Post Processing Effects

Whitepaper

Copyright © Imagination Technologies Limited. All Rights Reserved.

This publication contains proprietary information which is subject to change without notice and is supplied 'as is' without warranty of any kind. Imagination Technologies and the Imagination Technologies logo are trademarks or registered trademarks of Imagination Technologies Limited. All other logos, products, trademarks and registered trademarks are the property of their respective owners.

Filename : Post Processing Effects.Whitepaper
Version : PowerVR SDK REL_3.4@3164023a External Issue
Issue Date : 30 Sep 2014
Author : Imagination Technologies Limited

Contents

1. Introduction 4

2. Post Processing Overview..6

2.1. Basics 6
2.2. Technical details 7

3. Post Processing Training Course: Bloom...9

3.1. Bloom..9
3.2. Implementation 10
3.2.1. Resolution 11
3.2.2. Convolution 12
3.2.3. Blending...13

4. References..16

5. Contact Details..17

List of Figures

Figure 1. Sepia colour transformation..4
Figure 2. Radial twist..4
Figure 3. Depth of Field...5
Figure 4. Depth of field and sepia colour transformation 5
Figure 5. Image input and output 6
Figure 6. Image input and output chain 6
Figure 7. Gaussian blur filtering example 7
Figure 8. Post processing steps and framebuffer IDs...8
Figure 9. Tron 2.0 character without (left) and with glow (right) (source: Gamasutra.com) 9
Figure 10. Applying a high pass filter to obtain bright parts 9
Figure 11. Applying Gaussian blur...10
Figure 12. Additive blending of original and blurred high pass filtered image 10
Figure 13. Difference between non-textured (left) and textured (right) bloom.......................11
Figure 14. High pass filtering the brightness scalar by doing a texture lookup in an intensity map 12
Figure 15. Blurring the input image in a low resolution render target with a separated blur filter kernel 12
Figure 16. Reducing the amount of texture lookups by using hardware texture filtering.........13
Figure 17. Overview of the separate Bloom steps 14
Figure 18. Bounding box derived rectangle (red) used for final blending 14

List of Tables

Table 1. Result of applying vertical and horizontal filters...13

1. Introduction

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 very costly in terms of performance and is not suitable for applications that require interactive frame rates on mobile platforms. A more recent and practical approach is to use the processing power of the graphics hardware to do the image data manipulation.

Due to the ever increasing performance of graphics hardware, post processing effects are now quite common in video games. Depending on the power of the graphics accelerator that is available to the developer, the set of post processing effects can range from simple colour transformations and over image distortions, to complex depth of field computations (see examples in Figure 1 to Figure 4).

