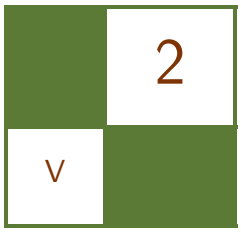

Post-Processing Effects on Mobile Devices

Marco Weber and Peter Quayle

Image processing on the GPU, particularly real-time post-processing, is an important aspect of modern graphical applications, and has wide and varied uses from games to user interfaces. Because of the continually improved performance levels of graphics hardware, post-processing effects are now common on the desktop and in the console space. With the current generation of OpenGL ES 2.0-enabled embedded GPUs, an increasing number of such effects are also possible on mobile devices. Bloom effects, for example, can add a vivid glow to a scene; color transforms and distortions can provide a stylistic flair; blur, and depth-of-field effects can draw attention to areas of interest, or highlight selected elements in a user interface. Figure 2.1 shows some generic examples of post-processing effects, and Figure 2.2 demonstrates the use of a blur effect in a user interface.

Post-processing is the modification and manipulation of captured or generated image data during the last stage of a graphics pipeline, resulting in the final output picture. Traditionally, these operations were performed by the host system CPU by reading back the input image data and altering it. This is a very costly operation and is not suitable for applications that require interactive frame rates on mobile platforms. Using the processing power of modern graphics hardware to do the image data manipulation is a more practical and efficient approach.

This chapter describes a general approach to post-processing and provides an example of a real-time effect that can be implemented on OpenGL ES 2.0-capable hardware, such as POWERVR SGX.

2.1 Overview

The general idea behind post-processing is to take an image as input and generate an image as output (see Figure 2.3). You are not limited to only using the provided input image data, since you can use any available data source (such as

Figure 2.1. Radial twist, sepia color transformation, depth of field, depth of field and sepia color transformation combined (clockwise from top left).

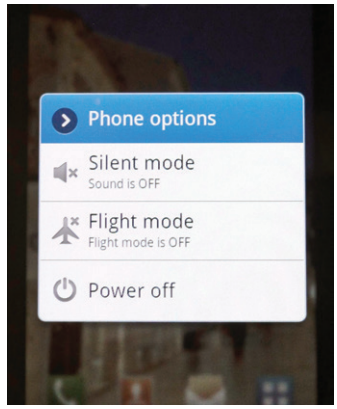

Figure 2.2. Post-processing effect on the Samsung Galaxy S. The background of the user interface is blurred and darkened to draw attention to items in the foreground.

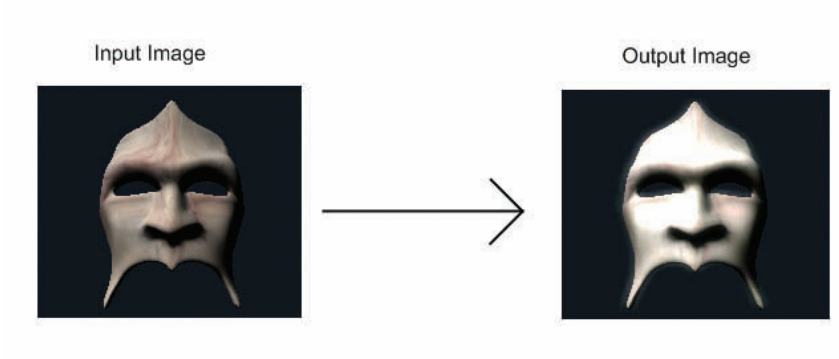

Figure 2.3. Image input and output.

depth and stencil buffers), as additional input for the post-processing step. The only requirement is that the end result has to be an image.

