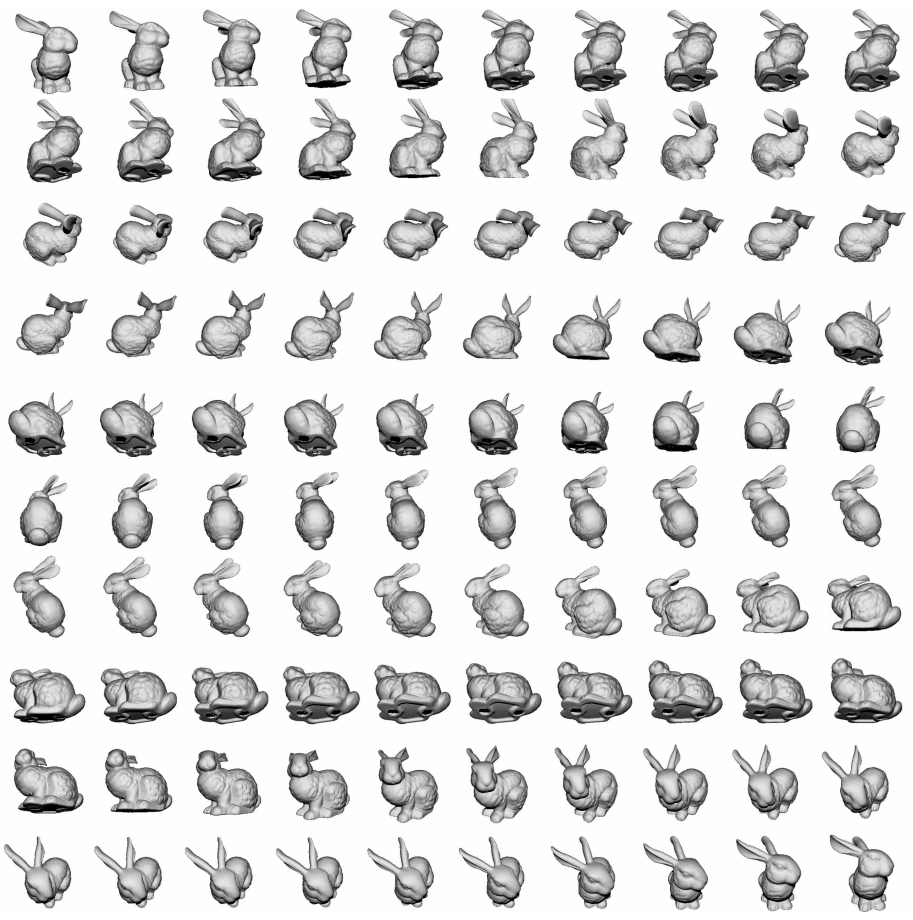
A Comparison of Acceleration Structures for GPU Assisted Ray Tracing



Master's thesis

Niels Thrane   Lars Ole Simonsen
thrane@daimi.au.dk   lo@daimi.au.dk

August 1, 2005

Advisor: Peter Ørbæk

Abstract

Recently, ray tracing on consumer level graphics hardware has been introduced. So far, most published studies on this topic use the uniform grid spatial subdivision structure for reducing the number of ray/triangle intersection tests. For many types of scenes, a hierarchical acceleration structure is more appropriate. We compare GPU based traversal of kd-trees and uniform grids with a novel bounding volume hierarchy traversal scheme. The three implementations are compared in terms of performance and usefulness on the GPU. We conclude that on the GPU, the bounding volume hierarchy traversal technique is up to 9 times faster than our implementations of uniform grid and kd-tree. Additionally, this technique proves the simplest to implement and the most memory efficient.

Danish summary

Raytracing er en velkendt teknik til frembringelse af foto-realistisk 3D grafik på computere. Whitted tildeles almindeligvis æren for den klassiske raytracer, som står beskrevet i hans artikel fra 1980 [Whitted]. Kort sagt virker ray tracing ved at man følger "lysstråler" (rays) rundt i scenen hvorved man kan beregne mængden af lys der rammer kameraet.

Korrekt simulering af lys involverer at man skal behandle millioner af rays. Da scenen ofte er sammensat af et antal trekanter i størrelsesordenen 100.000 - 1.000.000, er naiv test af hvorvidt hver ray rammer hver trekant for langsom i praksis.

Af denne grund arrangerer man trekanterne i en struktur, som tillader at man hurtigt kan se bort fra en stor mængde trekanter, som så ikke behøver at blive testet mod den givne ray. Sådanne strukturer kaldes accelerationsstrukturer og er et aktivt forskningsområde.

I 2004 viste Purcell hvordan en ray tracer kan implementeres på et almindeligt grafikkort. Det interessante ved at bruge grafikkortet er dels at der sidder kraftige graikkort i alle nyere hjemme-pc'er, dels at beregningskraften på grafikkort udvikler sig væsentligt hurtigere end på tidssvarende CPU'er.

Purcell anvendte accelerationsstrukturen uniform grid i sin raytracer. For en stor klasse af scener er hierarkiske strukturer – såsom kd-træer og bounding volume hierarkier (BVH) – et bedre valg. I denne afhandling beskriver og sammenligner vi tre forskellige accelerationsstrukturer i en raytracer, der kører udelukkende på grafikkortet. De tre strukturer er uniform grid, kd-træet og BVH.

Gennemløb af uniform grid og kd-træer er allerede beskrevet af henholdsvis Purcell og siden af Foley og Sugerman [Foley & Sugerman]. I denne afhandling præsenterer vi en ny måde at gennemløbe BVH'er, der viser sig at være specielt velegnet til udførelse på grafikkortet.

Vores målinger viser, at det nye BVH gennemløb giver op til 9 gange hurtigere gennemløb end tilfældet er med de to andre strukturer. På nogen scener er gevinsten mindre, men BVH-gennemløbet er konsekvent hurtigst. Vi tilskriver primært den lave gennemløbstid til simpliciteten af den kode der skal udføres på grafikkortet under træ-gennemløbet.

Contents

1 Introduction

In the entertainment industry today, there is an increasing demand for high quality computer generated imagery. A good example is the movie industry, where production costs are reduced significantly by generating large scale scenery on a computer instead of constructing the sets in real life.

In order to fool the human eye with artificial images, they must be as realistic as we can possible make them. This kind of realism in an image is usually referred to as photo realistic imagery.

Computer generation of photo realistic images is a well studied problem. The main problem is that of correctly simulating the interaction of light with life like materials. *Global illumination* denotes the family of algorithms used to solve this problem.

One of the most popular and simple ways of doing global illumination is known as *ray tracing*. Ray tracing involves tracing the path of light as it bounces around the scene. The main problem of ray tracing is that of determining which piece of the scene is hit by a given ray of light.

In order to render an image of sufficient quality, millions of rays must be traced into the scene. Scenes are generally represented as a large number of triangles. In the naive approach, each ray is tested for intersection with each triangle in the scene. Scenes can easily contain a hundred thousand triangles, leading to 100,000,000,000 ray/triangle intersection tests, which would take much too long in practice. For this reason, triangles are often arranged in such a way that one can quickly discard large portions of them, when searching for the nearest one intersected by a given ray. Grouping triangles together for the sake of faster ray intersection is known as an *acceleration structure*.

Acceleration structures serve to reduce the number of ray/triangle intersections tests and thus accelerate the image rendering process. Another way of increasing the speed of this expensive algorithm is to find faster and more suitable hardware configurations. Fast ray tracing has been accomplished with the use of pc clusters [OpenRT]. This solution is good news for movie makers and others who can afford to buy and maintain a cluster. It doesn't bring fast ray tracing to the end user, however.

Recently, work has been done to investigate the possibilities of the increasingly programmable graphics processing unit (GPU) with respect to ray tracing [Purcell]. The processing power and availability of the GPU make it an attractive alternative.

In 2000, Havran et al. carried out a study with the goal of identifying the fastest acceleration structure for ray tracing [Havran et al.]. The authors identified a small group of structures, each performing within a factor of two of the others. Which structure is faster depends – among other things – on the hardware running the ray tracer. For this reason, we find it interesting to investigate whether the same results hold for GPU implementations.

Purcell used the uniform grid structure in his GPU ray tracer. Since his focus was not on exploration of different acceleration structures, this choice was based mostly on ease of implementation.

The purpose of this thesis is three-fold:

- We provide a review of some of the best known acceleration structures: The uniform grid, kd-tree, and bounding volume hierarchy (BVH). Our aim is to determine if and how each of these acceleration structures allow efficient GPU traversal.

- We derive a new technique for efficient traversal of BVHs using elements of partial evaluation.

- Finally we do a comparative study including running implementations of all covered techniques and compare them with respect to performance and applicability on the GPU.

We also cover the basics within both ray tracing and general purpose computation on the GPU (GPGPU). In this way we aim to make this thesis readable to anyone with a background in general computer science.

We limit ourselves to static scenes, i.e. scenes where only the camera and light positions change. This means acceleration structures need only be constructed once. The construction algorithms do not parallellize as well as the traversal and so we build the structures on the CPU. For this reason the GPU applicability issue is only evaluated with respect to the traversal of the structures, not the construction.

We conclude that the kd-tree and the uniform grid have inherent drawbacks that make GPU traveral inefficient. Additionally they require tuning of several scene dependent parameters to achieve optimum performance. By comparison, BVHs provide simple construction in addition to simple and efficient traversal. Also there are no construction parameters to tune.

Our performance measurements reveal that BVH traversal is up to 9 times faster on average than the other structures. This is largely due to the BVH traversal technique introduced in this thesis.

The structure of this thesis is as follows: The first two sections supply background knowledge on GPGPU and ray tracing. We then show how ray tracing can be mapped to the GPU. After this we go through the acceleration structures one by one, demonstrating how they can each be traversed on the GPU. Finally, we present and discuss detailed performance measurements.

Acknowledgements

Thanks to Jeppe Brønsted, Jesper Mosegaard, Olivier Danvy, and of course our advisor, Peter Ørbæk for their useful suggestions. Thanks also to Tim Foley for the benchmark scenes and for fruitful discussions, to Tim Purcell for suggesting the topic of our thesis, and to the guys at www.gpgpu.org for their technical assistance.

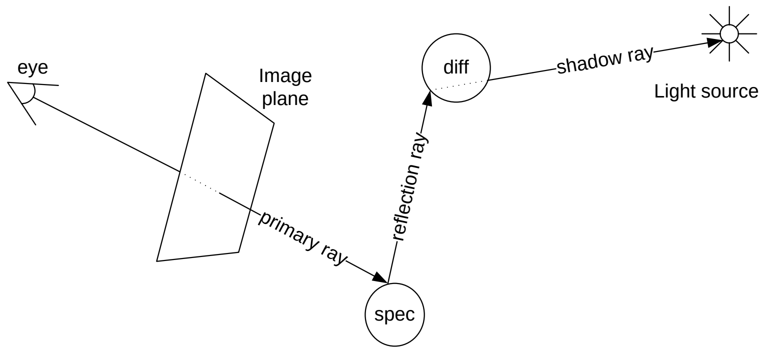


Figure 1: Basic ray tracing.

2 Ray tracing

In this section, we introduce ray tracing as a technique for generating photo-realistic synthetic images. We restrict ourselves to the aspects of ray tracing relevant to our experimental work on the GPU. As such, there is much more to ray tracing than what is covered here.

Ray tracing is classified as a global illumination rendering algorithm, which basically means that the appearance of an object is influenced by its surroundings (as opposed to local illumination algorithms). Global illumination is important for realistic images because it captures effects such as reflections, refractions, transparency, color bleeding, caustics and shadows.

The classic ray tracer was described by Whitted [Whitted] who could simulate reflection, refraction, hard shadows and diffuse surfaces.

2.1 Basic ray tracing

Generally speaking, global illumination algorithms generate an image by calculating the lighting of the scene and projecting it onto a plane in front of a virtual eye or camera. The projection plane can now be sampled to generate the final image.

In ray tracing, this calculation is done in reverse. That is, given an eye point and an image plane, we generate rays from the eye to the sample positions on the image plane. By following the rays beyond the image plane and into the scene we can determine which object is projected onto the image plane. Once the nearest intersection has been found, we can turn to calculating the correct color of that point. This is done by measuring the incoming light that is reflected in the viewing direction. This measurement can be accomplished by recursively spawning new rays into the scene, and measuring the light coming from those directions. This process is illustrated in figure 1. Here, we see a scene consisting of a specular (or reflective) sphere and a diffuse (or matte) sphere. As can be seen, the direction and type of newly spawned rays depend on the material of the intersected surface. Each sample on the image plane may thus give rise to many rays that must be traced through the scene before the final pixel color can be known.

A simple Whitted-style ray tracer yielding behavior as in the illustration can be expressed in pseudo-code like this (taken from [Jensen]):

```
function render()
  for each pixel
    generate a ray from the eye point through this pixel
    pixel color = trace(ray)


function trace(ray)
  find nearest intersection with scene
  compute intersection point and normal
  color = shade(point, normal)
  return color


function shade(point, normal)
  color = 0
  for each light source
    trace shadow ray to light source
    if shadow ray intersects light source
      color = color + direct illumination
  if surface is specular
    generate a new reflection or refraction ray
    color = color + trace(new ray)
  return color
```

A Whitted ray tracer is conceptually a simple program, and it fails to simulate several important effects, as will be explained in the following section.

2.2 The Rendering Equation

The problem of correctly computing a complete solution to the global illumination problem was formalised by Kajiya who introduced the so-called rendering equation [Kajiya]. The rendering equation gives a generalized description of how light behaves, based on known radiometric results.

The rendering equation is:

$$L(x,\varphi) = L_e(x,\varphi) + \int_{\Omega_x} \rho(x,\varphi,\theta)L(x',-\theta)(\theta\cdot n)d\theta$$

Where

- $L(x,\varphi)$ is the outgoing radiance of a surface at the point x in the direction of φ.

- $L_e(x,\varphi)$ is the emitted radiance from a surface at x in the direction of φ. This term does not include reflected radiance.

- Ω_x is the unit-hemisphere around x.

- x' is the location on the first surface hit by the ray leaving x in the direction of θ.

- n is the surface normal at x.

- $\rho(x,\varphi,\theta)$ is the fraction of light coming from direction θ that hits x and is reflected in the direction φ. This is also known as the bidirectional reflectance distribution function (BRDF).

Intuitively speaking, the rendering equation says that the outgoing light at x in the direction of φ is the light that the surface emits plus the sum of all the incoming light that is reflected in the direction of φ.

Whitted's classic ray tracer uses only perfect reflections and perfect refractions to account for indirect light and so is a very crude approximation to the rendering equation. For example, it does not take diffuse interreflections into account. A more complete method for approximating the rendering equation – known as path tracing – was given by Kajiya in the same article that introduced the rendering equation. In contrast to the Whitted ray tracer, path tracing accurately simulates color bleeding between diffuse surfaces, imperfect reflections and refractions, caustics, soft shadows, and motion blur.

A path tracer extends the Whitted ray tracer using Monte Carlo integration[1] to evaluate the rendering equation.

[1]Monte carlo integration evaluates the integral using random sampling

2.3 Parallel ray tracing

Whitted reported up to 95% of the total rendering time being spent in the intersection code. This means that almost all computation is carried out in calls to the `trace` function in the pseudo code. Additionally, calls to the `trace` function from the `render` function can be carried out in any order without affecting the results. These observations lead us to what turns out to be an important property of ray tracing: The heavy work of tracing rays through the scene can be carried out in parallel.

In principle, as many processors as there are primary rays can carry out the computation (though in practice, it may often be faster to use fewer). Ingo Wald reports a speedup almost linear in the number of CPUs used [OpenRT].

2.4 Summary

In this section we have briefly introduced ray tracing as a method for computing realistically looking images by tracing rays through the scene. Our main focus is on the relatively simple style of ray tracing known as Whitted ray tracing. We established that ray tracing is an algorithm extremely suited for parallel execution.

3 General purpose computation on the GPU

Having established in the previous section, that ray tracing is a good candidate for parallel execution, we will in this section look at the GPU's capacity for parallel computation and how to exploit it. Doing computations such as ray tracing on a GPU falls under the umbrella that is general purpose computation on the GPU (GPGPU). We cover the aspects of this field relevant to our ray tracer implementations.

3.1 Rasterized graphics

In spite of the fact that GPUs are designed to produce 3D graphics in fractions of a second, it turns out to be no trivial task to convert the algorithm of ray tracing so that it can be run efficiently on a GPU.

The reason is that the GPU is designed to produce what is often referred to as *rasterized graphics*. Rasterized graphics is a simple process in which each triangle in the scene is handled independently by the GPU. Triangles are projected onto the near plane of the viewing frustum, as seen in figure 2. The viewing frustum is the volume that bounds the field of view.

Handling each triangle independently makes it difficult to properly produce global illumination effects, such as shadows cast by other triangles or reflections of other triangles.

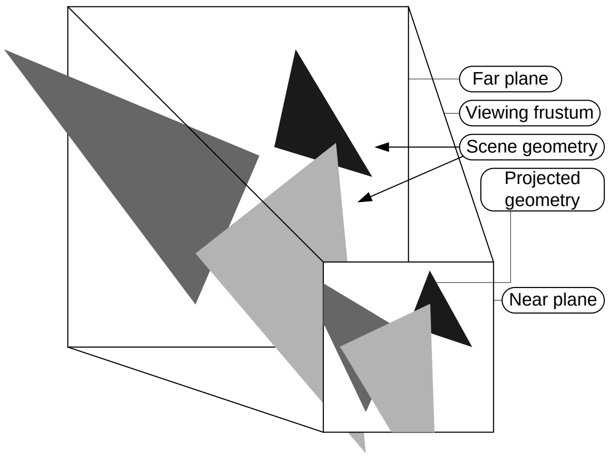


Figure 2: Rasterized graphics: Geometry within the viewing frustum is projected onto the near plane.

Though ray tracing is a simple algorithm as well, it is also a very different approach to rendering 3D scenes on 2D displays. For one thing every triangle in the scene can potentially influence the final color of a screen pixel. Given this apparent incompatibility between the hardware and the algorithm, one might wonder what the motivation is.

Modern day GPUs have up to 24 pixel pipelines operating in parallel. This means that GPUs have a great potential for parallel computation. As ray tracing speeds up linearly with the number of processors running it, the motivation for using the GPU is clear. On top of the parallel capacity, a GPU such as the Geforce 6800 ultra has an observed performance of over 50 GFlops. For the sake of comparison, a 3 GHz pentium 4 CPU has its theoretical peak at 6 GFlops [GPU Gems 2].

In the following subsections we cover the basic concepts of GPGPU. These include the OpenGL pipeline, fragment programs, and the streaming computational model. In the last part of this section some more advanced topics from GPGPU will be covered.

3.2 The OpenGL pipeline

In order to properly understand the framework in which GPGPU is done, a bit of knowledge of the standard graphics pipeline is required. Both OpenGL and DirectX can be used, but since this project relies on OpenGL exclusively, we will restrict ourselves to that framework.

Fixed function pipeline

When creating normal rasterized graphics, OpenGL provides a convenient abstraction, hiding the GPU details from the programmer. The fixed function pipeline denotes the default transform and lighting of vertices along with interpolation of colors between the vertices, when triangles are passed to OpenGL.

Programmable pipelines

Figure 3 illustrates the position in the pipeline where vertices are transformed and where fragments are processed. If, for some reason, we should want the pipeline to behave differently, we can supply OpenGL with custom vertex and fragment programs, effectively replacing the fixed function pipeline. This has been done extensively in recent computer games such as Far Cry, Doom 3, and Half Life 2. In these games, techniques such as bump mapping are implemented using fragment programs. On the left in figure 3, we illustrate how normal rasterized graphics applications may customize the vertex and fragment processing stages to achieve special effects.

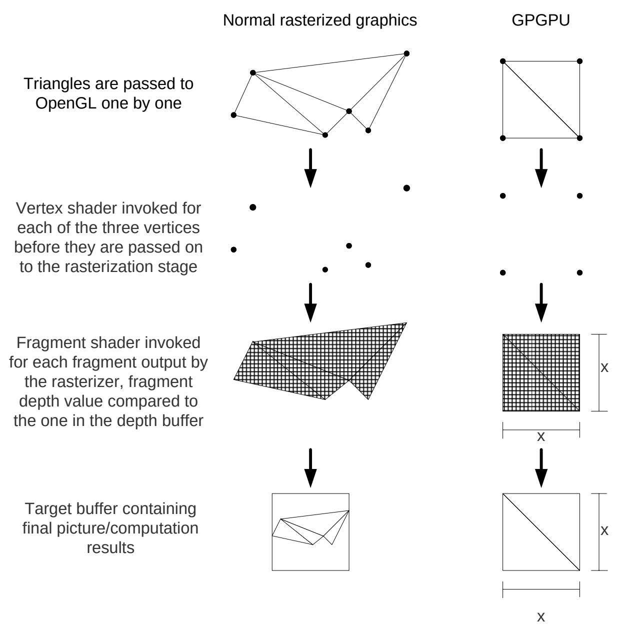

Normal rasterized graphics
GPGPU
Triangles are passed to OpenGL one by one
Vertex shader invoked for each of the three vertices before they are passed on to the rasterization stage
Fragment shader invoked for each fragment output by the rasterizer, fragment depth value compared to the one in the depth buffer
Target buffer containing final picture/computation results
x
x
x
x

Figure 3: Graphics pipeline from the GPGPU point of view.

The main idea behind GPGPU is to view the input and output of the graphics engine as general data as opposed to image-specific data as it is the case in the left column. The right side of figure 3 illustrates how a simple patch consisting of two triangles is sent through the pipeline. This quad gets transformed so that it precisely fills out the rendering window after rendering is done. In this way one fragment program invocation is done for each pixel in the resulting picture.

Say we have an algorithm suitable for parallel computation. If the problem can be divided into n smaller problems, that can each be solved independently, we setup OpenGL to render in a resolution $x \times x$ where $x^2 \geq n$, and x is a power of two. Keeping the rendering target square and at resolutions of a power of two is done to achieve optimum performance. We need to stay within the operating parameters that the hardware was optimized for. Now we can implement our parallel algorithm as a fragment program and have it run on our n subproblems simply by drawing the quad to an $x \times x$ buffer, and set up appropriate input. This is how GPGPU works in a nutshell.

Fragment programs are a key component in GPGPU and will be discussed in detail in the following section.

3.3 Fragment programs

The strength of the GPU lies in its capacity for parallel computation. In the Geforce 6800, up to 16 pipelines are available for fragment processing and 6 pipelines for vertex processing. As the numbers indicate, the main processing power of the GPU lies in the fragment processors. Our fragment programs are all written in a C-like language called C for graphics or Cg for short. The Cg code is compiled to assembly-like code which is then executed on the GPU.

Input to fragment programs comes in a few varieties. Vertex attributes such as texture coordinates, color, and position can be passed on from the vertex program through the rasterizing stage, and used as input to a fragment program. This means that each fragment gets different values as input with such interpolated parameters.

Parameters that are passed directly to the fragment program from the CPU are referred to as *uniform* parameters, since they are constant over all fragments. Uniform parameters can be one of several types. The following are the ones we have found useful.

1) A single floating point value.

2) A two to four dimensional vector of floating point values.

3) A texture handle.

A texture handle is a pointer to an array of floating point values, stored in GPU memory. Textures are normally used in rasterized graphics for simulating detailed surfaces. A picture of a piece of wood might be projected onto some triangles to make it appear they are made of wood. In GPGPU, we use textures to hold our input data.

Fitting our general purpose computations into the graphics framework carries with it some limitations. One consequence is that the only place a fragment program can store persistent data is in the pixel in the output buffer on which it was invoked. This renders certain types of computation inherently difficult to do on the GPU. Sorting is an example of such a computation [GPU Gems, p.627]. Most sorting algorithms require random access writes to run efficiently. Writing data in random locations in the output buffer is generally referred to as a *scatter* operation. The term *gather* operations refers to reading of data from random locations. As opposed to scatter, gather *is* supported since we can do texture lookups at coordinates calculated at runtime. This is also referred to as data dependent texture lookups.

3.4 Pbuffers

When doing GPGPU computations, fragment programs generally do not write their output to the screen like in normal graphics applications. In stead they write to special off-screen pixel buffers (widely known as *pbuffers*) which can later be used as input textures in subsequent rendering passes.

Each pbuffer has its own rendering context which may hold up to six drawable surfaces (for the Geforce 6800). A fragment program can write output to up to four such surfaces in each invocation. This is known as multiple render targets (MRT). Each surface may be treated as a texture, allowing it to be used as input to fragment programs. In addition to the mentioned surfaces, each pbuffer has its own depth buffer. Depth buffers are used in rasterized graphics to determine which of two overlapping triangles is closer to the camera.

Many types of computation require many iterations to reach a result and often it is the case that the output of iteration k is the input for iteration $k+1$. In GPGPU this scenario is traditionally handled using two pbuffers. In each iteration one pbuffer serves as input buffer and the other as the output buffer. After each iteration the two buffers swap roles and the next iteration can be carried out. This technique is known as *ping pong* [GPU Gems].

Redirecting output from one pbuffer to another requires a so-called context switch which is an expensive operation, so it is desirable to use the same buffer for both input and output, making the context switch redundant. This is supported so long as a program does not attempt to read from the buffer surface it will eventually write its results to. Reading from the output buffer is a convenient feature, but the extension specification[2] states that results from programs doing this are undefined. As it turns out we do in fact get precisely the behavior we want, despite the lack of guarantees. We use just a single buffer for our computation since it is faster than switching between two buffers.

