Raytracing Topics & Techniques - Part 1 - Introduction

Return to The Archives

by Jacco Bikker (29 September 2004)

Introduction

Hello and welcome to a new series of articles (or column, if you wish) on the topic of raytracing.

For those that do not know me: My name is Jacco Bikker, also known as 'Phantom'. I work as '3D tech guy' at Overloaded, a company that develops and distributes games for mobile phones. I specialize at 3D Symbian games, which require highly optimized fixed-point, non-HW-accelerated 3D engines, crammed into 250Kb installers. So basically I'm having fun.

As software rendering used to be my spare time activity, I was looking for something else. I tried some AI, which was great fun, and recently I dove into a huge pile of research papers on raytracing and related topics; such as global illumination, image based lighting, photon maps and so on.

One document especially grabbed my attention. It's titled: "State-of-the-Art in Interactive Ray Tracing", and was written by Wald & Slusallek. I highly recommend this paper. Basically, it summarizes recent efforts to improve the speed of raytracing, and adds a couple of tricks too. But it starts with a list of benefits of raytracing over rasterization-based algorithms. And one of those benefits is that when you go to extremes, raytracing is actually faster than rasterizing. And they prove it: Imagine a huge scene, consisting of, say, 50 million triangles. Toss it at a recent GeForce with enough memory to store all those triangles, and write down the frame rate. It will be in the vicinity of 2-5. If it isn't, double the triangle count. Now, raytrace the same scene. These guys report 8 frames per second on a dual PIII/800. Make that a quad PIII/800 and the speed doubles. Raytracing scales linearly with processing power, but only logarithmically with scene complexity.

Now that I got your attention, I would like to move on to the intended contents of this crash course in raytracing.

Contents

Over the next couple of articles I would like to introduce you to the beauty of raytracing. I would like to start with a really simple raytracer (spheres, planes, reflections) to get you familiar with the basic concepts.

After that it's probably a good idea to add things like refraction, area lights (resulting in soft shadows) and anti-aliasing to improve the quality of the graphics.

By that time, the raytracer will be painfully slow, so we'll add a simple spatial subdivision to speed it up.

And finally, I would like to introduce you to the wonderful world of Global Illumination, using photon maps. And believe me, you haven't really lived until you see your first colors bleeding from one surface to another due to diffuse photon scattering

Disclaimer

I would like to point out that I am pretty new to this. I've got some pretty decent results, and I'm quite sure my maths are OK, but in some cases I will undoubtedly make incredibly stupid mistakes. When that happens, don't forget to bash my sorry ass on the forum. I might learn from it.

Thanks!

I would like to thank Bram de Greve for proofreading these articles and providing very useful insights and corrections.

Basics

Raytracing is basically an attempt to imitate nature: The colors that you see are rays of light cast by the sun (probably), bouncing around the detailed scenery of nature, and finally hitting your eye. If we forget special relativity for a moment, all those rays are straight lines.

Consider the following illustration:

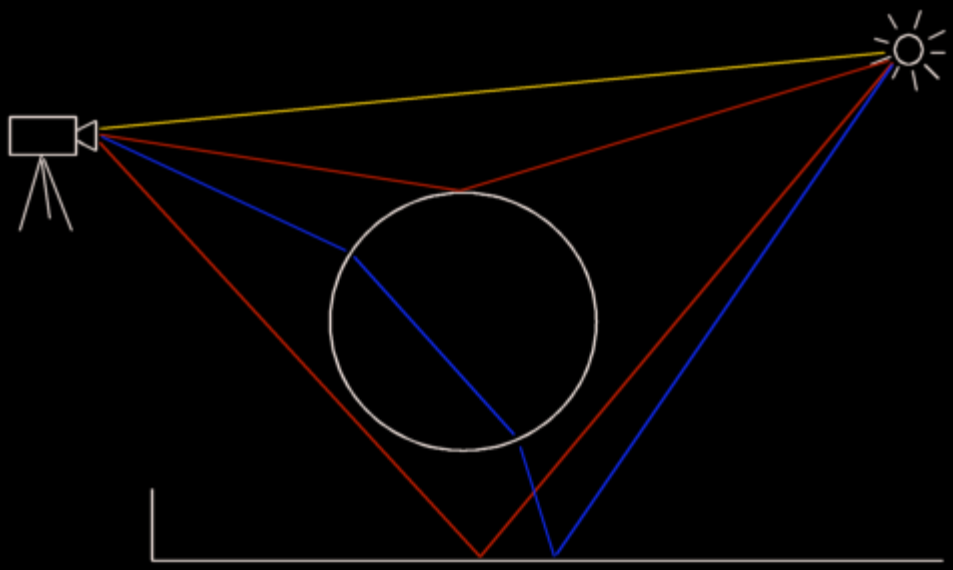


Fig. 1: Rays from sun to observer

I have drawn a couple of rays in this picture. The yellow one goes directly from the sun to the camera. The red ones reach the camera only after bouncing off scenery, and the blue one is bent by the glass sphere before hitting the camera.

What is missing in this picture are all the rays that never hit the observer. Those rays are the reason that a raytracer does not trace rays from a light source to a camera, but vice versa. If you look closely at the above picture, you can see that this is a fine idea, since the direction of a ray doesn't matter.

That means that we can have it our way: Instead of waiting for the sun to shoot a ray through that one pixel that is still black, we simply shoot rays from the camera through each pixel of the screen, to see what they hit.

Coding time

At the bottom of this article you'll find a link to a file kindly hosted by flipcode, containing a small raytracer project (VC6.0 project files included). It contains some basic stuff that I'm not going to explain here (winmain to get something on the screen and a surface class for easier pixel buffer handling and font rendering), and the raytracer, which resides in raytracer.cpp/.h and scene.cpp/.h. Vector math, pi and screen resolution #defines are in the file common.h.

Spawning rays

In raytracer.h, you will find the following class definition for a ray:

```
class Ray
{
public:
        Ray() : m_Origin( vector3( 0, 0, 0 ) ), m_Direction(
vector3( 0, 0, 0 ) ) {};
        Ray( vector3& a_Origin, vector3& a_Dir );
        
private:
        vector3 m_Origin;
        vector3 m_Direction;
};
```

A ray has an origin and a direction. When starting rays from the camera, the origin is usually one fixed point, and the rays shoot through the pixels of the screen plane. In 2D this looks like this:

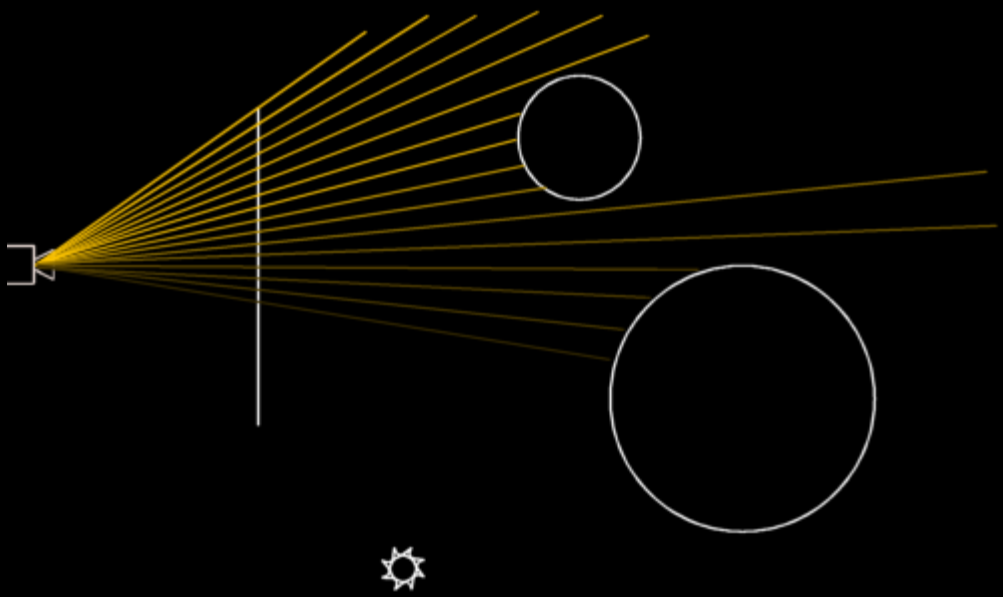


Fig. 2: Spawning rays from the camera through the screen plane

Have a look at the ray spawning code from the Render method in raytracer.cpp:

```
vector3 o( 0, 0, -5 );
vector3 dir = vector3( m_SX, m_SY, 0 ) - o;
NORMALIZE( dir );
Ray r( o, dir );
```

In this code, a ray is started at the origin ('o'), and directed to a location on the screen plane. The direction is normalized, and the ray is constructed.

A note about this 'screen plane': This is simply a rectangle floating in the virtual world, representing the screen. In the sample raytracer, it's centered at the origin, it's 8 world units wide and 6 world units high, which fits nicely for a 800x600 screen resolution. You can do all sorts of nice things with this plane: If you move it away from the camera, the beam of rays becomes narrower, and so the objects will appear bigger on the screen (use fig.2 to visualize this). If you rotate the plane (and the camera origin with it), you get a different view on the virtual world. It's rather nice that things like perspective and field of view are just a logical byproduct.

Building a scene

Next, we need a scene to raytrace. A scene consists of primitives: Geometric objects like spheres and planes. You could also decide to use triangles, and build all other primitives using those.

Take a look at the class definitions in scene.h. The primitives 'Sphere' and 'PlanePrim' are derived from 'Primitive'. Each primitive has a 'Material', and implements methods such as Intersect and GetNormal.

The scene itself is stored in a class named 'Scene'. Have a look at the InitScene method:

```
void Scene::InitScene()
{
	m_Primitive = new Primitive*[100];
	// ground plane
	m_Primitive[0] = new PlanePrim( vector3( 0, 1, 0 ),
4.4f );
	m_Primitive[0]->SetName( "plane" );
	m_Primitive[0]->GetMaterial()->SetReflection( 0 );
	m_Primitive[0]->GetMaterial()->SetDiffuse( 1.0f );
	m_Primitive[0]->GetMaterial()->SetColor( Color( 0.4f,
0.3f, 0.3f ) );
	// big sphere
	m_Primitive[1] = new Sphere( vector3( 1, -0.8f, 3 ),
2.5f );
	m_Primitive[1]->SetName( "big sphere" );
	m_Primitive[1]->GetMaterial()->SetReflection( 0.6f );
	m_Primitive[1]->GetMaterial()->SetColor( Color( 0.7f,
0.7f, 0.7f ) );
	// small sphere
	m_Primitive[2] = new Sphere( vector3( -5.5f, -0.5, 7
), 2 );
	m_Primitive[2]->SetName( "small sphere" );
	m_Primitive[2]->GetMaterial()->SetReflection( 1.0f );
	m_Primitive[2]->GetMaterial()->SetDiffuse( 0.1f );
	m_Primitive[2]->GetMaterial()->SetColor( Color( 0.7f,
0.7f, 1.0f ) );
	// light source 1
	m_Primitive[3] = new Sphere( vector3( 0, 5, 5 ),
0.1f );
	m_Primitive[3]->Light( true );
	m_Primitive[3]->GetMaterial()->SetColor( Color( 0.6f,
0.6f, 0.6f ) );
	// light source 2
	m_Primitive[4] = new Sphere( vector3( 2, 5, 1 ),
```

```
0.1f );
        m_Primitive[4]->Light( true );
        m_Primitive[4]->GetMaterial()->SetColor( Color( 0.7f,
0.7f, 0.9f ) );
        // set number of primitives
        m_Primitives = 5;
}
```

This method adds a ground plane and two spheres to the scene, and of course a light source (two, in fact). A light source is simply a sphere that is flagged as 'light'.

Raytracing

Now all is set up to start tracing the rays. First, let's have a look at some pseudo-code for the process:

```
For each pixel
{
        Construct ray from camera through pixel
        Find first primitive hit by ray
        Determine color at intersection point
        Draw color
}
```

To determine the closest intersection with a primitive for a ray, we have to test them all. This is done by the Raytrace method in raytracer.cpp.

Intersection code

After some initializations, the following code is executed:

```
// find the nearest intersection
for ( int s = 0; s < m_Scene->GetNrPrimitives(); s++ )
{
        Primitive* pr = m_Scene->GetPrimitive( s );
        int res;
        if (res = pr->Intersect( a_Ray, a_Dist ))
        {
                prim = pr;
                result = res; // 0 = miss, 1 = hit, -1 = hit
from inside primitive
        }
}
```

This loop processes all the primitives in the scene, and calls the Intersect method for each primitive. 'Intersect' takes a ray, and returns an integer that indicates a hit or a miss, and the distance along the ray for the intersection. The loop keeps track of the closest intersection found so far.

Colors

Once we know what primitive was hit by the ray, the color for the ray can be calculated. Simply using the material color of the primitive is too easy; this would result in boring colors without any gradient. Instead, the sample raytracer calculates a diffuse shading using the two lights. Since each light contributes to the color of the primitives, this happens inside a loop:

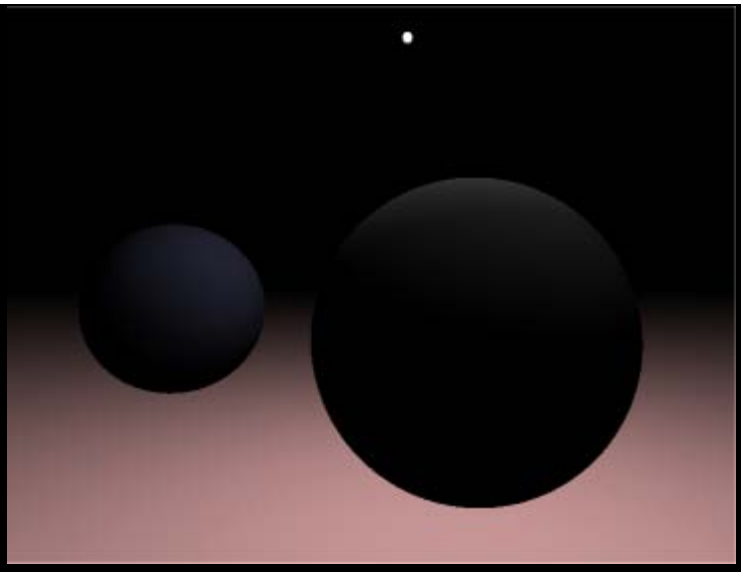
```
// determine color at point of intersection
pi = a_Ray.GetOrigin() + a_Ray.GetDirection() * a_Dist;
// trace lights
for ( int l = 0; l < m_Scene->GetNrPrimitives(); l++ )
{
	Primitive* p = m_Scene->GetPrimitive( l );
	if (p->IsLight())
	{
		Primitive* light = p;
		// calculate diffuse shading
		vector3 L = ((Sphere*)light)->GetCentre() -
pi;
		NORMALIZE( L );
		vector3 N = prim->GetNormal( pi );
		if (prim->GetMaterial()->GetDiffuse() > 0)
		{
			float dot = DOT( N, L );
			if (dot > 0)
			{
				float diff = dot * prim-
>GetMaterial()->GetDiffuse();
				// add diffuse component to
ray color
				a_Acc += diff * prim-
>GetMaterial()->GetColor() * light->GetMaterial()-
>GetColor();
			}
		}
	}
}
```

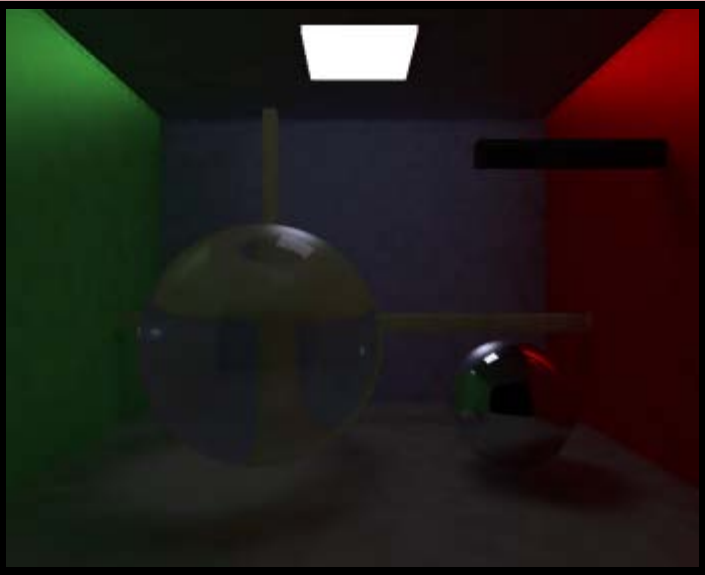
This code calculates a vector from the intersection point ('pi') to the light source ('L'), and determines illumination by the light source by taking the dot product between this vector and the primitive normal at the intersection point. The result is that a point on the primitive that is facing the light source is brightly illuminated, while points that are lit at an angle are darker. The test for 'dot > 0' prevents faces that are turned away from the light source get lit.

Last words

That's all for this article. In the next article I will explain how to add more interesting lighting and how to add shadows.

Here is a shot from the sample raytracer, and a preview of things to come.





See you next time,
Jacco Bikker, a.k.a. "The Phantom"

Download links:

- Sample project 1 - raytracer1.zip
- "State-of-the-Art in Interactive Raytracing", Wald & Slusallek, pdf

---

Article Series:

- ***Raytracing Topics & Techniques - Part 1 - Introduction***
- Raytracing Topics & Techniques - Part 2 - Phong, Mirrors and Shadows
- Raytracing Topics & Techniques - Part 3: Refractions and Beer's Law
- Raytracing Topics & Techniques - Part 4: Spatial Subdivisions
- Raytracing Topics & Techniques - Part 5: Soft Shadows
- Raytracing Topics & Techniques - Part 6: Textures, Cameras and Speed
- Raytracing Topics & Techniques - Part 7: Kd-Trees and More Speed

Raytracing Topics & Techniques - Part 2 - Phong, Mirrors and Shadows

Return to The Archives

by Jacco Bikker (06 October 2004)

Introduction

In the first article I described the basics of raytracing: Shooting rays from a camera through a screen plane into the scene, finding the closest intersection point, and simple diffuse shading using a dot product between the local normal of the primitive and a vector to the light source.

In the second article I would like to introduce you to mr. Phong, his bathroom mirrors and his shady sides. :)

For years I actually believed that 'phong' had something to do with the 'pong' sound that photons make when they bounce off a surface (if you listen really carefully), and I also believed that phong is normally implemented using a texture with a bright spot on it. But that appears to be phake phong...