One direct advantage of this approach is that, due to the identical basic format of input and output, it is possible to chain post-processing techniques. As illustrated in Figure 2.4, the output of the first post-processing step is reused as input for the second step. This can be repeated with as many post-processing

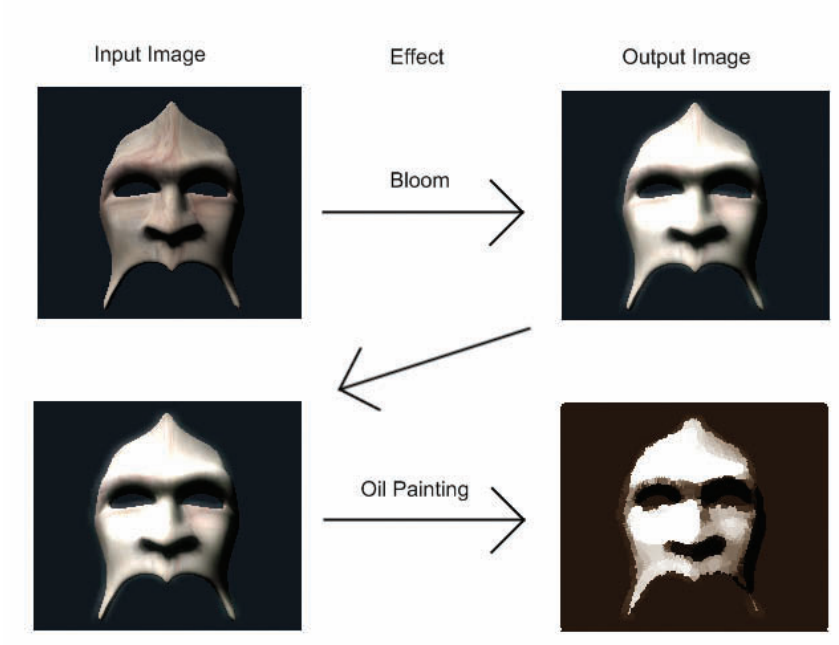

Figure 2.4. Image input and output chain.

techniques as required. Some sets of post-processing techniques can be merged to avoid the overhead of chaining, while others can exploit chaining to achieve a performance increase, as explained later.

Each step is performed by rendering to a frame buffer object (FBO). The final output image will be displayed on the screen.

2.2 Technical Details

To generate the first input image we must render our scene to a texture. As mentioned in the previous section, to do this we use FBOs. There are other ways to accomplish the same results, such as creating textures by copying data from the framebuffer, but using FBOs is better for performance (for a more detailed discussion see [Imagination Technologies 10]).

We will make use of pixel-shader support, as found in the POWERVR SGX graphics cores, since they provide a great deal of flexibility when developing post-processing effects. It is possible to implement some basic post-processing techniques with a fixed-function pipeline, but this is beyond the scope of this article.

The basic algorithmic steps for a post-processing effect are as follows:

1. Render the scene to a texture.
2. Render a full screen pass using a custom pixel shader, with the texture from the previous stage as input, to apply the effect.
3. Repeat step two until all effects are processed.

The first step is the most straightforward, because it simply requires setting a different render target. Instead of rendering to the back buffer, you can create an FBO of the same dimensions as your frame buffer. In the case that the frame buffer dimensions are not a power of two (e.g., 128×128, 256×256 etc.), you must check that the graphics hardware supports non-power-of-two (NPOT) textures. If there is no support for NPOT textures, you could allocate a power-of-two FBO that approximates the dimensions of the frame buffer. For some effects it may be possible to use an FBO far smaller than the frame buffer, as discussed in Section 2.4.

In step two, the texture acquired during step one can be used as input for the post-processing. In order to apply the effect, a full-screen quad is drawn using a post-processing pixel shader to apply the effect to each pixel of the final image.

All of the post-processing is executed within the pixel shader. For example, in order to apply an image convolution filter, neighboring texels have to be sampled and modulated to calculate the resulting pixel. Figure 2.5 illustrates the kernel, which can be seen as a window sliding over each line of the image and evaluating each pixel at its center by fetching neighboring pixels and combining them.

Figure 2.5. Gaussian blur filtering.

