

# Implementing a Ray Tracer on a GPU

Thomas Mølhave <[thomasm@daimi.au.dk](mailto:thomasm@daimi.au.dk)>

January 10, 2005

## 1 Introduction

Capabilities of modern day graphics processing units (GPU) are rapidly increasing due to the high demand from the games industry. Ray tracing is a very old method for generation of images and have traditionally been implemented on a CPU.

This paper describes an implementation of a Ray Tracing running purely on a mediocre NV3x GPU. The reader of this paper should be familiar with C++, OpenGL[OpenGL, ], Cg[NVIDIA, ] and GPU programming in general. The GPU terminology used is that of the OpenGL. The implementation can be downloaded from <http://www.daimi.au.dk/~thomasm/GPGPU/GI.zip>.

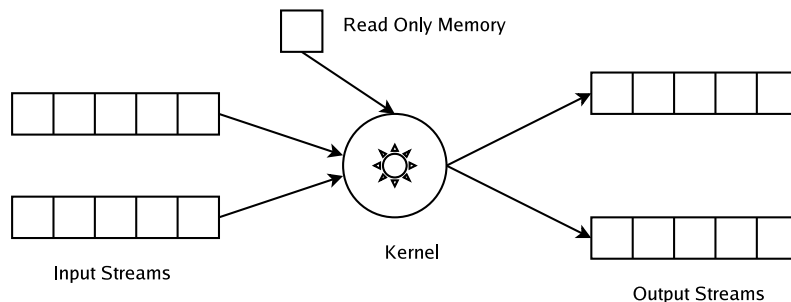
## 2 Related work

Many books have been written on the topic of general ray tracing (for instance [Shirley and Morley, 2003]). The idea of using the graphics processing unit to develop ray traced images is not new, and a couple of papers describing various methods are published.

One way of using the GPU is as a coprocessor for ray-triangle intersections ([Carr et al., 2002]), the ray tracing logic is running on the CPU and ray-triangle intersection tests are performed by the GPU. To optimize the system one needs to take care when deciding which ray-triangle pairs to test, a high degree of coherency is needed to get optimal performance. Another issue is the synchronization between GPU and CPU, the coprocessor should be able to run asynchronously with the CPU to be truly useful and this is quite possible with current graphic cards and their drivers.

More revolutionary, perhaps, is to run the entire ray tracer on the GPU using the CPU only as the trigger of certain events. The method was published in [Purcell et al., 2002] (and later in [Purcell, 2004], and it forms the foundation of the implementation described in this paper.

A lot of work on mapping the stream programming model to GPUs exist. The Brook for GPU's language[Stanford, ] is an abstraction of the GPU



**Figure 1:** The stream programming model schematically.

programming model, which allows the user to think purely in streams and kernels.

### 3 Theory

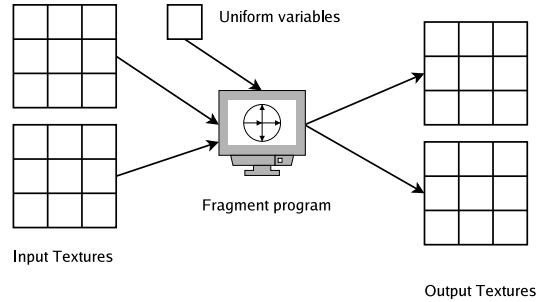
In [Purcell, 2004] a streaming ray tracing algorithm is developed. Algorithms based on streams, including this one, often map fairly well to GPU programming. This section will introduce the stream programming model and the ray tracing algorithm to be implemented. The algorithm will then be mapped to a completely general stream programming model, and then it is described how this is done on the GPU.

#### 3.1 The Stream Programming Model

Mapping an entire ray tracer to the GPU implies that we need to traverse and store acceleration data structures, store triangles and rays, generate eye rays, testing for intersections and computing the final pixel color using available geometry information - all of which are components not normally placed on the GPU.

The cost of computational power is significantly cheaper than that of bandwidth. Modern GPU's and dedicated stream processors (such as the Imagine processor [Khailany et al., 2000]) are therefore generally better at running low bandwidth algorithms optimized for cache coherency. The stream programming model was designed to make that fact explicit to the programmer thus making it easier to utilize the processor fully.

The two dominating concepts of the stream programming model are those of the *stream* and the *kernel*. A kernel is a function which operates on data fetched from a set of input streams and outputs data into a set of output streams. A schematic view can be seen in Figure 1. A stream holds the data operated on by kernels. If a kernel performs a substantial amount of computation for each set of input/output data it is said to have a high



**Figure 2:** The stream programming model mapped to the GPU.

*arithmetic intensity.* To maximize efficiency of a stream programming scenario one needs to balance the arithmetic intensity with the bandwidth of the processor, which in most cases favors kernels with many computations [Purcell, 2004].

