Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Московский государственный технический университет имени Н.Э. Баумана»
(МГТУ им. Н.Э.Баумана)

ФАКУЛЬТЕТ «ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ»

КАФЕДРА «ТЕОРЕТИЧЕСКАЯ ИНФОРМАТИКА И КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

«Формирование рельефа поверхности
методом наложения карт смещения при трассировке лучей»

Студент ИУ9-51 _____ А. В. Разборщикова
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсового проекта _____ И. Э. Вишняков
(Подпись, дата) (И.О.Фамилия)

Москва, 2016

Содержание

Введение

Применение реалистичных изображений безгранично. Однако вместе с повышением сложности модели, возрастает и ее вычислительная сложность, и затраты памяти для ее хранения, что может оказаться критическим, например, для приложений реального времени. Одной из важнейших задач, решаемых в компьютерной графике, является поиск компромисса между желаемым качеством изображения и затратами на его формирование. *Объектом исследования* данной работы являются методы повышения реалистичности синтезированных моделей путем наложения двумерных текстур. В качестве *предмета исследования* было выбрано семейство алгоритмов, создающих эффект рельефной поверхности методом наложения карт смещения. Их *практическое значение* определяется меньшими затратами памяти по сравнению с высокодетализированными моделями, а также простотой модификации мелких деталей на поверхности.

Целью работы является написание приложения, визуализирующего 3D-модели методом трассировки лучей с использованием наложения карт смещения.

В ходе работы ставятся следующие *задачи*:

1) изучение методов формирования рельефа поверхности;
2) разработка алгоритма трассировки лучей с учетом формирования рельефа поверхности методом наложения карт смещения;
3) реализация разработанного алгоритма с использованием технологии параллельного программирования CUDA;
4) тестирование разработанного приложения.

1. Обзор методов формирования рельефа поверхности

Детали, из которых состоит модель, можно классифицировать по размеру на: макрообъекты (геометрические примитивы, определяющие форму модели); микрообъекты (не различимы при отрисовке, но задают параметры материала, создавая такие эффекты, как блеск или рассеивание света), и мезообъекты, имеющие размер нескольких пикселей. Мезогеометрия включает детали, слишком сложные для эффективного представления отдельными полигонами, но достаточно большие, чтобы создавать заметную кривизну поверхности в несколько пикселей. Это могут быть морщины на лице персонажа, детали мускулатуры, складки и швы на одежде.

В данной работе рассматриваются методы, добавляющие элементы мезогеометрии к макрообъектам.

1.1 Методы наложения карт нормалей

В 1978 году Джеймс Ф. Блинн (James F. Blinn) предложил метод описания мезогеометрии модели с помощью двумерного массива. Он предположил, что мелкая деталь на поверхности может быть заменена внесением возмущения в нормаль к ней, а информация об этих возмущениях помещена в массив (см. рис. 1). Тогда от плоской поверхности свет отражается так же, как от искривленной, имитируя рельеф. Этот метод называется **рельефное текстурирование** (*bump mapping*) [1 , с. 183].

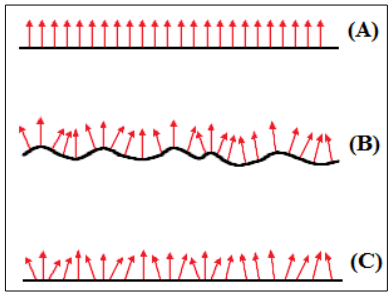
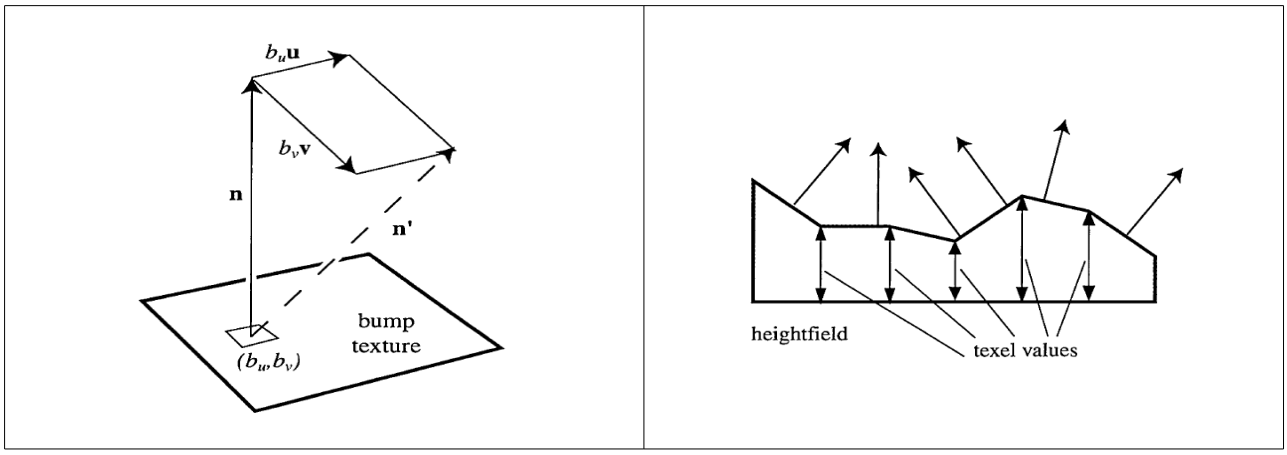


Рисунок 1 — Действие метода рельефного текстурирования: поверхность (C) получена наложением на плоскость (A) нормалей к поверхности (B).

Существует два основных метода представления карты возмущения нормалей [1 , с. 186-187]:

1. Как таблицы с точками (b_u , b_v), если поверхность задана параметрически функцией $\boldsymbol{P}(u , v)$, где b_u и b_v соответствуют изменению нормали вдоль осей ***u*** и ***v***; такой тип смещения называется рельефнымтекстурированием с вектором смещения (*offset vector bump map*), или смещенным отображением (o*ffset map*) (рисунок 2, а).
2. Как карты высот (*heightfield*), где одноцветное значение текстуры представляет собой высоту изменения поверхности (рисунок 2, б).



а) б)

Рисунок 2 — Интерпретация значения, хранимого в текстуре (bump texture): карта смещения (а) и карта высот (б).

При использовании метода рельефного текстурирования нормаль изменяет свое направление в соответствии с некоторой системой отсчета. Для этого в каждой вершине сохраняется базис *касательного пространства* (в объектных координатах), осями которого являются касательный и бикасательный векторы. Матрица преобразования, переводящая координаты светового луча из мировых координат в координаты касательного пространства, должна быть вычислена отдельно для каждой вершины [1, с. 184-185].

Если считать, что поверхность задана параметрически функцией $\boldsymbol{P}(u , v)$, то в случае использования карты высот изменение значения нормали в каждой

точке задается функцией $d(u, v)$. Тогда формула для измененной поверхности $\boldsymbol{P}_1(u, v)$ имеет вид (1):

$$\boldsymbol{P}_1(u, v) = \boldsymbol{P}(u, v) + d(u, v)\,\boldsymbol{n}. \quad (1)$$

После этого можно приближенно вычислить новый нормальный вектор смещения с помощью формул (2) и (3):

$$\boldsymbol{n} \approx \left(\frac{\partial \boldsymbol{P}}{\partial u} \times \frac{\partial \boldsymbol{P}}{\partial v}\right) + \left(\frac{\partial d}{\partial u}\boldsymbol{n} \times \frac{\partial \boldsymbol{P}}{\partial v}\right) - \left(\frac{\partial d}{\partial u}\boldsymbol{n} \times \frac{\partial \boldsymbol{P}}{\partial u}\right), \quad (2)$$

$$\boldsymbol{n}_0 = \frac{\boldsymbol{n}}{\|\boldsymbol{n}\|}, \quad (3)$$

где $\boldsymbol{n}$ — вектор, перпендикулярный поверхности $\boldsymbol{P}_1$, а $\boldsymbol{n}_0$ — искомая единичная нормаль (вывод формулы (2) см. в [2, с. 135-137]).

Таким образом, хранимые в текселях координаты b_u и b_v — это уже вычисленные приближенные значения частных производных. В случае использования карты высот направление нормали приходится вычислять «на месте», используя значения соседних текселей.

С усовершенствованием графических карт и усложнением моделей уменьшение вычислений получило больший приоритет, чем затраты памяти. Более предпочтительным стало хранение в памяти не двух значений смещения по координатам, а карты нормалей (*normal map*) целиком. Алгоритм и результат применения метода **наложения карт нормалей** (*normal mapping*) аналогичен применению рельефного текстурирования, изменяется только формат хранения данных: каждое направление кодируется цветом в модели RGB (см. рис. 3).

