microsoft Learn. — URL: https://learn.microsoft.com/en-us/windows/win32/api/d3d12/ne-d3d12-d3d12_resource_states (дата обращения: 2023-05-06).

[8] DD2018: Adam Sawicki - Porting your engine to Vulkan or DX12 - YouTube. — URL: https://youtu.be/6NWfznwFnMs (дата обращения: 2022-12-11).

[9] DIDSR/MCGPU: GPU-accelerated Monte Carlo x-ray transport code

to simulate medical x-ray imaging devices. — URL: https://github.com/DIDSR/MCGPU (дата обращения: 2023-03-26).

[10] Direct3D - Win32 apps | Microsoft Learn. — URL: https://learn.microsoft.com/en-us/windows/win32/direct3d (дата обращения: 2022-12-11).

[11] DirectX Raytracing - PIX on Windows. — URL: https://devblogs.microsoft.com/pix/directx-raytracing (дата обращения: 2022-12-11).

[12] DirectX Raytracing (DXR) Functional Spec | DirectX-Specs. — URL: https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html (дата обращения: 2022-12-11).

[13] DirectX-Specs | Engineering specs for DirectX features. — URL: https://microsoft.github.io/DirectX-Specs/d3d/CPUEfficiency.html#detailed-api-descriptions (дата обращения: 2022-12-11).

[14] DirectX-Specs | Engineering specs for DirectX features. — URL: https://microsoft.github.io/DirectX-Specs/d3d/HLSL_ShaderModel6_5.html (дата обращения: 2022-12-12).

[15] Documentation - PIX on Windows. — URL: https://devblogs.microsoft.com/pix/documentation (дата обращения: 2022-12-11).

[16] Exploiting Hardware-Accelerated Ray Tracing for Monte Carlo Particle Transport with OpenMC | IEEE Conference Publication | IEEE Xplore. — URL: https://ieeexplore.ieee.org/abstract/document/9059266 (дата обращения: 2022-09-12).

[17] Features — RenderDoc documentation. — URL: https://renderdoc.org/docs/getting_started/features.html (дата обращения: 2022-12-11).

[18] Frontiers | Evaluation of Single-Node Performance of Parallel Algorithms for Multigroup Monte Carlo Particle Transport Methods. — URL: https://www.frontiersin.org/articles/10.3389/fenrg.2021.705823/full (дата обращения: 2022-09-17).

[19] High-level shader language (HLSL) - Win32 apps. — URL: https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl (дата обращения: 2023-05-06).

[20] Home | Vulkan | Cross platform 3D Graphics. — URL: https://www.vulkan.org (дата обращения: 2022-12-11).

[21] ID3D12GraphicsCommandList::Dispatch (d3d12.h) - Win32 apps. — URL: https://learn.microsoft.com/en-us/windows/win32/api/d3d12/nf-d3d12-id3d12graphicscommandlist-dispatch (дата обращения: 2023-05-06).

[22] ID3D12GraphicsCommandList::ResourceBarrier (d3d12.h) - Win32 apps. — URL: https://learn.microsoft.com/en-us/windows/win32/api/d3d12/nf-d3d12-id3d12graphicscommandlist-resourcebarrier (дата обращения: 2023-05-06).

[23] LLNL Monte Carlo Transport Research Efforts for Advanced Computing Architectures. — URL: https://www.kns.org/files/int_paper/paper/MC2017_2017_2/P095S02-02BrantleyP.pdf (дата обращения: 2022-09-18).

[24] MIPT-NPM laboratory. — URL: https://npm.mipt.ru/ru/about (дата обращения: 2023-05-06).

[25] Modeling parameterized geometry in GPU-based Monte Carlo particle transport simulation for radiotherapy - IOPscience. — URL: https://iopscience.iop.org/article/10.1088/0031-9155/61/15/5851/meta (дата обращения: 2022-09-17).

[26] NVIDIA OptiX™ Ray Tracing Engine | NVIDIA Developer. — URL: https://developer.nvidia.com/rtx/ray-tracing/optix (дата обращения: 2022-12-10).

[27] NVIDIA Turing Architecture In-Depth | NVIDIA Technical Blog. — URL: https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth (дата обращения: 2022-12-09).

[28] The OpenMC Monte Carlo Code — OpenMC Documentation. — URL: https://docs.openmc.org/en/stable (дата обращения: 2022-09-12).

[29] RenderDoc — RenderDoc documentation. — URL: https://renderdoc.org/docs/index.html (дата обращения: 2022-12-11).