Оглавление

Введение

Современное компьютерное моделирование и визуализация требуют детального и реалистичного представления трехмерных объектов. Особенное внимание уделяется методам, позволяющим достичь высокой степени реалистичности без заметного увеличения вычислительной сложности или объема данных. Одним из таких методов является учет текстуры на поверхности тел путем внесения возмущений в нормаль.

Текстурирование позволяет не только изменять внешний вид объекта, но и модифицировать его геометрические и световые характеристики. С помощью текстур можно создавать реалистичные детали поверхности, такие как мельчайшие неровности, шероховатости и даже более сложные элементы, такие как: камни, узоры и рельефы, не добавляя при этом дополнительных геометрических объектов в сцену.

Цель работы – разработать программное обеспечения для учета текстуры на поверхности трехмерных тел с использованием метода внесения возмущения в нормаль.

Для достижения поставленной цели требуется решить следующие задачи:

1) Определить, какие объекты будут располагаться в сцене;

2) Выбрать алгоритмы для построения и обработки трехмерных объектов. Это включает в себя методы удаления невидимых поверхностей, создания теней и отражений;

3) Определить модель освещения, которая будет использоваться для создания реалистичных световых эффектов;

4) Спроектировать структуру программного обеспечения и выбрать подходящий способ представления данных;

5) Выбрать средства реализации алгоритмов;

6) Создать программное обеспечения, реализовав в нем выбранные алгоритмы;

7) Исследовать различные способы текстурирования, а также их временные характеристики на основе созданного программного обеспечения.

1 Аналитическая часть

В данном разделе производится формализация объектов сцены, анализ алгоритмов их визуализации, и выбираются наиболее подходящие для решения поставленной задачи.

1.1 Текстура, как характеристика поверхности трехмерного тела

В компьютерной графике текстурой называется детализация структуры поверхности. Текстура – это одномерное или двумерное изображение, которое имеет множество ассоциированных с ним параметров, определяющих, каким образом производится наложение изображения на поверхность.

Обычно рассматривается два вида детализации. Первый состоит в том, чтобы на гладкую поверхность нанести заранее заданный узор – так называемая детализация светом. После этого поверхность все равно остается гладкой. Второй тип детализации заключается в создании неровностей на поверхности, реализуется путем внесения возмущений в параметры поверхности – детализация фактурой. [1]

1.2 Описание метода внесения возмущений в нормаль

В этом методе для создания неровностей отдельно рассматривается каждый пиксел текстуры – текстел. Реалистичность изображения достигается за счет создания рельефа, а каждый текстел должен хранит информацию о возмущении в нормаль, которое требуется внести.

1.3 Текстурные карты

Для хранения информации о текстелях используют так называемые текстурные карты. Для метода внесения возмущений в нормаль чаще всего используют следующие:

- Карта высот (англ. height map). Каждый пиксел карты хранит одно значение. Поэтому изображения являются черно-белым, где белый – самые выпуклые точки на поверхности;
- Карта нормалей (англ. normal map). Представлены в виде RGB-изображений где каналы R и G описывают наклон каждой нормали;
- Карта параллактического отображения (англ. parallax map). Эта карта использует особый визуальный эффект, чтобы создать иллюзию глубины и объемности. Она изменяет положение текстурных координат в зависимости от уровня смещения, создавая ощущение сложной геометрии на плоской поверхности.

1.4 Представление объектов

Для решения данной задачи будем использовать сферы. Такой простой тип объектов позволит хорошо рассмотреть наложение текстур на поверхность, а так же позволит сильно сократить время работы программы, что очень важно, поскольку мы хотим достичь максимальной детализации используя вычислительные возможности только процессора. Это дает нам возможность выбрать более затратные по времени алгоритмы, отдав приоритет визуальной составляющей результата.

1.5 Анализ и выбор алгоритма удаления невидимых линий и поверхностей

Выделим самый важный критерий, которым должен обладать выбранный алгоритм:

- Получение высокореалистичных изображений. Для успешного наложения текстур важно учитывать даже небольшие изменения в освещении, тенях и других световых эффектах.

1.5.1 Алгоритм буфера глубины

Одним из самых распространенных алгоритмов удаления невидимых поверхностей является алгоритм буфера глубины (англ. Z-буфера). Здесь буфер глубины представляет собой дополнительное пространство в памяти, где для каждого пиксела хранится значение глубины объектов перед ним (расстояние от наблюдателя до поверхности объекта). Задача алгоритма заключается в нахождении минимального значения глубины z для каждого пиксела экрана. [2, 3]

Принцип работы заключается в следующем: сначала весь буфер глубины заполняется значениями, соответствующими максимальной глубине. В процессе растеризации для каждого пиксела рассчитывается его глубина и сравнивается с текущим значением в буфере глубины. Если рассматриваемый пиксел находится ближе, он рисуется, и его глубина заменяет значение в буфере. Если пиксел дальше, он не рисуется, и буфер остается неизменным – это позволяет отбросить невидимые поверхности.

Преимущества алгоритма:

- Простота реализации;
- Высокая скорость работы;
- Применим к различным типам объектов.

Недостатки:

- Производительность может снижаться при обработке сложных сцен;
- Могут возникнуть проблемы с отображением прозрачных объектов;
- Требуется значительный объем памяти.

1.5.2 Алгоритм Робертса

Алгоритм Робертса представляет собой первое известное решение задачи об удалении невидимых линий. Это математический метод, работающий в объектном пространстве. [2]

В алгоритме Робертса требуется, чтобы все изображаемые тела или объекты были выпуклыми. Невыпуклые тела должны быть разбиты на выпуклые части. Выпуклое многогранное тело с плоскими гранями должно представляться набором пересекающихся плоскостей.

Алгоритм Робертса включает в себя три этапа:

- На первом этапе каждое тело анализируется индивидуально с целью удаления нелицевых плоскостей;
- На втором этапе проверяется экранирование оставшихся в каждом теле ребер всеми другими телами с целью обнаружения невидимых отрезков;
- На третьем этапе вычисляются отрезки, которые образуют новые ребра при протыкании телами друг друга.

Преимущества алгоритма:

- Высокая точность изображения на выходе.

Недостатки:

- Сложность реализации;
- Высокий рост сложности алгоритма при увеличении числа объектов;
- Необходимость разбивать невыпуклые объекты на выпуклые, что может значительно замедлить выполнение программы.

1.5.3 Алгоритм обратной трассировки лучей

Методы трассировки лучей считаются наиболее мощными и универсальными методами создания реалистичных изображений. Существует множество успешных реализаций алгоритмов трассировки для отображения даже самых сложных трехмерных сцен. [2, 3]

Суть алгоритма заключается в том, что из источника наблюдения проводится луч в каждую точку картинной плоскости. Анализируя траекторию луча, мы можем определить, какие объекты он пересекает и от каких отражается. Это подобно тому, как человеческий глаз воспринимает световые лучи, но в обратном порядке.

При практической реализации метода обратной трассировки вводят ограничения. Вот некоторые из них:

1) Среди типов объектов выделяются источники света, они могут только излучать свет;

2) Свойства отражающих поверхностей описываются суммой двух компонент — диффузной и зеркальной;

3) Зеркальность разбивается на две составляющие: отражение от других объектов (кроме источников света) и световые блики от источников;

4) При диффузном отражении учитываются только лучи от источников света. Точки, находящиеся в тени, определяются тем, если луч на источник света блокируется другим объектом;

5) Для прозрачных объектов часто упрощается моделирование преломления: без учета зависимости от длины волны. Прозрачность может также моделироваться без преломления, когда направление преломленного луча совпадает с направлением падающего луча.

Преимущества алгоритма:

- Высокая степень реалистичности полученных изображений;

- Вычислительная сложность не сильно зависит от количества объектов на сцене.

Недостатки:

- Ограничения в производительности из-за интенсивного вычислительного процесса.

Алгоритм обратной трассировки лучей является предпочтительным для использования в задачах внесения возмущений в нормали и текстурной модификации по следующим причинам:

- Алгоритм обратной трассировки лучей уже включает в себя вычисление нормалей для каждой точки поверхности. Это означает, что мы можем легко внести дополнительные изменения в нормали, соответствующие текстуре;

- Алгоритм обратной трассировки лучей позволяет точно моделировать освещение, что имеет ключевое значение при работе с текстурными изменениями. Этот метод обеспечивает более реалистичные эффекты теней, подсветки и отражений, учитывая изменения, внесенные в нормали;

- В данной задаче акцент делается на визуальное воздействие, а не на обработку больших объемов данных в кратчайшие сроки. Хотя алгоритм обратной трассировки лучей может потреблять больше ресурсов, он предоставляет значительно более качественное визуальное представление сцены.

1.6 Выбор модели освещения

При текстурировании в трассировке лучей, предпочтительной моделью освещения является модель Фонга. Эта модель выделяется своей способностью учесть разнообразные характеристики поверхностей в зависимости от направления света и точки наблюдения. Такой выбор модели обеспечивает создание более реалистичных текстур на объектах.

Модель освещения Фонга включает как диффузную, так и зеркальную компоненты освещения. Диффузная составляющая позволяет текстуре равномерно рассеивать свет в разных направлениях, что особенно важно для отображения матовых и неровных поверхностей. Зеркальная составляющая создает блеск и отражения на более гладких участках поверхности, что позволяет подчеркнуть детали текстуры.

Вывод

При процессе текстурирования с внесением изменений в нормали, главной целью является достижение высокой степени реалистичности.

Применяя текстурные карты, необходимо использовать объемные модели, чтобы обеспечить правильное расположение текстуры на поверхности.

Выбор алгоритма обратной трассировки лучей обусловлен его способностью создавать более точное и реалистичное изображение поверхности, сохраняя детали и освещение. Отличным дополнением стала модель освещения Фонга, которая добавит еще больше реалистичности и позволит наиболее точно проанализировать наложение текстур.

Важно отметить, что благодаря такому подбору методов, при трассировке лучей мы сразу же учитываем отражающие и преломляющие способности поверхности, а также сразу можем накладывать текстурные карты. Это предоставляет возможность в несколько раз ускорить работу программы: нет необходимости производить вычисления поэтапно, анализируя одну и ту же поверхность несколько раз.

2 Конструкторская часть

В данном разделе будут более подробно рассмотрены выбранные в предыдущем разделе методы и алгоритмы, а также предоставлены требования к программному обеспечению.

