Hardware accelerated ray tracing through OpenGL

Max Reitz*

Abstract

This paper deals with using standard OpenGL graphics cards (2.0 and up with FBO extension) for hardware acceleration of ray tracing. For this it is necessary to transform the well-known recursive algorithms into iterative ones with comparably low execution time complexity.

The result is an algorithm capable of taking much advantage of today's standard rasterizing graphics hardware.

Note that any optimization of the standard ray tracing algorithm (e.g., BSPs) is not part of this paper – it uses the standard algorithm with linear execution time complexity, both regarding objects and light sources.

Introduction

Ray tracing has been used for long for high-quality 3D image rendering. Compared to the plain rasterization it is much easier parallelizable, however, it also requires heavy parallelization in order to run fast because it operates independently on every single pixel. Rasterization however may be executed comparably fast also on a single execution unit. This was maybe one of the reasons why early real-time 3D applications (e.g., games) used rasterization instead of ray tracing. Soon, acceleration hardware was created, featuring rasterization instead of other 3D image rendering techniques.

However, rasterization lacks several features ray tracing includes very easily, such as shadows or reflections and refractions at an arbitrary surface normal. Several techniques have been created in order to circumvent those limitations, though many are not very intuitive and the others are not very good. Those techniques were soon implemented using parallelized shader units on graphics hardware, because they often worked – just like ray tracing – on a per-pixel basis.

Ray tracing remained the standard rendering method for high-quality image generation (especially when used in conjunction with global illumination, which may however be implemented in rasterization, too). In the meantime, rasterizing became more and more complex while implementing features which ray tracing got all the time. As said before, those resulted in a heavy use of parallel units on graphics hardware, which were soon used for other tasks (GPGPU). Computation-intense parts of all different kinds of programs are since executed via OpenCL on GPUs.

There are also several attempts of using GPGPU for hardware-accelerated ray tracing (which are closely related to the approach presented in this paper). Others tried to accelerate the ray tracing algorithms itself by optimizing special use cases (e.g., only triangles as primitives) and thus using modern general-purpose CPUs for real-time rendering. Finally, there were those approaches creating specifically-designed hardware for acceleration, none of which actually succeeded.

I don't think that specific ray tracing hardware will replace standard graphics cards in the consumer area anytime soon. On the other hand, it is obvious that CPUs are not very suitable for ray tracing. They are designed for all types of computations, especially single-threaded ones. Ray tracing, in contrast, does not need most of the capabilities a CPU offers and is nearly arbitrarily parallelizable, directly suggesting GPGPU usage instead.

In conclusion we have to find a way to execute ray tracing on modern graphics hardware whose purpose is strictly speaking rasterizing, but which offer enough parallel computational power to also benefit ray tracing.

General implementation

As suggested by this paper's title, it will not deal with an OpenCL implementation, but with an OpenGL one (in practice, there is not big of a difference, anyway). There are (at least) two methods of involving a GPU in the rendering process:

1. GPU-assisted: Pass many simple calculations such as matrix multiplications to the GPU, retrieve the output and use the results in mostly

*Faculty of Computer Science, Dresden University of Technology, email: xanclic@googlemail.com

on-CPU ray tracing.

2. Fully GPU-based: Pass all input data (objects, object transformations, etc.) to GPU programs and retrieve the fully-rendered output image.

The first approach may seem simple at the first glance, however, it is not. One has to gather all required calculations (which have to be of the same kind), put them into a texture, read the results back and distribute them – such a gathering and distribution system seems rather complicated and will probably be even slower than doing everything right on the CPU. Thus, the only reasonable approach is the second one.

The approach taken here is synchronous multi-threaded. It is obviously multi-threaded, and it is synchronous, because it does one calculation for every pixel and stops after that calculation has been carried out on all of them. A drawback is that this is not as fast as maybe possible, but it also offers one great advantage when it comes to shadow calculation.

The desired result is now a fully rendered image which does not need any post-processing and may be displayed directly on screen. Hence, every calculation must be executed per pixel and will be executed on every single one. Every step will thus be carried out via frame buffer objects on textures.

Calculation framework

At this point, it is obvious that a framework is required to carry out calculations on textures, allowing an arbitrary texture input and output count. The texture count allowed by hardware as input is thus equivalent to the number of texture image units (samplers per fragment shader), the number of output textures is limited by the minimum of the number of draw buffers and the number of color buffer attachments per frame buffer object.

Calculations will be carried out by fragment shaders. Textures are used as full-size input, the output is written 1-to-1 to other textures. The vertex transformation is hence really simple: It ignores every Z component and just pipes the input vertex coordinates through so a quad drawn from $(-1,-1)$ to $(1,1)$ will span the whole output. It will furthermore assign a `uniform vec2` so that it will reach from $(0,0)$ to $(1,1)$ (UV coordinates of the input textures). Other input objects are matrices, vectors and scalars (booleans, floating point numbers, etc.), output is additionally allowed to a combined stencil/depth buffer.

