



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 КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

*«Программа моделирования работы
песочных часов»*

Студент ИУ7-52Б
(Группа)

(Подпись, дата) Е.В. Брянская
(И.О.Фамилия)

Руководитель курсового проекта

(Подпись, дата) М.Ю. Барышникова
(И.О.Фамилия)

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)

УТВЕРЖДАЮ
Заведующий кафедрой ИУ7
(Индекс)
И.В.Рудаков
(И.О.Фамилия)
« ____ » _____ 2020 г.

ЗАДАНИЕ на выполнение курсового проекта

по дисциплине Компьютерная графика

Студент группы ИУ7-52Б

Брянская Екатерина Вадимовна
(Фамилия, имя, отчество)

Тема курсового проекта Программа моделирования работы песочных часов

Направленность КП (учебный, исследовательский, практический, производственный, др.)
производственный

Источник тематики (кафедра, предприятие, НИР) кафедра

График выполнения проекта: 25% к 4 нед., 50% к 7 нед., 75% к 11 нед., 100% к 14 нед.

Задание Разработать программу моделирования работы песочных часов. Измерительное время выбирается пользователем из предлагаемых вариантов. Программа должна обеспечивать возможность задания в соответствующем поле положения источника освещения, камеры, её перемещения вокруг объекта. Пользователь может начать повторный отсчет по истечении времени работы прибора. Программа должна предусматривать построение теней часов и песка в них. При разработке программы необходимо учитывать прозрачность стекла и блики от источника освещения.

Оформление курсового проекта:

Расчетно-пояснительная записка на 25-30 листах формата А4.

Расчетно-пояснительная записка должна содержать постановку задачи, введение, аналитическую, конструкторскую, технологическую части, экспериментально-исследовательский раздел, заключение, список литературы и приложения.

Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.): на защиту проекта должна быть представлена презентация, состоящая из 15-20 слайдов. На слайдах должны быть отражены: постановка задачи, использованные методы и алгоритмы, расчетные соотношения, структура комплекса программ, диаграмма классов, интерфейс, характеристики разработанного ПО, результаты проведенных исследований.

Дата выдачи задания « ____ » _____ 2020 г.

Руководитель курсового проекта

Барышникова М.Ю.
(Подпись, дата) (И.О.Фамилия)

Студент

Брянская Е.В.
(Подпись, дата) (И.О.Фамилия)

Оглавление

Введение.....	5
1. Аналитический раздел	7
1.1 Объекты сцены.....	7
1.2 Анализ алгоритмов удаления невидимых линий и поверхностей	8
1.2.1 Алгоритм Робертса	8
1.2.2 Алгоритм, использующий z-буфер	9
1.2.3 Алгоритм обратной трассировки лучей	9
1.2.4 Алгоритм Варнока	10
1.2.5 Вывод	10
1.3 Анализ алгоритмов закраски	12
1.3.1 Простая закраска.....	12
1.3.2 Закраска методом Гуро	12
1.3.3 Закраска Фонга.....	12
1.3.4 Вывод	13
1.4 Анализ моделей освещения	13
1.4.1 Модель Ламберта.....	13
1.4.2 Модель Фонга	14
1.4.3 Модель Блинна-Фонга.....	16
1.4.4 Вывод	16
1.5 Физическая модель поведения объектов сцены.....	17
1.6 Вывод.....	19
2. Конструкторский раздел	20
2.1 Функционал программы	20
2.2 Общий алгоритм работы программы	20
2.3 Алгоритм z-буфера.....	22
2.4 Алгоритм закраски по Гуро	23
2.5 Физика песка	24
2.6 Создание полигональной сетки	25
2.7 Вывод.....	26
3. Технологический раздел	27
3.1 Выбор и обоснование языка программирования и среды разработки	27
3.2 Строение объектов сцены и отношения между ними.....	28
3.3 Формат входных данных	29
3.4 Описание интерфейса	30

Вывод	31
4. Исследовательский раздел.....	32
4.3 План эксперимента	32
Вывод	34
Заключение.....	35
Список использованной литературы.....	36
Приложение А.....	37
Приложение Б	58
Приложение В.....	64

Введение

В современном мире роль компьютерной графики огромна, она применима везде, где нужно создание и обработка изображений и каких-либо цифровых данных. Спецэффекты в фильмах, реклама, компьютерные игры – всё это проявления компьютерной графики, которая используется повсеместно. Она применяется в большинстве инженерных и научных дисциплинах для передачи информации, её наглядного восприятия.

Одной из основных задач компьютерной графики является создание наиболее приближенного к реальной жизни изображения. В процессе работы необходимо учитывать свойства не только самого объекта, но и окружающих тел, среды. Существует много алгоритмов, которые решают данную задачу, но зачастую они достаточно затратные как по времени, так и по используемой памяти. Поэтому при моделировании какого-либо объекта или процесса необходимо корректно подбирать алгоритм, исходя из конкретной цели.

Цель данной работы - разработать программу моделирования работы песочных часов.

Измерительное время пользователь выбирает из предлагаемых вариантов. Программа даёт возможность регулирования положения источника освещения, камеры, её перемещения вокруг объекта. Пользователь может начать повторный отсчет по истечении времени работы прибора. При разработке программы необходимо учитывать прозрачность стекла и блики от источника освещения.

Для достижения поставленной цели необходимо решить следующие задачи:

- 1) описать объекты сцены, как они представляются и как будут заданы в программе;

- 2) проанализировать алгоритмы для визуализации трёхмерной сцены, при необходимости рассмотреть модификации, и обосновать выбор конкретного алгоритма;
- 3) разработать физическую модель песочных часов, наиболее приближенную к реальности;
- 4) реализовать выбранные алгоритмы;
- 5) реализовать интерфейс, отвечающий заданным требованиям и предоставляющий пользователю заявленные возможности;
- 6) разработать программное обеспечение, выполняющее поставленную задачу.

1. Аналитический раздел

В этом разделе будут описаны объекты сцены, проведён анализ некоторых алгоритмов, а также сделан выбор методов для реализации поставленной задачи.

1.1 Объекты сцены

Сцена состоит из следующих объектов:

- ограничивающая плоскость – расположена параллельно плоскости oxz ; Песочные часы – располагаются на ограничивающей плоскости. В них можно выделить следующие составляющие:
 1. стеклянная часть – её свойства необходимо учитывать при изображении теней, бликов, моделировании песка;
 2. песок – совокупность мелких песчинок, которая приходит в движение при отсчете времени;
 3. непрозрачные подставки (сверху и снизу) – представляют собой параллелепипеды, основания которых параллельны ограничивающей плоскости, а боковые ребра перпендикулярны ей;
- источник освещения – предполагается, что источник находится в бесконечности. Начальное положение источника указывается по умолчанию, но пользователь может поменять его положение;
- камера – начальное положение указывается по умолчанию, но пользователь может его изменить, воспользовавшись соответствующими кнопками в предлагаемом интерфейсе.

Песок «разбивается» на части:

- песок, который ещё не пересыпался, и песок, который уже пересыпался – будут представляться в виде полигональной сетки;

- движущиеся частички – представляются в виде каркаса треугольных пирамид.

1.2 Анализ алгоритмов удаления невидимых линий и поверхностей

Задача удаления невидимых линий и поверхностей является достаточно сложной, поэтому существует много алгоритмов её решения. При этом универсального решения, которое можно считать наилучшим для любой задачи, нет, поэтому в каждом конкретном случае необходимо обосновывать выбор алгоритма, исходя из цели моделирования.

В рамках данной курсовой работы наиболее сложной проблемой является достижение реалистичной визуализации процесса пересыпания песка из одной части стеклянной колбы в другую. Для этого нужно добиться плавности и равномерности движения частиц, так как именно эти факторы будут влиять на качество анимации. Ещё одним важным критерием выбора алгоритма удаления невидимых линий является быстродействие.

Рассмотрим некоторые из основных алгоритмов удаления невидимых линий и поверхностей, проанализируем их и выберем наиболее подходящий.

1.2.1 Алгоритм Робертса

Плюсом данного алгоритма является то, что используемые в нём математические методы являются достаточно мощными и точными, а сам алгоритм прост для понимания.

На первом этапе удаляются те рёбра или грани, которые экранируются самим телом, далее те, что экранируются другими телами сцены. И если тела, связаны отношением взаимного «протыкания», то удаляются невидимые линии пересечения тел.

К минусам алгоритма следует отнести тот факт, что вычислительная ёмкость растёт как квадрат числа объектов. То есть при большом количестве объектов на сцене алгоритм будет работать достаточно медленно, однако существуют различные способы его оптимизации, например, с использованием предварительной приоритетной сортировки вдоль оси z . [1]

1.2.2 Алгоритм, использующий z -буфер

К достоинствам можно отнести то, что это один из простейших алгоритмов удаления невидимых линий и поверхностей. Также он позволяет визуализировать пересечения этих поверхностей. Элементы сцены не нужно сортировать, следовательно, на это не тратится время.

Сцена может быть любой сложности, а так как размеры изображения ограничены размером экрана, то трудоёмкость линейно зависит от числа поверхностей.

Но у этого алгоритма есть и существенные недостатки: большой объём требуемой памяти и трудоёмкость реализации таких эффектов, как прозрачность и освещение, но существуют и специальные модификации, направленные на разрешение проблемы с их визуализацией.

1.2.3 Алгоритм обратной трассировки лучей

В данном алгоритме считается, что точка зрения находится в бесконечности на положительной полуоси z и поэтому все световые лучи параллельны этой оси. Траектория каждого луча отслеживается, чтобы определить, какие именно объекты сцены, если таковые существуют, пересекаются с данным лучом.

Этот алгоритм позволяет получать такие эффекты как отражение, преломление и т.д. Другими словами, изображение становится более реалистичным. Расчёт отдельной точки изображения производится

независимо от других, поэтому в этом алгоритме поддерживается высокая степень параллельности вычислений.

К недостаткам алгоритма обратной трассировки лучей следует отнести высокую вычислительную стоимость расчетов. Каждая точка изображения требует множества вычислительных операций. Очень много времени тратится на поиск пересечений, что зачастую тормозит работу алгоритма. [1]

1.2.4 Алгоритм Варнока

Единой версии этого алгоритма не существует. Но все модификации основываются на рекурсивном разбиении окна изображения.

Для каждого окна определяются многоугольники, которые связаны с ним, и те, у которых легко определить видимость, изображаются. Если нет, то разбиение на подокна продолжается до тех пор, пока либо нельзя будет принять однозначное решение, либо размер окна не станет равным одному пикселю.

На каждом этапе разбиения осуществляется определение расположения многоугольников относительно текущего окна.

Насколько быстро будет работать данный алгоритм, зависит от сложности сцены. Так как было решено использовать полигональную сетку, то это может затормозить выполнение алгоритма.

1.2.5 Вывод

Для наибольшей наглядности была составлена Таблица 1:

Таблица 1 - Сравнение алгоритмов

<div>Алгоритм</div> <div>Критерии сравнения</div>	Алгоритм Робертса	Алгоритм z-буфера	Алгоритм обратной трассировки лучей	Алгоритм Варнока
Пространство, в котором работает алгоритм	Объектное пространство	Пространство изображения	Пространство изображения	Пространство изображения
Сложность, N – количество граней; C – количество пикселей	$O(N^2)$	$O(CN)$	$O(CN)$	$O(CN)$
Эффективность для сложных сцен	Низкая	Высокая	Низкая	Средняя
Сложность реализации	Высокая	Низкая	Средняя	Средняя

Проанализировав алгоритмы удаления невидимых линий и поверхностей, можно сделать вывод, что наиболее предпочтительным для данной задачи является алгоритм с использованием z-буфера, так как он позволяет работать со сценой любой сложности и обеспечивает её обработку за меньшее время. Кроме того, при работе с освещением и закраской z-буфер легко подстраивается под модификации.

1.3 Анализ алгоритмов закраски

Будут рассмотрены три основных алгоритма: простая закраска, закраска по Гуро и закраска по Фонгу.

1.3.1 Простая закраска

Один из самых быстрых алгоритмов. В его основе лежит закон Ламберта, который говорит о том, что интенсивность отражённого света пропорциональна косинусу угла между направлением света и нормалью к поверхности.

Поверхность предметов, которая была изображена с помощью этого алгоритма, выглядит визуально матовой и блеклой. Это происходит потому, что вся грань закрашивается с постоянной интенсивностью. [1]

1.3.2 Закраска методом Гуро

Благодаря этому алгоритму можно получить сглаженное изображение. Сначала определяется интенсивность вершин, а затем с помощью билинейной интерполяции вычисляется интенсивность соответствующего пикселя.

