Efficient Post-processing with Importance Sampling

Balázs Tóth, László Szirmay-Kalos, and Tamás Umenhoffer

Introduction

Texture filtering is a critical part in many rendering and post-processing methods. If we do it naively, the fragment shader needs to access the texture memory many times to fetch values in the neighborhood of the processed texel. This article presents an efficient filtering algorithm that minimizes the number of texture fetches. The algorithm is based on *importance sampling* and also exploits the bi-linear filtering hardware. We also present applications in one, two, and even in three dimensions, such as *tone mapping with glow*, *depth of field*, and *real-time local ambient occlusion*.

Problem Statement

Many rendering and post-processing algorithms are equivalent to the evaluation of spatial integrals. The general form of a two-dimensional image filter is:

$$L'(X,Y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} L(X-x,Y-y)w(X,Y,x,y)dxdy\,,$$

where $L'(X,Y)$ is the filtered value at pixel X,Y, $L(X,Y)$ is the original image, and $w(X,Y,x,y)$ is the filter kernel for this pixel. If the same filter kernel is used for all pixels, i.e. when kernel w is independent of pixel coordinates X,Y, then the filter is called *spatial-invariant*. For example, the spatial invariant two-dimensional *Gaussian filter* of variance σ^2 has the following kernel:

$$w(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}.$$

The integrals of the filter are usually approximated by finite sums:

$$L'(X,Y) \approx \sum_{i=-N/2}^{N/2}\sum_{j=-N/2}^{N/2} L(X-i,Y-j)w(i,j)\,.$$

This discrete integral approximation requires the evaluation of N^2 kernel values, multiplications, and additions, which is rather costly when repeated for every pixel of the screen.

The computation cost can be reduced for spatial-invariant *separable* filters. In case of separable filters the two-variate filter kernel can be expressed in a product form:

$$w(x,y) = w_x(x)\cdot w_y(y)\,.$$

For spatial-invariant separable filters, the double integral can be computed in two passes. The first pass results in the following 2D function:

$$L_h(X,Y) \approx \sum_{i=-N/2}^{N/2} L(X-i,Y)w_x(i)\,.$$

Then the resulting image is filtered again with a similar one-dimensional filter, but in the vertical direction:

$$L'(X,Y) \approx \sum_{i=-N/2}^{N/2} L_h(X,Y-i)w_y(i).$$

With this trick the filtering cost of a pixel can be reduced from N^2 to $2N$ kernel evaluations and multiplications, which may still be too high in interactive applications.

The Approach of Importance Sampling

In order to reduce the number of samples, instead of sampling the integration domain regularly, importance sampling takes more samples where the filter kernel is large. Let us consider the

$$L'(X) = \int_{-\infty}^{\infty} L(X-x)w(x)dx$$

one-dimensional convolution, and find integral $\tau(x)$ of the kernel and also its inverse $x(\tau)$ so that the following conditions hold

$$\frac{d\tau}{dx} = w(x), \quad \text{i.e.} \quad \tau(x) = \int_{-\infty}^{x} w(t)dt.$$

If kernel $w(t)$ is a probability density, i.e. it is non-negative and integrates to 1, then $\tau(x)$ is non-decreasing, $\tau(-\infty)=0$, and $\tau(\infty)=1$. In fact, $\tau(x)$ is the *cumulative distribution function* of the probability density.

If filter kernel w is known, then $x(\tau)$ can be computed and inverted off-line for sufficient number of uniformly distributed sample points. Substituting the $x(\tau)$ function into the filtering integral we obtain

$$L'(X) = \int_{-\infty}^{\infty} L(X-x)w(x)dx = \int_{-\infty}^{\infty} L(X-x)\frac{d\tau}{dx}dx = \int_{\tau(-\infty)}^{\tau(\infty)} L(X-x(\tau))d\tau = \int_{0}^{1} L(X-x(\tau))d\tau$$

Approximating the transformed integral taking uniformly distributed samples in τ corresponds to a quadrature of the original integral taking M non-uniform samples in x. In one-dimension we compute $x(\tau)$ for $\tau = 1/(2M), 3/(2M), \ldots, (2M-1)/(2M)$.

$$L'(X) = \int_{0}^{1} L(X-x(\tau))d\tau \approx \frac{1}{M}\sum_{i=1}^{M} L\left(X - x\left(\frac{2i-1}{2M}\right)\right).$$

This way we take samples densely where the filter kernel is large and fetch samples less often farther away, but do not apply weighting (Figure 1).

