UI Example

Whitepaper

Filename	:	UI Example.Whitepaper
Version	:	PowerVR SDK REL_16.2@4266559a External Issue
Issue Date	:	12 Oct 2016
Author	:	Imagination Technologies Limited

Contents

List of Figures

List of Tables

1. Introduction

Rendering graphical user interfaces may seem like a trivial exercise. However, without careful consideration a simple interface may be unresponsive and sluggish due to poor graphics performance. The purpose of this document is to highlight and demonstrate techniques that can be used to improve user interface performance for software running on PowerVR platforms. Indeed, any generic 2D sprite-based applications can benefit from the recommendations given in this document including games, applications and interfaces.

2. User Interface Example

Almost all applications utilize some form of user interface and traditionally the graphics implementation of this has not received high priority. However, with the advent of mobile computing and the rise of graphics acceleration on these platforms, the implementation of user interfaces and two dimensional graphics has become more important.

While the PowerVR hardware architecture is an extremely efficient and powerful solution, some naive techniques may have a more significant impact on performance than developers may be aware of. These issues are discussed at length in the document “PowerVR Performance Recommendations” and such will not be covered in depth here.

2.1. Traditional Approaches

The traditional approach to drawing 2D graphics and interfaces is to simply loop through each element or “sprite” in the scene, generate geometry (most likely a quad), apply render states and then call the graphics API to render the geometry and supplied sprite texture. This approach indeed works but is far from efficient. Analysing the above sequence, there are a number of stages that can be modified to improve performance and efficiency (Table 1).

Table 1. Issues with traditional approaches and solutions

Issue	Solution
Geometry constantly generated/uploaded per sprite.	Use Vertex Buffer Objects such that geometry can be built once, uploaded and then stored in memory.
Render state calls per sprite.	Batch sprites which use the same states together.
‘Draw’ call for each sprite.	Use indexed triangles and one VBO to render batches of sprites with one API call.
Texture state changes per sprite.	Use a texture atlas in conjunction with the above solutions to remove texture state changes.

2.2. Geometry Considerations

2.2.1. Vertex Buffer Objects

In general, the recommended method of geometry storage for the PowerVR architecture is to store geometry data in a Vertex Buffer Object (VBO). This removes the need to constantly pass data over the system bus and for the drivers to allocate and store the data in an appropriate format.

There exist several forms of VBO storage. In theory, this allows for either static or dynamic data to be used. In practice, however, there are no performance benefits to using a dynamic VBO and their use may even be suboptimal for performance. If geometry data is to be constantly modified it is actually beneficial to forgo the use of a VBO in place of a simple vertex array, though as stated a performance hit should be expected.

It is, however, possible to utilise a VBO while allowing some transformation of vertices through the use of a graphics core vertex program. In this instance, geometry can be built and uploaded into a VBO and at render time a matrix palette, typically used for hardware skinning, can be passed as a `uniform` variable to the vertex shader to transform the VBO’s vertices. Using `attribute` variables, an index can be passed with each vertex which then allows a lookup in to the matrix palette which can then be used to modify the vertex.

Implemented, the matrix palette is an array of 4x4 matrices which is passed using the `glUniformMatrix4fv` function. On most platforms the maximum number of 4x4 matrix arrays allowed in the vertex shader is 32. This can be uploaded at the start of the frame and referenced by the vertex shader throughout the render. If more than 32 transformations are required draw calls must be batched to allow for the maximum matrix palette size.

Implementing the palette index is more straightforward. However, there is a small caveat: the OpenGL ES 2.0 specification requires attribute variables to be floating point data types. This means we must cast the array index to type `float` before submitting the attribute data and then cast it back to an `int` type in the vertex shader before attempting to index the matrix palette. Finally, the palette index is passed to the vertex shader just as any other attribute:

```
glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, sizeof(PVRTVertex), (const
void*)i32VOffset);
…
glVertexAttribPointer(TRANSFORM_ARRAY, 1, GL_FLOAT, GL_FALSE, sizeof(PVRTVertex), (const
void*)i32TransIdxOffset);
```

Note: i32TransIdxOffset and i32VOffset are offsets into an interleaved VBO and are not data pointers.

