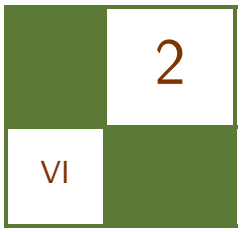

Shader Amortization using Pixel Quad Message Passing

Eric Penner

2.1 Introduction

Algorithmic optimization and level of detail are very pervasive topics in real-time rendering. With each rendering problem comes the question of the acceptable amount of approximation error and the quality vs. performance trade-off of increasing or decreasing approximation error. Programmable hardware pipelines play one of the largest roles in how we optimize rendering algorithms because they dictate where we can add algorithmic modification via programmable shaders.

In this article we analyze one particular aspect of modern programmable hardware—the pixel derivative instructions and pixel quad rasterization—and we identify a new level at which optimizations can be performed. Our work demonstrates how values calculated in one pixel can be passed to neighboring pixels in the frame buffer allowing us to amortize the cost of expensive shading operations. By amortizing costs in this manner we can reduce texture fetches and/or arithmetic operations by factors of two to sixteen times. Examples in this article include 4×4 percentage closer filtering (PCF) using only one texture fetch, and 2×2 bilateral upsampling using only one or two texture fetches. Our approach works using a technique we call pixel quad amortization (PQA). Although our approach already works on a large set of existing hardware, we propose some standards and extensions for future hardware pipelines, or software pipelines, to make it ubiquitous and more efficient.

2.2 Background and Related Work

As the performance of programmable graphics hardware increases exponentially, there has been a steady increase in the complexity of real-time rendering applications, often expressed as the number of arithmetic operations and texture accesses

required to shade each pixel. In response to increasing complexity, much recent research and development effort has focused on methods to reduce pixel processing workload. This includes techniques for simplifying shaders [Olano et al. 03], reusing data from previous frames [Zhu et al. 05, Nehab et al. 07] , or by simply using lower resolutions within a single frame.

One of the first upsampling approaches, known as dynamic video resizing [Montrym et al. 97], dynamically adjusts resolution based on performance, followed by simple bilinear or nearest-neighbor upsampling to a full-resolution frame. While this is effective for controlling pixel workload, artifacts are very noticeable in the upsampled frame. More recent techniques apply geometry-aware upsampling such as the joint bilateral filter [Tomasi and Manduchi 98] from either fixed size [Ren et al. 06] or dynamically resized [Yang et al. 08] frame buffers. What all of these techniques have in common is the requirement of an extra low-resolution pass, followed by upsampling. Our approach differs in that we are able to perform operations at two separate resolutions natively, in the same pass on existing hardware.

2.3 Pixel Derivatives and Pixel Quads

Before describing our technique, it is important to understand a few details of how modern graphics hardware works with respect to texture mapping, and why the pixel shader partial-derivative instructions exist. The need for partial derivatives arises from the simple problem of texture mapping a triangle. As a triangle becomes smaller on screen, one screen pixel will cover many texels, resulting in harsh aliasing unless the texture is adequately sampled. This issue is typically solved in graphics hardware with mipmapping, but a method is needed to compute which mipmap level to use.

Partial derivatives relate the infinitesimal change in one variable to the infinitesimal change in another variable at a particular location. Pixel shader partial derivatives refer to the rate of change of a shader value with respect to the screen-space x- and y-axes. When applied to texture coordinates, this can tell us how fast a texture coordinate is changing on the screen, and thus what mipmap level we should use. Before dependent texture fetches, derivatives could potentially be computed analytically, based on homogeneous barycentric texture coordinates calculated from three triangle vertices. However, dependent texture fetches can depend on arbitrary calculations including data from another texture; thus, no analytic solution exists for these cases.

The only solution remaining is to compute pixel shader derivatives discretely by looking at the value of a texture coordinate in neighboring screen pixels and computing the difference between them. Computing derivatives in this manner is called *forward differencing* or *backward differencing*, depending on whether you look at the pixel in front of or behind the current pixel, to compute the

derivative. For example, in a row of pixels p, $p[i+1]-p[i]$ is a forward difference while $p[i]-p[i-1]$ is a backward difference. Both of these typical schemes fail for parallel graphics hardware however, as they imply a dependency on the order in which pixels are computed. To solve this issue, modern hardware rasterizes triangles in quads, or 2×2 blocks of pixels, and uses custom derivative calculations that depend only on the values within a quad.

