COSC3000 - Lab 1 Worksheet [2018]

Introduction

This lab aims to provide exercises that help you understand how coordinate systems, transforms and spaces work in a simple OpenGL program. Along the way, the lab will also introduce basic OpenGL rendering using Python.

General instructions.

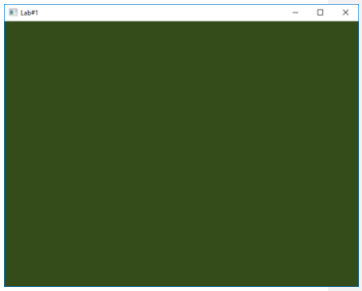
There are quite a few things needed to get even a simple OpenGL program running, and we will cover what is needed in due time. This lab focuses on coordinate systems and transformations, but we need a lot more to draw the examples. To reduce the clutter we will place code that deals with *future* lab/lecture material in a module called 'magic'. You are free, encouraged even, to look at this code, which will be commented to explain what it is used for and how it works, but it will be explained in the future (or it does not need explanation for this course). Conversely, there will be (in later tutorials) a module called 'lab_utils' where code that we have already covered will be placed – for example in this lab we will implement some matrix manipulation code, which will then move into 'lab_utils' so we can use them in the future.

First Time Setup

Follow the instructions on the course web-page.

Let's get started!

Now download lab1.zip and unpack somewhere sensible. When you run the file '`lab1_template1.py`' you should see a green window, like to the right here. If that is not the case, your setup might not be correct, make sure you read through setup instructions on the course website.



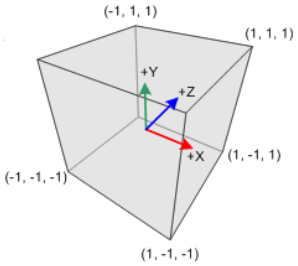
If it still is not working you may need to talk to a tutor. Check the console window for any error messages. If emailing a tutor for advice, **always include the full console output**, to help diagnosing the problem.

Tasks, part 1, 2D coordinate systems and transforms

The first set of tasks involves two coordinate systems:

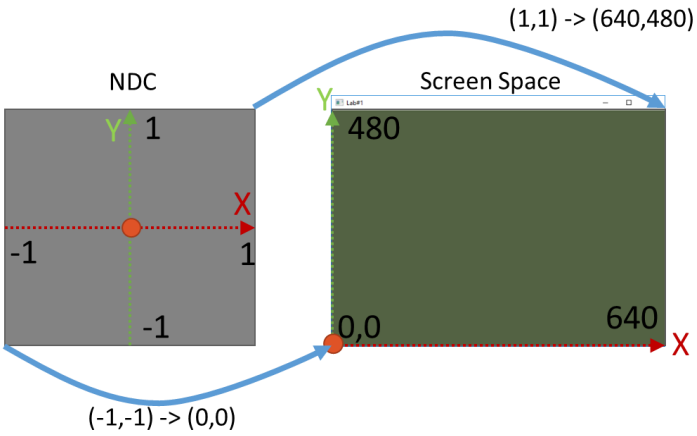
1. Normalized Device Coordinates (NDC), and
2. Screen space.

These are the only two that are pre-defined in OpenGL and we will discuss these coordinate systems in more details in lectures 3 and 4. For now, we will just need to know how they are defined such that we can draw a triangle using the correct coordinate system.



NDC is defined as a 3D cube with each axis is in the range [-1,1]. This is the space that all rendering ultimately hands data over to OpenGL. Any position within this space maps to a position that is *visible* on screen.

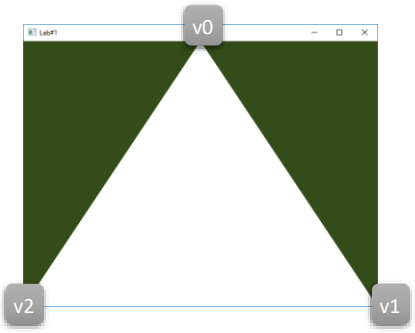
Leaving Z to the side for the moment, the mapping from this space to the screen is simple. The Y coordinate increases from left to right over the screen, and the Y coordinate increases from bottom to top. The mapping from [-1, 1] to pixels is trivial and is controlled using glViewport.



