# Rendering System Overview

This document is intended to present the current design of the Hieroglyph 3 rendering system. In general, the design is constantly evolving and hence may change at a later date. However, the items discussed in this document should remain mostly unchanged for the foreseeable future, and hence are worth documenting now.

The information in this document is also intended to supplement the actual source code of the Hieroglyph 3 engine. A significant effort is made to properly document the important parts of the source code, and additional updates are made as frequently as possible. Even so, this is an open source project and is therefore not a fulltime adventure. If there is something not correctly documented or missing documentation, please consider adding a discussion topic at the project page and we will try to update it as soon as possible.

The overall design of the rendering system makes use of four major functional groups, each of which will be discussed in more detail in the following sections.

## Renderer

The true heart of the rendering system is the Renderer itself. The RendererDX11 class is the source for all of the other portions of the rendering system, and hence is the primary class that an application will interface with to do some rendering. The renderer loosely follows the same responsibilities of the ID3D11Device interface from Direct3D 11 at a low level, but also coordinates higher level activities such as scene processing and rendering batches. Here is a short description of each of the renderer’s responsibilities.

### Creation/Management of Resources

The renderer provides the ability for other classes to create and reference the various Direct3D 11 objects. The primary objects that other non-rendering system classes will be interested in are the memory resource classes, which represent buffers and textures. The renderer allows creation of these objects, but retains ownership of them and returns only an ID to the caller. That ID is then used by the caller any time they want to reference a particular resource.

This pattern is also followed for non-resource classes as well, such as shader objects, samplers, input layouts, and so on. Instead of making the other classes work with direct references to D3D11 objects, they are instead referenced by IDs. This allows the renderer to keep control over how and when the D3D11 objects are used, and minimizes the direct knowledge needed by the application to use the engine.

### Coordinates Scene Rendering

The other primary task of the renderer is to coordinate the processing and rendering of a given scene. In Hieroglyph 3, scenes are represented as spatial scene graphs of Entity3D instances. A scene rendering is encapsulated into what is referred to as a render view. We will cover render views in more detail later, but for now knowing that they represent one scene rendering is enough to get started. As a scene is processed, some special effects require additional “views” of a scene to be rendered. For example, if there is a reflective object in the scene, then a reflection map is needed that represents the reflected view of the scene. The renderer provides the interface methods for aiding in a scene to recursively process multiple render views in the proper order.

In addition to this, there is also the ability to automatically utilize the multi-threading features of Direct3D11 to process render views in parallel. When selected to use multi-threading mode, the renderer will process each render view on a separate thread. Each render view produces a single command list that holds all of its submitted API commands. After all views are processed, they are executed on the immediate context. The number of threads is selected at compile time by a constant, with the current default set to 4.

The overall processing time may or may not be reduced by this parallel processing. In particular, very small scenes may not benefit very much, or could even suffer performance penalties for using the multithreading mode. To allow traditional processing, an interface method can be used to only use a single thread and only the immediate context for processing all of the views sequentially. This allows for an easy way to check and see if you are benefitting from parallel processing or not.

There are two major sub-systems used by the renderer in carrying out its scene processing tasks. Those are the Pipeline Manager and the Parameter Manager systems.

#### Pipeline Management

Pipeline management is performed by the PipelineManagerDX11 class. In general, this class loosely follows the idea of a context in Direct3D11, although it adds additional higher level capabilities as well. It is also used for both immediate and deferred context management, and is used to hide the details of using one or the other from the rest of the system. The following sections go into more detail about what this class is used for.

##### Setting States

The primary purpose of the pipeline manager is to provide an interface to the actual rendering and computation pipelines of Direct3D 11. As such, it has methods for setting the various states such as shader programs, resource views like SRVs or UAVs, state objects, and so on.

##### Managing Pipeline State

The pipeline manager is used to monitor and update the various pipeline states. This means monitoring the API calls that are made, and reflecting those calls accordingly in its model of the pipeline. This is especially important when trying to minimize the number of API calls being used, since it can eliminate any redundant state change requests.

This is also a very important topic to consider when working with immediate and deferred contexts. The pipeline state is propagated in different ways depending on how command lists are generated and consumed, so it is very important to update the pipeline model accordingly.

##### Coordinates pipeline execution

Once the pipeline has been configured as desired, the pipeline manager also has methods for invoking the pipeline. In the case of the rendering pipeline, this means calling one of the various DrawXXX calls. For DirectCompute workloads, this involves calling one of the DispatchXXX calls. The following sections describe these in more detail.