Unfortunately, neither the location of quads, nor derivatives within quads, nor even the use of pixel quads, is standardized by modern graphics APIs. Instead these details are left up to the vendor to implement as long as some form of derivative is provided. Since no documentation was provided, we turned to experimentation to determine exactly how derivatives are calculated on modern hardware. Not surprisingly, the implementations we found were exactly what one would expect, given the constraints. First and foremost, on all the hardware we tested, pixel quads have always been stationary in the same locations within the

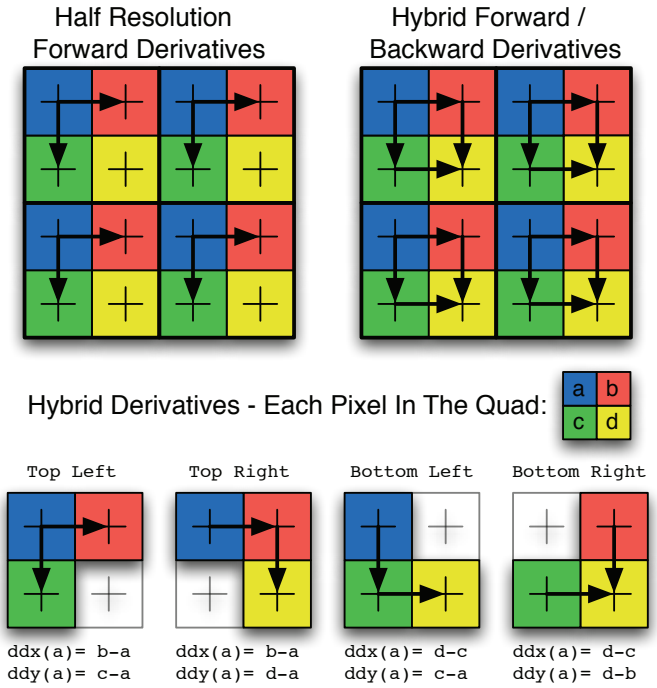

Figure 2.1. Derivative calculations used in practice in modern hardware. We found derivatives were either calculated at half-resolution using forward differencing, or using a hybrid of forward and backward differencing. In the hybrid case we have broken out each pixel's derivatives.

frame buffer; they are essentially double width pixels. Second, we found only two approaches to computing derivatives within these quads, as illustrated in Figure 2.1.

Interestingly, Shader Model 5 in Direct3D 11 has both *coarse* and *fine* versions of the derivative instructions, likely exposing the trade-offs of these two approaches to the developer. Half-resolution derivatives always return the same value within a quad, allowing for optimized texture sampling in some cases, while hybrid derivatives have the potential to provide slightly more accurate results.

It is important to note at this point that although derivative instructions were created to assist with texture mapping, they are not reserved for computing derivatives of texture coordinates. You can use the derivative instructions to calculate the derivative of *any value* in a shader. One obvious question that arises is, what happens when a triangle does not cover all the pixels in a given quad, or if some pixels in a quad are rejected by the depth test? Another question is, how does graphics hardware synchronize all the seemingly independent shader programs such that derivatives can be calculated anywhere? The answer is that in the real shader processing core the "loop" over all the pixels in a triangle is unrolled into blocks of at least four pixels. So all quad pixels are always calculated in lockstep and in parallel, likely even sharing the same set of real hardware registers. The shader program will execute for all the pixels in a quad even if only one pixel is actually needed. In the event that a quad pixel falls outside of a triangle, the values passed down from the vertices are extrapolated using the triangle's homogenous coordinates.

2.4 Pixel Quad Message Passing

Now that we have described pixel quads and why they exist in modern hardware, we turn to how we use them to our advantage. The obvious question is: Can some shaders, or some calculations within shaders, execute at the pixel quad level instead of the pixel level? If a graphics API were to theoretically support a "quad shader" it would lead to another dilemma for parallelism; we would essentially need another pipeline stage. For example, if the hardware was optimized for running 16 shaders in parallel, it would need to cache the output of 16 pixel quad shaders as input to 64 pixels shaders.

What might be a better compromise, and what we found we can already do with today's hardware, is to share values between the pixels in a given quad, but still execute a shader at the pixel level. If we choose problems that have inherent symmetries and are divisible into four identical operations, we can actually use the same pixel shader instructions to perform different "jobs" in each pixel, and then share the values in the quad. Now it should be clear why the derivative instructions are so important. They rely on the difference between two pixels, and thus can be used as a mechanism to share values between pixels, with

some simple arithmetic. We utilize the derivative instructions as message-passing instructions.

2.5 PQA Initialization

In the case of hybrid forward- and backward-differencing, each derivative is simply the positive or negative difference of the current pixel's value with the vertical/horizontal adjacent pixel. With this knowledge, it is easy to calculate what an adjacent pixel's value is. We simply subtract or add the derivative to the current pixel's value, based on the pixel's location in the quad. As an example, for the pixel in the quad in Figure 2.1(top left), we have

$$a + ddx(a) = a + (b - a) = b.$$

For the top-right pixel, we have

$$b - ddx(b) = b - (b - a) = a.$$

So to generically pass a value v horizontally within a quad and get the horizontal neighbor h, we compute

$$h = v - \text{sign}_x * ddx(v),$$

where sign_x denotes the sign of x in the quadrant of the current pixel within a quad. Although we can not access the pixel diagonally across from the current pixel directly, we can determine the horizontal neighbor followed by the vertical neighbor of that value. An example that computes all three neighbors is as follows:

```
//Gather four float4s
void QuadGather2x2(float4 value,
                   out float4 horz,
                   out float4 vert,
                   out float4 diag)
{
    horz = value + ddx(value) * QuadVector.z; //Horizontal
    vert = value + ddy(value) * QuadVector.w; //Vertical
    diag = vert  + ddx(vert)  * QuadVector.z; //Diagonal
}
```

