What is a scene graph?

- A set of nodes that hold transformation matrix data and pointers to other nodes and geometry
- Render multiple instances of the same geometry without having to construct multiple copies of that geometry
- Traverse a directed tree of transformations to render shapes
Traversing a scene graph

- Post-order traversal
- Recursively traverse each node’s children until you hit a leaf node
- Render any geometry at a node AFTER you render its children
- Since this function is recursive, it implicitly pops transformations off the transformation stack when it exits a recursion level

Traversal pseudocode

```cpp
Traverse(Node N, Matrix T)
{
    T = T * N.transform
    for each (Node CHILD in N.children)
    {
        Traverse(CHILD, T)
    }
    if (N.geometry != NULL)
    {
        Draw(N.geometry, T)
    }
}
```
Traversals visualization

- We begin the traversal at the root node.
- We put the root node’s transformation matrix onto our virtual stack of transformations.
We continue to the body node

Even though the body node has geometry, we won’t render it until we’ve processed all of the body node’s children.
Traversing visualization

- We’ve reached a leaf node, so we don’t have any nodes to traverse down
- We render the left arm node’s geometry using the compounded transformations of Root, Body, and Arm_L
- Remember, transformations are applied in order of “proximity” to geometry
  - e.g. Arm_L.transformation is applied before Body.transformation
Traversal visualization

- We continue to the other child node of the body node.
- It also has no children, so we render its geometry next.
Traversals visualization

- Our traversal has processed all of the body node’s children, so we now render its geometry.
- After this, our traversal returns to the root node, but since the root node has no geometry our traversal then terminates.
Traversal: Clicker

What does this scene graph produce?

Assume a transformation order of:
Translate * Rotate * Scale * Geometry
I’ve skipped to the first actual instance of geometry rendering.

We begin by applying the transformations at Node 3 to the square. It translates by 1 in the Y direction, so that the square now rests one unit above the origin.

Its local origin is therefore at its bottom edge.
Next, we apply the scale of Node 2. Remember, we’re going in the order of $T*R*S*Geometry$.

Since the square’s local origin is now its bottom edge, it scales up and away from the origin rather than directly outwards.
Now the rotation of Node 2 is applied.

Once again, since the square’s local origin is at its bottom edge, the square rotates about that point rather than its center.
Transformation process

Now we apply the translation of Node 2, which moves the square 2 units along the Y axis.

The square now sits high above the origin.
Transformation process

Next we’ll add Node 1’s geometry to the scene.

Note that in practice, your geometry is transformed, then rendered. We’re just adding the geometry to the scene first for illustrative purposes.
Transformation process

Now we apply the transformations of Node 1.

First up is its scale. The circle, which has just been added to the scene, sits at the origin so it scales away from the origin uniformly.

The square, on the other hand, moves even farther away from the origin when it scales because it sat three units away before the scale was applied.
Transformation process

I’ve skipped the rotation of Node 1 because it’s zero on all three axes.

Finally, we translate all geometry 2 units along the X axis. Now the circle sits with its leftmost edge at the origin.
Clicker Round 2

What does this scene graph produce?

Assume a transformation order of:
Translate * Rotate * Scale * Geometry
Let’s take a look at the homework

- The base code we’ve given you has many unfamiliar functions
- Many of these have to do with sending information to OpenGL shaders to draw your geometry
- Let’s go through them together to understand how to get your own code to interface with what we have
The OpenGL Pipeline

Vertex data

Vertex shader (Doesn't actually "shade")

Primitive assembly (Shown solid for clarity)

Rasterization

Fragment shader

Frame buffer
Why transform this guy in a shader?

The Stanford Dragon: 438,929 vertices!

http://graphics.stanford.edu/data/3Dscanrep/
Why are they called “shaders”??

- Original shaders were fragment-based and written in assembly code
  - Computed final color of a fragment, hence the name “shader”
- Nowadays, any programmable part of a GPU is called a “shader” because old habits die hard
- Perhaps think of the term “shader” as being synonymous with “customizer”
Shader types

- Vertex shader
  - Used to transform the input geometry efficiently
- Fragment shader
  - Used to determine the final color of a fragment (layer of a pixel)
- There are also tessellation shaders as of OpenGL 4.0, but we’re not going to use those in this course
  - Basically used to control geometry level of detail
How do you pass data to shaders?

- You use Vertex Buffer Objects (VBOs)
  - Analogy: a VBO is a bus into which you load your vertex data, which then drives off to OpenGL Land (i.e. your GPU), where you can no longer access your vertex data and you can’t see what happens to it.

- VBOs send per-vertex data to OpenGL, such as position, color, normal, and index

- For more efficient draw calls, you can use index buffers as well as vertex buffers