The calculation basically consists of GLSL code, which is split into common (or shared) code, which must be executed for every output object, and specific code describing what to write into which output buffer. For this reason it is possible to use more output buffers than physically allowed. If using e.g. 13 output buffers with 8 buffers physically allowed per FBO, this framework should create two FBOs with slightly different shader programs, one writing to the first 8 buffers, the other one accessing the last 5.

The `uniform` mentioned before is used to index the input textures in the fragment shader.

Since the ray tracing itself uses textures, too, the framework should also allow arbitrarily dimensioned textures which may be exchanged at runtime.

Such a framework with all of the described properties has been implemented under the name MACS[1]. It is built around a so-called render pass object, which is given the code and all input and output objects at initialization. The input object values may however be changed at runtime. This design has been chosen to expose the state-machine-like interface of OpenGL, thus improving efficiency.

Ray tracing

Basic input

The primary input objects are ray maps, one defining the ray starting points and one defining their directions. On the initial ray tracer call they will be initialized to a trivial map representing the rays appropriate for a viewer with a certain FOV and display aspect.

When doing recursive ray tracing, those initial maps will be replaced by ones generated from the ray tracer itself.

Ray tracing

The first step of ray tracing is to do ray tracing. This is generally implemented by finding the nearest intersection point for every pixel and object available and then returning the object with the minimal intersection distance, i.e.: Do for every object: Find nearest intersecting object.

This is not reasonably implementable in the given iteration order on hardware, as this would require a list of objects to be present there (which is not impossible, but hard to implement). However,

[1] Massive Arbitrary Computations by Shaders

this can be easily fixed by exchanging the inner and outer itation, i.e.: Do for every object: Find intersection for every pixel and store if nearer than already stored value. There actually is an OpenGL feature which allows exactly that behavior (store if nearer), called depth testing. Its only drawback is that it requires us to bring every intersection distance into a fixed range, since depth values must be in $[0, 1]$.

Generally, a program is created for every object class which finds the nearest intersection point for a certain ray. It also returns several surface attributes such as the surface normal for that point. Object instances are created by transforming such a prototype object (e.g. a sphere around the point of origin with radius 1) by an arbitrary transformation matrix. To transform a ray into the coordinates for the original prototype object, they are basically multiplied by the inverse of the transformation matrix.

This program writes its result to different output buffers. The intersection distance is divided by a fixed value which is expected to be the the maximum distance, and then written to the depth buffer. The automatic depth testing will assure that only the object with the minimal intersection distance will be stored in the output buffers. All relevant data will be written to the other buffers: global coordinates of the intersection point, surface normal, surface tangent and material properties (color, refractive index, etc.).

Each material property may be textured. This is done by another user-defined function which returns a UV coordinate for a given intersection point of the prototype object. This coordinate is then used to index the texture and set the material attribute value for the respective pixel.

Also, each material property may be freely assigned by a dynamic function (which is what shaders are used for in rasterization). This is really trivial since this is basically equal to texturing, but where the latter uses `texture2D(texture, uv_coordinate)`, the completely user-defined function would be called. Note that this has not been implemented in the example implementation because of optimizing problems (such a user-defined function would require a new object prototype, therefore requiring one object class per object instance). There is probably no solution to this problem, however, since arbitrary shading will presumably not be used too often, there is none needed whatsoever. Because of those circumstances, such dynamic functions have not been implemented yet.

In the end, the output buffers contain all surface intersection properties for every pixel there is.

Shadowing

Before applying the lighting, it is necessary to find all shadowed points. In the standard ray tracing model, this is achieved by testing for every pixel and every light source, whether the line in between intersects another object. However, if one were to implement this directly here, it would probably result in quadratic complexity regarding the object count ($O(n_l \cdot {n_o}^2)$, where n_o is the number of objects and n_l is the number of light sources), since one would have to check for every object were it is shadowed for every light source over all other objects. This is bad when compared to standard shadowing techniques as used for rasterization, which create all shadows of a light source by checking every object exactly once ($O(n_l \cdot n_o)$). Every method based off the mentioned one will thus share this disadvantage.

However, in the approach taken here we use the fact that we have the full image available, i.e., every global intersection point is known at this step (in contrast to the standard single-threaded or asynchronous multi-threaded approach). The method used in here resembles one commonly known as "shadow mapping" for rasterization. This one renders the scene as seen from the light source and uses the resulting depth buffer for determining whether a given surface is shadowed. Despite being relatively easy to implement and also reasonably fast, it has a major drawback: Since it renders the scene completely independent from the pixels seen by the viewer, the resulting shadows are not very exact. This can be compensated by using higher resolution textures, which in turn require more video memory and execution time.

