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

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

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

— формализовать задачу;

— выбрать алгоритмы для создания изображения;

— реализовать выбранные алгоритмы;

— провести исследование зависимости времени отрисовки кадра от количества источников света.

1 Аналитический раздел

В данном разделе рассматриваются теоретические основы алгоритмов компьютерной графики для создания трёхмерных изображений.

1.1. 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).

Можно выделить основные свойства шарика и качелей:

— цвет поверхности;

— выпуклость;

— оптические свойства материала;

— скорость движения.

1.2. Анализ способов задания трёхмерных объектов

Существуют несколько способов моделирования трёхмерных объектов:

— аппроксимация поверхностей объекта большим количеством многоугольников (например, треугольников) — данный тип называется **поверхностным**;

— задание объекта набором точек и рёбер, их соединяющих — это **каркасный** тип модели;

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

Сравнение способов задания трёхмерных объектов приведено ниже в таблице 1.1:

Таблица 1.1 – Сравнение способов задания моделей

Критерий сравнения	**Каркасный тип**	**Поверхностный тип**	**Модель сплошных тел**
Входные данные	Список точек, список рёбер	Список точек, список полигонов	Список тел, отношения, их связываюшие
Моделирование произвольных объектов	Нет	Да	Да

Вывод

Для выполнения курсовой работы был выбран поверхностный способ задания объектов, так как он позволяет отрисовывать произвольные объекты.

1.3. Анализ алгоритмов удаления невидимых ребер и поверхностей

Чтобы корректно отобразить один или несколько объектов на сцене, необходимо удалять невидимые рёбра и поверхности.

Для этого нужно:

1. удалить все не лицевые поверхности/рёбра;
2. удалить все видимые поверхности/рёбра, экранируемые другими телами.

Для решения данной задачи существуют несколько алгоритмов, которые можно разделить на три класса в зависимости от пространства работы: пространство объекта, пространство изображения и пространство объекта и изображения в сочетании.

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.
3. Алгоритмы, формирующие **список приоритетов**: алгоритм Ньюэла-Ньюэла-Санча.

   Далее будут рассмотрены теоретические основы алгоритмов первых двух классов.

1.3.1 Алгоритм плавающего горизонта

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

Трёхмерная задача сводится к двумерной путем пересечения исходной поверхности последовательностью параллельных секущих плоскостей, имеющих постоянные значения координат x, y или z (можно и последовательно применять разные постоянные значения координат для получения более реалистичного изображения).

Шаги работы алгоритма (при $z = const$, то есть если фиксируется z):

(a) плоскости $z = const$ упорядочиваются по возрастанию расстояния от них до точки наблюдения;

(b) для каждой плоскости, начиная с ближайшей к наблюдателю, строится кривая, лежащая на ней и принадлежащая объекту, то есть для каждого значения x вычисляется соответствующее значение y;

(c) происходит удаление невидимых линий: если на текущей плоскости при некотором заданном значении x соответствующее значение y больше значения y для всех предыдущих кривых при этом значении x, то текущая кривая видима в этой точке, в противном случае она невидима.

Алгоритм определяет обработку объекта чётко математически, что является несомненным достоинством, однако возникают несколько проблем, в числе которых:

— необходимость интерполяции значений y, если не всегда возможно вычислить значение функции;

— необходимость подбирать функцию, описывающую объект;

— ограниченный набор возможных для отрисовки объектов.

Типичный результат, получаемый в результате работы алгоритма представлен ниже:

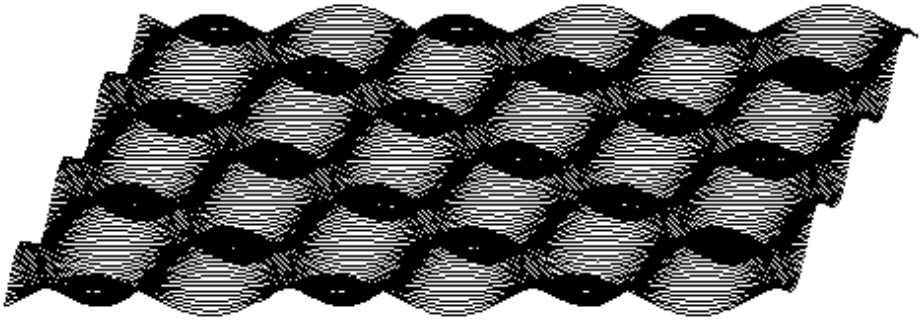

Рисунок 1.1 – Результат работы алгоритма плавающего горизонта

1.3.2 Алгоритм, использующий Z-буфер

Алгоритм, использующий Z-буфер, в специальном буфере для каждого пикселя изображения хранит координату Z и значения RGB-составляющих цвета пикселя. При занесении пикселя в буфер кадра значение его глубины сравнивается со значением глубины пикселя, находящегося на текущий момент в буфере кадра. Если значение координаты z текущего пикселя больше, то есть он ближе к наблюдателю, его атрибуты заносятся в буфер кадра.

Формальное описание алгоритма, использующего Z-буфер:

(a) буфер кадра заполняется фоновым значением интенсивности или цвета;

(b) Z-буфер заполняется минимальным значением глубины;

(c) каждый отрисовываемый многоугольник преобразуется в растровую форму в произвольном порядке;

(d) 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 $z(x, y)$;

Если известно уравнение плоскости, содержащей многоугольник, то вычисление глубины пикселя на сканирующей строке в пределах многоугольника можно делать пошагово по формуле

$$z_n = z - \frac{a}{c} \tag{1.1}$$

где z_n — глубина пикселя $Pixel_n$ на текущей сканирующей строке, z — глубина первого слева пикселя на текущей сканирующей строке, a, c — коэффициенты уравнения плоскости, содержащей обрабатываемый многоугольник.

Алгоритм имеет небольшое количество чётко определённых простых шагов, что делает его достаточно несложным для реализации, визуализация пересечений сложных поверхностей становится тривиальной задачей, но имеются и недостатки, в числе которых, например, большой объём потребляемой памяти для двух буферов.

Построение теней

Алгоритм, использующий Z-буфер, также позволяет строить падающие тени объектов при помощи так называемых карт теней (shadow mapping).

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

1.3.3 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.

Если бы лучи отслеживались, начиная от объекта, то алгоритм был бы весьма неэффективен, так как лишь немногие лучи в итоге доходят до наблюдателя. Поэтому было предложено отслеживать лучи в обратном направлении, то есть от наблюдателя к объекту.

Итак, из виртуального глаза через каждый пиксель (pixel) изображения испускается луч и находится его точка пересечения

с поверхностью сцены. Лучи, выпущенные из глаза, называются **первичными** (eye ray).