Недостаток алгоритма – появление полос Маха, которое можно исправить, если увеличить число обрабатываемых граней, но это приводит к замедлению процесса визуализации. [3]

Закраску Гуро лучше всего использовать с простой моделью с диффузным отражением, так получается более реалистичное изображение. [1]

1.3.3 Закраска Фонга

В закраске по Фонгу интерполирование происходит по вектору нормали, в отличие от закраски по Гуро, где используется значение интенсивности. Тем

самым изображение становится более реалистичным, а зеркальные блики выглядят более правдоподобно.

Но у этого метода есть существенный недостаток: большие вычислительные затраты, так как сначала производится работа с нормальными, а затем по ним находятся соответствующие интенсивности. [1]

1.3.4 Вывод

Так как в текущей задаче будут использоваться полигональные сетки и, желательно, чтобы были сглажены границы многоугольников, то лучше всего подойдёт алгоритм закрашки по Гуро.

1.4 Анализ моделей освещения

Освещение тоже играет не последнюю роль в данной задаче, а в силу того, что будет моделироваться стекло, нужно подобрать соответствующую модель. Будет использоваться локальная модель освещения, которая учитывает только свет от источника.

1.4.1 Модель Ламберта

Является одной из самых простых моделей освещения (Рисунок 1). Она моделирует идеальное диффузное освещение, сама поверхность выглядит одинаково яркой со всех направлений. При реализации используется формула (1).

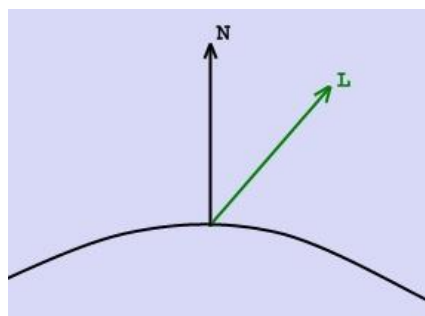


Рисунок 1 – Модель Ламберта

$$I = k_d(\vec{N}, \vec{L}), \quad (1)$$

где k_d – коэффициент диффузного освещения

\vec{N} – вектор нормали к поверхности в данной точке

\vec{L} – направление из точки на источник

Считается, что при попадании на поверхность свет равномерно рассеивается по всем направлениям. В основе расчетов лежит закон Ламберта.

1.4.2 Модель Фонга

Представляет классическую модель освещения, комбинируя диффузную составляющую (модель Ламберта) и зеркальную составляющую. Таким образом, на объекте может появиться ещё и блик, что придаёт больше реалистичности изображению. Нахождение блика на объекте определяется из закона равенства углов падения и отражения. Если наблюдатель находится вблизи углов отражения, то яркость соответствующей точки повышается. [6]

Падающий и отраженный лучи лежат в одной плоскости с нормалью к отражающей поверхности в точке падения, и она делит угол между лучами на две равные части (Рисунок 2), где \vec{V} – направление на наблюдателя, \vec{L} – направление на источник, \vec{R} – отраженный луч.

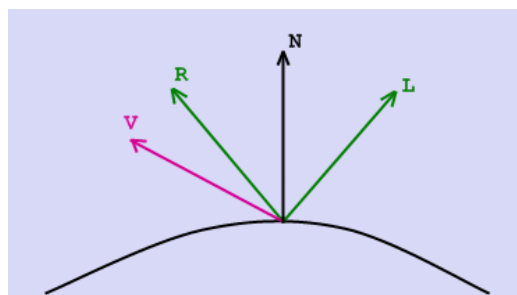


Рисунок 2 - Модель Фонга

Эта модель учитывает фоновую, рассеянную компоненты освещения и глянецовые блики.

Для удобства все векторы считаются единичными, чтобы сказать, что косинус угла между ними совпадает со скалярным произведением.

Интенсивность по модели Фонга рассчитывается по формуле (2).

$$I = k_a I_a + k_d (\vec{N}, \vec{L}) + k_s (\vec{N}, \vec{V})^p, \quad (2)$$

где k_a – коэффициент фонового освещения;

I_a – мощность фонового освещения (заранее задаётся для всей сцены);

k_d – коэффициент диффузного освещения;

\vec{N} – вектор нормали к поверхности в данной точке;

\vec{L} – направление из точки на источник;

k_s – коэффициент зеркального освещения;

\vec{V} – направление на наблюдателя;

p – коэффициент блеска.

При фиксированном положении поверхности относительно источника освещения фоновую и рассеянную составляющие можно вычислить только один раз, так как они не зависят от положения наблюдателя. А зеркальная компонента зависит, поэтому её надо вычислять каждый раз, когда меняется положение наблюдателя.

Модель Фонга учитывает только свойства текущей точки и источника освещения, а такие эффекты как рассеивание, отражение от других тел игнорируются.

1.4.3 Модель Блинна-Фонга

Есть большое сходство с моделью Фонга, только она исключает расчёт отражённого луча, что сильно упрощает вычисления, тем самым, экономится время работы. Но существенной разницы нет.

В этой модели используется медианный вектор, который является единичным и находится посередине между вектором, указывающим направление обзора, и вектором направления освещения.

Чем ближе такой вектор к нормали поверхности, тем больше будет вклад зеркальной компоненты.

Благодаря тому, что измеряется угол между нормалью и медианным вектором (а не между вектором направления наблюдения и вектором отражения, как в модели Фонга), не будет проблемы с резкой границей области зеркального отражения. [7]

1.4.4 Вывод

В качестве подходящей модели освещения была выбрана модель Фонга, она позволяет изобразить более реалистичное изображение, чем модель Ламберта, так как учитывает зеркальную составляющую и не требует дополнительных расчётов.

1.5 Физическая модель поведения объектов сцены

Как было уже упомянуто, основной целью является моделирование реалистичного процесса работы песочных часов. Вместе с этим большое внимание уделяется эффективности отобранных алгоритмов. Задача непростая, так как песок будет состоять из большого числа частиц, каждая из которых обладает скоростью, размером и т.д. и которые должны равномерно пересыпаться.

В начальном положении часов весь песок находится в верхней части колбы. Далее с течением времени он пересыпается в нижнюю. То есть, объём песка в верхней части должен равномерно уменьшаться, причём должны соблюдаться соответствующие физические законы. Аналогичная картина будет наблюдаться в нижней части, только количество песка должно увеличиваться. Время пересыпания должно соответствовать реальному.

Песок будет разделён на три части: верхнюю, пересыпающуюся и нижнюю. На Рисунок 3 показаны песочные часы в действии, можно различить все выделенные для работы части.



Рисунок 3 - Песочные часы

1. Верхняя часть

Представляет собой две связанные компоненты: боковая поверхность, повторяющая форму часов, и поверхность, образующая выемку в верхней

части всего объёма. Обе составляющие задаются с помощью полигональной сетки.

Поверхность, образующая выемку в песке, определяется уравнением (3).

$$y(x, z) = \frac{e^{-\frac{(\frac{x^2}{k} + \frac{z^2}{k})^2}{10}}}{t}, \quad (3)$$

где коэффициент k рассчитывается по формуле $k = 0.16 \sqrt{\frac{h^5}{100}}$ и регулирует ширину выемки, h – максимальная высота песка в верхней колбе, t – время, прошедшее с момента начала работы часов,

С течением времени, когда глубина выемки достигнет своего максимума, вся система верхней части песка начнёт “проседать”, то есть уменьшаться в объёме. Так будет продолжаться до тех пор, пока всё содержимое часов не окажется в нижней части.

2. Нижняя часть

Представляет собой растущую со временем горку из песка, поверхность которой описывается с помощью формулы (4).

$$y(x, z) = -\frac{e^{-\frac{(\frac{x^2}{k} + \frac{z^2}{k})^2}{5}}}{t}, \quad (4)$$

где коэффициент $k = 55$, был подобран экспериментально в целях придания наибольшей реалистичности, t – время, прошедшее с момента начала работы часов.

3. Пересыпающаяся часть

Представляет собой совокупность частиц, которые имеют форму треугольных пирамид, описанных с помощью каркасной модели.

Все частицы будут падать с ускорением, и их скорость будет изменяться по закону (5):

$$v = at, \quad (5)$$

где v – скорость, a – ускорение, t – время.

И соответственно ордината будет меняться в соответствии с формулой (6).

$$y = y_0 - \frac{at^2}{2}, \quad (6)$$

где y_0 , y – начальная и текущая ординаты, a – ускорение, t – время.

1.6 Вывод

В данном разделе были рассмотрены алгоритмы удаления невидимых линий и поверхностей, закраски, проанализирована физическая модель песочных часов.

В результате анализа для моделирования был выбран алгоритм z-буфера (удаление невидимых линий и поверхностей), метод Гуро (закраска), а песок будет представляться в виде системы частиц, которая подчиняется соответствующим физическим законам.

Входным данным является измерительное время, которое пользователь выбирает из предлагаемого списка. Пользователь может скорректировать расположение камеры, источника освещения. Все изменения, которые могут быть внесены пользователем, должны быть учтены при разработке графического интерфейса программного продукта.

2. Конструкторский раздел

В данном разделе будут более подробно описаны выбранные методы и алгоритмы, приведены схемы их работы, разобраны основные возможности программы.

2.1 Функционал программы

Программа обладает следующими возможностями:

- выбора из предлагаемого списка измерительного времени;
- корректировки положения источника освещения и камеры с помощью соответствующих кнопок;
- отслеживания пройденного времени в процентах;
- повторного отсчета по истечении времени работы прибора.

2.2 Общий алгоритм работы программы

Общий алгоритм выглядит следующим образом (Рисунок 4):

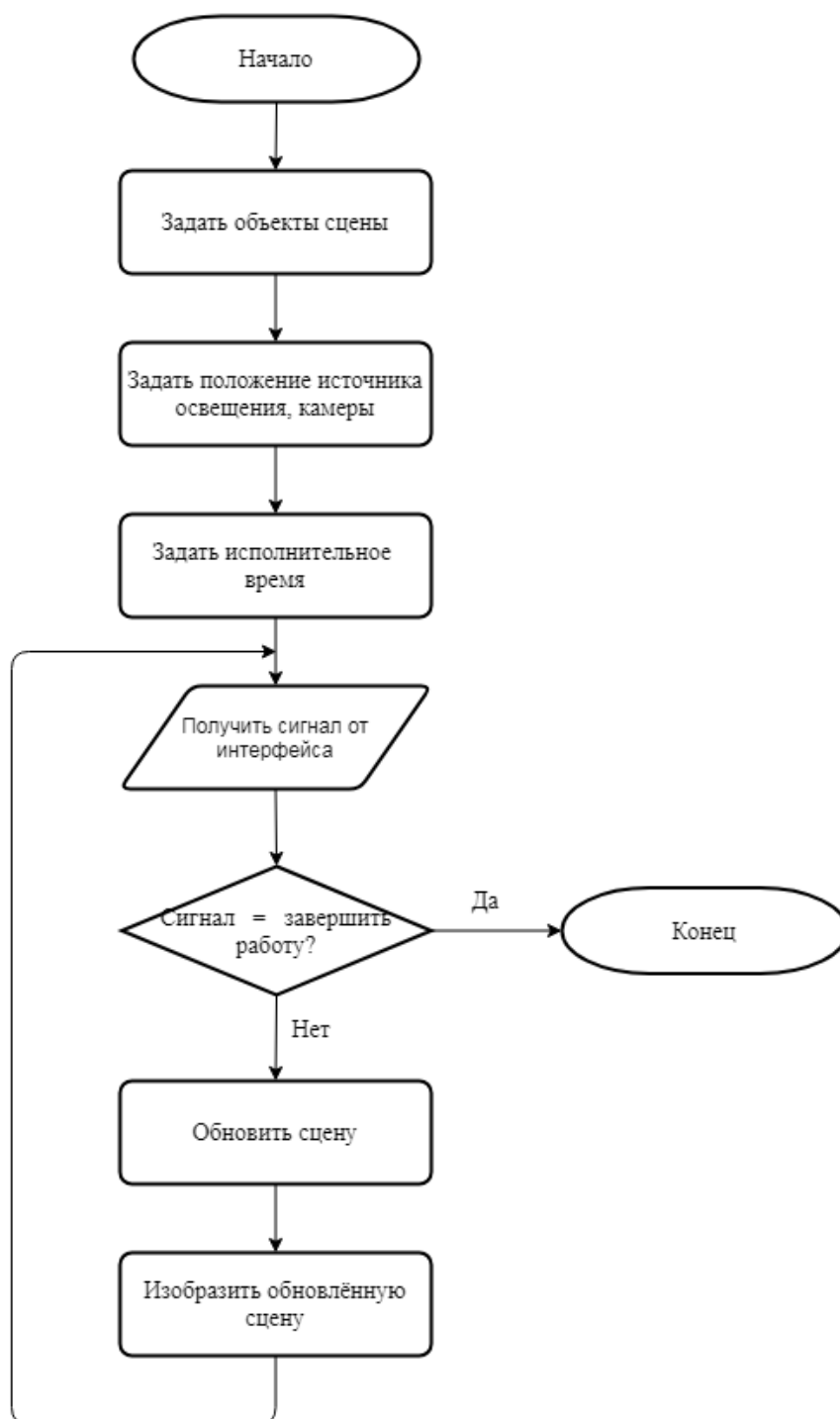


Рисунок 4 - Общий алгоритм

В качестве сигнала может выступать команда перемещения камеры вокруг объекта, смены положения источника освещения, начала работы, повторного отсчёта времени (только при окончании уже запущенного процесса), завершения работы.

