Faster GPU Computations Using Adaptive Refinement

Craig Donner Henrik Wann Jensen
University of California, San Diego *

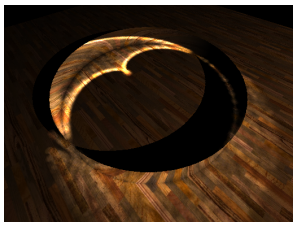
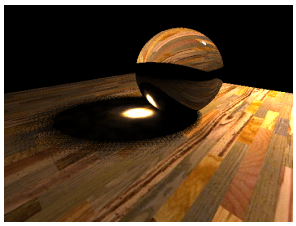

Figure 1: *Examples of scenes from [Purcell et al. 2003] rendered using our methods. Using adaptive refinement, the left image was rendered five times faster, and the right image was rendered four times faster.*

1 Introduction

We present a technique for improving the speed of multi-pass GPU computations by using adaptive refinement. We tile the screen and use occlusion queries to adaptively cull inactive parts of the computation. An implementation of this technique in a photon map renderer and a Mandelbrot fractal has resulted in speedups of up to one order of magnitude. Our technique is applicable to many of the recently developed multi-pass algorithms running on GPUs. It is easy to implement and often provides significant speedups by exploiting computational similarity, coherence, and locality.

2 Methods

We propose a method that utilizes the occlusion query feature present on modern GPUs to adaptively tile *sparse multi-pass computations*, increasing their overall performance. We define a sparse multi-pass computation as one where simple 2D primitives are rasterized to the viewport to generate fragments that execute shaders to perform computation, and where some parts of the solution are known before others. Most recent GPU algorithms (e.g. [Bolz et al. 2003]) can be classified as such.

A common method for generating fragments is to render a single viewport-sized quadrilateral, producing one fragment per pixel, and stopping when no fragments are active. The large quad is poor choice for sparse algorithms since it wastes processing power on fragments that do not perform useful work. Inactive fragments often execute a `KILL` instruction to indicate that their results should be ignored. Because the `KILL` instruction does not prevent the execution of the fragment shader, but rather the writing of its output, computation continues to occur over all fragments. The computation is done when all fragments execute a `KILL` instruction; this is detected using the occlusion query.

Our goal is to execute fragment shaders that produce useful results in every pass. By dividing the viewport into smaller fixed sized tiles we can increase the granularity of occlusion queries and thereby the efficiency of the computation. This intuition has its limits: many small tiles have higher instruction overhead and require more occlusion queries. By refining from larger to smaller tiles in areas where the solution converges quickly, we adaptively isolate the active areas of the viewport (see figure 2).

We keep a queue of active tiles with an associated occlusion query and active fragment count. After each pass of the computation, we perform an occlusion query on each tile. Once a tile no longer has any active fragments, we simply remove it from the active queue.

*{cdonner, henrik}@graphics.ucsd.edu

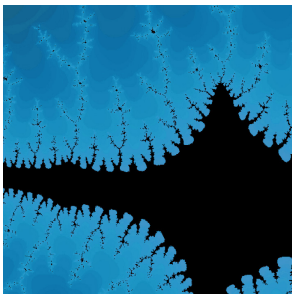
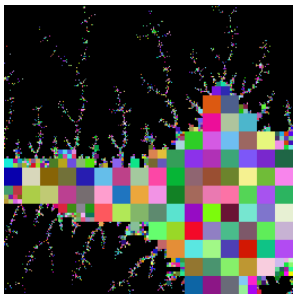

Figure 2: *A fragment shader implementing an iterative Mandelbrot Set solver. The left image is a visualization of the result, and the right image shows the active tiles.*

We have experimented with both points and quads when tiling the screen. A quad can be a single pixel pixel up to the full size of the viewport. Points are square-shaped and generally have limited size, which means more points are required to tile the viewport. Conceptually, points are easier to draw: only one vertex (rather than four) needs to be transformed, and the point can potentially be rasterized with a pixel template of some kind in screen space. Setting different point sizes when doing adaptive refinement, however, requires changing rendering state, which is known to be inefficient.

3 Results

To test our methods, we have used a GPU-based photon mapping implementation using the stencil sort routing method [Purcell et al. 2003]. To investigate the effectiveness of adaptive refinement for simple algorithms, we also implemented a small iterative Mandelbrot fractal fragment shader.

Our results have varied from small gains to speedups of almost three orders of magnitude, depending on tile size and the algorithm considered. In general, the results show significant speedups compared with the standard method that uses a single viewport-sized quad. The optimal tile size depends on the algorithm itself; slightly larger tiles work better with smaller fragment programs that finish quickly, while smaller tiles may be better suited to complex algorithms with long fragment programs.

When using a fixed tile size, our results indicate that the best tile size is from 8 to 32 pixels wide on current hardware. For adaptive refinement, our results show that subdividing when 40%-60% of fragments are active gives the best results. In addition, we found that setting a minimum tile size of 4 to 8 pixels wide prevents overloading the hardware; smaller tiles slow down the computation due to the instruction overhead as well as limited bandwidth from the CPU to the GPU. We did not find significant speed differences between using points or quads.

For very sparse computations with many passes, the use of adaptive refinement can result in significant speedups. As an example, we observed that the per-pass speedup in the last stages of the photon mapping renderer approached three orders of magnitude.

References

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRÖDER, P. 2003. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. In *Proceedings of ACM SIGGRAPH*.

PURCELL, T. J., DONNER, C., CAMMARANO, M., JENSEN, H. W., AND HANRAHAN, P. 2003. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Graphics Hardware*.