High performance stereo rendering for VR

Timothy Wilson
San Diego Virtual Reality Meetup
January 20, 2015

Stereo performance concerns

Myth: Everything has to be drawn twice - not exactly...

- Only *view dependent* items need to be drawn twice.
- Shadows do not need to be rendered twice - generally dependant on light transform.
- Can use head transform for PSM or CSM.
- Some reflections are view independent (ex. cube maps)
- Pixel fill requirements are not doubled. Because a single display is split, not shared for both eyes. HMDs *can* have higher res displays (1440p, 2160p), but the effect is the same for an application on the desktop with increasing monitor resolution.
- Post processing is performed across the entire physical frame buffer visiting pixels only once.
- Graphics API calls *do not* need to be doubled, which is what this slide stack covers.

Some tricks to save vertex performance on monitors (normal mapping, parallax mapping) no longer fools the eye at close distances, so we need real geometry.

Conclusion: Vertex processing is inescapably doubled, focus on reducing vertex cost.

One camera per eye

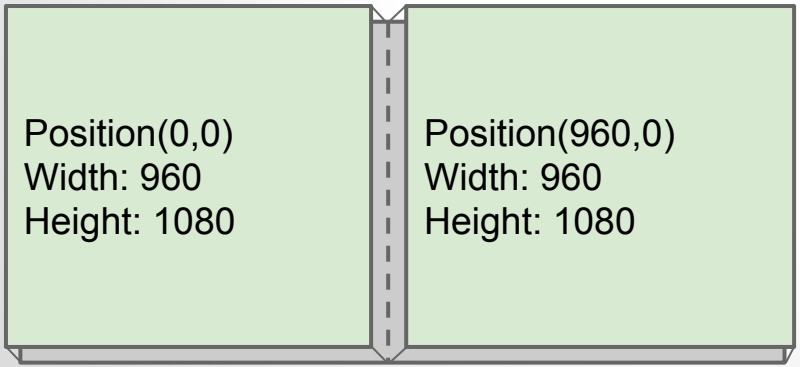
Left and right eye have different positions and viewports.

IPD = (Interpupillary distance), on average 66mm between each eye, results in two view transforms.

Oculus Rift SDK provides these two matrices.

Stereo Viewports

Viewports used to divide up single display into left and right views. A viewport is a position, width and height on the physical framebuffer, in pixels. Rasterizers only generate pixels inside of the current viewport, even if it only covers a portion of the display



Example DK2 Viewports

Approaches to stereo rendering

- Submit entire scene with left viewport and view transform, then re-submit scene again for right.
- Iterate over scene, but submit each object twice, changing viewport and view transform between each draw call.
- Submit entire scene using hardware to double the triangles with the Geometry Shader.
- Submit entire scene using hardware to double the triangles with instancing.

Submit entire scene twice

Pros:

Easy to implement.

Cons:

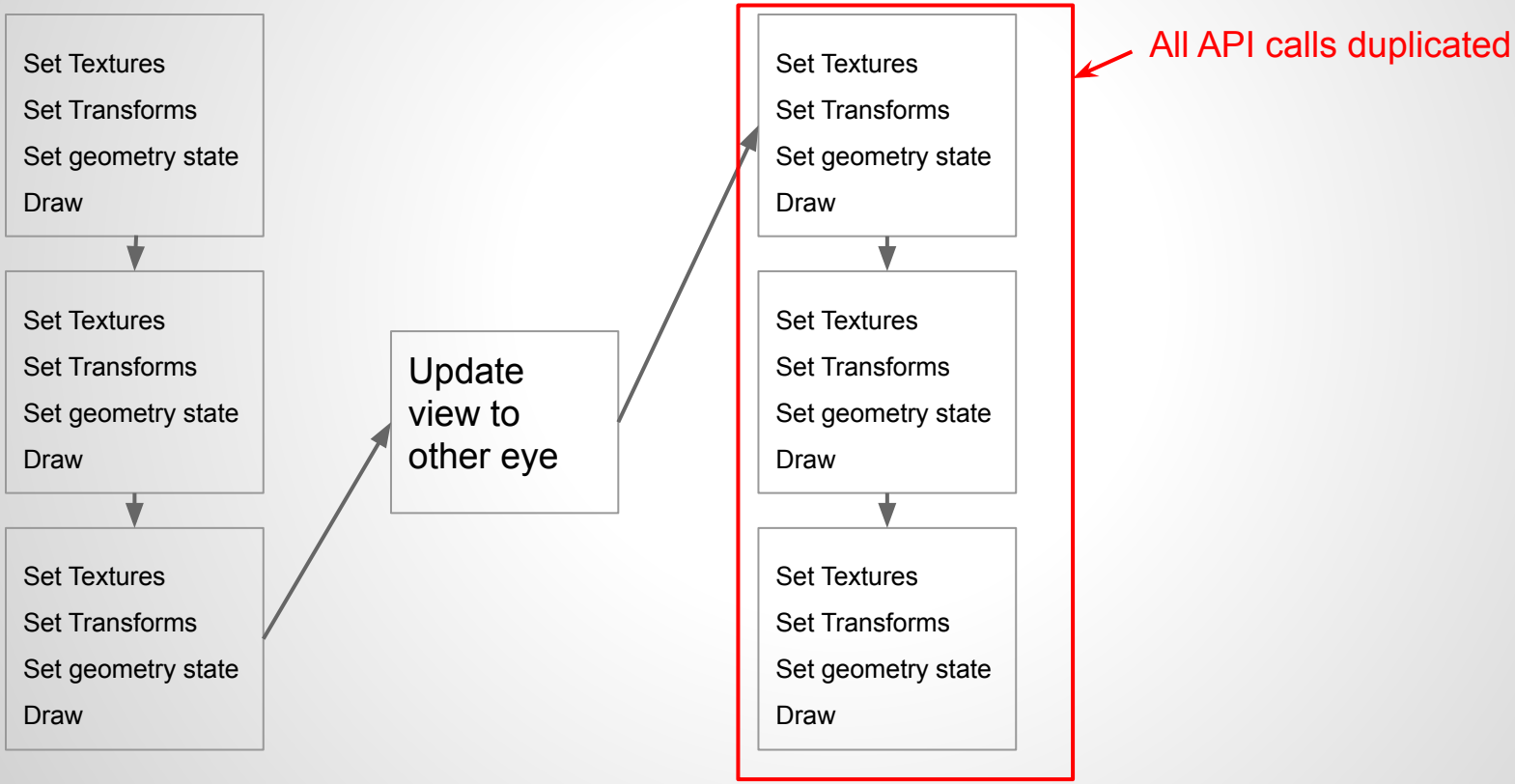
Draw calls *and* state changes are doubled. Not friendly to CPU or GPU caches.

