8

Variable Precision Pixel Shading for Improved Power Efficiency

Rahul P. Sathe

Intel

We propose a technique for selectively reducing the pixel shader precision for the purpose of efficient rendering in a low-power environment. Typically, pixel shading consumes the largest percentage of the power on GPUs [Pool 2012]. Modern APIs like Direct3D 11.1 allow users to reduce the shading precision to meet the low power requirements, but they don't allow doing so adaptively. Rendering at reduced precision can potentially produce artifacts in the rendered image. These artifacts are most noticeable where the user's attention is focused, which is typically at the center of the screen or, in the case of an eye tracking device, at the point provided as input by such a device. This chapter presents a scheme to render the scene at the highest precision where user's attention is focused and gradually reduce the precision at the points farther away from the focal point.

8.1 Introduction and Background

Image artifacts are more acceptable in some parts of the screen than others. Techniques proposed in the past like foveated 3D graphics [Guenter et al. 2012], coarse pixel shading [Vaidyanathan et al. 2014], and extending the graphics pipeline with adaptive, multirate shading [He et al. 2014] try to exploit this observation by reducing the sampling rate in less important parts of the screen. But none of these techniques propose reducing the shading precision. One can write a pixel shader that dynamically chooses the precision depending on the region of the screen that is being shaded. However, such a shader is less efficient because of the reduced SIMD usage due to the presence of dynamic control flow.

Forward shading refers to a technique where pixel shading is done immediately after rasterization (or after early and hierarchical Z/stencil testing, when applicable). Forward shading typically suffers from the issue of overdrawing the same pixel multiple times. Deferred shading overcomes the overdraw issue by decoupling the visibility determination and the shading. In the first pass, it writes out the pixel shader input (interpolated attribute values) into a buffer commonly called the *G-buffer* (Geometry buffer). In the second pass (a full-screen pass or a compute shader), it loads the G-buffer values and evaluates shading.

The key to lowering the pixel shader precision while shading certain parts of the screen is the ability to bind a lower precision shader while shading those pixels. With forward shading, one does not know where the polygons being shaded will land at the time that the pixel shader is bound. One can avoid shading regions that need different precision with the use of Z-buffer or stencil mask, but to shade the parts that require a different precision in a separate pass, the entire geometry processing stage needs to be done again. As a result, it is not efficient to use a specialized low-precision shader with the forward rendering. During the shading phase of the deferred shading process, one can bind a shader with a particular precision for shading the relevant portions of the screen. One can then repeat this with a different precision for different portions of the screen without processing the geometry multiple times. As a result, variable precision pixel shading fits well in the deferred rendering pipeline. We propose using our technique in conjunction with the tiled deferred renderer proposed by [Lauritzen 2010].

Texturing is one area that could be very sensitive to the precision. A small change in (u,v) values as a result of lowering the precision could mean vastly different looking texels. This is more likely to happen for large textures. Fortunately, texturing is typically done during the forward pass where we continue to use standard full-precision shading.

8.2 Algorithm

G-Buffer Generation

Just like in a normal deferred shading engine, our algorithm starts off by generating a G-buffer by writing out shading inputs at the pixel center. The G-buffer stores the derivatives of the view-space depth values in addition to the other surface data (position, normal, UVs, TBN basis, etc.) required for evaluating the BRDF during the shading pass. View-space depth derivatives are calculated by first multiplying the position with the camera-world-view matrix and evaluating

the `ddx_coarse()` and `ddy_coarse()` functions. We use spherical encoding to store the surface normal as a `float2` to save some G-buffer space and bandwidth. We pack the specular intensity and the specular power in the other two components to occupy a full `float4`. The G-buffer layout is given by the following structure.

```
struct GBuffer
{
    float4 normal_specular   : SV_Target0; // normal and specular params
    float4 albedo            : SV_Target1; // albedo
    float2 positionZGrad     : SV_Target3; // ddx, ddy of view-space depth
    float positionZ          : SV_Target4; // view-space depth
};
```

Shading Passes

Normally, deferred shading has only one shading pass. But because we propose using different precisions while shading different parts of the screen, we have to perform multiple shading passes, one corresponding to each precision level. The compute shader is launched such that one thread group processes one region of the screen, henceforth referred to as a *tile*. Figure 8.1 shows how we statically mark the regions on the screen. Region A corresponds to the center region where image artifacts would be most noticeable. As a result, that needs to be shaded at the highest available precision. Regions B and C are further away from the center of the screen, so artifacts are progressively less noticeable in those regions.

