О НЕКОТОРЫХ МЕТОДАХ ВИЗУАЛИЗАЦИИ ДИНАМИЧЕСКИХ 3D МОДЕЛЕЙ

М.Г. Городничев, *доцент кафедры «Математическая кибернетика и информационные технологии» МТУСИ, к.т.н. gorodnichev89@yandex.ru;*
Р.А. Гематудинов, *доцент кафедры «Автоматизация производственных процессов» МАДИ, к.т.н., rinatg86@mail.ru;*
А.М. Кухаренко, *студент РЭУ им. Г.В. Плеханова, alexandra.kukharenko@yandex.ru*

УДК 004.05

Аннотация. Проблема вычислений затратных по производительности алгоритмов в режиме реального времени вынуждает разработчиков программного обеспечения задумываться о поисках механизмов по оптимизации данных вычислений. Объектом исследования авторов является визуализация на основе алгоритма трассировки лучей с помощью параллельных вычислений. В данной статье исследована производительность алгоритма трассировки лучей посредством параллельных вычислений.

Ключевые слова: алгоритм; модель; параллельные вычисления; архитектура; оптимизация.

SOME METHODS OF RENDERING DYNAMIC 3D MODELS

Mikhail Gorodnichev, *associate professor of "Mathematical cybernetics and information technology" department MTUCI, Ph.D.;*
Rinat Gematudinov, *associate professor of "Automation of production processes" department MADI, Ph.D.;*
Alexandra Kukharenko, *student of the Plehanov Russian university of economics*

Annotation. The problem of computing performance-intensive algorithms in real time forces software developers to think about finding mechanisms to optimize these calculations. The object of research in this work is the visualization using the ray tracing algorithm using parallel calculations. In this paper the performance of ray tracing algorithm by means of parallel calculations is investigated.

Keywords: algorithm; model; parallel computing; architecture; optimization.

Постановка задачи

Целью проведенного авторами исследования является реализация алгоритмов визуализации с помощью параллельного вычисления на графическом процессоре в режиме реального времени.

Необходимо разработать и исследовать программное обеспечение (ПО) визуализации методом трассировки лучей посредством параллельных вычислений. Для решения поставленной задачи следует решить две подзадачи: реализация алгоритма трассировки лучей и оптимизация данного алгоритма для получения максимального результата в режиме реального времени.

Для оптимизации алгоритма важно понимание модели параллельного программирования. Для этого рассмотрим архитектуру модели NVIDIA CUDA с целью ее дельнейшего применения.

Метод «Трассировки лучей»

Суть данного метода заключается в отслеживании взаимодействия оптического луча, пущенного от точки обзора до источника освещения с моделями сцены.

Существуют несколько типов лучей при рассмотрении данного метода. Основные из них, это: луч обзора; луч, исходящий от точки обзора до объекта сцены; теневой луч (shadow

ray); луч от объекта сцены до источника сцены, который определяет нахождение объекта в тени от данного источника.

Луч обзора, луч началом которого является точка обзора, проходит через точку конечного изображения и заканчивается точкой первого пересеченного им объекта. При этом данный луч может не иметь конца на каком-либо объекте сцены, тогда конец луча уходит в бесконечность. При получении конечной точки луча, значение цвета и освещенности проецируются на плоскость изображения в точку прохождения данного луча. Для определения значения цвета данной точки проводят еще один луч обзора, началом которого является данная точка, а направление определяется типом материала.

Shadow ray – луч, который начинается в точке падения луча обзора и распространяется в направлении источника света.

Теневой луч определяет наличие тени от источника освещения в точке падения луча обзора. Если между точкой объекта и источником света находится другой объект сцены, то в точке объекта присутствует тень, параметры которой зависят от типа материала объектов между данной точкой и источником света.

Для каждого взаимодействия луча обзора с объектом существуют еще два луча: луч преломления и луч отражения.

Существует два типа отражения: зеркальное отражение, когда угол падения к поверхности равен углу отражения, и диффузное отражение, при котором угол отражения различается с углом падения.