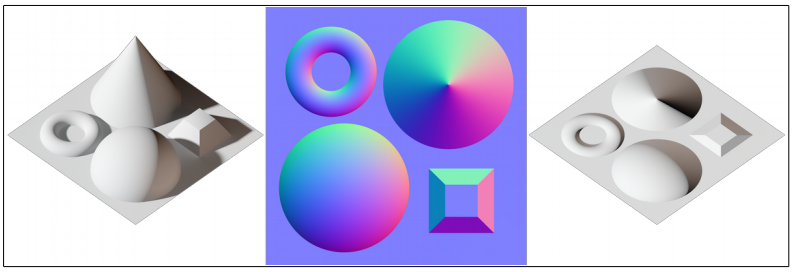


Рисунок 3 — Пример использования карты нормалей.

Важным аспектом при работе с картами нормалей является выбор системы координат для хранения векторов, так как при расчете освещения координаты луча и нормали должны быть представлены в одном базисе. Если хранить нормали к поверхности в мировых координатах, они потребуют пересчета при любых преобразованиях поверхности (повороте, сдвиге и пр.). Влияние выбора системы координат для хранения карты на эффективность метода рассмотрено в [1, с. 188-189].

Необходимо отметить, что методы рельефного текстурирования и наложения карт нормалей дают недостаточно реалистичный результат. Например, при взгляде на кирпичную стену под определенным углом швы между кирпичами должны перестать быть видны, но описанные выше методы не могут обеспечить эффект такого рода. Это происходит из-за несовпадения точки пересечения луча видимости с базовым полигоном $T_{видимая}$ и точки его пересечения с искривленной поверхностью $T_{корректная}$. Таким образом, полученное значение текстуры не соответствует ожидаемому (рис. 4). Поэтому возникает необходимость учитывать положение наблюдателя. Для решения этой проблемы Томомити Канэко (Tomomichi Kaneko) в 2001 году предложил использовать **параллактическое отображение**[1] (*parallax mapping*). Основная идея этого метода [3] заключается в том, чтобы на основе оценок, полученных опытным путем, и анализа рельефа изменить текстурные координаты таким образом, чтобы цвет видимой точки совпадал с ожидаемым. Серьезным недостатком такого метода является необходимость пересчета текстурных координат для каждой точки карты высот при каждом движении наблюдателя, что может стать критическим недостатком в приложениях реального времени.

1 **Паралла́кс** (от греч. parállaxis — отклонение) – видимое изменение положения предмета (тела) вследствие перемещения глаза наблюдателя. (ред. Прохоров А.М., Советский энциклопедический словарь. – М.: «Советская энциклопедия», 1980. – 1600 с.)

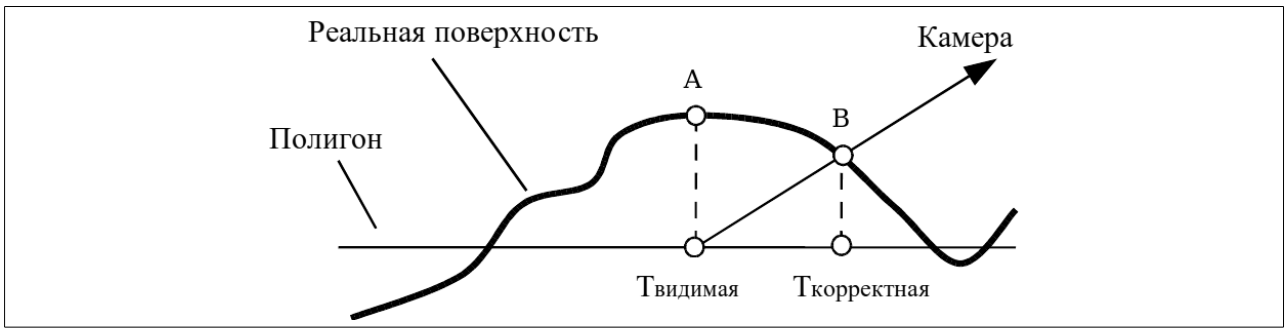


Рисунок 4 — Ошибка получения текстурных координат в случае, когда положение наблюдателя не учтено.

Следующим шагом в совершенствовании рассмотренного метода становится семейство методов, основанное на трассировке лучей (ray tracing) и иногда объединяемое под общим названием **parallax occlusion mapping** [1].

Данный метод, в отличие от упомянутого выше обыкновенного параллактического отображения, основан не на смещении, а на трассировке лучей: луч света выпускается в сторону карты высот (*height field*), и далее ищется ближайшая видимая точка на поверхности. Эту идею развивали независимо друг от друга несколько исследователей, и существует различные алгоритмы нахождения такой точки. В их основе лежит разбиение диапазона возможных значений высот на несколько равных отрезков (слоев) [1, с. 194]. Определив, в каком слое находится точка пересечения рельефной поверхности и луча, можно найти ее координаты с необходимой точностью. Это можно сделать, например, простым перебором слоев (*steep mapping*) или бинарным поиском (*relief mapping*). Наименьшими вычислительными затратами обладает метод, предложенный Натальей Татарчук (Natalya Tatarchuk) в работе [4]. Его суть заключается в том, что кривая рельефа в боковой проекции аппроксимируется кусочно-линейной (на рисунке 5 кривая заменяется отрезками на участках ширины δ).

Имея небольшую вычислительную сложность и простоту реализации, рассмотренные выше методы рельефного текстурирования обладают рядом существенных недостатков. Это ошибки некорректного отбрасывания тени (см. рисунок 3), в том числе самозатенения, и самоперекрытия в силу

сохранения очертания модели, что очень сильно влияет на реалистичность изображения. Усложняется задача отыскания точек пересечения моделей, к которым было применено такое текстурирование [5, с. 336].

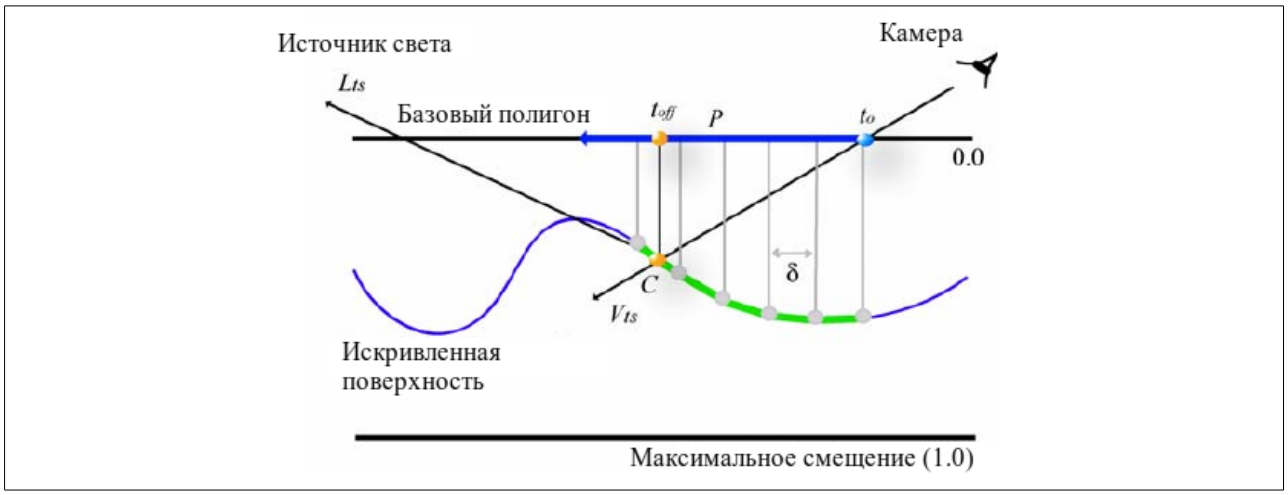


Рисунок 5 — Аппроксимация рельефа кусочно-линейной поверхностью.

Также возникают проблемы сглаживания изображения при удалении объекта от наблюдателя: при достаточно большом расстоянии отображаемые возмущения на поверхности могут становиться размером с пиксель или меньше. Как правило, вероятностная выборка может уменьшить ступенчатость изображения. Также в некоторых случаях возможно постепенно уменьшать высоту вносимых искажений в зависимости от степени детализации, с которой накладывается карта высот [2, с. 137].

Тем не менее в случаях отсутствия условий, в которых проявляются указанные выше недостатки, рельефное текстурирование дает хороший результат для диффузного затенения, может создать мельчайшие блики, корректно искривляет отражения [5, с. 336].

