Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский Авиационный Институт»
(Национальный Исследовательский Университет)

Факультет №8 «Компьютерные науки и прикладная математика»
Кафедра 805 «Прикладная математика»

Реферат
по курсу «Вычислительные системы»
2 семестр

Тема:

Расширение графических возможностей 3D движка на Java.

Автор работы:
студент 1 курса, гр. М8О-103Б-21
Фадеев Д.В.
Руководитель проекта:
Севастьянов В.С.
Дата сдачи:

Москва 2022 г

Содержание:

Введение.

В реферативной работе 1 семестра мною была создана программа, в которой можно было по координатам задавать трёхмерные вершины полигонов, из которых можно было, как конструктор, собирать объёмные модели. Тогда внутри программы было реализовано ядро, позволяющее производить все математические операции, чтобы правильно отображать на экране компьютера вращение модели.

Но итоговую программу нельзя было назвать идеальной, хоть математически вращение объекта происходило корректно, человеческому глазу было сложно понять как это вращение происходит, в какую сторону поворачивается модель. А корнем этой проблемы является человеческая биология. Мы привыкли видеть объекты искажёнными - чем дальше объект, тем он меньше. В имеющемся ядре строилась только изометрическая проекция объекта, без каких либо искажений. Значит необходимо добавить отображение объектов с учётом перспективной проекции.

Кроме того, большое влияние на ощущение объекта и преобразований над ним оказывает свойства его поверхности и освещение, которое падает на эту поверхность. При каркасном отображении объектов мы видим только линии, соединяющие вершины полигона. Решение этой проблемы может стать «заливка» полигона определённым цветом, придавая ему вид сплошной поверхности и последующий его рендеринг с созданием примитивного освещения.

Процесс 3D рендеринга очень ресурснозатратный, на аппаратные средства компьютера оказывается серьёзная нагрузка. В моём случае эта нагрузка не ощутима, потому что я реализую достаточно примитивный тип 3D движка, без сложных алгоритмов освещения, текстурирования, расчёта коллизий и т.п. Но будет отличным опытом отказаться от встроенных инструментов Java рисования и написать собственные функции растеризации. Тогда я смогу контролировать работу своей программы буквально попиксельно, потому что я же эти пиксели и буду просчитывать.

Таким образом, учитывая всё что было описано ранее, составляется не большой, но очень общий список всех задач, которые нужно решить в работе:

1. Отображение сцены с объёмными моделями, которые рендерятся с учётом настраиваемой перспективы.
2. Написание собственной функции растеризации прямых линий.
3. Растеризация полигонов модели.
4. Создание простейшего освещения сцены.

Начнём разбираться в математическом аппарате каждой задачи.

Перспективная проекция.

Наш мозг привык получать из глаз искажённое изображение, называемое перспективной проекций. Она позволяет ощущать расстояние до объекта, его размеры, а также общую глубину картинки. И наш мозг создал настолько хорошо и чётко работающие алгоритмы анализа изображения, что если ему дать объект в нестандартной проекции, то он потратит некоторое количество времени на осознание происходящего.

Существует один способ создания перспективной проекции, который широко используется в различных 3D движках, например OpenGL. Разберём мы этот метод в упрощённом варианте. Его суть довольно проста. Предположим у нас имеется некоторый отрезок CB, находящийся на расстоянии AB от наблюдателя. Мы предположим, что наблюдатель видит всю плоскость F, находящуюся от него на расстоянии AB1. Тогда перспективной проекцией отрезка CB (рис.1) будет отрезок C1B1, заключенный между AB и AC . Если бы CB был бы ближе к наблюдателю, то тогда и его проекция C1B1 была бы больше, т.е. свойства проекции выполняются.