Диффузное отражение происходит от шероховатой поверхности. При этом происходит рассеивание света во всех направлениях. При диффузном отражении, существует множество лучей, исходящих в разные направления из точки падения луча, интенсивность которых увеличивается с уменьшением угла.

Поскольку падающий луч света достаточно мал, он взаимодействует с микроскопической частью объекта. Отражение происходит от множества поверхностей разной ориентации, при этом образуются лучи отражения в различных направлениях. Несмотря на это, интенсивность данных отразившихся лучей различна. Максимальная интенсивность лучей отражения у углов, направленных к источнику света, при увеличении отклонения интенсивность лучей отражения снижается. Зависимость диффузного отражения характеризуется функцией Генриха Иоганна Ламберта, которая показывает пропорциональную зависимость к косинусу угла между нормалью к поверхности и лучом источника света: $I = I_0 \cdot cos(\theta)$, где, θ – угол между нормалью к поверхности и лучем источника света.

В отличии от диффузно-отражающей поверхности, полированные поверхности даже на микроскопическом уровне являются гладкими, при этом отражение происходит от одной поверхности. Поверхности этого типа при отражении лучей создают такие визуальные эффекты, как зеркальное отражение, блеск, блики и глянец.

При процессе зеркального отражения, угол отражения такой же, как и угол падения луча на поверхность. При отражении от зеркальной поверхности, изображения сохраняются. Тем не менее, большинство отражений происходят не от идеальной поверхности посеребренного зеркала.

Существует два наиболее значимых случая, при котором отражение не идеально. Они представляют собой частичные отражения и изогнутые поверхности. Частичные зеркальные отражения наиболее распространены для неметаллических поверхностей. Примером изогнутых поверхностей могут служить: глянцевая бумага, мокрая дорога, стеклянные стаканы. В местах изгиба отраженное изображение искажается. При единичном большом изгибе искажения будут выражены в увеличении или уменьшении масштаба изображения. Множественные малые изгибы и неровности сделают изображения более диффузными или размытыми при угле отражения равным углу падения.

При взаимодействии с большинством объектов отражение не может характеризоваться как строго диффузное или строго зеркальное. Для этих объектов применимы оба вида отражения.

Алгоритм работы метода «Трассировки лучей»

Для исследования значения каждого пикселя конечного изображения, из виртуальной камеры испускается множество первичных лучей (primary ray) [1]. Каждый луч из данного множества проходит через каждый пиксель конечного изображения. После установления направления первичного луча, каждый объект сцены проверяется на пересечение с данным лучом. Объектов на пути следования луча может быть несколько. Из данного множества объектов пересечения выбирается ближний к точке просмотра. Далее испускается теневой луч (shadow ray) из точки пересечения первичного луча и объекта к источнику света. При этом отслеживается пересечение данного луча с другими объектами, если теневой луч не пересекает на пути к источнику света других объектов, то точка освещена, при пересечении с другими объектами точка находится в тени. После вычисления освещенности точки из нее испускается луч отражения (reflection ray). Вычисление данного луча подобно вычислению первичного луча.

Достоинством данного алгоритма является слабая зависимость вычислительной сложности метода от сложности сцены, а также есть возможность распараллеливания вычисления. Например, можно параллельно вычислять несколько лучей или разделять экран на части.

Главным недостатком метода является требование к высокой производительности системы. Для рендеринга нового изображения требуется выполнить вычисления для каждого луча заново.

Технология NVIDIA CUDA

Nvidia CUDA Compiler (NVCC) – это компилятор, разработанный и запатентованный компанией Nvidia. Данный компилятор предназначен для раздельного компилирования исходного кода CUDA. Исходный код CUDA предназначен для работы на процессорах CPU и GPU. NVCC при компилировании разделяет код. Код для CPU отправляется на компилирование с помощью таких компиляторов, как GCC (GNU C Compiler), или ICC (Intel C++ Compiler) или Microsoft Visual C Compiler. Код для GPU компилирует для работы на процессоре GPU.

