V 1

Efficient Stereo and VR Rendering

Íñigo Quílez

1.1 Introduction

With the recent advent of consumer virtual reality (VR) through inexpensive head-mounted displays, real-time rendering for VR is a topic of great interest to academia, hobbyists, and businesses. From hardware to software developers, from presentation to rendering to content makers, all disciplines involved in visualizing content are rethinking their methods.

From a purely rendering standpoint, VR brings huge challenges, because current content needs to be rendered at high resolution (often twice the amount of pixels of HD), high frame-rate (usually 90 fps), and unusual levels of supersampling or antialiasing, making the amount of pixel computations required easily six times that of a conventional HD game running at 30 fps. This difference between the requirements of traditional 3D rendering and VR will only increase as the display resolutions for VR are expected to grow over the next decade. This disparity requires graphics programmers to change some of the established practices and optimize their engines differently. In this article we'll go through some of the basic ideas and methodologies recommended for efficient rendering in VR.

The first half of the article will focus on high-level architectural design, and the second half will discuss some of the lower-level optimizations available by using modern OpenGL 4.5.

1.2 Engine Design

1.2.1 The Point

Traditional engine design involves performing the posing and rendering of the whole content in a loop. Posing includes animating the characters in a film or cinematic moment and also more complex tasks, such as evaluating physics and simulations in a game. Rendering usually takes the whole pose and projects it into the screen in a single conceptual step, then proceeds to the next frame or iteration of the "pose and render" loop.

This works great for single monoscopic display systems such as a TV or a computer monitor. On the other hand, displays systems for VR are sophisticated and include several displays with stereoscopic images in each. Current VR headsets will typically have one virtual display that is stereoscopic. However, a CAVE VR system consists of five projection walls at 90 degree angles with stereo images. A fancy imaginary, but plausible, head-mounted device might one day contain one central display with stereo and two peripheral mono lower-resolution displays at 45 degree angles. So VR rendering is potentially done with multiple configurations of displays, where a display can be conceptually mapped to a planar projection with a monoscopic or stereoscopic frame buffer.

All these scenarios mean that rendering the whole pose of the game or film in a single step assuming a single pinhole camera doesn't work anymore. The renderer could treat each display and each left/right eye in each display as an individual render task, but that would make for a poorly performing renderer, since there are missed opportunities for reusing data which is neither eye nor display dependent.

1.2.2 Identifying Contexts

In order to exploit this optimization opportunity we need to identify the context in which data exists uniquely:

- *Frame context.* This is the context in which things that are constant across the frame exist and need to be processed. These include posing the worlds animated characters and performing sound simulations or global LOD computations, but also some rendering of global elements such as global shadow maps, reflection maps, cloud animated textures, water caustic maps, etc. Basically, anything that needs to be rendered and is view-orientation independent belongs in the frame context.

  The main subjects of the frame context are the viewer position (or player or camera) and perhaps the frustum that is the union of all the display frustums.

- *Display context.* This is the context where we want to do display-specific work, such as rendering a shadow map for the frustum of the display which can be shared across both stereo eyes on the display, or rendering distant geometry which won't benefit from stereo disparity and therefore can be rasterized once and shared between both eye projections. It is also in this context where we compile our command buffers or `glMultiDrawIndirect` buffers that will contain all the rendering work needed for the left and right eyes belonging to this display.