###### Drawing

Rendering in the pipeline is performed by first configuring the pipeline, and then calling an appropriate rendering method. This can include performing simple draw calls that only use vertex buffers, indexed draw calls that use vertex and index buffers, instanced drawing that uses both per-vertex and per-index vertex buffers, or even indirect drawing that uses another buffer resource to control the draw call. When possible, a geometry class is used to encapsulate the setting of parameters for each of these draw calls. Geometry classes are discussed in more detail later in this document.

###### Dispatching

Dispatching is a much more broad activity than drawing. Since DirectCompute allows for precise control over the threading mechanisms used by the GPU, it provides the ability to perform more general purpose computation and apply it to an action that isn’t necessarily related to rendering. There are direct dispatching calls as well as indirect dispatch calls to allow using buffer resources to control the pipeline execution.

While the draw calls typically use a geometry object to configure the pipeline, this is typically not the case with dispatching. The general nature of dispatching makes it less amenable to encapsulating its state in a class, and since the compute shader is only a programmable shader stage (i.e. no fixed function stages) then we can automate most of the configuration with the parameter manager system (which we discuss next).

#### Parameter System

Many of the settings that are used to configure the pipeline can be resolved to a particular text based name. For example, a shader program that uses a texture to provide the color of a model would use the name ColorMap to reference the texture. The application would likely want to specify which texture resource to use when this name is needed in a shader, as opposed to having to manually specify which shader resource view slot to bind it to. Managing the mapping from an application supplied parameter to the corresponding setting in the pipeline is handled by the parameter system in Hieroglyph 3.

The parameter system is implemented in the ParameterManagerDX11 class. As its name implies, it is responsible for coordinating the parameter data. Basically any client of the entire rendering system can acquire a handle to a parameter with its text based name, and then set that parameter’s value as it sees fit. Then during a later rendering pass, any shader parameter that is needed can easily be read out from the parameter manager. In this sense, the parameter manager can be thought of as a general storage area for named parameters where there are many writers of data and the rendering system is the reader of that data.

##### Multi-threaded Parameters

Since Hieroglyph 3 takes advantage of the multi-threading capabilities of Direct3D 11, that means that the rendering sub-systems must also take into consideration how it will support multiple threads. The parameter manager supports access from multiple threads by having each parameter store multiple copies of its data. Then each thread is assigned an index to write into, which tells it which copy of the parameter data can be written by it (and also which one to read from as well). This is depicted in Figure 1.

Figure 1: The handling of multiple threads by the parameter manager.

By allowing a parameter to support multiple clients with separate data areas, they can simultaneously allow access to multiple threads without trying to re-synchronize the data being used. This multiple copies of data are actually not inherently thread safe, but due to the sequencing of a rendering pass it becomes thread safe. We will walk through some additional details about this sequencing later on.

In addition to allowing thread safe access to the various parameters, this multiple data design allows for a single storage area to house all of the parameters. This is implemented as a single std::map<std::string, RenderParameter\*> which maps names to render parameters. This single map is accessed by all instances of a parameter manager, and then the individual data elements of each parameter are accessed according to the parameter manager’s specified index.

### The Multi-Threaded Renderer

With these two sub-systems defined in addition to the renderer itself, we can discuss how the multithreaded renderer is organized and how it performs the work that it does. As mentioned above, the general unit of GPU work in Hieroglyph 3 is called a render view. A render view is submitted directly to the renderer, where it is queued up until a later time when all of the queued render views are processed. Internally to the renderer, there are multiple copies of pipeline managers and parameter managers in use, and they are always paired together. To represent the immediate context, one pipeline manager is used and a corresponding parameter manager paired to it. Then for each additional thread that will be used in the renderer, another pipeline/parameter manager pair is created and assigned to that thread. For these subsequent pairs, a deferred context is used in the pipeline manager’s. This general concept is depicted in Figure 2.

Figure 2: The pipeline and parameter managers within the renderer.

When the renderer is configured for single threaded mode, only the first pair of pipeline/parameter managers is used to process the render views one at a time. In multithreaded mode, the render views are all processed by the additional pairs of pipeline/parameter managers to produce command lists. These command lists are then processed on the immediate context where they are submitted to the GPU/driver.

