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:

Реализация алгоритма иерархического z-буфера (с использованием октодерева) удаления невидимых поверхностей

Студент _____ИУ7-54Б_____ _____ ____**Д.В. Жабин**_____
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсового проекта _____ ___**Д.А. Погорелов**___
(Подпись, дата) (И.О.Фамилия)

2021 г.

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)
« _____ » _____ 20 ____ г.

З А Д А Н И Е
на выполнение курсового проекта

по дисциплине Компьютерная графика
Студент группы ИУ7-54Б

Жабин Дмитрий Владимирович
(Фамилия, имя, отчество)

Тема курсового проекта Реализация алгоритма иерархического z-буфера (с использованием октодерева) удаления невидимых поверхностей

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

учебный

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 программное обеспечение, которое предоставляет возможность загрузки моделей объемных тел из перечня файлов и визуализирует их с использованием выбранной модификации алгоритма z-буфера (простой алгоритм z-буфера, алгоритм z-буфера со сканирующей строкой, иерархический алгоритм z-буфера, иерархический алгоритм z-буфера с октодеревом). Должна быть предоставлена возможность вращать и приближать/отдалять модель, обеспечено освещение сцены зафиксированным источником света.

Оформление курсового проекта:
Расчетно-пояснительная записка на _25-30_ листах формата А4.
Расчетно-пояснительная записка должна содержать постановку задачи, введение, аналитическую часть, конструкторскую часть, технологическую часть, экспериментально-исследовательский раздел, заключение, список литературы, приложения.
Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.)
На защиту проекта должна быть предоставлена презентация, состоящая из 15-20 слайдов. На слайдах должны быть отражены: постановка задачи, использованные методы и алгоритмы, расчетные соотношения, структура комплекса программ, интерфейс, результаты проведенных исследований.

Дата выдачи задания « ___ » _____ 20__ г.

Руководитель курсового проекта _____ Д. А. Погорелов
(Подпись, дата) (И.О.Фамилия)

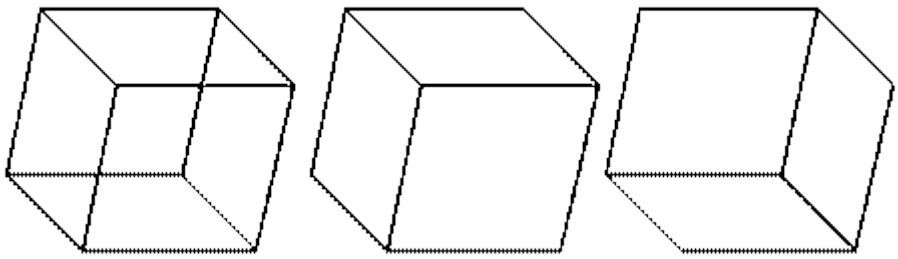
Студент _____ Д.В. Жабин
(Подпись, дата) (И.О.Фамилия)

Оглавление

Введение

Одной из основных задач компьютерной графики является визуализация трёхмерных сцен. Подобные задачи возникают в системах автоматизированного проектирования, пакетах моделирования физических процессов, средствах компьютерной анимации и виртуальной реальности. При отображении трёхмерной сцены на экране некоторые из объектов сцены могут заслонить другие объекты. Заслонённые части объектов невидимы и не должны рисоваться, или должны рисоваться иначе, чем видимые части, например, пунктиром. Если этого не делать, то изображение будет выглядеть неправильно. Такие задачи решают с помощью алгоритмов удаления невидимых линий и поверхностей. Невидимые линии удаляются при отображении сцены в каркасном виде (алгоритм выделяет части ребер объекта, которые заслонены и удаляет их). При отображении объектов с помощью закрашенных поверхностей удаляются их невидимые части.

Необходимость удаления невидимых линий, ребер, поверхностей или объемов проиллюстрирована на рис.1.



а б в

Рис.1. Необходимость удаления невидимых линий

На рис. 1.а приведен типичный каркасный чертеж куба. Каркасный чертеж представляет трехмерный объект в виде штрихового изображения его ребер. Рис. 1.а можно интерпретировать двояко: как вид куба сверху слева или снизу справа.

Удаление тех линий или поверхностей, которые невидимы с соответствующей точки зрения, позволяют избавиться от неоднозначности. Результаты показаны на рис. 1.б и 1.в.

Цель данной работы – реализовать алгоритм иерархического z-буфера (с использованием октодерева) удаления невидимых поверхностей и произвести анализ его эффективности по сравнению с более простыми модификациями алгоритма z-буфера.

Для достижения поставленной цели требуется решить следующие задачи:

- формализовать объекты сцены;
- изучить модификации алгоритма z-буфера удаления невидимых поверхностей;
- изучить существующие модели освещения сцены и выбрать наиболее подходящую для данной задачи;
- реализовать алгоритмы удаления невидимых поверхностей с использованием z-буфера;
- провести сравнение временных характеристик реализаций алгоритмов.

1. Аналитическая часть

В данном разделе проведены формализация объектов сцены, анализ модификаций алгоритма z-буфера удаления невидимых поверхностей и анализ существующих моделей освещения сцены.

1.1 Формализация объектов синтезируемой сцены

Сцена состоит из:

- Источника света – представляет собой материальную точку, из которой исходят лучи света во все стороны (в частном случае, когда источник расположен в бесконечности, имеет направленность).

  Положение источника зафиксировано и не меняется в ходе выполнения программы.

- Фигуры – модели объемного тела, добавленной в сцену пользователем (описывается в файле)

Фигуры наилучшим образом описываются через поверхностные модели.

Поверхностную модель можно задать несколькими способами:

Параметрическим представлением – для получения поверхности нужно вычислять функцию, зависящую от параметра. В сцене нет поверхностей вращения, поэтому использование параметрического представления будет затруднительно и нецелесообразно.

Полигональной сеткой – совокупностью вершин, ребер и граней, которые определяют форму объекта.

- **Вершинное представление** описывает объект как множество вершин, соединённых с другими вершинами. Это простейшее представление, но оно не широко используемое, так как информация о

гранях и рёбрах не выражена явно. Нужно обойти все данные, чтобы сгенерировать список граней для рендеринга. Кроме того, нелегко выполняются операции на рёбрах и гранях.

- **Список граней** представляет объект как множество граней и множество вершин. В отличие от вершинного представления, и грани и вершины явно представлены, так что нахождение соседних граней и вершин постоянно по времени. Моделирование требует лёгкого обхода всех структур. Однако, рёбра не заданы явно, так что поиск всё ещё нужен, чтобы найти все грани, окружающие заданную грань. Другие динамические операции, такие как разрыв или объединение грани, также сложны со списком граней.
- **«Крылатое» представление**: в нём каждая точка ребра указывает на две вершины, две грани и четыре (по часовой стрелке и против часовой) ребра, которые её касаются. Крылатое представление позволяет обойти поверхность за постоянное время, но у него большие требования по памяти хранения. Хорошо подходит для динамической геометрии.

В качестве способа хранения объектов сцены в данной работе было выбрано вершинное представление (список вершин и индексов), так как оно хорошо подходит под формат входных файлов, содержащих описание моделей.

1.2 Анализ алгоритмов удаления невидимых линий и поверхностей

Алгоритмы удаления невидимых линий или поверхностей можно классифицировать по способу выбора системы координат или пространства, в котором они работают. Выделяют три класса алгоритмов удаления невидимых линий или поверхностей:

- алгоритмы, работающие в объектном пространстве;

- алгоритмы, работающие в пространстве изображения (экрана);
- алгоритмы, формирующие список приоритетов.

Алгоритмы, работающие в объектном пространстве, имеют дело с физической системой координат, в которой описаны эти объекты. При этом получаются весьма точные результаты, ограниченные лишь точностью вычислений. Полученные изображения можно свободно увеличивать во много раз. Алгоритмы, работающие в объектном пространстве, особенно полезны в тех приложениях, где необходима высокая точность.

Алгоритмы же, работающие в пространстве изображения, имеют дело с системой координат того экрана, на котором объекты визуализируются. При этом точность вычислений ограничена разрешающей способностью экрана. Обычно разрешение экрана бывает довольно низким. Результаты, полученные в пространстве изображения, а затем увеличенные во много раз, не будут соответствовать исходной сцене. Например, могут не совпасть концы отрезков.

