ARCBALL CAMERA IN OPENGL WITH C++

Oguz D.

10/01/2022

We used quaternions to build an arcball camera. There are also other ways to do it. Opengl version is 3.30.

1 Quaternions

Quaternions are kind of number structure in mathematics and also used in physics and engineering. A quaternion consists of one real part and three imaginary-like part which are called "basic quaternions." It seems as $a + xi + yj + zk$. The real part is a and $i - j - k$ are basic quaternions. It can be thought that a quaternion is built by merging a real number and a three dimensional vector.

1.1 Properties of Quaternion

There is no need to focus all properties of quaternions. Just some of them which we use here will be given.

$$i^2 = j^2 = k^2 = ijk = -1$$

$$ij = k, \quad jk = i, \quad ki = jik = -j, \quad kj = -i, \quad ji = -k$$

A quaternion can be represented in form of $q = [s, v]$ where s is the real part and v is the vector part as $v = xi + yj + zk$. If

$$q_a = [s_a, v_a]$$
$$q_b = [s_b, v_b]$$

then

$$q_a + q_b = [s_a + s_b, v_a + v_b]$$
$$q_a - q_b = [s_a - s_b, v_a - v_b]$$

Assume that c is a constant;

$$cq_a = [cs_a, cv_a].$$

$$\begin{aligned} q_a.q_b &= (s_a + x_a i + y_a j + z_a k)(s_b + x_b i + y_b j + z_b k) \\ &= [s_a s_b - v_a v_b, s_a v_b + s_b v_a + v_a \times_b] \end{aligned}$$

These are some of properties which we need to build an arcball camera.

If θ is the rotation angle and $u_x i + u_y j + u_z k$ is the unit rotation axis which camera rotates about, then rotation quaternion is represented as $r = e^{\frac{\theta}{2}(u_x i + u_y j + u_z k)}$.

2 What We Do

2.1 Getting Screen Coordinates

This is how our code works.

When we click the mouse on the application window, the position where mouse cursor is on in that moment is our start position and the pc gets that position. Then we move the mouse and the cursor moves on the screen (we are still pushing the mouse button), every position on the screen that our mouse stands on with pushed button is our current position and we have to calculate the rotation for every current position with start position until the mouse button is released.

Below code piece is for getting start position.

```
void mouse_button_callback(GLFWwindow* window, int button, int action, int mods){
    action == glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT);
    if(action == GLFW_PRESS){

        double startXPos, startYPos; //screen coordinates when mouse clicks.
        glfwGetCursorPos(window, &startXPos, &startYPos);
        arcCamera.startPos.x = ((startXPos - (SCR_WIDTH/2) ) / (SCR_WIDTH/2)) * RADIUS; //co
        arcCamera.startPos.y = (((SCR_HEIGHT/2) - startYPos) / (SCR_HEIGHT/2)) * RADIUS;//
        arcCamera.startPos.z = arcCamera.z_axis(arcCamera.startPos.x, arcCamera.startPos.y);
        flag = true;
    }
     else if(action == GLFW_RELEASE){
        arcCamera.replace();
        flag = false;

        }
}
```

And for getting current position.

```
void mouse_pos_callback(GLFWwindow* window, double xpos, double ypos){
    if(flag == true){
        //Get the screen coordinates when mouse clicks.
        arcCamera.currentPos.x = ((xpos - (SCR_WIDTH/2) ) / (SCR_WIDTH/2)) * RADIUS;
        arcCamera.currentPos.y = (((SCR_HEIGHT/2) - ypos) / (SCR_HEIGHT/2)) * RADIUS;
        arcCamera.currentPos.z = arcCamera.z_axis(arcCamera.currentPos.x, arcCamera.currentP
        arcCamera.rotation();
    }
}
```

Time to explain something.
We have an "ArcballCamera" class and "arcCamera" is an object of this class (You'll see this class later). In the first piece above; when we click the button, the program gets cursor position on the application screen with "glfwGetCursorPos" and assign them to startXPos and startYPos. Then these start positions converts to normalized device coordinates(NDC) and are assigned to startPos vector(it is a glm::vec3 variable). But we need z axis, because, remember, arcball camera assumes that there is a sphere which we rotates and a sphere is an 3D shape. So we have to get z axis for certain location on the sphere. The z axis is calculated with this formula:

$$z = f(x,y) = \begin{cases} +\sqrt{1-x^2-y^2} & ; \quad x^2+y^2 \leq 1 \\ 0 & ; \quad x^2+y^2 > 1 \end{cases}$$

Now, we have normalized x,y and z coordinates for start position. Last thing we have to do in "mouse_button_callback" function is to assing "true" to the flag. This flag inform "mouse_pos_callback" function that the mouse button is pressed or released. When it is "true", "mouse_pos_callback" knows that the button is pressed and gets the current position of the cursor for every moment until the button is released. When the button is released, "false" is assigned to "flag" and the function finishes to get current position.