It is also possible in some stream models to have the kernels read from a read only memory in addition to the input streams. This is depicted in Figure 1 on the previous page as well.

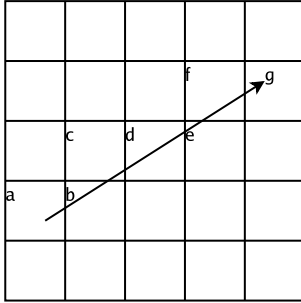
### 3.2 Mapping to the GPU

Mapping the concepts from stream programming to the GPU programming is fairly simple although the maturity of current GPU's impose severe limits that has to be dealt with in the mapping. That notwithstanding a simple mapping will now be presented. Details of the shortcomings of the mapping used in the implementation is presented in Section 4 on page 7.

To map a stream to the GPU a 2 dimensional texture is used, and kernels are represented using fragment programs. The use of textures as output streams requires a render-to-texture feature of the GPU to work efficiently in most cases. A number of uniform parameters can be used as the read only memory. The ideal GPU mapping of the stream model is represented in Figure 2.

### 3.3 Ray Tracing

The ray tracing algorithm exists in many variants with added features, most of which have to do with the performance or physical accuracy of the system. The ray tracer that this paper describes an implementation of is, however, fairly simple. As in any ray tracer the first step of the algorithm is to generate eye rays. No multi sampling is used so a single ray is generated for each pixel.



**Figure 3:** A correct traversal of the uniform grid, the voxel are visited in alphabetical order.

### 3.3.1 Accelerating Data Structure

The main work horse of any ray tracer is then the actual tracing of the rays through the scene. To avoid checking all rays with all objects, some kind of accelerating data structure is used. The high number of different strategies can be roughly divided into two categories. The spatial subdivision schemes subdivides three- or two-dimensional space into regions, and knowledge of this subdivision can then be utilized to skip intersection tests. The other scheme is using bounding volumes. The geometry of the bounding volumes is chosen to make intersection tests relatively cheap. The volumes are often arranged in a hierarchical structure.

In this case we use a spatial subdivision scheme, namely a uniform grid. For each voxel (box) in the grid we have a list of triangles intersection that grid, and when we trace a ray we just check the ray for intersections with the triangles contained in the voxels the ray passes. through.

### 3.3.2 Traversing the Grid

To make this work we need to be able to traverse the voxels along a given ray, which is a discretization problem similar to that of drawing a two dimensional line on a normal computer monitor. The line drawing is often done using a digital difference analyzer (DDA) such as Bresenham's algorithm, but a simple extension of Bresenham to three dimensions is not sufficient for our purpose since we need an additional property besides the added dimension.

The Bresenham algorithm and its derivatives were designed to produce rasterized lines that looked good and it tries to have a uniform width of the perceived line. The algorithm therefore skips some cells. For a correct traversal in the context of ray tracing we need all the cells which the ray intersects. This is demonstrated in Figure 3. In [Purcell, 2004] the algorithm presented in [Fujimoto et al., 1986] is used. In this implementation I have selected to use the 3D DDA described in [Amanatides and Woo, 1987] which is both elegant and fast.

```

1  while not done do
2    if  $tMaxX < tMaxY$  then
3       $tMaxX \leftarrow tMaxX + tDeltaX$ ;
4       $X \leftarrow X + stepX$ ;
5    end
6    else
7       $tMaxY \leftarrow tMaxY + tDeltaY$ ;
8       $Y \leftarrow Y + stepY$ ;
9    end
10   VisitVoxel(X,Y);
11 end

```

**Algorithm 1:** Uniform Grid Traversal

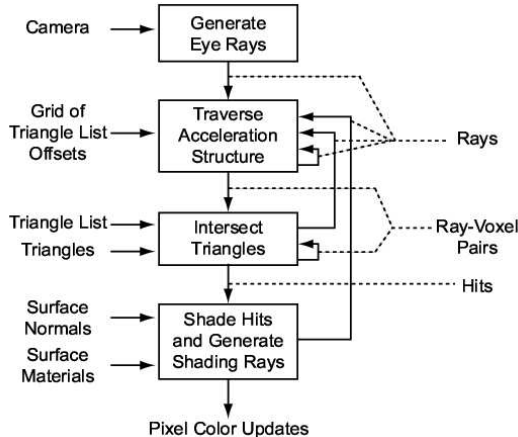
I will now describe how the traversal algorithm works in two dimensions. First we find the voxel in which the ray origin is located, call the index of this voxel  $X, Y$ . Then we initialize the two variables  $tMaxX, tMaxY$ .  $tMaxX$  holds the first parameter value at which the ray crosses a vertical voxel boundary, similarly  $tMaxY$  holds the first parameter for horizontal voxel boundaries. Another two variables  $tDeltaX, tDeltaY$  are also initialized.  $tDeltaX$  is the parameter for which the horizontal movement of the ray equals the voxel width. Finally  $stepX$  and  $stepY$  are computed.  $stepX$  is 1 if the ray traverses voxels of increasing x value, otherwise it is -1. The ray traversal algorithm from [Amanatides and Woo, 1987] is now formulated as can be seen in Algorithm 1. An extension to three dimension is fairly straightforward.