2.1 Требования к программному обеспечению

- Возможность отображать заданные сферы с учетом текстуры;
- Поддержка различных типов источников света: точечные, направленные и рассеянные;
- Учет теней, света и отражений;
- Поддержка различных текстурных карт: высот, нормалей, параллактического отображения.

2.2 Общий алгоритм решения задачи

- Разместить источники света: определить их позиции и характеристики (типы и интенсивность);
- Добавить сферы на сцену;
- Загрузить для каждой сферы текстурную карту;
- Трассировка лучей и визуализация:
    - Для каждой точки картинной плоскости сгенерировать луч, исходящий из источника наблюдения;
    - Проанализировать путь луча через сцену, учесть взаимодействия с поверхностью ближайшей сферы;
    - Внести изменения в нормали сферы с учетом текстурных карт;
    - Расчитать освещенность, тени и отражения на каждой точке поверхности сферы;

- Сформировать окончательное изображение на экране с учетом всех эффектов и модификаций.

2.3 Трассировка лучей

В реальном мире свет исходит от источников освещения, отражается от нескольких объектов и достигает наших глаз. Процесс моделирования пути света называется трассировкой лучей. В компьютерной графике же используются алгоритмы обратной трассировки. Основная идея заключается в анализе путей лучей, исходящих не от источников света, а от “глаз”, смотрящих на предметы. Такой подход соответствует всем привычным нам законам физики, при этом является реальным для реализации, в отличие от симуляции фотонов. [4]

2.3.1 Свет

Для наилучшего отображения реального освещения определим три типа источников света:

- Точечные источники света: испускают его равномерно во всех направлениях из определенной точки в пространстве;

- Направленные источники света: имеют фиксированное направление света, их можно рассматривать как бесконечно удаленные точечные источники, расположенные в заданном направлении;

- Рассеянные источники света: привносят часть освещения в каждую точку сцены, независимо от ее положения. Упрощает визуализацию реальной модели, когда свет, достигнув объекта, рассеивает часть обратно в сцену.

Таким образом, на сцене мы определим несколько источников света: рассеянные, точечные и направленные.

2.3.2 Наблюдатель и сцена

Первое, что определяется на сцене – положение наблюдателя и окна просмотра. Введем обозначения. Пусть точка $O = (O_x, O_y, O_z)$ – позиция камеры. Ширина и высота окна просмотра - V_w, V_h соответственно. Так же определим расстояние от точки O до плоскости окна – d, количество пикселей в окне приложения – C_w (по ширине), C_h (по высоте) и координаты пиксела на холсте - C_x, C_y. Для перехода от координат холста (C_x, C_y) к пространственным координатам (V_x, V_y, V_z) необходимо будет выполнить следующие преобразования:

$$V_x = C_x * \frac{V_w}{C_w}; \qquad V_x = C_y * \frac{V_h}{C_h}; \qquad V_z = d. \tag{2.1}$$

Итак, для каждой точки холста мы можем определить соответствующую ему точку в окне просмотра.

2.3.3 Лучи

Когда мы определили источник лучей (т. O), и их направления (V), можно задать луч (P). Удобнее всего это сделать с помощью параметрического уравнения:

$$P = O + t(V - O), \tag{2.2}$$

где t – любое неотрицательное вещественное число. Обозначим направление луча как: $\vec{D} = (V - O)$. Тогда:

$$P = O + t\vec{D}. \tag{2.3}$$

2.3.4 Сферы

Так как трассировка лучей – трудоемкий процесс, чаще всего вокруг всех объектов описываются сферы, и проверяется пересечение лучей сначала для

них. Таким образом, можно исключить множество ненужных вычислений, и выиграть во времени выполнении программы. Так же, чтобы хранить полную информацию о сфере не нужно так много памяти. Даже не нужно определять сторону для текстуры, поскольку ею будет всегда лишь единственная видимая.

Определим имеющиеся данные о сфере: радиус r, и центр S. Теперь эти данные необходимо преобразовать, чтобы с ними было удобнее работать. Пусть точка на поверхности сферы – T. По определению сферы:

$$\rho(T, S) = r, \tag{2.4}$$

где ρ – расстояние. Оно равно длине вектора:

$$|T - S| = r. \tag{2.5}$$

Длина вектора – корень из его скалярного произведения с самим собой:

$$\sqrt{\langle T - S, T - S\rangle} = r; \tag{2.6}$$

$$\langle T - S, T - S\rangle = r^2. \tag{2.7}$$

2.3.5 Пересечение луча и сферы

Составим систему из имеющихся выражений:

$$\begin{cases} \langle T - S, T - S\rangle = r^2 \\ P = O + t\vec{D}. \end{cases} \tag{2.8}$$

Предположим, что луч и сфера пересекаются, и $T = P$. Остается найти па- раметр t. Объединим имеющиеся уравнения из формулы (2.8) ($O - S = \vec{SO}$):

$$\langle \vec{SO} + t\vec{D}, \vec{SO} + t\vec{D}\rangle = r^2. \tag{2.9}$$

Используем дистрибутивность скалярного произведения:

$$\langle \vec{SO} + t\vec{D}, \vec{SO} \rangle + \langle \vec{SO} + t\vec{D}, t\vec{D} \rangle = r^2; \tag{2.10}$$

$$\langle \vec{SO}, \vec{SO} \rangle + \langle t\vec{D}, t\vec{D} \rangle + \langle t\vec{D}, \vec{SO} \rangle + \langle \vec{SO}, t\vec{D} \rangle = r^2; \tag{2.11}$$

$$\langle t\vec{D}, t\vec{D} \rangle + 2\langle t\vec{D}, \vec{SO} \rangle + \langle \vec{SO}, \vec{SO} \rangle = r^2. \tag{2.12}$$

Вынесем t за скалярное произведение:

$$t^2\langle \vec{D}, \vec{D} \rangle + 2t\langle \vec{D}, \vec{SO} \rangle + \langle \vec{SO}, \vec{SO} \rangle - r^2 = 0. \tag{2.13}$$

Мы получили квадратичное уравнение:

$$at^2 + bt + c = 0; \qquad \begin{cases} a = \langle \vec{D}, \vec{D} \rangle \\ b = \langle \vec{D}, \vec{SO} \rangle \\ c = \langle \vec{SO}, \vec{SO} \rangle - r^2. \end{cases} \tag{2.14}$$

Решив его, мы найдем пересечение луча со сферой. Определив пересечение луча со всеми сферами на сцене, мы получим наименьший параметр t, который будет соответствовать поверхности, расположенной как можно ближе к наблюдателю. Ее цвет и надо отобразить. В том случае, если луч не пересек ни одну сферу, мы закрасим пиксел цветом фона.

2.3.6 Нормали сферы

Для вычисления отражений нам понадобится уравнение нормали. Вектор нормали должен быть перпендикулярен поверхности и иметь длину 1, для любой точки сферы он лежит на прямой, проходящей через центр этой сферы:

$$\vec{N} = \frac{T - S}{|T - S|}. \tag{2.15}$$

2.3.7 Диффузное отражение

Диффузное отражение это процесс, при котором луч света, сталкиваясь с объектом, рассеивается обратно в сцену равномерно во всех направлениях. Именно оно придает матовым объектам матовость. Рассмотрим луч света с направлением $\vec{L}$ и интенсивностью I, который достигает поверхности с нормалью N . Представим интенсивность света как «ширину» луча. Его энергия распределяется по поверхности размером A. Отображу ситуацию на рисунке, обозначив вспомогательные углы и точки:

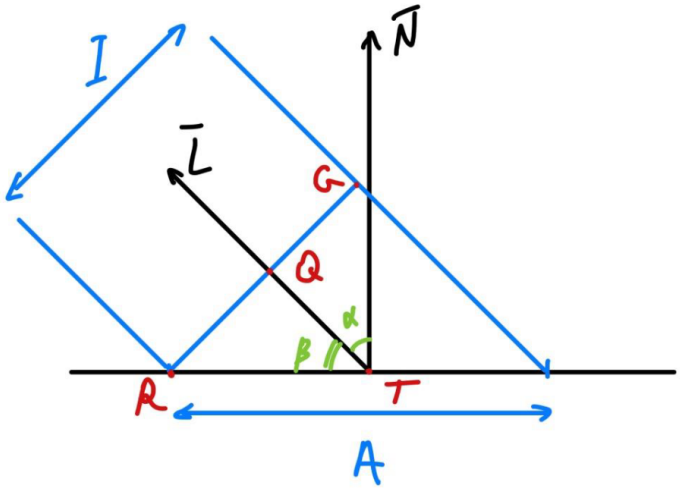

Рисунок 2.1 – Диффузное отражение света

Луч света шириной I достигает поверхности в точке T под углом β. Нормаль в T это $\vec{N}$, а переносимая лучом энергия распределяется по поверхности A. В случае, когда луч света имеет одинаковое направление с нормалью, $I = A$. С другой стороны, по мере того, как угол между $\vec{L}$ и $\vec{I}$ приближается к 90°, $A \to \infty$. Значит,

$$\lim_{A \to \infty} \frac{I}{A} = 0. \tag{2.16}$$

Выходит, для определения диффузного отражения, нам необходимо узнать что находится на промежутке, выяснив значение $\frac{I}{A}$. Рассмотрим RG – ширину луча. Так как эта прямая перпендикулярна лучу света, $QRT = \alpha$. В

треугольнике QRT сторона $QR = \frac{1}{2}$, а $RT = \frac{A}{2}$. По определению

$$\cos\alpha = \frac{QR}{RT} = \frac{\frac{1}{2}}{\frac{A}{2}} = \frac{I}{A}. \tag{2.17}$$

Так как α – угол между $\vec{(N)}$ и $\vec{L}$, мы можем использовать скалярное произведение этих векторов, чтобы выразить угол как:

$$\frac{I}{A} = \cos\alpha = \frac{\left\langle \vec{N}, \vec{L} \right\rangle}{\left|\vec{N}\right| \left|\vec{L}\right|}. \tag{2.18}$$

Мы получили простое уравнение, дающее нам отражаемую долю света в виде функции угла между нормалью поверхности и направлением света. Если у нас получилось, что значение $\alpha > 90$, это означает, что свет освещает тыльную сторону поверхности, и учитывать это не нужно.

