Федеральное агентство связи
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Кафедра вычислительных систем
Допустить к защите

Зав.каф. д.т.н. доцент
_____ *Мамойленко С. Н.*

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Реализация алгоритма трассировки пути
с использованием технологии OpenCL

Пояснительная записка

Студент _____ / Терешков Р. В. /

Факультет ИВТ Группа ИВ-222

Руководитель _____ / Фульман В. О. /

Новосибирск 2016 г.

Федеральное агентство связи
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

КАФЕДРА

ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

СТУДЕНТА Терешкова Р. В. ГРУППЫ ИВ-222

УТВЕРЖДАЮ

« ___ » _____ 20__ г.

Зав. кафедрой д.т.н. доцент
_____ / *С. Н. Мамойленко* /

Новосибирск 2016 г.

1. Тема выпускной квалификационной работы бакалавра
Реализация алгоритма трассировки пути с использованием технологии OpenCL.

утверждена приказом СибГУТИ от «20» февраля 2016 г. № **4/159о-16**

2. Срок сдачи студентом законченной работы «10» июня 2016 г.

3. Исходные данные к работе
1. Kevin Suffern. Ray Tracing from the Ground Up. – A. K. Peters, Ltd. : Los Altos, California, 2007. – 782 p.
2. The OpenCL Programming Book / Ryoji Tsuchiyama [et al.]. – Fixstars, 2012. – 324 p.

4. Содержание пояснительной записки (перечень подлежащих разработке вопросов)	**Сроки выполнения по разделам**
Введение	17.05
Описание предметной области	18.05-25.05
Реализация алгоритма	26.05-05.06
Результаты	06.06-08.06
Заключение	09.06

Дата выдачи задания «10» мая 2016 г.
Руководитель _____
подпись

Задание принял к исполнению «10» мая 2016 г.
Студент _____
подпись

АННОТАЦИЯ

Выпускная квалификационная работа Терешкова Р. В.
(Фамилия,И.О.)
по теме «Реализация алгоритма трассировки пути с использованием технологии OpenCL»

Объём работы – *48* страниц, на которых размещены *27* рисунков и *1* таблица. При написании работы использовалось *15* источников.

Ключевые слова: компьютерная графика, параллельные вычисления, рендеринг, трассировка лучей, трассировка пути, GPGPU, OpenCL.

Работа выполнена на кафедре ВС СибГУТИ
(название предприятия, подразделения)

Основные результаты*****

Целью бакалаврской работы являлась реализация параллельного алгоритма построения фотореалистичного трёхмерного изображения, а также исследование возможных усовершенствований данной реализации для повышения эффективности вычислений и качества финального изображения.

В рамках дипломного проекта был реализован и адаптирован для параллельных вычислений на графическом процессоре алгоритм трассировки пути. Проведено сравнение времени выполнения рендеринга на различных устройствах, поддерживающих стандарт OpenCL. Для редукции шума в изображении использовалась техника суперсэмплинга.

По затраченному времени на рендеринг изображения можно судить о том, что графический процессор является более подходящим устройством для выполнения задачи просчёта пути луча, в сравнении с центральным процессором.

***В данном разделе должны быть отражены основные результаты исследования**

Содержание

1 ВВЕДЕНИЕ 5
2 ОПИСАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ 6
2.1 Трассировка лучей 6
2.2 Глобальное освещение 7
2.3 Трассировка пути 9
2.4 Архитектура современных графических процессоров 13
2.5 Технология OpenCL 17
3 РЕАЛИЗАЦИЯ АЛГОРИТМА 25
3.1 Сторона хоста 25
3.2 Сторона OpenCL-устройства 27
4 РЕЗУЛЬТАТЫ 29
5 ЗАКЛЮЧЕНИЕ 31
ПРИЛОЖЕНИЕ А 33
ПРИЛОЖЕНИЕ Б 34
ПРИЛОЖЕНИЕ В 35

1 ВВЕДЕНИЕ

Основными показателями эффективности при визуализации трёхмерного изображения – рендеринга, в компьютерной графике являются затраченное время на создание изображения и его реалистичность. В действительности, получить достоверное, с точки зрения физики, изображение за допустимое время удаётся не всегда. В компьютерных играх, например, приходится жертвовать фотореалистичностью изображения, чтобы сформировать простейший кадр за несколько миллисекунд. Если же мы говорим о компьютерной графике в кинофильмах, где корректное представление теней, глубина резкости, каустика и другие оптические эффекты являются основными критериями при построении изображения, то процесс визуализации отснятых материалов может занимать гораздо больший период времени.

С развитием графических процессоров появилась возможность более эффективного использования алгоритмов построения трёхмерного изображения. Это стало возможным благодаря GPGPU-ориентированным инструментам программирования, таким как CUDA и OpenCL.

В настоящее время актуальной темой для исследований является проблема представления реалистичного поведения света в компьютерной графике. Большинство современных интерактивных приложений не воспроизводят реальное поведение света, а лишь имитируют его, используя другие подходы при визуализации изображений.

Целью данной работы является реализация алгоритма, решающего задачу глобального освещения, оптимизированного для параллельных вычислений на графическом ускорителе.

В главе 2 представлены теоретические сведения, необходимые для понимания физической корректности данного метода визуализации изображения, сопровождаемые математическим обоснованием уравнения рендеринга, используемого для решения задачи глобального освещения. Описан алгоритм трассировки пути. Дано подробное описание архитектуры графического процессора, на котором производились вычисления, а также технологии OpenCL.

Реализация алгоритма представлена в главе 3. Описан процесс запуска алгоритма, а также структура ядра – программы, выполняемой на графическом процессоре.

Глава 4 содержит результаты времени выполнения алгоритма на различных устройствах. Представлены итоговые изображения, полученные в результате рендеринга.

В главе 5 подведены итоги проделанной БР. Рассмотрены дальнейшие возможные усовершенствования реализованного алгоритма.

2 ОПИСАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ

В данной главе рассмотрены базовые понятия фотореалистичного рендеринга, описан алгоритм трассировки пути, даны краткие сведения о технике сэмплирования пикселя изображения. В конце главы дано описание архитектуры графического процессора, на котором выполнялись вычисления, а также краткий обзор технологии OpenCL.

2.1 Трассировка лучей

Трассировка лучей – один из методов построения изображения, основанный на оптике. Оптическое изображение – это картина, получаемая при прохождении световых лучей через оптическую систему (человеческий глаз, объектив фотоаппарата и т.д.). Таким образом, можно сделать вывод, что восприятие трёхмерного изображения невозможно без наличия источников освещения.

В трёхмерном пространстве из источника света испускается огромное количество лучей (в идеале их количество совпадает с количеством излучаемых фотонов). Каждый из этих лучей имеет несколько путей развития в пространстве. В первую очередь, луч может сразу из источника освещения попасть в оптическую систему (приёмник). Другой вариант – луч может, не пересекаясь ни с одним из объектов в сцене, уйти в пространство. Также возможен ещё один случай – луч может пересечься с одним из объектов, образовав новые лучи (отражённый и преломлённые), для которых будут возможны те же пути развития, а может не образовывать новые лучи и поглотиться объектом. Стоит заметить, что влияние на итоговое изображение оказывают лишь те лучи, которые достигли оптической системы. Поэтому целесообразно отслеживать историю только этих лучей. Учитывая тот факт, что время в оптике изотропно, мы можем изменить направление отслеживания лучей от источника к приёмнику на обратное (от приёмника к источнику), не потеряв при этом достоверности изображения.

Рассмотренный выше метод проиллюстрирован на рисунке 2.1. Алгоритм делится на несколько стадий, которые применяются отдельно для каждого пикселя изображения. Сначала генерируется зрительный луч (view ray), который начинает движение от точки зрения (point of view, pov) к текущему пикселю изображения. Пройдя сквозь пиксель, данный луч устремляется к сцене с объектами. История луча учитывается только в том случае, если он достиг источника освещения. Таким образом, сцена проектируется на плоскость изображения, а цвет пикселя определяется свойствами источника освещения и объектов, которые пересёк данный луч (если луч пересекал их).

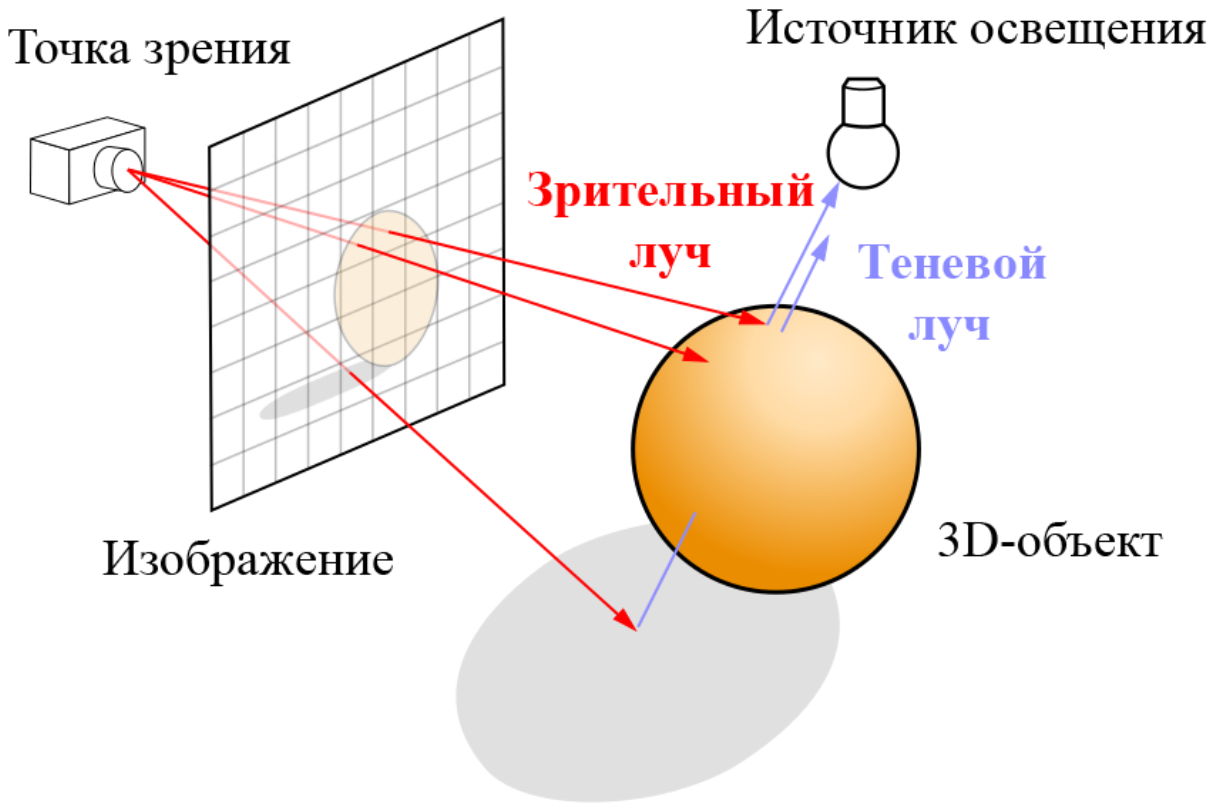

Рисунок 2.1 – Трассировка лучей.

Теневой луч образуется в результате прохождения луча через объект в направление к источнику освещения. В этом случае, в изображении будет наблюдаться идеально чёрная тень от объекта, что невозможно в реальном мире, ввиду особенностей источников освещения. Более того, в данном методе не учитывается свет, отражённый от других поверхностей, что также влияет на результирующее изображение.

Трассировку лучей относят к алгоритмам прямого освещения (direct illumination). Таким образом, невозможно воссоздать физически точное изображение, используя данную технику рендеринга.

2.2 Глобальное освещение

Глобальное освещение (global illumination) – это явление, при котором достигаются эффекты, возникающие в результате многократного отражения лучей света от поверхностей. К таким эффектам можно отнести диффузное отражение, каустику, тень от объекта.

Изображения, полученные в результате применения алгоритмов, решающих задачу глобального освещения, зачастую выглядят более реалистичными, нежели те, в процессе рендеринга которых применялись алгоритмы прямого освещения. Пример прямого и глобального освещения продемонстрирован на рисунке 2.2.

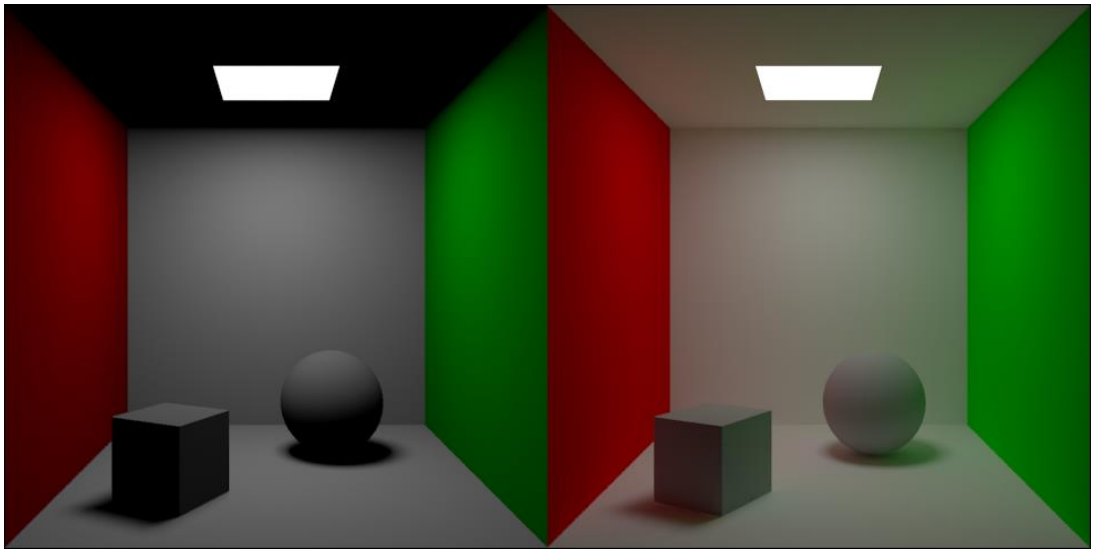

Рисунок 2.2 – Изображение, обработанное алгоритмом прямого освещения (слева) и глобального освещения (справа).

В 1986 году James Kajiya представил уравнение рендеринга [6], которое определяет количество светового излучения в определённом направлении как сумму собственного и отражённого излучения.

В математическом представлении уравнение рендеринга выглядит следующим образом:

$$L_o(x,\vec{w}) = L_e(x,\vec{w}) + \int_\Omega f_r(x,\vec{w}',\vec{w}) L_i(x,\vec{w}')(\vec{w}'\cdot\vec{n}) d\vec{w}' \qquad (2.1)$$

где $L_o(x,\vec{w})$ – свет, излучаемый поверхностью в точке x в направлении вектора w;

$L_e(x,\vec{w})$ – излучательные способности поверхности в точке x в направлении вектора w;

$\int_\Omega f_r(x,\vec{w}',\vec{w}) L_i(x,\vec{w}')(\vec{w}'\cdot\vec{n}) d\vec{w}'$ – бесконечная сумма излучений, пришедших в точку поверхности из пространственной полусферы и отражённых поверхностью в соответствии с её отражательными способностями, описываемыми с помощью функции $f_r(x,\vec{w}',\vec{w})$.

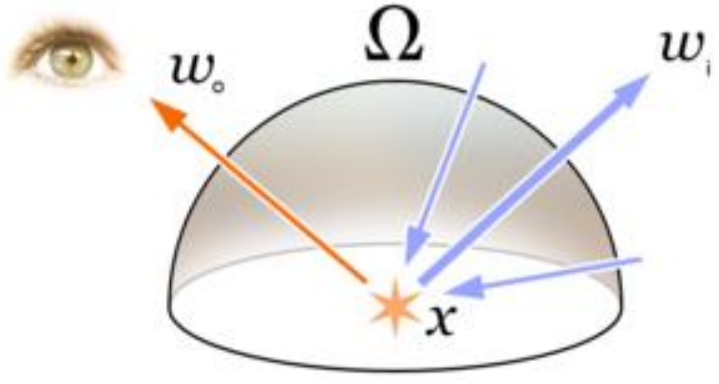

Рисунок 2.3 – Модель света.

