МИНОБРНАУКИ РОССИИ

федеральное государственное бюджетное образовательное учреждение

высшего образования

«Московский государственный технологический университет «СТАНКИН»

(ФГБОУ ВО «МГТУ «СТАНКИН»)

Институт информационных систем и технологий

Кафедра информационных систем

Программирование специализированных вычислительных устройств
Отчет по лабораторной работе

«3D моделлирование посредством OpenGL для Веба» Часть №2

Студент группы ИДБ-19-07: Касьян А.И.

Проверил доцент кафедры ИС: к.т.н. Волкова О.Р.

October 31, 2020

Оглавление

1 Вводное слово

- В предыдущем документе мы рассмотрели шейдеры и их обработку. В этом я сделаю упор на трансформациях и текстурах, в связи с этим я более детально рассмотрю код самих шейдеров.
- Все используемые технологии сохранились с предыдущего файла.

2 Результат выполнения задания

Данную секцию я разобью на две подсекции:

- Текстурирование
- Трансформация

2.1 Текстурирование

На данный момент, пока я не имею света, текстурирование - это обычная отрисовка картинки, как только появится свет можно будет применять материалы. Обработчик текстур я так же как и обработчик шейдеров занёс в отдельный класс, вот его прототип:

```
class Texture
{
  public:
    Texture(const char *file_name, unsigned int type);
    inline unsigned int getID() const;
    void bind(const int texture_unit);
    void unbind();
    void loadFromFile(const char *file_name);
    ~Texture();

  private:
    unsigned int id;
    int width;
    int height;
    unsigned int type;
};
```

Сам класс из себя ни чего не представляет. В конструкторе, посредством библиотеки SOIL2 мы подгружаем картинку. Далее биндим текстуру и генерируем мипмэп. `loadFromFile` выполняет то же смамое, что и конструктор. `bind`, `unbind` и `getID` говорят сами за себя. Поэтому я приведу листинг только конструктора/`loadFromFile`:

```
Texture::Texture(const char *file_name, unsigned int type)
{
    this->type = type;

    unsigned char *image = SOIL_load_image(
                                           file_name,
                                           &this->width,
                                           &this->height,
                                           NULL,
                                           SOIL_LOAD_RGBA
                                          );

```

```
    // generate texture names
    glGenTextures(1, &this->id);
    glBindTexture(type, this->id);
    // Set our texture parameters
    glTexParameteri(type, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(type, GL_TEXTURE_WRAP_T, GL_REPEAT);
    // Set texture filtering
    glTexParameteri(type, GL_TEXTURE_MAG_FILTER, GL_LINEAR_MIPMAP_LINEAR);
    glTexParameteri(type, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

    if (image)
    {
            // specify 2d img
        glTexImage2D(type, 0, GL_RGBA, this->width, this->height, 0, GL_RGBA, GL_UNSIGNED_BYTE, image);
        glGenerateMipmap(type);
    }
    else
    {
        std::cerr << "An error has occured in the " << __FILE__ << " in line "
                  << __LINE__ << "." << std::endl
                  << "Error occured in the: " << file_name << std::endl;
    }

    glActiveTexture(0);
    glBindTexture(type, 0);
    SOIL_free_image_data(image);
}
```

Пояснять нечего.

2.1.1 Обработка текстур в шейдерах

```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 2) in vec2 texCoord;

out vec2 TexCoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * view * model
                  * vec4(position, 1.0f);
    TexCoord = vec2(texCoord.x, 1.0 - texCoord.y);
}
```

(a) vertex shader

```
#version 330 core
in vec2 TexCoord;

out vec4 color;

uniform sampler2D ourTexture1;



void main()
{
    color = texture(ourTexture1, TexCoord);
}
```

(b) fragment shader

1. **vertex shader** - Фрагмент шейдер будет принимать координату текстуры `TexCoord`, устанавливаем её как out. Считаем эту координату.