Алгоритмы, формирующие список приоритетов, работают попеременно в обеих упомянутых системах координат.

Объем вычислений для любого алгоритма, работающего в объектном пространстве и сравнивающего каждый объект сцены со всеми остальными объектами этой сцены, растет теоретически, как квадрат числа объектов (n^2). Аналогично, объем вычислений любого алгоритма, работающего в пространстве изображения и сравнивающего каждый объект сцены с позициями всех пикселов в системе координат экрана, растет теоретически, как nN. Здесь n обозначает количество объектов (тел, плоскостей или ребер) в сцене, а N — число пикселов. Теоретически трудоемкость алгоритмов, работающих в объектном пространстве, меньше трудоемкости алгоритмов, работающих в пространстве изображения, при $n < N$. Однако на практике это не так. Дело в том, что алгоритмы, работающие в пространстве изображения, более эффективны потому, что для них легче воспользоваться преимуществом когерентности при растровой реализации.

1.3 Модификации алгоритма z-буфера удаления невидимых поверхностей

Алгоритм, использующий z–буфер (простой)

Идея *z*-буфера является простым обобщением идеи о буфере кадра. Буфер кадра используется для запоминания атрибутов (интенсивности) каждого пиксела в пространстве изображения, z-буфер - это отдельный буфер глубины, используемый для запоминания координаты *z* или глубины каждого видимого пиксела в пространстве изображения. В процессе работы глубина или значение *z* каждого нового пиксела, который нужно занести в буфер кадра, сравнивается с глубиной того пиксела, который уже занесен в *z*-буфер. Если это сравнение показывает, что новый пиксел расположен впереди пиксела, находящегося в буфере кадра, то новый пиксел заносится в этот буфер и, кроме того, производится корректировка *z*-буфера новым значением *z*. Если же сравнение дает противоположный результат, то никаких действий не производится. По сути, алгоритм является поиском по *x* и *y* наибольшего значения функции *z (x, y).*

Главное преимущество алгоритма – его простота. Кроме того, этот алгоритм решает задачу об удалении невидимых поверхностей и делает тривиальной визуализацию пересечений сложных поверхностей. Сцены могут быть любой сложности. Поскольку габариты пространства изображения фиксированы, оценка вычислительной трудоемкости алгоритма не более чем линейна. Поскольку элементы сцены или картинки можно заносить в буфер кадра или в *z*-буфер в произвольном порядке, их не нужно предварительно сортировать по приоритету глубины. Поэтому экономится вычислительное время, затрачиваемое на сортировку по глубине.

Формальное описание алгоритма z-буфера таково:

1. Заполнить буфер кадра фоновым значением интенсивности или цвета.

2. Заполнить *z*-буфер минимальным значением *z*.

3. Преобразовать каждый многоугольник в растровую форму в произвольном порядке.

4. Для каждого *Пиксел*(*x*,*y*) в многоугольнике вычислить его глубину *z*(*x*,*y*).

5. Сравнить глубину *z*(*x,y*) со значением *Zбуфер*(*x,y*), хранящимся в *z*-буфере в этой же позиции.

6. Если *z*(*x,y*) > *Zбуфер* (*x,y*), то записать атрибут этого многоугольника (интенсивность, цвет и т. п.) в буфер кадра и заменить *Zбуфер*(*x,y*) на *z*(*x,y*). В противном случае никаких действий не производить.

Алгоритм, использующий z–буфер со сканирующей строкой

Основная проблема компьютерной графики заключается в необходимости работать в трехмерном пространстве. Идея алгоритма построчного сканирования состоит в том, что мы рассматриваем плоскость сечения, проходящую через строку экрана перпендикулярно ему. В сечении этой плоскости оказываются некоторые объекты. После чего мы работаем только с этой двумерной плоскостью. Эта идея определяет сферу применимости метода. Он имеет эффективную реализацию для таких объектов, для которых можно легко находить пересечение с плоскостью.

Используемое в этом алгоритме окно визуализации имеет высоту в одну сканирующую строку и ширину во весь экран. Как для буфера кадра, так для z-буфера требуется память высотой в 1 бит, шириной в горизонтальное разрешение окна и глубиной в зависимости от требуемой точности. Обеспечиваемая точность по глубине зависит от диапазона значений, которые может принимать z.

Для каждой сканирующей строки буфер кадра инициализируется фоновым значением интенсивности, а z-буфер – минимальным значением z. Затем определяется пересечение сканирующей строки с двумерной проекцией каждого многоугольника сцены, если они существуют. При рассмотрении каждого пиксела на сканирующей строке в интервале между концами пар его глубина сравнивается с глубиной, содержащейся в соответствующем элементе z-буфера. Если глубина этого пиксела больше глубины из z-буфера, то рассматриваемый отрезок будет текущим видимым отрезком. И, следовательно, атрибуты многоугольника, соответствующего данному отрезку, заносятся в буфер кадра в позиции данного пиксела; соответственно корректируется и z-буфер в этой позиции. После обработки всех многоугольников сцены буфер кадра размером в одну сканирующую строку содержит решение задачи удаления невидимых поверхностей для данной сканирующей строки. Он выводится на экран дисплея в порядке, определяемом растровым сканированием, то есть слева направо.

Однако практически сравнение каждого многоугольника с каждой сканирующей строкой оказывается неэффективным. Поэтому используется некоторая разновидность списка упорядоченных ребер. В частности, для повышения эффективности этого алгоритма применяются групповая сортировка по оси y, список активных многоугольников и список активных ребер.

Пример работы алгоритма показан на рис. 1.1.

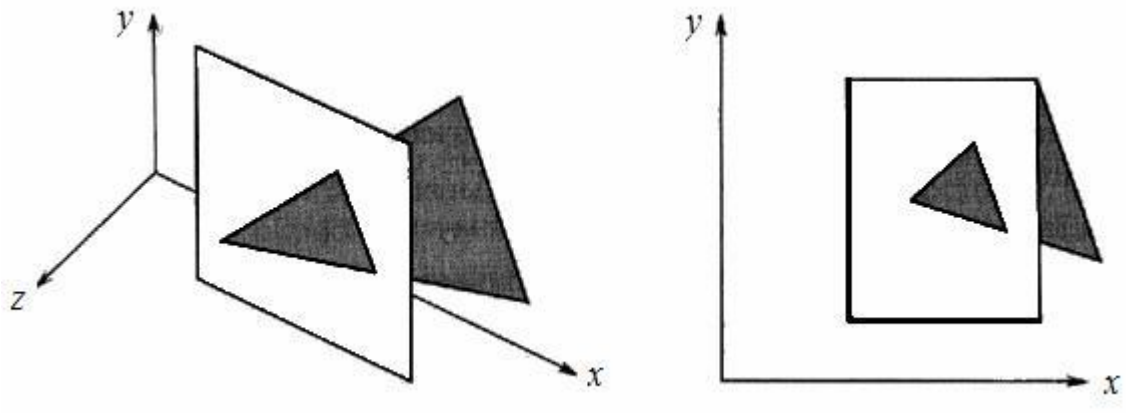


Рис. 1.1 – Протыкающий треугольник

Алгоритм, использующий иерархический z–буфер

Назовем грань скрытой (невидимой) по отношению к *z*-буферу, если для любого пикселя картинной плоскости, накрываемого этой гранью, глубина соответствующего пикселя грани не меньше значения в *z*-буфере. Ясно, что выводить скрытые грани не имеет смысла, так как они ничего не изменяют (они заведомо не являются видимыми).

Куб (прямоугольный параллелепипед) назовем скрытым по отношению к *z*-буферу, если все его лицевые грани являются скрытыми по отношению к этому *z*-буферу. Опишем куб вокруг некоторой группы граней. Перед выводом граней, содержащихся внутри этого куба, стоит проверить, не является ли куб скрытым. Если он скрытый, то все грани, содержащиеся внутри его, можно сразу же отбрасывать. Но, даже если куб не является скрытым, часть граней, содержащихся внутри его, все равно может быть скрытой, и поэтому нужно разбить его на части и проверить каждую из частей на скрытость.