2.3 Алгоритм z-буфера

Схема этого алгоритма представлена ниже на Рисунок 5.

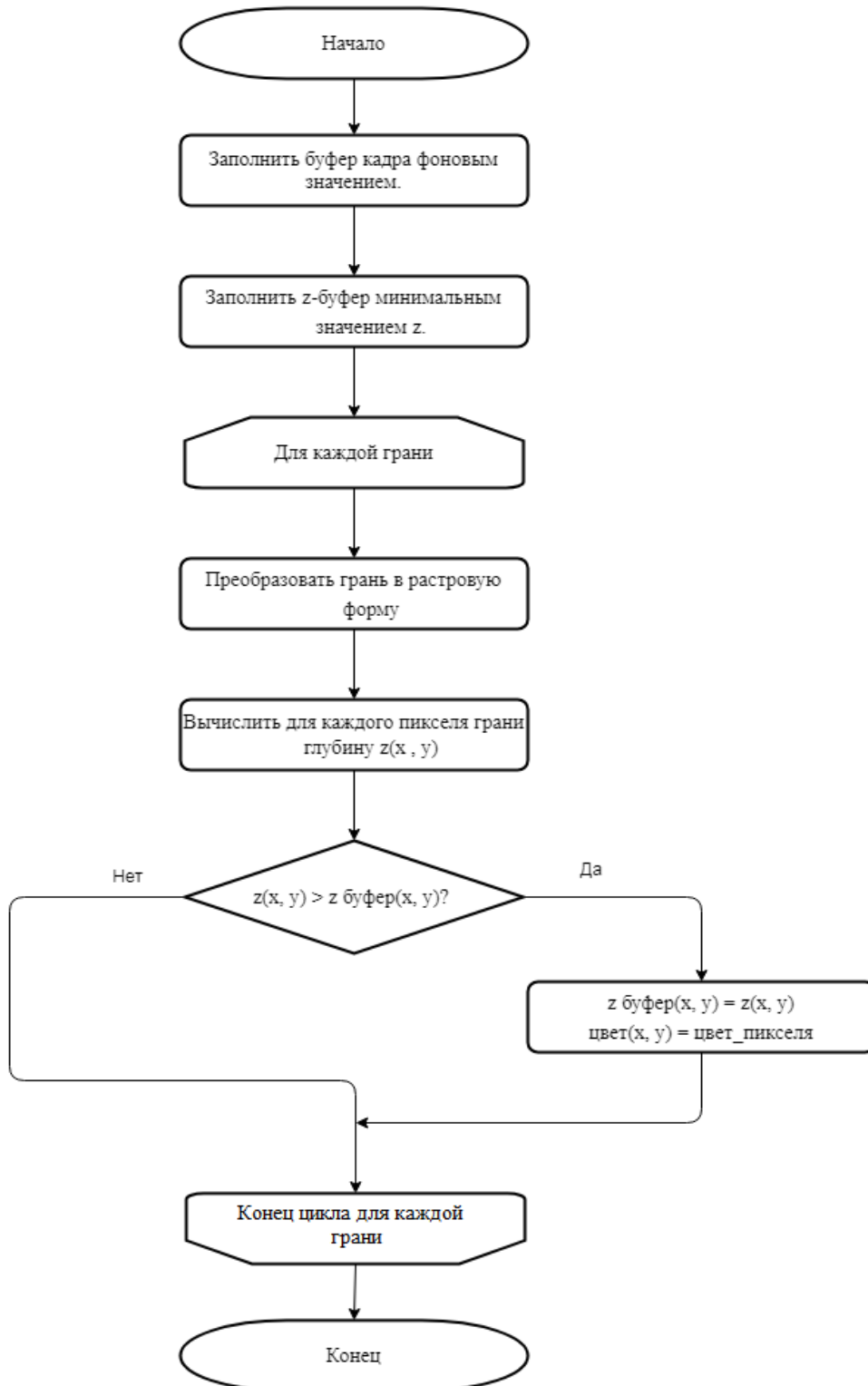


Рисунок 5 - Алгоритм z-буфера

Трудоёмкость этого алгоритма линейно зависит от числа поверхностей на сцене.

2.4 Алгоритм закрашки по Гуро

Сначала определяется интенсивность вершин, а затем вычисляется интенсивность соответствующего пикселя. Схема представлена на Рисунок 6.

Этот метод хорошо сочетается с построчным сканированием. Для каждой сканирующей строки определяются её точки пересечения с рёбрами. В этих точках интенсивность вычисляется с помощью линейной интерполяции интенсивностей в вершинах рёбра. Затем для всех пикселей, находящихся внутри многоугольника и лежащих на сканирующей строке, аналогично вычисляется интенсивность.

Можно заранее высчитать нормали к каждой вершине, чтобы не делать эти вычисления в процессе работы алгоритма. Интенсивность в вершинах полигона определяется как скалярное произведение вектора светового луча и нормали в вершине.

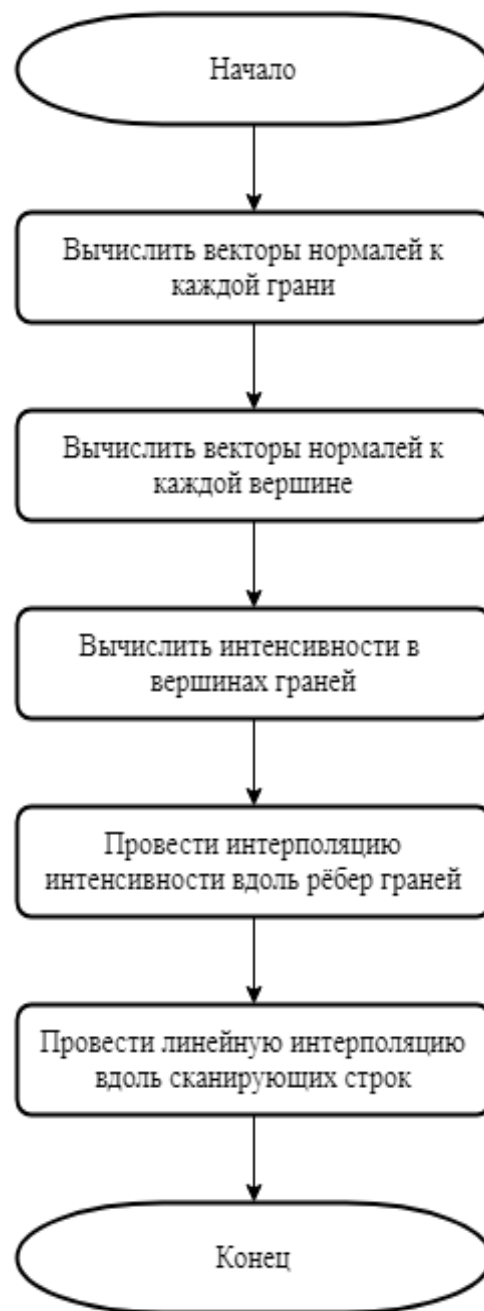


Рисунок 6 - Алгоритм закрашки по Гуро

2.5 Физика песка

На Рисунок 7 изображена схема однократного обновления составляющих песка.

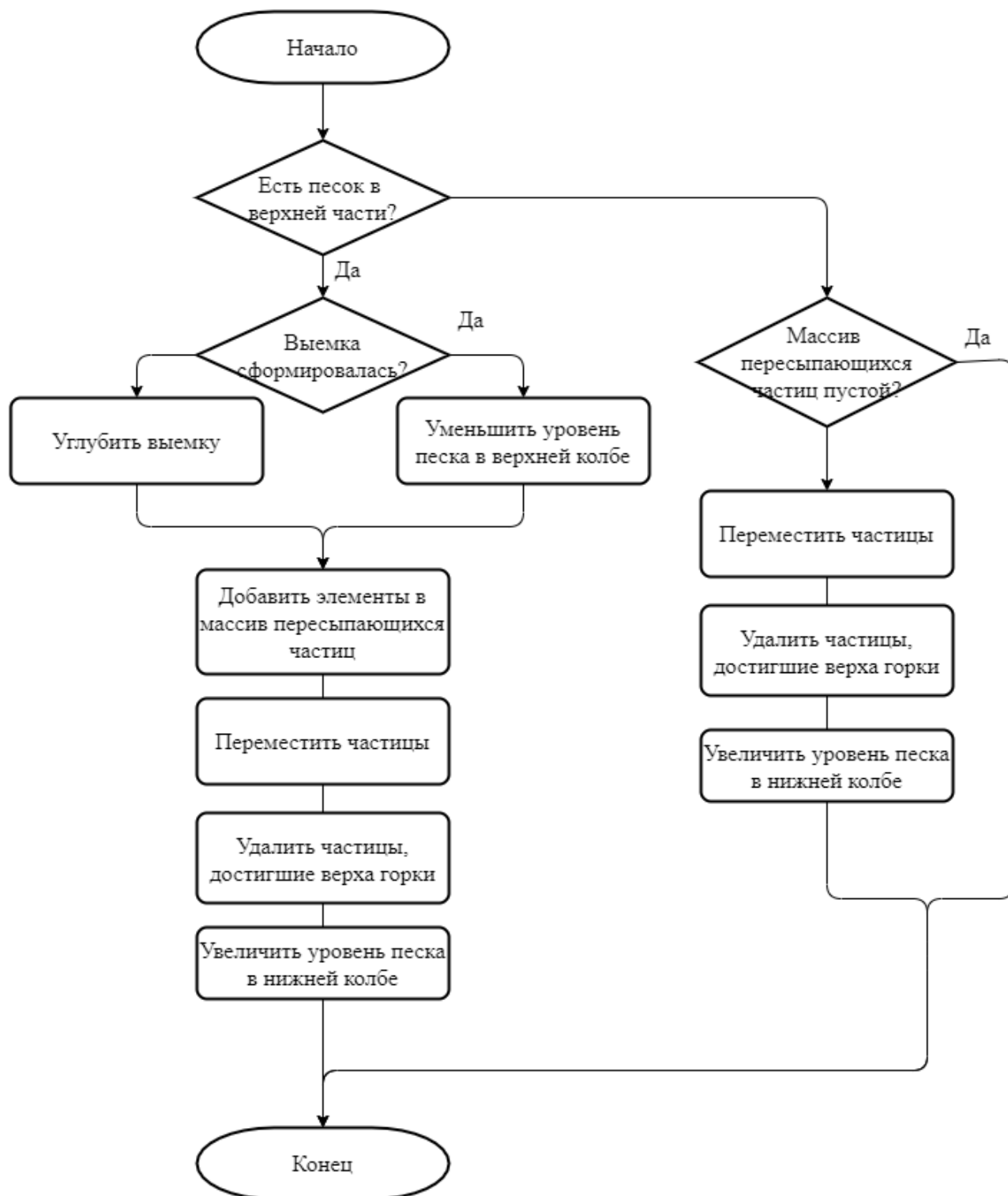


Рисунок 7 - Процесс однократного обновления песка

Изначально весь песок находится в верхней части часов. Постепенно он пересыпается в нижнюю. Весь процесс происходит до тех пор, пока песок полностью не окажется в нижней колбе. Количество песка в верхней части должно уменьшаться, в нижней, наоборот, увеличиваться. Отдельное внимание уделено формированию выемки в песке в верхней колбе и горки в нижней.

2.6 Создание полигональной сетки

В ходе работы программы активно используется полигональная сетка. Это совокупность вершин, рёбер и граней, которые определяют форму многогранного объекта в трёхмерной компьютерной графике и объёмном моделировании.

Гранями обычно являются треугольники, четырёхугольники или многоугольники, при этом у каждой грани высчитывается нормаль.

Недостатком этого метода является его приблизительность. Правда, видимые недочёты можно исправить, сделав разбиение более детальным, но это приведёт к дополнительным затратам по памяти и временным затратам.

Примеры использования полигональной сетки приведены на Рисунок 8.