На практике данное уравнение не может быть решено точно за конечное время. Однако алгоритмы, решающие задачу глобального освещения, позволяют получить достаточно близкий к теории результат за разумный период времени. Одним из таких алгоритмов является трассировка пути.

2.3 Трассировка пути

Трассировка пути (PT) является физически корректным подходом к визуализации изображения. Под физической корректностью подразумевается соответствие физических свойств объектов, визуализированных с помощью данного метода, таковым у реальных объектов. Данный метод наследует основные принципы метода трассировки лучей, однако имеет важное отличие – история луча, отразившегося от поверхности объекта, будет прослеживаться до тех пор, пока он не достигнет источника освещения. Стоит заметить, что количество отражений может быть неприемлемо велико, поэтому вводят специальное ограничение – максимальную глубину пути (PL). Иными словами, значение PL указывает на максимально возможное количество отражений от поверхностей объектов. Характер поведения луча, попавшего на поверхность объекта, зависит от материала соответствующего объекта.

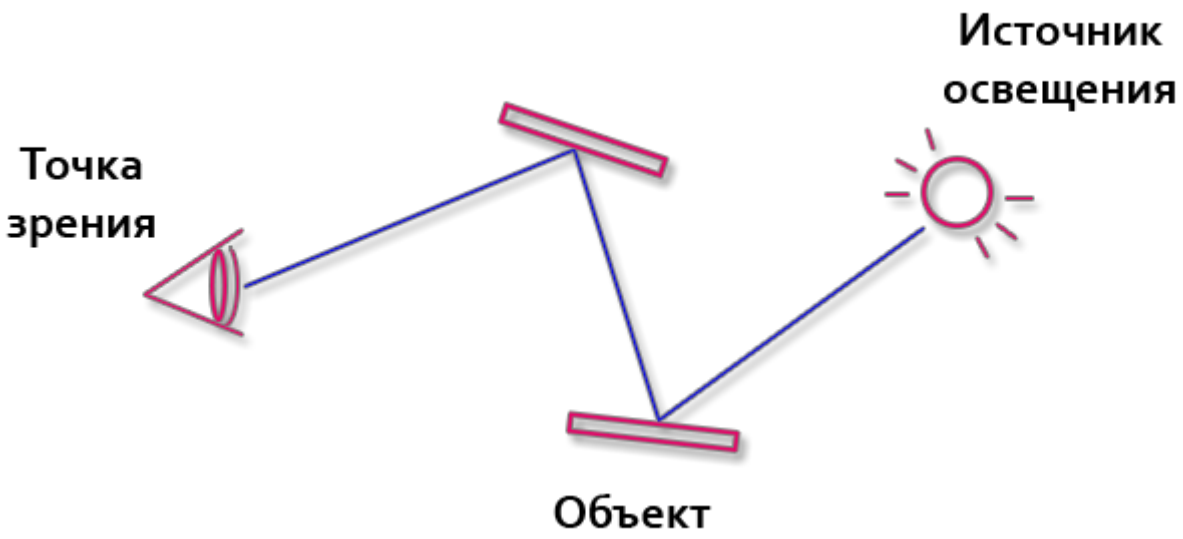

Рисунок 2.4 – Трассировка пути светового луча происходит от точки зрения к источнику освещения.

2.3.1 Световой луч, материал, объект, камера.

Световой луч – это линия, вдоль которой переносится световая энергия. В общем случае луч имеет начало (origin) и направление (direction). Чтобы получить луч по параметру t необходимо сложить начало луча с произведением направления на заданный параметр t, как показано в листинге 2.1.

Листинг 2.1 – Структура луча.

```
typedef struct Ray {
     float4 origin;
     float4 direction;
}Ray;

float4 get_ray(Ray r, float t) {
     return r.origin + t * r.direction;
}
```

Материал – это то, с чем взаимодействует световой луч. Именно от материала зависит дальнейшее поведение луча при пересечении с объектом. В рамках данной работы материал представлен в виде следующих параметров:

- *Цвет* (color) – указывает на цвет материала, принимает значения от 0 до 1.
- *Отражательная способность* (reflectivity) – указывает на возможность материала отражать от себя лучи, принимает значения от 0 до 1.
- *Преломляющая способность* (refractivity) – указывает на возможность материала преломлять лучи, принимает значения от 0 до 1.
- *Излучательная способность* (emissivity) – применяется для указания источника освещения, принимает значения от 0 до 1.
- *Показатель преломления* (index of refraction, ior) – значение зависит непосредственно от материала.

Листинг 2.2 – Структура материала.

```
typedef struct Material {
     float4 color;
     float  reflectivity;
     float  refractivity;
     float4 emissivity;
     float  ior;
}Material;
```

В качестве объекта в данной БР использовалась сфера. Структура сферы отображена в листинге 2.3.

Листинг 2.3 – Структура сферы.

```
typedef struct Sphere {
     float4    center;
     float     radius;
     Material  material;
}Sphere;
```

Чтобы найти пересечение со сферой, необходимо воспользоваться формулой

$$t^2 B \cdot B + 2t((A-C)\cdot(A-C)) + C \cdot C - R^2 = 0 \qquad (2.2)$$

где A – начало луча;

B – направление луча;

C – вектор из центра сферы к точке пересечения;

R – радиус сферы.

Данная формула, по сути, является квадратным уравнением. Вычислив дискриминант, находим корни уравнения (величину t). Далее выбирается наименьший из них и, если он попадает в область видимости камеры, то он заносится в структуру Hit_record, которая представлена в листинге 2.4.

Листинг 2.4 – Структура для хранения информации о пересечении луча с поверхностью объекта.

```
typedef struct Hit_record {
     float4 point;
     float4 normal;
     float depth;
     Material material;
}Hit_record;
```

Таким образом, данная структура хранит значение ближайшей точки пересечения с объектом по отношению к камере. Структура виртуальной камеры представлена в листинге 2.5.

Листинг 2.5 – Структура виртуальной камеры и её инициализация.

```
typedef struct Camera {
     float4 lowleftcorner;
     float4 horizontal;
     float4 vertical;
     float4 origin;
}Camera;
```

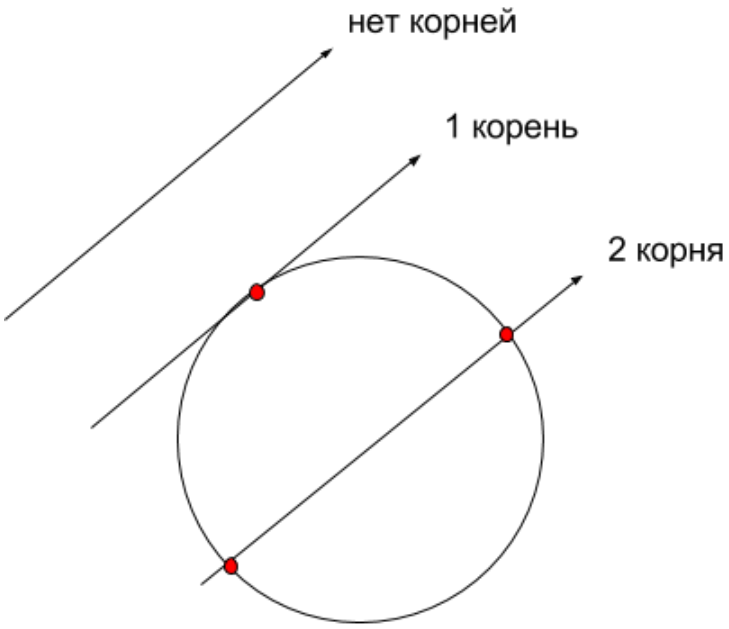

Рисунок 2.5 – Варианты пересечения луча со сферой.

2.3.2 Взаимодействие света с поверхностью

При попадании света на поверхность объекта, его дальнейшее поведение зависит от свойств, которыми обладает соответствующий объект. На рисунке 2.6 представлены возможные варианты дальнейшего движения луча, попавшего на определённую поверхность.

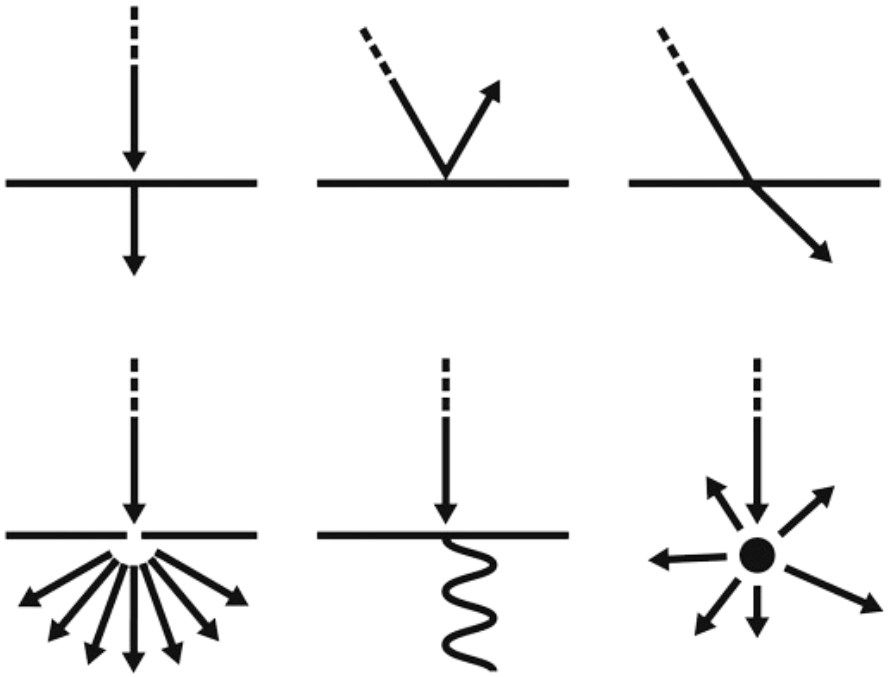

Рисунок 2.6 – 1) прохождение; 2) отражение; 3) преломление; 4) отклонение; 5) поглощение; 6) рассеивание.

В данной дипломной работе рассмотрены следующие основные примеры взаимодействий луча с поверхностью объектов:

- *Зеркальное отражение* (specular reflection). Исходной точкой является точка пересечения зрительного луча с объектом, направление высчитывается по формуле

$$\overrightarrow{d_r} = \overrightarrow{d_i} - 2 \cdot (\vec{n} \cdot \overrightarrow{d_i})\vec{n} \tag{2.3}$$

  где $\overrightarrow{d_i}$ – направление падающего на поверхность луча;
  $\overrightarrow{d_r}$ – направление отражённого луча;
  $\vec{n}$ – нормаль поверхности.

- *Внутреннее отражение/преломление* (specular refraction). Исходной точкой является точка пересечения зрительного луча с объектом, направление высчитывается по закону Снелла – угол падения света на поверхность связан с углом преломления соотношением

$$n_1 \cdot \sin\theta_1 = n_2 \sin\theta_2 \tag{2.4}$$

  где n_1 – показатель преломления среды, из которой свет падает на границу раздела;

θ_1 – угол между падающим на поверхность лучом света и нормалью к поверхности;

n_2 – показатель преломления среды, в которую попадает свет, пройденный границу раздела.

θ_2 – угол между прошедшим через поверхность световым лучом и нормалью к поверхности.

- *Диффузное отражение* (diffuse reflection). Исходной точкой нового луча является точка пересечения зрительного луча с объектом, направлением является нормализованный вектор, направленный вверх от центра полусферы.

2.3.3 Сэмплирование

Под сэмплированием подразумевается разбиение пикселя на отдельные фрагменты для последующего смешивания их цветов. Данная техника применяется для достижения эффекта сглаживания изображения. Пример сглаживания представлен на рисунке 2.5.

Сэмплирование характеризуется параметром SPP:

$$SPP = \frac{nrays}{pixel} \tag{2.5}$$

где nrays – количество лучей, пропущенных через один пиксель.

Чем больше параметр SPP, тем выше качество изображения, и тем дольше время выполнения алгоритма. Обычно, достаточно фотореалистичное изображение получается при 1000 < SPP < 10000. Разбиение пикселя на такое большое количество сэмплов называется избыточной выборкой сглаживания или суперсэмплингом.

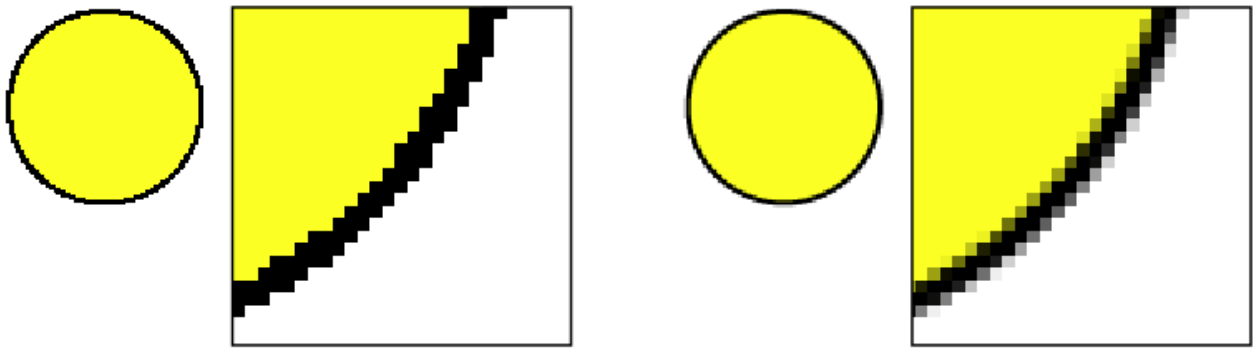

Рисунок 2.7 – Изображение слева не сглажено, к изображению справа применено сглаживание 4x.

2.4 Архитектура современных графических процессоров

Графический процессор – специализированное устройство ЭВМ, предназначенное для обработки графической информации. Основной отличительной особенностью графического процессора от центрального является наличие большого количества ядер в своём составе. Данное отличие отображено на рисунке 2.8.

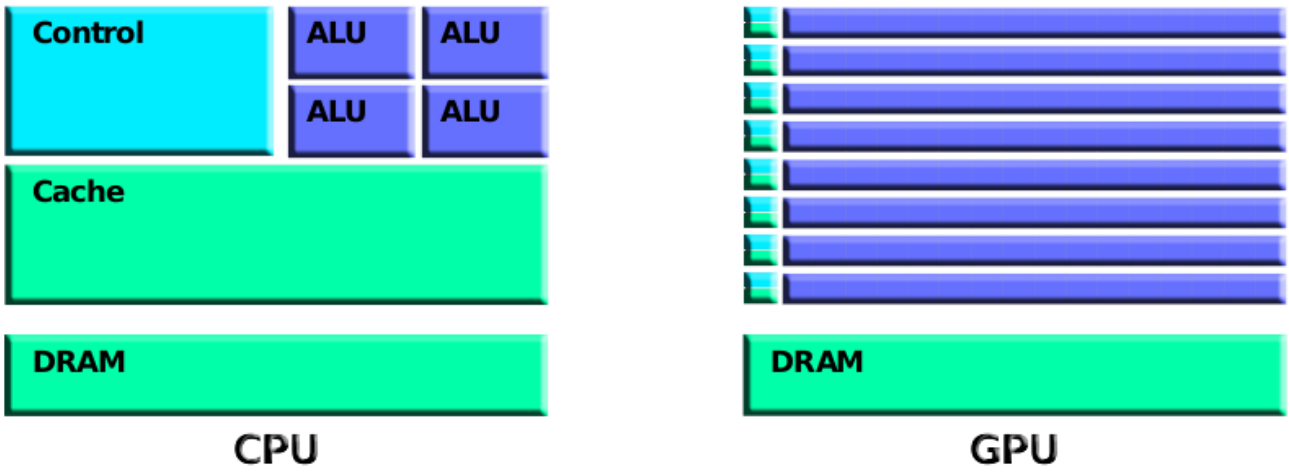

Рисунок 2.8 – Обобщённые архитектуры центрального (CPU) и графического (GPU) процессоров.