В итоге, мы можем записать уравнение диффузного отражения для вычисления общего количества света, получаемого точкой T с нормалью $\vec{N}$ в сцене с рассеянным светом с интенсивностью I_A и n точечных или направленных источников света с интенсивностью I_n, а так же световыми векторами $\vec{L_n}$:

$$I_T = I_A + \sum_{i=1}^{n} I_i * \frac{\left\langle \vec{N}, \vec{L} \right\rangle}{\left|\vec{N}\right| \left|\vec{L}\right|}. \tag{2.19}$$

2.3.8 Зеркальное отражение

Зеркальное отражение отображает глянцевость объектов. Их вид меняется при смещении точки обзора. Рассмотрим луч света $\vec{L}$. Нам нужно определить, сколько света от него отражается обратно в направлении точки обзора. Пусть $\vec{V}$ – вектор обзора, указывающий из точки T в сторону камеры, а α – угол между $\vec{R}$ и $\vec{V}$, тогда получим следующую картину:

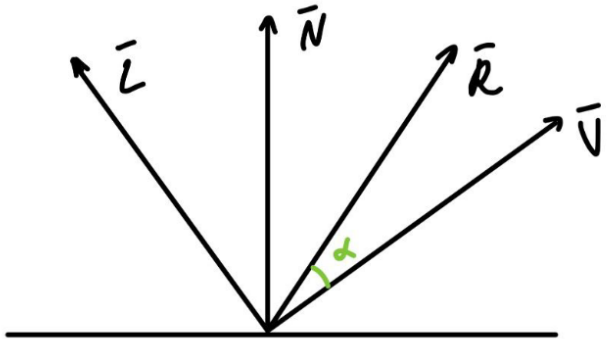

Рисунок 2.2 – Зеркальное отражение света

При $\alpha = 0$ весь свет отражается в направлении $\vec{V}$. При $\alpha = 90$ свет вообще не отражается. Как и в случае с диффузным отражением, нам нужно математическое выражение, позволяющее определить, что происходит при промежуточных значениях α.

Данная модель не имеет физического прототипа, но отлично отражает необходимые свойства и проста в вычислении.

Рассмотрим свойство $\cos\alpha$: $\cos 0 = 1 \cos \pm 90 = 0$. Это мы и будем использовать. Теперь необходимо учесть глянцевость поверхности. Глянцевость – это мера того, как быстро уменьшается функция отражения при возрастании α. Простой способ это сделать – возвести $\cos\alpha$ в положительную степень s:

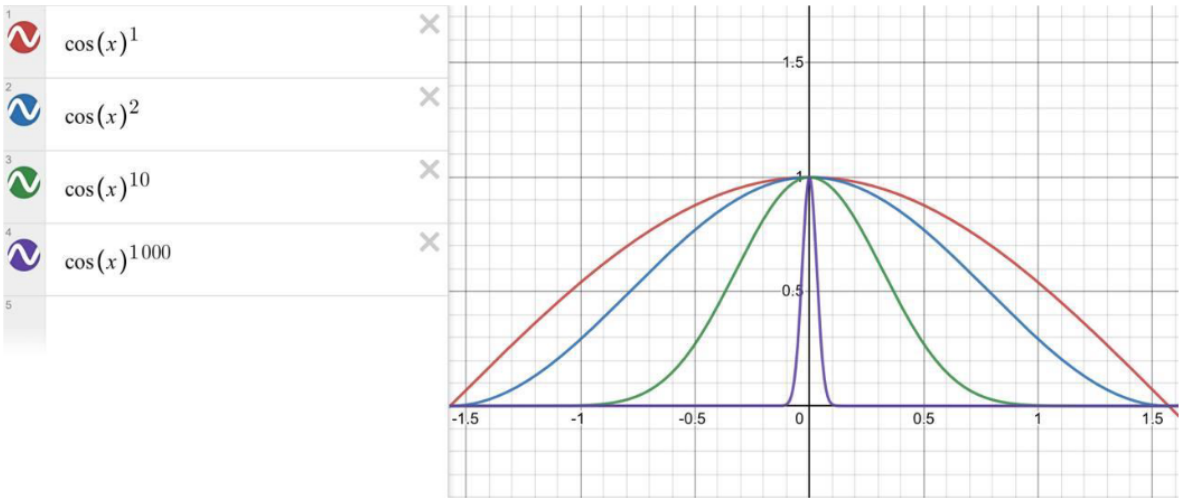

Рисунок 2.3 – График для $\cos x^s$

s – характеристика блеска поверхности. Для начала вычислим $\vec{R}$ из $\vec{N}$ и

$\vec{L}$. Для этого разделим $\vec{L}$ на $\vec{L_P}$ и $\vec{L_N}$ так, чтобы $\vec{L} = \vec{L_P} + \vec{L_N}$:

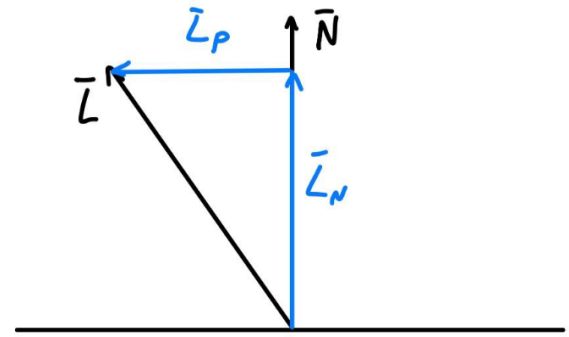

Рисунок 2.4 – Разделение $\vec{L}$ на $\vec{L_P}$ и $\vec{L_N}$

$\vec{L_N}$ – проекция $\vec{L}$ на $\vec{N}$. Исходя из свойств скалярного произведения и того, что длина нормали равняется единице, длина проекции равна $\left\langle \vec{N}, \vec{L} \right\rangle$ Так как $\vec{L_N} || \vec{N} = \vec{N} * \left\langle \vec{N}, \vec{L} \right\rangle$. Получим $\vec{L_P} = \vec{L} - \vec{N} * \left\langle \vec{N}, \vec{L} \right\rangle$. Теперь рассмотрим $\vec{R}$:

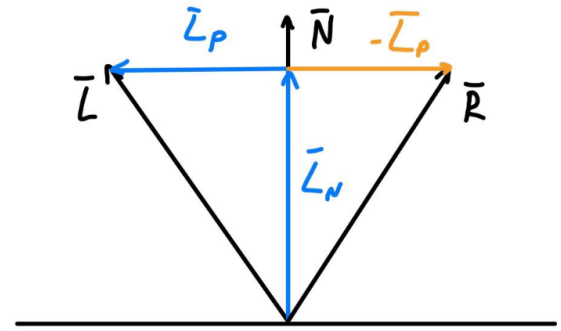

Рисунок 2.5 – Расположение векторов $\vec{R}$ и $\vec{L}$

Исходя из рисунка: $\vec{R} = \vec{L_N} - \vec{L_P}$. Подставив полученное выше:

$$\vec{R} = \vec{N} * \left\langle \vec{N}, \vec{L} \right\rangle - \vec{L} + \vec{N} * \left\langle \vec{N}, \vec{L} \right\rangle = 2 * \vec{N} * \left\langle \vec{N}, \vec{L} \right\rangle - \vec{L} \tag{2.20}$$

Теперь, по аналогии с диффузным отражением, можно составить уравнение для зеркального:

$$\vec{R} = 2 * \vec{N} * \left\langle \vec{N}, \vec{L} \right\rangle - \vec{L}; \tag{2.21}$$

$$I_S = I_L * \left(\frac{\left\langle \vec{R}, \vec{V} \right\rangle}{\left|\vec{R}\right| \left|\vec{V}\right|} \right)^S . \tag{2.22}$$

Как и в случае с диффузным отражением, если $\cos\alpha$ оказался отрицательным, его нужно игнорировать. Кроме того, для матовых объектов выражение зеркальности не должно вычисляться. Это можно предусмотреть заранее, пометив соответствующую поверхность, чтобы сократить количество вычислений.

Вычислив зеркальное и диффузное отражение, составим уравнение полного освещения в точке T:

$$I_T = I_A + \sum_{i=1}^{n} I_i * \left[\frac{\left\langle \vec{N}, \vec{L} \right\rangle}{\left|\vec{N}\right| \left|\vec{L}\right|} + \left(\frac{\left\langle \vec{R}, \vec{V} \right\rangle}{\left|\vec{R}\right| \left|\vec{V}\right|} \right)^S \right], \tag{2.23}$$

где I_A – интенсивность рассеянного света, $\vec{N}$ – нормаль к поверхности в точке T, V – вектор от T к камере, s – зеркальная характеристика поверхности, I_i – интенсивность потока света u, L_i – вектор из T к свету i, а R_i – вектор отражения в T для потока света i.

2.3.9 Тени

Тени возникают, когда лучи света не могут достичь объекта из-за встреченного на пути препятствия.

Начнем с направленного света. Нам известна точка T на поверхности, а также луч света $\vec{L}$. Зная это, мы можем определить луч $(O+t\vec{L})$, проходящий из этой точки поверхности к бесконечно удаленному источнику света. Если этот луч пересекается с чем-либо, то точка будет находится в тени и освещения от этого источника нужно игнорировать. В ином случае, мы добавляем освещение данного источника, как было показано ранее.

Точечные источники можно рассматривать так же, но надо учитывать то, что мы не хотим, чтобы объекты дальше источника света могли отбрасывать тени на T. Установим $t_{max} = 1$, чтобы при достижении источника света луч останавливался. Так же не стоит забывать, что $t_{min} = eps$, где aps – очень

близкое к нулю число справа, чтобы мы не нашли пересечение с поверхностью, из которой и пускаем луч.

2.3.10 Отражения

Когда мы смотрим в зеркало, мы видим отражаемые им лучи света. Они отражаются симметрично относительно нормали поверхности. Итак, нам необходимо выяснить, куда идет луч, отражаемый от зеркальной поверхности.

Таким образом, мы получим рекурсивный алгоритм. При его создании нам нужно убедится, что мы не порождаем бесконечный цикл. У нас будет два условия выхода: когда луч сталкивается с неотражающим объектом, и когда он ни с чем не сталкивается. Также важно вспомнить об эффекте «бесконечного коридора», который образуется, если поставить два зеркала друг напротив друга. Самый простой способ предотвратить бесконечный подсчет отражений: ограничить рекурсию каким-либо числом. В нашей задачи нет необходимости отображать зеркальные поверхности, а тем более визуализировать эффект «бесконечного коридора», поэтому можно обойтись достаточно малым значением в 2-3 единицы.