Primary vs Secondary Rays

Consider the following image:

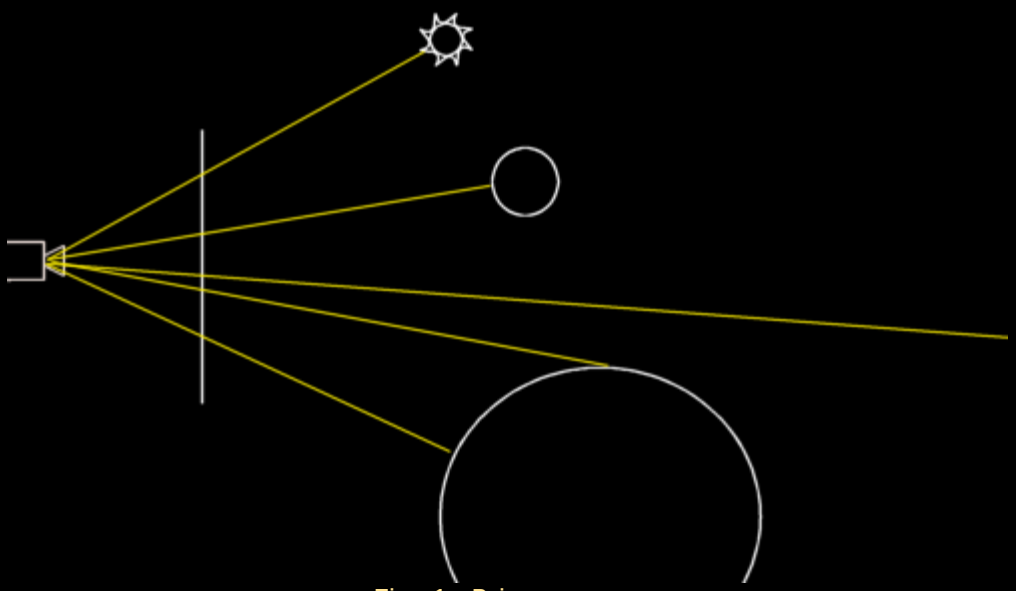


Fig. 1: Primary rays

This image shows the rays that the simple raytracer from the first article shoots into the scene. A ray can hit a light source, or a primitive, or nothing. There are no bounces and no refractions. These rays are called 'primary rays'.

Besides primary rays, you can use 'secondary rays'. These are shown in the next image (don't faint):

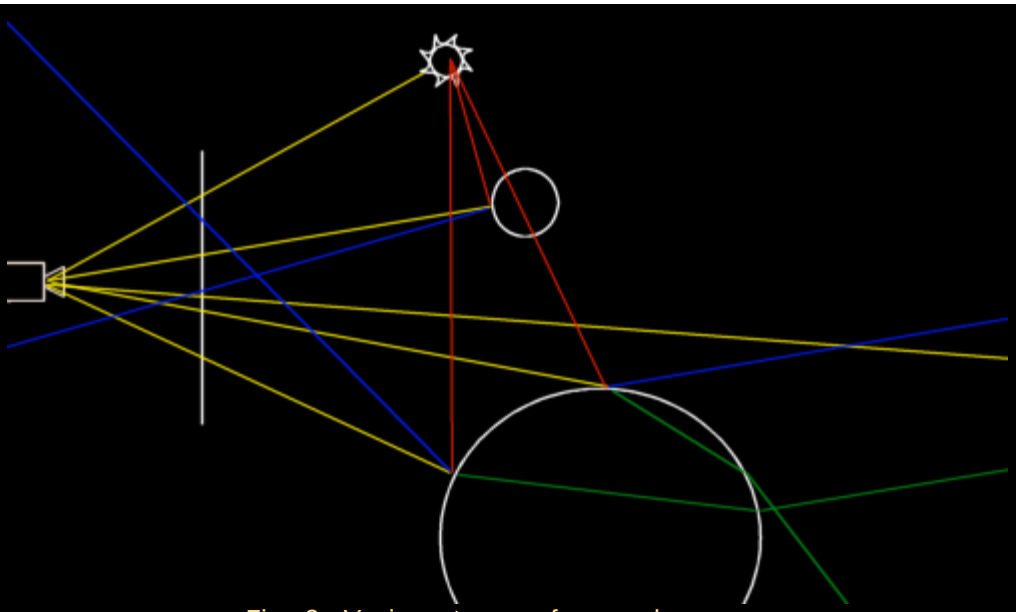


Fig. 2: Various types of secondary rays

The blue lines in this picture are reflected rays. For reflection, they simply bounce off the surface. How to do that exactly will be addressed in a moment.

The green lines are refracted rays. These are a bit harder to calculate than reflected rays, but it's quite doable. It involves refraction indices and a law formulated by mr. Snell (also known as Snellius, which is a rather strange habit of people of his period of time; imagine I called myself Phantomius That would be odd).

The red lines are rays used to probe a light source. Basically, when you want to calculate the diffuse lighting, you multiply the dot product by 1 if the light source is visible from the intersection point, or 0 if it is occluded. Or 0.5 if half of the light source is visible.

If you follow one of the yellow rays starting at the camera, you will notice that each ray spawns a whole set of secondary rays: One reflected ray, one refracted ray and one shadow ray per light source. After being spawned, each of these rays (except for shadow rays) is treated as a normal ray. That means that a reflected ray may be reflected and refracted again, and again, and again This technique is called 'recursive raytracing'. Each new ray adds to the color that its ancestor gathers, and so finally each ray contributes to the color of the pixel that the primary ray was originally shot through.

To prevent endless loops and excessive rendering time, there is usually a limit on the depth of the recursion.

Reflections

To reflect a ray of a surface with a known surface normal, the following formula is used:

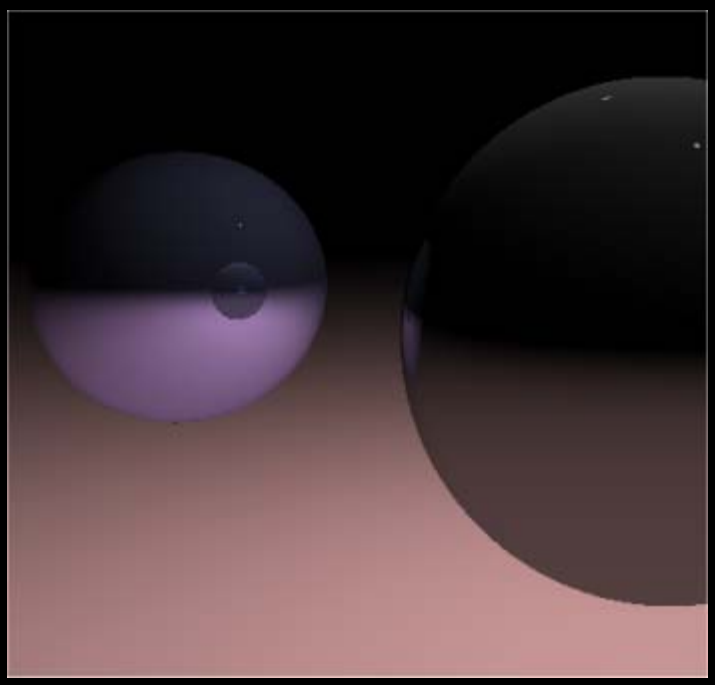
$$\vec{R} = \vec{V} - 2 * (\vec{V} \cdot \vec{N}) * \vec{N}$$

(where R is the reflected vector, V is the incoming vector and N is the surface normal)

This is implemented in the following code, which can be added to the raytracer right after the loop that calculates the diffuse illumination per light source.

```
// calculate reflection
float refl = prim->GetMaterial()->GetReflection();
if (refl > 0.0f)
{
        vector3 N = prim->GetNormal( pi );
        vector3 R = a_Ray.GetDirection() - 2.0f * DOT(
a_Ray.GetDirection(), N ) * N;
        if (a_Depth < TRACEDEPTH)
        {
                Color rcol( 0, 0, 0 );
                float dist;
                Raytrace( Ray( pi + R * EPSILON, R ), rcol,
a_Depth + 1, a_RIndex, dist );
                a_Acc += refl * rcol * prim->GetMaterial()-
>GetColor();
        }
}
```

If you didn't change the scene of the sample raytracer, you should now have something like this:



And that's quite an improvement. Note that both spheres reflect each other, and that the spheres also reflect the ground plane.

Phong

Creating 'perfect' lighting is extremely complex, so we will have to revert to an approximation. While the diffuse shading we used so far is excellent for soft looking objects, it's not so great for shiny materials. Besides, it doesn't give us any control at all, other than the intensity of the lighting.

Take a look at the following images:

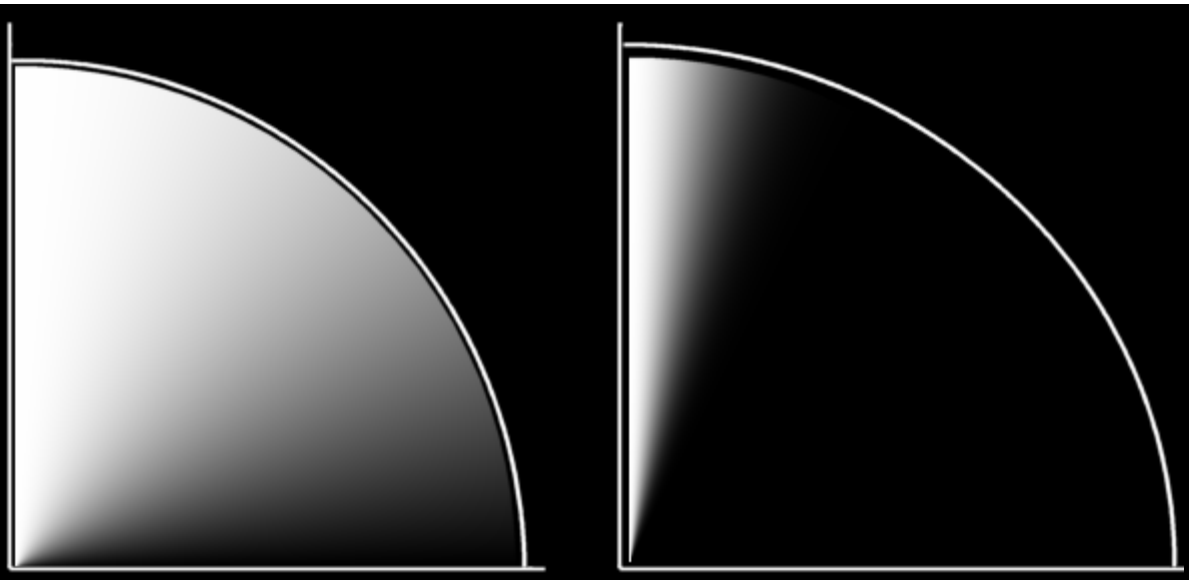


Fig. 3: Diffuse versus specular lighting

The left image shows the lighting that we used so far: The dot product of the normal and the light vector. There is a linear transition from white to black.

On the right side you see a graph of the same dot product, but this time raised to the power 50. This time, there is a very bright spot when the two vectors are close, and then a rapid falloff to zero.

Combining these improves matters quite a bit already: We get quite a bit of flexibility. A material can have some diffuse shading, and some specular shading; and we can set the size of the highlight by tweaking the power.

It's not quite right though.

The diffuse shading is OK: A diffuse material scatters light in all directions, and so its brightest spit is exactly there where the material faces the light source. Taking the dot product between the normal and a vector to the light gives this result.

Specular shading is a bit different: Basically, the specular highlight is a diffuse reflection of the light source. You can check this in real life: Grab a shiny object, put it on a table under a lamp, and move your head. You will notice that the shiny spot does not stay in the same position when you move: Since it's basically a reflection, its position changes when the viewpoint changes. Phong suggested the following lighting model, that indeed takes the reflected vector into account:

```
intensity = diffuse * (L.N) + specular * (V.R)n
```

(where L is the vector from the intersection point to the light source, N is the plane normal, V is the view direction and R is L reflected in the surface)

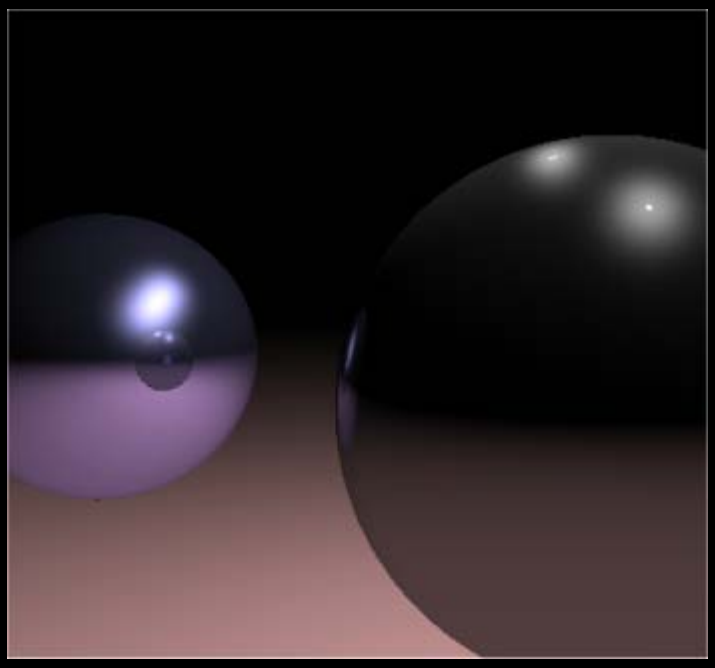
Notice that this formula covers both diffuse and specular lighting.

The code that implements this is shown below.

```
vector3 V = a_Ray.GetDirection();
vector3 R = L - 2.0f * DOT( L, N ) * N;
```

```
float dot = DOT( V, R );
if (dot > 0)
{
	float spec = powf( dot, 20 ) * prim->GetMaterial()->GetSpecular() * shade;
// add specular component to ray color
a_Acc += spec * light->GetMaterial()->GetColor();
}
```

After adding this to the lighting calculation, the raytracer produces an image like the one below:



Which is quite an improvement.

Shadows

The last type of secondary ray is the shadow ray. These are a bit different than the others: they do not contribute directly to the color of the ray that spawned them; instead they are used to determine whether or not a light source can 'see' an intersection point. The result of this test is used in the diffuse and specular lighting calculations.

The code below creates a shadow ray for each light source in the scene, and intersects this ray with all other objects in the scene.

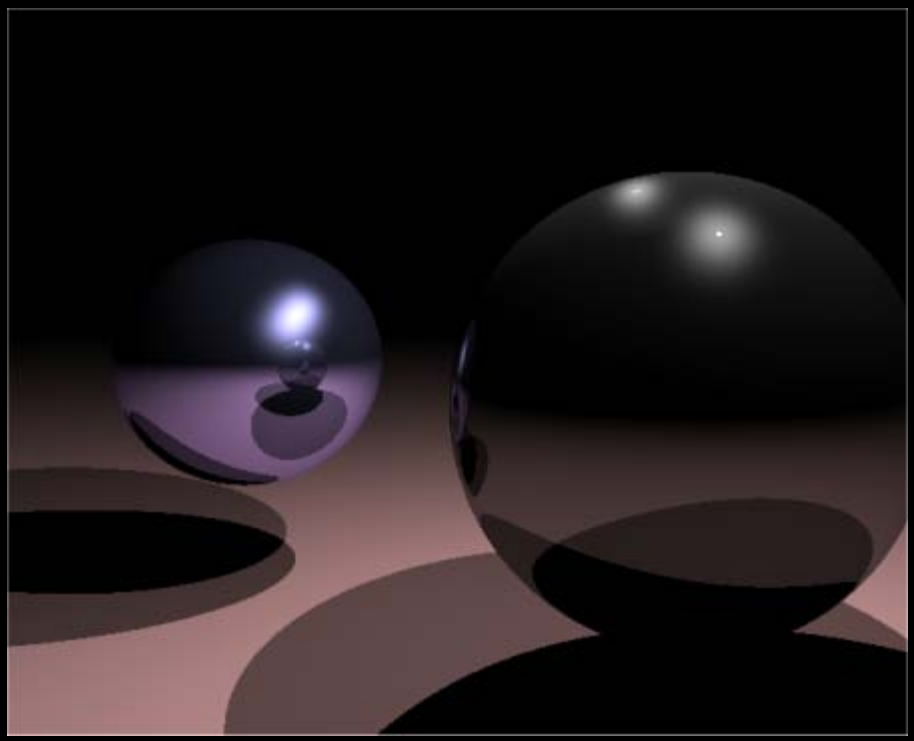
```
// handle point light source
float shade = 1.0f;
if (light->GetType() == Primitive::SPHERE)
{
	vector3 L = ((Sphere*)light)->GetCentre() - pi;
	float tdist = LENGTH( L );
	L *= (1.0f / tdist);
	Ray r = Ray( pi + L * EPSILON, L );
	for ( int s = 0; s < m_Scene->GetNrPrimitives(); s++ )
```

```
		{
			Primitive* pr = m_Scene->GetPrimitive( s );
			if ((pr != light) && (pr->Intersect( r,
tdist )))
			{
				shade = 0;
				break;
			}
		}
}
```

Most of this should be familiar by now. The result of the test is stored in a floating point variable 'shade': 1 for a visible lightsource, and 0 for an occluded light. Using a float for this might seem odd; however later on we will add area light sources, and those are often only partially visible. In that case, a 'shade' value between 0 and 1 would be used.

By the way, the above code does not always find the nearest intersection point of the shadow ray with the primitives in the scene. This is not necessary: any intersection with a primitive that is closer than the light source will do. This is quite an important optimization, as we can break the intersection loop as soon as an intersection is found.

Picture:



And there you have it. Two raytraced spheres, reflections, diffuse and specular shading, shadows from two light sources. The lighting on the plane subtly falls off in the distance due to the dot product diffuse lighting, resulting in a nice shading on the spheres. And all this renders within a second.

Notice how the shadows overlap to make the ground plane completely black. Notice how the color of the spheres affects the reflected floor plane color.

Raytracing is addictive, they say. :)

Final Words

One of the cool things about raytracing is that if you plug in something new, all the other things still work. For example, adding shadows and Phong highlights also adds reflected shadows and highlights. This is probably related to the parallel nature of raytracing: Individual rays are quite independent, which makes recursive raytracing very suitable for rendering on multiple processors, and also for combining various algorithms.

By the way, there's an error in the raytracer, which I will fix for the third revision: The result of the shadow test that is used for the diffuse component is also used for the specular component. Obviously, this is wrong. :) Send your solutions here for great prizes and eternal fame!

That's all for the second article. Next up: Refractions, Beer's law and adaptive supersampling.

An updated raytracer project is available using the link at the bottom of the page.

Greets,

Jacco Bikker, a.k.a. "The Phantom"

Link: Sample project 2 – raytracer2.zip

Article Series:

Raytracing Topics & Techniques - Part 3: Refractions and Beer's Law

Return to The Archives

by Jacco Bikker (13 October 2004)

Introduction

When you are deep in a raytracer project, you start looking at nature in an odd manner. I've noticed the same when working on a polygon engine: Building interiors suddenly appeared to have lots of 'polygonal detail' and 'wonderfully subdivided splines', but also 'poorly placed portals' and 'totally unrealistic soft shadows'.