В рамках текущей задачи в качестве граней выбраны четырёхугольники. И построение полигональной сетки осуществляется следующим образом: зная число разбиений по осям, можно по рассматриваемой формуле плоскости найти неизвестное, тем самым заполнив

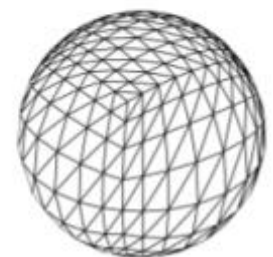
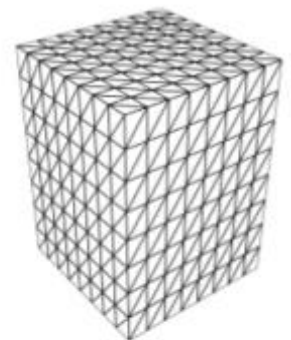


Рисунок 8 - Примеры использования полигональной сетки

«ячейку» сетки.

Далее все полученные точки соединяются в четырёхугольники. Все параметры разбиения заданы таким образом, чтобы обеспечить не только качественную визуализацию, но и минимальные, насколько это возможно, затраты по памяти.

2.7 Вывод

В данном разделе были подробно разобраны выбранные алгоритмы, рассмотрены схемы их работы, перечислены основные возможности программы.

3. Технологический раздел

В этом разделе будет обоснован выбор языка программирования и среды разработки, рассмотрена диаграмма основных классов, описан формат входных данных и разобран интерфейс, предлагаемый пользователю.

3.1 Выбор и обоснование языка программирования и среды разработки

При выборе языка программирования важно учитывать много факторов в зависимости от задачи. Из-за специфики модели в приложении на сцене будет большое количество объектов. К тому же изображение будет постоянно обновляться, что даёт дополнительную нагрузку.

В качестве языка программирования (ЯП) был выбран Си++. На это есть несколько причин.

1. При разработке этого программного продукта предусматривается использование объектно-ориентированного подхода, а его как раз поддерживает этот ЯП.
2. Этот язык обладает большой производительностью, что важно в данной задаче.
3. Предоставляется большое число шаблонов и библиотек, которые ускоряют работу и улучшают эффективность алгоритмов.
4. Накопившийся опыт работы с этим языком во время обучения упростит задачу реализации, также в свободном доступе в большом количестве есть необходимая литература и документация.

В качестве среды разработки был выбран QtCreator, который:

1. хорошо знаком, так как активно использовался во время обучения;
2. предоставляет удобную графическую библиотеку;
3. позволяет работать с графическим интерфейсом;

4. является бесплатным приложением.

3.2 Строение объектов сцены и отношения между ними

На Рисунок 9 представлена схема взаимодействия основных объектов сцены и показаны их составляющие.

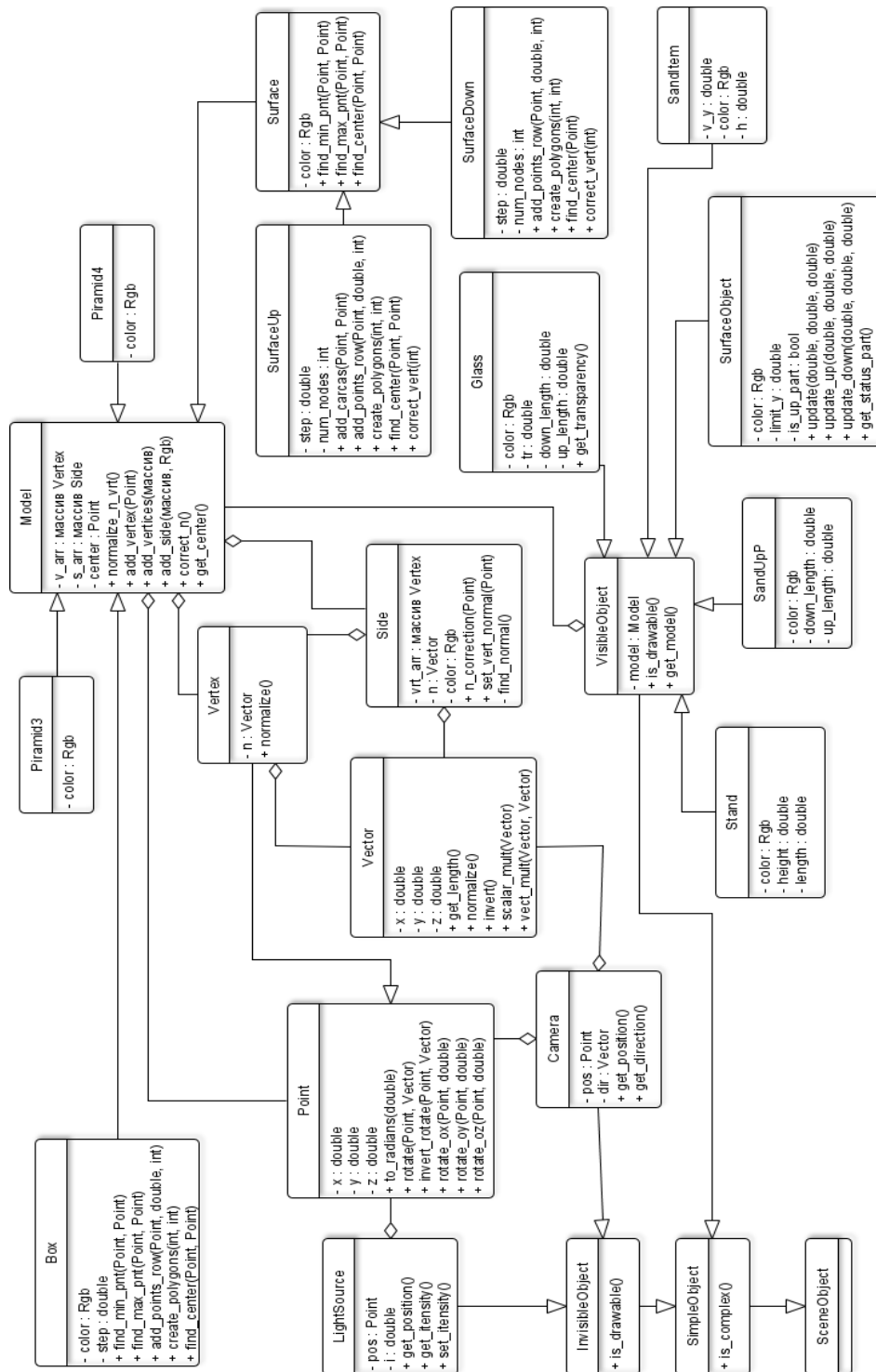


Рисунок 9 - Основные объекты сцены

Используются следующие основные классы:

- **Примитивы**
 - Point – класс точки в трёхмерном пространстве;
 - Vector – класс вектора в трёхмерном пространстве;
 - Vertex – класс вершины;
 - Side – класс грани;
 - Model – класс модели;
 - Box – класс параллелепипеда;
 - Piramid3 – класс треугольных пирамид;
 - Piramid4 – класс четырёхугольных пирамид;
 - Surface – класс поверхностей;
 - SurfaceUp – класс поверхности песка в верхней части часов;
 - SurfaceDown – класс поверхности песка в нижней части часов.
- **Основные объекты сцены**
 - Camera – класс камеры
 - LightSource – класс источника света
 - Glass – класс стекла
 - SandUpP – класс поверхности песка
 - SandItem - класс песчинок
 - Stand – класс подставок песочных часов

Также были реализованы классы, необходимые для корректного функционирования всей системы в целом, поддержания её структуры и обеспечения согласованной работы.

3.3 Формат входных данных

Входным данным выступает измерительное время, которое пользователь выбирает из предлагаемого списка.

Пользователь также может скорректировать через интерфейс расположение камеры, источника освещения.

3.4 Описание интерфейса

Пользователю предоставляется следующий интерфейс (Рисунок 10). На панели справа можно выбрать измерительное время из выпадающего списка (Рисунок 11), по умолчанию оно выставлено на 2 минуты. Также пользователь может изменять положение источника освещения, камеры через графический интерфейс, а также с помощью клавиатуры.

- Клавиши «w», «s» – переместить камеру вверх/вниз;
- «a», «d» – переместить камеру влево/вправо;
- «u», «j» – повернуть камеру вверх/вниз;
- «h», «k» - повернуть камеру влево/вправо.

Можно повторить отсчёт времени, нажав на кнопку «Старт», но это возможно только после того, как текущий процесс завершится).

Снизу есть специальная шкала, показывающая процент прошедшего времени.

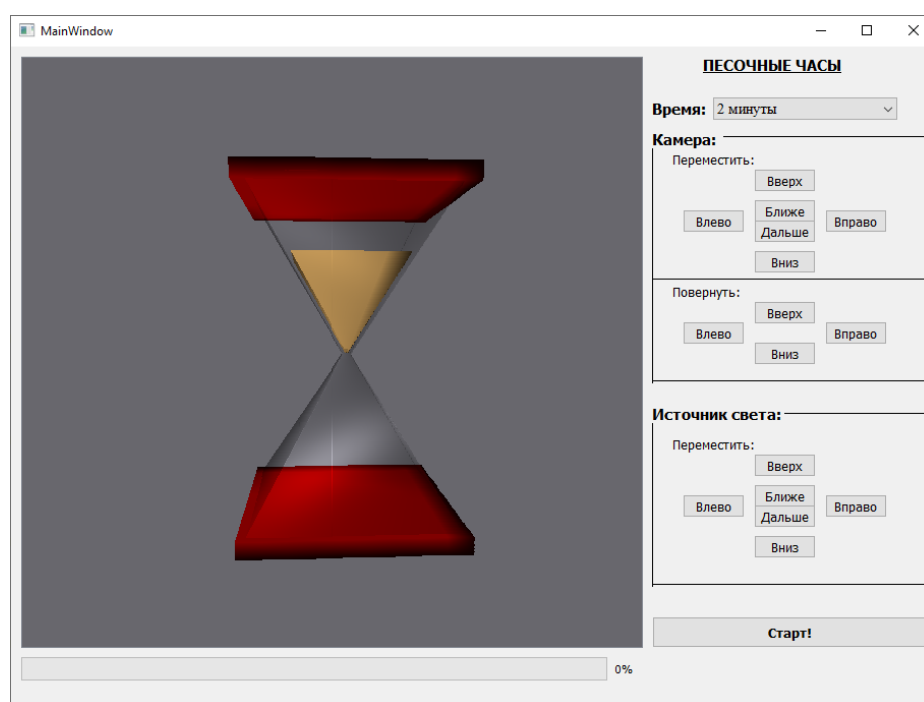


Рисунок 10 - Интерфейс

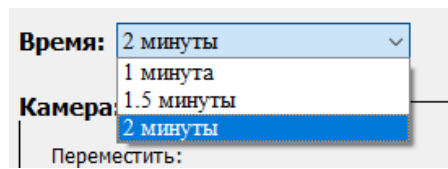


Рисунок 11 - Варианты измеряемого времени

Вывод

В этом разделе был выбран язык программирования и среда разработки, рассмотрена uml-диаграмма основных классов, описан формат входных данных и подробно разобран интерфейс приложения.

4. Исследовательский раздел

В текущем разделе будет поставлена цель эксперимента, проведён сам эксперимент и сделаны соответствующие выводы.

Поскольку на сцене присутствует большое число объектов, которые задаются с помощью полигональной сетки, остро возникает вопрос о том, насколько сильно сказывается величина шага на всё изображение, какое значение следует выбрать для её задания в конкретной задаче, чтобы картина была наиболее реалистичной, и при этом, минимизировать затраты как по памяти, так и по времени.

4.1 Характеристики ПК

Все исследования проводились на компьютере со следующими характеристиками:

- ОС – Windows 10 Pro
- Процессор – Inter Core i7 10510U (1800 МГц)
- Объём ОЗУ – 16 Гб

4.2 Цель эксперимента

Цель эксперимента - определить, как влияет шаг полигональной сетки на FPS (frame per second, количество кадров в секунду) и на визуальные характеристики.

4.3 План эксперимента

Эксперимент проводится на часах, в которых измерительное время выставлено в 1 минуту, и в качестве результирующего FPS берётся среднее значение, замеры производятся несколько раз для большей

правдоподобности. Оценка реалистичности изображения проводится субъективно путём сравнения полученных кадров.

Шаг сетки $step \in \{ 5, 10, 20, 30, 40, 50, 60 \}$.

4.4 Результаты эксперимента

Измерение FPS

Результаты измерения FPS представлены в Таблица 2 и на Рисунок 12.