Фаза компиляции – это логическое действие компилятора при компилировании всего проекта, которое может быть выбрано посредством параметров командной строки для NVCC. NVCC может разбить фазы компиляции на более мелкие шаги, на варианты реализации фазы, которые могут быть изменены при выходе новых версий CUDA в отличие от фаз.

NVCC выбирает фазы, комбинируя параметры командной строки и расширения входных файлов. Входные фазы определяются с помощью расширений входных файлов. Для определения выходных фаз используются параметры командной строки.

Этапы компилирования проекта

Компилирование CUDA файлов происходит с помощью преобразования исходных файлов, закодированных в расширенном языке CUDA в обычные исходные файлы ANSI C++, которые передаются компилятору C++ общего назначения.

При выполнении компиляции CUDA файлов входные файлы обрабатываются для компиляции на устройстве. При дальнейшей обработке исходные файлы компилируются в бинарные файлы cubin и промежуточные файлы PTX. Для создания встраиваемого файла fatbinary и преобразования специфичных C++ расширений в стандартные конструкции компилятор еще раз обрабатывает исходные файлы для компиляции на устройстве. На

последнем этапе C++ компилятор компилирует преобразованный код с встраиваемой fatbinary в объект хоста.

Архитектура параллельных вычислений Nvidia CUDA

Появление многоядерных графических процессоров GPU и многоядерных процессоров CPU дало возможность для разработки параллельных систем. При этом параллелизм данных систем масштабируется по закону Мура. На данный момент главной задачей масштабирования параллельности является увеличение эффективности использования большого количества процессорных ядер.

Модель параллельного программирования CUDA предназначена для решения данной проблемы, при этом она увеличивает доступность модели для программистов, знакомых со стандартными языками программирования, как C. В качестве минимального набора расширения языка программирования модель параллельного программирования содержит в себе три ключевые абстракции такие, как иерархия групп потоков, общая память (Shared memory) и барьерная синхронизация.

Данные абстракции обеспечивают параллелизм мелких данных и параллелизм потоков, которые встроены в параллелизм задач и параллелизм крупных данных. Все это позволяет разделить задачу на крупные подзадачи, которые решаются параллельно блоками мелких подзадач, которые можно решить параллельно с помощью потоков внутри блока.

Такое разложение позволяет потокам взаимодействовать при решении каждой подзадачи, в то же время обеспечивает автоматическую масштабируемость. Каждый блок потоков может выполняться на любом доступном мультипроцессоре GPU в любой последовательности, параллельно или последовательно, таким образом скомпилированная CUDA программа может выполняться на любом количестве мультипроцессоров, при этом на этапе исполнения, программа должна знать физическое число мультипроцессоров.

Архитектура NVIDIA GPU построена вокруг масштабируемого массива много потоковых мультипроцессоров (Streaming multiprocessors, SMs). При вызове функции ядра программой CUDA на хост-процессоре, блоки сетки распределяются по доступным мультипроцессорам GPU. На одном мультипроцессоре может одновременно выполняться, как потоки одного блока потоков, так и несколько блоков сразу. При завершении выполнения блоков потоков на освободившиеся мультипроцессоры запускаются для выполнения новые блоки потоков.

Мультипроцессор одновременно может выполнять несколько сотен потоков. Для управления таким большим количеством потоков используется архитектура SIMT (Single-Instruction, Multi-Tread). Конвейер инструкций использует параллелизм на уровне инструкций при выполнении одного потока. Также широкий параллелизм достигается на уровне потоков за счет одновременного использования аппаратной многопоточности (Hardware Multithreading). В отличии от ядер процессора CPU они выполняются по порядку, однако, нельзя предсказать ветвления и ход выполнения потоков.

Архитектура SIMT

Мультипроцессор выполняет такие действия, как создание, управление, распределение и выполнение потоков в группах по 32 параллельных потока в каждой [2]. Такие группы обозначаются, как основы (warp). Отдельные потоки из основной группы начинают выполняться вместе по одному адресу программы, при этом имея различный адрес команды счетчика и регистра состояние. Это позволяет потокам, образующим основу, быть независимыми друг от друга и свободно разветвляться.