If we need to gather only one or two values instead of a full `float4` vector, we can optimize this calculation down to as little as two MAD instructions and two derivative instructions:

```
//Gather four floats into one float4
float QuadGather2x2 (float value)
{
    float4 r = value;
    r.y  = r.x  + ddx(r.x)  * QuadVector.z; //Horizontal
    r.zw = r.xy + ddy(r.xy) * QuadVector.w; //Vertical /
        Diagonal
    return r;
}
```

In both of these examples we used the variable `QuadVector`. Figure 2.2 illustrates the value of `QuadVector` for each pixel in a quad. Most of the optimizations we perform in this chapter rely on this vector and one other variable called `QuadSelect`. `QuadVector` is used to divide two-dimensional symmetric problems into four parts, while `QuadSelect` is used to choose between two values based on the current pixel's quadrant.

The following code demonstrates one way to calculate `QuadVector` and `QuadSelect` from a pixel's screen coordinates. The negated/flipped values are also useful and are stored in z/w components.

```
void InitQuad(float2 screenCoord)
{
  //This assumes screenCoord contains an integer pixel
      coordinate
  ScreenCoord = screenCoord;
  QuadVector = frac(screenCoord.xy*0.5).xyxy;
  QuadVector = QuadVector*float4(4,4,-4,-4) + float4(-1,-1,1,1)
      ;
  QuadSelect = saturate(QuadVector);
}
```

While it takes a few instructions to initialize communication within a quad, this will allow us to amortize the cost of several costly shading operations. First, however, we will identify a few drawbacks and limitations when using PQA.

2.6 Limitations of PQA

There are a number of limitations to pixel quad amortization that become immediately apparent. First and foremost, pixel quad message passing works only on hardware that uses hybrid forward and backward derivatives as illustrated in Figure 2.2. When half-resolution derivatives are used, the derivative instructions never touch the bottom-right pixel in the quad. There is no way to communicate that pixel's value to the other pixels in the quad in that case, thus hybrid derivative support needs to be detected based on the graphics card. Appendix A

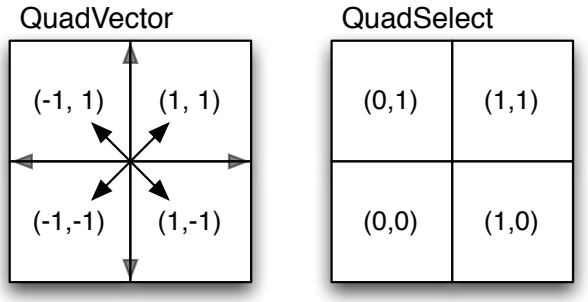

Figure 2.2. To initialize PQA we calculate two simple values for each pixel. `QuadVector` contains the x/y sign of the pixel within it's quad and is used to perform symmetric operations while `QuadSelect` is used to choose between values based on the pixel's location in the quad.

provides a list of hardware that supports hybrid derivatives at the time this article was written. It is also possible that a hardware vendor could change the way the derivative instructions work, breaking this functionality. Although this seems very unlikely, it is easy enough to write a detection routine to test which type of derivatives are used.

The second problem that becomes immediately apparent is that there is no interpolation between quads as there would be from a pre-rendered half-resolution buffer. Thus, if we output the same value for an entire quad, it will resemble unfiltered point sampling from a half-resolution frame buffer. This may be acceptable in certain situations, but if we want higher quality results, we still need to compute unique values for each pixel. Our ability to produce pleasing results really depends on the specific problem.

The third problem is that quad-level calculations work effectively only in the current triangle's domain. For example, we can use pixel quad amortization to accelerate PCF shadow-map sampling in forward rendering, but not nearly as easily in deferred rendering. This is because in the deferred case the quads being rendered are not in object space; thus, a pixel quad may straddle a depth discontinuity, creating a large gap in shadow space. In forward rendering, the entire quad will project into a contiguous location in shadow space, which is what we rely on to amortize costs effectively.

Although there are a number of drawbacks to PQA, we found we could solve these issues for several common graphics problems and still achieve large performance gains. In the following sections we will discuss how to optimize PCF, bilateral upsampling, and basic convolution and blurring with PQA.