Step three describes how easy it is to build a chain of post-processing effects by simply executing one after another, using the output of the previous effect as the input for the next effect. In order to accomplish this, it is necessary to allocate more than one frame buffer object as it is not possible to simultaneously read from and write to the same texture.

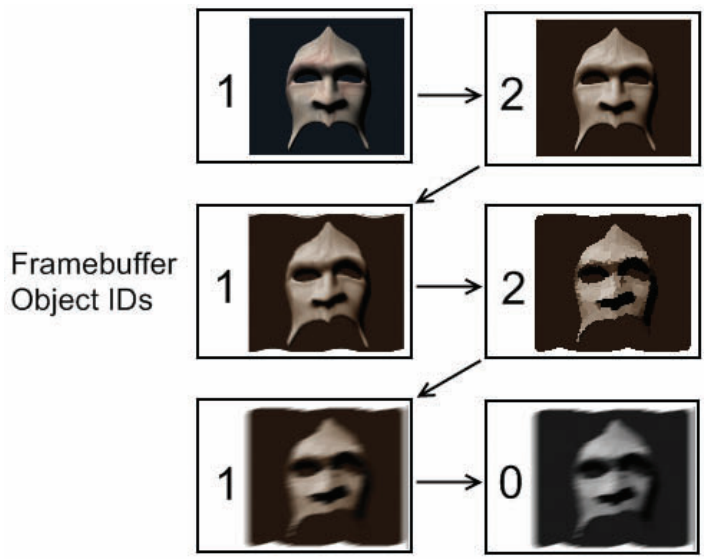

Figure 2.6. Post-processing steps and frame buffer IDs.

Figure 2.6 illustrates the re-use of several frame buffer objects:

- The initial step renders to the frame buffer object with ID 1
- The second step renders to the frame buffer object with ID 2, using the previous frame buffer object as input.
- The whole procedure is repeated for steps three and four, but instead of using frame buffer object 2 again for the last step, the back buffer is used since the final result will be displayed on the screen.

Depending on the individual effect, some of the illustrated passes may be merged into one pass. This avoids the bandwidth cost associated with processing the whole image for each effect that is merged, and reduces the total number of passes required.

2.3 Case Study: Bloom

The whole concept of post-processing, as presented in the previous section, is suitable for high-performance graphics chips in the desktop domain. In order to implement post-processing on mobile graphics chipsets, such as POWERVR SGX graphics cores, it is most important to act with caution. In this section, we illustrate an actual implementation of the bloom effect tailored for implementation on an embedded GPU, such as POWERVR SGX, at an interactive frame rate. The required optimizations and alterations to the original algorithm are explained throughout the following sections. At the beginning of this section the effect itself will be explained, followed by the actual implementation and some optimization strategies.

Figure 2.7. Mask without and with glow applied.

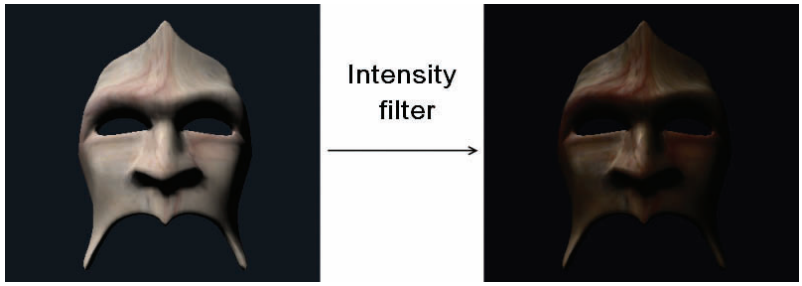

Figure 2.8. Applying an intensity filter to obtain bright parts.

The bloom effect simulates our perception of bright light by producing a pronounced glow and a halo around bright objects. Furthermore, it gives the impression of high dynamic range rendering despite being done in a low dynamic range. Another useful side effect is that it reduces aliasing artifacts due to the slight glow at the edges of objects. The whole effect and its intensity can be controlled by an artist and can be used to attract attention to objects and make them distinct (see Figure 2.7).