2.4 Связь различных текстурных карт с данными о нормали

Как было сказано ранее, существует несколько применяемых способов вносить возмущения в нормали. Рассмотрим соответствующие текстурные карты от наиболее примитивной, к наиболее сложной.

2.4.1 Карты высот (англ. height maps)

Карты высот имеют черно-белый цвет, что означает, что все каналы RGB (red, green, blue) равны между собой. Таким образом, они хранят единственное значение. Пример:

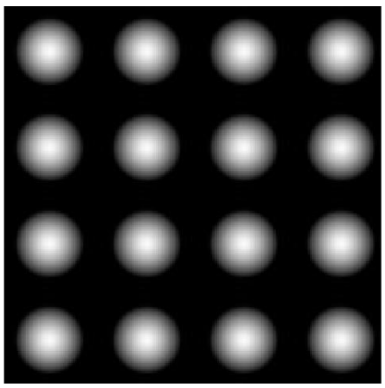

Рисунок 2.6 – Карта высот $h(x, y)$

Чтобы получить возмущение, сначала нужно вычислить аппроксимацию производных по направлениям x и y с помощью разностного аналога [5]:

$$h_x(x, y) = \frac{h(x + 1, y) - h(x - 1, y)}{2}; \qquad h_y(x, y) = \frac{h(x, y + 1) - h(x, y - 1)}{2}. \tag{2.24}$$

В изображении это будет выглядеть так:

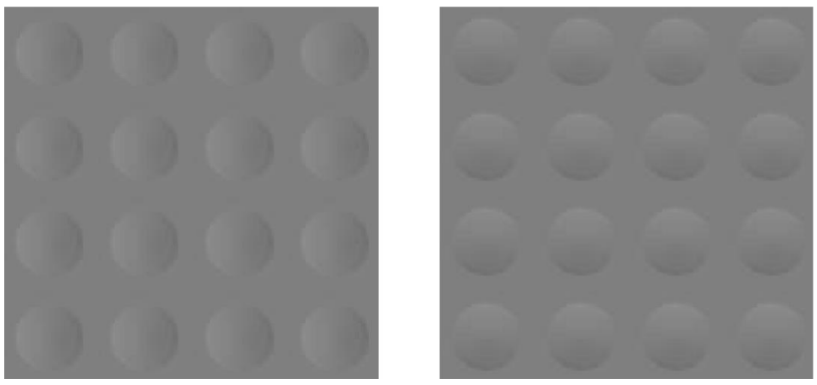

Рисунок 2.7 – Аппроксимация по x (слева) и y (справа)

Тогда ненормализованное значение нормали в текстеле (x, y):

$$N'(x, y) = (-h_x(x, y) - h_y(x, y), 1). \tag{2.25}$$

2.4.2 Карты нормалей (англ. normal maps)

Карты нормалей содержат значения в двух каналах. Принято хранить h_x в канале R, а h_y в G. Тогда значение канала B не используется, и все карты имеют голубоватый оттенок

В изображении это будет выглядеть так:

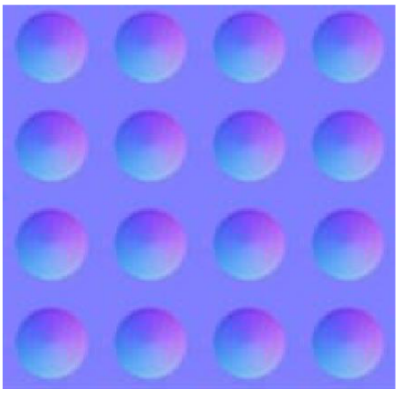

Рисунок 2.8 – Карта нормалей

2.4.3 Карта параллактического отображения (англ. paral map)

С предыдущими типами карт есть проблема: при смещении угла обзора, это никак не влияет на рельеф. Если мы посмотрим вдоль реальной кирпичной стены под некоторым углом, мы не увидим зазор между кирпичами. Это происходит, потому что мы лишь изменяли нормали.

Идея параллакса заключается в том, что положение объектов относительно друг друга должно изменятся с движением наблюдателя. Неровности должны увеличиваться в высоту [5].

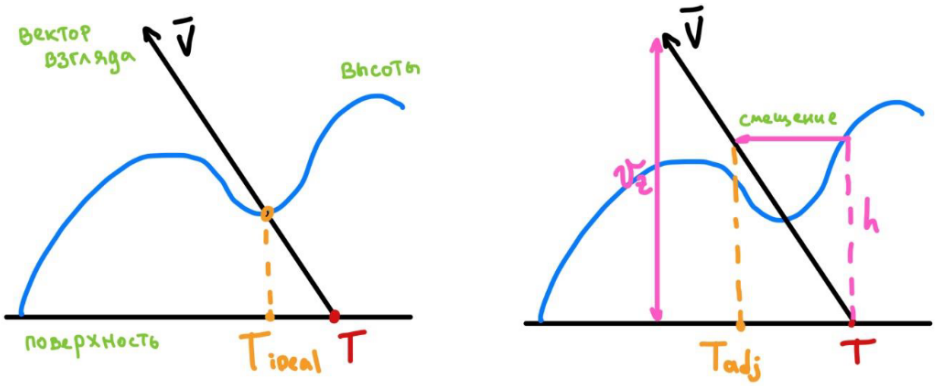

Рисунок 2.9 – Фактическое положение поверхности определяется лучом взгляда (слева). Параллактическое отображение выполняет аппроксимацию, используя высоту для нахождения положения новой точки (справа).

Теперь, помимо направления нормали, нам необходимо хранить высоту h. Учитывая местоположение с координатами текстуры T, высоту h, нормализованный вектор вида V со значением высоты V_z и горизонтальной составляющей V_{xy}, новая скорректированная по параллаксу текстурная координата:

$$T_{adj} = T + \frac{h * V_{xy}}{V_z}. \tag{2.26}$$

Однако, когда вектор обзора находится вблизи горизонта поверхности, небольшое изменение высоты приводит к большому смещению координат текстуры. Аппроксимация не будет корректной, поскольку полученное новое местоположение практически не коррелирует по высоте с исходным местоположением на поверхности. Чтобы решить эту проблему, мы может ограничить смещение: оно не должно превышать высоту. Тогда уравнение будет иметь вид:

$$T_{adj} = T + h * V_{xy}. \tag{2.27}$$

При крутых углах, это уравнение почти совпадает с исходным, поскольку $V_z \approx 1$. При малых углах смещение становится ограниченным. Но даже с такими ограничениями параллактическое смещение должно обеспечивать более реалистичное представление.

2.5 Наложение текстуры на поверхность

Теперь, когда у нас есть все необходимые уравнения и преобразования, остается лишь связать текстурную карту с поверхностью. То есть нужно определить, какой пиксел h_{xy} текстурной карты соответствует точке на сфере. Определим

$$x = u * M_w; \qquad y = v * M_h, \tag{2.28}$$

где M_w, M_h – угол между положительным направлением оси x и проекцией нормали к поверхности сферы на плоскость XY. Его мы можем вычислить по следующей формуле:

$$\phi = \arctan \frac{N_y}{N_x}. \tag{2.29}$$

Этот угол измеряет поворот вокруг вертикальной оси Z на плоскости XY. Он учитывает вращение окружности вокруг вертикальной оси, что является сферой. Переведя этот угол в диапазон $[0, 1]$ получим необходимое значение u:

$$u = \frac{\phi + \pi}{2\pi}. \tag{2.30}$$

Пусть θ – угол между нормалью и положительным направлением оси Z:

$$\theta = \arccos N_z. \tag{2.31}$$

Данный же угол определяет наклон нормали относительно вертикальной оси. На сфере он соответствует углу между нормалью и радиус-вектором точки на сфере. Он позволяет корректно распределить текстуру вдоль вертикальной оси сферы. Остается привести угол в необходимый диапазон, получив v:

$$v = \frac{\theta}{\phi}. \tag{2.32}$$

2.6 Схема алгоритма трассировки луча

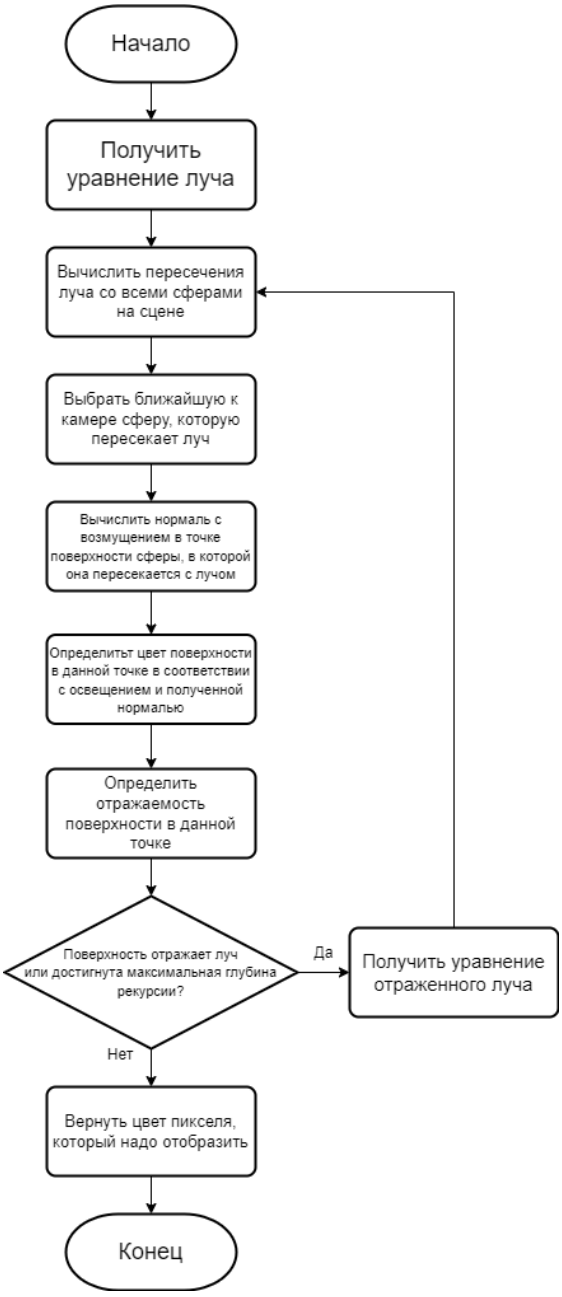

Рисунок 2.10 – Блок-схема алгоритма трассировки луча.