2.7 Cross Bilateral Sampling

The cross bilateral filter has been popularized as a means to provide geometry-aware upsampling. If a screen-space buffer is blurred or upsampled using a simple bilinear filter, the features in the low-resolution buffer will bleed across depth boundaries, creating artifacts. The basic idea behind the bilateral filter is to modify the reconstruction kernel to avoid integrating across depth or normal boundaries in the scene. This is achieved by storing a depth and/or normal for each low-resolution sample and assigning filter weight according to not only the distance in screen space to each sample, but also distance in depth and/or normal space. Bilateral filters usually use Gaussian weighting functions in both depth and screen space, however [Yang et al. 08] proposed to use a simple tent function in screen space, mimicking the effect of a bilinear upsample and therefore requiring only four depth/image samples. No matter what type of weighting function is used, the filter weight is accumulated such that the sample can be normalized by the total accumulated weight:

$$c_i^H = \frac{\Sigma c_j^L f(\hat{x}_i, x_j) g(|z_i^H - z_j^L|)}{\Sigma f(\hat{x}_i, x_j) g(|z_i^H - z_j^L|)}$$

In this example $f()$ is the normal linear filtering weight while $g()$ is a Gaussian falloff based on the difference in depth between the high-resolution and low-resolution depths. One disadvantage of bilateral upsampling is its cost compared with simple bilinear filtering. While a bilinear upsample requires only one hardware filtered sample, a bilateral upsample will require at minimum four point samples and four depth samples. This cost is incurred at the high resolution, thus it often partially defeats the purpose of performing calculations at a lower resolution in the first place. Obviously, if the calculation costs less than eight samples, it will be less expensive to just compute the value at the high resolution.

The bilateral filter is one example where PQA works without any of the drawbacks mentioned in the previous section. Since bilateral upsampling occurs in screen space, we can set up our low-resolution buffer such that all the pixels in the same high-resolution quad will share the same low-resolution samples. All that is needed then is to share the samples across the quad and let each pixel perform the bilateral filter independently. Here is an example for a 2X upsample of a low-resolution AO texture. To optimize this further to only one sample, the depth can be packed into extra channels of the AO texture.

```
//Gather quad horizontal / vertical / diagonal samples
float2 AO_D, AO_D_H, AO_D_V, AO_D_D;
AO_D.x = tex2D( lowResDepthSampler , coord ).x;
AO_D.y = tex2D( lowResAOSampler    , coord ).x;
QuadGather2x2( AO_D, AO_D_H, AO_D_V, AO_D_D );
```

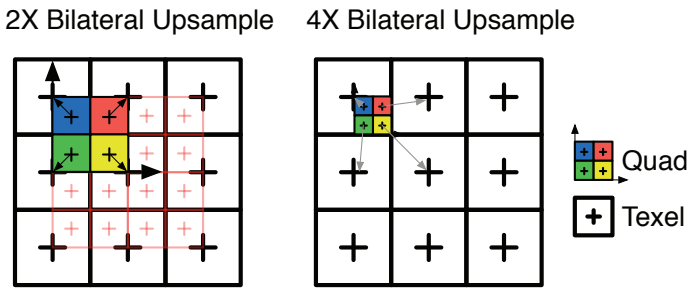

Figure 2.3. Bilateral upsampling from a half-resolution or quarter-resolution buffer. All quad pixels utilize the same four low-resolution samples. We can therefore perform a bilateral upsample with only one or two texture fetches and two derivative instructions, instead of eight texture fetches.

The bilateral upsample can then be performed as usual for each pixel, with the caveat that tent weights will need to flip to compensate for the samples being flipped in each pixel. A similar approach can be taken for a 4X upsample, or for bilateral blurring operations at any resolution. One extra thing to note is that the low-resolution buffer is shifted half a pixel (see Figure 2.3).

2.8 Convolution and Blurring

Convolution and blurring operations can also be accelerated using PQA. Although we are performing calculations at the pixel quad level, we would not want our result to be output at half-resolution or we might as well simply output a truly half-resolution texture! Thankfully, because we can share results at any point in the shader, we can customize the message delivered to other pixels in the quad in order to perform unique blurs for each pixel. The following code illustrates a 3×3 blur with four samples, while Figure 2.4 illustrates this process for a 5×5 blur using nine samples:

```
//Populate messages for neighbors
float4 m = 0;
m.rgba+= tex2D(imageSampler, coord ).x;
m.rb  += tex2D(imageSampler,coord+QuadVector*
            float2(TEXEL_SIZE.x,0)).x;
m.rg  += tex2D(imageSampler,coord+QuadVector*
            float2(TEXEL_SIZE.y,0)).x;
m.r   += tex2D(imageSampler,coord+QuadVector*
            float2(TEXEL_SIZE.xy )).x;
```

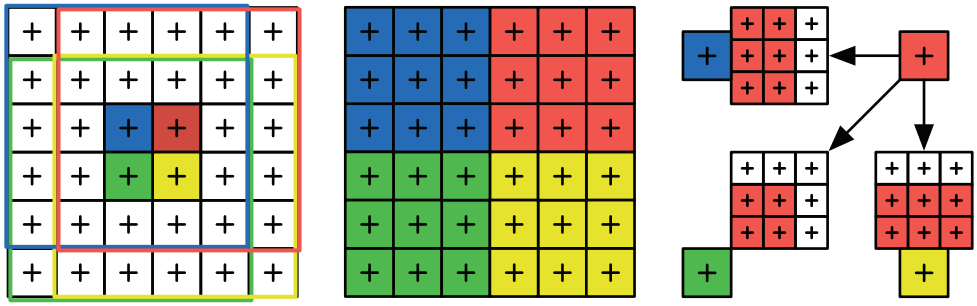

Figure 2.4. Illustration of a 5×5 blur using PQA. The blur kernel footprint of four pixels in a quad (left). Samples taken by each pixel in the quad (middle). Uniquely weighted messages from the red pixel to other pixels in the quad (right).

```
//Gather messages
float4 h, v, d;
QuadGather2x2( m, h, v, d );

//Weight results for 3x3 blur
float4 result = dot(float4(4,2,2,1) / 9.0 ,
                    float4(m.x,h.g,v.b,d.w) );
```