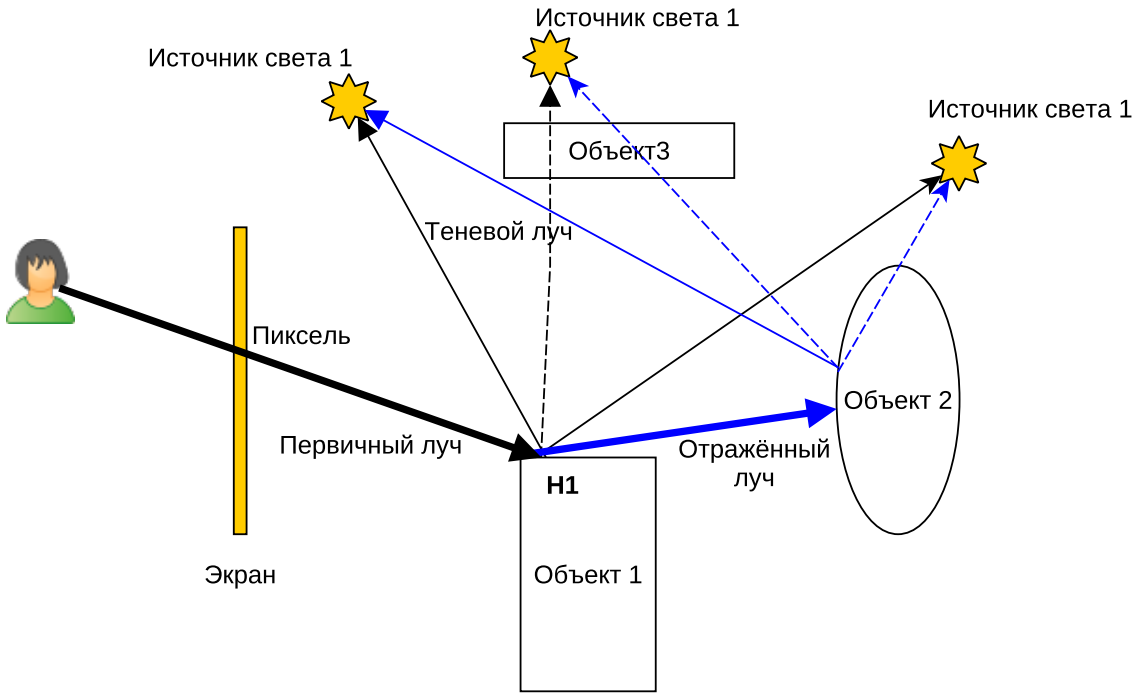

Рисунок 1.2 – Визуализация траектории лучей в алгоритме трассировки лучей

Пусть первичный луч пересекает объект 1 (object 1) в точке H1.

Шаги работы алгоритма трассировки лучей:

(a) необходимо определить для каждого источника освещения (light, в данном случае пусть они будут точечными), видна ли из него эта точка;

(b) для каждого источника света до него испускается **теневой луч** (shadow ray) из точки H1;

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** (Фонг, Гуро и т.д.);

(c) освещение со всех видимых (из точки H1) источников света складывается;

(d) 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 различным критериям.

(a) Локальные модели: учитывается только локальная геометрия объектов. Каждый объект рассматривается независимо от других, без учета их взаимодействия. Такие модели также называются объектно-ориентированными.

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

(c) Физические модели: аппроксимируют свойства реальных объектов, учитывая особенности поверхностной структуры и поведение материалов. Например, моделирование кожи или песчинок песка.

(d) Эмпирические модели: параметры моделей могут не иметь физической интерпретации, но подбор этих параметров позволяет получать реалистичные изображения.

(e) Гладкие модели: поверхность представляется как гладкая. Например, моделирование пластика.

(f) 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.

Для увеличения быстродействия алгоритма можно модифицировать алгоритм трассировки лучей, испуская лучи не через каждый пиксель, а через точки, аппроксимирующие полигоны объектов сцены.

Алгоритм приобретает следующий вид:

(a) цикл по всем полигонам сцены;

(b) генерация луча, проходящего через точку, аппроксимирующую данный полигон;

(c) нахождение точки пересечения луча со всеми объектами сцены, исключая пересечение с рассматриваемым полигоном.: если такого пересечения нет, точка видима, иначе выполнить следующий шаг цикла;

(d) вычисление в случае видимости точки её интенсивности согласно заданной модели освещения;

(e) высвечивание полигона с найденной интенсивностью в соответствии с заданным ему цветом при наложении текстур;

(f) конец цикла.

1.3.4 Алгоритм Робертса

Суть алгоритма Робертса заключается в следующем: объекты проецируются на плоскость экрана, и анализ видимости рёбер или граней происходит уже на плоскости. Каждое ребро последовательно сравнивается со всеми гранями (чаще всего это выпуклые многоугольники).

Варианты взаимного расположения ребра и грани:

— 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:

(a) Если ребро не перекрыто гранью, оно проверяется с последующей гранью.

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

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

Производительность можно повысить, если:

— не делить ребро после каждого сравнения, а запоминать невидимые интервалы ребра в отдельном массиве, границы интервалов можно отсортировать в порядке возрастания, и затем, перебрав границы в этом порядке, выделить видимые части ребра;

— использовать когерентность в пространстве (использовать простые геометрические тела, описанные вокруг объектов — оболочки).

1.3.5 Алгоритм Варнока

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

Ниже представлены основные шаги алгоритма.

(a) 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.

(b) 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.

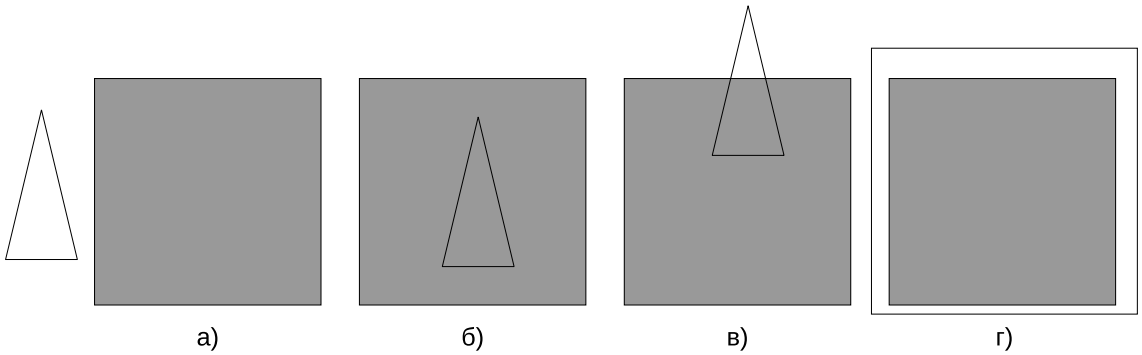

Рисунок 1.3 – Типы многоугольников: а — внешний, б — внутренний, в — пересекающий, г — охватывающий

Правила обработки для каждого окна:

(a) 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;

(b) 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;

(c) 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;

(d) если в сцене присутствует один охватывающий многоугольник, то окно изображается с интенсивностью или цветом соответствующими охватывающему многоугольнику;

(e) 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.

1.3.6 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.

Алгоритм Вейлера-Азертона содержит несколько шагов.

(a) Предварительно проводится сортировка по глубине.

(b) Осуществляется отсечение по границе ближайшего к точке наблюдения многоугольника, называемое сортировкой многоугольников на плоскости.

(c) Многоугольники, экранируемые более близкими к точке наблюдения многоугольниками, удаляются.

(d) Если необходимо, то проводится рекурсивное разбиение и новая сортировка.

