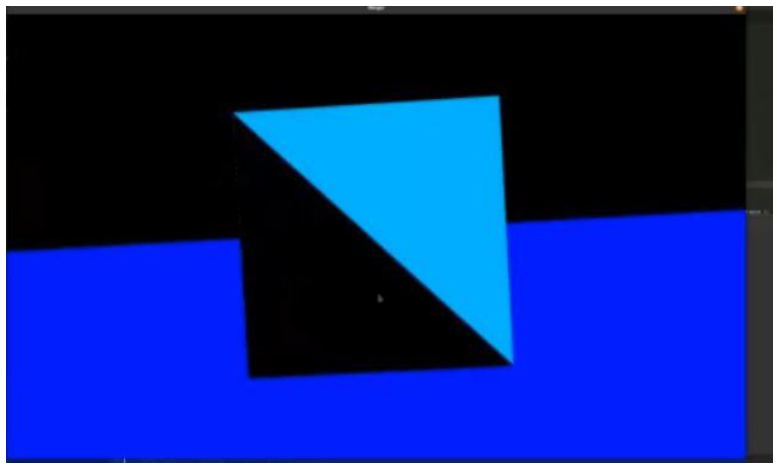
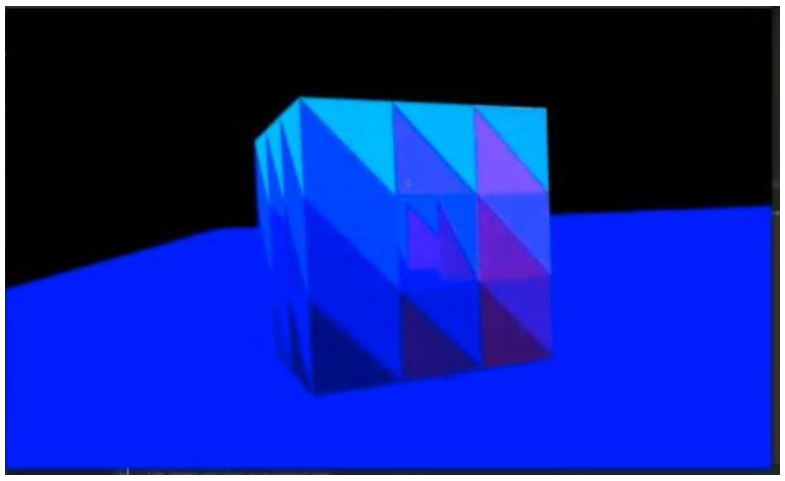
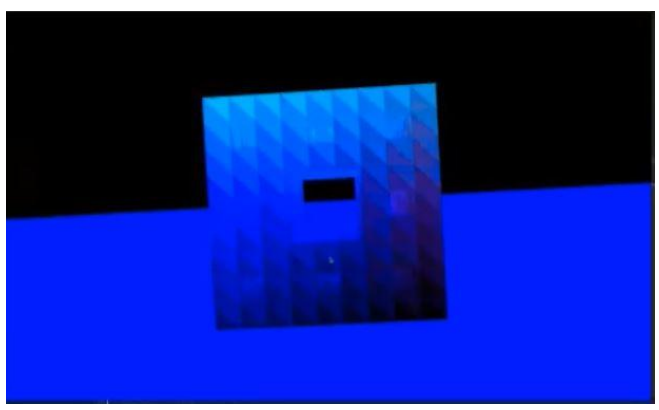
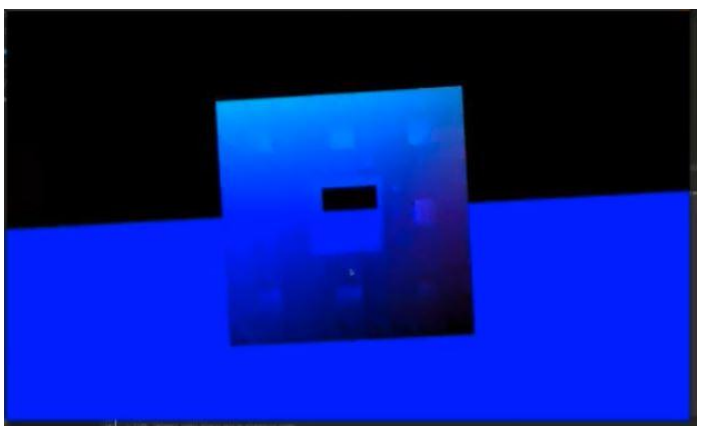
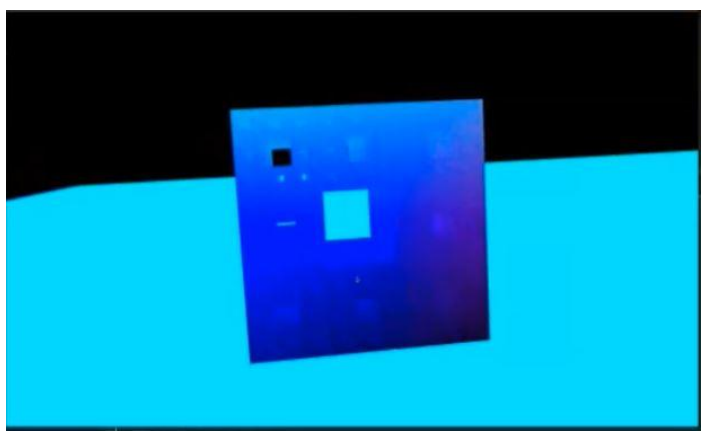
Menger Sponge Design Document
Remus Wong () Marlowe Johnson (msj879)
OS: Linux
Libraries : GLFW, GL