Using this technique, the performance benefit of using VBOs can be gained while still supporting dynamic, transformable objects.

2.2.2. Indexed Triangles versus Triangle Strip

Indexed triangle lists are recommended for PowerVR platforms over triangle strips, in general, and in the case of UIs they also have the advantage that they do not require an API call per quad/sprite, or the use of 'degenerate' triangles. If many sprites are to be drawn, as is common in user interfaces, then a triangle strip implementation may have hundreds of draw calls per frame. This is particularly poor as each call is quite expensive due to the hardware needing to lock and unlock various contexts and buffers.

A more optimal approach is to index each triangle and with one call to `glDrawElements`, any number of sprites can be rendered. This process of indexing triangles can be integrated into the building and submitting of geometry to a VBO. An example of building the indices is shown next:

```
#define INDICES PER QUAD (6)
PVRTQuad Quads[NumQuads];
GLushort u16Indices[NumQuads * 6];           // 6 indices per quad (to reference 2 triangles)
GLuint uiQuadIdx = 0;
for(int i = 0; i < NumQuads; ++i)
{
        BuildQuad(Quads[i]);

        GLushort u16Start = (GLushort)(uiQuadIdx << 2);      // 4 vertices per quad
        u16Indices[(uiQuadIdx*INDICES PER QUAD)+0] = u16Start+0;
        u16Indices[(uiQuadIdx*INDICES PER QUAD)+1] = u16Start+1;
        u16Indices[(uiQuadIdx*INDICES PER QUAD)+2] = u16Start+3;
        u16Indices[(uiQuadIdx*INDICES_PER_QUAD)+3] = u16Start+1;
        u16Indices[(uiQuadIdx*INDICES_PER_QUAD)+4] = u16Start+2;
        u16Indices[(uiQuadIdx*INDICES PER QUAD)+5] = u16Start+3;
}
```

The above arrays are then uploaded to OpenGL using the `glBindBuffer` function with either `GL_ELEMENT_ARRAY_BUFFER` or `GL_ARRAY_BUFFER` for indices and vertex data respectively. The call to `glDrawElements` would then look similar to this:

```
glDrawElements(GL TRIANGLES,
               QuadsToDraw * INDICES PER QUAD,
               GL UNSIGNED SHORT,
               (char*)0);
```

2.3. Multiple State Changes

Every call to OpenGL invokes a performance penalty, where the severity depends on the specific function. Thus the number of API calls should be minimised where possible. It is, therefore, prudent to batch similar geometry together and enable/disable OpenGL states a minimal amount of times while rendering the scene. This can easily be linked to the previously mentioned indexed triangle system. As `glDrawElements` takes an offset into the index buffer to begin rendering from, we can store specific quad offsets while building geometry and then when rendering the frame, pass the pre-calculated offset as the final parameter to `glDrawElements`. Modifying the previous code section to integrate this feature results in the following:

```
glDrawElements(GL_TRIANGLES,
               QuadsToDraw * INDICES_PER_QUAD,
               GL_UNSIGNED_SHORT,
               (unsigned char*)(StartQuad * INDICES_PER_QUAD * sizeof(GLushort));
```

This enables us to have only one VBO with all the sprite geometry stored in it, resulting in the advantage of less VBO creation time and fewer handles to track and manage. This system is integrated into the demo associated with this whitepaper. A dynamic array is used to store a determined amount of passes required for each group of sprites and then each pass is rendered with an associated render state. Render states should be tracked so that redundant state changes are minimised. This is also integrated into the demo (see function `UpdateRenderState()`).

2.4. Multiple Texture State Changes

Using texture atlases keeps the number of texture state changes to an absolute minimum. This is helpful for performance as using many single textures has a processing overhead due to texture bind operations and potentially inefficient use of memory caches for textures. Using a texture atlas is also necessary when batching ‘draw’ calls to the graphics API. If many textures are used, it is possible that only one sprite could be drawn per draw call due to the necessity to switch textures after each draw. Using an atlas works in conjunction with previous recommendations of batching geometry to keep the number of ‘draw’ calls to a minimum.