### 3.3.3 Ray-Triangle Intersection

Acceleration structures aside at the ray triangle intersection test is still needed at some point. I use the test from [Möller and Trumbore, 1997]. The only thing to note about the intersection routine is that it might be called for the same ray-triangle pair several times since a triangle can intersect several voxels. This can be prevented with a mail-boxing algorithm which simply marks the triangle with a unique index for the ray and then checking this index afterward[Amanatides and Woo, 1987], this requires one extra integer comparison but potentially saves many expensive intersection tests.

### 3.3.4 Shading

When the closest hit between a primary ray (eye ray) and an object is found a color needs to be calculated. Many algorithms spawn new rays and/or search for photons in a precomputed photon mapper, but for simplicity the implementation shades solely based on the normal of the surface, the position



**Figure 4:** Overall design of the streaming ray tracer. The figure is taken from [Purcell, 2004].

of lights and the intersection point. An extension to this should be relatively simple however. All surfaces are assumed to be perfect lambertian surfaces and the final color is calculated as

$$C = C_{diff} \langle N, L \rangle \quad (1)$$

where  $L$  is the vector from the intersection point to the (point) light source and  $C_{diff}$  is the diffuse color of the surface.

### 3.4 A Streaming Ray Tracer

Having introduced the ray tracing algorithm it is now time to map the ray tracing algorithm into the stream programming model. The overall concepts in this mapping are taken from [Purcell, 2004] and were first presented in [Purcell et al., 2002]. The overall design is depicted in Figure 4. Each box in the figure represents a kernel and the dotted lines are a rough estimate of the streams needed.

The first step is to generate the eye rays, the read only memory will contain the information required to generate a single eye ray for each pixel. The eye rays are put in an output stream. The rays should now be traced through the grid. To prepare for the grid traversal as depicted in Figure 4 a small kernel, not explicitly shown in the figure, is run. This kernel finds the starting voxel which is the one that the origin of the ray is located in, in addition a traversal state vector is required for proper operation of the traversal algorithm, this state vector is initialized with the needed values. To prepare for the grid traversal as depicted in Figure 4 a small kernel, not explicitly shown in the figure is run. This kernel finds the starting voxel which is the one the origin of the ray is located in, in addition a traversal

state vector is required for proper operation of the traversal algorithm, this state vector is initialized with the needed values.

The algorithm now enters a loop which search for the nearest triangle hit. The first kernel to be run after the traversal setup is the intersection kernel which performs ray-triangle intersections for each triangle found in the current voxel for that ray. If this kernel finds no intersection for a given ray it is run through the traversal kernel which advances the voxel of that ray. If an intersection is, in fact, found the ray is tagged. The intersection kernel is still running on that ray until every triangle in the voxel has been tested and the ray is then ignored in all future traversal and intersection tests. When a ray has passed through the entire grid it is marked as hitting the background.

The last kernel is the shading kernel. It computes a color for each of the rays marked as being done with traversal and intersection. If it is marked as hitting the background a constant color is returned, otherwise the color is produced based on the data on the triangle. The triangle is identified uniquely by its index as stored by the intersection tester, and the normal of the surface of the intersection point is found by interpolating across the triangle vertex normals by the Bayesian coordinates of the intersection, which was also stored by the intersection kernel.

The above procedure is looped as long as there are any rays still not shaded.

## 4 Implementation

Many of the implementation details will now be discussed. The ray tracer was implemented on a NVIDIA GeforceFX 5650GO graphics card. The textures used are 32bit precision floating point textures made available through the RenderTexture API [Harris, ]. The combination of a GeforceFX and floating point textures mandates the use of the NV\_TEXTURE\_RECTANGLE OpenGL extension which is therefore used. This is not an issue however as it makes the juggling of indices somewhat simpler.

The rendered scenes are drawn in 3D Studio MAX and exported using a custom plugin saving it in a custom file format. The file is then read by the ray tracer and the triangles are extracted.