Cube

20 points in order of left to right, back to front, bottom to top
Subdivides each point until a given branch of the recursive algorithm reaches a depth of 0.
Subdivision: Calculate L value which is used to modify and divide vector components based on the given depth of the current Menger. Each vector has its components re-modified according to its positioning in the cube and the recursively passed for a menger subdivision again.

The cube vertices are initially created based on 8 distinct points that come for a basis of near/far, left and right, and up/down. These 8 points are created and then those vertices are used to triangulate faces based on the positioning of the point and then stored into the faces vector, the face vector components are the number positioning of the vertex in the vertex vector.

Because my shader was broken, for the sake of these pictures I used a different shader so you can see the different nesting levels of the cube

CreateMenger(), the base level operation for the recursive function, creates a cube with the boundaries.

```
void
Menger::CreateMenger(glm::vec3 M, glm::vec3 m, std::vector<glm::vec4>& obj_vertices, std::vector<glm::uvec3>& obj_faces) const
{
    int offset = obj_vertices.size();
    // Create starting cube
    obj_vertices.push_back(glm::vec4(m.x,m.y,m.z, 1.0f));   // 0, far-bot left
    obj_vertices.push_back(glm::vec4(M.x,m.y,m.z, 1.0f));   // 1, far-bot right
    obj_vertices.push_back(glm::vec4(M.x,m.y,M.z, 1.0f));   // 2, near-bot right
    obj_vertices.push_back(glm::vec4(m.x,m.y,M.z, 1.0f));   // 3, near_bot left
    obj_vertices.push_back(glm::vec4(m.x,M.y,m.z, 1.0f));   // 4, far-top left
    obj_vertices.push_back(glm::vec4(M.x,M.y,m.z, 1.0f));   // 5, far-top right
    obj_vertices.push_back(glm::vec4(M.x,M.y,M.z, 1.0f));   // 6, near-top right
    obj_vertices.push_back(glm::vec4(m.x,M.y,M.z, 1.0f));   // 7, near-top left

    // Triangulate faces
    // first pass
    obj_faces.push_back(glm::uvec3(0+offset, 4+offset, 5+offset));
    obj_faces.push_back(glm::uvec3(1+offset, 5+offset, 6+offset));
    obj_faces.push_back(glm::uvec3(2+offset, 6+offset, 7+offset));
    obj_faces.push_back(glm::uvec3(3+offset, 7+offset, 4+offset));
    obj_faces.push_back(glm::uvec3(3+offset, 0+offset, 1+offset));
    obj_faces.push_back(glm::uvec3(4+offset, 7+offset, 6+offset));
    // second pass
    obj_faces.push_back(glm::uvec3(6+offset, 5+offset, 4+offset));
    obj_faces.push_back(glm::uvec3(1+offset, 2+offset, 3+offset));
    obj_faces.push_back(glm::uvec3(4+offset, 0+offset, 3+offset));
    obj_faces.push_back(glm::uvec3(7+offset, 3+offset, 2+offset));
    obj_faces.push_back(glm::uvec3(6+offset, 2+offset, 1+offset));
    obj_faces.push_back(glm::uvec3(5+offset, 1+offset, 0+offset));
}
```

The subdivide function is the recursive function. I cut the picture because I manually did the calculations for the 20 different cube positions that are subdivided.