"mouse_pos_callback" function, like we said before, gets the current position of the cursor on the window, converts them to NDCs, calculates z axis and starts the "rotation()" function of arcballCamera object. "rotation()" is the heart of this application. It's time to dive into it.

2.2 Rotation

Here is our algorithm:

1. Transform start and current positions as unit vectors.
2. Get rotational axis by cross product of start and current positions and transform it as unit vector.

3. Get cosine of the rotation angle by dot product of start and current unit vectors.

4. Get rotation angle θ.

5. Build currentQuaternion struct with θ and rotational axis by using required operations (assign the cosine of half the angle and rotational axis which is scaled by sine of half the angle).

6. Do $q' = q_{current} \cdot q_{last}$ product.

7. Replace lastQuaternion variables with new variables after the mouse button is released.

Of course, you wonder what currentQuaternion or lastQuaternion is. Here is the answer:

```
struct Quaternion{
    float cosine; //cosine of half the rotation angle
    glm::vec3 axis; //unit vector scaled by sine of half the angle

};
```

These are structures in type of the struct "Quaternion". We created "Quaternion" structure for describing rotation with quaternions. A Quaternion structure has a "cosine" variable and an "axis" vector.

The next one is the class which we used in this study.

```
class ArcballCamera{
public:

    glm::vec3 position = glm::vec3(0.0f, 0.0f, -3.0f);
    glm::vec3 startPos;
    glm::vec3 currentPos = startPos;
    glm::vec3 startPosUnitVector;
    glm::vec3 currentPosUnitVector;

    Quaternion currentQuaternion;
    Quaternion lastQuaternion = {0.0f, glm::vec3(1.0f, 0.0f, 0.0f)};

    float cosValue, cosValue_2;
    float theta;
    float angle = 180.0f;
    glm::vec3 rotationalAxis = glm::vec3(1.0f, 0.0f, 0.0f);
    glm::vec3 rotationalAxis_2;
    ArcballCamera (){};
    float z_axis(float,float);
    glm::vec3 getUnitVector(glm::vec3);
```

```
    float dotProduct();
    void rotation();
    void replace();


};
```

Here is some of functions and what they do:

- z_axis(float,float): Calculates z axis.
- getUnitVector(glm::vec3): Transforms its parameter to unit vector.
- dotProduct(): Calculates the dot product of start and current position's unit vector. It doesn't get any parameter.
- rotation(): Makes all rotation calculation.
- replace(): Replaces lastQuaternion variables with new ones.

The most important function is "rotation()". What other functions do is very clear and also they have just one job, so I will not explain them. However rotation() function has more calculation, each of them is important and has to be understood.Let's start and go on step by step to see what rotation() do.

```
startPosUnitVector = getUnitVector(startPos);
    currentPosUnitVector = getUnitVector(currentPos);
    currentQuaternion.axis = glm::cross(startPos, currentPos);
    currentQuaternion.axis = getUnitVector(currentQuaternion.axis);

    cosValue = dotProduct(); //q0 is cosine of the angle here.
    if(cosValue > 1) cosValue = 1; // when dot product gives '1' as result, it doesn't equal
    theta = (acos(cosValue) * 180 / 3.1416); //theta is the angle now.
```

These piece of codes shows the first four steps in the algorithm we mentioned above. Transforming start and current position vectors to unit vectors, getting rotation axis and rotation angle.In the bottom, we calculate rotation angle θ and completed the first four step of the algorithm. But maybe this line must be explained:

```
if(cosValue > 1) cosValue = 1;
```

I left a comment there but want to tell again. When dotProduct() calculates the dot product of vectors and if the result is 1, it may not equal to 1 totally. It may 1.00000001 but when you display the result on the screen, it seems as 1. Because of this, i put this line there and it checks the result if it is 1 certainly or not.

Then we build currentQuaternion assigning cosine of half the rotation angle and rotation axis which is scaled by sine of half the rotation angle.

```
currentQuaternion.cosine = cos((theta / 2) * 3.1416 / 180); //currentQuaternion.cosine is co

    currentQuaternion.axis.x = currentQuaternion.axis.x * sin((theta / 2) * 3.1416 / 180);
    currentQuaternion.axis.y = currentQuaternion.axis.y * sin((theta / 2) * 3.1416 / 180);
    currentQuaternion.axis.z = currentQuaternion.axis.z * sin((theta / 2) * 3.1416 / 180);
```

Remember, we represents rotation with quaternion. Let's write currentQuaternion in mathematical notation:

$$\begin{aligned} q_{current} &= e^{\frac{\theta}{2}(u_x i + u_y j + u_z k)} \\ &= \cos\frac{\theta}{2} + (u_x i + u_y j + u_z k)\sin\frac{\theta}{2} \\ &= [\cos\frac{\theta}{2}, (u_x i + u_y j + u_z k)\sin\frac{\theta}{2}] \end{aligned}$$

