Министерство образования и науки Российской Федерации
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)
Факультет прикладной математики и кибернетики
Кафедра программирования

ДОПУСТИТЬ К ЗАЩИТЕ В ГЭК
Руководитель ООП
д-р техн.наук, профессор
_____ А.М. Горцев
«_06_» _июня_ 2017 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

АНАЛИЗ АЛГОРИТМОВ, ИСПОЛЬЗУЮЩИХ СПИСКИ ПРИОРИТЕТОВ ДЛЯ ПОСТРОЕНИЯ РЕАЛИСТИЧНЫХ ИЗОБРАЖЕНИЙ

по основной образовательной программе подготовки бакалавров
по направлению подготовки 01.03.02 Прикладная математика и информатика

Казакова Анна Владимировна

Руководитель ВКР
д-р техн.наук, профессор
_____ А.Ю. Матросова
«_01_» _июня_ 2017 г.

Автор работы
студент группы № 1132
_____ А.В. Казакова

Томск – 2017

Министерство образования и науки Российской Федерации
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)
Факультет прикладной математики и кибернетики
Кафедра программирования

УТВЕРЖДАЮ
Руководитель ООП
д-р техн. наук, профессор
_____ А.М. Горцев
«16» января 2017 г.

ЗАДАНИЕ

по подготовке ВКР бакалавра
специалиста, бакалавра, магистра нужное вписать (напечатать)
студенту Казаковой Анне Владимировне группы № 1132

1. Тема ВКР «Анализ алгоритмов, использующих списки приоритетов для построения реалистичных изображений»

2. Срок сдачи студентом выполненной ВКР:
а) на кафедре 02.06.17г
б) в ГЭК 08.06.17г

2. Исходные данные к работе
Необходимо реализовать два представителя из алгоритмов, формирующих списки приоритетов: «алгоритм художника» и двоичное разбиение пространства, дать сравнительный анализ реализованных алгоритмов, а также сделать выводы на основе анализа.

3. Краткое содержание работы
Алгоритмы, использующие списки приоритетов. Реализация алгоритмов. Сравнительный анализ алгоритмов.
План работы включает: 1. Изучение алгоритмов, использующих списки приоритетов: алгоритм художника, двоичное разбиение пространства. 2. Реализация предложенных алгоритмов. 3. Проведение сравнительного анализа алгоритмов.

4. Указать предприятие, организацию, по заданию которого выполняется работа
Кафедра программирования ФПМК НИ ТГУ

6. Перечень графического материала (с точным указанием обязательных чертежей)
20 рисунков, 1 таблица

7. Дата выдачи задания «16» января 2017г.

Руководитель ВКР
д-р техн. наук, профессор — А.Ю. Матросова
должность, место работы — *подпись* — *инициалы, фамилия*

Задание принял к исполнению 16.01.17г
дата, подпись студента

РЕФЕРАТ

Цель работы: целью данной работы является реализация алгоритмов художника и двоичного разбиения пространства, а также проведение сравнительного анализа их качественных характеристик.

Объект исследования: алгоритмы, формирующие списки приоритетов, для решения проблемы видимости.

Ключевые слова: компьютерная графика, удаление невидимых поверхностей, алгоритмы, использующие списки приоритетов, алгоритм художника, двоичное разбиение пространства, BSP-деревья.

Выпускная квалификационная работа состоит из 3 разделов, 15 подразделов, 1 приложения, 20 страниц, включает 20 рисунков, 1 таблицу, 9 источников.

Результат работы:

Результатом работы является программная реализация алгоритма художника и двоичного разбиения пространства, а также анализ и сравнение алгоритмов для выявления критериев выбора наиболее подходящего алгоритма при заданных условиях.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 3
Удаление невидимых линий и поверхностей 3
1 Алгоритмы, формирующие списки приоритетов 6
1.1 Алгоритм художника 6
1.2 Двоичное разбиение пространства (Binary Space Partitioning) 7
2 Реализация алгоритмов 11
2.1 Классы, используемые для решения задачи 11
2.2 Реализация алгоритма художника 12
2.3 Реализация двоичного разбиения пространства 13
2.3.1 Определение положения точки относительно плоскости 13
2.3.2 Определение положения полигона относительно плоскости 15
2.3.3 Рассечение полигона плоскостью 16
2.3.4 Выбор оптимальной разделяющей плоскости 17
2.3.5 Построение BSP-дерева 18
3 Сравнительный анализ алгоритмов 22
3.1 Сравнение алгоритмов по качеству полученных изображений ... 22
3.1.1 Алгоритм художника 22
3.1.2 Двоичное разбиение пространства 24
3.1.3 Вывод 27
3.2 Сравнение алгоритмов по времени выполнения 27
ЗАКЛЮЧЕНИЕ 29
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 30
ПРИЛОЖЕНИЕ А Листинг программы 31

ВВЕДЕНИЕ

Компьютерная графика – одно из популярных и наиболее развивающихся направлений в компьютерных науках. Популярность компьютерной графики обусловлена тем, что виртуальные среды предоставляют возможность моделировать ситуации и объекты, не беспокоясь о последствиях, стоимости и других параметрах, которые в реальной жизни либо недоступны, либо ограничены в объемах. Так, например, архитекторы могут строить модели домов, не прилагая больших усилий в черчении, пилоты могут учиться управлять самолетами без риска для жизни, биохимики могут моделировать сложные молекулярные структуры и изучать их, а существующие пользовательские взаимодействия, такие как вращение и масштабирование, используются для лучшего обзора и анализа структуры. Но, конечно, наибольшее распространение в области компьютерной графики получила разработка игр, как сфера, наиболее востребованная у множества конечных пользователей, поскольку не является узкоспециализированной и нацеленной только на определенную группу людей специалистов.

Однако, никаких графических приложений не было бы, если бы не было алгоритмов, позволяющих преобразовывать 3D-графику для отображения в двухмерном пространстве и строить реалистичные изображения.

Удаление невидимых линий и поверхностей

Для построения реалистичного изображения нам необходимо знать какие грани являются видимыми, а какие скрыты от нас за другими, так появилась задача удаления невидимых линий и поверхностей. Необходимость определения видимости граней можно проиллюстрирована на рисунке 1. Здесь изображена реберная сцена куба, однако мы можем интерпретировать его положение по-разному, в зависимости от угла обзора и положения наблюдателя.

Сложность задачи привела к появлению большого числа различных способов ее решения. Большинство из них ориентированы на специализированные приложения. Однако наилучшего решения обшей задачи удаления невидимых линий и поверхностей не существует.

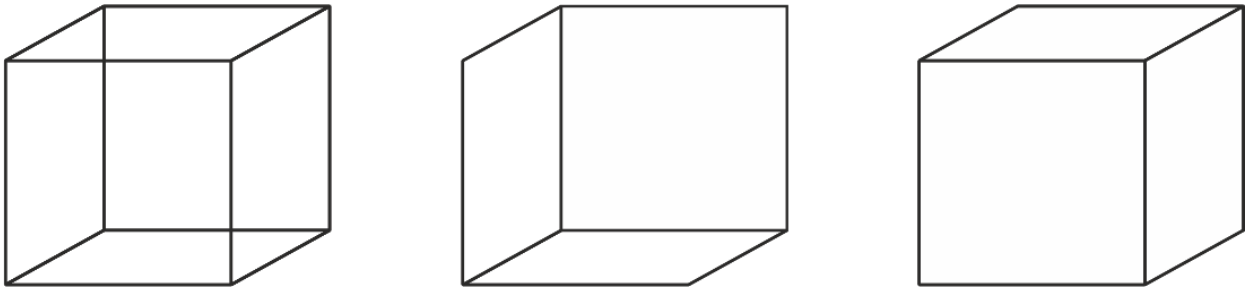


Рисунок 1 – Зависимость положения вида куба от угла обзора и положения наблюдателя

Для моделирования процессов в реальном времени требуются быстрые алгоритмы, с помощью которых можно достигнуть частоты видео 30 кадров в секунду. Для машинной мультипликации, требуются алгоритмы, которые могут генерировать сложные реалистические изображения с тенями, различными фактурами, прозрачностью, а также необходимо учитывать эффекты отражения и преломления цвета. Подобные алгоритмы работают медленно, и зачастую на вычисления требуется несколько минут или даже часов.

Стоит заметить, что учет эффектов прозрачности, фактуры, отражения и прочего не входит в задачу удаления невидимых поверхностей, однако они могут быть встроены в эти алгоритмы.

Рассматриваемые нами алгоритмы можно классифицировать по способу выбора пространства в котором они работают. Выделяют три класса алгоритмов удаления невидимых линий и поверхностей.

Алгоритмы, работающие в объектном пространстве. Они работают в системе координат объектов и получают результаты, точность которых

ограничена лишь точностью вычислений. В результате получаются хорошо масштабируемые изображения.