Generating a texture atlas (Figure 1) is not a difficult procedure but does require an algorithm for packing quads efficiently into a larger section. A simple algorithm for this is to utilise a binary tree and the source code for implementing this algorithm can be found in `‘OGLESExampleUI.cpp’`. The general idea is to use a binary tree to section up large quads into smaller sections. Each quad is a node in the tree and when a new quad is to be inserted, the tree is traversed to find either a quad which is a perfect fit, or a quad which is large enough to house the required area. This is subsequently sectioned.

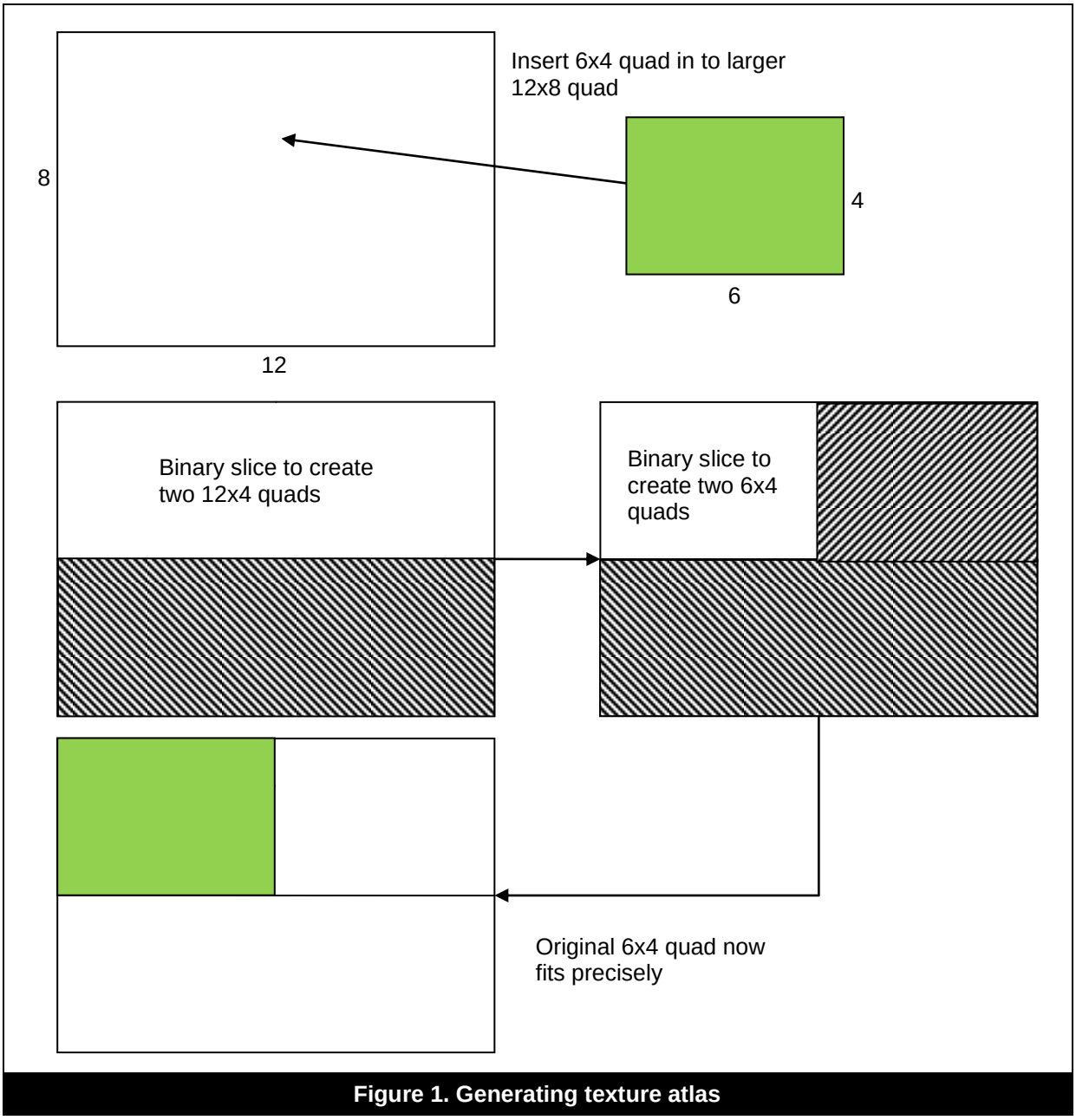

Figure 1. Generating texture atlas

3. Optimizations

Previously we have discussed various techniques for managing sprites and efficiently rendering them with a minimal impact on graphics performance. This section will discuss further techniques for performance optimizations.