Since the pipeline/parameter managers are designed to be modular, it is trivial to increase or decrease the number of threads that are used in the renderer. In addition, the number of additional threads is determined at compile time and the manager pairs are created at startup. This means that it is possible to switch between single threaded and multithreaded rendering mode during runtime, which can be used for debugging, performance comparisons between the two modes, and switching to single threaded mode to utilize specific features which can only be used on an immediate context.

## Object System

With a general idea of who does what in the rendering system, it is time now to turn our attention to the object system and see how it fits into the overall Hieroglyph 3 rendering system. The object system is what provides the objects within a scene their spatial frames of references, as well as the interfaces for using the rendering system. The following sections will provide an introduction to the object system and how it is implemented.

### Spatial Scene Representation

The heart of the object system is the Entity3D class. This class provides a spatial representation of an object, including its position and orientation. Any object that will end up being rendered will be represented as an instance of an Entity3D. With the position and orientation data, interface methods are provided for calculating an appropriate transformation matrix to position this object in the correct place for a given scene rendering.

To provide hierarchical connections within a scene, the Node3D class is provided. This class is a subclass of Entity 3D and provides the data and methods for creating a directed acyclic graph of entities. This allows for the typical hierarchical manipulation of scenes, and provides a fairly simple access mechanism for working with a scene. Since the graph must have a root node, then the entire scene can be handled with a single node reference.

### Coordinates scene processing

This concept of a scene is used in two different aspects. First, the state of a complete scene (which is ultimately just a graph of entities) is updated with a recursive call to the method Entity3D::Update(). This recursive call will propagate through the scene and have each entity update its world space location based on its own spatial data and that of its parent. This is performed for every frame update, and the delta time that has elapsed between frames is passed as an argument to the method.

Once the scene has been updated, it is ready to be processed by the rendering system. We have already mentioned render views several times in this article, and they are about to be discussed again... For any render view that will be providing a rendering of the scene, it must have a reference to the scene graph itself. This is typically performed by providing it with a reference to the root node of the scene. The graph structure of the scene is then used to provide a couple of methods for rendering the scene. The first is the method Entity3D::PreRender() method. This is a recursive call that provides each entity in the scene an opportunity to add additional render views to the renderer for processing. For example, if an object is shiny and needs a paraboloid environment map, then it would push its render view into the renderer’s queue during its pre-render method.

After all of the render views have been added to the queue, all of the render views are then processed by the renderer as detailed above (which will vary depending on if single- or multi-threaded rendering is used). Each object is processed for a particular view by calling the recursive Entity3D::Render(...) method. In this method, each entity performs the needed render pipeline configuration before actually invoking the pipeline and rendering its geometry. The exact setup of this rendering will depend on the type of view being processed, so the material system is used to provide all of the needed information for the expected view types in a given scene.

# Rendering System Details

With the general concepts in mind, we can move on to looking at the detailed responsibilities of each of the major classes that are used in the rendering system. After the this further introduction to the rendering system and its individual components, we will walk through a typical sequence of events that occurs while rendering a scene in Hieroglyph 3 to tie all of the pieces together.

## Entity3D

As previously mentioned, the Entity3D class provides the spatial representation of objects within a scene. The position and orientation of each object is managed by manipulating the Entity3D::Position() and Entity3D::Rotation() variables, which are later used to construct an appropriate world transformation to be used when rendering the entity.

While the entity provides the spatial information of an object in a scene, it isn’t concerned with what will actually be rendered. In fact, many Entity3D instances won’t have anything to render at all and instead just represent a location within a scene. Instead of complicating the Entity3D class with the configuration and management of the rendering system, it delegates the decision of what geometry to render and how to render it to two other classes: GeometryDX11 and MaterialDX11. These two classes can be seen as providing the input to the pipeline (in the case of GeometryDX11) and configuring the pipeline (in the case of MaterialDX11).

### GeometryDX11

In general, the GeometryDX11 class allows the user of the engine to specify the appropriate resources and their contents to be used as input to the pipeline. At the time of this writing, the class currently supports indexed rendering with a single vertex buffer and an index buffer. The vertex buffer is filled with array based data provided by the application to represent the per-vertex attributes (with VertexElementDX11 instances), while the index buffer is filled with data provided as points, lines, or triangles of indices. This provides a fairly convenient way to either load data from a file or to programmatically generate the desired data.[[1]](#footnote-1)

#### Provides pipeline input

The vertex and index buffers are then used to provide a geometric object to the pipeline for indexed rendering. The buffers themselves, the topology type of the index data, and the vertex layout are all supplied by the geometry class. Since this data is encapsulated within the GeometryDX11 class, the Entity3D class doesn’t need to be concerned with manually manipulating individual vertex elements. The actual binding of the geometry resources to the pipeline occurs in the PipelineManagerDX11::Draw(...) method.

