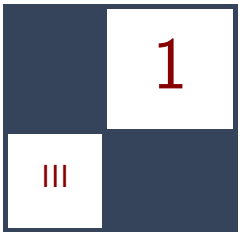

The Skylanders SWAP Force Depth-of-Field Shader

Michael Bukowski, Padraic Hennessy,
Brian Osman, and Morgan McGuire

1.1 Introduction

This chapter describes the depth-of-field (DoF) shader used in production at Vicarious Visions for the *Skylanders SWAP Force* game on multiple console platforms.

DoF is an important rendering effect. A real camera lens focuses on a single plane in the scene. Images of objects in that plane are perfectly sharp. Images of objects closer to or farther from the camera are blurry. Of course, most objects are typically outside the exact plane of focus. Photographers recognize that each point on an out-of-focus object blurs into the shape of the camera aperture, which is usually a disk, octagon, or hexagon. They call the bounding circle of a blurred point the *circle of confusion* (CoC) of that point. They say that an object is in focus (for a digital image) when the *radius* of the CoC is half a pixel or less. In that case, the object does not appear blurred because the blur falls below the image resolution. Photographers refer to the depth range over which the CoC radius is less than half a pixel as the *focus field*. They refer to the extent of this range as the *depth of field*. In computer graphics, that phrase is now associated with the effect of blurring images of objects outside of the field.

In a game, a DoF effect serves both gameplay and aesthetics. It allows the art director to control the player's attention by de-emphasizing background objects or those that are merely in the foreground to frame the shot. Rack-focus (Figure 1.1) in on a specific object can emphasize goals and powerups during gameplay without resorting to floating arrows or halos. In cut-scenes, DoF is a powerful cinematic tool. DoF also conceals many rendering limitations. Defocusing the background conceals aliasing and low level of detail. Defocusing the extreme foreground conceals texture magnification and tessellation limits.

Figure 1.1. Three frames from a cinematic "rack focus" transition designed to move the player's attention from the background to the extreme foreground. In each shot, the yellow dot shows the location of camera focus.

True DoF arises because each point on the camera lens has a slightly different viewpoint and the final image is a composite of images from all of them. Research papers have simulated this brute-force rendering of multiple viewpoints to an accumulation buffer, sampling viewpoints with distribution ray tracing [Cook et al. 84], and sampling with stochastic rasterization [McGuire et al. 10]. These methods are all too expensive to be practical today and are overkill for achieving a convincing effect. Since the goal is to blur parts of the image, we need not render a perfect result. It should be enough to selectively apply some post-processing blur filters to a typical frame. Like many other game engines, ours follows this approach. There's a good argument for this approximation over physically correct solutions: it is what Photoshop's Lens Blur and many film editing packages do. The DoF seen in advertisements and feature films is certainly of sufficient quality for game graphics.

We distinguish three depth ranges of interest: the *far field* in which objects are blurry because they are too far away, the *focus field* where objects are in focus, and the *near field* in which objects are blurry because they are too close to the camera.

A post-processing DoF shader is essentially a blur filter with a spatially varying kernel. The way that the kernel varies poses three challenges. First, unlike a typical Gaussian or box blur, DoF blur must respect depth edges. For example, the effect should not let a blurry distant object bleed over a nearer sharp object. We observe that it is most critical that the far-focus-near ordering of occlusion be preserved. However, incorrect blurring of images of neighboring objects *within* a field is often undetectable.

Second, when preventing bleeding, the effect must also not create sharp silhouettes on blurry objects in the near field. These sharp silhouettes are the primary visual artifact in previous methods.

Third, foreground objects have to be able to blur fairly far (up to 10% of the screen width) without compromising performance. The naïve approaches of scattering each pixel as a disk and the inverse of performing the equivalent gather operation are too slow—those methods require $O(r^2)$ operations for blur radius r and thrash the texture/L1 cache.

The DoF post-process in our engine is fast and produces good-quality near- and far-field blurring with little perceptible color bleeding. It reads a color buffer with a specially encoded alpha channel and produces a convincing DoF effect in three "full-screen" 2D passes over various size buffers. It uses 1.9 ms of GPU time running at 720p on the Xbox 360. On a PC it uses 1.0 ms of GPU time running at 1080p on a GeForce GTX 680.