При передаче мультипроцессору блоков потоков, он разбивает их на основы и распределяет их в расписании на выполнение с помощью планировщика основ. Мультипроцессор всегда разбивает потоки на основы одинаково. Потоки в основах содержатся

последовательно, при этом их идентификаторы увеличиваются с первой основы, содержащей поток с индексом 0.

Каждая основа выполняет одну общую инструкцию за один раз, поэтому наибольшая эффективность реализуется, когда все 32 потока в основе синхронизируются в процессе их выполнения. Если потоки основы расходятся из-за условной ветки зависящих данных, основа последовательно выполняет все ветви пути, исключая потоки, которые не должны выполняться по данному пути, а после завершения выполнения всех путей, потоки возвращаются на изначальный путь выполнения. Различные основы выполняются независимо друг от друга, даже при общих путях выполнения.

Для архитектуры SIMT характерен признак архитектуры SIMD (Single Instruction, Multiple Data) такой, как контроль одной командой нескольких элементов обработки. Ключевым различием является то, что в отличие от архитектуры SIMD, архитектура SIMT позволяет писать параллельный код на уровне потоков для независимых, скалярных потоков, в то время, когда параллельность на уровне данных требует скоординированных потоков. При написании приложений поведение SIMT может игнорироваться разработчиком, но для увеличения производительности потоки в основе должны как можно менее расходиться.

Аппаратная многопоточность

Контекст выполнения, такой как счетчики и регистры, для каждой основы обрабатываемой мультипроцессором поддерживается все время жизни основы. Благодаря этому, затраты на переключение с одного контекста исполнения к другой минимальны, и в каждый момент времени, планировщик выбирает основу с активными потоками, которые готовы к выполнению следующих инструкций. Для этого у каждого мультипроцессора есть набор 32-битных регистров, которые разделяются между основами и параллельный кэш данных или расширенная память, которая разделяется между блоками потоков.

Число блоков и перекосов, которые могут обрабатываться вместе на мультипроцессоре для данной функции ядра зависит от количества регистров и памяти, используемой данной функцией ядра, а также количеством регистров и общей памяти, доступной на мультипроцессоре. Кроме того, существует ограничение на число максимальных постоянных блоков и основ. Эти ограничения также ограничивают количество регистров и расширенной памяти, доступной на мультипроцессоре, и зависят от характеристик устройства. Если нет достаточного количества регистров и памяти, хотя бы на один блок, функция ядра не сможет запуститься.

Иерархия потоков

В функцию ядра передается несколько трехмерных векторов, первый из которых threadIdx, идентифицирующий потоки с использованием одно-, дву- и трехмерным индексом потока, тем самым образуя блок потоков. Это обеспечивает естественный способ вычисления таких структур, как вектор, матрица или объем.

Индекс потока и его идентификатор связаны друг с другом:

1. Для одномерного блока индекс потока совпадает с идентификатором.
2. Для двумерного блока размера $(D_x; D_y)$ идентификатор потока с индексом $(x; y)$ определяется $i = x + D_x \cdot y$
3. Для трехмерного блока размера $(D_x; D_y; D_z)$ идентификатор потока с индексом $(x; y; z)$ определяется $i = x + yD_x + zD_xD_y$

Существует ограничение на количество потоков в блоке, это связано с тем, что все потоки одного блока должны находиться на одном ядре процессора и использовать ресурсы памяти этого ядра. На данный момент, это ограничение равно 1024 потока на каждый блок потоков.

Тем не менее, возможно выполнение параллельно нескольких блоков, при этом общее количество потоков станет равно количеству потоков в одном блоке, умноженное на

количество блоков. Данные блоки организуются в *n*-мерные сетки. Количество блоков в сетке обычно определяется размерами обрабатываемых данных или количеством мультипроцессоров в системе.