1. Draw a triangle

Ok, finally we have set the stage for the first task. Open the solution '`lab1.sln`', and then the file '`lab1_template1.py`' (or just open the .py file in your favourite editor), and find the declaration of the variable '`g_triangleVerts`':

```
g_triangleVerts = [
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0],
]
```



These define the three coordinates of a triangle in the NDC space. Modify them to produce a triangle that looks like the picture to the right. Before you run the program work out what each corner ***should*** have and write down here (or on a separate piece of paper).

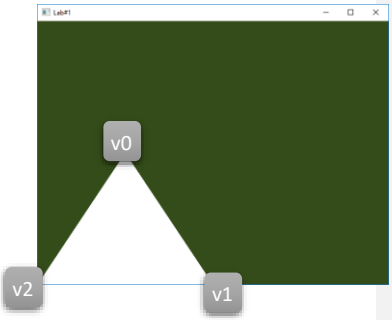
v0: _(, , 0.0)

v1: _(, , 0.0)

v2: _(, , 0.0)

2. Move the triangle to the lower left corner.

Change the coordinates such that the triangle uses only the lower left quarter of the screen.



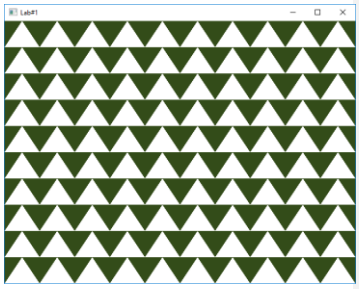
v0: _(, , 0.0)

v1: _(, , 0.0)

v2: _(, , 0.0)

3. Draw more triangles

Write a bit of code that produces something that looks like the image to the right. There are a couple of ways to achieve this. The first is to generate a list of vertices and then draw them in one go, something like the below:

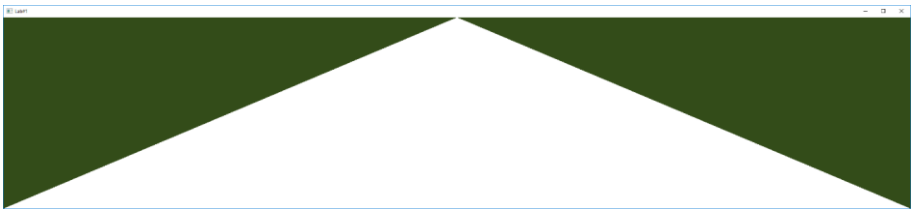


```
g_triangleVerts = []
N = 5
for y in range(-N,N):
    for x in range(-N,N):
        g_triangleVerts.append(???)
        g_triangleVerts.append(???)
        g_triangleVerts.append(???)
```

Another way is to define a single triangle, like before, and then create a loop to make multiple calls to: `magic.drawVertexDataAsTriangles(g_triangleVerts)`, each time with an offset.

4. Define triangles in screen space

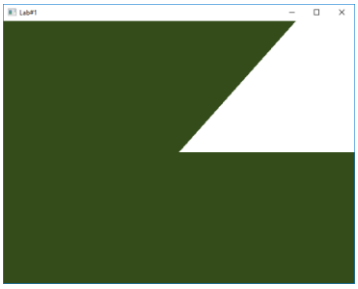
So far we've been defining our geometry (triangles) directly in NDC space. However, this means that the triangles get stretched when we change the size of the window. For some applications, it makes



more sense to define the geometry in **screen space**, and give coordinates in pixels.

```
g_triangleVerts = [
    [320, 480, 0],
    [640, 0, 0],
    [0, 0, 0],
]
```