Предложенный подход приводит к следующему алгоритму. Опишем куб вокруг всей сцены. Разобьем его на 8 равных частей.

Каждую из частей, содержащую достаточно много граней, снова разобьем и т. д. Если число граней внутри частичного куба меньше заданного числа, то разбивать его нет смысла, и он становится листом строящегося дерева. В результате получается восьмеричное дерево, где с каждым кубом связан список граней, содержащихся внутри его.

Вывод всей сцены можно представить следующим образом: очередной куб, начиная с корня дерева, проверяется на попадание в область видимости. Если куб в область видимости не попадает, то ни одна грань из него не видна, и мы его отбрасываем. В противном случае проверяем, не является ли этот куб скрытым. Скрытый куб также отбрасывается. В противном случае повторяем описанную процедуру для всех его восьми частей в порядке удаления: первым обрабатывается ближайший подкуб, последним – самый дальний. Для куба,

являющегося листом, вместо вывода его частей просто выводим все содержащиеся в нем грани.

Такой подход опирается на когерентность в объектном пространстве и позволяет легко отбросить основную часть невидимых граней.

Таким образом, данный алгоритм, во-первых, раскладывает в Z-буфер только те геометрические примитивы, которые содержатся в видимых ветвях дерева, то есть алгоритм не тратит время на ненужные ветви дерева разбиений, так как он посещает только те ветви, родительские структуры которых видимы, во-вторых, алгоритм никогда не посещает одни и те же ветви дважды.

1.4 Анализ моделей освещения

В стандартных моделях освещения итоговая освещённость складывается, обычно, из трёх компонент:

1. Фоновая составляющая
2. Рассеянная составляющая
3. Зеркальная составляющая

Фоновая составляющая: по сути это некоторая константа, которая прибавляется к освещённости в каждой точке освещаемой модели. Важно заметить, что фоновая составляющая не зависит от положения освещаемой точки в пространстве.

Рассеянная составляющая: эта компонента рассчитывается по закону Ламберта, исходя из предположения, что при попадании на поверхность свет рассеивается равномерно во все стороны.

Зеркальная составляющая: компонента, отвечающая за моделирование отражающих способностей материала. Добавление этой компоненты позволяет визуализировать блики на поверхности освещаемой модели.

Итоговая формула освещённости в точке выглядит примерно следующим образом:

$$I = Ia + Id + Is.$$

Модель освещения Ламберта

Вся грань закрашивается одним уровнем интенсивности, который высчитывается по закону Ламберта:

$$I = I_0 * \cos(\alpha), \text{ где}$$

I – результирующая интенсивность света в точке,

I_0 – интенсивность источника,

α – угол между нормалью к поверхности и вектором направления света.

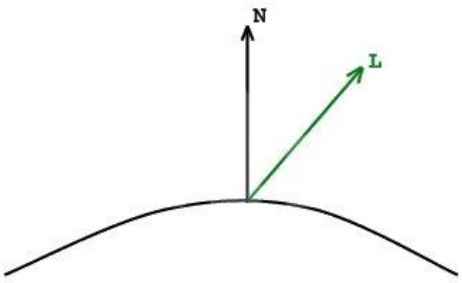


Рис. 1.2 – Векторы, используемые в модели освещения Ламберта

Этот метод крайне прост в реализации и совершенно не требователен к ресурсам. Однако плохо подходит для тел вращения, плохо учитывает отраженный свет.

Модель освещения Фонга

Модель Фонга базируется на модели Ламберта, но добавляет обработку зеркальной составляющей у визуализируемого материала.

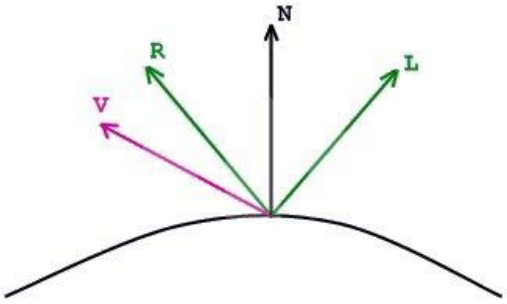


Рис. 1.3 – Векторы, используемые в модели освещения Фонга

В модели Фонга добавляется два новых вектора: направление из точки сцены на наблюдателя (камеру) *V* и отражённый от точки луч света *R*.

Модель Фонга работает следующим образом: чем меньше угол между векторами R и V, тем ярче должна быть освещена визуализируемая точка. Неформально эту логику можно объяснить следующим образом: маленький угол между этими векторами означает, что, отражаясь от точки на сцене луч из источника света попадает в камеру, "ослепляя её", что визуально отображается как блик.

При реализации, находится косинус между этими векторами (как скалярное произведение нормализованных векторов), который затем возводят в некоторую степень n, которая подбирается вручную и влияет на размер бликов. Чем меньше угол, тем больше косинус и ярче бликовая компонента.

Итоговая формула:

$$I=max(0,\ dot(N,L))+(dot(R,V))^{n}.$$

Модель освещения Блинна-Фонга

Модель Блинна-Фонга – это разновидность модели Фонга, исключающая расчет отраженного луча, что упрощает вычисления. Также она позволяет избавиться от характерной для модели Фонга проблемы с резкой границей области зеркального отражения.

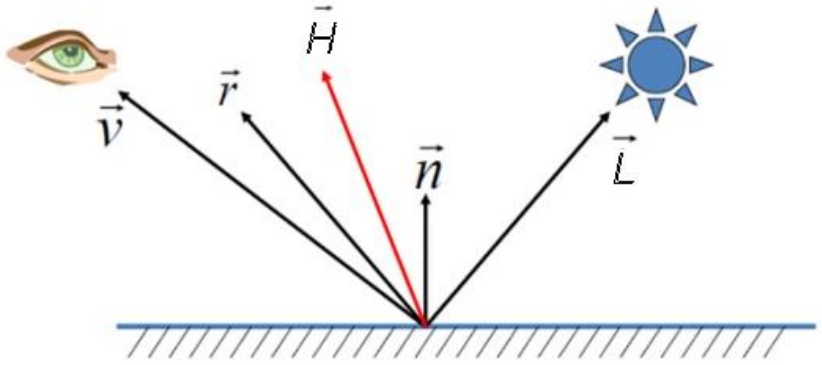


Рис. 1.4 – Векторы, используемые в модели освещения Блинна-Фонга

Вместо вычисления скалярного произведения между вектором отраженного луча и вектором на наблюдателя, в модели Блинна находится скалярное произведение среднего вектора (H) между вектором на наблюдателя (V) и вектором на источник света (L) и вектора нормали (N). А так как найти средний вектор между наблюдателем и источником света (фактически это просто нормализованная сумма двух векторов) с точки зрения вычислений проще, чем найти вектор отражения, модель Блинна более распространена в применении и используется во многих графических системах по умолчанию.

Итоговая формула:

$$I=max(0,\ dot(N,L))+(dot(H,N))^n.$$

В данной работе будет использована модель освещения Блинна-Фонга, так как она обеспечивает реалистичное освещение объектов сцены и при этом основана на более простых вычислениях, чем модель Фонга.

Вывод по аналитической части

В данном разделе была проведена формализация объектов сцены, изучены модификации алгоритма z-буфера удаления невидимых поверхностей, а также в результате анализа существующих моделей освещения сцены была выбрана наиболее подходящая для данной задачи модель.

2. Конструкторская часть

В данном разделе представлены требования к программному обеспечению, а также описаны изученные алгоритмы.

2.1 Требования к программному обеспечению

Программа должна предоставлять доступ к возможностям:

1) загрузки модели объемного тела из предоставленного перечня файлов
2) выбора модификации алгоритма z-буфера для визуализации модели
3) вращения и приближения, отдаления модели

Программа должна корректно реагировать на любые действия пользователя.

2.2 Разработка алгоритмов

Простой алгоритм Z-буфера

1. Всем элементам буфера кадра присвоить фоновое значение
2. Инициализировать Z буфер минимальными значениями глубины
3. Выполнить растровую развертку каждого многоугольника сцены:
   a. Для каждого пикселя, связанного с многоугольником вычислить его глубину z(x, y)
   b. Сравнить глубину пикселя со значением, хранимым в Z буфере.