### 4.1 Initializing Storage

Before we are ready to begin the ray tracing on the GPU we need the geometry data packed and ready in textures, including data on the uniform grid. The grid constructed has  $n$  voxels along each axis<sup>1</sup> and is constructed by the

---

<sup>1</sup>Of course, there is no theoretical reason as to why we have the same amount of voxels along each axis, but it makes it easier to index into the grid.

```

1  $o \leftarrow$  origin of the lower forward left point of the grid;
2  $w \leftarrow$  width of the box along all axis;
3 for  $z = 1$  to  $n$  do
4   for  $y = 1$  to  $n$  do
5     for  $x = 1$  to  $n$  do
6       for All triangles  $t$  do
7         if  $t$  intersects box with origin  $w \cdot (x, y, z) + o$  and width
            $w$  then
8           Add  $t$  to triangle list of voxel  $(x, y, z)$ ;
9         end
10      end
11    end
12  end
13 end

```

**Algorithm 2:** Uniform Grid Construction

naive algorithm shown in Algorithm 2.

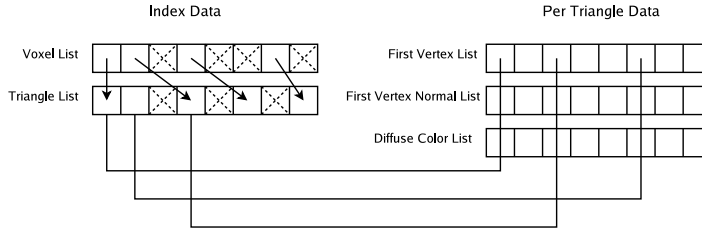
Once the grid has been constructed we now need to put the grid into a texture and allow access to triangle vertices, normals and diffuse color. The memory layout is depicted in Figure 5 on the following page. The grid is basically represented by one large array indexed by voxel number. For any given voxel the grid holds an index into a corresponding triangle list. The triangle list then contains a number of triangle indices delimited by a special delimiter value. The same delimiter value is used for voxels containing no triangles too.

Each index in the triangle list represents a triangle. That index can be used to look up the three vertices of that triangle and their normals (only the first vertex and normal is shown in the figure) as well as the per-triangle diffuse component. Figure 5 shows one dimensional lists, but we need to store two dimensional textures. The implementation maps an index list of size  $n$  to a texture of size  $k \times k$  where  $k = \lceil \sqrt{n} \rceil$ . The texture is filled from the first texel and forward leaving the remaining  $k^2 - n$  texels unused. All textures store one dimensional indices. Once we are on the GPU we need to translate from these indices into proper texture coordinates usable for texture lookups. Due to the use of the NV\_TEXTURE\_RECTANGLE extension these coordinates are still integers. If a list is saved in a texture of size  $k \times k$  a one dimensional index  $i$  is translated into texture coordinates by the following Cg function which is heavily used in the implementation:

```

float2 get_address(in int i, in int k) {
    return float2(0.25+i%k, i/k);
}

```



**Figure 5:** Memory layout on the GPU. The lists are packed into 2 dimensional textures. The lists on the left side have one component per entry and the lists on the right side contain three components per entry.

The addition of 0.25 to the result of the modulus operation was required for accuracy reasons. The result would give a floating point result very close to the correct integer, but the use of nearest texel sampling on the textures causes a result of 3.99999 when 4.0 is expected to give the wrong texel.

## 4.2 Implementing the Kernels

A thing never explicitly mentioned in Section 3.4 on page 6 is how we know what state a ray is in, we need to keep track on which kernel the ray should be processed in. In my implementation I maintain a four dimensional state vector for each ray. This state vector is kept in a separate stream of its own. The stream is as usual represented as a texture. The first component of the vector is interpreted as an state index, which keeps track on where in the pipeline the ray is located. The rest of the components of the state vector are introduced later. The following pipeline states are used:

**TRAV\_SETUP** The ray is about to enter the traversal algorithm. This is handled by a special traversal setup kernel.

**TRAV** The ray should be traversed to the next voxel. This is handled by the traversal kernel.

**ISECT\_NO\_HIT** The ray should be checked for intersection with the next relevant triangle, more on this later. No hit has yet been found for this ray. This is handled by the intersection kernel.

**ISECT\_HIT** This ray has hit a triangle in the current voxel. Proceed to check the rest of the triangles in the voxel. This is handled by the intersection kernel.

**SHADE** The ray has hit a triangle, a color value should now be calculated based on the information of the hit. This is handled by the shading kernel.