If we do this with our current setup, we'll get this, as we're only seeing one of the corners of a huge triangle:



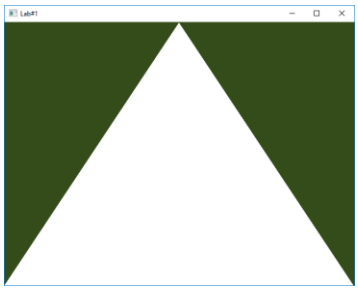
To make this work, we need to introduce a *transformation* to get **from screen** space (where we are now defining our triangle coordinates, call it our *world space* if you will) to **clip space** (which is where OpenGL expects coordinates of geometry to be expressed). This is exactly the reverse operation that OpenGL performs to get from clip space to screen space as defined for glViewport. Let's just inverse this then! To go from NDC we have (for x):

$x_s = (x_{nd} + 1)\left(\frac{width}{2}\right)$ (where we dropped the offset , x, as it is zero), and thus the inverse is: $x_{nd} = 2x_s/width - 1$. Finish implementing this in by completing the function given below, and call that instead of directly calling '`magic.drawVertexDataAsTriangles`'.

```
def drawScreenSpace(width, height, vertices):
    verticesNdc = []
    # transform each coordinate
    for [x,y,z] in vertices:
        verticesNdc.append([???, ???, ???])

    magic.drawVertexDataAsTriangles(verticesNdc)
```

The result should look like to the right, assuming the window size is as given in the program. Now, however, when we resize the window, the triangle stays the same size and shape.



This is because we defined the triangle in screen space – which has a well-defined meaning to us, and implemented the correct transformation to clip space.

5. The same with a Matrix

To start to approach how the real-time rendering pipeline works, lets generalize the function above using matrix algebra:

```
def drawTransformed(transformMatrix, vertices):
    verticesNdc = []
    # transform each coordinate
    for [x,y,z] in vertices:
        xt,yt,zt,wt = transformMatrix * [x,y,z,1.0]
        verticesNdc.append([xt,yt,zt] / wt)
    magic.drawVertexDataAsTriangles(verticesNdc)
```

Here we have assumed a 4x4 transformation matrix (which must be an instance of the class '`magic.Mat4`', which implements the multiplication operator to do matrix/vector or matrix/matrix multiplication). Note how the 3D vertex position is extended to a 4D homogeneous point by appending a 1 (`[x,y,z,1.0]`) – remember from Lecture 2 that this is what enables representing translation in the same matrix as rotation and scale. To get back to 3D we divide by '`wt`' after the transformation (`[xt,yt,zt] / wt`) – which follows from the definition of homogeneous coordinates. As long as we are only using *affine* transformations, such as translation, rotation and scaling '`wt`' will remain 1, but this will change when we get to projections.

The transformation matrix to do the screen to NDC transformation (i.e., exactly the same transformation as the previous task) can be built as:

```
screenSpaceToNdc = make_translation([-1, -1, 0]) *
                       make_scale([2/width, 2/height, 1])
```

Here we create the matrix by multiplying a translation with a scale. The way to think about this is that if we have a matrix: $M = T \times S$ (like the above), then $M \times P$ for some point P, is the same as

$T \times (S \times P)$, i.e., the scale *happens first*. In other words the above constructs a matrix that will **first** scale the point by half the screen size, and **then** translate it by -1.