Разница в архитектуре обусловливает и разницу в принципах работы данных устройств. CPU чаще применяется при обработке последовательной информации (современные центральные процессоры, однако, способны обрабатывать параллельно несколько задач), тогда как GPU оптимизирован для массовой параллельной обработки данных. Таким образом, в большинстве случаев графический процессор не является заменой центральному, а лишь выполняет роль вычислительного блока и работает одновременно с ним.

В качестве примера рассмотрим видеокарту Radeon R9 290x. В её основе лежит графический чип Hawaii XT, имеющий порядка шести миллиардов транзисторов. Процессор содержит 2816 потоковых ядер и 176 текстурных блоков. Тактовая частота Radeon R9 290x равна 800 МГц, с возможностью поднятия её в турбо-режиме до 1000 МГц.

В основе GPU лежат 4 больших кластера SE, называемых шейдерным движком. Каждый SE содержит 11 вычислительных блоков GCN и 4 укрупнённых растровых процессора.

Также, каждый из четырёх SE имеет по отдельному геометрическому процессору, что даёт четыре геометрических блока на GPU. При этом геометрические процессоры не являются жёстко привязанными к конкретному шейдерному движку и при необходимости могут направить свой поток данных на любой другой кластер SE.

Рисунок 2.9 – Архитектура Hawaii XT.

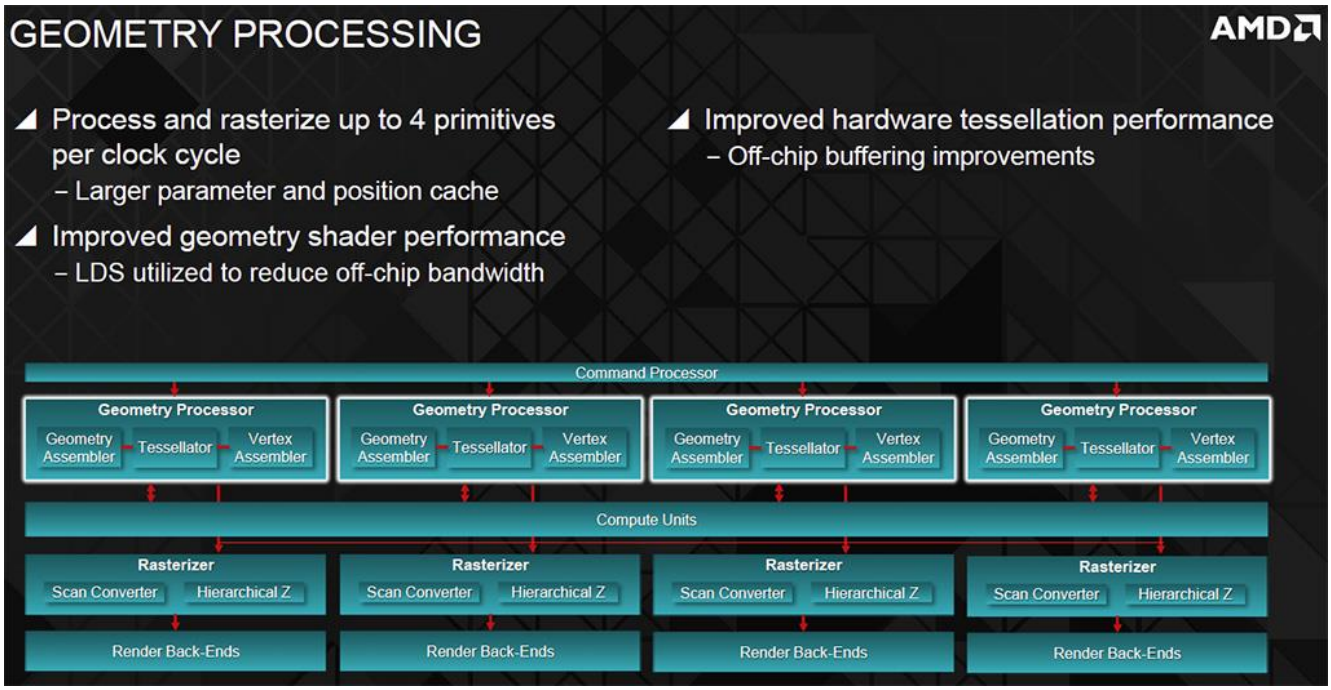

Рисунок 2.10 – Обработка геометрии в шейдерном движке.

Кроме того, для более эффективной работы геометрического блока были увеличены размеры и пропускная способность кэшей, а также оптимизирована загрузка/выгрузка данных. Основная цель данных оптимизаций – наличие данных, необходимых для вычислений в геометрических блоках, в исполнительных блоках CU или кэшах GPU.

Также Radeon R9 290x обладает большими возможностями в области общих вычислений на GPU, так называемых GPGPU – таких как OpenCL, DirectCompute, FireStream и т.д. R9 290x имеет в наличии девять командных процессоров – один для выполнения графических задач и восемь для неграфических. Командные процессоры ACE служат для создания и распределения вычислительных потоков, исполняемых на блоках GCN. Каждый подобный ACE-процессор доступен как отдельное устройство, кроме того ACE работают независимо от графического командного процессора.

Любой ACE-процессор способен одновременно управлять восемью очередями команд. Более того, каждый из ACE имеет доступ ко всем модулям GCN, что предоставляет возможности эффективно распределять нагрузку и задействовать неиспользуемые ресурсы.

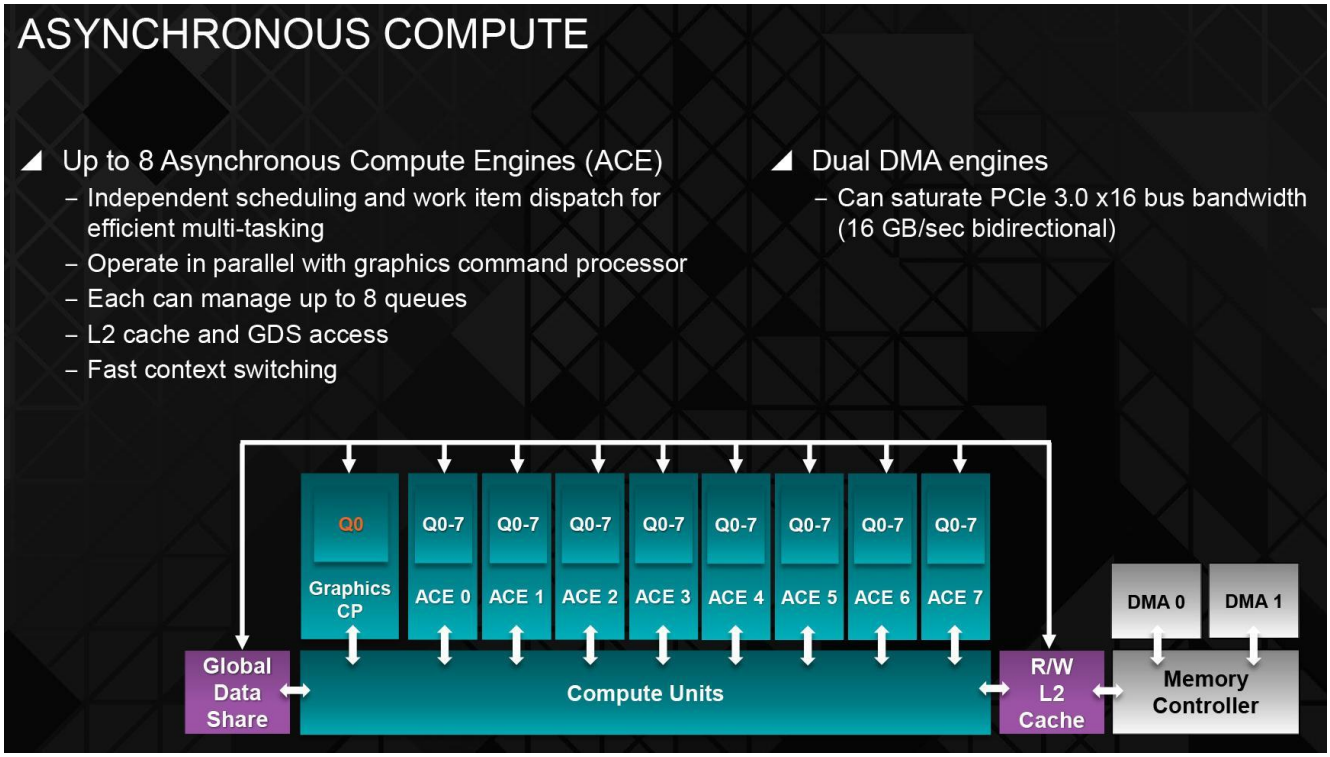

Рисунок 2.11 – Асинхронные командные процессоры.

Каждый модуль GCN содержит четыре векторных блока, планировщик, четыре текстурных блока и блок скалярных операций. Каждый из четырёх векторных блоков имеет по 16 ALU (64 ALU на один модуль GCN). Таким образом, 44 блока GCN в сумме дают 2816 потоковых ядер.

Планировщик распределяет команды по векторным блокам, причём каждый векторный блок является независимым и способен исполнять свой поток команд, никак не связанный с остальными векторными блоками.

Radeon R9 290x поддерживает до 4 ГБ памяти. Тип памяти – GDDR5. Сам контроллер памяти состоит из восьми 64-х битных контроллеров, каждый из которых способен одновременно работать с двумя микросхемами памяти.

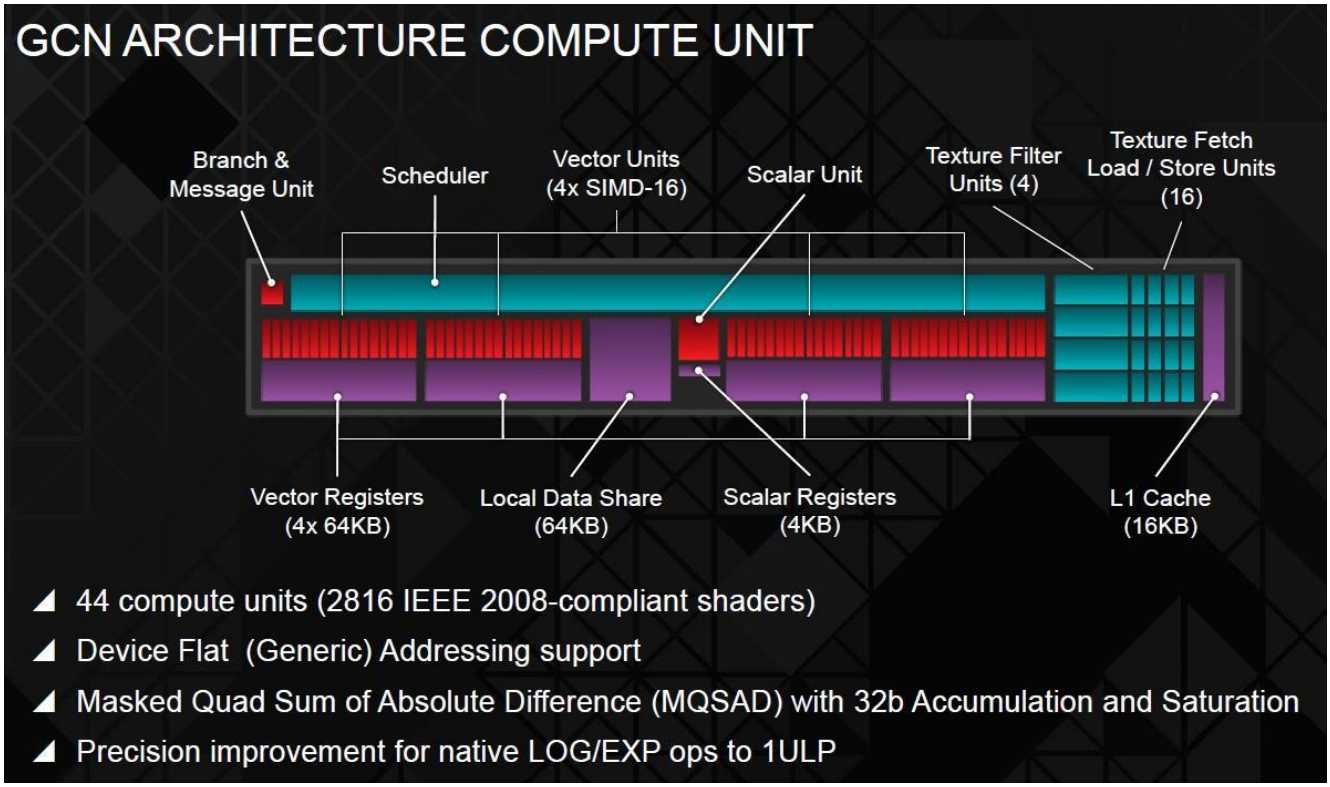

Рисунок 2.12 – Архитектура вычислительного блока GCN.

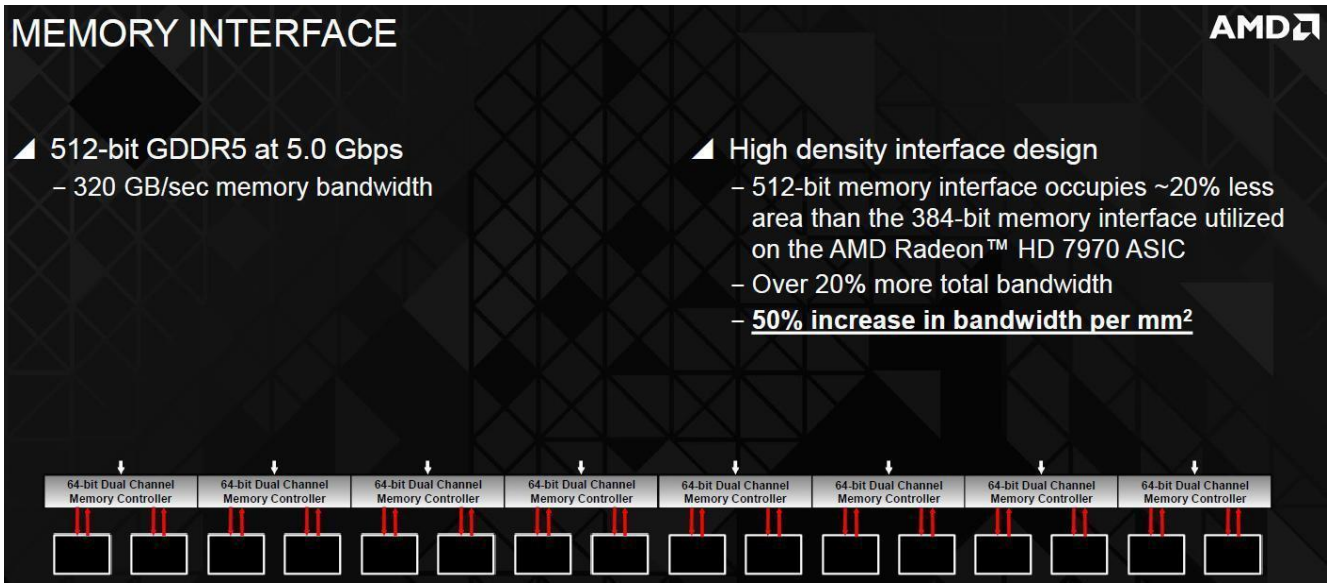

Рисунок 2.13 – Интерфейс памяти.