Вывод

В данном разделе были подробно рассмотрены алгоритмы, необходимые для реализации поставленной цели.

3 Технологическая часть

В данном разделе описаны средства реализации, диаграмма классов, а так же интерфейс ПО. Так же представлены реализованные алгоритмы, которые были выбраны ранее.

3.1 Средства реализации

Для реализации программного обеспечения был выбран язык C# по следующим причинам:

1. Он поддерживает объектно-ориентированный стиль программирования;
2. Личный опыт разработки на данном языке позволяет эффективно решать поставленные задачи;
3. Язык обладает высокой производительностью, что важно при трассировке лучей и выполнении вычислений на процессоре.

В качестве среды разработки была выбрана Visual Studio, поскольку она достаточно удобна и обладает всем необходимым функционалом для реализации поставленной задачи. В частности, использование библиотеки WPF позволяет реализовать современный и функциональный пользовательский интерфейс, а средства Visual Studio упрощают разработку этого интерфейса.

3.2 Диаграмма классов

На рисунке ниже представлена UML-диаграмма классов, необходимых для решения поставленной задачи. Она отображает структуру и взаимосвязи между классами, что позволяет наглядно представить архитектуру программного обеспечения и взаимодействие его компонентов.

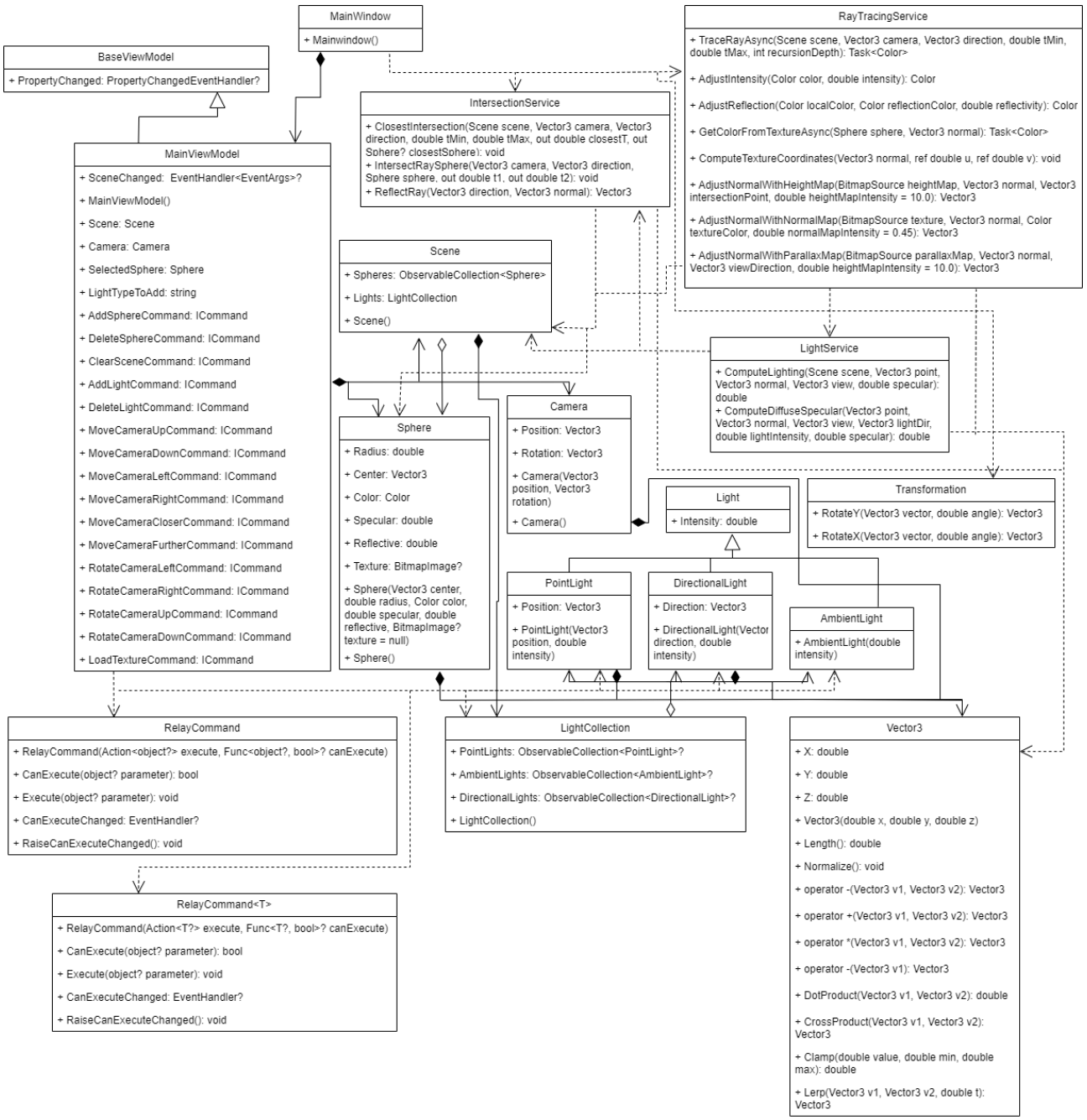

Рисунок 3.1 – UML-диаграмма

3.3 Реализация алгоритмов

Рассмотрим класс, реализующий логику пускания лучей и их пересечения с объектами.

Листинг 3.1 – Класс IntersectionService

```
public static class IntersectionService
{
```

```
public static void ClosestIntersection(Scene scene, Vector3 camera,
    Vector3 direction, double tMin, double tMax, out double closestT, out
    Sphere? closestSphere)
{
  closestT = double.PositiveInfinity;
  closestSphere = null;

  foreach (var sphere in scene.Spheres)
  {
    IntersectRaySphere(camera, direction, sphere, out double t1, out
        double t2);

    if (tMin <= t1 && t1 <= tMax && t1 < closestT)
    {
      closestT = t1;
      closestSphere = sphere;
    }

    if (tMin <= t2 && t2 <= tMax && t2 < closestT)
    {
      closestT = t2;
      closestSphere = sphere;
    }
  }
}

private static void IntersectRaySphere(Vector3 camera, Vector3 direction,
    Sphere sphere, out double t1, out double t2)
{
  Vector3 oc = camera - sphere.Center;
  double a = Vector3.DotProduct(direction, direction);
  double b = 2.0 * Vector3.DotProduct(oc, direction);
  double c = Vector3.DotProduct(oc, oc) - sphere.Radius * sphere.Radius;
  double discriminant = b * b - 4 * a * c;

  if (discriminant < 0)
  {
    t1 = t2 = double.PositiveInfinity;
  }
  else
  {
    double sqrtDiscriminant = Math.Sqrt(discriminant);
    t1 = (-b - sqrtDiscriminant) / (2.0 * a);
    t2 = (-b + sqrtDiscriminant) / (2.0 * a);
  }
}

public static Vector3 ReflectRay(Vector3 direction, Vector3 normal)
```

```
  {
    double dotProduct = Vector3.DotProduct(direction, normal);
    return  normal * (2 * dotProduct) - direction;
  }
}
```

Теперь представим класс, отвечающий за внедрение света.

Листинг 3.2 – Класс LightService

```
public static class LightService
{
  public static double ComputeLighting(Scene scene, Vector3 point, Vector3
      normal, Vector3 view, double specular)
  {
    double intensity = 0;

    foreach (var ambient in scene.Lights.AmbientLights)
    intensity += ambient.Intensity;

    foreach (var pointLight in scene.Lights.PointLights)
    {
      Vector3 lightDir = pointLight.Position - point;
      lightDir.Normalize();
      intensity += ComputeDiffuseSpecular(point, normal, view, lightDir,
          pointLight.Intensity, specular);
    }

    foreach (var directional in scene.Lights.DirectionalLights)
    {
      Vector3 lightDir = new Vector3(directional.Direction.X, directional.
          Direction.Y, directional.Direction.Z);
      lightDir.Normalize();
      intensity += ComputeDiffuseSpecular(point, normal, view, lightDir,
          directional.Intensity, specular);
    }

    return intensity;
  }

  private static double ComputeDiffuseSpecular(Vector3 point, Vector3 normal
      , Vector3 view, Vector3 lightDir, double lightIntensity, double
      specular)
  {
    double diffuseIntensity = Vector3.DotProduct(normal, lightDir) *
        lightIntensity;
    diffuseIntensity = Math.Max(diffuseIntensity, 0);

    double specularIntensity = 0;
```

```
    if (specular >= 0)
    {
      Vector3 reflectDir = IntersectionService. ReflectRay(lightDir, normal)
          ;
      double spec = Vector3.DotProduct(reflectDir, view);
      if (spec > 0)
      {
        specularIntensity = Math.Pow(spec, specular) * lightIntensity;
      }
    }

    return Math.Max(0, Math.Min(1, diffuseIntensity + specularIntensity));
  }
}
```

Осталось рассмотреть класс который отвечает за саму трассировку. Он так же должен выпонять учет текстуры, ведь при трассировке и так вычисляются нормали и достаточно эффективно сразу же вносить в них возмущения.