We developed our DoF effect from Gillham's *ShaderX*^{5} one [Gillham 07]. Like his and other similar techniques [Riguer et al. 03, Scheuermann 04, Hammon 07, Kaplanyan 10, Kasyan et al. 11], we work with low-resolution buffers when they are blurry and lerp between blurred and sharp versions of the screen. The elements of our improvements to previous methods are

- treating near field separately to produce blurry silhouettes on the near field,
- inpainting behind blurry near-field objects,
- a selective background blur kernel,
- using CoC instead of depth to resolve occlusion and blur simultaneously,
- processing multiple blurs in parallel with dual render targets.

1.2 Algorithm

Figure 1.3 shows the structure of our algorithm. The input is a color buffer with the (scaled and biased) *signed CoC radius* stored in the alpha channel. Two passes blur horizontally and then vertically in a typical separated blur pattern, and a final pass composites the blurred image over the sharp input. Each blur pass processes two textures: one that represents the focus and far field, and one that represents objects in the near field (with an alpha channel for coverage).

1.2.1 Input

The radius in the alpha channel of the color input buffer is signed, meaning that radius r at each pixel is on the range [–maximum blur, +maximum blur]. Far-field objects have a negative radius, near-field objects have a positive one, and $0.5 < r < 0.5$ in the focus field. Under a physically correct CoC model, this signed radius naturally arises. There, it models the fact that the silhouette of the aperture appears inverted in the far field. That inversion is irrelevant for the disc

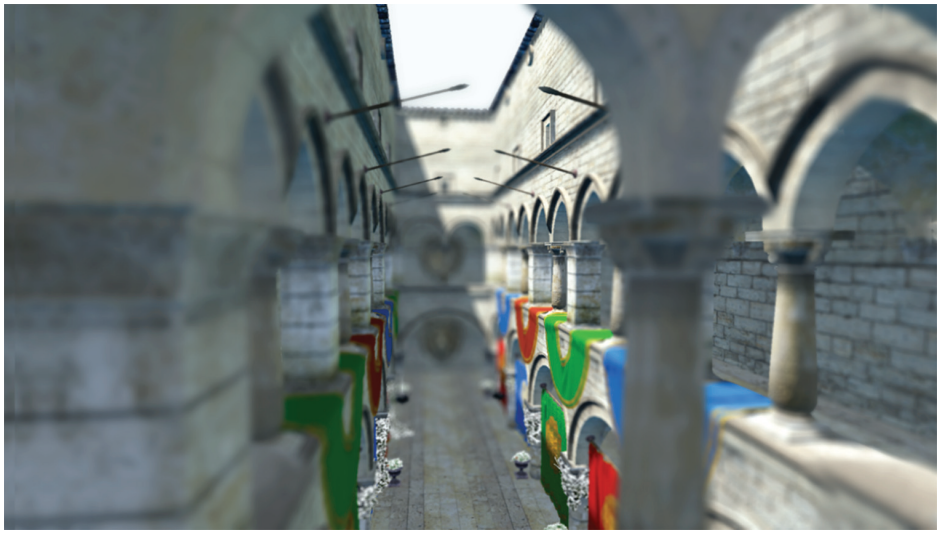

Figure 1.2. Extreme near- and far-field defocus with smooth transitions rendered by our algorithm. Note the blurry silhouettes on near-field objects and detail inpainted behind them.

aperture that we model, but we depend on the signed radius for another reason. Signed radius decreases monotonically with depth, so if $r_A < r_B$, then point A is closer to the camera than point B. Thus the single signed radius value at each pixel avoids the need for separate values to encode the field, radius, and depth of a point.