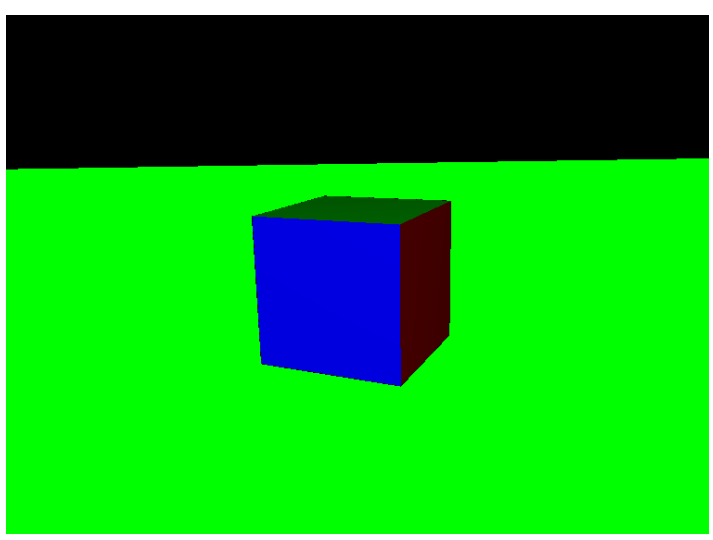
```
void
Menger::subdivide(int depth, glm::vec3 M, glm::vec3 m, std::vector<glm::vec4>& obj_vertices, std::vector<glm::uvec3>& obj_faces) const{
    double L = pow(1.0/3.0, 2+1-depth);
    L = abs(M.x - m.x)/3.0;
    if(depth == 0){
        Menger::CreateMenger(M, m, obj_vertices, obj_faces);
        return;
    }
    M = glm::vec3(M.x-L, M.y-L, M.z-L);
    m = glm::vec3(m.x+L, m.y+L, m.z+L);
    glm::dvec3 M_kj;        // max bounds
    glm::dvec3 m_kj;        // min bounds

    // Goes in order of left to right, back to front, bottom to top
    // bottom layer
    // 1
    M_kj = glm::vec3(M.x-L, M.y-L, M.z-L);
    m_kj = glm::vec3(m.x-L, m.y-L, m.z-L);
    subdivide(depth-1, M_kj, m_kj, obj_vertices, obj_faces);
    // 2
    M_kj = glm::vec3(M.x, M.y-L, M.z-L);
    m_kj = glm::vec3(m.x, m.y-L, m.z-L);
    subdivide(depth-1, M_kj, m_kj, obj_vertices, obj_faces);
    // 3
    M_kj = glm::vec3(M.x+L, M.y-L, M.z-L);
    m_kj = glm::vec3(m.x+L, m.y-L, m.z-L);
    subdivide(depth-1, M_kj, m_kj, obj_vertices, obj_faces);
```