There is only one problem with the single buffer approach: The programmer cannot know in advance the order in which fragments are processed by fragment programs. For this reason we must not read data from any pixels other than the one we will eventually write to. Any other data read from the output buffer could be either the result of the last iteration or the current one. One can still do gather operations on other input textures, and so this has not proven to be a serious limitation for our purposes.

In figure 4 we illustrate the general order of events, when rendering to a pbuffer. First, a pbuffer is bound as the current rendering context, and

[2]http://oss.sgi.com/projects/ogl-sample/registry/ARB/wgl_render_texture.txt

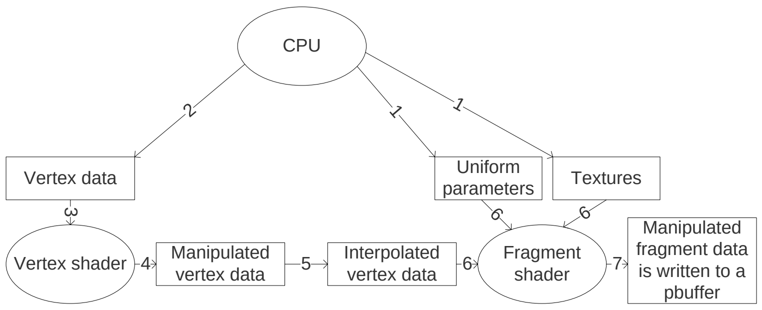


Figure 4: Pbuffer rendering: Ellipses indicate programmable stages. Boxes represent data.

the appropriate surfaces in the buffer are chosen to receive the output of the computation.

1) The CPU then binds input textures to be used by the fragment program and

2) starts the computation by passing a screen sized quad to OpenGL.

3-5) The quad is passed through the vertex processor

6) and finally, the fragment program is run with the parameters bound by the CPU in stage 1 along with the interpolated texture coordinates allowing us to look up the correct data in the input textures.

7) The results are written to the designated surfaces in the active rendering context, provided they pass the depth test.

3.5 Streaming computation

In the streaming computational model [Purcell], programs are denoted *kernels* and data is spoken of as *streams*. Computation is carried out by arranging input data in a stream and feeding the stream to a number of either physical or logical processors each executing a kernel on the stream elements one by one. The results of each kernel invocation is placed in an output stream. As pointed out by Purcell, programming for the GPU fits almost perfectly into this computational model. Data textures are operated on by fragment programs much like streams are operated on by kernels.

Due to the tidiness of this computational model, we decided to put together an abstraction layer allowing us to deal with kernel- and stream objects in our C++ code rather than dealing with the 3d-graphics interface through OpenGL directly. The work done on this streaming abstraction has paid for itself several times over, since much time was saved on development of the numerous Cg programs we have created since. The library *brook* [Buck et al.] is a high level streaming computation library for the GPU. We decided against using it because we want certain low level features available, so that the abstraction does not cost us performance.

A UML description of our library can be seen in figure 5. The diagram shows the inheritance hierarchy and lists main methods. Associations and non-essential methods have been left out for the sake of clarity.

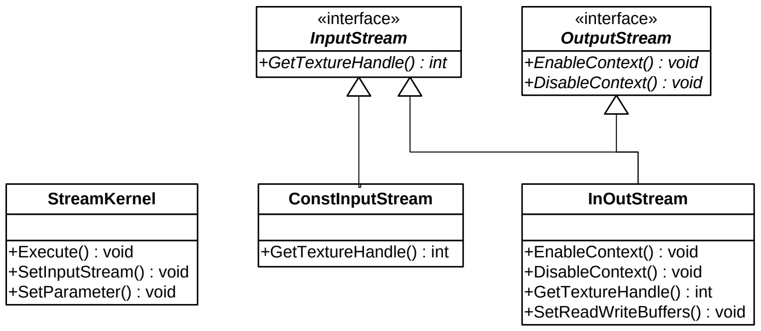


Figure 5: Library for streaming computation on the GPU

StreamKernel encapsulates fragment programs and their parameters. A rendering pass is done by simply invoking the 'Execute' method, once appropriate input and output has been set up. The 'Execute' method takes a boolean indicating whether to count the number of pixels written or not. This is called *occlusion querying* and is introduced in section 3.7.

InOutStream encapsulates a pbuffer and its associated rendering context as described in section 3.4. Any kernel invocations between calls to Enable- and DisableContext will be expected to write to the InOut-Stream in question. The SetReadWriteBuffers method is there to allow ping pong to occur (or rather just pong, since we only use one buffer) without unnecessary context switches. It allows setting up appropriate surfaces to be read from and written to, respectively.

ConstInputStream provides a wrapper for normal textures that cannot be written to. These are used for supplying the fragment programs with data that does not change, such as scene representations.

The main philosophy behind the abstraction classes is to ease the implementation of programs using the GPU while still leaving exposed those details that are important to reach peak performance. These details include support for occlusion querying and change of input and output buffers within the same rendering context.

In the following sections we will elaborate on these and other important aspects of getting good performance out of code running on the GPU.

3.6 Z-culling

In all but the simplest GPGPU applications some form of iterative computation is involved. It is often the case that one pixel in the result texture has its final result ready while other pixels still need more iterations through the fragment program. When this is the case, the kernels could be executed on all stream elements over and over until all pixels have their final value. This is not an optimal approach since many kernel invocations will be done on already finished data.

The solution can be found in a feature found in modern GPUs: Early z-rejection. This feature is designed to disregard portions of triangles which can be determined to be behind other geometry at the rasterization stage before the fragment has gone through the potentially costly shading stage.

The z-buffer or depth buffer is a part of the rendering context. It has the same dimensions as the surface being rendered to. When a triangle is sent through the pipeline, the rasterization stage will interpolate depth values between the vertices. Each screen pixel covered by the triangle is then updated only if a fragment with a smaller depth value has not already been drawn on the pixel in question. Whenever a pixel in the screen buffer is updated, so is the depth buffer. This way, the triangle closest to the camera can be drawn correctly in front of triangles further from the camera.

When used in rasterized graphics, the z-buffer will contain piecewise continuous values. This is no requirement, however, so there is nothing stopping us from manually setting appropriate depth values in a pbuffer's depth buffer. This makes it possible to tell the GPU to discard stream elements that need no more work without even invoking the fragment program on them.

The two following examples should clarify some central and somewhat tricky details regarding the depth buffer.

Example 1: Say the fragment program does not explicitly output a depth value. In this case a depth value is still found for the fragment during interpolation in the rasterization stage. If this interpolated depth value passes the depth test against the value in the depth buffer, the fragment program is run and the results are written to the target buffer.

If the interpolated depth value fails the depth test, early depth rejection takes place causing calculations on the fragment to cease after the rasterization stage.

Example 2: If the fragment program explicitly outputs a depth value the situation is different. No early z-rejection can take place since the depth value cannot be known before the fragment program has actually been run. When the fragment program has been run, the depth buffer is updated if and only if the computed depth value passes the depth test.

If we were to update the depth buffer from the kernel that does the actual computation, then no early z-rejection could be achieved as illustrated in the examples above. For this reason, manipulation of the depth buffer has to be done in a separate rendering pass with a specialized cull-kernel that checks the computational state of each fragment, and sets its depth value accordingly. It would be nice if it were possible to only run the cull-kernel on those fragments that are not yet finished. This isn't possible because the cull-kernel falls under the category of example 2 above, which means the kernel is invoked on all fragments regardless of the values in the depth buffer.

The cull pass is still quite cheap since the culling program has very few instructions and only a single path of execution. The issue of branching and multiple paths of execution is the topic of the next section.

3.7 Branching strategies

Fragment processors are SIMD (single instruction, multiple data) machines at the core. This means that concurrent fragment programs all perform the same instructions simultaneously. In the most recent GPUs, if-else constructs and loops have been made available. The control flow constructs should be used cautiously, however. GPUs execute some fixed number of kernels in parallel. Only when all concurrent kernels agree on the execution path, is the rejected code actually skipped. When a branch is reached where different paths of execution are required by concurrently executed programs, both sides of the conditional are evaluated by all kernels, and the needed results are stored using predicated writes [GPU Gems 2, p.553].

As a consequence, the use of branching and looping leads to performance penalties in programs where most neighboring fragments do not follow the same path of execution. A rule of thumb states that a branch is acceptable if,

```
if(foo)
   h = bar + baz;
else
   h = bar - baz;
=>
h = foo ? bar + baz : bar - baz;
```

Figure 6: Trivial example of if-else converted to conditional assignment.

on average, a square of 30x30 fragments all take the same path of execution [NVIDIA 2005].

What this means, is that while it is indeed possible to write and execute CPU-style code, it is generally suboptimal in terms of performance. To circumvent this problem, alternatives must be explored. Z-culling may be viewed as an alternative control flow mechanism since it allows us to mask out the fragments that do not require further computation. In the following we elaborate on other such mechanisms we have found useful.

Conditional assignments

If-else constructs can be rewritten to SIMD-friendly conditional assignments, with C-like semantics as illustrated in figure 6. In simple cases like the one illustrated, the Cg compiler will generally do the transformation automatically, but more complicated cases often need to be transformed by hand. The GPU handles these cases by using predicated instructions. This means that we are actually evaluating both the if- and the else part, while only keeping the results we need. In simple cases this is cheaper than true branching because it does not force the GPU to flush its instruction pipeline. For conditionals with many instructions in the if and else parts, it may not be beneficial to do the conversion because it does not allow us to skip over large code blocks.

CPU based branching

Another strategy is based on the idea of letting the CPU do the branching while the GPU does the rest of the work. One way of doing this is illustrated in figure 7. Kernel AB on the left has a expensive conditional, so it is replaced by kernels A and B. The CPU can now alternate between executing the two new kernels. With this technique it is, of course, necessary to handle the situation when a kernel is invoked on a fragment which does not require the execution path represented by the kernel. As illustrated, we handle

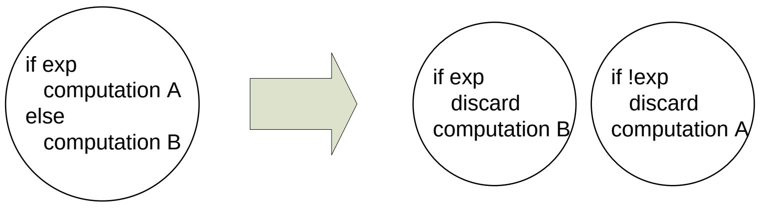


Figure 7: Kernel AB is replaced by kernels A and B to avoid branching.

this by invoking the *discard*[3] command when the wrong kernel is invoked. Alternating between the two kernels is rarely optimal. We pay a price every time we invoke the wrong kernel, so it would be nice if there were some way of knowing which choice of kernel is in greatest demand among the fragments to be processed. This is where occlusion querying comes into the picture.

Occlusion querying

OpenGL sports a feature allowing the CPU to read back the number of kernel invocations that actually resulted in an update of the target buffer. This technique is called occlusion querying. It allows CPU code to query how many pixels of the last rendering pass were occluded by previously rendered fragments. To be a bit more precise, the occlusion query counts the number of fragments that passed the depth test and did not invoke the discard command.

This means that after splitting our kernel into A and B, we can execute A and use the occlusion query to see how many fragments actually required the execution path represented by A. This number generally gives a useful hint as to which kernel to run next depending on the nature of the program. A drawback to the occlusion query is that it imposes a point of synchronization between the CPU and the GPU that in some cases may harm overall performance.

3.8 Hardware Limitations

Various limitations imposed by the hardware must be taken into account when designing fragment programs. The ones we have had to deal with are listed below along with ways to work around them. Note that these limits differ from GPU to GPU. The numbers here probably only apply to the Geforce 6800 GPU from NVIDIA that we used in our experiments.

[3]The discard command aborts execution on the fragment in question, and no result is written.

Instruction count: The number of instructions in a compiled fragment program must not exceed 1000 or the program will fail to load. To solve this problem the program may be divided into two smaller programs, to be run in succession.

Executed instruction count: Since the introduction of looping constructs the number of instructions executed at runtime often exceeds the compile time instruction count. If we attempt to execute more than 65535 instructions, results are undefined. If this limit is hit, it is a sign that a larger portion of the looping has to be done on the CPU, or the program must be split up in smaller programs.

Loop count When doing while-, for-, and do-while loops, the number of iterations cannot exceed 256. If a loop exceeds this limit, execution silently breaks out and resumes after the loop. The workaround is quite simple, though. Adding a second loop around the problem loop with the same termination criteria forces execution back into the inner loop when the loop limit is hit, effectively changing the limit to 2^{16}. Since this number matches the maximum number of executed instructions, two nested loops is all we could possibly need.

3.9 GPU applicability

Before deciding to use the GPU to boost the performance of some application, it is wise to carefully consider the hardware requirements of the fragment program code. In this section we will elaborate on a seemingly trivial problem that turns out to be hard to solve on the GPU. As an example we will discuss traversal of a binary tree as a candidate for GPU acceleration. The example serves to illustrate the importance of the concept of GPU applicability. It is sometimes difficult to predict how suitable a problem is for GPU implementation. The problem is not arbitrarily chosen as it is one we have spent a great deal of time solving as part of our experimental work.

If we cannot make any assumptions about the structure of a tree, or the traversal order, traversal requires a stack. On the CPU there is a choice between implicit and explicit stack representation, but there is no way around using some kind of stack in the most general case. In fragment programs, there is no indexable writable memory available[4]. The only places we can write data at runtime is in temporary registers, local to the fragment program, and in the output buffer, where the result of the fragment program

[4]At least not in state of the art graphics hardware. Preliminary specifications for the Sony Playstation 3 indicate that some form of ram may be available in the future.

computation goes.

We have attempted several stack implementations using temporary registers in a single pass traversal. One used texture space exponential in scene complexity and another had push and pop operations taking time proportional to maximum stack depth. Ernst et al [Ernst et al] describe a stack implementation using texture memory for storage. We found their approach impractical mainly because their push and pop operations need separate rendering passes and we need push and pop available inside our traversal kernels.

Our conclusion is that efficient general purpose tree traversal is not feasible on current graphics hardware. One must make the traversal implementation less general and handle each case separately. In the case of traversal of bounding volume hierarchies, we found an efficient strategy, which is described in detail in section 9.4.

3.10 Summary

In this section, we have gone through the technical as well as conceptual issues one faces when doing GPGPU. These include the OpenGL pipeline, fragment programs, pbuffers, and the streaming computational model. In addition, we have covered some more advanced topics including the use of early Z-rejection, occlusion querying, and other tricks to circumvent the penalties for writing CPU style code.

Lastly we have discussed some of the limitations of the GPU as a general streaming processor. We also used tree traversal as an example of a problem, that does not map well to the GPU. As it turns out, efficient tree traversal is difficult at best.

4 Mapping ray tracing to GPUs

Having introduced ray tracing and GPGPU, we now turn to describing ways of combining the two. We start by reviewing the previously proposed methods for using the GPU to accelerate ray tracing. This is followed by a more detailed description of our own approach.

4.1 Previous work

There have been two major types of approaches to using GPUs for accelerating ray tracing. The first, by Carr et al., used the GPU as a brute force ray/triangle intersection tester without any ray generation or shading [Carr et al.]. This means that a large part of the rendering still takes place on the CPU. Carr et al. report up to 120 million ray/triangle intersection tests per second on an ATI Radeon 8500. It is worth mentioning that the authors also reported that the limited floating point precision (24 bits) of this GPU gave rise to image artifacts (holes between triangles and so forth).

The second approach was due to Purcell who implemented a complete ray tracer on the GPU [Purcell]. Everything from ray generation through acceleration structure traversal to shading was done on the GPU. Since then, a number of similar projects have followed Purcell's model ([Foley & Sugerman], [Christen], [Karlsson & Ljungstedt]). Purcell did not report any precision related artifacts in his images but as his hardware also only supported 24 bits of precision, it is likely that certain types of scenes would have given rise to rendering errors.

4.2 A GPU mapping

It is obvious from the above that there is more than one way to map a traditional CPU ray tracer to a GPU assisted or GPU based ray tracer. In the following we detail the mapping used by Purcell and ourselves to implement a GPU based Whitted-style ray tracer. The general layout can be seen in figure 8.

Purcell has a similar diagram in his thesis [Purcell]. He split the intersection/traversal kernel into separate traversal- and intersection kernels. We thought it would be interesting to explore the branching capabilities that have been introduced since Purcell's work.

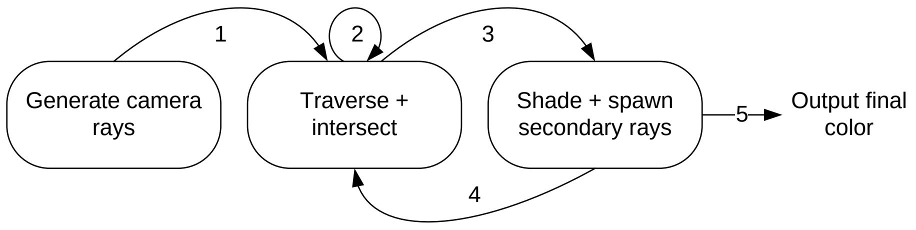


Figure 8: Division of ray tracing into kernels. Steps 3 and 4 may be taken several times.

Ray generation

Before we can start tracing rays we must generate primary rays pointing from the eye into the scene. On a GPU, we can use the interpolation capability of the rasterizer to generate all the primary rays[5] in a single kernel invocation.

Given the four corners of the viewing rectangle (the portion of the projection plane that is sampled to generate an image) and the eye point, we first compute four rays lining the corners of the view frustum. If we let the rasterizer interpolate the direction of these four rays across a 512 by 512 pixel region, we end up with all the primary ray directions needed to generate a 512 by 512 pixel image (with one sample per pixel). We can store these directions in a texture and use it as input for the intersection kernel. All primary rays have the same origin, but we still store it in a texture of the same dimensions as the direction-texture. This is necessary because the ray origin changes, when a shadow- or reflection ray is spawned.

Intersection testing

The traversal/intersection kernel takes as input ray origins and directions along with a representation of the scene. The exact representation and method of traversal is the topic of the following sections. After an appropriate number of kernel invocations[6], we output a hit record for each pixel. If a ray yielded an intersection, we report the three barycentric coordinates of the intersection, relative to the intersected triangle. Furthermore, we report how far along the ray the intersection was found and a reference to the material of the triangle. This leaves us with five floating point values making up a hit record. Textures only support four color channels (RGBA) and so it would be beneficial, if a hit record could be cut down to just four values.

[5]This assumes that we are using uniform sampling on the projection plane.

[6]As many invocations as are necessary for the last ray to have finished scene traversal.

We observe that we only need two of the three barycentric coordinates as they always sum to 1 within the triangle. This makes it possible to store intersection records in a single RGBA texture of the same dimensions as the two ray-textures.

Möller and Trumbore have proposed a ray/triangle intersection algorithm [Möller & Trumbore], which we have adapted to take advantage of the vector operations available on the GPU. The intersection code is listed in the following. The listing also serves as an example of how the vector instructions may be used to increase efficiency.

```
/**Used for triangle intersection testing.
The triangle is given by vertices a, b and c.
@param o is the ray origin
@param d is the ray direction
@param lasthit is the hitrecord for the previous hit
where .x=u, .y=v, .z=t, .w=material index for best hit
@return a new hit record with the same format as the input
*/
float4 Intersects(float3 a, float3 b, float3 c,
                  float3 o, float3 d, float minT,
                  float mat_index, float4 lasthit)
{
        float3 e1 = b - a;
        float3 e2 = c - a;
        float3 p = cross(d, e2);
        float det = dot(p, e1);
        bool isHit = det > 0.00001f;

        float invdet = 1.0f / det;
        float3 tvec = o - a;
        float u = dot(p, tvec) * invdet;

        float3 q = cross(tvec, e1);
        float v = dot(q, d) * invdet;
        float t = dot(q, e2) * invdet;

        isHit = (u >= 0.0f) && (v >= 0.0f)
                && (u + v <= 1.0f)
                && (t < lasthit.z)
                && (t > minT);

        return isHit ? float4(u, v, t, mat_index) : lasthit;
}
```

Shading

Having computed the intersection status for all the primary rays, we can look up the surface normals and material properties needed for shading. Each intersection record stores an index into the material texture that contains the triangle normals and material color and type. The three vertex normals are interpolated according to the barycentric coordinates of the hit record. The final pixel color can now be calculated as $(N \cdot L)C$ where N is the normal, L is the direction to the light and C is the triangle color.

Now, depending on the type of material we have hit (diffuse or specular), we need to generate secondary rays to account for shadows and reflections.

- If a ray misses the scene, the pixel gets assigned the color of the global background.
- If a ray hits a diffuse material, we generate a shadow ray. These rays originate at the intersection point and point towards the light source.
- If a ray hits a specular material, we generate a reflection ray. Reflection rays also originate at the intersection point, but the direction is a perfect reflection of the incoming ray relative to the interpolated normal at the intersection point. A true Whitted-style ray tracer also supports transparent materials giving rise to refraction rays. Because we are mainly focusing on acceleration, we have left out refraction rays from our implementation.
- If a shadow ray hits anything at all, it means geometry exists between the origin of the shadow ray and the light source, and so the pixel in question should be black. When tracing shadow rays, we are not interested in the nearest intersection but rather any intersection. For this reason, it is possible to speed up shadow ray tracing by stopping traversal as soon as a hit is found. In our implementation, it is possible to trace shadow and reflection rays in the same pass. For this reason we cannot use this optimization in general.

Having generated the appropriate secondary rays for each pixel, we can perform another traversal/intersection pass, if needed. Each invocation of the shading kernel returns both a color and a new ray for each pixel. The shading kernel also takes as input the color buffer output by previous shading passes. This makes it possible to combine the colors of successive specular surfaces as we trace successive rays.

In our benchmark scenes, there is only a single scene with a specular surface. When there is only a single specular surface, we may observe that we are always finished after three passes through the traversal/intersection and shading stages, since the longest path a ray can take is to first hit the mirror and secondly a diffuse surface requiring a shadow ray to be traced. If there is more than one specular surface (not in the same plane), an arbitrary number of such iterations may be necessary to account for all reflection rays.

Unlike Carr, we do not have problems with low floating point precision because our Geforce 6800 supports true 32-bit floating point operations throughout the pipeline. This makes our images visually indistinguishable from CPU renderings.

4.3 Summary

Having first established that it does indeed make sense to parallelize ray tracing, we describe how the ray tracing algorithm might be mapped to three kernels, suitable for execution on the GPU. The three kernels are: ray generation, traversal/intersection, and shading. The mapping described here is the result of an adaptation of Purcell's original mapping to one optimal for hardware supporting the fp40 profile.

5 Acceleration structure review

Now that the basics of ray tracing and GPGPU have been covered, we will turn our attention to the acceleration structures used in ray tracing. In the following section, we will make general observations regarding spatial subdivision schemes. Next, two sections follow introducing two concrete examples of such: the uniform grid and the kd-tree. Finally, we introduce the bounding volume hierarchy (BVH). Since our ultimate goal is to find good acceleration structures for ray tracing on the GPU, we will include discussions of GPU applicability for each of the structures. Each section will also include a description of our own implementation of the structure in question, including details regarding the texture representation of it on the GPU, and the algorithm to traverse it.

When doing a review of acceleration structures, one must choose which ones to include. As mentioned above, we have restricted our attention to uniform grids, kd-trees, and BVHs. These were not arbitrary choices, however.

The uniform grid is included, because it has been the structure of choice for both Purcell [Purcell] and those walking in his footsteps [Christen], [Karlsson & Ljungstedt]. As such, our review would hardly be complete without a thorough treatment of this structure.

The kd-tree has been crowned the performance winner on the CPU in a recent paper [Havran], and kd-tree traversal has recently been implemented on the GPU [Foley & Sugerman]. For these reasons, we felt it was appropriate to measure it up against our other structures.

The BVH has not been described as the fastest structure on the CPU, but traversal of a BVH is very simple and simple code generally runs much faster than complicated code on the GPU. We feel it should be examined if perhaps this could make the BVH a serious contender when it comes to ray tracing on the GPU.

5.1 A note on octrees

Another of the classic structures for ray tracing is the octree. We have chosen not to include this structure in our review for a number of reasons. The main one being that the octree is a special case of the kd-tree. In other words, any octree can be represented as a kd-tree. Therefore, the only thing that could make an octree perform better than the equivalent kd-tree is if we were somehow able to exploit the special structure of the octree in the traversal phase. Based on available litterature this appears not to be the case.

Various authors have proposed solutions to octree traversal that are more or less efficient:

- Glassner advances a point along the ray and locates the leaf voxel that contains it by searching down the octree, starting from the root [Glassner84]. In each octree node he then locates the sub-node by comparing the point against the three split planes that partition the node. In effect that means that he takes three kd-tree steps in quick succession. Therefore, the same technique could be used to traverse a kd-tree representation of an octree without loss of efficiency. However, the technique is inherently slow because each voxel step requires us to search down the tree all the way from the root.

- Sung suggests using a 3D digital differential analyzer (DDA) algorithm with a voxel size equivalent to the smallest voxel in the octree, but without using additional storage [Sung]. Because the traversal is nearly identical to that used for uniform grid, he achieves a running time that is only slightly lower than for a pure uniform grid but with less storage. Because storage is not a problem for us, this approach would defeat the purpose of using a hierarchical structure – namely the ability to move past large empty regions quickly.