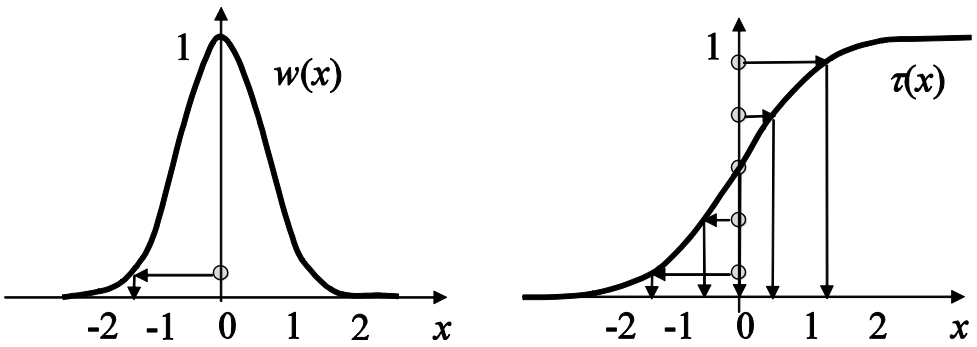


Figure 1. The kernel of the Gaussian filter of unit variance (left) and its cumulative distribution function $\tau(x) = \Phi(x)$. Sampling according to the inverse of this function is equivalent to taking uniform samples on the vertical abscissa and mapping it to the horizontal one.

The non-uniform sample positions are not necessarily on the grid, but may also be in between the texel centers. Such samples can be obtained assuming the original function to be piece-wise linear and exploiting the texture filtering hardware to provide us with these interpolated values at no additional cost [Sigg05].

Note that non-uniform sampling allows us to use a smaller number of samples than uniform sampling ($M<N$) while providing the same accuracy. Non-uniform sampling does not access every texel in the neighborhood since far from the center of the filter kernel the weighting would eliminate the contribution anyway, so taking dense samples far from the center would be waste of time.

The implementation of this approach is quite straightforward. Function $x(\tau)$ is computed by integrating the filter kernel and inverting the integral. For the Gaussian filter, $\tau(x)$ is the cumulative probability distribution function of the normal distribution, i.e. the famous Φ-function, $\tau(x) = \Phi(x/\sigma)$ [4]. Values of its inverse can be hardwired into the shader. For example, if M=5, we need the following pre-computed values:

$x(1/10)= -1.282\sigma$, $x(3/10)=-0.524\sigma$, $x(5/10)=0$, $x(7/10)= 0.524\sigma$, $x(9/10)= 1.282\sigma$.

The fragment shader implementing this idea along one dimension is shown below. The shader gets the texture coordinates of the current fragment in `Tex`, and filters the input image stored in texture `InputImage`. The horizontal resolution of the image is `HRES`.

```
float4 FilterImportancePS(in float2 Tex : TEXCOORD0) : COLOR {
    float2 du1 = float2(0.524/HRES * sigma, 0);
    float2 du2 = float2(1.282/HRES * sigma, 0);
    float3 filtered = tex2D(InputImage, Tex - du2) +
                      tex2D(InputImage, Tex - du1) +
                      tex2D(InputImage, Tex) +
                      tex2D(InputImage, Tex + du1) +
                      tex2D(InputImage, Tex + du2);
    return float4(filtered/5, 1);
}
```

In one dimension importance sampling takes optimally uniform series 1/(2*M*), 3/(2*M*), …, (2*M*-1)/(2*M*) in the unit interval, and transforms its elements with the inverse of the cumulative distribution (τ^{-1}) to obtain the non-uniform set of offsets for sampling. In two or higher dimensions the same concept is applied, but, unfortunately, we do not have the optimally uniform distribution of points in a unit square or in higher dimensional unit cubes. Regular grids, which repeat the same one-dimensional series independently along the coordinate axes, get very non-uniform in higher dimensions (we have large gaps between the rows and columns). Better options are the *low-discrepancy series*, such as the *Halton* or *Hammersley series*, or we can also produce our own uniform sample set with an iterative relaxation algorithm. An initial set of points are put into an (arbitrary dimensional) cube, and we assume that points repel each other. Moving the points in the direction of the resulting force and repeating this step iteratively, a uniform distribution, the Poisson disc distribution can be obtained.