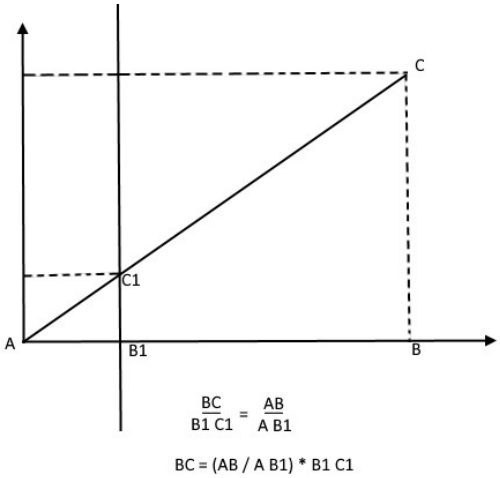

Рис.1

Найти это самое C1B1 очень просто, и школьник с этим справится, используя правило подобных треугольников: $B1C1=(\frac{AB1}{AB})\cdot BC$

В ядре программы алгоритм расчета проекции немного сложнее. Сначала нужно перевести все вершины в локальные координаты камеры. Затем уже находим проекцию вершин на плоскость (рис.2).

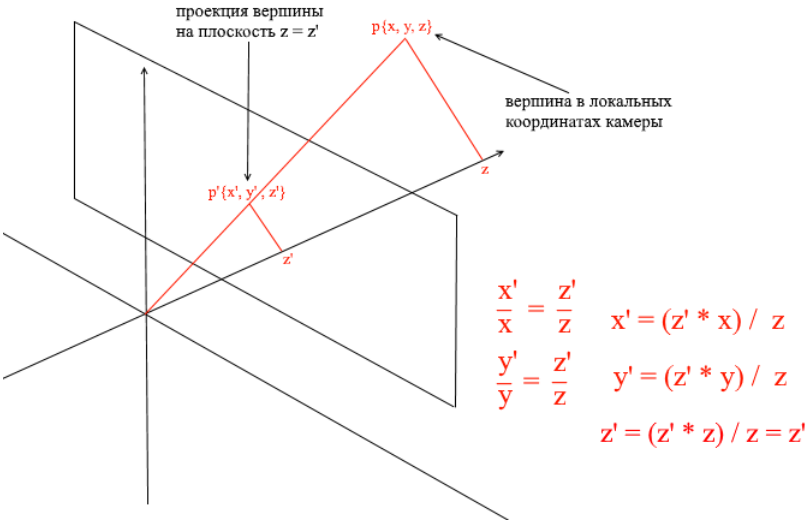

Рис.2

Здесь нам понадобятся только координаты проекции x` и y`, отвечающее за то, как на мониторе компьютера мы будем видеть вершину.

Для реализации в программе этого алгоритма, напишем отдельный класс камеры (рис. 3), которая будет высчитывать проекцию. В дальнейшем мы воспользуемся ею для растеризации и создания освещения.

```
public class Camera {
    double x;
    double y;
    double z;
    double h; //distance from coords
    double f; //distance of projection plane
    double xz = 0, yz = 0, xy = 0;
    List<Polygon> polygons;
    double[][] zBuffer;
```

Рис. 3

Переменные x, y, z - координаты точки, на которую всегда смотрит камера при вращении. *h* - расстояние от этой точки до камеры. Чтобы понять лучше как это работает, представим себе палку, один конец которой свободно вращается на шарнире, имеющий некоторые координаты в пространстве, а на другом конце находится камера, смотрящая вдоль этой палки на шарнир. *f* - расстояние от плоскости проекции до «глаза» камеры. На рисунке 1 переменной *f* соответствует AB1, а *h* — AB. Или же на рисунке 2, где *f* — z`, а *h* — z. Лист *polygons* хранит в себе все полигоны, которые существуют в сцене. Остальные переменные понадобятся в следующих разделах.

