# Lab 3

Arve Nygård

*Deliver A COMPLETE report explainig how, showing your math developed step by step as well as your results:*

# Plot a cube using 12 triangles. Plot it using your color of choice.

A cube has 8 corners. Instead of duplicating these vertices for each triangle touching that corner, I chose to create two buffers: One vertex buffer with 8 vertices, and one "index buffer" which defines the triangles using indexes pointing to the first buffer. This requires changing the call from `glDrawArrays` to `glDrawElements`.

I also noticed that we are basically drawing the excact same scene each time, just with a different animation. So i removed RenderScene1...5, and replaced these functions with a single function `RenderScene`. I then chose to handle changing the animation (and resetting transformation matrices) in the `Idle` function.

### RenderScene()

This function does a few basic things: * Set up *projection matrix* * Compute MVP matrix based on the previously created *projection matrix*, as well as the *Model* and *View* matrices defined by `Idle()`. * Set active buffer to our index buffer containing cube definition * draw it

## Scene 1: rotatethe cube in X

This is a "relative transformation", where we have a given change each frame, but don't care about the absolute value of the transformation. We use the previous frame's Model matrix as a base, and multiplies it with our "delta rotation matrix", before passing it on to `RenderScene()`.

```
[ 1   0    0    0 ]
[ 0 cosθ -sinθ  0 ] x PREV_MODEL_MATRIX
[ 0 sinθ  cosθ  0 ]
[ 0   0    0    1 ]
```

Here, θ is the angle we want to rotate (which is a set value multiplied by the time delta since the last frame).

## Scene 2: rotate the cube in X + Y in two different speeds

Mostly the same as in scene 1, however now we have two rotation matrices we need to apply on top of the previous Model matrix:

```
        Rotate Y axis                                  Rotate X axis
[ cosθ   0 sinθ 0 ]   [ 1   0    0   0 ]
[   0    1   0  0 ] x [ 0 cosθ -sinθ 0 ]  x PREV_MODEL_MATRIX
[ -sinθ  0 cosθ 0 ]   [ 0 sinθ  cosθ 0 ]
[   0    0   0  1 ]   [ 0   0    0   1 ]
```

## Scene 3: Rotate the cube in X +Y+ Z in 3 different speeds

Same as 1 and 2, except another matrix multiplication for Z axis.

## Scene 4: Translate the cube back and forth in X

This is slightly different. Instead of using a relative translation, we are now dealing with absolutes. Using sin(current_time/500), we get a periodic value for our position. However, we cannot use the previous frame's Model matrix as a base, since it contains a position value already, and adding the new position value as an offset would create **HUGE** movements. This is not what we want.

So instead, we use the identity matrix as our base.

```
Model = glm::translate(glm::mat4(1.0f)), glm::vec3(TRANSLATE_X_VALUE, 0 ,0);
```

This gives the following math:

```
Translation_matrix x identity_matrix
[ 1 0 0 X_POS ]   [1 0 0 0]
[ 0 1 0   0   ] x [0 1 0 0]
[ 0 0 1   0   ]   [0 0 1 0]
[ 0 0 0   1   ]   [0 0 0 1]
```

which boils down to

```
Translation_matrix
[ 1 0 0 X_POS ]
[ 0 1 0   0   ]
[ 0 0 1   0   ]
[ 0 0 0   1   ]
```

Hopefully glm::Translate has an optimization that detects the identity matrix and skips the unnecessary matrix multiplication. If not, there is room for optimization here: Define the Model matrix directly.

## Scene 5: Translate the cube back and forth in Z and rotate in X Y Z

This is where it gets interesting: We need to keep the previous rotation state (because we are applying incremental rotation), but translation is still absolute.

So for this scene we will construct the Model matrix by multiplying Translation * Rotation matrices together.

We keep a variable called `glm::mat4 Rotation`, that contains the rotation state of our cube. When constructing the Model matrix for this scene, we do the following:

```
Rotation = glm::rotate(Rotation, angle, rotation_axis) * Rotation
```

We do this for all three axes. The math is basically the same as in scene 1,2 and 3.

We then create our translation matrix (in the code i do this inline), using the identity matrix as a base:

```
Translation = glm::translate(glm::mat4(1.0f), glm::vec3(x,y,z));
```

Finally, we create the Model matrix:

```
Model = Translation * Rotation;
```

Which does the following:

```
Translation_matrix        Rotation_Z               Rotation_Y
[ 1 0 0 X_POS ]       [ cosθ   -sinθ  0  0 ]       [ cosθ   0 sinθ 0 ]
[ 0 1 0   0   ]   x   [ sinθ    cosθ  0  0 ]   x   [   0    1   0  0 ]
[ 0 0 1   0   ]       [   0      0    1  0 ]       [ -sinθ  0 cosθ 0 ]
[ 0 0 0   1   ]       [   0      0    0  1 ]       [   0    0   0  1 ]


     Rotation_X              old_rotation_matrix
    [ 1   0    0    0 ]       [? ? ? 0]
x   [ 0 cosθ -sinθ  0 ]   x   [? ? ? 0]
    [ 0 sinθ  cosθ  0 ]       [? ? ? 0]
    [ 0   0    0    1 ]       [0 0 0 1]
```

This ensures that the translation each frame is absolute, based on sin(current_time), while the rotation in each axis keeps on trucking at a regular pace :)