- Arvo explains how to traverse an octree in linear time in the number of voxels visited [Arvo]. He does so in the context of a kd-tree representation of the octree and so does not exploit any octree specific features.

In summary, we see no reason to distinguish between octrees and kd-trees and consequently leave out octrees from this study.

5.2 Hybrid structures

In recent years, several acceleration structures have been proposed for ray tracing that are each essentially a combination of two of the structures mentioned above. A BVH/uniform grid hybrid is described by Müller and Fellner [Müller & Fellner], who achieve good results by letting the two structures do only what they do best, as uniform grids are placed in leaf nodes of a coarse BVH.

Another hybrid is the hierarchical uniform grid (HUG) introduced by Cazals et al [Cazals et al.]. It is similar to the BVH/uniform grid hybrid in that local regions of dense geometry each get their own little uniform grid. The difference lies in how these grids are arranged. With HUG the small grids are placed in voxels of larger grids. This allows for multiple levels of grids within the hierarchy. See section 7 for details on the uniform grid.

When neither of these two hybrid solutions have made it into our review it is mainly due to the complexity of the traversal code necessary to traverse such complex structures. As noted above, there is a great penalty to running complicated code with many different states on the GPU. As pointed out in section 3.9, it is inherently difficult to traverse hierarchical structures since we do not have a stack. It may be possible to map traversal of these hybrids to the GPU, but we have focused our attention on the more well-known "pure" structures.

5.3 Summary

This section serves as an introduction for the following sections covering the different acceleration structures covered by our review. We have presented arguments for including BVH, uniform grid, and kd-tree in our survey and arguments for excluding octree and various hybrid structures.

6 Spatial subdivision schemes

A number of ray tracing acceleration schemes can be classified as spatial subdivision techniques. These include uniform grids, octrees, and kd-trees. They all share the common aspect of subdividing space as opposed to grouping objects. Sections 7 and 8 are devoted to the uniform grid and kd-trees, respectively, but because they are both spatial subdivision schemes, they share a number of features. To avoid repeating ourselves, we have taken the commonalities of the two and put them here.

6.1 Subdivision

All the subdivision techniques work by taking a volume of space and subdividing it into two or more sub-volumes. This subdivision procedure may be repeated recursively on the children of the original volume, thus generating a tree of partitions. Each internal node corresponds to a partitioning of space and each leaf corresponds to a volume. For example, the uniform grid divides the original scene box into a grid of leaf voxels that are not subdivided any furter whereas the kd-tree recursively subdivides each voxel into two sub volumes until a certain criteria is met.

The decision of when and where to subdivide is based on the geometry of the scene and the subdivision scheme used. In any case, each leaf-volume has a reference to all the triangles that intersect it.

The term voxel is usually used to denote a region of space. It is a generalization of the 2D term pixel (picture element) to 3D (volume element). Most subdivision schemes work only with axis aligned boxes as voxels, but any shape is possible in principle. General binary space partition trees, for example, do not always subdivide along the major axes and so generate subvolumes that are not axis aligned boxes.

Most authors reserve the term voxel for axis aligned boxes and we follow that convention in the following.

6.2 Multiple references

A property shared by all the subdivision techniques is that a scene triangle may intersect more than one voxel. This gives rise to a number of implementation considerations.

We have to decide what to do with a triangle when it straddles a voxel boundary. Suppose we have a voxel v that we split into two sub-voxels. If v contains a triangle that intersects both of the child-voxels then we can either split the triangle into two smaller triangles and put each half in the two children or we can give the children a reference to the original triangle. As a variation of the latter, the parent node v sometimes maintains a list of the straddling objects instead of propagating the object to the children.

If we decide to reference the original triangle, it is possible that the current voxel has a reference to a triangle that intersects the ray, but where the intersection lies beyond the current voxel (see figure 9).

If we decide to split the triangles instead, then we are always guaranteed that when we test a ray against the list of triangles in a voxel then any intersection found is going to be inside the voxel. This allows us to stop the traversal as soon as we have found an intersection, provided that we have visited the voxels in order of increasing distance from the ray origin. The reason is that all the other candidate volumes and triangles will be farther away.

Therefore, in order to be able to terminate the traversal early, it is important that we only count intersections that are inside the voxel we are currently processing, or we might miss an object that is nearer to the ray origin as in figure 9.

In practice, splitting is never used because it generates more (sometimes many more) primitives that all consume memory. Also, it is easy to handle the minor complications that follow from referencing primitives that are not fully contained in each voxel (by rejecting intersections that are not contained in the voxel).

Given that a triangle may be referenced by multiple voxels, when tracing a ray, we could potentially test an object for intersection against the same ray more than once, wasting computational resources. One solution to this problem is to employ a technique known as mailboxing [Amanatides & Woo]. Mailboxing works by storing the ID of a ray with the last triangle the ray was tested against. If a ray with the same ID is tested against the triangle again, we can skip the test the second time. Of course, each ray must be given a unique ID in order for this to work.

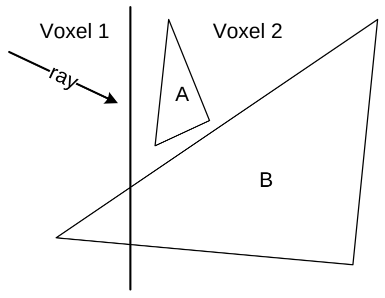


Figure 9: The intersection with triangle B is already detected in voxel 1. Stopping the traversal would mean missing the (nearer) intersection with trangle A in voxel 2.

6.3 Implications for GPUs

Mailboxing and multiple references to triangles are difficult to handle on a GPU. It is easy enough to test if an intersection point is within the voxel we are currently processing, but it may be difficult to visit the voxels in order of increasing distance from the ray origin. This is because we do not have a stack (see section 3). This is significant because then we will not always be able to stop the traversal once a voxel reports an intersection. The result is that we end up traversing more voxels and test more primitives for intersection than we stricktly need to. Consequently, we should rewrite stack based traversal schemes to an iterative approach whenever possible.

Mailboxing too causes problems in a GPU implementation. Firstly, because our triangles are stored in a texture, we are not able to write the ray's ID with the tested triangles because we do not have a random access write (scatter) operation in fragment programs. Secondly, mailboxing has an implicit assumption that rays are traced sequentially. Otherwise, the ray ID in a primitive could get overwritten by a different ray being traced simultaneously. As we are tracing multiple rays in parallel, we are effectively invalidating that assumption.

The complications with mailboxing were also identified by Purcell who decided not to include mailboxing in his implementation. In any case, the potential benefit of mailboxing is often minimal. Havran reports gains of only 5 - 6% while sometimes experiencing a slowdown [Havran02]

6.4 Summary

The common features of kd-trees and uniform grids were covered in this section. The main issues within these schemes all have to do with the fact that space is subdivided and so a triangle may find itself intersecting such a division. Different ways of handling this problem were described and reasons were given why we choose to use the multiple references – single representation strategy.

Finally, the mailboxing technique was introduced. We find that it is unlikely to yield improvements in our GPU implementations.

7 Uniform Grid

In this section we describe the uniform grid acceleration structure and its properties. The uniform grid is perhaps the most simple of all the acceleration structures for ray tracing. First described by Fujimoto et. al. [Fujimoto et al.], the main idea is to subdivide the scene into a uniform 3D-grid of voxels. Each voxel has a list of the triangles that intersect it. As such, the uniform grid is a special case of a spatial subdivision structure.

To intersect a ray with the grid, we walk through the voxels that are pierced by the ray in the order that they are pierced. For each voxel we visit, we test the list of triangles associated with it for an intersection with the ray and report the closest one, if any.

Once a voxel reports an intersection, we can stop the voxel traversal and report the intersection as the closest one. Note that no other intersection can be closer as we visit the voxels in order increasing distance from the ray origin. We also terminate the voxel walking when we reach the end of the grid.

7.1 Construction

Given a bounding box of the scene and a list of triangles, we can construct a uniform grid. The first thing we have to decide is the resolution of the grid along the three axes. Because the grid has a fixed structure, this is the only parameter we can tune in the construction phase.

A higher resolution will result in more voxels that are smaller in size. This in turn means fewer triangles per voxel and therefore fewer triangle intersection tests per voxel. On the other hand, more voxels means that we will spend more time traversing the grid.

Therefore, the best choice of resolution for a given scene is not apparent. Woo suggests using near-cubical voxels [Woo] and Havran and Sixta report experiments that show that a voxels to triangle ratio (usually referred to as the density) of 20 works well, always outperforming a the standard setting of 1 [Havran & Sixta]. Pharr and Humphreys [Pharr & Humphreys] use a setting of $3\sqrt[3]{N}$ voxels along the longest axis, where N is the number of triangles. The voxels are kept near-cubical. We have used a similar method for computing the resolution of the grid along the shortest axis. For some scenes, it is still necessary to tweak the resolution by hand to get optimal results. This has been done for all the scenes used in the benchmarks.

Once a resolution has been settled upon, we can allocate a 3D array of lists of triangles. For each triangle in the scene, we now find the voxels that it intersects and add a reference to the triangle in each such voxel.

7.2 Traversal

Like the other spatial subdivision schemes, the uniform grid achieves acceleration by only visiting a few voxels and therefore (hopefully) only a few of the triangles of the scene. Furthermore, the triangles are visited roughly in the order that they appear along the ray which makes it possible to stop the traversal as soon as a voxel reports an intersection.

The technique used for moving between voxels in the grid is the 3D equivalent of 2D line drawing – a 3D digital differential analyzer (DDA) algorithm. A basic version of this algorithm was presented by Fujimoto et al. in their original article, but we use the simpler and more efficient approach of Amanatides and Woo [Amanatides & Woo].

We present the DDA traversal in the 2D case as this makes it easier to explain. This algorithm extends trivially to 3D. We begin by identifying which voxel the ray starts in. If the origin of the ray is outside the voxel grid then we find the first point where the ray intersects the grid and use this point to locate the initial voxel. The coordinates of the original voxel are saved as integer variables X and Y. Furthermore, we create the variables $step_x$ and $step_y$ with initial values equal to ± 1 depending on the sign of the x and y components of the ray direction. These variables are used to increment or decrement the X and Y values as we step along the ray.

The next bit of information we need is the maximum distance we can travel along the ray before we cross a vertical or horizontal voxel boundary. We call these distances $tmax_x$ and $tmax_y$ – see figure 10. The minimum of these two numbers is the maximum distance we can travel along the ray and still remain inside the current voxel.

Finally, we compute $delta_x$ as the distance we must move along the ray in order for the horizontal component of the move to be equal to the width

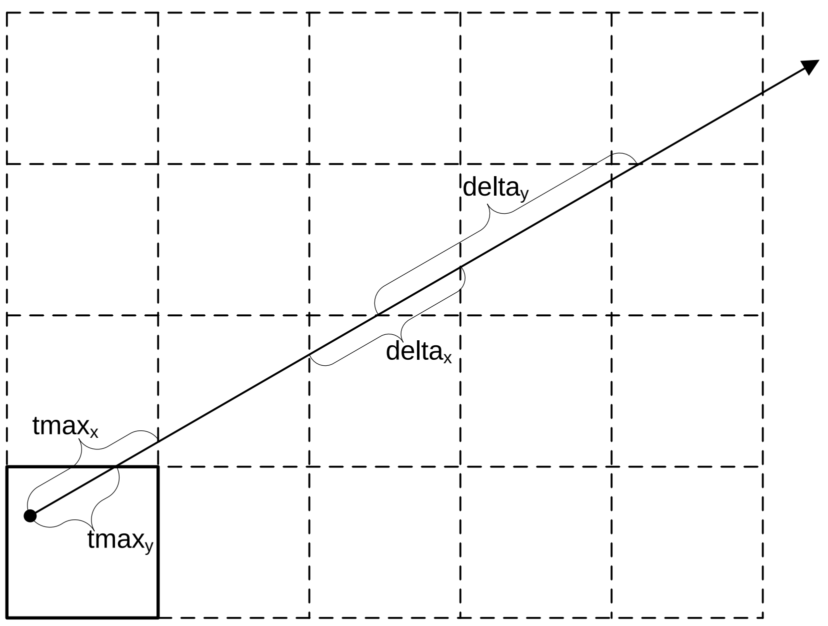


Figure 10: This figure shows the relationship between a ray, $tmax_x$, $tmax_y$, $delta_x$ and $delta_y$. The bottom left voxel contains the starting point.

of a single voxel. This is simply calculated as

$$delta_x = \frac{voxelsize_x}{raydirection_x}$$

A corresponding $delta_y$ is also calculated.

After initialization of these variable, we can now use a simple incremental algorithm to iterate over the voxels – see algorithm 1.

7.3 GPU applicability

The uniform grid was the first acceleration structure to be implemented on a GPU, due to Purcell [Purcell]. Purcell gives several reasons for choosing the grid:

> From these studies no single acceleration data structure appears to be most efficient; all appear to be within a factor of two of each other. Second, uniform grids are particularly simple for hardware implementations. Accesses to grid data structures require constant time; hierarchical data structures, in contrast, require variable time per access and involve pointer chasing. Code for

Algorithm 1 Incremental part of the DDA traversal.

```
while $X$ and $Y$ are inside grid do
    test triangles for intersection
    if we have an intersection inside this voxel then
        stop traversal and return intersection
    end if
    if $tmax_x < tmax_y$ then
        $X \leftarrow X + step_x$
        $tmax_x \leftarrow tmax_x + delta_x$
    else
        $Y \leftarrow Y + step_y$
        $tmax_y \leftarrow tmax_y + delta_y$
    end if
end while
return no intersection
```

grid traversal is also very simple and can be highly optimized in hardware.

Purcell does not argue in details why uniform grids should be more simple for hardware implementation than other acceleration structures, and so at first glance his choice may seem a bit arbitrary.

When we examine some of the key features of the uniform grid it becomes more clear why the uniform grid is a candidate for a good GPU acceleration structure:

- Each voxel of the traversal can be located and accessed in constant time using simple arithmetic. This eliminates the need for tree traversal and therefore a lot of repeated texture lookups which are expensive.
- Voxel walking is done incrementally using arithmetic. This eliminates the need for a stack and makes it possible to visit the voxels in order of increasing distance from the ray origin.
- We can exploit the visiting order to stop traversal as soon as an intersection is reported in a visited voxel.
- The traversal code is vector-oriented and highly suited for the GPU instruction set.

On the downside, the uniform grid is a special case of a spatial subdivision structure so may reference the same triangle from several voxels. As we are

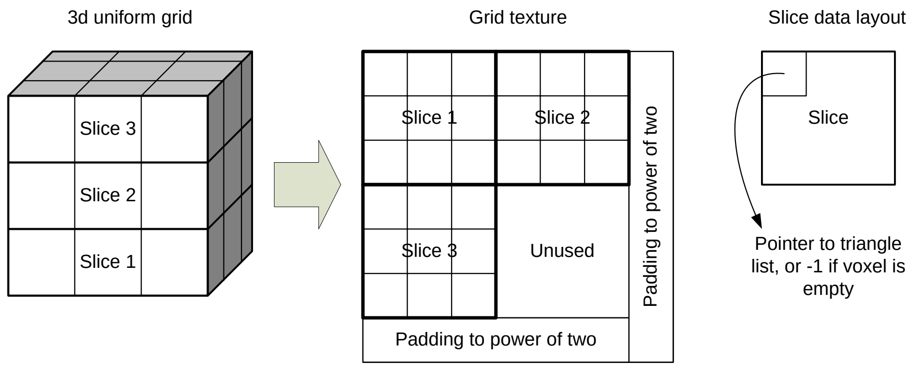


Figure 11: Uniform grid memory layout. The 3D grid of list pointers is mapped to a 2D texture.

unable to implement mailboxing, this means that we may need to test the same triangle for intersections against the same ray more than once.

7.4 GPU implementation details

Given an already constructed uniform grid, we can map the data structures to a texture layout that we can use as input on the GPU. The voxel grid can be stored directly as a 3D texture or we can pack the 2D slices of the grid into a single 2D texture. The memory layout we have chosen is the latter – a collection of 2D slices stored in a single texture; see figure 11. There are other ways of packing the data into a texture but we chose this one because it made debugging easier for us.

As each voxel only contains a single pointer to a list of triangles, we only need a texture with one 32-bit color channel which saves memory and bandwidth.

The triangle lists are stored in a separate floating point texture with one pointer to a triangle per texture element. Each list is terminated by a -1 value – see figure 12. In an RGBA texture, it would be possible to store four pointers per texture element. This would reduce the number of texture reads in the fragment program but it would also complicate the code. Unfortunately, due to time constraints we have not been able to incorporate this possible optimization into the traversal code.

Saving only pointers to triangles in the lists saves memory as we only need to store the triangle once. With that comes the penalty of having another pointer indirection per triangle and so is a typical time/space trade-off. While it may not always be necessary to make this trade-off, some scenes will reference the same triangles many times and quickly consume

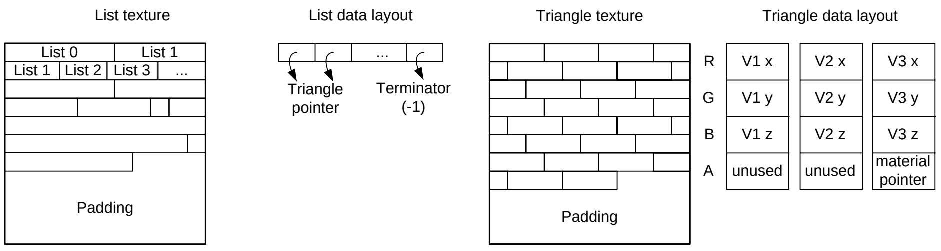


Figure 12: Memory layout for triangle lists and triangles.

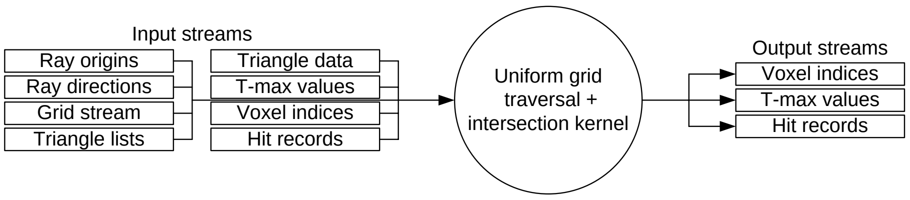


Figure 13: Uniform grid traversal as a stream/kernel computation.

large amounts of memory. In order to be able to represent as large scenes as possible, we have chosen always to use pointers.

The triangles themselves are stored in yet another texture with each triangle occupying three RGBA floating point vector values – one for each vertex of the triangle; see figure 12. This also implies that we do not share vertex data between triangles. Vertex sharing would be simple to implement as just another pointer indirection.

7.4.1 GPU Traversal

Traversing the grid requires three distinct steps:

Initialization For each ray have to find the voxel that contains the origin, or if the origin is outside the scene then the first voxel that is pierced by the ray.

Triangle testing When entering a new voxel with a non-empty triangle list, we have to test all the triangles in the voxel for intersections with the ray. To do this, we iterate over the list and test each triangle against the ray. If the end result is that we have found an intersection (within the current voxel) then we stop the traversal and return the intersection.

Traversal If the first voxel was empty or no intersections were found then we move to the next voxel using the 3D-DDA algorithm. We then start the triangle testing procedure again.

In Purcell's implementation, these three operations were handled by three different kernels. After running the initialization kernel, the traversal kernel is run repeatedly until about 20% of the kernels report that they require intersection testing (using occlusion querying). At this point Purcell then changes to the intersection kernel and runs that a number of times before changing back to the traversal kernel. Each kernel invocation only takes one voxel step (for the traversal) or performs a single ray/triangle intersection test (for the intersector). Thus, all looping and branching is done via the CPU (see section 3.7). The decision to change kernel is based on the result of an occlusion query.

In Purcell's implementation, a large number of rays may consequently be waiting for traversal while we are running the intersection kernel and vice versa. This is a great waste of resources, but without branching on the GPU Purcell really had no choice.

With the Geforce 6800 we have the fortune of true branching in our fragment programs. Consequently, we have been able to merge the intersection kernel and the traversal kernel into one – see figure 13. This means that each kernel invocation will always take either a traversal step or an intersection step. This means that no rays are left idle during a particular kernel invocation (unless we have already found the nearest intersection).

The intersection testing is performed by first running an initialization pass. This sets up the traversal state for each ray including the initial voxel. We can now run the combined traversal and intersection kernel as many times as we need to. The kernel is invoked by the CPU until all rays have found an intersection or missed the scene. We use periodic occlusion queries to determine when this is the case.

Between successive invocations of the traversal/intersection kernel, we need to store enough state to allow us to resume traversal. We store the following things:

t-max values This is a 3-vector of numbers used in the DDA traversal to determine in which direction to take the next step.

Traversing or iterating An indication of whether the kernel is traversing the grid or iterating over a list of triangles.

Voxel index The current voxel index saved as a 3-vector.

List index If we are iterating over a list, we store the index we have come to.

We map these values to two RGBA vectors (8 floats) in this way:

R$_1$ T-max for x-axis.

G$_1$ T-max for y-axis.

B$_1$ T-max for z-axis.

A$_1$ If A$_1$ is -1 we are traversing the grid. If A$_1$ is ≥ 0 then we are iterating over a list at index A$_1$.

R$_2$ Voxel-index for x-axis.

G$_2$ Voxel-index for y-axis.

B$_2$ Voxel-index for z-axis.

A$_2$ 1 if we have finised traversal, 0 if not.

Because we are using two RGBA vectors, this corresponds to two streams in figure 13.

It has been reported that current hardware does not perform well when branching is used extensively [GPU Gems 2]. We have observed that for our particular application, GPU based branching does not slow down our traversal significantly. We take this one step further by using the GPU to perform some of the looping too. While we can not always traverse the entire grid in one pass because of the GPU instruction count limit, we allow the fragment program to do as much work as possible per invocation.

It is beyond the scope of this thesis to provide detailed measurements of the consequences of using the GPU for branching and looping.

The complete code for GPU traversal is included in appendix B.

7.5 Summary

The uniform grid is a simple structure and was the first to be implemented on a GPU. The construction phase partitions the scene into a uniform grid of voxels which reference the triangles they intersect. The traversal phase uses 3D line drawing as its basis for locating the voxels that are pierced by a ray. Having located the right voxel, we can then test each triangle in that voxel against the ray.

In relation to GPUs, a uniform grid can be represented as a collection of textures. In our case, we have one texture for the grid, one for the triangle lists, and one for the actual triangles. We have implemented the traversal on a GPU using only two kernels. First, a single pass kernel initializes the traversal state. We can then use our combined traversal and itersection testing kernel for moving through the grid.

8 KD-trees

The recent statistical comparison of CPU-based acceleration structures by Havran et al. [Havran et al.] concludes that the kd-tree is the fastest acceleration structure on average for the tested scenes. Therefore, it is interesting to see if similar results apply to a GPU based kd-tree.

Like the uniform grid, the kd-tree is an instance of a spatial subdivision structure. Unlike the uniform grid, the kd-tree represents the scene as a hierarchical structure based on a binary tree.

We distinguish between internal nodes and leaf nodes in the tree. Leaf nodes correspond to voxels and have references to the triangles that intersect the respective voxels. An internal node corresponds to a partition of a region of space. Therefore, internal nodes contain a splitting plane and references to two subtrees. Leaf nodes only contain a list of triangles.

8.1 Construction

The construction of a kd-tree proceeds top-down in a recursive fashion. Given a bounding box and a list of triangles contained within, we choose an axis-perpendicular splitting plane that divides the box in two. Each primitive of the original box is assigned to the child node that contains it. If a primitive straddles the split plane then both child-volumes get a reference.

This procedure continues until we reach a maximum chosen depth or until the number of triangles in a node drops below a given threshold. In the article by Havran et al. a maximum depth of 16 and a minimum number of triangles equal to 2 is suggested for optimal performance. We find it an unlikely possibility that 16 should be the best max-depth for all realistic scenes. As the scene is halved at each level of the tree, a more plausable max-depth would be a logarithmic function in the number of scene triangles. This approach is also adopted by Pharr and Humphreys who use $8 + 1.3\ log(N)$ as their max-depth (N being the number of triangles).

The most difficult part of the construction lies in choosing where to place the splitting plane, given a set of geometry. MacDonald and Booth have analyzed this problem [MacDonald & Booth] and derived a cost function based on the fact that the probability of a ray intersecting a child node is proportional to the ratio of the surface area of the child node to the surface area of the parent [7].

This approach was later refined by Haines [Haines] who derived a different cost function that also took into account whats happens when a primitive

[7]This is usually referred to as the surface area heuristic although, strictly speaking, it is not a heuristic.