Таблица 2 - Результаты замеров

Шаг	5	10	20	30	40	50	60
FPS	24	53	71	80	82	81	86

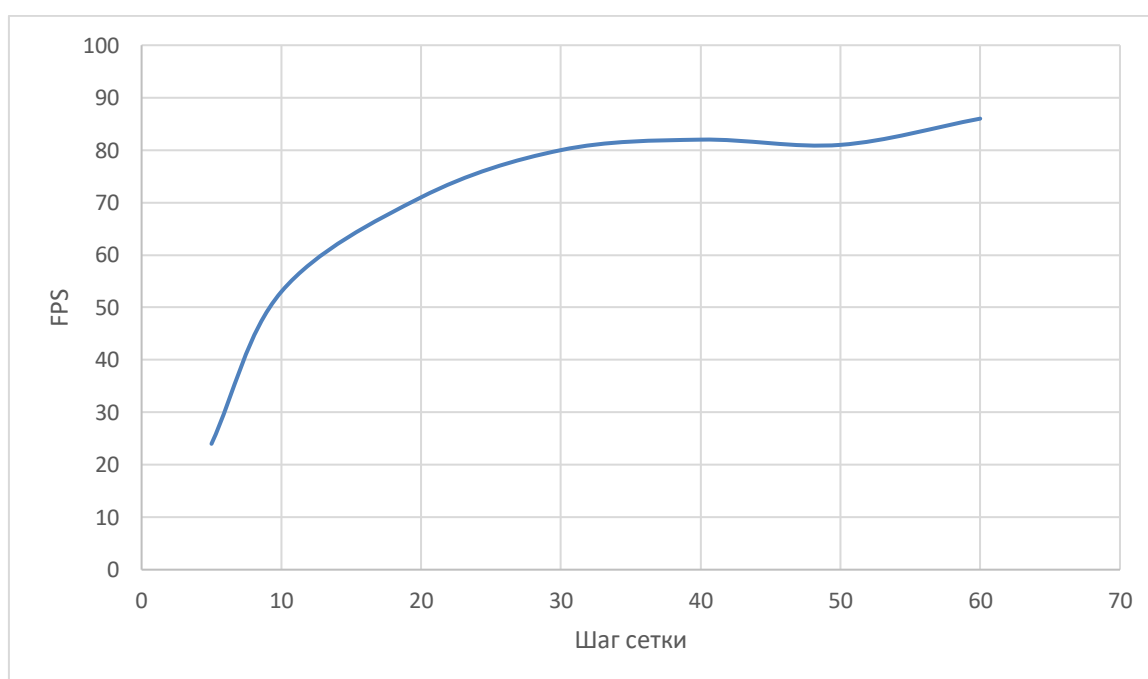
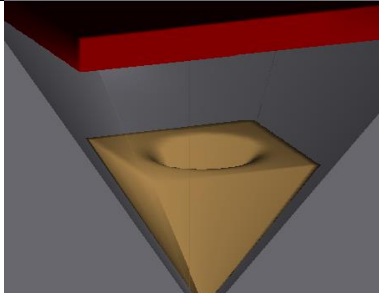
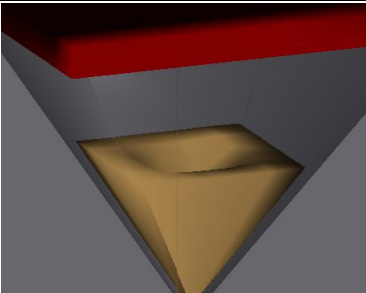
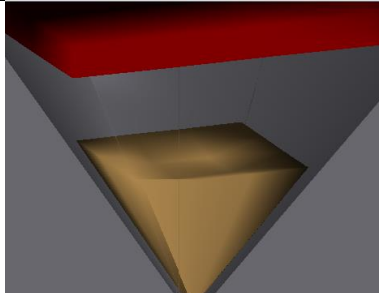
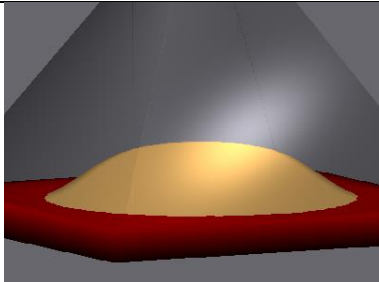
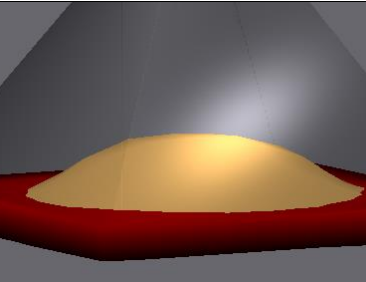
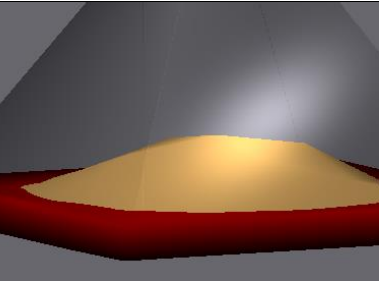


Рисунок 12- Результаты замеров

Визуальная оценка

Для наиболее наглядной демонстрации были выбраны эксперименты со $step \in \{5, 30, 60\}$, и в Таблица 3 представлены результаты для верхней и нижней колб песочных часов.

Таблица 3 - Визуальный результат эксперимента

5	30	60
		
		

Вывод

В результате эксперимента было выявлено, что при повышении детализации значительно возрастают временные затраты на визуализацию (например, количество кадров в секунду при шаге сетки равном 5 примерно в 3.3 раза меньше, чем при 30). Кроме того, сравнивая полученные результаты, можно заметить, что при шаге равном 30 изображение выглядит более правдоподобно, чем при очень маленьком значении. Таким образом, не всегда требуется использовать сильную детализацию, это зависит от поставленной задачи, в рамках текущего проекта лучше выбрать значение около 30.

Заключение

В процессе выполнения поставленной задачи в рамках курсового проекта была разработана программа моделирования работы песочных часов.

Были изучены, проанализированы и реализованы алгоритмы удаления невидимых линий и поверхностей, закраски. Все алгоритмы выбирались, исходя из поставленной цели. Была разработана физическая модель поведения объектов на сцене с учётом особенностей всей системы и создано программное обеспечение, отвечающее всем требованиям, которые были заявлены ранее.

По результатам проведенного эксперимента была выявлена зависимость количества кадров в секунду от шага полигональной сетки, а также найдено примерное значение шага, при котором изображение выглядит более реалистично.

Список использованной литературы

1. Роджерс Д. Алгоритмические основы машинной графики: Пер. с англ. – М.: Мир, 1989. – 512 с.
2. Методы представления дискретных трехмерных данных [Электронный ресурс]. – Режим доступа:
https://www.graphicon.ru/oldgr/ru/library/multires_rep/index.html (дата обращения 28.10.20)
3. Алгоритмы закраски [Электронный ресурс]. – Режим доступа:
https://studbooks.net/2248060/informatika/odnotonnaya_zakraska_metod_graneniya (дата обращения 29.10.20)
4. Обзор алгоритмов построения теней в реальном времени [Электронный ресурс]. – Режим доступа:
<https://www.ixbt.com/video/realtimeshadows.shtml> (дата обращения 04.11.20)
5. Ошаровская Е.В., Солодка В.И. Синтез трехмерных объектов с помощью полигональных сеток, Цифровые технологии, 2012, № 12, 2-9
6. Простые модели освещения [Электронный ресурс]. – Режим доступа:
<http://grafika.me/node/344> (дата обращения 10.11.20)
7. Продвинутое освещение. Модель Блинна-Фонга [Электронный ресурс]. – Режим доступа:
<https://habr.com/ru/post/353054/> (дата обращения 21.10.20)
8. Куров А.В., Курс лекций по дисциплине “Компьютерная графика” [Текст]
9. Польский С.В., Компьютерная графика: учебн.-методич. Пособие. – М.: ГОУ ВПО МГУЛ, 2008. – 38 с.

Приложение А

Приведены листинги классов примитивов.

Листинг А.1 – Класс Точки

```
class Point
{
public:
    double x, y, z;

    Point();
    Point(double x, double y, double z);
    Point(const Point& other);

    virtual ~Point();

    void operator =(const Point& other);

    double to_radians(double angle);
    void rotate(const Point& center, const Vector& angles);
    void invert_rotate(const Point &center, const Vector
&angles);

    void rotate_ox(const Point& center, double k);
    void rotate_oy(const Point& center, double k);
    void rotate_oz(const Point& center, double k);
};

Point::Point() : x(0), y(0), z(0) {}
Point::~~Point() {}

Point::Point(double data_x, double data_y, double data_z) :
    x(data_x), y(data_y), z(data_z) {}

Point::Point(const Point& other) :
    x(other.x), y(other.y), z(other.z) {}

void Point::operator=(const Point &other)
{
    this->x = other.x;
    this->y = other.y;
    this->z = other.z;
}

double Point::to_radians(double angle)
{
    return angle * PI / 180;
}

void Point::rotate(const Point &center, const Vector &angles)
```

```

{
    rotate_oy(center, angles.y);
    rotate_ox(center, angles.x);
    rotate_oz(center, angles.z);
}

void Point::invert_rotate(const Point &center, const Vector
&angles)
{
    rotate_oz(center, angles.z);
    rotate_ox(center, angles.x);
    rotate_oy(center, angles.y);
}

void Point::rotate_ox(const Point &center, double angle)
{
    double y_temp, z_temp;

    y_temp = center.y + (this->y - center.y) * cos(angle) +
(this->z - center.z) * sin(angle);
    z_temp = center.z + (this->z - center.z) * cos(angle) -
(this->y - center.y) * sin(angle);

    this->y = y_temp;
    this->z = z_temp;
}

void Point::rotate_oy(const Point &center, double angle)
{
    double x_temp, z_temp;

    x_temp = center.x + (this->x - center.x) * cos(angle) -
(this->z - center.z) * sin(angle);
    z_temp = center.z + (this->z - center.z) * cos(angle) +
(this->x - center.x) * sin(angle);

    this->x = x_temp;
    this->z = z_temp;
}

void Point::rotate_oz(const Point &center, double angle)
{
    double x_temp, y_temp;

    x_temp = center.x + (this->x - center.x) * cos(angle) +
(this->y - center.y) * sin(angle);
    y_temp = center.y + (this->y - center.y) * cos(angle) -
(this->x - center.x) * sin(angle);

    this->x = x_temp;
    this->y = y_temp;
}

```

Листинг A.2 – Класс Вектора

```
class Vector
{
public:
    double x, y, z;

    Vector();
    Vector(double data_x, double data_y, double data_z);
    Vector(const Vector& other);
    Vector(const Point& begin_pnt, const Point& end_pnt);

    virtual ~Vector();

    double get_length() const;
    void normalize();
    void invert();
    double scalar_mult(const Vector& other);
    Vector vect_mul(const Vector& v1, const Vector& v2) const;

    Vector operator +(const Vector& other);
    void operator +=(const Vector& other);
    void operator =(const Vector& other);
};

Vector::Vector() : x(0), y(0), z(0) {}

Vector::~Vector() {}

Vector::Vector(double data_x, double data_y, double data_z) :
    x(data_x), y(data_y), z(data_z) {}

Vector::Vector(const Vector& other) :
    x(other.x), y(other.y), z(other.z) {}

Vector::Vector(const Point &begin_pnt, const Point &end_pnt)
{
    x = end_pnt.x - begin_pnt.x;
    y = end_pnt.y - begin_pnt.y;
    z = end_pnt.z - begin_pnt.z;
}

double Vector::get_length() const { return sqrt(x*x+y*y + z*z); }

void Vector::normalize()
{
    double len = this->get_length();

    if (len < EPS)
```

```

        throw error::InvalidOperation(__FILE__, typeid
(*this).name(), __LINE__ - 1);

        x /= len;
        y /= len;
        z /= len;
    }

void Vector::invert()
{
    x = -x;
    y = -y;
    z = -z;
}

double Vector::scalar_mult(const Vector &other)
{
    return this->x * other.x + this->y * other.y + this->z *
other.z;
}

Vector Vector::vect_mul(const Vector &v1, const Vector &v2)
const
{
    Vector result;

    result.x = v1.y * v2.z - v1.z * v2.y;
    result.y = v1.z * v2.x - v1.x * v2.z;
    result.z = v1.x * v2.y - v1.y * v2.x;

    return result;
}

Vector Vector::operator+(const Vector &other)
{
    Vector result;

    result.x = this->x + other.x;
    result.y = this->y + other.y;
    result.z = this->z + other.z;

    return result;
}

void Vector::operator+=(const Vector &other)
{
    this->x += other.x;
    this->y += other.y;
    this->z += other.z;
}

void Vector::operator=(const Vector &other)
{

```



```

    this->x = other.x;
    this->y = other.y;
    this->z = other.z;
}