Implement the two matrix construction functions '`make_translation`' and '`make_scale`', that are given here just as identity matrices and use '`drawTransformed`' to draw the triangle. The result should look the same as in the previous task.

```
def make_translation(x, y, z):
    return magic.Mat4([[1,0,0,0],
                       [0,1,0,0],
                       [0,0,1,0],
                       [0,0,0,1]])
```

```
def make_scale(x, y, z):
    return magic.Mat4([[1,0,0,0],
                       [0,1,0,0],
                       [0,0,1,0],
                       [0,0,0,1]])
```

6. Rotation

Next we'll implement a rotation matrix for rotating points around the z axis. To the right you can find the skeleton, and the definition of how it works can be found in for example wikipedia. You'll need to import `math` for `math.sin` and `math.cos`.

```
def make_rotation_z(angle):
    return magic.Mat4([[1,0,0,0],
                       [0,1,0,0],
                       [0,0,1,0],
                       [0,0,0,1]])
```

We'll use using '`time.time()`' as the rotation angle such that it performs a full revolution every 2π seconds. Let's pop this rotation at the end of the transformation we just made, what happens?

```
rotTfm = make_rotation_z(time.time())
tfm = rotTfm * screenSpaceToNdc
drawTransformed(tfm, g_triangleVerts)
```

In what space is the rotation happening?

Let's change the order, but before you run this, try to predict what will happen: Around what point will the triangle rotate?

```
tfm = screenSpaceToNdc * rotTfm
```

Were you right? Can you explain what happens?

As a final step, try to figure out how to rotate the triangle around its own centre in screen space. You will need to introduce some more transformations. Keep in mind that the rotation matrix rotates **points** around the **origin,** nothing else. So you need to make sure the points are centred on the origin prior to the rotation, and put back after. Lecture #2 describes this process.

```
triTfm = ??? rotTfm ???

tfm = screenSpaceToNdc * triTfm
```

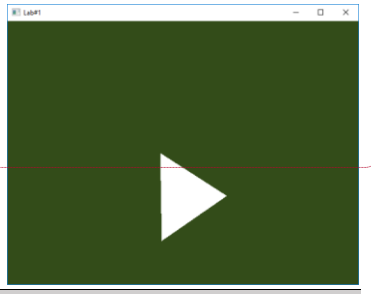
If you get stuck, you are encouraged to refer to the lecture slides, discuss with other students, and ask a tutor after a while!

Hint: Use pen and paper to draw what you think should happen.
Hint: `triCentre = np.average(g_triangleVerts, 0)`, could come in handy!

7. More scaling

To get some more practice with transformations we'll scale the triangle as well. Create a new matrix like this:

```
scale = 0.75 + math.sin(time.time()) * 0.5
scaleTfm = make_scale(scale,scale,scale)
```



This will create a transformation that scales (whatever coordinates it is applied to) over time, between 0.25x to 1.25x in all dimensions, i.e, *uniform scaling* (though we only care about x and y just at the moment).

Insert '`scaleTfm`' into the transformation stack to make the triangle scale around it's centre (as well as rotate).

```
triTfm = ??? scaleTfm ???
```

Commented [OO1]: TODO: insert into tfm stack, clean up things in previous exercises giving names to the different tfms.

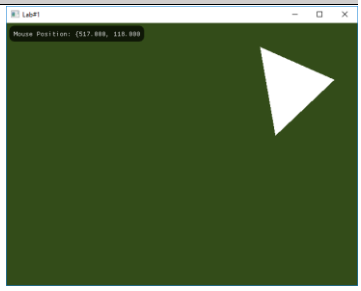
8. Follow the mouse

For extra fun we'll now make the triangle follow the mouse around, the mouse coordinates are tracked in '`g_mousePos`' whenever the mouse is over the window. Modify the transformation using a new translation matrix, to follow the mouse cursor (but keep it spinning and scaling around its own centre!):

```
mouseTfm = make_translation(magic.g_mousePos[0], magic.g_mousePos[1], 0)
triTfm = ??? mouseTfm ???
```

If you've done this correctly, you'll see that it works in the x- direction but is flipped in the y-direction. This is because mouse coordinates are in *yet another* coordinate system. This one is given by the windowing system (Windows in this case) and has the origin in the top-left corner. Since it makes more sense to stick with OpenGL in this case (our '`screenSpaceToNdc`' matches the transformation OpenGL does from NDC), it makes the most sense to flip the mouse Y-coordinate to make it agree with OpenGL's definition of screen space.

```
mouseTfm =
    make_translation(magic.g_mousePos[0], height - magic.g_mousePos[1], 0)
```



With this we now have a rotating triangle that moves with the mouse at its centre.