Листинг 3.3 – Класс RayTracingService

```
public static class RayTracingService
{
  public static async Task<Color> TraceRayAsync(Scene scene, Vector3 camera,
      Vector3 direction, double tMin, double tMax, int recursionDepth)
  {
    IntersectionService.ClosestIntersection(scene, camera, direction, tMin,
       tMax, out double closestT, out Sphere? closestSphere);

    if (closestSphere == null)
    {
      return Colors.LightBlue;
    }

    Vector3 intersectionPoint = camera + direction * closestT;
    Vector3 normal = intersectionPoint - closestSphere.Center;
    normal.Normalize();

    if (closestSphere.Texture != null)
    {

      switch (closestSphere.TextureType)
      {
        case TextureType.HeightMap:
        normal = AdjustNormalWithHeightMap(closestSphere.Texture, normal,
           intersectionPoint);
        break;
```

```
        case TextureType.NormalMap:
        Color textureColor = await GetColorFromTextureAsync(closestSphere,
            normal);
        normal = AdjustNormalWithNormalMap(closestSphere.Texture, normal,
            textureColor);
        break;
        case TextureType.ParallaxMap:
        normal = AdjustNormalWithParallaxMap(closestSphere.Texture, normal,
            intersectionPoint);
        break;
      }
      normal.Normalize();
    }

    Vector3 viewDirection = -direction;
    double intensity = LightService.ComputeLighting(scene, intersectionPoint
        , normal, viewDirection, closestSphere.Specular);
    Color localColor = AdjustIntensity(closestSphere.Color, intensity);

    double reflectivity = closestSphere.Reflective;
    if (recursionDepth <= 0 || reflectivity <= 0)
    {
      return localColor;
    }

    Vector3 reflectionDirection = IntersectionService.ReflectRay(
        viewDirection, normal);
    Color reflectionColor = await TraceRayAsync(scene, intersectionPoint,
        reflectionDirection, 0.001, double.PositiveInfinity, recursionDepth -
         1);
    Color finalColor = AdjustReflection(localColor, reflectionColor,
        reflectivity);

    return finalColor;
  }

  private static Color AdjustIntensity(Color color, double intensity)
  {
    byte r = (byte)Math.Min(255, Math.Max(0, color.R * intensity));
    byte g = (byte)Math.Min(255, Math.Max(0, color.G * intensity));
    byte b = (byte)Math.Min(255, Math.Max(0, color.B * intensity));
    return Color.FromRgb(r, g, b);
  }

  private static Color AdjustReflection(Color localColor, Color
      reflectionColor, double reflectivity)
  {
    byte r = (byte)Math.Min(255, Math.Max(0, localColor.R * (1 -
```

```
            reflectivity) + reflectionColor.R * reflectivity));
        byte g = (byte)Math.Min(255, Math.Max(0, localColor.G * (1 -
            reflectivity) + reflectionColor.G * reflectivity));
        byte b = (byte)Math.Min(255, Math.Max(0, localColor.B * (1 -
            reflectivity) + reflectionColor.B * reflectivity));
        return Color.FromRgb(r, g, b);
    }

    private static async Task<Color> GetColorFromTextureAsync(Sphere sphere,
        Vector3 normal)
    {
        double u = 0;
        double v = 0;

        ComputeTextureCoordinates(normal, ref u, ref v);

        int x = (int)(u * sphere.Texture?.PixelWidth ?? 0);
        int y = (int)(v * sphere.Texture?.PixelHeight ?? 0);

        var pixels = new byte[4];

        await Application.Current.Dispatcher.InvokeAsync(() =>
        {
            sphere.Texture?.CopyPixels(new Int32Rect(x, y, 1, 1), pixels, 4, 0);
        });

        return Color.FromArgb(pixels[3], pixels[2], pixels[1], pixels[0]);
    }

    private static void ComputeTextureCoordinates(Vector3 normal, ref double u
        , ref double v)
    {
        double phi = Math.Atan2(normal.Y, normal.X);
        double theta = Math.Acos(normal.Z);

        u = (phi + Math.PI) / (2.0 * Math.PI);
        v = theta / Math.PI;
    }

    private static Vector3 AdjustNormalWithHeightMap(BitmapSource heightMap,
        Vector3 normal, Vector3 intersectionPoint, double heightMapIntensity =
        10.0)
    {
        double u = 0;
        double v = 0;
        ComputeTextureCoordinates(normal, ref u, ref v);

        int width = heightMap.PixelWidth;
```

```
int height = heightMap.PixelHeight;

int x = (int)(u * width);
int y = (int)(v * height);

x = Math.Max(0, Math.Min(width - 1, x));
y = Math.Max(0, Math.Min(height - 1, y));

byte[] currentPixel = new byte[4];
heightMap.CopyPixels(new Int32Rect(x, y, 1, 1), currentPixel, 4, 0);

double h = currentPixel[0] / 255.0;

Vector3 gradient = new Vector3(0, 0, 1.0);

if (x > 0 && x < width - 1)
{
  byte[] leftPixel = new byte[4];
  byte[] rightPixel = new byte[4];

  heightMap.CopyPixels(new Int32Rect(x - 1, y, 1, 1), leftPixel, 4, 0);
  heightMap.CopyPixels(new Int32Rect(x + 1, y, 1, 1), rightPixel, 4, 0);

  double hLeft = leftPixel[0] / 255.0;
  double hRight = rightPixel[0] / 255.0;

  gradient.X = (hRight - hLeft) * heightMapIntensity;
}

if (y > 0 && y < height - 1)
{
  byte[] topPixel = new byte[4];
  byte[] bottomPixel = new byte[4];

  heightMap.CopyPixels(new Int32Rect(x, y - 1, 1, 1), topPixel, 4, 0);
  heightMap.CopyPixels(new Int32Rect(x, y + 1, 1, 1), bottomPixel, 4, 0)
      ;

  double hTop = topPixel[0] / 255.0;
  double hBottom = bottomPixel[0] / 255.0;

  gradient.Y = (hBottom - hTop) * heightMapIntensity;
}

Vector3 perturbedNormal = normal + new Vector3(-gradient.X, -gradient.Y,
    1.0);
perturbedNormal.Normalize();
```

```
  return perturbedNormal;
}


private static Vector3 AdjustNormalWithNormalMap(BitmapSource texture,
    Vector3 normal, Color textureColor, double normalMapIntensity = 0.45)
{
  double x = ((textureColor.R) / 255.0) * 2.0 - 1.0;
  double y = ((textureColor.G) / 255.0) * 2.0 - 1.0;

  double z = Math.Sqrt(Math.Max(0.0, 1.0 - Math.Clamp(x * x + y * y, 0.0,
     1.0)));

  Vector3 normalMapNormal = new Vector3(x, y, z);
  normalMapNormal.Normalize();

  return Vector3.Lerp(normal, normalMapNormal, normalMapIntensity);
}


private static Vector3 AdjustNormalWithParallaxMap(BitmapSource
    parallaxMap, Vector3 normal, Vector3 viewDirection, double
    heightMapIntensity = 10.0)
{
  double u = 0;
  double v = 0;
  ComputeTextureCoordinates(normal, ref u, ref v);

  int width = parallaxMap.PixelWidth;
  int height = parallaxMap.PixelHeight;

  int x = (int)(u * width) % width;
  int y = (int)(v * height) % height;

  x = (x + width) % width;
  y = (y + height) % height;

  byte[] currentPixel = new byte[4];
  parallaxMap.CopyPixels(new Int32Rect(x, y, 1, 1), currentPixel, 4, 0);
  double heightValue = currentPixel[0] / 255.0;

  Vector3 viewDirXY = new Vector3(viewDirection.X, viewDirection.Y, 0);
  viewDirXY.Normalize();

  double parallaxScale = 0.05;

  Vector3 parallaxOffset = viewDirXY * (heightValue * heightMapIntensity *
      parallaxScale);
```

```
parallaxOffset.X = (parallaxOffset.X + width) % width;
parallaxOffset.Y = (parallaxOffset.Y + height) % height;

double uAdj = (u - parallaxOffset.X / width + 1.0) % 1.0;
double vAdj = (v - parallaxOffset.Y / height + 1.0) % 1.0;

int adjX = (int)(uAdj * width) % width;
int adjY = (int)(vAdj * height) % height;

adjX = (adjX + width) % width;
adjY = (adjY + height) % height;

byte[] adjustedPixel = new byte[4];
parallaxMap.CopyPixels(new Int32Rect(adjX, adjY, 1, 1), adjustedPixel,
    4, 0);

double hX = 0.0;
double hY = 0.0;

if (adjX > 0 && adjX < width - 1)
{
  byte[] leftPixel = new byte[4];
  byte[] rightPixel = new byte[4];

  parallaxMap.CopyPixels(new Int32Rect((adjX - 1 + width) % width, adjY,
      1, 1), leftPixel, 4, 0);
  parallaxMap.CopyPixels(new Int32Rect((adjX + 1) % width, adjY, 1, 1),
     rightPixel, 4, 0);

  double hLeft = leftPixel[0] / 255.0;
  double hRight = rightPixel[0] / 255.0;

  hX = (hRight - hLeft) * heightMapIntensity;
}

if (adjY > 0 && adjY < height - 1)
{
  byte[] topPixel = new byte[4];
  byte[] bottomPixel = new byte[4];

  parallaxMap.CopyPixels(new Int32Rect(adjX, (adjY - 1 + height) %
     height, 1, 1), topPixel, 4, 0);
  parallaxMap.CopyPixels(new Int32Rect(adjX, (adjY + 1) % height, 1, 1),
      bottomPixel, 4, 0);

  double hTop = topPixel[0] / 255.0;
  double hBottom = bottomPixel[0] / 255.0;
```

```
        hY = (hBottom - hTop) * heightMapIntensity;
      }

      Vector3 perturbedNormal = new Vector3(-hX, -hY, 1.0);
      perturbedNormal.Normalize();

      return normal + perturbedNormal;
    }
}
```

Для ускорения работы программы методы трассировки был реализованы асинхронными, поскольку алгоритм позволяет производить вычисления для каждого из лучей не зависимо от предыдущих и последующих результатов. Это очень сильно ускоряет работу программы и позволяет сделать интерфейс более отзывчивым, а работу в программе более приятной для пользователя.

3.4 Интерфейс программного обеспечения

На рисунке 3.2 представлен интерфейс программы.