1.2 Скалярное смещающее отображение

Дальнейшим шагом в развитии идеи изменения рельефа при помощи текстур стали методы, все больше приближающие моделирование рельефа к его физической сущности, а именно **смещающее отображение** (*displacement mapping).*

Впервые данный метод был введен Робертом Куком (Robert L. Cook) в 1984 году. В отличие от рельефного текстурирования, смещающее отображение изменяет расположение элементов поверхности. Это позволяет добиться эффектов, недоступных при рельефном текстурировании (рисунок 6).

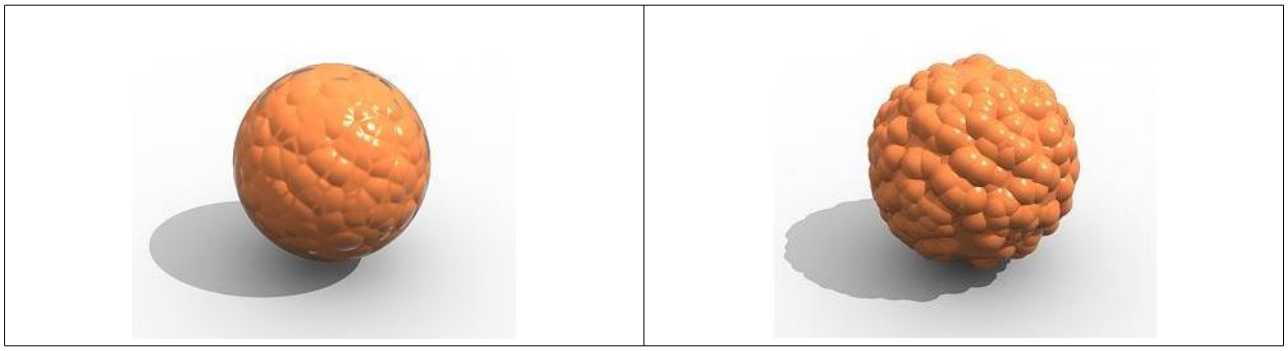


Рисунок 6 — Различия изображений при использовании рельефного текстурирования (слева) и смещающего отображения (справа).

Алгоритмы этого семейства работают следующим образом: на поверхности модели выбирается точка (*sample point*), которая сдвигается вдоль нормали к поверхности на расстояние, полученное из карты высот (в случае смещающего отображения ее называют *картой смещения* или *displacement map*). Выбранная точка может быть как вершиной исходной (или тесселированной) модели, так и точкой в центре текселя. В первом случае процесс будет носить название *вершинного смещения* (per-vertex displacement mapping), а во втором — *попиксельного смещения* (per-pixel displacement mapping).

В случае вершинного смещения изменение геометрии модели происходит в графическом конвейере. Для этого базовая поверхность итеративно тесселируется, после чего вершины смещаются вдоль нормали к базовой поверхности. Процесс продолжается до тех пор, пока получаемые полигоны не станут по размерам меньше одного пикселя. Данный алгоритм, реализованный на основе пиксельного шейдера, можно найти в [6].

В случае попиксельного смещения детали поверхности добавляются во время наложения цветовых текстур. Эта идея принадлежит Дж. Паттерсону (J.W. Patterson), назвавшему свой метод **обратным смещенным отображением**

(*inverse displacement mapping*) [7, с. 1-2]. Его особенностью является то, что вычисление пересечения луча и поверхности производится не в мировых координатах (как в большинстве реализаций), а в пространстве текстуры.

Первые реализации смещенного отображения работали медленно из-за аппаратных ограничений того времени и не давали желаемого результата в силу сложности создания карт смещения. Эти проблемы потеряли свою актуальность в 2000-е годы с появлением 3D-сканеров, позволявших считывать реальные карты высот поверхностей, а также мощных графических процессоров. Появилась возможность выполнять трассировку лучей в реальном времени, и она стала более предпочтительным решением для отображения карт смещений [8].

Несмотря на идейную простоту метода, наибольшая сложность заключается в поиске пересечения смещенной поверхности и луча видимости. Для этого были предложены различные методы, использовавшие, например, аффинную арифметику, сложные схемы кэширования и пересечения с решетками [7].

Самым простым является дискретный алгоритм (steeping algorithm), основанный на том, чтобы проходить вдоль луча видимости с определенным шагом [8]. Также могут использоваться алгоритмы, упомянутые ранее на стр. 8 применительно к параллактическому отображению с использованием трассировки. В результате развития этого направления появились и более сложные методы, например, **пирамидальное наложение смещения** (*pyramidal displacement mapping*) (см. рисунок 7) и **сферическая трассировка** (*sphere tracing*).

Пирамидальное наложение смещения аналогично использованию многоуровневой текстуры (*mip-map*), которая представляет собой квадродерево, где в каждый листовой тексель записана разность между максимальным смещением для данной поверхности и высотой текущего уровня для данного текселя. Корневой тексель на вершине построенной пирамиды задает

глобальную минимальную разность высот (значение d_0 на рис. 7). Каждый тексель на промежуточных уровнях хранит минимальную разность расстояний для четырех своих потомков (например, значение d_1 на рис. 7). Далее при прохождении лучом каждого уровня, он может продвинуться далее на расстояние, полученное из соответствующего узла текстуры, без риска пересечения с картой высот. Этот процесс продолжается до тех пор, пока не будет достигнут нижний уровень и вместе с ним искомая точка пересечения со смещенной поверхностью. Однако этот метод допускает случаи, при которых алгоритм дает неправильный результат, что делает его достаточно сложным в реализации [7].

Для выполнения сферической трассировки также сначала на основе двумерной текстуры вычисляется трехмерная карта расстояний (distance map), которая по сути является выровненной по осям охватывающей оболочкой (bounding box) для некоторой части смещаемой поверхности. Значение, хранимое в текселе для точки в трехмерном пространстве, интерпретируется как расстояние от этой точки до ближайшей к ней точки смещенной поверхности. Геометрически это может быть представлено как радиус наибольшей сферы с центром в данной точке, которая касается смещенной поверхности. Тогда из каждой точки на луче можно продвигаться вдоль него на расстояние, хранимое в трехмерной текстуре для данной точки, не пропустив точки пересечения (рисунок 7).

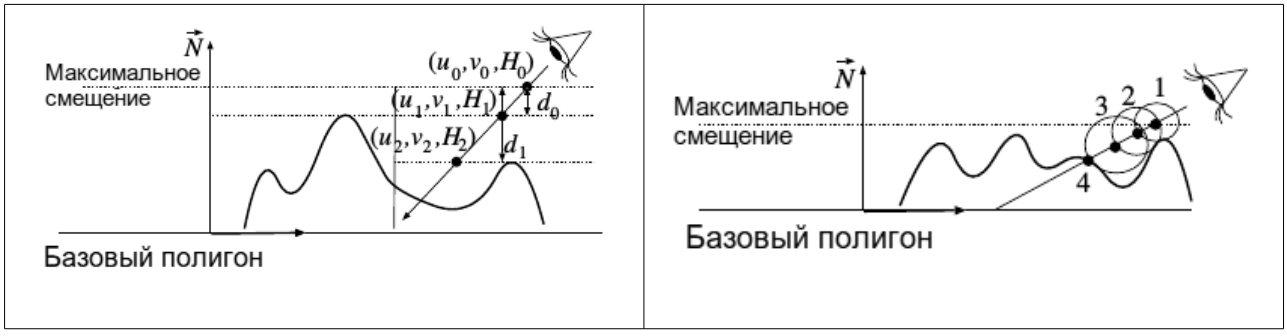


Рисунок 7 — Иллюстрация работы методов пирамидального смещения (слева) и сферической трассировки (справа).

Так как построение трехмерной карты требует значительных временных затрат, то целесообразно использовать этот метод в приложениях, где карта может быть вычислена один раз и сохранена для повторного использования.

После смещения помимо поиска пересечения луча с поверхностью возникает и другая задача — корректное изменение нормалей. Она решается теми же методами, которые были рассмотрены ранее для рельефного текстурирования в пункте 1.1. Применительно к смещающему отображению эти выкладки даны в работе [7]. Для некоторых методов дополнительные преобразования не нужны, например для вершинного смещения и алгоритма, описанного в работе [9]. Его основной принцип заключается в том, что базовый треугольник покрывается виртуальной сеткой из микротреугольников. Их вершины не хранятся в памяти постоянно, а вычисляются по очереди только для тех микротреугольников, над которыми проходит луч. Это позволяет значительно экономить память для хранения новых точек, а также сократить количество запросов из текстуры для получения величины смещения.

