Ray Tracing Arbitrary Objects on the GPU

Andrew Wood, Brendan McCane, and Scott A. King

University of Otago, Department of Computer Science.
{awood,mccane,sking}@cs.otago.ac.nz

Abstract

Adapting ray tracing algorithms to programmable graphics hardware has gained momentum recently by using the parallelism of the GPU to reduce the work done on the CPU. GPU methods for ray tracing scenes consisting of only triangles have since been proposed. In this paper, we present a general method for ray tracing objects other than triangles on the GPU. Using more advanced primitives allows for faster and better results by reducing the number of intersection tests, as well as providing a more accurate representation of the surface. Realistic images can then be produced containing complex shapes without being limited to a model completely made up of triangles.

Keywords: Programmable Graphics Hardware, Ray Tracing, Quadric Objects

1 Introduction

It has become popular to use the Graphics Processing Unit (GPU) to improve performance in current applications, primarily to reduce the work needing to be done on the CPU. Current research in implementing ray tracing algorithms on the GPU has been limited to scenes composed completely of triangles. The method for ray tracing triangles first presented in [1] runs almost entirely on the GPU, using a limited form of recursion to good effect in processing secondary rays. In [2], the GPU is used to accelerate only the ray-triangle intersections, with all runtime decisions made on the CPU. Problems with using only triangles include the need to increase the number of triangles the closer you get to the object in order to maintain a high level of detail. Using triangles to approximate curved surfaces also introduces artifacts, with a hard edge visible between two adjacent triangles. We present a method to ray trace objects other than triangles on the GPU. This reduces the number of objects in space subdivison structures, further simplifying processing of the scene. Quadric primitives are used here as an example, with extension possible to other surfaces.

In section 2, we briefly outline the capabilities of programmable GPUs. Section 3 looks at previous work in adapting ray tracing algorithms to the GPU. Sections 4 and 5 contain description and implementation details of our ray tracing and object intersection methods. Section 6 outlines possible improvements to the method, as well as directions for future research.

2 Programmable Graphics Hardware

Developments in graphics card technology in the last few years have added programmable functionality. On these new graphics cards, programs can now be run as part of the graphics pipeline [3]. Programs can be included at both the vertex and the fragment processing stages, allowing computation once per vertex and once per output pixel during each rendering pass (figure 1). These programs run as part of the pipeline, and so still take advantage of the functionality of the graphics API. These programs, also known as *shaders*, can be written in Cg, a c-like language supported in both DirectX and OpenGL, allowing it to be used in different environments [4].

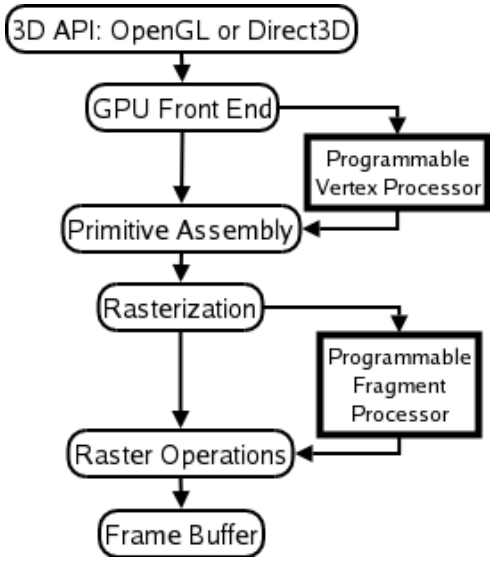


Figure 1: Graphics card programming model

3 Triangle Ray Tracing

The method for ray tracing triangles described in [5] uses a stream programming model, with a series of kernels that do the computation, connected by streams of information (figure 2). This is mapped to the graphics hardware using fragment shading programs as the kernels. Only one shading program can be executed on the card during each rendering pass. This algorithm must then use multiple rendering passes, using the rendered output of the current shading program as input to the next.
Using this method, a ray tracer can be built using the CPU to decide when to change from one operation to the next. The current fragment program is run once per fragment in each rendering pass, with multiple fragment processors running the same program in parallel on different fragments to compute the result. For example, GeForce FX series graphics cards have 12 parallel fragment processors. This method has similar performance to a software implementation, with most of the ray tracing work no longer being done on the CPU. This can give rise to more closely integrated applications, allowing the CPU to be either sharing the processing, or preparing scene information for the next frame while the current one is being rendered.