Функция perspective (рис. 4) вычисляет и возвращает перспективную проекцию для переданной вершины. В первых 4-ёх строках вершина переводится в локальные координаты камеры. После чего матрицей вращения *transform* вершина поворачивается на угол, под которым на неё смотрит камера. В полях trans.x и trans.y происходит вычисление тех самых x` и y`, которые потом мы увидим на экране компьютера.

```
Vertex perspective(Vertex v, Matrix transform) {
    Vertex trans = new Vertex(v.x, v.y, v.z);
    trans.x += x;
    trans.y += y;
    trans.z += z;
    trans = transform.transform(trans);
    trans.x = (trans.x * f) / (trans.z + h);
    trans.y = (trans.y * f) / (trans.z + h);
    return trans;
}
```

Рис. 4

Теперь всё готово для визуализации простого кубика (рис. 5 и 6).

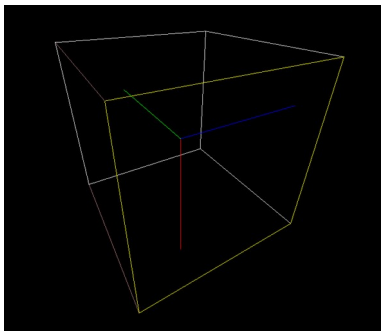

Рис. 5

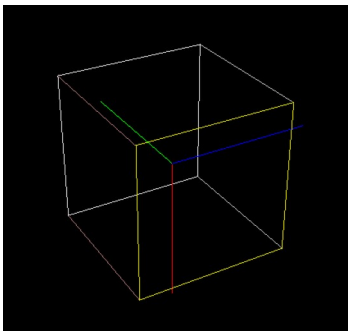

Рис. 6

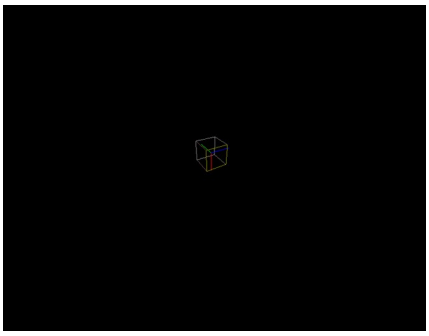

Рис. 7

Как видите, играясь со значениями расстояния до камеры *h*, мы получаем разное изображение с разным искажением рёбер. Хоть и на рисунках 5 и 6 кубик имеет одинаковый размер, в реальности в программе (рис. 7) кубик из рис. 6 имеет очень небольшой размер, который был смаштабирован переменной *f*.

Растеризация линий.

В реферативной работе 1-ого семестра я активно использовал графические инструменты JDK, в которые входило и рисование линий. Использование этого метода значительно упростило написание ядра графического движка, но чтобы улучшить его в этой работе, расширить функционал, необходимо отказаться от этого инструмента.

Связано это с тем, что в процессе самостоятельной растеризации линии мы можем высчитывать «глубину» каждого пикселя, и, если этот пиксель будет позади пикселя другого ребра или полигона, то он отображаться не будет. Таким образом возможно контролировать удаление скрытых поверхностей из итоговой картинки.

Другой значительный плюс самостоятельной растеризации будет раскрыт в следующем разделе.

В качестве алгоритма растеризации я выбрал алгоритм Брезенхэма:

5. Высчитываем Δx и Δy (разность координат) между двумя вершинами и выбираем преобладающую Δ, которая больше своего оппонента.
6. Находим количество n целочисленных значений между наиболее растущими координатами (например Δx > Δy, тогда $n = x_1 - x_2$) и находим изменение не преобладающей координаты при изменении преобладающей на единицу (пример $dy = (y_1 - y_2) / n$).
7. Проходясь по всем целочисленным значениям преобладающей координаты в цикле (i) меняем значение не преобладающей j на dj (например y = y + *dy*) после чего закрашиваем пиксель <i, j> в нужный цвет.