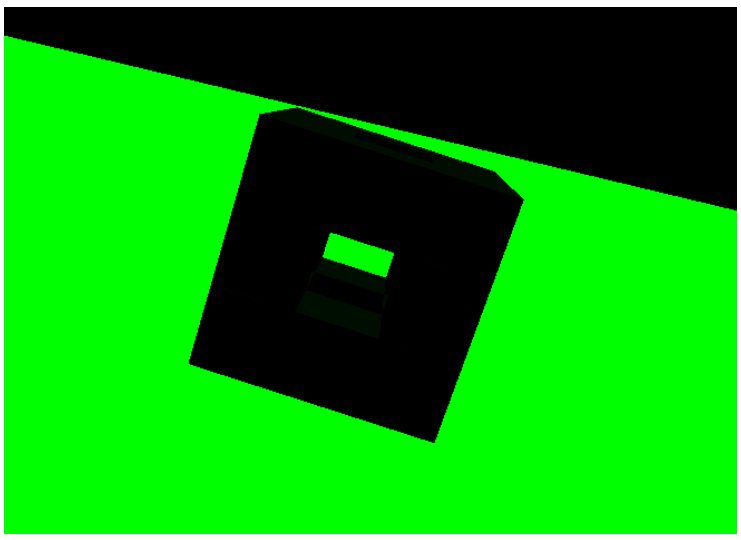
Shader

Vertex -
Take in vertex position, a view matrix and a vector light position. Output is a vector of the vs light position and some additional VertexData that contains the vertex's world position.
This was handled by multiplication of the view matrix and the vertex position vector, giving a vec4 of the view position. This vector is then used in calculation of the vs_light direction along the view matrix and the light_position originally.

```
const char* vertex_shader =
R"zzz(#version 330 core
in vec4 vertex_position;
uniform mat4 view;
uniform vec4 light_position;
out vec4 vs_light_direction;
out VertexData {
    vec4 world_pos;
} VertexOut;
void main()
{
    VertexOut.world_pos = vertex_position;
    gl_Position = view * vertex_position;
    vs_light_direction = -gl_Position + view * light_position;
}
)zzz";
```

Geometry-
For the geometry shader the normal needed to be properly calculated in order for the shader to be correct and to pass to the fragment shader for correctness in shading of the faces of the cube.
To correctly fix the normal calculation, the cross is taken between vertices of the triangles passed in and that normal value gets passed out. The coloration seems to be correct, However I was not able to figure out why the brightness of the face would vary with the camera movement. In addition, while the nesting level 0 seemed to provide a colored output, subsequent nesting levels would loose brightness quickly the deeper the nesting level is. This can be seen in the picture below.





```
const char* geometry_shader =
R"zzz(#version 330 core
layout (triangles) in;
layout (triangle_strip, max_vertices = 3) out;
uniform mat4 projection;
uniform mat4 view;
in VertexData {
    vec4 world_pos;
} VertexIn[3];
in vec4 vs_light_direction[];
flat out vec4 normal;
out vec4 light_direction;
void main()
{
    int n = 0;
    normal = vec4(0.0, 0.0, 1.0f, 0.0);
    vec4 a,b;
    a = (VertexIn[1].world_pos - VertexIn[0].world_pos);
    b = (VertexIn[2].world_pos - VertexIn[0].world_pos);
    vec3 a3, b3;
    a3 = vec3(a);
    b3 = vec3(b);
    vec3 cx = cross(a3, b3);
    vec4 c = vec4(cx, 0.0f);
    normal = c;
    for (n = 0; n < gl_in.length(); n++) {
        light_direction = vs_light_direction[n];
        gl_Position =  projection * gl_in[n].gl_Position;
        // normal =  projection * gl_in[n].gl_Position;

        EmitVertex();
    }
    EndPrimitive();
}
)zzz";
```

Fragment -
We use the normal and check which direction it is facing in order to see which color the face needs to be colored.

Floor Fragment Shader-
We check the direction of the normal again to give a checkerboard pattern to the floor.
Aligned to x and z axis, black for x normals, and white for the z normals. But for these pictures, because the shader did not work properly, we use the color of green for the floor. I wasn't entirely sure of how to extract the world position out of the fragment. I know that it would be interpolated but I couldn't correctly convert to world coordinates.

```
const char* fragment_shader =
R"zzz(#version 330 core
flat in vec4 normal;
in vec4 light_direction;
out vec4 fragment_color;
void main()
{
    vec4 color = normal;
    if (abs(normal.x) == 1.0)
        color = vec4(1.0, 0.0, 0.0, 1.0); //red
    if (abs(normal.y) == 1.0)
        color = vec4(0.0, 1.0, 0.0, 1.0); //green
    if (abs(normal.z) == 1.0)
        color = vec4(0.0, 0.0, 1.0, 1.0); //blue
    float dot_nl = dot(normalize(light_direction), normalize(normal));
    dot_nl = clamp(dot_nl, 0.0, 1.0);
    fragment_color = dot_nl * color;
}
)zzz";

//fragment_color = clamp(dot_nl * color, 0.0, 1.0);
// FIXME: Implement shader effects with an alternative shader.
const char* floor_fragment_shader =
R"zzz(#version 330 core
uniform mat4 projection;
uniform mat4 view;
flat in vec4 normal;
in vec4 light_direction;
out vec4 fragment_color;
void main()
{
    // vec4 color = vec4(1.0,1.0,1.0,0.0);
    // if (abs(normal.x) == 1.0)
    //    color = vec4(0.0,0.0,0.0,1.0); //black
    // else if (abs(normal.z) == 1.0)
    //    color = vec4(1.0,1.0,1.0,1.0); //white
    // color = vec4(0,1,0,1);
    // float dot_nl = dot(normalize(light_direction), normalize(normal));
    // dot_nl = clamp(dot_nl, 0.0, 1.0);
    // fragment_color = dot_nl * color;

    vec4 world_pos = projection * view * gl_FragCoord;
    vec4 color = normal;
    float dot_nl = dot(normalize(light_direction), normalize(normal));
    dot_nl = clamp(dot_nl, 0.0, 1.0);
    fragment_color = dot_nl * color;
    // fragment_color = vec4(0.0, 0.0, 1.0, 1.0);
}
)zzz";
```