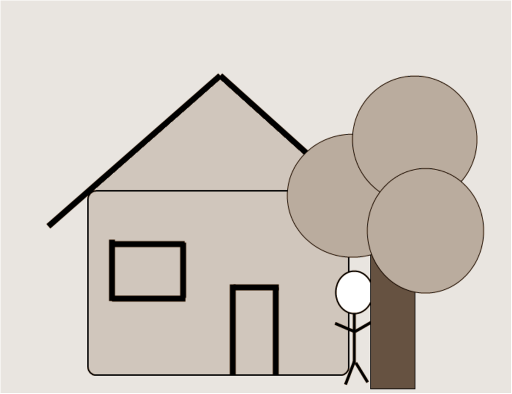

Figure 1. Sepia colour transformation

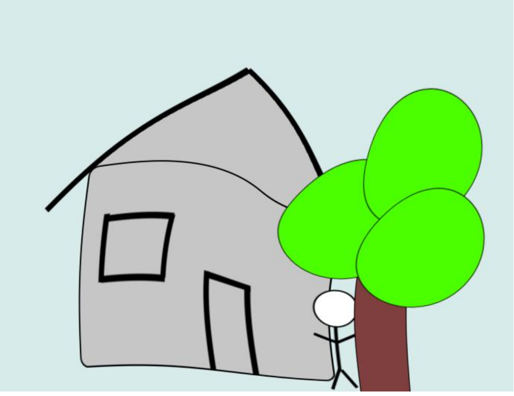

Figure 2. Radial twist

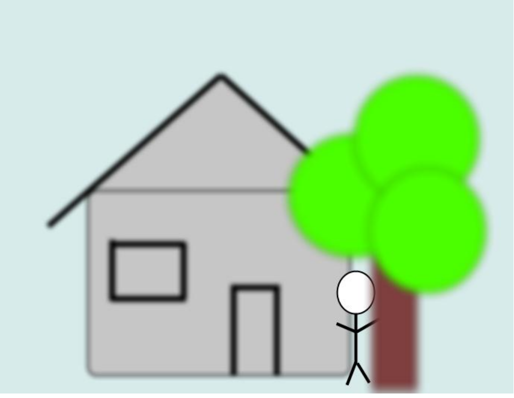

Figure 3. Depth of Field

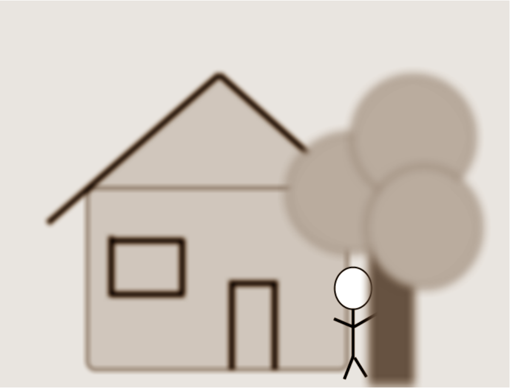

Figure 4. Depth of field and sepia colour transformation

This document describes a general approach to post processing and provides an example of a real-time effect that can be implemented for mobile platforms utilising PowerVR hardware. The remaining sections of this document provide the following information:

- Section 2 explains the common basic technique behind post processing effects.
- Section 3 offers a more detailed explanation based on a real example.
- Section 4 contains references to recommended reading material.

2. Post Processing Overview

2.1. Basics

Oversimplified, the general idea behind post processing is to take an image as input and generate an image as output (see Figure 5). The user is not limited to only using the provided input image data, as it is possible to use any available data source, such as 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.

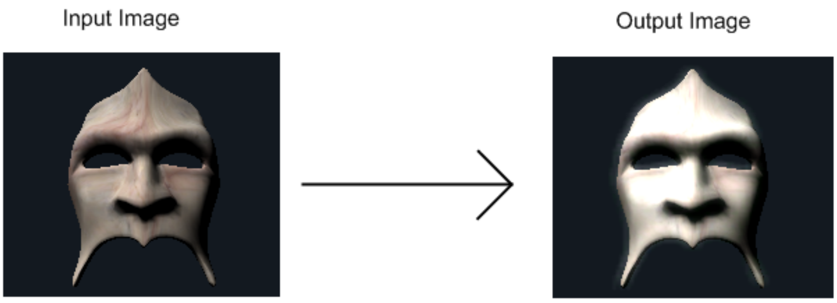

Figure 5. Image input and output

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 6, the output of the first post processing step is reused as input for the second step. This can be, of course, combined and repeated with as many post processing techniques as required.

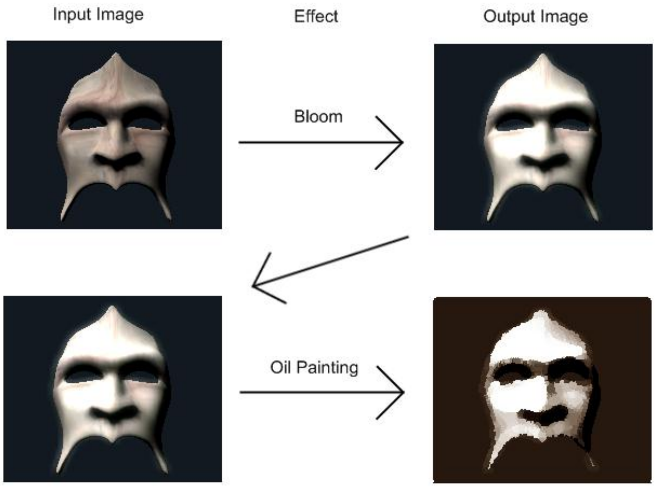

Figure 6. Image input and output chain

Each step is performed in a frame buffer object and makes intensive use of render to texture mechanisms. The final image will be considered as output and displayed on the screen.