   Если z(x, y) > zбуф(x, y), то zбуф(x,y) = z(x,y), цвет(x, y) = цветПикселя.

4. Отобразить результат

Алгоритм Z-буфера со сканирующей строкой

Подготовка информации:

- Для каждого многоугольника определить самую верхнюю сканирующую строку, которую он пересекает.

- Занести многоугольник в группу у, соответствующую этой сканирующей строке.

- Запомнить для каждого многоугольника как минимум следующую информацию: число сканирующих строк, которые пересекаются этим многоугольником; список ребер многоугольника; коэффициенты уравнения плоскости многоугольника; визуальные атрибуты многоугольника.

Решение задачи удаления невидимых поверхностей:

- Инициализировать буфер кадра дисплея.

- Для каждой сканирующей строки:

1. Инициализировать буфер кадра размером с одну сканирующую строку, заполнив его фоновым изображением

2. Инициализировать Z-буфер размером с одну сканирующую строку значением Zmin

3. Проверить необходимость добавления в список активных многоугольников (САМ) новых многоугольников

4. Если было добавление многоугольника в САМ, то добавить в САР соответствующие рёбра новых многоугольников

5. Если произведено удаление какого-либо элемента из пары рёбер САР, то проверить необходимость удаления всего многоугольника из САМ. Если он остаётся, то проверить необходимость удаления другого ребра из этой пары –

если его удалять не нужно, то доукомлектовать пару (добавив недостающее левое или правое ребро)

- В САР должна храниться следующая информация:

1. Хл – пересечение левого ребра с текущей сканирующей строкой
2. ΔХл – приращение Хл в интервале между соседними сканирующими строками
3. ΔYл – число сканирующих строк, пересекаемых левым ребром
4. Хп, ΔХп, ΔYп
5. ΔZx=-A/C для C<>0 (иначе ΔZx=0) – приращение по Z вдоль сканирующей строке
6. ΔZy=-B/C для C<>0 (иначе ΔZy=0) – приращение по Z в интервале между соседними сканирующими строками

- Для каждой пары ребёр многоугольника из САР выполнить:

1. Извлечь
2. Инициализировать Z значением Zл
3. Для каждого пикселя Хл≤X≤Хп вычислить Z(x,y=const). Z1=Zл, ..., Zk=Zk-1+ ΔZx
4. Если Z>Zбуф, то Zбуф=Z и занести атрибуты многоугольника в буфер кадра

- Записать буфер кадра сканирующей строки в буфер кадра дисплея

- Скорректировать САР

1. ΔYл--, ΔYп--
2. Хл=Хл+ΔХл, Хп=Хп+ΔХп
3. Zл=Zл+ΔZxΔX+ΔZy

4. Если ΔYл<0 или ΔYп<0, то удалить соответствующее ребро из списка, пометив положение обоих рёбер в списке и породивший их многоугольник

- Скорректировать САМ

1. ΔY--
2. Если ΔY<0, то удалить многоугольник из САМ

Алгоритм иерархического Z-буфера

1. Экран рекурсивно разбивается на 4 квадранта (вплоть до частей размером 1 пиксель) – создается дерево квадрантов (у каждого внутреннего узла ровно 4 потомка)
2. Всем элементам буфера кадра присвоить фоновое значение
3. Каждый многоугольник сцены добавить в корневой узел дерева:
    1) Если многоугольник полностью содержится одним из узлов-потомков, то он добавляется в этот узел-потомок
    2) Если же многоугольник не содержится полностью ни одним из узлов-потомков, то он растеризуется, а каждый пиксель из полученного множества аналогичным образом проверяется на принадлежность узлам-потомкам и добавляется в соответствующий узел (его параметры заносятся в буфер кадра в случае видимости)
4. Удалить дерево квадрантов
5. Отобразить результат

Алгоритм иерархического Z-буфера с октодеревом

1. Создается массив всех многоугольников сцены

2. Экран рекурсивно разбивается на 4 равных частей (вплоть до частей размером 1 пиксель) – создается дерево квадрантов (у каждого внутреннего узла ровно 4 потомка)

3. Пространство рекурсивно разбивается на 8 равных частей вплоть до тех пор, пока в каждом полученном фрагменте пространства не окажется не более 1 многоугольника – создается дерево октантов (у каждого внутреннего узла ровно 8 потомков), при этом множество многоугольников, принадлежащих данному узлу, делится между его потомками в случае принадлежности многоугольников какому-либо из конкретных узлов-потомков

4. Всем элементам буфера кадра присвоить фоновое значение

5. Добавить октодерево в корневой узел квадродерева – при этом рекурсивно обрабатываются все узлы октодерева:

   1) Все многоугольники, принадлежащие данному узлу, растеризуются, а каждый пиксель из полученного множества проверяется на принадлежность узлам-потомкам данного узла квадродерева и добавляется в соответствующий узел (его параметры заносятся в буфер кадра в случае видимости)

   2) Для каждого из узлов-потомков данного узла октодерева проверяется принадлежность какому-либо из узлов-потомков данного узла квадродерева – в случае принадлежности этот узел октодерева добавляется в соответствующий узел квадродерева

6. Удалить дерево квадрантов

7. Удалить дерево октантов

8. Отобразить результат

Вывод по конструкторской части

В данном разделе были представлены требования к программному обеспечению и описание реализуемых алгоритмов.

3. Технологическая часть

В данном разделе представлены средства разработки программного обеспечения, а также детали реализации рассмотренных алгоритмов.

3.1 Средства реализации

В качестве языка программирования был выбран C++.

Данный выбор обоснован высокой скоростью работы языка и поддержкой объектно-ориентированного подхода программирования.

Было принято решение использовать Qt Framework, который предоставляет большой набор базовых компонентов, необходимых для решения поставленной задачи. Кроме того, я ознакомлен со средой разработки Qt Creator, что облегчит процесс написания и отладки кода.

3.2 Реализация алгоритмов

В листингах 3.1 – 3.4 представлены реализации модификаций алгоритма z-буфера (реализации метода render для классов NaiveZBuffer, ScanlineZBuffer, NaiveHierarchicalZBuffer и OctreeHierarchicalZBuffer соответственно).

Листинг 3.1 – Реализация простого алгоритма z-буфера

```
      QImage NaiveZBuffer::render(std::vector <Vertex> &vertices,
std::vector<unsigned int> &indices) {
          std::vector<std::vector<float>> zBuffer(width);
          for (std::vector<float> &vector : zBuffer)
              vector = std::vector<float>(height, FLT_MAX);

          QImage ans(width, height, QImage::Format_RGB32);
          ans.fill(QColor(0, 0, 0));
          for (auto iter = indices.begin(); iter != indices.end(); ) {
              std::vector<Vertex> polygonVertices;
              for (int i = 0; i < 3; i++)
                  polygonVertices.push_back(vertices[*(iter++)]);
              Polygon polygon(polygonVertices, MVP, width, height);
              if (polygon.getDeltaX() > 0 && polygon.getDeltaY() > 0) {
                  std::vector<Pixel> pixels = calculatePixels(polygon, 0,
width - 1, 0, height - 1);
                  for (Pixel &pixel : pixels)
```

```
                    if (pixel.getZ() < zBuffer[pixel.getX()][pixel.getY()])
{
                        zBuffer[pixel.getX()][pixel.getY()] = pixel.getZ();
                        ans.setPixel(pixel.getX(), pixel.getY(),
pixel.getColor().rgb());
                    }
                }
            }
            return ans;
        }
```

