Санкт-Петербургский Политехнический Университет Петра Великого

Кафедра компьютерных систем и программных технологий

Отчёт по лабораторной работе №3

на тему: «Методы сглаживания изображений»

Курс: «Разработка графических приложений»

Выполнил студент:

Волкова М.Д.
Группа: 13541/2

Проверил:

Абрамов Н.А.

Санкт-Петербург
2018 г.

Содержание

1 Лабораторная работа №2 **2**
1.1 Цель работы . . . 2
1.2 Описание программы . . . 2
1.3 Ход работы . . . 3
1.3.1 Настройка фиксированной камеры . . . 3
1.4 Вывод . . . 4
1.5 Листинг . . . 5

Лабораторная работа №2

1.1 Цель работы

Разработать программу на языке C для растеризации загруженной модели на экран

1.2 Описание программы

1. Возможности программы:
   (a) Загрузка трехмерной модели из OBJ-файла
   (b) Растеризация каркаса трехмерной модели
   (c) Обеспечение вращения камеры вокруг трехмерной модели
2. Входные параметры программы:
   (a) Ширина и высота окна
   (b) Вертикальный угол обзора камеры для выполненя перспективной проекции
   (c) Ближняя и дальняя плоскости отсечения камеры
   (d) Дистанция от камеры до загруженной модели
   (e) Скорость вращения камеры вокруг модели (градус/сек)
3. Выходные параметры программы:
   (a) Последовательность кадров, выводимая на экран
4. Порядок работы программы:
   (a) Загрузка трехмерной модели в вершинные и индексные буфера
   (b) Определение центра модели (можно считать, что матрица мира для модели – единичная)
   (c) Формирование матрицы проекции
   (d) (Далее – для очередного кадра:)
      i. Формирование матрицы вида исходя из координат центра модели, дистанции до модели и скорости вращения камеры
      ii. Преобразование вершин модели в экранные координаты

1.3 Ход работы

1.3.1 Настройка фиксированной камеры

В OpenGL при использовании фиксированного конвейера есть ровно две матрицы, относящихся к трансформациям точек и объектов:

- GL PROJECTION моделирует ортографическое или перспективное преобразование от трёхмерной усечённой пирамиды (т.е. от области видимости камеры) к трёхмерному кубу с длиной ребра, равной 2 (т.е. к нормализованному пространству).
- GL MODELVIEW сочетает в себе два преобразования: от локальных координат объекта к мировым координатам, а также от мировых координат к координатам камеры.

За рамками фиксированного конвейера можно использовать столько матриц, сколько захочется.

- поведение камеры описывается как ортографическим или перспективным преобразованием, так и положением камеры в мировом пространстве, то есть для моделирования камеры нужны GL PROJECTION и GL MODELVIEW одновременно
- с другой стороны, для трансформаций над телами — вращение предмета с помощью умножения координат на матрицу — нужна матрица GL MODELVIEW.

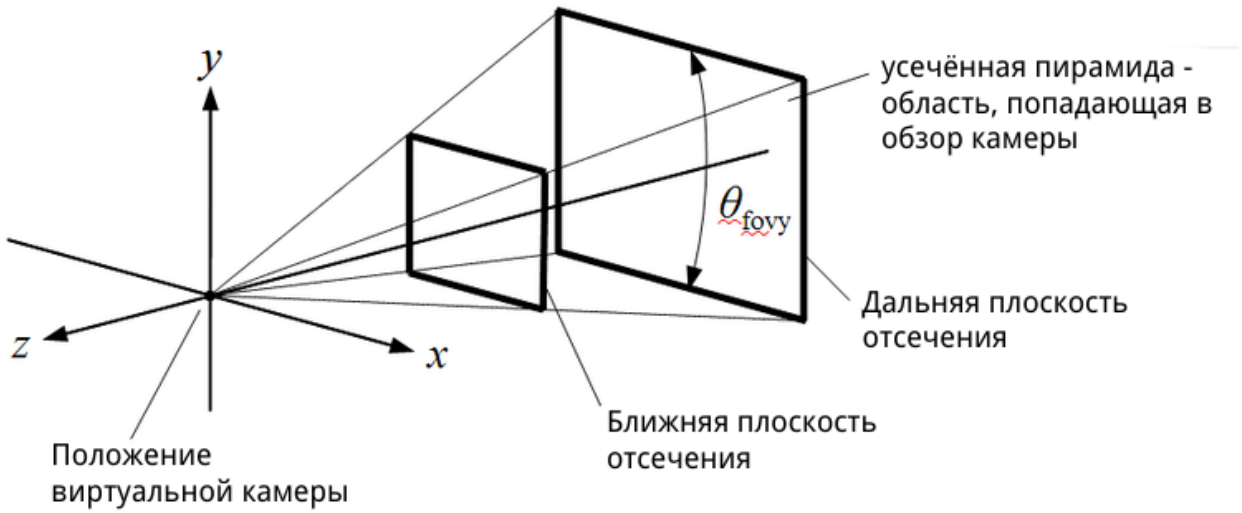
Настроим матрицу GL PROJECTION один раз для перспективного преобразования, а матрицу GL MODELVIEW будем постоянно модифицировать, когда локальная система координат очередного объекта не совпадает с мировой системой координат.

