Abstract—We present a simple method of achieving realistic extreme depth of field effects at real time rates with suitable hardware. The method combines the accumulation buffer with per-pixel stochastic sampling. We introduce a novel approach whereby the scene is drawn from a large number of different viewpoints simultaneously. The results avoid the multi-image effects that are present in earlier accumulation buffer methods, whilst still utilising the GPU hardware. The technique allows easy integration with other rendering algorithms. We present parallax mapping as an example, and show that correct “see-through” effects can be obtained within a parallax map.

Keywords — depth of field, real-time

I. INTRODUCTION

Depth of field, the distance range of objects within an image that are in focus, is due to the finite aperture of lenses. This aperture limitation results in any point in space being mapped to a ‘circle of confusion’, causing the blurring for those points outside the in-focus region (as shown in Figure 1). Computer graphics has often neglected the effect of the ‘out of focus’ parts of a scene. In the past this was due to performance issues - only computationally intensive ray-tracing could achieve these effects. More recently, the availability of programmable GPU cards has resulted in more interest in it, but complex methods have been needed to provide near real-time performance. Such complexity can work against general uptake of the techniques, particularly when they need to be combined with other effects.

This paper presents a simple method that allows per-pixel sampling to be performed in the context of the standard GPU pipeline. Real time performance is achievable for simple scenes on consumer-level hardware, and should be achievable for more complex scenarios by using high-end GPUs in a parallel configuration. We anticipate that in a relatively short time, Moore’s Law will enable complex scenes to be processed easily and economically.

The contributions from this paper are: (1) a simple model that does not require any arbitrary choices, (2) ease of integration with other effects due to its correct use of different viewpoints, (3) accurate rendering because the model is true to the physical situation, (4) a novel way of using the GPU to allow each pixel to use a different viewpoint.

A. Related Work

Depth of Field algorithms have been comprehensively reviewed by Barsky and Kosloff [1]. More recently, Jeong [2] summarised the research approaches in terms of multi-view sampling, single view sampling (and post processing) or some hybrid of both. For example, Lee [3], [4] used multiple depth layers on single view sampling whereas Jeong [2] combines depth layers with further performance improvements by considering level of detail. As the methods become more hybrid and complex, they are less likely to be more generally adopted, particularly when they may cause problems integrating with other required techniques.

The simple accumulation buffer technique [5] has only two defects: multiple images and the large number of samples. As GPU overall performance and programmability have been and are continuing to improve, it seems an appropriate point to use the performance for a larger number of samples, and the programmability to eliminate the multiple image problem. To this end we have therefore developed a new method of using the GPU to allow per pixel stochastic sampling.

II. METHOD

A. Overview

In a real lens system rather than a ’perfect’ pinhole, the rays of light from the scene can be considered to pass through any point within the aperture. The core of our new method is to find a good way to redraw the image multiple times...
from different points within the aperture. If such a process was applied naively, it would generate a sequence of separate multiple images rather than true depth of field. This would show up very quickly in extreme depth of field situations. Real lenses naturally avoid this by the integration of all the points within the aperture. A way to avoid this problem is to break up the coherence of the images by choosing different positions within the aperture as the eye point for each of the sample images. This would be a relatively simple process in ray tracing, as the whole geometry is available throughout the process. Unfortunately, in rasterization, the geometry information is not so accessible, as the transformations are done first, and the pixel operations are done afterwards.

In principle the use of a different aperture sample point within each pixel requires a complete re-run of all the geometry calculations. However, because the aperture has a limited size, each primitive has only a finite ‘footprint’ when considered across the range of sample points. By sending the whole footprint of the primitive to the rasterizer, we are able to defer the choice of sample point until the fragment (pixel) processing stage.

### B. Shaders

Figure 2 shows the sequence of operations through the shaders. In the vertex shader, we compute the circle of confusion for each vertex, based on the thin lens model.

Then, in the geometry shader, a new primitive is constructed from the input primitive. The new primitive is created as follows. First we find the upper and lower bounds of the primitive, taking into account the circle of confusion as shown in Figure 3a. In fact, it is possible to stop at this stage. However, in an attempt to reduce the number of pixels that will ultimately be discarded, it is also possible to trim the primitive using lines tangential to the circles of confusion as shown in Figure 3b. The consequences of this exercise are discussed later in the performance section.