In the following sections we present three applications for the discussed importance based filtering scheme. Tone mapping and glow (also called bloom) require spatial-invariant Gaussian filtering, which can be executed as a pass of one-variate horizontal, then vertical filtering. Then depth of field is attacked, where the filter size is not spatial-invariant. Thus, the two dimensions cannot be simply separated, but we apply the two-dimensional version of the discussed importance sampling scheme. Finally, a three-dimensional example is taken, the low-variance computation of screen space ambient occlusion.

Tone Mapping with Glow

Off the shelf monitors can produce light intensity just in a limited, *low dynamic range* (LDR). Therefore the values written into the frame buffer are unsigned bytes in the range of [0x00, 0xff], representing values in [0,1], where 1 corresponds to the maximum intensity of the monitor. However, realistic rendering often results in *high dynamic range* (HDR) luminance values that are not restricted to the range of the monitors. The mapping of HDR image values to displayable LDR values is called *tone mapping* [Reinhard06]. The conversion is based on the luminance the human eye is adapted to. Assuming that our view spans over the image, the adaptation luminance will be the average luminance of the whole image.

The main steps of tone mapping are as follows. The luminance value of every pixel is obtained with the standard CIE XYZ transform:
$Y = 0.2126\ R + 0.7152 G + 0.0722\ B$,
and these values are averaged to get adaptation luminance Y'. Since the human vision is sensitive to relative differences rather than to absolute ones, the geometric average is computed instead of the arithmetic mean. The geometric average can be calculated by obtaining the logarithm of the pixel luminances, generating the arithmetic mean of these logarithms, and finally applying exponentiation. Having adaptation luminance Y', pixel luminance values Y are first mapped to relative luminance Y_r:

$$Y_r = \frac{\alpha Y}{Y'},$$

where α is a user defined constant of the mapping, which is called the *key*. The relative luminance values are then mapped to displayable pixel intensities using the following function:

$$D = \frac{Y_r(1 + Y_r / {Y_w}^2)}{1 + Y_r},$$

where Y_w is another user defined value representing the relative luminance that is expected to be mapped onto the maximum monitor intensity. Colors of relative luminance higher than Y_w will burn out.

Having the display luminance, the original [*R,G,B*] data is scaled with it to provide color information [*r,g,b*]:

$$[r, g, b] = [R, G, B]\frac{D}{Y}.$$

The user defined key value controls where the average luminance is mapped in the [0,1] region. For α=0.05, α=0.2, and α=0.8, the average luminance is shown at *D*=0.048, *D*=0.17, and *D*=0.44, respectively.

Glow or *bloom* occurs when a very bright object causes the neighboring pixels to be brighter than they would be normally. It is caused by scattering in the lens and other parts of the eye, giving a glow around the light and dimming contrast elsewhere. To produce glow, first we distinguish pixels where glowing parts are seen from the rest by selecting pixels where the luminance is significantly higher than the average. The HDR color of glowing parts is distributed to the pixels nearby, which is a Gaussian blur, which results in a *glow map*. The glow map is added to the HDR image before tone mapping.

Implementation of Glow

Note that in this process we use Gaussian filtering during the computation of the glow map. The variance is constant, thus this is a spatial-invariant separable filter, that can be realized by two 1-dimensional filtering steps, which can exploit the proposed importance sampling scheme, which fetches samples non-uniformly, but with a distribution specified by the cumulative distribution of the Gaussian.

Figure 2. The Moria scene with tone mapping and glow.

Depth of Field

Computer graphics algorithms usually apply the pinhole camera model, while real cameras have lenses of finite dimensions, which let through rays coming from different directions. As a result, parts of the scene are sharp only if they are located at a specific focal distance.

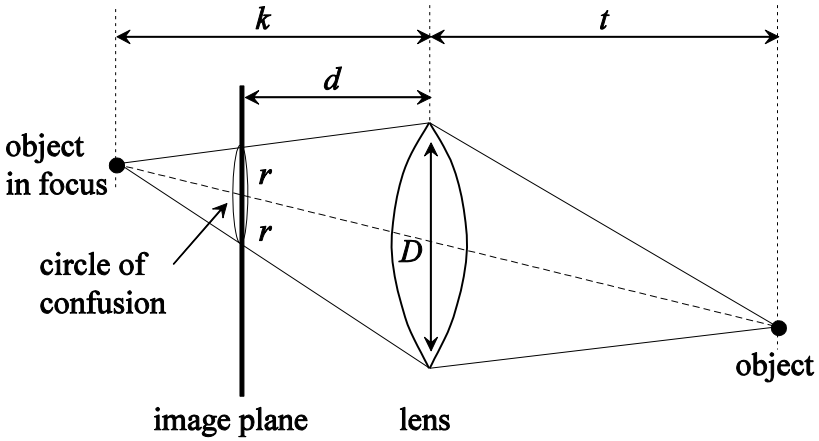


Figure 3. The computation of the circle of confusion.

