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

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Визуализация движения шарика на качелях»

Студент ИУ7-54Б _____ **А. Р. Алькина**
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель _____ **А. С. Кострицкий**
(Подпись, дата) (И.О.Фамилия)

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Одна из задач компьютерной графики — визуализация движения объектов на экране.

Целью курсовой работы является разработка программного обеспечения для визуализации движения шарика на качелях.

Для достижения поставленной цели необходимо решить следующие **задачи**:

— формализовать задачу в виде IDEF0-диаграммы;

— описать и проанализировать способы задания трёхмерных объектов, алгоритмы удаления невидимых рёбер и граней, алгоритмы построения теней, алгоритмы закрашивания;

— спроектировать программное обеспечение;

— выбрать средства реализации и реализовать спроектированное программное обеспечение;

— исследовать зависимость времени генерации кадра от количества источников света.

1 Аналитический раздел

В данном разделе выполнены формализация задачи в виде IDEF0-диаграммы, анализ способов задания трёхмерных объектов, дан обзор алгоритмов построения теней, удаления невидимых рёбер и граней, алгоритмов закрашивания.

1.1. Формализация задачи

Функциональная модель программы в нотации IDEF0 представлена на рисунке 1.1:

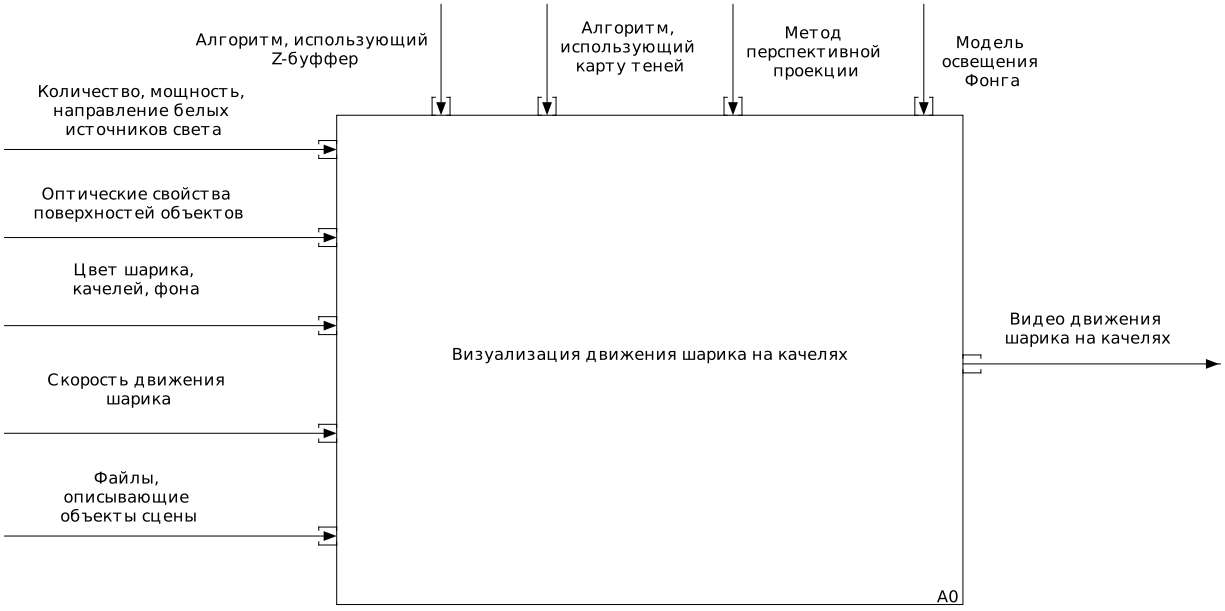

Рисунок 1.1 – Функциональная модель программы

1.2. Формализация объектов синтезируемой сцены

Исходя из ожидаемых результатов работы программы, кадр видео должен содержать следующие объекты:

— шарик, представляющий собой сферу;

— качели, имеющие форму параллелепипеда со скруглёнными углами (каждая качель имеет крепление).

Основные свойства шарика и качелей:

1. цвет поверхности;
2. выпуклость;
3. оптические свойства материала:
   — диффузная компонента освещённости k_d;
   — фоновая компонента освещённости k_a;
   — зеркальная составляющая освещённости k_s;
4. скорость движения.

Сцена содержит камеру, которая характеризуется свойствами:

1. положение в пространстве P;
2. вектор наблюдения V.

Сцена содержит направленные источники света, имеющие свойства:

1. вектор направления лучей D;
2. мощность S.

1.3. Анализ способов задания трёхмерных объектов

Существуют несколько способов моделирования трёхмерных объектов:

— аппроксимация поверхностей объекта многоугольниками (например, треугольниками) — данный тип называется **поверхностным** [1];

— задание объекта набором точек и соединяющих рёбер — это **каркасный** тип модели [1];

— формирование тела из отдельных простых геометрических объёмных элементов с помощью операций объединения, пересечения, вычитания и различных преобразований — модель **сплошных тел** [1].

Результат сравнения способов задания трёхмерных объектов приведен в таблице 1.1:

Критерий сравнения	**Каркасный тип**	**Поверхностный тип**	**Модель сплошных тел**
Структура	Список точек, список рёбер	Список точек, список полигонов	Список тел, отношения, их связываюшие
Возможность моделирования произвольных объектов	Нет	Да	Да

Таблица 1.1 – Сравнение способов задания моделей

Вывод

Для выполнения курсовой работы был выбран поверхностный способ задания объектов, так как существуют программы для 3D-моделирования (Blender, Autocad), которые позволяют экспортировать модели в поверхностном типе.

1.4. Анализ алгоритмов удаления невидимых рёбер и поверхностей

Удаление невидимых рёбер и поверхностей разделяют на две задачи [1]:

1. удаление всех не лицевых поверхностей/рёбер;
2. удаление всех видимых поверхностей/рёбер, экранируемых другими телами.

Алгоритмы удаления невидимых рёбер и поверхностей разделяют на классы в зависимости от пространства работы: пространство объекта или пространство изображения [1, 2].

1. Алгоритмы, работающие в пространстве **объекта**:
   - алгоритм Робертса;
   - алгоритм Вейлера-Азертона.
2. Алгоритмы, работающие в пространстве **изображения**:
   - алгоритм плавающего горизонта;
   - алгоритм с Z-буфером;
   - алгоритм трассировки лучей;
   - алгоритм Варнока.

1.4.1 Алгоритм плавающего горизонта

Алгоритм плавающего горизонта чаще всего используется для удаления невидимых линий трехмерного представления функций, описывающих поверхность в виде: $F(x, y, z) = 0$.

Данный алгоритм не подходит для моделирования объектов, заданных каркасным, поверхностным способом или моделью сплошных тел [2].

1.4.2 Алгоритм, использующий Z-буфер

Алгоритм, использующий Z-буфер, в специальном буфере для каждого пикселя изображения хранит координату Z и значения RGB-составляющих цвета пикселя. Чтобы отсечь невидимые поверхности, используется сравнение значений в буфере с координатами текущего пикселя [1].

Формальное описание алгоритма, использующего Z-буфер:

1. буфер кадра заполняется фоновым значением интенсивности или цвета;
2. Z-буфер заполняется минимальным значением глубины;
3. каждый отрисовываемый многоугольник преобразуется в растровую форму в произвольном порядке;
4. для каждого $Pixel(x, y)$ в многоугольнике:
   4.1. вычисляется его глубина $z(x, y)$;
   4.2. если $z(x, y) > Z_{buf}(x, y)$, то атрибут этого многоугольника (интенсивность, цвет и т. п.) записывается в буфер кадра и $Z_{buf}(x, y)$ заменяется на $z(x, y)$;

При использовании алгоритма Z-буфера визуализация пересечений сложных объектов не требует дополнительных модификаций. Однако, чтобы хранить два буфера кадра, необходим большой объём памяти [1, 3].

1.4.3 Алгоритм трассировки лучей

Главная идея, составляющая основу алгоритма трассировки лучей, заключается в том, что наблюдатель видит объекты посредством испускаемого неким источником луча света, который падает на объект и затем доходит до наблюдателя, отражаясь, преломляясь или каким-нибудь иным способом.

Для более эффективной работы алгоритма лучи отслеживаются от наблюдателя к объекту.

Итак, из позиции наблюдателя P через каждый пиксель изображения испускается луч и находится его точка пересечения с поверхностью сцены. Лучи, выпущенные из глаза, называются **первичными** [4].