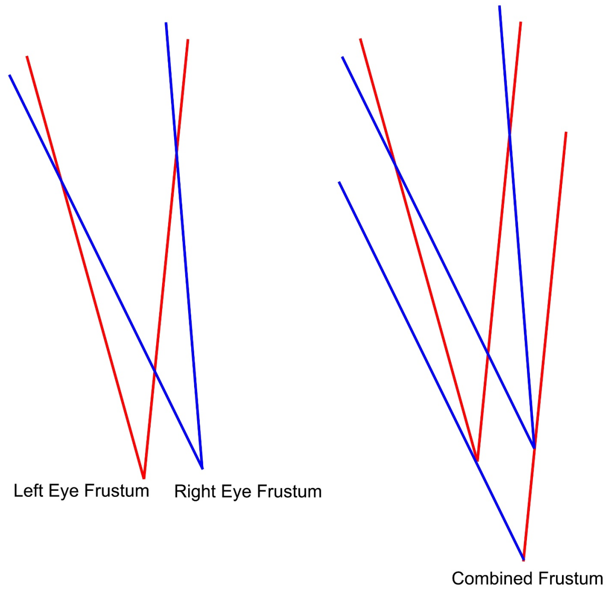

Figure 1.1. Viewer position, display frustum, and eye frustums and positions.

The subject of the display context is the frustum that is the union of the left and right eye projections for this display. Note that the apex of this frustum does not correspond to the viewer's position nor the midpoint location of both eyes. Usually the apex of the display frustum will be behind the viewer's head position and potentially skewed to one side as well if the displays are not parallel to the line connecting the eyes, such as in a CAVE, a tilted HMD, or peripheral display of an HMD (see Figure 1.1.

- *Eye context.* In this context belong the elements that are absolutely eye dependent and cannot be reused across left and right eyes, such as close-by objects which do need stereo disparity. When talking of shading, the eye context also includes the data necessary for correct specular computations.

  The subject of the eye context is the actual projection of the scene for this eye and its frustum (although it's not worth doing culling on this frustum). The center of projection (or apex of the frustum) is located at the position of the viewer's corresponding eye's eyeball in the world.

Given these three contexts, Listing 1.1 shows how a VR-ready engine could organize its rendering.

```
ComputeFrame( k )
{
    DoProcessFrame( k );                        // Compute animation, sound,
                                                //  LOD computation, simulations
    DoRenderFrame( k );                         // Render reflection maps,
                                                // clouds layers, global shadow maps

    for( int j=0; j<numDisplays; j++ )          // numDisplays = 5 for a CAVE, 1 for a
     HMD
    {
        DoProcessDisplay( k, j );               // Do frustum culling, compile draw
                                                // indirect buffers

        DoRenderDisplay( k, j );                // Render eye-shared shadow maps,
                                                // and distant geometry

        for( int i=0; i<2; i++ )                // stereo
        {
            DoRenderEye( k, j, i );             // Render regular geometry with parallax
                                                // and postpro
        }
   }
}
```

Listing 1.1. Organizing rendering in a VR-ready engine.

1.2.3 Culling and Contexts

Depending on your VR setup, frustum culling might need to happen at different stages. Most typically, for a single display stereoscopic system, the frame and display contexts can be considered the same, and we do the frustum culling in either. All that matters is that culling is not done per eye since both eyes' projections overlap considerably.

However, for a three-display system (a central and two peripheral displays), it might be a good idea to do a rough frustum-culling pass in the frame context and then do three frustum-culling passes at the display context on the results of the previous step. Depending on the nature of the content, skipping the cull at frame level might be an option.

Beware that this analysis is implementation agnostic. Even in the case of modern multi-projection enabled GPUs (such as NVidia's Pascal-powered hardware) where the vertex shader (or geometry or tessellation shader) can project the geometry into multiple viewports with different projection matrices and viewport transforms (hence, NDC locations) in a single non-instanced geometry pass, there's still the potential benefit of performing per-display (viewport) culling in the shaders themselves.

It might also happen that in such a multi-projection-enabled GPU, the limited amount of available viewport/projections is already in use for other purposes, such as stereoscopic lens-matched shading (which consumes eight viewports), so we might need to still iterate and render to the displays one at a time.