Начнём настройку камеры с GL MODELVIEW: зададим матрицу так, как будто бы камера смотрит с позиции camera position на точку model center, при этом направление “вверх” камеры задаёт вектор glm::vec3(0, 1, 0):

```
camera = glm::lookAt(
                camera_position,
                model_center,
                glm::vec3(0, 1, 0)
        );
```

Для перспективного преобразования достаточно создать матрицу с помощью функции glm::perspective. Она принимает на вход несколько параметров преобразования: горизонтальный угол обзора камеры, соотношение ширины и высоты, а также две граничных координаты для отсечения слишком близких к камере и слишком далёких от камеры объектов.

Эти параметры легко увидеть на следующей иллюстрации:



```
projection = glm::perspective(
             glm::radians(fovy),
             screen_ratio,
             front,
             back)
```

- glm::radians(fovy) - Вертикальное поле зрения в радианах.
- screen ratio - Отношение сторон.
- front - Ближняя плоскость отсечения.
- back - Дальняя плоскость отсечения.

1.4 Вывод

1.5 Листинг

```
#include <utility>

#include <iostream>
#include <any>
#include <OBJ_Loader.h>
#include <glm/vec3.hpp>
#include <glm/geometric.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <cxxopts.hpp>
#include <opencv2/core.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/imgproc.hpp>

#include "render.h"
#include "Drawer.h"
#include "transformers.h"

const int FRAME_PER_SECOND = 10;
const int FRAME_COUNT = 10000;


template<int index>
float min(const std::vector<glm::vec3> &vertices) {
    float result = FLT_MAX;
    for (auto &&vex : vertices) {
        result = std::min(result, vex[index]);
    }
    return result;
}

template<int index>
float max(const std::vector<glm::vec3> &vertices) {
    float result = FLT_MIN;
    for (auto &&vex : vertices) {
        result = std::max(result, vex[index]);
    }
    return result;
}

template<int index>
float getCenter(const std::vector<glm::vec3> &vertices) {
    auto &&min_point = min<index>(vertices);
    auto &&max_point = max<index>(vertices);
    return (min_point + max_point) / 2;
}

glm::vec3 getModelCenter(const std::vector<glm::vec3> &vertices) {
    auto &&center_x = getCenter<0>(vertices);
    auto &&center_y = getCenter<1>(vertices);
    auto &&center_z = getCenter<2>(vertices);
    return glm::vec3(center_x, center_y, center_z);
}

void render(Drawer &drawer, const std::vector<glm::vec3> &vertices, const std::vector<
    unsigned int> &indices) {
    for (auto i = 0; i < indices.size(); i += 3) {
        Triangle triangle{vertices[indices[i]], vertices[indices[i + 1]], vertices[
    indices[i + 2]]};
        drawer.draw(triangle);
    }
}