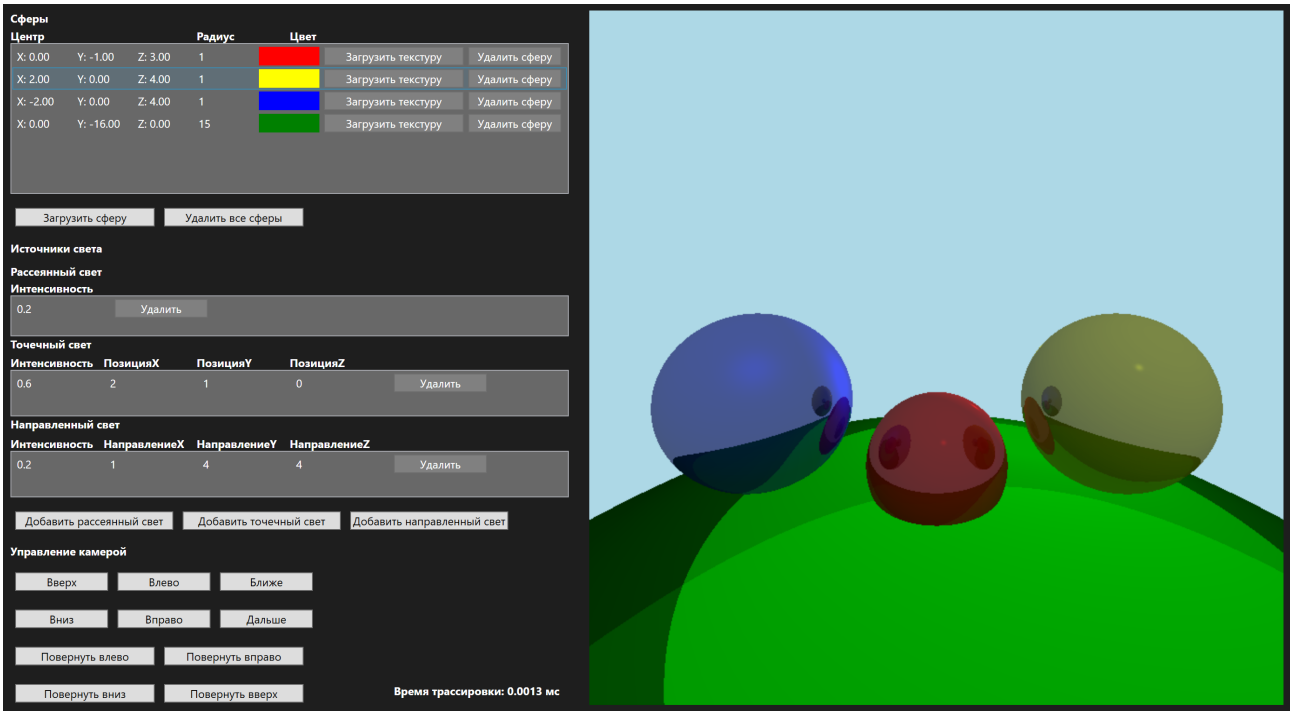

Рисунок 3.2 – Интерфейс ПО

В левой часте экрана расположены списки всех сфер и источников света

на сцене, а так же различные кнопки: добавления и удаления сфер, наложения текстур на сферу, перемещения и вращения камеры, добавления и удаления света. В нижней часте расположено поле, в котором отображается время выполнения трассировки сцены.

Вывод

В этом разделе были выбраны средства реализации программы и составлена диаграмма классов, представлена реализация алгоритмов и пользовательский интерфейс.

4 Исследовательская часть

В данном разделе будут приведены технические характеристики устройства, на котором проводился анализ алгоритмов, а также результаты работы программного обеспечения.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось исследование, следующие:

— Операционная система: Microsoft Windows 10 Pro;

— Оперативная память: 32 ГБ;

— Процессор: Intel(R) Core(TM) i9-9980HK CPU @ 2.40GHz, 2401 Mhz;

— Количество ядер: 8 физических и 16 логических.

4.2 Результаты работы ПО

На рисунках 4.1 – 4.5 представлены изображения, полученные с помощью разработанного ПО.

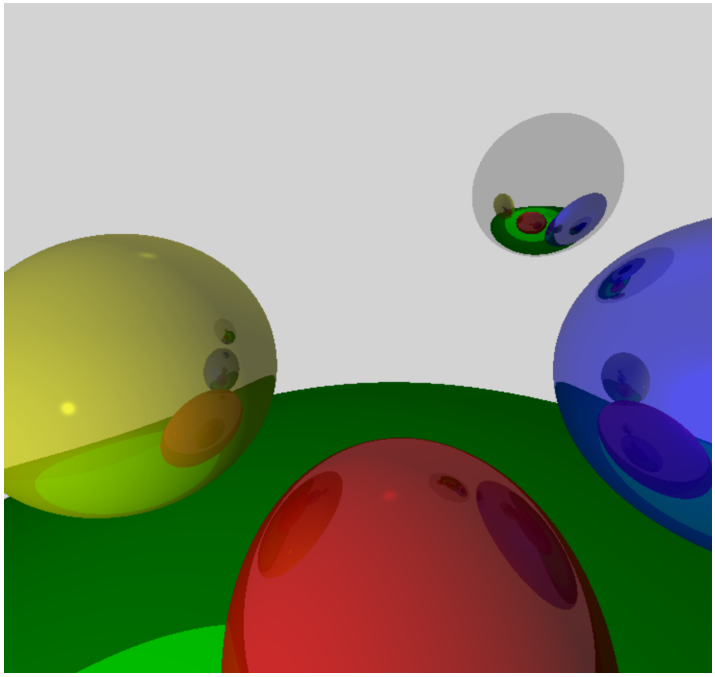

Рисунок 4.1 – Отражение сфер друг от друга.

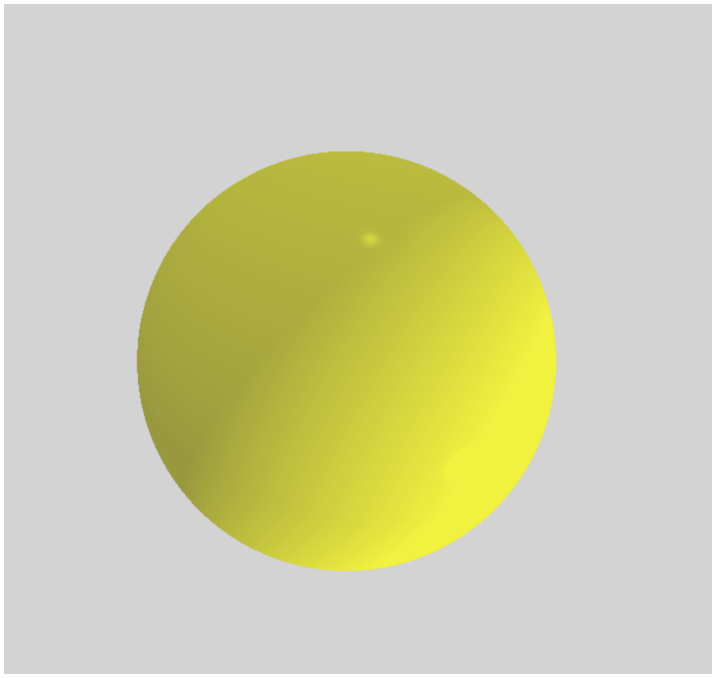

Рисунок 4.2 – Отображение света.

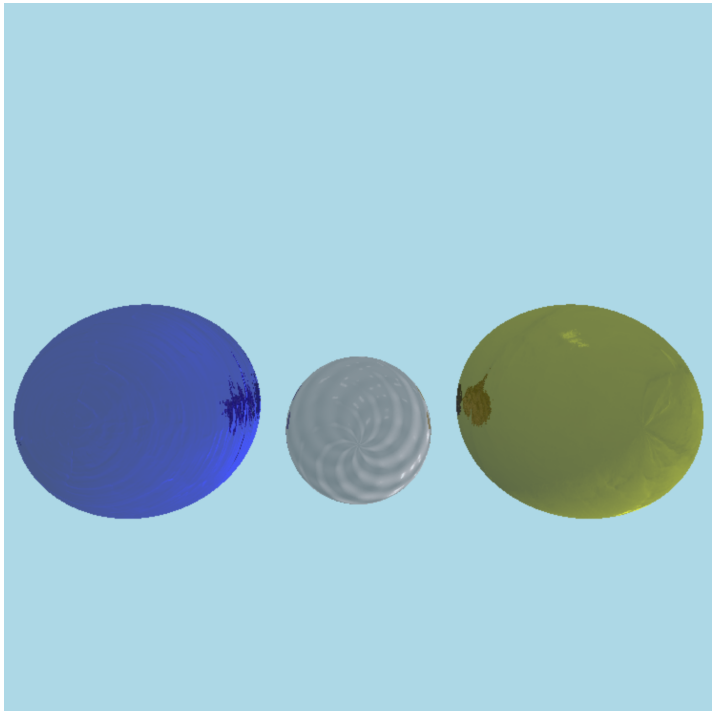

Рисунок 4.3 – Простая композиция шаров с текстурами, наложенными по карте высот.

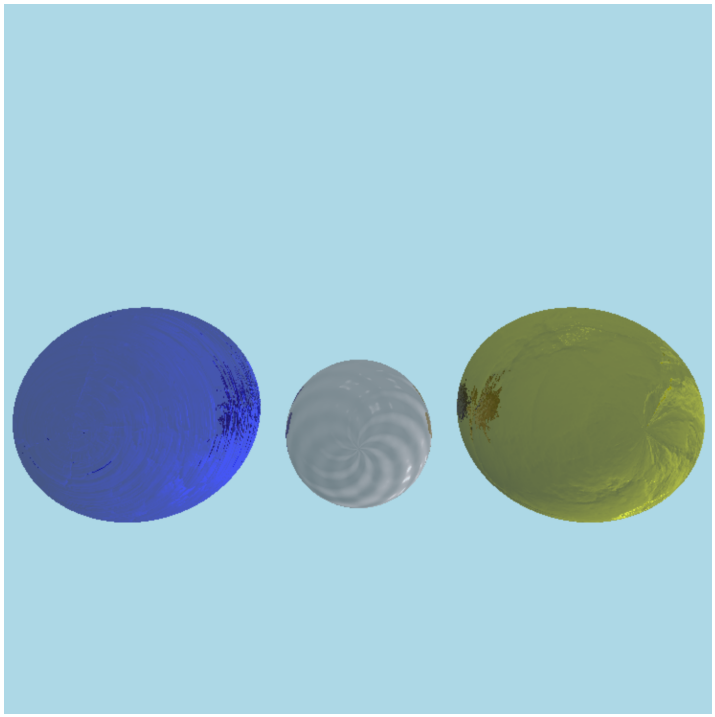

Рисунок 4.4 – Простая композиция шаров с текстурами, наложенными по карте нормалей.