Для вычисления покрытия одного пикселя другим заведём буферный двумерный массив *zBuffer* (та самая еще одна переменная из рис. 3) имеющий размерность окна программы. В нём будут храниться расстояния до каждого пикселя, цвет которого отображается на экране. Если расстояние от пикселя *A* меньше, чем у *B*, то на экране физический пиксель окрасится в цвет пикселя *A*. Расчёт расстояния до пикселя такой же, как изменение не пре-

обладающей координаты в алгоритме Брезенхэма. Весь алгоритм описан в методе drawLine(*v1, v2, c, img, border*) (рис. 8). Вершины *v1* и *v2* нам необходимо соединить линией цвета *c*, *img* - экземпляр BufferedImg, пиксели которого мы будем раскрашивать. Лист *border* понадобится позже при растеризации полигонов.

```
void drawLine(Vertex v1, Vertex v2, Color c, BufferedImage img, List<double[]> border) {
    int dx = (int) (v2.x - v1.x);
    int dy = (int) (v2.y - v1.y);
    if (Math.abs(dx) > Math.abs(dy)) {
        //для более горизонтальных прямых
        int x1 = (int) Math.floor(v1.x);
        int x2 = (int) Math.floor(v2.x);
        if (x1 > x2) {
            int a = x1;
            x1 = x2;
            x2 = a;
            Vertex b = v1;
            v1 = v2;
            v2 = b;

        }
        double d = (v2.y - v1.y) / (x2 - x1);
        double y = v1.y;
        double dz = (v2.z - v1.z) / (x2 - x1);
        double z = v1.z;
        int lastx = x1 + img.getWidth() / 2;
        for (int i = x1 + img.getWidth() / 2; i <= x2 + img.getWidth() / 2; i++) {
            int pixelY = (int) Math.floor(y + img.getHeight() / 2);
            if (i < img.getWidth() && i > 0 && pixelY < img.getHeight() && pixelY > 0) {
                if (z < zBuffer[i][pixelY]) {
                    zBuffer[i][pixelY] = z;
                    img.setRGB(i, pixelY, c.getRGB());
                }
            }
            if (lastx != i) {
                border.add(new double[]{i - img.getWidth() / 2, y, z});
                lastx = i;
            }



            y += d;
            z += dz;
        }
    } else {
        //для более вертикальных
        int y1 = (int) Math.floor(v1.y);
        int y2 = (int) Math.floor(v2.y);
        if (y1 > y2) {
            int a = y1;
            y1 = y2;
            y2 = a;
            Vertex b = v1;
            v1 = v2;
            v2 = b;

        }
        double d = (v2.x - v1.x) / (y2 - y1);
        ;
        double x = v1.x;
        double dz = (v2.z - v1.z) / (y2 - y1);
        double z = v1.z;
        int lastx = (int) Math.floor(x + img.getWidth() / 2);
        for (int i = y1 + img.getHeight() / 2; i <= y2 + img.getHeight() / 2; i++) {
            int pixelX = (int) Math.floor(x + img.getWidth() / 2);
            if (i < img.getHeight() && i > 0 && pixelX < img.getWidth() && pixelX > 0) {
                if (z < zBuffer[pixelX][i]) {
                    zBuffer[pixelX][i] = z;
                    img.setRGB(pixelX, i, c.getRGB());
                }
            }
            if (lastx != pixelX) {
                lastx = pixelX;
                border.add(new double[]{pixelX - img.getWidth() / 2, i - img.getHeight() / 2, z});
            }
            x += d;
            z += dz;
        }
    }
}
```

Рис. 8

С результатом работы метода drawLine вы уже знакомы, в изображениях куба из прошлого раздела (рис. 5, 6, 7) рендер использует именно этот метод. Но попробуем нарисовать что нибудь по-интереснее, например сферу (рис. 10). Тогда мы сможем заметить недостатки алгоритма, которые в дальнейшем можно будет улучшить.