The expanded primitive information is transmitted as a set of triangles from the geometry shader to the rasterizer via the built in GLPosition variables. The original primitive is transmitted in OpenGL flat form from the geometry shader to the fragment shader. The information transmitted for each triangle is the plane equation of the surface and the line equations of the edges of the original (i.e. unexpanded) triangle in texture coordinates.

In the fragment shader the first task is to compute the line of sight to be used. At the aperture end, we choose a random starting point. We also randomly displace the other end of the line of sight (by a small amount) in order to provide anti-aliasing for in-focus objects. However for correct z buffer operation, we have to choose the same random points for any line of sight within each rendering pass. Thus the sample points are different for each pixel within a given rendering pass, but when one visits the same pixel in that pass for different polygons, the sample point used must be the same.

The next task is to determine whether the viewing point is in front of or behind the surface. This of course can vary depending on which sample point is being used. We then compute the intersection point between the line of sight and the triangle, and transform that point into texture coordinates. The point is then tested against the edges of the triangle and the pixel is discarded if found to be outside the triangle. Any other linearly position-dependent quantities within the triangle can be derived as a linear function of the texture coordinates (tangent space).

Our implementation uses OpenGL, incorporating features up to version 3.2. All images, timing and videos have been generated on an MSI/NVidia GeForce GTX660 running at standard rates without overclocking. All images were generated at a resolution of 1024 by 768.
Our approach does not permit the standard perspective corrected interpolation to be used, as the aperture may cross the plane of a polygon, resulting in a singularity. To replace this functionality, we can compute the gradients and offsets of any other interpolated quantities, allowing them to be calculated accurately for each sample. It is worth noting that \( \frac{df}{dx} \) and \( \frac{df}{dy} \) functions cannot be used because of the pixel to pixel randomness. However, these values are available directly from the transformation if required.

To illustrate this capability to incorporate other algorithms, we have implemented steep parallax mapping [6]. The adjustable number of steps in this version of the algorithm also provides a useful performance test.

### III. RESULTS AND DISCUSSION

To show accurate depth of field effects, a system must satisfy the following requirements:

- Blur the edges of an object that is in front of or behind the focal plane
- Allow the viewer to see through foreground blur to the in-focus elements
- Show the correct visibility of oblique surfaces when extreme depth of field is applied (as illustrated in Figure 4).
- Avoid artefacts due to the simplification techniques used

In addition, a real-time method must show adequate performance under a range of conditions.

#### A. Visual Quality

Figure 5 shows a sequence of images of some children’s blocks. Note that as the front blocks become steadily more blurred, the see-through effect increases until, with an extremely blurred foreground, the distant child’s bed can clearly be seen due to the aperture width.

As a stress test, we created a ‘pathological’ yet simple object, with multiple long flat ‘legs’, with sides of different colours. Figure 6 shows the object, then a view with a single viewpoint, followed by a view using our new method. Notice that the new method allows us to see part of the right hand (magenta) side of the rectangle, and through on the left hand side to the base flat (green) surface. These are as suggested by Figure 4. If an alternative method had been used that just took a few samples around the edges of the aperture, then an object such as this would generate artifacts whereby some of the viewpoints would be blocked by adjacent ‘legs’.

The image also shows that the technique avoids the other problems enumerated above.

Figure 7 shows that the method is able to show depth of field blurring when used in conjunction with parallax mapping. The section of the image enlarged in Figure 8 also shows that the see-through effect can be achieved even within the parallax map.

#### B. Performance

Graphics performance can be measured in a number of ways. Frame rate is the basic parameter, but needs to be related to scene complexity. Polygon count is the most obvious measure of complexity, but resolution, depth complexity and the sophistication of the algorithms used on a per-pixel level must also be considered. Our method introduces extra workload compared to an in-focus pinhole camera model in a few ways. Firstly, there is the requirement to accumulate multiple samples per frame. There is an obvious trade-off between speed and quality here. Previous researchers (e.g. [3]) have suggested that with stochastic sampling, a sample count in the range of 16-32 is usually adequate. For the rest of this discussion, we will use the total sample frames per second (sfps) as the measure. Real time performance at 25 fps and 16 samples requires a rate of at least 400 sfps. Table I shows the impact of the method on the child bedroom scene as used in Figure 5. The full scene is as shown in Figure 9, and the figures are averaged over an automated low level pan around the room.