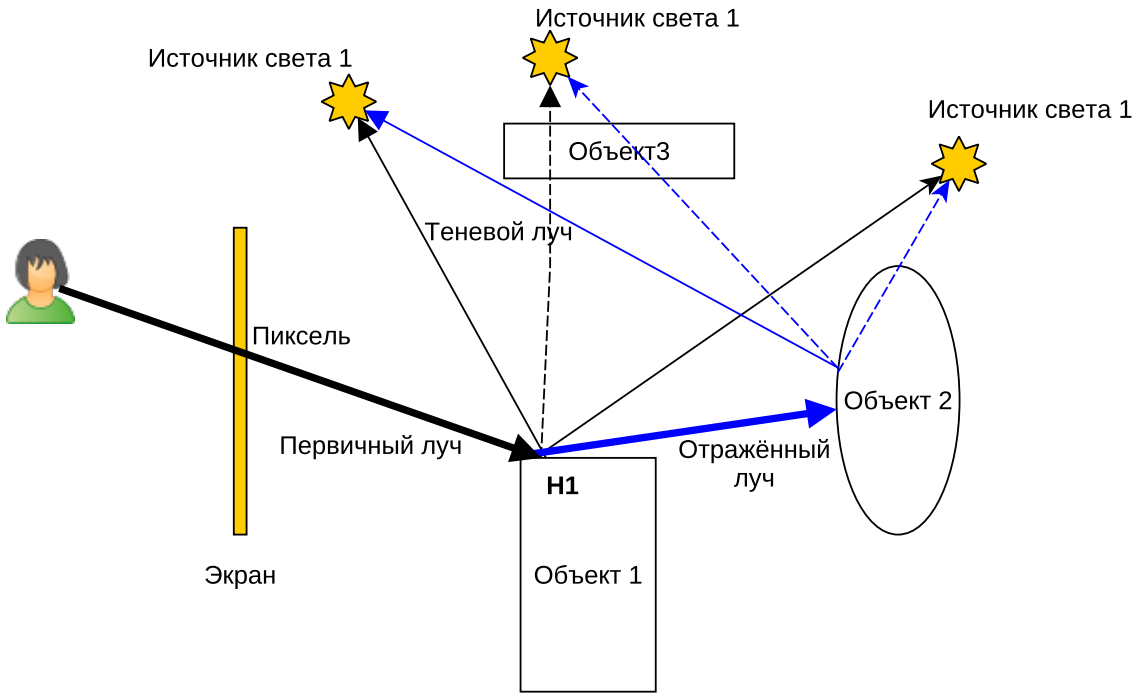

Рисунок 1.2 – Визуализация траектории лучей в алгоритме трассировки лучей

Пусть первичный луч пересекает объект 1 в точке H1.

Шаги работы алгоритма трассировки лучей:

1. необходимо определить для каждого источника освещения, видна ли из него эта точка;
2. для каждого источника света до него испускается **теневой луч** из точки H1 [4];
   2.1. если теневой луч пересекается с какими-либо объектами, расположенными между точкой и источником света, точка H1 находится в тени и освещать её не надо;
   2.2. иначе интенсивность считается по некоторой **модели** (например, Фонг или Гуро);
3. освещение со всех видимых (из точки H1) источников света складывается;
4. если материал объекта 1 имеет отражающие свойства, из точки H1 испускается отраженный луч и для него вся процедура трассировки рекурсивно повторяется.

Модели освещения (отражения) используются для имитации световых эффектов в компьютерной графике, когда свет аппроксимируется на основе физики света.

Существует много моделей освещения, и их можно классифицировать по критериям локальности, использования физических характеристик модели и гладкости модели [5].

1. По критерию локальности выделяют [5]:
   — локальные модели: учитывается только локальная геометрия объектов. Каждый объект рассматривается независимо от других, без учета их взаимодействия. Такие модели также называются объектно-ориентированными;

— глобальные модели: учитываются эффекты взаимодействия между объектами, такие как отражение света между поверхностями. Это приводит к более реалистичным результатам, но требует больших вычислительных ресурсов.

2. По критерию использования физических характеристик модели выделяют [5]:

   — физические модели: аппроксимируют свойства реальных объектов, учитывая особенности поверхностной структуры и поведение материалов. Например, моделирование кожи или песчинок песка;

   — эмпирические модели: параметры моделей могут не иметь физической интерпретации, но подбор этих параметров позволяет получать реалистичные изображения.

3. По критерию гладкости модели выделяют [5]:

   — гладкие модели: поверхность представляется как гладкая. Например, моделирование пластика;

   — негладкие модели: поверхность рассматривается как набор гладких микрозеркал. Например, моделирование металлических поверхностей.

В более сложных моделях освещения приходится также учитывать преломляющие свойства материала, то есть прозрачность поверхностей, что приводит к появлению преломлённых лучей.

1.4.4 Алгоритм Робертса

Суть алгоритма Робертса заключается в следующем: объекты проецируются на плоскость экрана, где каждое ребро последовательно сравнивается со всеми гранями.

Варианты взаимного расположения ребра и грани [1]:

— проекции ребра и грани не пересекаются, грань не заслоняет ребро;

— проекции ребра и грани пересекаются, но ребро лежит ближе грани, грань не заслоняет ребро;

— проекции ребра и грани пересекаются, и ребро лежит дальше грани, грань заслоняет всё ребро или часть ребра;

— ребро пересекает грань.

Определение видимости ребра [1]:

1. Если ребро не перекрыто гранью, оно проверяется с последующей гранью.

2. Если ребро полностью перекрыто, оно считается невидимым и проверка с остальными гранями не выполняется.

3. Если грань перекрывает часть ребра, ребро разделяется на видимые и невидимые части, и только видимые части продолжаются проверяться с другими гранями.

1.4.5 Алгоритм Варнока

Суть работы алгоритма Варнока заключается в анализе областей на экране на наличие в них видимых элементов.

Ниже представлены основные шаги алгоритма [6].

1. Если в текущем окне, получившемся в результате разбиения предыдущего окна, отсутствует объект или его содержимое просто для визуализации, то в зависимости от отсутствия/-присутствия части объекта происходит закрашивание цветом фона/объекта.

2. Разбиение продолжается до тех пор, пока содержимое фрагментов окна не станет достаточно простым для визуализации или их размеры не достигнут пределов разрешения.

В простейшей оригинальной версии алгоритма окно разбивается на четыре фрагмента (подокна), затем подокно, в котором есть содержимое, разбивается до достижения предела разрешения.

Выделяют следующие типы многоугольников для текущего окна:

— внешний — многоугольник располагается полностью вне окна;

— внутренний — многоугольник полностью располагается внутри окна;

— пересекающий — многоугольник пересекает как минимум одну границу окна;

— охватывающий — окно целиком располагается внутри многоугольника.

Рисунок 1.3 иллюстрирует данные типы многоугольников.

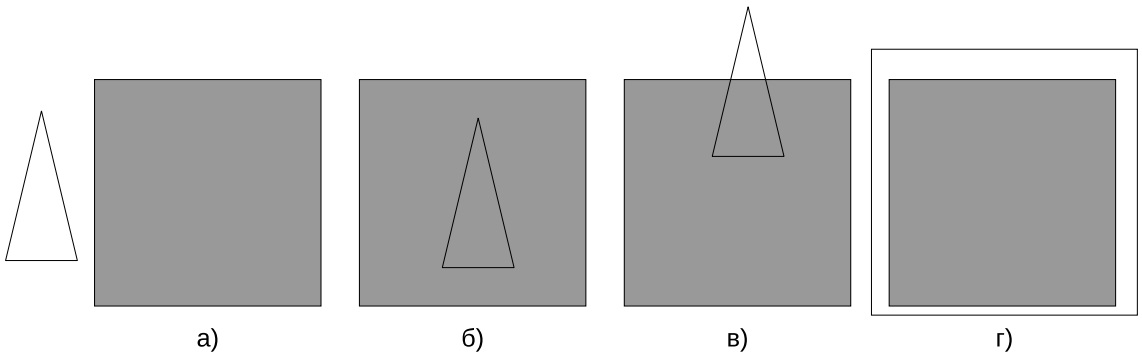

Рисунок 1.3 – Типы многоугольников: а — внешний, б — внутренний, в — пересекающий, г — охватывающий

Правила обработки для каждого окна [6]:

1. если в сцене присутствуют только внешние для текущего окна многоугольники, то это окно считается пустым — оно не разбивается и изображается с фоновой интенсивность или цветом;

2. если в сцене один внутренний многоугольник, то площадь вне многоугольника изображается с фоновой интенсивностью или цветом, а площадь внутри многоугольника, заполняется соответствующим ему цветом или интенсивностью;

3. если в сцене располагается пересекающий многоугольник, то площадь вне многоугольника изображается с фоновой интенсивностью или цветом, а площадь внутри многоугольника, которая располагается внутри окна, заполняется соответствующим ему цветом или интенсивностью;

4. если в сцене присутствует один охватывающий многоугольник, то окно изображается с интенсивностью или цветом соответствующими охватывающему многоугольнику;

5. если в сцене присутствует хотя бы один охватывающий многоугольник, расположенный ближе других к точке наблюдения, то окно заполняется интенсивностью или цветом, соответствующими охватывающему многоугольнику.

Последний пункт позволяет решать задачу удаления невидимых поверхностей.

1.4.6 Алгоритм Вейлера-Азертона

Алгоритм Вейлера-Азертона по сути является модификацией алгоритма Варнока. Вейлер и Азертон предложили перейти от прямоугольных разбиений к разбиениям вдоль границ многоугольников. Для этого был применён алгоритм отсечения многоугольников [1].

Алгоритм Вейлера-Азертона содержит несколько шагов.

1. Предварительно проводится сортировка по глубине.
2. Осуществляется отсечение по границе ближайшего к точке наблюдения многоугольника, называемое сортировкой многоугольников на плоскости.
3. Многоугольники, экранируемые более близкими к точке наблюдения многоугольниками, удаляются.
4. Если необходимо, то проводится рекурсивное разбиение и новая сортировка.

Вывод

В таблице 1.2 представлено сравнение алгоритмов удаления невидимых рёбер и поверхностей.

Критерий сравнения	**Алгоритм плавающего горизонта**	**Алгоритм Варнока**	**Алгоритм Робертса**	**Алгоритм Вейлера-Азертона**	**Алгоритм, использующий Z-буфер**	**Алгоритм трассировки лучей**
Пространство работы	Пространство изображения	Пространство изображения	Пространство объектов	Пространство объектов	Пространство изображения	Пространство изображения
Алгоритмическая сложность (количество объектов n, количество пикселей N)	$n \cdot N$	$n \cdot N$	n^2	n^2	$n \cdot N$	$n \cdot N$
Возможность использовать факт когерентности	Нет	Да	Да	Да	Да	Да
Возможность вычислять интенсивность закраски	Нет	Да	Да	Да	Да	Да
Возможность построения теней	Нет	Нет	Нет	Нет	Да	Да
Способ задания объектов	Только аналитически	Любой	Только выпуклые объекты	Объекты - плоские многоугольники	Любой	Любой

Таблица 1.2 – Сравнение алгоритмов удаления невидимых рёбер и поверхностей

Для выполнения курсовой работы был выбран алгоритм, использующий Z-буфер, так как он подходит для решения поставленных задач и обладает линейной оценкой трудоёмкости, также позволяет решать задачу построения теней.