Those drawbacks do not apply here. By using ray tracing, one can easily and exactly control which pixels shall be tested for shadowing. The output texture must only be as big as the final output and the resulting shadow map will be perfect. Other than that, the approach is exactly the same.

For every light source such a shadow map is to be created. For every pixel in the output, a ray (or more specifically: a line) is tested, reaching from the light source to the global intersection point found there. If that line intersects with anything but the object at its end, the point is considered shadowed. In the end, there will be a shadow map for every light source, which will be used during shading to test whether a given point is affected by a given light source.

In conclusion, the complexity of this approach is

linear again in regard to the object count ($O(n_l \cdot n_o)$).

Lighting

Lighting designates the process of applying incoming light to a surface point. There is not much to be said about this: A light model takes the position of the light source, the position of the lit point, the surface normal and tangent, the material property and several light properties such as an attenuation function as input. If the given point is affected by the light source (which can be determined by looking at the generated shadow map), a certain function returns the light intensity reflected by the point to the viewer for every color channel (often, all color channels are calculated simultanously).

The model used in the example implementation is the Schlick model [1], which can however be more or less easily replaced by other models such as the Phong model or different ones.

The results of all lighting operations are joined using additive blending.

Recursive ray tracing

Recursive ray tracing enables refraction and reflection. For both phenomenons two new maps have to be generated (ray starting points and directions) which are fed to a recursive call of the hardware ray tracer. The problem to find those recursive rays is a general ray tracing problem and will thus not be discussed further in here.

There is however an interesting problem: Finding the terminating condition. Normally, one would simply trace until no recursion is necessary anymore or a certain number of recursive calls have been made. Basically this would mean one would have to do recursive tracing until no pixel needs any recursion anymore (or there are too less). Generally, occlusion queries are ideal for this use case, they did, however, not work for me.

The other approach would then be to carry out a fixed number of recursion levels; this could be optimized by creating stencils for taking advantage of early-out.

Implementation

The ray tracing algorithm described here is implemented in the Betelgeuse library.

Results

Of course, the standard ray tracing approach cannot be directly implemented when transforming it into a synchronous multi-threaded model without inherent recursion. Generally, every operation is now executed on all pixels, which means, one has to generalize data so it can be represented by floating point vectors between iterative steps and that all pixels are always treated exactly the same way (which means one cannot store any special data, such as the intersected object type).

It is however in fact not too difficult to do this transformation. Every control structure which is not directly possible in a fragment shader such as iteration over a variable list of objects or recursion must simply be taken out of the shader to the next higher level, which is the program on the CPU calling the shader programs on the GPU.

It was thus possible to create a general ray tracing algorithm with a complexity of $O((n_p \cdot (n_o + n_l \cdot n_o + n_l))^{n_r}) = O((n_l \cdot n_o)^{n_r})$, where

- n_p is the number of pixels
- n_o is the number of objects
- n_l is the number of lights
- n_r is the number of recursions (plus one for the initial call)

This is exactly the same complexity as for the general ray tracing algorithm. Hence we were able to create an equivalent representation of that algorithm for usage on standard OpenGL 2.0 compatible graphics hardware, which was exactly the goal set in the first place.

Note that there are several techniques for bringing the time complexity of ray tracing down to logarithmic complexity (e.g., BSPs), but they are not part of the plain standard algorithm and thus not considered here.

Early-out

Many of the shaders are in fact quite complex programs, including conditional operations. These are not natively supported in early shader models, therefore heavily increasing execution time (especially since the `discard` keyword is not interpreted as intended).

Note however, that the number of GPUs supporting shading but no early-out is comparably small (and those are obviously quite old).

Example implementation

As mentioned in the respective sections, MACS and Betelgeuse together are a sample implementation of the methods mentioned here. MACS implements an interface for carrying out parallel computions on graphics hardware, while Betelgeuse uses that interface for implementing the ray tracing algorithm described here.

MACS/Betelgeuse proves that the approach taken here is actually applicable and yields the desired results (strong acceleration through OpenGL 2.0 compatible graphics hardware).

Comparison

Comparison between software and GPU ray tracing was done with two programs:

- Software ray tracer: softray (no real name), very similar to Betelgeuse, does, however, not support textures. Able to run multi-threaded arbitrarily up to one thread per row (i.e., 512 resp. 1024 pixels). Scenes are loaded via Lua scripts.
- GPU ray tracer: MACS/Betelgeuse. Scenes are given via programs linked to those libraries.

If softray uses multithreading, it creates two threads per logical processor core online (whatever `sysconf(_SC_PROCESSORS_ONLN)` returns). All of the CPU tests here were done in this way.