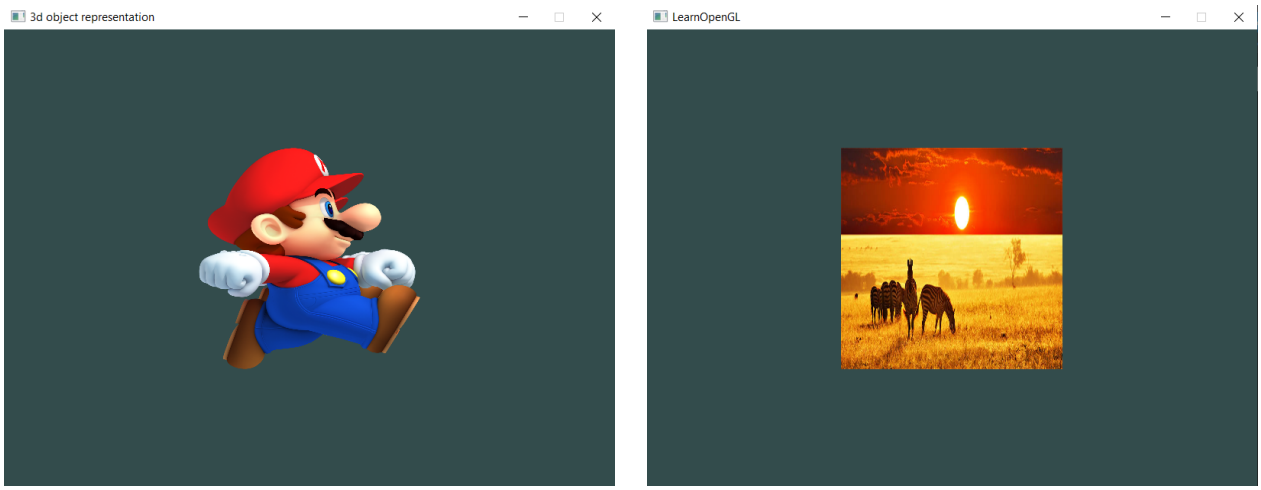
2. **fragment shader** - Во фрагмент шейдере, нам так же требуется получить доступ к объекту текстуры. В GLSL есть buid-in тип данных, предназначеных для текстур (`sampler*D`, где $* \in [1-3]$), поэтому добавляем глобальную текстурную переменную. Чтобы выбрать цвет текстуры, мы используем встроенную функцию `texture`, которая

принимает в качестве первого аргумента текстуру, а в качестве второго аргумента - соответствующие координаты текстуры. Затем функция `texture` выбирает соответствующее значение цвета, используя параметры текстуры, которые мы установили ранее. Результатом этого фрагментного шейдера является (отфильтрованный) цвет текстуры в (интерполированной координате текстуры.

2.1.2 Использование текстур

```
int main()
{
    ...
    // the type is GL_TEXTURE_2D
    Texture texture("assets/images/image1.jpg", GL_TEXTURE_2D);
#ifdef __EMSCRIPTEN__
    std::function<void()> mainLoop = [&]() {
#else
    while (!glfwWindowShouldClose(window))
    {
#endif
        ...

        // Bind Textures using texture units
        texture.bind(0);
        ourShader.set1i(0, "ourTexture1");

        ...

#ifdef __EMSCRIPTEN__
    };
    emscripten_set_main_loop_arg(dispatch_main, &mainLoop, 0, 1);
#else
    }
#endif
...
}
```

2.1.3 Результат работы текстурирования



(a) front (b) rotation

2.2 Трансформация

На данный момент я закончил камеру, фактически трансформация хорошо описана камерой. Однако в дальнейшем я хочу добавить возможность вращения объекта вокруг нулевой точки. Прототип класса камеры выглядит так:

```
// Defines several possible options for camera movement. Used as abstraction to stay away from window-s
// input methods
enum Camera_Movement
{
    FORWARD,
    BACKWARD,
    LEFT,
    RIGHT
};

// Default camera values
const float YAW = -90.0f;
const float PITCH = 0.0f;
const float SPEED = 6.0f;
const float SENSITIVTY = 0.15f;
const float ZOOM = 45.0f;

class Camera
{
  public:
    Camera(glm::vec3 position = glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f)
           float yaw = YAW, float pitch = PITCH);
    Camera(float pos_x, float pos_y, float pos_z, float up_x, float up_y, float up_z, float yaw, float
    // Returns the view matrix calculated using Eular Angles and the LookAt Matrix
    glm::mat4 GetViewMatrix();
    // Processes input received from any keyboard-like input system.
    // Accepts input parameter in the form of camera
    void ProcessKeyboard(Camera_Movement direction, float delta_time);
    // Expects the offset value in both the x and y direction.
    void ProcessMouseMovement(float x_offset, float y_offset, unsigned char constrain_pitch = true);
    // Only requires input on the vertical wheel-axis
    void ProcessMouseScroll(float y_offset);
    float GetZoom();

  private:
    // Camera Attributes
    glm::vec3 position;
    glm::vec3 front;
    glm::vec3 up;
    glm::vec3 right;
    glm::vec3 world_up;

    // Eular Angles
    float yaw;
    float pitch;
    float roll;

    // Camera options
    float movement_speed;
    float mouse_sensitivity;
    float zoom;

    // Calculates the front vector from the Camera's (updated) Eular Angles
    void updateCameraVectors();
};
```