With this separation of responsibilities, we can also extend the GeometryDX11 class to support other types of rendering operations in the future. For example, direct support for indirect rendering, instanced rendering, or any other combination thereof could be added with a subclass of GeometryDX11. The Entity3D class simply references the geometry object, but it can provide the appropriate resources that are needed for some other rendering type.

### MaterialDX11

With the pipeline input data provided by the GeometryDX11 class, the MaterialDX11 class is responsible for properly configuring the pipeline to render the geometry as desired. This includes the binding of shader programs, providing the needed constant parameters for those shader programs, and the setting of pipeline states (such as blend state or stencil test state). In fact, the MaterialDX11 class delegates this responsibility to the RenderEffectDX11 class. Instead, the material itself houses an array of RenderEffectDX11 instances to allow geometry to be rendered differently in different situations. This will be discussed further in the Render Views sections, but essentially the MaterialDX11 class provides one RenderEffectDX11 for any type of rendering condition that will be encountered in an application. The actual binding of the appropriate pipeline configuration is performed with the RenderEffectDX11::ConfigurePipeline() method.

### Geometry and Materials Together

When a GeometryDX11 and a MaterialDX11 are used together, the programmer must ensure that the geometry actually matches what is expected from the set of shader programs being provided in the material. This is logical, since you can’t execute the pipeline with mismatched input and pipeline configurations.

Where this becomes more interesting is when you can use the same set of geometry resources with multiple pipeline configurations. For example, when performing skinning with a particular geometric object, you can use it to both render a shadow map or to render a final perspective rendering. The geometry doesn’t need to change, but the material can simply provide two RenderEffectDX11 instances to render the object according to the current needs.

## Render Views

As described throughout this document, the render view is the key to actually performing any type of graphical operation with Direct3D 11 in the Hieroglyph 3 engine. In general, you can consider a render view as a way to group operations to be performed with the API into objects that can then be used by other portions of the engine, such as the material system. This section will give an introduction to the render view concept, and provide some detailed information about how they are currently being used in the engine.

### Basic Concept

The render view class hierarchy begins with the base class, IRenderView. Each subclass of IRenderView is used to perform a sequence of API calls to perform some useful work. This can include the rendering of an entire scene in a perspective view (as seen in the ViewPerspective class), the generation of a paraboloid environment map (as seen in the ViewParaboloidEnvMap class), and can also be used for non-rendering related operations such as performing a simulation on the GPU (as seen in the ViewSimulation class). There is no restriction put onto the operations that can be performed in a render view, so it can be used to encapsulate one or more rendering operations that must be performed sequentially.

Even though there are no restrictions imposed on the render view contents itself, it makes sense to consider what would really be sensible to batch together. With the previous discussion about The Multi-Threaded Renderer, we know that the render views are processed in parallel to one another when possible. Thus the encapsulated sequence of operations should be as long as possible, while still allowing for the total rendering to be performed in several chunks. In practice, this tends to suggest that one complete rendering pass is a good rule of thumb. A rendering pass here indicates the sequence of operations used to bind render targets and perform all necessary rendering operations on it until the next render target would need to be bound.

The actual execution of a render view is performed with two different methods, the IRenderView::PreDraw() and the IRenderView::Draw(). Examples of these methods taken from the ViewPerspective class are shown in Listing 1 and Listing 2.

void ViewPerspective::PreDraw( RendererDX11\* pRenderer )

{

if ( m\_pEntity != NULL )

{

Matrix4f view = m\_pEntity->GetView();

SetViewMatrix( view );

}

// Queue this view into the renderer for processing.

pRenderer->QueueRenderView( this );

if ( m\_pRoot )

{

// Run through the graph and pre-render the entities

m\_pRoot->PreRender( pRenderer, GetType() );

}

}

Listing 1: An example of the IRenderView::PreDraw() method.

The IRenderView::PreDraw method (shown in Listing 1) is responsible for doing what it sounds like – all operations that are needed to be performed prior to the view being drawn are done here. This can include collecting data from the scene (like view matrices from a camera entity) and preparing it for use later on. However, there are two key items that the pre-draw method performs. The first is that it must queue itself in the renderer for later processing. All render views are processed in this way – first they are queued into the renderer, and then the renderer will execute each view as is appropriate for its current configuration (i.e. single- or multi-threaded).