Figure 2 outlines the algorithm used to implement this method. Each step in this flow diagram represents a fragment shading program executed as part of a rendering pass. Each rendering pass involves setting a fragment shader as active and rendering a screen-filling quad. This allows the shader to be executed for each pixel on the screen, and the output to be saved to a texture. The texture represents the stream between two steps, which is then bound as input to the next step.
Generating the eye rays can be done in a single pass. Given the camera parameters, the rays are generated for each pixel and the output saved in a texture. In this method the entire triangle mesh is encoded in a uniform grid acceleration structure, all stored in texture memory on the graphics card. Traversing the acceleration structure involves multiple rendering passes. In each pass each eye ray is extended through the grid one voxel until it reaches a non-empty voxel.
Fragment programs can modify the value stored in the depth buffer, which can be used to keep track of the state of each pixel. Pixels can then be in either the 'traversal' or 'intersection' states, and a depth test can limit a rendering pass to only process those fragments in a given state. This reduces the work required in that rendering pass, and also makes it possible to count the number of fragments in that state. An occlusion query returns the number of fragments that were processed in the previous rendering pass. This allows looping to be controlled from the CPU, by running repeated rendering passes until there are no more fragments with that state in the depth buffer.
The eye rays are extended through the uniform grid until they all either reach a non-empty voxel, or pass out of the volume and require no further processing. The intersection routine is a similar process, looping over the triangles in the voxel until the closest intersection is found or the ray passes through the voxel and is returned to the traversal state. Switching between traversal and intersection states affects performance if not managed carefully. When all rays are either found to intersect a triangle or pass out of the scene, a rendering pass is initiated that shades these hitpoints and generates secondary rays if required. The recursive step in this method is limited to a single path, meaning that only one shadow, reflective or transmissive ray can be generated at each ray depth. The final result is then combined in the framebuffer and output to the screen after all processing is complete.

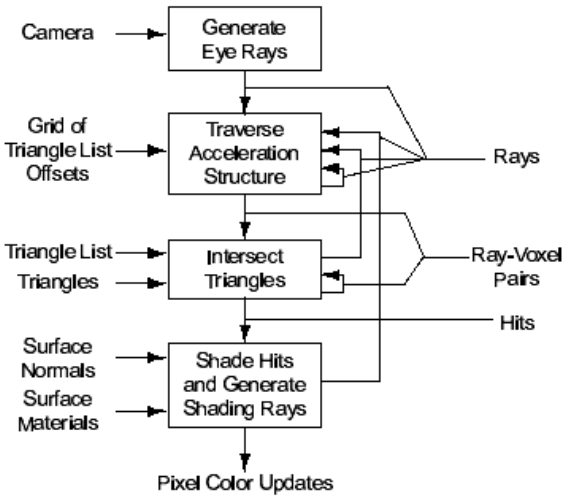


Figure 2: Streaming ray tracing model [5]

Another method that closely integrates processing between the CPU and the graphics hardware is the Ray Engine, proposed in [2]. This method uses the GPU to accelerate only the ray-triangle intersection tests. Ray tracers spend most of their time performing intersection tests, which proves to be the major bottleneck in the rendering process. Here the CPU runs the ray tracer, generating ray-triangle intersection tests as required. These tests are cached to make the best use of the GPU's parallel processing power. The CPU and GPU are very closely integrated, such that if the CPU finds that the GPU is still busy, it will calculate some of the intersection tests instead. Since only the ray-triangle intersection tests are performed on the GPU, the CPU can more easily maintain

acceleration structures, such as an octree - without having to try and implement this same structure on the GPU.