2.2. Technical details

As mentioned in the previous section, one of the most basic techniques used is render to texture. There are other ways of accomplishing the same results, such as creating textures from frame buffer content, but they are not recommended from a performance point of view (for a more detailed discussion see the “3D Application Development Recommendations” document, available in the documentation folder of the PowerVR SDK).

Another requirement is the availability of pixel shader support as it provides a great deal of flexibility when developing post processing effects. It is possible to implement some of the post processing techniques with a fixed function pipeline, but this is beyond the scope of this document. 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 to apply the effect.
3. Repeat step two until all effects are processed.

The first step is the most straight-forward one, as it simply requires setting a different render target. Instead of rendering to the back buffer, it is possible to create a framebuffer object (or render target in DirectX) of the same dimensions as the framebuffer. In the case that the framebuffer dimensions are not a power of two (e.g. 128x128, 256x256, etc.), the user must check that the graphic hardware supports non power of two (NPOT) textures. If there is no support for NPOT textures, the user should allocate a power of two framebuffer object that is larger than the framebuffer.

In stage two, the texture acquired during stage one can be used as input for the post processing. In order to apply the effect, a full screen quad must be 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, neighbouring texels have to be sampled and modulated to calculate the resulting pixel. Figure 7 illustrates the kernel, which can be seen as a window sliding over each line of the image and evaluating each pixel at its centre by fetching neighbouring pixels and combining them.

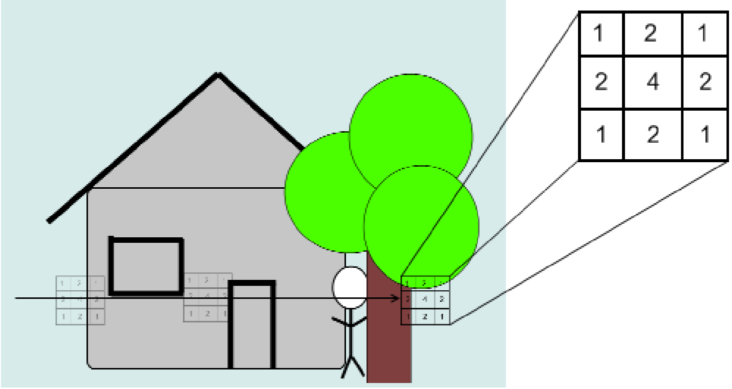

Figure 7. Gaussian blur filtering example

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 8 illustrates the reuse of several framebuffer objects:

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

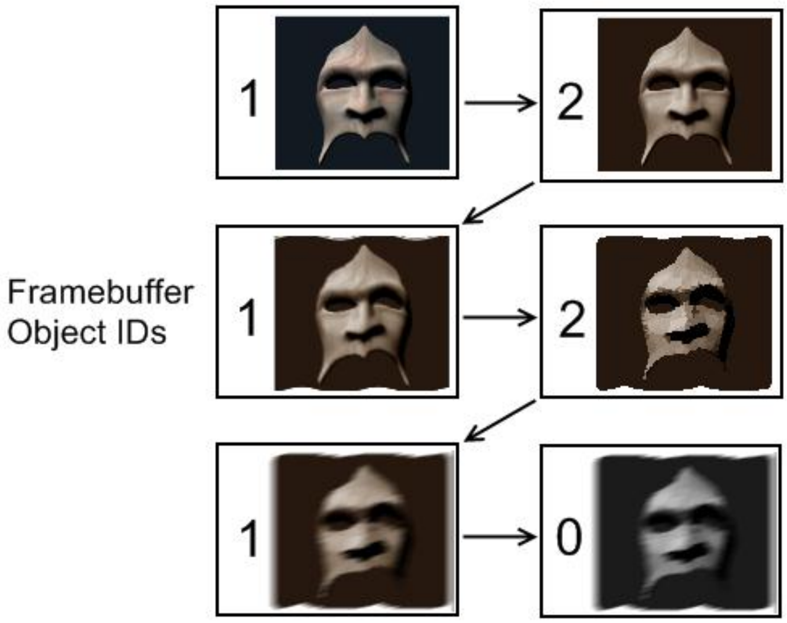

Figure 8. Post processing steps and framebuffer IDs

Depending on the individual effect, some of the illustrated passes may be merged into one pass, as the cost of processing the whole image data is quite severe.

3. Post Processing Training Course: 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 get them up and running on mobile graphics chipsets, it is most important to act with caution. The contents of this chapter illustrate an actual implementation of the Bloom effect tailored towards the low end configurations running at an interactive frame rate. The required optimizations and alterations to the original algorithm are explained throughout the following sections. The Bloom training course containing the source code can be found in the PowerVR SDK. At the beginning of this section the effect itself will be explained, followed by the actual implementation and concluded by general performance guidelines.

3.1. Bloom

The Bloom effect simulates the perception of bright light in our eyes, by producing a subtle glow and halo around bright objects. Furthermore, it gives the impression of high dynamic range rendering although the rendering is done in a low dynamic range. Another useful side effect is that it reduces aliasing artefacts at the edges of objects due to the slight glow. The whole effect and its intensity can be controlled and used by artists to make objects stand out of the scene or amongst other objects (see Figure 9).

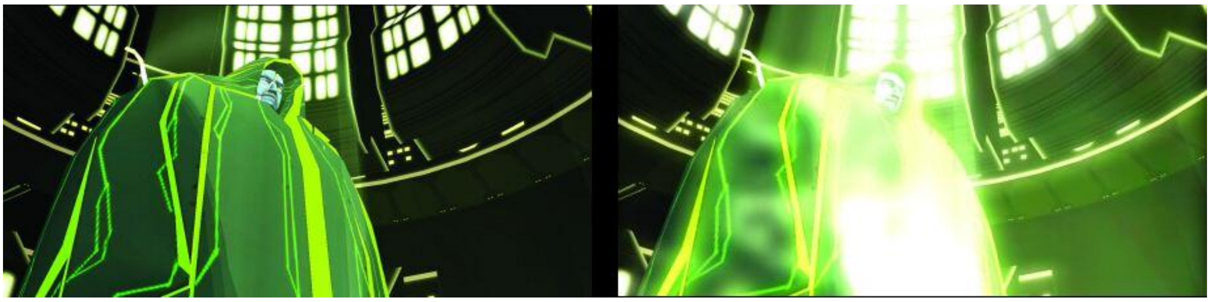

Figure 9. Tron 2.0 character without (left) and with glow (right) (source: Gamasutra.com)

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. This can either happen implicitly by applying a high-pass filter to the input image to extract the bright parts (see Figure 10) or explicitly by specifying the glowing parts through a separate data source, e.g., the alpha channel of the individual textures. Furthermore, the latter can be employed to animate the glow, e.g., doing a simple pulse.