While working on the raytracer, I experienced something similar. A glass of cool beer is very hard to render correctly. It has got a transparent hollow cylinder made of glass, a yellow substance inside it, and a highly complex matter near the top, consisting of tons of small spheres, with quite complex behaviour.

On the other hand, perhaps you have rendered too much if you start looking for suitable primitives and rendering challenges in a glass of beer...

In this article I would like to explain how to trace refracted rays. This involves spawning new rays at the point of intersection and calculating the new direction.

Besides this I will explain Beer's law: How light falls off inside a substance.

And finally, I would like to show how easy it is to get good anti-aliasing using a raytracer, and how to make it fast.

Refractions

Refraction is illustrated in figure 1. Notice how the rays bend at the surface of the primitve, and how they pass through one point behind the primitive. Objects behind this point will appear flipped and mirrored because of this.

How the rays are bent at the surface of the primitive depends on the refraction index of two materials: The material that the ray is in before it enters the primitive, and the material that the primitive is made of. Some examples: Air and vaccuum have a refraction index of about 1.0; water at 20 degrees Celcius has a refraction index of 1.33.

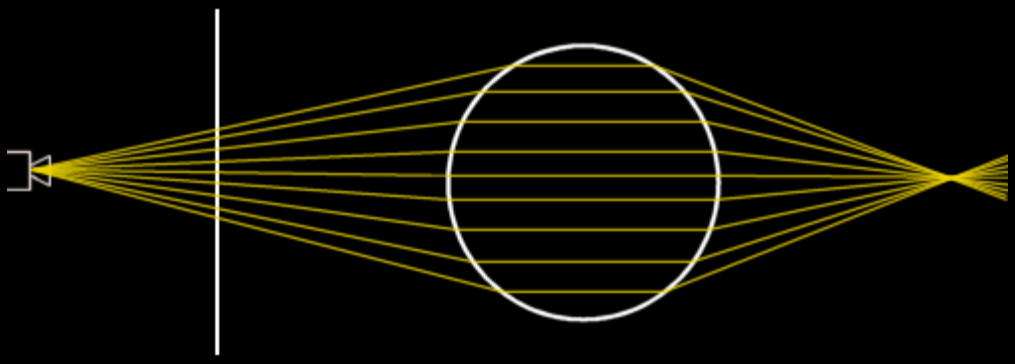


Fig. 1: Refraction

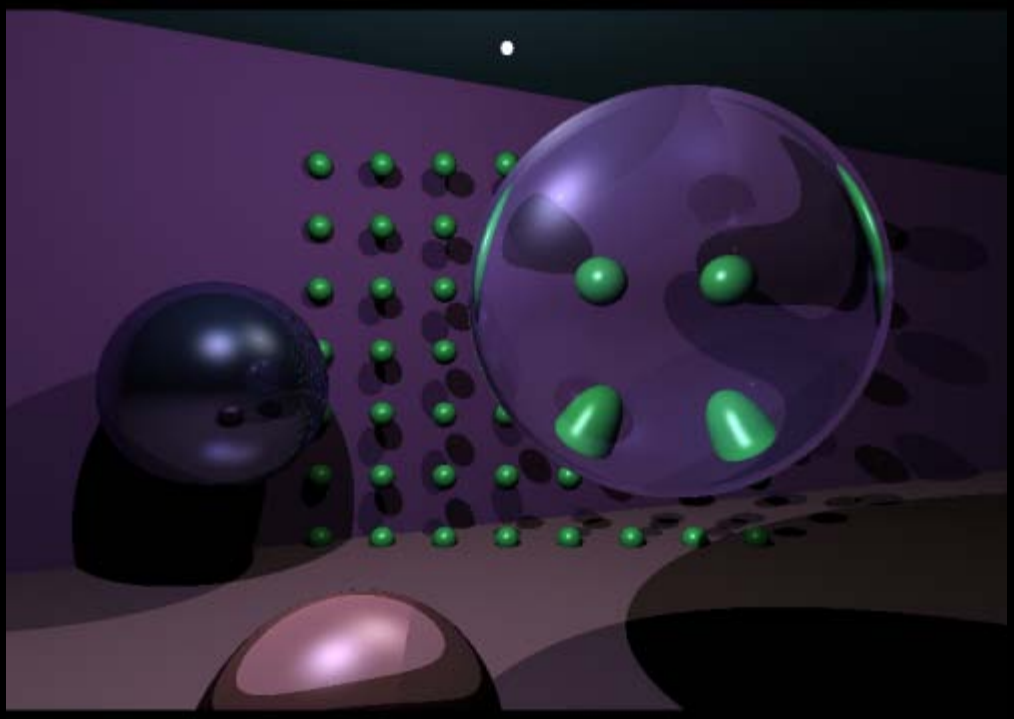
Regarding the exact maths for refraction, I'm not going to go into detail here. Instead, I would like to point to an article that Bram de Greve wrote on the subject. The pdf of this article is available at the end of the article, and will also appear separately at your favorite game coding site. You can view it here.

The code below should look pretty familiar by now. It constructs the refracted ray, traces it (recursively of course) and adds the resulting color to the ray that spawned the refracted ray. One note: The normal is multiplied by the value in the variable 'result'. This is a value that is filled by the intersection code for each primitive. It can either be 1 or 0, denoting a hit or a miss. There's a third option though: -1 means hit, but from within the primitive. This means that the origin of the ray that hit the primitive was inside the primitive itself. This is quite important: When a ray hits a primitive from the outside, basically it doesn't hit the primitive, but the matter around it. And therefore, the normal is reversed.

```
// calculate refraction
float refr = prim->GetMaterial()->GetRefraction();
if ((refr > 0) && (a_Depth < TRACEDEPTH))
{
        float rindex = prim->GetMaterial()->GetRefrIndex();
        float n = a_RIndex / rindex;
        vector3 N = prim->GetNormal( pi ) * (float)result;
        float cosI = -DOT( N, a_Ray.GetDirection() );
        float cosT2 = 1.0f - n * n * (1.0f - cosI * cosI);
        if (cosT2 > 0.0f)
        {
                vector3 T = (n * a_Ray.GetDirection()) + (n
* cosI - sqrtf( cosT2 )) * N;
                Color rcol( 0, 0, 0 );
                float dist;
                Raytrace( Ray( pi + T * EPSILON, T ), rcol,
a_Depth + 1, rindex, dist );
                a_Acc += rcol;
        }
}
```

I have added a slightly more interesting scene to the raytracer to make the effect of

refraction visible. It's included in the third release of the sample raytracer (see link at the end of the article). Here's a picture showing of the refraction code:



As you will undoubtedly have noticed, the raytracer now takes several seconds to render this image. This is only logical: There are a lot of primitives in the scene now, and every ray is intersected with every primitive to find the closest intersection point. Obviously, there's a better way to do this. We will use a spatial subdivision later on to limit the amount of intersection tests.

Beer's law

In the picture on the previous page, you can see that the sphere is blue, and so the refracted image is also slightly blueish. This is because the color returned by the refracted ray is multiplied by the primitive color. The same happens to reflections and diffuse and specular lighting. Many raytracers use the same technique for refracted rays. This is not entirely logical nor correct.

Imagine a pool with a colored substance (water mixed with blue ink, for example). At the shallow end of the pool the water is just 10 cm deep; at the other end it's over a meter. If you look from above to the bottom, it's rather obvious that at the shallow end, the bottom will be far less affected by the color of the ink than at the deep end. The effect of the colored medium is stronger over longer distances. This effect is called Beer's law. So, let's obey it.

Beer's law can be expressed in the following formula:

$$light\_out = light\_in * e^{-(e * c * d)}$$

This formula is primarily intended to calculate the light absorbance of a substance that is dissolved in water. 'e ' is some constant that specifies the absorbance of the solvent at hand (to be precise, the molar absorbtivity with units of L mol-1 cm-1); 'c' is the amount of this stuff, in mol L-1. 'd' is the path length for the ray. This is all extremely interesting if you want to study the behaviour of light in real materials, but if you just want the light to fall-off in a material that is not 100% translucent, basically all you're interested in is this

part:

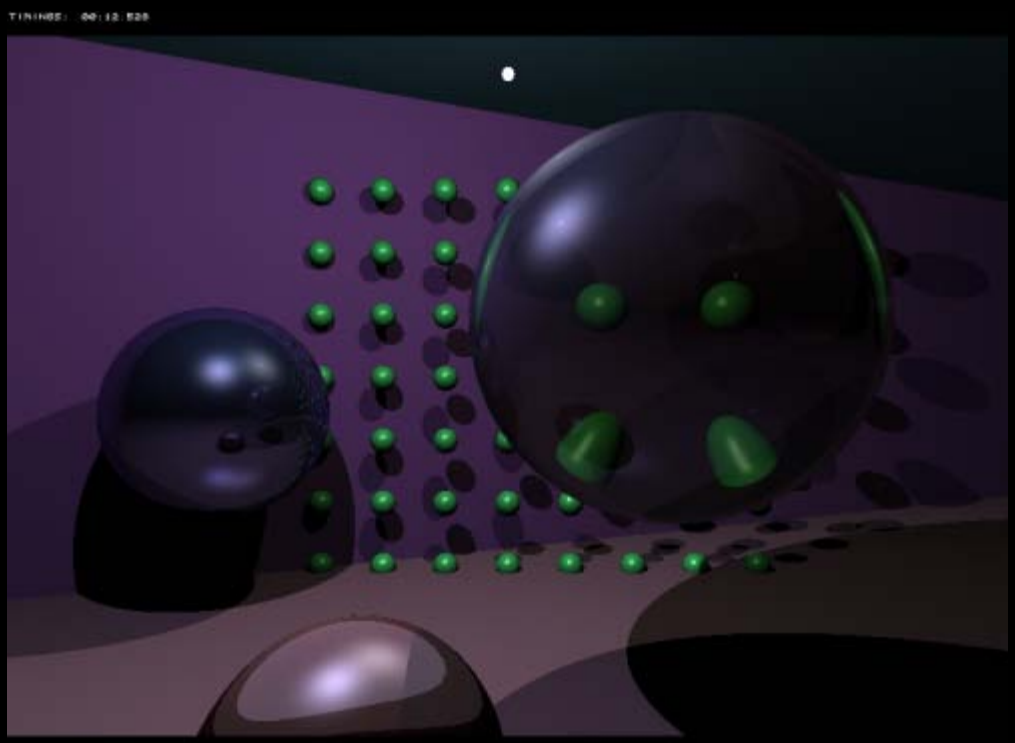
light_out = light_in * e$^{(d * C)}$

Where d is the path length, and C is some constant indicating the density of the matter. Lowering it will make light live longer while travelling through the substance.

Absorbance and thus transparency need to be calculated per color component. This leads to the following code:

```
Color absorbance = prim->GetMaterial()->GetColor() * 0.15f *
-dist;
Color transparency = Color( expf( absorbance.r ),
                            expf( absorbance.g ),
                            expf( absorbance.b ) );
a_Acc += rcol * transparency;
```

Here's the result:



Now I have to admit that this does not have an enormous impact on the image quality for the sample scene that we are using at the moment. However, once you start working with more complex materials, it can make quite a difference. And of course, there is the piece of mind that comes with an approach that is at least linked to a physical phenomenum.

Many raytracers use a simpler approach: Each material is assigned a 'reflectance' variable, which is multiplied with the color that the reflected ray returns, and a 'refractance', which is multiplied with the color that a refracted ray returns. For refracted rays, this doesn't work well: Every refracted ray is affected twice as it enters and leaves the primitive. And, travelling through a thin block will result in the same falloff as travelling through a rather thick block. The biggest problem though is that intuitively it doesn't feel right: Ray intensity doesn't drop at the surface of a primitive; it drops inside the primitive.

By the way, I have the same doubts with the lighting model: Diffuse shading combined with reflected specular shading feels like a hack, and in fact it is. Many raytracers even add an ambient component, which is simply added to the melting pot, just to cover the lack of global illumination. I guess I'll have to live with hacks to get a reasonable rendering speed, just because nature is incredibly complex...

Supersampling

OK, on to something lighter. :)

Suppose we replace the code that spawns a ray (in the Engine::Render method) with the following code:

```
for ( int tx = 0; tx < 4; tx++ ) for ( int ty = 0; ty < 4;
ty++ )
{
        vector3 dir = vector3( m_SX + m_DX * tx / 4.0f, m_SY
+ m_DY * ty / 4.0f, 0 ) - o;
        NORMALIZE( dir );
        Ray r( o, dir );
        float dist;
        Primitive* prim = Raytrace( r, acc, 1, 1.0f, dist );
}
int red = (int)(acc.r * 16);
int green = (int)(acc.g * 16);
int blue = (int)(acc.b * 16);
```

This code fires 16 rays through each pixel, and averages the result. The resulting image is anti-aliased. Nifty. However, it takes ages to render, and that's logical, since basically it's now rendering an image that's 16 times larger.

There's an easy way to have the best of both worlds though: Speed and quality. Well almost.

Perhaps you noticed that the raytracer returns a pointer to a primitive: This is the primitive that was hit by the primary ray. When we modify the ray spawn code so that it only traces 4x4 rays when we encounter a new primitive, performance improves dramatically:

```
// fire primary rays
Color acc( 0, 0, 0 );
vector3 dir = vector3( m_SX, m_SY, 0 ) - o;
NORMALIZE( dir );
Ray r( o, dir );
float dist;
Primitive* prim = Raytrace( r, acc, 1, 1.0f, dist );
int red, green, blue;
if (prim != lastprim)
{
        lastprim = prim;
        Color acc( 0, 0, 0 );
        for ( int tx = -1; tx < 2; tx++ ) for ( int ty = -
1; ty < 2; ty++ )
        {
                vector3 dir = vector3( m_SX + m_DX * tx /
2.0f, m_SY + m_DY * ty / 2.0f, 0 ) - o;
                NORMALIZE( dir );
                Ray r( o, dir );
```

```
                float dist;
                Primitive* prim = Raytrace( r, acc, 1, 1.0f,
dist );
        }
        red = (int)(acc.r * (256 / 9));
        green = (int)(acc.g * (256 / 9));
        blue = (int)(acc.b * (256 / 9));
}
else
{
        red = (int)(acc.r * 256);
        green = (int)(acc.g * 256);
        blue = (int)(acc.b * 256);
}
if (red > 255) red = 255;
if (green > 255) green = 255;
if (blue > 255) blue = 255;
```

This code runs about as fast as the version without supersampling, which is logical, as the extra code is only used at primitive boundaries.

It's not as good as full supersampling though:

- Shadow edges will still be blocky;
- The current code only detects vertical boundaries.

Both issues are quite easy to resolve. The raytrace method as it is now returns a pointer to a primitive, but you could also choose to return some other number, based on the primitive pointer, combined with reflected primitives and the number of visible lights. This would smooth out reflected primitive boundaries and shadow egdes.

Detecting horizontal primitive boundaries is also easy: Just store an array of pointers to primitives for the previous line. Now you can compare to the previous pixel on the left, but also on the previous line.

Obviously, the more boundaries you detect, the slower the code gets. But it will result in smoother images, at quite a low cost.

Final words

Well this pretty much concludes the basics of raytracing. I hope it's clear by now how intuitive raytracing is: After all it's just a bunch of rays doing funny things. It's also pretty close to the 'real thing', algorithm-wise.

There's lots more to explore though. First of all, there's the issue of performance: Raytracing quite quickly becomes slow, as you undoubtedly have noticed. So far I payed little attention to this issue.

The other thing is realism: The closer we get to the way light behaves in the real world, the better the results will be, obviously. Especially interesting is the topic of global illumination. This will replace the 'ambient shading' used in other raytracers with something that is far more realistic.

Both could be taken to extremes, but sadly they are more or less mutually exclusive. You can do real-time raytracing, but I doubt that anyone will do a real-time photon mapper in the next couple of years. And on the other hand, if you thought refraction was slow, wait till you see my photon mapper. :)

Next up: Spatial subdivisions.

That's all for today, see you next time!

Jacco Bikker, a.k.a. "The Phantom"

Link: Raytracer v3
Reference: Reflections and Refractions in Raytracing by Bram de Greve

---

Article Series:

Raytracing Topics & Techniques - Part 4: Spatial Subdivisions

Return to The Archives

by Jacco Bikker (26 October 2004)

Introduction

The last scene from the previous article (the one with the sphere grid to show off the new refraction code) took almost 9 seconds to render on my pimpy 1.7Ghz Toshiba laptop with 1600x1200 screen (I know resolution has nothing to do with it, but I thought I would mention it anyway). And that's just for simple stuff. Later on we will add area lights, and to test the visibility of those, we will need lots of rays for each pixel, that is.

And, I would like to be able to render triangle meshes exported from 3DSMax, consisting of 10.000 triangles or more. That's going to take ages!

It's easy to see what causes the delays: Every ray (primary and secondary) is intersected with about 100 spheres Even if it's painfully clear that the ray doesn't hit the primitive by a long shot.

In this article I will present some solutions for this.

First, I will outline some algorithms, and then I'll explain how to implement a simple spatial subdivision scheme: The regular grid. And finally we'll have a look at performance improvements and some alternatives.

Reducing Intersection Tests

A primary ray will hit only one primitive. In the raytracer that we built in the first three articles, this one primitive was found by trying them all, and keeping the closest one. This can take a lot of time if there are many primitives, and it's easy to see that rendering speed decreases linearly with the number of primitives in the scene.

There are a couple of ways to make things better.

Hierarchical Bounding Volumes

You could, for example group your objects, and encapsulate each group with a larger object (a sphere, or a box, for example). Now, if the ray misses the larger object, you can be sure it also missed the primitives in it. Storing multiple bounding volumes in an even larger volume reduces the number of tests even further. This is shown in figure 1:

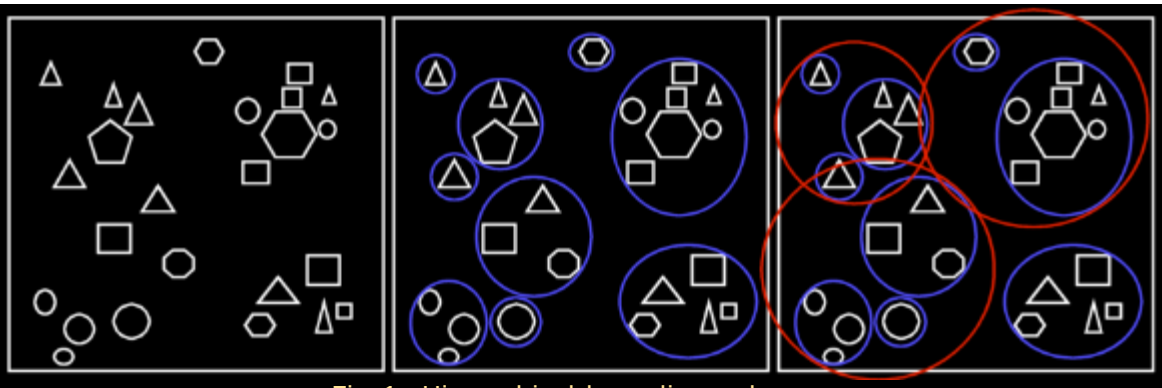


Fig 1: Hierarchical bounding volumes

The first image shows a scene with quite a bit of detail. The second one shows the same scene, but this time groups of objects are encapsulated by bounding spheres. In the third image, groups of bounding spheres are grouped.

