4.2

Adaptive Glare

Tiago Sousa

Introduction

As indicated by the number of image-space articles in this book and its predecessors, scene post-processing effects are fast becoming commonplace in games. Glare effects, also referred to as *glow* or *bloom*, are achieved through image post-processing techniques in order to produce a color bleeding effect around the brightest pixels in a scene. This is both an interesting stylistic element and a subtle but important cue to the user that conveys information about the relative brightness of different areas of an image. The technique described in this article is implemented in Crytek's CryEngine and used in the popular game *Far Cry*, as shown in Figure 4.2.1, and Color Plate 6.

FIGURE 4.2.1 *CryEngine screenshot, featuring the game Far Cry.*

In this article, we will discuss the implementation details of an inexpensive adaptive glare technique. The technique relies upon only 1.1 pixel shaders and hence may be performed on most mainstream gaming PCs and game consoles. Besides being inexpensive to execute, the technique is adaptive, which means that the threshold for what is considered "bright" in a given image can vary according to the brightness of the scene.

Overview

Before diving into the implementation details, it is important to understand the overall strategy of this image post-processing technique. A number of intermediate images will be generated and processed as shown in Figure 4.2.2.

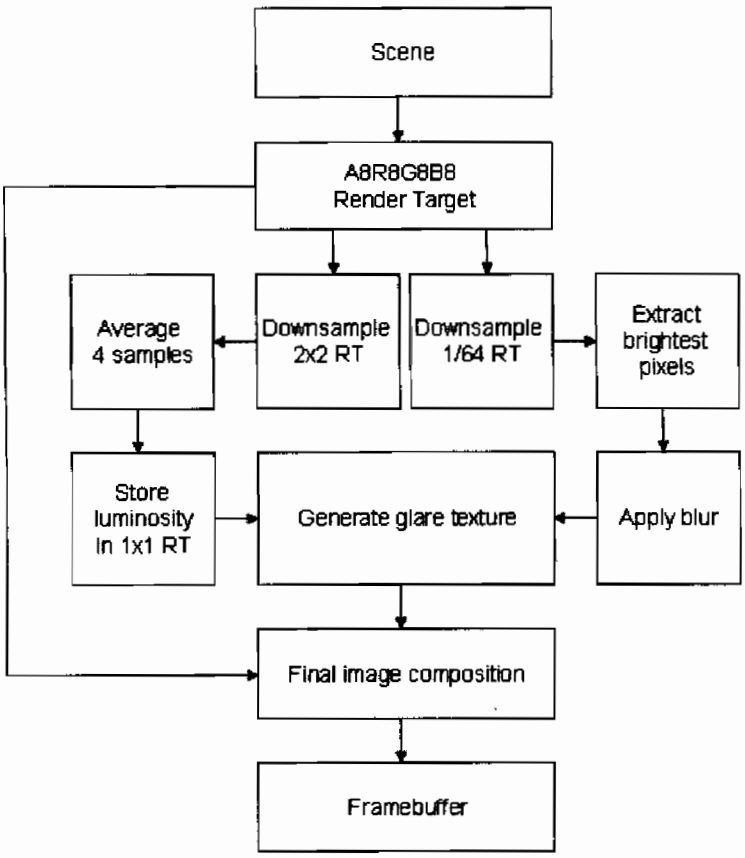


FIGURE 4.2.2 *Overview of adaptive glare implementation steps.*

The initial input to this process is our rendered scene. This can be generated by doing a `StretchRect` operation from the back buffer to a texture of the same size or by rendering the scene directly into a texture. In either case, this initial image is the same size as the back buffer and is stored in an RGBA texture with 8 bits per channel.

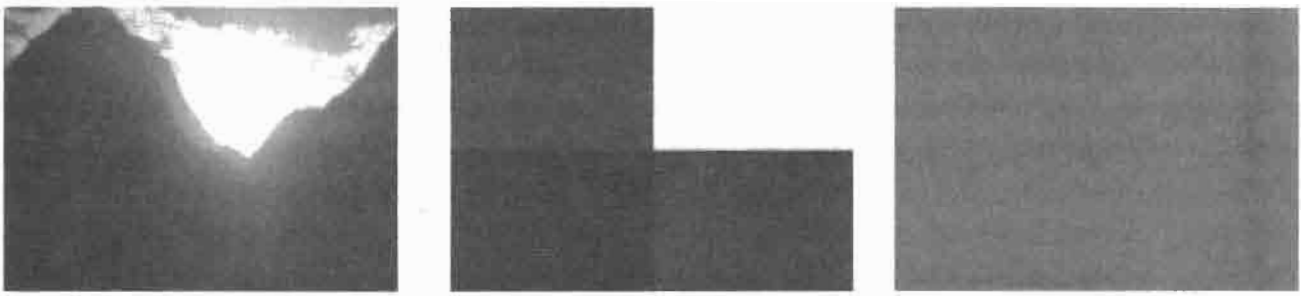
This input texture is resized into two other textures, a very small 2 × 2 texture and another which is 1/64 the size of the back buffer. This small 2 × 2 texture will be further downsampled to 1 × 1 in order to compute a single value for the average luminance of the scene. From the 1/64-sized texture, the brightest pixels will be blurred. This blurred texture will be composited with the initial scene image, including effects from the average luminance.