Naive implementations may re-calculate shadows or other non-stereo resources per eye.

Popular game engines utilize this method to your detriment.

D3D11CommandLists might reduce overhead for the 2nd submit, but the author has not benchmarked this.

Submit entire scene twice



Submit each object twice

Pros:

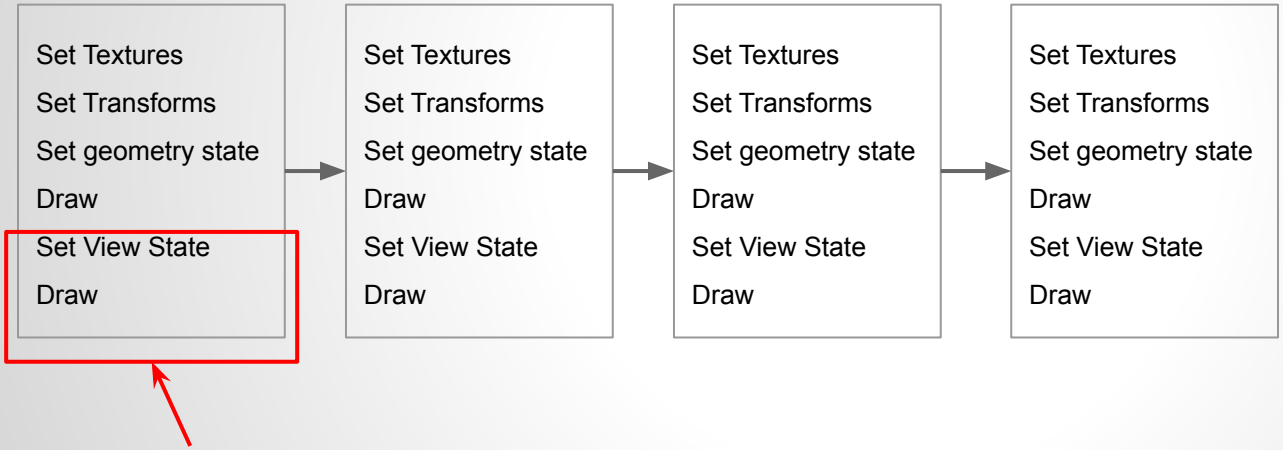
Somewhat easy to implement.

API calls reduced significantly. Only view state, transform and viewport, is doubled.

Cons:

Draw calls are still doubled.

Submit each object twice



Added view state, and second draw, bulk of state is ‘reused’

Hardware to draw stereo: Geometry Shader

Pros:

Hardware does heavy lifting.

Multiple D3D11 viewports handled automagically using `SV_ViewportIndex`.

Pixel shaders left untouched.

CPU side has minor changes to set up additional left/right eye data and bind GS.

No state duplication, no additional draw calls. Can handle texture-per-eye rendering with `SV_RenderTargetArrayIndex`, without additional cost.