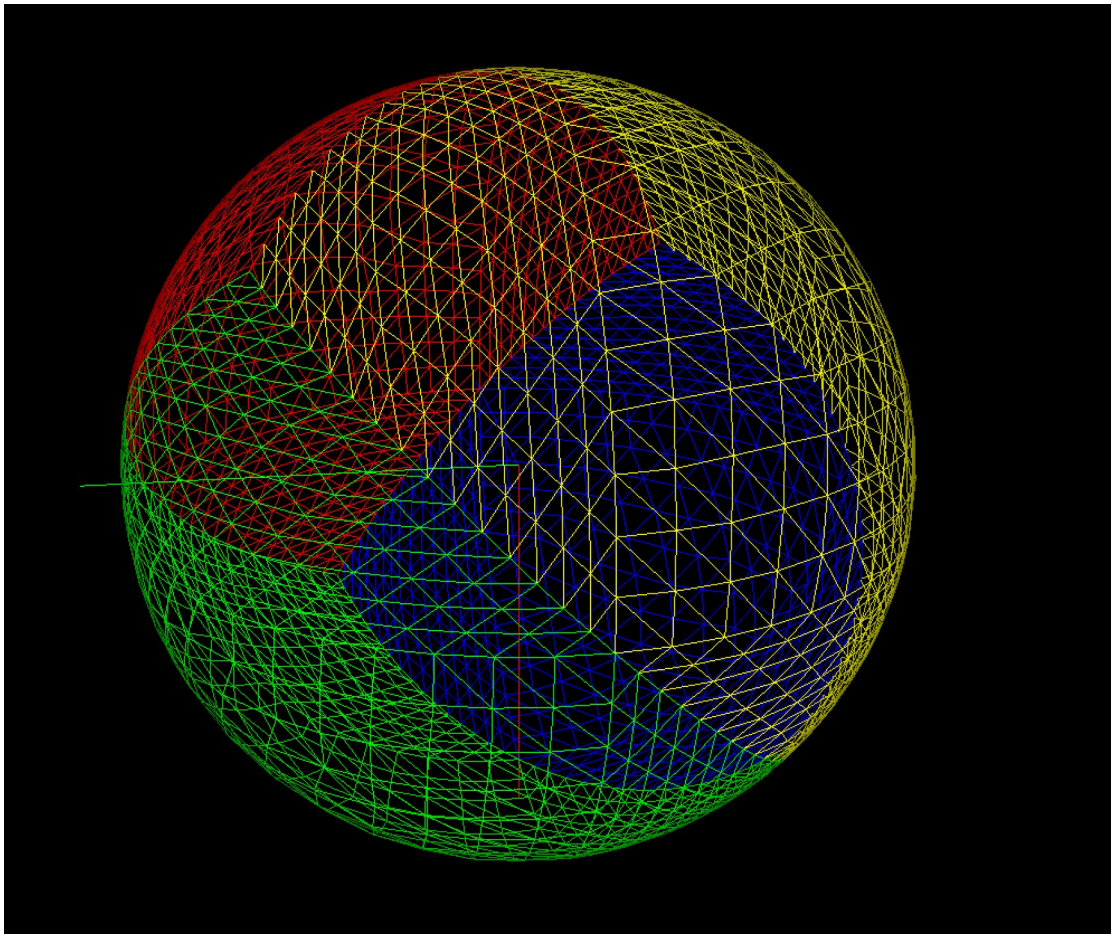

Рис. 10

Первое, что бросается в глаза - это ребристость всех линий, они похожи на лесенки, поэтому в дальнейшем было бы желательно добавить алгоритмы сглаживания изображения.

Растеризация полигонов.

Чтобы показать, что созданный полигон это поверхность, а не просто набор вершин, соединённых линиями, необходимо закрасить всю его внутреннюю площадь цветом.

Будь у меня каждый полигон представлен в виде набора только трёх вершин (т.е. треугольник), то алгоритм растеризации был бы в разы проще. Но в моём случае нужно закрашивать многоугольники, а значит требуется продуманный алгоритм, учитывающий потенциальную впуклусть и выпуклость многоугольника.