2.5 Технология OpenCL

Наличие большого количества ядер в составе графических процессоров породило концепцию GPGPU, в основе которой лежит перенос неграфических вычислений на графический процессор. Немаловажным является и тот факт, что ядра графического процессора потребляют небольшое количество энергии в сравнении с ядрами центрального процессора. В связи с этим GPU заметно выигрывает в производительности на ватт потребляемой мощности [11].

В компьютерной индустрии существует несколько крупных продуктов, которые позволяют производить общие вычисления на графических процессорах:

- *CUDA* – аппаратно-зависимая платформа, предназначенная для организации параллельных вычислений на графических процессорах фирмы Nvidia.
- *OpenCL* – универсальный фреймворк, позволяющий организовывать параллельные вычисления на различных графических и центральных процессорах.
- *DirectCompute и C++ AMP*.
- *OpenACC*.

Проанализируем данную концепцию на примере технологии OpenCL. OpenCL – это стандарт, разработанный компанией Khronos Group совместно с разработчиками ПО, вендорами аппаратных решений и производителями процессоров различных типов и назначений. Основная идея данной технологии заключается в предоставлении программисту универсального инструмента для использования всех вычислительных мощностей современных ВС. По задумке авторов, написав программу с использованием OpenCL, можно будет запускать её практически на любой ВС: телефонах, графических картах, ускорителях и т.п. [9] Рассмотрим четыре основные модели, на которых держится стандарт OpenCL.

2.5.1 Модель платформы

Модель платформы (platform model) даёт высокоуровневое описание гетерогенной системы. Центральным элементом данной модели выступает понятие хоста (host) – первичного устройства, которое управляет OpenCL-вычислениями и осуществляет все взаимодействия с пользователем. Хост всегда представлен в единственном экземпляре, в то время как OpenCL-устройства (devices), на которых выполняются OpenCL-инструкции, могут быть представлены во множественном числе. OpenCL-устройством может быть CPU, GPU, DSP или любой другой процессор в системе, поддерживающийся установленными в системе OpenCL-драйверами.

OpenCL-устройства логически делятся моделью на вычислительные модули (compute units), которые в свою очередь делятся на обрабатывающие элементы (processing elements). Вычисления на OpenCL-устройствах в действительности происходят на обрабатывающих элементах. На рисунке 2.14 схематически изображена OpenCL-платформа из 3-х устройств.

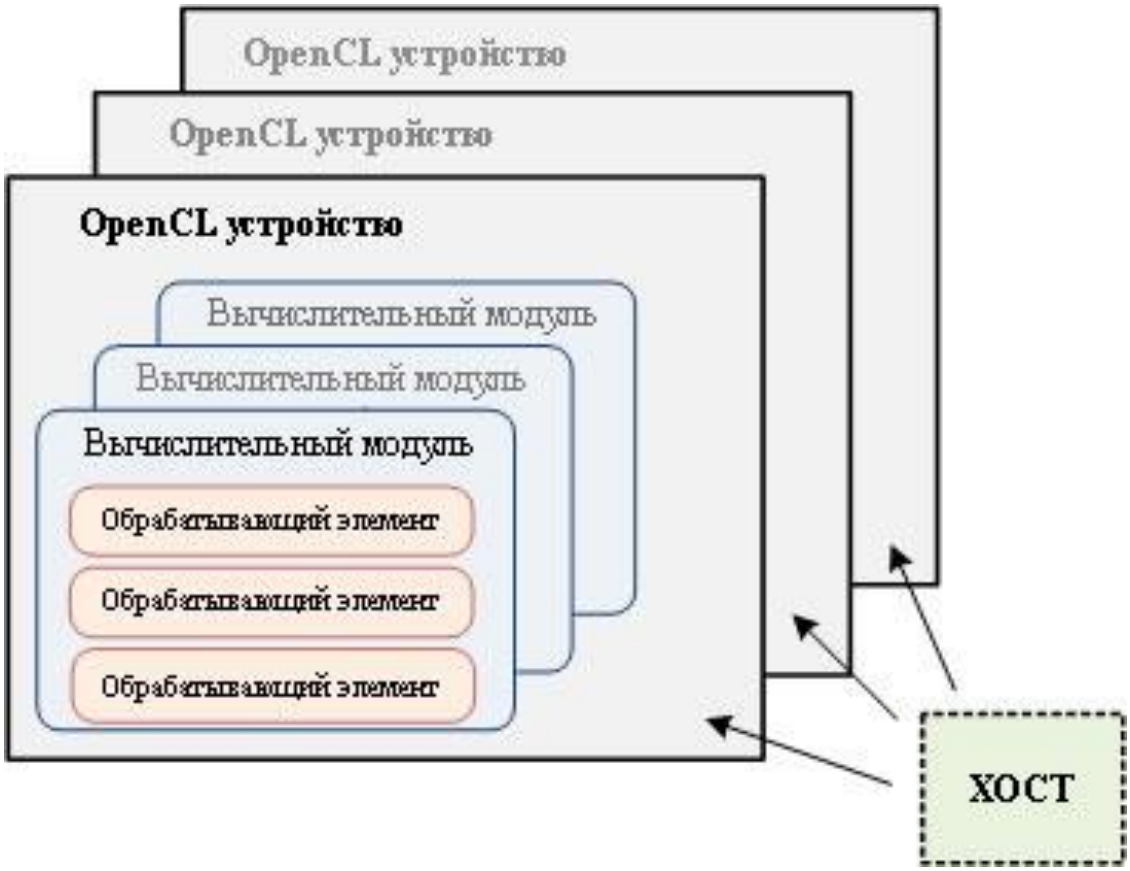

Рисунок 2.14 – Схематичное представление OpenCL-платформы.

2.5.2 Модель вычислений

Модель вычислений (execution model) описывает абстрактное представление того, как потоки инструкций выполняются в гетерогенной системе. С хостом неразрывно связано понятие хостовой программы (host program) – программного кода, выполняющегося только на хосте. OpenCL не указывает как именно должна работать хостовая программа, а лишь определяет интерфейс взаимодействия с OpenCL-объектами.

С точки зрения модели вычислений OpenCL-приложение состоит из хостовой программы и набора ядер (kernels). OpenCL-ядро в общем виде представляет собой функцию, написанную на языке OpenCL C (подмножество языка ISO C'99) и скомпилированную OpenCL-компилятором.

Ядро создается в хостовой программе и затем с помощью специальной команды ставится в очередь на выполнение в одном из OpenCL-устройств. Во время выполнения упомянутой команды OpenCL Runtime System создает целочисленное пространство индексов (integer index space), каждый элемент которого носит название глобального идентификатора (global ID). Каждый экземпляр ядра выполняется отдельно для каждого значения глобального идентификатора. Экземпляр ядра называют рабочим элементом (work-item). Таким образом, каждый work-item однозначно определяется своим глобальным идентификатором.

Множество всех work-item разбивается на группы – work-group. С каждой work-group сопоставляется свой уникальный идентификатор (work-group ID). Все work-item в одной work-group идентифицируются уникальным в пределах своей

группы номером – local ID. Таким образом, каждый work-item определяется как по уникальному global ID, так и по комбинации work-group ID и local ID внутри своей группы.

Все work-item в пределах одной work-group выполняются параллельно на обрабатывающих элементах одного вычислительного модуля OpenCL-устройства. Это гарантируется стандартом, в то время как совершенно не гарантируется, что несколько work-item из разных групп будут выполнены параллельно. Об этом важном свойстве параллелизма необходимо помнить при разработке OpenCL-программ.

Пространство индексов N-размерно и обычно носит название NDRange. В случае версии стандарта OpenCL 1.1 размерность N принимает значения 1,2 или 3 (в OpenCL 2.0 максимальная размерность определяется значением константы `CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS`). Таким образом, сетки координат global ID и local ID N-размерны, т.е. определяются N координатами. На рисунке 2.15 схематически показан двумерный NDRange.

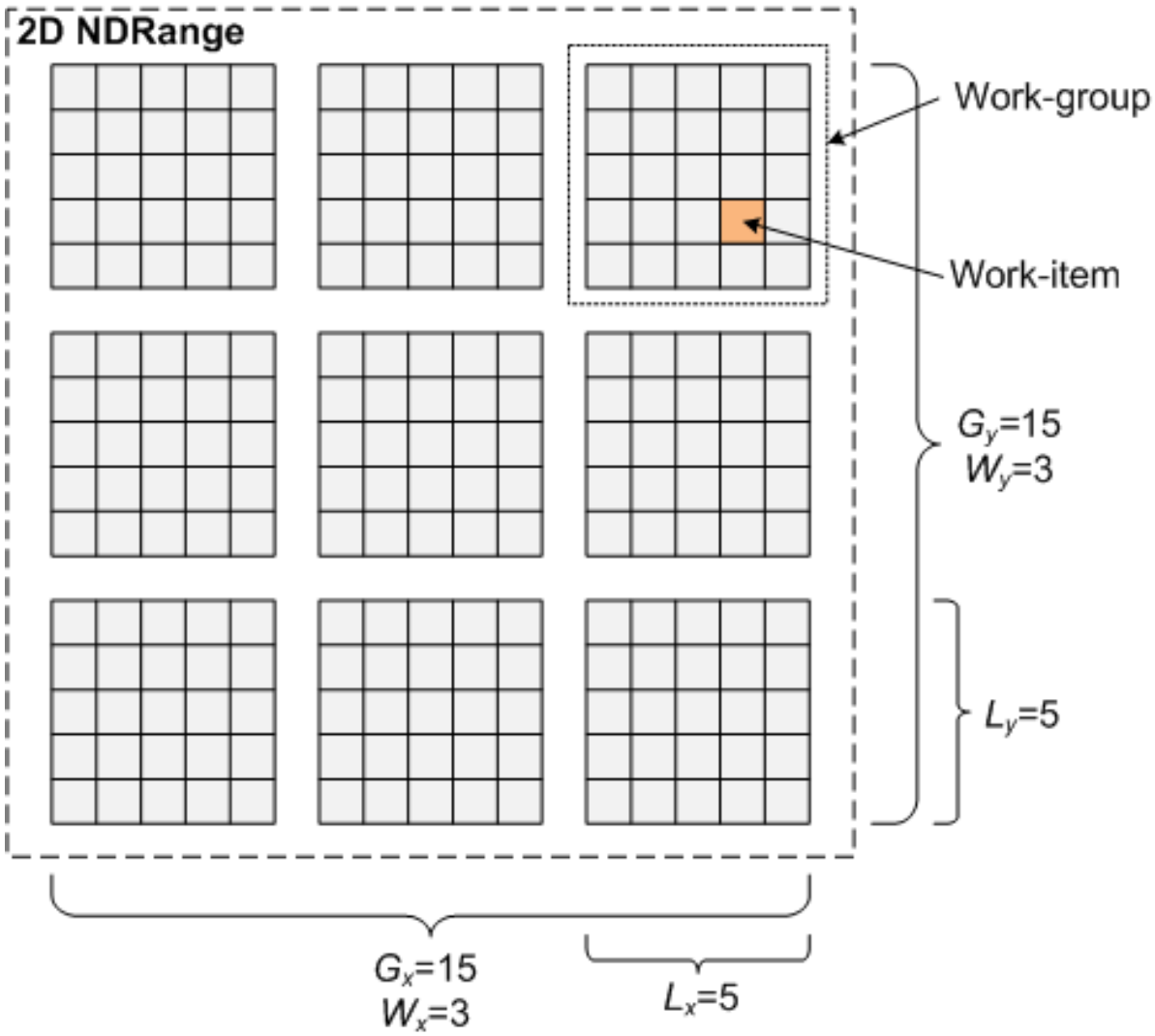

Рисунок 2.15 – Пример двумерного NDRange, где Gx и Gy - число глобальных идентификаторов, Wx и Wy - число групп, а Lx и Ly - число локальных идентифкаторов в NDRange.

Другим важным понятием модели вычислений является контекст (context), определение которого (при помощи вызова специальных функций OpenCL API) является первой задачей в OpenCL-приложениях. Контекст определяет среду выполнения ядер, в которую входят следующие компоненты: устройства, сами ядра, программные объекты (program objects, исходный и выполняемый код будущих ядер), объекты памяти (memory objects).

Взаимодействие между хостом и OpenCL-устройством происходит посредством команд, помещенных в командную очередь (command-queue). Данные команды ожидают в командной очереди своего выполнения на OpenCL-устройстве. Командная очередь создается хостом и сопоставляется одному OpenCL-устройству после того, как будет определен контекст. Команды делятся на те, что отвечают за: выполнение ядер, управление памятью и синхронизацию выполнения команд в очереди. Команды могут выполняться последовательно (in-order execution) или внеочередно (out-of-order execution). Второй вариант организации очередей поддерживается не всеми платформами, о чем необходимо помнить при разработке OpenCL-программы.

2.5.3 Модель памяти

Модель памяти (memory model) описывает набор регионов памяти и манипулирование ими во время проведения вычислений. OpenCL-объекты, инкапсулирующие регионы памяти, называются объектами памяти (memory objects). Объекты памяти бывают двух типов – буферные объекты (buffer objects) и объекты изображения (image objects).

Буферные объекты памяти инкапсулируют непрерывные участки памяти, доступные ядрам во время выполнения. Программист обычно производит отображение структур данных на данные объекты, а в коде ядра получает доступ к данным структурам посредством указателей.

Объекты изображений ограничены хранением изображений. При этом как именно изображение хранится не определяется стандартом и скрыто от программиста. Обычно хранение и доступ к изображению оптимизирован под конкретную аппаратную платформу.

Стандарт OpenCL имеет следующие пять различных регионов памяти в своём составе:

- *Память хоста* (host memory) – доступна только с хоста.
- *Глобальная память* (global memory) определяется в памяти, доступной на чтение и запись для всех work-item во всех work-group. Чтение и запись в глобальную память может кэшироваться, если OpenCL-устройство поддерживает данную возможность. В случае CPU глобальной является оперативная память. В подавляющем большинстве случаев глобальная (и константная) память самая медленная, так что использовать без необходимости ее не стоит.
- *Константная память* (constant memory) – глобальный регион памяти, который инициализируется хостом и является доступным любому из work-item для чтения данных из него.

- *Локальная память* (local memory) – доступна только в пределах одной work-group. Все work-item в данной work-group могут производить операции чтения и записи в данный регион памяти одновременно.
- *Приватная память* (private memory) – доступна одному конкретному work-item.

На рисунке 2.16 показаны отношения между перечисленными регионами памяти согласно стандарту OpenCL.

Рисунок 2.16 – Схематичное представление нескольких уровней памяти в OpenCL.

2.5.4 Модель программирования

Модель программирования (programming model) описывает варианты переноса абстрактного алгоритма на гетерогенные вычислительные ресурсы. OpenCL определяет два типа модели программирования – параллелизм по данным (data parallelism) и параллелизм по заданиям (task parallelism).

Первый тип модели программирования организован вокруг структур данных: каждый элемент структуры данных, определённый хостом, обновляется одновременно (параллельно) копиями одного и того же OpenCL-ядра. Дизайн

такой структуры должен поддерживать возможность одновременного изменения различных её частей.

Модель параллелизма по данным наиболее естественная модель программирования для OpenCL, так как NDRange создается непосредственно перед запуском ядра на устройстве. Роль программиста заключается в описании решения данной задачи в терминах упомянутой структуры данных, отобразив ее на NDRange и инкапсулировав в объекте памяти.

В случае если между несколькими work-item необходимо взаимодействие, то программист проектирует свой алгоритм с учетом разбиения множества всех work-item на некоторое количество work-group, в рамках которых несколько work-item могут осуществлять такое взаимодействие. Взаимодействие может осуществляться либо через чтение/запись локальных регионов памяти, либо через синхронизацию посредством группового барьера (work-group barrier).