Controls

Rotation of camera based of mouse drag

To calculate the direction based on the mouse, we separate the x and y axis and pass in those directions. We only care if its in the negative/positive x/y direction, not the actual value.

In the camRotation() function, we separate the rotation matrices. One for the x and y direction. I did this because at first I was using the mouse direction in world space and using a single rotation matrix but my camera would lock on the axis I was rotating around so I think I may have ben suffering from gimbal lock.

In orbit mode, we want to update the eye after rotating the look vector. So after rotating the look_ by the two matrices, we recompute the eye_ and the up_ vector.

In fps mode, we basically do the same thing except we do not need to recompute the eye because the eye remains fixed in fps while the look_ swivels around.

Note: In the video link below the camera rotation looks buggy. However the behavior seems the same as in the solution except it looks wonky because the shaders on my objects are not correct so it makes the world disorienting.

Demonstration of Camera Transformations:
https://drive.google.com/file/d/1g2WhwBxVE4N7TMD_GilAdsY6lgBxxMlS/view?usp=sharing

```
void
MousePosCallback(GLFWwindow* window, double mouse_x, double mouse_y)
{
    // cout << mouse_x << " " << mouse_y << " || " << prevX << " " << prevY << endl;
    if (!g_mouse_pressed){
        prevX = mouse_x;
        prevY = mouse_y;
        return;
    }
    if (g_current_button == GLFW_MOUSE_BUTTON_LEFT) {
        // FIXME: left drag
        glm::vec4 mouse_direction;
        mouse_direction.x = mouse_x - prevX;
        mouse_direction.y = -(mouse_y - prevY);
        g_camera.camRotation(mouse_direction);
    }
    else if (g_current_button == GLFW_MOUSE_BUTTON_RIGHT) {
        if (prevY > mouse_y){
            g_camera.zoomMouse(-1);
        }
        else if (prevY < mouse_y) {
            g_camera.zoomMouse(1);
        }
    }
    else if (g_current_button == GLFW_MOUSE_BUTTON_MIDDLE) {
    }
    prevX = mouse_x;
    prevY = mouse_y;
}
```

```
void Camera::camRotation(glm::vec4 mouse_dir)
{
    mouse_dir.y *= -1.0f;
    // glm::vec3 rotAxis = glm::normalize(glm::cross(look_, glm::vec3(-mouse_dir)));
    // glm::mat4 rot = glm::rotate(glm::mat4(1), rad, rotAxis);
    // two rotation matricies for each axis
    glm::mat4 rotX = glm::rotate(glm::mat4(1), 3.14f/180*rotation_speed*mouse_dir.x, up_);
    glm::mat4 rotY = glm::rotate(glm::mat4(1), 3.14f/180*rotation_speed*mouse_dir.y, getX());
    if(!fps_mode) {
        // ORBIT MODE: update eye
        glm::vec3 center = eye_ + camera_distance_*look_;
        glm::vec4 look4 = glm::vec4(look_, 0);
        look4 = look4 * rotX * rotY;
        look_.x = look4.x;
        look_.y = look4.y;
        look_.z = look4.z;
        eye_ = center - camera_distance_*glm::vec3(look_);
        up_ = glm::vec3(glm::vec4(up_, 0) * rotX * rotY);
    }
    else {
        // FPS MODE: update center
        glm::vec4 look4 = glm::vec4(look_, 0);
        look4 = look4 * rotX * rotY;
        look_.x = look4.x;
        look_.y = look4.y;
        look_.z = look4.z;
        up_ = glm::vec3(glm::vec4(up_, 0) * rotX * rotY);
    }
}
```

get_view_matrix() and camera::lookat()