Cons:

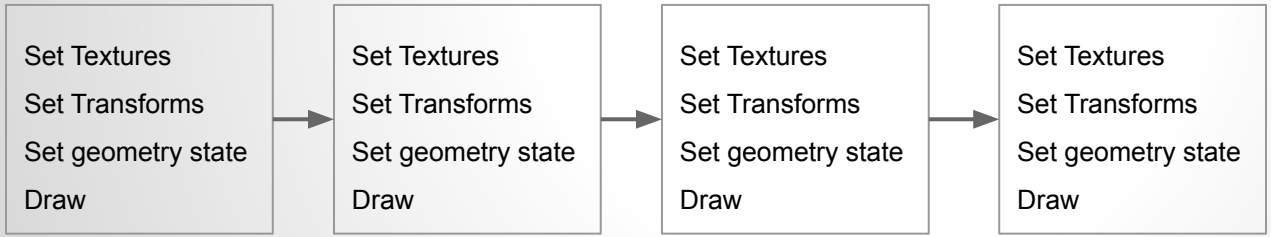
Platform may not have concept of a GS, (OpenGL ES) or even expose the functionality in the game engine.

Can be invasive to vertex shaders.

Measured 3x or more slower in geometry throughput in testing. (587 µsec versus 150 µsec for a sample mesh on an nVidia 660 GTX)

Hardware draws both eyes at once

In both GS and instanced case, API usage looks like a typical desktop application.



Hardware to draw stereo: Instancing

Pros:

Fast - doubles hardware geometry processing only. 3 times faster on the GPU than using the GS to amplify geometry.

No additional state changes.

Minor change to vertex shaders.

Cons:

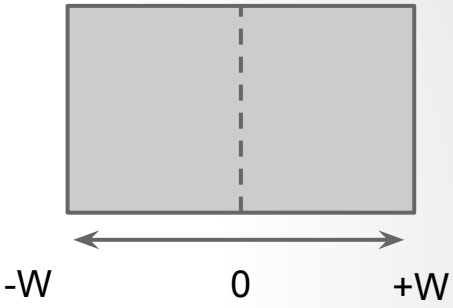
Won’t support per-eye render target, currently not an issue.

Makes regular instancing a tiny bit more complicated.

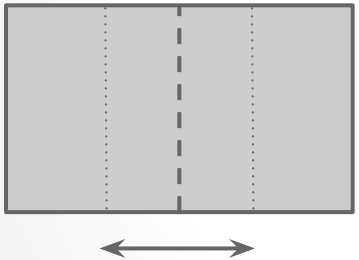
Graphics API may not support dynamic clipping. (OpenGL ES)

Using instancing to draw stereo: the magic

In clip space, X ranges from -W to +W



1. Scale X by 0.5

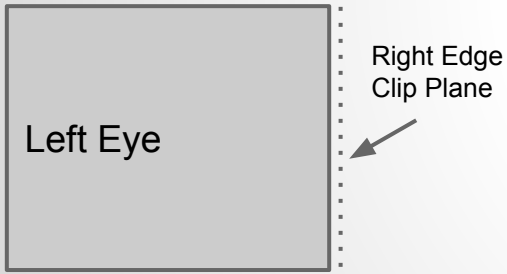


2. Shift X coordinate by half of W, left or right depending on the eye.
3. Use hardware triangle clipping to prevent spill over into the opposite eye.

Using instancing to draw stereo

For the left eye, clip against the right edge.

For right eye, clip against the left edge.



example: In clip space, the right frustum plane coefficients are (-1,0,0,1)

Render stereo with hardware: Details

Use the Instanced version of the draw API

1. in D3D11: `DrawIndexedInstanced()` or `DrawInstanced()`
2. Use an instance count of 2 for a single object, or multiply your original instance count by 2.
3. In the vertex shader, the `SV_InstanceID` is used as an index into 3 arrays.
4. Clip/Cull against left or right edge of viewport using outputs with `SV_ClipDistance`, and `SV_CullDistance` semantics

Render stereo with hardware: Code

Pseudocode follows:

```
Matrix WorldToEyeClipMatrix[2] // computed from SDK
Vector4 EyeClipEdge[2]={(-1,0,0,1), (1,0,0,1)}
float EyeOffsetScale[2]={0.5,-0.5}

uint eyeIndex = instanceID & 1   // use low bit as eye index.

Vector4 clipPos = worldPos * WorldToEyeClipMatrix[eyeIndex]
cullDistanceOut.x = clipDistanceOut.x = clipPos · EyeClipEdge[eyeIndex]

clipPos.x *= 0.5; // shrink to half of the screen
clipPos.x += EyeOffsetScale[eyeIndex] * clipPos.w; // scoot left or right.
clipPositionOut = clipPos
```

Contact the author

Timothy Wilson
tim@darkoaksoftware.com

Feedback and comments welcome.