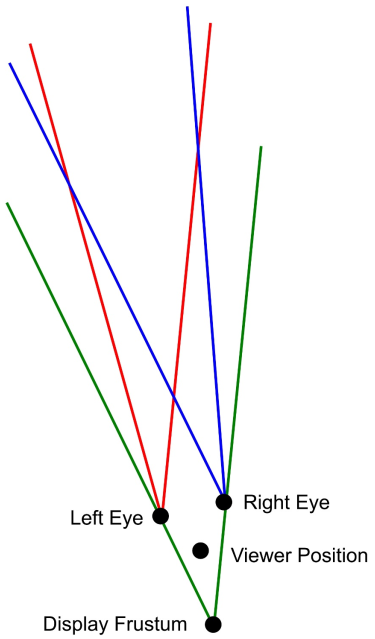

Figure 1.2. Combining left and right eyes frustums to create the Display frustum.

When it comes to combining display frustums to create a frame frustum, picking the correct clipping planes of each frustum gives a good solution.

When combining left- and right-eye projection frustums in order to generate a display frustum, a more elaborate technique is required. Something that works well is constructing a new frustum like this: the right plane of the new frustum will be constructed by taking the left eye's right-frustum plane and choosing a parallel plane that passes through the right eye's position. The left plane is constructed similarly, and the top and bottom planes can be picked from either eye, as shown in Figure 1.2.

Note that the combined frustum does not have its apex at or even near the average position of the eyes frustum's apices. Also, the near clipping plane of this frustum will need to be pushed forward to match the near clipping plane position of either eyes' frustum in order to perform efficient display-frustum culling.

1.2.4 Shader Uniforms per Context

At render time, the different shading stages will need to access data from the different contexts. For example, a tessellation shader might need to access the viewer position in order to choose the level of detail of a given mesh, which should not be computed based on an individual eyes position (as you would normally do). Conversely, the specular response of a material's surface should be computed based on the eye position for which we are rasterizing and not based on the viewer's (or camera's) position, as we would usually in a regular 3D engine (see Listing 1.2).

```
layout (row_major, binding=0) uniform FrameContext
{
    mat4x4      mLocation;
    mat4x4      mLocationInverse;
    vec3        mViewerPos;
    vec3        mViewerDir;
    float       mTime;
}frame;

layout (row_major, binding=1) uniform DisplayContext
{
    mat4x4      mLocation; // think modelview
    mat4x4      mLocationInverse;
    mat4x4      mProjection;
    mat4x4      mProjectionInverse;
    vec3        mPosition; // apex of the frustum
    vec2        mResolution;
}display;

layout (row_major, binding=2) uniform EyeContext
{
    mat4x4      mLocation; // think modelview
    mat4x4      mLocationInverse;
    mat4x4      mProjection;
    mat4x4      mProjectionInverse;
    vec3        mPosition; // apex of the frustum
}eye;
```

Listing 1.2. Shader uniforms grouped for virtual reality.

Please note again how the display and eye uniform blocks have both location and projection matrices which are different.

This data breakup works great conceptually. Conveniently, these three pieces of data can be updated at different time granularities (once per frame, per display, and per rendered eye).

However, for performance reasons, we'll shortly see that rendering the left and right eyes of a stereoscopic display is possible and recommended, so it makes sense to group the display and the two instances of the eye context together in practice to minimize the number of state changes in the renderer (see Listing 1.3).

```
layout (row_major, binding=0) uniform FrameContext
{
    mat4x4      mLocation;
    mat4x4      mLocationInverse;
    vec3        mViewerPos;
    vec3        mViewerDir;
    float       mTime;
}frame;

layout (row_major, binding=1) uniform DisplayContext
```

```
{
    mat4x4      mLocation; // think modelview
    mat4x4      mLocationInverse;
    mat4x4      mProjection;
    mat4x4      mProjectionInverse;
    struct
    {
        mat4x4      mLocation; // think modelview
        mat4x4      mLocationInverse;
        mat4x4      mProjection;
        mat4x4      mProjectionInverse;
        vec3        mPosition; // apex of the frustum
    }mEye[2];
    vec3        mPosition; // apex of the frustum
    vec2        mResolution;
}display;
```

Listing 1.3. Rendering-efficient shader uniform contexts.