9. Introducing shaders

As a final exercise before we leave 2D, we will implement the same thing but using a *vertex shader*. A shader is a program that executes on the GPU as part of a programmable stage in the pipeline. The primary output of a vertex shader is coordinates in *clip space*. Clip space is the homogeneous (4D) version of NDC.

We have in fact been using one, hidden within the magic, all along:

```
    #version 330
    in vec3 positionIn;
```

```
    void main()
    {
          gl_Position = vec4(positionIn, 1.0);
    }
```

This program gets executed once for each vertex in our triangle(s), each time the value of '`positionIn`' is a different vertex. All it does is perform the trivial conversion of a 3D **point** to a 4D homogeneous representation thereof, by adding a fourth component which is 1.

The program is not written in Python, but in GLSL (the Open**GL S**hading **L**anguage) – a C dialect specifically for shaders in OpenGL. In the Python program it is represented as a string which is handed over to and compiled by the OpenGL driver.

We will now extend this simple shader to perform the matrix version transformation in the function '`drawTransformed`' that we implemented earlier. This kind of thing is the bread and butter of shaders, so we can make use of built in types and operations. Replace the call to '`drawTransformed`' with the following:

```
magic.drawVertexDataAsTrianglesWithVertexShader(g_triangleVerts, tfm, """
    #version 330
    in vec3 positionIn;
    uniform mat4 transformationMatrix;
    void main()
    {
          gl_Position = transformationMatrix * vec4(positionIn, 1.0);
    }
    """)
```

The magic takes care of compiling the code and setting the transformation argument to the supplied value. To see how this happens, take a look inside the magic module. A key part of this program is the variable declared as '**`uniform`**', in GLSL, this indicates a value which is the same for all the instances of the program – i.e., for all of the three vertices, '`positionIn`' is different, but '`transformationMatrix`' is the same. '`transformationMatrix`' is set by the function '`drawVertexDataAsTrianglesWithVertexShader`' (to the value of '`tfm`') before drawing the triangles. '**`mat4`**' is the built-in 4x4 matrix type, so this should produce the same result as '`drawTransformed`'**.** What happened to the division by 'wt', I hear you ask! Good question! This is taken care of by the hardware, the shader writes the final clip space position as a 4D homogeneous coordinate to '`gl_Position`', the division takes place somewhere between here and the rasterization stage.

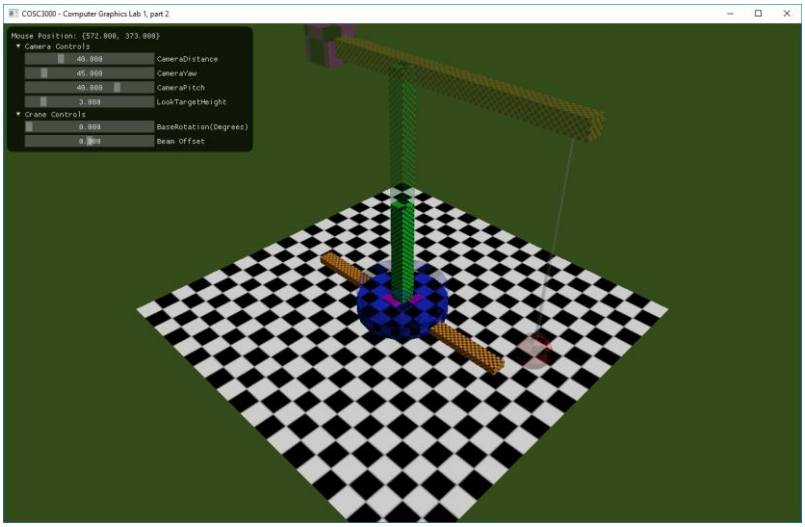
The key achievement here is a solid start on a general real-time rendering program. We now have a shader that runs on the GPU, with all its parallel power, that can perform any arbitrary transformation!