The bloom effect is achieved by intensifying bright parts of the image. In order to accomplish this, the bright parts of the image have to be identified and isolated. This can happen either implicitly by applying an intensity filter to the input image to extract the bright parts (see Figure 2.8), or explicitly by specifying the glowing parts through a separate data source, (e.g., the alpha channel of the individual textures).

The intensity filtered texture, which contains the bright parts of the image, will then be blurred with a convolution kernel in the next step. The weights of the kernel used in this example are chosen to resemble the weights of the Gaussian blur kernel. The kernel itself is applied by running a full shader pass over the whole texture and executing the filter for each texture element. Depending on the number of blur iterations and the size of the kernel, most of the remaining high frequencies will be eliminated and a ghostly image will remain (see Figure 2.9). Furthermore, due to the blurring, the image is consecutively smeared and thus enlarged, creating the halos when combined with the original image.

The final step is to additively blend the resulting bloom texture over the original image by doing a full screen pass. This amplifies the bright parts of the image and produces a halo around glowing objects due to the blurring of the intensity-filtered texture (see Figure 2.10).

The final amount of bloom can be controlled by changing the blend function from additive blending to an alpha-based blending function, offering even more artistic freedom. By animating the alpha value we can vary the amount of bloom and simulate a pulsing light.

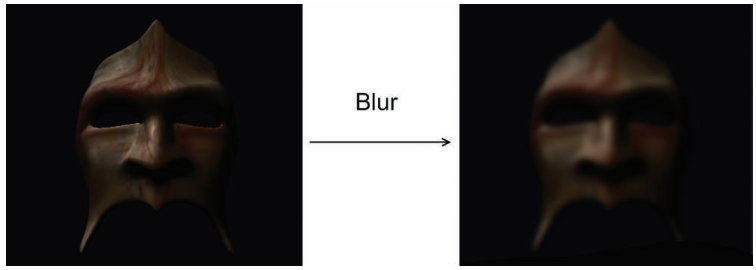

Figure 2.9. Applying Gaussian blur.

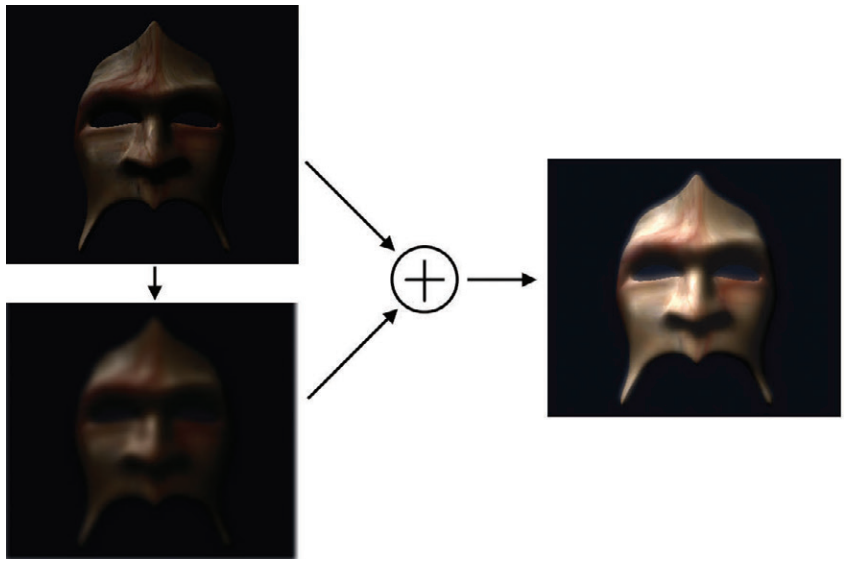

Figure 2.10. Additive blending of original and blurred intensity-filtered image.