```

Листинг А.3 – Класс Вершины

```

class Vertex : public Point
{
public:
    Vector n;

    Vertex();
    Vertex(double data_x, double data_y, double data_z);
    Vertex(const Point& other);
    Vertex(const Vertex& other);

    virtual ~Vertex();

    bool operator==(const Vertex& other);
    void normalize();
};

Vertex::Vertex() {}

Vertex::~~Vertex() {}

Vertex::Vertex(double data_x, double data_y, double data_z) :
    Point(data_x, data_y, data_z) {}

Vertex::Vertex(const Point& other) : Point(other) {}

Vertex::Vertex(const Vertex& other) : Point(other), n(other.n) {}

bool Vertex::operator==(const Vertex &other)
{
    if (fabs(this->x - other.x) > EPS) return false;
    if (fabs(this->y - other.y) > EPS) return false;
    if (fabs(this->z - other.z) > EPS) return false;

    return true;
}

void Vertex::normalize() { n.normalize(); }

```

Листинг А.4 – Класс Грани

```

class Side
{
public:

```

```

    vector<shared_ptr<Vertex>> vertex_arr;
    Vector n;
    QRgb color;

    Side();
    Side(vector<shared_ptr<Vertex>> vertex_arr, const Point&
control_p, QRgb color);

    virtual ~Side();

    void n_correction(const Point& control_p);
    void set_vert_normal(const Point& control_p);

private:
    void _find_normal();
};

Side::Side() {}

Side::~Side() {}

Side::Side(vector<shared_ptr<Vertex>> v_arr, const Point&
control_p, QRgb data_color)
{
    if (v_arr.size() < 2)
        throw error::DegenerateSide(__FILE__, typeid
(*this).name(), __LINE__ - 1, v_arr.size());

    color = data_color;
    vertex_arr = v_arr;

    _find_normal();
    n_correction(control_p);

    for (auto vertex : vertex_arr)
        vertex->n += this->n;
}

void Side::n_correction(const Point &control_p)
{
    Vector temp(control_p, *vertex_arr[0]);

    if (this->n.scalar_mult(temp) < 0)
        n.invert();
}

void Side::set_vert_normal(const Point& control_p)
{
    _find_normal();
    n_correction(control_p);
}

```

```

        for (auto vertex : vertex_arr)
            vertex->n += this->n;
    }

void Side::_find_normal()
{
    Vertex p1 = *vertex_arr[0];
    Vertex p2 = *vertex_arr[1];
    Vertex p3 = *vertex_arr[2];

    n.x = (p2.y - p1.y) * (p3.z - p1.z) - (p3.y - p1.y) * (p2.z - p1.z);
    n.y = (p3.x - p1.x) * (p2.z - p1.z) - (p2.x - p1.x) * (p3.z - p1.z);
    n.z = (p2.x - p1.x) * (p3.y - p1.y) - (p3.x - p1.x) * (p2.y - p1.y);

    n.normalize();
}

```

Листинг A.5 – Класс Модели

```

class Model
{
public:
    vector<shared_ptr<Vertex>> v_arr;
    vector<shared_ptr<Side>> s_arr;

    Model();
    Model(const vector<Point>& p_arr);
    explicit Model(const Model& other);

    virtual ~Model();

    void normalize_n_vrt();
    void add_vertex(const Point& pnt);
    void add_vertices(const vector<Point> &p_arr);
    void add_side(std::initializer_list<size_t> ind_arr, QRgb color);
    void add_side(vector<size_t> ind_arr, QRgb color);

    void correct_n();
    Point &get_center();

    void operator=(const Model& other);

protected:
    Point _center;

private:
    void _add_side(vector<shared_ptr<Vertex>> vertex_arr, QRgb color);
}

```

```

};

Model::Model() {}

Model::Model(const vector<Point>& p_arr)
{
    this->add_vertices(p_arr);

    _center.x = 0;
    _center.y = 0;
    _center.z = 0;

    for (Point pnt : p_arr)
    {
        _center.x += pnt.x;
        _center.y += pnt.y;
        _center.z += pnt.z;
    }

    _center.x /= p_arr.size();
    _center.y /= p_arr.size();
    _center.z /= p_arr.size();
}

Model::Model(const Model& other)
{
    _center = other._center;

    for (auto vertex : other.v_arr)
        this->add_vertex(*vertex);

    for (auto side : other.s_arr)
        this->_add_side(side->vertex_arr, side->color);
}

Model::~Model() {}

void Model::normalize_n_vrt()
{
    for (auto vertex : v_arr)
        vertex->n.normalize();
}

void Model::correct_n()
{
    for (auto vertex : v_arr)
    {
        vertex->n.x = 0;
        vertex->n.y = 0;
        vertex->n.z = 0;
    }
}

```

```

        for (auto side : s_arr)
            side->set_vert_normal(_center);

        normalize_n_vrt();
    }

void Model::_add_side(vector<shared_ptr<Vertex>> vertex_arr,
QRgb color)
{
    shared_ptr<Side> new_side(new Side(vertex_arr, _center,
color));
    s_arr.push_back(new_side);
}

void Model::add_vertex(const Point &pnt)
{
    shared_ptr<Vertex> new_vertex(new Vertex(pnt));
    v_arr.push_back(new_vertex);
}

void Model::add_vertices(const vector<Point> &p_arr)
{
    for (Point point : p_arr)
        add_vertex(point);
}

void Model::add_side(std::initializer_list<size_t> ind_arr, QRgb
color)
{
    vector<shared_ptr<Vertex>> new_side;

    for (size_t i : ind_arr)
    {
        if (i >= v_arr.size())
            throw error::WrongIndex(__FILE__, typeid
(*this).name(), __LINE__ - 1);

        new_side.push_back(v_arr[i]);
    }

    _add_side(new_side, color);
}

void Model::add_side(vector<size_t> ind_arr, QRgb color)
{
    vector<shared_ptr<Vertex>> new_side;

    for (auto i : ind_arr)
    {
        if (i >= v_arr.size())
            throw error::WrongIndex(__FILE__, typeid
(*this).name(), __LINE__ - 1);

```

```

        new_side.push_back(v_arr[i]);
    }

    _add_side(new_side, color);
}

Point& Model::get_center()
{
    return _center;
}

void Model::operator=(const Model &other)
{
    this->_center = other._center;
    this->v_arr = other.v_arr;
    this->s_arr = other.s_arr;
}

```

Листинг А.6 – Класс Коробки

```

class Box : public Model
{
public:
    Box(const Point& pnt1, const Point& pnt2, QRgb color);

    virtual ~Box();

    Point find_min_pnt(const Point& pnt1, const Point& pnt2);
    Point find_max_pnt(const Point& pnt1, const Point& pnt2);
    void add_points_row(const Point& pnt_min, double step, int
num);
    void create_polygons(int num, int i);

private:
    QRgb _color;
    double _step = 30;

    void _find_center(const Point& pnt1, const Point& pnt2);
};

Box::Box(const Point& pnt1, const Point& pnt2, QRgb color) :
    _color(color)
{
    Point pnt_min = find_min_pnt(pnt1, pnt2);
    Point pnt_max = find_max_pnt(pnt1, pnt2);

    _find_center(pnt1, pnt2);

    size_t num_x = static_cast<int>(round((pnt_max.x -
pnt_min.x) / _step));
}

```

```

    size_t num_z = static_cast<int>(round((pnt_max.z -
pnt_min.z) / _step));

    double step_x = (pnt_max.x - pnt_min.x) / num_x;
    double step_z = (pnt_max.z - pnt_min.z) / num_z;

    Point pnt = pnt_min;
    add_points_row(pnt, step_z, num_z);

    for (size_t i = 0; i < num_x; i++)
    {
        pnt.x += step_x;

        add_points_row(pnt, step_z, num_z);
        create_polygons(num_z, i);
    }

    int d = v_arr.size();

    pnt = pnt_min;
    pnt.y = pnt_max.y;
    add_points_row(pnt, step_z, num_z);

    for (size_t i = 0; i < num_x; i++)
    {
        pnt.x += step_x;

        add_points_row(pnt, step_z, num_z);
        create_polygons(num_z, i + num_x + 1);
    }

    for (size_t i = 0; i < num_z; i++)
    {
        size_t j = i + num_x * (num_z + 1);
        add_side({i, i + 1, i + 1 + d, i + d}, _color);
        add_side({j, j + 1, j + 1 + d, j + d}, _color);
    }

    size_t step = num_z + 1;
    for (size_t i = 0; i < num_x * step; i += step)
    {
        size_t j = i + num_z;
        add_side({i, i + step, i + step + d, i + d}, _color);
        add_side({j, j + step, j + step + d, j + d}, _color);
    }

    normalize_n_vrt();
}

Box::~Box() {}

void Box::_find_center(const Point &pnt1, const Point &pnt2)

```

```

{
    _center.x = (pnt1.x + pnt2.x) / 2;
    _center.y = (pnt1.y + pnt2.y) / 2;
    _center.z = (pnt1.z + pnt2.z) / 2;
}

Point Box::find_min_pnt(const Point& pnt1, const Point& pnt2)
{
    Point result;
    result.x = min(pnt1.x, pnt2.x);
    result.y = min(pnt1.y, pnt2.y);
    result.z = min(pnt1.z, pnt2.z);

    return result;
}

Point Box::find_max_pnt(const Point& pnt1, const Point& pnt2)
{
    Point result;
    result.x = max(pnt1.x, pnt2.x);
    result.y = max(pnt1.y, pnt2.y);
    result.z = max(pnt1.z, pnt2.z);

    return result;
}

void Box::add_points_row(const Point& pnt_min, double step, int
num)
{
    Point pnt = pnt_min;
    for (int i = 0; i < num + 1; i++)
    {
        add_vertex(pnt);
        pnt.z += step;
    }
}

void Box::create_polygons(int num, int cur_i)
{
    size_t p1 = cur_i * (num + 1);
    size_t p2 = cur_i * (num + 1) + 1;
    size_t p3 = (cur_i + 1) * (num + 1) + 1;
    size_t p4 = (cur_i + 1) * (num + 1);

    for (int i = 0; i < num; i++)
    {
        add_side({ p1, p2, p3, p4 }, _color);

        p1 += 1;
        p2 += 1;
        p3 += 1;
        p4 += 1;
    }
}

```



```
}
```

Листинг А.7 – Классы Пирамид

```
class Piramid_4 : public Model
{
public:
    Piramid_4(const Point& pnt1, const Point& pnt2, QRgb color,
double down_length, double up_length);

    virtual ~Piramid_4();

private:
    QRgb _color;
};

class Piramid_3 : public Model
{
public:
    Piramid_3(const Point& pnt1, const Point& pnt2, const Point&
pnt3, const Point& pnt_top, QRgb color);

    virtual ~Piramid_3();

private:
    QRgb _color;
};

Piramid_3::Piramid_3(const Point& pnt1, const Point& pnt2, const
Point& pnt3, const Point& pnt_top, QRgb color) :
    _color(color)
{
    _center = Point((pnt1.x + pnt2.x + pnt3.x + pnt_top.x) / 4,
                    (pnt1.y + pnt2.y + pnt3.y + pnt_top.y) / 4,
                    (pnt1.z + pnt2.z + pnt3.z + pnt_top.z) / 4);

    add_vertex(pnt1);
    add_vertex(pnt2);
    add_vertex(pnt3);
    add_vertex(pnt_top);

    add_side({1, 2, 3}, color);
    add_side({2, 0, 3}, color);
    add_side({0, 1, 3}, color);
    add_side({0, 1, 2}, color);

    normalize_n_vrt();
}

Piramid_3::~Piramid_3() {}
```

```

Piramid_4::Piramid_4(const Point& pnt1, const Point& pnt2, QRgb
color, double down_length, double up_length) :
    _color(color)
{
    _center = Point(pnt2.x, (pnt1.y + pnt2.y) / 2 , pnt2.z);

    add_vertex(pnt1);
    add_vertex(Point(pnt1.x, pnt1.y, pnt1.z - down_length));
    add_vertex(Point(pnt1.x + down_length, pnt1.y, pnt1.z -
down_length));
    add_vertex(Point(pnt1.x + down_length, pnt1.y, pnt1.z));

    add_vertex(pnt2);
    add_vertex(Point(pnt2.x, pnt2.y, pnt2.z - up_length));
    add_vertex(Point(pnt2.x + up_length, pnt2.y, pnt2.z -
up_length));
    add_vertex(Point(pnt2.x + up_length, pnt2.y, pnt2.z));

    add_side({1, 5, 6, 2}, color);
    add_side({0, 4, 5, 1}, color);
    add_side({2, 6, 7, 3}, color);
    add_side({0, 4, 7, 3}, color);

    normalize_n_vrt();
}

Piramid_4::~Piramid_4() {}

```

Листинг А.8 – Классы Поверхностей

```

class Surface : public Model
{
public:
    Surface(QRgb color);

    virtual ~Surface();

    Point find_min_pnt(const Point& pnt1, const Point& pnt2);
    Point find_max_pnt(const Point& pnt1, const Point& pnt2);
    void find_center(const Point& pnt1, const Point& pnt2);

protected:
    QRgb _color;
};

class SurfaceUp : public Surface
{
public:
    SurfaceUp(QRgb color, const Point& pnt1, const Point& pnt2,
const Point& pnt3);