Алгоритмы, работающие в пространстве изображения (экрана). Такие алгоритмы работают в системе координат экрана, на котором объекты визуализируются, а значит точность вычислений ограничены разрешающей способностью экрана.

Алгоритмы, формирующие список приоритетов. Являются комбинацией предыдущих двух методов и работают путем определения элементов сцены по удаленности от наблюдателя. А далее объекты выводятся на экран от дальнего к ближнему.

В данной работе будут детально рассмотрены и реализованы алгоритмы, относящиеся к классу формирующих списки приоритетов, а именно: алгоритм художника, как самый элементарный и интуитивно понятный, и двоичное разбиение пространства (Binary Space Partition или BSP-дерево), считающийся точным и быстрым. В процессе их реализации будут выявлены их слабые и сильные стороны, а также проведен анализ и сравнение их скорости и точности.

Стоит заметить, что с момента появления первых игр от первого лица и до настоящего времени используется такое решение как двоичное разбиение пространства. Оно было впервые реализовано Джоном Кармаком при разработке движка Doom, использовавшегося в одноименной игре от первого лица. Это решение до сих пор так или иначе применяется в игровой индустрии, поскольку его применение значительно упрощает решение проблемы видимости, а предварительное построение дерева для статичной сцены уменьшает в будущем время на обработку всех её элементов. Кроме того, с момента их первой реализации и использования лишь в качестве определения порядка полигона было обнаружено много других полезных свойств, которыми пользуются и по сей день.

1 Алгоритмы, формирующие списки приоритетов

Рассмотрим задачу построения сцены из трехмерных объектов. Все объекты могут быть заданы набором точек и граней, состоящих из этих точек. Нашей целью является построение упорядоченного списка граней, учитывающее местоположение наблюдателя. Таким образом входными данными для каждого алгоритма будут список полигонов, каждый полигон будет задан множеством трехмерных точек, и координаты точки, в которой находится наблюдатель.

Рассматриваемые алгоритмы используют идею упорядочивания всех элементов сцены, а точнее, многоугольников, по удаленности от наблюдателя. Итогом реализации такого алгоритма является список элементов сцены, в нем никакие два элемента не перекрывают друг друга. По такому списку очень просто построить двухмерное изображение, записывая в буфер кадра каждый элемент списка, начиная с наиболее удаленного от наблюдателя.

1.1 Алгоритм художника

Самым простейшим и интуитивно понятным алгоритмом для решения «проблемы видимости» является алгоритм художника. В нем, согласно разным источникам, грани сортируются либо по среднему значению координаты z каждого многоугольника

$$avg[i] = \frac{\sum_{j=1}^{n} coordinates[i][j].z}{n},$$

либо по минимальному значению координаты z в каждом из многоугольников

$$min[i] = \min_{1 \le i \le n} coordinates[i][j].z\,.$$

Здесь n – количество вершин в многоугольнике, coordinates[i][j].z – z-координата j-ой вершины i-го многоугольника сцены.

1.2 Двоичное разбиение пространства (Binary Space Partitioning)

Binary Space Partitioning представляет собой рекурсивное иерархическое разбиение в n-мерном пространстве. Построение дерева — процесс, использующий гиперплоскость для разбиения пространства. Под гиперплоскостью в n-мерном пространстве подразумевается (n-1)-мерный объект, который позволяет разделить пространство на две части. Например, в 3-мерном пространстве гиперплоскость — это просто плоскость, а в двумерном — линия. Взятое в целом BSP-дерево представляет собой все пространство, а каждый его узел ограничивает выпуклое подпространство.

Каждый из узлов дерева хранит информацию о гиперплоскости, делящей пространство узла на 2 части (переднюю и заднюю, что определяется направлением вектора нормали этой плоскости), а также ссылки на два новых узла, представляющих эти части. Помимо этого, в узлах может храниться информация об одном или нескольких полигонах, лежащих в этой "гиперплоскости".

Рассмотрим пример построения BSP-дерева в двумерном пространстве. Здесь точка – наблюдатель, линии – некоторые препятствия и направления их нормалей (Рисунок 1.1).

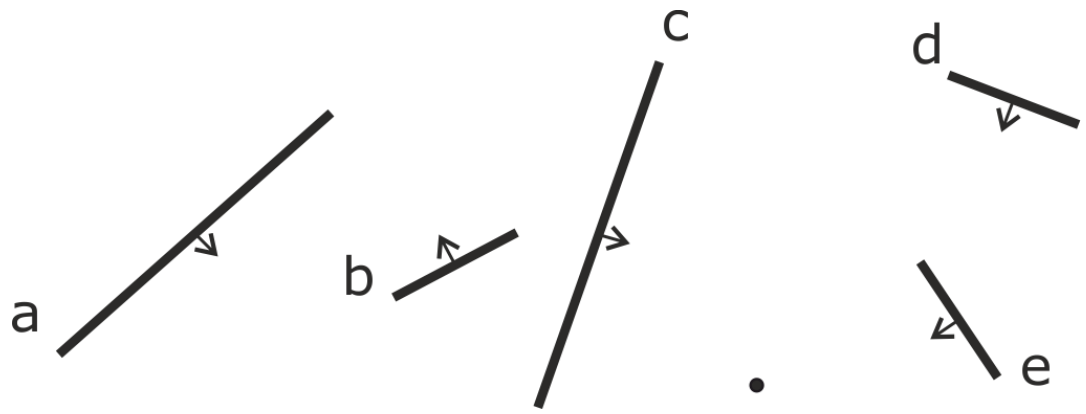


Рисунок 1.1 – Пример сцены в двумерном пространстве

Основой построения дерева является разбиение всего пространства некоторой плоскостью, которой принадлежит тот или иной полигон. Полигон располагается спереди, если он лежит в полуплоскости, в которое указывает направление нормали. Пусть первым разделителем (сплиттером) будет полигон **b**. На рисунке мы можем видеть, перед разделяющим полигоном лежит полигон **a**, позади лежат полигоны **d** и **e**. Полигон **c** разделяется на $\mathbf{c_1}$ и $\mathbf{c_2}$ (Рисунок 1.2).

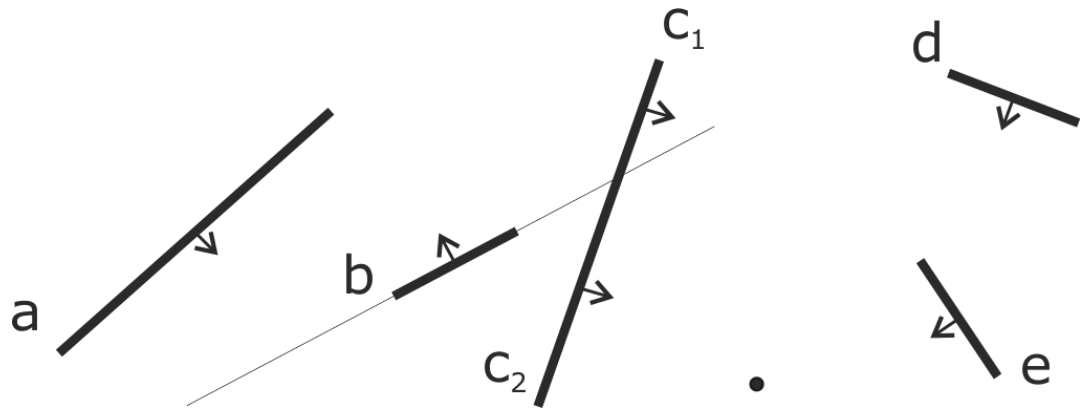


Рисунок 1.2 – Рассечение полигона **c** полигоном **b**

Таким образом, после рассечения имеем два списка полигонов:

- расположенные перед сплиттером: **a**, $\mathbf{c_1}$;
- расположенные позади: $\mathbf{c_2}$, **d**, **e**.

В узел дерева заносится сплиттер, и далее рекурсивно строится дерево, используя в качестве входных параметров список полигонов. Так, продолжаем разбивать пространство, пока во входных параметрах не останется полигонов.

Рассмотрим список **a**, $\mathbf{c_1}$. Возьмем в качестве сплиттера полигон **a**, тогда позади полигонов нет, а спереди полигон $\mathbf{c_1}$. У полигона $\mathbf{c_1}$ при разделении им пространства полигонов не окажется ни спереди, ни сзади.

Подобным образом поступим со вторым списком полигонов **d**, **e**, $\mathbf{c_2}$. Выбирая в качестве разделителя полигон **e**, получим:

- полигоны, расположенные перед сплиттером: $\mathbf{c_2}$;
- полигоны, расположенные позади сплиттера: **d**.