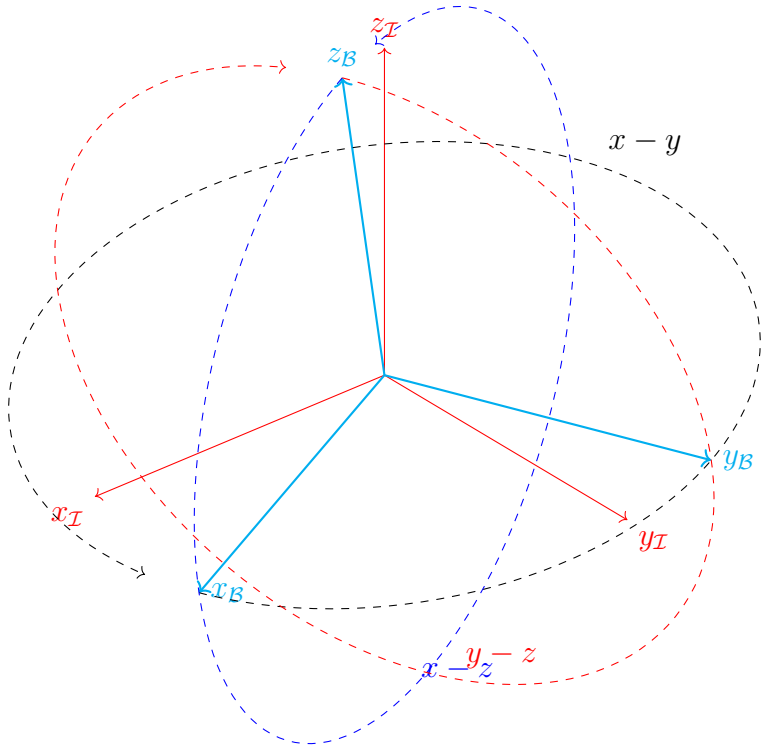
```cpp
void Camera::updateCameraVectors()
{
    // Calculate the new Front vector
    glm::vec3 front;
    front.x = cos(glm::radians(this->yaw)) * cos(glm::radians(this->pitch));
    front.y = sin(glm::radians(this->pitch));
    front.z = sin(glm::radians(this->yaw)) * cos(glm::radians(this->pitch));
    this->front = glm::normalize(front);
    // Also re-calculate the Right and Up vector
    this->right = glm::normalize(
        glm::cross(this->front, this->world_up));
        // Normalize the vectors, because their length gets closer to 0 the
        // more you look up or down which results in slower movement.
    this->up = glm::normalize(glm::cross(this->right, this->front));
}
void Camera::ProcessKeyboard(Camera_Movement direction, float delta_time)
{
    float velocity = this->movement_speed * delta_time;
    switch (direction)
    {
    case FORWARD: this->position += this->front * velocity; break;
    case BACKWARD: this->position -= this->front * velocity; break;
    case LEFT: this->position -= this->right * velocity; break;
    case RIGHT: this->position += this->right * velocity; break;
    default: break;
    }
}

void Camera::ProcessMouseMovement(float x_offset, float y_offset, unsigned char constrain_pitch)
{
    x_offset *= this->mouse_sensitivity;
    y_offset *= this->mouse_sensitivity;

    this->yaw += x_offset;
    this->pitch += y_offset;

    // Make sure that when pitch is out of bounds, screen doesn't get flipped
    if (constrain_pitch)
    {
        if (this->pitch > 89.0f) { this->pitch = 89.0f; }

        if (this->pitch < -89.0f) { this->pitch = -89.0f; }
    }

    // Update Front, Right and Up Vectors using the updated Eular angles
    this->updateCameraVectors();
}

void Camera::ProcessMouseScroll(float y_offset)
{
    if (this->zoom >= 1.0f && this->zoom <= 45.0f) { this->zoom -= y_offset; }

    if (this->zoom <= 1.0f) { this->zoom = 1.0f; }

    if (this->zoom >= 45.0f) { this->zoom = 45.0f; }
}
```

Так же оставляю без пояснений - коментарии всё покрывают. А вот сейчас самое интересное, я поясню - я не программист, я математик (аналитик) мне не интересен код, мне интересна логика, поэтому я немного разберу логику трансформаций.