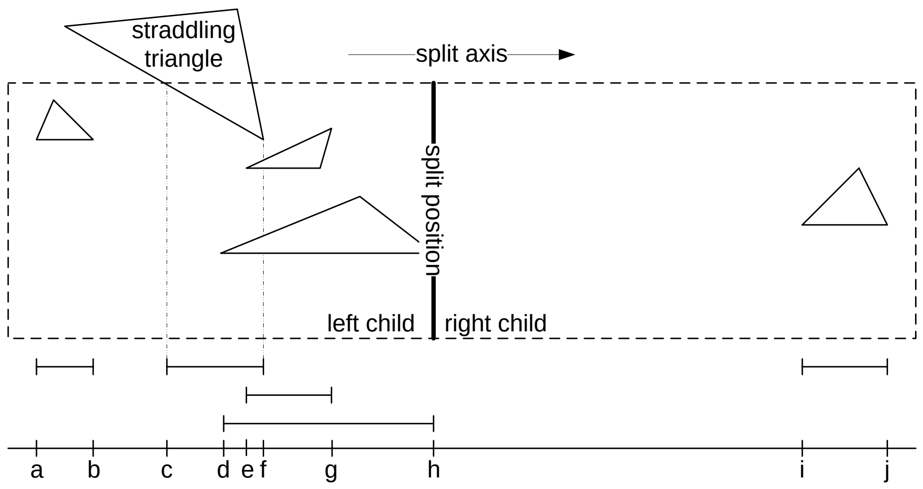


Figure 14: The cost function is evaluated at the positions a, b, $\ldots$, j. These positions correspond to the endpoints of the intervals that bound the individual triangles along the split axis. Note in particular that the bounding interval $[c, f]$ used for the straddling triangle is formed by clipping the triangle to the current voxel. Note also that the indicated split position is only our guess at where the cost function would have the lowest cost.

straddles a splitting plane. This is important because straddling triangles are propagated to both child volumes, thereby incurring a greater rendering cost. Pharr and Humphreys present a cost function that is very similar to Haines's and it is used as the basis for the construction that Foley and Sugerman use in their recent kd-tree implementation for GPUs [Foley & Sugerman].

The basis for choosing a split plane position is the evaluation of the cost function at all the edges of the bounding boxes of the triangles – see figure 14. The position with least cost is chosen as the split.

It may happen that a triangle is not completely contained in the voxel we are trying to split. In that case, we should not use the complete bounding box of the triangle, but rather clip the triangle against the voxel and then generate a new bounding volume using that instead. This is also depicted in figure 14 where the topmost triangle is not fully contained in the voxel. Havran identified this problem in his extensive treatise of kd-trees [Havran] where he dubbed the technique *split clipping* .

Note that although we clip the triangle against the voxel to get a new bounding box, the actual triangle propagated to the children of this node is the original triangle. After employing this technique, we observed improved performance of our kd-tree traversal as expected.

Algorithm 2 KD-tree construction code.

```
KDTREENODE $construct(voxel)$

if $numtriangles(voxel) \leq MIN\_TRIANGLES$ then
    return new leaf with a list of triangles
end if
if depth of tree $\geq MAX\_DEPTH$ then
    return new leaf with a list of triangles
end if
$best\ split \leftarrow$ empty
$best\ cost \leftarrow \infty$
for each axis in $\{x, y, z\}$ do
    $positions \leftarrow$ empty list
    for each triangle in voxel do
        clip triangle with respect to $voxel$
        calculate bounding box of clipped triangle
        find endpoints $a$ and $b$ of bounding box along axis
        add $a$ and $b$ to $positions$ list
    end for
    for each point $p$ in $positions$ do
        if $cost(p) < best\ cost$ then
            $best\ cost \leftarrow cost(p)$
            $best\ split \leftarrow (p, axis)$
        end if
    end for
end for
$(left\ voxel, right\ voxel) \leftarrow$ split $voxel$ according to $best\ split$
for each triangle $t$ do
    if $t$ intersects $left\ voxel$ then
        add $t$ to $left\ voxel$
    end if
    if $t$ intersects $right\ voxel$ then
        add $t$ to $right\ voxel$
    end if
end for
$left\ child \leftarrow construct(left\ voxel)$
$right\ child \leftarrow construct(right\ voxel)$
return new internal node$(left\ child, right\ child, best\ split)$
```

8.2 Traversal

Given a kd-tree node N, traversal of the subtree rooted at N proceeds as follows. If N is a leaf then test all the referenced triangles for intersection and return the nearest one if any. If N is an internal node then determine which of the two child nodes is first hit by the ray and call recursively into that node. If an intersection is found then we can return it as the nearest one. Otherwise we call recursively into the child node on the far side of the splitting plane.

In this way, we always visit the leaf voxels in the order that they are pierced by the ray. This allows us to stop the traversal as soon as we have found a leaf node with an intersection.

Havran presents an improved version of the above algorithm which takes a number of special cases into account [Havran].

8.3 GPU applicability

The immediate problem with implementing kd-trees on a GPU is that the standard traversal algorithm is recursive. Having no stack on the GPU makes this a problem. One solution would be to view the kd-tree as a special case of a bounding volume hierarchy[8] and then use the BVH traversal algorithm developed in section 9.4.1. This would mean traversing the children in a fixed order and leave us unable to stop the traversal as soon as an intersection is found, in general.

Also, because kd-trees are usually much larger than the corresponding BVHs for the same scene, we would effectively have created an inefficient BVH (see section 11.3 for memory consumption details).

The solution is therefore to employ a different traversal strategy. Before the recursive descent traversal was invented, kd-trees and octrees were traversed in a sequential manner, using a point location scheme [Glassner]. Glassner describes the technique as moving a point along the ray and descending from the root of the tree until the leaf containing the point is located [Glassner84]. He describes it as a technique for traversing octrees, but it can easily be adapted for kd-trees. This is very similar to what Foley and Sugerman do [Foley & Sugerman], except that instead of moving a point along the ray, they move a $[t_{min}, t_{max}]$ interval and use that to identify which voxels are interesting.

[8]A kd-tree is not actually a bounding volume hierarchy because the triangles that are referenced in the leaf nodes are allowed to protrude outside of the voxels. We can get around this obstacle by either clipping the triangles to the voxels or by only counting intersections that occur inside the referencing voxel.

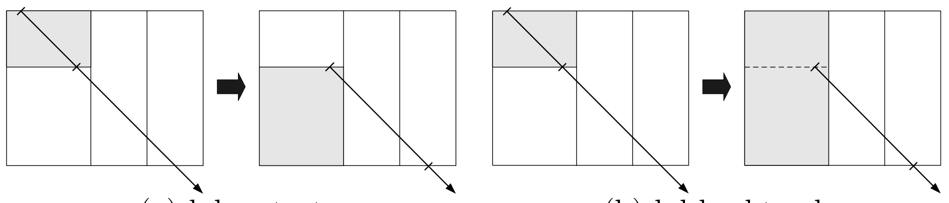


Figure 15: (a) After failing the intersection test against a leaf voxel the t-interval is set to begin at the end of the leaf. In this way we can locate the next leaf when restarting the traversal. (b) The interval is updated in the same way as for kd-restart, but instead of restarting the traversal from the root we walk up the tree until we find an ancestor which intersects the updated interval (marked with grey on the right side).

Initially, the interval spans the t-values where the ray is inside the top level scene box. For each level we move down the tree, t_{max} is set to $min(t_{max}, t_{split})$ where t_{split} is the distance along the ray to the split plane in the current node. When we are at a leaf node, the t-interval is precisely the parametric range in which the ray is inside that leaf.

If the leaf yielded no intersections, we update the t-interval. The interval we use for the continued traversal begins at the end of the current voxel and ends at the end of the scene box – see figure 15 (a).

Foley and Sugerman present their algorithm in two variations: *restart* and *backtrack*. The restart algorithm uses Glassner's approach of starting at the root of the tree for each voxel traversal step [Glassner84]. By only visiting voxels that intersect the t-interval we can guarantee that the voxels are visited in-order. Unfortunately, the restart algorithm has a slower worst case time complexity than a stack based algorithm.

To remedy this, the backtrack algorithm modifies restart by allowing us to move up the tree instead of beginning at the root each time. When a leaf voxel fails its intersection test, we move up the tree until we find an ancestor that intersects the new t-interval. Choosing the correct child node to visit when going from up-traversal to down-traversal is done in the same way as for the restart version. This gives us a the same time complexity as a stack based traversal but complicates the code a bit. Foley and Sugerman report slightly better performance with backtrack.

The use of an interval is a simpler and more elegant solution than Glassner's point location solution. It also eliminates a number of special cases from consideration.

With the stack problem out of the way there is no reason why a kd-tree could not be used in a GPU based implementation.

8.4 Our version

We have implemented the two varieties of traversal as presented by Foley and Sugerman; restart and backtrack.

Our construction algorithm uses the cost function presented by Haines [Haines] with the additional use of the split-clipping technique introduced by Havran [Havran]. For the scenes in the benchmark, a minimum-elements setting of 2 was found to be the best for all scenes. The best max-depth was scene dependent was found by exhaustive search up to the memory limit of our graphics card (see section 10.2 for details).

8.5 GPU implementation details

Mapping a kd-tree to texture representation is reasonably straight-forward. In the restart variant, each node occupies only one floating point RGBA vector (4 floats). The backtrack traverser needs two more RGBA vectors to store a per-node bounding box but is otherwise identical with respect to memory layout. Note that this means that the backtrack version uses three times as much memory as the restart version does. The per-node storage consists of:

Split axis: Which of the three axes is the split plane perpendicular to.

Split position: Where is the split plane located.

Child pointer: If this is an internal node, we need to know how to move to its two children. By always storing the left child as the node immediately following the current one, we only need to store a pointer to the right child.

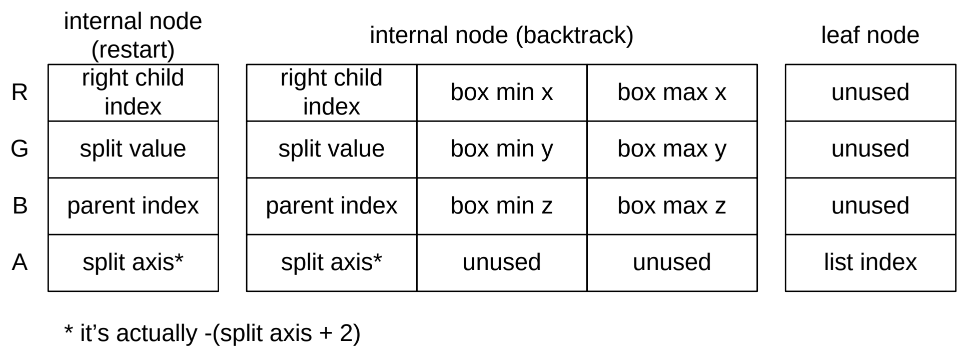
Parent pointer: For the backtrack-variant, we need to know how to move up the tree again.

List pointer: If this is a leaf node then we store the index to the list of triangles that are referenced. If the leaf is empty we store -1 to indicate that we can skip the list lookup.

Node type: Indicates whether this is an internal node or a leaf.

It is possible to store all six values in an RGBA vector by observing that they are not all used at once. For example, we store the split axis, node type, and list pointer in one floating point number. For that purpose we use this mapping:

- If the value is -1 then we have an empty leaf.



	internal node (restart)	internal node (backtrack)			leaf node
R	right child index	right child index	box min x	box max x	unused
G	split value	split value	box min y	box max y	unused
B	parent index	parent index	box min z	box max z	unused
A	split axis*	split axis*	unused	unused	list index

* it's actually -(split axis + 2)

Figure 16: The memory layout for kd-trees.

- If the value is either -2, -3, or -4 then we have an internal node and the split axis can be deduced by mapping -2 to X, -3 to Y and -4 to Z.
- If the value is 0 or greater then we have a leaf with a non-empty triangle list. The actual number stored also doubles as the position of the first element of the triangle list.

The exact memory layout is shown in figure 16.

Besides the actual tree we also need to represent the lists of triangles that leaf-nodes point to. This is done in the same way that we did for uniform grids: The leaf node points to the first element of the list and we terminate the list by storing an element with a -1 value. As was also the case for uniform grid, the list elements are only pointers to triangles and not triangles themselves. While this was a time/space tradeoff for uniform grid, we have found that it is more of a necessity for kd-trees because the number of triangle references can be very high.

8.5.1 GPU Traversal

The kd-tree and the uniform grid share the property of having voxels with lists of triangle references. This makes certain aspects of the traversal code for the two strucures very similar with the main difference lying in the voxel walking.

As was the case for uniform grid, we can break the traversal into three distinct stages:

Initialization For each ray have to find the parametric range $[t_{min}, t_{max}]$ where the ray intersects the scene box. These values are used by the traversal code.

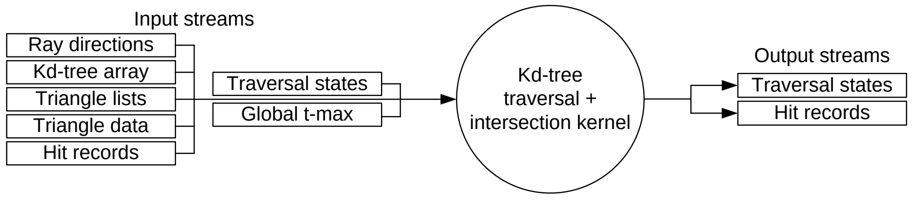


Figure 17: KD-tree traversal as a stream/kernel computation.

Triangle testing Triangle testing is identical to what happens for the uniform grid. We iterate over the list and test all triangles for intersection.

Traversal If we are not at a leaf node then we are either moving up or down the tree (only down for the restart version). Choosing the next node in the traversal depends on whether we are moving down or up in the tree, the orientation of the split plane, the position of the split plane relative to the ray, the sign of the ray direction, and the current t_{min} and t_{max} values for this ray. In the stream/kernel model, the traversal stage can be illustrated as in figure 17.

The initialization is handled prior to any traversal by a seperate kernel. As for the uniform grid, we handle both traversal and intersection in one kernel which also handles most of the looping. However, as we are usually not able to finish all the rays in one pass, the traversal/intersection kernel is invoked by the CPU until all rays have finished. After each pass, an occlusion query is used to determine the number of active rays, which signals when to stop traversal and intersection testing.

Between traversal passes, we need to store the state of the traversal so we can pick up at the right place in the next kernel invocation. The information we need is:

Traversing or intersecting The kernel can be in one of two modes: Traversing the tree or testing for intersections against a list of triangles.

List index If we were iterating over a triangle list then we need to know how far we were. To this end, we store the list index that we have come to.

Moving up or down For the backtrack version, we need to remember if we are moving up or down the tree.

Node index Whether we are moving up or down, we need to know the index of the node where we left off.

t_{min} and t_{max} The search interval for this ray is also stored so we know which branch to take in the tree.

These six values can be mapped to a single RGBA vector of four floats. We use the following mapping in our implementation:

R -2 means we are moving up the tree (for backtrack); -1 means we are moving down the tree. For $R \geq 0$, we are iterating over a triangle list at index R.

G The index to the tree node we have reached. If -1 then we have finished processing for this node.

B t_{min}

A t_{max}

This RGBA vector corresponds to the stream "traversal states" in figure 17. The complete source code of the GPU traversal for both restart and backtrack is included in appendix B.

8.6 Summary

We have reviewed the basic principles of kd-trees and explained the construction and traversal techniques. Construction starts top down and recursively splits the scene into two voxels by positioning a splitting according a cost function. We can traverse the kd-tree in recursive fashion, but this is not possible on GPUs because we do not have a stack. Instead, we can walk up and down the tree by remembering how far along the ray we have looked already. This eliminates the need for a stack and makes GPU traversal possible.

When used for GPU traversal, the kd-tree is represented as a collection of textures in much the same way as was the case for uniform grid. This means that we have one texture for the tree, one for the triangle lists and one for the actual triangles. GPU traversal proceeds by first invoking an initialization kernel and then invoking the combined traversal and intersection kernel as many times as needed.

9 Bounding volume hierarchies

In the following, we introduce the bounding volume hierarchy. We cover two types of construction algorithms and describe our GPU implementation of BVH traversal.

The BVH is a hierarchical scene partitioning structure. It differs from the spatial subdivision techniques by partitioning objects as opposed to space. The bounding volume of a piece of geometry is a simple geometric form that encloses geometry. Clearly, a ray that fails an intersection test against a bounding volume cannot hit any geometry contained within.

The motivation behind bounding volumes is that a simple bounding volume usually has a cheaper intersection test than the geometry it contains. This is more or less true depending on the complexity of the geometric primitives in question as well as the complexity of the bounding volume.

A hierarchy of bounding volumes consists of a root node with a bounding volume containing all other bounding volumes and thus all geometry of the scene. Each internal node in the tree has a number of children that are either internal nodes with associated bounding volumes or leaves with a number of triangles.

The traversal of a BVH is done using a simple and intuitive recursive descent.

9.1 Construction

A reasonable measure of the quality of a BVH, is the average cost of traversing it, given some arbitrary ray. There is no known algorithm for constructing optimal BVHs, just as it is not obvious how to evaluate the average traversal cost of a BVH.

Goldsmith and Salmon propose a cost function commonly referred to as the surface area heuristic [Goldsmith & Salmon]. They formalize this using the surface areas of the parent and the children as in the following relation:

$$p(hit(c)|hit(p)) \propto \frac{S_c}{S_p}$$

Where

- $hit(n)$ is the event that the ray hits the node, n.
- S_n is the surface area of the node n
- c and p are the child and parent nodes, respectively.

The cost function gives us an estimate of the cost of the hierarchy when intersecting it with an arbitrary ray.

Since no algorithm is available to efficiently construct an optimal BVH, different construction heuristics have been proposed. In the following some common variables are listed.

In practice the most widely used bounding volume for BVHs is the axis aligned bounding box (AABB). The AABB makes up for its loose fit by allowing fast ray intersection. It is also a good structure in terms of simplicity of implementation. Glassner gives an overview over studies done on more complex forms [Glassner, p.206-211].

Most papers on BVHs describe BVH construction schemes based on one of two basic ideas introduced in publications by Kay and Kajiya [Kay & Kajiya] and Goldsmith and Salmon [Goldsmith & Salmon] respectively. Kay and Kajiya suggest a recursive top down approach. The construction is outlined in algorithm 3.

Algorithm 3 Kay/Kajiya BVH construction

```
BVNODE BuildTree(triangles)

if we were passed just one triangle then
  return leaf holding the triangle
else
  Calculate best splitting axis and where to split it
  BVNODE $result$
  $result.leftChild \leftarrow$ BuildTree(triangles left of split)
  $result.rightChild \leftarrow$ BuildTree(triangles right of split)
  $result.boundingBox \leftarrow$ bounding box of all given triangles
  return $result$
end if
```

Goldsmith and Salmon proposed a more involved bottom up construction scheme. It does not lend itself well to a pseudo code listing. Instead, we will describe its main steps in the following.

The algorithm begins by assigning the first triangle as the root of the tree. For each additional object, the best position in the tree is found by evaluating a cost function in a recursive descent, following the cheapest path down the tree. Finally, the object is inserted as either just a new leaf or by replacing an existing leaf with an internal node containing the old leaf and the new object as its children. As a result of this approach, an internal node may have an arbitrary number of children, as opposed to the binary trees produced by the Kay/Kajiya approach.

Goldsmith and Salmon state that the quality of the resulting BVH depends strongly on the ordering of the triangles passed as input. As a remedy, they recommend randomizing the order of the triangles before the construction.

9.2 Traversal

The standard way of traversing a BVH is a recursive descent. For internal nodes we test the ray for intersection against the associated bounding volume. If an intersection is found, we test the ray recursively against the child nodes. Note that unlike the kd-tree, we have to visit all the children because they may overlap and are not sorted in any way by default. If the ray does not intersect a node's bounding volume, we skip its children. For leaves, the ray is simply tested for intersection against the triangle held by the leaf, and traversal resumes depending on the contents of the recursion stack.

The main issue in traversal of a BVH is the order in which a node's children are traversed. Kay and Kajiya propose a method, by which it is attempted to select the child closer to the ray origin along the direction of the ray. Their technique is quite involved in that it requires an additional priority queue to be maintained, in order to quickly extract the next node to be traversed. The authors do not have any mentioning of the performance gained by sorting. Shirley and Morley advise against sorting because no significant performance gain is to be expected [Shirley & Morley].

9.3 Our version

We have implemented both the top-down and the bottom up constructions, in order to do comparisons between the two on our test scenes. We use axis aligned bounding boxes as bounding volumes in both constructions. Our traversal strategy is to always traverse the children in a fixed left-to-right order. We discuss the consequences of this choice in section 9.4.3. First we elaborate on the specifics of our implementations of the two BVH construction schemes.

9.3.1 Kay & Kajiya

Our implementation of this variant was originally inspired by Shirley and Morley [Shirley & Morley, p.131-143]. The top-down approach automatically leads to a binary structure.

The authors suggest identifying the axis along which the current bounding box of the current node has its greatest extent and subdivide the contained geometry according to the spatial median along this axis. They further suggest distributing the triangles into the two child nodes according to their position relative to the spatial median.

As a slight variation over this technique, we compute the subdivision along all three axes and perform the actual split along the axis that yielded the smallest combined surface area of the child bounding boxes. We do this in an attempt to get more tightly fitting bounding volumes. This has yielded a small improvement in performance.

9.3.2 Goldsmith & Salmon

We implemented this variant according to the original article describing the technique [Goldsmith & Salmon]. To increase the likelihood of constructing a good tree, we construct the tree a number of times (15 times by default), while shuffling the list of triangles between consecutive constructions. We keep the one with the smallest estimated cost based on the cost function.

9.4 GPU implementation details

Before we can traverse a BVH on the GPU, we need to solve two problems. Firstly, we need to find a way to traverse the tree efficiently without a stack. Secondly, we need to find a suitable representation of the BVH, so that it can be passed to the fragment program in a texture. These problems are much the same as those we faced when implementing kd-tree traversal. In the following section, we will show how to take advantage of unique features of the BVH to make a simple and elegant solution to both problems.

Firstly, we will describe the algorithm used to convert a CPU tree representation, so that it can be traversed without a stack. Secondly, we look into the details of fitting the converted tree into texture memory, so it can be passed to the GPU.

9.4.1 Traversal representation

As it turns out, the traversal of the tree and the representation of the tree are interdependent. The solution lies in a careful choice of texture representation. What we store in the texture is in a sense the traversal of the tree rather than the tree itself.

The idea stems from the observation that all rays traverse the tree nodes in roughly the same order: Depth first, left to right. The idea is illustrated in figure 18.

The nodes of the tree are numbered sequentially according to the traversal order. The numbering also matches the sequence of the nodes in their array/texture representation. A dotted arrow $a \rightarrow b$ represents the fact that a ray that misses the bounding box of a must resume traversal at b. As is evident from the figure, such a dotted arrow can be stored in the array entry for a as the index of b. This forward pointer is what we call the *escape index.*

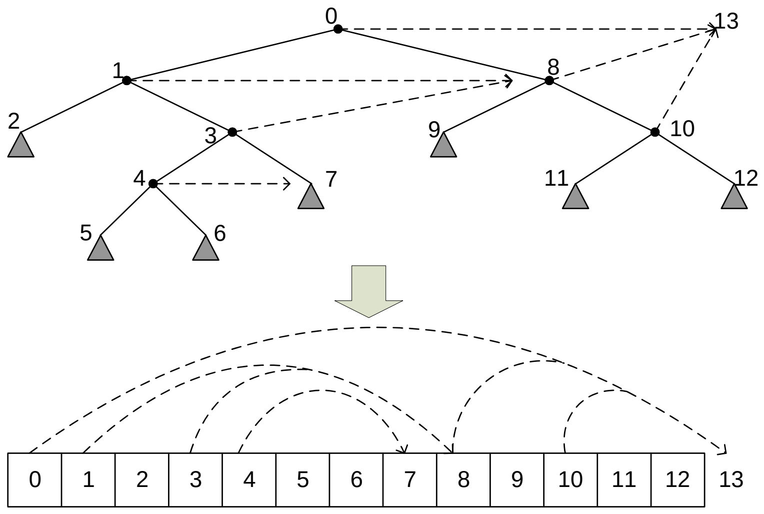


Figure 18: BVH traversal encoding example.

From the figure it can also be observed that all leaves have a relative escape index equal to one. As a consequence, we do not need to store escape indices with triangles. Note the convention hinted to in the figure that internal nodes on the right edge of the tree have escape indices equal to the total number of nodes. This is due to our loop termination criterium in the traversal algorithm.

Algorithm 4 The GPU traversal algorithm

```
$S \leftarrow$ The traversal sequence
$r \leftarrow$ The ray
$currentIndex \leftarrow 0$
**while** $currentIndex <$ length of $S$ **do**
   $currentNode \leftarrow$ element at position $currentIndex$ in $S$
   **if** $r$ intersects $currentNode$ **then**
      $currentIndex \leftarrow currentIndex + 1$
      save intersection data if $currentNode$ is a leaf/triangle
   **else**
      $currentIndex \leftarrow$ escape index of $currentNode$
   **end if**
**end while**
```

A traversal represented in this way provides for simple traversal code on the GPU. The algorithm required to execute the traversal, stored in the sequence, is sketched in algorithm 4. The while loop always terminates since the counter is always greater after an iteration than it was before; a con-

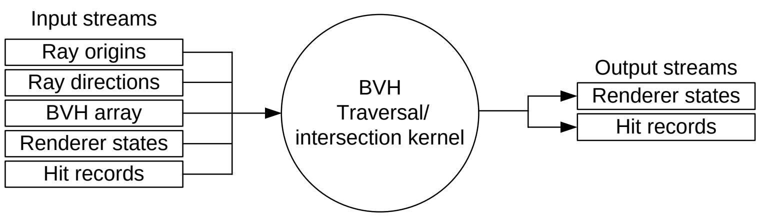


Figure 19: BVH traversal as a stream/kernel computation.

sequence of escape indices always pointing forwards in the sequence. The actual Cg traversal code can be found in appendix B.