1.3 Векторное смещающее отображение

Дальнейшим развитием технологии рассмотренного выше скалярного метода стало векторное смещающее отображение (*vector displacement mapping*) [10]. В случае обыкновенного смещающего отображения в текселе хранится расстояние, на которое должна быть перемещена данная точка вдоль нормали к поверхности. В векторном методе сдвиг может быть произведен в любом направлении. В текстуре, как и в случае наложения карт нормалей (см. пункт 1.1), хранятся координаты вектора смещения для всех точек по осям касательного пространства $\boldsymbol{t}$, $\boldsymbol{n}$, $\boldsymbol{b}$, где каждая координата представлена цветом в модели RGB. Этот метод требует бóльших затрат памяти, чем скалярное смещающее отображение, однако он дает возможность представления более сложных элементов на поверхности модели при более низкой ее исходной детализации (см. рисунок 8).

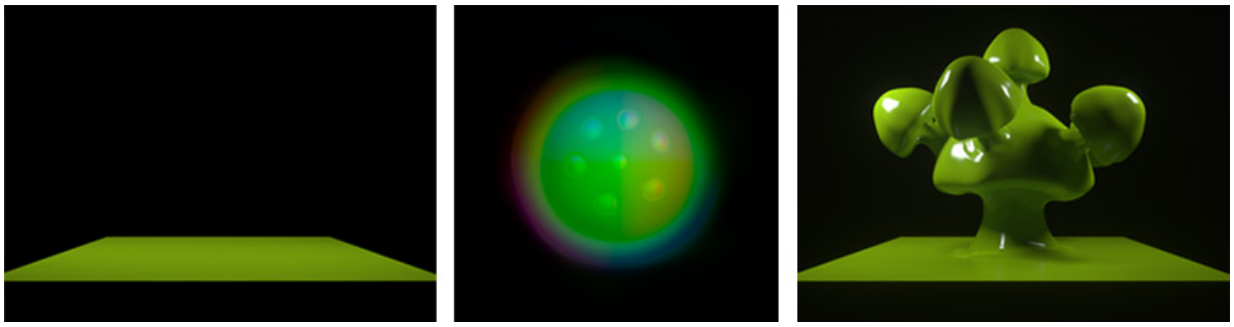


Рисунок 8 — Визуализация трехмерной модели с помощью векторного смещающего отображения на базовой поверхности, состоящей из одного полигона (по центру представлена векторная карта смещений).

Пусть для данной точки $\boldsymbol{P} = (P_x, P_y, P_z)$ известны $\boldsymbol{t}$, $\boldsymbol{n}$, $\boldsymbol{b}$ — нормальный, касательный и бикасательный векторы (единичной длины) соответственно. Тогда вычисление координат новой смещенной точки $\boldsymbol{P}'$ производится по следующей формуле (4):

$$\boldsymbol{P}' = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} + s \begin{bmatrix} t_x & b_x & n_x \\ t_y & b_y & n_y \\ t_z & b_z & n_z \end{bmatrix} \begin{bmatrix} d_r - o \\ d_g - o \\ d_b - o \end{bmatrix}, \quad (4)$$

где координаты по x, y, z векторов $\boldsymbol{t}$, $\boldsymbol{n}$, $\boldsymbol{b}$ заданы в объектном пространстве, $\boldsymbol{d} = (d_r, d_g, d_b)$ есть значение векторного смещения, полученное из текстуры, s — высота сдвига, o — значение текстуры, соответствующее нулевому изменению высоты. Для беззнаковых форматов o должно быть равным 0, а для знаковых — 0,5.

2. Разработка алгоритма трассировки лучей с учетом наложения карт смещения

Среди рассмотренных выше методов для реализации в данной работе был выбран алгоритм обхода лучом виртуальной сетки, описанный в работе [9].

Для его выполнения модель должна быть триангулирована. Ее элементы покрываются виртуальной сетью из микротреугольников, их вершины смещаются, после чего ищется пересечение луча с полученной поверхностью. Особенностью данного метода является система индексов для микротреугольников, которая позволяет определять те из них, над которыми проходит луч.

Каждая сторона исходного треугольника $\boldsymbol{P_0 P_1 P_2}$ разбивается на N частей (параметр разбиения). Границы ячеек проходят вдоль прямых, заданных фиксацией одной из барицентрических координат, кратных $1/N$. Тогда каждый полученный таким образом микротреугольник задается тройкой (i, j, k) индексов, заданных по правилу: $(i, j, k) = (\lfloor \alpha N \rfloor, \lfloor \beta N \rfloor, \lfloor \gamma N \rfloor)$, где (α, β, γ) есть барицентрические координаты любой точки, лежащей внутри него. Если сумма индексов i, j, k равна $N - 1$, будем называть такой микрополигон *нижним*, если $N - 2$ — *верхним* (см. рисунок 9).

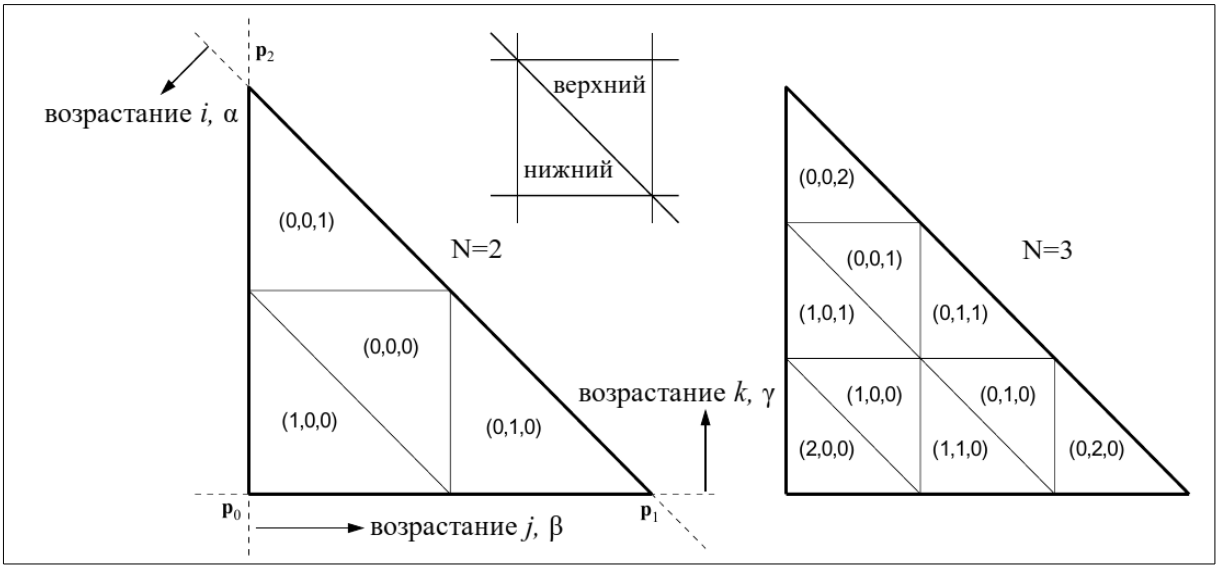


Рисунок 9 — Разбиение треугольника с параметрами $N = 2$ и $N = 3$.

Алгоритм разбивается на две стадии: стадию инициализации и сам обход сетки. В процессе обхода выполняется смещение текущего микротреугольника $\boldsymbol{abc}$, проверка на пересечение и переход к следующей ячейке. Для выбора следующей ячейки необходимо знать, с какой стороны луч «входит» и «выходит» из микротреугольника (см. рис. 10). Также необходимо хранить, какой индекс из тройки (i, j, k) был изменен и на какое значение (на +1 или -1). Для хранения этого параметра подходит множество поименованных констант *LastChange = {iplus, jminus, kplus, iminus, jplus, kminus}*.

Будем говорить, что *луч проходит справа (слева) от точки c* (точка на поверхности модели), если в формуле (5) значение *flag* больше (меньше) нуля.

$$flag = (\boldsymbol{n} \times (o - c)) \cdot \boldsymbol{v}, \qquad (5)$$

где c — координаты точки, $\boldsymbol{n}$ — нормаль к поверхности в этой точке, o — координаты начала луча, $\boldsymbol{v}$ — направление луча (рис. 10).

Вершины микротреугольника $\boldsymbol{abc}$ выбираются таким образом, чтобы луч «входил» в него со стороны $\boldsymbol{ab}$ (то есть проходил справа от точки $\boldsymbol{a}$ и слева от точки $\boldsymbol{b}$). Тогда следующая ячейка выбирается в зависимости от того, с какой стороны от вершины $\boldsymbol{c}$ проходит луч (см. рис. 10).