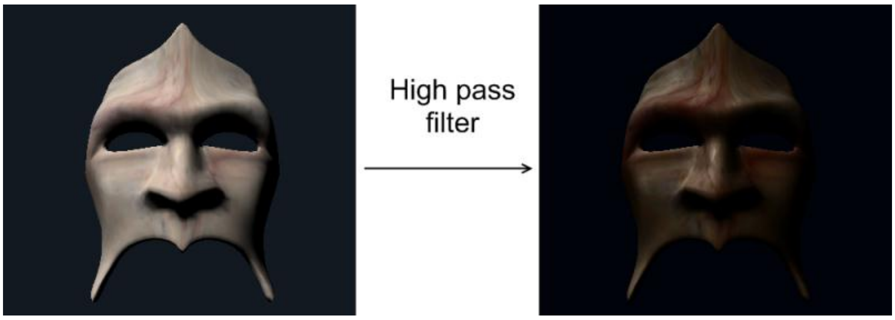

Figure 10. Applying a high pass filter to obtain bright parts

The high pass 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 applying a full shader pass over the whole texture and executing the filter for each texture element. Depending on the amount of blur iterations and the kernel size, most of the remaining high frequencies will be eliminated and a ghostly image will remain (see Figure 11). Furthermore, due to the blurring, the image is consecutively smeared and thus enlarged, creating the halos when combined with the original image.

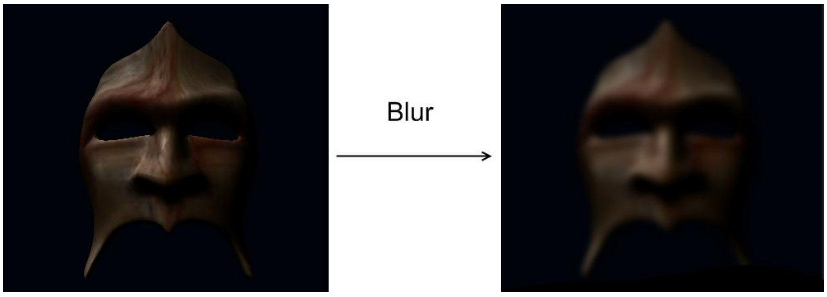

Figure 11. Applying Gaussian blur

The final step is to additively blend the resulting Bloom texture over the original image by doing a full screen pass. This brightens the already bright parts and produces the halo around glowing objects due to the blurring of the high pass filtered texture (Figure 12).

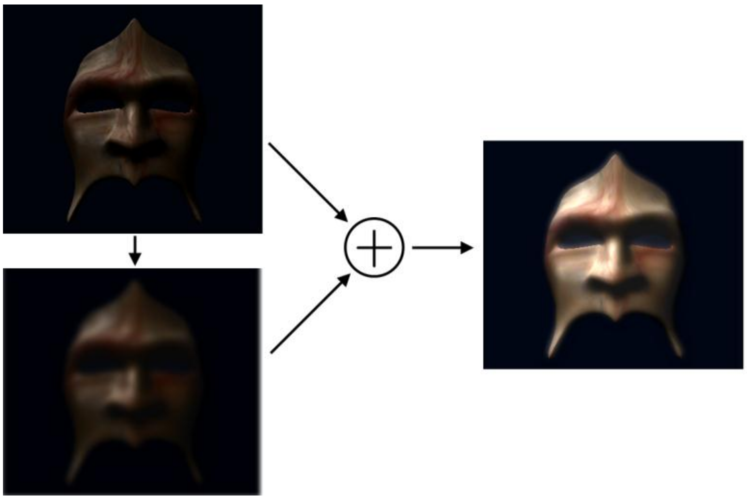

Figure 12. Additive blending of original and blurred high pass filtered image

The final amount of Bloom can be controlled by changing the blend function from additive blending to an alpha based blending function, giving even more artistic freedom. Again this alpha value can be used to animate the amount of bloom, e.g., to simulate a pulsing light.

3.2. Implementation

The Bloom algorithm presented in the previous section describes the general approach one might implement when processing resources are not very limited. Two full screen passes for high pass 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 processing environments, serious adjustments to the original algorithm have to be made in order to get it running on low profile hardware, even when paired with a highly efficient graphics core.

The end the result has to look convincing and be able to be run at interactive frame rates. Thus most of the steps illustrated in Section 3.1 have to be modified in order to meet the resource constraints. These alterations will be presented in the following subsections.

3.2.1. Resolution

One of the most critical limitations of mobile hardware graphics chipsets is the relatively low clock frequency available for processing.

Thus, the first and most important optimisation is to do the necessary calculations at a reduced resolution whenever possible, sacrificing image quality in favour of performance, but still producing convincing results.