3.1. Static Screens Using Frame Buffer Objects

One method for reducing the number of 'draw' calls and geometry bandwidth cost is to use pre-rendered sections of the interface which only require one 'draw' call and minimal geometry to process. Other optimizations can also be gained such as a reduction in blending, depth and stencil checking.

To implement this optimization a Frame Buffer Object (FBO) is utilised which is updated only when necessary and then used as a texture to be drawn to the main render applied to a simple quad. This is easy to add in to existing code as none of the original code need be modified. Only several calls to switch rendering to a user provided frame buffer need to be added before the geometry is to be rendered and then a single call to revert rendering to the main buffer. A trivial function can then be implemented which binds the texture associated with the frame buffer object and renders an appropriately sized quad to the screen. All that is left to implement is a simple check that only draws the interface to the framebuffer if something has changed or the interface needs updated, otherwise no rendering is necessary. Figure 2 next illustrated the flow diagram for utilising FBO for static elements.

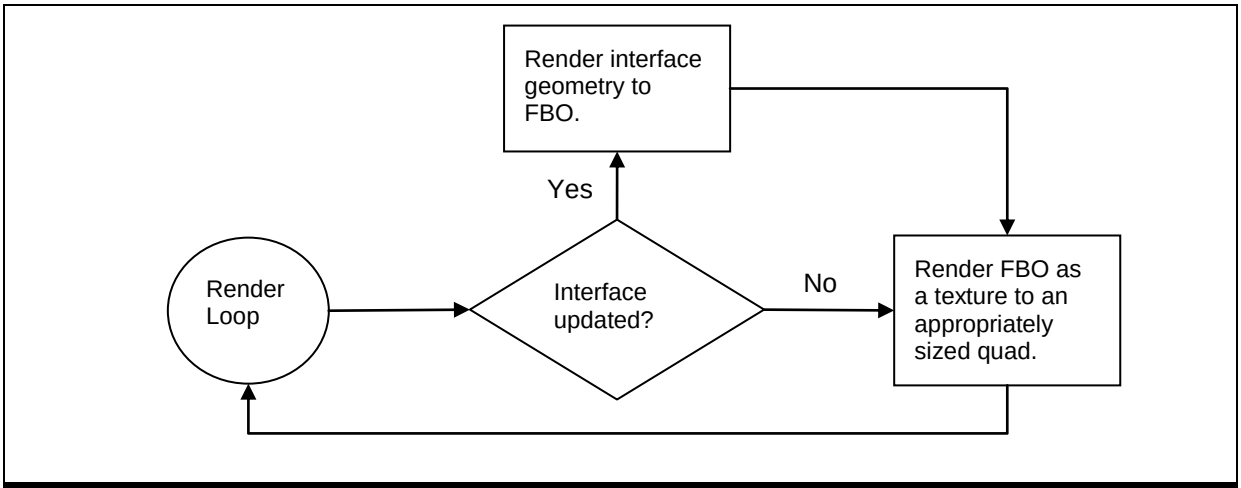

Figure 2. Flow diagram for utilising FBO for static elements

3.2. Disable Bilinear/Trilinear Texture Filtering

While texture filtering can drastically improve the quality of rendered images it is completely redundant when rendering interface and non-scaled sprites as these textures are predominantly mapped at a 1:1 pixel ratio between source texture and destination render. Disabling texture filtering (choosing `GL_NEAREST`) will improve application performance to varying degrees as the cost of filtering is based on the number of available texture units as well as available memory bandwidth.

Disabling texture filtering, however, has a noticeable effect on texture quality if pixels are not mapped 1:1 (i.e., scale or rotation is used) and in such cases it is probably beneficial to enable texture filtering. This optimization is heavily dependent on specific use-cases.