Basically followed the slides to calculate the lookat matrix. We calculate the axes of the camera and arrange them in a mat4 to form the view matrix.

```cpp
// FIXME: Calculate the view matrix
glm::mat4 Camera::get_view_matrix() const
{
    glm::mat4 CameraMatrix = glm::lookAt(eye_, glm::vec3(0,0,0), up_);
    CameraMatrix = Camera::lookAt();
    return CameraMatrix;
}

glm::mat4 Camera::lookAt() const
{
    glm::vec3 center = eye_ + camera_distance_*look_;
    glm::vec3 forward = glm::normalize(center - eye_);              // Z axis
    glm::vec3 right = glm::normalize(glm::cross(forward, up_));     // X axis
    glm::vec3 up = glm::normalize(glm::cross(right, forward));      // Y axis
    // Make them into vec4 with scaled w
    glm::vec4 f4 = glm::vec4(forward, glm::dot(forward, eye_));
    glm::vec4 l4 = glm::vec4(right, -1*glm::dot(right, eye_));
    glm::vec4 u4 = glm::vec4(up, -1*glm::dot(up, eye_));
    // reverse the Z because openGL
    f4.x *= -1;
    f4.y *= -1;
    f4.z *= -1;
    glm::mat4 view = glm::mat4();
    view[0] = l4;
    view[1] = u4;
    view[2] = f4;
    view[3] = glm::vec4(0,0,0,1);
    view = glm::transpose(view);
    return view;
}
```

ZoomMouse and Zoom KeyWS

Zooming with the right click works by modifying the camera_distance and recalculating the eye (this is for orbit mode). The mouse event will send it a value of 1 or -1 to indicate zooming in or out.
For the implementation of the ‘W’ and ‘S’ keys, if in orbital mode zoomMouse is reused. If In fps mode, the eye is translated by the zoom direction, zoom speed, and the look vector because we want to keep the distance from the eye to the center constant.

```cpp
void Camera::zoomMouse(float zoomDir)
{
    // if center = eye + camer_distance * look
    // center and look will remain constant while we update eye and camera distance
    if(fps_mode)
        return;
    glm::vec3 center = eye_ + camera_distance_*look_;
    camera_distance_ +=  zoomDir * zoom_speed;
    if (camera_distance_ >= 0 ){
        glm::mat4 trans = glm::translate(glm::mat4(), (glm::vec3(look_)*zoomDir*zoom_speed));
        // eye_ = glm::vec3(glm::vec4(eye_, 0) * trans);
        eye_ = center - camera_distance_*look_;
    }
    else
        camera_distance_ = 0;
}

void Camera::zoomKeyWS(float zoomDir)
{
    if(!fps_mode) {
        zoomMouse(zoomDir);
    }
    else {
        // FPS MODE: move eye and center at same rate
        eye_ += zoomDir * zoom_speed * look_;
    }
}
```

camKeyAD

The eye is translated by the direction passed in multiplied by the pan_speed of the camera and the tangent - or the x vector

```
void Camera::camKeyAD(float dir)
{
    glm::vec3 tangent = getX();
    eye_ += dir * pan_speed * tangent;
}

void Camera::camKeyUpDown(float dir)
{
    eye_ += dir * pan_speed * getY();
}
```

camKeyUpDown

The eye is translated by the direction multiplied by the pan speed and this time the y vector. This works by the same principle as the ‘A, D’ keys.

```
void Camera::camKeyAD(float dir)
{
    glm::vec3 tangent = getX();
    eye_ += dir * pan_speed * tangent;
}

void Camera::camKeyUpDown(float dir)
{
    eye_ += dir * pan_speed * getY();
}
```

camKey Left Right

A radian value is calculated and multiplied by the roll speed to give the angle of rotation. Then the rotation matrix is composed of the identity matrix rotated at the angle of the radian value with respect to the vector components of the z vector. The up vector is then translated using this matrix which in turn changes the viewing matrix.

```
void Camera::camKeyLeftRight(float dir)
{
    float rad = dir*3.14f/180*roll_speed;
    glm::mat4 rot = glm::rotate(glm::mat4(1), rad, getZ());
    glm::vec4 up4 = glm::vec4(up_, 0);
    up4 = up4 * rot;
    up_.x = up4.x;
    up_.y = up4.y;
    up_.z = up4.z;
}
```