When well executed, this technique can seriously improve rendering speed. The number of intersection tests is reduced to a fraction of the original count. There are however quite some cases where this algorithm will not perform so well. Looking at figure 1, it's easy to think up rays that hit several bounding spheres, but no primitives. And, in the above example you'll still need to test all three red spheres for every ray. Imagine a ray that hits the triangle in the upper left corner of the scene: The ray would first be tested against the three red spheres, then to the three blue spheres, and finally to the triangle. That's 7 tests.

Use the GeForce, Luke

There is another way. Suppose you draw the scene using your favorite 3D accelerator. Each primitive is drawn using a unique color (which limits the number of primitives somewhat, but let's ignore that). Now, primary rays can be omitted completely: For each pixel, the raytracer merely reads the color value to identify which primitive is closest. The z-buffer can be used to determine the intersection point (although that's probably not very accurate). From there on, secondary rays can be spawned.

This works quite well, especially when combined with the previous approach to accelerate the secondary rays. There's of course the limit to the number of primitives (not such a big problem when rendering to a 32bit target), and obviously you can't do supersampling like in the previous article. The biggest problem is obviously that this does not solve the problem for secondary rays, and once these start making up the majority of the spawned rays, you have to look for something else.

Spatial Subdivisions

Have a look at figure 2:

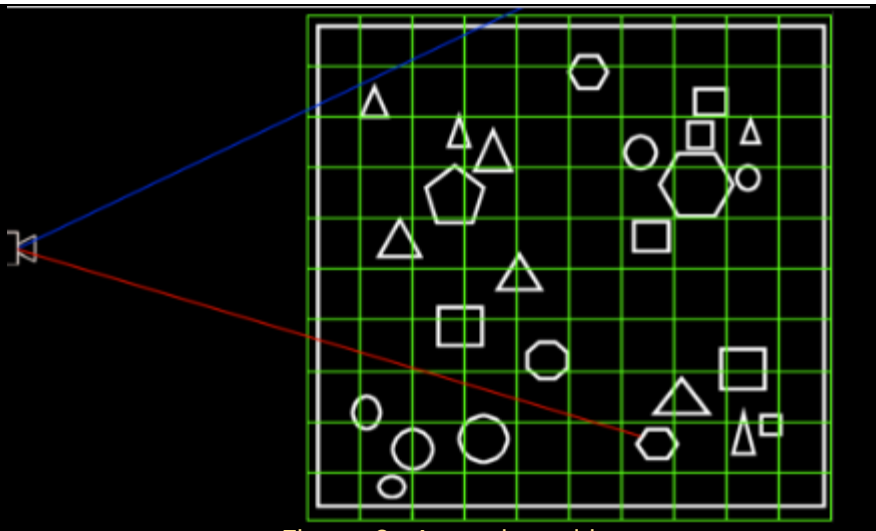


Figure 2: A regular grid

This is the same scene as was used in figure 1, but this time it's subdivided by a regular grid.

The camera in this image shoots two rays: The red one, and the blue one. Rays step through the grid, looking for primitives; as long as grid cells are empty, no intersection tests are performed. This means that the blue ray only tests one primitive. The red ray tests four primitives.

That appears to be quite a bit better than the other algorithms.

This algorithm has its problems too though: First, stepping through a regular grid is not very expensive, but not free either. Especially setting up the variables before the ray traversal starts can be quite costly. And, while in the above picture most cells only contain one primitive, this is obviously not always the case. And worse: Some cells may contain tons of primitives, while others are empty. An often used example is a football stadium with a highly detailed ball at the middle of the field. Storing this in a grid is a nightmare: There are lots of empty grid nodes, and then there is one grid node that contains the ball. Visiting this node means intersecting with all the triangles in it. Using a more detailed grid doesn't help (and if it does, we replace the ball with a detailed model of an ant). At the end of this article I will describe some algorithms that try to overcome these shortcomings.

Despite the shortcomings, the regular gird performs quite well, and is reasonably easy to implement. I implemented the algorithm described in a paper titled *"A Faster Voxel Traversal Algorithm for Ray Tracing"* by Amanatides & Woo; a link to this paper is available at the end of this article.

Grid Construction

First, the easy part. For the construction of the grid I have added a new primitive to scene.h / .cpp: An axis aligned box. This new primitive is used to represent cells in the grid, and also the bounding box of the grid. The implementation comes complete with an intersect method and normal calculation for the intersection point, so you can now add boxes to the scene. It's not very useful though, as you can't rotate the box: For the regular grid axis aligned boxes are sufficient.

Note: Between writing this and sending it to Kurt, this got slightly changed; there is now an aabb class in common.h, which is used in turn by the Box primitive. The reason for this is that you don't want to attach a material etc. to a bounding box. Strictly, the Box primitive is no longer necessary, but I left it in as it's a nice change after all those stupid spheres. :)

Next, all primitives now implement a new method: IntersectBox. This is used to determine whether or not a primitive is (partially) inside a cell of the grid. These are not terribly complicated; have a look at scene.cpp if you are curious.

Now the grid can be constructed. This is done by the (new) BuildGrid method in scene.cpp. Initially this code looked like this:

```
void Scene::BuildGrid()
{
	m_Grid = new ObjectList*[GRIDSIZE * GRIDSIZE * GRIDSIZE];
	memset( m_Grid, 0, GRIDSIZE * GRIDSIZE * GRIDSIZE * 4 );
	vector3 p1(-10, -5, -6), p2( 20, 8, 30 );
	float dx = (p2.x - p1.x) / GRIDSIZE;
	float dy = (p2.y - p1.y) / GRIDSIZE;
	float dz = (p2.z - p1.z) / GRIDSIZE;
	m_Extends = Box( p1, p2 - p1 );
	vector3 size = (p2 - p1) * (1.0f / GRIDSIZE);
	for ( int x = 0; x < GRIDSIZE; x++ ) for ( int y = 0; y < GRIDSIZE; y++ ) for ( int z = 0; z < GRIDSIZE; z++ )
	{
		int idx = x + y * GRIDSIZE + z * GRIDSIZE * GRIDSIZE;
		vector3 pos( p1.x + x * dx, p1.y + y * dy, p1.z + z * dz );
		for ( int p = 0; p < m_Primitives; p++ )
		{
			if (m_Primitive[p]->IntersectBox( pos, size ))
			{
				ObjectList* l = new ObjectList();
				l->SetPrimitive( m_Primitive[p] );
				l->SetNext( m_Grid[idx] );
				m_Grid[idx] = l;
			}
		}
		idx++;
	}
}
```

For each cell, this code checks all primitives to see if they are in the cell or not. If they are, they are added to a list of objects, and a pointer to the list is added to the cell. This works fine for a couple of primitives, but when the number of primitives increases, it gets a bit slow: The code loops over all the primitives for each cell, and does an intersection test for each iteration. That's x * y * z * primitives intersections (i.e., a lot). The final version is faster:

```
void Scene::BuildGrid()
{
	// initialize regular grid
	m_Grid = new ObjectList*[GRIDSIZE * GRIDSIZE * GRIDSIZE];
	memset( m_Grid, 0, GRIDSIZE * GRIDSIZE * GRIDSIZE * 4 );
	vector3 p1(-14, -5, -6), p2( 14, 8, 30 );
```

```
        // calculate cell width, height and depth
        float dx = (p2.x - p1.x) / GRIDSIZE, dx_reci = 1.0f
/ dx;
        float dy = (p2.y - p1.y) / GRIDSIZE, dy_reci = 1.0f
/ dy;
        float dz = (p2.z - p1.z) / GRIDSIZE, dz_reci = 1.0f
/ dz;
        m_Extends = aabb( p1, p2 - p1 );
        m_Light = new Primitive*[MAXLIGHTS];
        m_Lights = 0;
        // store primitives in the grid cells
        for ( int p = 0; p < m_Primitives; p++ )
        {
                if (m_Primitive[p]->IsLight())
m_Light[m_Lights++] = m_Primitive[p];
                aabb bound = m_Primitive[p]->GetAABB();
                vector3 bv1 = bound.GetPos(), bv2 =
bound.GetPos() + bound.GetSize();
                // find out which cells could contain the
primitive (based on aabb)
                int x1 = (int)((bv1.x - p1.x) * dx_reci), x2
= (int)((bv2.x - p1.x) * dx_reci) + 1;
                x1 = (x1 < 0)?0:x1, x2 = (x2 > (GRIDSIZE -
1))?GRIDSIZE - 1:x2;
                int y1 = (int)((bv1.y - p1.y) * dy_reci), y2
= (int)((bv2.y - p1.y) * dy_reci) + 1;
                y1 = (y1 < 0)?0:y1, y2 = (y2 > (GRIDSIZE -
1))?GRIDSIZE - 1:y2;
                int z1 = (int)((bv1.z - p1.z) * dz_reci), z2
= (int)((bv2.z - p1.z) * dz_reci) + 1;
                z1 = (z1 < 0)?0:z1, z2 = (z2 > (GRIDSIZE -
1))?GRIDSIZE - 1:z2;
                // loop over candidate cells
                for ( int x = x1; x < x2; x++ ) for ( int y
= y1; y < y2; y++ ) for ( int z = z1; z < z2; z++ )
                {
                        // construct aabb for current cell
                        int idx = x + y * GRIDSIZE + z *
GRIDSIZE * GRIDSIZE;
                        vector3 pos( p1.x + x * dx, p1.y + y
* dy, p1.z + z * dz );
                        aabb cell( pos, vector3( dx, dy, dz
) );
                        // do an accurate aabb / primitive
intersection test
                        if (m_Primitive[p]->IntersectBox(
cell ))
                        {
                                // object intersects cell;
add to object list
                                ObjectList* l = new
ObjectList();
                                l->SetPrimitive(
m_Primitive[p] );
                                l->SetNext( m_Grid[idx] );
                                m_Grid[idx] = l;
                        }
                }
        }
}
```

This time the code first asks the primitive for it's axis-aligned bounding box, so that only the cells that could contain the primitive are checked.

Note that this code also sets the extents of the grid. This is important, as rays won't be traced outside the grid. This does mean that a scene with an infinite ground plane suddenly won't be infinite anymore.

Grid Traversal

Once the grid is constructed, we can start traversing it. This is done using a 3D version of the DDA algorithm. It works like this (the following is heavily based on the paper by Amanatides & Woo):

First, we determine in which cell the ray starts. If the ray starts outside the grid, it must be advanced to a grid boundary; if it doesn't even hit the bounding box of the grid, obviously we are done with the ray. The coordinates of the first cell are stored in 'X', 'Y' and 'Z' (those are integers); we can use these coordinates to look up cells while traversing the grid.

Next, we determine how far we can travel along the ray before hitting a cell boundary. We do this for the axii; the result is stored in tMaxX, tMaxY and tMaxZ. This is illustrated in figure 3:

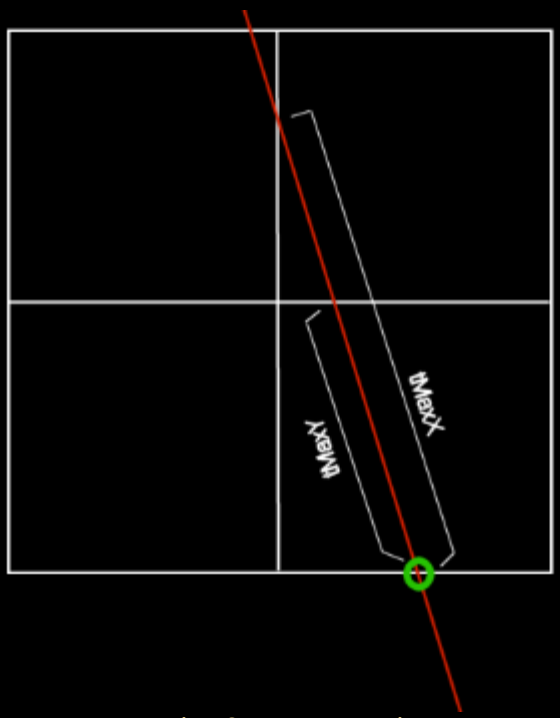


Fig. 3: tMaxX and tMaxY

In this image, the ray is first advanced to the boundary of the grid (the green circle). From that point, we have to move 'tMaxX' to cross the vertical cell boundary, or tMaxY to cross the horizontal cell boundary. The minimum of these three values determines how far we can travel before we hit the first cell boundary.

Finally, we determine how far we must move along the ray to cross one cell. Again, we do this for x, y and z; the result is stored in tDeltaX, tDeltaY and tDeltaZ. In the above image, tDeltaY equals tMaxY.

Once we have the variables set up, we can start stepping through the grid. Stepping through the cells is illustrated in the following pseudocode:

```
loop
{
        if (tMaxX < tMaxY)
        {
                tMaxX = tMaxX + tDeltaX;
                X = X + stepX;
        }
        else
        {
                tMaxY = tMaxY + tDeltaY;
                Y = Y + stepY;
        }
}
```

In this code, stepX and stepY are used: stepX is set to '-1' if the sign of the x direction of the ray is negative, or 1 otherwise; stepY is calculated in the same way.

The 3D version is a bit longer, but works in the same way. Have a look at the 'FindNearest' method in raytracer.cpp for the actual implementation.

Processing Grid Cells

Each cell of the grid contains a pointer to a list of objects. Often, this pointer will be NULL, so no intersection tests will have to be performed. When a non-empty object list is encountered, the objects in the list are intersected.

Take a look at the following situation:

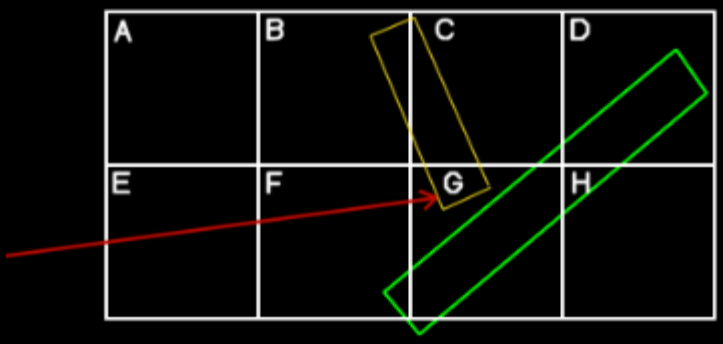


Fig. 4: Grid traversal problem

In this example case, a ray enters the grid at cell 'E', advances to cell 'F' and intersects the rotated yellow rectangle in cell 'G'. Right, but first it will encounter the green rectangle, in cell 'F'. And it intersects this rectangle, although the intersection point is not inside cell 'F'.

This problem could be solved by checking that intersection points are actually inside the cell that we are testing. But that's not a very nice solution: In the above example, the ray would miss the green rectangle in cell 'F', but it would test it *again* in cell 'G'. Two intersections for one primitive, that's not the way we want to go.

Amanatides & Woo to the Rescue

The paper that I mentioned before presents a solution to this problem.

First, the fact that the intersection is not occurring in the current cell is detected. While the intersection is outside the current cell, the ray is advanced to the next cell, and the objects in those subsequent cells are also tested. In the above case, this would mean that after finding an intersection (in cell 'H', but found while testing cell 'F') the algorithm will still check cells 'G' and 'H'.

Multiple intersection tests are prevented by labeling the rays with a unique number, and storing this number during an intersection test, in the primitive. This way, when the same ray tries to intersect the same primitive again, this is easily detected, and the test is skipped.

Implementation

There are some more implementation details, but the basics of the algorithm should be pretty clear. The full implementation can be found in the 'FindNearest' method in raytracer.cpp. Full discussion of the algorithm is in the mentioned pdf.

Performance Analysis

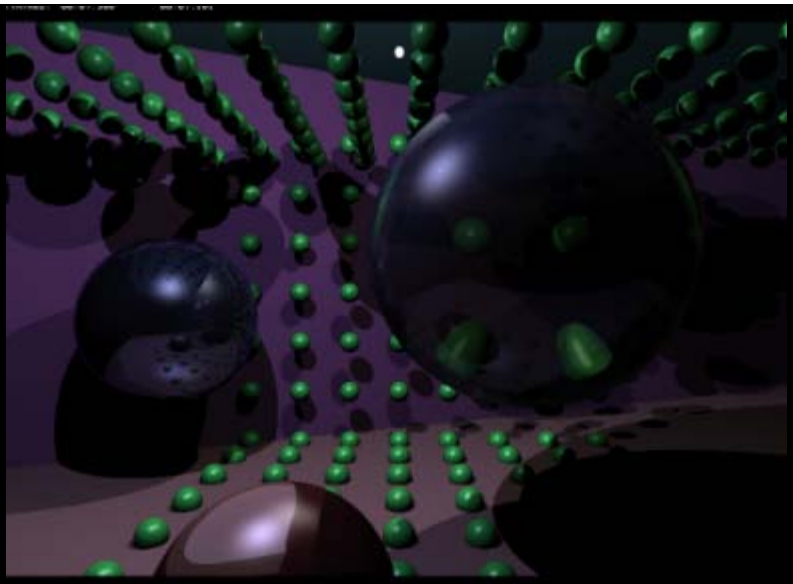
So how does it perform? You may be a bit disappointed to hear that the speed of the 'refraction scene' of the third article only doubled. At least initially I was disappointed.

But it's not as bad as it looks: If you increase the complexity of the scene, the speed will not be affected very much, in contrast to the original approach. The following table shows this:

primitives	64	128	256
Naïve implementation	**9.444s**	**17.886s**	**33.508s**
Regular grid, 8x8x8	**5.147s**	**5.799s**	**7.551s**
Regular grid, 16x16x16	**5.448s**	**6.069s**	**7.091s**

So, while it's only twice as fast at 64 primitives, it's three times faster at 128 primitives, and more than 4 times faster at 256 primitives.

But why is it still so slow? The main reason for this is the expensive setup for the grid traversal algorithm: Intersecting a couple of spheres takes less time than setting up the variables. And, rays that start outside the grid must first be advanced to the boundaries of the grid. This requires a box intersection and adjusting the ray origin. This makes the startup cost for the regular grid traversal quite high. Stepping is not free either: Try increasing the grid resolution and you'll notice that speed drops even further. For complexer scenes this will probably be necessary; careful tweaking is recommended.



