Lab 3 - Animation

Introduction
In this tutorial we are going to make things move, or animate. The goals are to be able to move the camera to look at things from different angles, and to make a part of the scene move as time progresses.

In this laboration, we will use a *Box* class (defined in "Box.h"), that when instantiated creates the buffer objects needed and that has a draw() method which renders the vertex array object. Look at this class and try to understand what happens. It does exactly what you have done in previous labs to draw a number of quads that make up a box.

Run the program, press mouse buttons and move the mouse around and see what happens. Then look through the code and make sure you understand it all so far.

Assignment: What function is run when you press a mouse button? _____

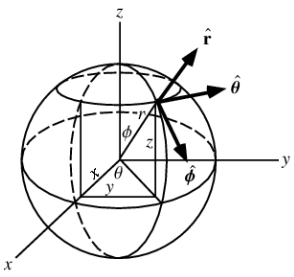
Assignment: What function is run when you move the mouse with a button held? _____

In this laboration, we finally move completely into the 3d space. That is, we will have a *Model*-, a *View*- and a *Projection* matrix which will be used to transform all vertices from their model-space coordinates to their window coordinates. In the current code, the *Model* and *View* matrix are both just the identity-matrix. We will implement the camera by modifying the *View* matrix and then add some animation by modifying the *Model* matrix.

Notice that the three matrices are multiplied together in the display() method and sent along to the vertex shader with this code:

```
// Concatenate the three matrices and pass the final transform
// to the vertex shader
float4x4 modelViewProjectionMatrix = projectionMatrix *
                                     viewMatrix * modelMatrix;
int loc = glGetUniformLocation(shaderProgram,
                               "modelViewProjectionMatrix");
glUniformMatrix4fv(loc, 1, false, &modelViewProjectionMatrix.c1.x);
```

Camera Movement
We will create a simple pivoting camera that is able to rotate around a fixed point in space, and zoom in and out. We represent the camera using spherical coordinates (r, θ, ϕ) instead of the cartesian *(x,y,z)* coordinates.



In this representation, *r* is the distance from origo to the camera, ϕ is the angle between the up-vector and the vector pointing at the camera and θ is the angle that vectors projection on the xz-plane makes with the "z" vector.

First then, we will define these variables globally so insert this somewhere in the beginning of main.cpp:

```
float camera_theta = 0.0f;
float camera_phi = M_PI/2.0f;
float camera_r = 8.0;
```

We will use mouse input to control this movement, which is we can do using GLUT. In the project this is set up using these two lines:

```
glutMouseFunc(mouse);          // callback function on mouse buttons
glutMotionFunc(motion);        // callback function on mouse movements
```

This instructs GLUT to send mouse button events to the function `mouse` and motion, when a button is held down, to the function `motion`. Study these two functions, in `mouse` there is code to handle button presses and set the variables, which represent the state of the mouse buttons:

```
bool leftDown = false;
bool middleDown = false;
bool rightDown = false;
```

Now let's look at the function `motion`. In this function we will translate the movement of the mouse into camera movement. This is done depending on which mouse button is held down. First we will affect the distance to the center of attention, in the `camera_r` variable, if the middle button is pressed, add:

```
if (middleDown)
{
      camera_r -= float (delta_y) * 0.3f;
      // make sure camera_r does not become too small
      camera_r = max(0.1f, camera_r);
}
```

after the line with `printf("Motion: %d %d\n", x, y);`

Next want to modify the two angles in the representation, θ and ϕ, this is done if the left button is down, add:

```
if(leftDown)
{
      camera_phi  -= float(delta_y) * 0.3f * M_PI / 180.0f;
      camera_phi = min(max(0.01f, camera_phi), M_PI - 0.01f);
      camera_theta += float(delta_x) * 0.3f * M_PI / 180.0f;
}
```

Next we must use this information to create view matrix for use when rendering. Find the line in *display()* where the View matrix is initialized and replace it with:

```
float3 camera_position = sphericalToCartesian(camera_theta,
                                              camera_phi,
                                              camera_r);
float3 camera_lookAt = make_vector(0.0f, 0.0f, 0.0f);
float3 camera_up = make_vector(0.0f, 1.0f, 0.0f)
float4x4 viewMatrix = lookAt(camera_position,
                             camera_lookAt, camera_up);
```

Now run the program and make sure you can move and zoom the camera using the mouse.

Animation

In this part of the tutorial we want to animate part of the scene, to do this we will add a second box, drawn using the same vertex data as the already existing box. We will achieve this by uploading a new `modelViewProjectionMatrix` matrix. Directly after the call to `myBox->draw()` add:

```
// Draw a second box
```

```
float4x4 r = make_rotation_y<float4x4>(currentTime*M_PI*0.5f);
float4x4 t = make_translation(make_vector(8.0f, 1.0f, 0.0f));
modelMatrix = r * t;
modelViewProjectionMatrix = projectionMatrix * viewMatrix *
                            modelMatrix;
// Update the modelViewProjectionMatrix used in the vertex shader
glUniformMatrix4fv(loc, 1, false, &modelViewProjectionMatrix.c1.x);
myBox->draw();
```

Assignment: Notice how the Model matrix is a concatenation of a rotation and a translation. What happens if the order is reversed (that is `modelMatrix = t * r`)? Try to predict the result before you try. What happened? (draw if it helps you)

_____

_____

For the fun of it, create a scaling matrix that varies with time:

```
float4x4 s = make_scale<float4x4>(sin(currentTime * 5.0f)*
                                  make_vector(1.0f, 1.0f, 1.0f));
```

And multiply the modelMatrix with it:

```
modelMatrix = r * t * s;
```

Run the program again and enjoy the result.

Assignments

Now, try to do the following things on your own:

- Add an animation (using `currentTime`) that only happens when the right mouse button is held down.
- Now make it do this for 3 seconds after each left button press.
- *Optional:* Add a new uniform float variable in the fragment shader called alpha, and use this to animate the transparency of the second triangle. For example using a periodic function such as sin.

When done, show your result to one of the assistants. Have the finished program running and be prepared to answer some questions about what you have done.