Если групповой барьер определен в коде ядра, то все work-item в пределах одной work-group должны дойти до этого барьера до того, как другие work-item из этой группы продолжат своё выполнение. Ввиду ориентированности технологии OpenCL на широкий круг устройств необходимо учитывать, что OpenCL не поддерживает механизмы синхронизации между несколькими work-item из разных work-group.

Также необходимо помнить не только о том, что все work-item в пределах work-group выполняются одновременно, но и о том, что несколько work-group могут выполняться параллельно, если устройство, на котором производятся вычисления, предоставляет такую возможность. Таким образом, стандарт OpenCL описывает иерархическую модель параллелизма по данным – внутри каждой группы и между различными группами.

При проектировании алгоритма программист имеет две возможности относительно определения work-group. Первое (explicit model) – определить размер групп самостоятельно. Второе (implicit model) – предоставить возможность разбиения на группы непосредственно самой системе.

Второй тип модели программирования организован вокруг ядер (заданий), выполняемых как единственный work-item. При этом вместе с таким заданием в устройстве могут одновременно выполняться и другие work-item. Необходимость параллелизма по заданиям может возникнуть, если параллелизм уже заключен в самом задании. Например, если в ядре задания производятся векторные операции над векторным типом данных.

Также в данном типе модели программирования может возникнуть необходимость тогда, когда несколько команд запуска ядер помещаются в очередь, в которой запуск происходит сразу же после того, как команда поместилась в данную очередь. В ряде случаев это позволяет увеличить степень использования OpenCL-устройств, позволяя системе самостоятельно планировать запуск множества различных заданий. Однако, опять же стоит учитывать, что параллелизм по заданиям с внеочередным запуском может не работать на некоторых вычислительных платформах и данный вариант модели программирования является скорее опциональной возможностью OpenCL.

Ещё один вариант необходимости использования параллелизма по заданиям возникает в том случае, когда множество заданий зависимо между

собой и объединяются в граф с применением OpenCL-событий. Одни команды, помещенные в очередь, могут генерировать события, а другие команды могут ожидать этих событий, чтобы начать своё выполнение.

3 РЕАЛИЗАЦИЯ АЛГОРИТМА

Параллельный алгоритм трассировки пути, предназначенный для вычислений на графическом процессоре, можно условно разбить на 3 этапа: инициализация устройства, инициализация ядра, запуск ядра. Первые два этапа, а также подготовка к запуску ядра, осуществляются на стороне хоста. Сами расчёты (выполнение алгоритма трассировки пути) происходят на стороне устройства.

3.1 Сторона хоста

3.1.1 Инициализация устройства

Инициализация устройства производится на хосте. На данном этапе осуществляется поиск и инициализация всех устройств, подключенных к ВС. Переменная `CL_DEVICE_TYPE_XXX` указывает на тип процессора, информацию о котором следует получить. Возможные варианты переменной:

- `CL_DEVICE_TYPE_CPU` – информация о центральном процессоре;
- `CL_DEVICE_TYPE_GPU` – информация о графическом процессоре;
- `CL_DEVICE_TYPE_ALL` – информация о всех устройствах.

К этапу инициализации устройств также можно отнести процесс создания контекста OpenCL и очереди команд. Рисунок 3.1 отображает вышеуказанные действия.

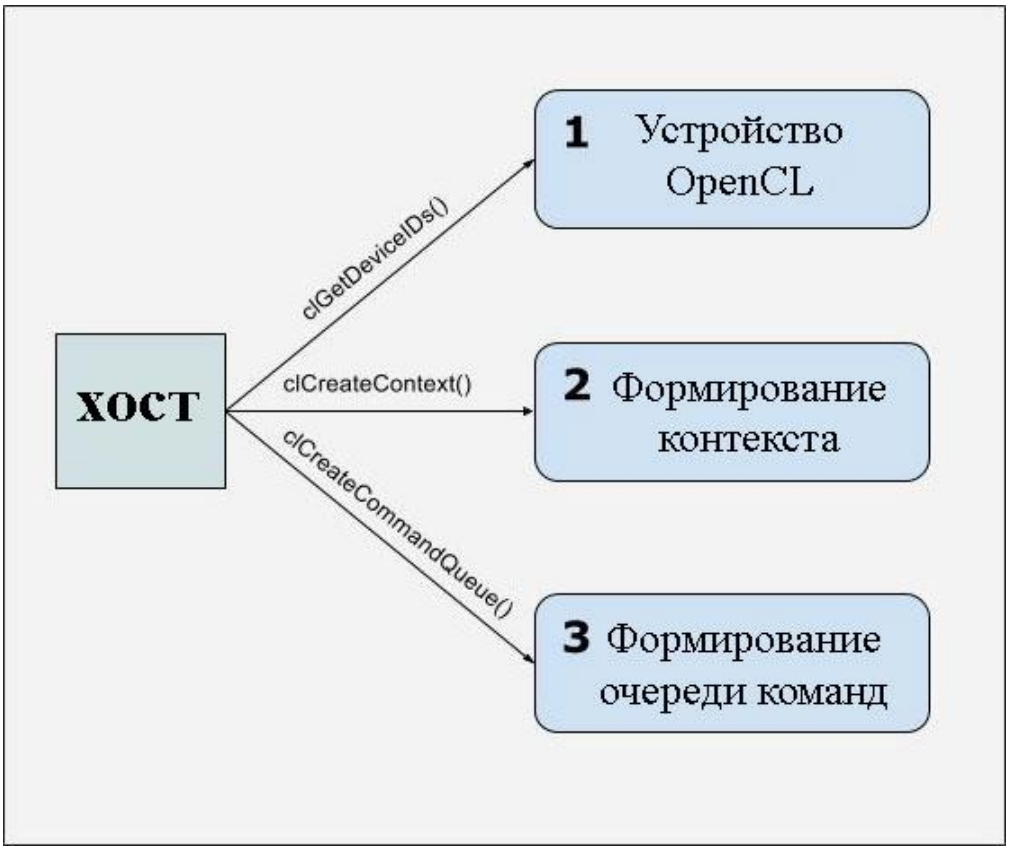

Рисунок 3.1 – Процесс инициализации устройств в системе.

3.1.2 Инициализация ядра

Инициализация ядра производится на хосте. На этом этапе происходит загрузка программы в буфер и последующий её перенос в ядро устройства, на котором будут происходить вычисления. Данные действия отображены на рисунке 3.2.

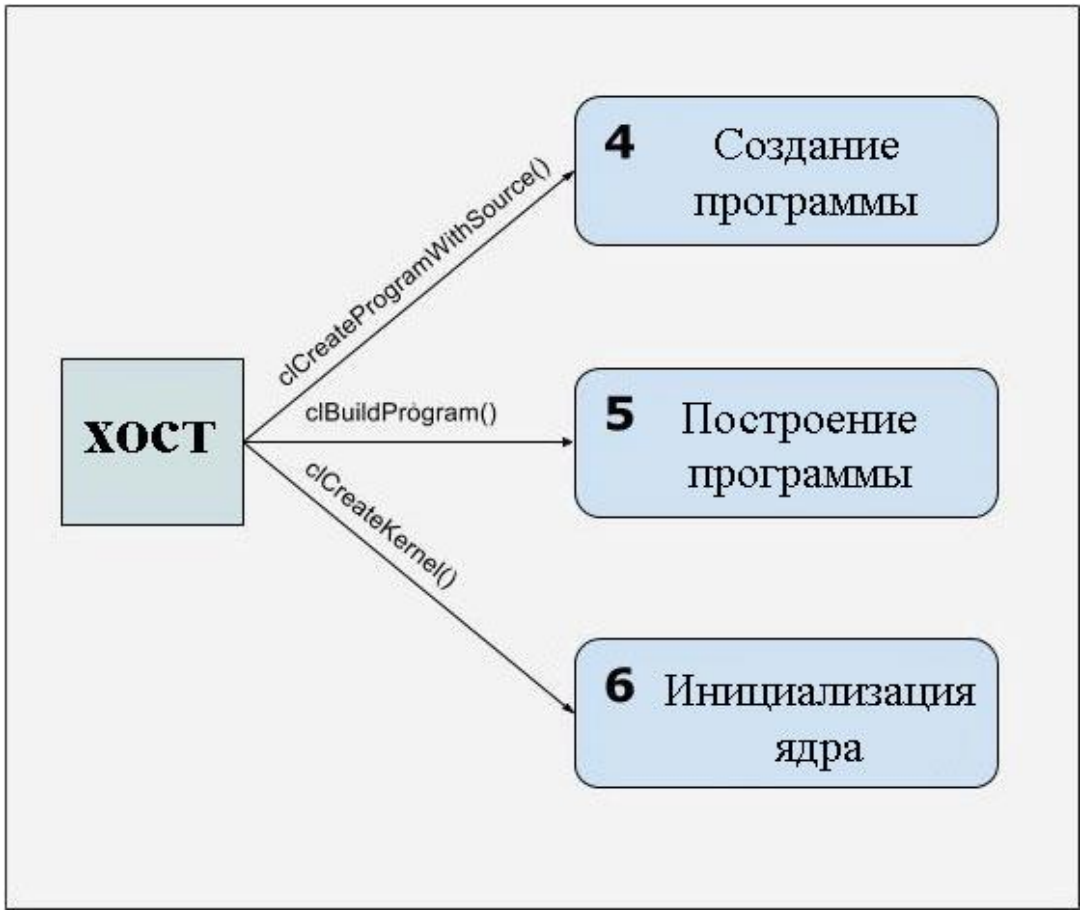

Рисунок 3.2 – Процесс инициализации ядра.

3.1.3 Запуск ядра

Подготовка к запуску ядра осуществляется также на хосте. Сначала хост устанавливает N аргументов kernel-функции, а затем с помощью инструкции clEnqueueNDRangeKernel() ставит в очередь команды (T_1 – T_K), которые затем будут выполнены на устройстве.

Инструкция clEnqueueReadBuffer() позволяет считать из очереди финальный результат (R_1 – R_K) в специально отведённый для этого буфер. Данный процесс схематично представлен на рисунке 3.3.

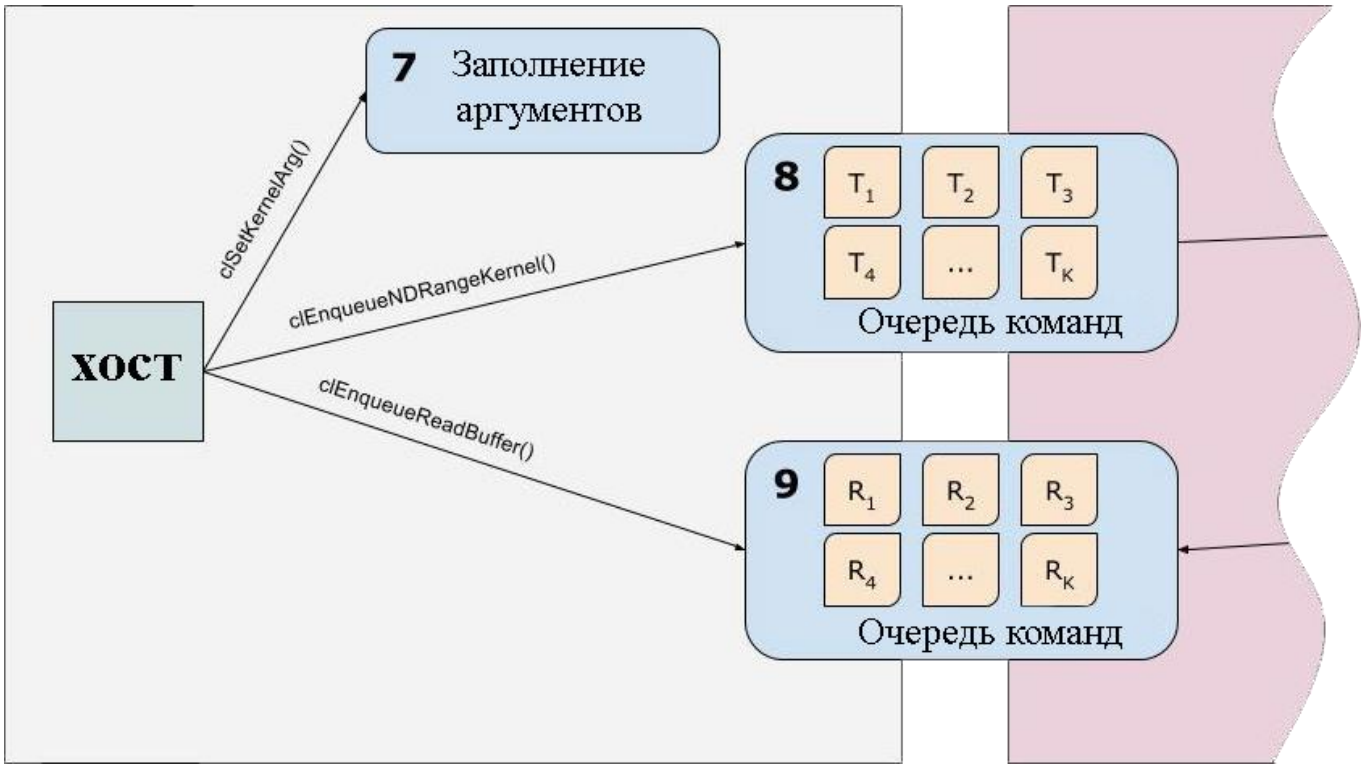

Рисунок 3.3 – Запуск ядра со стороны хоста и получение результатов.

3.2 Сторона OpenCL-устройства

В первую очередь, хост заполняет аргументы kernel-функции. Первым аргументом выступает массив A, равный размеру изображения. Этот массив использует каждый work-item, поэтому целесообразно хранить его в глобальной памяти устройства.

Следующим аргументом является массив, предназначенный для хранения начальных состояний (seed) для получения случайных чисел. Стандарт OpenCL не имеет в своём составе функций, способных генерировать случайные числа, которые являются необходимыми для данного алгоритма. Данная проблема была решена с помощью использования алгоритма PRNG [15]. Массив начальных состояний также хранится в глобальной памяти устройства.

Последним аргументом функции является сама 3D-сцена, предназначенная для рендеринга. Т.к. данная реализация алгоритма подразумевает рендеринг статических сцен, то нет необходимости генерировать каждый раз одну и ту же сцену для каждого экземпляра ядра. Более того, ядро никак не изменяет саму сцену, поэтому она так же была перенесена в глобальную память GPU.

Заполнив аргументы ядра и очередь задач, хост отправляет устройству задания на выполнение. В данной реализации задачей является запуск алгоритма трассировки пути для каждого пикселя изображения. Асинхронно извлекая задачи из очереди, каждое потоковое ядро обрабатывает только один рабочий элемент (пиксель изображения). Подробно это показано на рисунке 3.4.