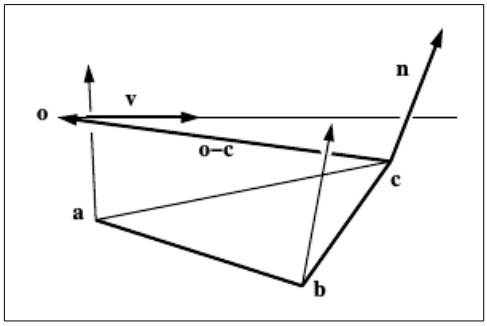


Рисунок 10 — Прохождение луча видимости над микротреугольником.

2.1 Стадия инициализации

На этом этапе вычисляются индекс стартовой ячейки (i, j, k), координаты вершин $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$ (с учетом того, что луч должен входить через сторону $\boldsymbol{ab}$) и параметр *change* из множества *LastChange*. Для этой цели

ищутся точки пересечения луча с ограничивающим объемом. Он получается в результате сдвига вершин вдоль своих нормалей на величину максимального смещения (см. рисунок 11).

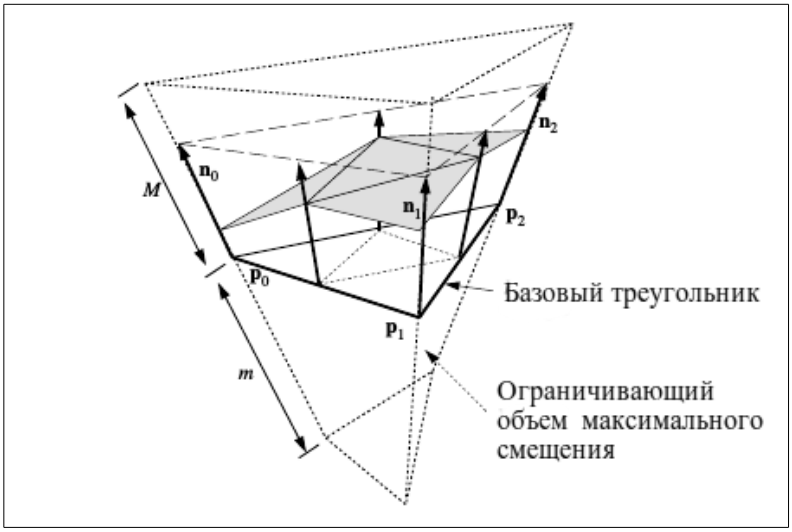


Рисунок 11 — Базовый треугольник и объем, заметаемый в процессе смещения.

Должны быть рассмотрены два случая: пересечение луча с торцевой треугольной крышкой и с боковой стороной. Последняя в общем случае представляет собой не плоскость, а билинейный патч (bilinear patch) (нормали, выходящие из разных концов одного ребра, могут не лежать в одной плоскости, см. рис. 11). Так как из-за искривленности боковой стенки луч может входить в ограничивающий объем более одного раза, алгоритм обхода должен быть запущен для каждой стартовой ячейки. Для этого по очереди проверяются каждая из пяти сторон (порядок проверки неважен):

1. Поиск пересечения с треугольной крышкой. Если пересечение найдено:
   1) найденная точка есть точка входа, если:
   - проверялась верхняя торцевая крышка и луч противоположно направлен с вектором $\boldsymbol{P_0P_1} \times \boldsymbol{P_0P_2}$ или
   - проверялась нижняя торцевая крышка и луч сонаправлен с вектором $\boldsymbol{P_0P_1} \times \boldsymbol{P_0P_2}$,

иначе перейти к следующей стороне.

2) $(i, j, k) := (\lfloor \alpha N \rfloor, \lfloor \beta N \rfloor, \lfloor \gamma N \rfloor)$, где (α, β, γ) — барицентрические координаты точки пересечения;

3) вычисляются мировые координаты вершин микротреугольника $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$; для каждой из сторон $\boldsymbol{AB}$, $\boldsymbol{BC}$, $\boldsymbol{CA}$ проверяется, пересекает ли ее луч, если да — это искомая сторона $\boldsymbol{ab}$ (оставшаяся точка становится вершиной $\boldsymbol{c}$); если не проходит ни через одну — порядок неважен;

4) параметр *change* устанавливается в зависимости от того, через какую сторону микротреугольника входит луч (см. рис. 9).

2. Поиск пересечения с боковой стенкой:

1) сторона треугольника разбивается на N частей $A_m B_m$, $m := 0 \ldots N-1$; если перебором находится такой m, что луч проходит справа от A_m и слева от B_m, то $\boldsymbol{a} := A_m$ и $\boldsymbol{b} := B_m$, иначе перейти к следующей стороне.

2) если такой отрезок найден и рассматривалась сторона:

- $\boldsymbol{P_0 P_1}$, тогда $(i, j, k) := (N-m-1,\ m,\ 0)$, *change* := *kplus;*
- $\boldsymbol{P_1 P_2}$, тогда $(i, j, k) := (0,\ N-m-1,\ m)$, *change* := *iplus;*
- $\boldsymbol{P_2 P_0}$, тогда $(i, j, k) := (m,\ 0,\ N-m-1)$, *change* := *jplus;*

3) на основе индексов (i, j, k) вычисляется вершина $\boldsymbol{c}$.

2.2 Обход виртуальной сетки

Если на стадии инициализации найден индекс стартовой ячейки $\boldsymbol{abc}$, от нее начинается обход виртуальной сетки по следующему алгоритму:

1. Если луч пересекает микротреугольник $\boldsymbol{abc}$, вернуть координаты точки пересечения и в качестве нормали — вектор $(\boldsymbol{b}-\boldsymbol{a}) \times (\boldsymbol{c}-\boldsymbol{a})$;
2. Вычислить *flag* по формуле (5).
3. Если *flag* > 0, $\boldsymbol{a} := \boldsymbol{c}$, иначе $\boldsymbol{b} := \boldsymbol{c}$.

4. Изменить параметр *change* из закольцованного списка $\{iplus, jminus, kplus, iminus, jplus, kminus\}$: если $flag > 0$ присвоить *change* значение справа от текущего, иначе слева (по формуле $change := (change + ((flag > 0)\ ?\ 1 : 5))\ mod\ 6$).
5. Найти *uvc* — барицентрические координаты вершины $\boldsymbol{c}$, $uvc := \{(u, v):\ (\alpha, \beta, \gamma) = (1-u-v,\ u,\ v)\}$:
   1) если $change = iminus$:
      - $i := i-1$, если $i < 0$ — выход за пределы треугольника, пересечения нет, выход;
      - $uvc := ((j+1)/N,\ (k+1)/N)$;
   2) иначе если $change = iplus$:
      - $i := i+1$, если $i \geq N$ — выход за пределы треугольника, пересечения нет, выход;
      - $uvc := (j/N,\ k/N)$;
   3) иначе если $change = jminus$:
      - $j := j-1$, если $j < 0$ — выход за пределы треугольника, пересечения нет, выход;
      - $uvc := (j/N,\ (k+1)/N)$;
   4) иначе если $change = jplus$:
      - $j := j+1$, если $j \geq N$ — выход за пределы треугольника, пересечения нет, выход;
      - $uvc := ((j+1)/N,\ k/N)$;
   5) иначе если $change = kminus$:
      - $k := k-1$, если $k < 0$ — выход за пределы треугольника, пересечения нет, выход;
      - $uvc := ((j+1)/N,\ k/N)$;
   6) иначе если $change = kplus$:

- $k := k+1$, если $k \geq N$ — выход за пределы треугольника, пересечения нет, выход;
- $uvc := (j/N,\ (k+1)/N)$;

7) получить координаты вершины $\boldsymbol{c}$ и нормали к ней, используя ее барицентрические координаты uvc внутри исходного треугольника.

6. Вернуться на шаг 1.

Выход из алгоритма происходит либо на шаге 1, если найдено пересечение, либо на шаге 5, когда луч выходит за пределы ограничивающего объема.

К достоинствам данного метода можно отнести отсутствие затрат памяти для хранения дополнительных вершин, а также необходимости вычислять частные производные к функции, задающей смещенную поверхность (по формуле (2) п.1.1) или решать уравнения, требующие вычисления алгебраических корней. Также алгоритм после триангуляции можно применять к поверхностям, заданным в любом виде. После этого для предотвращения разрывов между треугольниками интерполируются нормали внутри базового полигона (интерполирование Фонга, *Phong interpolation*). К недостаткам метода относится большое количество вычислений, а также проблемы точности.

3. Реализация разработанного алгоритма с использованием технологии CUDA

3.1 Трассировка лучей на GPU и использование технологии CUDA

CUDA (Compute Unified DeviceArchitecture) — программно-аппаратная платформа, предложенная компанией Nvidia для увеличения производительности приложений за счет использования графических процессоров (GPU, ГПУ) [11, 12].