2.2.1 Математика ♡

2.2.1.1 Камера: осмотр

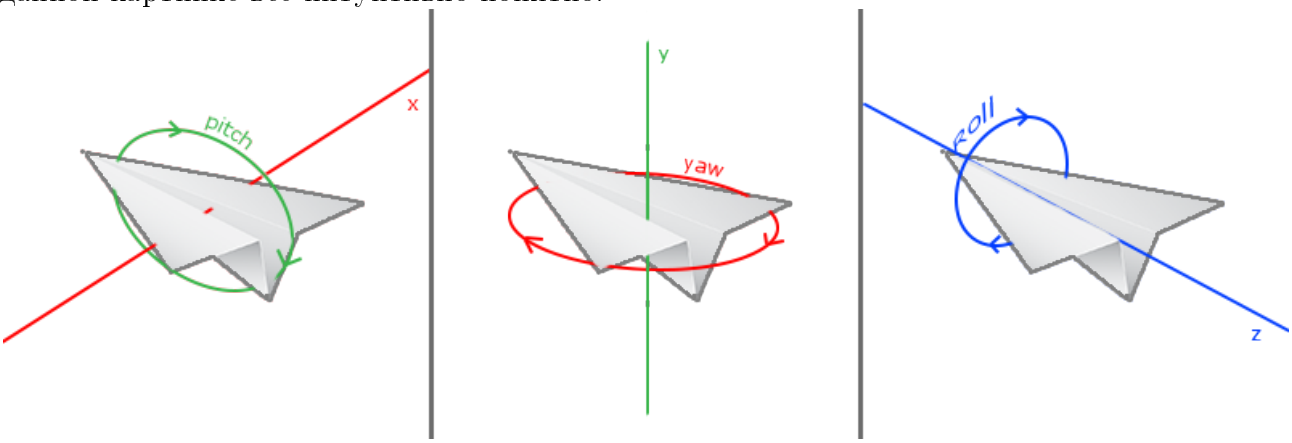
Знакомтесь, углы Эйлера:



Красиво, не правда ли. Углы Эйлера - это три угла для описания ориентации "твердого" тела относительно фиксированной системы координат. У этих углов есть названия:

- **pitch** - угол, отвечающий за вращение вокруг оси абсцисс (оси x)
- **yaw** - угол, отвечающий за вращение вокруг оси ординат (оси y)
- **roll** - угол, отвечающий за вращение вокруг оси аппликат (оси z)

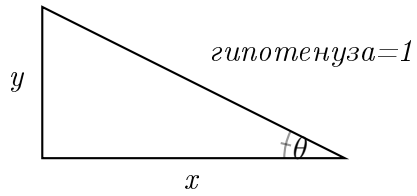
По данной картинке всё интуитивно понятно:



Каждый угол репрезентуется одним значением. Скомбинировав эти углы мы получим ротацию любого вектора в 3d пространстве.

Так как мы работаем с осмотром, посредством камеры - базовая логика углов Эйлера сохраняется, однако для нашей системы камеры roll-угол отпадает, за ненадобностью. Имея pitch и yaw значения мы можем конвертировать их в 3d вектор, который собой

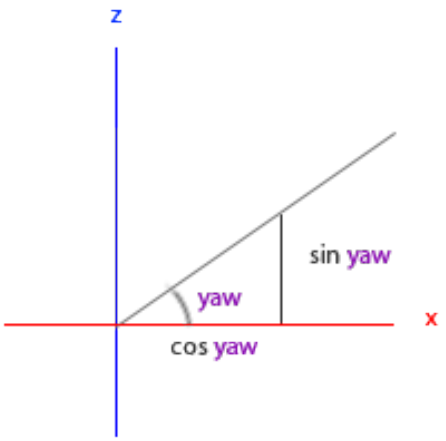
представляет Эвклидов вектор (или просто вектор). Теперь определимся с двумя аксиомами (в нашем случае пусть будут аксиомы):



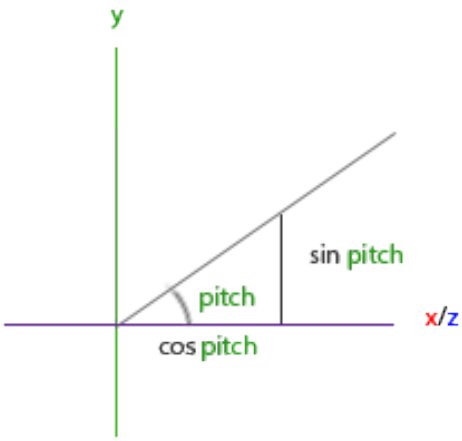
1. $\cos(\theta) = \frac{x}{\text{гипотенуза}}$

2. $\sin(\theta) = \frac{y}{\text{гипотенуза}}$

Если взять гипотенузу за 1, тогда мы получим $\cos(\theta) = x$ и $\sin(\theta) = y$.