2.4 Implementation

The bloom algorithm presented in the previous section describes the general approach one might implement when processing resources are vast. Two full screen passes for intensity filtering and final blending and several passes for the blur filter in the most naive implementation are very demanding even for the fastest graphics cards.

Due to the nature of mobile platforms, adjustments to the original algorithm have to be made in order to get it running, even when the hardware is equipped with a highly efficient POWERVR SGX core.

The end result has to look convincing and must run at interactive frame rates. Thus, most of the steps illustrated in Section 2.3 have to be modified in order to meet the resource constraints.

2.4.1 Resolution

One of the realities of mobile graphics hardware is a need for low power and long battery life, which demand lower clock frequencies. Although the POWERVR SGX cores implement a very efficient tile-based deferred rendering approach, it is still essential to optimize aggressively when implementing full screen post-processing effects.

In our implementation the resolution of the frame buffer object for the blurred texture was set to 128×128, which has shown to be sufficient for VGA (640×480) displays. Depending on the target device's screen and the content being rendered, even 64×64 may be adequate; the trade-off between visual quality and performance should be inspected by regularly testing the target device. It should be kept in mind that using half the resolution (e.g., 64×64 instead of 128×128) means a 75% reduction in the number of pixels being processed.

Since the original image data is not being reused because of the reduced resolution, the objects using the bloom effect have to be redrawn. This circumstance can be exploited for another optimization. As we are drawing only the objects which are affected by the bloom, it is possible to calculate a bounding box enclosing these objects that in turn will be reused in the following processing steps as a kind of scissor mechanism.

When initially drawing the objects to the frame buffer object, one could take the optimization even further and completely omit texture mapping (see Figure 2.11). This would mean that the vertex shader would calculate only the

Figure 2.11. Difference between nontextured (left) and textured (right) bloom.

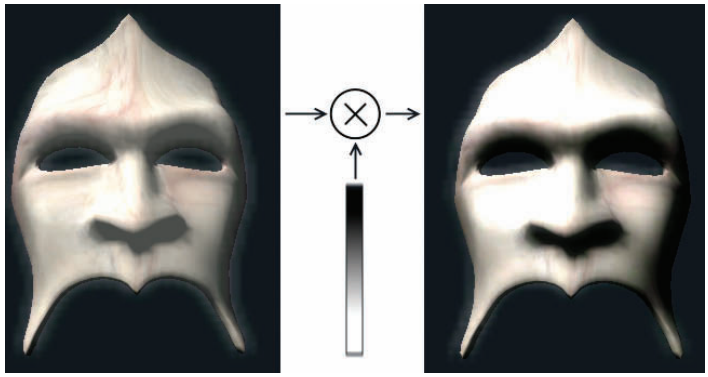

Figure 2.12. Transforming the brightness scalar by doing a texture lookup in an intensity map.

vertex transformation and the lighting equation, which would reduce the amount of data being processed in the fragment shader even further, at the expense of some detail. Each different case should be evaluated to judge whether the performance gain is worth the visual sacrifice.

When omitting texture mapping, the scalar output of the lighting equation represents the input data for the blur stage, but if we simply used scalar output for the following steps, the resulting image would be too bright, even in the darker regions of the input image, which is why the intensity filter has to be applied. Applying the intensity filter can be achieved by doing a texture lookup into a 1D texture, representing a transformation of the luminance (see Figure 2.12). This texture can be generated procedurally by specifying a mapping function, or manually, whereby the amount of bloom can be stylized to meet artistic needs. The lookup to achieve the intensity filtering can potentially be merged into the lighting equation. Other parts of the lighting equation could also be computed via the lookup table, for example, by premultiplying the values in it by a constant color.

2.4.2 Convolution

Once we've rendered our intensity-filtered objects to the frame buffer object, the resulting image can be used as input for the blur-filtering steps. This section explains the blur-filtering methods which are depicted in Figure 2.13.