256 primitives in 7 seconds

Alternatives

The regular grid is a rather simple spatial subdivision. Its regularity makes it easy to implement, and you don't have to take a lot of decisions when building the grid. Stepping is a bit complex but doable. Speed gain is OK. But isn't there anything better?

Sure there is. Some alternatives:

Nested grids: Imagine a grid cell that contains many primitives is subdivided in a grid. This would save many expensive intersection tests. And, it would let you use large empty cells in areas that contain little detail.

Kd-trees: These seem to be the best choice at the moment for complex scenes. A Kd-tree is an axis-aligned BSP tree. What this means is that you start by splitting the scene bounding box along one axis. Next, you split both halves using the next axis. This process is repeated, using the three axii in turn. The position of the splitting plane can be specified; it's usually chosen in a way that balances the number of primitives to the left and the right of the plane, while minimizing primitive splits.

A simpler version is the *octree*: In this version, you can't specify the position of the splitting plane; each cube is simply split in 8 equally sized cubes. As long as a cube contains more primitives than a certain threshold, cubes are split recursively. This results in large empty cubes for empty scene areas, and many smaller cubes for detailed areas.

There are more spatial subdivision schemes, but these are the most well-known.

Final Words

That's all for this article. The new gained performance paves the way for some new time consuming tricks, which I will discuss in the next article: Soft shadows and other effects that require lots of rays (but produce pretty pictures).

By the way, if you improve the raytracer, please share your experiences and code! New primitives, cool scenes, speed enhancements are all welcome. If it's good stuff, I will add it to the raytracer and distribute it with the final package.

Have fun,

Jacco Bikker, a.k.a. "The Phantom"

Links:

- raytracer4.zip
- "A faster voxel traversal algorithm for ray tracing", pdf
- Heuristic ray shooting algorithms by Vlastimil Havran, a very detailed discussion of various spatial subdivisions.

Article Series:

Raytracing Topics & Techniques - Part 5: Soft Shadows

Return to The Archives

by Jacco Bikker (05 November 2004)

Introduction

It's time to do something useful with the new speed that the regular grid brought us. Ray tracing speed is just like money: You always spend twice the amount you actually have. :)

One of the finest things to spend it on is soft shadows. They are reasonably easy to do, so this article will be a bit lighter than the previous one. Since we are cranking up image quality anyway, I will also show how to add diffuse reflections.

Enjoy the ride.

Area Lights

A typical scene with a point light source is shown in the following image:

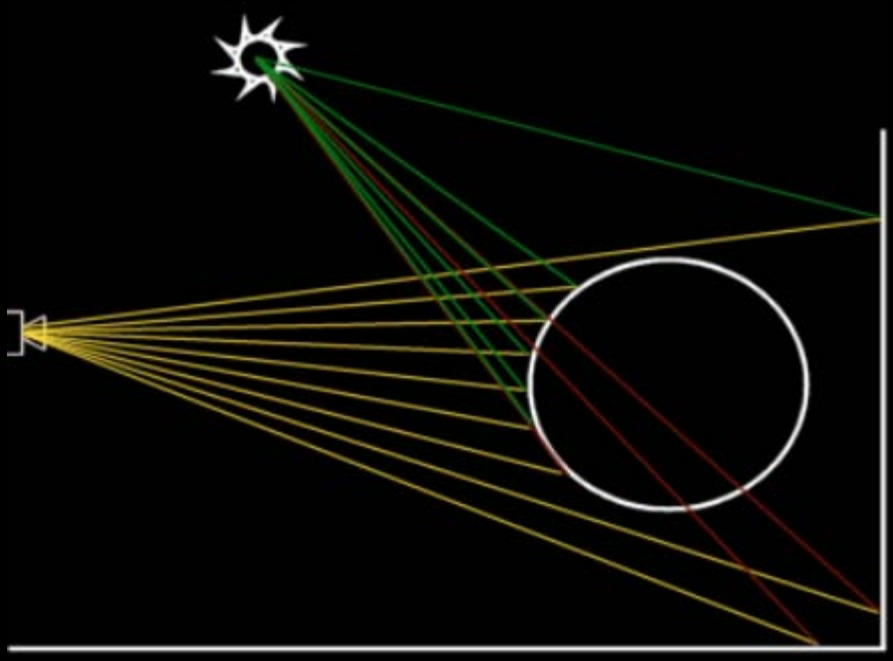


Fig. 1: Point light source

In this picture, intersection points of the primary rays with the primitives are used to spawn secondary rays towards the light source to check whether the point is shadowed or not. Red lines are shadow rays that hit something before reaching the light source.

Now consider the same scene, but this time with an area light source instead of the point light source:

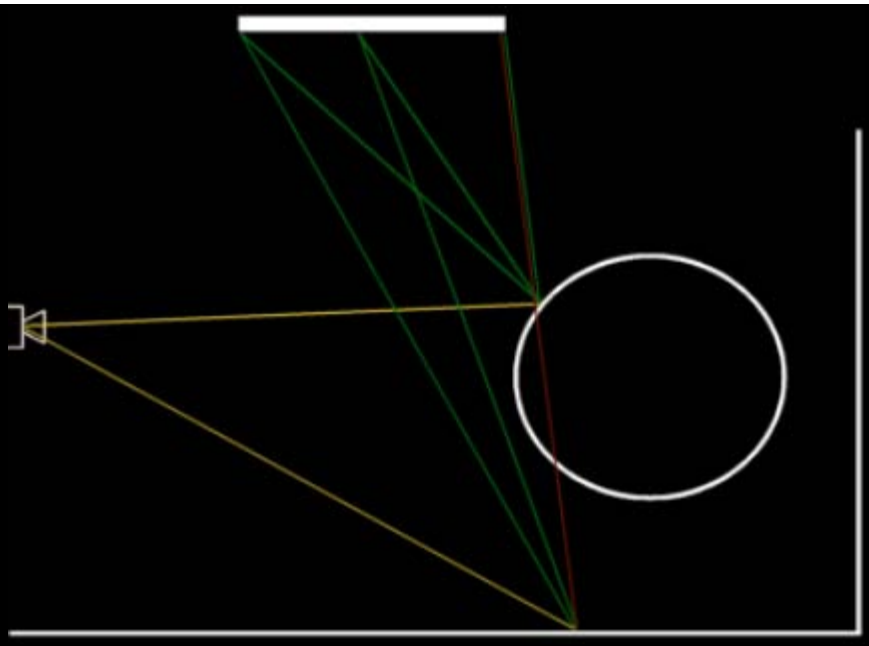


Fig. 2: Area light source

I have only drawn two primary rays in this case. The intersection point on the sphere is used to spawn *three* shadow rays towards the area light. In this case, none of them is occluded, so the point is fully lit by the area light.

The intersection point on the floor plane also spawns three rays. This time however only two reach the area light without problems.

Approximating Illumination

Why three rays? Well, I used three rays because I didn't feel like drawing a hundred green lines, obviously. :) But seriously, to determine how much light an area light casts on a particular spot, you would have to calculate how much of the area of the light source is visible from that point. While this is theoretically possible, it's not exactly practical. Therefore, we approximate the result by sending multiple rays towards the light source and averaging the results. When the number of rays goes to infinity, the solution converges to the correct result.

Obviously, an unlimited amount of rays is not practical either, so we will have to find out just how many rays are needed.

Coding Time

I moved the code that determines the visibility of a light source as seen from an intersection point to a separate method, 'CalcShade'. Here it is:

```
float Engine::CalcShade( Primitive* a_Light, vector3 a_IP,
vector3& a_Dir )
{
	float retval;
	Primitive* prim = 0;
	// handle point light source
	retval = 1.0f;
	a_Dir = ((Sphere*)a_Light)->GetCentre() - a_IP;
	float tdist = LENGTH( a_Dir );
	a_Dir *= (1.0f / tdist);
```

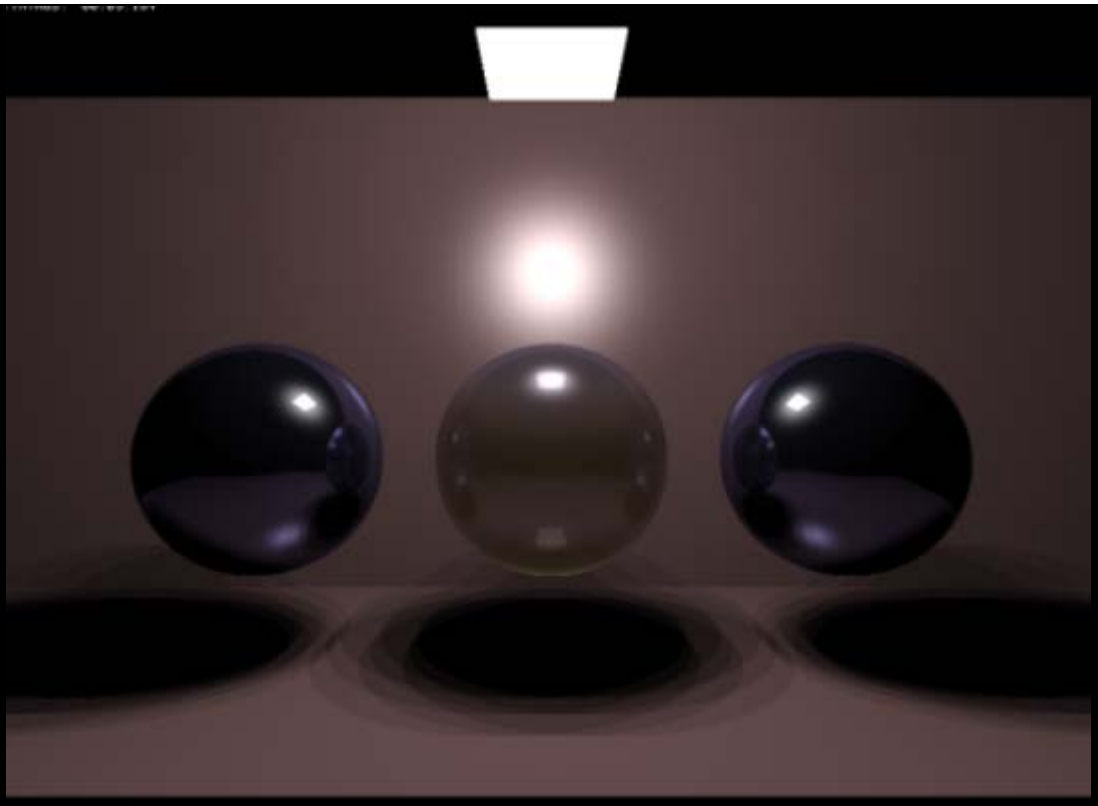
```
        FindNearest( Ray( a_IP + a_Dir * EPSILON, a_Dir,
++m_CurID ), tdist, prim );
        if (prim != a_Light) retval = 0;
        return retval;
}
```

Nothing special here, it's the same code as used before. Now let's add some support for area lights:

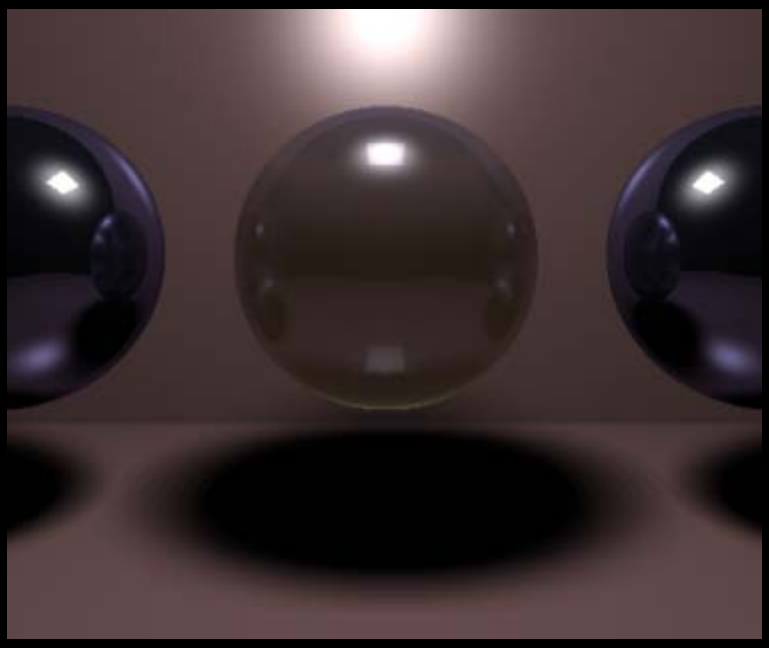
```
float Engine::CalcShade( Primitive* a_Light, vector3 a_IP,
vector3& a_Dir )
{
        float retval;
        Primitive* prim = 0;
        if (a_Light->GetType() == Primitive::SPHERE)
        {
                // handle point light source
                retval = 1.0f;
                a_Dir = ((Sphere*)a_Light)->GetCentre() -
a_IP;
                float tdist = LENGTH( a_Dir );
                a_Dir *= (1.0f / tdist);
                FindNearest( Ray( a_IP + a_Dir * EPSILON,
a_Dir, ++m_CurID ), tdist, prim );
                if (prim != a_Light) retval = 0;
        }
        else if (a_Light->GetType() == Primitive::AABB)
        {
                retval = 0;
                Box* b = (Box*)a_Light;
                a_Dir = (b->GetPos() + 0.5f * b->GetSize())
- a_IP;
                NORMALIZE( a_Dir );
                for ( int x = 0; x < 3; x++ ) for ( int y =
0; y < 3; y++ )
                {
                        vector3 lp( b->GetPos().x + x, b-
>GetPos().y, b->GetPos().z + y );
                        vector3 dir = lp - a_IP;
                        float ldist = (float)LENGTH( dir );
                        dir *= 1.0f / ldist;
                        if (FindNearest( Ray( a_IP + dir *
EPSILON, dir, ++m_CurID ), ldist, prim ))
                                if (prim == a_Light) retval
+= 1.0f / 9;
                }
        }
        return retval;
}
```

In this code, 'SPHERE' primitives are still handled as point light sources. Boxes are treated different: A series of rays is cast towards the surface of the box to calculate the average visibility. In the above case, 3x3 rays are generated, so each ray accounts for 1/9th of the full light intensity.

Here is a screenshot of the result:



Hmm. Banding. Perhaps we didn't use enough rays? Let's see how 8x8 rays look:



Definitely much better. But, it took 1 minute and 10 seconds to render 3 balls this way. And I still have the feeling that I see some banding, and that this banding would be worse if I zoomed in on the shadow.

Poker Faces and Pimples

The banding that you see when sampling 3x3 or 8x8 is caused by the fact that basically you're rendering 9 or 64 shadows. No matter how many samples you take this way, there

will always be banding. But there is a better way.

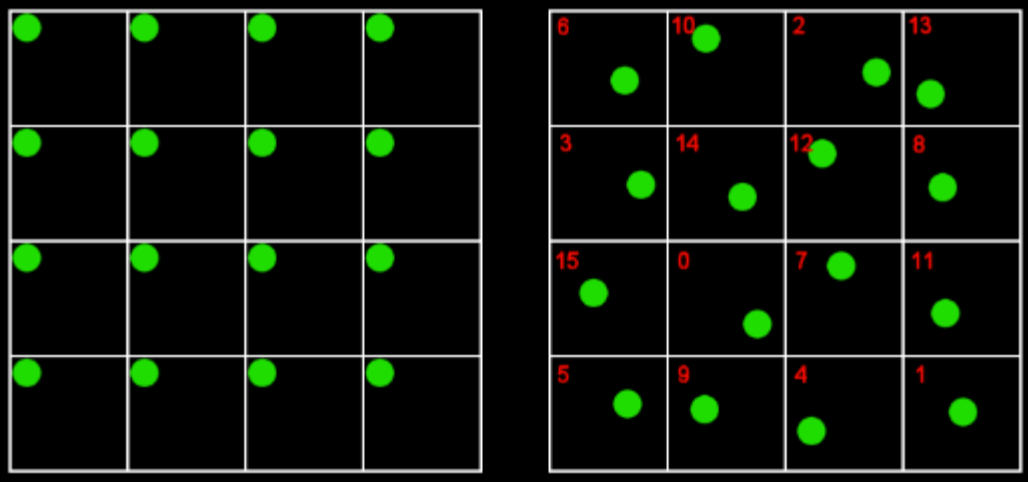


Fig. 3: Random samples

In figure three, the left image shows the approach we've taken so far: We took samples in a regular grid. This causes banding. The image to the right shows a different approach: We still use the grid, but inside the grid cells, we use a random offset. This replaces the banding with noise.

So is noise better than banding? The answer is: It definitely is. Our eyes are used to noise: When you try to find your way around the house in the dark, you will see plenty of noise as your eyes try to process the faint lighting.

I think here some terminology is in place: The above sampling process (using multiple rays to determine a color) is called distribution ray tracing, or Monte Carlo ray tracing. What we are trying to do is to resolve an integral: We are estimating the area of the surface of the light source that we can see from a certain point. This is very hard to solve analytically. Monte Carlo says that when you take enough random samples, the average will converge to the correct solution.

In the right image I have numbered the 16 cells. This is useful when you need more than 16 samples: For sample 'x' you can use cell 'x % 16'. This way, you can increase the number of samples without reformatting the grid. Of course, we could also have used a 2x2, or 8x8 or 1x1 grid, but 4x4 seems to be the best balance: 1x1 does not guarantee a good distribution of the rays over the entire area, and neither does 2x2. 8x8 means that we must sample at least 64 rays to cover all cells, and that might be a bit too much in most cases.