**BACKGROUND** The ray hit nothing and should produce a color accordingly. This is handled by the shading kernel.

**DONE** No more processing is needed for this ray.

This combination of kernels and states and the way they are managed are key concepts in the implementation and will be explained thoroughly. Common to all the kernels below are that they check the pipeline state in the state vector and discard any fragments that shouldn't be processed.

#### 4.2.1 Triggering the kernels

All kernels are triggered by drawing a quad the size of the desired rendered picture in an orthographic view mode setup as to allow each pixel of the output image to correspond exactly to one fragment.

#### 4.2.2 Generating Eye Rays

The eye ray generator is one of the structurally simplest kernels and transforms readily into a fragment program. For the eye ray generator uniform variables describing the viewing frustum planes, the eye position, the size of the view rectangle and the viewer orientation are used. These uniform variables represent the read only data for this kernel. The position of the hit point between the eye ray and the viewing plane is found simple by translating the texture coordinate of the fragment (which is an integer due to the `NV_TEXTURE_RECTANGLE` extension) such that the fragment at the center of the rendered quad is origin of the view plane.

The eye ray generator is the starting point of the ray tracer and has no input streams. The output stream of the kernel contains, according to Section 3.4 on page 6 the eye rays. Since a stream is represented by a texture this causes a problem. A single ray requires 6 floating point values and we only have 4 (RGBA) in a single floating point texture. This problem is solved by using two output streams for rays, one for ray origins and another for ray directions. The state of each ray is reset to `TRAV_SETUP`, so we need the state vectors as output as well. Note that all the output streams have the same size, and each stream has one component for each ray. This means that the state stream is indexed with the same index as the ray origin and direction streams.

#### 4.2.3 Traverser Setup

This kernel takes a ray and starts by locating the voxel in which the ray is initially positioned which is the one the origin of the ray is in. The voxel is easily found by translating the ray origin relative to the grid origin and then doing a integer division. This is done by the following simple piece of Cg code:

```

//o is the origin of the ray relative to the grid origin
int3 grid_offsets = int3(floor(o.x/voxel_width),
                        floor(o.y/voxel_width),
                        floor(o.z/voxel_width));

```

To store space the voxel index is not stored as a three dimensional vector, it is instead stored as a single component in the state vector stream. We need be able to translate back and forth from these indices. To do that we take advantage of the fact that we can consider a three dimensional grid index as being a number in a number system with base *grid\_count* (the number of voxels along each axis of the grid). When this is realized the index translation functions are simple to write.

As we saw in Algorithm 1 on page 5 each step of the algorithm needs to update the *tMax* vector which is now three-dimensional. This has the impact that we need a stream to hold the vector. This stream is initialized as explained in Section 3.3.2 on page 4. Here is the code to compute *tMaxX*

```

float4 plane;

//Plane parallel to the YZ-axis
plane.xyz = 0;
plane.x = 1;
plane.w = grid_offsets.x*voxel_width
         + (1.0 + step.x)*voxel_width/2.0f;

tMax.x = dot(r.d, plane.xyz);
if (abs(tMax.x) < 0.001) //no hit!
    tMax.x = INF;
else
    tMax.x = (plane.w - dot(r.o,plane.xyz))/tMax.x;

```

The *tDelta* and *step* vectors are static and could be precomputed and stored as well. I have not done this, however. As will be shown shortly, the code to compute these vectors are fairly simple and does not require any texture lookup. Besides saving a lot of video memory this might also be the fastest way since stream programming benefits from computational intensive kernels as mentioned in Section 3.1. A GPU specific study of this was done in [Fatahalian et al., 2004].

#### 4.2.4 Traverser

The traverser reads the voxel state field of the state vector, transforms the one dimensional index into (X,Y,Z) voxel coordinates which are then fed to the traverser algorithm. The bulk of the traverser fragment program is shown in Listing 1 on the next page. This also shows the three dimensional version of Algorithm 1.

**Listing 1:** An excerpt from the traversal fragment program. The "r" variable is an instance of the Ray structure which has two components, o and d. The o component is the ray origin and the d component is the direction.

```

//Compute static traversal data
tDelta.x = (voxel_width)/r.d.x;
tDelta.y = (voxel_width)/r.d.y;
tDelta.z = (voxel_width)/r.d.z;
tDelta.xyz = abs(tDelta.xyz);
float3 voxel = index_1d3d(statevec.z);
float X = voxel.x; float Y = voxel.y; float Z = voxel.z;
float3 step = sign(r.d.xyz);
//Run algorithm..

if (tMax.x < tMax.y) {
    if (tMax.x < tMax.z)
    {
        X += step.x;
        tMax.x += tDelta.x;
    } else {
        Z += step.z;
        tMax.z += tDelta.z;
    }
} else {
    if (tMax.y < tMax.z) {
        Y += step.y;
        tMax.y += tDelta.y;
    } else {
        Z += step.z;
        tMax.z += tDelta.z;
    }
}
float3 voxel;
voxel.x = X; voxel.y = Y; voxel.z = Z;

// "voxel" now contains the new voxel index
//Write output stream

datavec.xyz = tMax.xyz;

statevec.x = STATE_ISECT_NO_HIT; //Test intersections next
statevec.w = 0; //Reset triangle offset
statevec.z = index_3d1d(voxel); //translate index to 1d offset

// Detect rays traversed outside the grid
if (statevec.z >= grid_count*grid_count*grid_count)
    statevec.x = STATE_BACKGROUND;
if (voxel.x < 0 || voxel.y < 0 || voxel.z < 0)
    statevec.x = STATE_BACKGROUND;