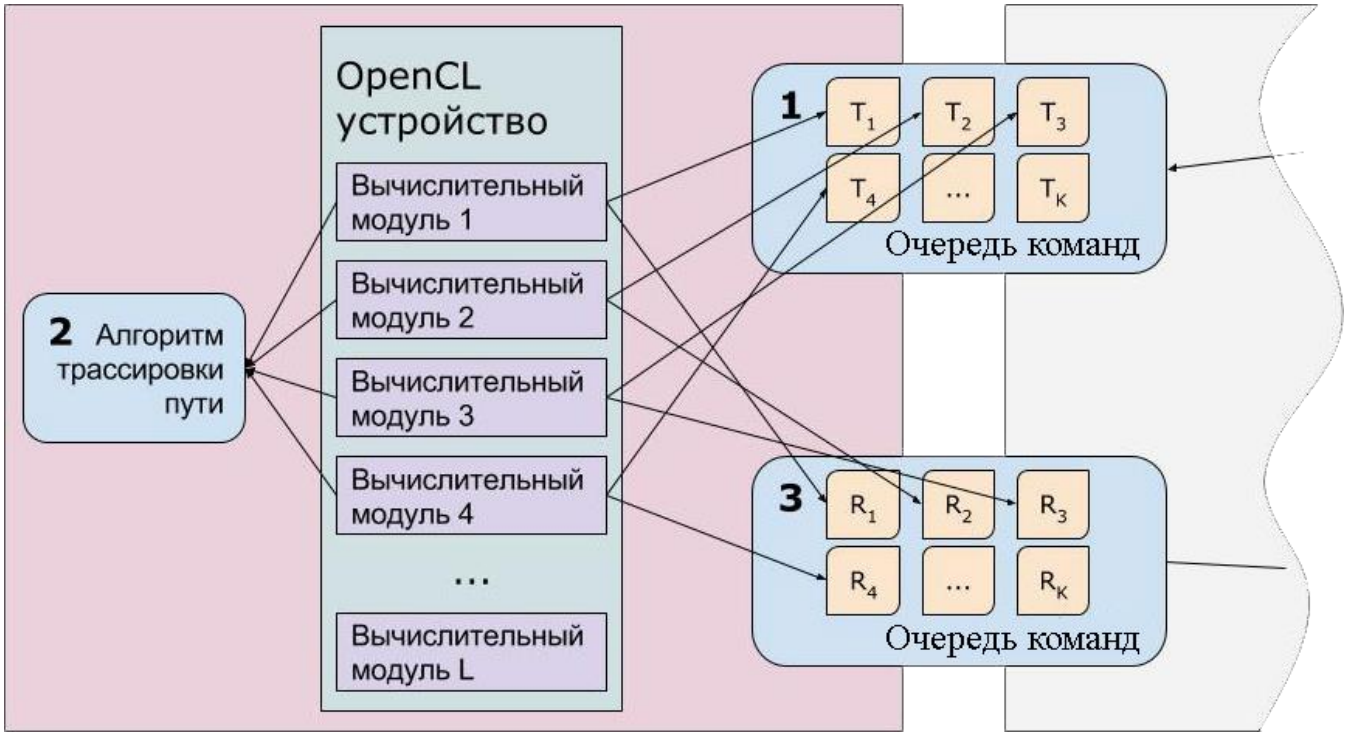

Рисунок 3.4 – Процессы, происходящие на стороне OpenCL-устройства.

Завершив алгоритм трассировки пути, итоговое значение помещается в массив A по индексу, равному глобальному ID, на котором выполнялась kernel-функция. Это даёт гарантию, что цвет, полученный в результате трассировки, будет принадлежать строго определённому пикселю. Таким образом, массив A хранит информацию о цвете каждого пикселя изображения, что отображено на рисунке 3.5.

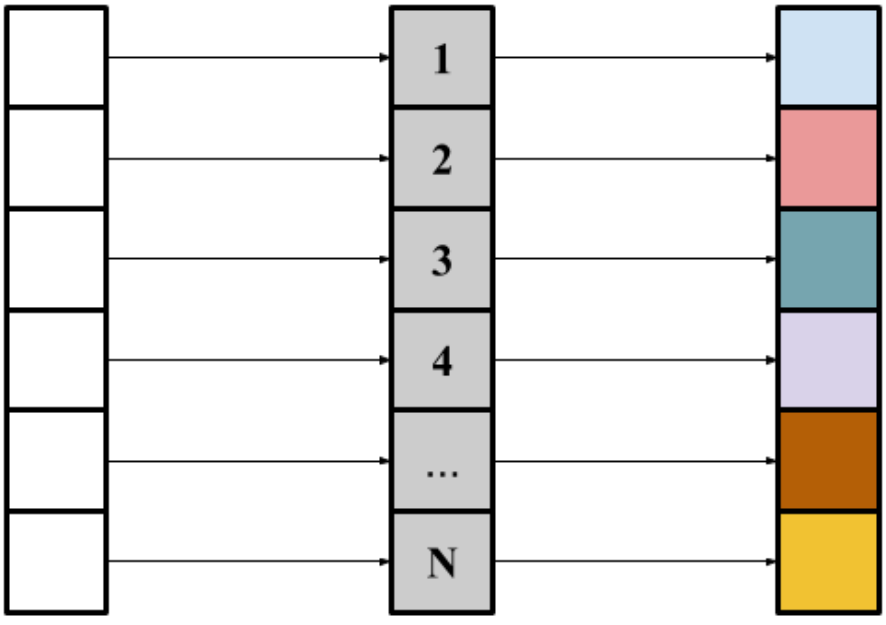

Рисунок 3.5 – Первоначальный массив (слева), массив после завершения работы программы (справа).

4 РЕЗУЛЬТАТЫ

Реализованный алгоритм способен выдавать фотореалистичное изображение за короткий период времени, если производить вычисления на графическом процессоре. Центральный процессор справляется с данной задачей значительно хуже. Затраченное время на рендеринг для различных устройств представлено в таблице 1. Полученные изображения представлены на рисунках 4.1 – 4.4.

Таблица 1 – Время выполнения алгоритма.

Производитель	**Тип**	**Устройство**	**SPP**	**Время (мс)**
AMD	GPU	Radeon r9 290x	10	2589
AMD	GPU	Radeon r9 290x	100	27655
AMD	GPU	Radeon r9 290x	1000	269815
Nvidia	GPU	Geforce GT 630	10	3492
Nvidia	GPU	Geforce GT 630	100	33215
Nvidia	GPU	Geforce GT 630	1000	312652
AMD	CPU	FX-6300 Vishera	10	10322
AMD	CPU	FX-6300 Vishera	100	130246
AMD	CPU	FX-6300 Vishera	1000	1405217

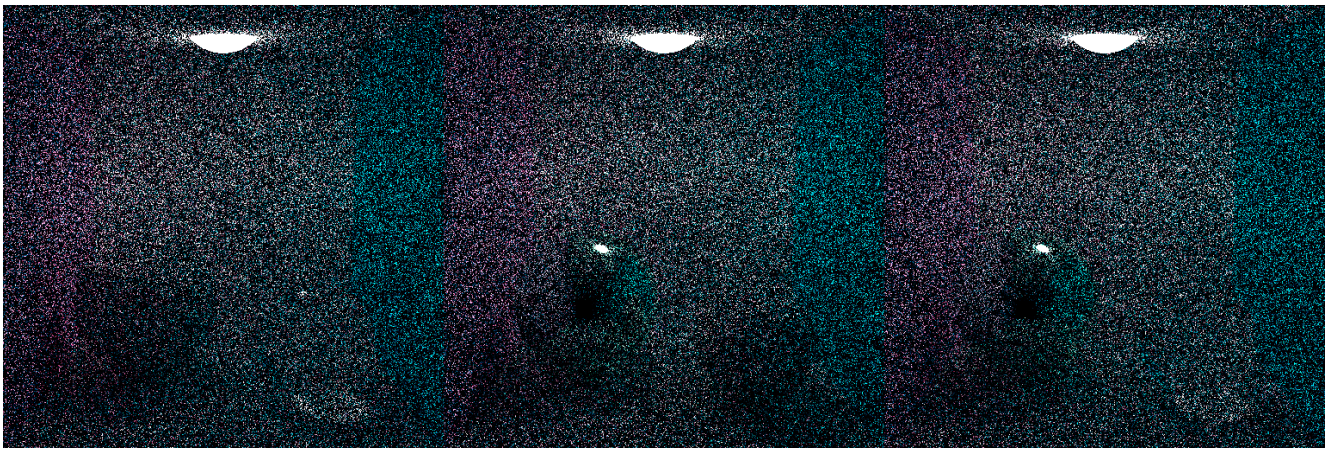

Рисунок 4.1 – PL: 128, SPP: 10.

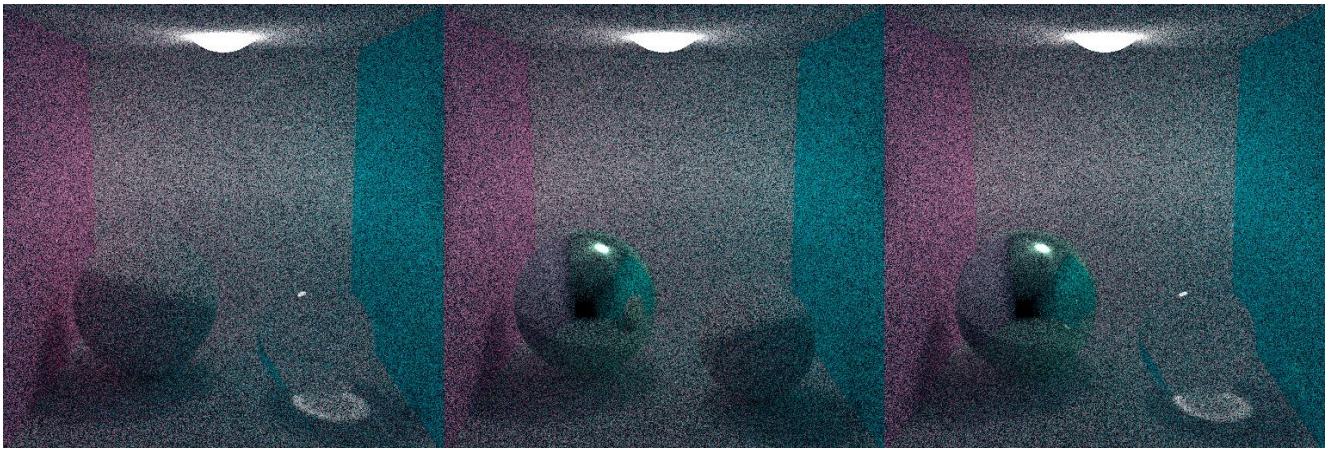

Рисунок 4.2 – PL: 128, SPP: 100.

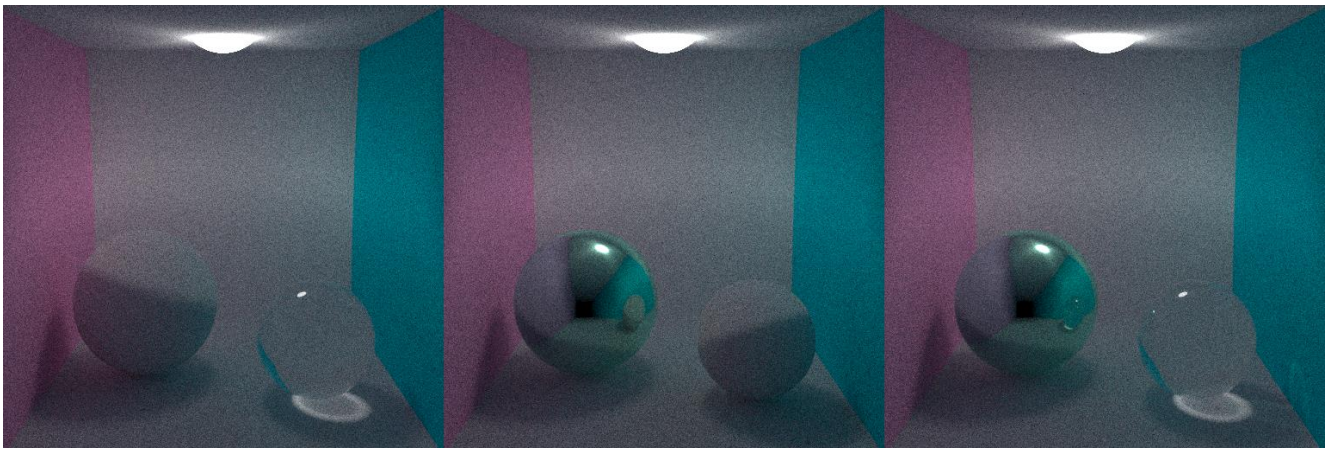

Рисунок 4.3 – PL: 128, SPP: 1000.

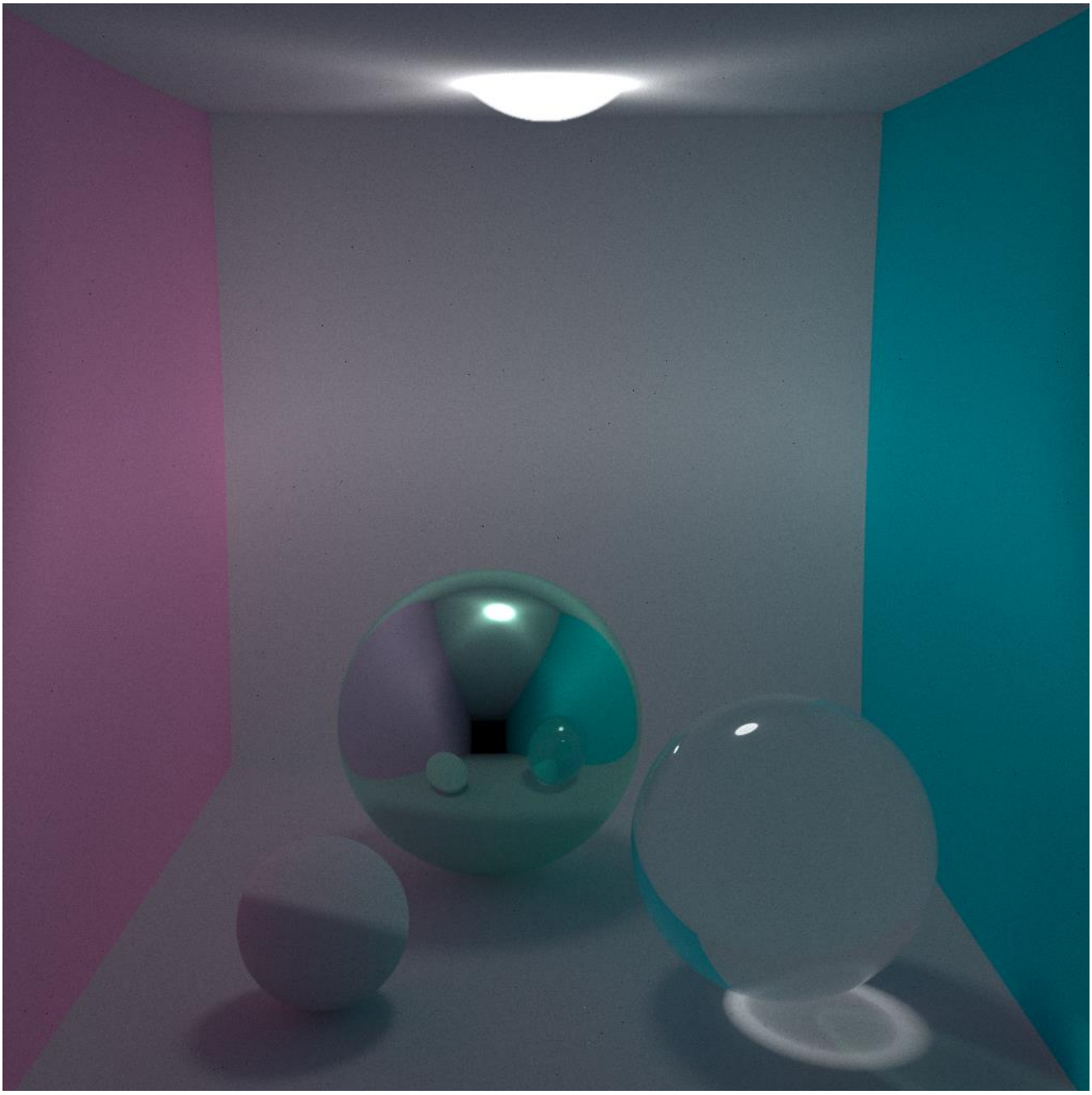

Рисунок 4.4 – PL: 128, SPP: 10000.

5 ЗАКЛЮЧЕНИЕ

С повышением производительности устройств, предназначенных для массово-параллельных вычислений, раздел алгоритмов, относящихся к просчёту глобального освещения, станет ещё более популярным. Современные подходы к симуляции поведения света в различных приложениях являются чрезмерно усложнёнными. Трассировка пути решает данную проблему за короткий период времени уже сейчас, эффективно задействуя вычислительные мощности современных процессоров.