In the stream/kernel model, the traversal can be depicted as in figure 19. The loop in algorithm 4 is split up, so that some fixed number of iterations can be done on the GPU in a single pass. The CPU uses occlusion querying to determine if there are still fragments that require more iterations. If this is the case, another rendering pass is done with the traversal kernel. This approach has the disadvantage that we must store the state of the traversal in texture memory, whenever we exit from a rendering pass without having found the closest intersection. For the BVH, this is easy, since all we need to store is the index of the next node to be processed. We store this in the stream marked "Renderer states". In addition, we also need to store a hit record holding information of the closest intersection so far.

9.4.2 Partial evaluation

Automatically preparing a tree traversal once and executing it many times could be viewed as an application of partial evaluation. We have implemented this technique as an acceleration structure for our CPU ray tracer as well, and observed a consistent speedup over traversal of the same BVH structure in the recursive manner. The reason for this is likely to be the lack of function calls and stack operation with the new representation.

9.4.3 Fixed order traversal

The main disadvantage of our traversal approach is that we are forced to traverse the children of a node in a fixed order, regardless the direction of the ray. A scene illustrating a problematic scenario can be seen in figure 20. The problem is in the fact that the ray hits all the bounding boxes of the hierarchy. The grey boxes indicate a plausible set of bounding boxes for a hierarchy built over the given set of triangles. Triangle A yields the intersection we are looking for. As we will demonstrate, this can be determined very fast or

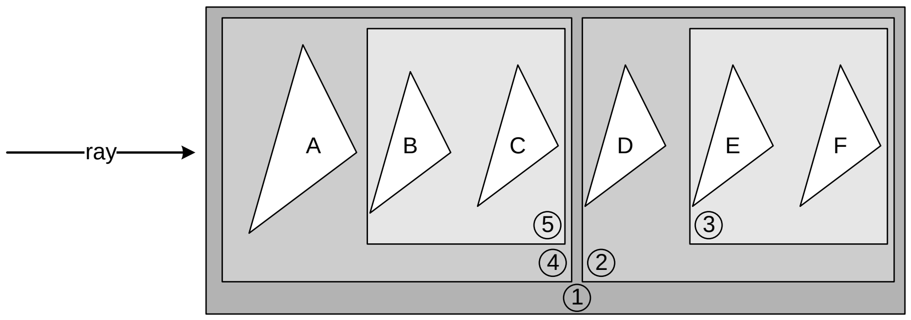


Figure 20: Potentially expensive scene layout for the Kay/Kajiya BVH construction.

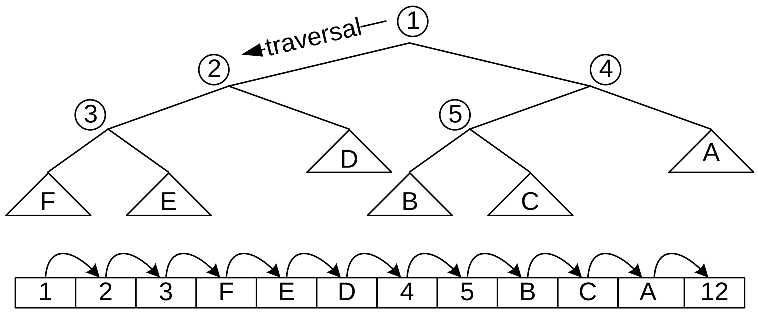


Figure 21: Worst case hierarchy corresponding to the geometry in figure 20.

very slowly, depending on the construction of the hierarchy, specifically the ordering of a node's children.

In the worst case, nodes may be arranged as depicted in figure 21. The arrows show how the traversal would play out on this particularly unfortunate scenario. As illustrated, such a traversal would require up to $2n - 1$ loop iterations (n leaves and up to $n-1$ internal nodes). Note that the motivation for using acceleration structures in the first place is to improve on the brute force approach costing up to n loop iterations.

In the best case, the hierarchy is constructed as in figure 22. The time to find the correct intersection here takes time linear in the height of the tree. Once we have found this intersection we discard the bounding boxes 5 and 2 because they are farther away than the nearest recorded triangle intersection.

In order to minimize the likelihood of ever encountering scenarios like that of figure 21, Tim Foley suggested that we flip a coin at every internal node during construction to choose an ordering of its children. The idea is to minimize the probability of traversing the nodes in the worst-case order (as in figure 21) by not sorting the internal nodes along any axis. Note however,

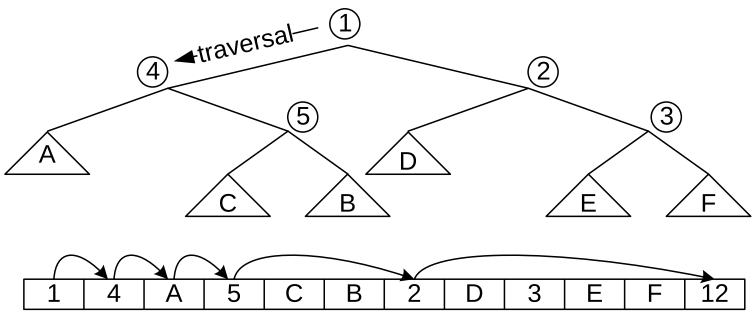


Figure 22: Best case hierarchy.

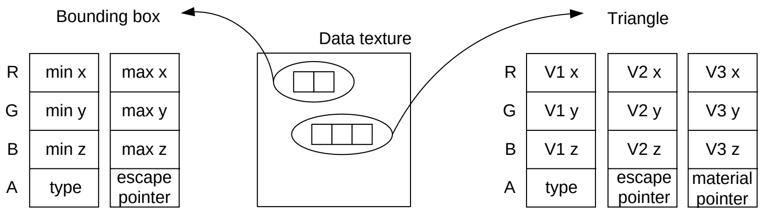


Figure 23: Data texture layout.

that the average traversal time for an arbitrary ray is unlikely to change as a result of this coin flip. All we can hope to accomplish is a reduction of the render time for the most costly camera angles.

9.4.4 Data representation

Since we now have a way of representing the tree traversal all we need is a format defining how the raw data is stored in the texture. We propose a representation in which nodes are represented as either a pair or triple of texture elements. The exact representation can be seen in figure 23. The node sequence output by the conversion described above is simply stored in the texture so that internal nodes are represented by a bounding box and leaves by a triangle as indicated on the figure.

The format allows us to read the first two texture reads without knowing if we are reading data representing a box or a triangle. The fourth component of the first block will then indicate how we should interpret the data and if we need the data from a third block in the case of the triangle.

9.5 Summary

In this section, we have introduced the BVH. Among these, we have chosen two for implementation and come up with a way of transforming both types of trees to allow traversal on the GPU. Our playback traversal method was described in detail and pros and cons were discussed. The biggest advantage is the simplicity of the traversal code, while the biggest disadvantage is likely to be the fixed order traversal.

10 Experimental setup

An inherent problem within the field of ray tracing is that of accurately comparing efficiency of one rendering system against another. In order to do this, we must find a suitable unit for measuring a system's performance. Results are generally influenced greatly by rendering parameters such as image resolution, number of triangles, rays per pixel, the camera parameters, the number of light sources and their position, geometric and spatial coherence in the scene, and occlusion of parts of the scene.

We would like to be able to compare our results with those from other publications within the field. Doing this properly, however, requires agreement on all of the above mentioned parameters. Attempts have been made to reach agreement on a standardized set of benchmark scenes. One such attempt is the standard procedural database (SPD) [Haines87]. A problem with the SPD is that it factors out very few of the above listed parameters. Even parameters such as scene complexity and triangle count are still variable using the SPD since they are generated procedurally.

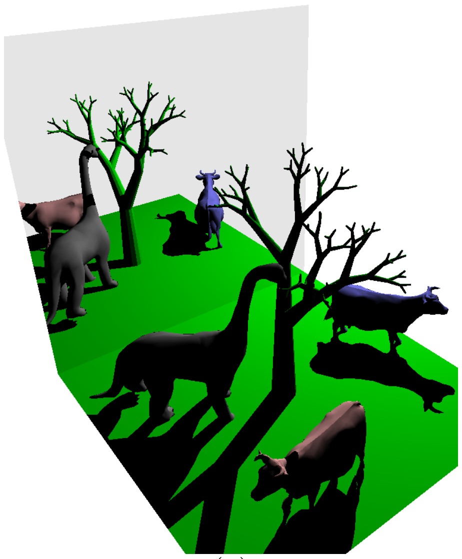
These days, ray tracing is entering the realm of interactive computer graphics, and so a representative benchmark scene should reflect the challenges of current rendering applications. A newer collection of scenes is “Benchmark for Animated Ray Tracing” (BART) [Lext et al.]. Since we only render static scenes, the animation features of these scenes are of little use. We found, however, that they reflect better the challenges faced by a modern real time rendering system, such as the “teapot in a stadium” problem, see section 10. We have included two of these scenes in ourbenchmark suite: Robots and kitchen.

In the following, we list all the scenes we have used to test our system.

Cows: 30226 triangles

Figure 24 (a) shows a scene we have constructed from meshes of varying origins. The tree is from the SPD and the cow and dinosaur meshes are from the public domain. The tree mesh combines large triangles at the bottom with very fine triangle patches in the small branches.

The top of the tree is expensive to trace rays through since a high percentage of rays graze areas of fine geometry. This stresses all the acceleration structures albeit for different reasons. The cows and the dinosaur contribute with additional local areas of high geometric density. The bottom and back planes are important since they allow us to test how well the different structures are able to benefit from complex geometry being occluded by simpler geometry. The back plane is the only specular surface in all our test scenes.


(a)

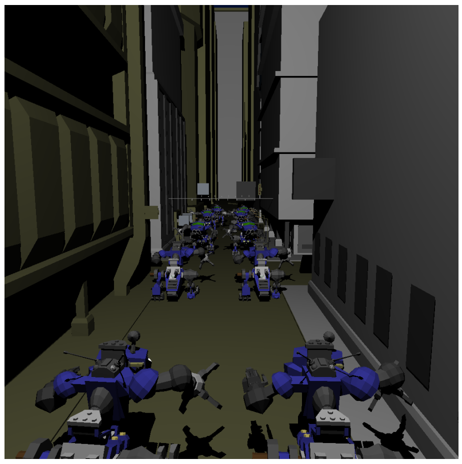

(b)

Figure 24: Cow (a) and robot (b) scenes.

Robots: 71708 triangles

Tim Foley was kind enough to send us this scene. It was used as a benchmark in a recent article [Foley & Sugerman]. We include it so that we may compare our results to theirs as accurately as possible. It is originally from the BART [Lext et al.] repository. This scene allows us to test how well our acceleration structures behaves in conditions where there are small, highly tesselated objects in a large scene. This is sometimes referred to as the “teapot in a stadium problem”. A rendering of the robot scene can be seen in figure 24 (b).

Kitchen: 110561 triangles

Another scene from BART that was supplied by Tim Foley. This scene (see figure 25 (a)) sports the highest number of triangles among all our scenes. As such it allows us to determine whether large scenes can be represented and efficiently accessed in GPU memory.

Bunny: 69451 triangles

This is the well-known Stanford bunny. It has been used in almost every ray tracing project for the last ten years. We include it here to test how our structures perform on a single, highly tessellated mesh – see figure 25 (b).

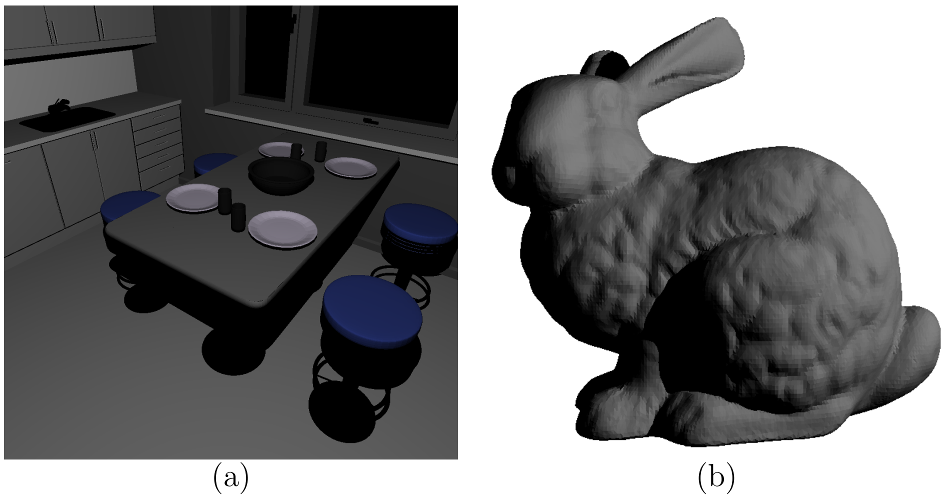


Figure 25: The kitchen scene and the Stanford bunny

Cornell Box: 32 triangles

This scene is another ray tracing classic. We use it to determine which structures gain most when the number of triangles is low. It is depicted in figure 26.

10.1 Fair comparisons

When doing a comparative study such as this one, it is of great importance to have strong arguments for why people should believe that the losers of the comparison have been treated fairly. This is the topic of the following subsections.

10.1.1 Fair traversal measurements

Our main experiments consist of the timing of each GPU acceleration structure rendering each scene. There are a few problems with this kind of experiments. We need to handle the fact that the CPU is occasionally disturbed by OS interrupts and so our measurements may become contaminated. We observe that the actual CPU time spent by our program on a single frame is always less than or equal to the real world timing we get as output. For this reason we alleviate the problem by repeating each experiment a number of times and keeping the fastest timing as our result.

Another potential problem is that a single snapshot of a scene might favor one accelerator over the others. Suppose we have a scene with a highly tessellated teapot on a table and we make the unlikely decision to place the

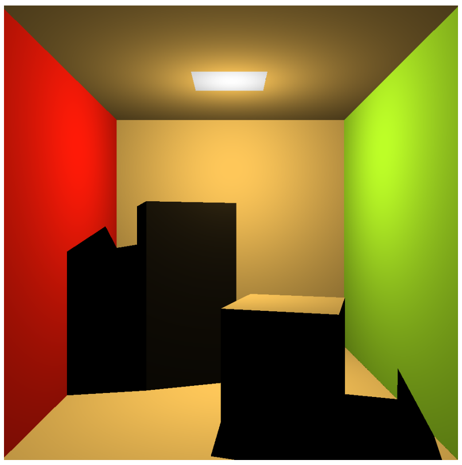


Figure 26: The cornell box. The lighting of the ceiling looks wrong because we use a point light suspended slightly below the white square.

camera under the table facing upwards. In this scenario the BVH might have the bad fortune of entering the subtree with the teapot geometry before having tested the table for intersection. In contrast both uniform grid and kd-tree accelerators will always test the table first. To compare the performance of the accelerators one should ideally take snapshots from all possible angles and positions and compare the average render time. Lacking the time to iterate over all angles we have come up with a camera path that alleviates the problem to some degree. We let the camera move along the curve shown in figure 27 with the scene being rendered in the center. The distance between the camera and the center of the scene is a function of the dimensions of the scene. We use this approach for the scenes cows and bunny.

The scenes robots, kitchen, and cornell are included to allow a comparison of our results to those produced by Foley and Sugerman. In these scenes, we only render a single frame as this was the approach taken by said authors. We have approximated their camera parameters to make the comparison as straightforward as possible when we render that same frame.

10.1.2 Fair construction

Our performance measurements are of questionable value, if we cannot say with reasonable certainty, that our three main competing structures are all built equally well. It is inherently difficult to say anything formal about the quality of the construction of an acceleration structure since the construction algorithms are based on heuristics. What we do instead is to try and build structures allowing as fast traversal as others have achieved before us.

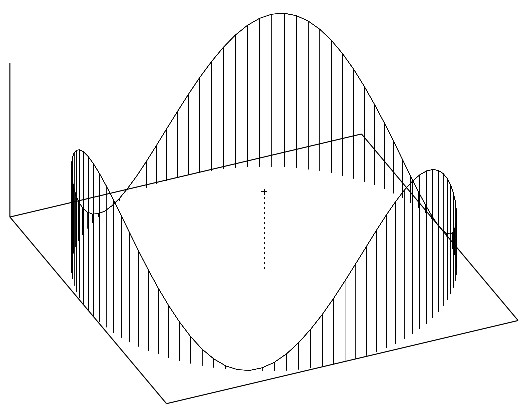


Figure 27: The camera curve $f(t) = (cos(t), sin(t), sin(3t))$. The dot in the middle marks the center of the scene.

We have used CPU implementations of the structure traversal for all our structures to help ensure the quality of their construction. We have prioritized efficient GPU traversals over their CPU counterparts. Consequently, we do not expect our CPU implementations to match cache optimized implementations from other research projects. Instead, we have aimed for the relative performance between the structures to match roughly what is reported in the literature.

Havran [Havran] and later Foley and Sugerman [Foley & Sugerman] state that uniform grid outperforms kd-trees on scenes with small variance in geometric density, such as the bunny scene. Conversely, on scenes displaying the “teapot in a stadium” effect, both agree that kd-tree is best. We have confirmed this relationship in our experiments. See section 11.1.6.

During development, we have observed BVHs consistently outperforming kd-trees on the CPU as well as on the GPU. This was surprising to us, because Havran ranked the BVH as the worst performing structure in his survey [Havran et al.]. The only two possible explanations to this conflict are that

A: Our kd-tree *and* uniform grid implementations are of poor quality, or that

B: Havran’s BVH implementation is flawed.

In an attempt to verify the quality of our kd-tree construction, we have compared the quality of our construction against that of Pharr and Humphreys on the benchmark scenes [Pharr & Humphreys]. We compared the two structures by converting the in-memory representation of their construction to the internal representation of our own kd-tree. This allowed us to use the same traversal algorithm on both, leaving the construction techniques to be the only difference. The results showed that our own construction is slightly better than theirs. One reason why our construction is better might be our use of split clipping (as described in section 8.1), which is absent from the kd-tree construction of Pharr and Humphreys.

The traversal speed of the BVH has exceeded our expectations by performing extremely well. For this reason, we are confident in the quality of construction algorithms.

The only heuristic involved in the construction of a uniform grid is the choice of grid resolution. It is a structure simple enough that it may be verified simply by inspection using a visual debugger.

10.2 Benchmark parameters

Our rendering system accepts a number of command line parameters. A comprehensive list with descriptions can be found in appendix A. In order to allow the interested reader the opportunity to verify our results, the parameters used in our experiments are listed in the following.

-r 2 Tells the system to do the rendering on the GPU.

-d *maximumdepth* Sets the maximum depth of the constructed kd-trees. This value was obtained by doing exhaustive search in the interval $[1; 40]$, or up to the memory limit of the graphics card. The depths used for restart and backtrack, respectively, in the benchmarks are as follows. Cows: 24 and 33, Robots: 30 and 30, Kitchen: 30 and 33, Bunny: 30 and 30, Cornell: 1 and 1 (effectively a list of triangles).

-e 2 Sets the minimum number of elements required for the kd-tree to make another internal node. Two elements gives deep trees, but short leaf lists. This value was found to be the best for all scenes by means of interval search.

-l 20 Sets the number of traversal/intersection steps to do in each fragment program invocation. Doing more than 20 has no effect and less than 20 usually degrades performance. There is an hardware limit on the number of executed instructions per pass, so higher values may cause strange behavior.

-n *angles* Sets the number of angles to take snapshots from. For the cows- and bunny scenes it is set to 100. For the rest it is set to 1.

-b *bounces* Sets the number of bounces to allow a ray to take. Shadow rays and primary rays are included in this number. Since the only scene with a specular surface is cows, this scene was rendered using 3 bounces. The rest only used 2 bounces (primary and shadow rays).

-g *resolutionmultiplier* Normally, the resolution of the uniform grid is a function of the number of triangles in the scene. This parameter is used to multiply that function value by a constant to increase or lower the overall resolution of the grid. We have found the following values to work best. Cows: 1.1, Robots: 1.4, Kitchen: 1.3, Bunny, 1.1, Cornell: 0.6

-w *resolution* Render the images in $resolution \times resolution$. We have used 512 by 512 as the size for our benchmark images.

As can be seen in the usage appendix, there are several other parameters available from the command line. In our experiments these were all left to their default values which are specified in appendix A.

We ran our tests on a system with the following specifications:

CPU: 3,2GHz Pentium 4 with HT technology

RAM: 1GB 400MHz DDR ram

GPU: Geforce 6800 Ultra at 400 MHz with 256MB DDR3 ram at 1100 MHz

Operating system: Windows XP Professional, service pack 2

GPU Driver: NVIDIA 77.62 (beta)

Cg compiler Version 1.3, used with the fp40 profile

The Geforce 6800 supports the fp40 fragment program profile. The program should run on any other card with full support for fp40. Note that the program only uses the graphics adapter, when run with the -r 2 parameter as specified above.

10.3 Summary

In this section we have presented arguments why our measurements are reasonably accurate. We also describe how all acceleration structures are subjected to both favorite and least favorite viewing angles on all our scenes.

We also present arguments why we believe that all our structures are constructed correctly.

Finally, we list the parameters with which our benchmarks are run. These, along with the description of our benchmarking system, should allow the interested reader to verify our results.

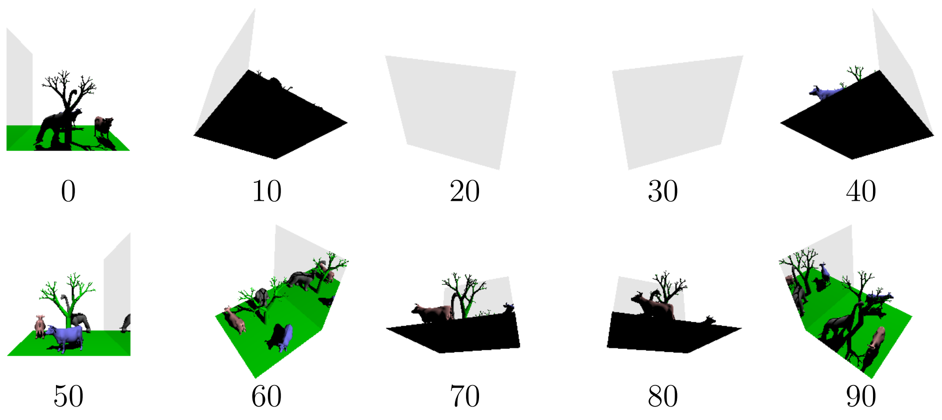


Figure 28: Frames 0, 10, ..., 90 from the measurements on the cow scene.

11 Results and Discussion

In this section we present the results of our experimental work. We first go through the scenes one by one emphasizing relevant aspects of the performance graphs. We include measurements of the memory consumption of the three acceleration structures. We relate these results to those of Foley and Sugerman. Finally, we iterate over the acceleration schemes, discussing the measurements and their significance.

11.1 The scenes

In the following, the results of our measurements are presented. We present them in visual form to allow easy comparisons between the structures. The results are presented as a table in section 11.1.6.

11.1.1 Cow scene

First thing to observe from the graph in figure 29 is that the time it takes to render this scene varies greatly with the camera angle and position. We may also observe that the curves have points of symmetry around frames 25 and 75, and that the curve wraps continuously from frame 99 to frame 0. These facts are due to the function describing the path of the camera (displayed in figure 27), combined with the partial symmetry of the scene.

Several features of the setup contribute to the shape of the curves. Most notably, frames 20, 30, and to some degree 75 show fast render times. In frames 20 and 30 the scene is occluded by just two triangles as can be seen in figure 28. Between the frames 70 and 80 the camera dips down allowing the ground plane to occlude everything but the top of the tree. There are

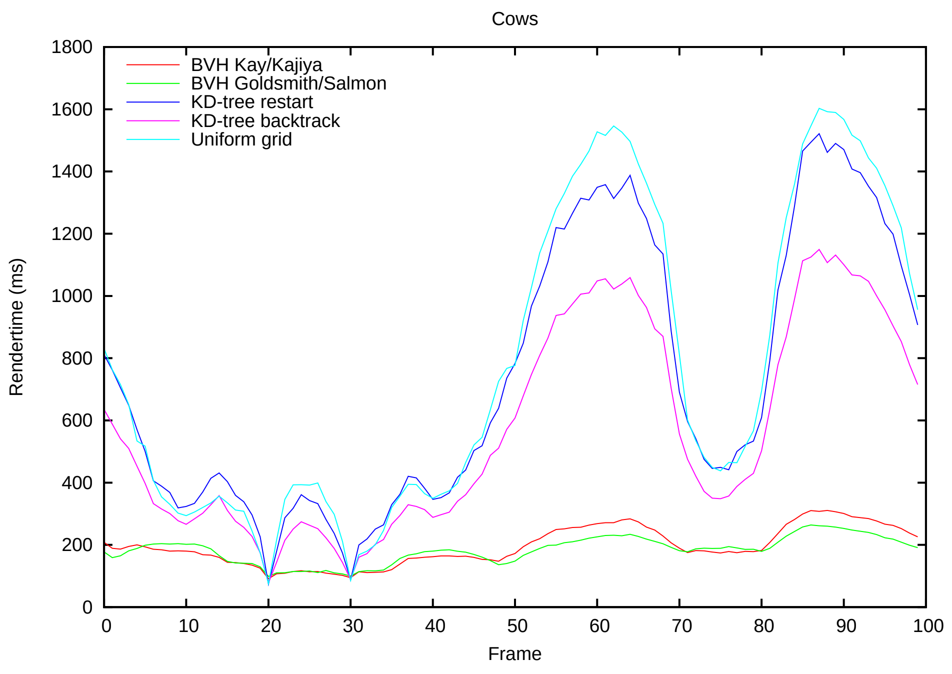


Figure 29: Timings for the cow scene.

two reasons why occlusion leads to such fast rendering. First of all, most of the acceleration structures guarantee a very low number of intersection tests in such scenarios. Secondly, all rays that hit the bounding box of the scene (thus leading to traversal of the structures) all hit one of the same two triangles. This means that nearly all concurrent fragment programs do the same intersection tests in the same order. As explained in section 3.7, this is the sort of behavior that leads to optimal utilization of the GPU.