Блоки потоков должны выполняться независимо друг от друга в любом порядке. При этом обеспечивается независимость от параллельного или последовательного выполнения. Данное требование позволяет планировать выполнение блоков в любом порядке, не зависимо от числа ядер в системе.

Иерархия памяти

Потоки CUDA могут получить доступ к данным из нескольких ячеек памяти во время их выполнения. Для этого каждый поток имеет свою локальную память, каждый блок имеет расширенную память, видимую только потокам этого блока, каждый поток также может обращаться к глобальной памяти, которая доступна всем потокам.

В дополнение к этому существует еще два типа памяти, к которой потоки имеют доступ только в режиме чтения – это постоянное пространство памяти и текстурное. Текстурная память использует различную адресацию памяти, а также фильтрацию для различных специфичных форматов данных.

Модель программирования CUDA предполагает систему с разделением на хост-устройство и графическим устройством, каждое из данных устройств имеет свою отдельную память.

AMD OpenCL

Модель программирования OpenCL [3] основывается на понятии хост-устройства, которое поддерживается с помощью основного API приложения, и на устройствах, подключаемых через шину. Такие устройства поддерживаются API OpenCL C. API хост-устройства разделено на различные платформы и на группу слоев, поддерживающих разные среды выполнения. OpenCL C представляет расширенный язык программирования C для параллельного программирования. Как пример такого расширения могут быть представлены операции над памятью и барьеры.

Модель программирования OpenCL позволяет устройствам работать с данными и задачами параллельного вычисления. Ядро при этом выполняется в виде функции, работающей в многомерной области индексов. В данной области каждый элемент называется рабочим элементом (work-item), а общая область элементов рабочей областью (work-size). Рабочую область можно разделить на под-домены и рабочие группы (work-group). Каждый рабочий элемент при этом может взаимодействовать с расширенной локальной памятью или глобальной областью памяти. Рабочие элементы синхронизируются с помощью барьеров и ограничивающих операций.

Приложения OpenCL построены таким образом, что первый запрос от среды выполнения нацелен на определение платформ, присутствующих в системе с последующим выбором нужной. В ходе следующего шага создается контекст OpenCL из связанных устройств. При этом обеспечивается оптимальная согласованность этих устройств в ходе работы. Для обеспечения этого объекты памяти, такие как буферы и изображения, выделяются над контекстом. При этом гарантируется видимость изменений данных объектов другим устройствам в определенных точках синхронизации.

Выполнение работы с устройством обеспечивается с помощью очереди команд на соответствующем устройстве. При этом организуются очереди команд для каждого устройства контекста, работа с данным устройством происходит только через данный интерфейс.

Требования к разрабатываемому приложению:

- Разрабатываемое приложение должно осуществлять визуализацию объектов сцены.
- Разрабатываемое приложение должно осуществлять визуализацию в режиме реального времени.

- Разрабатываемое приложение должно корректно реализовывать алгоритм трассировки лучей.
- Разрабатываемое приложение должно уметь принимать на вход файлы с информацией о сцене и объектах данной сцены.
- Разрабатываемое приложение должно реализовываться с использованием технологии параллельного вычисления.

Структурное программирование

Структурное программирование [4] является одной из парадигм программирования, направленной на улучшение таких характеристик разработки программного обеспечения, как чистота и качество исходного кода и время разработки. Для этого широко применяются подпрограммы, блочные структуры, циклы и ветвления, в отличие от простых выражений, как прыжки или метка goto.

Теорема структурного программирования гарантирует, что любая часть блок-схемы программы может быть разложена на три типа конструкций, таких как:

- f THEN g – блок-схема f следует за блок-схемой g.
- IF p THEN f ELSE g – ветвление, при условии p выполняется блок-схема f иначе блок-схема g.
- WHILE p DO f – цикл, блок-схема f циклически выполняется пока условие p истинно.

где: f, g – блок-схемы с одним входом и одним выходом, p – условие проверки, IF, THEN, ELSE, WHILE, DO – ключевые слова.