Unfortunately, though we can gather more samples, it becomes cumbersome to apply unique weights for more complicated filters, especially when bilinear filtering is also applied to increase the kernel width. In our example it would also take several `QuadGather` operations for a multiple channel texture. While this can be optimized significantly by separating vertical and horizontal messages, we recommend this approach primarily for performing nonseparable and/or nonlinear blurring operations on one or two channel data. In the case that only approximate results are required, we discuss a gradient approximation to support bilinear filtering in Section 2.9.

In the case of Direct3D 11 hardware, it should be noted that PQA should not be used for simple image blurring. In this case DirectCompute or OpenCL can achieve much better performance by applying the same idea in a compute shader. For example, one could output in quad-sized groups of pixels, or even output an entire row of quads in one shader. For this reason PQA should be used only during geometry rasterization on hardware that supports compute shaders. PQA will remain a valid technique in these cases since rasterization is only a semi-parallelizable task.

2.9 Percentage Closer Filtering

Percentage closer filtering refers to filtering in which a nonlinear operation is required before the filter can take place. In graphics, PCF usually refers to shadow-map filtering, where the nonlinear operation is a depth comparison. A naive N × N PCF filter looks something like this:

```
for( int i = 0; i < N; i++ )
for( int j = 0; j < N; j++ )
{
    shad += ShadowSample(Map, Coord,
                SM_TEXEL * float2( i-(N/2.0-0.5),j-(N/2.0-0.5)
                    ) );
}
shad /= (N*N);
```

Many graphics cards support native bilinear PCF filtering, and this section assumes we have at least bilinear PCF support. Some more recent graphics cards support fetching four depth values at once, allowing the user to arbitrarily filter them in the shader. Since utilizing bilinear PCF is more difficult in our case, but is supported on a much wider set of hardware, we will focus on using bilinear PCF. Extensions to `Gather` instructions can further improve results.

Since we cannot access the result of each pixel when using bilinear PCF, we start by applying an approach from [Sigg and Hadwiger 05, Gruen 10], which uses bilinear samples to build efficient larger filters. This involves using sample offsets such that each bilinear sample fetches four uniquely weighted samples. In the most simple case, where we want equal weights, this simply means placing a bilinear PCF sample in the middle of the four texels we want:

```
//Fraction of a pixel
float2 a = frac(Coord.xy * SM_SIZE - 0.5 );

//Negative/Positive offsets to compute equal weights
float4 Offset = a.xyxy * -(SM_TEXEL) +
                    float4(-0.5,-0.5,1.5,1.5)*SM_TEXEL;
float4 taps;
     taps.x = ShadowSample( Map, Coord, Offset.xw );
     taps.y = ShadowSample( Map, Coord, Offset.zw );
     taps.z = ShadowSample( Map, Coord, Offset.xy );
     taps.w = ShadowSample( Map, Coord, Offset.zy );

float shadow = dot(taps,0.25);
```

This approach can apply to arbitrary separable filters as we will see later, but for now we will keep things simple. To apply PQA, we replace the offset calculation with one that uses the quadrant vector, and then take one sample at each pixel, followed by a quad average:

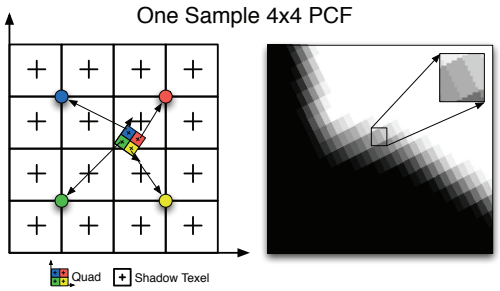

Figure 2.5. Half-resolution 4×4 PCF using quad LOD. The colored pixels correspond to the projection of one pixel quad into shadow space. Each pixel performs only one texture fetch, followed by a pixel quad average. The close-up illustrates half-resolution point sampling artifacts.

```
//Average coordinate for quad
Coord.xy = QuadAve2x2(Coord.xy);

//Fraction of a pixel
float2 a = frac(Coord.xy * SM_SIZE - 0.5 );

//Negative or positive offset to compute equal weights
float2 Offset = (-a + 0.5 + QuadVector.xy) * SM_TEXEL;
float tap = ShadowSample( Map, Coord, Offset );
float shadow = QuadAve2x2(tap);
```

We first compute the average texture coordinate for the quad. We then use the quadrant vector to select only the offset we need. Last, we take one sample in each pixel and then average the results. This is illustrated in Figure 2.5. Note that the offset calculation was also reduced from a `float4` to `float2` calculation.

