4

II

Hi-Z Screen-Space Cone-Traced Reflections

Yasin Uludag

4.1 Overview

This chapter will introduce a novel approach for calculating reflections on dynamic 3D scenes, one that works on arbitrary shaped surfaces. Algorithms and techniques that were researched during the early development of *Mirror's Edge* are presented and shared.

The methods we will look into outperform any other methods both in terms of performance and image quality, as can be seen in Section 4.8, "Performance," and Section 4.9, "Results."

We will take a look into the latest work done in the area of real-time reflections, analyze their pros and cons and where they fail to deliver. Then we'll look into a new approach for calculating reflections in real time at game interactive frame rates.

First we will present the algorithm itself, which uses a screen-space aligned quad tree we call *Hierarchical-Z* (Hi-Z) buffer to accelerate the ray tracing. The hierarchy is stored in the MIP channels of the Hi-Z texture. This acceleration structure is used for empty space skipping to efficiently arrive at the intersection point. We will further discuss all the pre-computation passes needed for glossy reflections and how they can be constructed. We will also look into a technique called screen-space cone tracing for approximating rough surfaces, which produce blurred reflections. Moreover, performance and optimization for the algorithm are discussed. Then extensions to the algorithm are shown for improving and stabilizing the result. One such extension is temporal filtering, which allows us to accumulate the reflection results of several previous frames to stabilize the output result of the current frame by re-projecting the previous images even when the camera moves. The ability to approximate multiple ray bounces for reflections within reflections comes for free when doing temporal filtering because the previous frames already contain the reflections. Research and development techniques currently being developed will be mentioned, such as packet tracing for grouping several rays together into a packet and then refining/subdividing and shooting smaller ray packets once a coarse intersection is found. Another direction for future research that will be mentioned is screen-space tile-based tracing where if an entire tile contains mostly rough surfaces we know we can shoot fewer rays because the result will most likely be blurred, thereby gaining major performance, which gives us more room for other types of calculations for producing better images.

Finally timers will be shown for PCs. For the PC we will use both NVIDIA- and AMD-based graphics cards. Before we conclude the chapter, we will also mention some future ideas and thoughts that are being currently researched and developed.

This novel and production-proven approach (used in *Mirror's Edge*), proposed in this chapter guarantees maximum quality, stability, and good performance for computing local reflections, especially when it is used in conjunction with the already available methods in the game industry such as local cube-maps [Bjorke 07, Behc 10]. Specific attention is given to calculating physically accurate glossy/rough reflections matching how the stretching and spreading of the reflections behave in real life from different angles, a phenomenon caused by micro fractures.

4.2 Introduction

Let's start with the actual definition of a reflection:

> *Reflection* is the change in direction of a wave, such as a light or sound wave, away from a boundary the wave encounters. Reflected waves remain in their original medium rather than entering the medium they encounter. According to the *law of reflection*, the angle of reflection of a reflected wave is equal to its angle of incidence (Figure 4.1).

Reflections are an essential part of lighting; everything the human eye perceives is a reflection, whether it's a specular (mirror), glossy (rough), or diffusive

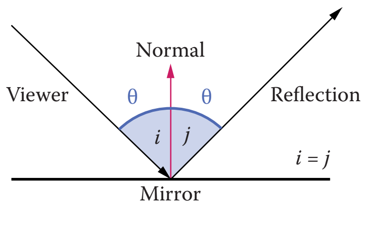


Figure 4.1. Law of reflection states that incident angle i equals reflection angle j.

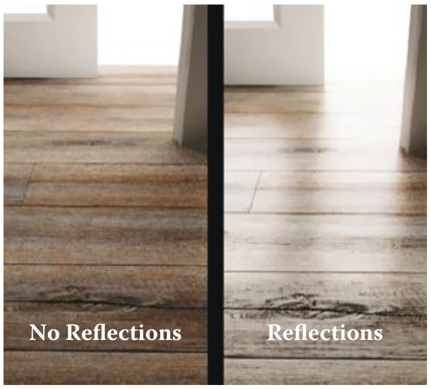


Figure 4.2. Impact of reflections toward the goal of achieving photorealism. Notice the reflection occlusion near the contact points of the floor and the door blocking the incident light.

reflection (matte). It's an important part of achieving realism in materials and lighting. Reflection occlusion also helps out with grounding the object being reflected into the scene at the contact points, as we can see in Figures 4.2 and 4.3. It's an important part of our visual understanding of reality and it shouldn't be taken lightly, as it can make a big difference in achieving realism.

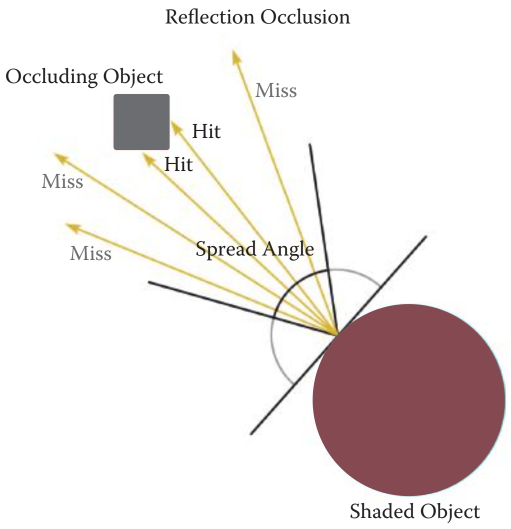


Figure 4.3. Illustration showing reflection occlusion where the hit rays block the light just like the door blocks the white light in Figure 4.2.

There has been little development lately for producing accurate reflections, especially glossy reflections, in the real-time graphics industry at high performance game frame-rate levels, meaning an algorithm has to run at a fraction of our per frame millisecond budget.

Solving reflections in computer games has been a big challenge due to the high performance requirements of the computations. We have a limited budget of milliseconds to spare for each frame, 16.6 milliseconds for 60 FPS and 33.33 milliseconds for 30 FPS, to recalculate everything and present an image to the user. This includes everything from game simulation, physics, graphics, AI to Network, etc. If we don't keep the performance level at such small fractions of a second, the user will not experience feedback in real time when giving input to the game. Now imagine that a fraction of those milliseconds needs to go to reflections only. Coming up with an algorithm that runs as fast as a few milliseconds and still keeps the quality level at maximum is hard using rasterization-based techniques that GPU hardware runs on.

Though game developers have been able to produce fake reflections for a very long time on simple cases, there is no solution that fixes every issue up to an acceptable level of realism with the performance levels required. For planar surfaces, meaning walls and floors, it's easy to flip the camera and re-render the entire scene and project the resulting image onto the planar surface to achieve what we today call planar reflections. This works for planar surfaces such as floors and walls but it's a completely different story for arbitrarily shaped surfaces that can reflect toward any direction per pixel. Re-rendering the entire scene and re-calculating all the lightings per plane is also an expensive operation and can quickly become a bottleneck.

The only solution that gives perfect results existing today is what we call ray tracing. But, tracing reflected rays and mathematically intersecting geometric primitives (a bunch of small triangles that make up the 3D world) is computationally and also memory heavy both in terms of bandwidth and size, because the rays could really go anywhere and we would need to keep the entire 3D scene in memory in a traversable and fast-to-access data structure. Even today, with the most optimized algorithms and data structures, ray tracing is still not fast enough in terms of performance to be deployed on games.

4.3 Previous Work

Generating 100% accurate and efficient reflections is difficult if not impossible with rasterization-based hardware used in GPUs today. Though we have moved on toward more general computing architectures allowing us more freedom, it's still not efficient enough to use a real ray tracer. For this reason game developers have for a long time relied on planar reflections where you re-render the scene from a mirrored camera for each plane, such as floors or walls, and project the image

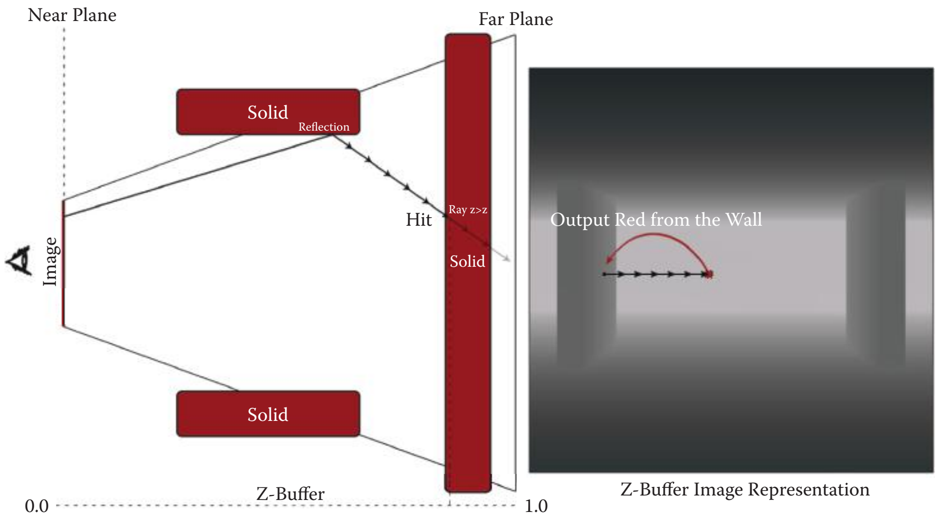


Figure 4.4. Ray marching by taking small steps in screen space using the Z-buffer (depth buffer, an image representing scene depth in floating point numbers) until the ray depth is below the surface. Once it is below the surface, we can stop the ray marching and use the new coordinates to acquire the reflection color and apply it on the pixel we started the marching from.

to create a planar reflection. Another technique that has been relied upon for a very long time is cube-maps, six images capturing 360 degrees of the surrounding environment from a single point with a 90-degree field of view for each side, hence these reflections are only valid from that specific point only.

A new idea called screen-space local reflections was first showed by [Graham 10] at Beyond3D forums and then later introduced into Crysis 2 DX11 patch by Crytek [Tiago et al. 12]. They both proposed a simple ray marching algorithm in screen space. Screen space means that we do everything in 2D framebuffer objects, images, as a post-processing effect.

It's a fairly simple idea; you just compute a screen-space reflection vector using the scene normal buffer and ray-march through the pixels at a certain step size until the ray depth falls below the scene depth stored in what we call a depth buffer. Once the ray depth is below the scene depth, we detect a hit and use the new screen-space coordinate to read the scene color, which is used as reflection for the pixel we started the ray-marching from. This technique is illustrated in Figure 4.4.

However since this computation is performed in screen space, there are limitations that need to be taken care of. A built-in problem with this kind of technique is that not all of the information is available to us. Imagine a mirror reflecting the ray in the opposite direction of the view direction; this information is not available in screen space. This means that occlusion and missing information is a huge challenge for this technique and if we do not deal with this problem we will have artifacts and produce incorrect reflection colors. Smoothly fading rays that fall outside the screen borders, fall behind objects that occlude information and rays that point toward the camera are recommended.

On the other hand this type of linear ray marching can be really efficient if you do a low number of steps/samples for very short range reflections. As soon as you have really long rays this method starts to perform really slowly because of all the texture fetches it requires at each loop to acquire the scene depth from the Z-buffer. Due to this latency hiding starts to diminish and our cores basically stall, doing nothing.

It's also error prone such that it can miss very small details due to taking a fixed constant step size at each sample. If the small detail next to the ray is smaller than the step size, we might jump over it and have an incorrect reflection. The number of steps taken and how large those steps are make a huge difference in terms of quality for this kind of linear ray marching. This technique also produces staircase artifacts, for which you have to employ some form of a refinement once an intersection point is found. This refinement would be between the previous ray-march position and the ray-march intersection point to converge into a much more refined intersection. A low number of binary search steps or a single secant search is usually enough to deal with the staircase artifacts for pure specular rays. (See Figure 4.5.) Crytek employs ray length jittering at each iteration step to hide the staircase artifacts.

A simple linear ray-marching algorithm that traverses the depth buffer can be written with fewer than 15 lines of code, as we can see in Listing 4.1.