1.5. Анализ алгоритмов построения теней

Выделяют три класса алгоритмов построения теней [7].

1. Вычисление затенения в процессе преобразования в растровый вид.

   К **первому классу** относятся алгоритмы, основанные на принципе отбрасывания лучей, и алгоритмы, которые включают метод трассирования лучей как составную часть.

2. Разделение поверхностей объекта на теневые и нетеневые площади, предшествующее преобразованию в растровый вид.

   Второй класс алгоритмов использует известные алгоритмы определения видимых поверхностей. При этом используются двухпроходные реализации алгоритмов построчного сканирования или Z-буфера.

3. Включение значения теней в данные, описывающие объект.

   Алгоритмы, относящиеся к **третьему классу**, заносят в базу данных информацию о «затененных» полигональных областях.

Так как в разделе анализа алгоритмов удаления невидимых рёбер и поверхностей был выбран алгоритм Z-буфера, то будет рассмотрен только алгоритм построения теней с использованием карт теней, который сочетается с алгоритмом Z-буфера [7, 8].

1.5.1 Алгоритм построения карты теней

Для построения теней с использованием алгоритма Z-буфера требуется два прохода: один относительно источника света и другой относительно наблюдателя. Для этого используется отдельный «теневой» буфер глубины [7].

1. Во время первого прохода определяются точки, видимые со стороны источника света.
2. Во втором проходе сцена отображается с позиции наблюдателя с учетом того, что точки, которые были невидимы со стороны источника света, находятся в тени.

Вывод

Для построения теней был выбран алгоритм, использующий карты теней, так как он использует алгоритм Z-буфера, выбранный для удаления невидимых рёбер и поверхностей.

1.6. Анализ алгоритмов закрашивания

Среди методов закрашивания выделяют простую закраску, закраску методом Гуро и закраску методом Фонга [1].

Задача закраски состоит в определении интенсивности цвета объекта или части объекта I в зависимости от параметров объектов сцены (раздел 1.2):

— оптических свойств объекта (k_d, k_a, k_s);

— вектора наблюдения V;

— направления D и мощности P источника света.

1.6.1 Простая закраска

Для изображаемой грани объекта вычисляется один уровень интенсивности освещения, который и используется для закраски всего объекта [1]. Этот уровень интенсивности освещения вычисляется по **закону косинусов Ламберта**: интенсивность **диффузно отраженного** света пропорциональна косинусу угла между направлением света и нормалью к поверхности n [1, 5]:

$$I = Pk_d(D, n). \tag{1.1}$$

Необходимо также учитывать отражённый от других объектов свет — **рассеянный** свет — и **зеркальную** составляющую света. В простых моделях можно использовать **эмпирическую модель Фонга** [1, 5]:

$$I = Pk_a + P(k_d(D, n) + k_s(D_n, V)^m)/(d + K) \tag{1.2}$$

где D_n — вектор отражённого относительно нормали n луча, m — степень, аппроксимирующая пространственное распределение зеркально отраженного света, K — произвольная постоянная, d — расстояние от объекта до источника света.

Для того чтобы применить данную модель освещения, необходимо вычислить нормаль и вектор отражения в каждой точке поверхности. Такой подход будет создавать изображение, состоящее из отдельно видимых полигонов [1, 5].

Этот эффект продемонстрирован на рисунке 1.4.

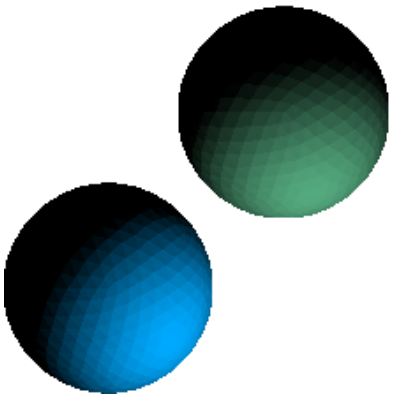

Рисунок 1.4 – Эффект отсутствия сглаживания

1.6.2 Закраска методом Гуро

Метод закраски Гуро использует аппроксимацию нормалей к поверхности в вершинах многоугольников и билинейную интерполяцию интенсивности освещения каждого пикселя сканирующей строки [5].

Интенсивность в точке (x_3, y_3) на рисунке 1.5 — I_3 — вычисляется в несколько шагов.

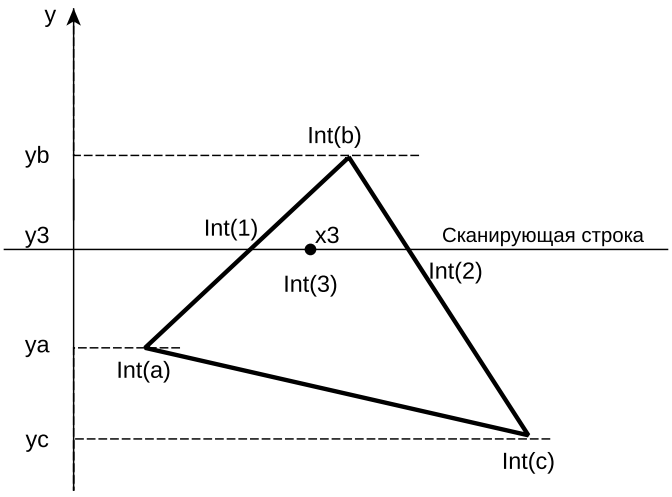

Рисунок 1.5 – Интерполяция интенсивности

1. Необходимо ввести следующие векторы для интенсивностей в точках:
$$I^1 = \begin{bmatrix} I_b \\ I_c \end{bmatrix}, I^2 = \begin{bmatrix} I_b \\ I_c \end{bmatrix}, I^3 = \begin{bmatrix} I^1 \\ I^2 \end{bmatrix} \tag{1.3}$$
2. Также необходимы следующие векторы для расчёта коэффициентов при интенсивностях:
$$y^1 = \begin{bmatrix} \frac{y_3 - y_a}{y_b - y_a} \\ \frac{y_b - y_3}{y_b - y_a} \end{bmatrix}, y^2 = \begin{bmatrix} \frac{y_3 - y_c}{y_b - y_c} \\ \frac{y_b - y_3}{y_b - y_c} \end{bmatrix} \tag{1.4}$$
3. Тогда с учётом формул 1.3 и 1.4 можно вычислить интенсивность $Int(3)$:
$$I_3 = I^1 \cdot y^1 \cdot \frac{x_b - x_3}{x_b - x_a} + I^2 \cdot y^2 \cdot \frac{x_3 - x_a}{x_b - x_a} \tag{1.5}$$

Для цветных объектов интерполируется каждая компонента цвета.

Лучше всего закраска Гуро выглядит в сочетании с простой моделью освещения с диффузным отражением. Блики при зеркальном отражении могут выглядеть нереалистично [1, 5].

1.6.3 Закраска методом Фонга

Закраска методом Фонга потребляет гораздо больше вычислительных ресурсов, нежели закраска методом Гуро, однако объекты выглядят более реалистично, в частности качественнее отображаются зеркальные блики [5].

При закраске Фонга аппроксимация кривизны поверхности осуществляется сначала в вершинах многоугольников путём аппроксимации нормали к вершине. То есть по нормалям к грани определяются нормали к вершинам (в каждой точке закрашиваемой гра-

ни интерполируется вектор нормали). Таким образом интерполируется не значение интенсивности каждого пикселя сканирующей строки, а нормали.

Интенсивность в точке (x_3, y_3) на рисунке 1.5 — I_3 — вычисляется в несколько шагов.

1. Необходимо ввести векторы (n_i — нормаль к точке i):

$$a^1 = \begin{bmatrix} a \\ (1-a) \end{bmatrix}, a \in \{u, w, t\} \tag{1.6}$$

$$n^1 = \begin{bmatrix} n_2 \\ n_1 \end{bmatrix}, n^2 = \begin{bmatrix} n_1 \\ n_3 \end{bmatrix}, n^3 = \begin{bmatrix} n^1 \\ n^2 \end{bmatrix} \tag{1.7}$$

2. С использованием рисунка 1.5 получается:

$$n_1 = u^1 \cdot n^1, n_2 = u^2 \cdot n^2 \tag{1.8}$$

3. В итоге интенсивность в точке $Int(3)$ вычисляется по формуле:

$$I_3 = t^1 \cdot n \tag{1.9}$$

где $u = \frac{y_a - y_2}{y_1 - y_2}, w = \frac{y_1 - y_b}{y_1 - y_3}, t = \frac{x_p - x_a}{x_b - x_a}, n = \begin{bmatrix} n_1 \\ n_2 \end{bmatrix}$.

Вывод

В таблице 1.3 представлено сравнение алгоритмов закраски:

Критерий сравнения	Простая модель освещения	Закраска методом Гуро	Закраска методом Фонга
Возможность использования когерентности изображения	Нет	Да	Да
Возможность построения кривых гладких поверхностей	Нет	Частично	Да

Таблица 1.3 – Сравнение алгоритмов закраски

Для курсовой работы был выбран метод закраски Гуро в сочетании с моделью освещения Фонга, так как он даёт возможность построения кривых гладких поверхностей.

