Rendering Architecture

Table of Contents

[2 Overview 1](#_Toc382652441)

[2.1 2D Rendering 2](#_Toc382652442)

[2.2 3D Rendering 2](#_Toc382652443)

[2.2.1 Vertex Formats 2](#_Toc382652444)

[2.2.2 GeoPooling and SurfacePooling 2](#_Toc382652445)

[2.2.3 Materials 3](#_Toc382652446)

[2.2.4 Design 4](#_Toc382652447)

[2.2.5 Particles 5](#_Toc382652448)

[2.2.6 Draw mechanics 5](#_Toc382652449)

# Overview

Rendering can be conceptualized as a series of layers. These aren’t explicit layers created as multiple surfaces, but they are layers in the sense of work being done, and ordering between them.

Given the design of the game, the most complicated case is a game that is running with HUD visible, and a menu on top. This can be visualized in the following way:

Game Rendering

HUD and UI

Optional filter (dim, blur, sepia, etc…)

Menu(s)

The core game renders first, then the HUD and other UI elements are rendered overtop. Then, depending on our final design, some sort of filter may be run on that combined result before laying down the menu. This can include fade-out, wash-out of colors, sepia, blur, etc… to really make the menu stand out. When no menu is up, that filter is removed.

## 2D Rendering

For 2D rendering, we will have a virtual screen bounds (ex: 1280 x 720) value that the UI and other visual elements can layout against. There will also be a switch to control how that virtual canvas gets mapped to the screen, which may be any number of resolutions depending on graphics settings, etc… The two mapping options are stretch and center. In stretch, the visual bounds are stretched to the output resolution while maintaining aspect ratio. In center, the contents aren’t resized to match resolution, the UI is just centered instead.

There will be intelligent batching and coalescing of 2D rendering calls to improve performance, and 2D rendering will support transparency.

Finally, we will have support to provide both input source rect and final destination rect. The source rect allows using a portion of a source image (ex: fonts).

## 3D Rendering

Inputs are object transforms, object poses, object bounds, geometries, textures, materials (shaders/effect), lights, and some game object flags like IsVisible, CastsShadows, and ContainsTransparency.

Supported lighting: Ambient, Specific Tone-Mapping Hues, Directional, Point, Spotlight.

Supports shadow mapping with filtering.

### Vertex Formats

There will be 2 supported 3D vertex formats. This helps simplify the geopooling and materials discussed later. One format will be used for non-skinned objects, and contain basic information about position, tangent basis, and texture coordinates. The second format contains everything in the basic one, but also includes bone weight and index data.

### GeoPooling and SurfacePooling

Each piece of 3D geometry in the game will share one of 2 supported vertex formats. We also do not support implicit or non-poly surfaces, meaning no dynamically generated terrains and no other formats where geometry is determined in the shader.

This allows us to create larger buffers to hold vertex and index data, and for each level we can pack the objects required for that level into these buffer pools. The size of each pool will be determined based on the available video memory of the machine to minimize thrashing over the bus. If we can keep the pools paged in on the GPU, there will be a good performance benefit to having them pooled. It also avoids state changes between draws. If we exceed the size of one pool, we create another pool and add the rest of the objects there. There is no limit to the number of pools, but the size of each is important for performance.

Similarly, texture arrays will be used for each object, and these can also be pooled. For instance, all objects with 512x512 textures can have their images packed into a 512x512 texture array, and just remember which offset into the array their images start at. This allows us to keep a single texture array bound for longer, and gives us another metric on which to pool/batch draws against.

Basic GeoPool data:

* Vertex format
* Vertex buffer
* Index buffer
* Space available (next index)

Basic Geometry data:

* GeoPool ID
* BaseVertex
* BaseIndex
* NumVertices
* NumIndices

Basic SurfacePool data:

* Texture2D Array
* Width
* Height
* Format
* Space available (next index)

Basic Texture data:

* SurfacePool ID
* BaseIndex

There will be a GeoPoolManager whos job is to find an appropriate pool to add an incoming object to, create new pools, and track/bind pools as necessary (no-oping if a request to bind already bound pool is made).