According to geometric optics (see Figure 3), if the *focal length* of a lens is *f* and an object point is at distance *t* from the lens, then the corresponding image point will be in sharp focus on an image plane at distance *k* behind the lens, where *f*, *t*, and *k* satisfy the following equation:

$$\frac{1}{f} = \frac{1}{k} + \frac{1}{t}.$$

If the image plane is not at proper distance k from the lens of diameter D, but at distance d, then the object point is mapped not onto a point but onto a circle of radius r:

$$r = \frac{|k-d|}{k}\frac{D}{2}.$$

This circle is called the *circle of confusion* corresponding to the given object point. It expresses that the color of the object point affects the color of not only a single pixel but all pixels falling into the circle.

A given camera setting can be specified by the *focal distance P*, which is the distance of those objects from the lens, which appear in sharp focus on the image plane (note that the focal distance must not be confused with the focal length). The focal distance and the distance of the image plane also satisfy the basic relation of the geometric optics:

$$\frac{1}{f} = \frac{1}{d} + \frac{1}{P}.$$

Putting the three equations together, we obtain the following formula for the radius of the circle of confusion:

$$r = \left|\frac{1}{t} - \frac{1}{P}\right|\frac{Dd}{2}.$$

According to this formula the radius is proportional to the difference of the reciprocals of the object distance and of the focal distance. Since the projective transform and the homogeneous division translates camera space depth z to screen space depth Z as $Z=a + b/z$ where a and b depend on the front and back clipping space distances (yet another camera parameters), the radius of the circle of confusion is just proportional to the difference of the object's depth coordinate and the focal distance, interpreting them in screen space:

$$r = |Z - P'|S$$

where Z is the screen space depth of the point, P' is the distance of the focal plane transformed to screen space, and $S=Ddb/2$ is the camera's scaling parameter composed from the size of the lens, the distance of the image plane, and the front/back clipping plane distances.

This theory of depth of field describes the phenomena from the point of view of object points projected onto the view plane. The power reflected off by a point is distributed in a circle, usually non-uniformly, producing higher power density in the center. The affected region grows with the area of the circle, but the contribution to a particular point decreases proportionally. However, in rendering we take an opposite approach and should consider the phenomenon from the point of view of the pixels of the image. If we can assume that the depth is similar in the neighborhood to the depth of the current fragment, the colors of the neighborhood pixels should be blended with

the current color, using a filter that decreases with the distance to a negligible value when the distance is greater than the radius of the circle of confusion. A good candidate for such filters is the Gaussian filter setting its standard deviation σ to the radius of the circle of confusion.

Depth of Field Implementation

Depth of field simulation consists of two phases. In the first pass, the scene is rendered into textures of color and depth values. In the second pass, the final image is computed from the prepared textures with blurring. Blurring is performed using a variable-sized Gaussian filter since the variance (i.e. the circle of confusion) changes from pixel to pixel, making the filtering process not spatial-invariant. Note that this prohibits us to replace the 2D filtering by two 1D filtering phases. So we have to implement the method as a real 2D filtering scheme, which makes importance sampling even more indispensable. The depth of field shader should use 2D offset positions stored in a pre-computed array, which is generated by the discussed importance sampling method. We have two options. On the one hand, we can exploit the separability of the 2D Gaussian filter and repeat the method proposed for 1D filtering to independently find the *x* and *y* coordinates of the sample locations.