Рекурсивно продолжая процесс разделения пространства с всё возникающими новыми наборами входных параметров, строим дерево. В результате всех действий получим дерево, представленное на рисунок 1.3. Это и есть BSP-дерево.

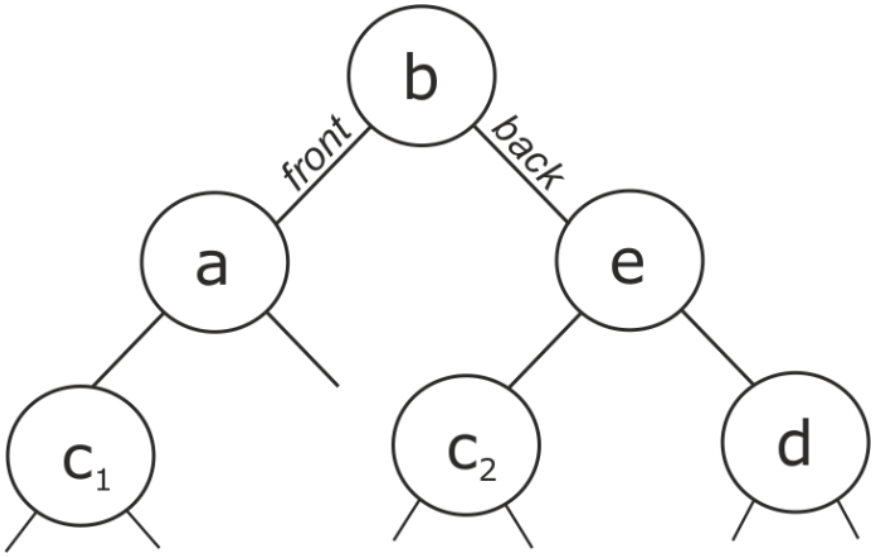


Рисунок 1.3 – BSP-дерево для сцены на рисунке 1.1

Кроме того, может возникнуть ситуация, когда несколько полигонов лежат на одной плоскости. В таком случае, если один из таких полигонов будет являться сплиттером, все полигоны принадлежащие плоскости сплиттера заносятся вместе с ним в узел дерева.

Рассмотрим, как из BSP-дерева получить список полигонов, расположенных в порядке убывания от наблюдателя.

```
void MakeList (BSPTree *tree)
{int result = ClassifyPoint(onlookPoint, tree->polygon);
    if (result == FRONT){
        if (tree->back != NULL) MakeList (tree->back);
        AddToList(tree->list_of_polygons);
        if (tree->front != NULL) MakeList (tree->front);}
    else{if (tree->front != NULL) MakeList (tree->front);
         AddToList (tree-> list_of_polygons);
```

```
        if (node->back != NULL) MakeList (tree->back);}
    return;}
```

Здесь функция ClassifyPoint() определяется положение наблюдателя относительно плоскости полигона находящегося в узле дерева. Функция AddToList() добавляет полигоны в список отображения в порядке приближения к наблюдателю. Впоследствии, передав полученный нами список в функцию, отображающую на экран сцену, все полигоны будут рисоваться именно в таком порядке, такой метод отображения называется back-to-front. Таким образом по разным направлениям будут отображаться сначала наиболее удаленные полигоны.

2 Реализация алгоритмов

2.1 Классы, используемые для решения задачи

Итак, поскольку минимальным элементом графики является точка, то необходимо иметь такой класс, который будет ее описывать. Точка в пространстве имеет три измерения, а соответственно и три координаты: x, y, z. А значит, ее необходимо описать следующим образом:

```
class Vertex3D {
    public:
    double x;
    double y;
    double z;
//…
}
```

Полигон мы можем задать набором вершин, его составляющих:

```
class _Polygon {
public:
    Vertex3D *polygon;
    int countOfTops;
    double koeficient
}
```

Для построения дерева нам понадобится класс, член-данными которого будут разделяющий полигон, список полигонов в узле дерева (на случай, если в одной плоскости будет лежать не один полигон, а несколько) и указатели на передний узел дерева и на узел позади. Для хранения списка полигонов и манипуляций с ними будем использовать шаблонный класс list.

```
class BSPTree {
public:
    _Polygon polygon; // полигон – часть делящей плоскости
    BSPTree *front; // указатель на передний узел
```

```
    BSPTree *back; // на задний узел
    std::list<_Polygon> list_polygons; //список полигонов в узле дерева
};
```

2.2 Реализация алгоритма художника

Алгоритм художника предполагает сортировку полигонов по некоторому признаку: либо минимальная координата z по всем вершинам многоугольника, либо среднее значение. Для записи этого признака в классе _Polygon есть поле koeficient. Для сортировки полигонов воспользуемся сортировкой Шелла.

Полученный список подается на вход функции, которая будет прорисовывать каждый полигон из списка. Так, например, на рисунке 2.1 показана работа программы при сцене из трех параллельных треугольников разного размера при работе любого из этих алгоритмов:

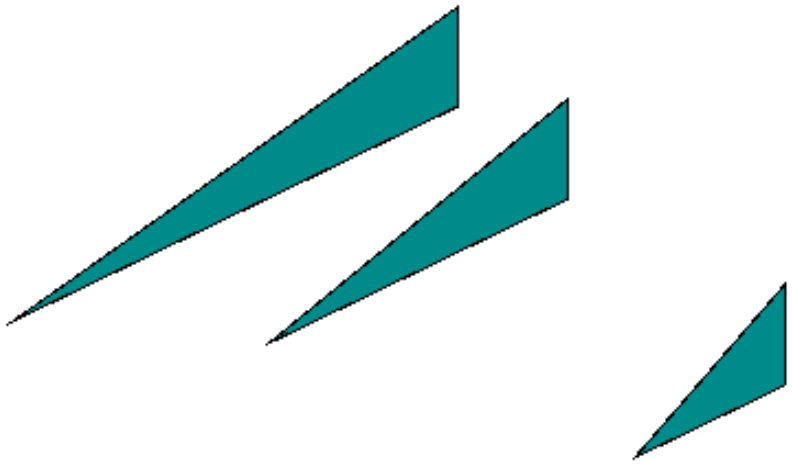


Рисунок 2.1 – Сцена из трах параллельных треугольников разного размера, полученная после работы алгоритма художника

Однако, если попытаться отсортировать и далее изобразить два пересекающихся треугольника, то сцена будет выглядеть неверно (Рисунок 2.2).

Одним из недостатков этого алгоритма является то, что перед каждой сортировкой всех полигонов необходимо приводить всю сцену к системе

координат наблюдателя, поскольку все полигоны сортируются относительно него. Это влияет на скорость обработки сцены, однако в этом есть и свои преимущества, если и сама сцена динамична, то данное решение будет более выгодным, в отличие от двоичного разбиения пространства.

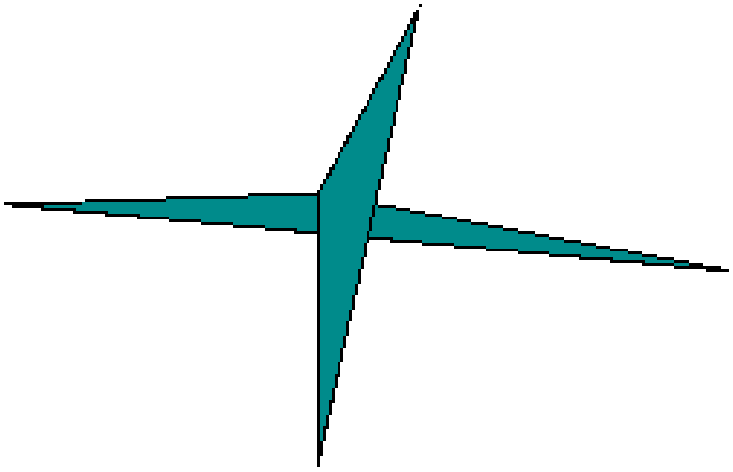


Рисунок 2.2 – Сцена из двух пересекающихся треугольников, полученная после работы алгоритма художника

Реализация обоих алгоритмов представлена в Приложении А.

2.3 Реализация двоичного разбиения пространства

Как выяснили выше, для построения BSP-дерева нам необходимо знать с какой стороны от плоскости-сплиттера будет находиться наблюдатель, как друг относительно друга располагаются полигоны, а также уметь делить полигоны на части, если они будут рассекаться сплиттером.

Обход дерева и занесение в список полигонов в необходимом нам порядке будем производить методом back-to-front, описанным ранее, реализация которого представлена в Приложении А.

