REAL-TIME AMBIENT OCCLUSION ON THE PLAYSTATION3

Dominic Goulding[1], Richard Smith[1], Lee Clark[1], Gary Ushaw[2] and Graham Morgan[2]
[1]*CCP Games, Gateshead, U.K.*
[2]*School of Computing Science, Newcastle University, Newcastle, U.K.*
{dgoulding, rsmith, leec}@ccpgames.com, {gary.ushaw, graham.morgan}@ncl.ac.uk

Keywords: Ambient Occlusion, Playstation3, Graphics.

Abstract: This paper describes how to implement ambient occlusion effects on the Playstation3 (PS3) while alleviating processing demands on the GPU. The solutions proposed here are implementations that utilize the parallel processing available on the PS3's synergistic processing units (SPUs). Two well-known ambient occlusion techniques are evaluated as candidate solutions for PS3 SPU implementations.

1 INTRODUCTION

Ambient occlusion (AO) is a technique for enhancing the perception of three-dimensional space in computer graphics. The technique enhances an image via the shadowing of ambient light. The accentuating of small surface details and the provision of spatial clues via contact shadows provide an increased degree of realism (Hoberock and Jia, 2008) (McGuire et al., 2011). This makes ambient occlusion a popular technique in the context of film and video games (Loos and Sloan, 2010).

To achieve ambient occlusion in real-time an approach based on approximation is required. Such techniques are convincing on the latest graphics cards and allow the modern PC gamer to enjoy the heightened realism afforded by ambient occlusion. However, as current console graphic card technology is dated the ability to achieve convincing ambient occlusion in console games is difficult.

The Playstation3 (PS3) does provide an opportunity to move some graphics calculations away from the graphics card and onto its Cell Architecture (which consists of 8 processing units). However, the Cell Architecture affords quite a different processing style than a GPU. This requires a different approach to implementation and re-integration to a graphics scene (generated by the GPU).

In this paper we describe an engineering approach to achieving ambient occlusion on the PS3. As such, we are not proposing a new technique in ambient occlusion but are proposing an implementation suitable for deployment on the Cell Architecture.

2 BACKGROUND

2.1 Playstation3 Architecture

The Playstation3's CPU architecture is cell-based, consisting of six Synergistic Processing Units (SPUs) around the central processor (plus a further two which are not accessible to the developer). These cells have a limited amount of memory (256k) for combined program and data, and the DMA access to this memory is comparatively slow. Efficient programming of SPUs is therefore reliant on identifying jobs which can run independently within that memory, with infrequent calls on main memory. The SPU processors are single instruction multiple data (SIMD) devices.

2.2 Ambient Occlusion

Ambient occlusion is defined as the amount of ambient light that is able to reach a point, which is not occluded by other points (i.e. it simulates the shadowing caused by nearby objects from indirect light). This can be achieved by casting 'rays' from the point, and determining if these rays are obstructed. AO is then calculated as the integral of a visibility function over a unit hemisphere (Loos and Sloan, 2010).

Screen Space Ambient Occlusion (SSAO), is a technique used to approximate the obscurance integral (Shanmugam and Arikan, 2007). Implementations of SSAO in games use a point sampling technique to approximate the occlusion integral. This involves computing the obscurance for each pixel on screen by taking samples around the pixel. The corresponding depth information from the depth buffer is then used to compute how much of a surrounding neighbourhood of the point in the scene is obscured by objects.

Whilst making the assumption that the falloff function is constant allows for efficient calculations of the obscurance integral, it is also possible to select a specific falloff function for an efficient implementation that maintains the complexity of the full radiometric model (McGuire et al., 2011).

2.3 Contribution of Paper

This paper shows that it is possible to move ambient occlusion calculations to the Playstation3's SPUs, freeing up processing time on the GPU, without a noticeable reduction in quality. The paper introduces a method for distributing full-screen ambient occlusion into ”SPU-sized” chunks of calculation. Two techniques for achieving ambient occlusion are implemented and compared - line-sampling, and taking a specific fall-off function - both shown to be viable approaches on the Sony hardware. A number of optimisations, taking advantage of the Playstation3 architecture, are also presented.

3 IMPLEMENTATION

Both the line-sampling technique (Loos and Sloan, 2010) and (Ownby, 2010), and the technique of taking the specific falloff function (McGuire et al., 2011) were implemented. The line sampling algorithm (which only requires the depth buffer values), is advantageous due to the limited local memory of each SPU. Whilst this technique boasts reduced sample counts in comparison to point sampling, further reductions in the sample count can be made by using the fall-off function.

3.1 Performing Calculations on the SPUs

A GPU based implementation uses a fullscreen 32bit depth buffer for SSAO calculations; at 1280 × 720 screen resolution, this requires approximately 3.5MB. However, each SPU on the Playstation3 has 256kB of local memory and can only access external memory through direct memory access (DMA), which can have a significant delay between requests and completion (Engstad, 2010).