(a) yaw angle



(b) pitch angle

Давайте представим, что yaw - это угол, идущий против часовой стрелки, начиная с оси абсцисс. Тогда, длина стороны по OX равна cos(yaw), длина стороны по OZ равна sin(yaw). И так мы разобрались с ккординатой в 3d пространстве, относительно оси OX. Теперь перейдём к оси OY, и если представить, что мы "смотрим" на OY с плоскости XZ, тогда pitch строит схожий треугольник, где длина стороны по OX = cos(pitch), длина стороны по OZ = cos(pitch) и по OY = sin(pitch).

Всё описанное выше расматривалось относительно какой-то оси. Но т.к. камера - объект, расположенный относительно какой-то точки, а не оси, то объединим всё в одно. Т.к. длины по OX и по OZ обе равны cos(pitch), однако в это же аремя $\|X\| = cos$(yaw), а $\|Z\| = sin$(yaw). Тем самым, мы получили финальный вектор, транслированный из двух углов Эйлера.

$$direction = \begin{cases} x = cos(\text{pitch}) \times cos(\text{yaw}) \\ y = sin(\text{pitch}) \\ z = cos(\text{pitch}) \times sin(\text{yaw}) \end{cases}$$

2.2.1.2 Камера: движение

Ни чего интересного, просто берём позицию камеры и прибавляем к ней определённый вектор, помноженный на скаляр скорости. Даже замарачиваться и отрисовывать что-то не хочу.

2.2.1.3 Модель: вращение

Чуть по-позже я добавлю вращение модели. Выполняется посредством всё тех же углов Эйлера, только матрицы немного поменялись:

$$
\begin{aligned}
\begin{bmatrix} x \\ y \\ z \end{bmatrix} &= R_z(\psi)R_y(\theta)R_x(\phi)\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \\
&= \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix}\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \\
&= \begin{bmatrix} \cos\theta\cos\psi & -\cos\phi\sin\psi + \sin\phi\sin\theta\cos\psi & \sin\phi\sin\psi + \cos\phi\sin\theta\cos\psi \\ \cos\theta\sin\psi & \cos\phi\cos\psi + \sin\phi\sin\theta\sin\psi & -\sin\phi\cos\psi + \cos\phi\sin\theta\sin\psi \\ -\sin\theta & \sin\phi\cos\theta & \cos\phi\cos\theta \end{bmatrix}\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}
\end{aligned}
$$