Starting with DirectX 11.1 on Windows 8, new HLSL data types were introduced that allow applications to use lower precision, namely `min16float` and `min10float`. One can find out which types are supported on a given GPU by using the following snippet.

```
D3D11_FEATURE_DATA_SHADER_MIN_PRECISION_SUPPORT  minPrec;

hr = pd3dDevice->CheckFeatureSupport(
    D3D11_FEATURE_SHADER_MIN_PRECISION_SUPPORT, &minPrec, sizeof(minPrec));

if (FAILED(hr)) memset(&minPrec, 0, sizeof(minPrec));
```

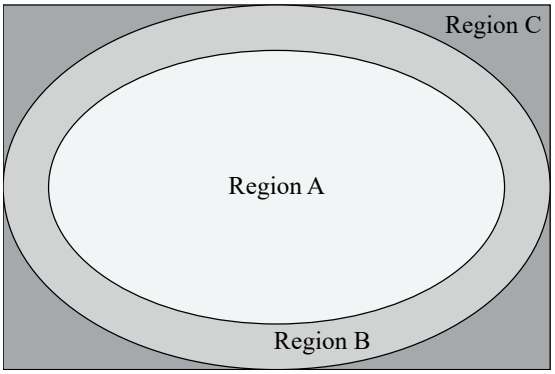

Figure 8.1. The screen is statically partitioned into the elliptical regions A, B, and C. Axis dimensions for region A are chosen arbitrarily and can be set to the desired values with the experimentation.

The value of `minPrec` in the above code tells the user what precisions are supported by the GPU for different shader types. If available, we compile three separate versions of the deferred shader, the one with the full precision for region A, the one with `min16float` precision for region B and the one with `min10float` precision for region C. If the `min10float` type is not supported, then we just use `min16float` for region C and full precision for regions A and B. The major and minor axes for elliptical regions corresponding to region A are a function of viewing distance. Since we use a compute shader for deferred shading, we do not mark these regions in the stencil buffer to selectively shade those regions. Instead, we check whether the tile on the screen is within the region of interest that is being shaded. We simply skip rendering if the tile is outside of region being shaded and free the corresponding hardware resources. Since the regions of interest are elliptical in shape, we use the following equation to test whether a point (x,y) is inside an ellipse centered at (h,k) with the major and minor axis lengths a and b.

$$\frac{(x-h)^2}{a^2}+\frac{(y-k)^2}{b^2}<1$$

We perform the test at the each of the tile corners, and we shade all the pixels of the tile only if all the corners of the tile are within the region being shaded. If multisampling is enabled during the G-buffer generation pass, one should be careful to evaluate the above equation at the corners of the tile and not at the cen-

ters of the corner pixels. The reason for doing this is the fact that the sample locations can vary from one hardware vendor to another and only way to guarantee that all the samples in the tile are in a particular region is to do the in-out test at the tile corners. If multisampling is enabled during the G-buffer generation pass, one should use the technique discussed in [Lauritzen 2010] during the lighting pass in order to shade at the sample rate only when necessary.

After the in-out test for the entire tile, the compute shader is conceptually divided into the following phases:

1. Light tiling phase.
2. Analysis phases (if multisampling is enabled).
3. Pixel shading phase.
4. Sample shading phase (if multisampling is enabled).

At the end of each phase, the threads within the thread group synchronize. The details of the phases can be found in [Lauritzen 2010]. Listing 8.1 shows the pseudocode for the shading pass.