```

```
62
63 int main(int argc, char **argv) {
64     cxxopts::Options options("Lba3", "Render teapot and maybe something else");
65     std::string default_file_path = "../teapot.obj";
66     std::string default_save_path = "../teapot.avi";
67
68     options.add_options()
69             ("w,width", "Width of image", cxxopts::value<int>()->default_value("800"))
70             ("h,height", "Height of image", cxxopts::value<int>()->default_value("600"))
71             ("s,speed", "Camera speed", cxxopts::value<float>()->default_value("2.0"))
72             ("v,fovy", "fovy", cxxopts::value<float>()->default_value("-50.0"))
73             ("dx", "Distance to model", cxxopts::value<int>()->default_value("120"))
74             ("dy", "Distance to model", cxxopts::value<int>()->default_value("100"))
75             ("f,front", "Front cut plane", cxxopts::value<float>()->default_value("0.1"))
76             ("b,back", "Back cut plane", cxxopts::value<float>()->default_value("10000.0"
    ))
77             ("i,in_file", "Input filename ", cxxopts::value<std::string>()->default_value
    (default_file_path))
78             ("o,out_file", "Output filename ", cxxopts::value<std::string>()->
    default_value(default_save_path));
79
80
81     auto &&arguments = options.parse(argc, argv);
82
83     auto &&width = arguments["width"].as<int>();
84     auto &&height = arguments["height"].as<int>();
85     auto &&speed = arguments["speed"].as<float>();
86     auto &&fovy = arguments["fovy"].as<float>();
87     auto &&distanceX = arguments["dx"].as<int>();
88     auto &&distanceY = arguments["dy"].as<int>();
89     auto &&front = arguments["front"].as<float>();
90     auto &&back = arguments["back"].as<float>();
91     auto &&file_name = arguments["in_file"].as<std::string>();
92     auto &&res_file_name = arguments["out_file"].as<std::string>();
93
94     objl::Loader loader;
95     loader.LoadFile(file_name);
96     auto &&mesh = loader.LoadedMeshes[0];
97
98     auto &&model_vertices = ToGLMVertices().applyList<objl::Vertex, glm::vec3>(mesh.
    Vertices);
99     auto &&model_center = getModelCenter(model_vertices);
100
101    auto &&screen_ratio = static_cast<float>(width) / static_cast<float>(height);
102    auto &&projection = glm::perspective(
103            glm::radians(fovy),
104            screen_ratio,
105            front,
106            back
107    );
108
109    float angle = 0;
110    float angle_per_frame = speed / FRAME_PER_SECOND;
111
112    auto &&start_camera_position = glm::vec4(distanceX, distanceY, 0, 1);
113
114    //    cv::VideoWriter result(res_file_name, -1, FRAME_PER_SECOND, cv::Size(width,
    height));
115
116    //    CvLineDrawer drawer(width, height);
117 //  LineDrawer drawer(width, height);
118    TriangleDrawer drawer(width, height);
119
120
121    for (auto i = 0; i < FRAME_COUNT; i++) {
122        drawer.resetImage();
```

```
123         glm::mat4 rotation_matrix = glm::rotate(glm::mat4(1), angle, glm::vec3(0, 1, 0));
124         glm::vec3 camera_position = (rotation_matrix * start_camera_position);
125         auto &&camera = glm::lookAt(
126                 camera_position,
127                 model_center,
128                 glm::vec3(0, 1, 0)
129         );
130 
131 //    drawer.updatePipeline(std::make_unique<LineTransformationPipeline>(camera,
    projection, width, height));
132         drawer.updatePipeline(std::make_unique<TriangleTransformationPipeline>(camera,
    projection, width, height));
133         render(drawer, model_vertices, mesh.Indices);
134 
135 //          double min, max;
136 //          cv::minMaxLoc(drawer.zBuffer, &min, &max);
137 //          cv::Mat zNorm;
138 //          cv::normalize(drawer.zBuffer, zNorm, 0.0, 1.0, cv::NORM_MINMAX, CV_64F);
139 //          cv::imshow("ZZZZ", zNorm);
140 
141         cv::imshow("Aaaa", drawer.getImage());
142         //        result.write(drawer.getImage());
143         cv::waitKey(2000);
144 
145         angle += angle_per_frame;
146     }
147     //    result.release();
148     return 0;
149 }
```