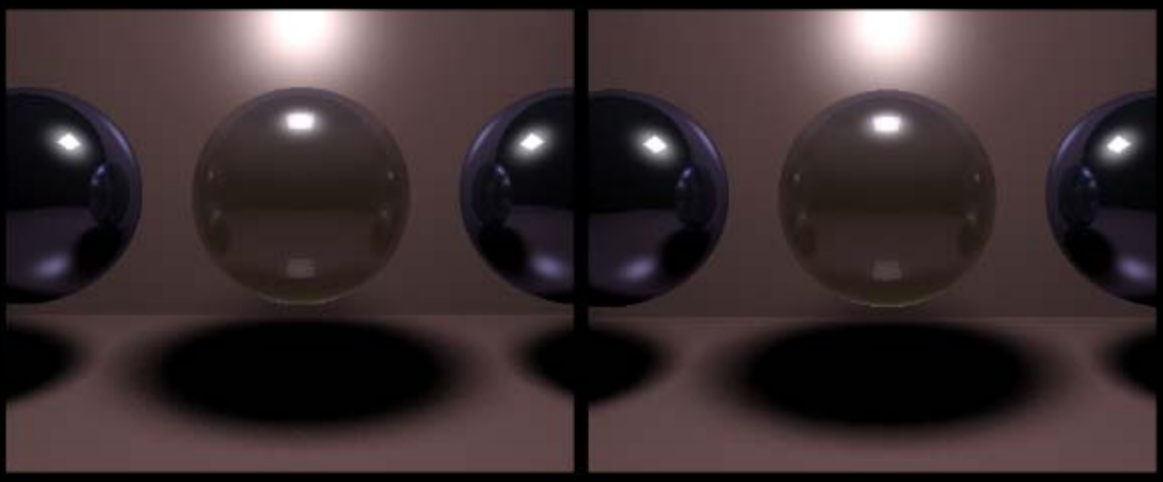
Here is the modified sampling code:

```
float deltax = b->GetSize().x * 0.25f, deltay = b-
>GetSize().z * 0.25f;
for ( int i = 0; i < SAMPLES; i++ )
{
        vector3 lp( b->GetGridX( i & 15 ) + m_Twister.Rand()
* deltax, b->GetPos().y, b->GetGridY( i & 15 ) +
m_Twister.Rand() * deltay );
        vector3 dir = lp - a_IP;
        float ldist = (float)LENGTH( dir );
        dir *= 1.0f / ldist;
        if (FindNearest( Ray( a_IP + dir * EPSILON, dir,
++m_CurID ), ldist, prim ))
                if (prim == a_Light) retval += m_SScale;
}
```

Some notes about this code:

- The code uses a 'GetGridX' and 'GetGridY' method from the Box class. These return the x and y for cell 'i % 16' (the red numbers in figure 3).
- The code also uses a new random number generator: The Mersenne Twister. This is a random number generator that gives better results than the build-in Rand() function, and is a bit faster as well. The Mersenne Twister resides in twister.cpp and twister.h.

'SAMPLES' is a constant defined in common.h. When we set it to 32 and 64 the results looks like this:



Rendering with 32 samples took 30 seconds; with 64 samples it took 58 seconds.

Diffuse Reflections

The same technique can be applied to reflections. Figure 4 illustrates what we are trying to do.

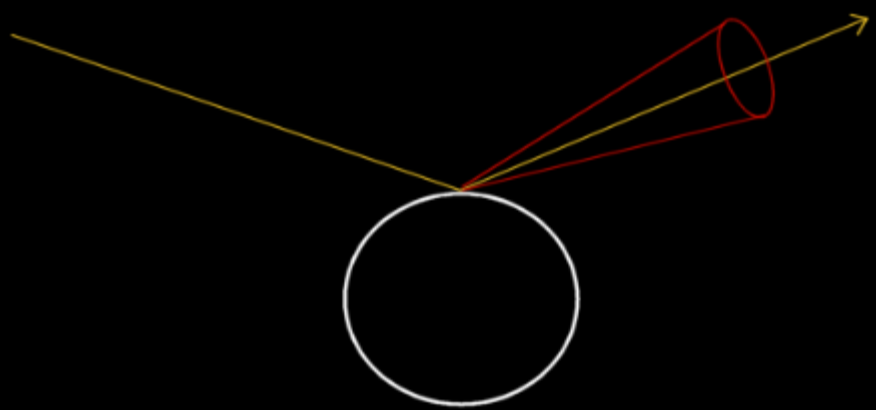


Figure 4: Diffuse reflection

Instead of reflecting in a precisely calculated direction, we want to calculate the average of all rays inside the red cone. It would probably look better if there where more rays in the

centre of the cone than at the edges, but let's ignore that for a moment.

Here is the code that performs the diffuse reflection:

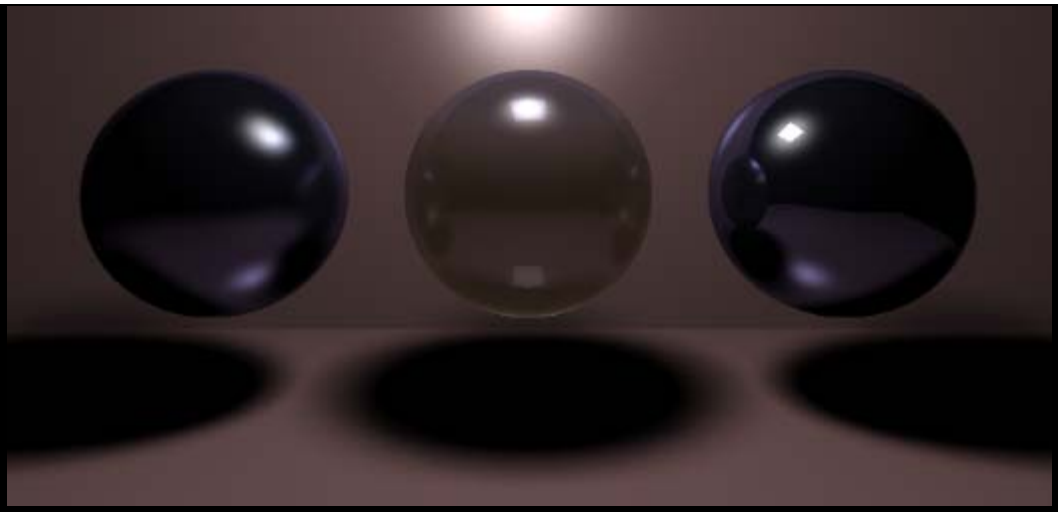
```
float drefl = prim->GetMaterial()->GetDiffuseRefl();
if ((drefl > 0) && (a_Depth < 2))
{
	// calculate diffuse reflection
	vector3 N = prim->GetNormal( pi );
	vector3 RP = a_Ray.GetDirection() - 2.0f * DOT(
a_Ray.GetDirection(), N ) * N;
	vector3 RN1 = vector3( RP.z, RP.y, -RP.x );
	vector3 RN2 = RP.Cross( RN1 );
	refl *= m_SScale;
	for ( int i = 0; i < SAMPLES; i++ )
	{
		float xoffs, yoffs;
		do
		{
			xoffs = m_Twister.Rand() * drefl;
			yoffs = m_Twister.Rand() * drefl;
		}
		while ((xoffs * xoffs + yoffs * yoffs) >
(drefl * drefl));
		vector3 R = RP + RN1 * xoffs + RN2 * yoffs *
drefl;
		NORMALIZE( R );
		float dist;
		Color rcol( 0, 0, 0 );
		Raytrace( Ray( pi + R * EPSILON, R,
++m_CurID ), rcol, a_Depth + 1, a_RIndex, dist );
		a_Acc += refl * rcol * prim->GetMaterial()-
>GetColor();
	}
}
```

In this code, the original reflection vector (here 'RP') is modified using two perpendicular vectors (RN1 & RN2), scaled by a random value that is not larger than the value set for the diffuse reflection. The 'do...while' makes sure the generated offset is inside the cone. Finally, a lot of rays are cast to determine an estimate of the diffuse reflection.

Note that I only perform diffuse reflection for primary rays; otherwise it's just too slow.

Also note that the cone is not nicely subdivided like we did for the shadows. The sad result is that the color of the diffuse reflection converges even slower; 128 samples is just barely enough.

Here's an image generated using 128 samples:



The left sphere is rendered using a diffuse reflection value of 0.6; the middle uses 0.4 and the right one shows perfect reflection.

This image took 11 minutes to render... Don't try this at home, kids. :) But it's quite beautiful.

Final Words

This concludes the fifth installment of the ray tracing column. I tried to make clear that distribution ray tracing is a valuable extension to the basic ray tracing algorithm; it allows for all sorts of cool effects that just can't be done using single rays / secondary rays. Obviously, these techniques require huge amounts of processing power... Or time.

As usual, an updated VC6 project is available at the end of this article. You may want to disable the diffuse reflection for the spheres to reduce the rendering time to something reasonable. :)

In the next article I will show you how to add texturing and a camera to the raytracer.

That's all, have fun!

Jacco Bikker, a.k.a. "The Phantom"

Links:

- raytracer5.zip
- Ray Tracing News Guide - Condensed information and discussions from the Ray Tracing News mailing list. Great stuff.

Article Series:

Raytracing Topics & Techniques - Part 6: Textures, Cameras and Speed

Return to The Archives

by Jacco Bikker (10 November 2004)

Introduction

Welcome to the sixth installment of my ray tracing column. This time I would like to fix some things that I left out earlier: Texturing and a camera.

Before I proceed I would first like to correct an error in the previous article. Some readers (Bramz, Thomas de Bodt) pointed out that the way I calculate the diffuse reflection is not correct. Bramz explained that to calculate two perpendicular vectors to the reflected vector, you could use the following method:

1.Pick a random vector V.
2.Now the first perpendicular vector to the reflected vector R is RN1 = R cross V.
3.The second perpendicular vector is RN2 = RN1 cross R.

There are some obvious cases where this will not work, but these are easy to work around (e.g., if V = (0,0,1) and R = (0,0,1) then you should pick (1,0,0) for V instead).

Also, related to this: The correct term is 'glossy reflection', rather than diffuse reflection.

Now that the bugs are out of the way, let's proceed.

Perhaps it's good to explain the way these articles are written: Three months back I started working on a raytracer, after reading lots of stuff during my summer holidays. The stuff that interested me most is global illumination, so I pushed forward to get there as fast as possible. This means that I didn't really take a 'breadth-first' approach to ray tracing, but instead built more advanced features upon a simple basis. The raytracer that you see evolving with these articles used to be a very stripped-down version of my own version, with bits and pieces of the full thing being added every week.

Last week I added the first feature to the 'article tracer' that was not in the original project (the diffuse reflections). And now I feel it's necessary to add texturing and a free camera, which I also didn't implement yet.

The consequence of this is that the code that is included (as usual) with this article is pretty fresh, as it was written specifically for this article. It works though. :)

The Camera

So far we have worked with a fixed viewpoint (at 0,0,-5) and a fixed view direction (along the z-axis). It would be rather nice if we could look at objects from a different perspective. In modeling packages like 3DS Max this is usually done using a virtual camera.

In a ray tracer, a camera basically determines the origin of the primary rays, and their direction. This means that adding a camera is rather easy: It's simply a matter of changing the ray spawning code. We can still fire rays from the camera position through a virtual screen plane, we just have to make sure the camera position and screen plane are in the right position and orientation. Rotating the screen plane in position can be done using a 4x4

matrix.

Look At Me

A convenient way to control the direction of the camera is by using a 'look-at' camera, which takes a camera position and a target position.

Setting up a matrix for a look-at camera is easy when you consider that a matrix is basically a rotated unit cube formed by three vectors (the 3x3 part) at a particular position (the 1x3 part). We already have one of the three vectors: The z-axis of the matrix is simply the view direction. The x-axis of the matrix is a bit tricky: if the camera is not tilted, then the x-axis of the matrix is perpendicular to the z-axis and the vector (0, 1, 0). Once we have two axii of the matrix, the last one is simple: It's perpendicular to the other two, so we simply calculate the cross product of the x-axis and the z-axis to obtain the y-axis.

In the sample raytracer, this has been implemented in the InitRender method of the Raytracer class. This method now takes a camera position and a target position, and then calculates the vectors for the matrix:

```
vector3 zaxis = a_Target - a_Pos;
zaxis.Normalize();
vector3 up( 0, 1, 0 );
vector3 xaxis = up.Cross( zaxis );
vector3 yaxis = xaxis.Cross( -zaxis );
matrix m;
m.cell[0] = xaxis.x, m.cell[1] = xaxis.y, m.cell[2] =
xaxis.z;
m.cell[4] = yaxis.x, m.cell[5] = yaxis.y, m.cell[6] =
yaxis.z;
m.cell[8] = zaxis.x, m.cell[9] = zaxis.y, m.cell[10] =
zaxis.z;
m.Invert();
m.cell[3] = a_Pos.x, m.cell[7] = a_Pos.y, m.cell[11] =
a_Pos.z;
```

For once, this is more intuitive in code than in text. :)

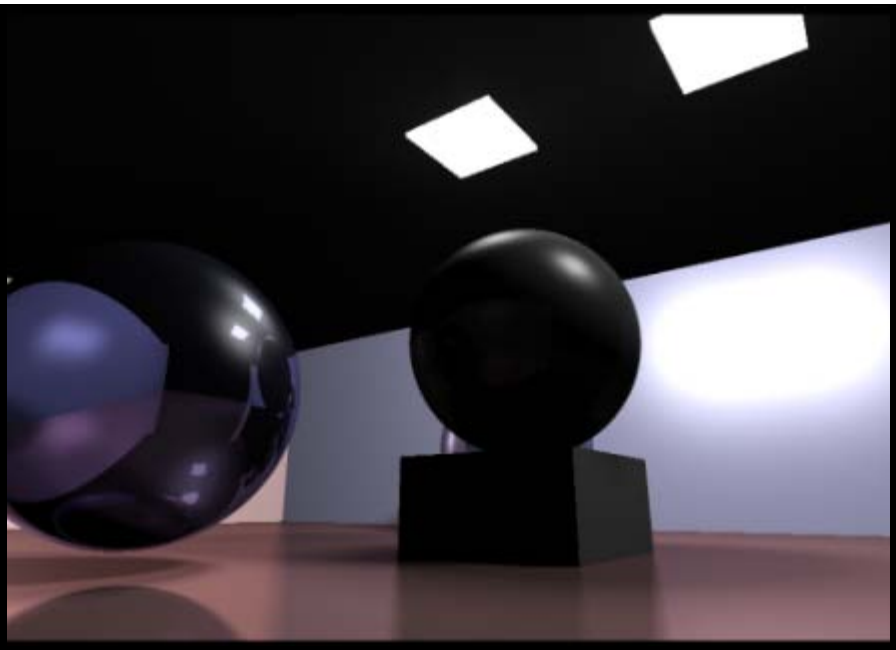
Note that the y-axis is calculated using the reversed z-axis, the image turned out to be upside down without this adjustment. I love trial-and-error. :)

Also note that the rotation part of the matrix needs to be inverted to have the desired effect.

Interpolate

The matrix is used to transform the four corners of the virtual screen plane. Once that is done, we can simply interpolate over the screen plane.

Here is a screenshot of the new camera system in action. Also note the cool diffuse reflection for the floor.



Texturing

Looking at the screenshot at the end of the previous paragraph, I really wonder why anyone would want texturing anyway. Graphics without textures have that distinct 'computerish' look to them, a lack of realism that just makes it nicer, in a way. I once played an FPS (I believe it was on an Amiga, might even have been Robocop) that didn't use any texturing at all; all detail like stripes on walls was added using some extra polygons.

Anyway. I suppose one should at least have the option to use textures. :)

Texture Class

I have added some extra code to accommodate this. First, a texture class: This class loads images from disk, knows about things like image resolution and performs filtering.

To load a file, I decided to use the TGA file format, as it's extremely easy to load. A TGA file starts with an 18 byte header, from which the image size is easily extracted. Right after that, the raw image data follows. The texture class stores the image data using three floats per color (for red, green and blue). This may seem a bit extreme, but it saves expensive int-to-float conversions during rendering and filtering, and it allows us to store HDR images.

Intersections & UV

Next, we need to know what texel to use at an intersection point. In a polygon rasterizer, you would simply interpolate texture coordinates (U, V) from vertex to vertex; for an infinite plane we have to think of something else. Have a look at the new constructor for the PlanePrim primitive:

```
PlanePrim::PlanePrim( vector3& a_Normal, float a_D ) :
        m_Plane( plane( a_Normal, a_D ) )
{
        m_UAxis = vector3( m_Plane.N.y, m_Plane.N.z, -
m_Plane.N.x );
```

```
        m_VAxis = m_UAxis.Cross( m_Plane.N );
}
```

For each plane, to vectors are calculated that determine how the texture is applied to the plane: one for the u-axis and one for the v-axis. Once we have these vectors, we can fill in the intersection point to obtain the u and v coordinates:

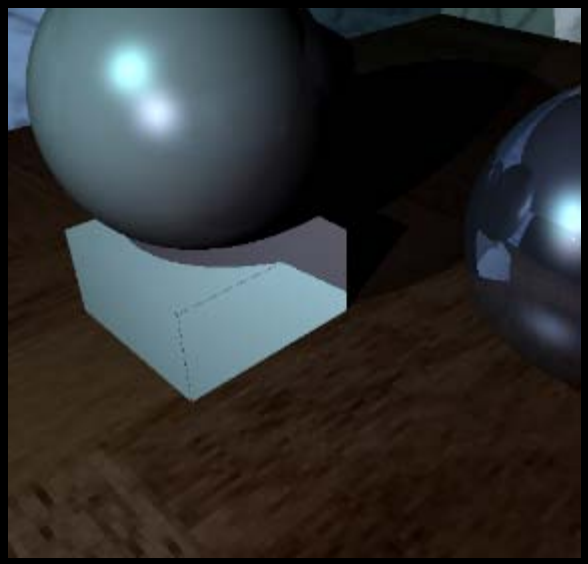
```
float u = DOT( a_Pos, m_UAxis ) * m_Material.GetUScale();
float v = DOT( a_Pos, m_VAxis ) * m_Material.GetVScale();
```

Scaling the texture is now a matter of stretching the u-axis and v-axis for the plane. This is done by multiplying the dot product by a scaling factor, which is retrieved from the Material class.

This procedure works well, and we can now apply textures to the walls, ceiling and floor. Note however that this is actually a rather limited way of texturing: We can't set the origin of the texture, and we can't specify an orientation for the texture. Adding this is straightforward though.

Filtering

Once the u and v coordinates for an intersection point are calculated, we can fetch the pixel from the texture map and use its color instead of the primitive color. The result looks like this:



Ouch. That looks like an old software rasterizer. We can easily solve this by applying a filter. There are many filtering techniques, but let's have a look at a simple one: the bilinear filter. This filter works by averaging 4 texels, based on their contribution to a virtual texel. The idea is illustrated in figure 1.