Вывод

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

Таблица 1.2 – 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	$n \cdot N$	$n \cdot N$	n^2	n^2	$n \cdot N$	$n \cdot N$
Использование факта 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
Ограничения на отрисовываемые объекты	Объекты задаются только аналитически	Нет	Только выпуклые объекты	Объекты - плоские многоугольники	Нет	Нет

Для выполнения курсовой работы был выбран алгоритм, использующий Z-буфер, так как его идея проста, он подходит для решения поставленных задач и обладает приемлемой оценкой трудоёмкости.

1.4. Анализ алгоритмов закрашивания

Так как на выходе работы программы необходимо получить цветное реалистичное изображение, нужно подобрать алгоритм, позволяющий закрасить полигоны, из которых состоят объекты, учитывая тени и расположение источника света.

Рассмотрим три метода закрашивания: простую закраску, закраску методом Гуро и закраску методом Фонга.

1.4.1 Простая закраска

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

$$I_d = I_l k_d \cos\theta, \tag{1.2}$$

где I_d — интенсивность диффузно отраженного света; I_l — интенсивность источника; k_d — коэффициент диффузного отражения ($0 \leq k_d \leq 1$), который зависит от материала объекта и длины волны падающего на него света, но в простых моделях освещения обычно является постоянным; θ — угол между направлением на источник света и нормалью к поверхности.

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

$$I = I_a k_a + I_l(k_d \cos\theta + k_s \cos^n\alpha)/(d + K) \tag{1.3}$$

где k_a — коэффициент диффузного отражения рассеянного света ($0 \leq k_a \leq 1$) и I_a — интенсивность рассеянного света, I_l — интенсивность источника, k_d — коэффициент диффузного отражения ($0 \leq k_d \leq 1$), θ — угол между направлением на источник света и нормалью к поверхности, n — степень, аппроксимирующая пространственное распределение зеркально отраженного света, k_s — коэффициент зеркально отраженного све-

та, α — угол между отражённым лучом и вектором наблюдения, 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 нереалистичное изображение, состоящее из отдельно видимых полигонов.

Похожий эффект продемонстрирован на рисунке 1.4.

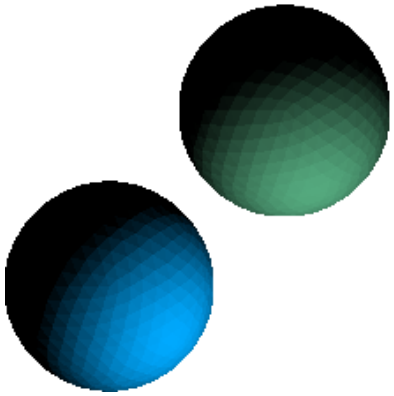

Рисунок 1.4 – Эффект отсутствия сглаживания

1.4.2 Закраска методом Гуро

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

Интенсивность в точке P на рисунке 1.5 — I_3 — вычисляется следующим образом:

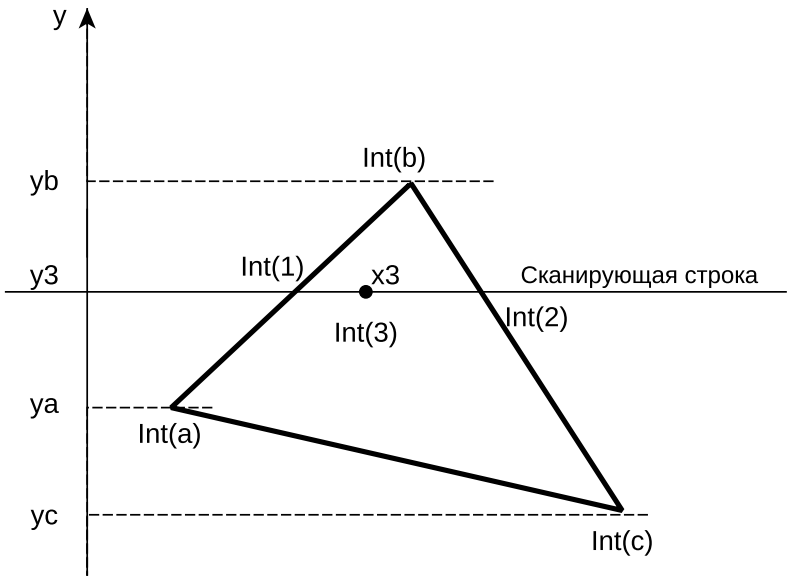

Рисунок 1.5 – Интерполяция интенсивности

Необходимо ввести следующие векторы для интенсивностей в точках:

$$I^1 = \begin{bmatrix} I_b \\ I_c \end{bmatrix}, I^2 = \begin{bmatrix} I_b \\ I_c \end{bmatrix}, I^3 = \begin{bmatrix} I^1 \\ I^2 \end{bmatrix} \tag{1.4}$$

Также необходимы следующие векторы для расчёта коэффи-

циентов при интенсивностях:

$$y^1 = \begin{bmatrix} \frac{y_3 - y_a}{y_b - y_a} \\ \frac{y_b - y_3}{y_b - y_a} \end{bmatrix}, y^2 = \begin{bmatrix} \frac{y_3 - y_c}{y_b - y_c} \\ \frac{y_b - y_3}{y_b - y_c} \end{bmatrix} \tag{1.5}$$

Тогда с учётом формул 1.4 и 1.5 можно вычислить интенсивность $Int(3)$:

$$I_3 = I^1 \cdot y^1 \cdot \frac{x_b - x_3}{x_b - x_a} + I^2 \cdot y^2 \cdot \frac{x_3 - x_a}{x_b - x_a} \tag{1.6}$$

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.

1.4.3 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.

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

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

$$a^1 = \begin{bmatrix} a \\ (1-a) \end{bmatrix}, a \in \{u, w, t\} \tag{1.7}$$

$$n^1 = \begin{bmatrix} n_2 \\ n_1 \end{bmatrix}, n^2 = \begin{bmatrix} n_1 \\ n_3 \end{bmatrix}, n^3 = \begin{bmatrix} n^1 \\ n^2 \end{bmatrix} \tag{1.8}$$

Используя рисунок 1.5, получим:

$$n_1 = u^1 \cdot n^1, n_2 = u^2 \cdot n^2 \tag{1.9}$$

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

$$I_3 = t^1 \cdot n \tag{1.10}$$

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:

Таблица 1.3 – Сравнение алгоритмов закраски

Критерий сравнения	**Простая модель освещения**	**Закраска методом Гуро**	**Закраска методом Фонга**
Использование когерентности изображения	Нет	Да	Да
Скорость работы	1	2	3
Характеристика получаемого изображения	Объекты выглядят нереалистично, происходит разбиение на полигоны, то есть отсутствует сглаживание	Более реалистичное изображение, чем при использовании простой модели, однако появляется эффект полос Маха, контуры изображения - многоугольники, на некоторых участках поверхность может выглядеть плоской	Самый реалистичный результат из трёх рассматриваемых методов, лучшая локальная аппроксимация кривизны поверхности, более правдоподобные зеркальные блики

Для курсовой работы был выбран метод закраски Гуро в сочетании с моделью освещения Фонга, так как он даёт высокую степень реалистичности изображения и имеет удовлетворительную скорость работы.