2.3.1 Определение положения точки относительно плоскости

Из курса линейной алгебры известно, что плоскость может быть задана уравнением Ax+By+Cz+D=0, где A, B и C – координаты нормали к плоскости. Значит, нам необходимо узнать координаты нормали. Нормаль плоскости

будет совпадать с нормалью самого полигона, принадлежащего этой плоскости. Поскольку плоскость может быть построена вокруг трех точек, не лежащих на одной прямой, то, чтобы найти нормаль, необходимо взять три точки полигона, получить из них 2 вектора, с помощью которых, перемножив их векторно и нормализовав результат, получим нормаль.

Чтобы определить положение точки относительно плоскости заданной нормалью необходимо скалярно перемножить вектор нормали и вектор началом которого является точка наблюдения, а концом – точка, принадлежащая полигону.

По знаку скалярного произведения можно определить сзади, спереди или на плоскости находится точка, поскольку скалярное произведение

$$(\vec{n}, \vec{c}) = |\vec{n}||\vec{c}| * cos(\vec{n} \wedge \vec{c}), |\vec{n}| > 0, |\vec{c}| > 0,$$

если $(\vec{n}, \vec{c}) > 0$, тогда $cos(\vec{n} \wedge \vec{c}) > 0$, а из этого следует, что точка находится за плоскостью.

Если $(\vec{n},\vec{c})<0$, то и $\cos(\vec{n}\hat{}\vec{c})<0$ и точка находится перед плоскостью.

При $(\vec{n}, \vec{c})=0$ получим, что точка лежит на плоскости (Рисунок 2.3).

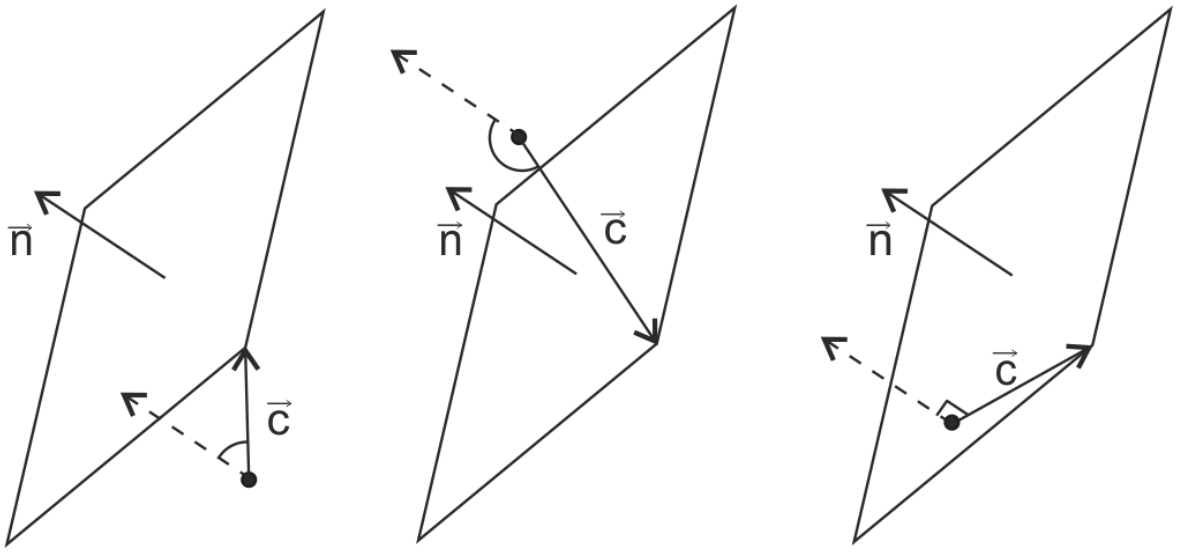


Рисунок 2.3 – Положение точки относительно плоскости

Таким образом реализация функции, определяющей местоположение точки относительно плоскости, будет выглядеть следующим образом

```
int ClassifyPoint(_Polygon & p, Vertex3D & a)
{
        double *vector = new double[3];
        vector[0] = p.polygon[0].x - a.x;
        vector[1] = p.polygon[0].y - a.y;
        vector[2] = p.polygon[0].z - a.z;
        double result = DotProduct(Normalize(CrossProduct(p.polygon[0],
        p.polygon[1], p.polygon[2])),vector);
        if (result < -0.000001) return 1;//точка лежит спереди плоскости
        if (result > 0.000001) return 2;//точка лежит позади плоскости
        return 0;//точка лежит на плоскости
}
```

Здесь DotProduct() – функция возвращающая значения скалярного произведения, CrossProduct() – функция возвращающая вектор полученный при векторном умножении, а Normalize() – нормализует вектор, который подается ей на вход.

2.3.2 Определение положения полигона относительно плоскости

Полигон задается точками в пространстве, все его точки лежат на одной плоскости. Для каждой точки полигона мы определим ее положение относительно плоскости, затем подсчитаем количество точек, расположенных перед, позади и на плоскости. Если все точки полигона лежат перед плоскостью и на ней, то полигон лежит перед плоскостью; если позади нее (и на ней) — то полигон позади плоскости; а если все точки лежат на плоскости — то и полигон принадлежит этой плоскости.

Однако, может возникнуть случай, когда точки лежат и перед плоскостью, и на ней, и позади, в таком случае полигон рассекается плоскостью. Реализация функции определения положения полигона относительно плоскости представлена в Приложении А.

2.3.3 Рассечение полигона плоскостью

Поскольку в условиях 3D мира может возникнуть ситуация, когда один полигон (или его плоскость) будет рассекать другой полигон, значит нам необходимо найти точки пересечения полигона и плоскости.

Поскольку многоугольник задается множеством точек, являющихся его вершинами, то нужно найти не все точки пересечения, а лишь те, в которых плоскость рассекает ребро полигона.

Рассмотрим задачу поиска точки пересечения отрезка и плоскости. Плоскость задана координатами вектора нормали, длина которого равна единице. Скалярное произведение векторов $\vec{x}$ и $\vec{y}$ можно определить не только как произведение длин векторов на косинус угла между ними, но также данной операции соответствует умножение длины вектора $\vec{x}$ на проекцию вектора $\vec{y}$ на направление вектора $\vec{x}$. А это значит, что при скалярном умножении векторах $\vec{x}$ на единичный вектор, получим проекцию этого вектора на направление единичного вектора.

Из рисунка 2.4 видно, что с помощью скалярного произведения вектора $\vec{n}$ и $\vec{x}$ можно найти длину вектора $\vec{y}$, поскольку $|\vec{y}|=|\vec{x}|*\cos(\phi)$.

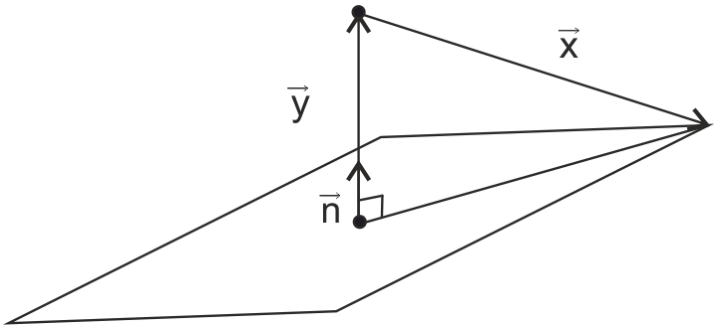


Рисунок 2.4

Рассмотрим рисунок 2.5, отрезок CV пересекает плоскость в точке x, и эта точка будет разбивать отрезок CV в таком же отношении, как относятся друг к другу отрезки CO и CP.

$$\overrightarrow{CO} = (\overrightarrow{CA}, \vec{n});$$

$$\overrightarrow{CP} = (\overrightarrow{CV},\vec{n}).$$

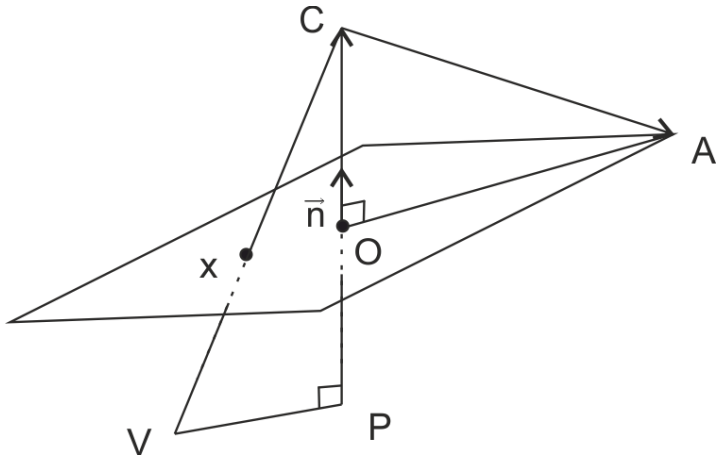


Рисунок 2.5 – Рассечение вектора $\overrightarrow{CV}$ плоскостью

Реализация функции рассечения полигона плоскостью представлена в Приложении А.