Our demo supports two methods to compute the signed radius. The first is the physically correct model derived from Figure 1.4. Let a be the radius of the lens, $z_\mathrm{F} < 0$ be the depth of the focus plane, and R be the world-space (versus screen-space) radius. By similar triangles, the screen-space radius r for a point at depth z is

$$\frac{R}{|z_\mathrm{F} - z|} = \frac{a}{|z_\mathrm{F}|}, \quad r \propto \mathrm{a}\frac{|z_\mathrm{F} - z|}{z_\mathrm{F} \cdot z}.$$

The proportionality constant depends on screen resolution and field of view. Our art team felt that this physical model gave poor control over the specific kinds of shots that they were trying to direct. They preferred the second model, in which the artists manually place four planes (near-blurry, near-sharp, far-sharp, far-blurry). For the near- and far-blurry planes, the artists specify the CoC radius explicitly. At the near-sharp and far-sharp planes, the radius is $1/2$ pixel. The CoC at depths between the planes is then linearly interpolated. Depths closer than the near-blurry and farther than the far-blurry have radii clamped to the values at those planes.

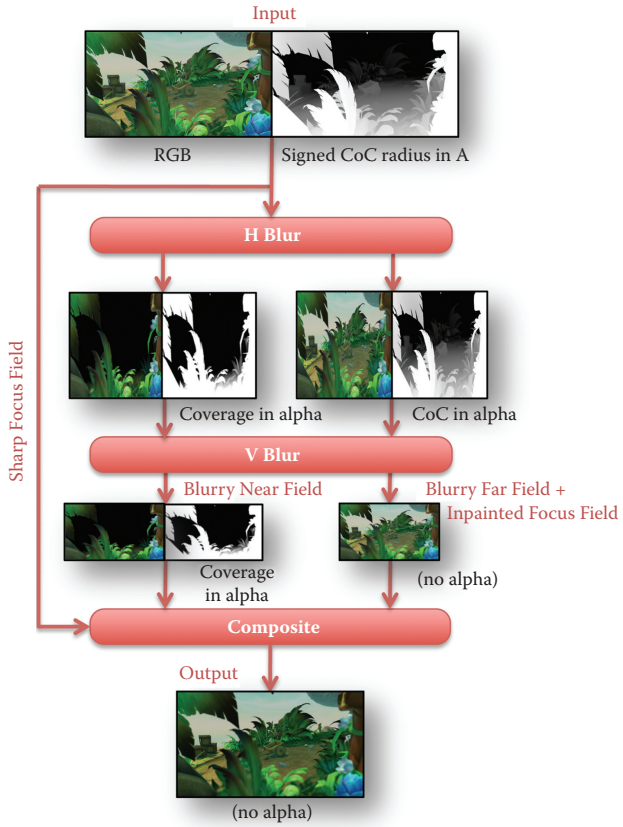

Figure 1.3. Diagram of shading passes with buffers shown. See the supplemental material for high-resolution images.

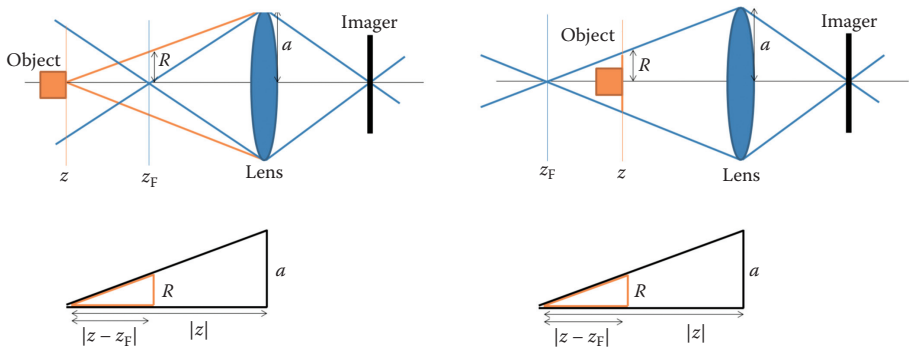

Figure 1.4. The geometry of lens blur. Left: the orange cube is in the far field and out of focus. Right: the orange cube in the near field and out of focus. The radius of the CoC (in camera space) is given by using similar triangles to find R in each case.