Применительно к задаче трассировки лучей параллельные вычисления могут быть использованы следующим образом. Трассировка лучей представляет собой процесс создания виртуального луча (первичный луч), выходящего из точки наблюдения (позиции камеры), через каждый пиксель экрана. После этого ищется точка ближайшего пересечения луча со сценой, откуда строится луч в сторону источника света (вторичный луч), и выясняется, освещается ли она или находится в тени от других объектов. На основе этого вычисляется цвет точки и устанавливается соответствующий цвет пиксела в буфере кадра. Существуют алгоритмы, строящие отраженные лучи и учитывающие свет, который попал в точку не напрямую из источника света, а при отражении от других объектов сцены [1, с. 412-413]. Но такой метод сильно увеличил бы время построения кадра, не принося существенных изменений в принцип построения изображения, поэтому в приложении был реализован классический вариант, не учитывающий отраженный свет.

Так как для каждого луча трассировка производится независимо от остальных, *эта задача может быть эффективно выполнена с помощью параллельных вычислений*, если обрабатывать каждую точку экрана в отдельном потоке. Эти вычисления переносятся на графический процессор с помощью технологии CUDA .

3.2 Реализация приложения и основные структуры данных

При считывании модель разбивается на треугольники и заносится в массив, который копируется на GPU (см. Листинг 1).

Листинг 1 — Представление треугольника.

```
struct Triangle {
    float3 p0, p1, p2; //Координаты вершин: стандартный обход против часовой стрелки
    float3 n0, n1, n2; //Нормали
    float2 uv0, uv1, uv2; //Текстурные координаты

};
```

Перед копированием вершины треугольников раздвигаются на малую величину (10^{-6} единичного вектора) от центра. Это не влияет на вид модели, но предотвращает выпадение пикселей на границах треугольников.

После копирования всех данных на GPU запускается цикл отрисовки кадров. Каждый пиксел буфера кадра обрабатывается в отдельном потоке на графическом процессоре. По индексу потока вычисляются координаты пиксела сначала в оконных, потом в мировых координатах. Строится луч из точки нахождения камеры через данный пиксел и выполняется его трассировка. Однако она выполняется достаточно долго, если пересечение с моделью ищется полным перебором треугольников в массиве. Возникает критическая проблема для высокополигональных моделей: если нить выполняется дольше определенного количества времени, установленного драйвером видеокарты, он прекращает ее выполнение. В разделе поддержки на сайте Nvidia [13] в качестве решения предложено отключить систему X Windows либо сократить время выполнения потока до 0,1 сек. В реализованном приложении для решения этой проблемы были выполнены две оптимизации:

1. Введена ускоряющая структура данных — дерево ограничивающих оболочек HLBVH [14]. Каждый его листовой элемент содержит оболочку области, полученной смещением треугольника вдоль нормалей на величину смещения. (Дерево также строится на GPU с использованием

параллельных вычислений технологии CUDA.) Трассировщик лучей вначале проходит по дереву ограничивающих оболочек, отбрасывая элементы, с которыми луч заведомо не может пересечься. После этого для каждого из оставшихся треугольников запускается алгоритм поиска пересечения с учетом наложения карты смещения, описанный в разделе 2.

2. Изображение строится не целиком, а по частям: при меньшем количестве потоков ядро завершается быстрее, что уменьшает риск превышения времени работы.

Для улучшения качества изображения и уменьшения артефактов, вызванных особенностями алгоритма наложения карты смещений, предусмотрена возможность включить сглаживание (раскомментировав строку «#define SMOOTH» в заголовочном файле test.h перед компиляцией). Сглаживание реализовано с помощью вероятностного метода (*stochastic sampling*) [1, с. 131]. Каждый кадр луч выпускается из камеры не в центр пиксела экрана, а в случайную точку внутри него. Полученное в результате трассировки значение прибавляется к аккумулирующему буферу, после чего суммарное значение делится на количество построенных кадров, давая среднее значение цвета в пределах одного пиксела. Качество изображения улучшается с каждым кадром, в какой-то момент вклад нового значения станет минимальным и изображение стабилизируется. При перемещении камеры аккумулирующий буфер заполняется нулевыми значениями, а счетчик кадров устанавливается в единицу. Вычисление среднего арифметического значения для каждого пиксела также выполняется параллельно на GPU. Случайные значения для сдвига получаются с помощью модуля cuRand из CUDA Toolkit 8.0 и для каждого пиксела хранятся в массиве gpu_buffs.rand_buff, который передается ядру.

3.3 Выбор средств программирования

Приложение реализовано на языке C++ (стандарт ISO/IEC 14882:2011), так как этот язык позволяет написать приложение с высокой скоростью работы и эффективным использованием аппаратных ресурсов и при этом поддерживается платформой CUDA.

Использовались следующие сторонние библиотеки:

1) Assimp 3.3.1 — для считывания модели из файла и триангуляции [15];
2) Класс QImage библиотеки Qt5 для считывания файлов, содержащих текстуры [16].
3) CUB — для числовой сортировки и префиксного сканирования на GPU при построении дерева ограничивающих оболочек для ускорения поиска видимого участка поверхности [17];
4) GLFW3 и GLEW — для вывода построенного изображения на экран [18, 19];
5) GLM — для матричных вычислений при переходе между различными системами координат [20];
6) CUDA Toolkit 8.0 — для реализации параллельных вычислений на графическом процессоре [21].

3.4 Запуск и использование приложения

В рамках данного курсового проекта было реализовано приложение для отображения трехмерных моделей с наложением поверхностных смещений, хранимых в двумерных текстурах в виде карты высот.

Системные требования:

- графическая карта NVIDIA, поддерживающая архитектуру CUDA с compute_capability 2.0,
- дистрибутив операционной системы GNU/Linux,
- компилятор gcc версии не ниже 4.8,
- утилита cmake версии не ниже 3.0.0.

Аргументы командной строки:

1. Путь к файлу в формате OBJ, содержащему модель, в которой должны быть заданы текстурные координаты. В случае возникновения ошибки в консоль выводится сообщение вида «[readObjFile]: <Сообщение об ошибке, генерируемое классом Importer библиотеки Assimp>». В случае успешного считывания в консоль выводится сообщение «[main]: Importing model SUCCESS. Imported <количество загруженных треугольников> triangles».
2. Путь к файлу изображения с текстурой для наложения цвета. Изображения могут иметь любые из наиболее распространенных форматов (например, JPEG, PNG, BMP), считывание производится классом QImage библиотеки Qt. О результате считывания также выводится сообщение в консоль.
3. Путь к файлу изображения с текстурой, содержащей карту высот. Ограничений на входной формат так же как в п. 2) определяются классом QImage. Если изображение монохромное, значение для сдвига получается из текстуры непосредственно, иначе берется значение яркости пиксела.
4. Число, определяющее нулевой сдвиг (float shift). Это число будет вычтено из каждого тексела при загрузке текстуры. Для текстур, где отсутствию сдвига соответствует черный цвет, это значение должно равняться нулю (в карте смещений будут хранится значения из диапазона [0; 1]; если серый цвет — то 0,5, тогда диапазон возможных смещений соответственно [-0,5; 0,5]). Этот параметр может принимать и другие значения в зависимости от желаемого эффекта.
5. Растяжение (float scale). На эту величину будет домножен каждый тексел на этапе считывания (таким образом диапазон значений возможных сдвигов изменяется от [0; 1] до [-shift*scale; (1-shift)*scale]). Чем больше это значение, тем более выпуклым будет рельеф.

6. Параметр разбиения *N* при обходе виртуальной полигональной сетки. Параметр следует подбирать в зависимости от модели и желаемого качества изображения.

Если указаны не все параметры (возможно не указать ни одного) — вместо них будут использоваться значения, установленные до компиляции. Если тип данных аргумента не соответствует ожидаемому — приложение выводит в консоль подсказку и завершается.

После запуска появляется окно, в которое выводится полученное изображение. Происходит захват мыши окном и после этого каждое ее движение воспринимается как поворот камеры в соответствующем направлении. Для движения камеры в стороны используются клавиши A (влево), S (назад), D (вправо), W (влево). Закрытие окна осуществляется нажатием клавиши Esc.

4. Тестирование приложения

Целью тестирования является демонстрация работоспособности и производительности приложения.