Part 1 Summary

Stepping back for a moment, we have defined a very simple rendering pipeline. We have defined a 'world space' coordinate system – i.e., screen space. We implicitly also defined a 'model space', namely the space where the triangle is centred. We also implemented a projection in our transformation from world space to NDC (or clip space in the last version) – this is an example of a very simple *orthographic* projection. We also managed to create our first shader, which is going to be important as we go.

Part 2: Hierarchical transformations in 3D

Now let's switch over to 3D and play around with hierarchical transformations a bit more. For this we will switch to the file `lab1_template2.py`. It makes use of a lot more magic that enables rendering 3D models with textures and basic shading under perspective. We will not concern ourselves with those things for now, but instead just practice using the transformation tools we're built up so far. Running the template should look like this:



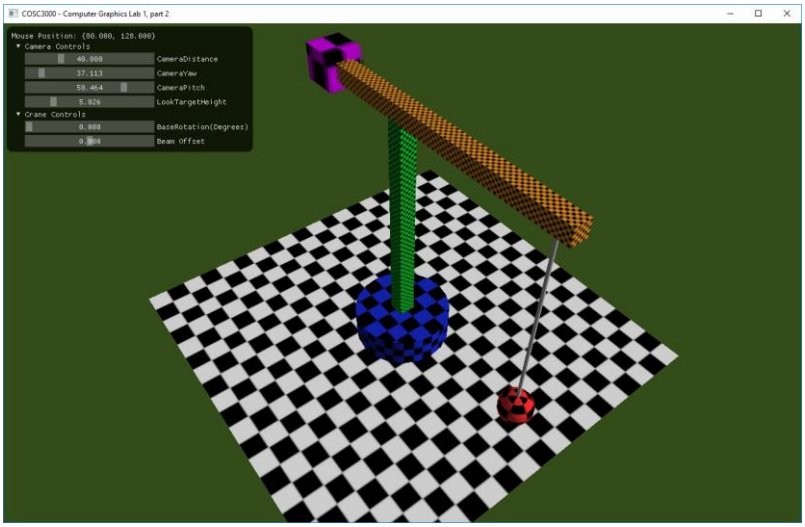
You can change the view point using the camera controls, try it to see how it works. These parameters are used to construct the camera location and point where it should look towards (edited for readability):

```
Ry = lu.make_rotation_y(math.radians(g_cameraYaw))
Rx = lu.make_rotation_x(-math.radians(g_cameraPitch))
eyePos = lu.Mat3(Ry * Rx) * [0.0, 0.0, g_cameraDistance]
```

The magic provides functions to create the view matrix ('`magic.make_lookAt`') and projection ('magic.make_perspective') matrix, we will cover these in the next lab, so no need to go too deeply into that now. However, note that there is nothing particularly special about the matrices they return, they are just 4x4 matrices that can be combined with others to form the full transformation needed. In particular, note that '`viewToClipTfm * worldToViewTfm`' defines a transformation from world space to clip space – in effect what ended up with in the first part of the lab. All the data is different, but the principle is the same.

10. Assemble the Crane

As you can see above, the template has a faint version of the crane as it should look. This is there as a help and is done with the last few lines of '`renderFrame`'. You can also see the jumble of parts in the centre. This is because they are all rendered with an identity model-to-world transform. It is your first task to fix this by creating appropriate model-to-world transforms for each of the parts such that it matches the reference crane. The result should looks something like this:

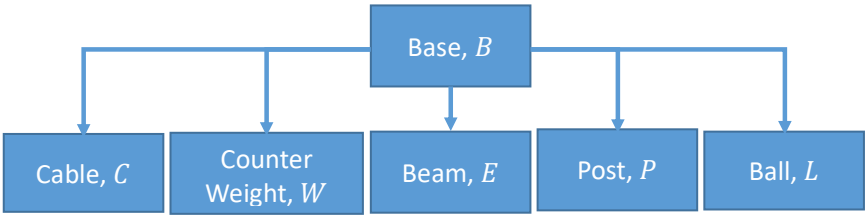


The astute of eye can see that the counter weight is somewhat offset, but we're ok with that. The rest look spot on.