В ходе БР был реализован алгоритм трассировки пути, осуществляющий фотореалистичный рендеринг. Данный алгоритм написан на специализированном языке OpenCL, что позволило осуществить его запуск на графическом процессоре. По затраченному времени на рендеринг изображения можно судить о том, что графический процессор является более подходящим устройством для выполнения задачи просчёта пути луча, в сравнении с центральным процессором.

Дальнейшее усовершенствование алгоритма для повышения его эффективности является открытым вопросом. Одним из возможных вариантов улучшения является применение BVH-деревьев. Данная структура данных позволяет эффективнее производить выборку объектов, с которыми пересекается луч. Эффективное построение BVH-деревьев с пространственным разбиением описано в работе [12].

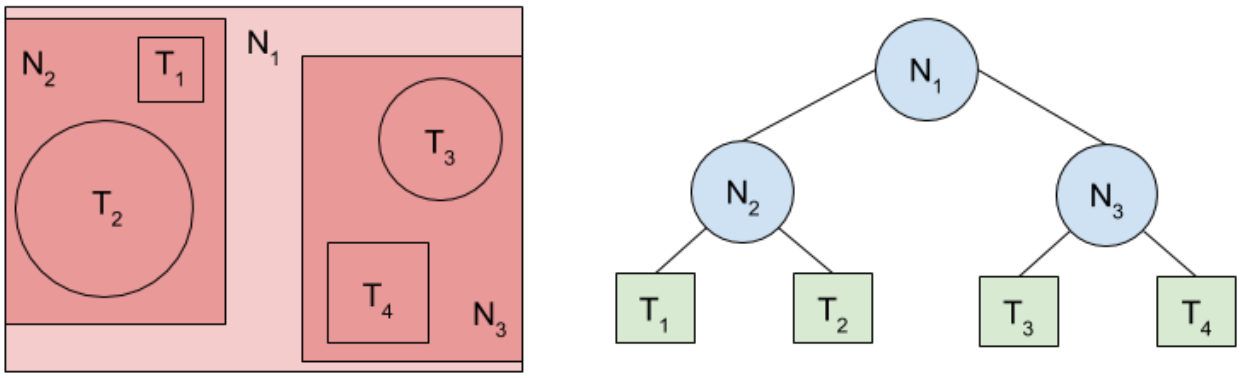

Рисунок 5.1 – BVH-деревья.

Таким образом, применение BVH-деревьев призвано сократить время выполнения алгоритма, увеличив при этом накладные расходы на содержание самого дерева в памяти устройства.

Продолжением алгоритма трассировки пути является двунаправленная трассировка пути (BDPT) – расширенная версия алгоритма, в основе которой лежит метод просчёта пути не только от виртуальной камеры к источнику освещения, но и обратный метод. Данная техника является более сложной с точки зрения реализации, однако позволяет получить менее «шумное» изображение за тот же период времени выполнения алгоритма, за который выполняется обычная трассировка пути. Также BDPT позволяет решить проблему локальных и

труднодоступных источников света в обычном алгоритме, что отображено на рисунке 5.1.

Возможны также улучшения непосредственно двунаправленной трассировки пути. Одним из методов редукции шума в изображении является алгоритм G-BDPT (gradient-domain bidirectional path tracing), с помощью которого можно получить не только более чёткое изображение, но и уменьшить время выполнения самого рендеринга [13].

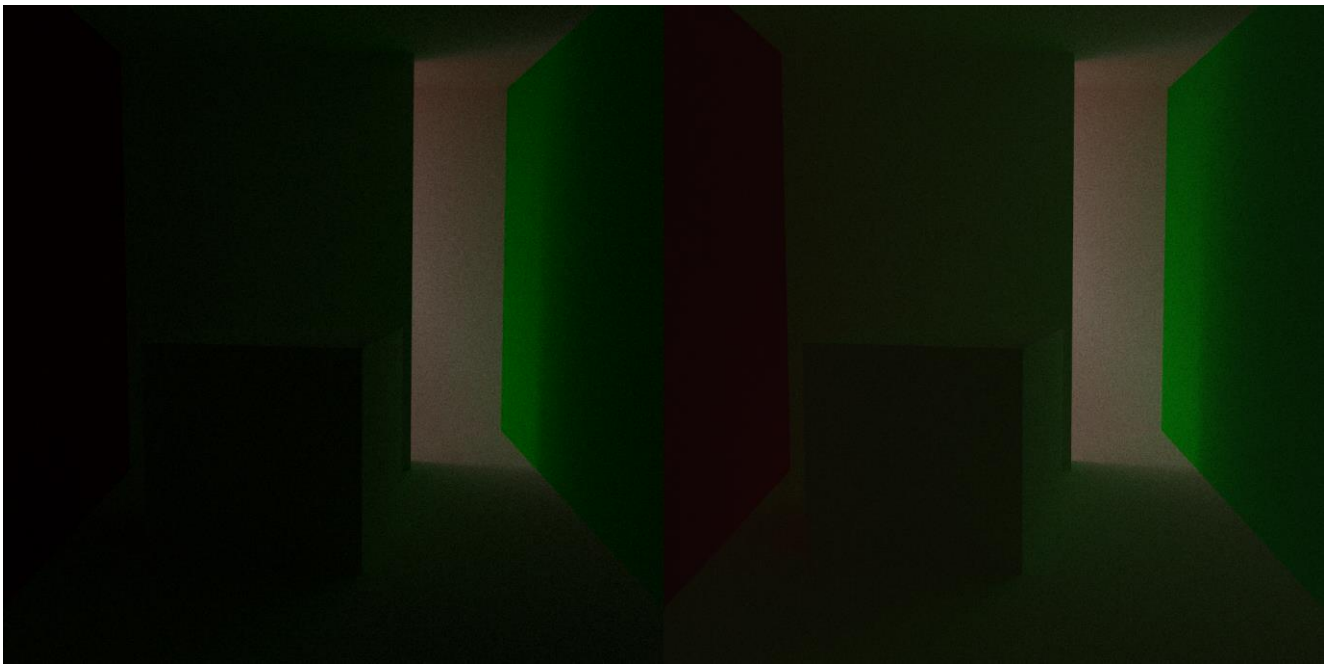

Рисунок 5.2 – Труднодоступный источник освещения в 3D-сцене (слева PT, справа BDPT).

ПРИЛОЖЕНИЕ А

Библиография

1 Хорошевский В.Г. Архитектура вычислительных систем: Учеб. пособие / В.Г. Хорошевский. – 2-е изд., перераб. и доп. – М.: Изд-во МГТУ им. Н.Э. Баумана, 2008. 121 с. – (Информатика в техническом университете, ISBN 987 5 7038 3175 5)

2 Основы компьютерной графики и технологии трехмерного моделирования [Электронный ресурс] : учеб. пособие / Л. Ю. Забелин, О. Л. Конюкова, О. В. Диль ; Сиб. гос. ун-т телекоммуникаций и информатики. - Новосибирск : СибГУТИ, 2015. - 259 с. : ил. - Библиогр.: с. 257-258. - Б. ц.

3 Kevin Suffern. Ray Tracing from the Ground Up. – A. K. Peters, Ltd. : Los Altos, California, 2007. – 782 p.

4 The OpenCL Programming Book / Ryoji Tsuchiyama [et al.]. – Fixstars, 2012. – 324 p.

5 AMD OpenCL Programming User Guide / AMD Developer Central [Electronic resource] // Сайт AMD-developers. – Режим доступа : http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_OpenCL_Programming_User_Guide2.pdf, свободный. – 2015. – 180 с.

6 James T Kajiya. The rendering equation. – ACM SIGGRAPH Computer Graphics, 1986. – Volume 20 Issue 4.

7 Heterogeneous Computing With OpenCL 2.0. David R. Kaeli [et al.]. – Morgan Kaufmann, 2015. – 308 p.

8 AMD Radeon R9 290 ‘Hawaii’ GPU Block Diagram Pictured and Detailed / WCCFtech [Electronic resource] // Новинки аппаратного обеспечения, мобильных технологий и игровой индустрии. – URL : http://wccftech.com/amd-radeon-r9-290-hawaii-gpu-block-diagram-pictured-detailed/ (дата обращения : 15.06.2016).

9 OpenCL Programming Guide / Aaftab Munshi [et al.]. – Addison-Wesley, 2011. – 648 p.

10 Казённов А.М. Основы технологии CUDA и OpenCL. – Москва, 2013. – 67 с.

11 Струняшев А.Н. Анализ практического применения технологии GPGPU // Современная техника и технологии. 2013. №3 [Электронный ресурс]. URL : http://technology.snauka.ru/2013/03/1714 (дата обращения: 15.06.2016).

12 Martin Stich, Heiko Friedrich, Andreas Dietrich. Spacial splits in bounding volume hierarchies. – Proceedings of the Conference on High Performance Graphics, HPG`09. – Pages 7-13.

13 Gradient-Domain Bidirectional Path Tracing. Marco Manzi [et al.]. – Eurographics Symposium on Rendering, 2015. – 10 p.

14 Rendering: теория / Render.ru [Электронный ресурс] // Российский интернет ресурс по компьютерной графике и анимации. – URL : http://render.ru/books/show_book.php?book_id=521 (дата обращения 15.06.2016).

15 Stephen K. Park, Keith W. Miller. Random Number Generators: Good ones are hard to fine. – Communications of the ACM 31, 1192-1201.

ПРИЛОЖЕНИЕ Б

Наиболее употребляемые текстовые сокращения

ВС – вычислительная система
ПЗ – пояснительная записка
БР – бакалаврская работа
PL – path length
SPP – samples per pixel
ALU – arithmetic and logic unit
SE – shader engine
CU – compute unit
GCN – graphics core next
ACE – asynchronous command engine
CUDA – Compute Unified Device Architecture
OpenCL – Open Computing Language
GPGPU – general-purpose computing for GPU`s
CPU – central processing unit
GPU – graphics processing unit
PT – path tracing
BDPT – bidirectional path tracing
BVH – bounding volume hierarchy