Listing 8.1. Pseudocode for the shading pass. It has four phases: the light tiling phase, the analysis phase, the pixel shading phase, and the sample shading phase. Phases are separated by a call to `Groupsync()`.

```
#define GROUP_DIM 16
#define GROUP_SIZE (GROUP_DIM * GROUP_DIM)

groupshared uint sMinZ, sMaxZ; // Z-min and max for the tile.

// Light list for the tile.
groupshared uint sTileLightIndices[MAX_LIGHTS];
groupshared uint sTileNumLights;

[numthreads(GROUP_DIM, GROUP_DIM, 1)] // Coarse pixel is NxN.

void ComputeShaderTileCS(...)
{
    // Check to see if each of the corners of the tile lie within the
    // region being shaded. Proceed only if the tile lies inside.
    Groupsync();
```

```
// Load the surface data for all the pixels within NxN.
// Calculate the Z-bounds within the coarse pixel.
// Calculate min and max for the entire tile and store as sMinZ, sMaxZ.

// One thread processes one light.
for (lightIndex = groupIndex..totalLights)
{
    // If light intersects the tile append it to sTileLightIndices[].
}

Groupsync();

// Read the lights that touch this tile from the groupshared memory.
// Evaluate and accumulate lighting for every light for top left pixel.

// Check to see if per sample lighting is required.
bool perSampleShading = IsPerSampleShading(surfaceSamples);
if (perSampleShading)
{
    // Atomically increment sNumPerSamplePixels with the read back.
    // Append the pixel to the sPerSamplePixels[].
}
else
{
    // Store the results in the intermediate buffer in groupshared or
    // global memory OR if no per pixel component, splat the top-left
    // pixel's color to other pixels in NxN.
}

GroupSync();

uint globalSamples = sNumPerSamplePixels * (N * N - 1);
for (sample = groupIndex..globalSamples..sample += GROUP_SIZE)
{
    // Read the lights that touch this tile from the groupshared memory.
    // Accumulate the lighting for the sample.
    // Write out the results.
}

GroupSync();
```

```
}
```

8.3 Results

We measured the image quality on Intel Core i7-6700K 8M Skylake Quad-Core running at 4.0 GHz. Intel hardware does not support `min10float`, so our screen was divided in two regions. Regions A and B were shaded with the full-precision shader, and region C was shaded with the half-precision shader. We used the assets that we thought were representative of real game assets. We distributed 1024 lights in our scenes.

Figure 8.2 shows images produced at different shading precisions. The top row of the Figure 8.2 shows the images rendered at 1600×1200 resolution with a full-precision shader used for the rendering pass. The second row shows the same scenes rendered with the half-precision shader used for shading every pixel on the screen. The third row shows the images where region B was shaded with full precision and region C was shaded with half precision during the rendering pass. The last row shows scaled (100 times) image differences between screenshots with full precision and mixed precision (rows 1 and 3). The PSNR for the mixed precision with respect to full precision was 41.22 for the power plant scene and 38.02 for the Sponza scene.

8.4 Discussion

Following the trajectory of the evolving GPU languages, dynamic binding in the shaders is a real possibility in the near future. With dynamic shader binding, there won't be a need to bind a specialized shader prior to the draw or dispatch call. With this restriction removed, the technique could be used during forward shading as well, but one has to be mindful of texture sampling issues and SIMD efficiency when using such a technique with forward rendering.