Splitting the screen into sections and performing SSAO calculations on a block at a time is not a viable solution, as pixels at the edge of a block will not have access to the required depth buffer samples. This issue also occurs at the edge of the screen in a fullscreen implementation, however this can either be solved by rendering to a slightly larger image and cropping (McGuire et al., 2011), or ensuring that samples outside of the screen return a very large depth value, meaning they never contribute to occlusion (Filion and McNaughton, 2008). Storing the full screen depth buffer in main memory and then using DMA calls for each sample when it is needed is also inefficient due to the large number of DMA transfers.

3.1.1 Arranging the Input

Whilst the Playstation3 allows access to the depth buffer, the data are stored in a specific tiled format that is not suitable for our SSAO calculations. Before reading the information for the SPU tasks this tiled depth buffer must be reordered into a linear buffer. This was performed in a pre-pass rendering stage, storing the detiled depth buffer in main memory.

The next step is to arrange the data for concurrently running SPUs. The screen was split into 64 × 64 pixel tiles, with each SPU calculating occlusion values on a single tile at a time. A 128 × 128 tile of depth information was read from the pixels surrounding and including the inner 64 × 64 tile. We therefore restricted samples to a maximum of 32 pixels away from the target pixel. While this does cause a slight loss of accuracy for occlusion values, particularly with objects very near to the camera, for the majority of cases this restriction was not noticeable (indeed, this problem was further reduced by using downsampled buffers, described below). Depth values overlapping the screen edges were set to very large depths, as in many full-screen implementations. This approach means that there is an overlap of reads from the depth buffer, but no overlap when writing to the buffer that stores the occlusion values.

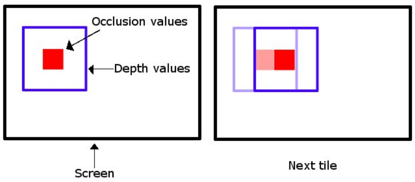

Figure 1: Arranging the depth information for input.

Using this configuration each SPU was required to store a 128 × 128 buffer of 4 byte depth values, and a 64 × 64 buffer of single byte occlusion values. Both of these were ‘double buffered’ (see below), meaning that approximately 140kB of local memory was needed. This is comfortably within the 256kB local memory available to a SPU. For any DMA transfer of more than 16 bytes the size of the transfer must be a multiple of 16 bytes, and must be aligned on a 16 byte boundary (Augonnet, 2007). The size of tiles used ensures safe DMA transfer requests for each tile.

3.1.2 Combining with Lighting

The occlusion values must now be combined with the current scene lighting. The occlusion values from each of the SPUs' local memory is written to a single external buffer in the RSX graphics memory. This is a little slower than writing to main memory, but only the GPU requires access to these occlusion values, so storing the values in RSX memory reduces main memory usage whiles providing fast access to the occlusion values during the lighting stage.

The occlusion values are stored as a single byte for each pixel. They are read as values from zero to one and combined with the scene lighting in a pixel shader by multiplying each pixel's colour value by the corresponding occlusion value. This has the effect of dimming the lighting where AO occurs.

3.2 Optimizations

3.2.1 Downsampling

The first optimization was to perform calculations on a $\frac{1}{4}$ sized buffer. The depth buffer was downsampled while it was detiled, so the reduced buffer was stored in main memory. SSAO calculations were then performed at this reduced resolution and output to a texture of the same resolution. This reduced the memory used, and also significantly reduced calculation times.

Downsampling also had the advantage of increasing our sampling range in screen space. Each SPU still performs calculations on the same number of pixels, but with a downsampled buffer these pixels correspond to a larger amount of screen space. Taking samples from a maximum of 32 pixels away is equivalent to taking samples up to 64 pixels away in a full resolution buffer, allowing for a wider ambient effect and improving the accuracy of occlusion results for nearby objects in the scene.

Using a reduced resolution buffer is a common way to increase the performance of SSAO algorithms, providing a significant performance increase with only minimal loss of detail. The decision whether to use a fullscreen or downsampled buffer is a compromise between performance, memory and visual quality, and will be application specific.

3.2.2 Double Buffering

A further performance increase came from double buffering, allowing each SPU to perform occlusion calculations on its current tile at the same time as sending its previous tile's occlusion results and retrieving the next tile's depth information.

A pair of input buffers (for the depth values) and a pair of output buffers (for the occlusion values) were created, which were then alternated so that at any specific time, one of each is being used for memory transfer while the other is being accessed by the AO calculations. Each SPU requests a DMA transfer of the previous tile's occlusion values to main memory, and a further DMA transfer to fill the free input buffer with the next tile's depth information. While these transfers are occurring, the SPU uses the current depth information to calculate the occlusion values. In this way a SPU task does not have to wait for DMA requests to complete before being able to perform calculations on the current tile, improving performance.