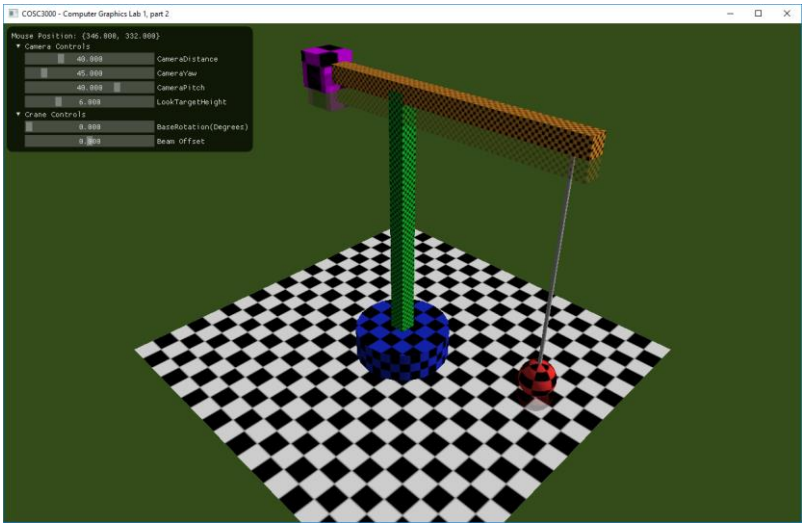
Hint: The positioning only involves translation.
Hint: No decimals are needed.

11. Make it rotate with the base

Now we want to make the whole crane rotate with the base. First make the base rotate, using the value (in degrees) of the variable '`g_baseRotationDeg`' around the up axis (the Y axis). Then make this a hierarchical scene. In this task we will use the flattest hierarchical model where all the other parts inherit the transformation of the base. This is the simplest way, and meets the goal of making everything rotate with the base.



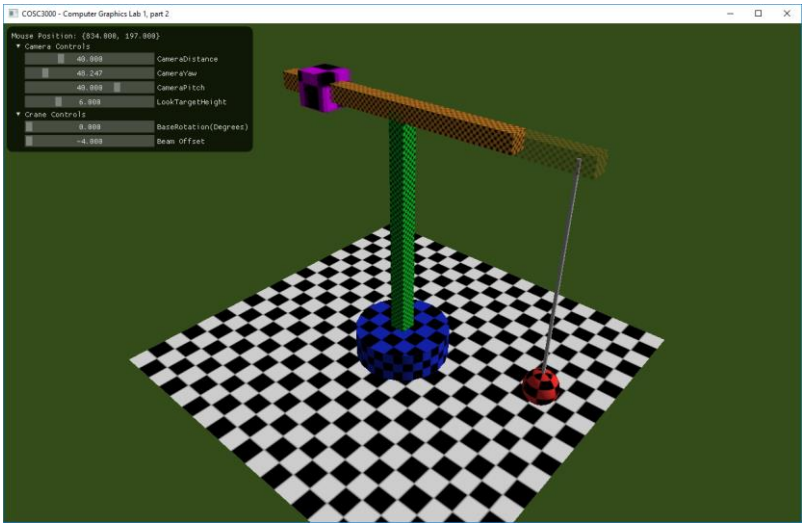
Now, if you, like me just went ahead and did this without modifying the individual transforms, you would see something like the below. Everything is offset upwards. Why? By how much? What should we do to fix this? (There are several solutions actually). Can you think of a general solution when we want to insert a transform in a hierarchy but retain the world space transform of the object?