Листинг 3.2 – Реализация алгоритма z-буфера со сканирующей строкой

```
        QImage ScanlineZBuffer::render(std::vector<Vertex> &vertices,
std::vector<unsigned int> &indices) {
            std::vector<Polygon> polygons;
            for (auto iter = indices.begin(); iter != indices.end(); ) {
                std::vector<Vertex> polygonVertices;
                for (int i = 0; i < 3; i++)
                    polygonVertices.push_back(vertices[*(iter++)]);
                Polygon polygon(polygonVertices, MVP, width, height);
                if (polygon.getDeltaX() > 0 && polygon.getDeltaY() > 0)
                    polygons.push_back(polygon);
            }
            std::sort(polygons.begin(), polygons.end());

            std::multimap<int, ActivePolygon> activePolygons;
            QImage ans(width, height, QImage::Format_RGB32);
            ans.fill(QColor(0, 0, 0));
            for (int index = 0, scanlineY = polygons.begin()->getY(); scanlineY
< height; ) {
                for (; index < polygons.size() && polygons[index].getY() ==
scanlineY; index++)
                    activePolygons.insert(std::make_pair(polygons[index].getY()
+ polygons[index].getDeltaY(), ActivePolygon(polygons[index])));
                for (std::pair<const int, ActivePolygon> &pair : activePolygons)
                    pair.second.check(scanlineY);

                if (scanlineY >= 0) {
                    std::vector<float> zBuffer(width, FLT_MAX);
                    for (std::pair<const int, ActivePolygon> &pair :
activePolygons) {
                        Segment segment = pair.second.segment();
                        int x = segment.getX() + segment.getDeltaX();
                        ActiveSegment activeSegment(segment);

                        for (int scanlineX = segment.getX(); scanlineX <= x &&
scanlineX < width; scanlineX++) {
                            if (scanlineX >= 0 && activeSegment.getZ() <
zBuffer[scanlineX]) {
                                zBuffer[scanlineX] = activeSegment.getZ();
                                QVector3D p = activeSegment.getP(), n =
activeSegment.getN();
                                QColor color = calculateColor(p, n);
                                ans.setPixel(scanlineX, scanlineY, color.rgb());
                            }
                            activeSegment.update();
                        }
                    }
```

```
                }

                while (!activePolygons.empty() && activePolygons.begin()->first
== scanlineY)
                    activePolygons.erase(activePolygons.begin());
                for (std::pair<const int, ActivePolygon> &pair : activePolygons)
                    pair.second.update();

                if (activePolygons.empty())
                    scanlineY = index < polygons.size() ? polygons[index].getY()
: height;
                else
                    scanlineY++;
            }

            return ans;

        }
```

Листинг 3.3 – Реализация алгоритма иерархического z-буфера

```
        QuadTree::QuadTree(int minX, int maxX, int minY, int maxY) {
            this->minX = minX;
            this->maxX = maxX;
            this->minY = minY;
            this->maxY = maxY;
            z = FLT_MAX;
            if (minX != maxX || minY != maxY) {
                int middleX = (minX + maxX) / 2, middleY = (minY + maxY) / 2;
                children.push_back(new QuadTree(minX, middleX, minY, middleY));
                if (middleX + 1 <= maxX)
                    children.push_back(new QuadTree(middleX + 1, maxX, minY,
middleY));
                if (middleY + 1 <= maxY)
                    children.push_back(new QuadTree(minX, middleX, middleY + 1,
maxY));
                if (middleX + 1 <= maxX && middleY + 1 <= maxY)
                    children.push_back(new QuadTree(middleX + 1, maxX, middleY +
1, maxY));
            }
        }

        bool QuadTree::contain(Pixel &pixel) {
            return minX <= pixel.getX() && maxX >= pixel.getX() && minY <=
pixel.getY() && maxY >= pixel.getY();
        }

        bool QuadTree::contain(Polygon &polygon) {
            return minX <= polygon.getX() && maxX >= polygon.getX() +
polygon.getDeltaX() && minY <= polygon.getY() && maxY >= polygon.getY() +
polygon.getDeltaY();
        }

        void QuadTree::update() {
            z = -FLT_MAX;
            for (QuadTree *child : children)
                z = std::max(z, child->z);
        }

        void QuadTree::addPixels(std::vector<Pixel> &pixels, QImage &image) {
            float minZ = FLT_MAX;
            for (Pixel &pixel : pixels)
```

```
                minZ = std::min(minZ, pixel.getZ());
            if (z <= minZ)
                return;

            if (minX == maxX && minY == maxY) {
                for (Pixel &pixel : pixels)
                    if (pixel.getZ() < z) {
                        z = pixel.getZ();
                        image.setPixel(pixel.getX(), pixel.getY(),
pixel.getColor().rgb());
                    }
            } else {
                std::vector<std::vector<Pixel>> childPixels(children.size());
                for (Pixel &pixel : pixels)
                    for (int i = 0; i < children.size(); i++)
                        if (children[i]->contain(pixel)) {
                            childPixels[i].push_back(pixel);
                            break;
                        }
                for (int i = 0; i < children.size(); i++)
                    if (!childPixels[i].empty())
                        children[i]->addPixels(childPixels[i], image);
                update();
            }
        }

        void QuadTree::addPolygon(Polygon &polygon, QImage &image, ZBuffer
*zBuffer) {
            if (z <= polygon.getZ())
                return;

            bool flag = true;
            for (QuadTree *child : children)
                if (child->contain(polygon)) {
                    child->addPolygon(polygon, image, zBuffer);
                    flag = false;
                    update();
                    break;
                }

            if (flag) {
                std::vector<Pixel> pixels = zBuffer->calculatePixels(polygon,
minX, maxX, minY, maxY);
                addPixels(pixels, image);
            }
        }

        QImage NaiveHierarchicalZBuffer::render(std::vector <Vertex> &vertices,
std::vector<unsigned int> &indices) {
            QuadTree *root = new QuadTree(0, width - 1, 0, height - 1);
            QImage ans(width, height, QImage::Format_RGB32);
            ans.fill(QColor(0, 0, 0));
            for (auto iter = indices.begin(); iter != indices.end(); ) {
                std::vector<Vertex> polygonVertices;
                for (int i = 0; i < 3; i++)
                    polygonVertices.push_back(vertices[*(iter++)]);
                Polygon polygon(polygonVertices, MVP, width, height);
                if (polygon.getDeltaX() > 0 && polygon.getDeltaY() > 0)
                    root->addPolygon(polygon, ans, this);
            }
            delete root;
            return ans;

        }
```