It is also clear from the curves that the acceleration structures gaining most from the occlusion are the spatial ones. More on this in the discussion of the acceleration structures below.

All structures have a small peak around frame 25. In this frame, the camera comes up high enough for the top of the tree to show, adding greatly to the diversity of the paths of execution followed by concurrent fragment programs, consequently leading to slightly higher render times.

The peaks around frames 60 and 90 may again be explained by the snapshots in figure 28. As can be seen here, nearly all geometry is visible, and the most triangle rich parts of the scene are reflected in the mirror. This means that all three traversal stages (camera rays, reflection rays, and shadow rays) involve traversal through dense parts of the scene.

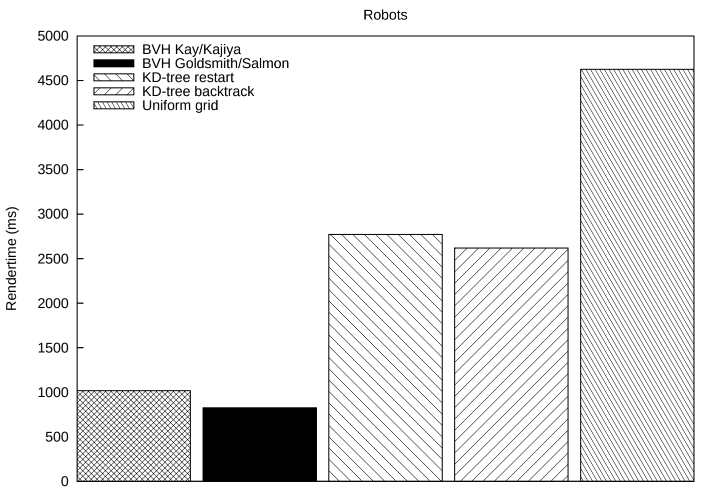


Figure 30: Timings for the robot scene. Unlike the scenes "Cows" and "Bunny" this measurement represents a single snapshot and not a whole range of camera angles (hence the bars instead of a graph).

11.1.2 Robots

This scene shares many of the characteristics of the cow scene. The large empty regions and local areas with high geometric detail make this scene give much the same relative results as the cow scene, as can be seen in figure 30. One difference is the size of the empty region down the center of the street. For structures unable to skip empty regions quickly, the large empty region makes rendering of this scene extra slow. There are also more rays grazing highly tessellated objects. This is expensive for all acceleration structures, and it is evident from the scale of the numbers that the rendering times are of a different magnitude than those from the cow scene measurements.

Another reason why this scene takes so long to render is that all camera rays hit the scene box as opposed to the cow scene setup where a large portion of rays miss the bounding box of the scene. This leads to a greater computational burden. Even if many rays hit the simple geometry on the sides of the buildings, they still serve to reduce control flow coherence, when they are processed along side rays hitting more complex surfaces.

11.1.3 Kitchen

The measurements in figure 31 show a similar relationship between the different accelerators. Note, however, that the magnitude of the rendering times has changed once again. In spite of this scene containing roughly 50%

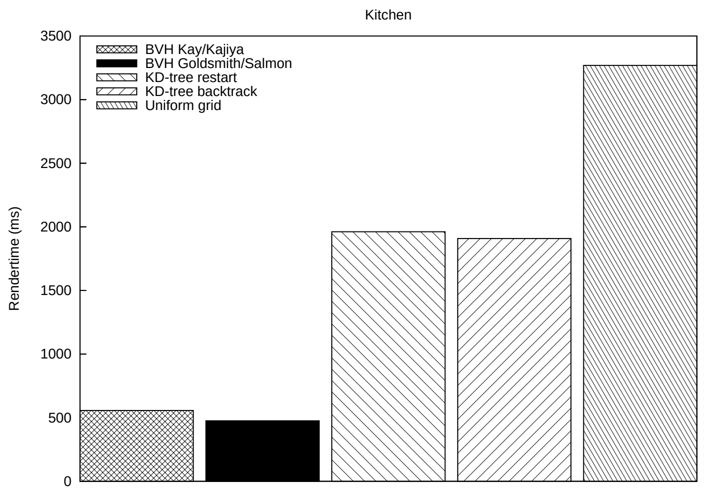


Figure 31: Timings for the kitchen scene.

more triangles than the robot scene, the render times average to approximately 2/3 of those from the robots scene. A possible reason is that a larger part of this scene is behind the camera than is the case for the robot scene.

11.1.4 Bunny

The Stanford bunny is rendered like the cow scene with a camera moving along the camera curve. All the renderings from the bunny scene can be seen on the front page. The timings are shown in figure 32. Rendering times are of the same magnitude as those in the cow scene. Timings vary as the camera position changes, but not nearly as much as in the cow scene. This is most likely due to the fraction of occluded triangles varying very little from frame to frame.

The fluctuations that remain are likely to be caused by variations in the number of primary rays and shadow rays hitting the bunny.

Because this scene is so different from the other scenes, it shows a different ordering of the accelerators, performance wise. In this scene, the uniform grid outperforms the two kd-tree variants.

11.1.5 Cornell box

As mentioned in section 10, this scene is extreme in its low number of triangles. As such, it tells us something about the time required to initialize the traversal of the different acceleration structures. This would account for the more even performance between the structures, as seen in figure 33.

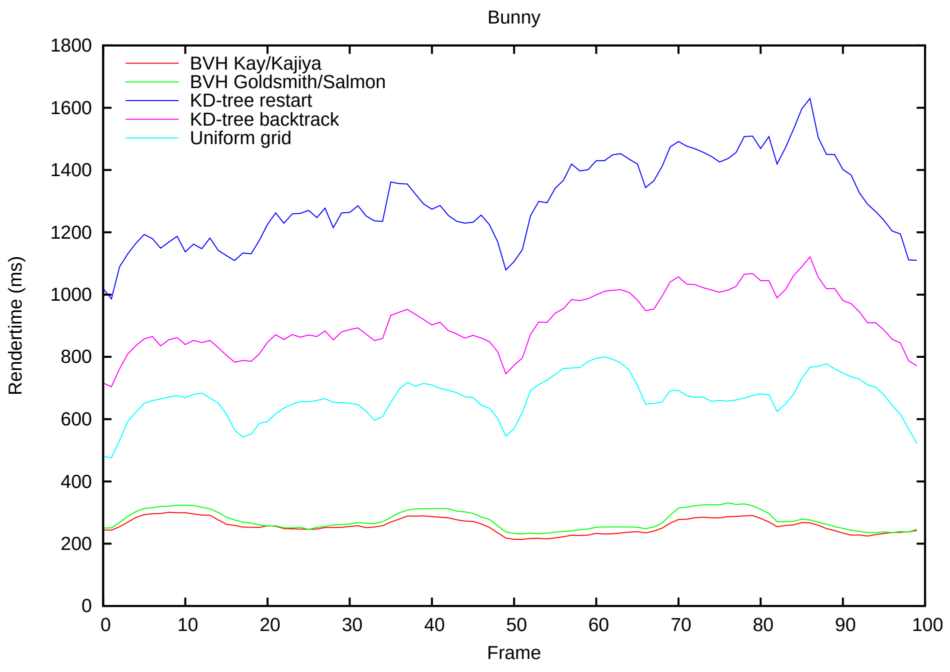


Figure 32: Timings for the bunny scene.

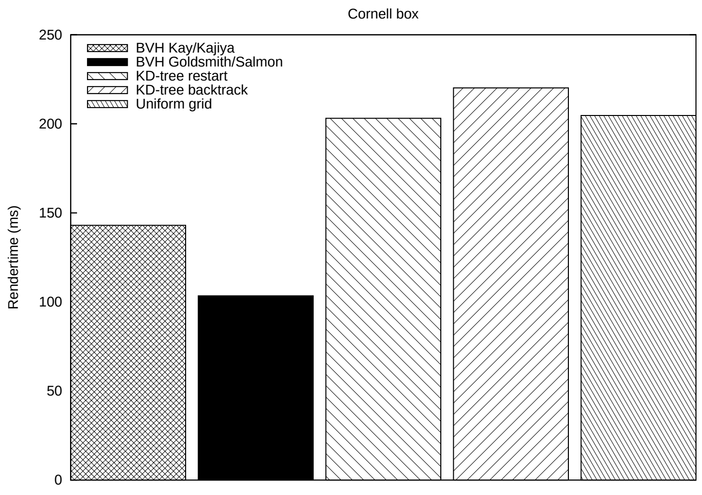


Figure 33: Timings for the Cornell box.

It is also interesting to note that the increase in triangle count from 32 in the cornell box to 30226 in the cow scene, leads only to a doubling of the rendering time. There are several reasons to this. Firstly, the rays that miss the cow scene are very cheap to process, whereas all primary rays hit the cornell box in our setup. Secondly, the hierarchical structures we use have been shown to provide traversal in logarithmic time [Glassner, p.206], [Havran]. Consequently, this scene illustrates the fact that ray tracing scales well in the number of triangles rendered. Especially when compared to rasterized graphics, where execution time is always linear in the number of triangles.

On a scene this small, it is likely, however, that rendering would be faster without the overhead imposed by acceleration structures.

11.1.6 Overview of results

To sum up the general picture from the graphs above, we have put the average timings of all accelerators rendering all scenes in table 1. All experiments were repeated 100 times, and the minimum rendering time achieved is listed in the table. For Cows and Bunny, the number is an average over all the angles, from which renderings were done.

	Cows	Robots	Kitchen	Bunny	Cornell
Uniform grid	770	4626	3269	667	205
Kd-tree (Restart)	728	2771	1961	1299	203
Kd-tree (Backtrack)	567	2619	1908	913	220
BVH (Kay/Kajiya)	195	1017	556	257	143
BVH (Goldsmith/Salmon)	183	825	476	275	103

Table 1: Average rendering times in milliseconds per frame, including shadow rays and reflection rays where applicable.

In presenting our results, we had a choice between two units: milliseconds per frame and rays per second (rps). We display the rps measurements from the cow scene in figure 34. When compared to the graph in figure 29, it is clear that they both give the same picture of the relative performance between the structures. This is not surprising, since the two measures are inversely proportional. For these reasons we deemed it redundant to display rps graphs for all our experiments.

11.2 Related results

In order to put our results into context, we will compare them to those reported by Foley and Sugerman in their recent article [Foley & Sugerman].

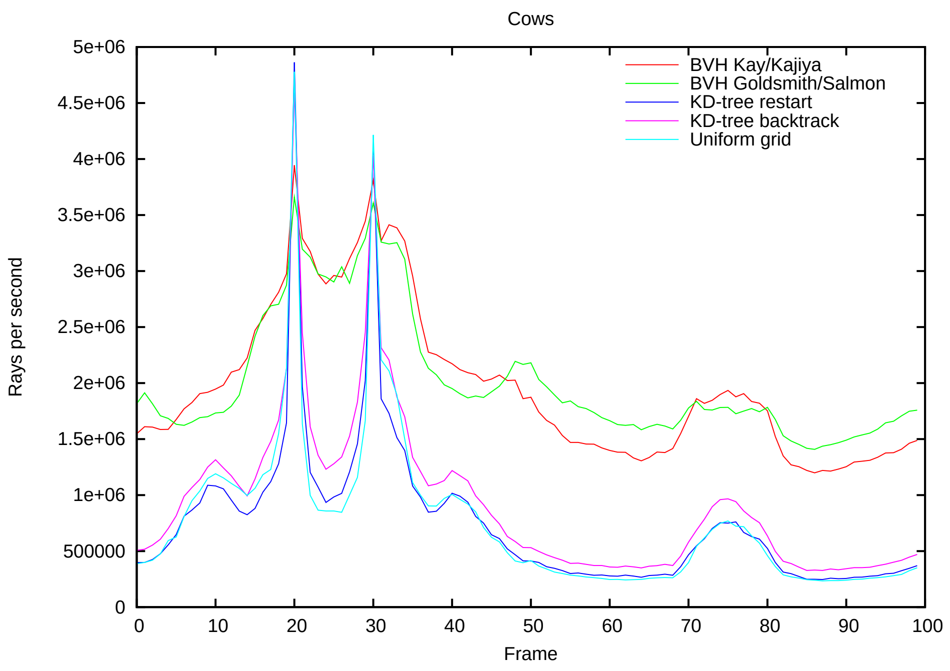


Figure 34: Rays per second measurements for the cow scene. Inversely proportional to figure 29.

As discussed in section 10, it is difficult to directly compare absolute results from other research projects to ours. Foley and Sugerman list rendering times for the tracing of their primary rays only. Since our timings account for ray generation, primary and secondary rays, and shading, the numbers cannot be compared directly. For the scenes robots and kitchen, however, all primary rays give rise to exactly one shadow ray, and so our timings are the results of roughly twice the amount of work as that measured in Foley and Sugerman's article. One might therefore compare their timings to ours divided by two[9]. We have halved some of our measurements and listed them alongside their measurements in table 2.

Conclusions from the table above should be made with care. It is a rough comparison, since its validity depends on a number of unconfirmed assumptions. In the following we list some observations.

BVH The difference between our BVH timings and the rest leave little doubt that it is the faster structure.

[9]Note that we do not treat shadow rays any differently from primary or reflection rays, although this is possible.

	Kitchen		Robots	
	Ours	Theirs	Ours	Theirs
BVH GS	238	-	413	-
UG	1635	2687	2313	8344
Restart	981	992	1386	968
Backtrack	954	857	1310	946

Table 2: A rough comparison of our results to Foley's and Sugerman's. Numbers indicate time spent tracing primary rays.

Uniform grid There appears to be a rather large difference between the performance of our uniform grid traversal and that reported by Foley and Sugerman. Part of the explanation could be that they use equal resolution along all three axes although the scenes are not of cubic shape. We use a resolution of $2\sqrt[3]{N}$ along the shortest axis, and choose resolutions along the other axes so as to get near cubic voxels. We have had the best results on the robot and kitchen scenes with grid resolutions of $116 \times 189 \times 242$ (-g 1.4) and $246 \times 123 \times 248$ (-g 1.3), respectively. See section 10.2 for an explanation of the -g parameter.

Kd-tree Our kd-tree traversals are beaten with a factor of 1.2 on average. There are many possible reasons for this. Having established that our kd-tree construction is of reasonable quality (see section 10.1.2), there is only the traversal to blame.

Foley's and Sugerman's article does not specify whether ray generation and shading is included in their measurements. If it is not, then this may account for part of the difference.

Foley and Sugerman decomposed their kd-tree traversal into six different kernels. This has the advantage that all kernels share the same path of execution. The disadvantage is that only the kernels that require the control flow represented by the active kernel actually make progress. This approach has been discussed in section 3.7.

Our implementation style relies heavily on the branching capabilities of the Geforce 6800 GPU. The numbers in the table indicate that traversal of the kd-tree is complex enough, that neighboring fragments rarely give rise to the same path of execution.

The question then is if our BVH traversal would also execute faster if split up in several kernels. Due to the simplicity of the BVH traversal kernel, we do not expect there to be much to gain by decomposing it, but this remains to be resolved.

It should perhaps also be taken into consideration that Foley's and Sugerman's experiments were done on an ATI GPU and ours on an NVIDIA GPU. Both are equally new, but there are numerous architectural differences. An example is the internal floating point precision, which is 24 bit on the ATI GPU and 32 bit on ours. This could account for some of the difference. Due to the hardware differences, there is no guarantee that splitting the kd-tree traversal kernel up would result in a speedup on our GPU, but we do consider it a possibility.

We would like to compare our implementation to Purcell's, but his experiments were done on a much slower GPU, and in a lower resolution, so direct comparisons with his results would carry little meaning.

11.3 Memory consumption

For each of the benchmark scenes, we have calculated the amount of memory consumed per triangle by each structure. The construction parameters used for the test are the same as for the benchmarks. The results are listed in table 3. As indicated in the table caption, the representation of the triangles themselves are not included, as it is constant. This means the numbers may be seen as a measure of the overhead required for the acceleration itself, by the different structures. As is evident from the numbers, there is great variation in the memory consumption with regards to both the structures and the scenes. These will be examined more closely in the discussion below.

	Cows	Robots	Kitchen	Bunny	Cornell
BVH (Goldsmith/Salmon)	4.1	3.6	4.1	4.6	5.0
BVH (Kay/Kajiya)	8.0	8.0	8.0	8.0	7.8
Uniform grid	24.1	154.0	140.0	38.6	6.1
Kd-tree (Restart)	66.0	109.2	68.5	109.4	2.5
Kd-tree (Backtrack)	371.4	192.8	158.5	197.1	3.3

Table 3: Memory consumption in floats per triangle. The triangles themselves are not included in the numbers.

In spite of the low memory consumption by the BVHs, all our traversal kernels are bandwidth limited. This can be deduced from the graphs in figure 35. The graph in (a), shows the decrease in performance measured while rendering the robot scene with the GPU memory operating at decreasing speeds. It is evident that reduction of bandwidth capacity has an impact on performance.

To produce the graph in (b) we repeated the process, with reduced GPU core frequency and memory operating at full speed. Note the scaling on the y-axis. Fluctuations are less than 0.5% even when the GPU is computing at 1/4 capacity. This method of identifying the bottleneck in GPU computations is suggested in the GPU programming guide [NVIDIA 2005].

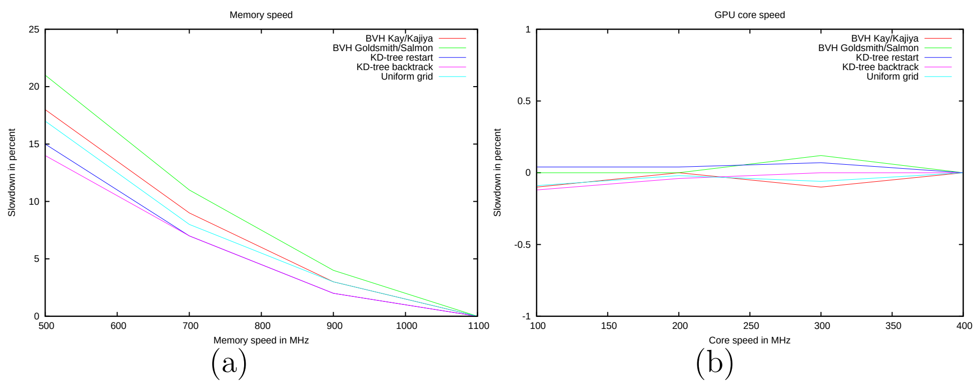


Figure 35: Graphs showing the results of down clocking the memory (a) and GPU core (b) frequencies.

11.4 Discussion of results

In the following, we discuss the results presented above. We present the discussion as an iteration over the acceleration structures.

11.4.1 Uniform grid

In table 1 it is evident that most of our benchmark scenes are the wrong kind of scenes to render using the uniform grid. The large, empty regions of both the robot and cow scenes are expensive for the uniform grid to traverse. The problem in the robot scene is that increasing the grid resolution helps reduce the number of rays grazing the detailed robots, but at the same time the large number of rays, that hit triangles in the opposite end of the street, must traverse a much higher number of empty voxels. Reducing the resolution gives faster traversal of empty space, while increasing the number of expensive rays that hit voxels with long triangle lists (often without hitting anything).

What the uniform grid lacks, is the ability to adapt to varying geometric density. This ability is crucial for the cow, robot, and kitchen scenes. For more uniformly distributed objects, the uniform grid performs reasonably well. It does not come near the performance of the BVHs, but it does beat the kd-tree on the bunny scene. Here, the resolution calculating heuristic is better able to choose a resolution that gives an even distribution of triangles in voxels. A certain degree of control flow coherence can also be accomplished, since neighboring rays generally enter the same voxels and traverse the same triangle lists on this type of scene. For more complex scenes this will not so often be the case.

Another important scene property is that of occlusion. In the cow scene, the uniform grid displays a drastic speedup in those frames, where most or all of the scene is hidden behind the mirror. In these frames, uniform grid is actually faster than the BVHs. Since this is the only situation, where uniform grid comes close to the kind of performance we see with BVH, it is worth considering what causes this major speedup.

One contributing factor is that uniform grid never continues traversal once the list holding the intersected triangle has been traversed. More importantly, the intersection with the mirror is always found in the very first voxel a ray hits and so no traversal is necessary when looking at the back of the mirror. All rays can stop after traversing the list of the first voxel. Since no voxel traversal and no more than two triangle intersections are necessary, it must be considered a very special case that allows the uniform grid to match the speed of BVH in this case.

A unique advantage of the uniform grid is its ability to start traversal from the position of the camera, completely disregarding everything behind the camera. Most of our tests were not designed to fully demonstrate this, but the relative performance improvement seen in the kitchen scene could indicate that this is an important factor.

In terms of memory consumption, the uniform grid fares reasonably well. The numbers for the robot and kitchen scenes do indicate a weakness, when it comes to complex scenes. The high grid resolution needed for optimal performance on these scenes requires a lot of memory.

Another weakness of the uniform grid traversal is the amount of information having to be stored after each rendering pass, so that we may resume traversal where we left off. Passing this state information to and from the fragment programs costs memory bandwidth, which has turned out to be the bottleneck of all our traversal kernels.

11.4.2 KD-tree

The ability to adapt to the scene geometry makes the kd-tree a good candidate for more complex scenes with higher variation in geometric density. For single-object scenes, it may not be beneficial to use the kd-tree because it has a tendency to subdivide the scene too finely for these types of scenes, resulting in deep trees. This problem falls under the general problem with kd-trees, which is that of choosing good parameters, given a scene. The maximum depth parameter has an especially strong impact on the traversal speed.

Another drawback of the kd-tree is that the recursive traversal generally preferred in CPU implementations cannot be ported to the GPU, as explained in the example in section 3.9. Asymptotically speaking, the kd-restart scheme is inferior to recursive traversal, and in contrast to the conclusions of Foley and Sugerman, the backtrack method yields the best results on all our scenes.

We established above that the restart and backtrack implementations of Foley and Sugerman are faster than ours. Since our implementations are clearly bandwidth limited, it is worth considering if our implementations might somehow be wasting bandwidth.

Consider two concurrently executing restart kernels, A and B. Say A needs to take a step down the tree in the next invocation, while B needs to resume iteration over a triangle list. As explained in section 3.7, A and B must both execute the same instructions, even though they perform entirely disjoint computations. For A to take a step down the tree, data must be read from the tree texture. For B to advance in its iteration over a triangle list, it must read the triangle data and do the intersection test. The end result is that both A and B must read data required for both traversal and intersection testing. We believe this problem to be the main reason behind the somewhat disappointing performance of our kd-tree traversal. We still find that it was a worthwhile experiment, doing the implementation in a single kernel. The multiple kernel approach employed by Foley and Sugerman wastes resources as well, and so it was not obvious in advance which approach would be better.

In choosing between kd-restart and kd-backtrack, we notice that kd-backtrack is always faster than kd-restart. Backtrack uses significantly more memory, however. The extreme memory consumption limits its practicality to relatively small scenes. We believe that the reason for the high memory consumption is that the kd-tree construction method has a tendency to subdivide the scene very finely near the scene geometry. This causes the trees to be very deep and hence use a lot of memory. Furthermore each triangle may be referenced many times which also increases the memory consumption. We have observed that while deep trees require a lot of memory, they also provide the fastest traversal. This is most likely due to the short triangle lists that result from fine subdivision. Short lists result in fewer kernels iterating over triangle lists at any given time. This will again result in better bandwidth utilization as explained above.

While generally faster than uniform grids, the kd-trees are still not able to keep up with the much faster BVHs.

11.4.3 Bounding volume hierarchy

For all the tested scenes, the two variants of BVHs clearly outperformed the other acceleration structures. The BVH constructed using the Goldsmith/Salmon approach generally outperform the binary Kay/Kajiya ones. Constructing a Kay/Kajiya BVH is many times faster than constructing a Goldsmith/Salmon BVH. Each of the 15 separate constructions we do, while attempting to find an efficient Goldsmith/Salmon tree, take longer than a single Kay/Kajiya construction. If the construction time is of importance, it is hardly worth the effort constructing Goldsmith/Salmon trees, when there is so little performance gained from them.

A number of factors contribute to the high performance of the BVHs. The traversal algorithm is extremely simple with minimal branching. Since there is only one triangle at every leaf of the tree, most rays are only tested for intersection against a few triangles. This comes at the cost of a much higher number of bounding box intersections, but since this means most fragment programs will be executing the same code, we get a high degree of control flow coherence. Additionally, many of the neighboring and thus concurrent traversal kernels will be reading from the same area in memory, thereby potentially[10] improving cache utilization.

When we are unlucky in our BVH traversal, we do not get to jump forward in the sequence at all. This unlikely worst case scenario is described in detail in section 9.4.3. The upside is that this leads to sequential texture reads, which are much faster due to texture caching on the GPU. In other words: The more texture reads we need in our traversal, the faster reads we get.