Secondly, there is extra workload caused by redraw resulting from the expanded footprint of each triangle. This will tend to increase as the polygon count rises, as each overlaps its neighbours. The impact of this can be seen in Table I. Here the higher polygon count scenes showed significant reduction in sfps when very blurred. However, it should be noted that when only the foreground or only the background was blurred, the figure remained near to that required for real time use. Unfortunately, our attempt to reduce this effect by trimming the footprint as in Figure 3b resulted in an increase of approximately half in the processing time. We are unsure as to why this happens; to resolve this will require analysis of the GPU behaviour.

The extra computation caused by the processes of the algorithm itself must also be considered. Comparison between the 200 polygon scene and the 9000 polygon scene without blur shows a decrease of approximately one third. This indicates that the computation cost of pixel drawing must dominate over that of polygon processing. Since the most complex parts of our algorithm fall within the vertex and geometry
shaders the extra workload associated with them cannot be a
dominant factor.

As an alternative to large polygon counts, parallax map-
ing can provide highly detailed images with relatively few
polygons. Since overdrawn pixels are discarded early, the
fragment shader can contain complex algorithms without
incuring any extra penalty. This is illustrated by Figure
10, which shows the effect of increasing the number of
steps in the parallax mapping on the overall time to execute
a render. The image used for these measurements is that
shown in Figure 7. Since the parallax mapping code is
performed entirely after the new depth of field algorithm,
then the overall time for render is the sum of both. If the
new algorithm was the main contributor to the time taken
to render, then any changes to the parallax mapping would
have little effect on the overall time. However, it is evident
that the time taken is significantly affected by the number of
steps in the parallax mapping, showing that the contribution

<table>
<thead>
<tr>
<th>Faces</th>
<th>Scene Blur Used</th>
<th>SFPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>None</td>
<td>667</td>
</tr>
<tr>
<td>200</td>
<td>All</td>
<td>480</td>
</tr>
<tr>
<td>9000</td>
<td>None</td>
<td>486</td>
</tr>
<tr>
<td>9000</td>
<td>All</td>
<td>138</td>
</tr>
<tr>
<td>9000</td>
<td>Foreground</td>
<td>367</td>
</tr>
<tr>
<td>9000</td>
<td>Background</td>
<td>340</td>
</tr>
</tbody>
</table>
from the new depth of field algorithm is low.

**IV. CONCLUSIONS AND FUTURE WORK**

The technique shows that, with the advent of more programmable graphics cards, we can now use novel techniques to achieve in real-time what was once the province of offline ray tracing. Our method is capable of generating the correct form of blurring for extremely out of focus objects, with visibility of oblique faces, no gaps where faces meet, and no multiple images.

We have also shown that it can be combined with parallax mapping, and allows correct blurring within a parallax map. We believe that the simple structure of our method should allow other complex fragment shader algorithms to be incorporated easily.

Real time performance is achievable for modest polygon counts on relatively inexpensive hardware even for extremely blurred images such as the rightmost image of Figure 5. At present it is not possible to process models with very large numbers of (inevitably very small) polygons in real time. However it is worth noting that the use of such models is unnecessary, since parallax mapping can provide the same quality for much lower polygon counts. We believe that where extreme depth of field is a priority, our method provides an easy to implement and practical solution.

We have not yet explored the possibilities of improving realism by manipulating the pseudo-random sample patterns to model lens behaviour accurately. This should be simple to do. There may also be scope for improved performance by adjusting the way in which the algorithm is implemented on the GPU, and comparing the behaviour of different GPUs. Another obvious extension to this method is to apply it to motion blur.

**ACKNOWLEDGMENT**

Thanks to Andy Burton, Tim Randall and James Lewis for models, and Morgan McGuire for the parallax mapping textures.

**REFERENCES**