3.2.3 Single Instruction, Multiple Data

The SPUs are capable of single instruction multiple data (SIMD) operations (Gschwind, 2006). As the SSAO algorithms perform the same operations on each pixel in turn, the code was adapted to perform SSAO calculations on four pixels at a time using SIMD. Downsampling the depth and normal buffers, and blurring results were also performed using SIMD instructions. As SIMD instructions were included throughout the SPU code there was an improvement in the calculation time by approximately four times.

3.3 Bilateral Filter

To smooth the results a 2D Gaussian blur was applied which split the buffer into horizontal strips, followed by vertical strips, allowing a single SPU to perform the blur calculations one strip at a time.

While a Gaussian blur successfully smooths the results, it causes artifacts (known as 'halos') in the image. This effect occurs due to occlusion values bleeding across the edges of objects (Filion and McNaughton, 2008). This lightens areas that should be dark due to occlusion and darkens edges that should not be occluded. To reduce the halo effect we used a bilateral filter, where every sample is replaced by a weighted average of its neighbours' depth values (Elad, 2002).

4 RESULTS AND EVALUATION

The two SSAO techniques were implemented using the Playstation 3's SPU architecture.We also implemented fullscreen and downsampled versions for both approaches. We found that using 12 samples for the volumetric obscurance method and 6 samples for the falloff function method provided similar performance and both produced a good quality image.

4.1 Visual Quality

Visually comparing the two implementations at both fullscreen and downsampled resolution shows that all versions are of a high quality. In the game Uncharted 2 (Hable, 2010), where SSAO was also implemented on the SPUs, the two visual downfalls were a visible cross pattern of occlusion, and halos around objects. Neither of our implementations suffer from a cross pattern, whilst halos have been significantly reduced.

The volumetric obscurance method gives defined occlusion values with little noise, and appears to be of a similar quality to that seen in Toy Story 3: The Game (Ownby, 2010). However this method suffers from creating occlusion values that are focused around the edges of objects, not providing wide area results as seen in the falloff function method.

The falloff function method provides the best results for wide area ambience, capturing a greater sense of the overall geometry of the scene. As this method uses the surface normals, it is also able to highlight details from the normal maps of the objects' surfaces. The falloff function method can however suffer from objects causing self-occlusion (this is consistent with the findings of (McGuire et al., 2011)), and the image as a whole is more noisy than that seen in the volumetric occlusion method.

4.2 Performance

In our implementation, both SSAO techniques can run on as many SPUs as desired. Whilst performance obviously improves with more SPUs, it is unlikely to be possible to allocate all 6 SPUs to perform SSAO calculations. Table 1 shows performance results, measured in total SPU time. In our fastest implementation (using the falloff function at a downsampled resolution), occlusion results were calculated in 46.8ms, if all six SPUs are assigned to this task therefore, the time it takes for this task to complete is 7.8ms.

Table 1: Performance figures showing total SPU time.

	Volumetric Obsc.	Falloff Fn.
Downsampled	55.2*ms*	46.8*ms*
Fullscreen	209*ms*	166*ms*

We also created a version using only the PPU, no tiling was required (much like a GPU implementation) and we tested on a fullscreen resolution buffer. This was significantly slower, increasing frame render times by over 1500ms.

Whilst all four of the SPU implementation results allow for fully interactive frame rates in test levels, the large increase in speed seen in the downsampled methods make them much more desirable than the fullscreen equivalent. Our fastest result remains slower than GPU implementations however. Despite this,our implementation has the advantage of not impacting GPU performance, giving a trade-off between 2.3ms of GPU time and 7.8ms of CPU time.

5 CONCLUSIONS

We have successfully achieved high quality SSAO effects using the Playstation3's SPUs, confirming that this technique can be used as an alternative to GPU-based implementations. Our techniques are currently viable in an application that is hindered by GPU performance, but not with CPU performance.

REFERENCES

Augonnet, C. (2006-2007). An introduction to IBM cell processor.

Elad, M. (2002). Algorithms for noise removal and the bilateral filter.

Engstad, P.-K. (2010). Introduction to SPU optimizations. Naughty Dog.

Filion, D. and McNaughton, R. (2008). Starcraft 2 effects and techniques. In *Advances in Real-Time Rendering in 3D Graphics and Games Course*, SIGGRAPH.

Gschwind, M. (2006). The cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. Technical report, IBM Research Division.

Hable, J. (2010). Uncharted 2: HDR lighting. Game Developers Conference.

Hoberock, J. and Jia, Y. (2008). High-quality ambient occlusion. In *GPU Gems 3*. Addison-Wesley.

Loos, B. J. and Sloan, P.-P. (2010). Volumetric obscurance.

McGuire, M., Osman, B., Bukowski, M., and Hennessy, P. (2011). The alchemy screen-space ambient obscurance algorithm. In *High-Performance Graphics 2011*.

Ownby, J.-P. (2010). Toy Story 3: The video game rendering techniques. In *Advances in Real-Time Rendering Course*, SIGGRAPH.

Shanmugam, P. and Arikan, O. (2007). Hardware accelerated ambient occlusion techniques on GPUs. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 73–80. ACM.