The fixed order traversal remains an issue with the sequential traversal, however. In the Goldsmith/Salmon construction, the order of a node's children is automatically randomized. In contrast, the Kay/Kajiya construction does not force any ordering of a node's children, so we can choose one as we see fit. Through this ordering, we may influence the performance impact of changing the viewing angle. In order to determine the impact of the fixed order traversal on performance, we have run tests in which we traverse left child first, right child first, and one in which we traverse a random child first at every node. The curves in figure 36 mostly show what one might expect: The two deterministic orderings of the children either yield near-optimal performance, or near-worst-case performance depending on the position of the camera. When the camera looks in the positive direction, along two or more axes, the left child first approach will do well, and the right child first ap-

[10]Without detailed knowledge of the internal workings of the texture cache it is difficult to say anything with certainty. Unfortunately, NVIDIA do not share this type of information in sufficient detail.

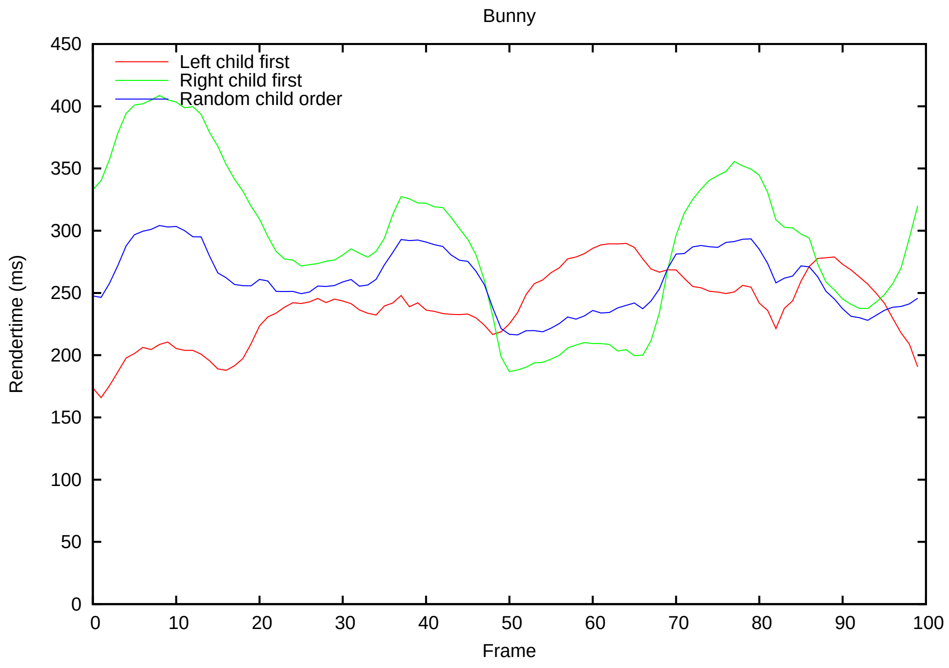


Figure 36: Kay/Kajiya BVHs constructed so that the left, right or a random child is visited first, on the bunny scene

proach not so well. It is not surprising that the randomized approach results in a less varying performance, which is roughly an average between those of the two deterministic tree constructions.

We also make an observation regarding the two different BVH construction approaches. The one constructed with the Goldsmith/Salmon method has a potentially large number of children at each node, and is thus a very different type of BVH than the strictly binary one you get from the Kay/Kajiya construction. Despite these differences, they both lead to roughly the same high performance. The only thing shared between the two types of BVH is our GPU traversal. This is a hint that the high performance of BVHs is not caused by the general properties of BVHs. We believe the efficiency of the GPU traversal makes most of the difference. The BVH traversal is bandwidth limited like the rest of the GPU traversal kernels. When it is still faster than the others, it is an indication that the BVH traversal either requires fewer texture accesses per ray, or utilizes the cache better.

Consider the problem discussed above regarding two concurrent kd-tree traversal kernels. In traversal of a BVH, each kernel invocation carries out the same three sequential texture reads whether it is going to traverse the tree or perform a ray/triangle intersection. This results in an improved bandwidth utilization, even when concurrently executed BVH traversal kernels are doing different computations.

The two BVH variants are also the most memory efficient, sometimes using as little as four floats per triangle to store acceleration information. This makes BVH a practical structure in situations with limited available memory. This is often the case for GPUs which have 256-512 megabytes for current generation consumer cards. While the storage required by the Goldsmith/Salmon trees varies with the different scenes, it is always bounded upwards by 8 floats per triangle, since all internal nodes require 8 floats and the maximum number of internal nodes is $n-1$, where n is the number of triangles. This limit is always reached by binary trees such as those constructed using the Kay/Kajiya method.

11.5 Summary

We have presented the results of our experimental work in the form of timing graphs for each of the benchmark scenes and a table summarizing the results. Furthermore, we have included measurements of memory consumption. These results are compared to those of Foley and Sugerman. Finally, our results are discussed in the context of each acceleration structure.

12 Conclusions

In this section, we sum up the main contributions of our thesis and discuss related future work.

12.1 Results

We have successfully implemented traversal of three acceleration structures for ray tracing on GPUs:

Uniform grid This first structure to be implemented for GPUs has turned out also to be the slowest structure, except for single-object type scenes. We have confirmed that the uniform grid is not suited for scenes with high variance in geometric density because it is incapable of adapting to such changes. Additionally, traversal on the GPU suffers from a relatively large amount of data required to represent the current state of the traversal. We conclude that while the uniform grid does allow traversal on the GPU, it yields comparatively low performance.

KD-tree Foley and Sugerman have already shown that on most scenes, the kd-tree is faster than uniform grid. We confirm their findings with our results. Both the restart and the backtrack variants of the kd-tree traversal suffer from a complex traversal algorithm in the context of fragment programs. The kd-tree makes up for this to some degree by its ability to adapt to changes in geometric density.

BVH Our BVH traversal scheme has proven to consistently outperform the two other schemes; sometimes by as much as a factor of nine. We believe our choice of benchmark scenes show sufficient variation to counter any claims that the scenes favor BVHs in some way. We conclude that in the current state of affairs, the BVH is the acceleration structure allowing the fastest traversal on the GPU. As an added bonus, implementation of BVH construction and traversal is simpler than for any other structure traversed on the GPU.

Since the BVH handles all scenes better than its alternatives, we consider a genuine contribution within the field GPU assisted ray tracing. The two main explanations to the speed achieved with our BVH traversal are:

Simplicity The BVH traversal is the simplest in two ways. Firstly, it takes only a hit record and an index to uniquely identify how far we are in the traversal. Secondly, the traversal code is extremely simple with only one branch and no costly branch dependent texture fetches.

Coherence The simplicity of the code causes many concurrently executed fragment programs to share the same execution path. Many of these adjacent kernels will also look up the same addresses in memory as they traverse the same parts of the BVH, which is good for cache utilization. Cache usage and low divergence in execution paths are two key elements to using GPUs with any measure of efficiency.

12.2 Future work

While we have not compared CPU implementations of the three structures on a scientific level, we have used CPU implementations as an aid in debugging the construction algorithms. Tests of these implementations indicate that BVH structures are also competitive on the CPU. This contrasts with Havran's findings, and it would therefore be interesting to implement optimized versions of the three structures and make proper comparisons on the CPU.

Our purpose with this thesis has been to improve the speed of intersection testing, and consequently we have put very little effort into shading. With our acceleration scheme, it would be feasible to use the GPU as an intersection test co-processor, where ray generation and shading is done on the CPU.

CPUs are designed to minimize the penalties of branching using branch prediction, but lack the floating point processing power of GPUs. GPUs handle branching much less gracefully due to their SIMD nature, but make up for this with their superior raw processing power. In applications like photon mapping and path tracing, it could prove beneficial to use the GPU for intersection testing and the CPU for everything else. This type of setup would allow both the CPU and the GPU to do what they do best in a concurrent manner.

Another way of expanding on our work would be to implement GPU traversal of some of the hybrid structures mentioned in section 5. We speculate that they would not allow sufficiently simple traversal to allow them to compete with the BVH, but the starting point in our work with the BVH was similar, as Purcell had named the uniform grid the most suitable structure for GPU traversal.

References

[Amanatides & Woo]	John Amanatides, Andrew Woo: A Fast Voxel Traversal Algorithm for Ray Tracing, Eurographics Conference Proceedings 1987
[Arvo]	Jim Arvo: Linear-Time Voxel Walking for Octrees, Ray Tracing News, Volume 1, Number 5, 1988 http://www.acm.org/tog/resources/RTNews/html/
[Buck et al.]	Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, Pat Hanrahan: Proceedings of ACM SIGGRAPH 2004
[Carr et al.]	Nathan A. Carr, Jesse D. Hall, John C. Hart: The Ray Engine, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2002, p. 37-46. ISBN 1-58113-580-7
[Cazals et al.]	Frédéric Cazals, George Drettakis, Claude Puech: Filtering Clustering and Hierarchy Construction: a New Solution for Ray-Tracing Complex Scenes, Computer graphics forum, Vol. 14, issue 3, p. 371, August 1995.
[Christen]	Martin Christen: Ray Tracing on GPU, Diploma thesis, 2005, University of Applied Sciences Basel, http://gpurt.sourceforge.net/DA07_0405_Ray_Tracing_on_GPU-1.0.5.pdf
[Ernst et al]	Manfred Ernst, Christian Vogelgsang, Günther Greiner: Stack Implementation on Programmable Graphics Hardware, Proceedings of Vision, Modeling, and Visualization, november 2004, p 255-262.
[Foley & Sugerman]	Tim Foley, Jeremy Sugerman: KD-Tree Acceleration Structures for a GPU Raytracer. (draft)
[Fujimoto et al.]	Akira Fujimoto, Takayuki Tanaka, Kansei Iwata: ARTS: Accelerated Ray Tracing System, IEEE Computer Graphics and Applications, Volume 6, Issue 4, 1986, p. 16-26

[Glassner]	Andrew S. Glassner: An introduction to Ray Tracing, 1989, Morgan Kaufmann Publishers, Inc. San Francisco, California. ISBN 0-12-286160-4.
[Glassner84]	Andrew S. Glassner: Space Subdivision for Fast Ray Tracing, IEEE Computer Graphics and Applications, 4(10): p. 15-22, 1984
[Goldsmith & Salmon]	Jeffrey Goldsmith, John Salmon: Automatic Creation of Object Hierarchies for Ray Tracing. IEEE Computer Graphics and Applications, Volume 7, Issue 5, 1987 p. 14-20, IEEE Computer Society Press, Los Alamitos, CA, USA.
[GPU Gems]	NVIDIA Corporation: GPU Gems, 2004, Addison-Wesley, ISBN 0-321-22832-4
[GPU Gems 2]	NVIDIA Corporation: GPU Gems 2, 2005, Addison-Wesley, ISBN 0-321-33559-7
[Haines]	Eric Haines: BSP Plane Cost Function Revisited, Ray Tracing News, Volume 17, Number 1, 2004, http://www.acm.org/tog/resources/RTNews/html/
[Haines87]	Eric Haines: A Proposal for Standard Graphics Environments, IEEE Computer Graphics & Applications, Vol. 7, No. 11, Nov. 1987, p. 3-5, IEEE Computer Society Press, Los Alamitos, CA, USA
[Havran]	Vlastimil Havran: Heuristic ray shooting algorithms, Ph. D. thesis, 2000 Czech Technical University, Faculty of Electrical Engineering, Department of Computer Science and Engineering
[Havran02]	Vlastimil Havran, Questions and Answers, Ray Tracing News, Volume 15, Number 1, 2002 http://www.acm.org/tog/resources/RTNews/html/
[Havran & Sixta]	Vlastimil Havran, Filip Sixta: Comparison of Hierarchical Grids, Ray Tracing News, Volume 12, Number 1, 1999, http://www.acm.org/tog/resources/RTNews/html/

[Havran et al.] Vlastimil Havran, Jan Přikryl, Werner Purgathofer: Statistical Comparison of Ray-Shooting Efficiency Schemes, Technical Report TR-186-2-00-14, Institute of Computer Graphics, Vienna University of Technology, 2000

[Jensen] Henrik Wann Jensen: Realistic Image Synthesis Using Photon Mapping, AK Peters, 2001, ISBN 1-56881-147-0.

[Kajiya] James T. Kajiya: The Rendering Equation, Proceedings of the 13th annual conference on Computer graphics and interactive techniques, p. 143-150, 1986

[Karlsson & Ljungstedt] Filip Karlsson, Carl Johan Ljungstedt: Ray tracing fully implemented on programmable graphics hardware, master's thesis, 2005, Chalmers University of Technology, http://www.ce.chalmers.se/edu/proj/raygpu/downloads/raygpu_thesis.pdf

[Kay & Kajiya] Timothy L. Kay, James T. Kajiya: Ray Tracing Complex Scenes, ACM Computer Graphics, Volume 20, Issue 4, p. 269-278, 1986, ACM Press, New York, USA

[Lext et al.] Jonas Lext, ulf Assarsson, Tomas Möller: BART: A Benchmark for Animated Ray Tracing, IEEE Computer Graphics and Applications, Vol. 21, Issue 2, March 2001, p. 22-31, IEEE Computer Society Press, Los Alamitos, CA, USA

[MacDonald & Booth] J. David MacDonald, Kellogg S. Booth: Heuristics for ray tracing using space subdivision. The Visual Computer, Volume 6, Issue 3, 1990, Springer-Verlag New York Inc. Secaucus, NJ, USA

[Möller & Trumbore] Tomas Möller, Ben Trumbore: Fast, minimum storage ray-triangle intersection, Journal of Graphics Tools, Volume 2, Issue 1, 1997, p. 21 - 28

[Müller & Fellner] Gordon Müller, Dieter W. Fellner: Hybrid Scene Structuring with Application to Ray Tracing, Proceedings of International Conference on Visual Computing (ICVC'99), pp. 19-26, Goa, India, Feb 1999.

[NVIDIA 2005] GPU Programming Guide, NVIDIA Corporation, 2005. http://developer.nvidia.com/object/gpu_programming_guide.html

[OpenRT] The OpenRT Real-Time Ray-Tracing Project, www.openrt.de.

[Pharr & Humphreys] Matt Pharr, Greg Humphreys: Physically based rendering, Morgan Kauffman, 2004, ISBN 012553180X

[Purcell] Timothy Purcell: Ray tracing on a stream processor, Ph. d. thesis, 2004

[Shirley & Morley] Peter Shirley, R. Keith Morley: Realistic Ray Tracing Second Edition, AK Peters, 2003, ISBN 1-56881-198-5.

[Sung] Kelvin Sung: A DDA Octree Traversal Alorithm for Ray Tracing, Proceedings of Eurographics 1991, p. 73-85

[Whitted] Turner Whitted: An Improved Illumination Model for Shaded Display, Communications of the ACM, 1980.

[Woo] Andrew Woo: Ray Tracing Polygons using Spatial Subdivision, Proceedings of the conference on Graphics interface 1992, p. 184-191

APPENDICES

A Usage

Listed below are the command line parameters of our program *tracy*. To obtain a copy of the binary program, contact one of the authors by email. In a command prompt, change to the directory holding the file `tracy.exe`. Now invoke the program with the desired combination of parameters and switches. In the following example the program is run in interactive mode with the CPU renderer using uniform grid with a slightly increased resolution on the default scene, Cows:

```
./tracy.exe -i -r 1 -a 4 -g 1.1
```

-r is used to choose between the OpenGL renderer (-r 0), the CPU renderer (-r 1), and the GPU renderer (-r 2). The default value is 2.

-a chooses the accelerator the renderer should use. Values have slightly differing meanings depending on the active renderer. For the OpenGL renderer this parameter is ignored. For the CPU- and GPU renderers the values and the corresponding accelerators can be seen in table 4. Default value is 0.

value	CPU renderer	GPU renderer
0	K/K BVH, sequential	K/K BVH
1	G/S BVH, sequential	G/S BVH
2	Kd-tree	Kd-tree restart
3	Kd-tree (pbrt)	Kd-tree backtrack
4	Uniform grid	Uniform grid
5	K/K BVH, recursive	N/A
6	G/S BVH, recursive	N/A
7	Octree	N/A

Table 4: Choosing accelerator for CPU and GPU renderers

-i is a switch used to activate interactive mode. This mode opens a window where camera and light positions may be manipulated. The interactive window has two modes that can be activated. Space bar activates the mode that shows the path of the camera in our benchmarks. 'r' enables rotation of the camera around the scene with the directional keys. Shift + up/down zooms in and out. 'd' dumps the current frame to the file dump.ppm.

-s selects the scene to render. The values accepted are: $\{0, 1, 2, 3, 4\}$. They select the scenes Cows, Robots, Kitchen, Bunny, Cornell in that order. Default is 0.

-w selects the image resolution. In interactive mode the window is always 512x512, but the image can be rendered at higher or lower resolutions and stretched/shrunk to fit the window. Values following w should be a power of two. Default value is 256.

-b sets the number of bounces to allow rays to take. Shadow rays and primary rays are included in this number. Note that only scenes with reflective surfaces will benefit from higher values than 2. Default is 3.

-p Switch for non-interactive mode only. Tells the program to dump all the rendered images for inspection.

-t Sets the number of times to construct new Goldsmith/Salmon BVHs before choosing the best one. Defaulting to 10.

-g Allows for tweaking of uniform grid resolution. Default value of 1.0 keeps the system's heuristically determined value. 2.0 doubles the resolution, 0.5 halves it etc.

-e Kd-tree construction parameter. Sets the required number of triangles for making an internal node. Default is 2.

-d Kd-tree construction parameter. Sets the maximum depth of the kd-tree.

-m Non-interactive mode only. Sets the number of times to repeat each measurement. The minimum of the resulting timings will be output. Default is 10.

-n Non-interactive mode only. Sets the number of angles to render from. Default is 10.

-l GPU renderer only. Sets the number of iterations to do on the GPU in each pass. Default is 20.

-c A parameter for Kay/Kajiya BVH construction. Determines the probability that the children of each internal node are swapped. Accepted values are 0,1,2. The probability is determined by multiplying the argument by 0.5. Default is 1.

B Cg code

This appendix contains all the Cg code used for acceleration structure traversal on the GPU.

B.1 Shared code

```
//used as a symbolic constant representing "infinity"
float INF() {return 100000000.0f;}

/** Translates a 1D-index to a 2D-position in a texture of width "width".
* @param index the 1D-index in the range [0, width*width - 1]
* @param width the pixel width of the lookup texture
* @return a 2D position in the range [0;1]x[0;1] */
float2 Get2dCoords(float index, float width)
{
   float delta = 1.0f / width;
   float2 lookup_const = float2(delta, delta * delta);
   //0.5 means we use the center of the pixel
   float2 coord = lookup_const * (index + 0.5f);
   return coord;
}

/** Uses the data texture as an array to look up a value
* @param the data texture that we should interpret as a 1D-array
* @param index the position in the array we want to find.
* @param width the width in pixels of the data texture.
* @return data[index]. */
float4 ReadIndex(sampler2D data, float index, float width)
{
   return tex2D(data, Get2dCoords(index, width));
}


/** Used for ray/triangle intersection testing.
* @param a first triangle vertex
* @param b second triangle vertex
* @param c third triangle vertex
* @param o is the ray origin
* @param d is the ray direction
* @param minT the intersection only counts if the t-value is greater
* @param mat_index the material index of the tested triangle.
* @param lasthit is the hitrecord for the previous hit where
* .x=u, .y=v, .z=t, .w=material index for best hit
* where u and v are two of the barycentric coordinates of the hit.
* @return a new hit record with the same format as the input.
* It is a new hit record if we found a hit, otherwise we return
the old one.*/
float4 Intersects(float3 a, float3 b, float3 c,  float3 o, float3 d,
                  float minT, float mat_index, float4 lasthit)
{
   // uses intersection test from
   //"a fast minimum-storage triangle intersection test"
   //lasthit contains information of the previous hit -
   float3 e1 = b - a;
   float3 e2 = c - a;
   float3 p = cross(d, e2);
   float det = dot(p, e1);
   bool isHit = det > 0.00001f; //the triangle is nearly edge-on
   float invdet = 1.0f / det;
   float3 tvec = o - a;
   float u = dot(p, tvec) * invdet;

   float3 q = cross(tvec, e1);
   float v = dot(q, d) * invdet;
   float t = dot(q, e2) * invdet;

   isHit = (u >= 0.0f) && (v >= 0.0f)
      && (u + v <= 1.0f)
      && (t >= 0.0f)
      && (t < lasthit.z)
      && (t > minT);

   return isHit ? float4(u, v, t, mat_index) : lasthit;
}
```

B.2 BVH

```
/** Checks for intersection between a ray and a box.
* @param box_min the minimum corner of the box.
* @param box_max the maximum corner of the box.
* @param o the origin of the ray
* @param d the direction of the ray
* @param bestHit the previously best known hit.
* @param tMin the hit only counts if the t-value is greater than tMin
* @return true if the ray intersects inside [tMin, bestHit.z] */
bool BoxIntersects(float3 box_min, float3 box_max, float3 o,
                    float3 d, float4 bestHit, float tMin) {
   float3 tmin, tmax;

   tmin = (box_min - o) / d;
   tmax = (box_max - o) / d;

   float3 real_min = min(tmin, tmax);
   float3 real_max = max(tmin, tmax);

   float minmax = min(min(real_max.x, real_max.y), real_max.z);
   float maxmin = max(max(real_min.x, real_min.y), real_min.z);

   bool res = minmax >= maxmin;
   return res && bestHit.z >= maxmin && tMin < minmax;
}



//this is the type for the output of our main program
struct fragment_out
{
   float4 bestHit      : COLOR0; //best hit so far
   float4 renderState   : COLOR1; //records how far we are
};



/** the main ray/scene intersection/traversal program (kernel).
* returns the best hit so far and the index to the next element */
fragment_out multipass_main
(uniform sampler2D geometry, //a texture representation of the BVH and the triangles
uniform sampler2D rayOrigin, //a list of ray origins
uniform sampler2D rayDirection, //a list of ray directions
uniform sampler2D bestHits, //the best hits found so far
uniform sampler2D renderStates, //records how far we are in the traversal
uniform float texWidth, //the width of the geometry texture
uniform float maxIndex, //maximum legal index in geometry texture
float2 streampos : TEXCOORD0, //texcoord to ray and best hit
uniform float looplimit) //maximum number of allowed loops
{
   float4 renderState   = tex2D(renderStates, streampos);
   float datapos         = renderState.x;

   if (datapos > maxIndex) discard;

   //find the ray and the previously best hit
   float3 origin      = tex2D(rayOrigin, streampos);
   float3 direction   = tex2D(rayDirection, streampos);
   float4 bestHit      = tex2D(bestHits, streampos);
   int loopcount = looplimit;

   while (loopcount > 0 && renderState.x < maxIndex)
   {
      //vertex A or box_min depending on the type of node
      float4 data1 = ReadIndex(geometry, datapos++, texWidth);
      //vertex B or box_max depending on the type of node
      float4 data2 = ReadIndex(geometry, datapos++, texWidth);
      //vertex C (we don't use it for bounding boxes)
      float4 data3 = ReadIndex(geometry, datapos++, texWidth);

      if (data1.w > 0)  // current element is a bbox
      {
         if (BoxIntersects(data1, data2, origin, direction, bestHit, 0.0f))
            renderState.x += 2; //we only needed two chunks of data
         else //data2.w points to the next piece of data in the traversal
            renderState.x = data2.w;
      }
      else //current element is a triangle
      {
         renderState.x += 3; //we needed 3 chunks of data.
         //data3.w contains the material index of this triangle
         bestHit = Intersects(data1.xyz, data2.xyz, data3.xyz,
                              origin.xyz, direction.xyz, 0.0, data3.w,
                              bestHit);
      }
      datapos = renderState.x;
      loopcount--;
   }

   fragment_out result;
   result.bestHit         = bestHit;
   result.renderState   = renderState;
   return result;
}


/** This kernel is used to set up early z-rejection and is run with
* occlusion querying so we know how many rays (pixels) still
* need processing. */
float cull_pass(   uniform sampler2D renderStates,
                  uniform float maxIndex,
                  float2 streampos : TEXCOORD0) : DEPTH
{
   return  tex2D(renderStates, streampos).x >= maxIndex ? 0.25f : 0.75f;
}


/** is used as output by the initialization kernel.
* Is mapped the same way as for the intersection kernel. */
struct setup_out
{
   float4 bestHit         : COLOR0;
   float4 renderState   : COLOR1;
};


/** Initialization kernel. Sets up empty intersection records and
* traversal positions start at index 0. */
setup_out rt_setup(float2 tc : TEXCOORD0,
               uniform sampler2D rayDirs)
{
   setup_out result;
   float finishedFlag = tex2D(rayDirs, tc).w;//finished if this flag is negative
   result.bestHit = float4(0, 0, INF(), 0); //"no hit"
   result.renderState = float4((finishedFlag < 0 ? INF() : 0.0f), 1.0f, 1.0f, 1.0f);
   return result;
}
```

