Scene Graphs and Instancing

So far we have studied how to take objects and transform them utilizing the model matrices to take them into their final positions in world space. While the concatenation of different matrices gives us a lot of control over the exact behaviour of the objects and can be implemented to be very fast at the hardware level, the way we have been specifying individual objects is less than ideal.

Consider the following example: you are in charge of helping an artist to model a scene for a movie. The scene will take place in a street, and among the several assets that need to be placed, you need to take care of placing the lamp posts, which are going to be set at predictable intervals. Suppose that this is a rather long stretch of road, and there are a total of 50 lamp posts that need to be placed. With our current setup, we would need to *manually* create and specify the positions of each one of the 50 lamp posts! Ignoring the memory cost of having this many duplicate meshes, the sheer amount of time that it would take to code this would be ridiculously high. More importantly, if the positions need to change, you would have to change them for all 50! Clearly, there has to be a better way!

Instancing

In this example, we can begin by noticing that the lamp posts are all *identical* with the exception of their location. Even if their sizes were different, the base model is the same, with the only variant being the model matrix that each one has. Supposing that a formula could be created to specify the positions of the lamp posts automatically, the only thing that we really care about, and need to generate are the model matrices themselves. What we would then be doing is creating different *instances* of the same model with a different matrix applied to them. This process is known as *instancing* and when it relates to rendering it is called *instanced rendering*.

The base idea is the following: instead of duplicating a mesh that we already have, what we can do instead is to simply specify a different model matrix per instance of the object that we require in our scene. If we were to consider the dragon mesh which consists of 871,306 triangles, for the cost of only 16 floats per matrix, we could have as many copies of the dragon as we wished.

Instancing is a very powerful technique that allows for the rendering of very complex scenes with little cost to performance and with huge savings in memory. It is particularly useful in real-time graphics where memory is a scarce resource (comparatively speaking), but it is also very helpful for offline rendering where the meshes can be in the order of millions of triangles. How instancing is implemented depends very much in the situation you are in. OpenGL does provide a way of instancing objects by specifying a buffer of matrices along with the vertices of the mesh.

While we will not discuss the details of how OpenGL manages it, it is important

to understand how it would work. In essence, we would provide OpenGL with the buffers of the mesh we wish to instance, and a separate buffer that would contain the model matrices of all the instances. Since the buffer would not change for the duration of the frame, we would create something called a *uniform buffer* which is essentially an extension of the uniform variables we saw earlier. From here we would simply tell OpenGL to render the mesh with the uniform buffer and that's it. Everything else would be taken care of in the GPU.

Scene Graphs

Instancing solves the following problem: "I have n identical objects that I need to place in world space. The only difference is their model matrix". There is another problem that is similar, which would require a different approach. Consider this problem: you are designing a character for a video game. For the sake of simplicity, let's assume that the character is composed of simple boxes for the legs, torso, arms, and head (similar to Minecraft). Clearly the arms and legs can rotate independently of each other, and the head and torso must be tied together. However, their sum position is directly linked. We cannot have a circumstance where the torso is allowed to move without the legs, or the head to just walk off with the legs and leaving the torso and arms behind (as amusing or scary as this may be). Ultimately, all of their positions rely on each other, and in fact we can define a type of hierarchy with them. Say that we divide the torso into two halves to represent the chest and hips. Then we would end up with the following relationships:

- The legs can rotate around the hip joints, but their position depends on the position of the hips.
- The arms can rotate around the shoulder joints, but their position depends on the chest.
- The head can rotate, but its position also depends on the chest.
- Finally, the position of the hips depends on the position of the chest.

What I have essentially described here is a hierarchy of dependency. If we were to draw this out, we would end up with a structure that you will recognize as a tree (or a graph). This type of structure is called a *scene graph*. Keep in mind that this structure can be a lot more expansive and complex than what we have described here. We can extend this system as far as we like, to tying the position of multiple characters together, to objects in the scenery, etc.

Continuing with the example of the character, we will now see how to represent the scene graph and traverse it. Let's simplify the character to just the two legs and the hips. If we draw out the graph, we would end up with a single root at level 0 (the hips), and two leaves at level 1 (the legs). Say we wanted to render these. Well in order to position the hips, we would just grab the model matrix for that model and that's it. But what about the legs? They have their own model matrices, but their position depends on the position of the hips. So, we would concatenate the model matrices of the legs with the model matrix of the

hips. More concisely, let M_h be the model matrix of the hips, and M_{l_0}, M_{l_1} be the model matrices for the legs. We then have the following:

- The final position of the hips is determined solely by M_h.
- The final position of leg 0 is determined by $M_h M_{l_0}$.
- The final position of leg 1 is determined by $M_h M_{l_1}$.

Notice that we are multiplying M_h on the left of the matrices for the legs. The reason behind this boils down to the order of operations. We first want to rotate the legs by whatever angle they need and *then* we want to position them next to the hips. Note that we would also have an offset from the centre of the hips attached to each of the model matrices of the legs to ensure they don't end up on top of each other, but the principle remains the same.

We can see how this would be a similar system to the arms and head in relation to the chest. Finally, we could decide that the hips depend on the chest, and therefore to compute the position of the legs we would concatenate the model matrix of the hips with the one from the chest.

The only real cost associated to the scene graph is its traversal, in particular for real-time graphics. Since the graph could become very complex, traversal times could increase. That being said, there are ways of solving this by caching the matrices at each level. If they don't change, there's no need to traverse the entire hierarchy. We could also keep track of the structures and prune areas that don't have to be connected (even if they seem to be). The major advantage of this technique is that it allows us a very intuitive way of thinking and organizing our objects in space.