For this training course the resolution of the frame buffer object for the blurred texture was set to 128x128 (width x height), which has shown to be sufficient for high-resolution QVGA (320x240) displays. Depending on the target device's screen 64x64 might be even adequate. The individual result has to be checked when running on the targeted device. It should be kept in mind that using half the resolution (e.g., 64x64 instead of 128x128) means a reduction of pixels being processed by 75 percent.

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

When implementing the first part of this algorithm by drawing the objects to the frame buffer object, one could take the optimization even a bit further and completely omit texture mapping (see Figure 13). This means that the vertex shader only calculates the vertex transformation and the lighting equation, which reduces the amount of data being processed in the fragment shader even further. On the other hand, the resulting glow will be monochromatic and it has to be evaluated for each different case if the performance gain is worth the visual sacrifice.

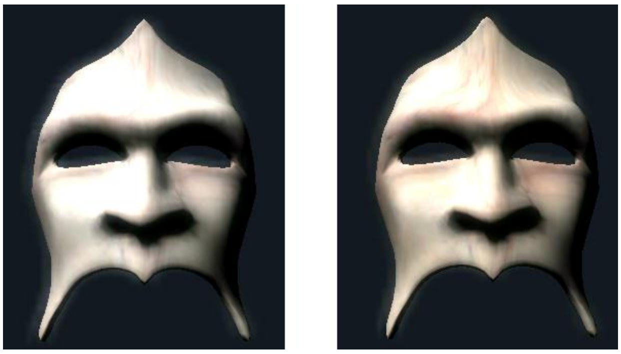

Figure 13. Difference between non-textured (left) and textured (right) bloom

The scalar output of the lighting equation represents the input data for the blur stage, but if we simply used this for the following steps the resulting image would be too bright, even in the darker regions of the input image. This is why the high pass filter has to be applied. This can be achieved by doing a texture lookup into a 1D texture, representing a direct mapping of luminance values to high pass filtered luminance values (see Figure 14). This texture can be generated procedurally by specifying a mapping function or manually, where the amount of bloom can be stylized to meet artistic needs. After the rendering of the objects to the lower resolution framebuffer object and the high-pass filtering of the output, the resulting image can be used as input for the blur filtering steps.

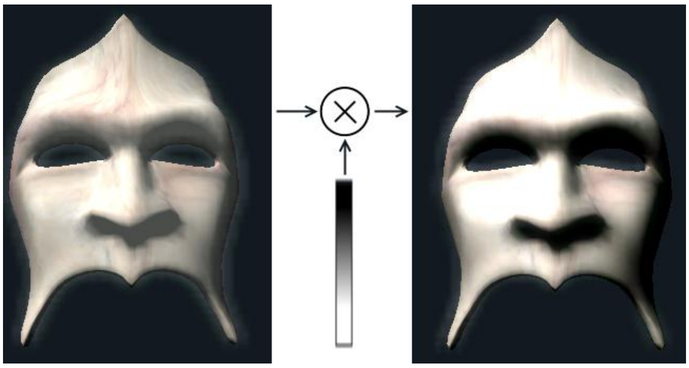

Figure 14. High pass filtering the brightness scalar by doing a texture lookup in an intensity map

3.2.2. Convolution

After generating the initial bloom texture we have to filter and blend it over the existing back buffer content. This section explains the blur filtering methods which are depicted in Figure 15.