The tested CPUs were the following:

- Intel Pentium D Presler 930 (3.0 GHz) – a lower mid-range dual-core NetBurst CPU for desktop computers in 2006 with 12 GFLOPS.
- Intel Pentium Dual-Core T2310 (1.46 GHz) – a relatively low-end Core CPU for notebooks in 2007 with 11.7 GFLOPS.
- Intel Core i5-3570K (3.4 GHz) – a higher mid-range Ivy Bridge CPU for desktop computers in 2012 with 108.8 GFLOPS.

The tested GPUs were the following:

- ATI Radeon X1600 – a mid-range graphics card for desktop computers in 2006 (OpenGL 2.1) with 73 GFLOPS.
- Intel GM965 (GMA X3100) – a standard integrated graphics chip for notebooks in 2007 (OpenGL 2.1) with probably between 10 and 30 GFLOPS.
- AMD Radeon HD 7850 – a high mid-range graphics card for desktop computers in 2012 (OpenGL 4.2) with 1761 GFLOPS.

One object, two lights

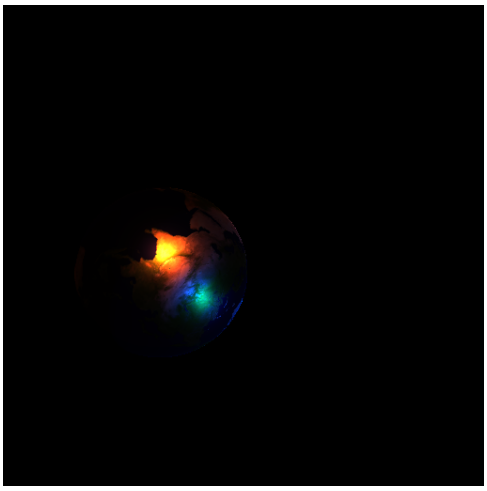

Figure 1: Sphere with an earth surface texture, being lit by two different point light sources

Resolution	CPU/GPU	Render time
512 × 512	Presler 930	300 ms
	T2310	220 ms
	GM965	83 ms
	3570K	35 ms
	X1600	19 ms
	HD 7850	1.19 ms
1024 × 1024	Presler 930	1.15 s
	T2310	720 ms
	GM965	290 ms
	3570K	125 ms
	X1600	79 ms
	HD 7850	4.4 ms

(Refer to Fig. 1 for the rendered image)

Two objects, two lights

Resolution	CPU/GPU	Render time
512 × 512	T2310	780 ms
	3570K	150 ms
	GM965	120 ms
	HD 7850	2.8 ms
1024 × 1024	T2310	2600 ms
	3570K	520 ms
	GM965	430 ms
	HD 7850	9.7 ms

(Refer to Fig. 2 for the rendered image)

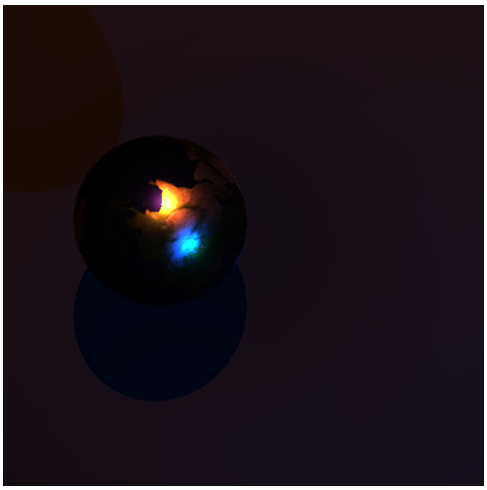

Figure 2: Sphere with an earth surface texture in front of a non-textured white plane, being lit by two different point light sources

Comparison conclusion

The first result is one that does not support the thesis that CPUs are always worse than GPUs for ray tracing, though it did not appear unexpectedly: The i5-3570K offers 108.8 GFLOPS of computing power, much more than the X1600 with 73 GFLOPS – the latter is however significantly faster in all test runs. This is probably because it is much harder to optimize correctly for SIMD on CPU than on GPU. The sole purpose of shader units on a GPU is to do SIMD calculations, therefore every fragment shader will be compiled and deployed to every execution unit and may thus use the full computational power. The CPU however is grained much finer: First, there are the cores and second, there are the SIMD units per core. Most programming languages are by design SISD. It is hence difficult for a compiler to find vectorizable code and transform it accordingly – deploying independent code to every core is easy, but using all SIMD units at full capacity is not.

The only practical solution would be to write specialized routines in assembly and using them just like shaders for the on-GPU calculations. This is, however, ineffective, because of three reasons:

1. Writing good assembly is hard.

2. One would lose the possibility for nice high-level optimizations (effective early-out, very dynamic recursion depth, etc.).

3. GPUs are better anyway (so why care?).

References

[1] Schlick, C. A customizable reflectance model for everyday rendering, 1993.