```

```

    virtual ~SurfaceUp();

    void add_carcas(const Point& pnt1, const Point& pnt2);
    void add_points_row(const Point& pnt_min, double step, int
num);
    void create_polygons(int num, int i);
    void find_center(const Point& pnt1, const Point& pnt2);
    void correct_vert(size_t ind);

private:
    double _step = 30;
    size_t _num_nodes;
    double _down_length = 140;
    double _up_length = 2;
    double _height;
    double _cur_length = 140;
};

class SurfaceDown : public Surface
{
public:
    SurfaceDown(QRgb color, const Point& pnt1, const Point&
pnt2);

    virtual ~SurfaceDown();

    void add_points_row(const Point& pnt_min, double step, int
num);
    void create_polygons(int num, int i);
    void find_center(const Point& pnt);
    void correct_vert(size_t ind);

private:
    double _step = 20;
    size_t _num_nodes;
    double _down_length = 140;
    double _cur_length = 140;
};

Surface::Surface(QRgb color) : _color(color) {}

Surface::~Surface() {}

Point Surface::find_min_pnt(const Point& pnt1, const Point&
pnt2)
{
    Point result;

    result.x = min(pnt1.x, pnt2.x);
    result.y = min(pnt1.y, pnt2.y);

```

```

        result.z = min(pnt1.z, pnt2.z);

        return result;
    }

    Point Surface::find_max_pnt(const Point& pnt1, const Point&
pnt2)
    {
        Point result;

        result.x = max(pnt1.x, pnt2.x);
        result.y = max(pnt1.y, pnt2.y);
        result.z = max(pnt1.z, pnt2.z);

        return result;
    }

    void Surface::find_center(const Point&, const Point&) {}

    SurfaceUp::SurfaceUp(QRgb color, const Point& pnt1, const Point&
pnt2, const Point& pnt3) : Surface(color)
    {
        Point pnt_min = find_min_pnt(pnt1, pnt2);
        Point pnt_max = find_max_pnt(pnt1, pnt2);
        find_center(pnt3, pnt_max);

        _height = abs(pnt1.y - pnt3.y);

        size_t num_x = static_cast<int>(round((pnt_max.x -
pnt_min.x) / _step));
        size_t num_z = static_cast<int>(round((pnt_max.z -
pnt_min.z) / _step));

        if ((num_x + 1) % 2) num_x += 1;
        if ((num_z + 1) % 2) num_z += 1;

        double step_x = (pnt_max.x - pnt_min.x) / num_x;
        double step_z = (pnt_max.z - pnt_min.z) / num_z;

        Point pnt = pnt_min;
        add_points_row(pnt, step_z, num_z);

        for (size_t i = 0; i < num_x; i++)
        {
            pnt.x += step_x;

            add_points_row(pnt, step_z, num_z);
            create_polygons(num_z, i);
        }
        _num_nodes = v_arr.size();
    }

```

```

add_carcas(pnt1, pnt3);

if ((num_z + 1) % 2)
{
    for (size_t i = 1; i * 2 < (num_z + 2); i++)
        add_side({i - 1, i, _num_nodes}, _color);
    for (size_t i = (num_z + 2) / 2; i < num_z + 1; i++)
        add_side({i - 1, i, _num_nodes + 1}, _color);
    add_side({_num_nodes, num_z / 2, _num_nodes + 1},
_color);

    size_t d = num_x * (num_z + 1);

    for (size_t i = 1; i * 2 < (num_z + 2); i++)
        add_side({i - 1 + d, i + d, _num_nodes + 3},
_color);
    for (size_t i = (num_z + 2) / 2; i < num_z + 1; i++)
        add_side({i - 1 + d, i + d, _num_nodes + 2},
_color);
    add_side({_num_nodes + 3, num_z / 2 + d, _num_nodes +
2}, _color);
}
else
{
    for (size_t i = 1; i * 2 < (num_z + 1); i++)
        add_side({i - 1, i, _num_nodes}, _color);
    for (size_t i = (num_z + 1) / 2 + 1; i < num_z + 1; i++)
        add_side({i - 1, i, _num_nodes + 1}, _color);
    add_side(({num_z + 1) / 2 - 1, (num_z + 1) / 2,
_num_nodes + 1, _num_nodes}, _color);

    size_t d = num_x * (num_z + 1);

    for (size_t i = 1; i * 2 < (num_z + 1); i++)
        add_side({i - 1 + d, i + d, _num_nodes + 3},
_color);
    for (size_t i = (num_z + 1) / 2 + 1; i < num_z + 1; i++)
        add_side({i - 1 + d, i + d, _num_nodes + 2},
_color);
    add_side(({num_z + 1) / 2 - 1 + d, (num_z + 1) / 2 + d,
_num_nodes + 2, _num_nodes + 3}, _color);
}

_num_nodes += 4;
if ((num_x + 1) % 2)
{
    size_t n = (num_x + 1) * (num_z + 1);
    size_t step = num_z + 1;

    for (size_t i = step; i * 2 < n + step; i += step)
        add_side({i - step, i, _num_nodes}, _color);
    for (size_t i = (n + step) / 2; i < n; i += step)
        add_side({i - step, i, _num_nodes + 3}, _color);
}

```

```

        add_side({_num_nodes, (n - step) / 2, _num_nodes + 3},
_color);

        size_t d = num_z;

        for (size_t i = step; i * 2 < n + step; i += step)
            add_side({i - step + d, i + d, _num_nodes + 1},
_color);
        for (size_t i = (n + step) / 2; i < n; i += step)
            add_side({i - step + d, i + d, _num_nodes + 2},
_color);
        add_side({_num_nodes + 1, (n - step) / 2 + d, _num_nodes
+ 2}, _color);
    }
    else
    {
        size_t n = (num_x + 1) * (num_z + 1);
        size_t step = num_z + 1;

        for (size_t i = step; i * 2 < n; i += step)
            add_side({i - step, i, _num_nodes}, _color);
        for (size_t i = n / 2 + step; i < n; i += step)
            add_side({i - step, i, _num_nodes + 3}, _color);
        add_side({n / 2 - step, n / 2, _num_nodes + 3,
_num_nodes}, _color);

        size_t d = num_z;

        for (size_t i = step; i * 2 < n; i += step)
            add_side({i - step + d, i + d, _num_nodes + 1},
_color);
        for (size_t i = n / 2 + step; i < n; i += step)
            add_side({i - step + d, i + d, _num_nodes + 2},
_color);
        add_side({n / 2 - step + d, n / 2 + d, _num_nodes + 2,
_num_nodes + 1}, _color);
    }
    normalize_n_vrt();
}
void SurfaceUp::correct_vert(size_t ind)
{
    v_arr[ind]->n.y = 0;
    double length = v_arr[ind]->n.get_length();
    v_arr[ind]->n.x /= length;
    v_arr[ind]->n.z /= length;
}

void SurfaceUp::add_carcas(const Point &, const Point &pnt2)
{
    add_vertex(Point(pnt2.x, pnt2.y, pnt2.z - _up_length));
    add_vertex(Point(pnt2.x + _up_length, pnt2.y, pnt2.z -
_up_length));
    add_vertex(Point(pnt2.x + _up_length, pnt2.y, pnt2.z));
}

```

```

        add_vertex(pnt2);

        add_vertex(Point(pnt2.x, pnt2.y, pnt2.z - _up_length));
        add_vertex(Point(pnt2.x + _up_length, pnt2.y, pnt2.z -
_up_length));
        add_vertex(Point(pnt2.x + _up_length, pnt2.y, pnt2.z));
        add_vertex(pnt2);
    }

SurfaceUp::~SurfaceUp() {}

void SurfaceUp::add_points_row(const Point& pnt_min, double
step, int num)
{
    Point pnt = pnt_min;

    for (int i = 0; i < num + 1; i++)
    {
        add_vertex(pnt);
        pnt.z += step;
    }
}

void SurfaceUp::create_polygons(int num, int cur_i)
{
    size_t p1 = cur_i * (num + 1);
    size_t p2 = cur_i * (num + 1) + 1;
    size_t p3 = (cur_i + 1) * (num + 1) + 1;
    size_t p4 = (cur_i + 1) * (num + 1);

    for (int i = 0; i < num; i++)
    {
        add_side({ p1, p2, p3, p4 }, _color);

        p1 += 1;
        p2 += 1;
        p3 += 1;
        p4 += 1;
    }
}

void SurfaceUp::find_center(const Point &pnt1, const Point
&pnt2)
{
    _center.x = 0;
    _center.y = (pnt1.y + pnt2.y) / 2;
    _center.z = 0;
}

```

```

SurfaceDown::SurfaceDown(QRgb color, const Point& pnt1, const
Point& pnt2) : Surface(color)
{
    Point pnt_min = find_min_pnt(pnt1, pnt2);
    Point pnt_max = find_max_pnt(pnt1, pnt2);
    find_center(pnt1);

    size_t num_x = static_cast<int>(round((pnt_max.x -
pnt_min.x) / _step));
    size_t num_z = static_cast<int>(round((pnt_max.z -
pnt_min.z) / _step));

    double step_x = (pnt_max.x - pnt_min.x) / num_x;
    double step_z = (pnt_max.z - pnt_min.z) / num_z;

    Point pnt = pnt_min;
    add_points_row(pnt, step_z, num_z);

    for (size_t i = 0; i < num_x; i++)
    {
        pnt.x += step_x;

        add_points_row(pnt, step_z, num_z);
        create_polygons(num_z, i);
    }
    _num_nodes = v_arr.size();

    normalize_n_vrt();
}

void SurfaceDown::correct_vrt(size_t ind)
{
    v_arr[ind]->n.y = 0;
    double length = v_arr[ind]->n.get_length();
    v_arr[ind]->n.x /= length;
    v_arr[ind]->n.z /= length;
}

SurfaceDown::~SurfaceDown() {}

void SurfaceDown::add_points_row(const Point& pnt_min, double
step, int num)
{
    Point pnt = pnt_min;

    for (int i = 0; i < num + 1; i++)
    {
        add_vertex(pnt);
        pnt.z += step;
    }
}

void SurfaceDown::create_polygons(int num, int cur_i)

```



```

{
    size_t p1 = cur_i * (num + 1);
    size_t p2 = cur_i * (num + 1) + 1;
    size_t p3 = (cur_i + 1) * (num + 1) + 1;
    size_t p4 = (cur_i + 1) * (num + 1);

    for (int i = 0; i < num; i++)
    {
        add_side({ p1, p2, p3, p4 }, _color);

        p1 += 1;
        p2 += 1;
        p3 += 1;
        p4 += 1;
    }
}

void SurfaceDown::find_center(const Point &pnt)
{
    _center.x = 0;
    _center.y = pnt.y - 30;
    _center.z = 0;
}

```

Приложение Б

Приведены листинги классов некоторых объектов сцены.

Листинг Б.1 – Класс Камеры

```
class Camera : public InvisibleObject
{
public:
    Camera();
    Camera(const Point& position);
    Camera(const Point& position, const Vector& direction);
    explicit Camera(const Camera& other);
    virtual ~Camera();

    Point& get_position();
    Vector& get_direction();

    const Point& get_position() const;
    const Vector& get_direction() const;

    void operator=(const Camera& other);

    virtual void accept(ObjectVisitor &visitor);
    virtual SceneObject* clone();

private:
    Point _pos;
    Vector _dir;
};

Camera::Camera() : _pos(Point(10, 10, 100)) {}

Camera::Camera(const Point& position) : _pos(position) {}

Camera::Camera(const Point& position, const Vector& direction) :
    _pos(position), _dir(direction) {}

Camera::Camera(const Camera& other) : _pos(other._pos),
    _dir(other._dir) {}

Camera::~~Camera() {}

Point& Camera::get_position() { return _pos; }

Vector& Camera::get_direction() { return _dir; }

const Point& Camera::get_position() const { return _pos; }

const Vector& Camera::get_direction() const { return _dir; }
```

```

void Camera::operator=(const Camera &other)
{
    this->_pos = other._pos;
    this->_dir = other._dir;
}

void Camera::accept(ObjectVisitor& visitor)
{
    visitor.visit(*this);
}

SceneObject *Camera::clone()
{
    return (new Camera(*this));
}

```

Листинг Б.2 – Класс Источника освещения

```

class LightSource : public InvisibleObject
{
public:
    LightSource();
    LightSource(const Point& position);
    LightSource(const Point& position, double intensity);
    explicit LightSource(const LightSource& other);

    virtual ~LightSource();

    Point& get_position();
    double get_intensity();
    void set_intensity(double intensity);

    const Point& get_position() const;

    void operator =(const LightSource& other);

    virtual void accept(ObjectVisitor&);
    virtual SceneObject* clone();

private:
    Point _pos;
    double _i;
};

class LightSource : public InvisibleObject
{
public:
    LightSource();
    LightSource(const Point& position);