1.4.4 Функциональная модель программы

Функциональная модель программы в нотации IDEF0 представлена на рисунке 1.6:

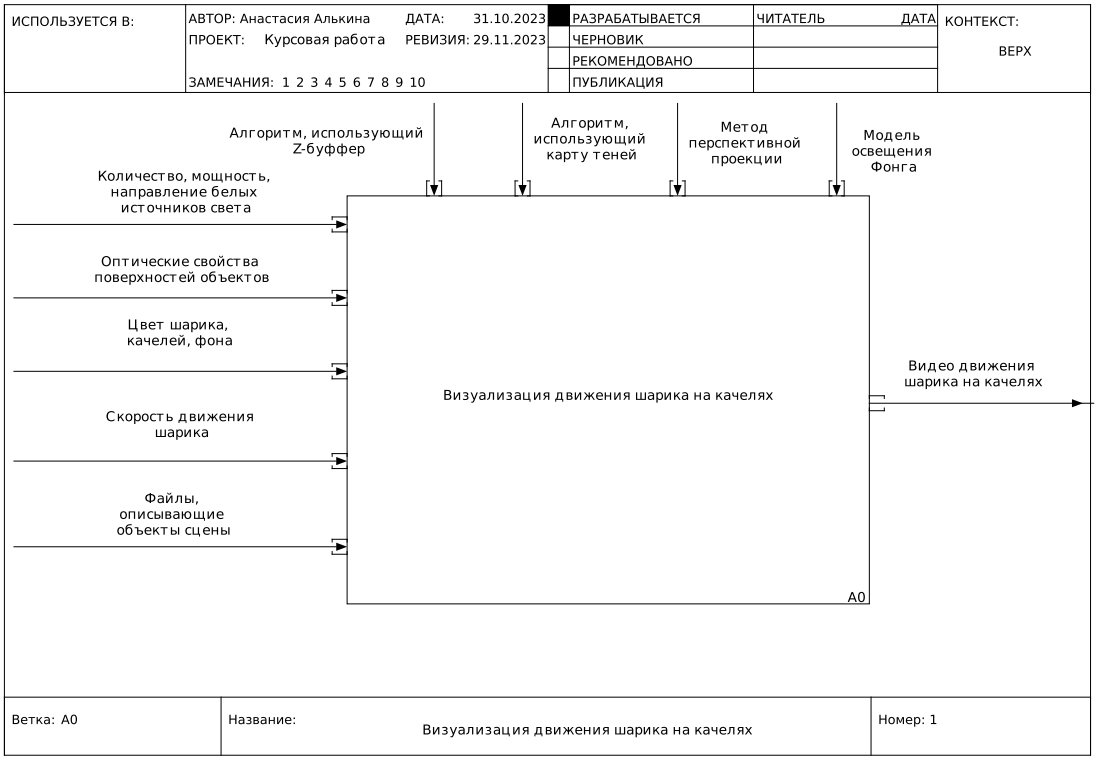

Рисунок 1.6 – Функциональная модель программы

Вывод

В данном разделе было произведено исследование предметной области, проведена формализация объектов сцены, обзор способов задания трёхмерных объектов, алгоритмов удаления невидимых рёбер, 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, а также даны описания алгоритмов, выбранных для решения поставленной задачи.

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;
- результатом работы программного обеспечения является видео движения шарика на качелях;
- кадры должны генерироваться со скоростью 30 кадров в секунду.

2.2. Функциональная модель программы первого уровня в нотации idef0

Функциональная модель программы в нотации IDEF0 представлена на рисунке 2.1:

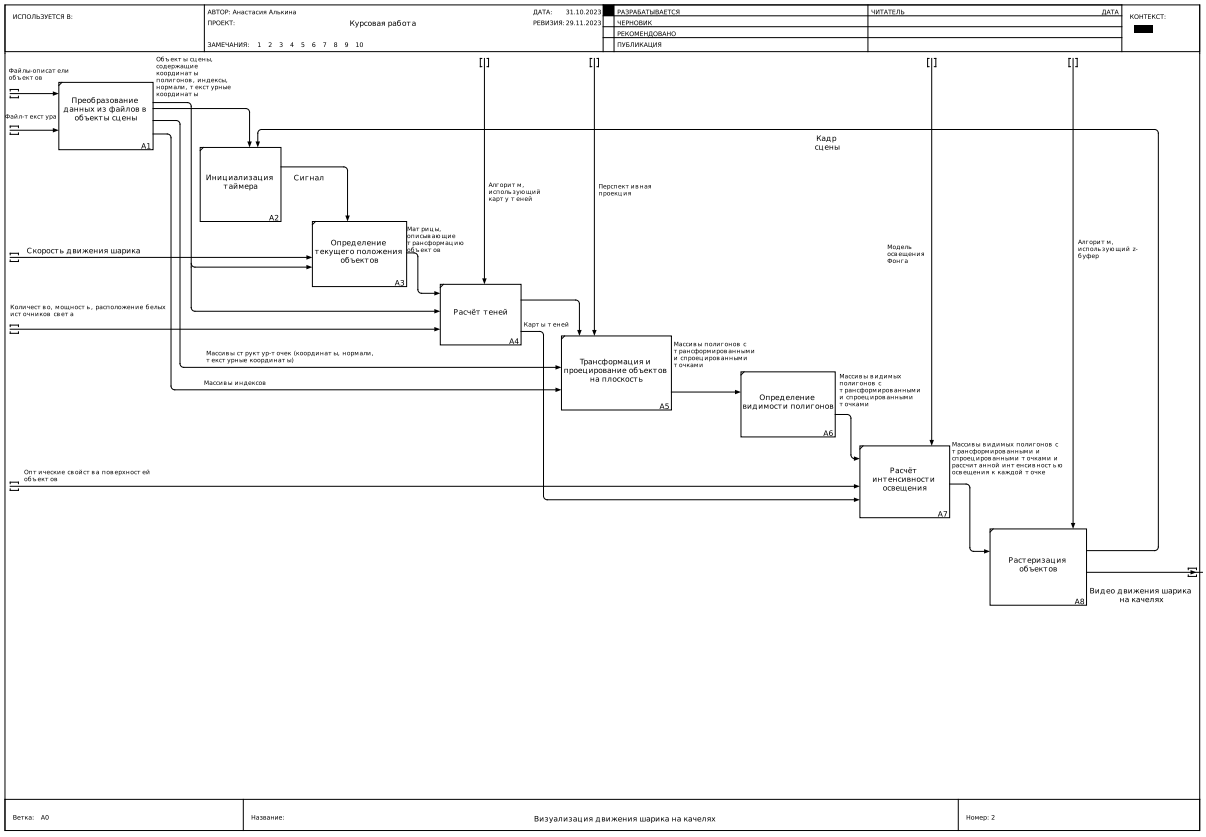

Рисунок 2.1 – Функциональная модель первого уровня

2.3. Структуры данных

В таблице 2.1 представлены описания структур данных для объектов:

Таблица 2.1 – Структуры данных 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
Камера	Структура, содержащая матрицу вида
Источник света	Структура, содержащая матрицу источника света, интенсивность, направление, позицию, цвет источника света