```

#### 4.2.5 Intersection Kernel

The intersection kernel is the most complex of the kernels and is active for rays in state `ISECT_HIT` and `ISECT_NO_HIT`. The kernel needs the following input data: ray data, voxel data, triangle data, current closest intersection point and current triangle offset. The current triangle offset is used when looping through the list of triangles contained in a voxel and is stored in the  $w$  component of the state vector. The component is reset to 0 by the traverser as can be seen in Listing 1 and is incremented after each successful pass of the intersection kernel. The output of the kernel is a data stream that holds information of any hit, the information is stored in 4 floats and contains the Bayesian coordinates of the intersection and the index of the triangle. Some of the code to the kernel can be seen in Listing 2 on the following page.

There are a couple of rather subtle points that should be stressed here. The two first if statements detect the case where a voxel is empty, or all triangles for a given voxel have been traversed. In this case we have no intersections to test and the state should be advanced. This is where the two `ISECT` states comes into play. If the state is `ISECT_NO_HIT` no hit has been found with any of the triangles in the voxel and the ray should be traversed to the next voxel. Otherwise (the state is `ISECT_HIT`) an intersection has been found and this must then be the closest one since no voxel farther away can contain a point closer to any point in the current voxel, this causes the state of the ray to change to the `SHADE` state.

The `maybe_discard` code is unique to this shader. The intersection kernel is run twice, once for outputting intersection data and another for outputting the state vectors. This is due to the lack of multiple render targets and will be touched upon again later. The intersection data for a given ray should of course not be overwritten when no intersection occurs, and we therefore need to discard the fragment. But if we discard the fragment blindly we will not be able to update the state vector and we would be looping indefinitely. The `maybe_discard` statement is replaced with an ordinary call to `discard` by the preprocessor when compiling the program for the data output and completely removed when compiling for the state vectors. This ensures that the code in Listing 2 can be shared.

#### 4.2.6 Shading

The last state is the simple shading state. If the state is `BACKGROUND` a constant color is produced, otherwise the intersection data produced by the intersection kernel is fetched and then used to gather the triangle data. After interpolating the normals formula 1 is used to produce the final color. The intersection point is computed from the ray data and the parameter value found in the state vector.

**Listing 2:** Some of the intersection kernel code

```
int tri_count = statevec.w;
float2 grid_index = get_address(voxel, grid_size);
int trilist_index_1d = texRECT(grid, grid_index).x;
if (trilist_index_1d < 0)
{
    maybe_discard;
    statevec.x = (statevec.x == STATE_ISECT_NO_HIT) ?
                STATE_TRAV : STATE_SHADE;
} else {
    trilist_index_1d += tri_count;
    float2 trilist_index = get_address(trilist_index_1d,
                                      trilist_size);
    int vertex_index_1d = texRECT(trilist, trilist_index).x;
    if (vertex_index_1d < 0)
    {
        maybe_discard;
        statevec.x = (statevec.x == STATE_ISECT_NO_HIT) ?
                    STATE_TRAV : STATE_SHADE;
    } else {
        float2 vertex_index = get_address(vertex_index_1d,
                                          vtxlist_size);

        triangle t;
        t.a = texRECT(v0list, vertex_index).xyz;
        t.b = texRECT(v1list, vertex_index).xyz;
        t.c = texRECT(v2list, vertex_index).xyz;
        isect is;
        bool h = hit(r,t, is);
        float3 hit = evaluate(r,is.t);
        if (h)
            h = in_voxel(hit, voxel);
        if (h && is.t < statevec.y && is.t > 0)
        {
            data.xy = is.barycentric;
            data.zw = vertex_index;
            statevec.x = STATE_ISECT_HIT;
            statevec.y = is.t;
        } else {
            maybe_discard;
        }
        statevec.w++; // go to next triangle
    }
}
```

In this kernel new rays could be produced to generate a more advanced shading.