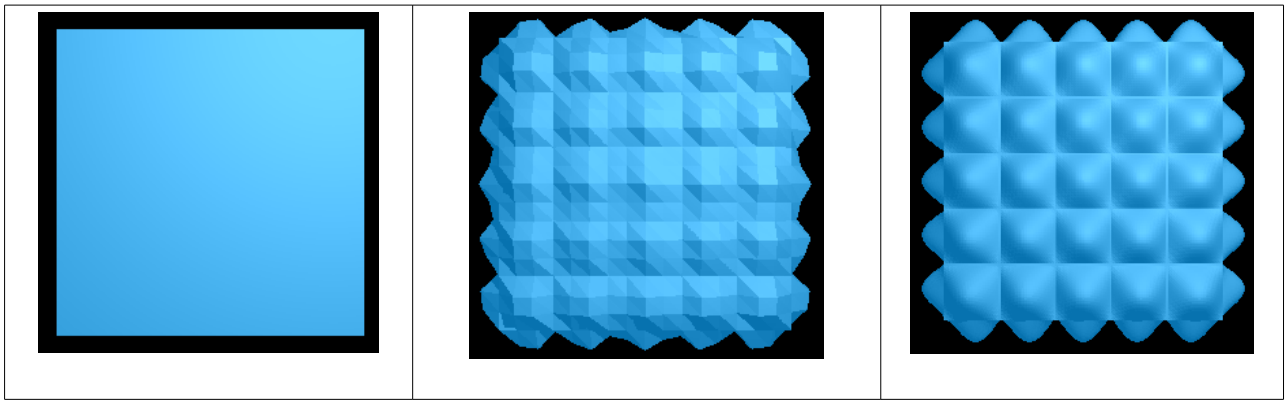
Тесты проводились на графической карте NVIDIA GeForce GT 620M.

4.1 Наложение процедурной текстуры

Модель: куб. Количество треугольников: 12.

Текстура процедурная. Для текстурных координат u и v в диапазоне $[0;1]$ используется формула $|\sin(5*u*\pi)\sin(5*v*\pi)|/0{,}4$.

Результат наложения представлен на рис. 12.



а) б) в)

Рисунок 12 — Наложение процедурной текстуры: исходная модель (а), смещение с параметом разбиения $N = 16$ (б) и $N = 128$ (в).

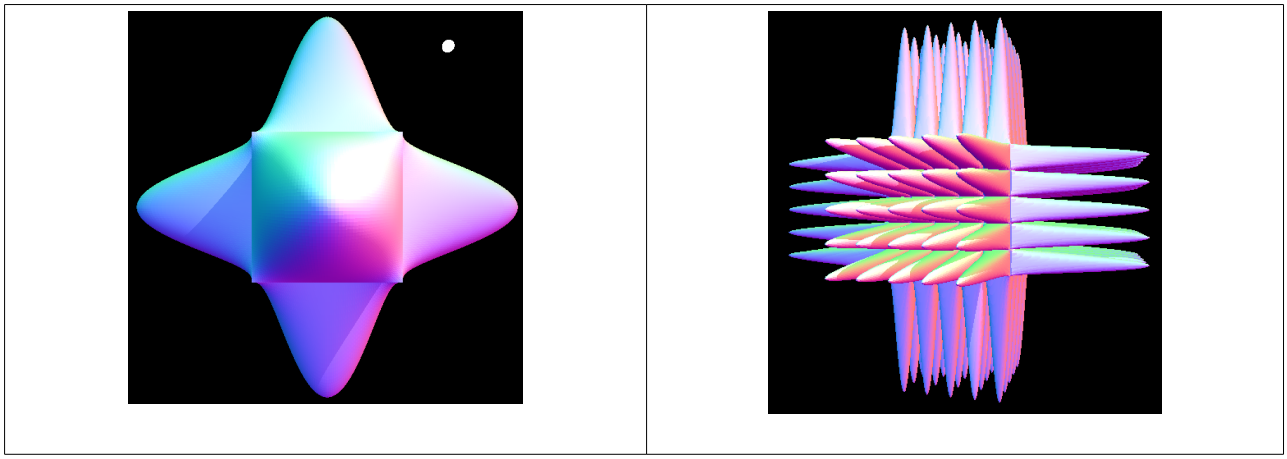
Производительность без наложения смещений: 39 кадров/с.

Замеры производительности с различными параметрами представлена в таблице 1.

Таблица 1 — Производительность при отрисовке куба.

Параметр разбиения N	Производительность, кадров/с
1	32
16	10
128	5

На рисунке 13 представлен результат работы программы, в качестве цвета используется направление нормали в данной точке (параметр $N = 128$).



а) б)

Рисунок 13 — Наложение на куб процедурной текстуры, заданной формулой $|\sin(u*\pi)\sin(v*\pi)|$ и $|\sin(5*u*\pi)\sin(5*v*\pi)|$.

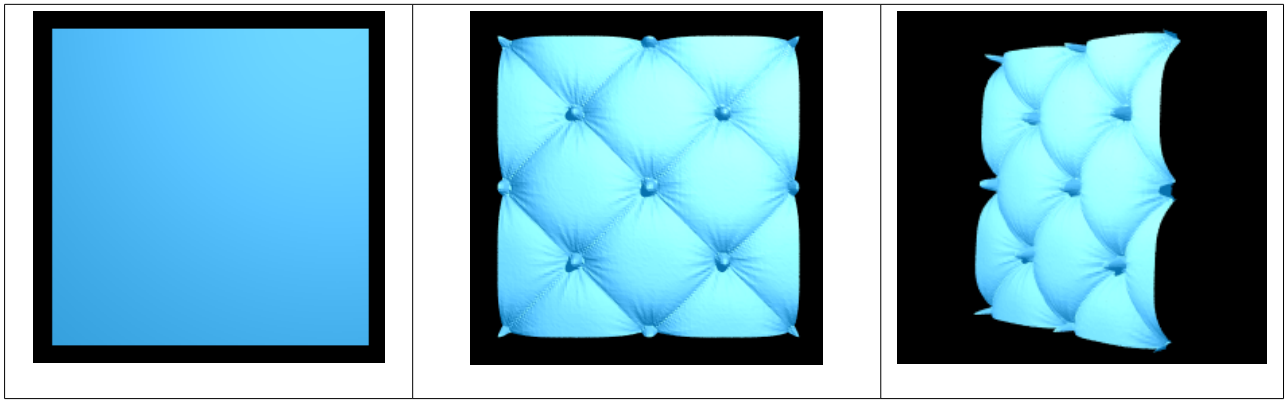
На изображении видно, что алгоритм позволяет добавлять к модели детали, сравнимые по размеру с ней.

4.2 Изменение формы плоскости

Модель: квадрат. Количество треугольников: 2.

Текстура: chesterfield-height.jpg (http://romain.vergne.free.fr).

Результат наложения представлен на рис. 14.



а) б) в)

Рисунок 14 — Наложение карты смещений:

исходная модель (а), результат наложения текстуры (б), вид сбоку (в).

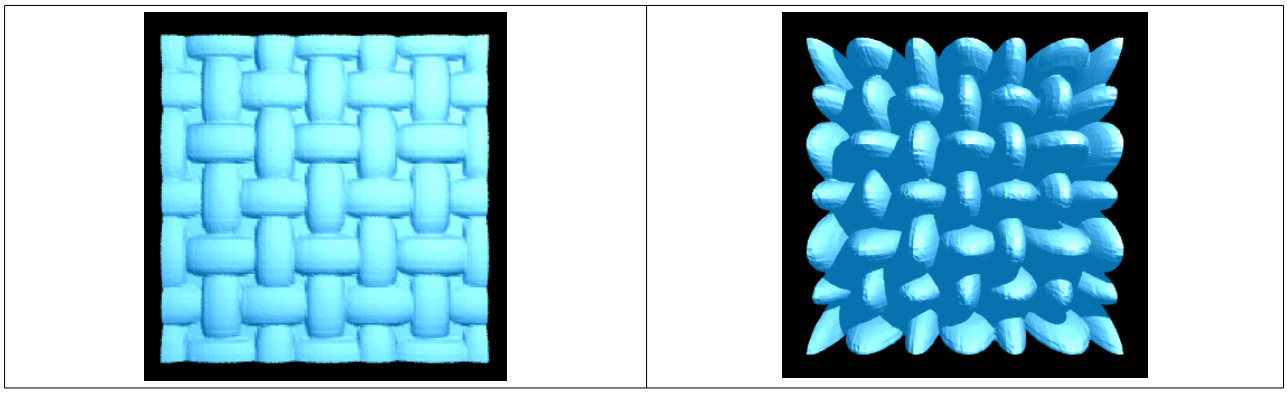
Производительность без наложения смещений: 40 кадров/с.

Замеры производительности с различными параметрами представлена в таблице 2.

Таблица 2 — Производительность при отрисовке квадрата.	
Параметр разбиения *N*	Производительность, кадров/с.
1	34
16	23
128	8

Текстура: 10.jpg (http://blender3d.org.ua).

Результат изменения параметра растяжения *scale* для значений из карты высот показан на рисунке 15 (параметр разбиения $N = 128$).