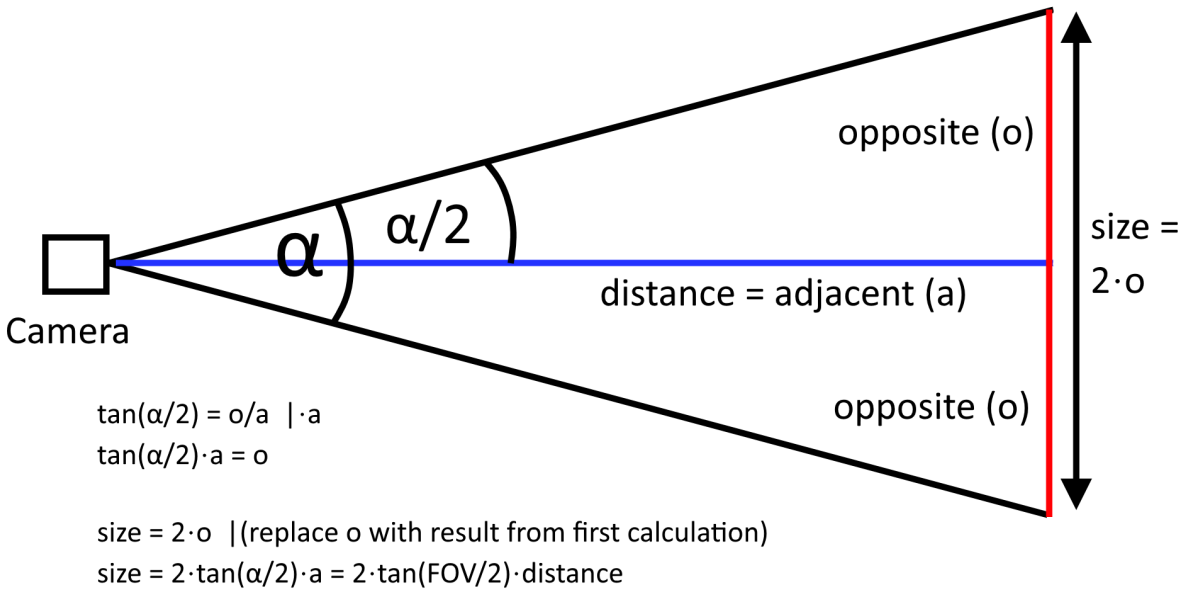
2.2.2 Трансформация в OpenGL

Кратко, как всё описанное выше работает в OpenGL:

1. **Model matrix** - Преобразует позицию модели в позицию в мире. Это положение зависит от положения, масштаба и поворота отрисованной модели.

2. **View matrix** - Если в реальном мире мы двигаем камерой вокруг мира, то в OpenGL мир двигается вокруг камеры камера установленна на нулевой точке, с -Z направлением, т.е. если вы хотите поднять камеру - вы опускаете мир.

3. **Projection matrix** - после трансформации мира, выполняется проецирование (отдаляем камеру - объект уменьшается). У матрицы проекции есть своя фишка, её нужно нормализовать. После того как вы премените матрицу проекции - вы получите некоторые координаты (clip coordinates), однако знаете момент, когда вы отдаляете объект, а он растягивается. Это связано с FOV, чем больше угол FOV - тем больше точек в себя включает сцена.



Исходя из FOV и дистанции мы расчитываем размер объекта. Однако, конечный результат будет лежать в отрезке $[-1; 1]$. И только потом подгоняем под размер

сцены. А теперь вспомноим, что мы получили координаты в ненормализованной форме, а следовательно их нужно нормализовать.

$$v_{\text{normalized}} = \begin{pmatrix} x_{clip}/w_{clip} \\ y_{clip}/w_{clip} \\ z_{clip}/w_{clip} \end{pmatrix}$$

w - показатель на сколько далеко объект находится от камеры (дистанция).

Далее, чтобы нормально всё отобразить - мы объединяем все три матрицы в одну:

$$v' = M_{\text{proj}} * M_{\text{view}} * M_{\text{model}} * v$$

Вообщем-то данную операцию мы выполняем в вертекс шейдере.

```
gl_Position = projection * view * model * vec4(position, 1.0f);
```