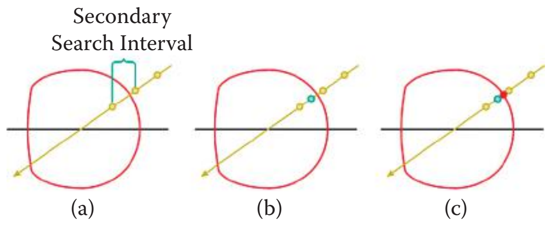


Figure 4.5. Binary search illustration between the intersection position and the last position of the ray. Basically it takes the middle point of the two and checks if it's still intersecting; if true it does it again until it can resurface, which is a refined position. We can also visualize the constant step sizes linear ray marching takes and end up in the wrong coordinate, which results in staircase artifacts, so we need some form of a refinement. Binary search and secant search are popular ones. [Original image courtesy of [Risser 07].]

```
#define LINEAR_MARCH_COUNT 32
for(int i = 0; i < LINEAR_MARCH_COUNT; ++i)
{
    // Read scene depth with current ray.
    float d = depthBuffer.SampleLevel( pointSampler, ray.xy, 0 );

    // Check if ray is greater than the scene, it means we
    // intersected something so end.
    if( ray.z > d )
        break;

    // Else advance the ray by a small step and continue the
    // loop. Step is a vector in screen space.
    ray += step;
}
```

Listing 4.1. A simple linear ray marching to illustrate the concept of walking a depth buffer until an intersection is found. We can quickly see that this technique is fetch bound. The ALU units are not doing a lot of work as there are too few ALU instructions to hide the latency of the global memory fetches the shader has to wait to complete.

So let's take a close look at some major problems for this kind of a technique.

1. It takes small-sized steps, it conducts many fetches, and latency starts to bottleneck quickly.

2. It can miss small details in the case that the step it takes is larger than the small detail next to the ray.

3. It produces staircase artifacts and needs a refinement such as a secant or binary search.

4. It is only fast for short travels; ray-marching an entire scene will stall the cores and result in slow performance.

Our goal is to introduce an algorithm that can solve all four points.

All of those points can be solved by introducing an acceleration structure, which can then be used to accelerate our rays, basically traversing as much distance as the ray can possibly take without risking missing any details at all. This acceleration structure will allow the ray to take arbitrary length steps, and especially large ones as well whenever it can. It's also going to conduct fewer fetches and thereby perform faster. The acceleration structure is going to produce great results without needing an extra refinement pass, although it doesn't hurt to do a final secant search between the previous pixel and the current pixel because an acceleration structure is usually a discrete set of data. Since we gain major speedups by using an acceleration structure, we will also be able to travel longer and ray-march an entire scene with much better performance.

The algorithm called Hi-Z Screen-Space Cone Tracing proposed in this chapter can reflect an entire scene with quick convergence and performs orders of magnitude faster than the linear constant step-based ray-marching algorithm.

4.4 Algorithm

The proposed algorithm can be divided into five distinct steps:

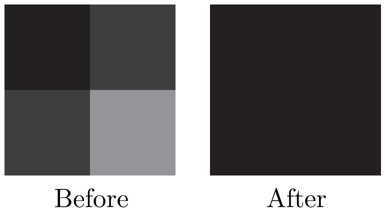
1. Hi-Z pass,
2. pre-integration pass,
3. ray-tracing pass,
4. pre-convolution pass,
5. cone-tracing pass.

We will go through each step by step now.

4.4.1 Hi-Z Pass

The *Hierarchical-Z buffer*, also known as the Hi-Z buffer, is constructed by taking the minimum or maximum of the four neighboring values in the original Z-buffer and storing it in a smaller buffer at half the size. In our case for this chapter, we will go with the minimum version.

The Z-buffer holds the depth values of the 3D scene in a buffer such as a texture/image. The figure below represents the minimum value version of how a Hi-Z construction works:



The result is a coarse representation of the original buffer. We do this consecutively on the resulting buffers until we arrive at a buffer with the size of 1, where we no longer can go smaller. We store the computed values in the mip-channels of a texture. This is represented in Figure 4.6.

The result is what we call a Hi-Z buffer, because it represents the Z values (also known as scene depth values) in a hierarchical fashion.

This buffer is the heart of the algorithm. It's essentially a screen/image aligned quad tree that allows us to accelerate the ray-tracing algorithm by noticing and

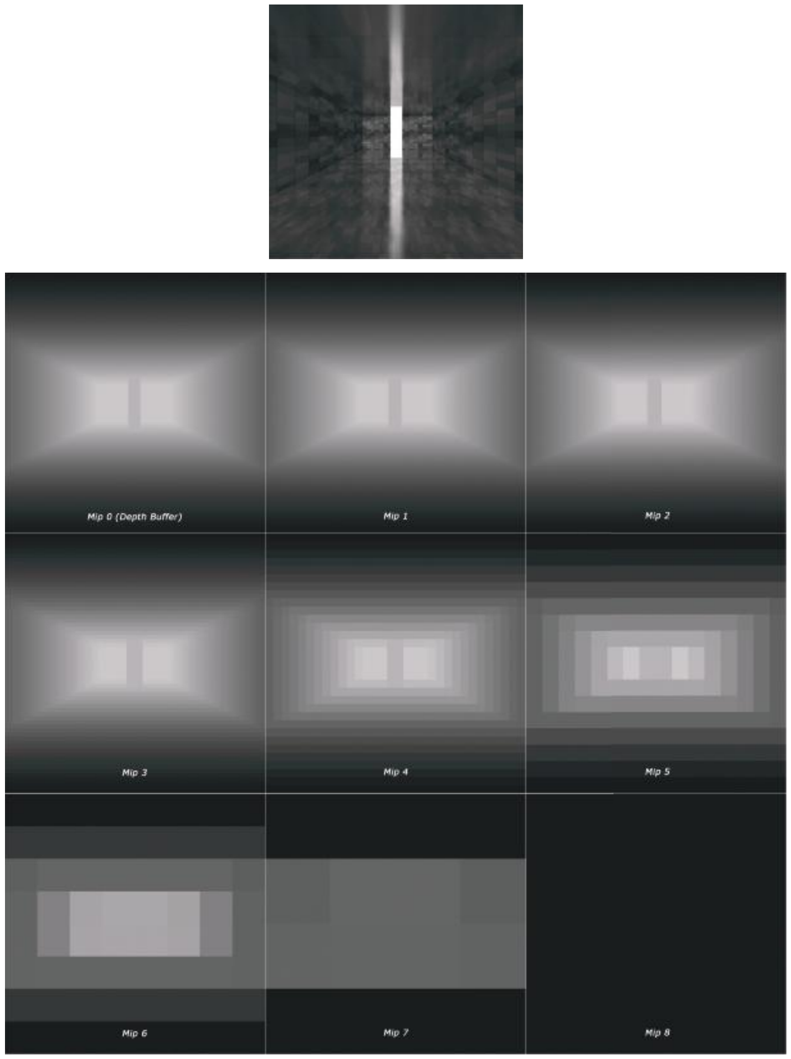


Figure 4.6. The original scene (top) and the corresponding Hi-Z (Hierarchical-Z) buffer (bottom) where the 2×2 minimum depths have been used successively to construct it. It serves as an acceleration structure for our rays in screen space. Mip 0 is our depth buffer that represents the scene depth per pixel. At each level we take the minimum of 2×2 pixels and produce this hierarchical representation of the depth values.

skipping empty space in the scene to efficiently and quickly arrive at our desired intersection point/coordinate by navigating in the different hierarchy levels. Empty space in our case is the tiles we see in the image, the quads.

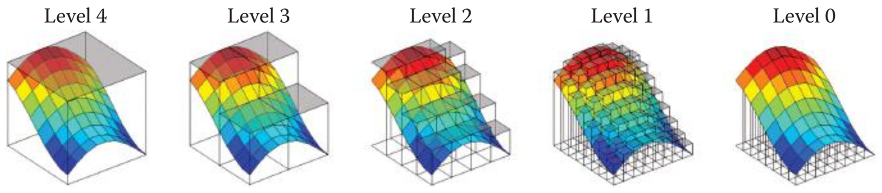


Figure 4.7. The Hi-Z (Hierarchical-Z) buffer, which has been unprojected from screen space into world space for visualization purposes. [Image courtesy of [Tevs et al. 08].]

Unlike the previously developed methods, which take constant small steps through the image, the marching method we investigate runs much faster by taking large steps and converges really quickly by navigating in the hierarchy levels.

Figure 4.7 shows a simple Hierarchical-Z representation unprojected from screen space back into world space for visualization purposes. It's essentially a height field where dark values are close to the camera and bright values are farther away from the camera.

Whether you construct this pass on a post-projected depth buffer or a view-space Z-buffer will affect how the rest of the passes are handled, and they will need to be changed accordingly.

4.4.2 Pre-integration Pass

The pre-integration pass calculates the scene visibility input for our cone-tracing pass in a hierarchical fashion. This pass borrows some ideas from [Crassin 11], [Crassin 12], and [Lilley et al. 12] that are applied to voxel structures and not 2.5D depth. The input for this pass is our Hi-Z buffer. At the root level all of our depth pixels are at a 100% visibility; however, as we go up in this hierarchy, the total visibility for the coarse representation of the cell has less or equal visibility to the four finer pixels:

$$\text{Visibility}_n \leq \text{Visibility}_{n-1}.$$

(See also Figure 4.8.) Think about the coarse depth cell as a volume containing the finer geometry. Our goal is to calculate how much visibility we have at the coarse level.

The cone-tracing pass will then sample this pre-integrated visibility buffer at various levels until our ray marching accumulates a visibility of 100%, which means that all the rays within this cone have hit something. This approximates the cone footprint. We are basically integrating all the glossy reflection rays. We start with a visibility of 1.0 for our ray; while we do the cone tracing, we will keep subtracting the amount we have accumulated until we reach 0.0. (See Figure 4.9.)

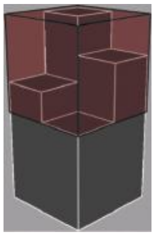


Figure 4.8. The area of interest between the minimum and maximum depth plane of the four pixels for which we calculate the visibility; basically, take the percentage of empty volume.

We cannot rely on only the visibility buffer, though. We must know how much our cone actually intersects the geometry as well, and for that we will utilize our Hi-Z buffer. Our final weight will be the accumulated visibility multiplied by how much our cone sphere is above, in between, or below the Hi-Z buffer.

The format for this pre-integration buffer is an 8 bit per channel texture that gives 256 values to represent our visibility. This gives 0.390625% of increments for our visibility values (1.0/256.0), which is good enough precision for transparency.

Again, this pass is highly dependent on whether we have a post-projected depth Hi-Z or a view-space Hi-Z buffer.

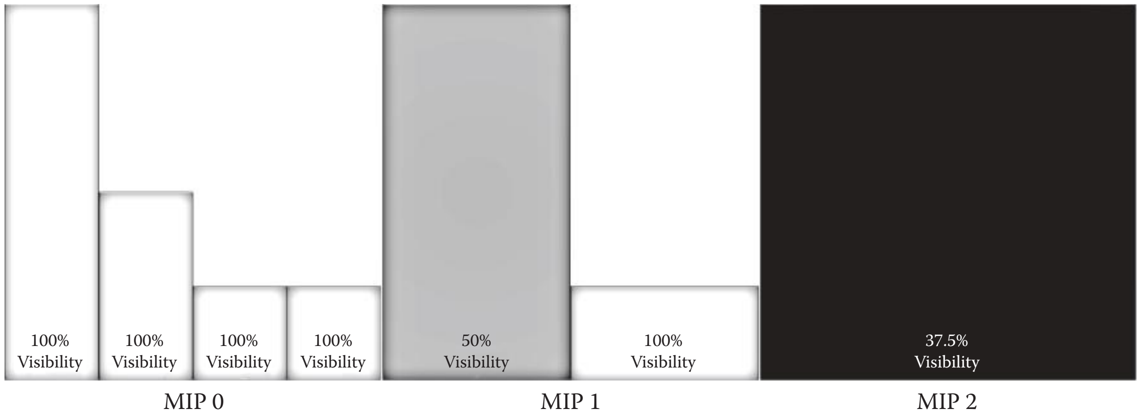


Figure 4.9. A 2D representation of the hierarchical pre-integrated visibility buffer. The percent is calculated between the minimum and maximum depths. The height is the depth and the color is the amount of visibility.

4.4.3 Ray-Tracing Pass

The following function is the reflection formula where $\vec{V}$ is the view direction and $\vec{N}$ is the surface normal direction (surface orientation) and the return value is the reflection vector:

$$Reflect(\vec{V}, \vec{N}) = 2(\vec{V} \cdot \vec{N})\vec{N} - \vec{V}.$$

The dot is the dot product, also known as the scalar product, between the two vectors. We will later use this function to calculate our reflection direction for the algorithm.

Before we continue with the ray-tracing algorithm, we have to understand what the depth buffer of a 3D scene actually contains. The depth buffer is referred to as being nonlinear, meaning that the distribution of the depth values of a 3D scene does not increase linearly with distance to the camera. We have a lot of precision close to the camera and less precision far away, which helps with determining which object is closest to the camera when drawing, because closer objects are more important than farther ones.