Image convolution is a common operation and can be executed very efficiently on the GPU. The naive approach is to calculate the texture-coordinate offsets (e.g., 1/width and 1/height of texture image) and sample the surrounding texels. The next step is to combine these samples by applying either linear filters (Gaussian, median, etc.) or morphologic operations (dilation, erosion, etc.).

Figure 2.13. Blurring the input image in a low-resolution render target with a separated blur filter kernel.

In this case we will apply a Gaussian blur to smooth the image. Depending on the size of the filter kernel, we have to read a certain amount of texture values, multiply each of them by a weight, sum the results, and divide by a normalization factor. In the case of a 3×3 kernel this results in nine texture lookups, nine multiplications, eight additions, and one divide operation, which is a total of 27 operations to filter a single texture element. The normalization can be included in the weights, reducing the total operation count to 26.

Fortunately, the Gaussian blur is a separable filter, which means that the filter kernel can be expressed as the outer product of two vectors:

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} = (1 \quad 2 \quad 1) \otimes (1 \quad 2 \quad 1).$$

Making use of the associativity,

$$t \cdot (v \cdot h) = (t \cdot v) \cdot h,$$

where t represents the texel, v the column, and h the row vector, we can first apply the vertical filter and, in a second pass, the horizontal filter, or vice versa. This results in three texture lookups, three multiplications, and two additions per pass, giving a total of 16 operations when applying both passes. This reduction in the number of operations is even more dramatic when increasing the kernel size (e.g., 5×5, 7×7, etc.)(see Table 2.1):

Kernel	Texture Lookups	Muls	Adds	No. Of Operations
3x3 (standard)	9	9	8	26
3x3 (separated)	6	6	4	16
5x5 (standard)	25	25	24	74
5x5 (separated)	10	10	8	28
9x9 (standard)	81	81	80	242
9x9 (separated)	18	18	17	53

Table 2.1.

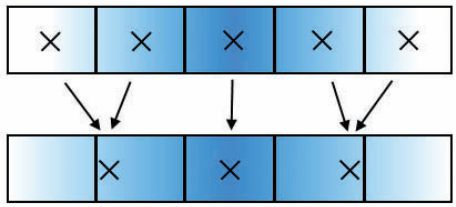

Figure 2.14. Reducing the number of texture lookups by using hardware texture filtering.

In most cases, separating the filter into horizontal and vertical passes results in a large performance increase. However, the naive single-pass version may be faster in situations in which bandwidth is severely limited. It is always worth benchmarking to ensure the best solution for a given platform or scenario.

The number of texture lookups can be decreased again by exploiting hardware texture filtering. The trick is to replace the texture lookup for the outer weights with one which is between the outer texels, as shown in Figure 2.14.

The way this works is as follows: when summing the contribution, s, of the outer texels, t_0 and t_1, in the unoptimized version, we use

$$s = t_0 w_0 + t_0 w_1. \tag{2.1}$$

When we sample between the outer texels with linear filtering enabled we have

$$s = t_0(1-u) + t_1 u, \tag{2.2}$$

where u is the normalized position of the sample point in relation to the two texels. So by adjusting u we can blend between the two texel values. We want to blend the texels with a value for u such that the ratio of $(1-u)$ to u is the same as the ratio of w_0 to w_1. We can calculate u using the texel weights

$$u = w_1 / (w_0 + w_1). \tag{2.3}$$

We can then substitute u into Equation (2.2). Because u must be a value between 0 and 1, we need to multiply s by the sum of the two weights. Our final equation looks like this:

$$s = (t_0 (1-u) + t_1 u)(w_0 + w_1). \tag{2.4}$$

Although this appears to contain more operations than Equation (2.1), the cost of the term in the first set of brackets is negligible because linear texture filtering is effectively a free operation. In the case of the 5×5 filter kernel, the number of texture lookups can be reduced from ten to six, yielding the identical number of computation necessary as for the 3×3 kernel.