At this point we are doing a lot of extra work to save only three samples, but once we extend this to larger kernels it starts to become quite effective. For example, if we use four samples per pixel we can now achieve 8×8 PCF (64 total texels) with only four bilinear samples, for a 16X improvement over the naive approach. The layout of these samples is illustrated in Figure 2.6 (right).

```
//Low and high offsets for this pixel
float4 lOhO = (-a.xyxy + QuadVector.xyxy + 0.5 +
                         float4(-2,-2,2,2) ) * SM_TEXEL;

float4 t;
       t.x = ShadowSample( Map, Coord, lOhO.xy );
       t.y = ShadowSample( Map, Coord, lOhO.xw );
       t.z = ShadowSample( Map, Coord, lOhO.zy );
       t.w = ShadowSample( Map, Coord, lOhO.zw );

float shadow = PixelAve2x2(dot(t,0.25));
```

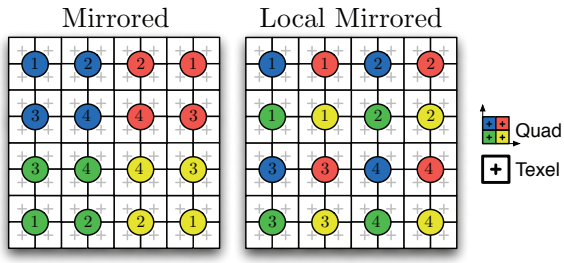

Figure 2.6. Different sample placements color coded by quad pixel. Simply mirroring samples is easier but results in samples that are very far apart, which can degrade cache performance (left). Local mirroring results in samples that are closer together but offsets can be more difficult to calculate symmetrically (right).

Although we can now sample very large kernels, we are outputting the same value for each pixel in the quad, resulting in quad-sized point sampling artifacts. Noncontinuous PCF is also quite undesirable, so it is important to add at least first-order continuity to our filter. We will now tackle both of these issues.

Higher-order filtering is more complicated since shadow texels are not located at fixed distances from the sampling location, thus weights need to be calculated dynamically. The most recent approach [Gruen 10] to achieving higher-order PCF filtering involves solving a small linear system for each sample to find the correct weights and offsets. The linear system is based on all the bilinear samples that would have touched the same texels.

We note that this can be largely simplified by using the work from [Sigg and Hadwiger 05]. Instead of replicating the weights produced by several bilinear samples and a grid of weights, we determine the weight for each texel using an analytic filter kernel. Because the kernel is separable, we can compute the sample offsets and weights separately for each axis. This is demonstrated using a full-sampled Gaussian kernel below.

```
{
    #define SIGMA (SM_TEXEL*2)
    #define ONE_OVER_TWO_SIGMA_SQ (1.0/(2.0*SIGMA*SIGMA))
    #define GAUSSIAN(v) (exp(-(v*v)*ONE_OVER_TWO_SIGMA_SQ))

    float4 GaussianFilterWeight(float4 offset)
    {
    return GAUSSIAN(offset) - GAUSSIAN(4*SM_TEXEL);
    }

    float4 linstep(float4 min, float4 max, float4 v)
    {
    return saturate((v-min)/(max-min));
    }
```

```
float4 LinearStepFilterWeight(float4 offset)
{
return linstep(SM_TEXEL*4.0,SM_TEXEL*2.0,abs(offset));
}

float4 SmoothStepFilterWeight(float4 offset)
{
return smoothstep (SM_TEXEL*4.0,SM_TEXEL*1.0,abs(offset));
}

float4 FilterWeight(float4 offset, const int filterType)
{
switch(filterType)
    {
    case 0:
    return LinearStepFilterWeight(offset,texelWidth);
    case 1:
    return SmoothStepFilterWeight(offset,texelWidth);
    case 2:
    return GaussianFilterWeight(offset,texelWidth);
    }
}

float Shadow8x8Hlaf(Texture2D Map, float4 Coord, const int
    filterType)
{
    //Compute average coord, and fraction of pixel
    Coord = QuadAve(Coord);
    float2 a = frac(Coord.xy*SM_SIZE - 0.5);

    //Low and high pixel center offsets (local mirrored)
    float4 offsets0 = (-a.xyxy + QuadVector.xyxy + float4
        (-2,-2,2,2))*SM_TEXEL;
    float4 offsets1 = offsets0 + SM_TEXEL;

    //Filter weights and offsets
    float4 g0 = FilterWeight(offsets0,filterType);
    float4 g1 = FilterWeight(offsets1,filterType);
    float4 g01 = g0 = g1;
    float4 bilinearOffsets = offsets0 + (g1/g01)*SM_TEXEL;

    //Gather 64 shadow map texels with 4 samples
    float4 taps;
    taps.x = ShadowSample(Map, Coord, bilinearOffsets.xy);
    taps.y = ShadowSample(Map, Coord, bilinearOffsets.zy);
    taps.z = ShadowSample(Map, Coord, bilinearOffsets.xw);
    taps.w = ShadowSample(Map, Coord, bilinearOffsets.zw);
    float4 weights = g01.xzxz*g01.yyww;

    //Sum weights and samples across the quad.
    float4 shadow_weight;
    shadow_weight.x = dot(taps,weights);
    shadow_weight.y = dot(1,weights);
```

```
            shadow_weight.xy = QuadAve(shadow_weight.xy);

            //Normalize our sample weight
            float shadow = shadow_weight.x/shadow_weight.y;
            return shadow;
        }
    }
```