The other possibility is to replace Cartesian coordinates by polar coordinates *r*, ϕ in the filtering integral:

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} L(X-x,Y-y)\frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}}dxdy = \int_{\phi=0}^{2\pi}\int_{r=0}^{\infty} L(X-r\cos\phi,Y-r\sin\phi)\frac{1}{2\pi\sigma^2}e^{-\frac{r^2}{2\sigma^2}}rdrd\phi$$

This filter can be separated to a filter for angular angles ϕ, where the filter kernel is constant $1/2\pi$ in filtering domain $[0,2\pi]$, and to filter for radius *r*, where the filtering kernel is

$$w(r)=\frac{1}{\sigma^2}e^{-\frac{r^2}{2\sigma^2}}r\,.$$

Since the filtering kernel of polar angle ϕ is constant, such samples are taken uniformly in the angular domain $[0,2\pi]$. Considering distance *r*, the cumulative distribution

$$\tau(r)=\int_0^r \frac{1}{\sigma^2}e^{-\frac{t^2}{2\sigma^2}}tdt = 1-e^{-\frac{r^2}{2\sigma^2}}$$

can be analytically inverted:

$$r=\sigma\sqrt{-2\log(1-\tau)}\,.$$

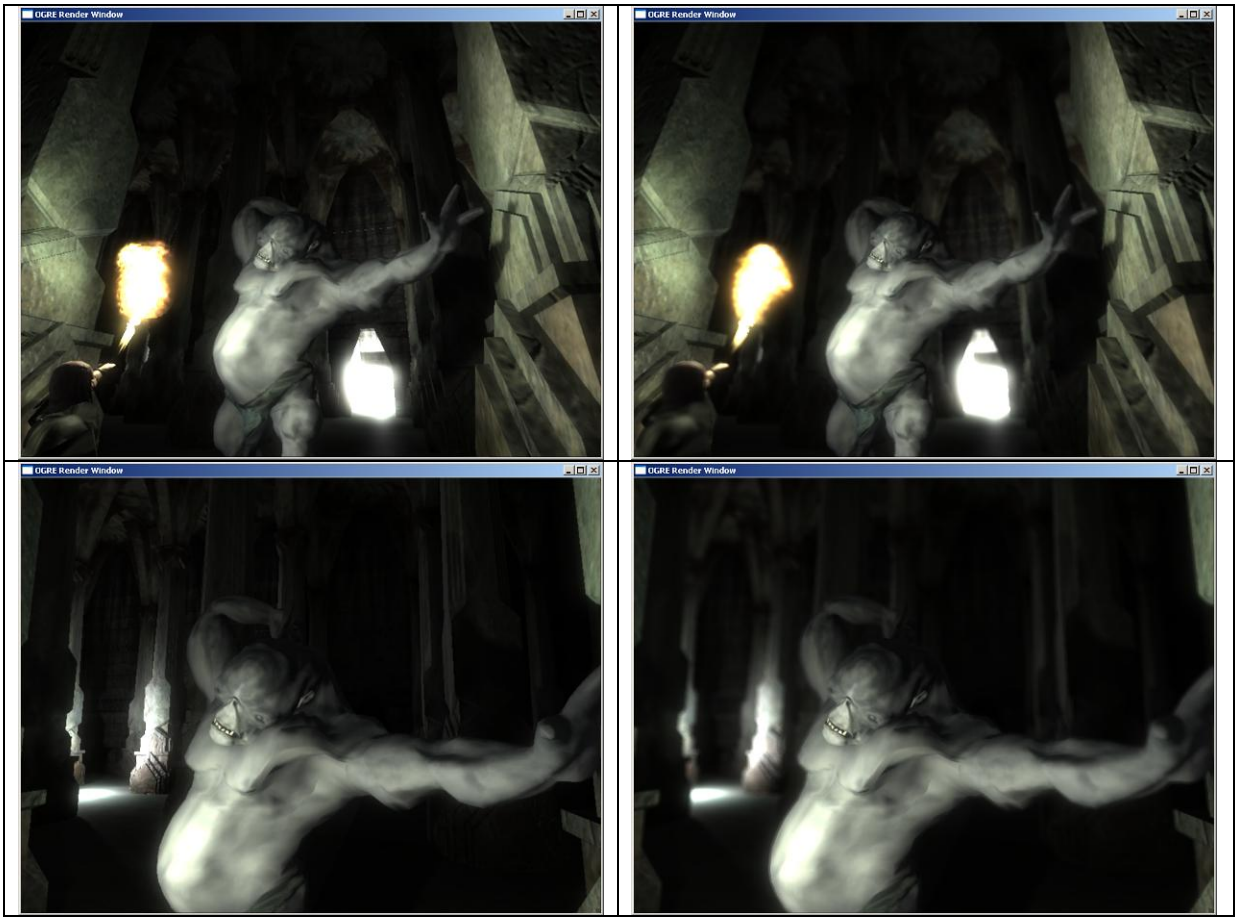
Despite to the analytical expression, it is still worth pre-computing the sample location offsets, and pass them to the shader as a global variable (`filterTaps[i]` in the program below). This array is constructed by taking `NUM_DOF_TAPS` number of uniformly distributed (τ, υ) samples in the unit square, transforming them as ($\sqrt{-2\log(1-\tau)}$, $2\pi\upsilon$) to (r, ϕ) polar coordinate pairs, and finally obtaining the offset positions in Cartesian coordinates. The offset positions will be multiplied by the

radius of the circle of confusion (σ) in the shader since this parameter varies from fragment to fragment.