We have also another quaternion, called lastQuaternion, which keeps the last values of cosine and rotation axis.

$$\begin{aligned} q_{last} &= e^{\frac{\theta'}{2}(u'_x i + u'_y j + u'_z k)} \\ &= \cos\frac{\theta'}{2} + (u'_x i + u'_y j + u'_z k)\sin\frac{\theta}{2} \\ &= [\cos\frac{\theta'}{2}, (u'_x i + u'_y j + u'_z k)\sin\frac{\theta}{2}] \end{aligned}$$

We have arrived the most important step. How do we calculate the rotation? We do this with this product:

$$q' = q_{current} q_{last}$$

It was given how to product two quaternion before, so I will not mention about it again but tell how to code it. The real part of the quaternion q' is calculated by below lines:

```
cosValue_2 = (currentQuaternion.cosine * lastQuaternion.cosine)
                        - glm::dot(currentQuaternion.axis, lastQuaternion.axis);
```

Then the vector part of the quaternion is calculated by:

```
glm::vec3 temporaryVector;

    temporaryVector = glm::cross(currentQuaternion.axis, lastQuaternion.axis);
```

```
    rotationalAxis_2.x = (currentQuaternion.cosine * lastQuaternion.axis.x) +
                          (lastQuaternion.cosine * currentQuaternion.axis.x ) +
                          temporaryVector.x;

    rotationalAxis_2.y = (currentQuaternion.cosine * lastQuaternion.axis.y) +
                          (lastQuaternion.cosine * currentQuaternion.axis.y ) +
                          temporaryVector.y;

    rotationalAxis_2.z = (currentQuaternion.cosine * lastQuaternion.axis.z) +
                          (lastQuaternion.cosine * currentQuaternion.axis.z ) +
                          temporaryVector.z;
```

We need a temporary(glm::vec3 temporaryVector) vector to calculate the cross product of vector parts of current and last quaternions.

As a result, we have cosine value and vector part of q'. What we have to do is getting rotation angle from the cosine value and getting rotation axis from the vector part.

```
angle = (acos(cosValue_2) * 180 / 3.1416) * 2;

    rotationalAxis.x = rotationalAxis_2.x / sin((angle / 2) * 3.1416 / 180);
    rotationalAxis.y = rotationalAxis_2.y / sin((angle / 2) * 3.1416 / 180);
    rotationalAxis.z = rotationalAxis_2.z / sin((angle / 2) * 3.1416 / 180);
```

Remember that the vector part of a rotation quaternion is in scaled form by sine of half the angle. To extract the rotation axis, we need to divide the vector part by sine.
Then "angle" and "rotationalAxis" is put into glm::rotate function.

```
view = glm::rotate(view, glm::radians(arcCamera.angle), arcCamera.rotationalAxis);
```

Finally, put cosine and vector part(rotationalAxis_2) of q' into lastQuaternion when mouse button is released. lastQuaternion variables must be updated after rotation because we calculate the rotation by combining last two rotations which is the operation represented by $q' = q_{current}q_{last}$.

```
if(action == GLFW_RELEASE){
        arcCamera.replace();
        flag = false;

        }

void ArcballCamera::replace(){
    lastQuaternion.cosine = cosValue_2;
    lastQuaternion.axis = rotationalAxis_2;
}
```

References

[1] Marie ? How to implement a simple arcball camera. https://asliceofrendering.com/camera/2019/11/30/ArcballCamera/, 2019. Accessed: 2022-01-17.

[2] amdj1. Arcball controller. https://pixeladventuresweb.wordpress.com/2016/10/04/arcball-controller/, 2016. Accessed: 2022-01-17.

[3] Robert Eisele. Trackball rotation using quaternions. https://www.xarg.org/2021/07/trackball-rotation-using-quaternions/, 2021. Accessed: 2022-01-17.

[4] Kevin Li. Introduction to computer graphics, assignment 3 lecture notes. http://courses.cms.caltech.edu/cs171/assignments/hw3/hw3-notes/notes-hw3.html#ref1, 2016. Accessed: 2022-01-17.

[5] Ken Shoemake. Arcball rotation control (paper written by ken shoemake). https://research.cs.wisc.edu/graphics/Courses/559-f2001/Examples/Gl3D/arcball-gems.pdf, 1994. Accessed: 2022-01-17.

[6] Brad Smith. Arcball. http://rainwarrior.ca/dragon/arcball.html, 2008. Accessed: 2022-01-17.

[7] Wikibooks. Opengl programming/modern opengl tutorial arcball. https://en.wikibooks.org/wiki/OpenGL_Programming/Modern_OpenGL_Tutorial_Arcball, 2020. Accessed: 2022-01-17.