There will be a similar SurfacePoolManager who does the same for surface pools.

### Materials

Materials are a construct encapsulating shader information with additional parameters and smarts. They are implemented in code, and backed by HLSL.

Material data:

* Shaders
* InputLayout(s) (May be two if the material supports both vertex types, and just downgrades the more complicated one to the simpler via an input layout)
* Constant buffers
* Method to apply the material to the pipeline (binds resources, sets per-frame constants, etc…)
* Method to draw an object with the current settings (set per-object constants, draw)

Shaders will have a set of registers reserved for system managed stuff (for example, vertex & index slot 0 are reserved for the active geopool, texture slot 0 is reserved for the current surface pool). Any additional parameters the shaders need can be assigned to non-reserved slots, and it’s up to the implemented Material class to manage those.

### Design

Other tasks for 3D rendering:

* Optimal visibility culling
* Optimal management of the GPU (batches, pooling, state change minimization)

Bucket objects that need rendering into 2 groups (lists): opaque and “might contain transparency”.

Each list is made up of sorted RenderData structures. The list is insertion sorted on each insertion (during culling pass).

struct RenderData

{

Uint64\_t Key;

GameObject\* GameObject;

};

The elements are sorted by their Key, which is comprised of material ID, geopool ID, and surfacepool ID, which will help ensure that we don’t ping pong between objects with very different resources.

When rendering for shadows, we don’t need any materials since we just render into the depth buffer, no lighting or texturing is done. Therefore, the shadow render list(s) can set the Keys to just be the geopool ID.

Render flow:

1. For each camera
   1. Set the viewport for the camera.
   2. Build renderlists (1 opaque, 1 transparent) of objects visible to camera.
   3. Build renderlist of shadow casters per light.
   4. Execute opaque renderlist and/or shadow list(s).
   5. Execute transparent renderlist(s) and/or shadow list(s).
   6. Post processing

A couple of issues we need to sort out:

1. If we have a lot of shadow casting lights, and we can’t hold all generated shadow maps in memory at once, then we’ll have a lot of wasted computation if we have to render all opaques with shadows, then render transparent objects while possibly recomputing shadowmaps that we evicted in order to render shadows on the transparent objects.
2. Transparency has a fair set of challenges, depending on what rendering technology we pick (forward, deferred) and also how many features we want for the transparent objects (do they cast translucent, colored shadows? Do they received shadows? Do they receive rich lighting? etc…)
3. I suggest we simplify for this release, and I propose the following restrictions:
   1. Transparent objects do not cast shadows, or their shadows are fully opaque (looks bad)
   2. Transparent objects can receive shadows and lights (otherwise, particles may look bad)
   3. Keep the number of shadow casting lights limited to what fits in memory without eviction. This will scale up with the power of hardware the user has. We select the closest shadow areas to keep and drop further lights/shadows if we don’t have room.

### Particles

Many effects can be achieved with a simple particle system. We don’t need anything fancy, and we don’t need to support dynamically/physically simulated particles. They can be mostly precanned, with some basic simulation for movement. No collision (smoke will go through walls like in most games, it won’t realistically slide around in the volume of the room), and follows the same rules as other transparent objects (no shadow casting, but receives lighting and shadows).

The number of particles (including option to turn them off) will be determined based on available memory and power of the GPU being used. More beefy hardware can enjoy more particles.

### Draw mechanics

Before an object can be drawn, the following prerequisites must be met by the renderer:

* Required material is bound (and all per-frame values updated)
* Required geopool is bound
* Required surfacepool is bound

Drawing the object requires a few pieces of data, which by default can be applied as global constants to the material on a per-object basis:

* Transform(s) (In the case of a skinned object, there is one per bone)
* Base surface index – which index in the surfacepool do this object’s textures start at

More parameters may be added later if necessary, but this is the absolute minimum. If we find that many levels contain lots of similar objects using the exact same geometry (even if their textures are different), we can use instancing if profiling shows it will improve performance.

The data above would be packed into each instance in the instance buffer to be sent along with the same base vertex & index values used in the DrawInstanced call (since we’re using the same portion of the geopool for all of the objects).