By definition division is a nonlinear operation and there is a division happening during perspective correction, which is where we get the nonlinearity from. A nonlinear value can't be linearly interpolated. However, while it is true that the Z-values in the depth buffer are not linearly increasing relative to the Z-distance from the camera, it is on the other hand indeed linear in screen space due to the perspective. Perspective-correct rasterization hardware requires linear interpolation across an entire triangle surface when drawing it from only three vertices. In particular the hardware interpolates 1/Z for each point that makes up the surface of the triangle using the original three vertices.

Linear interpolation of Z directly does not produce correct depth values across the triangle surface, though 1/Z does [Low 02]. Figure 4.10 explains why non-perspective interpolation is wrong.

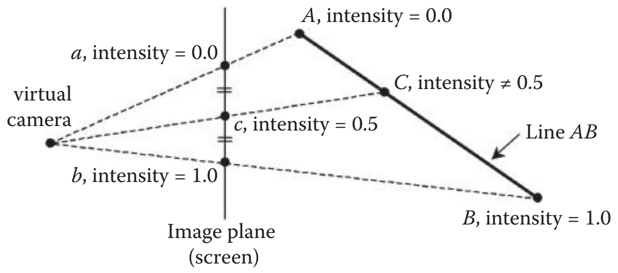


Figure 4.10. Illustration of interpolating an attribute directly in screen space giving incorrect results. One must do perspective correct interpolation as described in [Low 02]. The depth buffer value, 1/Z, is perspective correct so this allows us to interpolate it in screen space without any further computation.

We can observe the fact that the depth buffer values are linear in screen space, due to perspective, by taking the partial derivatives, gradients, of them using `ddx` and `ddy` instructions in Microsoft HLSL and outputting them as color values. For any planar surface the result is going to be a constant color, which tells us a linear rate of change the farther the planes are from the camera in screen space.

Anything that behaves linearly is also going to allow us to interpolate it, just like the hardware, which is a very powerful fact. It's also the reason we did the Hi-Z construction on the nonlinear depth buffer. Our ray tracing will happen in screen space, and we would like to exploit the fact that the depth buffer values can be interpolated correctly in screen space because they're perspective-corrected. It's like the perspective cancels out this nonlinearity of the values.

In the case that one desires to use a view-space Hi-Z buffer and not a post-projected buffer, one has to manually interpolate the Z-value just as perspective interpolation does, 1/Z. Either case is possible and affects the rest of the passes as mentioned earlier. We will assume that we use a post-perspective Hi-Z from now on. Now that we know the depth buffer values can be interpolated in screen space, we can go back to the Hi-Z ray-tracing algorithm itself and use our Hi-Z buffer.

We can parameterize our ray-tracing algorithm to exploit the fact that depth buffer values can be interpolated. Let $\mathbf{O}$ be our starting screen coordinate, the origin, let the vector $\vec{D}$ be our reflection direction, and finally let t be our driving parameter between 0 and 1 that interpolates between the starting coordinate $\mathbf{O}$ and ending coordinate $\mathbf{O} + \vec{D}$:

$$Ray(t) = \mathbf{O} + \vec{D} * t,$$

where the vector $\vec{D}$ and point $\mathbf{O}$ are defined as

$$\vec{D} = \vec{V}_{ss}/\vec{V}_{ss_z},$$

$$\mathbf{O} = \mathbf{P_{ss}} + \vec{D} * -\mathbf{P_{ss_z}}.$$

$\vec{D}$ extends all the way to the far plane now. The division by $\vec{V}_z$ sets the Z-coordinate to 1.0, but it still points to the same direction because division by a scalar doesn't change a vector's direction. $\mathbf{O}$ is then set to the point that corresponds to a depth of 0.0, which is the near plane. We can visualize this as a line forming from the near plane to the far plane in the reflection direction crossing the point we are shading in Figure 4.11.

We can now input any value t to take us between the starting point and ending point for our ray-marching algorithm in screen space. The t value is going to be a function of our Hierarchical-Z buffer.

But we need to compute the vector $\vec{V}$ and $\mathbf{P}$ first to acquire $\mathbf{O}$ and $\vec{D}$. $\mathbf{P}$ is already available to us through the screen/texture coordinate and depth. To get $\vec{V}$ we need another screen-space point $\mathbf{P}'$, which corresponds to a point somewhere

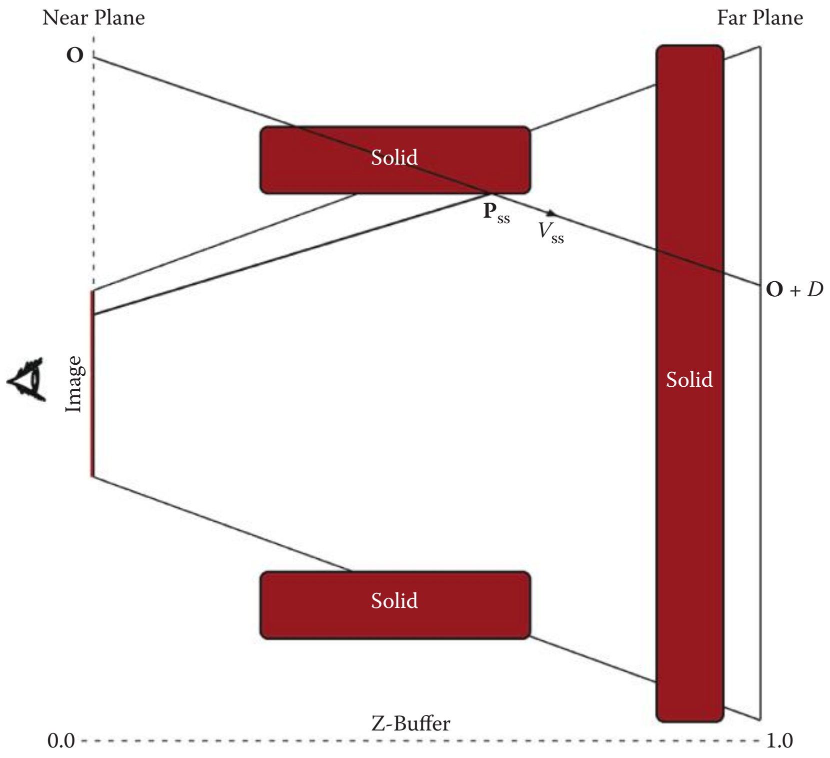


Figure 4.11. An illustration showing $\mathbf{O}$, $\vec{D}$, $\mathbf{P}$, and $\vec{V}$ variables from the equations. $\mathbf{O} + \vec{D} * t$ will take us anywhere between starting point $\mathbf{O}$ and ending point $\mathbf{O} + \vec{D}$ where t is between 0 and 1. Note that $\vec{V}$ is just a vector, direction, and has no position. It was put on the line for visualization purposes.

along the reflection direction. Taking the difference of these two will yield us a screen-space reflection vector:

$$\vec{V}_{ss} = \mathbf{P}'_{\mathbf{ss}} - \mathbf{P}_{\mathbf{ss}},$$

where the available $\mathbf{P}$ is defined as

$$\mathbf{P}_{\mathbf{ss}} = \{\mathbf{texcoord}_{\mathbf{xy}} \ \ \mathbf{depth}\}.$$

The other $\mathbf{P}'$ along the reflection direction can be computed by taking the view-space point, view-space direction, and view-space surface normal, computing a view-space reflection point, projecting it into clip space $[-1, 1]$ range, and finally converting from clip space into screen space $[0, 1]$ range, as we can see below:

$$\mathbf{P_{cs}} = (\mathbf{P_{vs}} + Reflect(\vec{V}_{vs}, \vec{N}_{vs})) * \mathbf{M_{proj}},$$
$$\mathbf{P'_{ss}} = \frac{\mathbf{P_{cs}}}{\mathbf{P_{cs_w}}} * [0.5 \;\; -0.5] + [0.5 \;\; 0.5].$$

Once we have a screen-space reflection vector, we can run the Hi-Z traversal to ray-march along the acceleration structure using $\mathbf{O}$, $\vec{D}$, and t.

We'll first look at the pseudo code in Listing 4.2. The algorithm uses the Hi-Z buffer we constructed earlier to accelerate the ray marching. To visualize the algorithm in Listing 4.2 step by step, follow Figure 4.12.

Once the ray-tracing algorithm has run, we have our new coordinate in screen space that is our ray intersection point. Some ideas are borrowed from displacement mapping techniques found in the literature [Hien and Lim 09, Hrkalovic and Lundgren 12, Oh et al. 06, Drobot 09, Szirmay-Kalos and Umenhoffer 06]. One major difference is that we start on the root level while displacement techniques start on the leaf level for marching a ray. Ray-marching with a view-space Z-buffer is a bit more involved because we have to manually interpolate the Z-coordinate as it is not possible to interpolate it in screen space.

```
level = 0 // starting level to traverse from

while level not below N // ray-trace until we descend below the
                        // root level defined by N, demo used 2

    minimumPlane  = getCellMinimumDepthPlane( . . . )
    // reads from the Hi-Z texture using our ray
    boundaryPlane = getCellBoundaryDepthPlane( . . . )
    // gets the distance to next Hi-Z cell boundary in ray
    // direction

    closestPlane  = min( minimumPlane, boundaryPlane )
    // gets closest of both planes

    ray = intersectPlane( . . . )
    // intersects the closest plane, returns O + D * t only.

    if intersectedMinimumDepthPlane
    // if we intersected the minimum plane we should go down a
    // level and continue
       descend a level

    if intersectedBoundaryDepthPlane
    // if we intersected the boundary plane we should go up a
    // level and continue
       ascend a level

color = getReflection( ray ) // we are now done with the Hi-Z ray
                       // marching so get color from the intersection
```

Listing 4.2. Pseudo code for implementing the Hi-Z ray tracing.

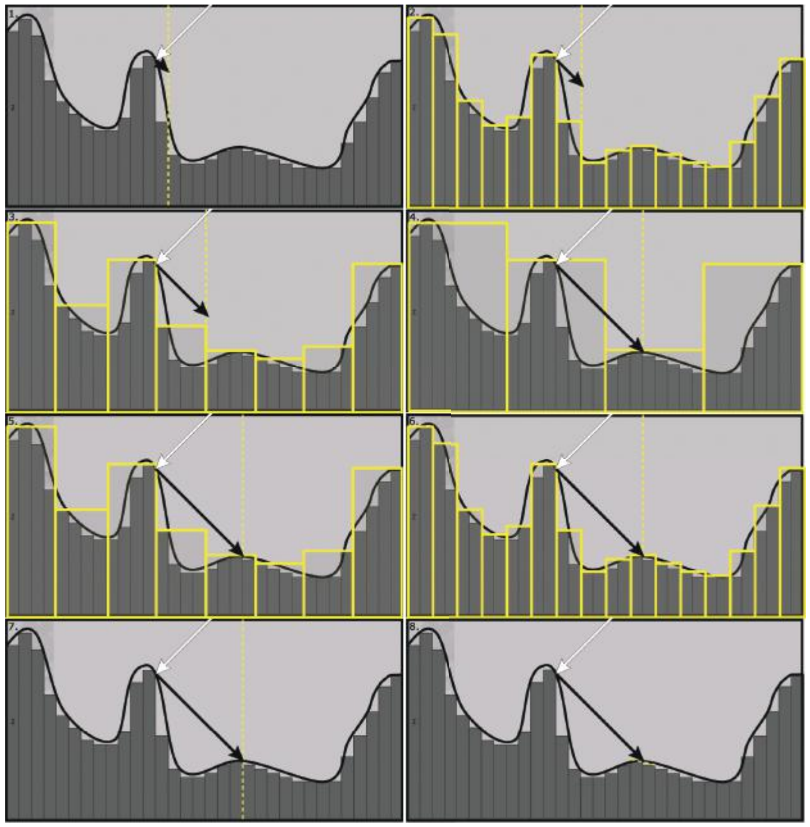


Figure 4.12. Hi-Z ray-tracing step by step going up and down in the hierarchy of the buffer to take longer jumps at each step. [Source image courtesy of [Drobot 09].]

4.4.4 Pre-convolution Pass

The pre-convolution pass is an essential pass for the algorithm for computing blurred glossy reflections emitted from microscopic rough surfaces. Just like in the Hi-Z pass, which outputs a hierarchy of images, this pass also does so, but with a different goal in mind.

We convolve the original scene color buffer to produce several different blurred versions out of it as we can see in Figure 4.13. The final result is another hierarchical representation, images at different resolutions with different levels of convolution, which is stored in the mip-map channels.

These blurred color buffers will help out with accelerating rough reflections to achieve results similar to what we can see in Figure 4.14.