2.4. Схема алгоритма, использующего Z-буфер

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

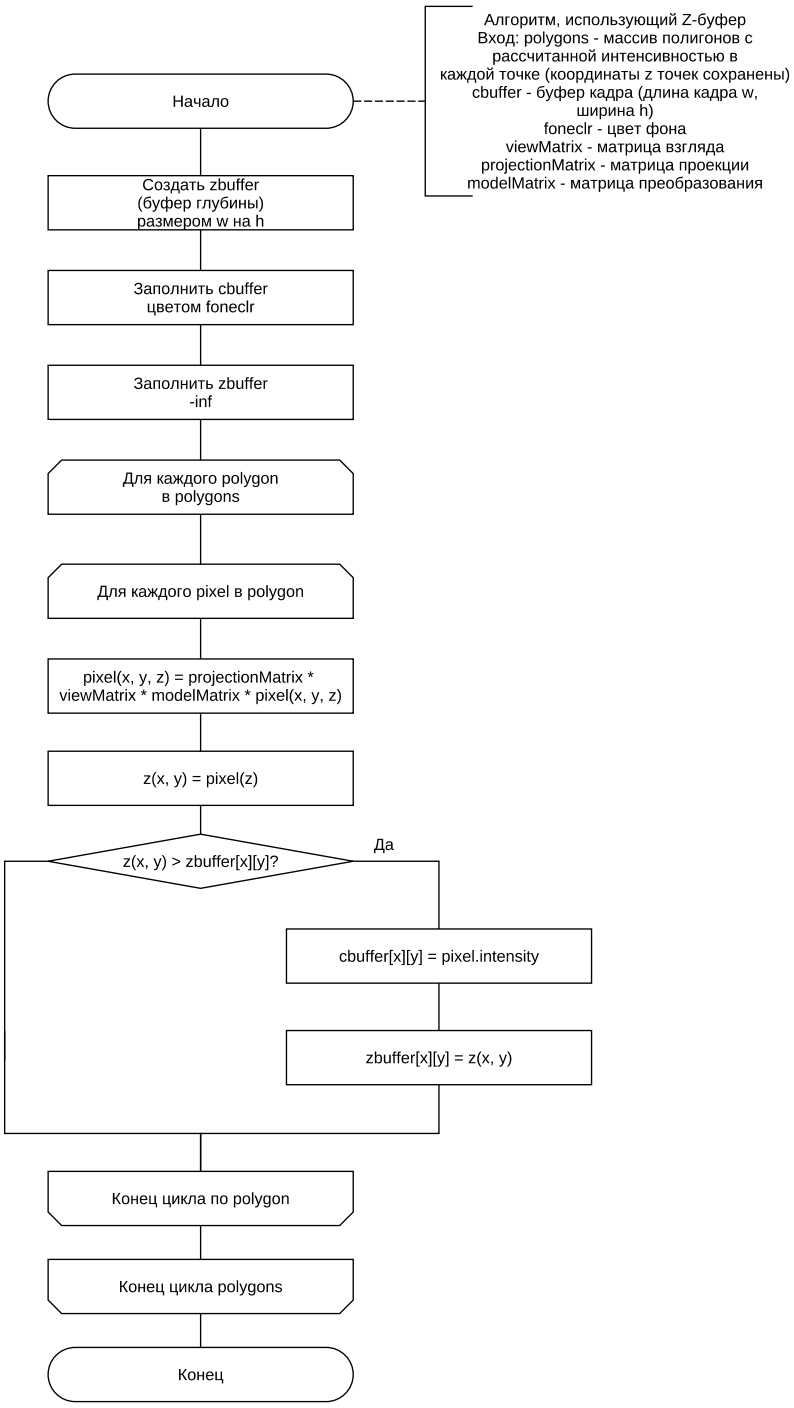

Рисунок 2.2 – Схема алгоритма, использующего Z-буфер

2.5. Схема алгоритма построения карты теней

На рисунке 2.3 приведена схема алгоритма построения карты теней.

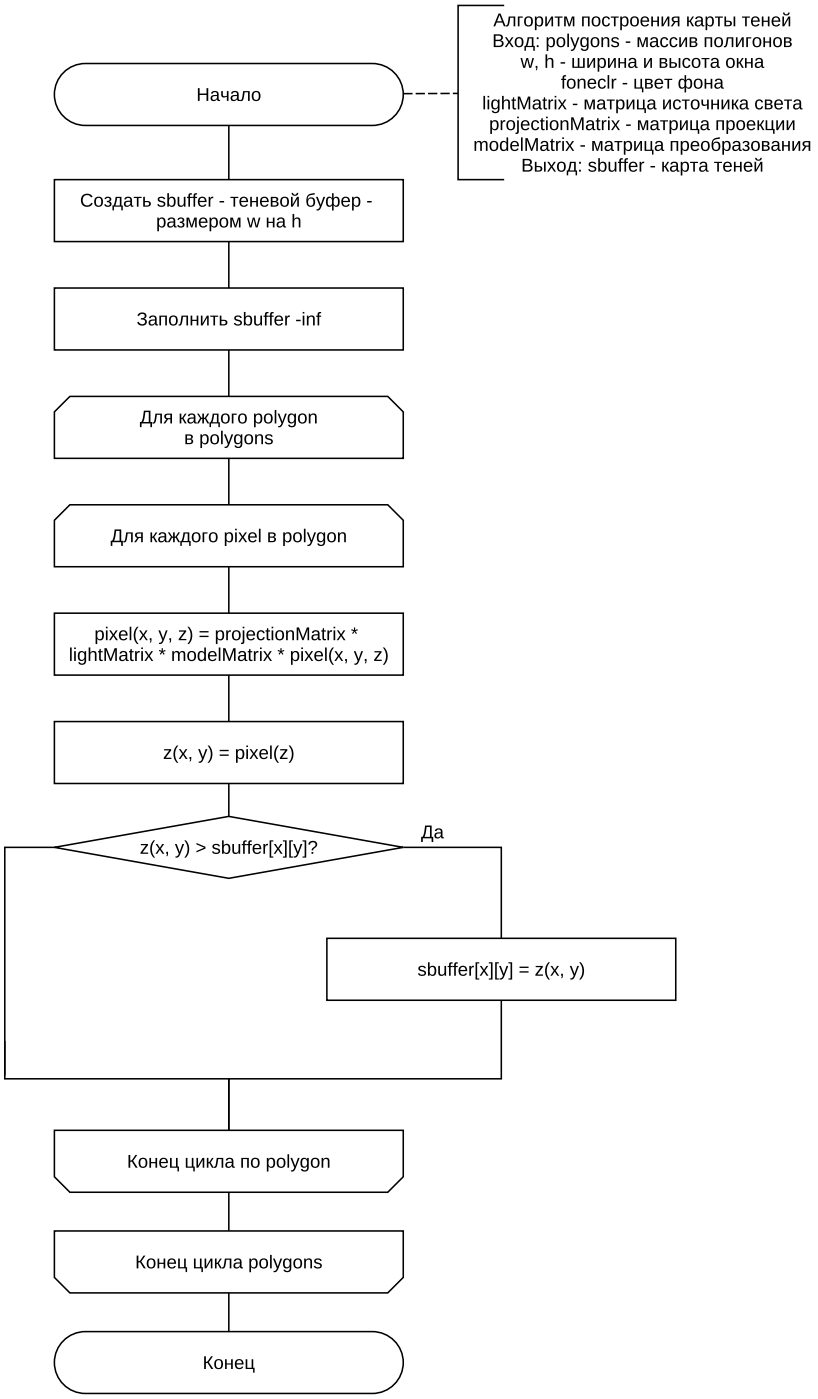

Рисунок 2.3 – Схема алгоритма построения карты теней

2.6. Схема алгоритма закраски Гуро с моделью освещения Фонга

На рисунке 2.4 приведена схема алгоритма построения карты теней.