We have shown a few Gaussian filters for both simplicity and readability; in practice, we prefer to use linear, quadratic, or cubic B-spline kernels. Note that we do not need to calculate weights for each texel, but rather for each row and column of texels. Bilinear offsets can then similarly be computed separately and weights simplified to the product between the sum of X and Y weights. The same approach can be applied for piecewise polynomial filters such as B-Splines, or using arbitrary filters with the offsets and weights stored in lookup textures as in [Sigg and Hadwiger 05].

At this point we now have very smooth shadows but still have the same value for all pixels in a quad. To smooth the point-sampled look, it would be optimal to bound all quad texels in shadow space and create a uniquely weighted kernel for each pixel, but without `Gather()` capability that would involve performing four

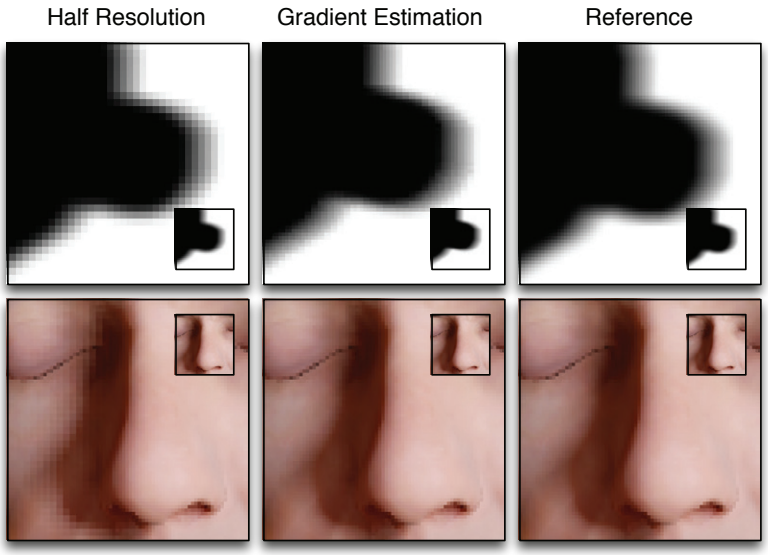

Figure 2.7. Gradient estimation for 8×8 bilinear PCF (four samples). These images are magnified to illustrate how even very naive gradient estimation can hide most quad artifacts. If using Shader Model 4 or 5, `Gather` samples can be used to avoid these artifacts altogether.

times the samples. We found that a good compromise when using bilinear PCF is to compute a simple gradient approximation along with the shadow value. Rather than using every texel to compute the gradient, we simply reuse the bilinear-filtered samples as if they had come from a lower-resolution shadow map. Although this is somewhat of a hack, thankfully it actually works quite well (see Figure 2.7). The weights for the derivative calculation will depend on the kernel itself (see Figure 2.8). The following code calculates a 4×4 Prewitt gradient which works well for low-order B-spline filters:

```
//Gradient estimation using Prewitt 4x4 gradient operator
float4 s_dxdy;
s_dxdy.xy = dot( taps, weights );
//Prewitt (x)
s_dxdy.z  = dot( taps, float4(3,3,1,1) * QuadVector.x );
//Prewitt (y
s_dxdy.w  = dot( taps, float4(3,1,3,1) * QuadVector.y );)
s_dxdy = QuadAve( s_dxdy ) * 4;
float shadow = s_dxdy.x;
```

Figure 2.8. All of the images in Part II, Chapter 1 also make use of PQA for PCF shadows. This image uses 8×8 bilinear PCF filtering (four samples) and half the original ALU operations. Part of the shadow penumbra is used to mimic scattering fall-off, thus an inexpensive wide PCF kernel is crucial. Mesh and textures courtesy of XYZRGB.

```
shadow += dot((Coord.xy-realCoord), s_dxdy.zw );
return shadow;
```

The last issue that needs to be mentioned is handling anisotropy and minification. Our gradient estimate works well on close-ups and will handle minification up to the size of the kernel used. However, under extreme minification the distance between the quad pixels in shadow space increases, and the linear gradient estimate breaks down. There are a number of ways we can deal with this. Firstly, if using a technique like cascading shadow maps (CSMs) we are unlikely to experience extreme minification since shadow resolution should be distributed somewhat equally in screen space. In other cases, one option is to generate mipmaps of the shadow map, allowing us to increase the kernel size to fit the footprint of the quad in shadow space. Alternatively, we can also forgo generating mipmaps and just sparsely sample a larger footprint in the shadow map. We have found that both of these solutions work adequately. Again, having `Gather()` support opens up several more options.