1.2.5 Other Considerations

Before diving into efficient stereo rendering, we'd love to make the note that a VR-ready engine should respect a real physical scale for the scenes and objects in it. Make sure modellers are modelling and lighters are lighting your world using real units, otherwise the VR experience will be wrong. Hacking scene scales or eye-separation units will lead you and your team to an ugly, messy situation very quickly. For the same reason that ad-hoc shading models have been replaced by physically based shading and materials in many pipelines, you want your VR to be physically based, not random-scale-factors-based. When done properly, the size of objects in VR are unambiguously perceived as the correct size. If you currently don't, don't compensate with scale factors—go and fix your math or the assets.

1.3 Stereo Rendering

Within a given display, rendering in stereo is normally sped up by sharing the left- and right-eye rendering work to some degree. Different techniques achieve a greater or lesser degree of sharing when trying to render both eyes simultaneously, but in general they are all faster than rendering each eye separately.

At the most basic level, instead of rendering all the objects for the left eye and then all the objects for the right eye, one can iterate all the objects and render them twice sequentially, first for the left and then for the right eye. This reduces the number of state changes (texture, shader, and uniform bindings) to half, which is always good. In this case the rendering architecture is as given in Listing 1.4.

This requires allocating frame-buffer space for both left and right eyes, which can be expensive in some cases. For example, it is recommended to use 8X MSAA buffers for rendering in VR. Under these conditions, the cost of duplicating the depth and color buffers, or G-Buffer if in a deferred engine, can be too high.

```
ComputeFrame( k )
{
    DoProcessFrame( k );                       // Compute animation, sound,
                                               // LOD computation, simulations
    DoRenderFrame( k );                        // Render reflection maps, clouds layers,
                                               // global shadow maps

    for( int j=0; j<numDisplays; j++ )         // numDisplays = 5 for a CAVE, 1 for HMD
    {
        DoProcessDisplay( k, j );              // Do frustum culling, compile draw
                                               // indirect buffers

        DoRenderDisplay( k, j );               // Render eye-shared shadow maps,
                                               // and distant geometry

        DoRenderLeftAndRightEye( k, j );       // Stereo render regular geometry
                                               // with parallax and postpro
   }
}
```

Listing 1.4. Rendering architecture.

However, the efficiency gained by rendering left and right eyes at the same time is well worth the trouble if possible.

An even greater speed-up can be achieved by reducing the amount of draw calls as well. These days the standard way to do so is to use instancing in combination with multiple viewports or layered textures.

Using multiple viewports involves setting up a large texture that will hold both the left- and right-eye rendered content, with twice the size of the individual framebuffers, by calling `glTextureStorage2D` as usual for the color and depth (or G-buffer). Then the two viewports are prepared by using `glViewportIndexedf`, making each viewport occupy half the space reserved in the texture.

Another way to set up the framebuffers that is somewhat easier is by using layered textures. It involves creating only a single texture of the original resolution with two layers, one for each eye, by calling `glTextureStorage3D`.

Once the framebuffer is ready and multiple viewports have been set in the case of the non-layered framebuffer approach, the rendering happens by using the instancing-flavored version of the usual render commands, such as `glDrawElementsInstanced`. The point is to have the draw function called only once but set the invocation count to two, so that the graphics pipeline will be invoked twice for the whole piece of geometry. This alleviates the stress on the graphics driver, which no longer has to process the render call twice.

Then, in the shaders, the geometry must be routed to the left or right viewport or layer (depending on the choice made) for each one of the two instancing invocations. This can be done by using `gl_ViewportIndex` and `gl_Layer`, respectively, in the vertex, geometry or tessellation shader. Note that though this is primitive routing, which usually is done in the geometry or tessellation evaluator, it can still be done at vertex level in the vertex-shader stage as well, provided the extension GL_ARB_shader_viewport_layer_array is present in the system. The viewport or layer to which a triangle will be routed depends on the index set by the provoking vertex of the triangle, which is GL_LAST_VERTEX_CONVENTION by default.