Мой выбор пал на алгоритм краевого списка. Если постараться самостоятельно придумать алгоритм растеризации, то скорее всего первым придёт в голову метод, очень похожий на алгоритм краевого списка. И это не удивительно, краевой список интуитивно понятен. Заключается он в простой идее: давайте запоминать, какие координаты *y* имеют пиксели (соединяющие вершины полигона), находящиеся на координате *x*. Тогда у нас появится список из пар значений <x, y> для всего полигона. Далее нам необходимо отсортировать пары по возрастанию *x*, а если *x* равны у пар, то по возрастанию *y*. Затем остаётся только соединить соседние пары вертикальными линиями. Получается суть алгоритма заключается в заполнении полигона вертикальными линиями, заключёнными между верхними и нижними краевыми пикселями, слева на право.

На рисунке 11 показан тот же самый алгоритм, только в нём происходит заполнение по вертикале.

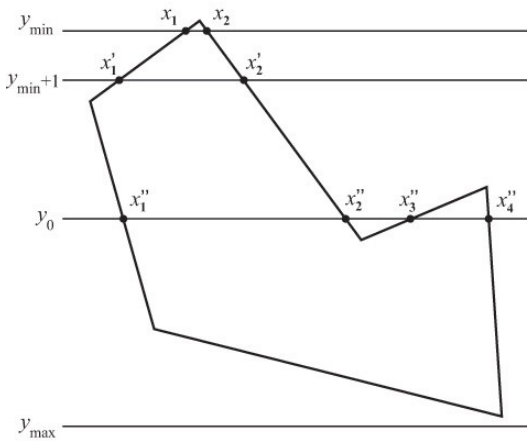

Рис. 11

В блоке растеризации полигонов создан метод drawPolygon (рис. 12). Сначала в нём к каждой вершине полигона применяется трансформация перспективы. Затем уже выполняется алгоритм растеризации рёбер и запись в лист border пары координат *x*, *y*. Затем компаратором сортируется этот список и соединяются линиями соседние пары.

```
Vertex start = vertexes.get(0);
Vertex last = start;
List<double[]> border = new ArrayList<>(); //border buffer;
for (Vertex v : vertexes) {
    drawLine(last, v, c, img, border);
    last = v;
}
drawLine(last, start, c, img, border);
border.sort(new Comparator<double[]>() {
    @Override
    public int compare(double[] o1, double[] o2) {
        if (o1[0] > o2[0]) return 1;
        else if (o1[0] < o2[0]) return -1;
        else if (o1[0] == o2[0]) {
            if (o1[1] > o2[1]) return 1;
            else if (o1[1] < o2[1]) return -1;
        }
        return 0;
    }
});
for (int i = 0; i < border.size() - 1; i += 2) {
    Vertex v1 = new Vertex(border.get(i)[0], border.get(i)[1], border.get(i)[2]);
    Vertex v2 = new Vertex(border.get(i + 1)[0], border.get(i + 1)[1], border.get(i + 1)[2]);
    drawLine(v1, v2, c, img, new ArrayList<>());
}
```

Рис. 12

Давайте взглянем на итог работы алгоритма на примере уже привычного нам куба и новом тетраэдре(рис. 13).

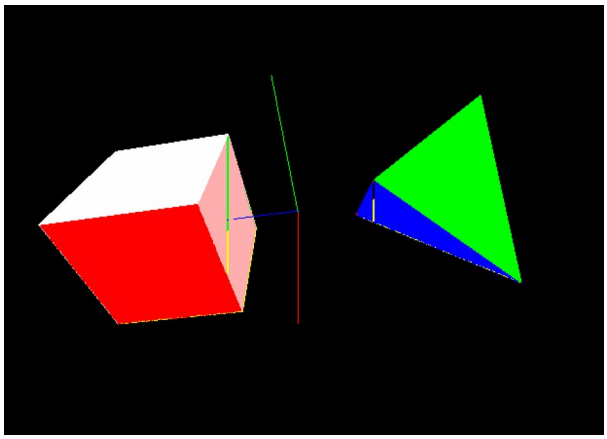

Рис. 13

Как можно заметить, растеризация полигонов без багов не обошлась. Два полигона отрисовались с коллизией — они имеют щели в себе. Решить этот баг, к сожалению, мне не удалось, но, скорее всего, его природа заключается в постоянном преобразовании одного типа данных в другой в алгоритме отрисовки линий, из-за чего часть координат теряется.