Листинг 3.4 – Реализация алгоритма иерархического z-буфера с октодеревом

```
      Octree::Octree(int minX, int maxX, int minY, int maxY, float minZ, float
maxZ, std::vector<Polygon> &polygons) {
          this->minX = minX;
          this->maxX = maxX;
          this->minY = minY;
          this->maxY = maxY;
          this->minZ = minZ;
          this->maxZ = maxZ;
          if (polygons.size() <= THRESHOLD) {
              z = minZ;
              this->polygons = polygons;
          } else {
              int middleX = (minX + maxX) / 2, middleY = (minY + maxY) / 2;
              float middleZ = (minZ + maxZ) / 2;
              std::vector<std::vector<Polygon>> childPolygons(8);
              for (Polygon &polygon : polygons)
                  if (polygon.getX() + polygon.getDeltaX() <= middleX &&
polygon.getY() + polygon.getDeltaY() <= middleY && polygon.getZ() +
polygon.getDeltaZ() <= middleZ)
                      childPolygons[0].push_back(polygon);
                  else if (polygon.getX() + polygon.getDeltaX() <= middleX &&
polygon.getY() + polygon.getDeltaY() <= middleY && polygon.getZ() > middleZ)
                      childPolygons[1].push_back(polygon);
                  else if (polygon.getX() + polygon.getDeltaX() <= middleX &&
polygon.getY() > middleY && polygon.getZ() + polygon.getDeltaZ() <= middleZ)
                      childPolygons[2].push_back(polygon);
                  else if (polygon.getX() + polygon.getDeltaX() <= middleX &&
polygon.getY() > middleY && polygon.getZ() > middleZ)
                      childPolygons[3].push_back(polygon);
                  else if (polygon.getX() > middleX && polygon.getY() +
polygon.getDeltaY() <= middleY && polygon.getZ() + polygon.getDeltaZ() <=
middleZ)
                      childPolygons[4].push_back(polygon);
                  else if (polygon.getX() > middleX && polygon.getY() +
polygon.getDeltaY() <= middleY && polygon.getZ() > middleZ)
                      childPolygons[5].push_back(polygon);
                  else if (polygon.getX() > middleX && polygon.getY() >
middleY && polygon.getZ() + polygon.getDeltaZ() <= middleZ)
                      childPolygons[6].push_back(polygon);
                  else if (polygon.getX() > middleX && polygon.getY() >
middleY && polygon.getZ() > middleZ)
                      childPolygons[7].push_back(polygon);
                  else
                      this->polygons.push_back(polygon);
              children.push_back(new Octree(minX, middleX, minY, middleY,
minZ, middleZ, childPolygons[0]));
              children.push_back(new Octree(minX, middleX, minY, middleY,
middleZ, maxZ, childPolygons[1]));
              children.push_back(new Octree(minX, middleX, middleY + 1, maxY,
minZ, middleZ, childPolygons[2]));
              children.push_back(new Octree(minX, middleX, middleY + 1, maxY,
middleZ, maxZ, childPolygons[3]));
              children.push_back(new Octree(middleX + 1, maxX, minY, middleY,
minZ, middleZ, childPolygons[4]));
              children.push_back(new Octree(middleX + 1, maxX, minY, middleY,
middleZ, maxZ, childPolygons[5]));
              children.push_back(new Octree(middleX + 1, maxX, middleY + 1,
maxY, minZ, middleZ, childPolygons[6]));
              children.push_back(new Octree(middleX + 1, maxX, middleY + 1,
maxY, middleZ, maxZ, childPolygons[7]));
```

```
                for (Octree *child : children)
                    z = std::min(z, child->z);
            }
        }

        bool QuadTree::contain(Octree *octree) {
            return minX <= octree->getMinX() && maxX >= octree->getMaxX() &&
minY <= octree->getMinY() && maxY >= octree->getMaxY();
        }

        void QuadTree::addOctree(Octree *octree, QImage &image, ZBuffer
*zBuffer) {
            if (z <= octree->getZ())
                return;

            std::vector<Polygon> polygons = octree->getPolygons();
            for (Polygon &polygon : polygons) {
                std::vector<Pixel> pixels = zBuffer->calculatePixels(polygon,
minX, maxX, minY, maxY);
                addPixels(pixels, image);
            }

            std::vector<Octree *> octreeChildren = octree->getChildren();
            for (Octree *octreeChild : octreeChildren)
                for (QuadTree *child : children)
                    if (child->contain(octreeChild))
                        child->addOctree(octreeChild, image, zBuffer);
            update();
        }

        QImage OctreeHierarchicalZBuffer::render(std::vector<Vertex> &vertices,
std::vector<unsigned int> &indices) {
            float minZ = FLT_MAX, maxZ = -FLT_MAX;
            std::vector<Polygon> polygons;
            for (auto iter = indices.begin(); iter != indices.end(); ) {
                std::vector<Vertex> polygonVertices;
                for (int i = 0; i < 3; i++)
                    polygonVertices.push_back(vertices[*(iter++)]);
                Polygon polygon(polygonVertices, MVP, width, height);
                if (polygon.getDeltaX() > 0 && polygon.getDeltaY() > 0) {
                    polygons.push_back(polygon);
                    minZ = std::min(minZ, polygon.getZ());
                    maxZ = std::max(maxZ, polygon.getZ() + polygon.getDeltaZ());
                }
            }

            QuadTree *rootQuadTree = new QuadTree(0, width - 1, 0, height - 1);
            Octree *rootOctree = new Octree(0, width - 1, 0, height - 1, minZ,
maxZ, polygons);
            QImage ans(width, height, QImage::Format_RGB32);
            ans.fill(QColor(0, 0, 0));
            rootQuadTree->addOctree(rootOctree, ans, this);
            delete rootQuadTree;
            delete rootOctree;
            return ans;

        }
```

Все перечисленные классы наследуются от базового класса ZBuffer, в котором определен метод calculatePixels для растеризации многоугольников. Реализация метода представлена в листинге 3.5.

Листинг 3.5 – Реализация алгоритма растеризации многоугольников

```
      std::vector<Pixel> ZBuffer::calculatePixels(Polygon &polygon, int minX,
int maxX, int minY, int maxY) {
          std::vector<Pixel> ans;
          ActivePolygon activePolygon(polygon);
          for (int scanlineY = polygon.getY(); scanlineY <= polygon.getY() +
polygon.getDeltaY() && scanlineY <= maxY; scanlineY++) {
              activePolygon.check(scanlineY);

              if (scanlineY >= minY) {
                  Segment segment = activePolygon.segment();
                  ActiveSegment activeSegment(segment);

                  for (int scanlineX = segment.getX(); scanlineX <=
segment.getX() + segment.getDeltaX() && scanlineX <= maxX; scanlineX++) {
                      if (scanlineX >= minX) {
                          float z = activeSegment.getZ();
                          QVector3D p = activeSegment.getP(), n =
activeSegment.getN();
                          QColor color = calculateColor(p, n);
                          ans.push_back(Pixel(scanlineX, scanlineY, z,
color));
                      }
                      activeSegment.update();
                  }
              }
              activePolygon.update();
          }
          return ans;

      }
```

Вывод по технологической части

В данном разделе были рассмотрены средства, с помощью которых было реализовано программное обеспечение, а также представлены листинги кода с реализацией рассмотренных алгоритмов.

4. Исследовательская часть

В данном разделе приведены примеры работы разработанного программного обеспечения, а также проведено исследование, в котором будут сравнены временные характеристики реализаций алгоритмов.

4.1 Примеры работы программного обеспечения

На рисунках 4.1 – 4.4 представлены примеры работы разработанного программного обеспечения с разными моделями из предоставленного перечня.

Продемонстрирована возможность выбора конкретной модификации алгоритма z-буфера. На рисунках 4.3 – 4.4 показана возможность вращать и приближать модель.