We also use the stream programming model in our more general ray tracing framework. This structure makes it easier to see the logical connection between different steps, which is then implemented using multiple rendering passes and textures as storage. The GPU is used primarily to speed up the intersection tests and compile the final output image.

4 Ray Tracing Framework

Backward ray tracers operate by casting rays from the eye through each pixel on the screen, to discover what the eye would see in the scene. This is a process that easily maps to graphics hardware designed to run the same fragment shading program once for every pixel in the window currently being rendered. Ray casting is the process of tracing only the primary rays through the scene, shading on the closest hit. This is not a full ray tracing model with secondary shadowing, reflection, or refraction rays. The framework shown in figure 3 outlines our ray casting method, using a CPU driven approach to allow for optimisation algorithms to be included. Each of the kernels is implemented by a fragment program running on the GPU, one per rendering pass.

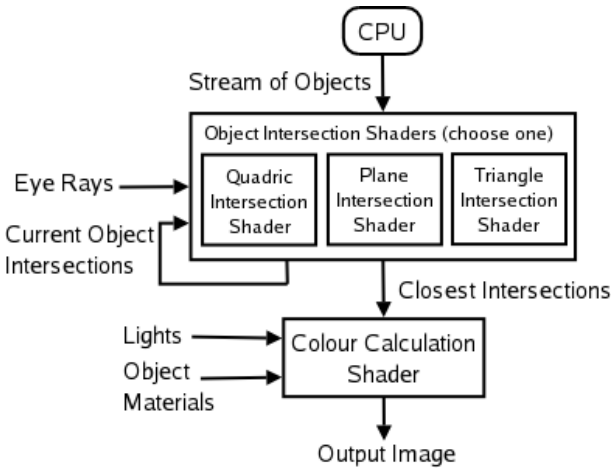


Figure 3: Ray tracing framework

The CPU runs a loop or traversal of the current scene, running an object intersection routine for each object with all initial eye rays. Any resulting intersections are stored in an intersection depth buffer updated in each intersection pass, retaining only the closest intersection point. When all objects have been processed, the shading routine is used to colour each pixel using the stored intersection point, lighting information and material properties.

Since the type of object is known on the CPU, the correct object intersection routine can then be chosen. This means that an object intersection shader can be built for each different type of object, rather than trying to produce a shader that can intersect all possible objects. These intersection routines are then kept quite short and, more importantly, fast, while being easy to swap in and out as the current intersection method. This allows a range of objects to be included in ray-traced scenes, encoding the ray-intersection routine as a fragment shader. Section 5 outlines the method for intersecting rays with the full range of quadric primitives.

The CPU can initiate this object shading process and continue processing any change in scene information for the next frame. This provides good results for single frame rendering, while being practical for the production of animation. The CPU and GPU are now working together more closely, reducing CPU workload and increasing performance.

Because object rendering is initiated on the CPU, it is possible to include a host of optimisation algorithms. It is also possible to store all the objects in either a heirarchical tree or even a grid acceleration structure. Here the simplest intersection method is used; each object is intersected with all of the rays cast into the scene.