Вывод

В данном разделе была формализована поставленная задача, была проведена формализация объектов сцены, был дан обзор способов задания трёхмерных объектов, алгоритмов удаления невидимых рёбер и повехностей, алгоритмов построения теней, алгоритмов закраски.

Был выбран поверхностный способ задания объектов, алгоритм Z-буфера для удаления невидимых рёбер и поверхностей с использованием карт теней и метод закраски Гуро с моделью освещения Фонга.

2 Конструкторская часть

В данном разделе представлены требования к программному обеспечению, даны описания алгоритмов, выбранных для решения поставленной задачи, приведены выбранные структуры данных.

2.1. Требования к программному обеспечению

Программное обеспечение должно предоставлять следующие возможности для пользователя:

— изменение скорости движения шарика;

— изменение цвета шарика, качелей и фона;

— добавление направленных источников света;

— изменение оптических свойств шарика и качелей.

Выделяются следующие требования к программному обеспечению:

— входные данные для построения объектов задаются в файлах;

— кадры должны генерироваться со скоростью 30 кадров в секунду [1].

[1] Для приложений визуализации комфортный уровень fps — до 40 кадров в секунду [9]

2.2. Функциональная модель программы первого уровня в нотации idef0

Функциональная модель программы в нотации IDEF0 представлена на рисунке 2.1:

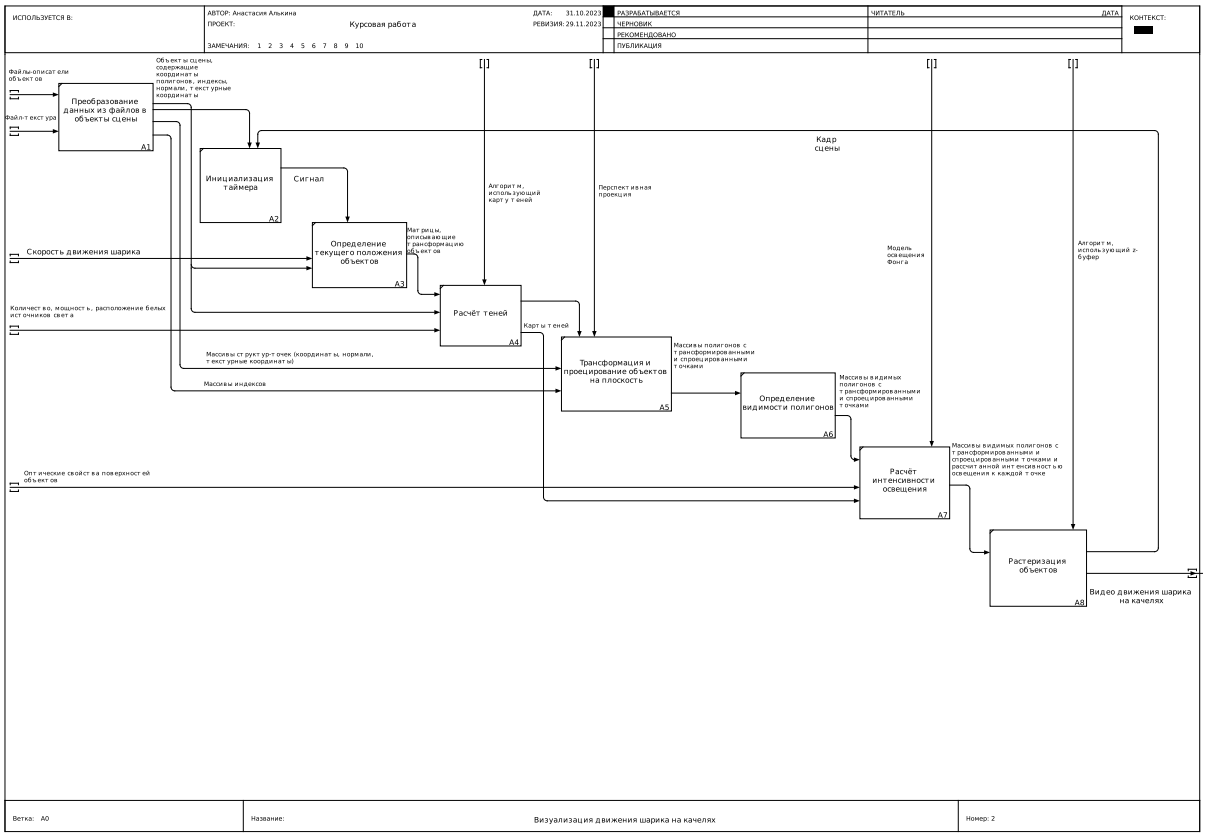

Рисунок 2.1 – Функциональная модель первого уровня

2.3. Структуры данных

В таблице 2.1 представлены описания структур данных для объектов:

Объект	Структура данных
Координаты точки	Трёхкоординатный вектор
Текстурные координаты	Двухкоординатный вектор
Нормаль	Трёхкоординатный вектор
Вершина	Структура, содержащая поля: координаты точки, текстурные координаты, нормаль
Свойства материала	Структура, содержащая три вещественных поля для диффузной, рассеянной и зеркальной составляющей
Карта теней	Структура, содержащая буфер теней, его высоту и ширину
Объект сцены	Структура, содержащая массив вершин, индексов вершин, структуру свойства материала
Объекты сцены	Список, содержащий объекты сцены
Камера	Структура, содержащая матрицу вида
Источник света	Структура, содержащая матрицу источника света, интенсивность, направление, позицию, цвет источника света
Источники света	Список, содержащий источники света

Таблица 2.1 – Структуры данных объектов

2.4. Схема алгоритма, использующего Z-буфер

На рисунке 2.2 приведена схема алгоритма, использующего Z-буфер.

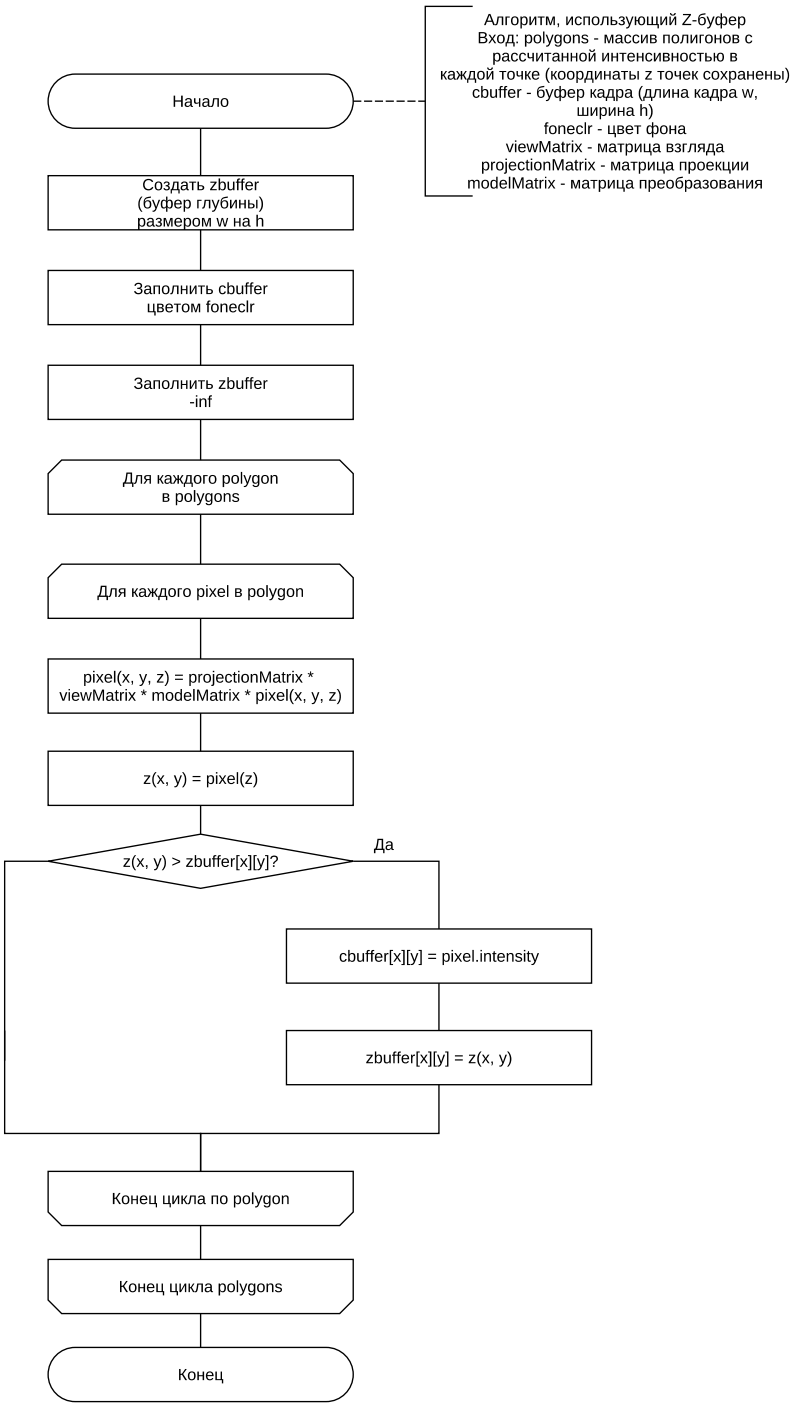

Рисунок 2.2 – Схема алгоритма, использующего Z-буфер

2.5. Схема алгоритма построения карты теней

На рисунке 2.3 приведена схема алгоритма построения карты теней.