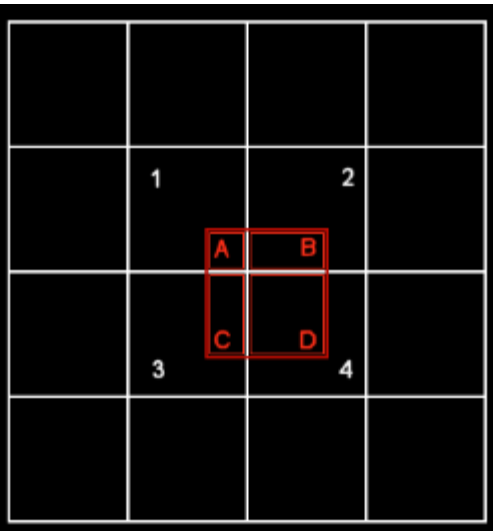


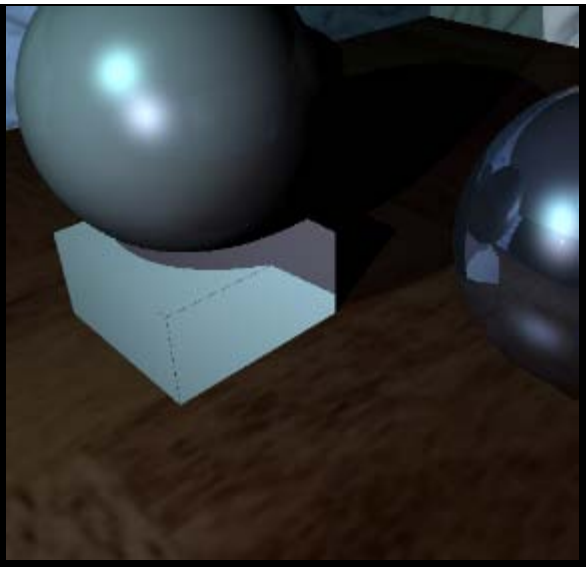
Figure 1: Bilinear filtering

The pixel that we would like to fetch from the texture is shown as a red square in the above image. Since we can only get textures at integer coordinates, we get one pixel instead; probably the one labeled as '1' in figure 1. Bilinear filtering considers all four texels covered by the red square. The color value of each texel is multiplied by the area that it contributes to the full texel, so '1' contributes far less then '4'.

This is implemented in the following code:

```
Color Texture::GetTexel( float a_U, float a_V )
{
	// fetch a bilinearly filtered texel
	float fu = (a_U + 1000.5f) * m_Width;
	float fv = (a_V + 1000.0f) * m_Width;
	int u1 = ((int)fu) % m_Width;
	int v1 = ((int)fv) % m_Height;
	int u2 = (u1 + 1) % m_Width;
	int v2 = (v1 + 1) % m_Height;
	// calculate fractional parts of u and v
	float fracu = fu - floorf( fu );
	float fracv = fv - floorf( fv );
	// calculate weight factors
	float w1 = (1 - fracu) * (1 - fracv);
	float w2 = fracu * (1 - fracv);
	float w3 = (1 - fracu) * fracv;
	float w4 = fracu *  fracv;
	// fetch four texels
	Color c1 = m_Bitmap[u1 + v1 * m_Width];
	Color c2 = m_Bitmap[u2 + v1 * m_Width];
	Color c3 = m_Bitmap[u1 + v2 * m_Width];
	Color c4 = m_Bitmap[u2 + v2 * m_Width];
	// scale and sum the four colors
	return c1 * w1 + c2 * w2 + c3 * w3 + c4 * w4;
}
```

This improves the rendered image quite a bit:



This time the texturing is blurry... Obviously, bilinear filtering is not going to fix problems caused by low-resolution textures, it merely makes it less obvious. Bilinear filtering works best for textures that are the right resolution, and also improves the quality of textures that are too detailed: Using no filter at all would result in texture noise, especially when animating.

Spheres

Finally, let's look at spheres. To determine u and v coordinates at an intersection point on a sphere, we use polar coordinates, r (the radial coordinate, or latitude) and *T* (the angular coordinate, or polar angle, or longitude).

We can calculate the latitude and the u coordinate as follows:

$$r = \arccos(-\vec{V}_n \cdot \vec{V}_p)$$
$$v = r / PI$$

Where $\vec{V}_n$ is a vector pointing to the north pole of the sphere, and $\vec{V}_p$ a vector from the centre of the sphere to the intersection point. 'v' is the desired v coordinate of a texel from the texture.

The longitude is calculated as follows:

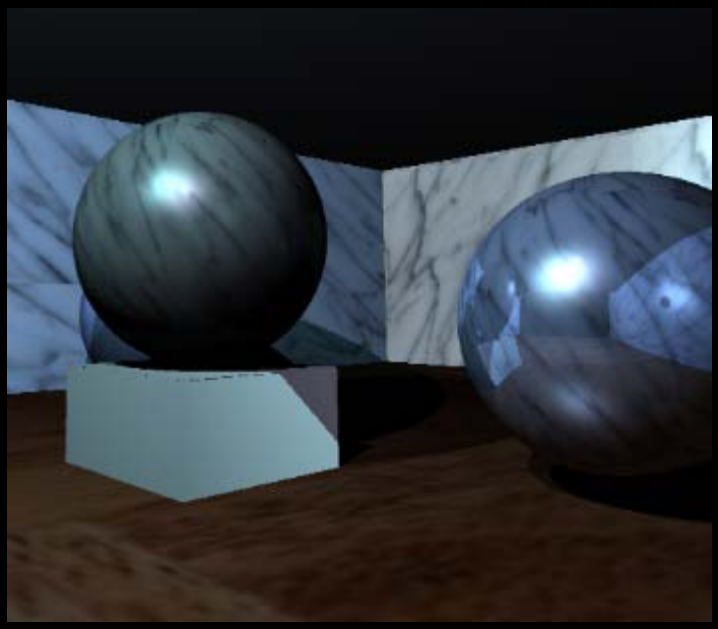
$$\Theta = (\arccos(\vec{V}_p \cdot \vec{V}_n) / \sin(r))) / (2 PI)$$
$$\text{if } (((\vec{V}_n \times \vec{V}_n) \cdot \vec{V}p) > 0) \; u = \Theta \text{ else } u = 1 - \Theta$$

Once we have the u and v coordinates for a texel on the sphere, we can use the same filtering as we used for texturing a plane. The full code:

```
Color Sphere::GetColor( vector3& a_Pos )
{
        Color retval;
        if (!m_Material.GetTexture()) retval =
```

```
m_Material.GetColor(); else
        {
                vector3 vp = (a_Pos - m_Centre) * m_RRadius;
                float phi = acosf( -DOT( vp, m_Vn ) );
                float u, v = phi *
m_Material.GetVScaleReci() * (1.0f / PI);
                float theta = (acosf( DOT( m_Ve, vp ) /
sinf( phi ))) * (0.5f / PI);
                if (DOT( m_Vc, vp ) >= 0) u = (1.0f -
theta) * m_Material.GetUScaleReci();
                                        else u = theta *
m_Material.GetUScaleReci();
                retval = m_Material.GetTexture()->GetTexel(
u, v ) * m_Material.GetColor();
        }
        return retval;
}
```

And an image of the new feature in action:



I'm sure it will look better with a nice earth texture on it. :)

Importance Sampling

One last thing that I would like to discuss briefly in this article is importance sampling. A recursive raytracer spawns secondary rays at each intersection point. In case of soft shadows and diffuse reflections, this can result in enormous amounts of rays. An example: A primary ray hits a diffuse reflective surface, spawns 128 new rays, and each of these rays probe an area light using 128 shadow rays. Or worse: Each of these rays could hit another diffuse reflective surface, and *then* probe area light sources. Each primary ray would spawn an enormous tree of rays, and each secondary ray would contribute to the final color of a pixel only marginally.

This situation can be improved using importance sampling. For the first intersection, it's still a good idea to spawn lots of secondary rays to guarantee noise-free soft shadows and high quality diffuse reflections. The secondary rays however could do with far less samples: A

soft shadow reflected by a diffuse surface will probably look the same when we take only a couple of samples.

Take a look at the new method definitions for Engine::Raytrace and Engine::CalcShade (from raytrace.h):

```
Primitive* Raytrace( Ray& a_Ray, Color& a_Acc, int a_Depth,
float a_RIndex, float& a_Dist, float a_Samples, float
a_SScale );

float CalcShade( Primitive* a_Light, vector3 a_IP, vector3&
a_Dir, float a_Samples, float a_SScale );
```

The two new parameters (a_Samples and a_SScale) control the number of secondary rays spawned for shading calculations and diffuse reflections. For diffuse reflections, the number of samples is divided by 4:

```
Raytrace( Ray( pi + R * EPSILON, R, ++m_CurID ), rcol,
a_Depth + 1, a_RIndex, dist, a_Samples * 0.25f, a_SScale * 4
);
```

'a_SScale' is multiplied by 4 in this recursive call to Engine::Raytrace, sine the contribution of individual rays increases by a factor 4 if we take 4 times less samples.

In the new sample raytracer included with this article, perfect reflections spawn only one half of the number of secondary rays that primary rays cast; refracted rays also lose 50% of their samples, and diffuse reflections skip 75%.

Improved Importance

This is only a very simple implementation of importance sampling. You could tweak importance of new rays based on material color (reflections on dark surfaces will contribute less to the final image), on reflectance (a surface that is only slightly reflective requires less precision), the length of the path that a refracted ray takes through a substance and so on.

Final Words

Ray tracing is quite a broad subject. Things we have covered so far:

- Spawning primary rays, ray / primitive intersections and diffuse shading (article one);
- Secondary rays, reflections, the Phong illumination model and shadows (article two);
- Refractions, Beer's law and supersampling (article three);
- Spatial subdivisions and regular grids (article four);
- Area lights, Monte Carlo approximation and diffuse reflections (article five);
- Camera's, look-at matrices, texturing of planes and spheres, bilinear filtering and importance sampling (this article).

For most of these topics I only scratched the surface of what you can do. For example, besides bilinear filtering there is trilinear filtering and anisotropic filtering, to name a few.

You could spend a lot of time on a subject like that. The same goes for Phong: This is really a relatively inaccurate model of how light behaves in real life. You could read up on Blinn or Lafortune, for better models.

And then there are the topics that I didn't cover at all, all worth a lifetime of exploration and research:

- Real-time ray tracing;
- Image based lighting;
- (Bi-directional) path tracing and photon maps (or global illumination in general);
- Triangles, cylinders, torii, cones and other primitives;
- Bump mapping and displacement mapping;
- Procedural textures;
- Distributed ray tracing;
- Non-Photorealistic Rendering...

The list is long. And that is good, because it's a wonderful world to explore. It becomes clear however that you can't do it all, unless you make it a full-time job, perhaps. You have to perform importance sampling. :)

For the next article I have some very good stuff for you: Highly optimized kd-tree code, pushing the performance of the ray tracer to 1 million rays per second on a 2Ghz machine. The scene from article four is rendered in 1 second using this new code. I will also show how to add triangles to the scene.

After that, I have a problem: The writing has caught up with the coding. This means that I am not sure if I will be able to keep the pace of one article each week after article 7. I'll keep you informed.

Download: raytracer6.zip

See you next time,
Jacco Bikker, a.k.a. "The Phantom"

Article Series:

Raytracing Topics & Techniques - Part 7: Kd-Trees and More Speed

Return to The Archives

by Jacco Bikker (23 November 2004)

Introduction

OK, this is it: The 7th installment of the ray tracing tutorial series. Over the last few weeks I worked very hard to improve the speed of the ray tracer, and I am quite proud of the results: Render times for the scene in article 4 (you know, the sphere grids) dropped from 7 seconds to less than 2 seconds. To put this power to good use, I added a triangle primitive, and a 3ds file loader.

The downside is that the number of changes to the raytracer is enormous. I removed the plane and box primitives, put the triangle and sphere in a single class, removed the regular grid, replaced it with a kd-tree, added a memory manager to align objects in a cache friendly manner, exchanged Phong's expensive pow() for Schlicks slick model, gave the lights their own class and so on. For this article I suggest you unpack the sample code first, you're going to need it while reading this article. As I understand from the feedback that I got that many people like to have the stuff explained without code as much as possible, I'm going to do my best to limit 'code-only' explanations as much as possible.

The good part is that I'm not going to get away with 9 pages in Word for this article (typical size for the previous articles).

Triangles

Before we dive into the hardcore kd-tree generation and traversal, let's first have a look at ray-triangle intersections.

An efficient way to calculate the intersection point of a ray and a triangle is to use barycentric coordinates. It works like this:

First, we calculate the signed distance of the ray to the plane of the triangle. This is the same test that we used for the ray-plane intersections. If the ray does not intersect the plane, we are done (this happens if the ray is parallel to the plane, or starts behind the plane). If the ray intersects the plane, we can easily calculate where this happens, and the problem is reduced to determining whether the intersection point is inside the triangle.

Barycentric Coordinates

This is where the barycentric coordinates come in handy. Barycentric coordinates were invented by mister M&#65533;bius (the guy that invented the famous one-sided belt). I'm not covering them in too much detail here; if you want to know all about them then I suggest you check out the MathWorld explanation: http://mathworld.wolfram.com/BarycentricCoordinates.html.

It works like this: Suppose you have a 2D line, from P1 to P2. One way of representing this line is this:

```
P = P1 + a (P2 – P1)
```

As long as a is between 0 and 1, we are on the line segment between P1 and P2. We can rewrite the formula like this:

```
P = a1 P1 + a2 P2
```

In this formula, a1 and a2 are the barycentric coordinates of point P with respect to the end points P1 and P2. Note that the sum of a1 and a2 must be 1 if we want P to lie between P1 and P2. We can extend this to triangles:

```
P = a1P1 + a2P2 + a3P3
```

In this formula, P1, P2 and P3 are the coordinates of the vertices of the triangle, and a1, a2, a3 are the barycentric coordinates of a point in the triangle. Again, the sum of a1, a2 and a3 must be 1. You could visualize it like this: a1, a2 and a3 balance P between the vertices of the triangle.

What makes barycentric coordinates so interesting for ray/triangle intersections is the fact that the barycentric coordinates must all be greater or equal than zero, and summed they may not exceed 1. In all other cases, it's clear that P is not inside the triangle. So, if we know the barycentric coordinates of the hit point, we can accept or reject the ray with very simple tests: Point P is outside the triangle if one of the barycentric coordinates a is smaller than zero, or the summed coordinates are greater than 1.

Calculating the barycentric coordinates can be done like this: First, we start with the previous equation:

```
P = a1P1 + a2P2 + a3P3
```

Since the sum of the coordinates must be one, we can substitute a1 with 1 – a2 - a3. A bit of rearranging gives us the following equation:

```
a2 (P2 – P1) + a3(P3 – P1) = P – P1
```

This can be solved in 3D, but it's cheaper in 2D. Therefore we project the triangle on a 2D plane. Convenient planes for this are the XY, YZ or XZ planes, obviously. To pick the best one of these, we take a look at the normal of the triangle: By using the dominant axis of this vector we get the projection with the greatest projected area, and thus the best precision for the rest of the calculations.

In 2D a2 and a3 can be calculated as follows:

```
a2 = (bx hy – by hx) / (bx cy – by cx)
a3 = (hx cy – hy cx) / (bx cy – by cx)
```

where b = (P3 – P1) and c = (P2 – P1), bx and by are the components of the first projected triangle edge and cx and cy are the components of the second projected triangle edge.

Using the above formulas to calculate the barycentric coordinates for the projected intersection point P the intersection code is now rather simple. In pseudo code:

```
vector b = vertex3 – vertex1
vector c = vertex2 – vertex1
vector Normal = cross( c, b )
distance = -dot( O – vertex1, N ) / dot( D, N )
// distance test
if (distance < epsilon) || (distance > max_distance) return
MISS
// determine dominant axis
if (abs( N.x ) > abs( N.y ))
  if (abs( N.x ) > abs( N.z )) axis = X; else axis = Z
else
  if (abs( N.y ) > abs( N.z )) axis = Y; else axis = Z
// determine project plane axii
Uaxis = (axis + 1) % 3
Vaxis = (axis + 2) % 3;
// calculate hitpoint
Pu = Ou + distance * Du
Pv = Ov + distance * Dv
// check intersection
a2 = (bu * Pv – bv * Pu) / (bu * cv – bv * cu)
if (a2 < 0) return MISS
a3 = (cv * Pu – cu * Pv) / (bu * cv – bv * cu)
if (a3 < 0) return MISS
if (a2 + a3 > 1) return MISS
// ray intersects triangle
return HIT
```

This approach was suggested by Ingo Wald, in his PhD paper on real-time ray tracing (link at the end of this tutorial). While this method is already pretty fast, it can be done much better. Wald suggests precalculating some data to improve the speed of the algorithm. We can precalculate the dominant axis for each triangle, plus the vectors for the projected triangle edges. Laid out in a way that is perfect for the intersection code:

```
nu = Nu / Naxis
nv = Nv / Naxis
nd = dot( N, vertex1 ) / Naxis
// first line equation
bnu = bu / (bu * cv – bv * cu)
bnv = -bv / (bu * cv – bv * cu)
// second line equation
cnu = cv / (bu * cv – bv * cu)
cnv = -cu / (bu * cv – bv * cu)
```

Now the actual intersection code can be simplified like this:

```
// determine U and V axii
Uaxis = (axis + 1) % 3
Vaxis = (axis + 2) % 3
// calculate distance to triangle plane
```

```
d = 1.0f / (Daxis + nu * DUaxis + nv * DVaxis
distance = (nd - Oaxis - nu * OUaxis - nv * OVaxis) * d
if (!(0 < t < maxdistance)) return MISS
// calculate hit point
Pu = OUaxis + t * DUaxis - AUaxis
Pv = OVAxis + t * DVAxis - AUAxis
a2 = Pv * bnu + Pu * bnv
if (a2 < 0) return MISS
a3 = Pu * cnu + Pv * cnv
if (a3 < 0) return MISS
if ((a2 + a3) > 1) return MISS
return HIT
```

For a detailed derivation I would like to point you to Wald's thesis. The actual implementation of this code for the sample ray tracer is in scene.cpp (method Primitive::Intersect). My implementation is slightly different than Wald's; the method I used requires fewer precalculated values (so it takes less storage), but it does use vertex A to calculate the intersection point. Which method is better depends on whether you add the precalculated values to the primitive class or not; Wald stores these values in a 'TriAccel' structure that contains all the data needed to test for intersections.

Note that the modulo operations at the beginning of this code are quite expensive; you can use a look-up table instead to speed these up.

Normals and Textures

Obviously we are interested in slightly more than just the fact that the ray intersects the triangle. We probably would like to interpolate vertex normals to make the mesh look smooth, and to texture the object, we need to interpolate U and V over the triangle.

Luckily, this is quite simple. Using the barycentric coordinates we can quickly interpolate values over the triangle. Calculating the normal for a given beta and gamma can thus be done as follows:

```
vector3 N1 = m_Vertex[0]->GetNormal();
vector3 N2 = m_Vertex[1]->GetNormal();
vector3 N3 = m_Vertex[2]->GetNormal();
vector3 N = N1 + m_U * (N2 - N1) + m_V * (N3 - N1);
NORMALIZE( N );
return N;
```

This code is from method Primitive::GetNormal. It's a bit sad that the interpolated normal requires normalization; this is a significant extra cost.

Calculating U and V is similar; check out Primitive::GetColor for the actual implementation.

Calculating Vertex Normals

To interpolate vertex normals, you need to have those available, obviously. I used an extremely simple 3ds loader that does not import things like smooth groups and stuff like that (I didn't want to turn this project into a 3ds file loading project), so I needed some code to quickly generate these normals.