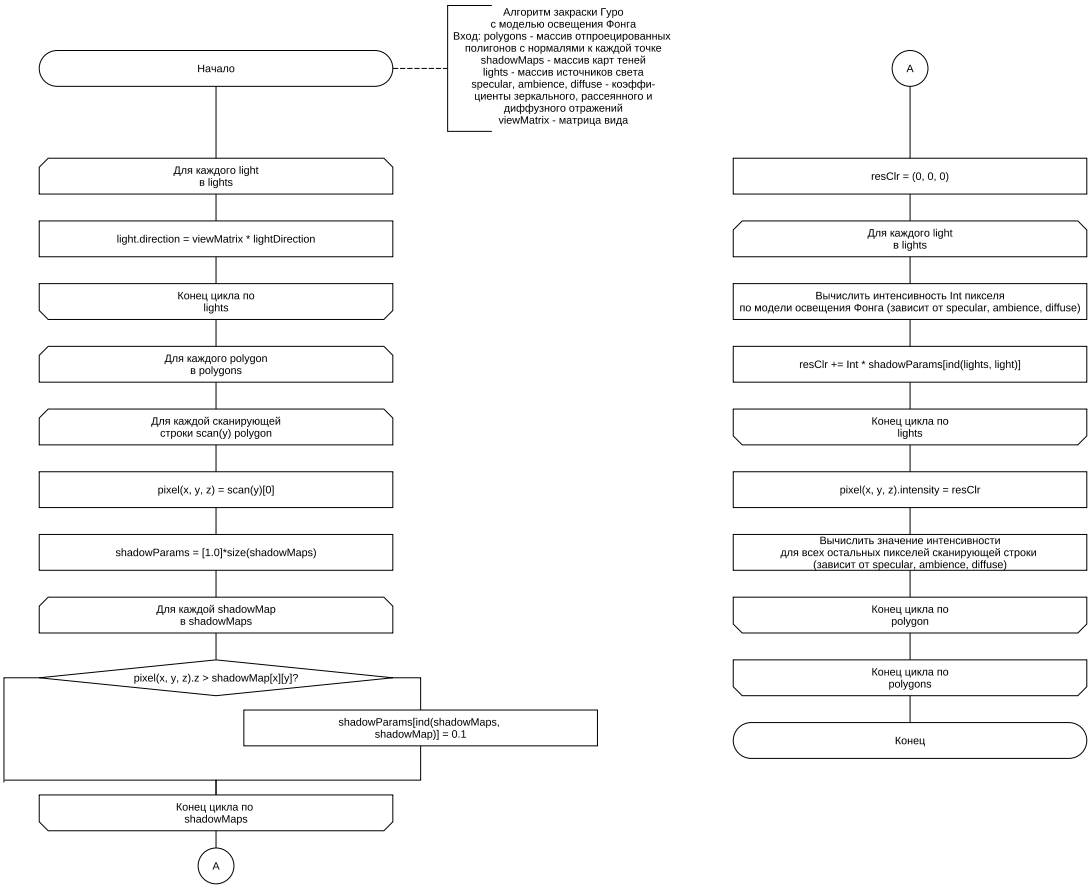

Рисунок 2.4 – Схема алгоритма закраски Гуро с моделью освещения Фонга

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 представлены средства реализации и интерфейс разработанного программного обеспечения, а также приводятся листинги кода.

3.1. Средства реализации

В качестве языка программирования для реализации курсовой работы был выбран язык C++ с использованием OpenGL Framework, так как язык C++ предоставляет необходимые средства для реализации программы. OpenGL Framework обеспечивает удовлетворительную скорость генерации изображения, так как вычисления выполняются на графическом процессоре. Скорость генерации кадра особенно важна вследствие требования на частоту генерации кадра, которая составляет 30 кадров в секунду.

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

3.2. Листинг кода

В листинге 3.1 приведена инициализация окна OpenGL. В функции initializeGL с помощью вызова функции glEnable с параметром GL_DEPTH_TEST включается тест на глубину, то есть сравнение глубины текущей точки полигона и значения в буфере глубины. Также присутствует вызов функции glEnable с

параметром GL_CULL_FACE для удаления невидимых задних граней.

Листинг 3.1 – Инициализация opengl-окна

```
    context()->functions()->glEnable(GL_DEPTH_TEST);
    context()->functions()->glEnable(GL_CULL_FACE);

    initShaders();
    for (int i = 0; i < lights.size(); i++)
        shadowBuffers[i]->shadowBuff = new
            QOpenGLFramebufferObject(shadowBuffers[i]->width,
                                     shadowBuffers[i]->height,
                                     QOpenGLFramebufferObject::Depth);

    materialProperties_t material = {0.1f, 0.9f, 10.0f};
    for (int i = 0; i < OBJ_NUMBER; i++)
        objects.push_back(new BaseObject(objPaths[i], texturePaths[i], material));

    objects[0]->translate(QVector3D(0.0, SPHERE_Y, 0.0));
}

void GLWidget::resizeGL(int w, int h)
```

В листинге 3.2 приведена реализация метода отрисовки кадра OpenGL.

Листинг 3.2 – Отрисовка кадра

```
    context()->functions()->glClearColor(foneClr.x(), foneClr.y(), foneClr.z(), 1.0f);
    for (int i = 0; i < lights.size(); i++)
        if (lights[i]->getUsed() == true)
            getShadowMap(i, shadowTextures[i]);
    context()->functions()->glViewport(0, 0, this->width(), this->height());
    context()->functions()->glClear(GL_COLOR_BUFFER_BIT |
        GL_DEPTH_BUFFER_BIT);
    shaderProgram.bind();
    sendLightsIntoShader(&shaderProgram);
    cam->set(&shaderProgram);
    shaderProgram.setUniformValue("numberLights", getCurLights());
    shaderProgram.setUniformValue("numberShadows", getCurLights());
    shaderProgram.setUniformValue("qt_ProjectionLightMatrix", projectionLightMatrix);
    sendShadowIntoShader(&shaderProgram);
    shaderProgram.setUniformValue("qt_ProjectionMatrix", projectionMatrix);
    sendMaterialIntoShader(&shaderProgram, 0);
    objects[0]->draw(&shaderProgram, context()->functions());
    sendMaterialIntoShader(&shaderProgram, 1);
    for (int i = 0; i < objects.size() && draw; i++)
        objects[i]->draw(&shaderProgram, context()->functions());
    shaderProgram.release();
}

```

```
23 void GLWidget::initShaders()
```

В строках 3-4 происходит расчёт карт теней.

В листингах 3.3–3.5 приведены основные вершинный и фрагментный шейдеры, участвующие в отрисовке изображения. Фрагментный шейдер обеспечивает закраску пикселей в соответствии с выбранными алгоритмами.