а) б)

Рисунок 15 — Растяжение текстуры: *scale* = 0,07 (а), *scale* = 0,7 (б).

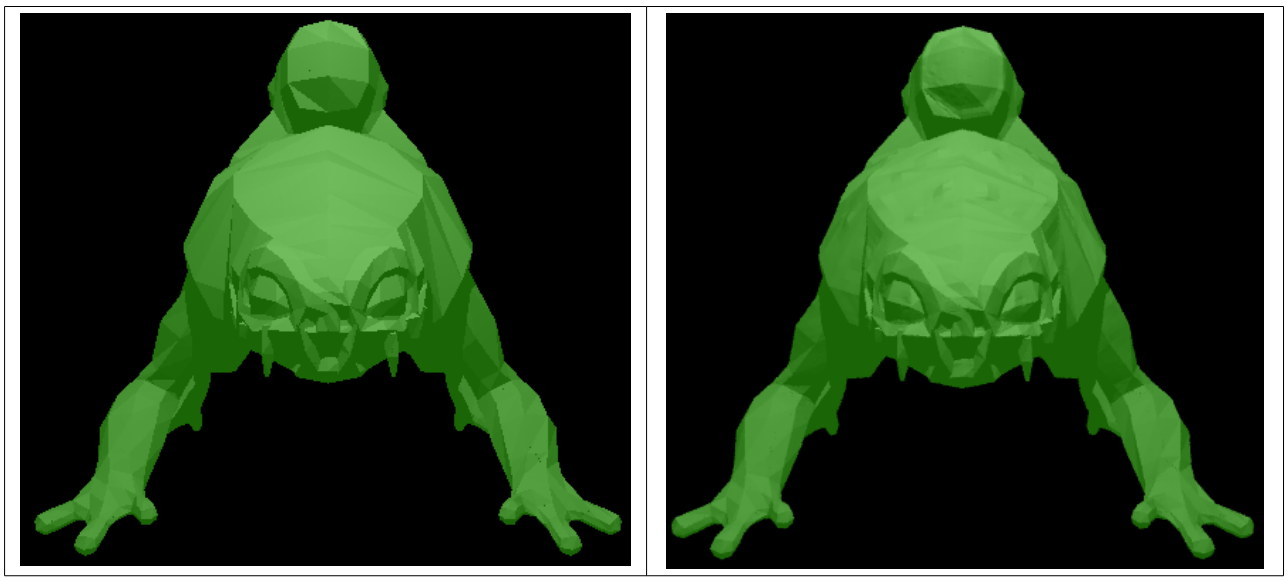
Тесты с плоскостью демонстрируют, насколько сильную деформацию поверхности можно произвести с помощью данного метода.

4.3 Высокополигональная модель

Модель и текстура: Monsterfrog (NVIDIA).

Количество треугольников: 2584.

Исходная модель и измененная представлены на рисунке 16.



а) б)

Рисунок 16 — Наложение карты смещений:
исходная модель (а), результат(б).

Производительность без наложения смещений: 14 кадров/с.

Замеры производительности с различными параметрами представлена в таблице 3.

Таблица 3 — Производительность при отрисовке квадрата.

Параметр разбиения N	Производительность, кадров/с.
4	3
8	2

На рисунке 17 видно влияние параметра разбиения и наличия сглаживания на количество выпадающих пикселей.

а) б) в)

Рисунок 17 — Результат изменения параметров:
$N = 4$ без сглаживания (а), $N = 8$ без сглаживания (б),
$N = 8$ со сглаживанием (в).

Данный тест показывает, что в некоторых случаях луч проходит мимо необходимой точки пересечения. Это происходит на границе полигонов из-за ограничений точности арифметики с плавающей запятой либо в силу особенности алгоритма. При смещении точек микротреугольника вдоль нормалей получается объем с билинейным патчем в качестве боковой стенки так же, как при смещении вершин исходного треугольника. И луч также может пересечь этот патч дважды. Для исходного треугольника этот случай обрабатывается в стадии инициализации (см. п. 2.1), а при обходе виртуальной сетки такая возможность не учитывается. Этот случай описан в оригинальной статье [9], и в качестве решения авторами предлагается увеличить параметр разбиения.

В данном тесте луч проходит между исходными треугольниками. Увеличение параметра разбиения, а так же применение стохастического сглаживания позволяют свести такие ошибки к минимуму.

Заключение

В рамках данной работы были выполнены следующие задачи: изучены различные методы формирования рельефа поверхности, был разработан и реализован алгоритм трассировки лучей, позволяющий создавать рельеф поверхности методом наложения карт смещения с использованием технологии параллельного программирования CUDA, а также выполнено тестирование приложения.

Данное приложение позволяет визуализировать трехмерные модели, накладывать на них цветовые текстуры и карты смещений, регулировать величину смещения, а также параметр разбиения для создания смещенных микротреугольников.

Для улучшения качества изображения могут быть сделаны следующие модификации программы:

1) динамическое вычисление параметра разбиения в зависимости от расстояния от камеры до треугольника так, чтобы площадь проекции на экран микротреугольника была меньше одного пиксела;
2) использование после наложения смещения одного из методов преобразования нормалей при небольшом параметре разбиения.

Список использованных источников

[1] Akenine-Möller T., Haines E., Hoffman N. Real-Time Rendering, Third Edition. — CRC Press, 2008. — 1045 p.

[2] Buss. S. 3D Computer Graphics: A Mathematical Introduction with OpenGL. — Cambridge University Press, 2003. — 371 p.

[3] Kaneko T., Takahei T., Inami M., Kawakami N., Yanagida Ya., Maeda T., Tachi S. Detailed Shape Representation with Parallax Mapping // ICAT 2001. December 5-7, Tokyo, Japan. 4 p.

[4] Tatarchuk N. Practical Parallax Occlusion Mapping with Approximate Soft Shadows for Detailed Surface Rendering. — Advanced Real-Time Rendering in 3D Graphics and Games, SIGGRAPH 2006. Chapter 5. — 32 p.

[5] Birn J. Digital Lighting and Rendering. — Pearson Education, 2013. — 464 p.

[6] Pharr M., Fernando R. GPU Gems 2: Programming Techniques For High-Performance Graphics And General-Purpose Computation. — Pearson Addison Wesley Prof, 2005. — 814 p.

[7] Szirmay-Kalos L., Umenhoffer T. Displacement Mapping on the GPU — State of the Art. The Eurographics Association and Blackwell Publishing 2008. — 24 p.

[8] Fourquet E., Cowan W., Mann S. Geometric Displacement on Plane and Sphere. — Proceedings of Graphics Interface 2008. — 193-202 p.

[9] Smits S., Shirley P., Stark M. Direct Ray Tracing of Smoothed and Displaced Mapped Triangles. — Eurographics Rendering Workshop, 2000. — 8 p.

[10] Volovar M. Rendering High Detail Models from Displacement Maps. — Proceedings of CESCG 2016: The 20th Central European Seminar on Computer Graphics. — 8 p.

[11] CUDA Toolkit Documentation. Programming Guide. Режим доступа: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (дата обращения 30.01.2017).

[12] Боресков А. В., Харламов А. А. Основы работы с технологией CUDA. — М.: ДМК Пресс, 2010. — 232 с.: ил. ISBN 978-5-94074-578-5.

[13] Using CUDA and X. Режим доступа: http://nvidia.custhelp.com/app/answers/detail/a_id/3029/~/using-cuda-and-x (дата обращения 30.01.2017).

[14] Garanzha K., Pantaleoni J., McAllister D. Simpler and faster hlbvh with work queues. — 2011. Режим доступа: https://research.nvidia.com/sites/default/files/publications/main.pdf (дата обращения 30.01.2017).

[15] Open Asset Import Library. Режим доступа: http://www.assimp.org/ (дата обращения 30.01.2017).

[16] Qt Documentation. Qimage class. Режим доступа: http://doc.qt.io/qt-5/qimage.html (дата обращения 30.01.2017).

[17] CUB: main page. Режим доступа: https://nvlabs.github.io/cub/ (дата обращения 30.01.2017).

[18] GLFW — An OpenGL Library. Режим доступа: http://www.glfw.org/ (дата обращения 30.01.2017).

[19] GLEW: The OpenGL Extension Wrangler Library. Режим доступа: http://glew.sourceforge.net/ (дата обращения 30.01.2017).

[20] OpenGL Mathematics. Режим доступа: http://glm.g-truc.net/0.9.8/index.html (дата обращения 30.01.2017).

[21] NVIDIA Developer. CUDA Toolkit. Режим доступа: https://developer.nvidia.com/cuda-toolkit (дата обращения 30.01.2017).