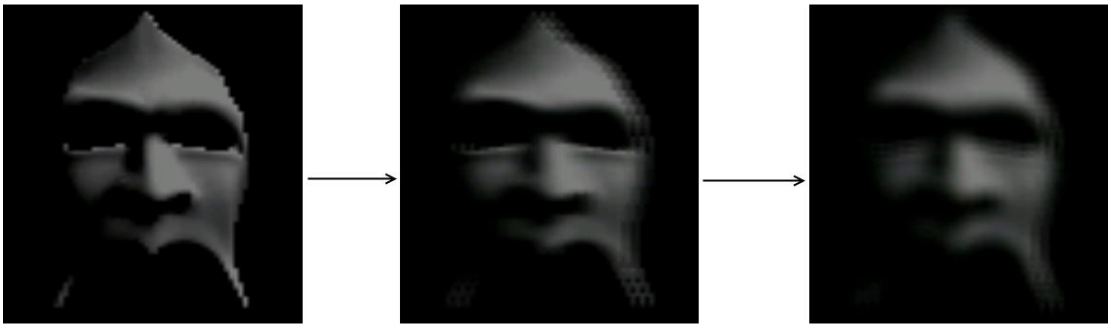

Figure 15. Blurring the input image in a low resolution render target with a separated blur filter kernel

Image convolution is a very common operation and can be executed on the graphics core very efficiently. 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.).

In this case we apply a Gaussian blur to smooth the image, which falls into the category of linear filters. 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 3x3 kernel this results in nine texture lookups, nine multiplications, eight additions and one divide operation, which is a total of twenty-seven operations to filter a single texture element. The normalization can be included in the weights, reducing the total operation count to twenty-six. Fortunately the Gaussian blur falls into the subcategory of separable filters, which means that the filter kernel can be expressed as the outer product of two vectors:

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 2 & 1 \end{pmatrix}$$

Making use of the associativity,

$$t * (v * h) = (t * v) * h$$

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 (Table 1). This reduction of operations is even more dramatic when increasing the kernel size (e.g., 5x5, 7x7, etc.)

Table 1. Result of applying vertical and horizontal filters

Kernel	Texture lookups	Multiplications	Additions	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

The amount of texture lookups can be even more decreased when using hardware texture filtering. In the case of the 5x5 filter kernel the amount of texture lookups can be reduced from 10 to 6, yielding the exact same amount of computation necessary as for the 3x3 kernel. The trick is to replace the texture lookup for the outer weights with one which is in-between the outer texels (Figure 16).

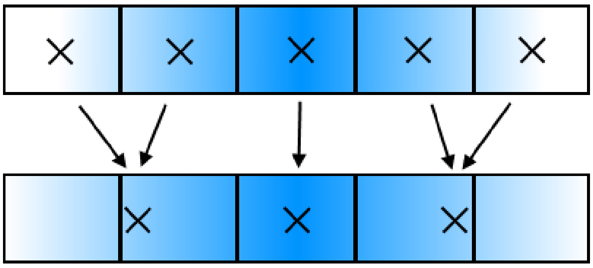

Figure 16. Reducing the amount of texture lookups by using hardware texture filtering

The shift itself is calculated with respect to the original weights and the hardware filtering will linearly combine the texels. The final step is to sum up the texture values and write the resulting value to the colour output. It is important that the texture coordinates, which are required for the texture lookups in the pixel shader, have to be calculated in the vertex shader or have to be passed as uniform shader variables in order to avoid dependant texture reads. Although these are supported, they incur a substantial performance hit. Doing no dependent texture reads means that the texture sampling hardware can fetch the texels sooner and hide the latency of accessing memory.

3.2.3. Blending

The last step is to blend the blurred image over the original image to produce the final result. This is illustrated in Figure 17.

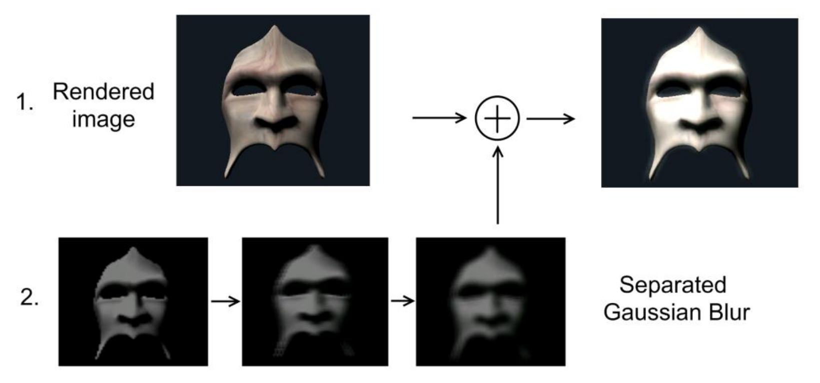

Figure 17. Overview of the separate Bloom steps

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, it is possible to setup an alpha value based modulation scheme to control the amount of Bloom in the final image.