ПРИЛОЖЕНИЕ В

Код программы

Листинг В.1 – Main.cpp.

```
#include <fstream>
#include <iostream>
#include <string>

#include <cfloat>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <ctime>

#define MAX_SOURCE_SIZE (0x100000)
#include <CL/cl.h>

#define WIDTH      1024
#define HEIGHT     1024
#define ASPECT     WIDTH / HEIGHT
#define NPIX       WIDTH * HEIGHT

#define SAMPLES    10000
#define SCALE      1.0F / SAMPLES
#define MAX_DEPTH  128
#define NOBJ       9

typedef struct Material {
    cl_float4 color;

    float reflectivity;
    float refractivity;
    cl_float4 emissivity;
    float ior;
} Material;

 typedef struct Sphere {
    cl_float4 center;
    cl_float radius;
    Material material;
} Sphere;

cl_int init_gpu(cl_device_id &, cl_context &, cl_command_queue
    &);
cl_int init_kernel(cl_device_id &, cl_context &, cl_kernel &,
    cl_program &);
cl_int run_kernel(cl_context &, cl_command_queue &, cl_kernel
    &);
cl_int free_kernel(cl_context &, cl_command_queue &, cl_kernel
    &, cl_program &);

int gamma_correction(float val);
```

```
Sphere *Cornell_Box();

int main(void)
{
    srand((unsigned)time(NULL));

    /* Initialize all the GPU-side stuff */

    cl_device_id        device_id;
    cl_context              context;
    cl_command_queue    command_queue;
    cl_kernel           kernel;
    cl_program              program;
    cl_int                  ret;

    std::cout << "Initialize device ... ";
    if (!(ret = init_gpu(device_id, context, command_queue)))
         std::cout << "OK!\n";
    else {
         std::cout << "Something is going wrong. Error code: "
              << ret << "\n"; return -1;
    }

    std::cout << "Initialize kernel ... \n";
    if (!(ret = init_kernel(device_id, context, kernel,
         program))) std::cout << "OK!\n";
    else {
         std::cout << "Something is going wrong. Error code: "
              << ret << "\n"; return -1;
    }

    std::cout << "Run kernel ... ";
    if (!(ret = run_kernel(context, command_queue, kernel)))
         std::cout << "OK!\n";
    else {
         std::cout << "Something is going wrong. Error code: "
              << ret << "\n"; return -1;
    }

    std::cout << "Free kernel`s memory ... \n";
    if (!(ret = free_kernel(context, command_queue, kernel,
         program))) std::cout << "OK!\n";
    else {
         std::cout << "Something is going wrong. Error code: "
              << ret << "\n"; return -1;
    }

    return 0;
}

cl_int init_gpu(cl_device_id &device_id, cl_context &context,
    cl_command_queue &command_queue)
{
    cl_platform_id platform_id;
```

```
cl_uint ret_num_platforms, ret_num_devices;
cl_int ret;

ret = clGetPlatformIDs(1, &platform_id,
     &ret_num_platforms);
ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1,
     &device_id, &ret_num_devices);

/* GET INFO */

size_t retSize;
cl_ulong size;
cl_uint mcu;

bool *isAvail = NULL;
clGetDeviceInfo(device_id, CL_DEVICE_AVAILABLE,
     sizeof(cl_bool), &isAvail, 0);
if (isAvail) std::cout << "DEVICE IS AVAILABLE.\n";


std::cout <<
     "*****************************************\n";

ret = clGetDeviceInfo(device_id,
     CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(cl_uint), &mcu,
     0);
std::cout << " CL_DEVICE_MAX_COMPUTE_UNITS:\t" << mcu <<
     "\n";

ret = clGetDeviceInfo(device_id,
     CL_DEVICE_MAX_MEM_ALLOC_SIZE, sizeof(cl_ulong), &size,
     0);
std::cout << " CL_DEVICE_MAX_MEM_ALLOC_SIZE:\t" << size <<
     "\n";

ret = clGetDeviceInfo(device_id, CL_DEVICE_GLOBAL_MEM_SIZE,
     sizeof(cl_ulong), &size, 0);
std::cout << " CL_DEVICE_GLOBAL_MEMORY_SIZE:\t" << size <<
     "\n";

ret = clGetDeviceInfo(device_id, CL_DEVICE_LOCAL_MEM_SIZE,
     sizeof(cl_ulong), &size, 0);
std::cout << " CL_DEVICE_LOCAL_MEMORY_SIZE:\t" << size <<
     "\n";

ret = clGetDeviceInfo(device_id,
     CL_DEVICE_MAX_WORK_GROUP_SIZE, sizeof(size_t),
     &retSize, 0);
std::cout << " CL_DEVICE_MAX_WORK_GROUP_SIZE:\t" << retSize
     << "\n";

ret = clGetDeviceInfo(device_id,
     CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS, sizeof(cl_uint),
     &mcu, 0);
```

```
    std::cout << " CL_DEVICE_MAX_WORK_ITEM_DIMEN:\t" << mcu <<
        "\n";

    ret = clGetDeviceInfo(device_id,
        CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE, sizeof(cl_ulong),
        &size, 0);
    std::cout << " CL_DEVICE_MAX_CONST_BUFF_SIZE:\t" << size <<
        "\n";

    std::cout <<
        "*****************************************\n";

    /* CREATE CONTEXT & COMMAND QUEUE */

    context = clCreateContext(NULL, 1, &device_id, NULL, NULL,
        &ret);
    command_queue = clCreateCommandQueue(context, device_id, 0,
        &ret);

    return ret;
}

cl_int init_kernel(cl_device_id &device_id, cl_context
    &context, cl_kernel &kernel, cl_program &program)
{
    program = NULL;
    kernel = NULL;

    cl_int ret;

    FILE *fp;
    const char fileName[] = "kernel.cl";
    size_t source_size;
    char *source_str;


    fp = fopen(fileName, "r");
    if (!fp) {
        fprintf(stderr, "Failed to load kernel.\n");
        exit(1);
    }
    source_str = (char *)malloc(MAX_SOURCE_SIZE);
    source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
    fclose(fp);

    program = clCreateProgramWithSource(context, 1, (const char
        **)&source_str, (const size_t *)&source_size, &ret);
    ret = clBuildProgram(program, 1, &device_id, NULL, NULL,
        NULL);
    kernel = clCreateKernel(program, "test", &ret);

    return ret;
}
```

```
cl_int run_kernel(cl_context &context, cl_command_queue
    &command_queue, cl_kernel &kernel)
{
    cl_mem res_arr = NULL;
    cl_mem rand_seed = NULL;
    cl_mem spheres = NULL;

    size_t mem_length = NPIX;
    cl_float4 *total = new cl_float4[mem_length];
    Sphere *s = Cornell_Box();
    cl_int *rs = new cl_int[mem_length];
    cl_int ret;

    for (int i = 0; i < mem_length; ++i)
         rs[i] = rand();

    res_arr = clCreateBuffer(context, CL_MEM_READ_WRITE,
         mem_length * sizeof(cl_float4), NULL, &ret);
    ret = clEnqueueWriteBuffer(command_queue, res_arr, CL_TRUE,
         0, mem_length * sizeof(cl_float4), total, 0, NULL,
         NULL);
    ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void
         *)&res_arr);

    rand_seed = clCreateBuffer(context, CL_MEM_READ_WRITE,
         mem_length * sizeof(cl_int), NULL, &ret);
    ret = clEnqueueWriteBuffer(command_queue, rand_seed,
         CL_TRUE, 0, mem_length * sizeof(cl_int), rs, 0, NULL,
         NULL);
    ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void
         *)&rand_seed);

    spheres = clCreateBuffer(context, CL_MEM_READ_WRITE, NOBJ *
         sizeof(Sphere), NULL, &ret);
    ret = clEnqueueWriteBuffer(command_queue, spheres, CL_TRUE,
         0, NOBJ * sizeof(Sphere), s, 0, NULL, NULL);
    ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void
         *)&spheres);

    size_t global_work_size[1] = { (size_t)mem_length };

    ret = clEnqueueNDRangeKernel(command_queue, kernel, 1,
         NULL, global_work_size, NULL, 0, NULL, NULL);
    ret = clEnqueueReadBuffer(command_queue, res_arr, CL_TRUE,
         0, mem_length * sizeof(cl_float4), total, 0, NULL,
         NULL);

    std::ofstream fout("super_img.ppm");
    fout << "P3\n" << WIDTH << " " << HEIGHT << "\n255\n";
    for (int i = NPIX - 1; i >= 0; --i) {
         fout << gamma_correction(total[i].x) << " "
              << gamma_correction(total[i].y) << " "
              << gamma_correction(total[i].z) << "\n";
    }
```

```
    fout.close();

    std::cout << "Done !!!\n";
    delete[] total;
    ret = clReleaseMemObject(res_arr);

    return ret;
}

cl_int free_kernel(cl_context &context, cl_command_queue
    &command_queue, cl_kernel &kernel, cl_program &program)
{
    cl_int ret;

    ret = clReleaseKernel(kernel);
    ret = clReleaseProgram(program);
    ret = clReleaseCommandQueue(command_queue);
    ret = clReleaseContext(context);

    return ret;
}

inline float ffmin(float a, float b) { return a < b ? a : b; }
inline float ffmax(float a, float b) { return a > b ? a : b; }

int gamma_correction(float val)
{
    return int(255.0f * sqrt(ffmax(0.0f, ffmin(1.0f, val))));
}

Sphere *Cornell_Box()
{
    std::cout << "Initialize scene ...\n\n";

    Sphere *list = new Sphere[NOBJ];

    int i = 0;
    cl_float4 s1 = { -10277.5f,  0.0f,  0.0f  };
    cl_float4 s2 = {  10277.5f,  0.0f,  0.0f  };
    cl_float4 s3 = {  0.0f,  10277.5f,  0.0f  };
    cl_float4 s4 = {  0.0f, -10277.5f,  0.0f  };
    cl_float4 s5 = {  0.0f,  0.0f,  10555.0f  };
    cl_float4 s6 = {  0.0f,  337.5f,  277.5f  };
    cl_float4 s7 = {  140.5f, -187.5f, 50.0f  };
    cl_float4 s8 = { -40.5f, -167.5f, 250.0f };
    cl_float4 s9 = { -130.5f, -227.5f, 30.0f };

    cl_float4 gn = { 1.0f, 0.41f, 0.7f };
    cl_float4 rd = { 0.0f, 0.69f, 0.78f };
    cl_float4 bl = { 0.0f, 0.0f, 0.0f };
    cl_float4 gh = { 0.7f, 0.7f, 0.7f };
    cl_float4 aq = { 0.5f, 1.0f, 0.8f };
    cl_float4 super = { 15.0f, 15.0f, 15.0f };
```

```
    Material lwall = { gn, 0.0f, 0.0f, bl, 1.0f };
    Material rwall = { rd, 0.0f, 0.0f, bl, 1.0f };
    Material owall = { gh, 0.0f, 0.0f, bl, 1.0f };
    Material metal = { aq, 1.0f, 0.0f, bl, 1.0f };
    Material glass = { bl, 1.0f, 1.0f, bl, 1.5f };
    Material light = { bl, 0.0f, 0.0f, super, 1.0f };

    Sphere a1 = { s1, 10000.0f, lwall };
    Sphere a2 = { s2, 10000.0f, rwall };
    Sphere a3 = { s3, 10000.0f, owall };
    Sphere a4 = { s4, 10000.0f, owall };
    Sphere a5 = { s5, 10000.0f, owall };
    Sphere a6 = { s6, 80.0f, light };
    Sphere a7 = { s7, 90.0f, glass };
    Sphere a8 = { s8, 110.0f, metal };
    Sphere a9 = { s9, 50.0f, owall };

    list[i++] = a1;
    list[i++] = a2;
    list[i++] = a3;
    list[i++] = a4;
    list[i++] = a5;

    list[i++] = a6;
    list[i++] = a7;
    list[i++] = a8;
    list[i++] = a9;

    std::cout << "Number of objects: " << i << "\n\n";

    return list;
}
```

Листинг В.2 – Kernel.cl.

```
typedef struct Material {
    float4 color;

    float reflectivity;
    float refractivity;
    float4 emissivity;
    float ior;
}Material;

typedef struct Hit_record {
    float4 point;
    float4 normal;
    float depth;
    Material material;
}Hit_record;

typedef struct Ray {
    float4 origin;
```

```
    float4 direction;
}Ray;

typedef struct Sphere {
    float4 center;
    float radius;
    Material material;
}Sphere;

typedef struct Camera {
    float4 lowleftcorner;
    float4 horizontal;
    float4 vertical;
    float4 origin;
}Camera;

int random_number_kernel(int *seed)
{
    int const a = 16807; //ie 7**5
    int const m = 2147483646; //ie 2**31-1
    *seed = ( (long)(*seed * a)) % m;
    return (*seed);
}

int rand(__global int* seed_memory)
{
    int global_id = get_global_id(1) * get_global_size(0) +
         get_global_id(0);
    int seed = seed_memory[global_id];
    int result = seed;
    result = result & 0x7FFFFFFF;
    int random_number = random_number_kernel(&seed);
    seed_memory[global_id] = seed;
    return result;
}

float drand(__global int* seed_memory)
{
    return rand(seed_memory) / (float)2147483646;
}

/********************* KERNEL *********************/

bool refraction(float4 v1, float4 v2, float nu, float4 *refr)
{
    float4 uv = normalize(v1);
    float t = dot(uv, v2);
    float D = 1.0f - nu * nu * (1.0f - t * t);

    if (D > 0.0f) {
         *refr = nu * (v1 - v2 * t) - v2 * sqrt(D);
         return true;
    }
    else
```

```
        return false;
}

float4 reflection(float4 v1, float4 v2)
{
    return v1 - 2.0f * dot(v1, v2) * v2;
}

float shlick_approximation(float cosine, float refract)
{
    float p = (1.0f - refract) / (1.0f + refract);
    p *= p;
    return p + (1.0f - p) * pow((1.0f - cosine), 5);
}

float4 get_ray(Ray r, float k)
{
    return r.origin + k * r.direction;
}

Ray shoot_ray(Camera cam, float w, float h)
{
    Ray r = { cam.origin, cam.lowleftcorner + w*cam.horizontal
        + h*cam.vertical - cam.origin };
    return r;
}

bool diffusion(Ray r, Hit_record rec, float4 *fade, Ray *scat,
    __global int* seed_memory)
{
    if (length(rec.material.emissivity) > 0.0f) return false;

    if (rec.material.reflectivity == 0.0f &&
        rec.material.refractivity == 0.0f) {
        // matte
        float4 v1 = {drand(seed_memory) * 2.0 - 1.0,
            drand(seed_memory) * 2.0 - 1.0,
            drand(seed_memory) * 2.0 - 1.0, 0};
        v1 = normalize(v1);
        if (dot(v1, rec.normal) < 0.0f) v1 = -v1;
        Ray scat_ray = { rec.point, v1};
        *scat = scat_ray;
        *fade = rec.material.color;

        return true;
    } else if (rec.material.reflectivity > 0.0f &&
        rec.material.refractivity == 0.0f) {
        // metal

        float4 refl = reflection(normalize(r.direction),
            rec.normal);
        Ray scat_ray = { rec.point, refl };
        *scat = scat_ray;
        *fade = rec.material.color;
```

```
            return (dot(scat->direction, rec.normal) > 0);
        }
        else {
            // glass

            float4 out_normal;
            float4 refr;
            float4 refl = reflection(normalize(r.direction),
                rec.normal);

            float nu, prob, cosi;

            *fade = (float4){1.0f, 1.0f, 1.0f, 0.0f};

            if (dot(r.direction, rec.normal) > 0) {
                out_normal = -rec.normal;
                nu = rec.material.ior;
                cosi = dot(r.direction, rec.normal) /
                    length(r.direction);
                cosi = sqrt(1.0f -
                    rec.material.ior*rec.material.ior * (1.0f -
                    cosi*cosi));
            } else {
                out_normal = rec.normal;
                nu = 1.0f / rec.material.ior;
                cosi = -dot(r.direction, rec.normal) /
                    length(r.direction);
            }

            float4 v1 = r.direction;
            float4 v2 = out_normal;
            float4 uv = normalize(v1);
            float t = dot(uv, v2);
            float D = 1.0f - nu * nu * (1.0f - t * t);

            if (D > 0.0f) {
                refr = nu * (v1 - v2 * t) - v2 * sqrt(D);
                prob = shlick_approximation(cosi,
                    rec.material.ior);
            } else prob = 1.0f;

            if (drand(seed_memory) < prob) {
                Ray r = { rec.point, refl };
                *scat = r;
            } else {
                Ray r = { rec.point, refr };
                *scat = r;
            }

            return true;
        }
    }
```

```
bool get_intersection(Ray r, __global Sphere *s, float
    min_depth, float max_depth, Hit_record *rec)
{
    float4 oc;
    float4 ro = r.origin;
    float4 ce = s->center;
    oc = ro - ce;

    float a = dot(r.direction, r.direction);
    float b = 2.0f * dot(oc, r.direction);
    float c = dot(oc, oc) - s->radius * s->radius;
    float D = b*b - 4.0f*a*c;

    if (D > 0) {
         float d1 = (-b - sqrt(D)) / (2.0f * a);
         float d2 = (-b + sqrt(D)) / (2.0f * a);
         float root = (d1 < d2) ? d1 : d2;

         if (root < max_depth && root > min_depth) {
              rec->point = get_ray(r, root);
              rec->normal = (get_ray(r, root) - ce)/s->radius;
              rec->depth = root;
              rec->material = s->material;

              return true;
         }
    }

    return false;
}

bool nearest_intersection(Ray r, __global Sphere *scene, float
    min_depth, float max_depth, Hit_record *rec, int n)
{
    Hit_record tmp;

    bool hit = false;
    float dist = max_depth;

    for (int i = 0; i < n; ++i) {
         if (get_intersection(r, &scene[i], min_depth, dist,
              &tmp)) {
              hit = true;
              dist = tmp.depth;
              *rec = tmp;
         }
    }

    return hit;
}


float4 raytrace(Ray r, __global Sphere *scene, __global int
    *seed_memory, int nobj)
```

```
{
    Ray primary = r;
    Ray scattered;
    float4 attenuated;
    float4 emitted;
    float4 composite = {0.0f, 0.0f, 0.0f, 0.0f};
    float4 counted = {1.0f, 1.0f, 1.0f, 0.0f};

    Hit_record rec;
    for (int i = 0; i < 128; ++i) {
        if (nearest_intersection(primary, scene, 0.001f,
            1E+37f, &rec, nobj)) {
            emitted = rec.material.emissivity;
            if (diffusion(primary, rec, &attenuated,
                &scattered, seed_memory)) {
                primary = scattered;
                composite += emitted * counted;
                counted *= attenuated;
            } else
                return composite + emitted * counted;
        } else
            return (float4){0.0f, 0.0f, 0.0f, 0.0f};
    }

    return (float4){0.0f, 0.0f, 0.0f, 0.0f};
}

void create_cam(Camera *cam, float4 pov, float4 target, float
    fov, float aspectratio)
{
    cam->lowleftcorner = 0.0f;
    cam->horizontal = 0.0f;
    cam->vertical = 0.0f;
    cam->origin = pov;

    float4 upward = { 0.0f, 1.0f, 0.0f, 0.0f };

    float alpha = fov * 3.14159265359f / 180.0f;
    float vfov = tan(alpha / 2.0f);
    float hfov = aspectratio * vfov;

    float4 dz = normalize(pov - target);
    float4 crossed = cross(upward, dz);
    float4 dx = normalize(crossed);
    float4 dy = cross(dz, dx);

    cam->lowleftcorner.x = cam->origin.x - hfov*dx.x -
        vfov*dy.x - dz.x;
    cam->lowleftcorner.y = cam->origin.y - hfov*dx.y -
        vfov*dy.y - dz.y;
    cam->lowleftcorner.z = cam->origin.z - hfov*dx.z -
        vfov*dy.z - dz.z;

    cam->horizontal.x = 2.0f * hfov * dx.x;
```

```
    cam->horizontal.y = 2.0f * hfov * dx.y;
    cam->horizontal.z = 2.0f * hfov * dx.z;

    cam->vertical.x = 2.0f * vfov * dy.x;
    cam->vertical.y = 2.0f * vfov * dy.y;
    cam->vertical.z = 2.0f * vfov * dy.z;
}

__kernel void test(__global float4 *result, __global int
    *seed_memory, __global Sphere *scene)
{
    int SAMPLES = 100;
    int WIDTH = 1024;
    int gid = get_global_id(0);
    int index = gid;

    float4 zero = 0.0f;

    result[gid] = zero;
    float scale = 1.0f / (float)SAMPLES;

    Ray primary;
    float4 gidded, scaled, new_val;

    Camera cam;
    create_cam(&cam, (float4){0.0f, 128.0f, -800.0f, 0.0f},
         (float4) {0.0f, 0.0f, 0.0f, 0.0f}, 40.0f, 1.0f);
    primary.origin = cam.origin;
    float4 part1 = cam.lowleftcorner;
    float4 part2 = cam.horizontal;
    float4 part3 = cam.vertical;

    __local float w_off, h_off;

    float4 w, h;

    for (int spp = 0; spp < SAMPLES; ++spp) {
         w_off = (float)(index % WIDTH + drand(seed_memory)) /
              (float)(WIDTH);
         h_off = (float)(index / WIDTH + drand(seed_memory)) /
              (float)(WIDTH);

         w = w_off * part2;
         h = h_off * part3;
         primary.direction = part1 + w + h - primary.origin;

         new_val = raytrace(primary, scene, seed_memory, 9);
         scaled = scale * new_val;
         gidded = result[gid];
         result[gid] = scaled + gidded;
    }
}
```