Jerzy Jaskuc **13.11.2020**

Computer Graphics Mid-Term Report

In this report I will go over some of the more important features of the code, explaneing the ideas and design behind them.

Shaders

In addition to shaders given to us as an example, I also created one extra vertex and fragment shader for the plane, since I wanted to have a plane coloring different to the ant. The vertex shader uses pretty much the same principle and uniform variables as the given one, so that we have an option to apply transformations to the plane if needed. However, in addition to that it also takes in the texture coordinates that are then passed to fragment shader. All that fragment shader does, is to apply the texture to the passed coordinates, using the 2D sampler.

In order to interchange the use of the shaders, I created an extra function that compiles these shaders for texture loading and saves its ID in a different variable. The texture shaders are not made specifically for plane and can be reused to apply texture to anything.

```
// Set up the shaders
GLuint shaderProgramID = CompileShaders();
GLuint textureShaderProgID = CompileTextureShader();
```

Mesh Loading

Currently there are 8 meshes in the scene. A plane, body of an ant and six of its legs. All of the meshes are stored as .obj files and loaded using *Assimp* library. Plane object also has texture coordinates loaded with it, which I applied in Blender myself (apparently, *Assimp* does not increase *mNumTextures* when loading .obj files. Texture coordinates are loaded nonetheless). I store the objects in a vector, since before implementing hierarchies I was drawing them in a loop, however I plan to store them in the struct object and store multiple structs in a vector later, to support multiple ants. I structured the code in a way that will allow encapsulating all the processes if needed, which in turn will make a lot of code reusable.

```
#pragma region SimpleTypes
typedef struct mData
{
    size_t mPointCount = 0;
    std::vector<vec3> mVertices;
    std::vector<vec3> mNormals;
    std::vector<vec2> mTextureCoords;
    GLuint VAO;
} ModelData;
#pragma endregion SimpleTypes
```

```
// Buffers
std::vector<GLuint> vVBOs;
std::vector<GLuint> nVBOs;
std::vector<ModelData> meshes;
```

Texture Loading

To load a texture I use lightweight *stb_image* library. For plane texture I use linear filtering, since it's more important to not have pixelated large plane texture, rather than having details on it. After loading the texture I save it and it's associated VBO, so that I can bind it later when needed. Currently texture loading is specialized for plane grass texture (linear filtering and repeating the pattern), but with minor adjustments it can be used for other things, as its not hardcoded.

Hierarchies and transformations

Leg Movements

In terms of leg movements, it's one to many hierarchy model with just one level of depth. That means that all legs are children of a body object, but are also the deepest children on the hierarchy

As the ant moves, I increase the angle of rotation for each leg. In order for legs to not rotate 360 around them, I utilize cos and sin functions on the angle of rotation first, before passing it into *glm::rotate()*. However, since sin and cos will return a value between 1 and -1, I multiply it by a factor of 15, to give it a meaningful value in radians. I also control the leg rotation speed by multiplying the angle change by *leg_speed* variable.

For the leg movement to look sort of natural I interchange sin and cos functions, as well as passing the angle as a negative or positive value. From watching an ant in a slow motion, I realized that middle legs usually work as support, while front and back ones are doing the actual movement, exactly what I tried to recreate during the forward/backward movement. For the turning, middle legs are more motion active and are supposed to help with the turning, and so I used different leg angle change for the middle legs to try and reflect that.

Another caveat was the fact that originally models are offset on the scene, away from the scene origin. So in order to rotate them around their own axis I had to translate them to the scene origin first, rotate them and translate them back. I took the offset from object translation coordinates in blender scene.

Here is an example of one of the legs transformations.

```
glm::mat4 legR(1.0f);
legR = glm::translate(legR, glm::vec3(x_off, 0, 0));
legR = glm::rotate(legR, glm::radians(cos(glm::radians(up_rota)*leg_speed) * 15.0f), glm::vec3(0, 1, 0));
legR = glm::translate(legR, glm::vec3(-x_off, 0, 0));
upLeftLegModel = upLeftLegModel * legR;
// Connect to parent
upLeftLegModel = model * upLeftLegModel;
// Update and draw
glUniformMatrix4fv(matrix_location, 1, GL_FALSE, glm::value_ptr(upLeftLegModel));
drawPart(antPartIndex++);
```