3.3. 2-Pass Rendering (Opaque/Translucent)

The architecture of PowerVR, TBDR (Tile Based Deferred Rendering), features pixel-perfect hidden surface removal such that if an opaque fragment is submitted, no other hidden fragments that are associated with this pixel will be processed. This is extremely powerful when rendering fully opaque geometry. However, some of this advantage is lost when translucent geometry is introduced.

To overcome this issue all opaque geometry should be drawn first, followed by translucent geometry. This is generally good practice regardless of architecture to achieve correct blending results. This is easily integrated with previously mentioned considerations such as batching geometry to reduce 'draw' calls and simply requires a number of indices to mark the beginning and end of opaque and translucent geometry within the vertex buffers.

3.4. Reduce Alpha Blending by Increasing Geometry Complexity

One method to minimise the impact of several layers of blended sprites is to increase the geometry complexity of the sprites in order to reduce the amount of wasted transparent fragments. For example, if a sprite is circular in shape and is rendered using the most optimal fitting quad, 22% of the geometry will still be wasted in processing the fully transparent fragments. Significant performance improvements can be gained by reducing the wasted transparency by increasing geometry complexity.

PowerVR graphics processes have excellent vertex processing capabilities and are designed to handle large amounts of geometry data, far in excess of what is present in most UI views. So increasing complexity should have minimal performance impact and any impact this may have will almost certainly be outweighed by the savings of rendering less transparency.

If we increase the complexity of the previous case of a perfectly fitting quad around a circular sprite to that of a dodecagon (twelve sided polygon) we can reduce the amount of wasted fragment processing to just 3%. Assuming radius `r` of 64, the expressions below then reveal the consequence of increasing complexity and reducing processing.

$$A = 1 - \frac{\pi r^2}{(2r)^2} \qquad \text{vs} \qquad A = 1 - \frac{\pi r^2}{12\left(2-\sqrt{3}\right)r^2}$$

$$A = 0.214 \qquad A = 0.029$$

$$A = 21.4\% \qquad A = 2.9\%$$

3.5. Use Stencil Buffer for Efficient Clipping

There are many instances when there is a necessity to clip certain aspects of the screen to prevent overdraw or restrict certain geometry to a particular section of the screen (i.e., inside a pop-up window). The traditional approach to accomplishing this task is to make use of a 'scissor' function such as `glScissor()` which culls any pixels outside of a defined rectangle.

Using a 'scissor' function, however, has a number of inherent flaws: not only is the scissor functionality confined to a screen-aligned rectangular shape but it might affect performance in some situations.

The solution to these problems is to use the stencil buffer which enables rendering on a per-pixel basis, effectively clipping subsections of the screen. Stencil tests are extremely fast on PowerVR graphics cores as they are designed to perform visibility determination at a very high speed on-chip and without using up precious system memory bandwidth (see the document on "PowerVR Hardware Architecture Guide for Developers" for more information). Contrastingly, extensively utilising the stencil buffer on other hardware implementations can incur fill-rate and bandwidth issues.

The process for implementing stencil buffer clipping is fairly simple and, like in Section 3.1, requires minimal modification of existing code. An overview of the technique is as follows:

1. Enable stencil test.
2. Clear the existing stencil buffer.
3. Set stencil operation to write '1' to the stencil buffer.
4. Render geometry to the stencil buffer that will be used to 'clip' or 'mask' future geometry.
5. Set the stencil function to only pass if a '1' is detected in the stencil buffer for a given fragment.
6. Render geometry as normal.

For the sake of simplicity OpenGL ES 2.0 code will be included to demonstrate the process:

```
glEnable(GL_STENCIL_TEST);

glStencilFunc(GL ALWAYS, 0x1, 0xFFFFFFFF);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
```

Stencil test should first be enabled and set to write a '1' in to the buffer in all instances. The last parameter to `glStencilOp()`, `GL_REPLACE`, instructs the graphics hardware to write the reference value (0x1) in any instance. It is wise to also set the colour mask at this point so that geometry rendered to 'clip' or 'mask' the section of the screen is not drawn to the colour buffer and would, subsequently, be made visible on the screen.