### 4.3 Tying it Together

With all the kernels taken care of we need some way to control the kernels from the CPU. After the eye rays have been generated and the traversal is setup we need to find out which kernel to run next. To do this we need to know how many fragments were active in a given pass, this is done with the aid of the `NV_OCCLUSION_QUERY` extension which returns just that. The various passes then return the number of active fragments when run. The implementation tries to interleave the intersection passes and the traversal passes and looks something like the following:

```
generate_eye_ray();
setup_traverser();
do {
    bool done = (intersector() == 0);
} while (done || traverser());
shade();
```

Due to the lazy evaluation of the condition this ensures that the loop keeps running the intersection kernel until no rays are active there. We then run the traverser once and continue to run the intersection kernel if any rays were traversed. The loop is stopped when no more fragments are active for intersection or traversal and the shading is then performed.

When the grid size increases, so does the traversal needed to finish the computation, and not many voxels has more than one triangle, this suggests another controlling strategy:

```
generate_eye_ray();
setup_traverser();
do {
    cont = (intersector() > 0);
    cont = (traverser() > 0) || cont;
} while (cont);
shade();
```

### 4.4 Hardware Issues

There is a rather large amount of control CPU code written to control the ray tracer, a great part of this code could be removed had the GPU been more capable.

#### 4.4.1 Multiple Output Streams

Nearly all of the kernels have more than one output stream. It is, however, not possible to use multiple render target on the GeForce FX5650 GO and this feature therefore had to be emulated, both in the Cg code and the control code. I will now describe a generic way to handle  $n$  output textures from a fragment program  $F$ . The control code has created the  $n$  textures and for each texture,  $F_i$  we compile a corresponding Cg program  $F_i$  with entry point  $F_i$ . The kernel is now executed as:

```
for (int i = 0; i < n; i++)
{
    render_to_texture_i_with_program_Fi;
}
```

The Cg code then has the following structure:

```
void F(out float4 result1, ... , out float4 resultn)
{
    //perform kernel
}

void Fi(out resulti)
{
    float dummy1,... dummyn;
    F(dummy1, ... , dummyi-1, resulti, dummyi+1, dummyn)
}
```

This is a simple implementation and allows the Cg compiler to optimize the unused dummy variables away for a given entry point. The problem is, of course that we have to run  $n$  fragments programs and, just as bad, perform  $n$  context switches.

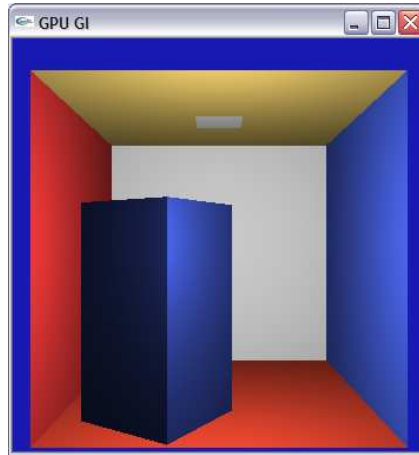
#### 4.4.2 Reading and Writing from the same texture

Among certain other texture the texture holding the state vectors needs to be both written to and read from in the execution of some kernels. This is not possible in the current NVIDIA drivers to avoid read-write hazards. [Purcell, 2004] uses specially modified ATI drivers which allowed it anyways, and since we are using streams this is not a problem since we only read from the same texel as we write.

This implementation uses double buffering of the textures in question instead, it creates two textures for each texture which should be used as both input and output and makes sure to bind the right one for index and write to the other one. There is, however, one catch to this solution in context of the rather complex logic of the ray tracer. The correctness of the ray tracer relies on the correct discard of irrelevant fragments for a certain state. This causes inconsistencies in the double buffered texture. Therefore

Kernel	Output Textures
eye ray generator	eye ray origin, eye ray direction, state vector
traversal setup	ray origin, ray direction, state vector, traversal data
traversal	state vector, traversal data
intersection	state vector, intersection data
Shader	Color

**Table 1:** Overview of the output textures



**Figure 6:** A screenshot of the ray traced Cornell Box

the texture is duplicated before the fragment program is run to ensure the the value of discarded fragments is correct. So the overhead of the workaround is a context switch, an extra texture in memory and the copying.

#### 4.5 Texture Overview

The implementations use a grand total of 20 floating point textures, out of these 9 are used as output texture at some point. Table 1 provides an overview of the output textures of the various kernels and represents a good overview of the overall construction of the ray tracer.