The fragment shader gets the location of the current fragment and also its depth value in screen coordinates, computes the circle of confusion radius, and scales the sample offset with this value. The original color image is fetched at the offset positions and the colors are added up without any additional weighting.

```
float4 DepthBlurPS(in float2 Tex :TEXCOORD0): COLOR
{
  float4 colorSum = tex2D(InputImage, Tex);     // Center sample
  float  depth = tex2D(depthMapSampler, Tex).a; // Current depth
  float sizeCoC = abs(depth - FOCAL_DISTANCE) * DOF_SCALE;
  for (int i = 0; i < NUM_DOF_TAPS; i++) // Filter
  {
     float2 tapCoord = Tex + filterTaps[i].xy * sizeCoC;
     colorSum += tex2D(InputImage, tapCoord);
  }
   return colorSum / (NUM_DOF_TAPS + 1); // Normalize
}
```

Note that this implementation assumes that the depth values are similar in the neighborhood as the depth of the current pixel. If this assumption fails, artifacts may show up, which can be reduced by skipping those candidate fragments where the depth difference is too large.



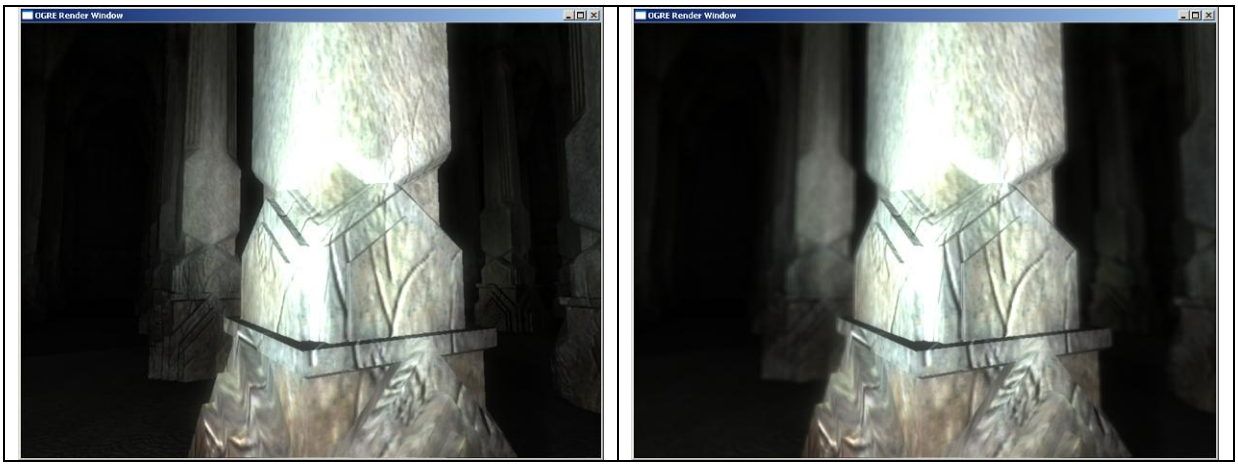


Figure 4. The Moria scene without (left) and with depth of field (right).

Real-time obscurances and ambient occlusion

The *obscurances* [Iones03] model approximates the indirect lighting in point $\vec{x}$ of the scene by

$L^i(\vec{x}) \approx a(\vec{x})W(\vec{x})L^a$,

where a is the albedo, L^a is the ambient light intensity, and W is the obscurances value of the point, which expresses how open this point is for ambient illumination. A point is *open* in a direction if no occluder can be found close by. The obscurances value is defined by

$$W(\vec{x}) = \frac{1}{\pi}\int \mu(d(\vec{\omega}))\cos\theta d\omega,$$

where d is the distance of the occluding surface in direction $\vec{\omega}$ enclosing angle θ with the surface normal, and μ is a fuzzy measure of openness. If d is small, then the point is closed, thus the fuzzy measure is small. If the distance is large, the point gets more open. We consider just a neighborhood of radius R, and assume the point to be totally open if the occluder is farther.