This mechanism of using vertex shader-level routing and indexed viewports or using layered rendering has two benefits. First, the impact in the code is minimal, since there's no need to add geometry shaders to the system to do the routing. Second, we avoid using geometry shaders, which are not recommended because of their poor performance. Furthermore, since we are using viewport arrays or texture layers, we also do not need to set up custom clipping planes to prevent leaking pixels across eyes. Basically, the technique is very non-intrusive and trivial to implement without any serious modifications to the engine.

Since we already set up our uniform blocks to have all the display- and eye-context information ready, the only changes needed to make a regular shader work in stereo is the following:

```
// regular 3D rendering vertex shader:

vec3 worldPos =  ;

gl_Position = display.mProjection *
              display.mLocation *
              vec4( worldPos, 1.0 );


// instanced stereo rendering vertex shader:

vec3 worldPos =  ;

gl_ViewportIndex = gl_InstanceID;
gl_Position = display.mEye[gl_InstanceID].mProjection *
              display.mEye[gl_InstanceID].mLocation *
              vec4( worldPos, 1.0 );
```

Listing 1.5. Regular and stereo view vertex shader code.

Note again the lack of gl_ClipDistance[0] or any other hacks to deal with a single viewport and how minimal the change is. The geometry is still sent twice down the input assembler and vertex shader, but both state changes and render calls have been reduced to half.

If hardware instancing was already in use for fast rendering of object instances, then the number of instances to be submitted should be twice the original. In the shaders, dividing gl_InstanceID by two can be used to index into the correct instance data, such as object to world matrix and material information, and the last bit of the same built-in variable can be used to index into the correct viewport (Listing 1.6).

A similar option, which makes use of multiple viewports but does not require instancing, is to use viewport multicast, which is available through the NV_stereo_view_rendering and NV_viewport_array2 extensions. This method

```
// instanced stereo rendering of instanced objects, vertex shader:

int eyeID = gl_InstanceID & 1;
int insID = gl_InstanceID >> 1;

vec3 worldPos =  ; // use insID to index object to world matrix

gl_ViewportIndex = eyeID;
gl_Position = display.mEye[eyeID].mProjection *
              display.mEye[eyeID].mLocation *
              vec4( worldPos, 1.0 );
```

Listing 1.6. Eye selection using the instance ID.

processes the geometry only once, which makes it potentially faster than stereo instancing. By simply performing both left- and right-eye transformation in the vertex shader (or geometry or tessellation), one can avoid submitting the geometry twice. The left eye's clip-space coordinates are typically output to `gl_Position` as usual, and the right's eye coordinates to `gl_SecondaryPosition`. At the same time, `gl_ViewportMask[0]` and `gl_SecondaryViewportMask[]` must be used to route each polygon to the correct viewport (Listing 1.7).

```
// regular 3D rendering vertex shader:

vec3 worldPos =  ;

gl_Position = display.mProjection *
              display.mLocation *
              vec4( worldPos, 1.0 );


// stereo rendering vertex shader:

vec3 worldPos =  ;

gl_ViewportMask[0] = 1;
gl_Position = display.mEye[0].mProjection *
              display.mEye[0].mLocation *
              vec4( worldPos, 1.0 );

gl_SecondaryViewportMask[0] = 2;
gl_SecondaryPosition = display.mEye[1].mProjection *
                       display.mEye[1].mLocation *
                       vec4( worldPos, 1.0 );
```

Listing 1.7. Viewport multicasting.

1.4 Conclusion

Architecting a rendering engine to work in VR is not difficult, but needs some care from developers who are accustomed to traditional single-display monoscopic rendering. There no longer exists a single frame of reference or "context" that combines viewer position, display, and eye spaces. These three concepts are different and play different roles at different moments along the rendering process.

Rendering stereo can be done at less than twice the cost of a mono rendering if instancing and other techniques are used which can reduce the renderer's state changes, the number of draw calls, and the amount of geometry flowing down the shading pipeline.