4.1 Intersection Kernel

An intersection shader is built for each type of object in the scene. An example of how to construct such an intersection shader is given in figure 4.
The input definitions allow textures (lines 2-3) and variables to be made available (lines 4-5). Lines 7 and 16 list the lookups to these textures, using texture coordinates for the current screen pixel. The ray position and direction are then transformed to object space (lines 14-15). The intersection test function call on line 20 uses a function that would be defined specific to the type of object. The hitpoint and normal for the closest intersection are calculated and left in the 'hp1' and 'normal1' variables. Also output is a boolean value indicating whether or not this intersection test was a 'miss'. The new intersection point will only be output if there was a valid intersection, and the new intersection point is closer than any value currently stored in the intersection texture (line 25).
The output on a successful intersection test will overwrite any value currently being stored, if it is the closest hitpoint. The intersection depth buffer has to be maintained by the intersection shader on the GPU since there is more information

```
1  float4 main(float2 texcoords                 : TEXCOORD0,
2              uniform samplerRECT eyeray       : TEXUNIT0,
3              uniform samplerRECT intersection : TEXUNIT1,
4              uniform float4x4 AFT,
5              uniform float object_id) : COLOR {
6   float3 hp1,normal1; bool miss;
7   float4 temp = texRECT(eyeray, texcoords);
8   float4 world_ray_from = float4(unpack_2half(temp.x).x,
9                                  unpack_2half(temp.y).x,
10                                 unpack_2half(temp.z).x,1.0);
11  float4 world_ray_dir = float4(unpack_2half(temp.x).y,
12                                unpack_2half(temp.y).y,
13                                unpack_2half(temp.z).y,0.0);
14  float4 ray_from = mul(AFT,world_ray_from);
15  float4 ray_dir = mul(AFT,world_ray_dir);
16  float4 curr_pt = texRECT(intersection, texcoords);
17  float obj_dist = distance(float3(unpack_2half(curr_pt.x).x,
18                                   unpack_2half(curr_pt.y).x,
19                                   unpack_2half(curr_pt.z).x),world_ray_from.xyz);
20  intersect(AFT,ray_from,ray_dir,world_ray_from,world_ray_dir,hp1,normal1,miss);
21  float4 hit_pt = float4(pack_2half(half2(hp1.x,normal1.x)),
22                         pack_2half(half2(hp1.y,normal1.y)),
23                         pack_2half(half2(hp1.z,normal1.z)),object_id);
24  //conditionals are used here because the entire Cg shader is always executed
25  return ((distance(hp1,world_ray_from.xyz) < obj_dist) && !miss) ? hit_pt : curr_pt;
```

Figure 4: Cg code for the construction of an intersection shader.

than just the current depth being stored. It is only possible to output one four-component colour (32-bit floating point components) from a fragment shader, requiring that each component be packed with two 16-bit values (See figure 5 and lines 21-23). This output colour value from each intersection pass is stored in a texture the size of the output image, and fed as input into the next pass, ensuring each object intersection is supplied with correct data.

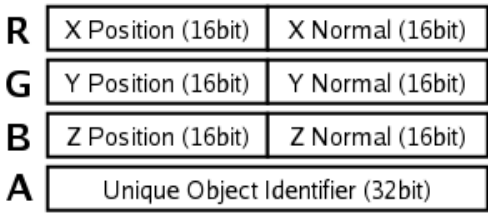


Figure 5: Intersection texture organisation.

Textures are used as temporary storage, providing a means to implement the streams in the stream programming model. The result of a rendering pass is output to a texture, which can be bound as an input to a shader in the next rendering pass. Implementation of this functionality is different on each platform: under DirectX this texture can be bound directly as a render target, but under OpenGL on Linux the framebuffer must be copied to a texture, a much more costly process.

4.2 Shading Kernel

Currently the Blinn-Phong[6] shading method is used to calculate the output colour for object intersections. Since there are currently no shadow rays being traced, there is no object shadowing and the output colour is made up of only the ambient, diffuse and highlight lighting components.

5 Quadric Intersection

The class of quadric objects includes cylinder, cone, ellipsoid, hyperboloid and others, with spheres and planes as special cases. Using a general quadric definition allows intersection with all possible shapes, without separate routines for each type of object. Treatment of the general case of quadrics is taken from [7].

$$Ax^2 + 2Bxy + 2Cxz + 2Dx + Ey^2 \\ +2Fyz + 2Gy + Hz^2 + 2Iz + J = 0 \qquad (1)$$