1.2.2 Blur Passes

The two blur passes are similar to one another. Each reads a fixed-length set of adjacent samples, either horizontal or vertical, and computes two weighted sums. The key outer loop section of the code is shown below.

```
// Accumulates the blurry image color
blurResult.rgb = float3(0.0f);
float blurWeightSum = 0.0f;

// Accumulates the near-field color and coverage
nearResult = float4(0.0f);
float nearWeightSum = 0.0f;

// Location of the central filter tap (i.e., "this" pixel's location)
// Account for the scaling down by 50% during blur
int2 A = int2(gl_FragCoord.xy) * (direction + ivec2(1));

float packedA = texelFetch(blurSourceBuffer, A, 0).a;
float r_A = (packedA * 2.0 - 1.0) * maxCoCRadiusPixels;

// Map large negative r_A to 0.0 and large positive r_A to 1.0
float nearFieldness_A = saturate(r_A * 4.0);

for (int delta = -maxCoCRadiusPixels; delta <= maxCoCRadiusPixels;
                                                ++delta) {
   // Tap location near A
   int2 B = A + (direction * delta);

   // Packed values
   float4 blurInput = texelFetch(blurSourceBuffer, clamp(B, int2(0),
                        textureSize(blurSourceBuffer, 0)
                         - int2(1)), 0);

   // Signed kernel radius at this tap, in pixels
   float r_B = (blurInput.a * 2.0 - 1.0) * float(maxCoCRadiusPixels);

    // [Compute blurry buffer]
    ...

    // [Compute near-field super blurry buffer and coverage]
    ....
}

blurResult.a = packedA;
blurResult.rgb /= blurWeightSum;
nearResult /= nearWeightSum;
```

The details of the two blur kernel sections follow. See also our demo source code, which contains extensive comments explaining optimizations and alternative implementations.

Let A be the center sample of the kernel and B be a nearby sample (note that $B = A$ is included in the set of samples that we consider). We compute the weight of sample B as follows.

```
If A is in the near field:
    // Inpaint behind A using some arbitrary constant weight k ≈ 1.
     w_B = k
    else if B is not in the near field:
    // Obey occlusion; note that both r values are always negative in this case.
      w_B = max(0, min(1, |r_A − r_B + 1.5|)) · Gaussian (BA)
    else:
    // Avoid divide-by-zero if every sample pair hits this case.
     w_B = ε
```

In practice, we smooth the transitions by implementing the branches with lerps. The relevant section of VVDoF_blur.glsl in our demo source code is

```
float weight = 0.0;

float wNormal =
    // Only consider mid- or background pixels (allows inpainting of the
    // near field).
    float(! inNearField(r_B)) *

    // Only blur B over A if B is closer to the viewer (allow 0.5 pixels
    // of slop and smooth the transition).
    saturate(abs(r_A) - abs(r_B) + 1.5) *

    // Stretch the Gaussian extent to the radius at pixel B.
    gaussian[clamp(int(float(abs(delta) * (GAUSSIAN_TAPS - 1)) /
            (0.001 + abs(r_B * 0.5))), 0, GAUSSIAN_TAPS)];

weight = lerp(wNormal, 1.0, nearFieldness_A);
// far- + mid-field output
blurWeightSum += weight;
blurResult.rgb += blurInput.rgb * weight;
```

We compute the coverage value (alpha) for the separate near-field buffer in the horizontal pass as

$$\alpha_B = \begin{cases} \min\left(1, \dfrac{r_B}{\text{maximum near-field blur}}\right)^4 & \text{if } |A - B| < r_B, \\ 0 & \text{otherwise;} \end{cases}$$

in code, this is somewhat more verbose:

```
float4 nearInput;
#if HORIZONTAL
   nearInput.a = float(abs(delta) <= r_B) *
        saturate(r_B * invNearBlurRadiusPixels * 4.0);
   nearInput.a *= nearInput.a; nearInput.a *= nearInput.a;

   // Compute premultiplied-alpha color.
   nearInput.rgb = blurInput.rgb * nearInput.a;

#else
```

```
    // On the second pass, use the already-available alpha values.
     nearInput = texelFetch(nearSourceBuffer, clamp(B, int2(0),
                 textureSize(nearSourceBuffer, 0) - int2(1)), 0);
#endif

    weight = float(abs(delta) < nearBlurRadiusPixels);
    nearResult += nearInput * weight;
    nearWeightSum += weight;
}
```

We empirically tuned this coverage falloff curve to provide good coverage when the near field is extremely blurry and to fade smoothly into the focus field. For example, in Figure 1.5, the out-of-focus fence in the near field must have sufficient coverage to smear white pixels over a large region, while we still want the transition of the ground plane into the focus field to look like gradual focusing and not simply a lerp between separate blurry and sharp images. This is an extremely hard case for a post-processing DoF algorithm that previous real-time methods do not handle well.