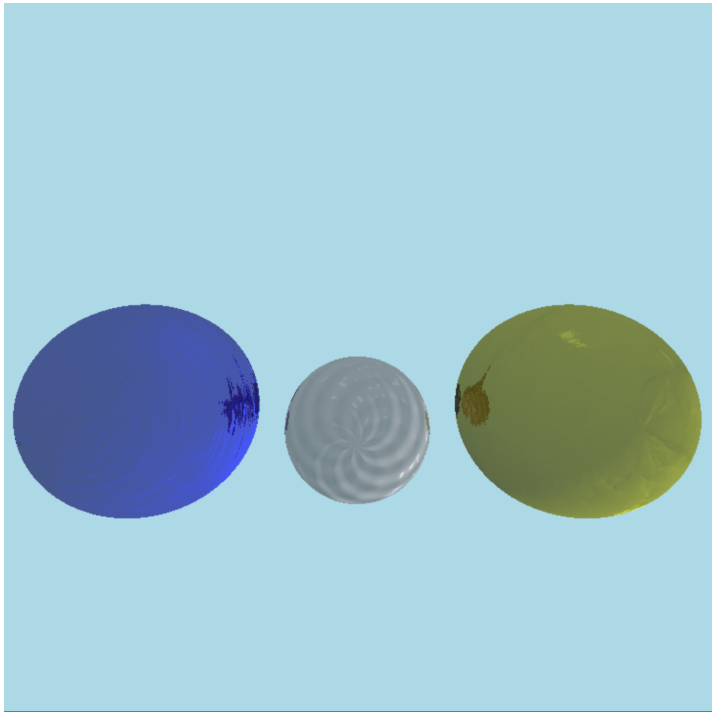

Рисунок 4.5 – Простая композиция шаров с текстурами, наложенными по карте паралактического отображения.

4.3 Анализ наложения текстур

Как мы можем заметить, на получившихся сценах (4.1 – 4.5) текстуры накладываются визуально схожим образом, все три варианта содержать правильное отображение теней, бликов и отражений. Таким образом на практике применимы все варианты, но чтобы более точно проанализировать эти методы, проведем еще одну проверку.

Согласно теории, методы наложения текстур по карте высот и нормалей очень схожи, просто по-разному хранятся данные. А для текстурирования по картам параллактического отображения, нам нужны дополнительные математические вычисления, но в итоге должна выйти более привлекательная сцена в тех местах, где мы смотрим на объект под маленьким углом. Чтобы в этом убедиться построим сцену, где мы будем нахожится ближе к поверхности большого шара и будем смотреть вдоль нее. Результаты работы программы с такой сценой представлены на рисунках 4.6 – 4.8.

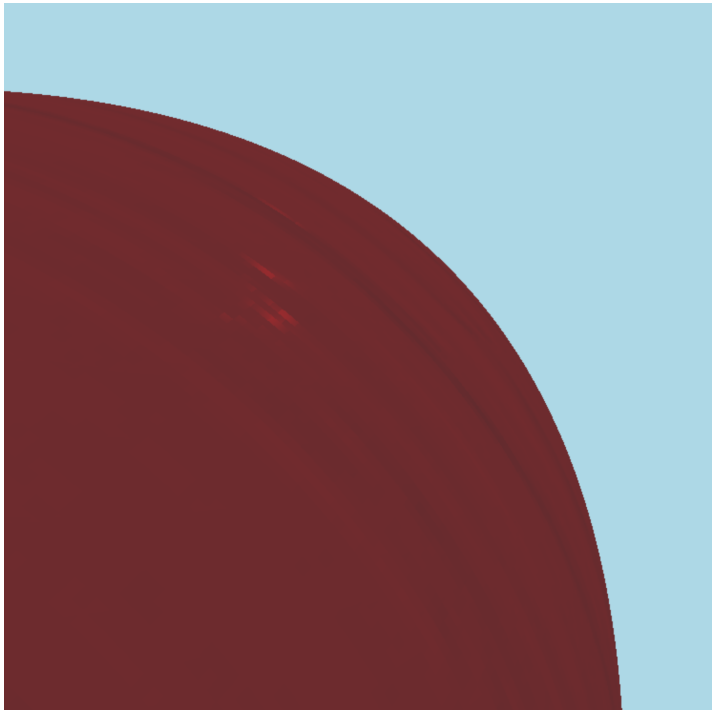

Рисунок 4.6 – Поверхность шара под маленьким углом с текстурами, наложенными по карте высот.

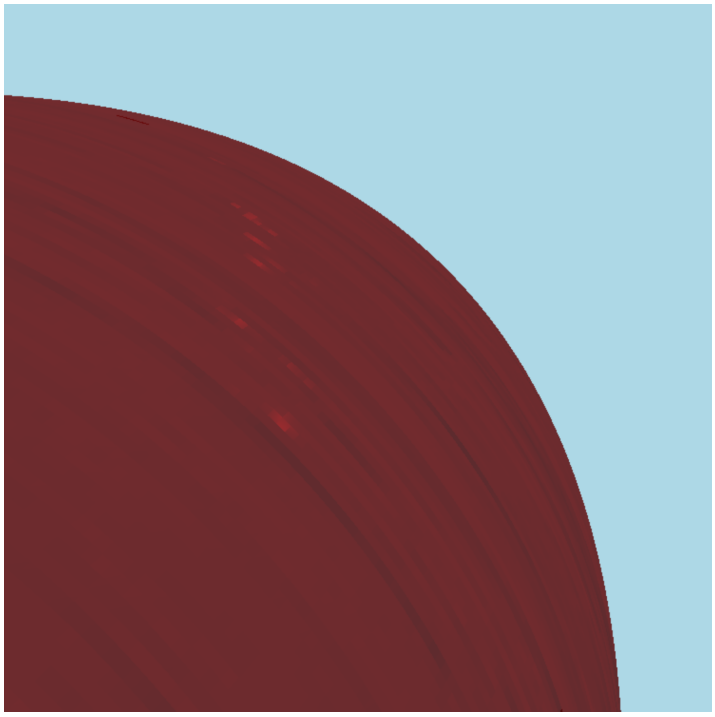

Рисунок 4.7 – Поверхность шара под маленьким углом с текстурами, наложенными по карте нормалей.

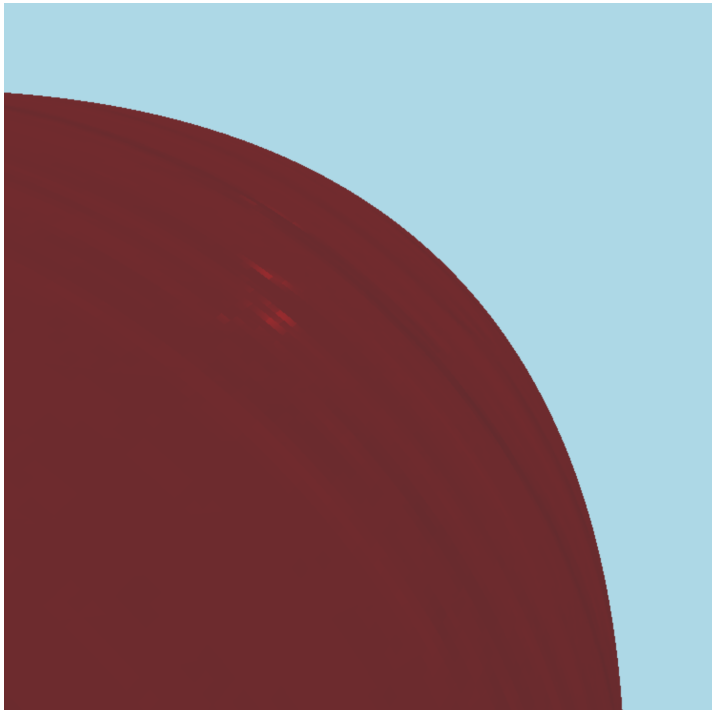

Рисунок 4.8 – Поверхность шара под маленьким углом с текстурами, наложенными по карте паралактического отображения.

Сравнивая полученные изображения можно с уверенностью сказать что улучшения, которые дает нам параллакс действительно видны, и в работах, требующих визуализации, более приближенной к реальности, такой алгоритм вполне подойдет. Что касается карты нормалей – поскольку там нет математических вычислений и возмущение полностью заложено в текстурную карту, результат получается более точным. Такой вариант подойдет, когда мы можем отдавать приоритет анализу больших данных, нежели вычислениям.

4.4 Анализ производительности

Для завершения исследования осталось проверить скорость выполнения алгоритом. Для этого рассмотрим сцены разной степени нагруженности по количеству шаров, а так же разные способы наложения текстур. Результаты работы программы приведены в таблице 4.1.

Таблица 4.1 – Производительность ПО при разном размере тканевой сетки.

Номер сцены	Карта высот. Время работы, мс.	Карта нормалей. Время работы, мс.	Карта параллакса. Время работы, мс.
1	301.1235	524.8413	429.1257
2	446.9992	671.1294	558.0016
3	512.6823	790.8529	602.9895
4	632.1282	993.8422	767.7887
5	728.1388	1053.046	950.8592
6	1068.7003	1514.2481	1168.2758
7	1388.5561	3143.9859	1401.0582

По результатам исследования времени работы ПО мы видим, что самым эффективным способом текстурирования является наложение карт высот, чуть медленнее – карт параллакса, и самым неудачным способом – использование карт нормалей.

Вывод

В данном разделе были представлены результаты работы ПО, проанализировано наложение текстур, а так же время работы программы. В итоге было выявлено, что самым менее затратным по времени, и при этом достаточно наглядным, является наложение текстур с помощью карт высот. Более визуально точным является наложение текстур по параллаксу, но это требует чуть больше времени. Самым визуально точным является текстурирование по картам нормалей, что требует самого большого количества времени, которое сильно растет с усложнением сцены. Это может вызвать больше проблемы с производительностью, поэтому такой метод стоит внедрять только с более проработанной обработкой больших данных.

Заключение

В ходе выполнения курсовой работы было разработано программное обеспечение, которое позволяет составлять сцены из различных сфер и накладывать на них текстуру с помощью карт разного типа. Для достижения цели были выполнены следующие задачи:

1) Было решено, какие параметры могут иметь сферы, а так же как они будут представляться;

2) Выбраны алгоритмы отображения сфер;

3) Определена модель освещение, отлично сочетающаяся с ранее выбранными алгоритмами отображения;

4) Спроектирована структура программного обеспечения, а так же выбраны подходящие способы представления данных;

5) Были выбраны средства реализации алгоритмов;

6) Созданно программное обечпечение на основе выбранных методов и алгоритмов;

7) Проведено исследование временных характеристик, и наложения текстур на основе созданного ПО;

Приложение А

Презентация к курсовой работе.