Renderer & Introduction to Scene Management

Scene Management

- We will define Scene Management as a system that allows for fast queries of geometry.
  - Handle collision detection quickly.
    - Spatial Systems will allow us to quickly cull objects from collision tests that have no chance of colliding with each other.
  - Provide information for AI systems.
  - Process geometry of our scene in a way that allows it to be quickly rendered.
    - The Scene Managment system will be in charge of deciding what objects need to be rendered as well as sorting those objects into batches of like objects that should be rendered together.

Scene Management

Our Scene Management system will have the following stages:

1. Hierarchy Flattening
    a. This is how we attach one mesh to another
2. Getting the Viewable Set
    a. Frustum Culling
    b. Spatial System
3. Sorting
    a. Opaque Set
    b. Transparent Set
    c. RenderContext
4. Rendering

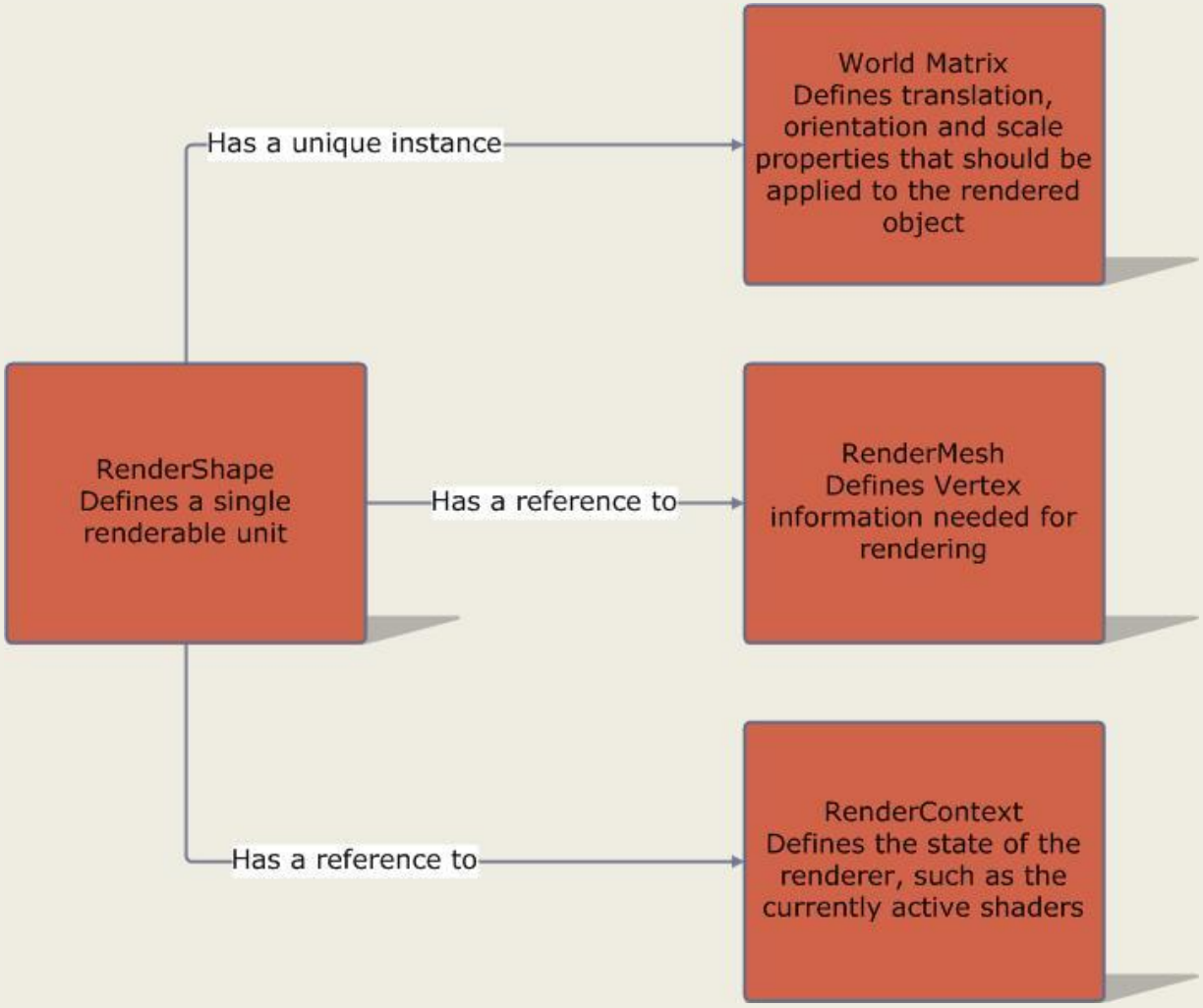
Renderer

- The Renderer will be responsible for processing batches of renderable objects.
  - The batches, or RenderSets, will be pre-sorted linked lists of renderables
  - This will be a "Stupid" renderer
    - The Renderer is stupid, because it will do **no** processing on the objects to be rendered. There will be no branching.
  - We will also attempt to keep all Rendering API code contained in our Renderer library.
    - This makes the system easier to replace or update the API specific code, such as DirectX or OpenGL.

Renderer

- Before we can move onto discussing the details of how our renderer will function, we need to define what data a renderable object will need.
  - To the white board!

RenderShape



Renderable/RenderShape

- RenderShape contains
  - Reference to RenderMesh
    - Vertex positions, Indices, Normals, Texture Coordinates, more advanced vertex elements...
  - Reference to RenderContext
    - Shader, textures, blend states, vertex and index buffers...
  - A unique world-space matrix
- The referenced RenderMesh and RenderContext allows for various objects to share common data.
  - Maybe we want to render all the laptops in the room. We might be able to use the same RenderMesh for them all, but changing the RenderContext would allow each instance to have a unique texture.
  - Maybe your laptop could use the same texture as your neighbors, but you dropped it down the stairs and it is no longer the same shape, requiring a different RenderMesh.