2.3.4 Выбор оптимальной разделяющей плоскости

Выбор оптимального сплиттера — сложный вопрос, который необходимо решить, поскольку, если использовать в качестве разделяющий плоскости первый в списке полигон, то глубина рекурсии может быть очень большой. Кроме того, принимая во внимание то, что на экран выводятся все многоугольники, но в определенном порядке, есть смысл в минимизации количества «распиливаний» полигонов плоскостью выбранного сплиттера, поскольку это увеличивает число полигонов в сцене, и соответственно полигонов, которые затем нужно отобразить.

В то же время возникает необходимость построения сбалансированного дерева. Сбалансированным называется дерево, у которого для каждой вершины выполняется требование: число вершин в левом и правом поддеревьях различается не более, чем на 1.

Общепринятым способом является последовательный проход по всем потенциальным сплиттер полигонам, и подсчете неких баллов.

Рассмотрим один из критериев выбора:

$$Score = abs(front - back)$$

В данном случае front — количество полигонов перед сплиттером, back — позади него, abs() – модуль числа внутри скобок. Исходя из данной формулы, сплиттер с наименьшим количеством очков признается наиболее оптимальным. Однако эта формула не учитывает, что сплиттер может «разрезать» полигоны, которые он пересекает. Чтобы понизить количество таких «распиливаний» и понизить увеличение количества полигонов в уровне, можно усовершенствовать формулу следующим образом:

$$Score = abs(front - back) + splits * w$$

Splits в этом примере — количество возможных распиливаний. A w в данном случае является коэффициентом «веса».

Однако и в этой формуле учитываются не все возможные расположения полигонов друг относительно друга. Полигоны могут лежать в одной плоскости со сплиттером. Добавим еще одно слагаемое:

$$Score = abs(front - back) + splits * w1 \pm coplanar,$$

где coplanar — количество полигонов. лежащих на одной плоскости с потенциальным сплиттером. Если использовать знак минус, сплиттер с большим количеством компланарных полигонов будет иметь больше шансов быть выбранным.

Обобщив предложенные формулы, получим:

$$Score = w_1 * abs(front - back) + w_2 * splits + w_3 * coplanar.$$

Реализация функции выбора оптимального сплиттера представлена в Приложении А.

2.3.5 Построение BSP-дерева

Используя класс BSPTree и описанный выше алгоритм построения дерева опишем функцию Build_BSPTree() следующим образом:

```
BSPTree* Build_BSPTree(BSPTree * tree, std::list<_Polygon> &polygons)
    {
        _Polygon p = SelectPolygon(polygons); //выбираем сплиттер
```

```
		tree->polygon = p;
		tree->list_polygons.push_back(tree->polygon);
		std::list<_Polygon> front_list, back_list;
		std::list<_Polygon>::iterator ipolygons;
		for (ipolygons = polygons.begin(); ipolygons != polygons.end();
ipolygons++) {
			if (*ipolygons != tree->polygon)
			{
			switch (ClassifyPolygon(*ipolygons, tree->polygon))
			{
			case 0: tree->list_polygons.push_back(*ipolygons);
				break;
			case 1: front_list.push_back(*ipolygons);
				break;
			case 2: back_list.push_back(*ipolygons);
				break;
			case 3:
				{
				_Polygon new1, new2;
				Split_Polygon(*ipolygons, tree->polygon, new1, new2);
				if (ClassifyPolygon(new1, tree->polygon) == 1 &&
				ClassifyPolygon(new2, tree->polygon)== 2)
				{
					front_list.push_back(new1);
					back_list.push_back(new2);
				}
				else {
					front_list.push_back(new2);
					back_list.push_back(new1);
					}
```

```
                }
                        break;
                }
            }
        }
        if (!front_list.empty())
            tree->front = Build_BSPTree(tree->front, front_list);
        if (!back_list.empty())
            tree->back = Build_BSPTree(tree->back, back_list);
        return tree;
}
```

Далее обойдя полученное дерево методом back-to-front, получив список полигонов, подав его на вход функции прорисовывающей всю сцену получим, к примеру, три параллельных прямоугольника изображенных на рисунке 2.6.

Главным преимуществом данного алгоритма является то, что после построения дерева, если сцена не меняется, то при изменении местоположения наблюдателя все дерево перестраивать не нужно, необходимо лишь заново его обойти, чтобы узнать порядок полигонов.

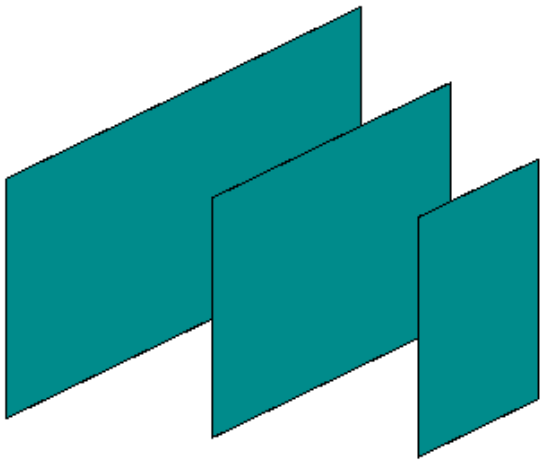


Рисунок 2.6 – Сцена из трех параллельных прямоугольников после работы алгоритма двоичного разбиения пространства

Кроме того, за счет функции рассечения полигонов изображение сцены с пересекающимися телами не представляет никакой трудности и будет выглядеть так, как необходимо. Рассмотрим такие же два пересекающихся треугольника как в алгоритме художника, работа алгоритма изображена на рисунке 2.7.

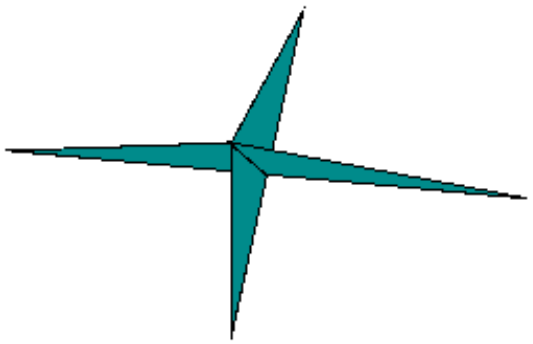


Рисунок 2.7 – Сцена из двух пересекающихся треугольников после работы алгоритма двоичного разбиения пространства

3 Сравнительный анализ алгоритмов

3.1 Сравнение алгоритмов по качеству полученных изображений

3.1.1 Алгоритм художника

В ходе работы были рассмотрены две модификации алгоритма художника и выяснилось, что у каждого из них получаются свои результаты, различающиеся точностью и качеством.

Так, например, даже при изображении такой элементарной фигуры как тетраэдр получаются разные результаты, причем качество изображения после сортировки по минимальной координате z хуже нежели после сортировки по среднему значению координаты z, результат изображен на рисунке 3.1, слева сортировка по минимальной координате z, справа – по среднему значению координаты z.

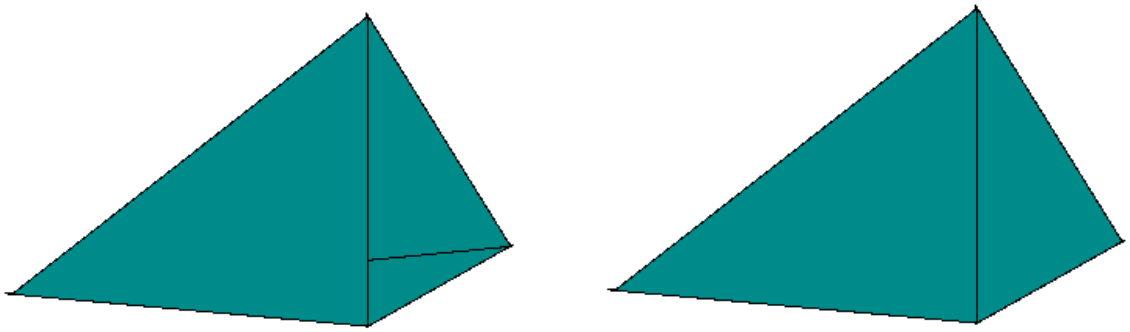


Рисунок 3.1 – Сцена из одного тетраэдра после работы алгоритма художника, слева – сортировка по минимальной координате z, справа – по среднему значению координаты z

Кроме этого выше было в разделе 2.2 было замечено, что при взаимном перекрытии полигонов оба алгоритма работают не корректно и результат не соответствует действительности, на рисунке 3.2 слева направо изображены результаты работы алгоритмов сортировки по минимальной координате z и по среднему значению координаты z и правильный вариант соответственно.