Листинг 3.3 – Вершинный шейдер

```
1  attribute highp vec4 qt_Vertex;
2  attribute highp vec2 qt_MultiTexCoord0;
3  attribute highp vec3 qt_Normal;
4  uniform highp mat4 qt_ProjectionMatrix;
5  uniform highp mat4 viewMatrix;
6  uniform highp mat4 modelMatrix;
7
8  uniform highp mat4 qt_ProjectionLightMatrix;
9  uniform highp mat4 shadowMatrixes[10];
10 uniform int numberShadows;
11
12 varying highp vec4 qt_VertexesLightMatrix[10];
13
14 varying highp vec4 qt_Vertex0;
15 varying highp vec2 qt_TexCoord0;
16 varying highp vec3 qt_Normal0;
17 varying highp mat4 qt_viewMatrix;
18
19 void main(void)
20 {
21     mat4 mvMatrix = viewMatrix * modelMatrix;
22     gl_Position = qt_ProjectionMatrix * mvMatrix * qt_Vertex;
23     qt_TexCoord0 = qt_MultiTexCoord0;
24     qt_Normal0 = normalize(vec3(mvMatrix * vec4(qt_Normal, 0.0)));
25     qt_Vertex0 = mvMatrix * qt_Vertex;
26     qt_viewMatrix = viewMatrix;
27
28     for (int i = 0; i < numberShadows; i++)
29     qt_VertexesLightMatrix[i] = qt_ProjectionLightMatrix * shadowMatrixes[i] * modelMatrix *
           qt_Vertex;
30 }
```

Листинг 3.4 – 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);
}

void main(void)
{
    for (int i = 0; i < numberLights; i++) {
    tmp[i] = qt_VertexesLightMatrix[i].xyz / qt_VertexesLightMatrix[i].w;
    tmp[i] = tmp[i] * vec3(0.5) + vec3(0.5);
    }

    vec3 clr = vec3(1.0f, 1.0f, 1.0f);
    float shadowParam = 1.0f;

    for (int i = 0; i < numberLights; i++) {
    shadowParam = ShadowMapping(qt_ShadowMaps0[i], tmp[i].xy, tmp[i].z * 255.0f - 0.6f);
    shadowParam += 0.1f;
```

Листинг 3.5 – Продолжение листинга 3.5

```
    if (shadowParam >= 1.0f)
        shadowParam = 1.0f;
    resShadowParams[i] = shadowParam;
    }

    for (int i = 0; i < numberLights; i++) {
    corLights[i].direction = qt_viewMatrix * lights[i].direction;
    corLights[i].pos = qt_viewMatrix * lights[i].pos;
    corLights[i].color = lights[i].color;
    corLights[i].power = lights[i].power;
    corLights[i].type = lights[i].type;
    }

    vec4 resClr = vec4(0.0f, 0.0f, 0.0f, 0.0f);
    vec4 eyePos = vec4(0.0f, 0.0f, 0.0f, 1.0f);

    for (int i = 0; i < numberLights; i++) {
    vec3 light_vec;
    light_vec = normalize(corLights[i].direction.xyz);
    vec4 srcClr = texture2D(qt_Texture0, qt_TexCoord0);
    float len = length(qt_Vertex0.xyz - eyePos.xyz);
    vec3 eye = normalize(qt_Vertex0.xyz - eyePos.xyz);
    vec3 reflectlLight = normalize(reflect(light_vec, qt_Normal0));
    vec4 diffFactor = diffParam * srcClr * max(0.0f, dot(qt_Normal0, -light_vec)) *
        corLights[i].power;
    vec4 ambFactor = ambParam * srcClr;
    vec4 specFactor = vec4(corLights[i].color, 1.0f) * corLights[i].power * pow(max(0.0f,
        dot(reflectlLight, -eye)), specParam);
    resClr += ((diffFactor * vec4(clr, 1.0f)) + (ambFactor * vec4(clr, 1.0f)) + (specFactor *
        vec4(clr, 1.0f)) * resShadowParams[i]);
    }

    gl_FragColor = resClr;
}
```

В листингах 3.6–3.8 представлены листинги шейдеров, создающих карты теней.

Листинг 3.6 – 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 3.7 – Продолжение листинга 3.6

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

Листинг 3.8 – Фрагментный шейдер для создания карты теней

```
varying highp vec4 qt_Vertex0;

void main(void)
{
    float depth = qt_Vertex0.z / qt_Vertex0.w;
    depth = depth * 0.5f + 0.5f;
    gl_FragColor = vec4(depth, 0.0f, 0.0f, 0.0f);
}
```

В листинге 3.9 приведена функция, создающая текстуры, содержащие карты теней.

Листинг 3.9 – Создание текстур карт теней