Блок-схема может быть любого размера, но при этом теорема структурного программирования гарантирует, что логика управления может быть представлена в конечном базисе с соответствующим снижением сложности характеристики произвольных блок-схем. Это обеспечивает каноничность форм для документации и тестирующих программ.

Вторым следствием теоремы структурного программирования является возможность прочтения программы в режиме «Top-Down». При данном методе прочтения программы сначала исследуют большие логические блоки алгоритма, потом, при надобности, каждый такой блок исследуется по частям. При написании приложений в режиме «Top-Down», сначала создают интеграционную часть, затем подсистемы приложения и, в последнюю очередь, функциональные модули.

Процедурная парадигма программирования

Процедурная парадигма программирования использует в качестве объектов линейные процедуры или связанные в единый блок последовательность операторов программирования [5]. Ключевой особенностью парадигмы является разделение функций в небольших дискретных многоразовых модулей, которые действуют как маленькие программы сами со своими собственными входными и выходными параметрами. Процедурный пример кода выполнен из единой точки управления или точки входа.

Парадигма процедурного программирования была одной из первых парадигм высокого уровня и в настоящее время является наиболее распространенной и хорошо понимаемой формой программирования. Новые парадигмы построены на принципах процедурного программирования.

Для процедурного программирования характерна модульность, особенно в больших и сложных проектах. При этом входные параметры определяются синтаксическими правилами языка программирования, а выходные параметры при этом передаются как возвращаемые значения.

Для поддержания модульности процедур существует разграничение области видимости. Обеспечение области видимости позволяет ограничивать доступ к переменным из других модулей, в том числе предыдущих экземпляров переменных.

Возможность организации простого, самодостаточного интерфейса делает процедуры удобным средством для разработки программного обеспечения.

Объектно-ориентированное программирование

Объектно-ориентированное программирование является парадигмой программирования, основанной на манипулировании «объектов», которые могут содержать данные и код. Методы объекта могут получать доступ к полям, с которыми связан объект. При построении программ объекты чаще всего взаимодействуют друг с другом.

Языки, поддерживающие ООП, используют наследование для повторного использования кода и расширяемость с помощью классов и прототипов. При использовании классов языки программирования поддерживают два понятия:

- Классы – определения для формата данных, доступных процедур для типа или класса объекта. Классы содержат в себе данные и процедуры-методы класса.
- Объект – экземпляр класса.
- Объекты могут соответствовать вещам, которые характеризуют, или более абстрактные объекты, такие как услуги.

Каждый объект программы является экземпляром определенного класса, процедуры которого являются методами, а переменные – поля или атрибуты класса. Для объекта характерны следующие термины:

- Переменные класса – относятся к классу в целом, существуют в единственной копии.
- Атрибут класса – данные, характерные для определенного объекта. Каждому объекту сопоставлен отдельный атрибут.
- Методы класса – относятся к классу в целом и имеют доступ только к переменным класса и входным параметрам от вызова процедуры.
- Методы экземпляра – относятся к отдельному экземпляру, имеют доступ к переменным экземпляра для соответствующего объекта и входным параметрам и переменным класса.

Объекты доступны как переменные со сложной внутренней структурой, в большинстве языков они являются указателями. Структура объектов состоит из реальных ссылок на один экземпляр указанного объекта в памяти внутри кучи или стека. При этом обеспечивается уровень абстракции, позволяющей разделять внутренний от внешнего кода. Внешний код использует объект с помощью вызова метода экземпляра, также данный код может читать или перезаписывать переменную экземпляра. При создании объекта процедура вызывает специальную функцию-конструктор. Программа может создавать несколько независимо работающих экземпляров одного класса.

Особенности реализации

Для реализации приложения визуализации, в качестве основной парадигмы программирования выбрана парадигма объектно-ориентированного программирования. Для данного проекта разбиение на классы можно сделать следующим образом:

- frameworkManager – основная задача данного класса организовывать и управлять окном и событиями.
- sceneManager – класс, отвечающий за хранение и манипулирование информацией о сцене.
- renderManager – класс, отвечающий за рендеринг сцены при запросе.