Equation 1 shows the full expansion of the quadric expression. This can also be expressed in matrix form, $x^T Qx$, with $x^T = [x\ y\ z\ 1]$ and

$$\mathbf{Q} = \begin{bmatrix} A & B & C & D \\ B & E & F & G \\ C & F & H & I \\ D & G & I & J \end{bmatrix}.$$

In order to build the ray-quadric intersection test, a quadric is then defined as $x^T Q x = 0$ with the ray definition $x = p + tv$. Substituting the ray definition for x gives:

$$\begin{aligned} (p+tv)^T Q(p+tv) &= 0 \\ p^T Qp + p^T Qtv + tv^T Qp + tv^T Qtv &= 0 \\ (v^T Qv)t^2 + (p^T Qv + v^T Qp)t + p^T Qp &= 0 \\ (v^T Qv)t^2 + (2v^T Qp)t + p^T Qp &= 0 \qquad (2) \\ (\text{Q is symmetric}) & \end{aligned}$$

The result of Equation 2 is in the form of a quadratic expression ($At^2 + Bt + C = 0$), allowing extraction of the terms A ($v^T Qv$), B ($2v^T Qp$) and C ($p^T Qp$). These can be used to calculate the discriminant ($B^2 - 4AC$) which, if less than or equal to zero, means there are no intersections with this quadric (tangential intersections are treated as nonintersecting). The quadratic equation can then be used to calculate the two possible intersection values:

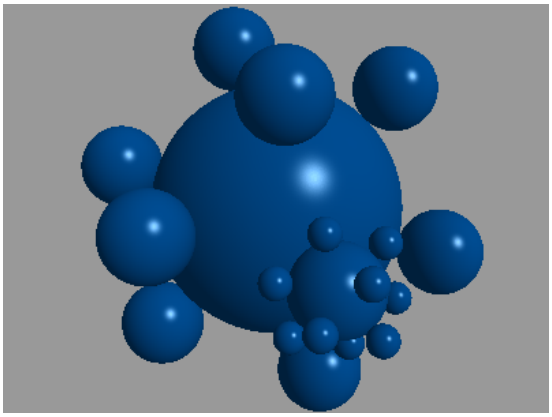
$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

The object space normal for each hitpoint can then be calculated for each t intersection value. This will need to be transformed back to world space for use in shading.

$$normal = normalize(Q(p + tv))$$

In order to keep the intersection routines simple, rather than transforming the objects themselves when we want to construct a deformation, we transform the rays, allowing us to operate on unit primitives. Each time we want to do a primitive intersection test, the ray is transformed into the object space of the unit primitive.

Since the entire quadric expression is used, it is possible to define the full complement of quadric shapes using combinations of the 10 variables. A unique matrix can then be built to define any of these primitives. Sample primitives are depicted in figures 6,7,8,10 and 11 with associated quadric matrices. These images were ray traced using a naive ray casting application built with the framework defined in section 4.



(a) $\mathbf{Q} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$

Figure 6: Part of a sphereflake, using the sphere matrix definition in figure 6(a).

6 Conclusion & Extension

In this paper, we have presented a framework for ray tracing arbitrary objects on programmable graphics hardware. This allows accurate images to be produced with fewer intersection tests and less processing. We built our GPU ray tracer alongside an existing software implementation, utilising the same underlying scene structure. In generating the sphereflake scene (figure 6), the GPU method was able to achieve 1.11 frames per second (FPS) while the software implementation ran at 0.53 FPS. These results were measured by timing the generation of a series of frames with the data structures rebuilt for each frame. Performance for this method is inhibited by the need to read back a lot of data to main memory under linux, as noted at the end of section 4.1. This research was carried out on a Pentium 4 2.4GHz box running Fedora Core 1, with an nVidia Quadro FX 3000 graphics card.