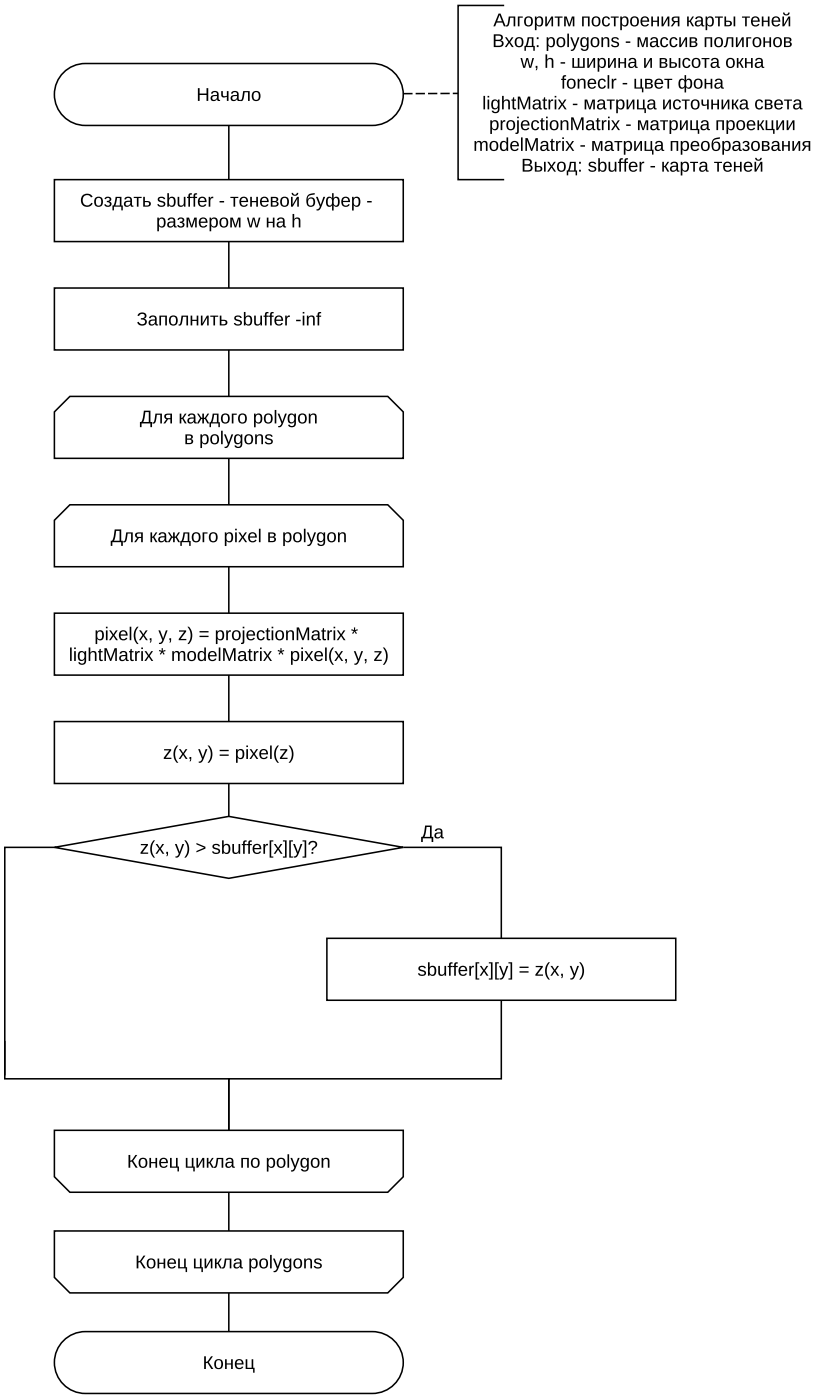

Рисунок 2.3 – Схема алгоритма построения карты теней

2.6. Схема алгоритма закраски Гуро с моделью освещения Фонга

На рисунке 2.4 приведена схема алгоритма построения карты теней.

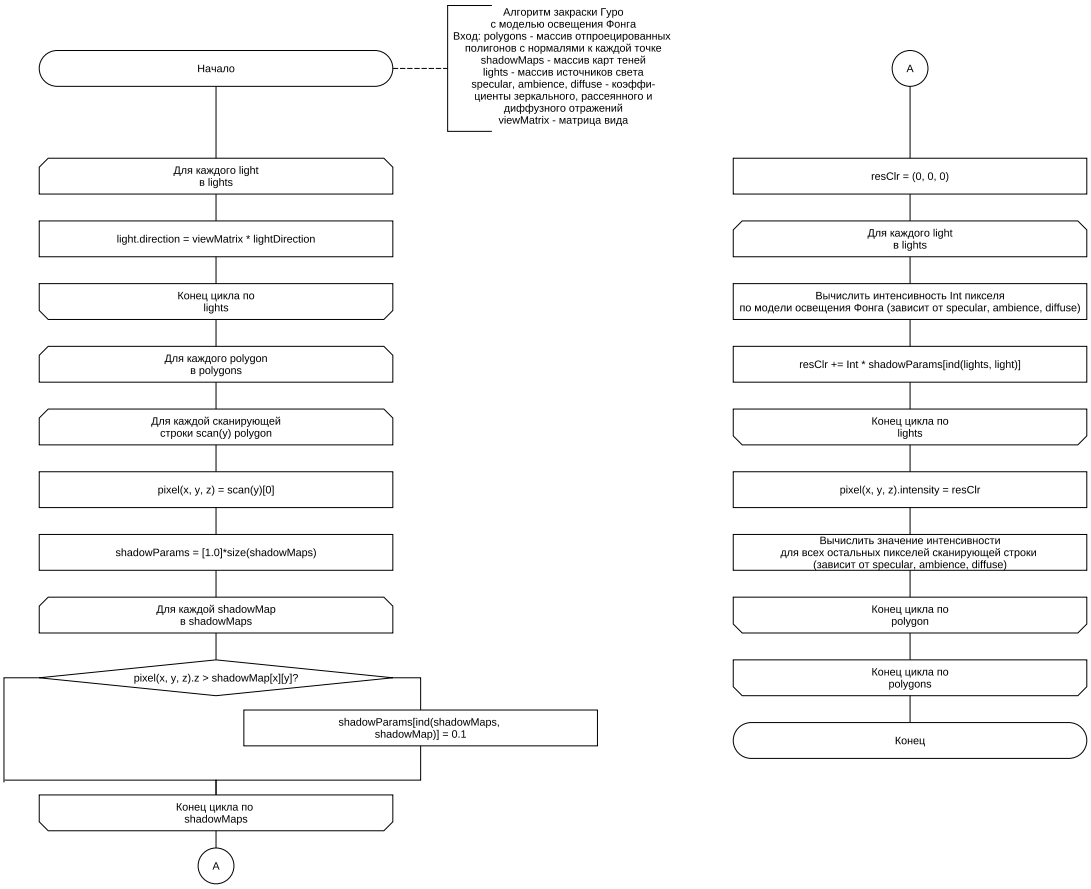

Рисунок 2.4 – Схема алгоритма закраски Гуро с моделью освещения Фонга

Вывод

В данном разделе представлены схема алгоритма, использующего Z-буфер, алгоритма построения карты теней, алгоритма закраски Гуро с моделью освещения Фонга, также представлены требования к разрабатываемому программному обеспечению и обеспечиваемые им возможности, описаны структуры данных и приведена модель программы первого уровня в нотации idef0.

3 Технологическая часть

В данном разделе представлено обоснование выбранных средств программной реализации, описаны основные моменты программной реализации и методики тестирования.

3.1. Средства реализации

В качестве языка программирования для реализации курсовой работы был выбран язык C++ с использованием OpenGL Framework и Qt Framework, так как язык C++ предоставляет необходимые средства для реализации выбранных в результате проектирования алгоритмов. OpenGL Framework используется для организации вычислений на графическом процессоре.

В качестве среды разработки выбрана Qt Creator, так как она поддерживает язык C++ и имеет встроенные средства для создания пользовательского интерфейса.

Для тестирования компонент программного обеспечения был выбран модуль Qt Test, так как он интегрирован в Qt Framework и не требует установки дополнительного программного обеспечения.

3.2. Примеры реализации программного обеспечения

3.2.1 Примеры описания структур данных

В листинге 3.1 приведено описание структуры данных vertex_t, которая описывает вершину объекта сцены.

Листинг 3.1 – Структура, описывающая вершину объекта

```
struct vertex_t
{
    vertex_t() {}
    vertex_t(QVector3D p, QVector2D t, QVector3D n)
        : position(p), textcoord(t), normal(n) {}
    QVector3D position;
    QVector2D textcoord;
    QVector3D normal;
};
```

В листинге 3.2 приведено описание класса ShadowBuffer для реализации карт теней.

Листинг 3.2 – Класс для реализации карты теней

```
class ShadowBuffer
{
public:
    ShadowBuffer(int w, int h) : width(w), height(h), shadowBuff(nullptr) {};
    ~ShadowBuffer() { delete shadowBuff; }
    QOpenGLFramebufferObject *shadowBuff;
    quint64 width;
    quint64 height;
    bool isUsed;
};
```

В листинге 3.3 приведено описание структуры materialProperties_t, хранящей оптические свойства материала объекта.

Листинг 3.3 – Структура, описывающая оптические свойства материала объекта

```
struct materialProperties_t
{
    float ambParam;
    float diffParam;
    float specParam;
};
```

В листинге 3.4 приведено описание класса BaseObject, представляющего изображаемый на сцене объект.