RenderNode

- RenderNode will be the base class object from which we will inherit all objects that need to be processed by the renderer.
  - The Renderer will only know about RenderNode, not its derived classes such as RenderShape and RenderContext
- RenderNode will contain a function pointer, in our implementation we call this function pointer the RenderFunc
  - This RenderFunc will need to be pointed to a method that will handle rendering the object, but it is not limited to rendering code
- In order to render a RenderNode instance, the Renderer will call the RenderProcess method, which in turn will call the RenderFunc

RenderNode Definition

```
typedef void (*RenderFunc)(RenderNode &rNode);
class RenderNode
{
  RenderNode *next;   // Used to define list of nodes
  RenderFunc func;   // Function pointer

  // Process used by the Renderer
  inline void renderProcess() { func(*this) };
};
```

Why the RenderFunc?

- Why do we have function pointer, instead of an actual render function?
  - The function pointer gives us a layer of abstraction.
    - This abstraction allows us to have various methods of rendering supported.
    - Instead of branching on the state of the object in render loop, we can assign an appropriate render method when we load the object.
    - An example of this need would be transparent objects. We typically render transparent objects twice, first back faces then front to ensure proper blending. At load time, a transparent object can be assigned a render method to support his need.

Why the RenderFunc?

- Why do we have function pointer, instead of an actual render function? Continued...
  - The function pointer tends to provide a simpler and faster path to this abstraction than a virtual inheritance model.
    - Virtual function calls have an increased cost
    - Changing the render method at run-time with the RenderFunc just involves assigning a new method. With the inheritance model we would have to copy from the old object to a new one, put the new one in the list, and delete the old one.
  - Allows the end user a means to change rendering code without actually needing the source code of our

RenderSet

- The RenderSet class will define a linked list of RenderNodes that need to be rendered this frame.
- We will eventually implement this class ourselves, but for now we will just need to be able to iterate through the contained RenderNodes.
  - The RenderSet has an accessor method for retrieving the head node.
  - Each RenderNode has a pointer to the next node in the list.
  - Once we find a NULL RenderNode we have reached the end of the list.

Extending RenderNode

RenderContext

- The job of the RenderContext is to put the rendering system into the correct state for rendering specific objects.
  - This will consist of things like setting the vertex stream, turing on shaders, as well as changing engine states like the blend mode.
  - Changing these states, can be very costly as they must use the motherboard bus to communicate between the CPU and GPU and these changes may be stalled, or cause stalls, depending on what the GPU is doing when a context change is applied.
  - We will sort objects based on RenderContext to reduce how often we have to make these changes per-frame.

RenderContext continued...

- RenderContext is a (inherits from) RenderNode
  - RenderContext will have all the functionality of RenderNode, including the RenderFunc function pointer.
  - What is the main responsibility of the RenderFunc for a RenderContext?
    - We do not have access to enough data to actually draw anything...
    - We will have to point the RenderContext instances' RenderFunc at a method that will apply the context changes desired.
    - We will likely have multiple methods for the RenderFunc, each giving different functionality such as using different vertex buffers or different engine states like blending.

RenderContext Defined

```
class RenderContext : public RenderNode
{
private:
    Shader shader;
    std::vector<TexID> texIds;
    // All the other rendering states
public:
    // RenderFunc methods
    static void ContextPosNormalUVRenderFunc(RenderNode &node);
    static void ContextTransparentPosNormalUVRenderFunc
(RenderNode &node);
...
};
```

RenderShape

- The RenderShape will have the task of grouping data and functionality of rendering a single object.
- The RenderShape will store a reference to the RenderMesh, a reference to the RenderContext that is used with this RenderShape, as well as a unique matrix defining the position, orientation and scale of a renderable object.
  - Referencing the RenderMesh and RenderContext allows for smaller memory usage, as well as reducing the need for context switches.
- A RenderShape method that is to be used as a RenderFunc will have the task of actually rendering the object.
  - The referenced RenderMesh and RenderContext should contain all the data needed for rendering.

RenderShape Defined

```
class RenderShape : public RenderNode
{
private:
    RenderContext &context;
    RenderMesh &mesh;
    Matrix worldMatrix;
    Sphere boundingVolume;
public:
    // RenderFunc methods
    static void IndexedPrimitiveRenderFunc(RenderNode &node);
    static void PrimitiveRenderFunc(RenderNode &node);
...
};
```

RenderMesh

- Our RenderMesh will be our interface to vertex data.
- The vertex data can consist of many things, not limited to vertex positions, normals, texture coordinates, vertex colors, incides, tangents, bone indices, skinning weights... and so many more possible vertex data data types.
  - We will not directly store and vertex arrays. Instead, vertex data will be read from file and then added to buffers on the GPU.
- We will also need to be able to read and write meshes from file.

Index and Vertex Buffers

- We can put all the indices of different RenderMeshes into one buffer.
  - We will end up with multiple vertex buffers, each with a different vertex type, but data sharing a vertex type will share the same vertex buffer.
- Using only one buffer means we do not have to switch the buffer per-object, saving us costly context changes.
- There are many ways to create a system with this functionality. A reasonably easy way to go about it is to recreate the vertex or index buffer each time new values need to be added. Each time we recreate the buffer, we will make a new buffer that is big enough to hold what has already been stored, as well as our new data. Then we copy from the old buffer to the new one, and delete the old buffer.

Typical use of Renderer