Рассмотрим несколько другую сцену, в которой также есть перекрытие одного полигона другим (Рисунок 3.3). Тут один из алгоритмов (работа

которого изображена справа), а именно сортировка по среднему значению координаты z, будет работать корректно, а второй (изображен слева) нет.

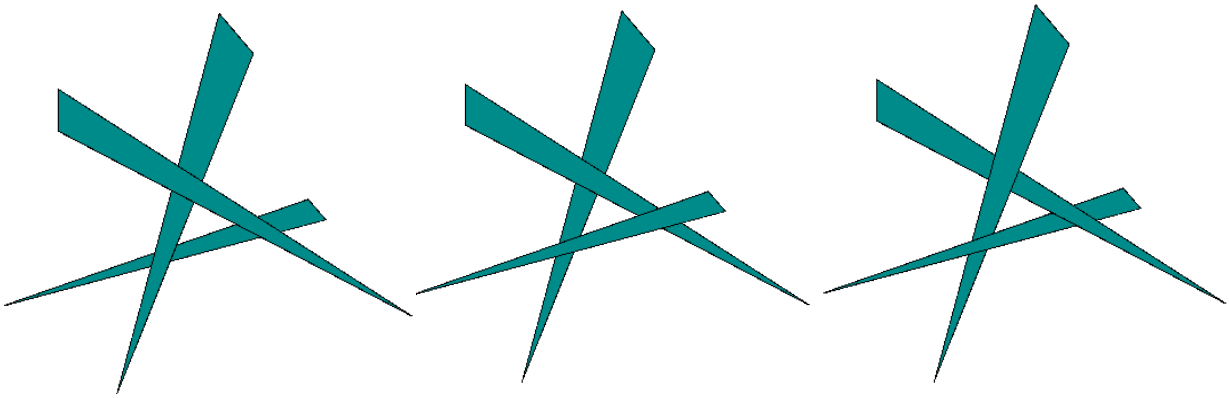


Рисунок 3.2 – Сцена со взаимным перекрытием полигонов

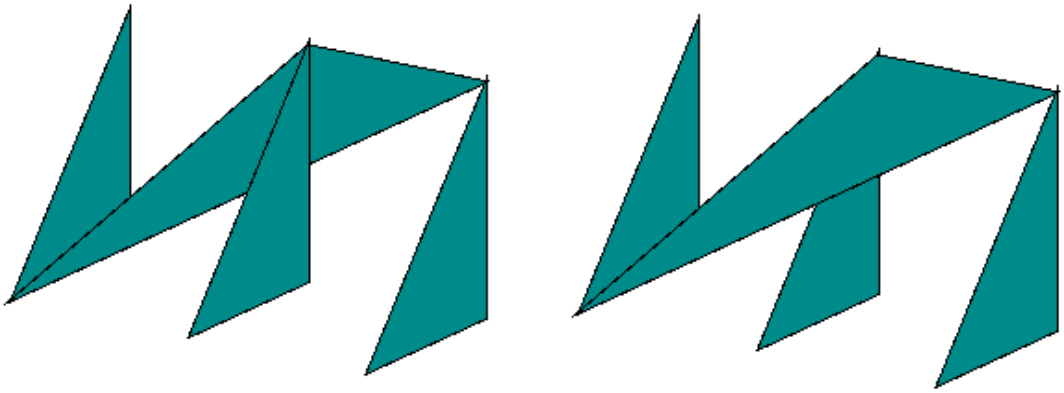


Рисунок 3.3 – Сцена с перекрывающимися полигонами

Кроме этого если начать изменять координаты наблюдателя, то и правильно работающий алгоритм начнет работать некорректно, это обусловлено тем, что, при перемещении точки зрения, меняются коэффициенты плоскостей и под некоторыми углами нет того фактора, который вызывает неправильное отображение сцены. Результат работы отражен на рисунке 3.4.

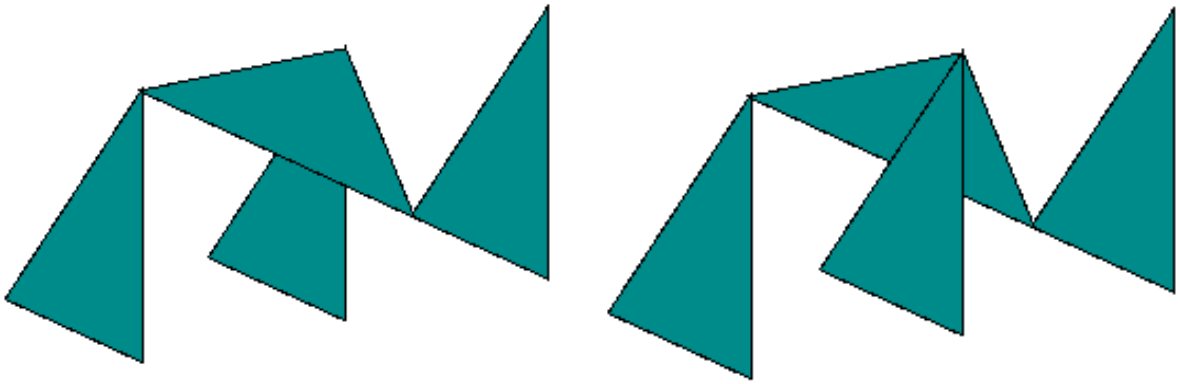


Рисунок 3.4 – Сцена с перекрывающимися полигонами при изменении точки зрения, слева – результат работы алгоритма, справа – ожидаемый результат

Рассмотрим результаты работы алгоритмов не только на простых фигурах, но и на более сложных, таких как конусы и цилиндры, оба алгоритма работают корректно с такими фигурами. На рисунке 3.5 имеем результат работы алгоритма художника, здесь изображены цилиндры, аппроксимированные 8, 16 и 32 прямоугольниками.

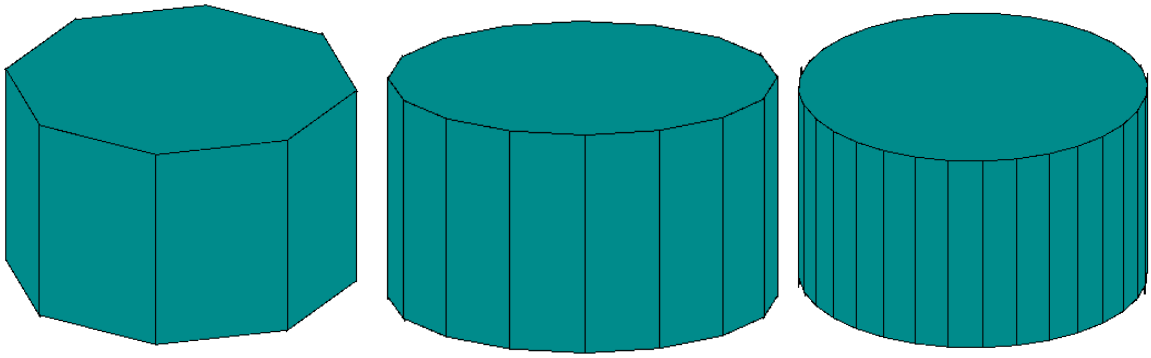


Рисунок 3.5 – Цилиндры, аппроксимированные 8, 16, 32 прямоугольниками после работы алгоритма художника

3.1.2 Двоичное разбиение пространства

Реализованный алгоритм построения BSP-дерева, а также последующий его обход методом back-to-front имеет множество сильных

сторон. Главное его преимущество состоит в том, что изображения, независимо от количества и формы объектов, всегда получаются необходимой точности. Так, например, Тетраэдр, построенный с помощью двоичного разбиения пространства, изображенный на рисунке 3.6, соответствует действительности и отображается корректно.

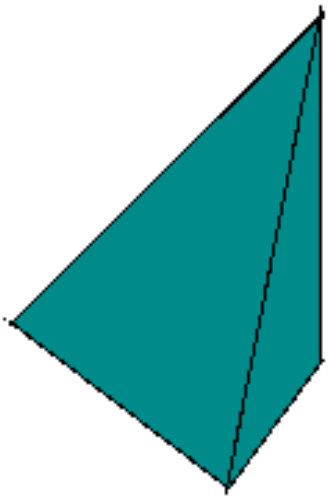


Рисунок 3.6 – Сцена с тетраэдром после работы алгоритма двоичного разбиения пространства

В процессе реализации алгоритма была определена функция рассечения полигонов. Это является преимуществом этого алгоритма, поскольку теперь можно обрабатывать сцены, содержащие в себе пересечения тел и отдельных полигонов. Одна из таких сцен показана на рисунке 3.7, здесь видны места, где полигоны рассекались, что является неэстетичным, однако точность и правильность оправдывают этот недостаток.