Листинг 3.4 – Класс, представляющий объект сцены

```
class BaseObject
{
public:
    BaseObject() = delete;
    BaseObject(const QString &filename,
               const QString &texturePath,
               materialProperties_t mat);
    BaseObject(const QVector<vertex_t> &vertexes,
               const QVector<GLuint> &indexes,
               const QString &texturePath,
               materialProperties_t mat);
    ~BaseObject();

    void rotate(const QQuaternion &r);
    void translate(const QVector3D &t);
    void scale(const float &s);
    void setGlobalTransform(const QMatrix4x4 &gt);
    void resetTransformations();

    void draw(QOpenGLShaderProgram *program,
              QOpenGLFunctions *functions);
    materialProperties_t getMaterial();
    void changeTexture(const QString &texturePath);
    void changeMaterial(int n, float val);
protected:
    void free();
    void loadFromFile(const QString &filename);
    void init(const QVector<vertex_t> &vertexes,
              const QVector<GLuint> &indexes,
              const QString &texturePath);
private:
    QOpenGLBuffer vertexesBuffer;
    QOpenGLBuffer indexesBuffer;
    QString texturePath;
    QOpenGLTexture *texture;

    QQuaternion rotation;
    QVector3D transposition;
    float scaleFactor;
    QMatrix4x4 globalTransform;

    QVector<GLuint> indexes;
    QVector<vertex_t> vertexes;

    materialProperties_t material;
};
```

3.2.2 Примеры реализации функций

В листинге 3.5 приведена реализация функции-обработчика сигнала таймера, при помощи которой происходит генерация очередного кадра.

Листинг 3.5 – Обработчик сигнала таймера

```
void MainWindow::handleTimerSignal()
{
    sphereMovement(matrixes[0], sphereSpeed);
    oglw->getObject(0)->setGlobalTransform(matrixes[0]);

    for (int i = 0; i < OBJ_NUMBER-1; i++)
        if (flags[i]) {
            swingSpeeds[i] = 0.0;
            flags[i] = false;
            timers[i]->start(sleep);
        }

    for (int i = 0; i < OBJ_NUMBER-1; i++)
        swingsMovement(matrixes[i+1], degrees[i],
                       RADIUS, swingSpeeds[i],
                       &drives[i], oglw->getObject(i+1),
                       &flags[i]);

    oglw->update();
}
```

В листингах 3.6 — 3.7 приведена реализация функции для создания текстур карты теней.

Листинг 3.6 – Функция для создания текстур карты теней

```
void GLWidget::getShadowMap(int ind, int textInd)
{
    shadowBuffers[ind]->shadowBuff->bind();
    context()->functions()->glViewport(0, 0, shadowBuffers[ind]->width, shadowBuffers[ind]->height);
    context()->functions()->glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    shadowShaderProgram.bind();
    shadowShaderProgram.setUniformValue("qt_ProjectionLightMatrix", projectionLightMatrix);
    shadowShaderProgram.setUniformValue("qt_ShadowMatrix", lights[ind]->getLMatrix());
    for (auto obj : objects)
        obj->draw(&shadowShaderProgram, context()->functions());
    shadowShaderProgram.release();
```

Листинг 3.7 – Продолжение листинга 3.6

```
    GLuint shadowTexture = shadowBuffers[ind]->shadowBuff->texture();
    context()->functions()->glActiveTexture(textInd);
    context()->functions()->glBindTexture(GL_TEXTURE_2D, shadowTexture);
```

В листингах 3.8–3.10 приведены основные вершинный и фрагментный шейдеры, участвующие в отрисовке изображения. Фрагментный шейдер обеспечивает закраску пикселей в соответствии с выбранными алгоритмами.

Листинг 3.8 – Вершинный шейдер

```
attribute highp vec4 qt_Vertex;
attribute highp vec2 qt_MultiTexCoord0;
attribute highp vec3 qt_Normal;
uniform highp mat4 qt_ProjectionMatrix;
uniform highp mat4 viewMatrix;
uniform highp mat4 modelMatrix;

uniform highp mat4 qt_ProjectionLightMatrix;
uniform highp mat4 shadowMatrixes[10];
uniform int numberShadows;

varying highp vec4 qt_VertexesLightMatrix[10];

varying highp vec4 qt_Vertex0;
varying highp vec2 qt_TexCoord0;
varying highp vec3 qt_Normal0;
varying highp mat4 qt_viewMatrix;

void main(void)
{
    mat4 mvMatrix = viewMatrix * modelMatrix;
    gl_Position = qt_ProjectionMatrix * mvMatrix * qt_Vertex;
    qt_TexCoord0 = qt_MultiTexCoord0;
    qt_Normal0 = normalize(vec3(mvMatrix * vec4(qt_Normal, 0.0)));
    qt_Vertex0 = mvMatrix * qt_Vertex;
    qt_viewMatrix = viewMatrix;

    for (int i = 0; i < numberShadows; i++)
    qt_VertexesLightMatrix[i] = qt_ProjectionLightMatrix * shadowMatrixes[i] * modelMatrix *
        qt_Vertex;
}
```

Листинг 3.9 – Фрагментный шейдер

```
struct Light {
    float power;
    vec3 color;
    vec4 pos;
    vec4 direction;
    int type;
};

uniform sampler2D qt_Texture0;
uniform sampler2D qt_ShadowMaps0[10];
uniform int numberLights;

uniform Light lights[10];
Light corLights[10];
vec3 tmp[10];
float resShadowParams[10];

uniform highp float specParam;
uniform highp float ambParam;
uniform highp float diffParam;

varying highp vec4 qt_Vertex0;
varying highp vec2 qt_TexCoord0;
varying highp vec3 qt_Normal0;
varying highp mat4 qt_viewMatrix;
varying highp vec4 qt_VertexesLightMatrix[10];

//float ShadowMapping(sampler2D map, vec2 coordinates, float cur_depth)
//{
// vec4 cur = texture2D(map, coordinates);
// float val = cur.x * 255.0 + 0.5f;
// return step(cur_depth, val);
//}

float SampleShadowMap(sampler2D map, vec2 coords, float compare)
{
    vec4 v = texture2D(map, coords);
    float value = v.x * 255.0f + (v.y * 255.0f + (v.z * 255.0f + v.w) / 255.0f) / 255.0f;
    return step(compare, value);
}

float SampleShadowMapLinear(sampler2D map, vec2 coords, float compare, vec2 texelsize)
{
    vec2 pixsize = coords / texelsize + 0.5f;
    vec2 pixfractpart = fract(pixsize);
    vec2 starttexel = (pixsize - pixfractpart) * texelsize;
    float bltexel = SampleShadowMap(map, starttexel, compare);
```

Листинг 3.10 – Продолжение листинга 3.10

```
    float brtexel = SampleShadowMap(map, starttexel + vec2(texelsize.x, 0.0f), compare);
    float tltexel = SampleShadowMap(map, starttexel + vec2(0.0f, texelsize.y), compare);
    float trtexel = SampleShadowMap(map, starttexel + texelsize, compare);

    float mixL = mix(bltexel, tltexel, pixfractpart.y);
    float mixR = mix(brtexel, trtexel, pixfractpart.y);

    return mix(mixL, mixR, pixfractpart.x);
}

float SampleShadowMapPCF(sampler2D map, vec2 coords, float compare, vec2 texelsize)
{
    float result = 0.0f;
    float spcfq = 1.0;
    for(float y = -spcfq; y < spcfq; y += 1.0f)
    for(float x = -spcfq; x < spcfq; x += 1.0f)
    {
        vec2 offset = vec2(x, y) * texelsize;
        result += SampleShadowMapLinear(map, coords + offset, compare, texelsize);
    }
    return result / 9.0f;
}

float CalcShadowAmount(sampler2D map, int i)
{
    vec3 value = qt_VertexesLightMatrix[i].xyz / qt_VertexesLightMatrix[i].w;
    value = value * vec3(0.5f) + vec3(0.5f);
    float offset = 2.0f * dot(qt_Normal0, lights[i].direction.xyz);

    return SampleShadowMapPCF(map, value.xy, value.z * 255.0f + offset, vec2(1.0f / 1024.0f));
}
```

В листингах 3.11–3.13 представлены листинги шейдеров, создающих карты теней.

Листинг 3.11 – Вершинный шейдер для создания карты теней

```
attribute highp vec4 qt_Vertex;
uniform highp mat4 qt_ShadowMatrix;
uniform highp mat4 qt_ModelMatrix;
uniform highp mat4 qt_ProjectionLightMatrix;
varying highp vec4 qt_Vertex0;

void main(void)
```

Листинг 3.12 – Продолжение листинга 3.11

```
{
    qt_Vertex0 = qt_ProjectionLightMatrix * qt_ShadowMatrix * qt_ModelMatrix * qt_Vertex;
    gl_Position = qt_Vertex0;
}
```

Листинг 3.13 – Фрагментный шейдер для создания карты теней

```
varying highp vec4 qt_Vertex0;
float tmp[11];

void main(void)
{
    float depth = qt_Vertex0.z / qt_Vertex0.w;
    depth = depth * 0.5f + 0.5f;
    tmp[0] = tmp[1] = depth;

    for (int i = 0; i < 3; i++) {
    int cur = i * 3 + 2;
    tmp[cur] = tmp[cur-2] * 255.0f;
    tmp[cur+1] = fract(tmp[cur]);
    tmp[cur+2] = floor(tmp[cur]) / 255.0f;
    }
    gl_FragColor = vec4(tmp[4], tmp[7], tmp[10], tmp[9]);
}
```

В листинге 3.14 приведена функция, создающая текстуры, содержащие карты теней.

Листинг 3.14 – Создание текстур карт теней

```
    if (!qFuzzyCompare(vec1.x(), vec2.x()))
        return false;
    if (!qFuzzyCompare(vec1.y(), vec2.y()))
        return false;
    if (!qFuzzyCompare(vec1.z(), vec2.z()))
        return false;
    if (!qFuzzyCompare(vec1.w(), vec2.w()))
        return false;

    return true;
}

void GLWidget::delLight(QVector4D direction, float power)
{
    for (int i = 0; i < lights.size(); i++)
        if (lights[i]->getUsed() == true && areVectorsEqual(lights[i]->getDirect(), direction)
                && qFuzzyCompare(lights[i]->getPower(), power)) {
            lights[i]->setUsed(false);
            break;
```

3.3. Тестирование

В курсовом проекте было реализовано модульное тестирование отдельных компонент разработанного программного обеспечения.