The *ambient occlusion* is a special case of this model, which uses a clear, non-fuzzy distinction of open and closed directions [Hayden02]. If the occluder is closer than R, then the point is closed, otherwise it is opened, which can be modeled by a step-like fuzzy membership function. However, better results can be obtained with smoother membership functions, as proposed by Mendez et al. [Mendez05]. For example, $\mu(d) = \sqrt{d/R}$ works well in practice.

The evaluation of the directional integral of the obscurances formula requires rays to be traced in many directions, which is rather costly. The expensive ray tracing operation can be replaced by a simple containment test if neighborhood R is small enough to allow the assumption that the ray intersects the surface at most once in the R-interval [Iones02]. This imposes restrictions on the surface curvature.

In order to find an efficient method for the evaluation of the obscurances, we express it as a three dimensional integral. First the fuzzy measure is expressed as

$$\mu(d) = \int_0^d \frac{d\mu(r)}{dr} dr = \int_0^R \frac{d\mu(r)}{dr} \varepsilon(d-r) dr,$$

where $\varepsilon(r)$ is the step function, which is 1 if r is not negative and zero otherwise. Substituting this integral into the obscurances formula we get

$$W(\vec{x}) = \int_\Omega \int_0^R \frac{d\mu(r)}{dr} \frac{\cos\theta}{\pi} \varepsilon(d-r) dr d\omega.$$

Let us consider a ray of equation $\vec{x} + \vec{\omega}r$ where shaded point $\vec{x}$ is the origin, $\vec{\omega}$ is the direction, and distance r is the ray parameter. If we assume that the ray intersects the surface at most once in the R-neighborhood, then $\varepsilon(d\text{-}r)$ is equivalent to the condition that the ray has not intersected the surface yet. If it has not intersected the surface, and other objects are far, then this condition is equivalent to the visibility of the sample point $\vec{x} + \vec{\omega}r$, which can be checked using the content of the z-buffer [Mittring07] (Figure 5). Note that this integral is a filtering scheme where we filter the 0, 1 values of visibility indicator ε with the following kernel:

$$w(r, \vec{\omega}) = \frac{d\mu(r)}{dr} \frac{\cos\theta}{\pi}.$$

This filter is the product of a density $d\mu(r)/dr$ of distances and density $\cos\theta/\pi$ of ray directions.

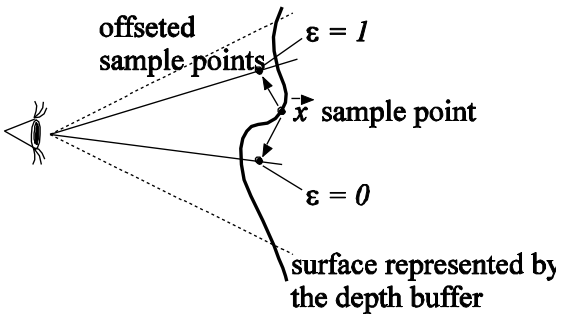


Figure 5. The idea of replacing ray tracing by visibility tests using the content of the depth buffer as a sampled representation of the surface.

Implementation of Real-time Obscurances

To construct the non-uniform locations representing the filter kernel, we start with point set $[\tau, \upsilon, \xi]$ that is uniformly distributed in a 3D unit cube, and transform the uniform distribution to mimic the filtering kernel. As the first step of the transformation, two coordinates $[\tau, \upsilon]$ are transformed to a direction that has cosine distribution. Let us consider a unit radius sphere centered at the considered point. The sphere intersects the tangent plane xy in a circle. If we take points in a unit radius circle uniformly, and then map up the point onto the sphere, we get samples with cosine distribution. In order to generate uniformly distributed points in a unit radius circle, we take values τ, υ and transform them linearly from [0,1] to [-1,1]. The two scalars are considered as x,y coordinates of a 2D point and it is checked whether the point is inside the unit radius circle. If not, this pair is rejected, and another pair is

taken until we get a point that is really in the circle, and finally project it up to the unit sphere.

Having found a cosine distributed direction, distance r is sampled with density $d\mu(r)/dr$. If we take the third coordinate ξ of the uniform 3D point set, $r = \mu^{-1}(\xi)$ will have exactly the required density. Sample points generated this way are stored in a global array `OFFSET`. These points correspond to translations in the tangent space.