Создание освещения сцены

Распространение света в реальном мире это чрезвычайно сложный процесс, зависящий от слишком многих факторов, и, располагая ограниченными вычислительными ресурсами, мы не можем себе позволить учитывать в расчетах все нюансы. Существуют компании, которые развивают технологии расчёта освещения, например NVidia с технологией RTX. Но для неё нужно покупать особые видеокарты.

В рамках своего реферата я могу использовать только упрощённые математические модели. Одним из самых распространённых уже долгие годы является модель Фонгу. Она состоит из трёх компонентов: фонового, диффузного и бликового освещения. Для нас будет интересно на данном этапе только диффузное освещение, так как оно вносит наиболее весомой вклад в общее визуальное ощущение освещения сцены.

Диффузное освещение рассчитывается достаточно легко - чем ближе к перпендикуляру угол между лучом света и поверхностью, тем большую яркость имеет компонент объекта. И наоборот, если полигон будет почти отвёрнут от нас, он значительно потемнеет.

Расчёт угла достаточно прост. Берём три случайных точки полигона с координатами (x_i, y_i, z_i) $i = 0..2$. После этого составляем уравнение плоскости, проходящей через три точки:

$$\begin{vmatrix} x-x_0 & y-y_0 & z-z_0 \\ x_1-x_0 & y_1-y_0 & z_1-z_0 \\ x_2-x_0 & y_2-y_0 & z_2-z_0 \end{vmatrix} = 0$$

Решив это уравнение, мы можем найти коэффициенты A, B, C вектора нормали полигона. Далее остаётся решить простейшую задачки по поиску косинуса угла из 10 класса: $\cos(\alpha) = \frac{C}{\sqrt{A^2+B^2+C^2}}$.

Зная цвет полигона в формате HSB, мы можем просто умножить значение составляющей brightness на $\cos(\alpha)$, получив эффект затемнения. В коде весь описанный выше алгоритм делится на два блока: первый реализован в методе drawPolygon (рис. 14), где ищется нормаль полигона, и второй - метод getShadeColor (рис. 15), где высчитывается затемнение полигона.

```
Color c = p.color;
//поиск угла наклона полигона
//направляющие векторы
if(vertexes.size() >=3) {
    Vertex p1 = new Vertex( x: vertexes.get(1).x - vertexes.get(0).x,  y: vertexes.get(1).y - vertexes.get(0).y,  z: vertexes.get(1).z - vertexes.get(0).z);
    Vertex p2 = new Vertex( x: vertexes.get(2).x - vertexes.get(0).x,  y: vertexes.get(2).y - vertexes.get(0).y,  z: vertexes.get(2).z - vertexes.get(0).z);
    //вектор нормали
    Vertex n = new Vertex(
            x: p1.y * p2.z - p1.z * p2.y,
            y: p1.x * p2.z - p1.z * p2.x,
            z: p1.x * p2.y - p1.y * p2.x);
    //угол наклона к плоскости
    double length = Math.sqrt(n.x*n.x + n.y*n.y + n.z*n.z);
    float cosAngle =(float) (Math.abs(n.z) / length);
    c = getShadeColor(p.color, cosAngle);
}
```

Рис. 14

```
Color getShadeColor(Color c, float cosAngle) {
    float[] hsb = new float[3];
    hsb = Color.RGBtoHSB(c.getRed(), c.getGreen(), c.getBlue(), hsb);
    hsb[2] = 0.3f + 0.7f * cosAngle;
    return Color.getHSBColor(hsb[0], hsb[1], hsb[2]);
}
```

Рис. 15

Давайте взглянем на результат рендеринга с тенями кубика и сферы (рис. 16).