It is the intent of our research to further investigate the use of programmable graphics hardware in ray tracing and related applications. We will continue building a full ray tracing model, including shadow, reflection and refraction rays. Recursion is not possible on graphics hardware, so a mutiple-pass method will be required. By using the GPU to do the final rendering of scenes, interactive animation for such systems may become a reality. The CPU will be able to process information for the next frame, using the GPU to finish rendering the current frame. This will make it possible to quickly produce ray traced images using commodity graphics hardware.

Figure 7: Cone primitive, defined using figure 9(a).

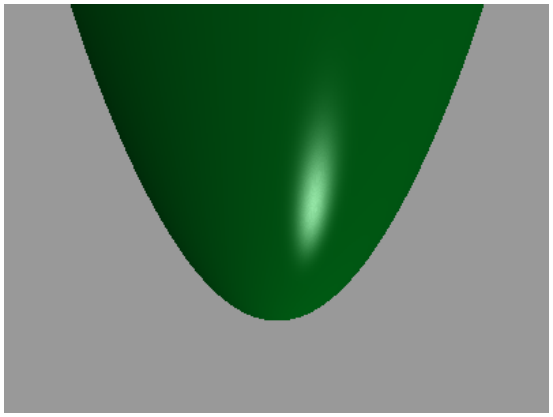


Figure 8: Paraboloid primitive, defined using figure 9(b).

$$\mathbf{Q} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{Q} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad \mathbf{Q} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \quad \mathbf{Q} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(a) (b) (c) (d)

Figure 9: Matrices defining the unit cone, paraboloid and hyperboloid primitives.

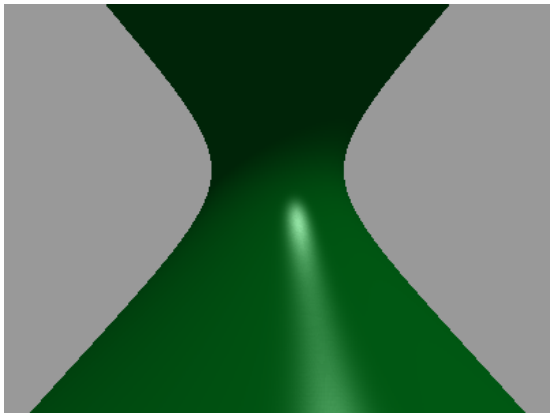


Figure 10: Hyperboloid (1 Sheet) primitive, defined using figure 9(c).

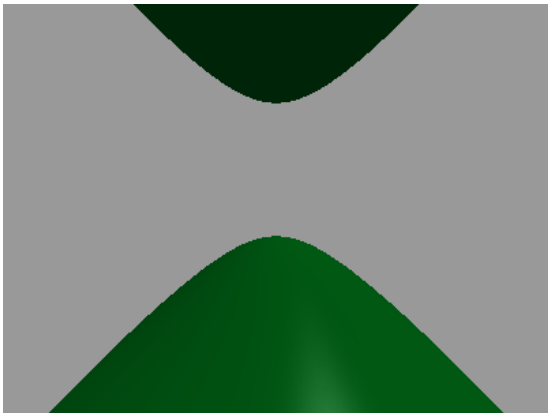


Figure 11: Hyperboloid (2 Sheets) primitive, defined using figure 9(d).

References

[1] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pp. 703–712, ACM Press, 2002.

[2] N. A. Carr, J. D. Hall, and J. C. Hart, "The ray engine," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 37–46, Eurographics Association, 2002.

[3] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: a system for programming graphics hardware in a c-like language," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 896–907, 2003.

[4] NVIDIA Corporation, *Cg Toolkit, Release 1.2*, January 2004. Software and Documentation available at http://developer.nvidia.com/object/cg_toolkit.html.

[5] T. J. Purcell, *Ray Tracing on a Stream Processor*. PhD thesis, Stanford University, March 2004.

[6] J. F. Blinn, "Models of light reflection for computer synthesized pictures," in *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pp. 192–198, ACM Press, 1977.

[7] C. A. Lindley, *Practical Ray Tracing in C*. Wiley, 1992. ASIN: 0471573019.