The near-field buffer is written with premultiplied alpha values for the color channel, and the color and alpha are both blurred during the subsequent vertical pass.

1.2.3 Compositing

The compositing pass (shown below) reads the original input buffer along with the low-resolution blurry near- and far-field buffers. It interpolates pixels between the original input and the blurred, inpainted far-field buffer based on the CoC at each

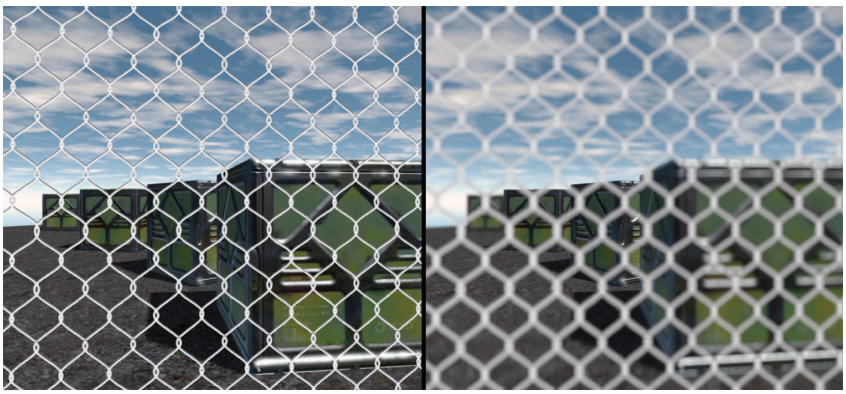

Figure 1.5. Input image with a chain-link fence very close to the camera (left), and near field under extreme blur with inpainted details visible through the "solid" parts of the fence and a smooth ground-plane transition between depth regions (right).

pixel. It then blends the near-field buffer over that result with premultiplied alpha blending. Near-field pixels exhibit inpainted detail from the far-field buffer and existing detail from the input buffer, both of which then receive the significantly blurred near-field content over them. Far-field pixels are blurry from the far-field image, the focus field is sharp and from the original image, and all transition regions are smooth because of the lerp.

```
uniform sampler2D packedBuffer;
uniform sampler2D blurBuffer;
uniform sampler2D nearBuffer;

out vec3 result;

const float coverageBoost = 1.5;

float grayscale(float3 c) {
   return (c.r + c.g + c.b) / 3.0;
}

void main() {
  int2 A = int2(gl_FragCoord.xy);

  float4 pack = texelFetch(packedBuffer, A, 0);
  float3 sharp = pack.rgb;
  float3 blurred = texture(blurBuffer,
           gl_FragCoord.xy / textureSize(packedBuffer, 0));
  float4 near = texture(nearBuffer,
           gl_FragCoord.xy / textureSize(packedBuffer, 0));

  // Normalize radius.
  float normRadius = (pack.a * 2.0 - 1.0);

  if (coverageBoost != 1.0) {\{}
      float a = saturate(coverageBoost * near.a);
      near.rgb = near.rgb * (a / max(near.a, 0.001f));
      near.a = a;
  }

  // Decrease sharp image's contribution rapidly in the near field.
  if (normRadius > 0.1) {\{}
      normRadius = min(normRadius * 1.5, 1.0);
  }

  result = lerp(sharp, blurred, abs(normRadius)) * (1.0 - near.a)
         + near.rgb;
}
```

The effect is extremely robust to camera and object movement and varying blur radii, independent of the scene. It should be tuned for two application-specific cases: transitions from mid to near depending on the field of view, and objects that don't write the depth buffer.