```
    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;
        }
}

void GLWidget::setFone(QColor clr)
```

3.3. Описание интерфейса программы

Демонстрация интерфейса программы приведена на рисунке 3.1.

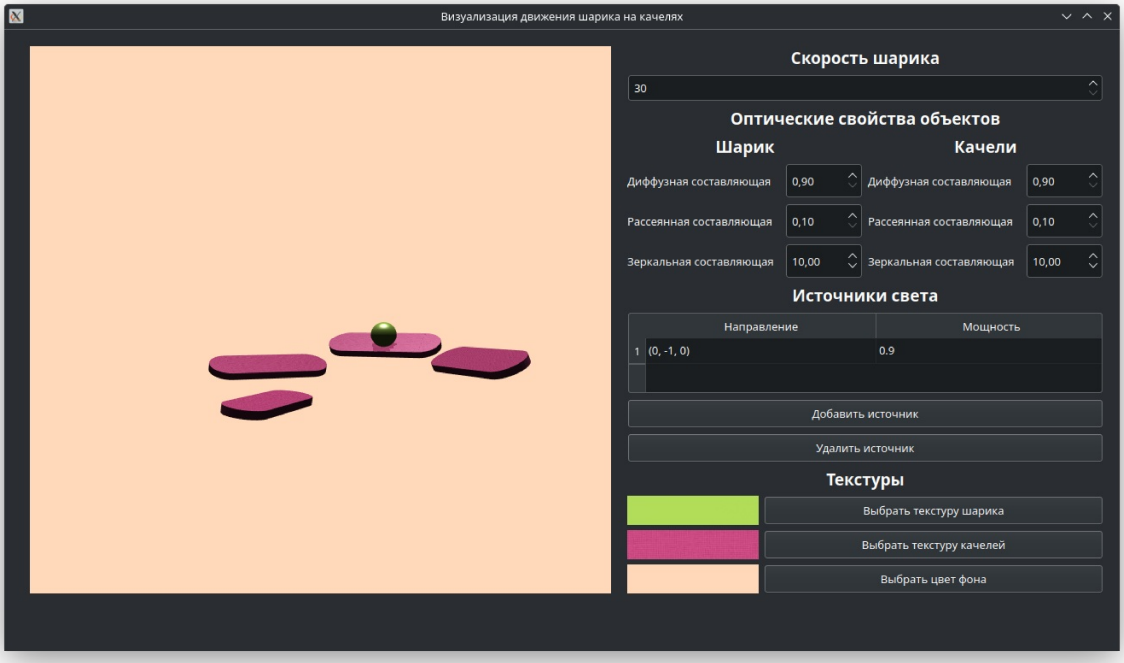

Рисунок 3.1 – Демонстрация интерфейса программы

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.

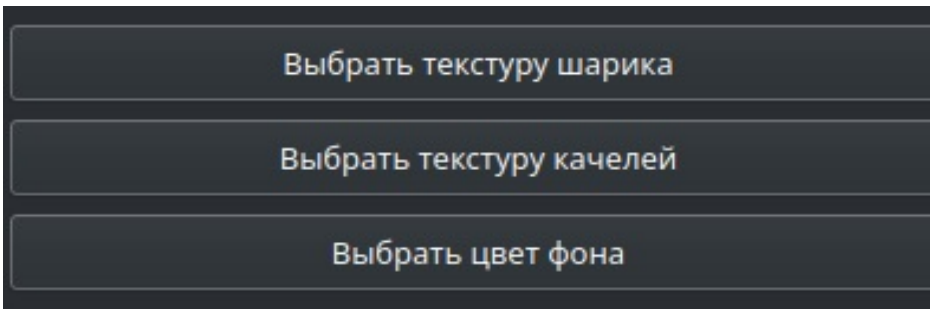

Рисунок 3.2 – Кнопки выбора текстур

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

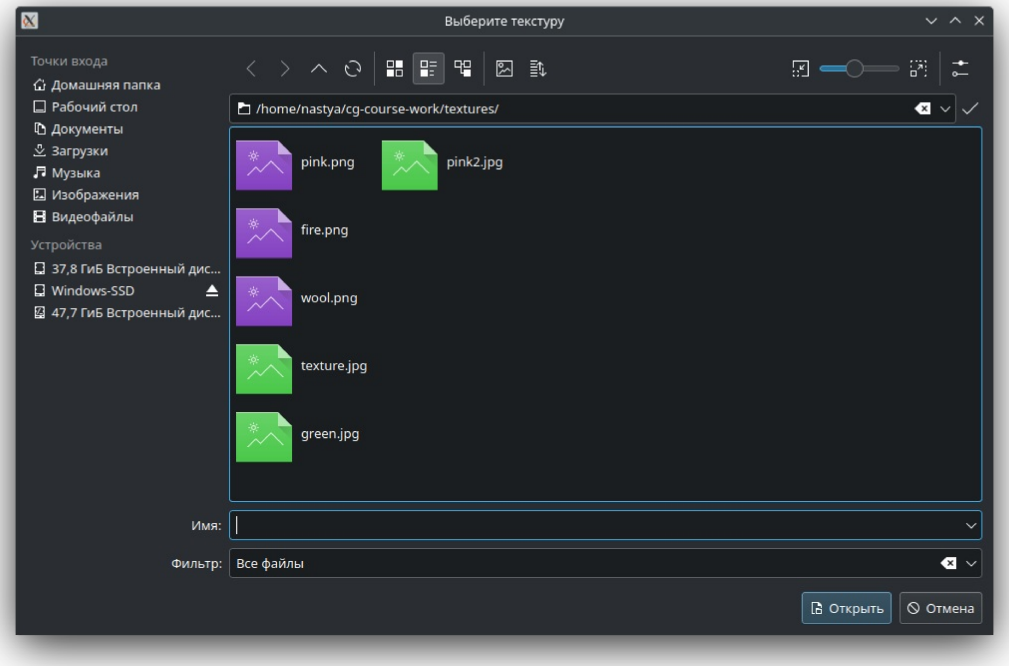

Рисунок 3.3 – Окно выбора текстуры

Нажатие на кнопку выбора цвета фона приводит к появлению соответствующего окна — рисунок 3.4.

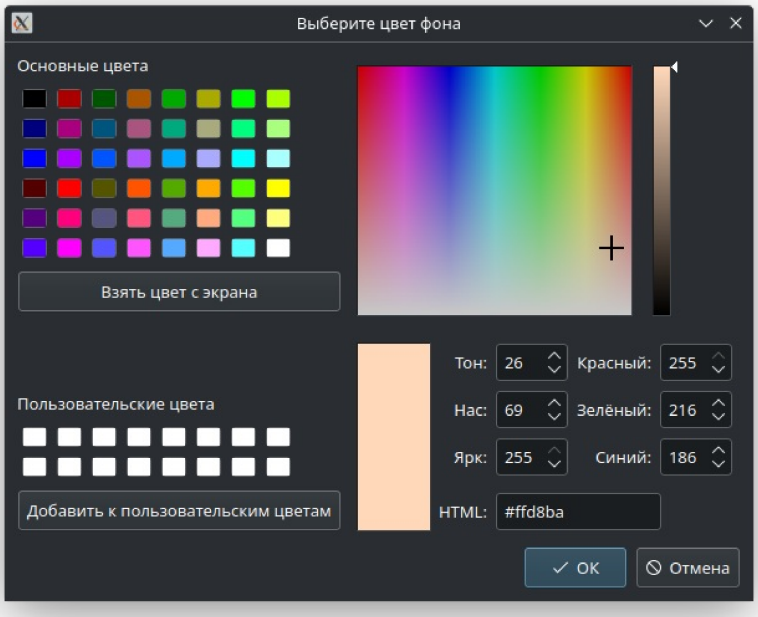

Рисунок 3.4 – Окно выбора цвета фона

Добавление и удаление источника света происходит с помощью кнопок, представленных на рисунке 3.5.

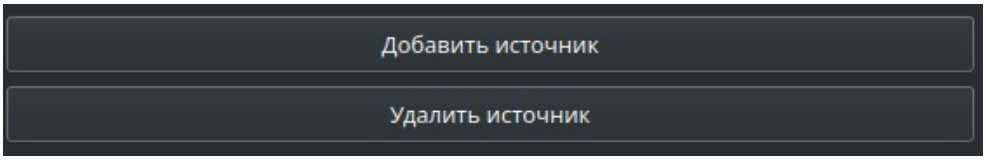

Рисунок 3.5 – Кнопки добавления и удаления источника света

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

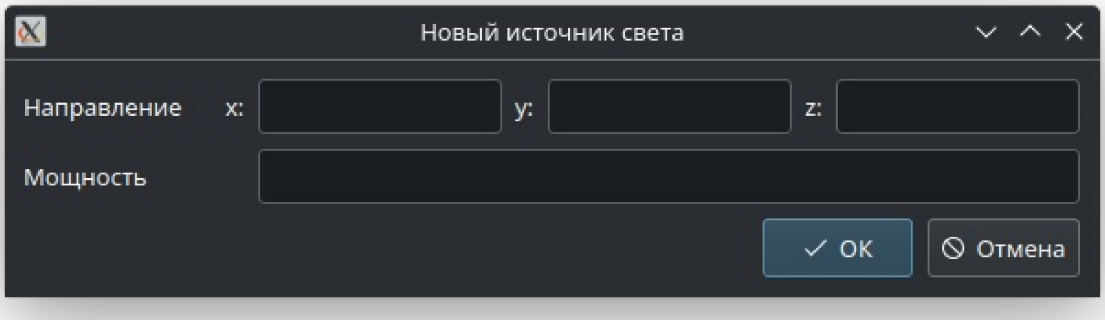

Рисунок 3.6 – Окно ввода значений

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.