Для определения положения точки относительно полигона была необходимость использовать координаты нормали к плоскости. Здесь возникает проблема перечисления вершин полигона в определенном порядке (всегда по часовой стрелке или всегда против). От этого будет зависеть направление нормалей плоскостей, а от их направления будет зависеть спереди или сзади плоскости находится точка, так, если задать неправильным

образом полигоны в тетраэдре, то получим неверное изображение, показанное на рисунке 3.8.

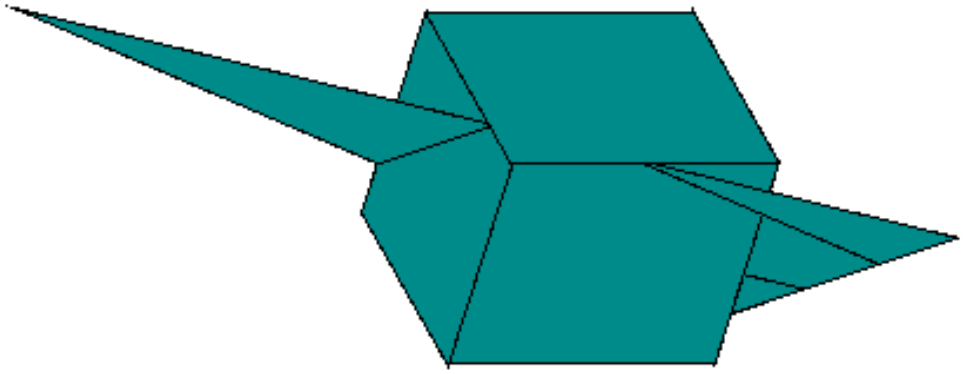


Рисунок 3.7 – Сцена с пересечением треугольника и куба после работы алгоритма двоичного разбиения пространства

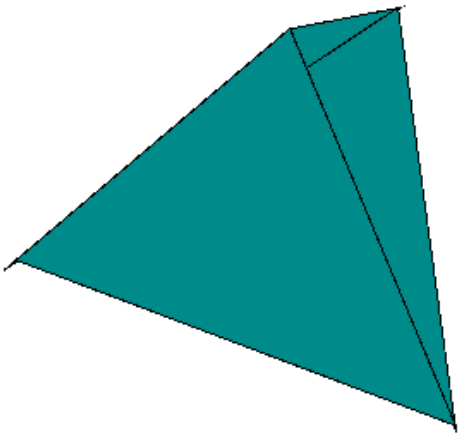


Рисунок 3.8 – Сцена с тетраэдром, вершины каждого из треугольников которого перечислены в неправильном порядке

Теперь рассмотрим гладкие фигуры, в нашем случае цилиндр, также, как и в алгоритме художника, аппроксимируем цилиндр 8, 16 и 32 прямоугольниками (Рисунок 3.9).

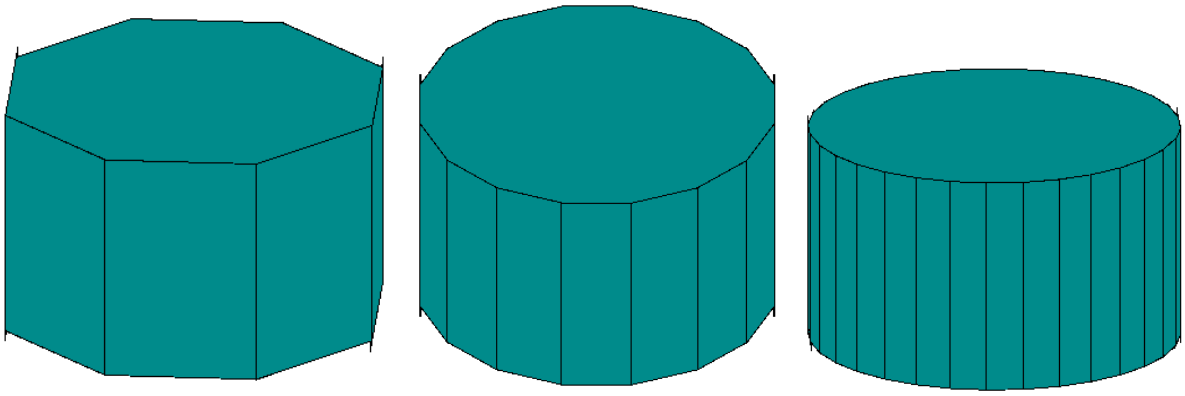


Рисунок 3.9 – Цилиндры, аппроксимированные 8, 16, 32 прямоугольниками после работы алгоритма художника

3.1.3 Вывод

Таким образом, можно сделать заключение о том, что алгоритм двоичного разбиения пространства качественно воспроизводит сцены, в которых есть пересечения тел и полигонов.

Однако существует проблема в корректном отображении тел, в которых полигоны заданы неправильным образом.

Алгоритм художника не имеет проблемы с заданием полигонов и отображением простых сцен, например, с помощью сортировки по среднему значению координаты z, однако некорректное отображение взаимно перекрывающих друг друга полигонов делает неудобным использование этого алгоритма.

3.2 Сравнение алгоритмов по времени выполнения

Для сравнения алгоритмов можно воспользоваться такими критериями как время выполнения и правильность сортировки. В разделе 4 были описаны все преимущества и недостатки алгоритмов в плане качества и правильности сортировки.

Сравним эти алгоритмы по времени сортировки. Рассмотрим большую сцену с четырьмя цилиндрами, каждый из которых аппроксимирован 60 прямоугольниками. Результаты времени работы приведены в таблице 1.

Таблица №1 – Результаты сравнения времени работы алгоритмов на сложной сцене

Двоичное разбиение пространства		**Алгоритм художника**	
Время построения дерева, сек.	4,097	Время сортировки, сек.	0,012
Время обхода дерева, сек.	0,002		
Время прорисовки, сек.	0,377	Время прорисовки сцены, сек.	0,328

Из таблицы №1 видно, что построение дерева занимает много времени, однако обход его занимает гораздо меньше времени, нежели сортировка в алгоритме художника. Значит можно сделать вывод о том, что для статичных сцен, для которых можно заранее построить дерево, двоичное разбиение пространства - лучший из этих двух вариантов. Это обусловлено тем, что при перемещении точки зрения нет необходимости перестраивать все дерево, нужно просто произвести новый обход. В противоположность BSP деревьям алгоритм художника требует полной пересортировки полигонов при изменении положения наблюдателя.

ЗАКЛЮЧЕНИЕ

В данной работе были реализованы два алгоритма для построения списка приоритетов полигонов на сцене: алгоритм художника и двоичного разбиения пространства. Были решены главные проблемы алгоритма двоичного разбиения пространства, такие как выбор оптимальной разделяющей плоскости, определения положения точки и полигона относительно плоскости, а также деление полигона плоскостью.

В процессе реализации этих алгоритмов обнаружены достоинства и недостатки каждого из них, проведен сравнительный анализ как по качеству изображения сцены, так и по времени работы алгоритма. Установлено, что точность и правильность сцены присуща двоичному разбиению пространства, за исключением некоторых особенностей. Для небольших простых, но изменяющихся сцен алгоритм художника выигрывает по времени реализации, а для статичных сцен имеет смысл использовать BSP-дерево, которое строится один раз для всей сцены, а далее проводится лишь процедура обхода методом back-to-front.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. КОМПЬЮТЕРНАЯ ГРАФИКА. А.Ю. Дёмин, А.В. Кудинов [Электронный ресурс] // URL: http://compgraph.tpu.ru/index.html (дата обращения: 21.01.2017)
2. Машинная графика. Удаление невидимых линий и поверхностей. [Электронный ресурс] // URL: https://www.bsu.by/Cache/Page/353833.pdf (дата обращения: 21.01.2017)
3. Удаление невидимых граней и методы оптимизации [Электронный ресурс] // URL: http://shackmaster.fdd5-25.net/optimize.htm (дата обращения: 07.02.107)
4. Удаление невидимых поверхностей. Алгоритм художника. [Электронный ресурс] // URL: http://www.opita.net/node/54 (дата обращения: 13.02.2017)
5. Удаление невидимых частей. Алгоритм художника. [Электронный ресурс] // URL: http://algolist.manual.ru/graphics/3dfaq/articles/32.php (дата обращения: 13.02.2017)
6. BSP-дерево и способы его применения в трехмерной графике [Электронный ресурс] // URL: http://eways.narod.ru/HomePage/3d/bsp_tree.htm (дата обращения: 04.04.2017)
7. Введение в BSP деревья или BSP для самых «маленьких». Часть первая, теоретическая. [Электронный ресурс] // URL: http://www.gamedev.ru/code/articles/BSP (дата обращения: 23.04.2017)
8. Andrew Steven – AN INVESTIGATION INTO REAL-TIME 3D POLYGON RENDERING USING BSP TREES. – 1999.
9. Пересечение отрезка с плоскостью. [Электронный ресурс] // URL: http://programmingcpp.narod.ru/otrezok.htm (дата обращения: 10.05.2017)