Далее рассмотрен каждый класс более подробно. Также представлена схема взаимодействия классов.

Реализация параллельного программирования с помощью технологии Nvidia CUDA [6]

Для организации параллельного программирования требуется, чтобы функция, которая выполняется над несколькими потоками, была глобальной. Данной функцией будет одна из функций класса sceneManager, которая определяет параметры первичных лучей и запускает дальнейшие функции. Методы поиска пересечения луча с объектами и определения цвета и материала объекта определяются, как __device__, данные функции будут выполняться только на устройстве GPU.

Класс renderManager

При проектировании данного класса были описаны основные методы класса:

- rendFrame – открытый метод класса – на вход принимает сцену, которую нужно визуализировать, параметры экрана. Возвращает массив пикселей, изображение, которое в дальнейшем потребуется вывести на экран.
- rayTracing – закрытый метод класса – на вход принимает адрес памяти GPU с информацией о сцене, параметры окна и адрес памяти GPU с результатом. Данный метод запускает трассировку первичных лучей, а также вычисляет данные лучи.
- colorForRay – закрытый метод класса – на вход принимает луч, сцену в которой его требуется трассировать, и номер отражения. Данный метод требуется для трассировки луча в сцене и возвращения значения пикселя.

Поскольку трассировка луча должна выполняться на устройстве GPU для данного класса требуется реализовать технологию Nvidia CUDA. Для этого требуется функция подготовки памяти устройства GPU и вызова функции-ядра. В качестве такой функции реализована функция rendFrame. В качестве функции-ядра реализована функция rayTracing. Для реализации алгоритма трассировки луча требуется функция с рекурсивным вызовом, чтобы вызывать саму себя для трассировки луча отражения. Такая функция реализована в методе класса colorForRay.

Для реализации класса renderManager потребуется два файла: файл с описанием класса и его методов – renderManager.hpp и файл с определением всех методов класса с возможностью компиляции CUDA – renderManager.cu.

Реализация класса frameWorkManager

От данного класса требуется выполнение нескольких основных задач, связанных взаимодействием с операционной системой и внутренней системой рендеринга:

- Инициализация и открытие окна, инициализация рендера.
- Взаимодействие с библиотекой SDL.
- Взаимодействие с операционной системой, обработка событий, возникающих при работе пользователя с окном программы.
- Взаимодействие с системой рендеринга. Для реализации обновления экрана при обновлении состояния сцены требуется постоянная обработка сцены.

Для реализации всех требований к данному классу, возможно разбиение на два подкласса, которые будут выполнять функции обработки окна и обработки событий операционной системы. При этом в основном классе будут присутствовать следующие методы:

- Конструктор класса frameworkManager принимает на вход параметры окна и в процессе своей работы инициализирует окно с помощью конструктора подкласса windowManager.
- Деструктор класса очищает все требуемые классом ресурсы и деинициализирует окно.
- Метод класса manageWindow запускает с помощью подкласса eventManager обработку событий операционной системы.

При работе данному классу требуется работа с операционной системой с ресурсами CPU, поэтому выполнение функций на устройстве GPU не требуется.

Реализация класса sceneManager

Класс sceneManager разделен на несколько подклассов, каждый из которых отвечает за свой уровень графических примитив сцены, таких уровней, как объект, полигон или вершина. Также в число подклассов вошли классы, описывающие камеру и материал объекта.

Класс sceneManager выполняет несколько функций, которые обеспечивают работу других классов программы со сценой и доступ к данным, хранящимся в подклассах:

- Функция работы с внешними данными. Реализация данной функции обеспечивает загрузку сцены из конфигурационных файлов с помощью библиотеки libconfig++ и загрузку объектов сцены с помощью библиотеки FBX.
- Доступ к данным при надобности обеспечивается методами класс носителя данных.
- Функция взаимодействия с классом renderManager. Для данного класса требуется найти точку пересечения луча и объектов сцены. Класс sceneManager ищет точку пересечения и характеристики материала и луч отражения и возвращает классу визуализации.
- Функция взаимодействия с классом frameworkManager для изменения положения камеры и объектов. Реализуется с помощью функций манипулирования объектами сцены.