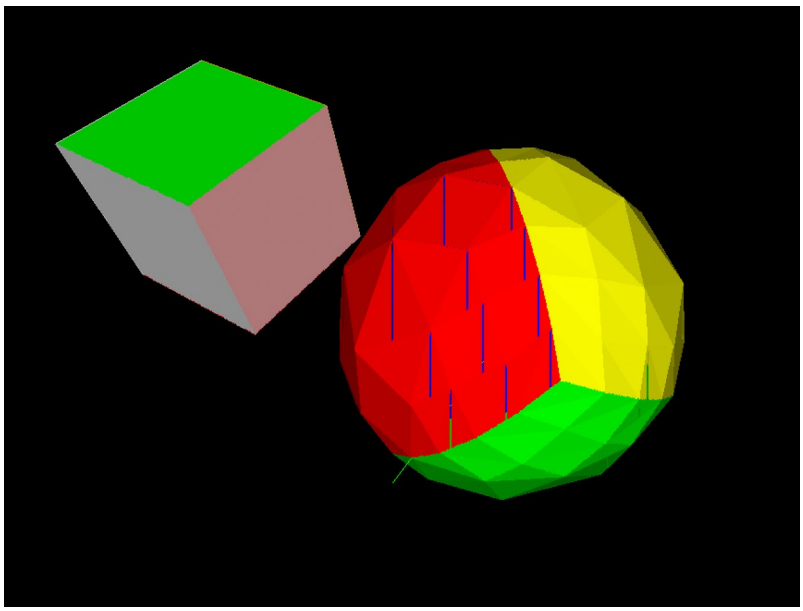

Рис. 16

Выглядит очень красиво, результат радует глаз.

Вывод.

Результатом проделанной работы стало значительное улучшение визуальной составляющей движка. Теперь все объекты в сцене ощущаются намного объёмнее, интереснее и понятнее. Больше не возникает вопросов о том, где вверх кубика, низ, зад и перед. Перспектива решила эту проблему.

Вращение сцены также стало удобнее, растеризация полигонов и диффузный свет придают ему ощущение динамики. А затемнение объектов с увеличением расстояния придаёт сцене ощущение глубины.

Написание реферата подтолкнуло меня к изучению алгоритмов и методов, используемых в 3D рендеринге. Этот путь был долгим и непростым, нужно было погрузиться в мир сложных, иногда абсолютно непонятных алгоритмов, которые нужно было не просто понять, но и реализовать с учётом архитектуры ядра моего рендера. Особенно сложно было решить все проблемы с растеризацией полигонов и картой глубин.

Итоговая программа не лишена недостатков. Всё ещё имеется баг с пустотами в полигонах, иногда окрашиваются крайние пиксели, находящиеся на пересечении полигонов. Убрать эти ошибки алгоритмов можно будет в следующей реферативной работе уже 3 семестра. Но главная цель расширения графических возможностей 3D движка на Java достигнута.

СПИСОК ЛИТЕРАТУРЫ.

8. Трёхмерная графика с нуля. Часть 2: растеризация. – Текст : электронный // Хабр : интернет-сайт. – 24 ноября 2017 г. – URL: https://habr.com/ru/post/342708/ (дата обращения 30.03.21 г.)
9. learnopengl. Урок 2.2 - Основы освещения. – Текст : электронный // Хабр : интернет-сайт. – 23 июля 2017 г. - URL: https://habr.com/ru/post/333932/ (дата обращения 26.03.21 г.)
10. Основы компьютерной геометрии. Написание простого 3D-рендера. – Текст : электронный // Хабр : интернет-сайт. – 21 сентября 2020 г. - URL: https://habr.com/ru/post/520090/. (дата обращения 24.03.21 г.)
11. Растеризация полигонов. – Текст : электронный // Википедия : интернет-энциклопедия. - URL: https://deru.abcdef.wiki/wiki/Rasterung_von_Polygonen. (дата обращения 01.04.21 г.)