Adaptive Glare Algorithm

As mentioned above, there are two main textures required to implement this effect: the *luminance texture* and the *glare texture*.

Luminance

The luminance texture is generated by performing a `StretchRect` from the original scene image into a 2 × 2 texture. This 2 × 2 texture is further downsampled to 1 × 1 and converted to a single scalar luminance value as shown in Figure 4.2.3. Naturally, all of these operations are performed on the GPU in order to avoid any CPU intervention.



(a) Original scene texture (b) Scene scaled to 2 × 2 (c) Scene luminance

FIGURE 4.2.3 *Luminance computation.*

This single average luminance is computed by the GPU in a pixel shader by rendering a screen-aligned quadrilateral into the 1 × 1 luminance texture from the intermediate 2 × 2 texture. The shader samples the intermediate 2 × 2 texture four times using pick-nearest filtering. The samples use texture coordinates which have been offset by 0.5 in the vertex program:

```
OUT.Tex0.xy = IN.TexCoord0.xy + vTexOffset01.xy;
OUT.Tex1.xy = IN.TexCoord0.xy + vTexOffset01.zw;
OUT.Tex2.xy = IN.TexCoord0.xy + vTexOffset02.xy;
OUT.Tex3.xy = IN.TexCoord0.xy + vTexOffset02.zw;
```

where:

```
vTexOffset01.xyzw = ( -0.5, -0.5 , -0.5, 0.5)
vTexOffset02.xyzw = ( 0.5, -0.5, 0.5, 0.5)
```

In the pixel shader, these four samples are averaged and converted to luminance:

```
// average color samples
float3 vAvgColor = saturate(vColor01.xyz * 0.25 +
                            vColor02.xyz * 0.25 +
                            vColor03.xyz * 0.25 +
                            vColor04.xyz * 0.25);

// only output luminance
OUT.Color.xyz = dot(float3(0.3, 0.59, 0.11), vAvgColor.xyz);

// glare amount interpolation value
OUT.Color.w = vGlareAmountLerp.w;
```

These four samples could also be automatically averaged using a single sample with bilinear filtering, saving some instructions in the pixel shader. In order to smooth out the luminance over time and avoid popping, this result is blended with the luminance of the previous frame. The blending ratio can be tuned to various values depending on the desired effect.

Glare

Now that we have computed a single luminance value for the scene and blended it with the running luminance from the previous frames, we can compute the *glare texture*. The first step in computing this texture is downsampling the original scene image to an extremely low resolution texture with 1/64 of the original resolution (1/8 in height and 1/8 in width). This can be done by using the `StretchRect` API or by manually rendering a screen-aligned quadrilateral and performing the filtering in a pixel shader. From this 1/64-sized texture, the brightest pixels are extracted using a simple thresholding operation.

```
// remove low-luminance colors
float3 vFinalGlare = saturate(vGlareColor.xyz - vGlareParams.xyz)
                     * vGlareParams.w;

// modulate brightest pixels by inverse luminance
OUT.Color.xyz = vFinalGlare.xyz *(1- vGlareAmount.w);
```

where:

- vGlareColor is the current glare texture
- vGlareParams.xyz is the threshold value (0.4 is usually a good value)
- vGlareParams.w is the maximum user defined amount of glare
- vGlareAmount is the luminance

After this thresholded image is computed, a simple blur operation is performed, resulting in a glare texture like the one shown in Figure 4.2.4(c).

(a) Original scene texture (b) Thresholded image (c) Blurred glare texture

FIGURE 4.2.4 *Glare texture creation steps.*

Final Image Composition

In our final step, we will composite the glare texture over the original scene image. A simple approach would be to just add the glare texture to the original scene image. In this case, care must be taken when generating the glare texture or the scene can end up looking extremely bright in some situations. To avoid this, the threshold could be adjusted according to the precomputed ambient lighting in the current sector/scene, but this will depend on the game engine structure. A more robust method of compositing the glare texture with the original scene image is to make the glare contribution proportional to the luminance of the pixel in the original scene image. Composing the final image this way will display a visible but subtle glare on low-luminance pixels, avoiding the extremely bright cases.

```
// Compute scene with full glare
float3 vFinalGlareColor = saturate(vGlareColor.xyz +
                                   vScreenColor.xyz);

// Compute scene luminance
float  fLum = dot(vScreenColor.xyz, float3(0.3, 0.59, 0.11));

// Interpolate between current scene and scene with full glare
OUT.Color.xyz = saturate(vScreenColor.xyz*(1-fLum) +
                         vFinalGlareColor.xyz*fLum );
```

For yet another look, it is possible to combine the glare texture with the original scene image using the following inverse multiplication operation:

```
vFinalGlareColor = saturate(vGlareColor.xyz + vScreenColor.xyz –
                            vGlareColor.xyz * vScreenColor.xyz);
```

Minimizing Artifacts

Simply using the `StretchRect` API to perform many of the downsampling operations in this algorithm is prone to artifacts on high-frequency scenes since not enough filtering is performed. This can lead to wild variations in the luminance computation and

flickering as the scene changes or the user's view changes. An alternative method which minimizes this problem is to first resize the original scene image into a low-resolution texture (128 × 128 or less) and generate mipmaps all the way down to 1 × 1. This will create a more accurate and stable luminance result but is more costly than the method outlined in this article.