The single most important optimization in this case is to minimize the blending area as far as possible. Blending is a fill-rate intensive operation, especially when being done over the whole screen. It is best to avoid drawing as much as possible, especially on mobile platforms. For example, when the Bloom effect shall be only applied to a subset of the visible objects, it is possible to optimize the final blending stage:

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

Figure 18 shows the blending rectangle which is derived from the objects 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 edges of the bounding box are transformed into clip space coordinates, where the minimum and maximum coordinate components are determined. The blending rectangle is then derived from these coordinates and the texture coordinates are adapted to the vertex positions. Depending on the shape of the object, more suitable bounding volumes might be appropriate, e.g., a bounding sphere.

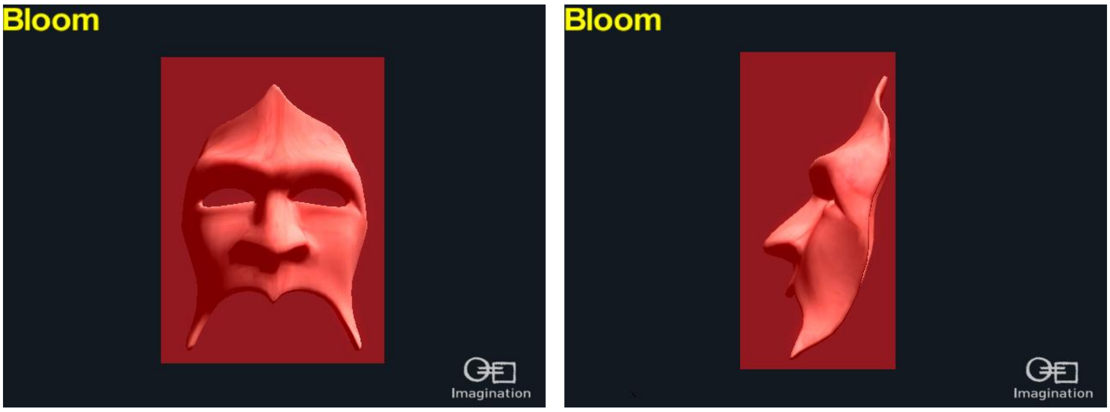

Figure 18. Bounding box derived rectangle (red) used for final blending

This bounding box derived blending can lead to artefacts when the blending rectangles of two or more objects overlap, resulting in sharp edges and highlights that are too bright. A workaround 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 blending is only applied to pixels where the stencil is zero and increment after the operation.
- Draw all bounding volumes and disable stencil test.

This prevents multiple blend operations to a single pixel and produces the same result as a single full-screen blend.

4. References

Further Performance Recommendations can be found in the Khronos Developer University Library:
http://www.khronos.org/devu/library/

Developer Community Forums are available at:
http://www.khronos.org/message_boards/

5. Contact Details

For further support, visit our forum:
http://forum.imgtec.com

Or file a ticket in our support system:
https://pvrsupport.imgtec.com

To learn more about our PowerVR Graphics SDK and Insider programme, please visit:
http://www.powervrinsider.com

For general enquiries, please visit our website:
http://imgtec.com/corporate/contactus.asp

Imagination Technologies, the Imagination Technologies logo, AMA, Codescape, Ensigma, IMGworks, I2P, PowerVR, PURE, PURE Digital, MeOS, Meta, MBX, MTX, PDP, SGX, UCC, USSE, VXD and VXE are trademarks or registered trademarks of Imagination Technologies Limited. All other logos, products, trademarks and registered trademarks are the property of their respective owners.