2.2.3 Использование камеры и трансформаций

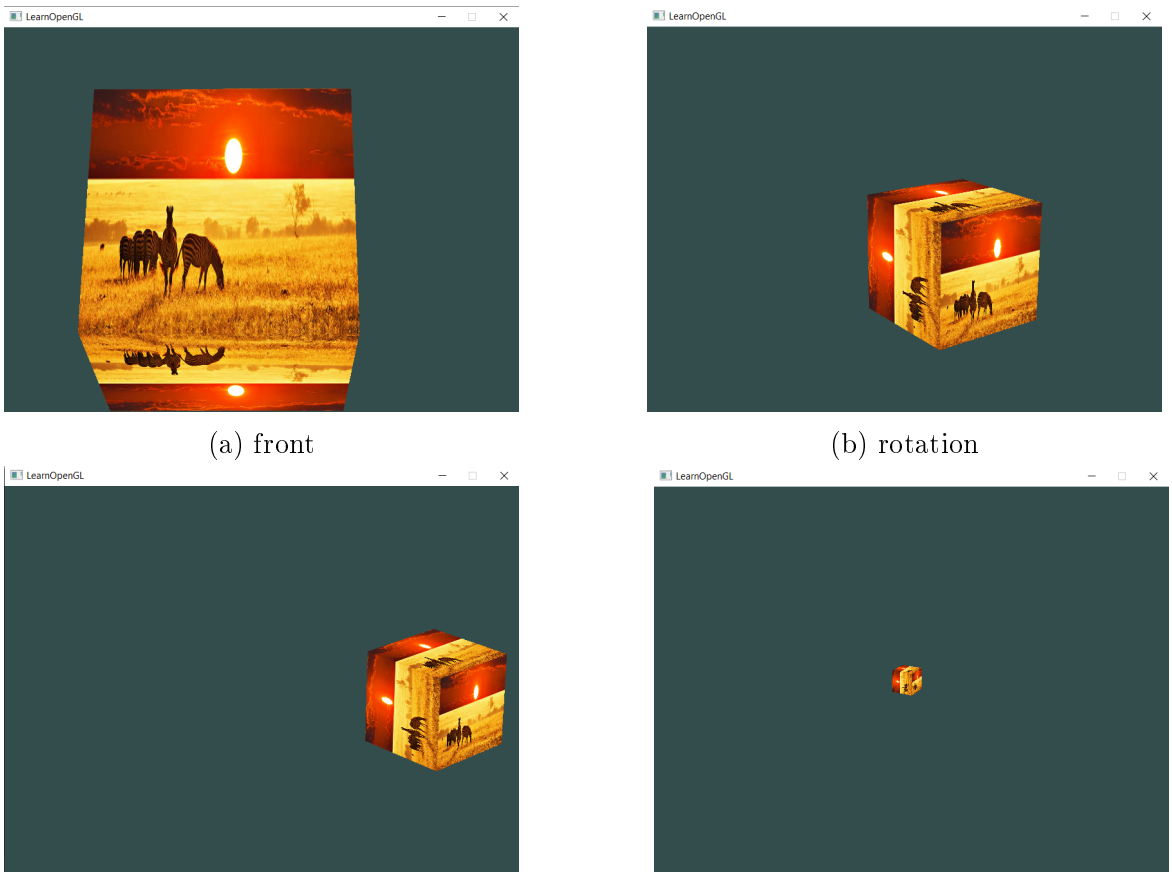
```
const GLuint WIDTH = 800, HEIGHT = 600;
int SCREEN_WIDTH, SCREEN_HEIGHT;
// Function prototypes
void KeyCallback(GLFWwindow *window, int key, int scancode, int action, int mode);
void ScrollCallback(GLFWwindow *window, double xOffset, double);
void MouseCallback(GLFWwindow *window, double xPos, double yPos);
void DoMovement();
Camera camera(glm::vec3(0.0f, 0.0f, 3.0f));
GLfloat lastX = WIDTH / 2.0;
GLfloat lastY = HEIGHT / 2.0;
bool keys[1024];
bool firstMouse = true;
GLfloat deltaTime = 0.0f;
GLfloat lastFrame = 0.0f;
int main()
{
 ...
    Texture texture("assets/images/image1.jpg", GL_TEXTURE_2D);
#ifdef __EMSCRIPTEN__
    std::function<void()> mainLoop = [&]() {
#else
    while (!glfwWindowShouldClose(window))
    {
#endif
...
        glm::mat4 projection(1);
        projection = glm::perspective(glm::radians(camera.GetZoom()),
                    (GLfloat)SCREEN_WIDTH / (GLfloat)SCREEN_HEIGHT, 0.1f, 1000.0f);
        // Create camera transformation
        glm::mat4 view(1);
        view = camera.GetViewMatrix();
        // Pass the matrices to the shader
        ourShader.setMat4fv(view, "view");
        ourShader.setMat4fv(projection, "projection");
        glBindVertexArray(VAO);
        for (GLuint i = 0; i < 10; i++)
        {
            // Calculate the model matrix for each object and pass it to shader before drawing
            glm::mat4 model(1);
            model = glm::translate(model, cubePositions[i]);
```

```
            GLfloat angle = 20.0f * i;
            model = glm::rotate(model, angle, glm::vec3(1.0f, 0.3f, 0.5f));
            ourShader.setMat4fv(model, "model");
            ourShader.Use();
            glDrawArrays(GL_TRIANGLES, 0, 36);
        }
...
#ifdef __EMSCRIPTEN__
    };
    emscripten_set_main_loop_arg(dispatch_main, &mainLoop, 0, 1);
#else
    }
#endif
...
}
// Moves/alters the camera positions based on user input
void DoMovement()
{
    // Camera controls
    if (keys[GLFW_KEY_W] || keys[GLFW_KEY_UP]) { camera.ProcessKeyboard(FORWARD, deltaTime); }
    if (keys[GLFW_KEY_S] || keys[GLFW_KEY_DOWN]) { camera.ProcessKeyboard(BACKWARD, deltaTime); }
    if (keys[GLFW_KEY_A] || keys[GLFW_KEY_LEFT]) { camera.ProcessKeyboard(LEFT, deltaTime); }
    if (keys[GLFW_KEY_D] || keys[GLFW_KEY_RIGHT]) { camera.ProcessKeyboard(RIGHT, deltaTime); }
}

// Is called whenever a key is pressed/released via GLFW
void KeyCallback(GLFWwindow *window, int key, int scancode, int action, int mode)
{
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS) { glfwSetWindowShouldClose(window, GL_TRUE); }

    if (key >= 0 && key < 1024)
    {
        if (action == GLFW_PRESS) { keys[key] = true; }
        else if (action == GLFW_RELEASE)
        {
            keys[key] = false;
        }
    }
}

void MouseCallback(GLFWwindow *window, double xPos, double yPos)
{
    if (firstMouse)
    {
        lastX = xPos;
        lastY = yPos;
        firstMouse = false;
    }

    GLfloat xOffset = xPos - lastX;
    GLfloat yOffset = lastY - yPos; // Reversed since y-coordinates go from bottom to left

    lastX = xPos;
    lastY = yPos;

    camera.ProcessMouseMovement(xOffset, yOffset);
}

void ScrollCallback(GLFWwindow *window, double xOffset, double yOffset)
{
   camera.ProcessMouseScroll(yOffset);
}
```