## 5 Discussion & Results

A rendering of a typical Cornell box is shown in Figure 6. This implementation of the ray tracer proves that even GeForceFX's have enough capabilities to run a ray tracer. The efficiency of the system is not very good however due to the many context switches and lack of multiple render targets. If the system was updated to the new pixel shader model available in NV4x chips and MRT one would see dramatically higher frame rater. Render time for

Strategy	Resolution	Seconds to render
First	100x100	5.5
First	256x256	22
First	512x512	79
First	756x756	170
Second	100x100	3
Second	256x256	12.5
Second	512x512	46
Second	756x756	100

**Table 2:** Overview of render times of the cornell box scene with 26 triangles and a grid size of 20x20x20

Resolution	Seconds to render
100x100	1.3
256x256	4.3
512x512	15
756x756	31

**Table 3:** Overview of render times of the cornell box scene with 26 triangles and no grid

the Cornell box can be seen in Table 2, the table also shows that the second controlling strategy, in this case, is superior.

If do not use the uniform grid the render times decrease significantly as seen in Table 3, this is rather unfortunate and defeats the purpose of having the grid in the first place. The reason for this increase in framerate is that the grid code relies heavily on the fragment programs ability do discard fragments in the wrong. A much better way is to avoid eveng running the kernels on rays with incorrect state. This could be done some kind of early-z discard technique where the state of the texture is used as a depth value. Using that the fragments can be discarded in a OpenGL pipeline state prior to the fragment processor.

I do not however believe that streaming ray traces implemented on traditional GPU's will see any serious adaption during the next couple of years. It is simply too cumbersome and complex to code them, the ray tracer presented here would hardly take a day take to implement traditionally.

It is however an interesting endeavor to port such algorithms to the GPU and the value of the theory will perhaps increase if high performance general purpose stream processors become mainstream. It is hard to tell how general the stream processing capabilities of the GPU will become, a native scatter operation, for instance, will not emerge for a while, if ever.

## 5.1 Future work

An obvious extension of the implementation presented here would be to add more advanced ray tracing capabilities such as shadow tracing, path tracing and/or photon mapping. Photon mapping on GPUs is described in [Purcell, 2004] as well and requires a bitonic search as the most complex GPU element. Much of the grunt work is already present in the implementation, all that is needed is a new shader and a couple of new ray textures.

## 5.2 Conclusion

The product of this paper is the ray tracer. The ray tracer is structurally fairly complex and uses 16 Cg programs and 20 floating point textures. Excluding the RenderTexture library and the graphics engine loading the scenes, a total of about 3200 lines of C++ to control the textures and Cg programs and constructing the uniform grid. Besides the C++ code 912 lines of Cg code were written. The program can be run on any 3d Studio MAX scene exported with the custom exporter. The finished ray tracer uses a simple algorithm but is powerful and easily extensible.

## References

- [Amanatides and Woo, 1987] Amanatides, J. and Woo, A. (1987). A fast voxel traversal algorithm for ray tracing.
- [Carr et al., 2002] Carr, N. A., Hall, J. D., and Hart, J. C. (2002). The ray engine.
- [Fatahalian et al., 2004] Fatahalian, K., J. S., and Hanrahan, P. (2004). Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. Available online at <http://graphics.stanford.edu/papers/gpumatrixmult>.
- [Fujimoto et al., 1986] Fujimoto, A., Tanaka, T., and Iwata, K. (1986). Arts: Accelerated ray-tracing system. *IEEE Computer Graphics & Applications*, pages 16–26.
- [Harris, ] Harris, M. J. Rendertexture library. Available online at <http://sourceforge.net/projects/gpgpu>.
- [Khailany et al., 2000] Khailany, B., Dally, W. J., Rixner, S., Kapasi, U. J., Mattson, P., Namkoon, J., Owens, J. D., and Towles, B. (2000). imagine: Signal and image processing using streams. *Hot Chips 12*.
- [Möller and Trumbore, 1997] Möller, T. and Trumbore, B. (1997). Fast, minimum storage ray-triangle intersection.

- [NVIDIA, ] NVIDIA. Cg user's manual. Available online at <http://developer.nvidia.com>.
- [OpenGL, ] OpenGL. Opengl 1.5 specification. Available online at <http://www.opengl.org>.
- [Purcell, 2004] Purcell, T. J. (2004). *Ray Tracing on a Stream Processor*. PhD thesis, Stanford University.
- [Purcell et al., 2002] Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. (2002). Ray tracing on programmable graphics hardware.
- [Shirley and Morley, 2003] Shirley, P. and Morley, R. K. (2003). *Realistic Ray Tracing*. A K Peters LTD, second edition.
- [Stanford, ] Stanford. Brookgpu. Available online at <http://graphics.stanford.edu/projects/brookgpu/>.