B.3 Uniform Grid

```
/** is used to look up the contents of the grid cells. Maps 3D-coords
* to 2D textures.
* @param cellIndex the coordinates of the cell (in integers)
* @param texWidth the width of the 2D texture
* @param resolutions the size of the grid along the three axes
* @param grid the texture with the grid representation.
* @return a pointer to the list contained in this cell */
float GetTriangleListIndex(float3 cellIndex, float texWidth,
float3 resolutions, sampler2D grid)
{
   float gridWidth = floor(texWidth / resolutions.x);//number of layers in a row

   float2 coords;
   coords.y = floor(cellIndex.z / gridWidth);
   coords.x = fmod(cellIndex.z, gridWidth);
   coords *= resolutions.xy;
   coords += cellIndex.xy;
   coords /= texWidth;
   coords += float2(0.5f / texWidth);

   return tex2D(grid, coords).a;
}



/** Checks for intersection between a ray and a box.
* @param box_min the minimum corner of the box.
* @param box_max the maximum corner of the box.
* @param o the origin of the ray
* @param d the direction of the ray
* @param t_hit returns the t-value where the ray enters the box
* @return true if the ray intersects the box */
bool BoxIntersects(float3 box_min, float3 box_max, float3 o, float3 d, out float t_hit) {
   float t0 = 0.0f;
   float t1 = INF();

   float3 tmin, tmax;

   tmin = (box_min - o) / d;
   tmax = (box_max - o) / d;

   float3 real_min = min(tmin, tmax);
   float3 real_max = max(tmin, tmax);

   float minmax = min(min(real_max.x, real_max.y), real_max.z);
   float maxmin = max(max(real_min.x, real_min.y), real_min.z);
   t_hit = maxmin;
   return minmax >= maxmin;
}


/** calculates the cell index of a point p which is assumed to be inside the grid.
* @param p a point such that the return value is the voxel that containes it.
* @param sceneMin the minimum corner of the scene box
* @param resolutions the grid resolution along the three axes
* @param cell_width the width of a single cell along the three axes
* @return the index of the voxel that contains p. */
float3 GetCellIndicesOf(float3 p, float3 sceneMin, float3 resolutions, float3 cell_width)
{
   return clamp(floor((p - sceneMin) / cell_width) ,
     float3(0.0f,0.0f,0.0f),
     resolutions - float3(1.0f, 1.0f, 1.0f));
}



/** calculates the bounding box for a given cell
* @param scene_min the minimum corner of the scene box
* @param cell_width the width of a single cell along the three axes
* @param indices the index of the cell in question
* @param cell_min the minimum corner of the bbox is returned here
* @param cell_max the maximum corner of the bbox is returned here */
void GetCellBounds(float3 scene_min, float3 cell_width, float3 indices,
                   out float3 cell_min, out float3 cell_max)
{
   cell_min = scene_min + cell_width * indices;
   cell_max = cell_min + cell_width;
}


/** return value type of the initialization kernel*/
struct initializer_out
{
   //used for dda traversal.
   //xyz are the t-values
   //w is the triangle list index we are at in the current list
   float4 tMax         : COLOR0;

   //current cell index
   //xyz are the indices into the grid
   //w is the render state: 1 means finished, 0 means not finished
   float4 cellIndex   : COLOR1;

   //best hit so far
   //.x = barycentric u
   //.y = barycentric v
   //.z = t value such that (hitpoint = ray.origin + t * ray.direction)
   //.w = material index for the hit triangle (used for shading)
   float4 bestHit      : COLOR2; //best hit so far
};


/** Kernel used to initialize traversal.
* returns the initial state for the given ray. */
initializer_out initializer
(uniform sampler2D rayOrigin, //the texture of ray origins
uniform sampler2D rayDirection, //the texture of ray directions
uniform sampler2D grid, //the 2d-texture representation of the grid
uniform float3 sceneMin, //the minimum corner of the scene box
uniform float3 sceneMax, //the maximum corner of the scene box
uniform float3 resolutions, //the grid resolutions along the three axes
uniform float3 cell_width, // the xyz width of a single voxel
uniform float grid_width, //the width in pixels of the grid texture
float3 tc: TEXCOORD0) //texcoord indicating which ray we look up
{
   initializer_out o;

   //set initial best hit
   o.bestHit      = float4(0.0f, 0.0f, INF(), 0.0f);

   //look up ray
   float3 orig   = tex2D(rayOrigin, tc.xy).xyz;
   float3 dir   = tex2D(rayDirection, tc.xy).xyz;

   //set renderState to finished or traversing
   float3 gridOrig = orig;
   float t;
   if(!BoxIntersects(sceneMin, sceneMax, orig, dir, t)) {
      o.cellIndex.w = true;//isfinished because we missed the scene
   }
   else
   {
      gridOrig = t > 0.0f ? orig + dir * t : orig;
      //set start cell index
      o.cellIndex.xyz = GetCellIndicesOf(gridOrig, sceneMin, resolutions, cell_width);
      //find the index of the first cell
      float list_index = GetTriangleListIndex(o.cellIndex, grid_width, resolutions, grid);

      o.cellIndex.w = false;
      o.tMax.w = list_index;
   }

   //set initial tMax values
   float3 cell_min;
   float3 cell_max;
   GetCellBounds(sceneMin, cell_width, o.cellIndex, cell_min, cell_max);
   float3 t1s = (cell_min - orig) / dir;
   float3 t2s = (cell_max - orig) / dir;
   o.tMax.xyz = max(t1s, t2s);
   o.tMax.xyz = max(o.tMax.xyz, float3(0, 0, 0));

   return o;
}


/** see initialization documentation - this structure is identical */
struct UGOut
```

```
{
   float4 tMax : COLOR0;
   float4 cellIndex : COLOR1;
   float4 bestHit : COLOR2;
};



/** this is the main intersection/traversal kernel.
* returns a new best hit and updated traversal state.*/
UGOut traversalAndIntersection
(float2 tc : TEXCOORD0, // texcoord pointing to the ray and state
uniform sampler2D rayDirections, // the texture of ray directions
uniform sampler2D rayOrigins, //the texture of ray origins
uniform sampler2D grid, // the texture representation of the 3d grid
uniform sampler2D lists, //the texture containing the triangle lists
uniform sampler2D triangles, //the actual triangle vertices
uniform sampler2D tMaxes, // the t-values used in the dda traversal
uniform sampler2D cellIndices, //the index to the current voxel
uniform sampler2D bestHits, //the best hit so far
uniform float grid_width, //width in pixels of the grid texture
uniform float lists_width, //width in pixels of the list texture
uniform float triangles_width, // width in pixels of the triangle texture
uniform float3 resolutions, //the xyz resolution of the grid
uniform float3 cell_dims, //the size of each individual voxel
uniform float looplimit) //maximum number of allowed iterations
{
   UGOut OUT;

   OUT.cellIndex = 0.0f;
   OUT.tMax = 0.0f;
   OUT.bestHit = 0.0f;
   OUT.cellIndex      = tex2D(cellIndices, tc);

   bool isFinished = OUT.cellIndex.w;

   //this may happen if z-culling does not work.
   if(isFinished) discard;

   OUT.tMax = tex2D(tMaxes, tc);
   //loook up ray
   float3 dir = tex2D(rayDirections, tc).xyz;
   float3 orig = tex2D(rayOrigins, tc).xyz;
   //calculate variables for dda traversal
   float3 steps = sign(dir);
   float3 delta = abs(cell_dims / dir);
   float3 tMax = OUT.tMax.xyz;
   float3 cellIndex = OUT.cellIndex.xyz;
   float4 bestHit = tex2D(bestHits, tc);
   int loopsleft = looplimit;
   float list_index = OUT.tMax.w;

   while (!isFinished && loopsleft > 0)
   {
      //find the current index into the triangle list we are processing
      float tri_index  = ReadIndex(lists, list_index, lists_width).a;

      //iterate through triangles and test for intersections
      while (tri_index >= 0.0f && list_index >= 0.0f && loopsleft > 0)
      {
         float3 a      = ReadIndex(triangles, tri_index, triangles_width);
         float3 b      = ReadIndex(triangles, tri_index + 1, triangles_width);
         float4 c      = ReadIndex(triangles, tri_index + 2, triangles_width);
         bestHit = Intersects(a, b, c.xyz, orig, dir, 0.0f, c.w, bestHit);

         list_index += 1.0f; //move to next triangle
         tri_index = ReadIndex(lists, list_index, lists_width).a; //pointer deref
         loopsleft--;
      }
      if(loopsleft > 0) {
         if (bestHit.z < INF()) //we have a hit
         {
            float tMin = min(tMax.x, min(tMax.y, tMax.z));
            //if hit was inside voxel we are done
            isFinished = bestHit.z < tMin ? true : false;
         }

         //clever way of implementing the dda step from
         //Amanatides and Woo's article
         float m = min(tMax.x, min(tMax.y, tMax.z));
         //mask will be 1 where t is least, for example (0, 1, 0)
         float3 mask = float3(m, m, m) == tMax;

         //move to the next cell in the direction of least t-value
         cellIndex += steps * mask;

         //update the t value
         tMax   += delta * mask;

         //test for "beyond grid" and terminate if so
         float3 lt = cellIndex >= resolutions;
         float3 gt = cellIndex < float3(0.0f, 0.0f, 0.0f);
         if (any(lt) || any(gt))
            isFinished = true;

         //look up the list from the new voxel
         list_index = GetTriangleListIndex(cellIndex, grid_width, resolutions, grid);
         loopsleft--;
      }
   }

   OUT.tMax.xyz = tMax;
   OUT.tMax.w = list_index;
   OUT.cellIndex.xyz = cellIndex;
   OUT.cellIndex.w = isFinished;
   OUT.bestHit = bestHit;

   return OUT;
}



/** this kernel is used to set up early z-rejection and is run with
* occlusion querying on so we know how many rays (pixels) still
* need processing.
* @param tc indicates which renderstate we should look up.
* @param cellIndices contains part of the traversal state. the W
* component stores "isFinished"
* @param dirs the ray directions from the shader. if .w is -1
* then this ray need no processing
* @return a depth value that  culls this pixel if needed*/
float cull(   float2 tc : TEXCOORD0,
            uniform sampler2D cellIndices,
            uniform sampler2D dirs) : DEPTH
```

```
{
   float4 cellIndex = tex2D(cellIndices, tc);
   bool shaderFinished = tex2D(dirs, tc).w == -1.0f;
   //remember that cellIndex.w := isFinished
   return (shaderFinished || cellIndex.w) ? 0.25f : 0.75f;
}
```

B.4 KD-restart

```
/** Checks for intersection between a ray and a box.
* @param box_min the minimum corner of the box.
* @param box_max the maximum corner of the box.
* @param o the origin of the ray
* @param d the direction of the ray
* @param t_enter is used to return the t-value where the ray enters the box
* @param t_exit is used to return the t-value where the ray exits the box
* @return true if the ray intersects the box*/
bool BoxIntersects(float3 box_min, float3 box_max, float3 o, float3 d,
                     out float t_enter, out float t_exit) {
   float t0 = 0.0f;
   float t1 = INF();

   float3 tmin, tmax;

   tmin = (box_min - o) / d;
   tmax = (box_max - o) / d;

   float3 real_min = min(tmin, tmax);
   float3 real_max = max(tmin, tmax);

   float minmax = min(min(real_max.x, real_max.y), real_max.z);
   float maxmin = max(max(real_min.x, real_min.y), real_min.z);
   t_enter         = maxmin;
   t_exit         = minmax;
   return minmax >= maxmin;
}



struct main_out
{
   float4 bestHit : COLOR0;
   float4 renderState  : COLOR1;
};

/** The main kd-restart traversal/intersection kernel
* returns a new traversalState and a new bestHit*/
main_out kd_main
(uniform sampler2D rayOrigins, //the texture of ray origins
uniform sampler2D rayDirections, //the texture of ray directions
uniform sampler2D bestHits, //the best hit so far
uniform sampler2D traversalStates, //traversal state
uniform sampler2D traversalStates2, //more traversal state
uniform sampler2D kdtree, //a texture representation of the kd-tree
uniform sampler2D triLists, //a texture with  the triangle lists
uniform sampler2D triangles, //the actual triangle vertices
uniform float kdtreeWidth, //width in pixels of the kdtree texture
uniform float listsWidth, //width in pixels of the list texture
uniform float trianglesWidth, //width in pixels of the triangle texture
float2 tc : TEXCOORD0, //texture coordinate indicating which ray we look up
```

```
uniform float loopLimitConst) //maximum number of loops allowed
{
   //find the ray and the previous best hit
   float4 bestHit = tex2D(bestHits, tc);
   float3 orig = tex2D(rayOrigins, tc).xyz;
   float4 dir = tex2D(rayDirections, tc);
   float4 traversalState = tex2D(traversalStates, tc);
   float traversalState2 = tex2D(traversalStates2, tc).x;
   //traversalState.x = down: -1, up: -2: [0;INF] traversing triangle list
   //traversalState.y = tree pointer
   //traversalState.z = t_min
   //traversalState.w = t_max
   //traversalState2.x = global_t_max

   int kdtreeIndex      = traversalState.y;
   float t_min            = traversalState.z;
   float t_max         = traversalState.w;
   float global_t_max   = traversalState2;
   float listIndex;

   bool isFinished = false;
   float loopLimit = loopLimitConst;


   while(!isFinished && loopLimit > 0) {
      float4 node_info      = ReadIndex(kdtree, kdtreeIndex++, kdtreeWidth);
      //node_info.x = right child pointer
      //node_info.y = split value along split axis
      //node_info.z = parent pointer (not used in restart)
      //node_info.w = split axis if in {-2,-3,-4}
      //node_info.w = empty leaf if == -1
      //node_info.w = leaf with list pointer if >= 0

      if(node_info.w < -1) {
         //this is an internal node -> traverse downwards a step
         float3 splits = (float3(node_info.y) - orig) / dir.xyz;
         float t_split = node_info.w == -2 ?
    splits.x :
    node_info.w == -3 ? splits.y : splits.z;

         float3 dirSigns = sign(dir);
         float dirSign = node_info.w == -2 ?
            dirSigns.x :
            node_info.w == -3 ? dirSigns.y : dirSigns.z;

         //determine which child is nearest
         float first = dirSign == 1 ? kdtreeIndex : node_info.x;
         float second = dirSign == 1 ? node_info.x : kdtreeIndex;

         if(t_split >= t_max || t_split < 0.0f) {//ray only passes through first
            if (t_split < 0.0)   kdtreeIndex = second;
            else                 kdtreeIndex = first;
         } else if(t_split <= t_min) {//ray only passes through second
            kdtreeIndex = second;
         } else {//ray passes through first - then second. Go through the first now.
            kdtreeIndex = first;
            t_max = t_split;
         }
      } else {
         //this is a leaf - lookup list index and traverse it
         //Are we just starting list traversal at this node then start at the beginning,
         //else start where we got to
         listIndex = traversalState.x == -1 ? node_info.w : traversalState.x;

         float triIndex = ReadIndex(triLists, listIndex, listsWidth).a;
         triIndex = listIndex == -1.0f ? -1.0f: triIndex;//-1.0 indicates an empty leaf

         //traverse triangle list
         while(triIndex >= 0.0f && loopLimit > 0) {
            float3 a = ReadIndex(triangles, triIndex, trianglesWidth).xyz;
            float3 b = ReadIndex(triangles, triIndex + 1, trianglesWidth).xyz;
            float4 c = ReadIndex(triangles, triIndex + 2, trianglesWidth);

            bestHit = Intersects(a, b, c.xyz, orig, dir, 0.0f, c.w, bestHit);

            listIndex++;
            triIndex = ReadIndex(triLists, listIndex, listsWidth).a;
            loopLimit--;
         }
         traversalState.x = listIndex; //remember the list position we got to
         if(triIndex < 0.0f) { //Did we traverse until the terminator?
            //if a hit is found -> return if it is inside this voxel
            isFinished = bestHit.z <= t_max ? true : false;
            if(t_max == global_t_max)
               isFinished = true; //ray missed all triangles in the scene
            t_min = t_max;
            t_max = global_t_max;
            kdtreeIndex = 0;//start at root again since this is the 'restart' version
            //we are now traversing down the tree again (not running through a list)
            traversalState.x = -1;
         } else {
            //must point to the current leaf node if we are to resume list traversal
            kdtreeIndex-=1.0f;
         }
      }
      loopLimit--;
   }

   traversalState.y = isFinished ? -1 : kdtreeIndex;
   traversalState.z = t_min;
   traversalState.w = t_max;

   main_out OUT;
   //set output variables
   OUT.renderState = traversalState;
   OUT.bestHit = bestHit;
   return OUT;
}


/** output type for the initialization kernel */
struct initializer_out
{
   //best hit so far
   //.x = barycentric u
   //.y = barycentric v
   //.z = t value such that (hitpoint = ray.origin + t * ray.direction)
   //.w = material index for the hit triangle (used for shading)
   float4 bestHit : COLOR0;

   //state used during traversal
   //.x :-1 = traversing down,
   //      [0; inf] = traversing triangle list at position .x
   //.y: pointer to the node we have reached or -1 if finished
   //.z: t_min
   //.w: t_max
   float4 traversalStates : COLOR1;

   //state used during traversal (but only generated here)
   //.x = global t_max
   //yzw not used
   float4 traversalStates2 : COLOR2;

   //used for culling pixels that don't need processing
   float depth : DEPTH;
};


/** Initialization kernel used for both restart and backtrack
* @param origins the ray origins
* @param dirs the ray directions
* @param sceneMin the minimum corner of the scene bbox
* @param sceneMax the maximum corner of the scene bbox
* @param tc the texture coordinate of the ray
* @return state pointing traverser to root node */
initializer_out initializer(uniform sampler2D origs,
                          uniform sampler2D dirs,
                          uniform float3 sceneMin,
                          uniform float3 sceneMax,
                          float2 tc : TEXCOORD0)
{
   //find the ray
   float3 orig = tex2D(origs, tc).xyz;
   float4 dir= tex2D(dirs, tc);
   initializer_out OUT;

   bool hit = BoxIntersects(sceneMin, sceneMax, orig, dir.xyz,
                              OUT.traversalStates.z, OUT.traversalStates.w);
   OUT.depth = hit && (dir.w > -1.0f) ? 0.55f : 0.25f;
   OUT.bestHit = float4(0.0f, 0.0f, INF(), 0.0f); //"no hit yet"
   OUT.traversalStates.xy = float2(-1.0f, hit ? 0.0f : -1.0f);
   OUT.traversalStates2 = float4(OUT.traversalStates.w, 0.0f, 0.0f, 0.0f);
   return OUT;
}

/** used for early z-rejection and occlusion querying
* @param tc the texture coordinate of the ray we are processing
* @param traversalState the state of the traversal through the tree
* @param dirs the ray directions. .w = -1 if ray is finished
* @return a z-value that possibly culls this fragment in future runs */
float cull(   float2 tc : TEXCOORD0,
            uniform sampler2D traversalState,
            uniform sampler2D dirs) : DEPTH
{
   float finishedIfNegative = tex2D(traversalState, tc).y;
   bool shaderFinished = tex2D(dirs, tc).w == -1.0f;

   return (finishedIfNegative < 0) || shaderFinished ? 0.25f : 0.75f;
}
```

B.5 KD-backtrack

```
/** Checks for intersection between a ray and a box.
```

```
* @param box_min the minimum corner of the box.
* @param box_max the maximum corner of the box.
* @param o the origin of the ray
* @param d the direction of the ray
* @param t_enter is used to reject certain intersections.
* @param t_exit is used to return the t-value where the ray exits the box
* @return true if the ray intersects the box*/
bool BoxIntersects(float3 box_min, float3 box_max, float3 o, float3 d,
                   in float t_enter, inout float t_exit) {
   float t0 = 0.0f;
   float t1 = INF();

   float3 tmin, tmax;

   tmin = (box_min - o) / d;
   tmax = (box_max - o) / d;

   float3 real_min = min(tmin, tmax);
   float3 real_max = max(tmin, tmax);

   float minmax = min(min(real_max.x, real_max.y), real_max.z);
   float maxmin = max(max(real_min.x, real_min.y), real_min.z);
   bool res = minmax >= maxmin && maxmin <= t_exit && t_enter < minmax;
   t_exit         = res ? minmax : t_exit;
   return res;
}


struct main_out
{
   float4 bestHit : COLOR0;
   float4 renderState  : COLOR1;
};


/** The main kd-backtrack traversal/intersection kernel.
* returns a new traversalState and a new bestHit*/
main_out kd_main
(uniform sampler2D rayOrigins, //the texture of ray origins
uniform sampler2D rayDirections, //the texture of ray directions
uniform sampler2D bestHits, //the best hit so far
uniform sampler2D traversalStates, //traversal state
uniform sampler2D traversalStates2, //more traversal state
uniform sampler2D kdtree, //a texture representation of the kd-tree
uniform sampler2D triLists, //a texture with  the triangle lists
uniform sampler2D triangles, //the actual triangle vertices
uniform float kdtreeWidth, //width in pixels of the kdtree texture
uniform float listsWidth, //width in pixels of the list texture
uniform float trianglesWidth, //width in pixels of the triangle texture
float2 tc : TEXCOORD0, //texture coordinate indicating which ray we look up
uniform float loopLimitConst) //maximum number of loops allowed
{
   //find the ray and the previous best hit
   float4 bestHit = tex2D(bestHits, tc);
   float3 orig = tex2D(rayOrigins, tc).xyz;
   float4 dir = tex2D(rayDirections, tc);
   float4 traversalState = tex2D(traversalStates, tc);
   float traversalState2 = tex2D(traversalStates2, tc).x;

   //traversalState.x = down: -1, up: -2: [0;INF] traversing triangle list
   //traversalState.y = tree pointer
   //traversalState.z = t_min
```

```
//traversalState.w = t_max
//traversalState2.x = global_t_max

int kdtreeIndex      = traversalState.y;
float t_min            = traversalState.z;
float t_max         = traversalState.w;
float global_t_max   = traversalState2;
float listIndex;

bool isFinished = false;
int loopLimit = loopLimitConst;

while(!isFinished && loopLimit > 0) {
   //look up the first internal node
   float4 node_info   = ReadIndex(kdtree, kdtreeIndex++, kdtreeWidth);
   //node_info.x = right child pointer
   //node_info.y = split value along split axis
   //node_info.z = parent pointer
   //node_info.w = split axis if in {-2,-3,-4}
   //node_info.w = empty leaf if == -1
   //node_info.w = leaf with list pointer if >= 0

   //the bounding box of the node
   float3 box_min      = ReadIndex(kdtree, kdtreeIndex++, kdtreeWidth).xyz;
   float3 box_max      = ReadIndex(kdtree, kdtreeIndex++, kdtreeWidth).xyz;

   //are we moving up or down the tree ?
   if(traversalState.x == -2) { //we are going upwards from a leaf
      //test if we have come far enough up the tree
      float3 splits = (float3(node_info.y) - orig) / dir.xyz;
      float t_split = node_info.w == -2 ?
 splits.x :
 node_info.w == -3 ? splits.y : splits.z;
      if(BoxIntersects(box_min, box_max, orig, dir.xyz, t_min, t_max)) {
         //this node is where we need to be going back down again
         traversalState.x = -1; //-1 == "down"
         kdtreeIndex-= 3.0f; //to stay at the same node in next iteration
      } else { //else take another step upwards
         kdtreeIndex = node_info.z;
      }
   } else {
      //moving down the tree

      if(node_info.w < -1) {
         //this is an internal node -> traverse downwards a step
         float3 splits = (float3(node_info.y) - orig) / dir.xyz;
         float t_split = node_info.w == -2 ?
    splits.x :
    node_info.w == -3 ? splits.y : splits.z;

         float3 dirSigns = sign(dir.xyz);
         float dirSign = node_info.w == -2 ?
               dirSigns.x :
               node_info.w == -3 ? dirSigns.y : dirSigns.z;

         //determine which child is nearest
         float first = dirSign == 1 ? kdtreeIndex : node_info.x;
         float second = dirSign == 1 ? node_info.x : kdtreeIndex;

         if(t_split >= t_max || t_split < 0.0f) {//ray only passes through first
            if (t_split < 0.0)    kdtreeIndex = second;
            else                kdtreeIndex = first;
         } else if(t_split <= t_min) {//ray only passes through second
            kdtreeIndex = second;
         } else {//ray passes through first - then second. Go through the first now.
            kdtreeIndex = first;
            t_max = t_split;
         }
      } else {
         //this is a leaf - lookup list index and traverse it
         //Are we just starting list traversal at this node then start at the beginning,
         //else start where we left off
         listIndex = traversalState.x == -1 ? node_info.w : traversalState.x;

         float triIndex = ReadIndex(triLists, listIndex, listsWidth).a;
         triIndex = listIndex == -1.0f ? -1.0f: triIndex;//-1.0 indicates an empty leaf

         //traverse triangle list
         while(triIndex >= 0.0f && loopLimit > 0) {
            float3 a = ReadIndex(triangles, triIndex, trianglesWidth).xyz;
            float3 b = ReadIndex(triangles, triIndex + 1, trianglesWidth).xyz;
            float4 c = ReadIndex(triangles, triIndex + 2, trianglesWidth);

            bestHit = Intersects(a, b, c.xyz, orig, dir.xyz, 0.0f, c.w, bestHit);

            listIndex++;
            triIndex = ReadIndex(triLists, listIndex, listsWidth).a;
            loopLimit--;
         }
         //remember the list position we got to
         traversalState.x = listIndex;
         if(triIndex < 0.0f) {//Did we traverse until the terminator?
            //if a hit is found -> return if it is inside this voxel
            isFinished = bestHit.z <= t_max ? true : false;
            if(t_max > global_t_max - 0.00001f)
               isFinished = true; //ray missed all triangles in the scene

            t_min = t_max;
            t_max = global_t_max;

            kdtreeIndex = node_info.z;//go to parent node (as this is 'backtrack')
            traversalState.x = -2;//we are now traversing up the tree
         } else {
            //must point to the current leaf node if we are to resume list traversal
            kdtreeIndex-=3.0f;
         }
      }
   }
   loopLimit--;
}

traversalState.y = isFinished ? -1 : kdtreeIndex;
traversalState.z = t_min;
traversalState.w = t_max;

main_out OUT;
//set output variables
OUT.renderState = traversalState;
OUT.bestHit = bestHit;

return OUT;
}
```