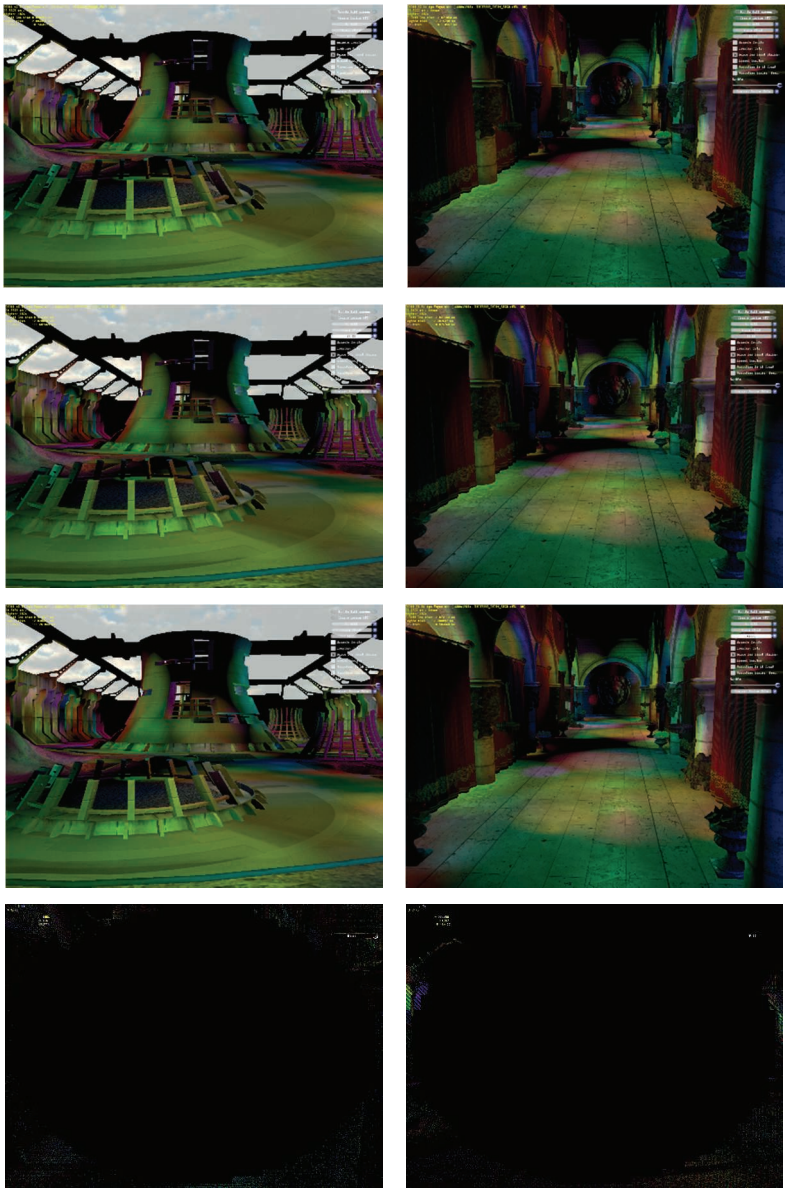

Figure 8.2. Images in the top row were rendered entirely with the full-precision shader, and images in the second row were rendered entirely with the half-precision shader. The third row shows the images when screen was partitioned into regions B and C as per Figure 8.1, region B was shaded at full precision, and region C was shaded at half precision. The last row shows the differences between rows 1 and 3 scaled 100 times.

Acknowledgements

Thanks to Tomas Akenine-Möller and Karthik Vaidyanathan for their help with writing this chapter. Tomas Akenine-Möller is a Royal Swedish Academy of Sciences Research Fellow supported by a grant from the Knut and Allice Wallenberg foundation.

References

[Guenter et al. 2012] Brian Guenter, Mark Finch, Steven Drucker, Desney Tan, and John Snyder. "Foveated 3D Graphics". *ACM Transactions on Graphics*, Vol 31, No. 6 (November 2012), Article 164.

[Vaidyanathan et al. 2014] Karthik Vaidyanathan, Marco Salvi, Robert Toth, Tim Foley, Tomas Akenine-Möller, Jim Nilsson, Jacob Munkberg, Jon Hasselgren, Masamichi Sugihara, Petrik Clarberg, Tomasz Janczak, and Aaron Lefohn. "Coarse Pixel Shading". *High Performance Graphics*, 2014.

[Lauritzen 2010] Andrew Lauritzen. "Deferred Rendering for Current and Future Rendering Pipelines". Beyond Programmable Shading, Siggraph Course, 2010.

[Pool 2012] Jeff Pool. "Energy-Precision Tradeoffs in the Graphics Pipeline". PhD dissertation, 2012.

[He et al. 2014] Yong He, Yan Gu, and Kayvon Fatahalian. "Extending the graphics pipeline with adaptive, multi-rate shading". *ACM Transactions on Graphics*, Vol. 33, No. 4 (July 2014), Article 142.