A compile-time constant, `coverageBoost`, allows increasing the partial coverage (alpha) of the near field to make it feel more substantial. This should always be greater than or equal to 1. If the near-field objects seem too transparent, then

increase `coverageBoost`. If an obvious transition line is visible between the blurred near-field region and the sharp mid-field, then decrease the `coverageBoost`. Which of these cases an application is in largely depends on whether the field of view makes the ground plane visible within this transition region. For example, a first-person camera typically cannot see the ground plane in this region but a third-person camera often can. The third-person camera benefits from a smaller `coverageBoost` setting.

Because the effect assumes a single depth at each pixel in the input, we process particle systems separately by MIP-biasing their textures during a forward rendering pass rather than relying on the post-processing. For non-particle, translucent and reflective objects such as glass, we simply choose to use the depth of the translucent object or the background depending on the amount of translucency.

1.3 Conclusion

We knew that depth of field was an essential effect for the art direction of *Skylanders SWAP Force*, where the visuals resemble a CG animated film more than a traditional video game. By addressing the perception of the phenomenon of blurring instead of the underlying physics, we were able to achieve both high quality and high performance on a range of target platforms.

The primary limitations of previous real-time depth-of-field approaches are poor near-field blur and poor transitions between blurred and sharp regions. Figure 1.2 shows that even under a narrow depth of field, our effect overcomes both of those limitations. The interaction of depth of field with translucent surfaces remains problematic in the general case; however, we've described the forward-rendering techniques that we applied successfully to such surfaces in this specific game.

It is important for game graphics to serve game design for engagement as well as to please the eye. In this game, we've found depth of field to be a powerful tool for both gameplay and cinematic expression. Designers employ it for controlling attention and indicating gameplay elements as well as the artists using it to enhance visuals and mitigate certain artifacts. We hold this effect as an example of a technological advance serving to enhance all aspects of the player's experience.

Bibliography

[Cook et al. 84] Robert L. Cook, Thomas Porter, and Loren Carpenter. "Distributed Ray Tracing." In *SIGGRAPH '84: Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 137–145. New York: ACM, 1984.

[Gillham 07] David Gillham. "Real-Time Depth-of-Field Implemented with a Postprocessing-Only Technique. In *ShaderX^5: Advanced Rendering Techniques*, edited by Wolfgang Engel, pp. 163–175. Boston: Charles River Media, 2007.

[Hammon 07] Earl Hammon, Jr. "Practical Post-Process Depth of Field." In *GPU Gems 3*, edited by Hubert Nguyen, Chapter 28. Upper Saddle River, NJ: Addison-Wesley, 2007.

[Kaplanyan 10] Anon Kaplanyan. "CryENGINE 3: Reaching the Speed of Light." Talk, SIGGRAPH 2010, Los Angeles, CA, July 28, 2010. (Available at http://www.crytek.com/sites/default/files/AdvRTRend_crytek_0.ppt.)

[Kasyan et al. 11] Nickolay Kasyan, Nicolas Schulz, and Tiago Sousa. "Secrets of CryENGINE 3 Graphics Technology." SIGGRAPH Course, Vancouver, Canada, August 8, 2011. (Available at http://www.crytek.com/sites/default/files/S2011_SecretsCryENGINE3Tech_0.ppt.)

[McGuire et al. 10] Morgan McGuire, Eric Enderton, Peter Shirley, and David Luebke. "Real-Time Stochastic Rasterization on Conventional GPU Architectures." In *Proceedings of teh Converence on High Performance Graphics*, pp. 173–182. Aire-la-Ville, Switzerland: Eurographics, 2010.

[Riguer et al. 03] Guennadi Riguer, Natalya Tatarchuk, and John Isidoro. "Real-Time Depth-of-Field Simulation." In *ShaderX^2: Shader Programming Tips and Tricks with DirectX 9.0*, edited by Wolfgang Engel, pp. 529–556. Plano, TX: Wordware Publishing, Inc., 2003

[Scheuermann 04] Thorsten Scheuermann. "Advanced Depth of Field." Presentation, Game Developers Conference 2004, San Francisco, CA, 2004. (Available at http://www.amddevcentral.com/media/gpu_assets/Scheuermann_DepthOfField.pdf.)