A vertex normal is calculated by averaging the directions of the normals of all faces that the vertex belongs to. A na&#65533;ve implementation can be really slow: For each vertex, you

scan all faces to see if the vertex is used in that face, and if so, you add the face normal to a vector that accumulates these normals. Once you have checked all the faces, you normalize the accumulated vector to get the final vertex normal.

A programmer that worked on 3d studio once sent me an e-mail, suggesting a faster method:

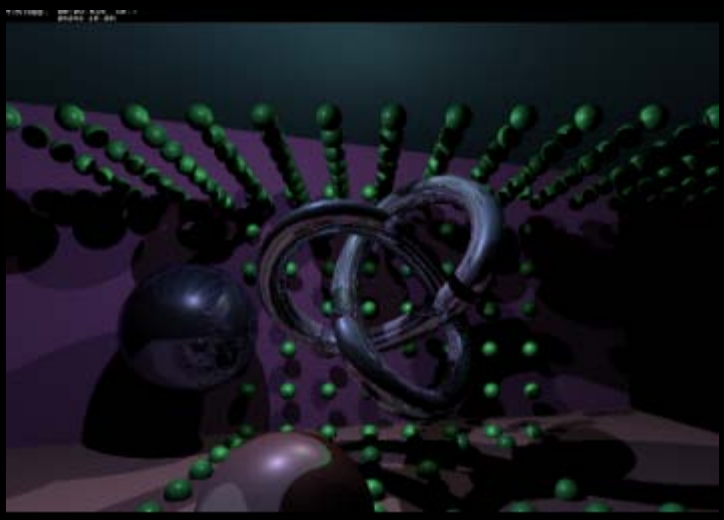
While reading the mesh, make a list of faces that each vertex uses. This can be done without slowing down the loading: Each time a vertex is used to create a face, this face is added to a list of faces for that vertex. Once the object is loaded, you have for each vertex a list of faces that use that particular vertex. Calculating the vertex normal is now a matter of summing the normals of these faces and normalizing. This gives you the vertex normals virtually for free. The actual implementation is 'interesting' (as in 'challenging'): It requires an array of pointers to primitives per vertex, so the array declaration looks as follows:

```
Primitive*** vertface = new Primitive**[m_NrVerts];
```

If you ever used the even more complex '****' in a real-world application, I would like to hear from you. :)

You can check out the tiny 3ds loader and vertex normal calculation code in scene.cpp.

Here is a screenshot of an earlier scene, combined with a triangle mesh:



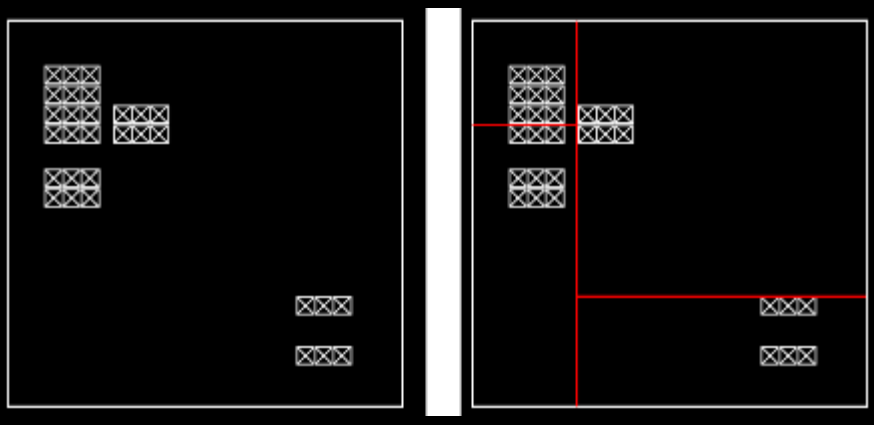
Kd-trees

The triangle intersection code that I explained is fast. Very fast, in fact: The version with the precomputed data is twice as fast as the M&#65533;ller-Trumbore code that everyone seems to be using. I think my ray-sphere intersection code is still faster, but spheres are not very useful for generic scenes.

So now we are ready for the next step: Implementing a better spatial subdivision scheme, so that we find the right triangle as quickly as possible. Vlastimil Havran did an extensive study of available spatial subdivision schemes (including regular grids, nested grids, octrees and kd-trees) and concludes in his thesis that the kd-tree beats the other schemes in most cases. And Ingo Wald claims that it is possible to limit the number of ray-triangle intersections to three or less using a well-build kd-tree.

Building a Kd-tree

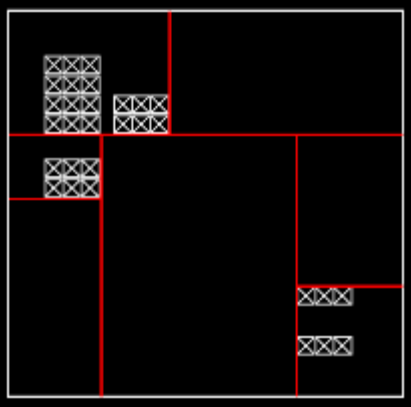
A kd-tree is an axis-aligned BSP tree. Space is partitioned by splitting it in two halves, and the two halves are processed recursively until no half-space contains more than a preset number of primitives. While kd-trees may look like octrees at first, they actually are quite different: An octree always splits space in eight equal blocks, while the kd-tree splits space in two halves at a time. The most important difference though is that the position of the splitting plane is not fixed. In fact, positioning it well is what differentiates a good kd-tree from a bad one. Consider the following images:



The left image shows a scene with some objects in it. In the right image, this scene is subdivided; the first split is the vertical line, then both sides are split again by the horizontal lines. The split plane position is chosen in such a way that the number of polygons on both sides of the plane is roughly the same.

While this may sound like a good idea at first, it actually isn't: Imagine a ray traversing this scene; it will never pass through an empty voxel. If you keep adding planes to this tree using the same rule, you'll end up with a tree that does not contain a single empty node. This is pretty much the worst possible situation, as the raytracer has to check all primitives in each node it travels through.

Now consider the following subdivision:



This time, a different rule (or 'heuristic') has been used to determine the position of the split plane. The algorithm tries to isolate geometry from empty space, so that rays can travel freely without having to do expensive intersection tests.

A good heuristic is the SAH [illegible] or surface area heuristic. The basic idea is this: The chance that a ray hits a voxel is related to it's surface area. The area of a voxel is:

```
area = 2 * width * height * depth
```

The cost of traversing a voxel can be approximated by the following formula:

```
cost = Ctraversal + area * prims * Cintersect
```

Since splitting a voxel always produces two new voxels, the cost of a split at a particular position can be calculated by summing the cost of both sides.

The split plane position that is the least expensive is then used to split the voxel in two new voxels, and the process is repeated for these voxels, until a termination criterion is met: The process can for example be halted when there are three or less primitives in the voxel, or when the recursion depth is 20 (these are values that I use right now).

An interesting extra termination criterion follows from the cost function: If the cost of splitting the voxel is higher than leaving the voxel as it is, it is also a good idea to stop splitting.

Implementation

OK, that was a lot of theory. But it's worth the trouble: Moving from a regular grid to a na&#65533;ve kd-tree implementation almost doubled the speed. Moving from a simple heuristic to the SAH doubled the speed again.

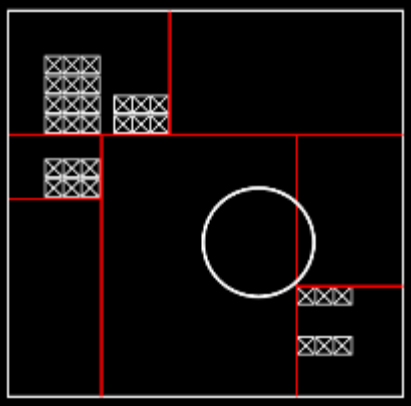
So let's have a look at the actual implementation of the kd-tree construction algorithm. Building a kd-tree generally looks like this, in pseudo-code:

```
void buildkdtree( Node* node )
{
        if (stopcriterionmet()) return
        splitpos = findoptimalsplitposition()
        leftnode = new Node()
        rightnode = new Node()
        for (all primitives in node)
        {
                if (node->intersectleftnode()) leftnode-
>addprimitive( primitive )
                if (node->intersectrightnode()) rightnode-
>addprimitive( primitive )
        }
        buildkdtree( leftnode )
        buildkdtree( rightnode )
}
```

Building a tree is really that simple. The first version of the kd-tree compiler for the sample raytracer was a mere 40 lines of code. There are some tricky parts though: Finding the split plane position using SAH, and checking intersection of a primitive and an axis aligned box.

Box-primitive Intersections

If you go back to one of the previous versions of the sample ray tracer, you can see that box-sphere intersections are handled by checking the box against the bounding box of the sphere. This is not optimal. Consider the following situation:



Does this box intersect the lower-right voxel? Using the old test it does. And that's not good; every ray that visits the lower-right voxel will check intersection with the sphere. I found a better method on a site with BlitzBasic code snippets:

```
bool Primitive::IntersectSphereBox( vector3& a_Centre, aabb&
a_Box )
{
	float dmin = 0;
	vector3 spos = a_Centre;
	vector3 bpos = a_Box.GetPos();
	vector3 bsize = a_Box.GetSize();
	for ( int i = 0; i < 3; i++ )
	{
		if (spos.cell[i] < bpos.cell[i])
		{
			dmin = dmin + (spos.cell[i] -
bpos.cell[i]) * (spos.cell[i] - bpos.cell[i]);
		}
		else if (spos.cell[i] > (bpos.cell[i] +
bsize.cell[i]))
		{
			dmin = dmin + (spos.cell[i] -
(bpos.cell[i] + bsize.cell[i])) * (spos.cell[i] -
(bpos.cell[i] + bsize.cell[i]));
		}
	}
	return (dmin <= m_SqRadius);
}
```

I'm not terribly in the exact details of this code, so if you don't mind I'm just going to accept that it works.

Initially I used a similar simplification for triangles. You can check the new box-triangle intersection in scene.cpp, I used the version made by Tomas Akenine-M&#65533;ller (the same guy that worked with Trumbore on ray/triangle intersections), as it's fast and short. The effect on the performance of the ray tracer is dramatic. Without the improved box/primitive intersection code, each ray was checked against 22 primitives on average. With the new code, this figure went down to 6.7. VTune made the impact even clearer: Without proper box/primitive intersection code, the intersection tests took 80% of the clock cycles (compared to 15% for the traversal code). With the new code, the same amount of time is

spent inside the intersection code as in the traversal code.

The impact on rendering speed is dramatic, as expected: The scene from article 4 now renders in less than a second when rendering to a 512x384 canvas.

Implementing SAH

Let's go back to the formula for the cost of a single voxel using the surface area heuristic:

```
cost = Ctraversal + area * prims * Cintersect
```

This is the cost for a single voxel. The cost of a voxel after splitting it is the sum of the cost of the two new voxels:

```
cost = costleft + costright
```

To find the optimal split position, we have to check them all. Luckily, the number of interesting split plane positions is limited: A split plane should always be situated at the boundary of a primitive. All other positions don't make sense, since the number of primitives won't change between two primitive boundaries. Sadly, even with this restriction, the number of candidate split plane positions is daunting. Imagine a scene containing 8000 triangles. For the first split, we have to consider roughly 16000 positions. For every position, we need to count the number of primitives that would be stored in the left and right voxel.

In pseudo code:

```
bestpos = -1
bestcost = 1000000
for (each primitive in node)
{
        left_extreme = primitive->getleftextreme( axis )
        right_extreme = primitive->getrightextreme( axis )
        if (cost = calculatecost( left_extreme ) < bestcost)
        bestcost = cost, bestpos = left_extreme
if (cost = calculatecost( right_extreme ) < bestcost)
        bestcost = cost, bestpos = right_extreme
}

float calculatecost( splitpos )
{
        leftarea = calculateleftarea( splitpos )
        rightarea = calculaterightarea( splitpos )
        leftcount = calculateleftprimitivecount( splitpos )
        rightcount = calculaterightprimitivecount( splitpos )
        return costtrav + costintersect * (leftarea *
leftcount + rightarea * rightcount)
}
```

You can check out the actual implementation in the sample ray tracer project. I tried some things to speed up the code: The candidate split plane positions are first stored in a linked list. During this first loop, duplicate positions are eliminated, and the positions are sorted.

In a second loop, the left and right voxel primitive counts are calculated for each split

plane position. This is where the sorted list comes in handy; as we are walking from left to right, we can tag primitives that are to the left of the current plane, so we can skip them in the next iteration.

Even using these tricks the kd-tree compiler is too slow. If someone has a good suggestion to improve this, I would like to hear it. :)

The generated tree is quite good however. It can be improved, consider this work in progress: Right now, triangles are not clipped to the parent voxel before their extends are determined. Doing this clipping allows split planes to be situated at the exact position where a triangle leaves a node. This is not a very difficult extension, but it will make the tree compiler even slower.

Traversing the Kd-tree

To traverse the kd-tree, I implemented Havran's TA^{B}_{rec} algorithm (Havran tried a lot of spatial subdivisions and traversal algorithms; TA stands for 'traversal algorithm', rec means recursive and B is the version His code is a modification of an existing algorithm A). The code is not overly complex, but instead of copying a huge amount of text from his thesis I rather point you to the original document.

Ingo Wald apparently found an even more efficient way to traverse a kd-tree, but in my experience, his implementation suffers badly from numerical stability problems. His implementation notes are quite sketchy (which is why I didn't feel so guilty about heavily quoting his work) and he only mentions these problems, without the solutions. Wald on the other hand designed his approach specifically to cope with stability issues.

The actual code is much smaller than the regular grid traversal code, which is a nice bonus. :)

Finetuning

Once the ray tracer is in good shape algorithm-wise, it's time to revert to old-skool hardcore optimizing.

Check out the class declaration of the KdTreeNode (I know, it's code, but it's interesting):

```
class KdTreeNode
{
public:
        KdTreeNode() : m_Data( 6 ) {};
        void SetAxis( int a_Axis ) { m_Data = (m_Data &
0xfffffffc) + a_Axis; }
        int GetAxis() { return m_Data & 3; }
        void SetSplitPos( real a_Pos ) { m_Split = a_Pos; }
        real GetSplitPos() { return m_Split; }
        void SetLeft( KdTreeNode* a_Left ) { m_Data =
(unsigned long)a_Left + (m_Data & 7); }
        KdTreeNode* GetLeft() { return
(KdTreeNode*)(m_Data&0xfffffff8); }
        KdTreeNode* GetRight() { return
((KdTreeNode*)(m_Data&0xfffffff8)) + 1; }
        void Add( Primitive* a_Prim );
        bool IsLeaf() { return ((m_Data & 4) > 0); }
        void SetLeaf( bool a_Leaf ) { m_Data = (a_Leaf)?
(m_Data|4):(m_Data&0xfffffffb); }
        ObjectList* GetList() { return
(ObjectList*)(m_Data&0xfffffff8); }
```

```
        void SetList( ObjectList* a_List ) { m_Data =
(unsigned long)a_List + (m_Data & 7); }
private:
        // int m_Flags;
        real m_Split;
        unsigned long m_Data;
};
```

I bet this looks extremely obscure. :) There is a reason for this though: To optimize cache behavior, I tried to reduce the amount of space that a single KdTreeNode uses to 8 bytes. That means that 4 KdTreeNodes fit in a single cache line. The data for a KdTreeNode is laid out in a peculiar manner: First, there is the split plane position, a float that uses 4 bytes. The other data member is 'unsigned long m_Data'. This field actually contains three values: A pointer to the left branch, a bit indicating whether this node is a leaf or not, and two bits that determine the axis for the split plane. A pointer to the list of primitives in a leaf node is also available: Since a leaf node does not need a pointer to it's left and right kids, this space can be used for a pointer to the primitive list.

Since the object takes 8 bytes, it's good for the cache to align the address for each node to an 8 byte boundary. That also means that the lower three bits of every KdTreeNode pointer are zeroes. These bits can thus be used for other purposes, but only if addresses are properly masked when they are needed. That is the reason for the bitmasks in many of the methods: These bitmasks simply filter out the requested data.

The multiple-of-8 addresses for KdTreeNodes are enforced by a custom memory manager. This manager allocates a large chunk of memory, and returns new KdTreeNodes upon request.

I already mentioned that the m_Data field also contains one implicit value: A pointer to the right branch. KdTreeNode allocations are always done in pairs: This guarantees that the address of the right branch KdTreeNode is always 4 bytes more than the left branch KdTreeNode. Now all required data fits in 2 dwords.

Further Improving Cache Performance

One other small thing that I added to improve cache behavior is tiled rendering: Instead of drawing entire scanlines at a time, the new sample ray tracer renders tiles of 32x32 pixels. You may think that this can't possibly have a big impact on the performance of the ray tracer, but the opposite is true: The impact on rendering time is no less than 10%.

This can be explained: When sending rays in bundles, they are much more likely to use the same data. Many rays that share a tile will also hit the same voxels and possibly also the same primitives. I have to admit that the impact on performance did surprise me.

I tried various configurations, but 32x32 tiles seem to give the best performance.

Future Work

So much to do, so little time. And I just bought Half Life 2, so that's not going to help either. :)

Future work can be divided in two main branches:

First, the ray tracer can be made much faster. Don't get me wrong, the current ray tracer is quite fast, and does not use any tricks that limit it's usability. It's a solid basis for future development. However, Wald is reporting speeds of several frames per second for scenes

consisting of thousands of polygons at a resolution of 1024x768 pixels, on a single 2.5Ghz laptop. That would mean that I am at only 20% of his speed. A coders nightmare... The biggest gain could be obtained by tracing ray packets of 4 rays simultaneously; this makes the algorithm exceptionally suitable for SSE optimizations. Even without SSE code this approach is interesting: Since all rays in the packet use the same data, the amount of data that is used by the ray tracer is reduced considerably. Wald reports speedups of 200-300% using this approach.

Second, there's functionality. The current ray tracer really is a bare bones program. I would really like to work on global illumination (like I promised), and on materials (bump maps, BRDF's), or on real-time ray tracing.

Right now the plan is to let the performance issue rest for a while. After I complete Half Life 2 I would like to integrate the photon mapping code that I did earlier in the current ray tracer. That's going to take some time, as my knowledge of ray tracing evolved considerably. It was quite a ride. :)

Last Words

That's it for the moment. At the end of this article you will find a download link to raytracer7.zip, Wald's thesis and Havran's thesis. Both documents are very good reads, so if you didn't check them earlier, do so now.

Thanks for your attention, and see you soon.

Jacco Bikker, a.k.a. "The Phantom"

Links:
"Heuristic Ray Shooting Algorithms", Vlastimil Havran:
http://www.cgg.cvut.cz/~havran/phdthesis.html

"Real Time Ray Tracing and Interactive Global Illumination", Ingo Wald:
http://www.mpi-sb.mpg.de/~wald/PhD/

And the latest VC6 project: raytracer7.zip

Article Series: