**Rendering and the Scene Graph**

**The Scene Graph**

Everything's based around nodes; shader effects are attached to nodes in the scene graph. Individual mesh instances are represented by GeometryNodes, and things are organized into coherent groups (like turrets on a ship) via GroupingNodes. Shader effects are *attached* to nodes; if attached to a GeometryNode, the shader just affects the one mesh. If the effect's attached to a GroupingNode, it affects all the nodes under it.

How can we use the scene graph's shader hierarchy, *and* use a spatial organizer to cull geometry? One way is to have some list of the global shaders affecting a node – when it's moved, it clears the list, then recursively asks parents for their global shaders. Inefficient on hierarchy change, but plausible and removes need for sentinel nodes.

**Shader Parameters**

For uniforms and such, there'll be key names that the engine will look to set - for lighting, "lEmitColor", "lConeAngle", "lLightPower" and the like; for materials, "lDiffuse", "lSpecular", "lNormTex". If the shader doesn't contain the keyword, it can be noted or ignored since shaders are just scanned in for uniforms.

Unity does something interesting – a shader's your usual HLSL code, but it has a properties header specifying the shader's name, what uniform name maps to what, and what that uniform's name should be in the editor. Passes and fallback shaders can also be specified. One file contains the entire shader, not just an individual fragment/vertex shader.

**Handling Lights**

Lights are attached like shaders are, but are also hierarchy elements. The shader needs to specify that it's looking for a type of light and how many it may be looking for. On a draw call, the renderer will then pass whatever light parameters are needed to the shader.