```

```

LightSource(const Point& position, double intensity);
explicit LightSource(const LightSource& other);

virtual ~LightSource();

Point& get_position();
double get_intensity();
void set_intensity(double intensity);

const Point& get_position() const;

void operator =(const LightSource& other);

virtual void accept(ObjectVisitor&);
virtual SceneObject* clone();

private:
    Point _pos;
    double _i;
};

```

Листинг Б.3 – Класс Стекла

```

class Glass : public VisibleObject
{
public:
    Glass(const Point& pnt1, const Point& pnt2);
    explicit Glass(const Glass& other);

    virtual ~Glass();

    double get_transparency();

    virtual void accept(ObjectVisitor& visitor);
    virtual SceneObject* clone();

private:
    QRgb _color;
    double _tr = 0.3;
    double _down_length = 240;
    double _up_length = 10;
};

Glass::Glass(const Point& pnt1, const Point& pnt2)
{
    _color = QColor(Qt::green).rgba();

    Model* model_ptr = new Piramid(pnt1, pnt2, _color,
    _down_length, _up_length);
    _model = shared_ptr<Model>(model_ptr);
}

```

```

Glass::Glass(const Glass& other) : VisibleObject(other)
{
    _color = other._color;
    _tr = other._tr;
}

Glass::~Glass() {}

double Glass::get_transparency() { return _tr; }

void Glass::accept(ObjectVisitor &visitor) {
    visitor.visit(*this); }

SceneObject* Glass::clone() { return new Glass(*this); }

```

Листинг Б.4 – Классы составляющих песка

```

class SandUpP : public VisibleObject
{
public:
    SandUpP(const Point& pnt1, const Point& pnt2);
    explicit SandUpP(const SandUpP& other);

    virtual ~SandUpP();

    virtual void accept(ObjectVisitor& visitor);
    virtual SceneObject* clone();

private:
    QRgb _color;
    double _down_length = 140;
    double _up_length = 2;
};

class SandItem : public VisibleObject
{
public:
    double v_y = 0;

    SandItem(const Point& pnt1, const Point& pnt2, const Point&
pnt3, const Point& pnt_top);
    SandItem(const Point& pnt_top);
    explicit SandItem(const SandItem& other);

    virtual ~SandItem();

    virtual void accept(ObjectVisitor& visitor);
    virtual SceneObject* clone();

```

```

private:
    QRgb _color;

    double _h = 4;
};

SandUpP::SandUpP(const Point& pnt1, const Point& pnt2)
{
    _color = QColor("#b0894f").rgba();

    Model* model_ptr = new Pyramid_4(pnt1, pnt2, _color,
    _down_length, _up_length);
    _model = shared_ptr<Model>(model_ptr);
}

SandUpP::SandUpP(const SandUpP& other) : VisibleObject(other)
{
    _color = other._color;
}

SandUpP::~SandUpP() {}

void SandUpP::accept(ObjectVisitor &visitor) {
    visitor.visit(*this); }

SceneObject* SandUpP::clone() { return new SandUpP(*this); }

SandItem::SandItem(const Point& pnt1, const Point& pnt2, const
Point& pnt3, const Point& pnt_top)
{
    _color = QColor("#b0894f").rgba();

    Model* model_ptr = new Pyramid_3(pnt1, pnt2, pnt3, pnt_top,
    _color);
    _model = shared_ptr<Model>(model_ptr);
}

SandItem::SandItem(const Point& pnt_top)
{
    _color = QColor("#b0894f").rgba();

    Point pnt1(pnt_top.x - _h * 1.73 / 4, pnt_top.y - _h,
    pnt_top.z + _h / 2);
    Point pnt2(pnt_top.x + _h * 1.73 / 4, pnt_top.y - _h,
    pnt_top.z + _h / 2);
    Point pnt3(pnt_top.x, pnt_top.y - _h, pnt_top.z - _h);

    Model* model_ptr = new Pyramid_3(pnt1, pnt2, pnt3, pnt_top,
    _color);
    _model = shared_ptr<Model>(model_ptr);
}

```

```

SandItem::SandItem(const SandItem& other) : VisibleObject(other)
{
    _color = other._color;
}

SandItem::~SandItem() {}

void SandItem::accept(ObjectVisitor &visitor) {
    visitor.visit(*this); }

SceneObject* SandItem::clone() { return new SandItem(*this); }

```

Приложение В

Приведены листинги классов некоторых трансформаций объектов сцены.

Листинг В.1 – Классы трансформаций

```
class Transformation
{
public:
    Transformation();
    virtual ~Transformation() = 0;

    virtual void rotate(const Vector& vect) = 0;

    virtual void execute(double& x, double& y, double& z) = 0;
    virtual void execute(Point& pnt) = 0;
    virtual void execute(Vertex& vertex) = 0;
    virtual void execute(Vector& vect) = 0;
    virtual void execute(Camera& camera) = 0;
};

class Move : public Transformation
{
public:
    Move(double dx, double dy, double dz);
    Move(const Vector& vect);
    virtual ~Move();

    virtual void rotate(const Vector& vect);

    virtual void execute(double& x, double& y, double& z);
    virtual void execute(Point& pnt);
    virtual void execute(Vertex& vertex);
    virtual void execute(Vector& vect);
    virtual void execute(Camera& camera);

private:
    Vector _dir;
};

class Rotate : public Transformation
{
public:
    Rotate(const Vector& vect, const Point& pnt);
    Rotate(const Vector& vect);
    virtual ~Rotate();

    virtual void rotate(const Vector& vect);

    virtual void execute(double& x, double& y, double& z);
    virtual void execute(Point& pnt);
    virtual void execute(Vertex& vertex);
```



```

    virtual void execute(Vector& vect);
    virtual void execute(Camera& camera);

    void to_radians();

private:
    Vector _dir;
    Point _center;

    double _to_radians(double angle);

    void rotate_x(double& x, double& y, double& z);
    void rotate_y(double& x, double& y, double& z);
    void rotate_z(double& x, double& y, double& z);

    void rotate_ox(double& x, double& y, double& z);
    void rotate_oy(double& x, double& y, double& z);
    void rotate_oz(double& x, double& y, double& z);
};

class Scale : public Transformation
{
public:
    Scale(const Vector& vect);
    Scale(const Vector& vect, const Point& pnt);
    virtual ~Scale();

    virtual void rotate(const Vector& vect);

    virtual void execute(double& x, double& y, double& z);
    virtual void execute(Point& pnt);
    virtual void execute(Vertex& vertex);
    virtual void execute(Vector& vect);
    virtual void execute(Camera& camera);

private:
    Vector _dir;
    Point _center;
};

Transformation::Transformation() {}
Transformation::~Transformation() {}

Move::Move(double dx, double dy, double dz) : _dir(dx, dy, dz)
{}
Move::Move(const Vector& vect) : _dir(vect) {}
Move::~Move() {}

void Move::rotate(const Vector &vect)
{
    Vector v_try{-vect.x, -vect.y, vect.z};
    Rotate action(v_try);

```

```

        action.execute(_dir);
    }

void Move::execute(double &x, double &y, double &z)
{
    x += _dir.x;
    y += _dir.y;
    z += _dir.z;
}

void Move::execute(Point &pnt)
{
    pnt.x += _dir.x;
    pnt.y += _dir.y;
    pnt.z += _dir.z;
}

void Move::execute(Vertex &vertex)
{
    vertex.x += _dir.x;
    vertex.y += _dir.y;
    vertex.z += _dir.z;
}

void Move::execute(Vector&) {}

void Move::execute(Camera &camera) {
    execute(camera.get_position()); }

Rotate::Rotate(const Vector& vect, const Point& pnt) :
    _center(pnt)
{
    _dir.x = vect.x;
    _dir.y = vect.y;
    _dir.z = vect.z;
}

Rotate::Rotate(const Vector& vect) : _center()
{
    _dir.x = vect.x;
    _dir.y = vect.y;
    _dir.z = vect.z;
}

Rotate::~Rotate() {}

void Rotate::to_radians()
{
    _dir.x = _to_radians(_dir.x);
    _dir.y = _to_radians(_dir.y);
    _dir.z = _to_radians(_dir.z);
}

```

```

double Rotate::_to_radians(double angle){ return angle * PI /
180; }

void Rotate::rotate(const Vector&) {}

void Rotate::execute(double &x, double &y, double &z)
{
    rotate_x(x, y, z);
    rotate_y(x, y, z);
    rotate_z(x, y, z);
}

void Rotate::execute(Point &pnt)
{
    rotate_x(pnt.x, pnt.y, pnt.z);
    rotate_y(pnt.x, pnt.y, pnt.z);
    rotate_z(pnt.x, pnt.y, pnt.z);
}

void Rotate::execute(Vertex &vertex)
{
    rotate_x(vertex.x, vertex.y, vertex.z);
    rotate_y(vertex.x, vertex.y, vertex.z);
    rotate_z(vertex.x, vertex.y, vertex.z);
}

void Rotate::execute(Vector& vect)
{
    rotate_ox(vect.x, vect.y, vect.z);
    rotate_oy(vect.x, vect.y, vect.z);
    rotate_oz(vect.x, vect.y, vect.z);
}

void Rotate::execute(Camera &camera)
{
    Vector& vect = camera.get_direction();
    vect.x += _dir.x;
    vect.y += _dir.y;
    vect.z += _dir.z;
}

void Rotate::rotate_x(double&, double &y, double &z)
{
    double y_temp, z_temp;

    y_temp = _center.y + (y - _center.y) * cos(_dir.x) + (z -
_center.z) * sin(_dir.x);
    z_temp = _center.z + (z - _center.z) * cos(_dir.x) - (y -
_center.y) * sin(_dir.x);

    y = y_temp;
    z = z_temp;
}

```

```

}

void Rotate::rotate_y(double &x, double&, double &z)
{
    double x_temp, z_temp;

    x_temp = _center.x+(x-_center.x)*cos(_dir.y)-(z-_center.z)*sin(_dir.y);
    z_temp = _center.z+(z-_center.z)*cos(_dir.y)+(x-_center.x)*sin(_dir.y);

    x = x_temp;
    z = z_temp;
}

void Rotate::rotate_z(double &x, double &y, double&)
{
    double x_temp, y_temp;

    x_temp = _center.x+(x-_center.x)*cos(_dir.z)+(y-_center.y)*sin(_dir.z);
    y_temp = _center.y+(y-_center.y)*cos(_dir.z)-(x-_center.x)*sin(_dir.z);

    x = x_temp;
    y = y_temp;
}

void Rotate::rotate_ox(double&, double& y, double& z)
{
    double y_temp, z_temp;
    y_temp = y*cos(_dir.x) + z*sin(_dir.x);
    z_temp = -y*sin(_dir.x) + z*cos(_dir.x);

    y = y_temp;
    z = z_temp;
}

void Rotate::rotate_oy(double& x, double&, double& z)
{
    double x_temp, z_temp;
    x_temp = x*cos(_dir.y) - z*sin(_dir.y);
    z_temp = x*sin(_dir.y) + z*cos(_dir.y);

    x = x_temp;
    z = z_temp;
}

void Rotate::rotate_oz(double& x, double& y, double&)
{
    double y_temp, x_temp;
    x_temp = x*cos(_dir.z) + y*sin(_dir.z);
    y_temp = -x*sin(_dir.z) + y*cos(_dir.z);

    x = x_temp;

```

```

        y = y_temp;
    }

Scale::Scale(const Vector& vect, const Point&
pnt):_dir(vect),_center(pnt) {}
Scale::Scale(const Vector& vect) : _dir(vect), _center() {}
Scale::~Scale() {}

void Scale::rotate(const Vector&) {}

void Scale::execute(double &x, double &y, double &z)
{
    x = _dir.x * (x - _center.x) + _center.x;
    y = _dir.y * (y - _center.y) + _center.y;
    z = _dir.z * (z - _center.z) + _center.z;
}

void Scale::execute(Point &pnt)
{
    pnt.x = _dir.x * (pnt.x - _center.x) + _center.x;
    pnt.y = _dir.y * (pnt.y - _center.y) + _center.y;
    pnt.z = _dir.z * (pnt.z - _center.z) + _center.z;
}

void Scale::execute(Vertex &vertex)
{
    vertex.x = _dir.x * (vertex.x - _center.x) + _center.x;
    vertex.y = _dir.y * (vertex.y - _center.y) + _center.y;
    vertex.z = _dir.z * (vertex.z - _center.z) + _center.z;
}

void Scale::execute(Vector &vect)
{
    vect.x *= _dir.x;
    vect.y *= _dir.y;
    vect.z *= _dir.z;
}

void Scale::execute(Camera &camera)
{
    execute(camera.get_position());
}

```