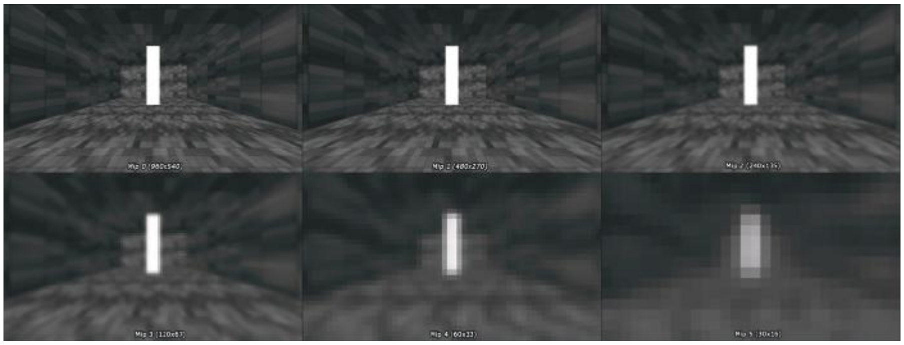


Figure 4.13. Convolved color texture of a simple scene with different level of convolution at each level. This will be later used to create our rough reflections.

Usually to simulate this kind of blurred reflections in ray-tracing-based renderers, we would shoot a lot of diverged rays defined by the cone aperture, say 32 more, and average the resulting colors together to produce a blurred reflection. (See Figures 4.15 and 4.16.)

However, this quickly becomes a very expensive operation and the performance decreases linearly with the number of rays we shoot, and even then the technique produces noisy and unacceptable results that need further processing to smooth out. One such technique is called image-space gathering [Robison and Shirley 09], which works really well even on pure mirror reflections to make them appear like rough reflections as a post-process.

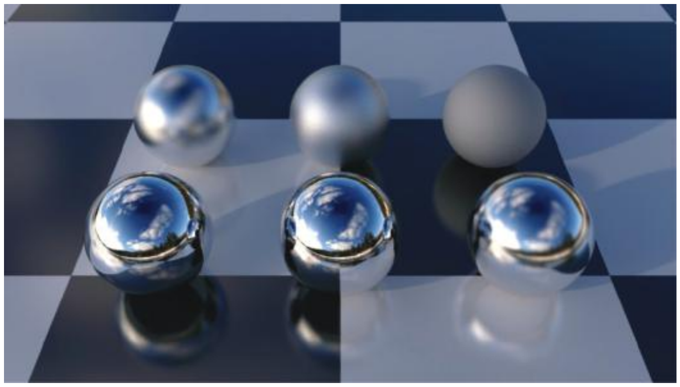


Figure 4.14. Different levels of roughness on the spheres produce diverged reflection rays, which results in the viewer perceiving blurred reflections. [Image courtesy of [David 13].]

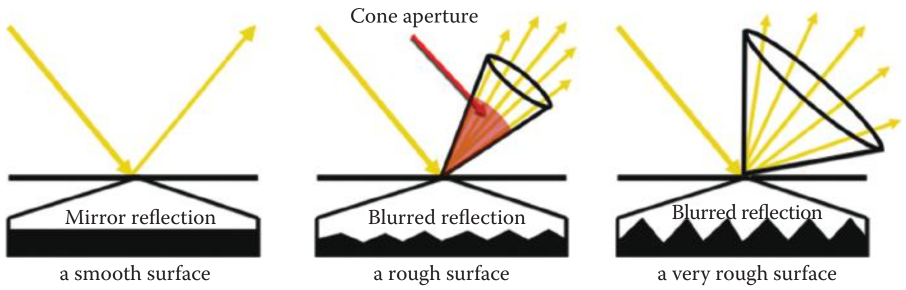


Figure 4.15. The rougher the surface is at a microscopic level, the more blurry and weaker the reflection appears. Fewer rays hit the iris of the perceivers' eye, which gives the weaker appearance. [Original image courtesy of [ScratchaPixel 12].]

Another drawback of shooting randomly jittered rays within the cone aperture is the fact that parallel computing hardware such as the GPU tends to run threads and memory transaction in groups/batches. If we introduce jittered rays we slow down the hardware because now all the memory transactions are in memory addresses far away from each other, slowing the computation by tremendous amounts because of cache-misses and global memory fetches, and bandwidth becomes a bottleneck.

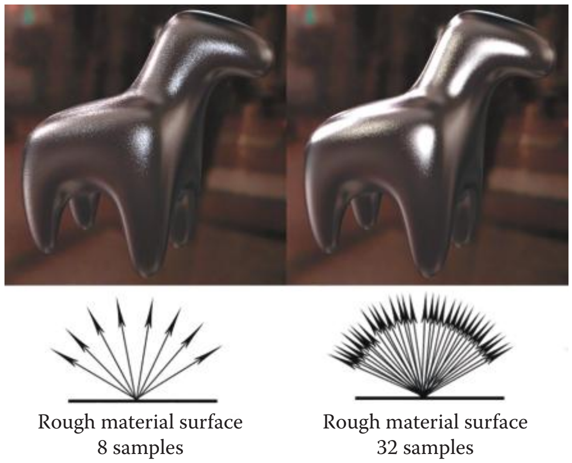


Figure 4.16. Noisy reflections produced by firing multiple diverged rays and averaging the results together for glossy reflections. Even 32 rays per pixel are not enough to create a perfectly smooth reflection and the performance decreases linearly with each additional ray. [Original images courtesy of [Autodesk 09] and [Luxion 12].]

Section 4.4.5 will propose a method that is merely an approximation but runs really fast without the need for firing multiple rays or jittering. By just navigating in the hierarchy of the blurred color images we discussed in this pass, depending on the reflection distance and roughness of the surface we are reflecting from, we can produce accurate glossy reflections.

4.4.5 Cone-Tracing Pass

The cone-tracing pass runs right after the Hi-Z ray-tracing pass finishes, and it produces the glossy reflections of the algorithm. This pass uses all our hierarchical buffers.

The output from the ray-tracing pass is our screen-space intersection coordinate as we saw earlier. With that we have all the knowledge to construct a screen-space aligned cone, which essentially becomes an isosceles triangle.

The idea is simple; Figure 4.17 shows a cone in screen space that corresponds to how much the floor diverges the reflection rays at maximum. Our goal is to accumulate all the color within that cone, basically integrate for every single ray that diverges. This integration can be approximated by sampling at the circle centers, where the size of the circle decides at which hierarchy level we read the color from our textures, as we saw in Section 4.4.3. Figure 4.18 illustrates this.

We determine whether the cone intersect our Hi-Z at all. If it does, we determine how much it intersects and multiply this weight with the pre-integrated visibility for that level and point. This final weight is accumulated until we reach 100%, and we weigh the color samples as well during the traversal. How you determine whether the cone intersects the Hi-Z for empty space is highly dependent on whether you use a post-projected Hi-Z or view-space Hi-Z.

First, we need to find the cone angle for a specific roughness level. Our reflection vector is basically the Phong reflection model because we just compute a reflection vector by reflecting the view direction on the normal. To approximate

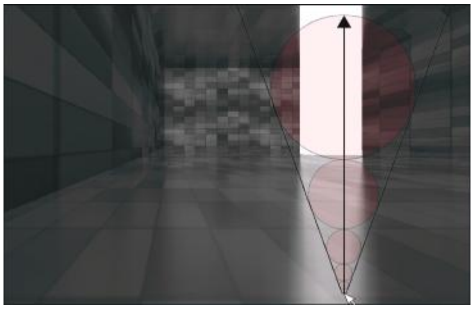


Figure 4.17. A cone in screen space is essentially an isosceles triangle with the in-radius circles that will be able to sample/read the hierarchical buffers.

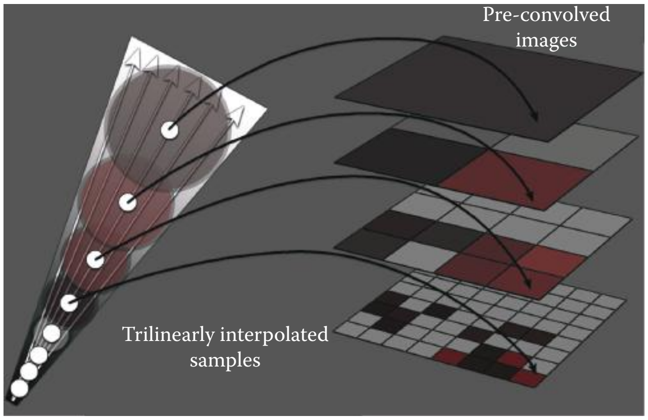


Figure 4.18. The cone integrates the rays, arrows, we see in the image by sampling, reading, at different levels in our convolved, blurred, color image depending on the distance and surface roughness. It blends both between the neighboring pixels and between the hierarchy levels, which is what we call *trilinear interpolation* for smooth transitions and blended results.

the cone angle for the Phong model, we use

$$\theta = \cos\left(\xi^{\frac{1}{\alpha+1}}\right), \tag{4.1}$$

where α is the specular power and ξ is hard-coded to 0.244. This is the basic formula used for importance sampling of a specular lobe; it is the inverse cumulative distribution function of the Phong distribution. Importance-sampling applications generate a bunch of uniform random variables [0–1] for ξ and use Equation 4.1 to generate random ray directions within the specular lobe in spherical coordinates [Lawrence 02]. The hard-coded value 0.244 seems to be a good number for covering a decent range of cone-angle extents. Figure 4.19 shows how well this equation maps to the cone angle extents of the Phong specular lobe in polar coordinates.

To get a perfectly mirrored ray with the Phong model, the specular power value would need to be infinity. Since that will not happen and graphics applications usually have a cap on their specular power value, we need a threshold value to support mirror reflections for the cone tracer. We can clearly see that there is not much change between a power of 1024 and 2048. So, any specular power in the range of 1024–2048 should be interpolated down to an angle of 0.

If we want another type of reflection model with more complicated distribution, we would need to pre-compute a 2D lookup table and index it with roughness as the u coordinate and $\vec{V}\cdot\vec{N}$ as the v coordinate, which would then return a local reflection direction. This local reflection direction would need to be transformed into a global reflection vector for the Hi-Z tracing pass.

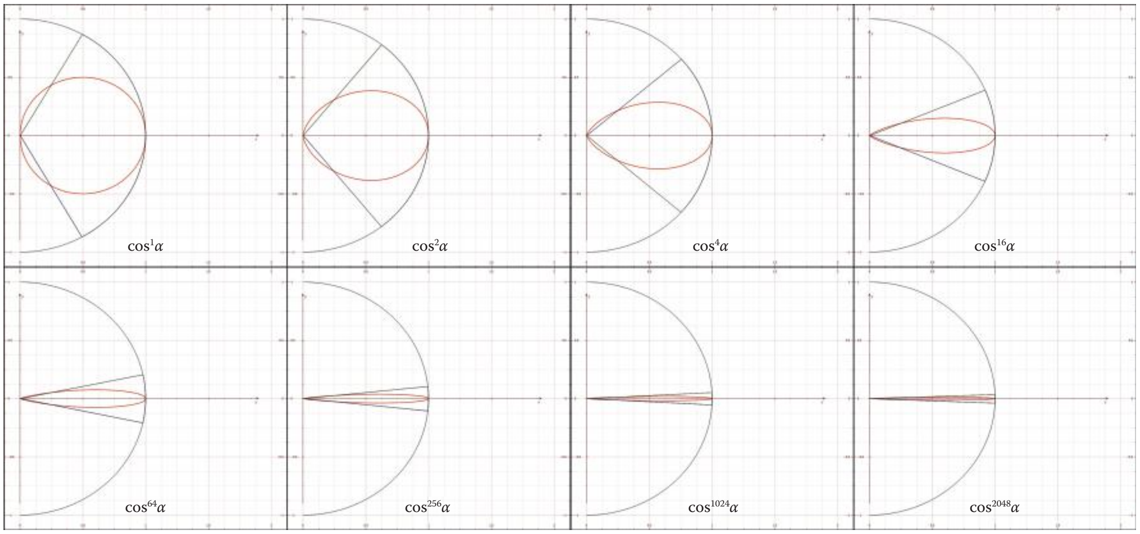


Figure 4.19. Polar coordinate plot of the specular lobe with various specular power values. The red ellipse is the specular lobe and the black isosceles triangle shows the cone angle extents using the formula presented earlier; α is the angle and $\cos\alpha$ is powered to various specular power values.

So, for any distribution model, we average at pre-computation time all the reflection vectors within the specular lobe—importance sampling using uniform random variables [0–1]—with a specific roughness value that gives the vector where the reflection vector is strongest, and then we store this vector in a 2D texture table. The reason we average all the reflection vectors within the lobe is the fact that complicated BRDF models often don't produce a lobe with respect to the pure specular reflection vector. They might be more vertically stretched or behave differently at grazing angles, and we are interested in finding the reflection vector that is the strongest within this specular lobe, which we can clearly see in Figure 4.20.

The RGB channel of this table would contain the local reflection vector and the alpha channel would contain either an isotropic cone-angle extent with a single value or anisotropic cone-angle extents with two values for achieving vertically stretched reflections, which we revisit later.

For this chapter, we just assume that we use a Phong model. We need to construct an isosceles triangle for the cone-tracing pass using the newly obtained angle θ. Let $\mathbf{P_1}$ be the start coordinate of our ray in screen space and $\mathbf{P_2}$ be the end coordinate of our ray, again in screen space. Then the length l is defined as

$$l = \|\triangle_{\mathbf{P}}\| = \|\mathbf{P_2} - \mathbf{P_1}\|.$$

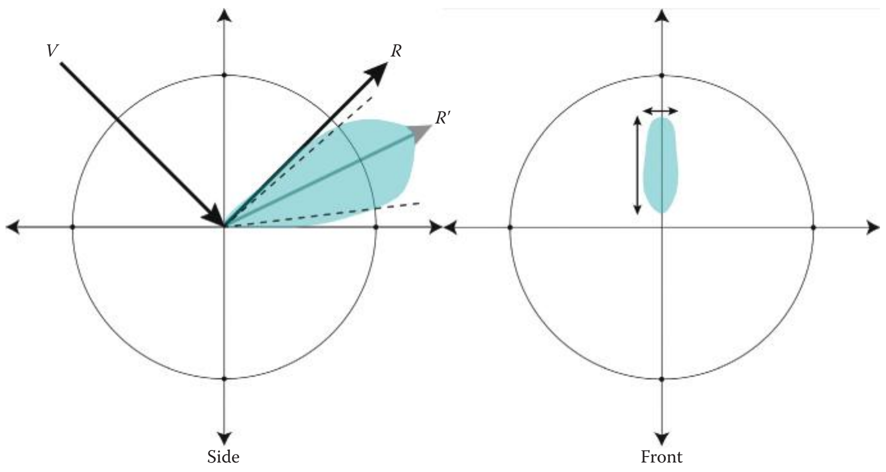


Figure 4.20. Spherical coordinate preview of a specular lobe for a complicated distribution. We can clearly see that the lobe does not necessarily need to be centered around the pure reflection vector. If we average all the vectors within the lobe, we get a new reflection vector $\vec{R}'$ that represents our reflection direction more precisely.

Once we have the length for our intersection, we can assume that it's the adjacent side of our isosceles triangle. With some simple trigonometry we can calculate the opposite side as well. Trigonometry says that the tangent of θ is the opposite side over the adjacent side:

$$\tan(\theta) = \frac{opp}{adj}.$$

Using some simple algebra we discover that the opposite side that we are looking for is the tangent of θ multiplied by the adjacent side:

$$opp = \tan(\theta)adj.$$

However, this is only true for right triangles. If we look at an isosceles triangle, we discover that it's actually two right triangles fused over the adjacent side where one is flipped. This means the opposite is actually twice the opposite of the right triangle:

$$opp = 2\tan(\theta)adj \tag{4.2}$$

Once we have both the adjacent side and the opposite side, we have all the data we need to calculate the sampling points for the cone-tracing pass. (See Figure 4.21.)

To calculate the in-radius (circle radius touching all three sides) of an isosceles triangle, this equation can be used:

$$r = \frac{a(\sqrt{a^2 + 4h^2} - a)}{4h},$$

where a is the base of the isosceles triangle, h is the height of the isosceles triangle, and r is the resulting in-radius. Recall that the height of the isosceles triangle is

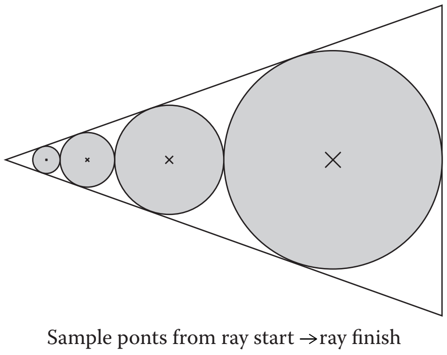


Figure 4.21. The isosceles triangle with the sampling points for the cone-tracing pass in screen space. To find the sample points of our cone in screen space, we have to use some geometry and calculate the in-radius of this isosceles triangle. Note that this is an approximation and we are not fully integrating the entire cone.

the length of our intersection we calculated before and the base of our isosceles triangle is the opposite side. Using this formula we find the radius of the in-radius circle. Once we have the in-radius of the isosceles triangle, we can take the adjacent side and subtract the in-radius from it to find the sampling point we are interested in. We can now read the color from the correct coordinate in screen space.

To calculate the rest of the sampling points, all we have to do is subtract the in-radius another time to reach the leftmost side of the circle and then recalculate the opposite side with this new adjacent side using equation 4.2, and then rerun the in-radius formula to get the next smaller circle. We do this successively for as many samples as we want to take.

We accumulate the correct color by using a trilinear filtering scheme (smoothly filtering between the neighboring pixels and between the hierarchy levels). We also weigh the color with the transparency buffer and by how much our cone-sphere intersects the coarse depth cells. This is done in front-to-back order, so it is basically a linear search algorithm. The larger the cone is, the faster it runs. The weight is accumulated to know how much visibility is integrated. One might want to take smaller offsets between the circles to achieve smoother results; however, that gets more expensive. If the cone tracer doesn't accumulate a visibility of 100%, we can blend in the rest of the visibility using, say, cube-maps with the same roughness.

Again depending on the format of the Hi-Z buffer, if we use a view-space Z version, then how we determine whether the cone-sphere intersects the Hi-Z buffer—as well as how we calculate the sampling points on the cone—is different One can use the cone angle with the view-space Z distance to find the sphere size and then project this using perspective division into screen space, keeping aspect ratio in mind.

4.5 Implementation

We have now looked at all the algorithms; let's go through them in the same order again, looking at sample code for implementation details.

4.5.1 Hi-Z Pass

The code snippet in Listing 4.3 shows how to implement a Hi-Z construction pass in DirectX using Microsoft HLSL (High Level Shading Language). This shader is executed successively, and the results are stored in the mip-channels of the Hi-Z buffer. We read from level $N - 1$ and write to N until we reach a size of 1×1 as mentioned in Section 4.4.1.

To render into the same texture that we read from in DirectX 11 terms, we will have to make sure that our `ID3D11ShaderResourceView` objects point to a single mip-channel and not the entire range of mip-channels. The same rule applies to our `ID3D11RenderTargetViews` objects.

4.5.2 Pre-integration Pass

The code snippet in Listing 4.4 shows how to implement a pre-integration pass in DirectX using Microsoft HLSL. It basically calculates the percentage of empty

```
float4 main( PS_INPUT input ) : SV_Target
{
    // Texture/image coordinates to sample/load/read the depth
    // values with.
    float2 texcoords = input.tex;

    // Sample the depth values with different offsets each time.
    // We use point sampling to ignore the hardware bilinear
    // filter. The constant prevLevel is a global that the
    // application feeds with an integer to specify which level
    // to sample the depth values from at each successive
    // execution. It corresponds to the previous level.
    float4 minDepth;

    minDepth.x = depthBuffer.SampleLevel( pointSampler,
        texcoords, prevLevel, int2(  0,  0) );
    minDepth.y = depthBuffer.SampleLevel( pointSampler,
        texcoords, prevLevel, int2(  0, -1) );
    minDepth.z = depthBuffer.SampleLevel( pointSampler,
        texcoords, prevLevel, int2( -1,  0) );
    minDepth.w = depthBuffer.SampleLevel( pointSampler,
        texcoords, prevLevel, int2( -1, -1) );

    // Take the minimum of the four depth values and return it.
    float d = min( min(minDepth.x, minDepth.y), min(minDepth.z,
        minDepth.w) );

    return d;
}
```

Listing 4.3. How to implement the Hierarchical-Z buffer taking the minimum of 2×2 depth values from the depth buffer.

space within the minimum and maximum of a depth cell and modulates with the previous transparency.

```
float4 main( PS_INPUT input ) : SV_Target
{
    // Texture/image coordinates to sample/load/read the depth
    // values with.
    float2 texcoords = input.tex;

    float4 fineZ;
    fineZ.x    = linearize( hiZBuffer.SampleLevel( pointSampler,
        texcoords, mipPrevious, int2(  0,  0) ).x );
    fineZ.y    = linearize( hiZBuffer.SampleLevel( pointSampler,
        texcoords, mipPrevious, int2(  0, -1) ).x );
    fineZ.z    = linearize( hiZBuffer.SampleLevel( pointSampler,
        texcoords, mipPrevious, int2( -1,  0) ).x );
    fineZ.w    = linearize( hiZBuffer.SampleLevel( pointSampler,
        texcoords, mipPrevious, int2( -1, -1) ).x );

    // hiZBuffer stores min in R and max in G.
    float minZ = linearize( hiZBuffer.SampleLevel( pointSampler,
        texcoords, mipCurrent ).x );
    float maxZ = linearize( hiZBuffer.SampleLevel( pointSampler,
        texcoords, mipCurrent ).y );

    // Pre-divide.
    float coarseVolume = 1.0f / (maxZ - minZ);

    // Get the previous four fine transparency values.
    float4 visibility;
    visibility.x = visibilityBuffer.SampleLevel( pointSampler,
        texcoords, mipPrevious, int2(  0,  0 ) ).x;
    visibility.y = visibilityBuffer.SampleLevel( pointSampler,
        texcoords, mipPrevious, int2(  0, -1 ) ).x;
    visibility.z = visibilityBuffer.SampleLevel( pointSampler,
        texcoords, mipPrevious, int2( -1,  0 ) ).x;
    visibility.w = visibilityBuffer.SampleLevel( pointSampler,
        texcoords, mipPrevious, int2( -1, -1 ) ).x;

    // Calculate the percentage of visibility relative to the
    // calculated coarse depth. Modulate with transparency of
    // previous mip.
    float4 integration = fineZ.xyzw * abs(coarseVolume)
        * visibility.xyzw;

    // Data-parallel add using SIMD with a weight of 0.25 because
    // we derive the transparency from four pixels.
    float coarseIntegration = dot( 0.25f, integration.xyzw );

    return coarseIntegration;
}
```

Listing 4.4. The demo uses both a minimum Hi-Z buffer and a maximum Hi-Z buffer. With them, we calculate how much empty space there is in between the hierarchy depth cells. We linearize the post-projected depth into view-space Z for the computation. We could also output a linear Z-buffer during the Hi-Z pass, but this would require some changes in the ray-tracing pass and cone-tracing pass because view-space Z cannot be interpolated in screen space by default.

4.5.3 Ray-Tracing Pass

The implementation in Listing 4.5 is the Hi-Z ray-tracing code in Microsoft HLSL. The code snippet is heavily commented and should be easy to follow once the algorithm presented in Section 4.4.3 is clear.

```
float3 hiZTrace( float3 p, float3 v )
{
    const float rootLevel = mipCount - 1.0f; // Convert to 0
                                             // based indexing

    float level         = HIZ_START_LEVEL; // HIZ_START_LEVEL was
                                           // set to 2 in the demo
    float iterations    = 0.0f;

    // Get the cell cross direction and a small offset to enter
    // the next cell when doing cell crossing.
    float2 crossStep, crossOffset;
    crossStep.x    = ( v.x >= 0 ) ? 1.f : -1.f;
    crossStep.y    = ( v.y >= 0 ) ? 1.f : -1.f;
    crossOffset.xy = crossStep.xy * HIZ_CROSS_EPSILON.xy;
    crossStep.xy   = saturate( crossStep.xy );

    // Set current ray to the original screen coordinate and
    // depth.
    float3 ray = p.xyz;

    // Scale the vector such that z is 1.0f
    // (maximum depth).
    float3 d = v.xyz /= v.z;

    // Set starting point to the point where z equals 0.0f (↩
        minimum depth).
    float3 o = intersectDepthPlane(p.xy, d.xy, -p.z);

    // Cross to next cell so that we don't get a self-
    // intersection immediately.
    float2 rayCell     = getCell(ray.xy, hiZSize.xy);
    ray                = intersectCellBoundary(o.xy, d.xy, ↩
        rayCell.xy, hiZSize.xy, crossStep.xy, crossOffset.xy);

    // The algorithm loop HIZ_STOP_LEVEL was set to 2 in the
    // demo; going too high can create artifacts.
    [loop]
    while( level >= HIZ_STOP_LEVEL && iterations < MAX_ITERATIONS↩
        )
    {
        // Get the minimum depth plane in which the current ray
        // resides.
        float minZ = getMinimumDepthPlane( ray.xy, level, ↩
            rootLevel );

        // Get the cell number of our current ray.
        const float2 cellCount  = getCellCount(level, rootLevel);
        const float2 oldCellIdx = getCell(ray.xy, cellCount);

        // Intersect only if ray depth is below the minimum depth
        // plane.
        float3 tmpRay           = intersectDepthPlane( o.xy, d.xy↩
            , max( ray.z, minZ) );
            // Get the new cell number as well.
            const float2 newCellIdx = getCell(tmpRay.xy, cellCount);

            // If the new cell number is different from the old cell
            // number, we know we crossed a cell.
            [branch]
            if( crossedCellBoundary(oldCellIdx, newCellIdx) )
            {
                // So intersect the boundary of that cell instead,
                // and go up a level for taking a larger step next
                // loop.
                tmpRay = intersectCellBoundary(o, d, oldCellIdx,
                    cellCount.xy, crossStep.xy, crossOffset.xy);
                level  = min(HIZ_MAX_LEVEL, level + 2.0f);
            }

            ray.xyz = tmpRay.xyz;

            // Go down a level in the Hi-Z.
            --level;

            ++iterations;
        } // end while

        return ray;
    }
```

Listing 4.5. Some of the functions are not shown because of code length. This is only a minimum tracing for the sake of simplicity. The full implementation of those functions can be seen with the demo code on the book's website. The demo uses minimum-maximum tracing, which is a bit more complicated than this. View-space Z tracing is a bit more complicated and not shown.

4.5.4 Pre-convolution Pass

The pre-convolution pass is just a simple separable blur with normalized weights so that they add up to 1.0 when summed—otherwise we would be creating more energy than what we had to begin with in the image. (See Figure 4.22.)

The filter is executed successively on the color image and at each step we reduce the image to half the size and store it in the mip-channels of the texture. Assuming that we are at half resolution, this would correspond to 960 × 540; when convolving level 2 (240 × 135), we read from level 1 (480 × 270) and apply the separable blur passes.

The 1D Gaussian function for calculating our horizontal and vertical blur weights is

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}.$$

So, for example, a 7 × 7 filter would have an inclusive range from −3 to 3 for x.

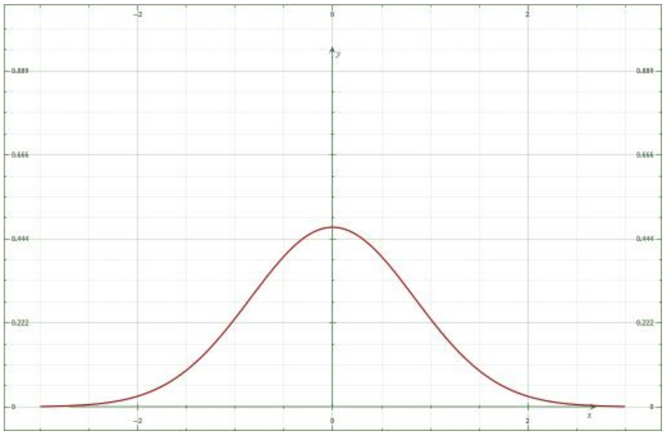


Figure 4.22. The normalized Gaussian curve used in the demo for weighting colors.

The normalized Gaussian weights then would be 0.001, 0.028, 0.233, 0.474, 0.233, 0.028, and 0.001, which when summed equal 1.0 exactly.

We don't want to produce more energy when doing the local blurring in the image, we want to keep the total energy in the image the same so the weights must equal 1.0 when summed—otherwise we are going to end up with more energy than what we started with so our image would have become brighter.

Listing 4.6 is a simple simple horizontal and vertical Gaussian blur implementation in the shading language Microsoft HLSL.

The final convolved images are produced by running the horizontal blur passes first and then the vertical blur passes successively for each level, which then are stored in the mip-channel of our color texture.

The choice of 7×7 seems to give good matching results for the needs of the demo. Using a wider kernel with a wider range of weights will cause wrong results because our transparency buffer and the colors associated with it will not match on a pixel basis anymore. Notice that our Gaussian weights take the colors mostly from the two neighboring pixels.

4.5.5 Cone-Tracing Pass

The cone-tracing pass is one of the smaller and easier-to-understand passes. In short it calculates the in-radiuses for the triangle made up from the cone and samples at different levels in our hierarchical color convolution buffer and pre-integrated visibility buffer (see Listing 4.7). Refer back to Section 4.4.5 for the algorithm explanation. The result can be seen in Figure 4.23

```
// Horizontal blur shader entry point in
// psHorizontalGaussianBlur.hlsl
float4 main( PS_INPUT input ) : SV_Target
{
    // Texture/image coordinates to sample/load the color values
    // with.
    float2 texcoords = input.tex;
    float4 color;

    // Sample the color values and weight by the pre-calculated
    // normalized Gaussian weights horizontally.
    color += colorBuffer.SampleLevel( pointSampler, texcoords,
        prevLevel, int2( -3,  0 ) ) * 0.001f;
    color += colorBuffer.SampleLevel( pointSampler, texcoords,
        prevLevel, int2( -2,  0 ) ) * 0.028f;
    color += colorBuffer.SampleLevel( pointSampler, texcoords,
        prevLevel, int2( -1,  0 ) ) * 0.233f;
    color += colorBuffer.SampleLevel( pointSampler, texcoords,
        prevLevel, int2(  0,  0 ) ) * 0.474f;
    color += colorBuffer.SampleLevel( pointSampler, texcoords,
        prevLevel, int2(  1,  0 ) ) * 0.233f;
    color += colorBuffer.SampleLevel( pointSampler, texcoords,
        prevLevel, int2(  2,  0 ) ) * 0.028f;
    color += colorBuffer.SampleLevel( pointSampler, texcoords,
        prevLevel, int2(  3,  0 ) ) * 0.001f;

    return color;
}

// Vertical blur shader entry point in
// psVerticalGaussianBlur.hlsl
float4 main( PS_INPUT input ) : SV_Target
{
    // Texture/image coordinates to sample/load the color values
    // with.
    float2 texcoords = input.tex;
    float4 color;

    // Sample the color values and weight by the pre-calculated
    // normalized Gaussian weights vertically.
    color += colorBuffer.SampleLevel( pointSampler, texcoords,
        prevLevel, int2( 0,  3 ) ) * 0.001f;
    color += colorBuffer.SampleLevel( pointSampler, texcoords,
        prevLevel, int2( 0,  2 ) ) * 0.028f;
    color += colorBuffer.SampleLevel( pointSampler, texcoords,
        prevLevel, int2( 0,  1 ) ) * * 0.233f;
    color += colorBuffer.SampleLevel( pointSampler, texcoords,
        prevLevel, int2( 0,  0 ) ) * 0.474f;
    color += colorBuffer.SampleLevel( pointSampler, texcoords,
        prevLevel, int2( 0, -1 ) ) * 0.233f;
    color += colorBuffer.SampleLevel( pointSampler, texcoords,
        prevLevel, int2( 0, -2 ) ) * 0.028f;
    color += colorBuffer.SampleLevel( pointSampler, texcoords,
        prevLevel, int2( 0, -3 ) ) * * 0.001f;

    return color;
}
```

Listing 4.6. Simple horizontal and vertical separable blur shaders, with a 7×7 kernel.

```
// Read roughness from a render target and convert to a BRDF
// specular power.
float specularPower  = roughnessToSpecularPower( roughness );

// Depending on what BRDF used, convert to cone angle. Cone
// angle is maximum extent of the specular lobe aperture.
float coneTheta        = specularPowerToConeAngle(
    specularPower );

// Cone-trace using an isosceles triangle to approximate a cone
// in screen space
for(int i = 0; i < 7; ++i)
{
    // Intersection length is the adjacent side, get the opposite
    // side using trigonometry
    float oppositeLength   = isoscelesTriangleOpposite(
        adjacentLength, coneTheta );

    // Calculate in-radius of the isosceles triangle now
    float incircleSize     = isoscelesTriangleInradius(
        adjacentLength, oppositeLength );

    // Get the sample position in screen space
    float2 samplePos       = screenPos.xy + adjacentUnit *
        (adjacentLength - incircleSize);

    // Convert the in-radius into screen size (960x540) and then
    // check what power N we have to raise 2 to reach it.
    // That power N becomes our mip level to sample from.
    float mipChannel       = log2( incircleSize *
        max(screenSize.x, screenSize.y) );

    // Read color and accumulate it using trilinear filtering
    // (blending in xy and mip direction) and weight it.
    // Uses pre-convolved image and pre-integrated transparency
    // buffer and Hi-Z buffer. It checks if cone sphere is below,
    // in between, or above the Hi-Z minimum and maxamimum and
    // weights it together with transparency.
    // Visibility is accumulated in the alpha channel.
    totalColor += coneSampleWeightedColor( samplePos, mipChannel ↩
        );

    if( totalColor.a > 1.0f )
        break;

    // Calculate next smaller triangle that approximates the cone
    // in screen space.
    adjacentLength         = isoscelesTriangleNextAdjacent(
        adjacentLength, incircleSize );
}
```

Listing 4.7. Again the full implementation of some of the functions is not shown because of code length. The demo code available online has the full implementation; the demo also comes with alternative toggle-able code for accumulating the colors such as basic averaging, distance-based weighting, and hierarchical pre-integrated visibility buffer weighting.

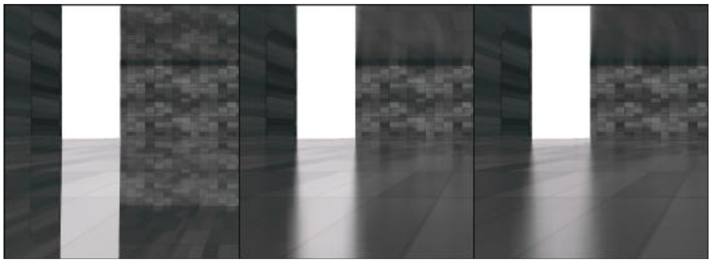


Figure 4.23. Cone-tracing algorithm capable of producing glossy reflections in screen space. Notice how the algorithm ensures that the further the reflection is from caster, the more spread it becomes like in the real world.

There is a conflict between the Hi-Z tracer and the cone tracer. Hi-Z tries to find a perfect specular as fast as possible while the cone tracer needs to take linear steps to integrate the total visibility in front-to-back order for correct occlusion.

This is not shown in this chapter because of complexity but the Hi-Z buffer is actually used together with the cone angle to find an early intersection at a coarse level to early exit out of the Hi-Z loop; then, we jump straight to the cone-tracing pass to continue with the linear stepping for glossy reflections. The Hi-Z functions as an empty-space determiner; once we have a coarse intersection with the cone, we can jump straight into the cone tracer to integrate the visibility and colors from that point onwards.

The more roughness we have on our surfaces, the cheaper this technique gets because we sample bigger circles and do larger jumps. Conversely, the less rough the surface is, the further the Hi-Z can travel for a perfect specular reflection so it all balances out evenly. Again, implementation is dependent on whether you use post-projected Hi-Z or view-space Hi-Z.

4.6 Extensions

4.6.1 Smoothly Fading Artifacts

We already talked about the inherent problems with screen-space local reflection in Section 4.2. Without further ado, rays traveling the opposite direction of the viewing ray and rays that fall close to the screen borders or outside the screen should be faded away due to lack of information available to us in the screen space. We can also fade rays based on ray travel distance.

A quick and simple implementation is shown in Listing 4.8. The demo ships with a more robust implementation.

One could fade away based on local occlusion as well, where a ray starts traveling behind an object and fails to find a proper intersection. One would

```
// Smoothly fade rays pointing towards the camera; screen space
// can't do mirrors (this is in view space).
float fadeOnMirror    = dot(viewReflect, viewDir);

// Smoothly fade rays that end up close to the screen edges.
float boundary      = distance(intersection.xy,
                           float2(0.5f, 0.5f) ) * 2.0f;
float fadeOnBorder  = 1.0f - saturate( (boundary  - FADE_START) /
    (FADE_END - FADE_START) );

// Smoothly fade rays after a certain distance (not in
// world space for simplicity but shoudl be).
float travelled   = distance(intersection.xy, startPos.xy);
float fadeOnTravel    = 1.0f - saturate( (travelled - FADE_START)
     / (FADE_END - FADE_START) );

// Fade the color now.
float3 finalColor = color * ( fadeOnBorder  * fadeOnTravel *
    fadeOnMirror );
```

Listing 4.8. Artifact removal snippet for fading the rays that have a high chance of failing and computing incorrect reflection results. `FADE_START` and `FADE_END` drive how quickly the fading should happen, where they are between 0 and 1. Though the code snippet shows the same parameters used for both the fading techniques, one should use different parameters and tweak them accordingly.

store when the ray entered such a state and then, depending on the distance traveled, fade the ray during that state to remove such unwanted artifacts.

4.6.2 Extrapolation of Surfaces

Since the ray-marching step might not find a true intersection, we could potentially extrapolate the missing information. Assuming the screen is covered with mostly rough surfaces with glossy reflections, we could run a bilateral dilation filter, which basically means take the surface normal and depth into account when extrapolating the missing color (i.e., flood-filling the holes). For any surface other than rough surfaces, the dilation filter might fail horribly because of potential high-frequency reflection colors.

One might be able to use a tile-based filter that finds good anchor points per tile and then run a clone brush filter to extrapolate the missing information for non-rough surfaces. The tile-based approach should work well for running the dilation only on the needed pixels.

4.6.3 Improving Ray-Marching Precision

If we use a nonlinear, post-projected depth buffer, most of the depth values fall very quickly into the range between 0.9 and 1.0, as we know. To improve the precision of the ray marching, we can reverse the floating-point depth buffer.

This is done by swapping the near and the far planes in the projection matrix, changing the depth testing to greater than and equal instead of lesser than and equal. Then, we could clear the depth buffer to black instead of white at each frame because 0.0 is where the far plane is now. This will turn 1.0 to the near plane in the depth buffer and 0.0 to the far plane. There are two nonlinearities here: one from the post-perspective depth and one from the floating point. Since we reversed one, they basically cancel each other out, giving us better distribution of the depth values.

Keep in mind that reversing the depth buffer affects our Hi-Z construction algorithm as well.

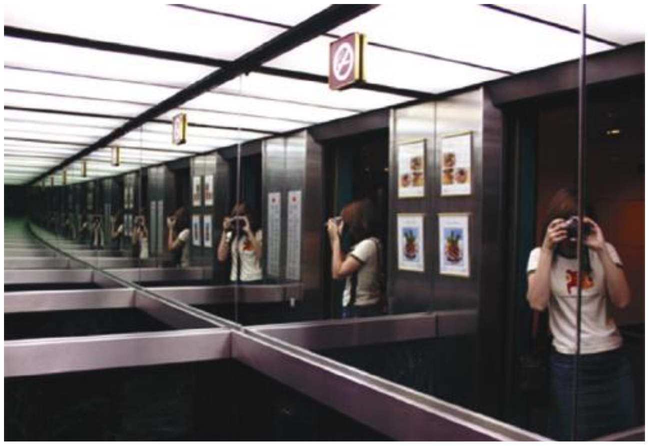
One should always use a 32-bit floating-point depth buffer; on AMD hardware the memory footprint of 24-bit and 32-bit depth buffers is the same, with which the fourth generation consoles are equipped also.

Another technique that can be used to improve depth precision is to actually create the Hi-Z buffer over a view-space Z depth buffer. We would need to output this in the geometry pass into a separate render target because recovering it from a post-perspective depth is not going to help the precision. This gives us uniformly distributed depth values. The only issue with a view-space Z depth buffer is that since it's not post-perspective, we can't interpolate it in screen space. To interpolate it we would have to employ the same technique as the hardware interpolator uses. We take 1/Z and interpolate it in screen space and then divide this interpolated value again by 1/Z' to recover the final interpolated view-space Z. However, outputting a dedicated linear view-space Z buffer might be too costly. We should test a reversed 32-bit floating-point depth buffer first. The cone-tracing calculations are also a bit different with a view-space Z buffer. We would need to project the sphere back into screen space to find the size it covers at a particular distance. There are compromises with each technique.

4.6.4 Approximate Multiple Ray Bounces

Multiple bounces are an important factor when it comes to realistic reflections. Our brain would instantly notice that something is wrong if a reflection of a mirror didn't have reflections itself but just a flat color. We can see the effect of multiple reflections in Figure 4.24.

The algorithm presented in this chapter has the nice property of being able to have multiple reflections relatively easily. The idea is to reflect an already reflected image. In this case the already reflected image would be the previous frame. If we compute the reflection of an already reflected image, we'll accumulate multiple bounces over time. (See Figure 4.25.) But since we always delay the source image by a frame, we'll have to do a re-projection of the pixels. To achieve this re-projection, we'll basically transform the current frame's pixel into the position it belonged to in the previous frame by taking the camera movement into account [Nehab et al. 07].



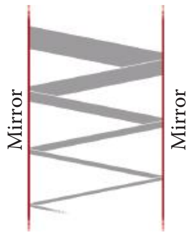


Figure 4.24. Infinite reflections when two mirrors are parallel against each other. Notice how the strength of the reflections decreases with the number of bounces we have due to absorption where the light is transferred to heat and some is reflected back again. The result is darker reflections at further ray depths. Notice the green tint as well that the ray accumulates over time from the glass as it bounces, due to iron oxide impurities in an ordinary soda-lime glass [Wikipedia 14]. The green tint is usually most noticeable on the edges of a glass. [Image courtesy of [Merry Monk 11].]

Once we know the position of where it belonged in the previous frame, we'll also need to detect if that pixel is valid or not. Some pixels might have moved outside the screen borders and some might have been blocked/occluded, etc. If the camera has moved drastically between the previous frame and the current frame, we might have to reject some of the pixels. The easiest way of doing this would be to store the previous frame's depth buffer; once we have done a re-projection of the current pixel into the previous frame, we just compare them by an epsilon and detect a fail or success. If they are not within an epsilon value we know the pixel is invalid. To get even more accurate results, one could also use the normal (surface orientation) of the previous frame and the normal of the re-projected pixel. The demo uses only the depth for rejection invalid pixels.

Mathematically speaking, we need to use the current inverted camera projection matrix and the previous camera projection matrix to take us from the current frame's pixel into the previous frame:

$$\mathbf{M}' = \mathbf{VP}_{\mathbf{curr}}^{\mathbf{-1}}\mathbf{VP}_{\mathbf{prev}},$$

where $\mathbf{M}'$ is the concatenated re-projection matrix, $\mathbf{VP}_{\mathbf{curr}}^{\mathbf{-1}}$ is the inverse view projection matrix from the current frame, and $\mathbf{VP}_{\mathbf{prev}}$ is the view projection matrix from the previous frame. When multiplied with a pixel $\mathbf{P}_n$ in clip space, this will take us to the corresponding pixel $\mathbf{P}_{n-1}$ in the previous frame in homoge-

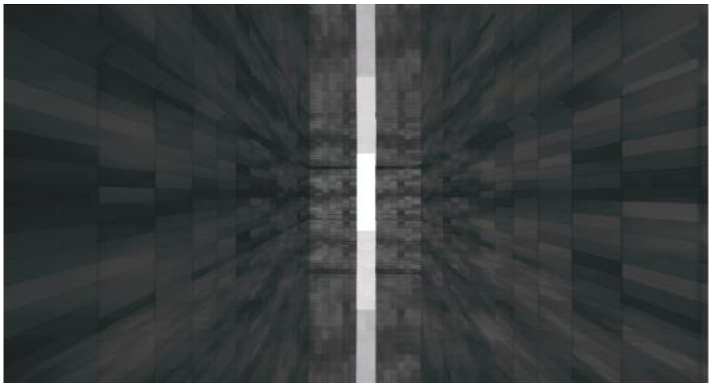


Figure 4.25. The effect of re-projecting an already reflected image and using it as a source for the current frame. This produces multiple reflections and just like in the real world the reflections lose intensity the further in reflection depth it gets.

nous space. We just need to divide the result with the *w* component to finally get the clip-space coordinate. Then we can just map it into screen space and start reading from the previous frame color buffer and thereby have an infinite number of reflection bounces. Figure 4.26 and Listing 4.9 show the concept of un-projecting and re-projecting a pixel into the previous camera's pixel position.

Another benefit of using the previous frame is the fact that we are taking the final lighting and all transparent object information into account as well as the possibility of including post-processing effects that have been applied to the image. If we would have used the current unfinished frame, we would lack all of those nice additions—though not all post-process effects are interesting.

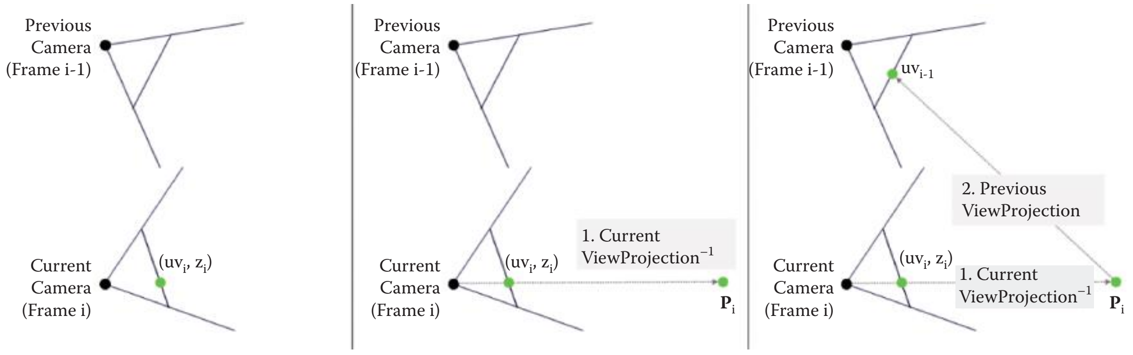

Previous Camera (Frame i-1)
Current Camera (Frame i)
(uv_i, z_i)
Previous Camera (Frame i-1)
Current Camera (Frame i)
(uv_i, z_i)
1. Current ViewProjection^{-1}
P_i
Previous Camera (Frame i-1)
uv_{i-1}
2. Previous ViewProjection
Current Camera (Frame i)
(uv_i, z_i)
1. Current ViewProjection^{-1}
P_i

Figure 4.26. Illustration of how a pixel in screen space is transformed into its old coordinate by re-projection, which can be used to read from the previous color buffer.

```
float2 texcoords = input.tex.xy;

// Unpack clip position from texcoords and depth.
float depth          = depthBuffer.SampleLevel(pointSampler,
    texcoords, 0.0f);
float4 currClip      = unpackClipPos(texcoords, depth);

// Unpack into previous homogenous coordinates:
// inverse(view projection) * previous(view projection).
float4 prevHomogenous = mul(currClip,
    invViewProjPrevViewProjMatrix);

// Unpack homogenous coordinate into clip-space coordinate.
float4 prevClip       = float4( prevHomogenous.xyz /
    prevHomogenous.w, 1.0f);

// Unpack into screen coordinate [-1, 1] into [0, 1] range and
// flip the y coordinate.
float3 prevScreen     = float3(prevClip.xy * float2(0.5f, -0.5f)
    + float2(0.5f, 0.5f), prevClip.z);

// Return the corresponding color from the previous frame.
return prevColorBuffer.SampleLevel(linearSampler, prevScreen.xy,
    0.0f);
```

Listing 4.9. Re-projecting a pixel into its previous location in the previous frame's color image. Implementation details for rejecting pixels are omitted. Demo comes with the full code.

If you have a motion blur velocity vector pass, more specifically 2D instantaneous velocity buffer, you can use that instead of re-projecting with the code above. Using 2D instantaneous velocity is more stable but that is beyond the topic of this chapter.

4.6.5 Temporal Filtering

Temporal filtering is another enhancer that helps the algorithm to produce even more accurate and stable results by trying to recover and reuse pixels over several frames, hence the name *temporal*, over time. The idea is to have a history buffer that stores the old reflection computation, and then we run a re-projection pass over it just like we saw in Section 4.6.4, and reject any invalid pixels. This history buffer is the same buffer we write our final reflection computation to, so it acts like an accumulation buffer where we keep accumulating valid reflection colors. In case the ray-marching phase fails to find a proper intersection, due to the ray falling behind an object or outside of the screen, we can just rely on the previous re-projected result that is already stored in the history buffer and have a chance of recovering that missing pixel.

Temporal filtering helps stabilize the result because a failed pixel in frame N due to occlusion or missing information might be recovered from frame $N-1$, which was accumulated over several frames by re-projection of pixels. Having the possibility of recovering pixels that were valid in the previous frames but invalid in the current frame is essential and is going to give much better results and stabilize the algorithm. It's also possible to get huge speedups by not running the ray-marching code if the recovery is accurate enough so that we don't need to recalculate the reflection at all.

This little enhancer walks hand in hand with Section 4.6.2, "Multiple Ray Bounces," since both rely on re-projection.

4.6.6 Travel Behind Surfaces

It is possible to travel behind objects using a Minimum and a Maximum Hi-Z buffer. In case the ray crosses a cell and ends up behind both the minimum and the maximum depth of the new cell, we can just cross that cell as well and continue with the Hi-Z traversal. This assumes that the surface has not infinitely long depth. A global small epsilon value can also be used, or a per object thickness epsilon value into a render target. Traveling behind objects is really a hard problem to solve if we do not have information on object thickness.

4.6.7 Ray Marching Toward the Camera

We've looked at the algorithm using a Minimum Hi-Z hierarchy for rays to travel away from the camera. It's also possible for mirror-like reflected rays to travel toward the camera and thereby have a chance of hitting something, though this chance is very small and would mostly benefit curved surfaces. A small change is required for the algorithm, which makes use of an additional hierarchy, the Maximum Hi-Z, for any ray that would want to travel toward the camera.

A texture format such as R32G32F would be appropriate for this change, the R channel would store the minimum and the G channel would store the maximum.

There is a very small amount of pixels that have a chance of actually hitting something, so this change might not be worth it as this would add an overhead to the entire algorithm.

4.6.8 Vertically Stretched Anisotropic Reflections

In the real world the more grazing angles we see on glossy surfaces, the more anisotropic reflections we perceive. Essentially the reflection vectors within the reflection lobe are spread more vertically than horizontally, which is the main reason why we get the vertical stretching effect.

To achieve this phenomenon with screen-space reflections, we have to use the `SampleGrad` function of our color texture during the cone-tracing accumulation pass of the reflection colors, give this sampler hardware some custom calculated vertical and horizontal partial derivatives, and let the hardware kick the anisotropic filterer to stretch the reflections for us. This is highly dependent on the BRDF used and how roughness values map to the reflection lobe.

We could also manually take multiple samples to achieve the same result. Basically, instead of sampling quads, we sample elongated rectangles at grazing angles.

We saw earlier in Section 4.4.5 that for complicated BRDF models we would need to pre-compute a 2D table of local reflection vectors and cone angles. A texture suited for this is R16G16B16A16. The RGB channels would store the local vector and the alpha channel would store either one isotropic cone-angle extent or two anisotropic vertical and horizontal cone-angle extents. These two anisotropic values for the cone would decide how many extra samples we would take vertically to approximate an elongated rectangle to stretch the reflections.

4.7 Optimizations

4.7.1 Combining Linear and Hi-Z Traversal

One drawback of the Hierarchical-Z traversal is that it is going to traverse down to lower hierarchy levels when the ray travels close to a surface. Evaluating the entire Hierarchical-Z traversal algorithm for such small steps is more expensive than doing a simple linear search with the same step size. Unfortunately the ray starts immediately close to a surface, the surface we are reflecting the original ray from. Doing a few steps of linear search in the beginning seems to be a great optimization to get the ray away from the surface and then let the Hierarchical-Z traversal algorithm do its job of taking the big steps.

In case the linear search finds intersections, we can just early-out in the shader code with a dynamic branch and skip the entire Hi-Z traversal phase. It's also worth it to end the Hi-Z traversal at a much earlier level such as 1 or 2 and then continue with another linear search in the end. The ending level could be calculated depending on the distance to the camera, since the farther away the pixel is, the less detail it needs because of perspective, so stopping much earlier is going to give a boost in performance.

4.7.2 Improving Fetch Latency

Partially unrolling dynamic loops to handle dependent texture fetches tends to improve performance with fetch/latency-bound algorithms. So, instead of handling one work per thread, we would actually pre-fetch the work for the next N loops. We can do this because we have a deterministic path on our ray. However, there is a point where pre-fetching starts to hurt performance because the register usage rises and using more registers means less buckets of threads can run in parallel. A good starting point is $N = 4$. That value was used on a regular linear tracing algorithm and a speedup of $2\times$–$3\times$ was measured on both NVIDIA and AMD hardware. The numbers appearing later in this chapter do not include these improvements because it wasn't tested on a Hi-Z tracer.

4.7.3 Interleaved Spatial Coherence and Signal Reconstruction

Because most of our rays are spatially coherent, we can shoot rays every other pixel—so-called interleaved sampling—and then apply some sort of signal reconstruction filter from sampling theory. This works very well with rough reflections since the result tends to have low frequency, which is ideal for signal reconstruction. This was tested on linear tracing-based algorithms, and a performance increase of about 3×–4× was achieved. The interleaving pattern was twice horizontally, twice vertically. These improvements were also not tested on a Hi-Z tracer, so the numbers presented later do not include these either.

4.7.4 Cross-Bilateral Upsampling or Temporal Super-Sampling

Since we run the ray tracing at half-resolution, we do need a smart upsampling scheme to make up for the low number of pixels. A cross-bilateral image upsampling algorithm is a perfect fit for this kind of a task [Kopf et al. 07], but a temporal super-sampling algorithm is even better: after four frames we will have full-resolution traced results using the temporal re-projection that was explained earlier.

For the cross-bilateral upsampler, The full-resolution depth buffer would be an input together with the half-resolution reflection color buffer. The algorithm would upsample the reflection color buffer to full resolution while preserving silhouettes and hard edges. It's way faster and cheaper to calculate the reflections at half-resolution than full-resolution. However, to recompose the image back to the original screen, at full-resolution, we need to scale it up while preserving the hard edges, and that's exactly what the Cross-Bilateral Upsampling algorithm is good for.

While upsampling one could also use another approach and append the pixels at depth discontinuities to an append/consume buffer and re-trace only those pixels at high resolution later for higher quality. This was not tested.

4.8 Performance

The demo runs at half-resolution, meaning 960×540, and it's running super-fast:

- 0.35–0.39 ms on NVidia GTX TITAN,
- 0.70–0.80 ms on NVidia GTX 670,
- 0.80–0.90 ms on AMD 7950.

The timers are the Hi-Z Ray-Marching and Cone-Tracing combined.

The demo is memory latency bound, and the memory unit is 80–90% active, which gives little to no room for our ALU units to work because they just sit there waiting for a fetch to complete.

According to GPU PerfStudio 2, we are having 50% cache misses because of nonlocal texture buffer accesses when traversing using the Hi-Z acceleration structure and we also suffer from noncoherent dynamic branching since a GPU executes branches in lock-step mode. If an entire bucket of threads (group of 32 threads for Nvidia called Warp, 64 for AMD called Wavefront) does not take the same branch, then we pay the penalty of stalling some threads until they converge again into the same path. This gets worse as the threads keep taking different branches for some pixels.

One optimization that was not tried, but mentioned by [Tevs et al. 08], is using a 3D texture to store the Hi-Z instead of a 2D texture. According to [Tevs et al. 08], using a 3D texture for a displacement mapping technique, where each slice represents the hierarchy levels of our Hi-Z, gives better cache hits and a performance boost of 20% due to less L2 traffic and more texture cache hits.

Since we are memory latency bound due to cache misses and incoherent texture accesses, while jumping up and down in the hierarchy, this might be a good optimization to try, though it would use much more memory.

4.9 Results

The presented algorithm works really well and produces great reflections, both specular and glossy, and it runs at easily affordable speeds for games. The most noticeable detail is the spread of reflections as they get farther away from the source, which is the selling point of the entire algorithm. (See Figures 4.27 and 4.28.)

4.10 Conclusion

In this chapter we looked at Hi-Z Screen-Space Cone Tracing to compute both specular and glossy reflections at game interactive frame rates and performance

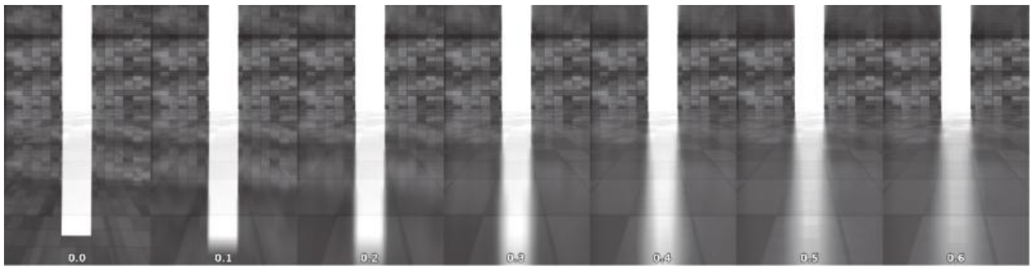


Figure 4.27. Cone-tracing algorithm with different level of glossiness on the tile material, giving the appearance of diverged reflection rays. The reflection becomes more spread the farther away it is, and it is stretching just like the phenomena we see in the real world.

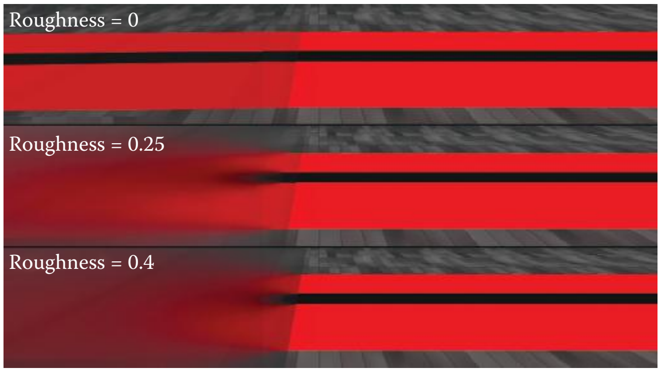


Figure 4.28. Another example of Hi-Z Screen-Space Cone Tracing producing spread reflections the farther it travels due to micro fracture simulation using material roughness.

levels. While the algorithm works well for local reflections, there are some edge cases where it may fail because of insufficient information on the screen. This algorithm can only reflect what the original input image has, as we have seen. You will not be able to look at yourself in the mirror since that information is not available to us in screen space. Hi-Z Screen-Space Cone Tracing is more of a supplementary effect for dynamic reflections in dynamic 3D scenes with the cheap glossy appearance on them, and it's recommended that you combine it with other types of reflection techniques such as local box projected [Behc 10] or sphere projected [Bjorke 07] cube-maps to take over when Hi-Z Screen-Space Cone Tracing fails as a backup plan. Hi-Z Screen-Space Cone Tracing should not be used on its own as a single solution because of the inherent problems the screen-space algorithm has, unless you have a very specific controlled scene with a specific camera angle where you can avoid the problems to begin with, such as flat walls and no mirrors, etc. The glossy reflections help hide artifacts that are otherwise visible as mirror reflections.

4.11 Future Work

The system could be extended in many ways. One idea is to take a screenshot of a 3D scene and store the color and depth information in conjunction with the camera basis. With this we could at run time re-project this local screenshot without any dynamic objects obscuring interesting information from us. The screenshots would act like local reflection probes, and we would pick the closest interesting one and do our Hi-Z traversal.

The Hi-Z Screen-Space Cone-Tracing technique could also be applied on cube-maps, where we construct a cube-mapped Hi-Z acceleration structure and ray-march within this cube volume. This would allow us to reflect anything outside the screen as well with pixel perfect ray-tracing-like results. This technique, Hi-Z Cube-Map Ray Tracing, is ongoing research and it will be published at a later time.

Another expansion could be packet tracing. This is ongoing research also and will be published at a later time. The basic idea is to group several rays together and basically shoot a packet of rays, like a frustum. As we intersect the coarse Hi-Z, we can refine/subdivide the packet and shoot several more rays. This way we can quickly intersect coarse Hi-Z levels and then do fewer operations as we travel, though this makes the implementation more complex and harder to maintain and relies heavily on compute shaders. Grouping coherent rays should give an excellent boost in performance.

One could also do tiled tracing, where each tile identifies the roughness value of the scene. In case the entire tile is very rough, we can probably get away with shooting fewer rays and extrapolating most of the pixel colors. This multi-resolution handling should also give an excellent boost in speed, though again this also relies heavily on compute shaders.

All of those topics are ongoing research and will be, as said before, published at a later time if it makes it out of the research and development phase.

4.12 Acknowledgments

I would like to give my special thanks to my co-workers who supported me at EA DICE—Mikael Uddholm, Rendering Engineer of *Mirror's Edge*, and Charles de Rousiers, Rendering Engineer of *Frostbite*—and to my leads for proofreading this chapter, also at EA DICE—Jonas Åberg, Technical Director of *Mirror's Edge*, and Daniel Johansson, Lead Software Engineer of *Mirror's Edge*.

I would also like to give my special thanks to the editors of *GPU Pro 5*—Michal Valient of Guerilla Games and Wolfgang Engel of Confetti FX—and to project editor Charlotte Byrnes of CRC Press.

Bibliography

[Autodesk 09] Autodesk. "General Utility Shaders." *mental ray Architectural and Design Visualization Shader Library*, http://download.autodesk.com/us/maya/2011help/mr/shaders/architectural/arch_util.html, 2009.

[Behc 10] Behc. "Box Projected Cubemap Environment Mapping." http://www.gamedev.net/topic/568829-box-projected-cubemap-environment-mapping/, 2010.

[Bjorke 07] Kevin Bjorke. "Image-Based Lighting." In *GPU Gems*, edited by Randima Fernando, Chapter 19. Upper Saddle River, NJ: Addison-Wesley, 2007.