Further Improvements

One of the goals of any kind of glare effect is to approximate the look of high-dynamic range rendering. Once we do have access to real-time high-dynamic range rendering, it will possible to use this luminance computation technique to dynamically control the tone mapping step in the high-dynamic range rendering process.

Sample Application

A sample application is available in the companion CD-ROM. The sample application uses a very simple scene showing three brightly colored teapots inside an opened dark cube as shown in Figure 4.2.5. This is done so that you can easily see the result of the adaptive glare when moving from inside to outside of the cube. The mouse and the W and S keys can be used to move the camera. F1 enables the glare effect described in this article while F2 disables it.

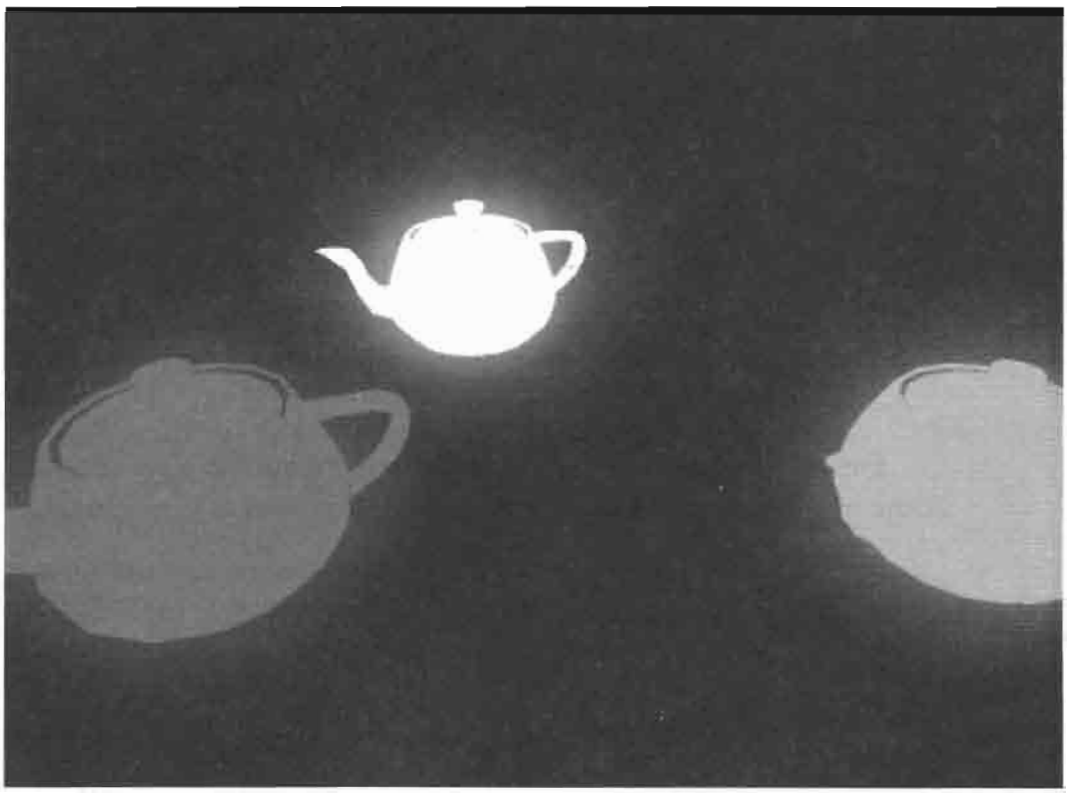


FIGURE 4.2.5 *Adaptive glare demo.*

Conclusion

The article presented an empirical technique to achieve a well-behaved and natural looking glare effect. The final post-processed image contains a subtle glare which adjusts smoothly based upon the luminance of the rendered scene. This technique is currently being used in Crytek's CryEngine, which is used by the recent popular game *Far Cry*.

Acknowledgments

Special thanks to Wolfgang Engel for his great help and patience in reviewing and suggesting ways to improve this article. Thanks also to Martin Mittring and Carsten Wenzel for reviewing the article and suggesting ideas for higher quality image compositing and artifact minimization. Also, special thanks to Márcio Martins for reviewing the article and providing the model loader used in the sample application.

Additional References

[Kawase03] Masaki Kawase, "Frame Buffer Post-processing Effects in Double-S.T.E.A.L," GDC 2003.

[Kawase04] Masaki Kawase, "Practical Implementation of High-dynamic Range Rendering," GDC 2004.

[Vlachos03] Alex Vlachos, Greg James, "Special Effects with DirectX 9," GDC 2003.