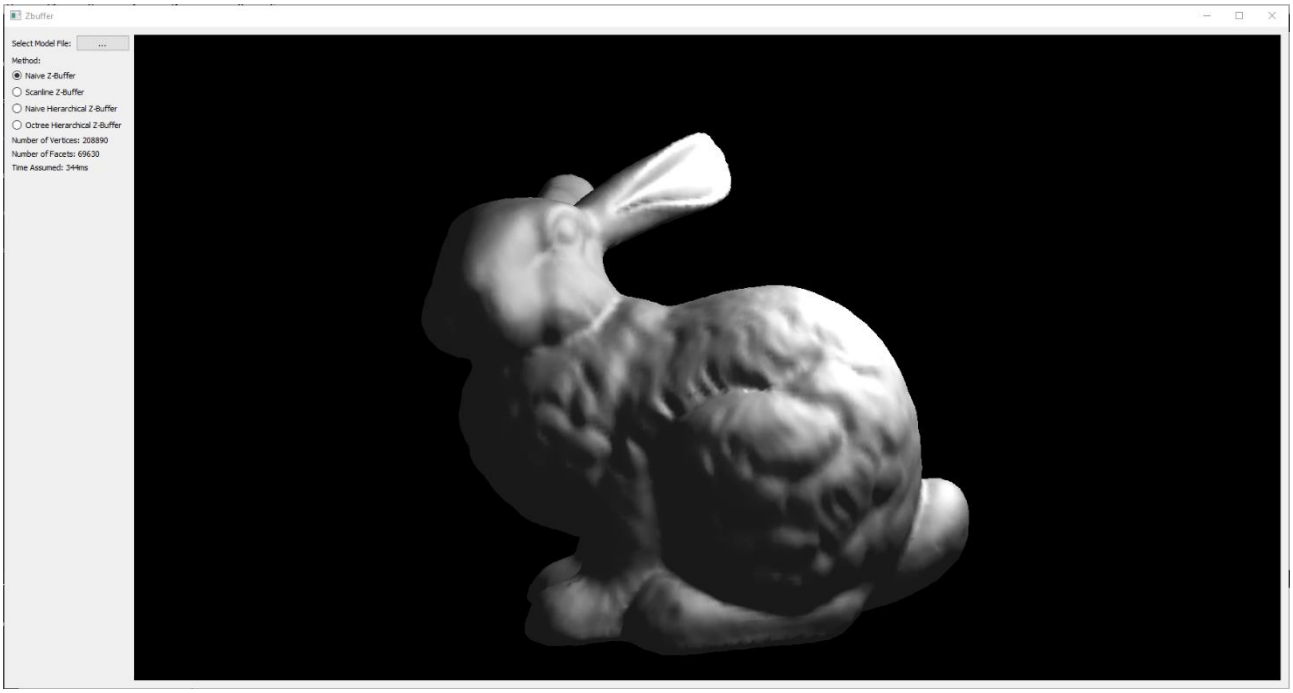


Рис. 4.1 – Визуализация bunny.obj с помощью простого алгоритма z-буфера

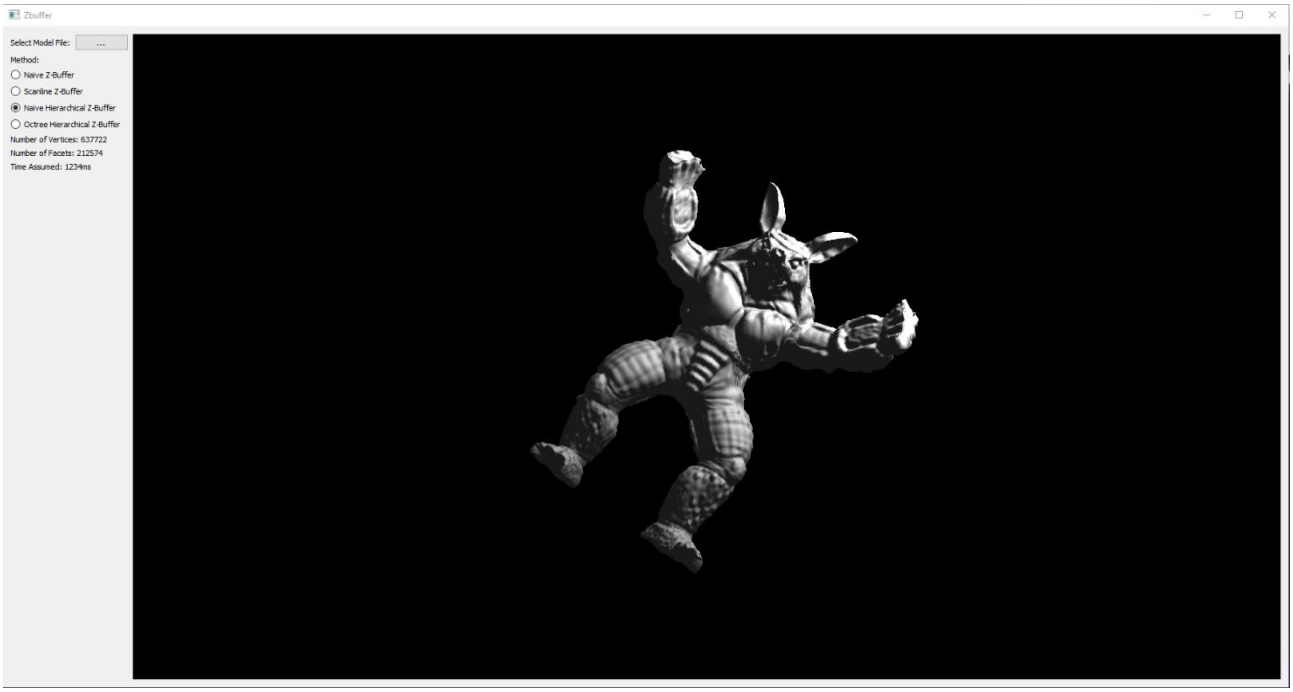


Рис. 4.2 – Визуализация armadillo.obj с помощью алгоритма иерарх. z-буфера

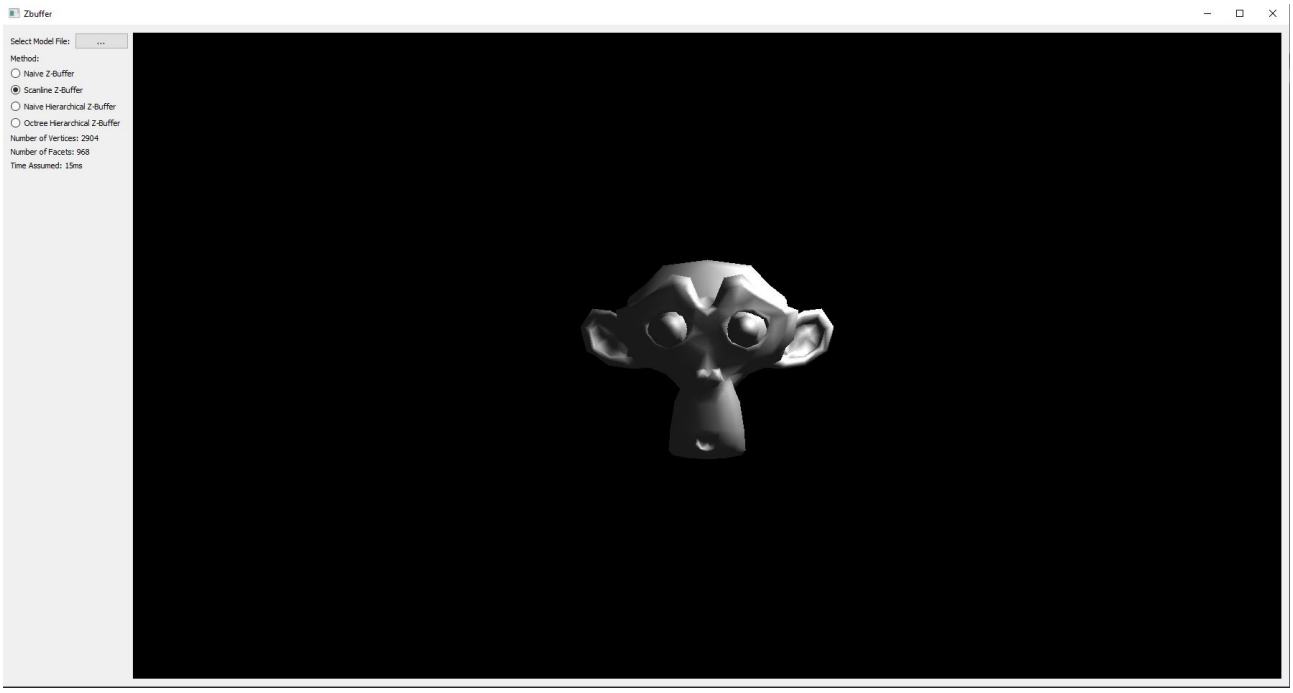


Рис. 4.3 – Визуализация monkey.obj с помощью алгоритма z-буфера со сканирующей строкой

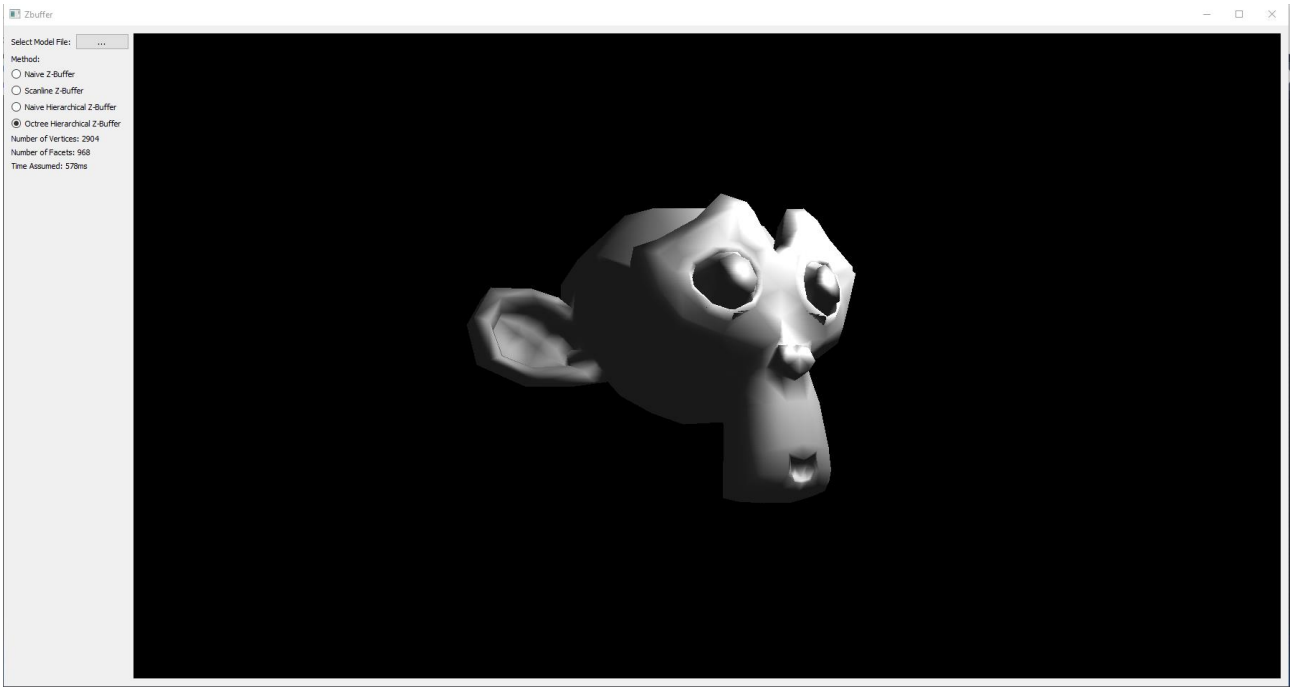


Рис. 4.4 – Визуализация monkey.obj с помощью алгоритма иерархического z-буфера с октодеревом

4.2 Постановка исследования

Цель исследования – сравнить временные характеристики реализаций алгоритмов на моделях разной сложности и сделать вывод о целесообразности применения того или иного алгоритма для конкретной задачи.

Модели, используемые для сравнения реализаций алгоритмов, представлены в таблице 4.1.

Таблица 4.1 – Характеристики используемых моделей

Файл	Количество вершин	Количество граней
cube.obj	36	12
monkey.obj	2904	968
bunny.obj	208890	69630
armadillo.obj	637722	212574

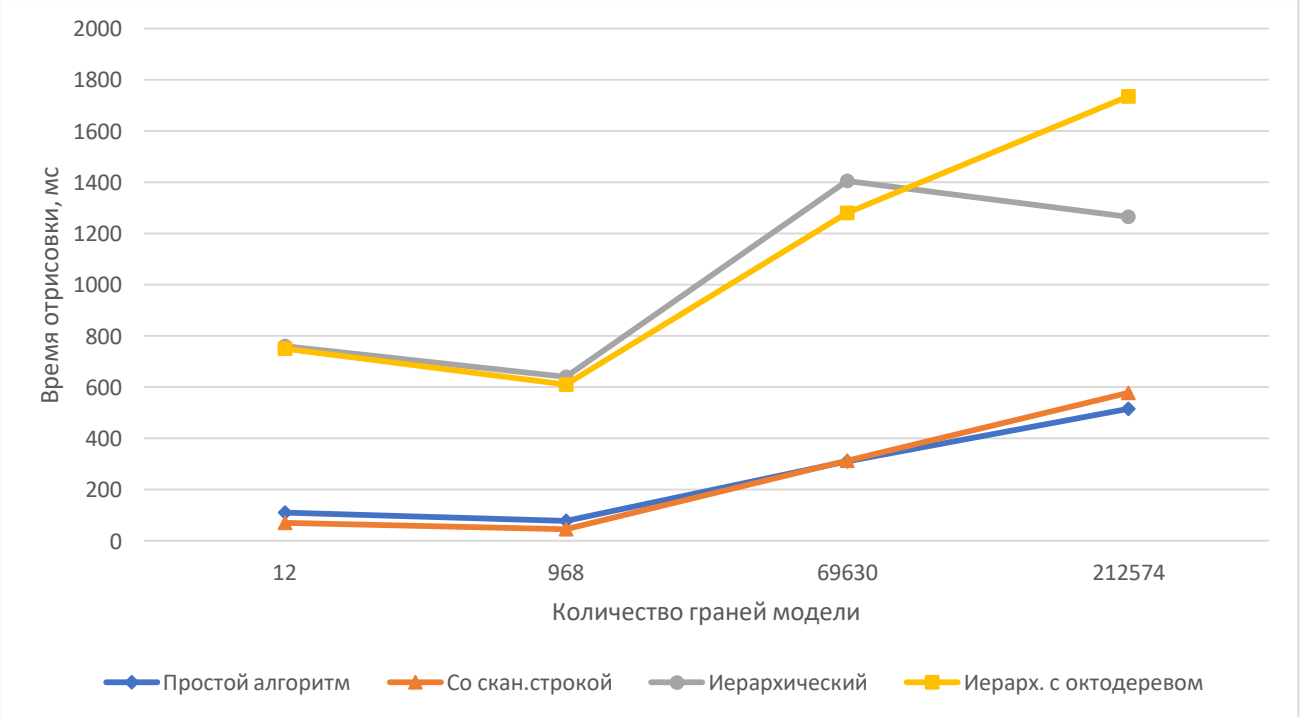
Время, необходимое для визуализации каждой модели с использованием различных алгоритмов, показано в таблице 4.2.

Таблица 4.2 – Временные характеристики реализаций алгоритмов

Время (мс)	Cube.obj	Monkey.obj	Bunny.obj	Armadillo.obj
Простой алгоритм	110	78	312	515
Алгоритм со сканирующей строкой	70	47	313	578
Иерархический алгоритм	761	641	1406	1266
Иерархический алгоритм с октодеревом	750	610	1281	1735

Зависимость времени визуализации модели от количества её граней для каждого из рассмотренных алгоритмов показана на графике 4.1.

График 4.1 – Зависимость времени отрисовки модели от количества граней



Можно сделать вывод, что, хотя алгоритм иерархического Z-буфера требует больше времени, чем обычный алгоритм Z-буфера в различных тестовых случаях, временные затраты на иерархический алгоритм значительно медленнее увеличиваются с увеличением сложности модели. Алгоритм иерархического Z-буфера хорошо подходит для сцен с большим количеством многоугольников и перекрытий между ними.

Вывод по исследовательской части

В данном разделе были представлены примеры работы разработанного программного обеспечения, а также проведено исследование, целью которого было сравнение временных характеристик реализаций алгоритмов.

Заключение

В ходе выполнения курсового проекта было разработано программное обеспечение, которое предоставляет возможность загрузки моделей объемных тел из перечня файлов и визуализирует их с использованием выбранной модификации алгоритма z-буфера, а также позволяет вращать, приближать и отдалять полученное трехмерное изображение модели. В процессе выполнения данной работы были выполнены следующие задачи:

- формализованы объекты сцены;
- изучены модификации алгоритма z-буфера удаления невидимых поверхностей;
- изучены существующие модели освещения сцены и выбрана наиболее подходящая для данной задачи модель;
- реализованы алгоритмы удаления невидимых поверхностей с использованием z-буфера;
- проведено сравнение временных характеристик реализаций алгоритмов.

В ходе выполнения исследовательской части было установлено, что реализация алгоритма иерархического z-буфера проигрывает реализации простого алгоритма z-буфера по временным характеристикам на использованных моделях, но при усложнении модели временные затраты на выполнение иерархического алгоритма растут значительно медленнее, чем затраты на выполнение простого алгоритма. Из этого можно сделать вывод о том, что при достижении определенного уровня сложности модели иерархический алгоритм окажется эффективнее простого алгоритма.

Список использованной литературы

1. Методы представления дискретных данных [Электронный ресурс]. – Режим доступа: https://www.graphicon.ru/oldgr/ru/library/multires_rep/index.html (дата обращения 27.11.21)
2. Полигональная сетка [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%B3%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%B5%D1%82%D0%BA%D0%B0 (дата обращения 27.11.21)
3. Е. А. Снижко. Компьютерная геометрия и графика [Текст], 2005. - 17 с.
4. Qt Widgets 5.15.5 [Электронный ресурс]. — Режим доступа: https://doc.qt.io/qt-5/qtwidgets-index.html (дата обращения: 10.12.2021).
5. Алгоритм Z-буфера [Электронный ресурс]. — Режим доступа: http://compgraph.tpu.ru/zbuffer.htm (дата обращения: 03.12.2021).
6. Алгоритм иерархического Z-буфера [Электронный ресурс]. — Режим доступа: https://mybiblioteka.su/tom3/2-76939.html (дата обращения: 03.12.2021).