During run time, the post-processing filter computes the obscurances from the depth buffer (`depthMap`) storing camera space z values. The shader gets the current point's texture coordinates (`Tex`) and its projection onto the first clipping plane (`Dir`). From the projection and the stored depth, the point is reconstructed in camera space. The shader program transforms the offsets of `OFFSET` to camera space (`cOff`) and translates the camera space point with them obtaining `sampleCPos` in camera space and then by projection transformation `sampleSPos` in screen space. The visibility of the translated point is checked by projecting it onto the screen, and comparing its depth to the depth obtained in the direction of the translated point. If the sample point passes the depth test, then value 1 is added to the filtered variable, otherwise the variable is not changed. Note that to transform the offsets from tangent space to camera space we need the camera space normal vector of the current point. We can store these normals in the first three channel of the depth buffer.

```
float4 SSAOPS(in float2 Tex : TEXCOORD0, // pixel coords in [0,1]
              in float4 Dir : TEXCOORD1, // on front clipping plane
              ) : COLOR
{
   float depth = tex2D(depthMap, Tex).a;  // camera space depth
   Dir.xyz /= Dir.w;
   float3 cPos = Dir.xyz * depth / Dir.z; // camera space location

   float3 T, B, N;                        // Determine tangent space
   N = tex2D(depthMap, Tex).rgb;          // camera space normal
   T = cross(N, float3(0, 1, 0));
   B = cross(N, T);
   float3x3 TangentToView = float3x3(T, B, N);

   float occ = 0;
   for(int k = 0; k < NUM_SAMPLES; k++)
   {
      // Transform offsets from tangent space to camera space
      float3 cOff = mul(OFFSET[k].xyz, TangentToView) * R;
      float3 sampleCPos = cPos + cOff; // Sample pos in camera space

      // Compute screen coordinates
      float4 sampleCPos = mul(float4(sampleCPos, 1), projMatrix);
      float2 sampleSPos = sampleCPos.xy / sampleCPos.w;
      sampleSPos.y *= -1;
      sampleSPos = (sampleSPos + 1.0) * 0.5;

      // Read depth buffer
      float sampleDepth = tex2D(depthMap, sampleSPos).a;

      // Compare sample depth with depth buffer value
      if(sampleDepth >= sampleCPos.z)         occ++;
      else if(sampleCPos.z - sampleDepth > R) occ++;
```

```
    }
    occ = occ / NUM_SAMPLES; // Normalize
    return occ * tex2D(InputImage, Tex); // Compose with shaded image
}
```

Figure 6. The Rocky Scene and the Space Station Scene rendered with classical ambient reflection model $a(\vec{x})L^a$ (top row), obscurances only (middle row), and using the obscurances reflection model $a(\vec{x})W(\vec{x})L^a$. The obscurances values are generated from 16 samples per pixel in real-time.

Conclusion

This article presents a method to improve the efficiency of filtering algorithms and three applications to demonstrate the power of the method. The first application is a separable Gaussian filtering used for tone mapping and glow. The second application is a not spatial-invariant 2D filter producing depth of field effect. The final application filters in 3D and provides real-time ambient occlusion. A real-time demo implemented using DirectX9.

Acknowledgments

This work has been completed in the framework of the GameTools project.

References

[Reinhard06] E. Reinhard and G. Ward and S. Pattanaik and P. Debevec. High Dynamic Range Imaging. Morgan Kaufmann, 2006.

[Mittring07] Martin Mittring. Finding Next Gen - CryEngine 2. In Advanced Real-Time Rendering in 3D Graphics and Games Course - Siggraph 2007. pp 97-121.

[Sigg05] C. Sigg and M. Hadwiger. Fast Third-Order Texture Filtering. In GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Matt Pharr(ed.), Addison-Wesley, 2005. pp 313-329.

[Hayden02] Hayden, L. Production-Ready Global Illumination. SIGGRAPH Course notes 16. 2002.

[Iones03] Iones, A. and Krupkin, A. and Sbert, M. and Zhukov, S. Fast realistic lighting for video games. IEEE Computer Graphics and Applications, Vol. 23, No 3, 2003. pp 54-64.

[Mendez05] A. Mendez and M. Sbert and J. Cata and N. Sunyer and S. Funtane. Real-time Obscurances with Color Bleeding. In ShaderX4: Advanced Rendering Techniques, Wolfgang Engel (ed.) Charles River Media. 2005.