Evan Lancaster
CS 480
11/20/22

Project Assignment #4

Part 1: Pyramid Class Implementation

Vertex Setup

```
Vertices = {
        {{.0f, 2.f, .0f}, {1,1,0}}, // top
        {{1.0f, .0f, 1.0f}, {1, 0.1, 0.5}},
        {{-1.0f, .0f, 1.0f}, {1, 0.1, 0.5}},
        {{-1.0f, .0f, -1.0f}, {1, 0.1, 0.5}},
        {{1.0f, .0f, -1.0f}, {1, 0.1, 0.5}} // four base
};

Indices = {
  0, 1, 2, //E
  0, 2, 3, //N
  0, 3, 4, //W
  0, 4, 1, //S
  1, 2, 3, // bottom
  3, 4, 1 // bottom
};
```

This code happens in the pyramid SetupVertices function, which was inherited and overridden from the Object class. The vertices object consists of the top vertex and the four bottom vertices of a pyramid, with the top vertex being yellow and the rest being a mix of mostly red. The indices create six triangles, four on the top for each side, and two to create the base.

Buffer Setup

```
void Object::Initialize(GLint posAttribLoc, GLint colAttribLoc) {

    // Set up your VOA
    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);

    // setting the Vertex VBO
    glGenBuffers(1, &VB);
    glBindBuffer(GL_ARRAY_BUFFER, VB);
    glBufferData(GL_ARRAY_BUFFER, sizeof(Vertex) * Vertices.size(), &Vertices[0], GL_STATIC_DRAW);
    glVertexAttribPointer(posAttribLoc, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), 0);
    glVertexAttribPointer(colAttribLoc, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, color));

    // Setting the Index VBO
    glGenBuffers(1, &IB);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IB);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(unsigned int) * Indices.size(), &Indices[0], GL_STATIC_DRAW);
}
```

The buffer setup for each object is done in the base class, Object's, initialize function. This is the same code from the first project assignment, and simply creates a VAO, a VBO, and has the attributes set to read an array of 3 position floats and 3 color floats for each vertex, then to create the index VBO based on our indices from above.

Model Matrix Setup

```
Object::Object(Object* _pivot, glm::vec3 location, float angle, float scale)
{
    createVertices();
    pivot = _pivot;

    model = glm::translate(glm::mat4(1.0f), location);
    model *= glm::rotate(glm::mat4(1.0f), angle, glm::vec3(0, 1.0f, .0f));
    model *= glm::scale(glm::vec3(scale, scale, scale));
}
```

The model matrix is created after setting up the vertices. The user inputs the location, angle, and scale they wish for the model to have, and then we create a matrix based on the translation, then rotate, then scale.

Model Matrix Updating

```
void Object::Update(unsigned int dt)
{
    model = glm::translate(model, m_speed);
    model *= glm::rotate(glm::mat4(1.0f), m_rotation_speed, m_rotation_vector);
    m_orbit_progress += m_orbit_speed;
}
```

The model matrix is updated through the update function of the Object class. This is called every rendered frame. This simply translates the model by the speed, and rotates it based on the given speed and desired vector of rotation. This function also helps to progress the object's orbit by its orbital speed. The calling of this function is handled by the graphics pipeline.

Rendering Details

```
void Object::Render(GLint posAttribLoc, GLint colAttribLoc)
{
    // Bind VAO
    glBindVertexArray(vao);

    // Bind VBO(s)
    glBindBuffer(GL_ARRAY_BUFFER, VB);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IB);

    // enable the vertex attribute arrays
    // this is the poistion attrib in the vertex shader
    glEnableVertexAttribArray(posAttribLoc);
    // this is the color attribe in the vertex shader
    glEnableVertexAttribArray(colAttribLoc);

    // Draw call to OpenGL
    glDrawElements(GL_TRIANGLES, Indices.size(), GL_UNSIGNED_INT, 0);

    // disable the vertex attributes
    glDisableVertexAttribArray(posAttribLoc);
    glDisableVertexAttribArray(colAttribLoc);

    // unbind VBO(s) and ElementBuffer(s)
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

}
```

The rendering is handled by the base class, Object. This class follows the usual steps to render, binding VAO/VBOs, setting the vertex attribute arrays, calling for OpenGL to draw the VBOs, then disabling and unbinding our inputs. The calling of this function is done by the graphics pipeline.

Part 2: Solar System Implementation

Model Stacks

```
Object::Object(Object* _pivot, glm::vec3 location, float angle, float scale)
{
    createVertices();
    pivot = _pivot;

    model = glm::translate(glm::mat4(1.0f), location);
    model *= glm::rotate(glm::mat4(1.0f), angle, glm::vec3(0, 1.0f, .0f));
    model *= glm::scale(glm::vec3(scale, scale, scale));
}
```

```
void Object::setOrbitTranslation()
{
    glm::mat4 pivotMatrix = getParentMatrix();
    model[3] = pivotMatrix[3] + glm::vec4(m_orbit_radius * glm::sin(m_orbit_progress), 0.0f, m_orbit_radius * glm::cos(m_orbit_progress), 0.0f);
}

glm::mat4 Object::GetModel()
{
    setOrbitTranslation();
    return model;
}
```

Each object has a reference to the object we want it to pivot around. I decided to implement it in a similar way that game engines handle parents. The code is specifically tailored to orbiting to make things easier. When the model matrix is requested by something, the object sets its position in the world to be the pivot's position, and then adds the relative position it would be in orbit based on the m_orbit_progress. The base object rotates on the XZ plane, but the cube object has settings to change that.

An example of how the stack would put planets in the correct place, is that we would have the Moon, which calls for the Earth's model matrix, which calls for the Sun's model matrix, and the matrix' position is moved based on the unraveling of the other two matrices positions.

Updates
The m_orbit_progress is updated in the Update function of the Object class. The translation is set on the call to the GetModel().

Rendering
All of the rendering is done the same as the Object class' rendering picture.

Part 3: Camera Movements
The camera movements are done by the engine's cursorPositionCallback function, which tracks the delta of the mouse. Based on the difference of the x and y values, the engine updates the camera's yaw and pitch according to the sensitivity. It then tells the camera to update its matrix.

Part 4: Graphics/Engine Pipeline Changes
The engine was not changed at all, other than removing the input to move the objects.
The graphics object also wasn't changed, but I did make it easier to add/remove objects to be rendered instead of having to add them into the code every time I wanted a new object. This was done by creating an objects list, and each time we add an object, we add it to the objects list. The objects list is in the graphics class, and essentially loops through each object in areas required for rendering, like when we call for them to render, or when we want them to update. The graphics pipeline handles creating and setting of variables for each object, parenting, as well as adding them to the rendering list.