The second task performed in the IRenderView::PreDraw method is to give any objects that will be rendered by this render view a chance to perform any operations that they need prior to rendering. This is done with a recursive method call to Entity3D::PreRender(...). This method is called on the root node of the scene being rendered, which then propagates the call to each of its children. This pre-render operation allows an object’s material to also queue additional render views if they are needed before rendering the current render view[[2]](#footnote-2).

void ViewPerspective::Draw( PipelineManagerDX11\* pPipelineManager,

IParameterManager\* pParamManager )

{

if ( m\_pRoot )

{

// Set the parameters for rendering this view

pPipelineManager->ClearRenderTargets();

pPipelineManager->BindRenderTargets( 0, m\_RenderTarget );

pPipelineManager->BindDepthTarget( m\_DepthTarget );

pPipelineManager->ApplyRenderTargets();

pPipelineManager->SetViewPort( m\_iViewport );

// Set default states for these stages

pPipelineManager->SetRasterizerState( 0 );

pPipelineManager->SetDepthStencilState( 0 );

pPipelineManager->SetBlendState( 0 );

pPipelineManager->ClearBuffers( m\_vColor, 1.0f );

// Set this view's render parameters

SetRenderParams( pParamManager );

// Run through the graph and render each of the entities

m\_pRoot->Render( pPipelineManager, pParamManager, GetType() );

}

}

Listing 2: An example of the IRenderView::Draw() method.

After the view has performed its PreDraw() method, it remains dormant until the RendererDX11 class processes it. The processing essentially consists of calling the render view’s Draw method, as shown in Listing 2. The method is where the render view actually accomplishes its primary goal – to get some work done on the GPU. In this example the view sets up the rendering pipeline, clears the attached render targets, sets its own rendering parameters, and then calls the recursive Entity3D::Render() method.

The configuration of the pipeline depends on what the goal of this view is, but in general a rendering based render view will have a similar batch of setup calls as shown here. In the next step, the call to SetRenderParameters() is a method of the render view which allows it to set its own parameters into the parameter manager before the scene is processed and rendered. In the case of a ViewPerspective, it will set the view and projection matrices before the scene is rendered so that it uses the appropriate parameters.

The final step in this render view is to render the scene. This performs a depth-first traversal of the scene graph and renders each Entity3D in the scene with its Entity3D::Render() method. The contents of this method are shown in Listing 3.

void Entity3D::Render( PipelineManagerDX11\* pPipelineManager,

IParameterManager\* pParamManager,

VIEWTYPE view )

{

// Test if the entity contains any geometry, and it has a material

if ( ( m\_sParams.pGeometry ) && ( m\_sParams.pMaterial ) )

{

// Only render if the material indicates that you should

if ( m\_sParams.pMaterial->Params[view].bRender )

{

// Set the material render parameters. This is done before

// the entity render parameters so that unique values can be

// set by the individual entities, and still allow the

// material to set parameters for any entities that don't

// specialize the parameters.

m\_sParams.pMaterial->SetRenderParams( pParamManager, view );

// Set the entity render parameters

this->SetRenderParams( pParamManager );

// Send the geometry to the renderer using the appropriate

// material view effect.

pPipelineManager->Draw(

\*m\_sParams.pMaterial->Params[view].pEffect,

\*m\_sParams.pGeometry,

pParamManager );

}

}

}

Listing : The Entity3D render method.

This method makes a few checks to see if the entity has both a geometry object and a material object set in it, then to see if the material has been configured to render for this particular type of view. This allows for selectively including or excluding an entity from different types of render views. Next, the material’s render parameters are set, followed by the entity’s own set render parameters call (these are intentionally named the same as the render view method – they all accomplish the same thing for their respective objects). Finally the Entity3D class draws its geometry with its material into the pipeline manager. This will either immediately render the object (in single-threaded mode) or write the commands into a command list (in multi-threaded mode).

### Scene Processing

#### Recursive Scene Graph Traversal

#### Pre-Rendering

#### Rendering

### Specialized Processing

#### Dispatches

#### Non-scene based rendering passes

## Scene Rendering

### Camera Object

#### Provides a render view for the main window view

1. In fact, there are two static classes that provide geometry loading and generation: GeometryLoaderDX11 and GeometryGeneratorDX11. [↑](#footnote-ref-1)
2. The prototypical example of this operation is generating an environment map for an object, which must be performed prior to actually rendering the object itself. [↑](#footnote-ref-2)