2.10 Discussion

We have demonstrated a new approach for optimizing shaders, by amortizing costly operations across pixel quads, that is natively supported by a large set of existing hardware. Our approach has the advantage of not requiring additional passes over the scene unlike other frame buffer LOD approaches. It also potentially allows for sharing redundant calculations and temporary registers between pixels, while still performing the final calculation at full resolution. We have also demonstrated how gradients can be used to generate smooth results within a quad while still supporting bilinear texture fetches. The primary drawback of our approach remains the lack of interpolation between neighboring quads that would be provided with something like bilateral upsampling. Interestingly, however, our technique can help in either case, since our technique can also accelerate the bilateral upsampling operation itself.

Should PQA become a popular technique, hardware or software pipelines could make it much more efficient by exposing the registers of neighboring pixels directly in the pixel shader. Native API support for sharing registers between pixels would greatly simplify writing amortized shaders. The current cost of sharing results via derivative instructions makes it prohibitive in some cases.

We have found that our approach can also be applied to other rendering problems, such as shadow-contact hardening, ambient occlusion, and global illumination. Although we can not verify this at the time of writing, it also appears that all future hardware that supports Direct3D 11's fine derivatives will support PQA.

2.11 Appendix A: Hardware Support

At the time of writing we have verified that PQA works on all recent NVIDIA hardware. We have tested several 8000 and 9000 series cards including mobile cards in laptops. Unfortunately all the ATI hardware we have tested so far, including the Xbox 360, do not support PQA as they use half-resolution derivatives. The PlayStation 3 console, on the other hand, does support PQA since it uses an NVIDIA GPU, making PQA feasible for current console games. Since Direct3D 11 specifies two types of derivatives, PQA will likely be supported by all hardware that supports Shader Model 5. At the time of writing we do not have access to any Intel graphics cards and thus we do not know which form of derivative Intel graphics cards use.

To detect if message passing works on an arbitrary card, we draw a small `rect` with a custom shader and look at (or read back) the results. The custom shader sets a variable to four in only one quad pixel and zero otherwise. The result of calling `QuadAve()` on that variable will be one for all pixels if message passing worked and something else otherwise. This test is repeated for all quad pixels.

Bibliography

[Gruen 10] Holger Gruen. "Fast Conventional Shadow Filtering." In *GPU Pro: Advanced Rendering Techniques*, pp. 415–445. Natick, MA: A K Peters, 2010.

[Montrym et al. 97] J.S. Montrym, D.R. Baum, D.L. Dignam, and C.J. Migdal. "InfiniteReality: A Real-Time Graphics System." In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, *SIGGRAPH '97*, pp. 293–302. New York: ACM Press/Addison-Wesley Publishing Co., 1997.

[Nehab et al. 07] Diego Nehab, Pedro V. Sander, Jason Lawrence, Natalya Tatarchuk, and John R. Isidoro. "Accelerating Real-Time Shading with Reverse Reprojection Caching." In *ACM Siggraph/Eurographics Symposium on Graphics Hardware*, pp. 25–35. Aire-la-Ville, Switzerland: Eurographics Association, 2007.

[Olano et al. 03] Marc Olano, Bob Kuehne, and Maryann Simmons. "Automatic shader level of detail." In *ACM Siggraph/Eurographics Conference on Graphics Hardware*, pp. 7–14. Aire-la-Ville, Switzerland: Eurographics Association, 2003.

[Ren et al. 06] Zhong Ren, Rui Wang, John Snyder, Kun Zhou, Xinguo Liu, Bo Sun, Peter-Pike Sloan, Hujun Bao, Qunsheng Peng, and Baining Guo. "Real-Time Soft Shadows in Dynamic Scenes using Spherical Harmonic Exponentiation." *ACM Siggraph Transactions on Graphics* 25:3 (2006), 977–986.

[Sigg and Hadwiger 05] Christian Sigg and Markus Hadwiger. "Fast Third-Order Filtering." In *GPU Gems 2*, Chapter 20. Reading, MA: Addison-Wesley Professional, 2005.

[Tomasi and Manduchi 98] C. Tomasi and R. Manduchi. "Bilateral Filtering for Gray and Color Images." In *Proceedings of the Sixth International Conference on Computer Vision*, *ICCV '98*, pp. 839–. Washington, DC: IEEE Computer Society, 1998.

[Yang et al. 08] Lei Yang, Pedro V. Sander, and Jason Lawrence. "Geometry-Aware Framebuffer Level of Detail." 27:4 (2008), 1183–188.

[Zhu et al. 05] T. Zhu, R. Wang, and D. Luebke. "A GPU Accelerated Render Cache." In *Pacific Graphics*, 2005.