ПРИЛОЖЕНИЕ А

Листинг программы

1 Функция сортировки полигонов по минимальному значению координаты z

```
void minSortTriangle(_Polygon *T, int size) {
	for (int i = 0; i < size; i++) {
			T[i].koeficient = DBL_MAX;
			for (int j = 0; j < T[i].countOfTops; j++)
				if (T[i].koeficient > T[i].polygon[j].z)
					T[i].koeficient = T[i].polygon[j].z;
		}
	ShellSort(T,size);
}
```

2 Функция сортировки полигонов по среднему значению z

```
void avgSortTriangle( _Polygon * T, int size){
	for (int i = 0; i < size; i++) {
			for (int j = 0; j < T[i].countOfTops; j++)
				T[i].koeficient = T[i].koeficient + T[i].polygon[j].z;
			T[i].koeficient = T[i].koeficient / T[i].countOfTops;
		}
	ShellSort(T,size);
}
```

3 Функция определения положения полигона относительно плоскости

```
int ClassifyPolygon(Polygon & p,_Polygon & p1){
	int countFront = 0, countBack = 0, countOn = 0;
	for (int i = 0; i < p.countOfTops; i++) {
		int res = ClassifyPoint(p1, p.polygon[i]);
```

```
            switch (res){
                case 1: countFront++; break;
                case 2: countBack++;      break;
                case 0: countOn++; break;
            }
        }
        if (countFront == p.countOfTops || (countFront + countOn) ==
p.countOfTops)
            return 1;//front
        if (countBack == p.countOfTops || (countBack + countOn) ==
p.countOfTops)
            return 2;//back
        if (countOn == p.countOfTops)
            return 0; //on plane
        return 3;//cross
}
```

4 Функция рассечения полигона плоскостью

```
void Split_Polygon(_Polygon & devide,_Polygon & main,_Polygon &
new1,_Polygon &new2){
        std::list<Vertex3D> front;
        std::list<Vertex3D> back;
        Vertex3D a, b;
        for (int i = 0; i < devide.countOfTops; i++) {
            if (i < devide.countOfTops - 1) {
                a = devide.polygon[i];
                b = devide.polygon[i + 1];
            }
            else {
                a = devide.polygon[i];
```

```
                b = devide.polygon[0];
            }
            if (ClassifyPoint(main, a) == 1) {
                front.push_back(a);
                if (ClassifyPoint(main, b) == 2) {
                    Vertex3D d = IntersectionPoint(a, b, main);
                    front.push_back(d);
                    back.push_back(d);
                }
                if (ClassifyPoint(main, b) == 0) {
                    front.push_back(b);
                    back.push_back(b);
                }
            }
            if (ClassifyPoint(main, a) == 2) {
                back.push_back(a);
                if (ClassifyPoint(main, b) == 1) {
                    Vertex3D d = IntersectionPoint(a, b, main);
                    front.push_back(d);
                    back.push_back(d);
                }
                if (ClassifyPoint(main, b) == 0) {
                    front.push_back(b);
                    back.push_back(b);
                }
            }
        }
        _Polygon newp1(front.size()), newp2(back.size());
        int i = 0;
        for (std::list<Vertex3D>::iterator ifront = front.begin(); ifront !=
front.end(); ifront++)
```

```
            newp1.polygon[i++] = *ifront;
        i = 0;
        for (std::list<Vertex3D>::iterator iback = back.begin(); iback !=
back.end(); iback++)
            newp2.polygon[i++] = *iback;
        new1 = newp1;
        new2 = newp2;
}
```

5 Функция определения точки пересечения луча и плоскости

```
Vertex3D IntersectionPoint(Vertex3D &startPoint, Vertex3D&endPoint,
_Polygon &devider) {
        Vertex3D iPoint;
        double *normal = new double[3]/*вектор нормали*/,
        *vector1 = new double[3]/*вектор из начала луча в вершину
полигона*/,
        *vector2 = new double[3]/*вектор луча start-end*/,
        k/*коэфициент приближения к плоскости*/;
        normal = CrossProduct(devider.polygon[0], devider.polygon[1],
devider.polygon[2]);
        vector1[0] = devider.polygon[0].x - startPoint.x;
        vector1[1] = devider.polygon[0].y - startPoint.y;
        vector1[2] = devider.polygon[0].z - startPoint.z;
        vector2[0] = endPoint.x - startPoint.x;
        vector2[1] = endPoint.y - startPoint.y;
        vector2[2] = endPoint.z - startPoint.z;
        k = DotProduct(normal, vector1) / DotProduct(normal, vector2);
        iPoint.x = startPoint.x + vector2[0] * k;
        iPoint.y = startPoint.y + vector2[1] * k;
        iPoint.z = startPoint.z + vector2[2] * k;
        return iPoint;
```

```
}
```

6 Функция выбора оптимального сплиттера

```
_Polygon& SelectPolygon(std::list<_Polygon> &masp) {
	double min = DBL_MAX;
	std::list < _Polygon > ::iterator memory;
	for (std::list < _Polygon > ::iterator i = masp.begin(); i != masp.end();
i++){
	int countFront = 0, countBack = 0, countOnPolygon = 0,
	countCrossPolygon = 0;
	for (std::list < _Polygon > ::iterator j = masp.begin(); j != masp.end();
j++){
		if (i != j) {
			int res = ClassifyPolygon(*j, *i);
			switch (res){
			case 1: countFront++;	break;
			case 2: countBack++; break;
			case 0: countOnPolygon++; break;
			case 3: countCrossPolygon++; break;
			}
		}
	}
	  double tmp;
	  tmp = abs(countFront - countBack) + (countCrossPolygon * 8) -
countOnPolygon;
	  if (tmp < min) {
		min = tmp;
		memory = i;
		}
	  }
}
```

(/index.php/ru/) (/index.php/en/)

Поиск заимствований в научных текстах$^{\beta}$

Введите текст:

...или загрузите файл:

Файл не выбран... **Выбрать файл...**

Укажите год публикации: 2017

Выберите коллекции

- Все
- Рефераты
- Авторефераты
- Иностранные конференции
- PubMed
- Википедия
- Российские конференции
- Иностранные журналы
- Российские журналы
- Энциклопедии
- Англоязычная википедия

Анализировать

Проверить по расширенному списку коллекций системы Руконтекст (http://text.rucont.ru/like)

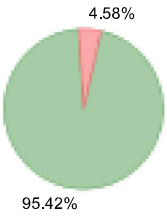
Обработан файл:
ВКР.docx.

Год публикации: 2017.

Оценка оригинальности документа - 95.42%

Процент условно корректных заимствований - 0.0%

Процент некорректных заимствований - 4.58%



Просмотр заимствований в документе

Время выполнения: 19 с.

Документы из базы

Источники заимствования	В списке литературы	Источники Заимствования
1. Реферат: Трёхмерная компьютерная графика (http://www.bestreferat.ru/files/97/bestreferat-30697.docx) Год публикации: 2016. Тип публикации: реферат. http://www.bestreferat.ru/files/97/bestreferat-30697.docx (http://www.bestreferat.ru/files/97/bestreferat-30697.docx) Показать заимствования (11)		4.34%
2. Трехмерная компьютерная графика (http://limej.ru/index.php/home/136-stat/47391-Trehmernaya_komp_yuternaya_grafika.html) Год публикации: 2016. Тип публикации: реферат. http://limej.ru/index.php/home/136-stat/47391-Trehmernaya_komp_yuternaya_grafika.html (http://limej.ru/index.php/home/136-stat/47391-Trehmernaya_komp_yuternaya_grafika.html) Показать заимствования (11)		4.34%

3. Трёхмерная компьютерная графика (http://limej.ru/index.php/home/136-stat/30105-Tryohmernaya_komp_yuternaya_grafika.html)

Год публикации: 2016. Тип публикации: реферат. 4.15%
http://limej.ru/index.php/home/136-stat/30105-Tryohmernaya_komp_yuternaya_grafika.html
(http://limej.ru/index.php/home/136-stat/30105-Tryohmernaya_komp_yuternaya_grafika.html)
Показать заимствования (10)

4. Учебное пособие: Методические указания по выполнению курсовой работы по дисциплине «машинная графика» Москва 1995 (http://www.bestreferat.ru/files/08/bestreferat-411708.docx)

Год публикации: 2016. Тип публикации: реферат. 2.47%
http://www.bestreferat.ru/files/08/bestreferat-411708.docx
(http://www.bestreferat.ru/files/08/bestreferat-411708.docx)
Показать заимствования (7)

Дополнительно

Значимые оригинальные фрагменты

Библиографические ссылки

Искать в Интернете