At this point in the program, geometry is rendered that will function as the mask. Unlike using 'scissor' functionality, this geometry can be as simple or complex as is necessary and is not restricted to being screen-aligned. Indeed any transformation matrix can be used to scale, rotate and manipulate the mask section. The mask is not limited to two-dimensional coordinates systems either. If interface elements are drawn with perspective projection, then using the stencil buffer to clip is the perfect solution as the mask geometry can be projected with the same matrix as the rendered geometry, giving pixel-perfect clipping in three-dimensions.

```
        glStencilFunc(GL_EQUAL, 0x1, 0xFFFFFFFF);
        glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
        glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
```

Finally once the mask geometry has been written to the stencil buffer the stencil function is set to only pass if the value in the buffer for a given fragment is equal to the supplied reference value (0x1). Colour buffer writing must be re-enabled if previously disabled so that further geometry will be visible. Now any geometry required to be clipped by the previous mask can be rendered and will, subsequently, be clipped by the stencil buffer.

```
glDisable(GL_STENCIL_TEST);
```

It is necessary to remember to disable stencil test once the geometry to be clipped has been rendered.

4. Further Considerations

In this section further optimizations will be discussed which are possible to implement to increase efficiency while rendering interfaces and sprite-based systems.

4.1. Further Sprite Optimization

The demo included with this whitepaper uses hardcoded sprite optimizations for a specific shape or sprite. In real-world examples this might not be practical due to the time taken to code a shape for each specific sprite. Further work could include an algorithm to procedurally generate optimised geometry according to Section 3.4. Care needs to be taken to not increase complexity to such a significant amount that the performance gain of less alpha blending is outweighed by the amount of geometry processing required. An example of optimised geometry would be as shown in Figure 3.

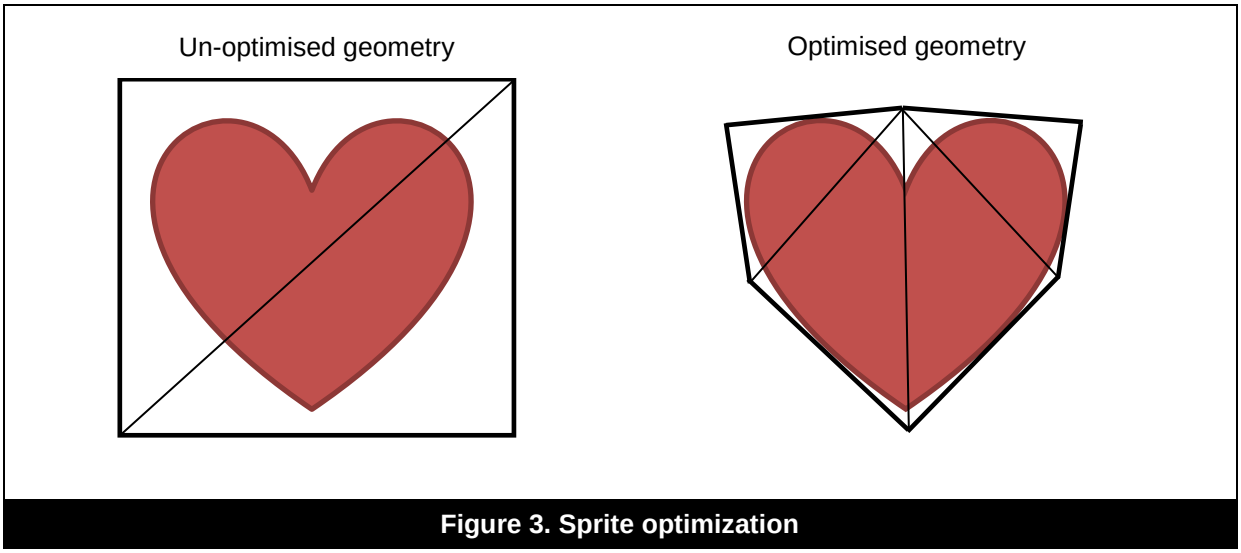

Figure 3. Sprite optimization

5. 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/

6. 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