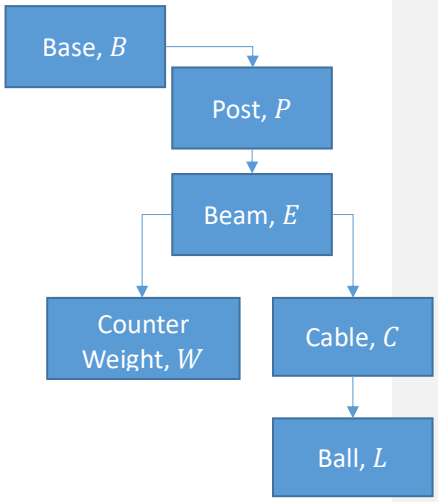
Once fixed it should be back to how it was before.

12. Animate the beam & kids

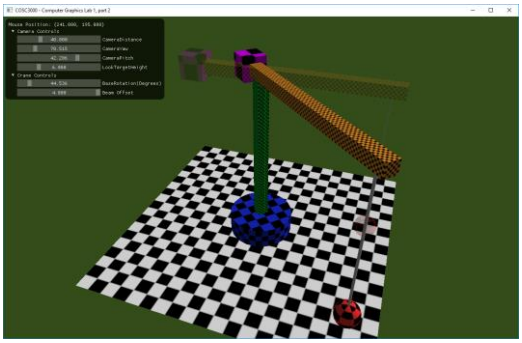
To animate the beam, modify its model transform to translate back and forth using the value of '`g_beamOffset`'. Now if you run the program before changing anything else, it ought to look like this. The UI slider moves the beam and nothing else.



Not quite what we wanted, so we'll keep going. Recall from Lecture 2, the more complex scene structure shown to the right.

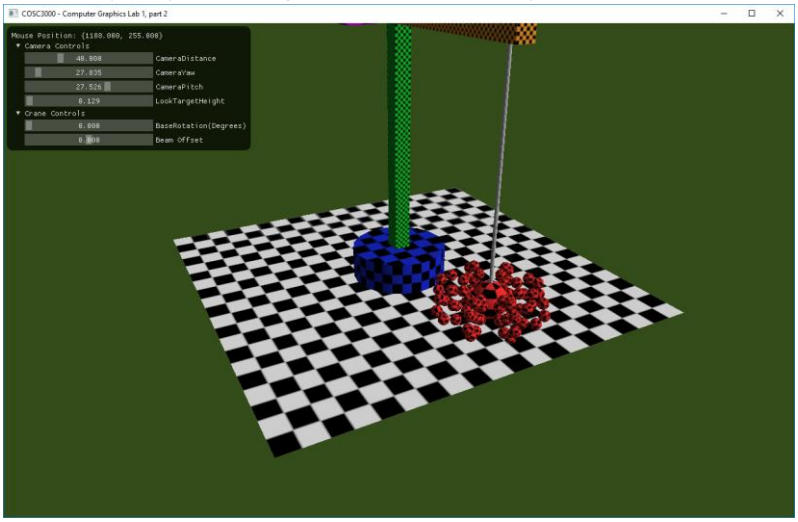


If we implement this, most 'realistic' (as in most like how a real crane would be stuck together) structure things will behave as we'd expect. However, in graphics we can take shortcuts and for example skip the Post->Beam, and Cable->Ball steps. It is up to you, as long as the final version works and looks as expected.



13. Play around!

Now go crazy and add more parts that are transformed relative to each other. Why not a bunch of smaller balls that spin around the one already there? Make sure to investigate when the visual result is not what you expected! Doing this twice (in a nested loop) looks like this:



Part 2 Summary

Hope you enjoyed the labs, and feel confident in using transformations. Don't hesitate to talk to a tutor to discuss any questions you might have, or anything that seems unclear or mystical!