Rotation and translation of the main body

One of more challenging things was the fact that as the ant moves away from the origin, the rotation of the parent body would be done with respect to scene origin and not with respect to the object origin. The fix for that is doing the transformations in a proper order, where we do rotation before translation. However with that, it makes it very hard to translate the ant in the direction that its facing, by simply translating by the offset from the key input. In order to overcome that, I calculate the *x* and *z* translation offset based on cos and sin functions, since as the ant rotates, one of them will grow larger and the other one smaller, proportionally.

```
trs_x += 0.08f * cos(glm::radians(rotate_y));
trs_z -= 0.08f * sin(glm::radians(rotate_y));
```

And the main body transformations would look like this.

```
glm::mat4 model(1.0f);
glm::mat4 T(1.0f);
glm::mat4 R(1.0f);
glm::mat4 S(1.0f);

R = glm::rotate(R, glm::radians(rotate_x), glm::vec3(1, 0, 0));
R = glm::rotate(R, glm::radians(rotate_y), glm::vec3(0, 1, 0));
R = glm::rotate(R, glm::radians(rotate_z), glm::vec3(0, 0, 1));

S = glm::scale(S, glm::vec3(scale_x, scale_y, scale_z));
T = glm::translate(T, glm::vec3(trs_x, trs_y, trs_z));

model *= T * R * S;
```

Camera movements

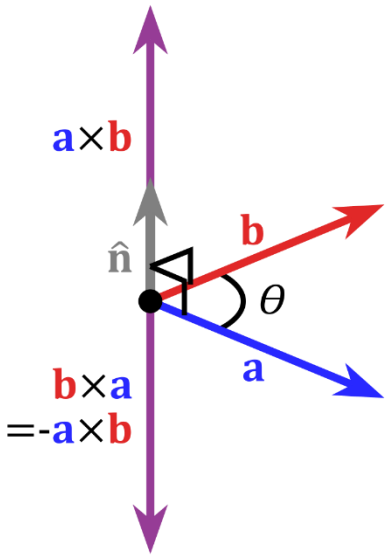
For camera I use perspective projection with view set up by *glm::lookAt()* function, which makes it really easy to control the position of the camera. Camera target is defined by using a forward vector added to the current camera position. This way to move camera forward or backwards we can simply add or subtract this forward vector to camera position, modified by camera speed variable.

```
// Cam forward
else if (key == GLUT_KEY_UP) {
    cameraPos += forwardVector * move_speed;
}
// Cam backwards
else if (key == GLUT_KEY_DOWN) {
    cameraPos -= forwardVector * move_speed;
}
```

```
glm::vec3 forwardVector = cameraTarget;
```

```
// View
glm::mat4 V(1.0f);
V = glm::lookAt(cameraPos, cameraPos + cameraTarget, glm::vec3(0.0f, 1.0f, 0.0f)); //position, target, up direction
```

For right and left movements I use cross product of y-axis unit vector using the rule on the image below.



Basically, since our forward vector in reality is a unit vector looking forward (starting with -z), then by getting cross product with y-axis unit vector we get either left or right unit vector (with respect to current camera angle), depending on the order of the vectors in the multiplication.

For camera rotation with mouse there had to be a bit more logic implemented. First of all when we left click, the program saves the original coordinates of the click. Then the camera angle offset is calculated based on the change of coordinates from the offset. This process is done every frame for the smooth experience. Then for each axis of the direction, I take sin and cos of the camera x and y angles (depending on the axis respectively) and calculate and new, normalized forward vector from that.

Additionally I also restrict y angle change to max 90 to -90 degrees, to prevent unnatural motions and flipping of the camera from going too much up or down.

```
// glut origin is top left
if (clickOriginX >= 0) {
    angleChangeX = (x - clickOriginX) * mouseSens;
    angleChangeY = (clickOriginY - y) * mouseSens;

    // update camera's target angles
    x_angle += angleChangeX;
    y_angle += angleChangeY;

    clickOriginX = x;
    clickOriginY = y;

    if (y_angle > 89.0f)
        y_angle = 89.0f;
    if (y_angle < -89.0f)
        y_angle = -89.0f;

    glm::vec3 direction;
    direction.x = cos(glm::radians(x_angle)) * cos(glm::radians(y_angle));
    direction.y = sin(glm::radians(y_angle));
    direction.z = sin(glm::radians(x_angle)) * cos(glm::radians(y_angle));
    cameraTarget = glm::normalize(direction);
}
glutPostRedisplay();
```

Video Link

Link to a video demo. https://youtu.be/KaBpIf4f64g