**Graphics Coursework 2**

**Camera/Projection**

The camera system has been implemented as a base Camera class, with both a Orbit and FPS camera subclass.

The base class is made of the following variables

* Camera Position
* Camera Target (center of the scene)
* Up Vector
* Right Vector
* Yaw
* Pitch
* Field of View

Within the class we have a function which we use to get the view matrix of the camera, which makes use of the LookAt function which has been implemented to create a viewing matrix from an eye point (camera position), the target, and the Up vector. These three vectors are used to construct a et of orthonormal basis vectors that define how the camera coordinate system is oriented in relation to the world coordinate system.

Vec3f zAxis = normalize(position - target);

Vec3f xAxis = normalize(cross(normalize(worldUp), zAxis));

Vec3f yAxis = cross(zAxis, xAxis);

We create an affine translation matrix to transform vertices from camera space to world space which is a linear transformation followed by a translation. The linear transformation is comprised of the transposed orthonormal basis vectors, and the translation is made of the negated translation components. We then multiply these matrices together as follows to create our lookAt matrix which will function as the view matrix for our camera.

translation(0, 3) = -position.x;

translation(1, 3) = -position.y;

translation(2, 3) = -position.z;

Mat44f rotation = kIdentity44f;

rotation(0, 0) = xAxis.x;

rotation(0, 1) = xAxis.y;

rotation(0, 2) = xAxis.z;

rotation(1, 0) = yAxis.x;

rotation(1, 1) = yAxis.y;

rotation(1, 2) = yAxis.z;

rotation(2, 0) = zAxis.x;

rotation(2, 1) = zAxis.y;

rotation(2, 2) = zAxis.z;

return rotation \* translation;

Within our main program loop, we request the camera view matrix for each frame as well as calculating the perspective projection matrix from the camera’s FOV and the window size as follows, and both of these are passed to the shader to create our scene.

projection = make\_perspective\_projection(makeRadians(fpsCamera.getFOV()), fbwidth / float(fbheight), 0.1f, 100.0f);

To implement an FPS camera, we have also added functions to move the camera position.

Within the main program, we have setup a key callback to track the WASD-EQ keys being pressed, which then call the move function. The fpsCamera.move function simply adds an offset position to the camera’s position variable and then calls the updateCameraVectors function to update the rest of the variables (covered later). This offset position is calculated by multiplying the movement sensitivity variable by the time between this frame and the last to see how far the camera should move. Depending on the key pressed, we then times this by the positive or negative version of the cameras look (forwards or backwards), right (left or right) or up (up or down) vector to move the camera in the correct direction. For example, to move right:

if (glfwGetKey(window, GLFW\_KEY\_D) == GLFW\_PRESS)

fpsCamera.move(MOVE\_SPEED \* (float)elapsedTime \* fpsCamera.getRight());

We can alter the movement sensitivity by pressing control or shift.

The camera can also rotate/look around. This is done by tracking the cursor position for each frame. In order for this to work, we have disabled the cursor, and for each frame we then set the cursor position to the center of the screen. We can then see how much the cursor has moved from here each frame, and pass this offset to our fpsCamera.rotate function.

The rotate function converts the x/yaw offset to radians and then adds this to the camera’s current yaw variable, and converts the y/pitch offset to radians and adds this to the cameras pitch variable.

The pitch is clamped between -0.5pi + 0.1 radians and pi/2-0.1 radians to keep the up and down rotation within a functional range. Again, we then update the camera vectors using our function which is explained below.

The updateCameraVetors function calculates the view direction vector from the yaw and pitch as follows, converting the spherical coordinates to cartesian coordinates as explained at <https://en.wikipedia.org/wiki/Spherical_coordinate_system> .

look.x = cosf(mPitch)\* sinf(mYaw);

look.y = sinf(mPitch);

look.z = cosf(mPitch) \* cosf(mYaw);

mLook = normalize(look);

We then recalculate the right and up vector, and the camera target position.

mRight = normalize(cross(mLook, WORLD\_UP));

mUp = normalize(cross(mRight, mLook));

mTargetPos = mPosition + mLook;

For reference the normalize function is implemented as follows:

float lengthV = length(aVec);

return(Vec3f{ aVec.x / lengthV, aVec.y / lengthV, aVec.z / lengthV });

**Object Loading**

Objects are loaded into a custom mesh class through the RapidOBJ loader which parses the file.

Our mesh class has vectors which contain vectors for each of the positions, normal, texture coordinates and colors defined in the OBJ file. We first read the information from the OBJ into these, and then create the VBO’s for each of these and pass the information to them.

glGenBuffers(1, &mvbo);

glBindBuffer(GL\_ARRAY\_BUFFER, mvbo);

glBufferData(GL\_ARRAY\_BUFFER, positions.size() \* sizeof(Vec3f), positions.data(), GL\_STATIC\_DRAW);

Once we have set up each of the VBO’s, we create a VAO and load the data for each attribute into it in the following fashion.

glBindBuffer(GL\_ARRAY\_BUFFER, mvbo);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 0, 0);

glEnableVertexAttribArray(0);

We can later use our draw call for the mesh object which will bind its VAO and call glDrawArrays to draw the object within the scene.

glDrawArrays(GL\_TRIANGLES, 0, positions.size());

**Textures**

To handle textures, we have implemented a custom texture class with methods to load, bind and unbind textures.

To load textures, we start by loading the image data into an unsigned char pointer using the stbi library as follows.

unsigned char\* imageData = stbi\_load(filename.c\_str(), &width, &height, &components, STBI\_rgb\_alpha);

This loads the image inverted, so we iterate through the pixels in the image and flip them from top to bottom.

We then call glGenTextures(1, &mTexture); and then bind the GLFL texture within the texture object, and configure the texture wrapping/filtering options for the texture. Once we have configured the texture, we call glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA, width, height, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, imageData);to map the image data bits to the GLFL teture that we have created.

**Complex Object**

The complex object that has been created is a fence. This was done by combining several cylinders and cubes. Three cubes for the “walls” of the fence, and six cylinders for the “bars” of the fence. First the code for creating cylinders (make\_cylinder) and cubes (make\_cube) was written, with help from exercises G3 and G4. Additionally, a function (make\_fence) has been made for creating the fence, this calls the make\_cylinder, and make\_cube functions using the relevant transformations to create the design of the fence. Furthermore, this function takes the number of fences, which replicates a single fence several times. It can also have a preTransform, which can change the position of the fence.

Image X below shows what this object looks like.