В качестве примера модульных тестов можно рассмотреть класс CameraTest.

В листингах 3.15 — 3.16 приведёны данный класс и пример модульного теста.

Листинг 3.15 – Класс, тестирующий камеру

```
TestCamera::TestCamera()
{

}

TestCamera::~TestCamera()
{

}


void TestCamera::testScale()
{
```

Листинг 3.16 – Пример теста

```
    auto actual = cam.getTransposition();
    auto expected = QVector3D(0.0, 1.0, 0.0);
    QVERIFY2(qFuzzyCompare(actual, expected), "Векторы не равны");
}
```

Результат прохождения тестов представлен в листинге 3.17:

Листинг 3.17 – Результат прохождения тестов

```
******** Start testing of CameraTest ********
Config: Using QtTest library 5.15.3, Qt 5.15.3
PASS : CameraTest::initTestCase()
PASS : CameraTest::test_case1()
PASS : CameraTest::cleanupTestCase()
Totals: 3 passed, 0 failed, 0 skipped, 0 blacklisted, 1ms
******** Finished testing of CameraTest ********
```

3.4. Описание интерфейса программы

Демонстрация интерфейса программы приведена на рисунке 3.1.

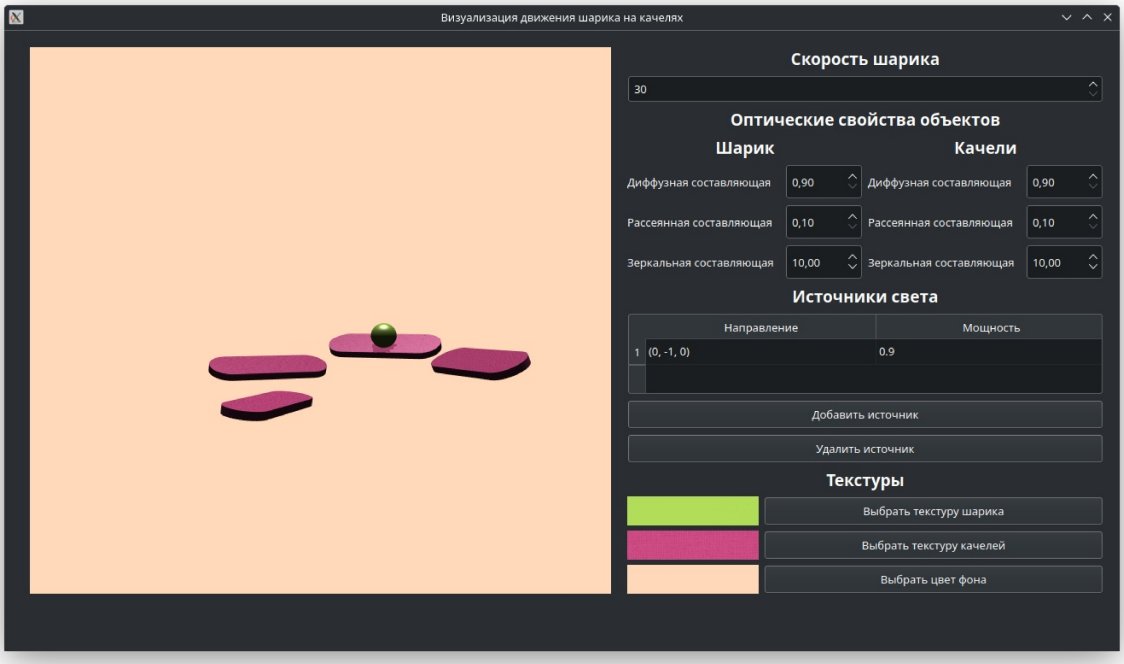

Рисунок 3.1 – Демонстрация интерфейса программы

Приложение можно запустить двумя способами: через среду Qt Creator или через терминал, выполнив команды qmake, make и ./sphere-mov-viz.

При запуске приложения визуализация запускается со значениями параметров по умолчанию.

Оптические свойства объектов можно варьировать с помощью

полей ввода со спиннером, скорость шарика также регулируется с помощью данных полей ввода.

Чтобы изменить текстуры шарика и качелей, необходимо нажать на кнопки, представленные на рисунке 3.2.

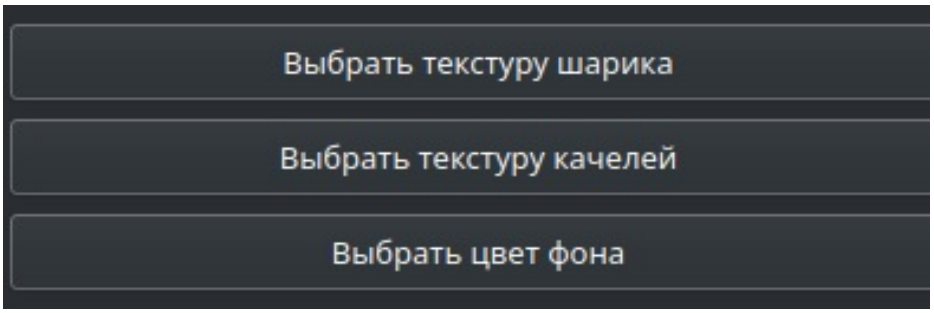

Рисунок 3.2 – Кнопки выбора текстур

Нажатие первых двух кнопок, представленных на рисунке 3.2, приводит к появлению окна выбора текстуры — рисунок 3.3.

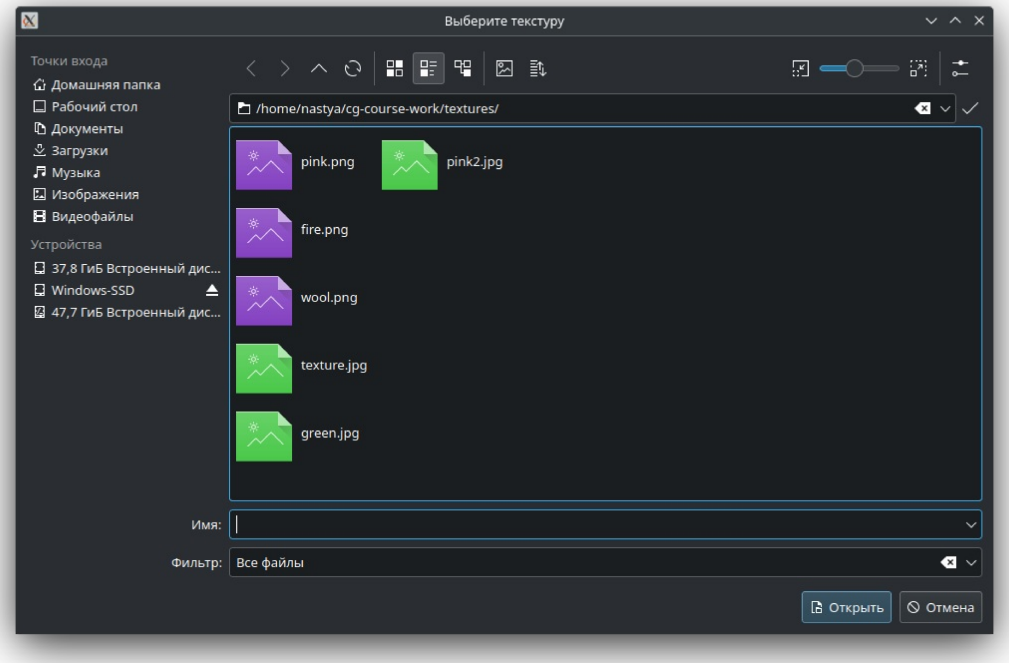

Рисунок 3.3 – Окно выбора текстуры

Добавление и удаление источника света происходит с помощью кнопок, представленных на рисунке 3.4.

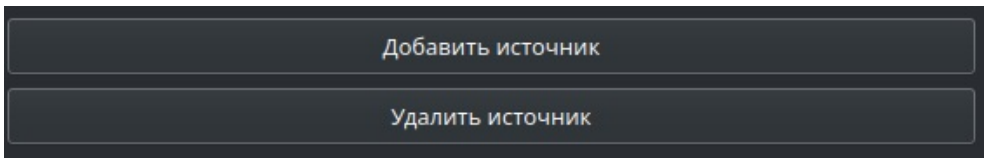

Рисунок 3.4 – Кнопки добавления и удаления источника света

Нажатие на кнопку выбора цвета фона приводит к появлению соответствующего окна — рисунок 3.5.

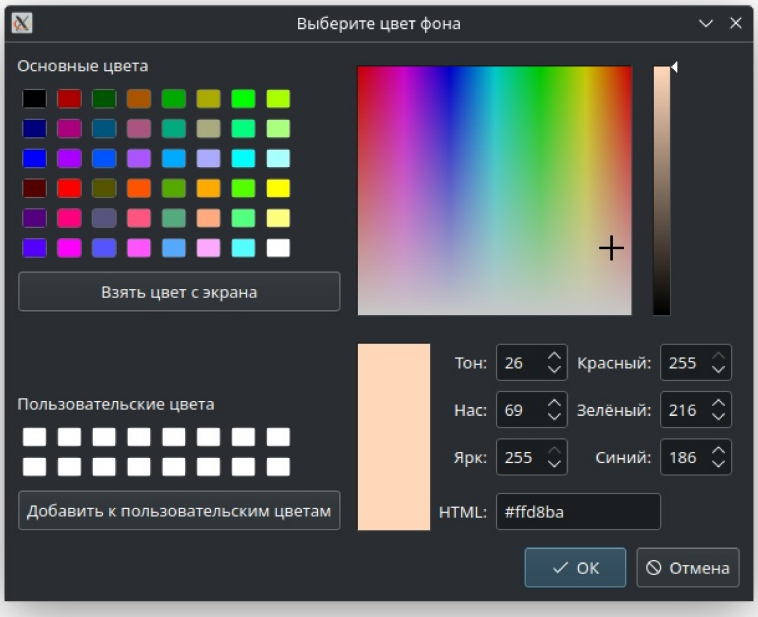

Рисунок 3.5 – Окно выбора цвета фона

Нажатие кнопки, отвечающей за добавление источника, приводит к появлению окна для ввода значений — рисунок 3.6.