2.2.4 Результат работы программы

2.2.4.1 MSVC компилятор

На данный момент результат работы программы выглядит так:



(a) front (b) rotation

(c) movement (d) zoom

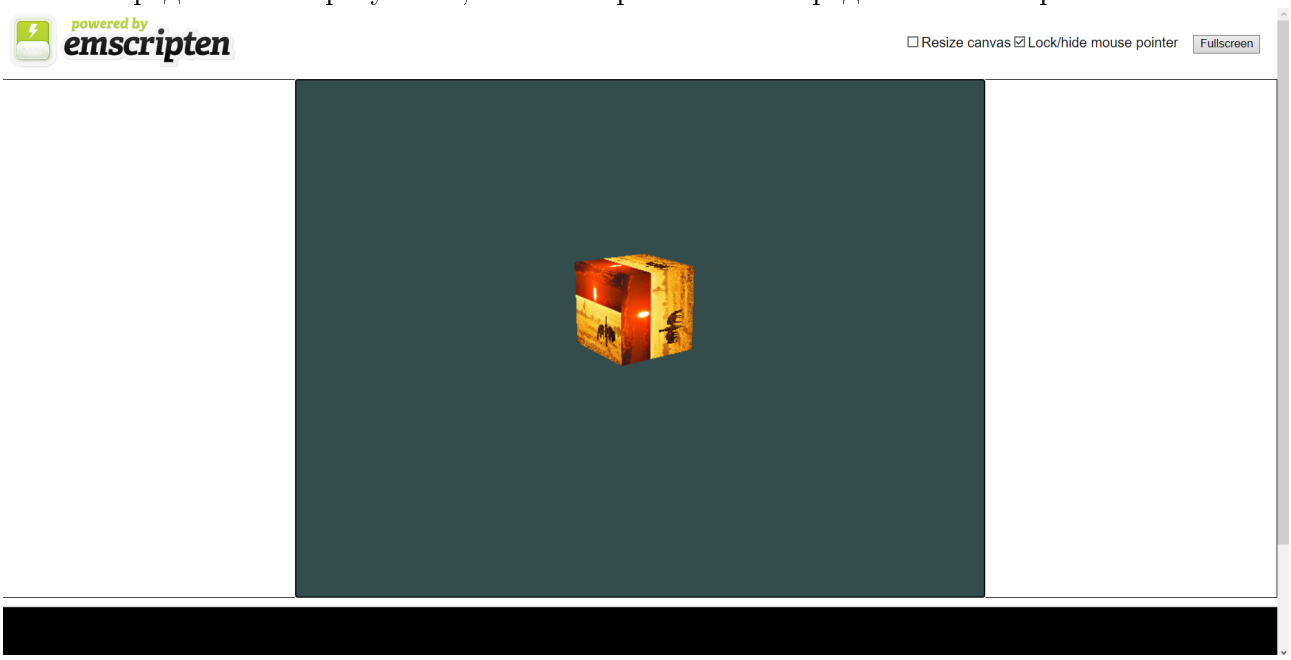
Figure 4: Transformations

2.2.4.2 Emscripten компилятор

В коде синтаксис emscripten'а обрамляется `ifndef-else-endif`. Так же флаги, которые я использовал (их значения можно увидеть тут):

- `FORCE_FILESYSTEM=1`
- `USE_WEBGL2=1`
- `USE_GLFW=3`
- `FULL_ES3=1`
- `ALLOW_MEMORY_GROWTH=1`
- `ASSERTIONS=1`
- `WASM=1`

Так же предоставляю результат, скомпилированный посредством emscripten:



2.3 Улучшения

Как я и говорил, мне осталось сделать свет и 3d object loader.