It is important that the texture coordinates are calculated in the vertex shader and passed to the pixel shader as varyings, rather than being calculated in the pixel shader. This will prevent dependent texture reads. Although these are supported, they incur a potentially substantial performance hit. Avoiding dependent texture reads means that the texture-sampling hardware can fetch the texels sooner and hide the latency of accessing memory.

2.4.3 Blending

The last step is to blend the blurred image over the original image to produce the final result, as shown in Figure 2.15.

Therefore, the blending modes have to be configured and blending enabled so that the blurred image is copied on top of the original one. Alternatively, you could set up an alpha-value-based modulation scheme to control the amount of bloom in the final image.

In order to increase performance and minimize power consumption, which is crucial in mobile platforms, it is best that redundant drawing be avoided as much as possible. The single most important optimization in this stage is to minimize the blended area as far as possible. Blending is a fill-rate intensive operation, especially when being done over the whole screen. When the bloom effect is applied only to a subset of the visible objects, it is possible to optimize the final blending stage:

- In the initialization stage, calculate a bounding volume for the objects which are affected.
- During runtime, transform the bounding volume into clip space and calculate a 2D bounding volume, which encompasses the projected bounding volume. Add a small margin to the bounding box for the glow.

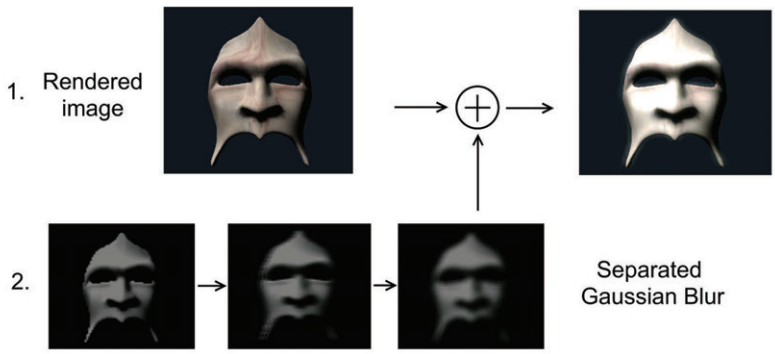

Figure 2.15. Overview of the separate bloom steps.

Figure 2.16. Bounding box derived rectangle (red) used for final blending.

- Draw the 2D bounding object with appropriate texture coordinates to blend the blurred texture over the original image.

Figure 2.16 shows the blending rectangle which is derived from the object's bounding box. The bounding box in this case is a simple axis-aligned bounding box which is calculated during the initialization. At runtime, the eight vertices of the bounding box are transformed into clip space and the minimum and maximum coordinate components are determined. The blended rectangle is then derived from these coordinates and the texture coordinates are adapted to the vertex positions. Depending on the shape of the object, other, more suitable, bounding volumes might be appropriate (e.g., bounding spheres).

This bounding-box-directed blending can lead to artifacts when the blending rectangles of two or more objects overlap, resulting in sharp edges and highlights that are too bright. A work-around for this overlap issue is to use the stencil buffer:

- Clear the stencil buffer to zero and enable stencil testing.
- Configure the stencil test so that only the pixels with a stencil value of zero are rendered, and the stencil value is always incremented.
- Draw all bounding volumes and disable stencil test.

Use of this algorithm prevents multiple blend operations on a single pixel and produces the same result as a single full screen blend. On a tile-based deferred renderer like POWERVR SGX, stencil operations are low cost.

2.5 Conclusion

We have presented a brief introduction to post-processing, followed by a detailed case study of a well-known post-processing effect. We have illustrated optimization techniques that make it possible to use the effect while achieving interactive

framerates on mobile hardware. Many of the optimization techniques shown can be adapted and applied to other post-processing effects.

Bibliography

[Imagination Technologies 10] *PowerVR SGX OpenGL ES 2.0 Application Developer Recommendations*. 2010. http://www.imgtec.com/POWERVR/insider/POWERVR-sdk.asp.