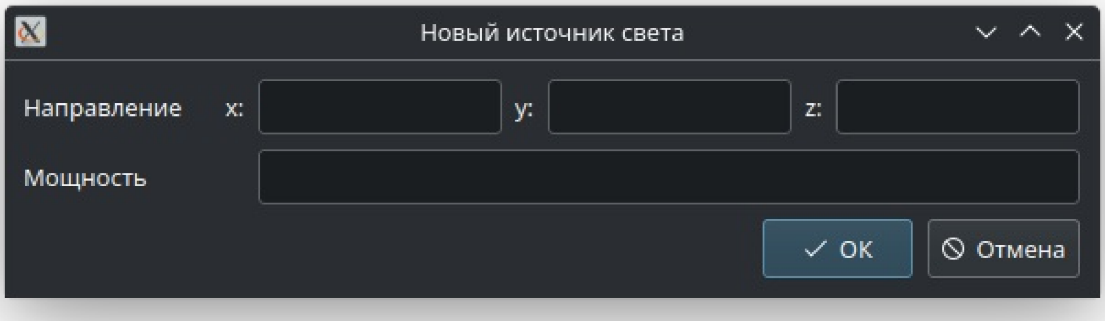

Рисунок 3.6 – Окно ввода значений

Удаление источника света производится нажатием на соответствующую строку таблицы и нажатием кнопки «Удалить источник».

Вывод

В данном разделе были выбраны средства реализации, приведены листинги кода и был описан интерфейс программы.

4 Исследовательская часть

В данном разделе представлено описание планирования исследования характеристик разработанного программного обеспечения и его результаты.

4.1. Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

— операционная система: Ubuntu [10] 22.04.3 LTS;

— оперативная память: 15 ГБ;

— процессор: 12 ядер, AMD Ryzen 5 4600ГЦ with Radeon Graphics [11].

4.2. Объект исследования

Так как для отрисовки теней используется создание карт теней, можно предположить, что существует зависимость между количеством источников света и временем отрисовки кадра.

Чтобы исследовать, существует ли эта зависимость и каков её характер, необходимо проверить, как меняется время отрисовки кадра в зависимости от количества источников света.

4.3. Условия исследования

Так как количество источников света ограничено десятью, то исследование было проведено на количестве источников света от одного до десяти.

Остальные параметры сцены на протяжении замеров оставались, чтобы исключить их влияние на время отрисовки кадра.

Для одного количества источников света отрисовка кадра проводилась несколько раз (например, 10000), чтобы снизить влияние флуктуаций, вызванных внешними факторами, такими как нагрузка на систему или процессами, выполняемыми на компьютере.

4.4. Измерение времени отрисовки кадра

Замеры времени отрисовки кадра проводились с помощью библиотеки chrono языка C++.

Результаты замеров приведены в таблице 4.1.

Таблица 4.1 – Замеры времени отрисовки кадра при количестве источников от 1 до 10

Количество источников света, шт	**Время отрисовки кадра, нс**	**Количество источников света, шт**	**Время отрисовки кадра, нс**
1	88485	6	707738
2	239052	7	843580
3	370524	8	1001790
4	468000	9	1140690
5	588258	10	1281620

На графике 4.1 представлена зависимость времени отрисовки кадра от количества источников света.

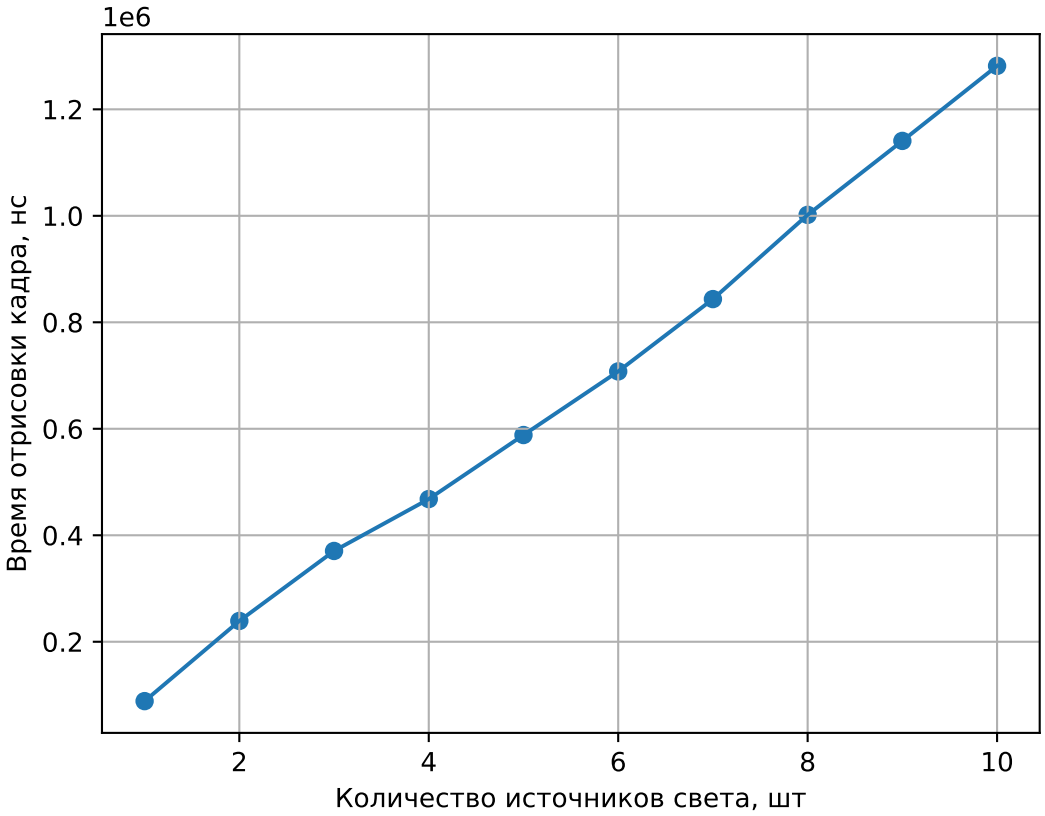

Рисунок 4.1 – Время работы алгоритмов вычисления расстояния между строками

Вывод

Время отрисовки кадра зависит от количества источников линейно: чем больше количество источников света, тем больше карт теней необходимо предварительно рассчитать и тем больше время отрисовки одного кадра.

ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы было разработано программное обеспечения для визуализации движения шарика на качелях.

В ходе выполнения проекта были выполнены все поставленные задачи:

— была формализована задача проекта;

— были рассмотрены и выбраны алгоритмы простроения трёхмерного изображения;

— были реализованы выбранные алгоритмы;

— было проведено исследование характеристик разработанного программного обеспечения.

В результате исследования была обнаружена линейная зависимость скорости отрисовки кадра от количества источников света.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Rogers David F. Алгоритмические основы машинной графики. — М.: Мир, 1989. с. 512.

[2] А Никулин Е. Компьютерная геометрия и алгоритмы машинной графики. — СПб.: БХВ-Петербург, 2003. с. 560.

[3] В Боресков А. Программирование компьютерной графики. — М.: ДМК-Пресс, 2019. с. 372.

[4] Обратная трассировка лучей — Обзор при поддержке Лаборатории компьтерной графики и мультимедия ф-та ВМК МГУ [Электронный ресурс]. Режим доступа: http://ray-tracing.ru/articles164.html (дата обращения: 11.12.2023).

[5] Г Задорожный А. Модели освещения и алгоритмы затенения в компьютерной графике. — Новосибирск: Изд-во НГТУ, 2020. с. 80.

[6] Абрамова О. Ф. Никонова Н. С. Сравнительный анализ алгоритмов удаления невидимых линий и поверхностей, работающих в пространстве изображения // NOVAINFO : электрон. науч. журн. 2015. URL: http://novainfo.ru (дата обращения: 11.12.2023).

[7] Романюк А. Н. Куринный М. В. Алгоритмы построения теней // КОМПЬЮТЕРЫ+ПРОГРАММЫ : науч. журн. 2000.

[8] В. Гиль С. Использование карты глубины для построения областей видимости. — Кафедра информатики БГУИР, 2014. с. 3.

[9] Мустафин М. Б. Турар О. Н. Ахмед-Заки Д. Ж. Тестирование визуализации Vulkan для геомоделей на системах с

графическими процессорами для трассировки лучей. URL: https://journal.neark.kz/wp-content/uploads/pdf/2020-3/6-%D0%A2%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%20%D0%B2%D0%B8%D0%B7%D1%83%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D0%B8%20VULKAN.pdf (дата обращения: 12.12.2023).

[10] Linux — Официальная документация [Электронный ресурс]. Режим доступа: https://www.linux.org/ (дата обращения: 15.09.2023).

[11] AMD Processors [Электронный ресурс]. Режим доступа: https://www.amd.com/en.html (дата обращения: 15.09.2023).