Для реализации первой функции требуется функция loadedObj, которая получает на вход имя файла с параметрами сцены. В ходе выполнения создается нужное количество объектов сцены, далее происходит загрузка нужных объектов, которая происходит с помощью библиотеки FBX и функции подкласса object.

Для реализации второй функции в каждом подклассе существуют функции, возвращающие и принимающие значения полей класса.

Третья функция требует реализации функции поиска в классах sceneManager, object и polygon. Каждая из данных функций, кроме функции в классе polygon, просматривает объекты или полигоны, вызывая функцию поиска для каждого объекта. В функции поиска пересечения луча с полигоном, находится точка пересечения луча и плоскости в которой лежит полигон, и в дальнейшем проверяется принадлежность этой точки многоугольнику – полигону.

Первая и четвертая функции должны быть реализованы только для выполнения на устройстве CPU, в то время как третья и вторая функции должны быть скомпилированы для выполнения на устройстве GPU.

Результат работы программы

В результате работы разработанного программного обеспечения программа выводит на экран визуализацию сцены с помощью метода трассировки лучей. При этом учитывается время, с которой работает алгоритм трассировки лучей.

В ходе разработки программного обеспечения удалось добиться определенного уровня оптимизации благодаря внедрению технологии параллельного вычисления. Но для лучшей производительности и лучшего качества изображения требуется система с как можно мощными характеристиками.

При создании приложения использовалась система со следующими характеристиками:

- Операционная система: Arch Linux x86_64.
- Процессор: Intel i7-4710HQ 2.5 GHz.
- GPU: Nvidia GeForce GTX 850M.
- CUDA Version 7.5.

При данных характеристиках достигается умеренная производительность и основной параметр – количество кадров в секунду, в среднем равен 30 на простых сценах. Для комфортного использования системы требуются более мощные характеристики системы.

Заключение

Визуализация реалистичных 3D моделей в режиме реального времени на данный момент перспективно развивается. Различные технологии визуализации используются в различных индустриях, таких как визуализация объектов виртуальной реальности в различных видах тренажеров, например, авиа-тренажеры.

Взаимодействие алгоритма трассировки лучей и технологии параллельного программирования Nvidia CUDA позволяет оптимизировать вычисления алгоритма.

В процессе разработки программного обеспечения, реализующего алгоритм трассировки лучей в режиме реального времени, использовались различные дополнительные библиотеки, обеспечивающие взаимодействие с операционной системой, ресурсными файлами и технологией параллельного программирования.

В качестве технологии параллельного программирования принята технология Nvidia CUDA, обеспечивающая параллельные вычисления на базе вычислительных процессоров GPU от производителя Nvidia. В качестве аналогов рассмотрена технология AMD OpenCL от производителя видеочипов AMD.

При проектировании приложения были рассмотрены такие парадигмы, как структурное программирование, процедурное программирование и объектно-ориентированное программирование. При проектировании использовалась парадигма ООП, обеспечивающая хороший уровень стилизации кода и разделения функций, требуемых для реализации приложения.

Литература

1. Introduction to Ray Tracing: a Simple Method for Creating 3D Images [Электронный ресурс] //scratchapixel:URL:http://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/implementing-the-raytracing-algorithm

2. MOORE G.E. Cramming More Components onto Integrated Circuits, 1998.

3. Advanced Micro Devices I. OpenCL User Guide. – Sunnyvale. 2015.

4. Cockshott P. SIMD Programming Manual for Linux and Windows. – London: Springer, 2004.

5. Juds S. Photoelectric Sensors and Controls: Selection and Application, First Edition. – New York: Marcel Dekker, inc, 1988.