Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS

Pavel Herrmann

GPU Raytracer

Department of Software and Computer Science Education

Supervisor of the bachelor thesis:	RNDr. Josef Pelikán
Study programme:	Computer Science
Specialization:	Programming

Prague 2011

Název práce: GPU Raytracer

Autor: Pavel Herrmann

Katedra: Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Josef Pelikán

Abstrakt: Ray tracing je oblíbená metoda pro vytváření realistické grafiky, s velkou výpočetní složitostí a velkým potenciálem pro paralelizaci. Moderní GPU mohou sloužit jako výkonné paralelní koprocesory, díky čemuž se zdají být ideální nástroj pro ray tracing.

Tato práce obsahuje přehled o technikách ray tracingu, přehled o technikách programování GPU, a představuje software vytvořený pro použití GPU pro ray tracing. Tento software se snaží spojit klasické metody pro ray tracing se specifickými vlastnostmi programování GPU, při zachování rozšiřitelnosti a vysoké rychlosti.

Klíčová slova: GPGPU, Ray tracing, OpenCL, Počítačová grafika

Title: GPU Raytracer

Author: Pavel Herrmann

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Josef Pelikán

Abstract:

Ray tracing is a popular method for generating realistic imagery, with high computation complexity and high potential for parallelization. Modern GPUs can be used as a high performance parallel co-processor, making them seemingly ideal for tasks such as ray tracing.

This thesis will give an overview of ray tracing methods, overview of GPU computing methods, and present a piece of software designed for using GPU for ray tracing. This software tries to integrate classic ray tracing algorithms with specifics of GPU programming, while providing extendability and high performance.

Keywords: GPGPU, Ray tracing, OpenCL, Computer graphics

Contents

Introduction

The goal of this thesis is to create a modular ray tracer using OpenCL technology to take advantage of the computing power of modern GPUs.

Motivation

Ray tracing is an advanced form of image generation in computer graphics with many specifics. It can be used to create photo-realistic imagery without complex tricks, by simply simulating how the material in question behaves, and tracing the causes of color in the image. The downside of ray tracing is the large complexity put into each pixel of the image, which prevents ray tracing from becoming the first pick for any computer graphics.

On the other hand, ray tracing is very parallel task, easily spreading to multiple computing units. Current CPUs have as much as 10 physical cores, but GPUs have over 500 cores. That seems like GPUs are an ideal target for ray tracing implementations, but that is not entirely accurate. GPU cores often operate on much lower frequency than CPU cores, have different memory hierarchy, and often are not as independent as CPU cores. However, when used properly, GPU outperforms CPU hands down. Due to this, ray tracing on GPU has been an interest to many researchers.

1. Introduction to ray tracing

Ray tracing is a name for several techniques used for generating two-dimensional views of a three-dimensional scene. All these techniques can be characterized as simplified simulation of light transport using geometric optics,

1.1 History of ray tracing

The simplest version of ray tracing, today more often referred to as 'Ray Casting', was introduced by Arthur Appel in 1968[1]. The idea was to create a ray coming from the eye through each pixel of the image and intersecting it with the scene, finding the closest object that intersects with the given ray. At this point, ray tracing was used for solving visibility problems for solid objects, rather than for creating realistic images, due to not simulating reflection and refraction effects. Appel did, however, use shadow rays to determine whether the point is in shadow or not, although that is not considered a part of ray casting.

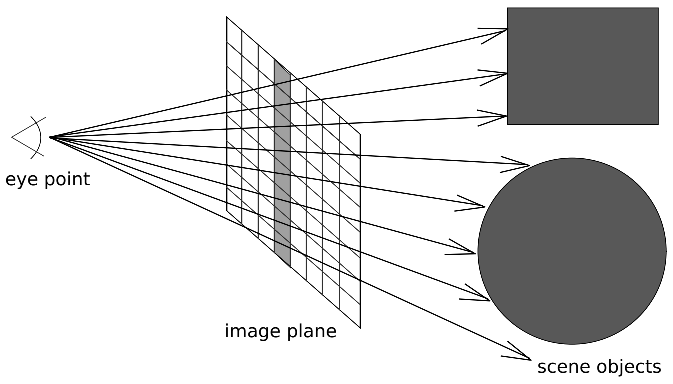

Figure 1.1: Ray casting, with highlighted primary rays

Probably the most significant improvement was brought in by Turner Whitted in 1980[2], creating what is today commonly known as 'Ray Tracing' or 'Whitted Ray Tracing' or 'Recursive Ray Tracing'. His idea was to use rays not only for finding the nearest object for each pixel, but also to determine the color of this pixel. In his work, Whitted introduced three types of secondary rays[1] cast from the intersection point: reflection rays, refraction rays [2], and shadow rays [3]. Refraction and reflection rays were used to simulate optical properties of reflective and (semi-)transparent materials, while the shadow rays were used to find any objects obstructing the light source, and therefore create very accurate shadows and add realism to the image. The reflection and refraction rays are traced in recursive fashion, creating more secondary rays on new intersections with the scene, until the rays contribution to the result is considered minimal and the ray

[1]Secondary rays are rays not originating in the eye, but rather in the intersection point
[2]at most one per intersection each
[3]one per intersection per light source

is terminated[4]

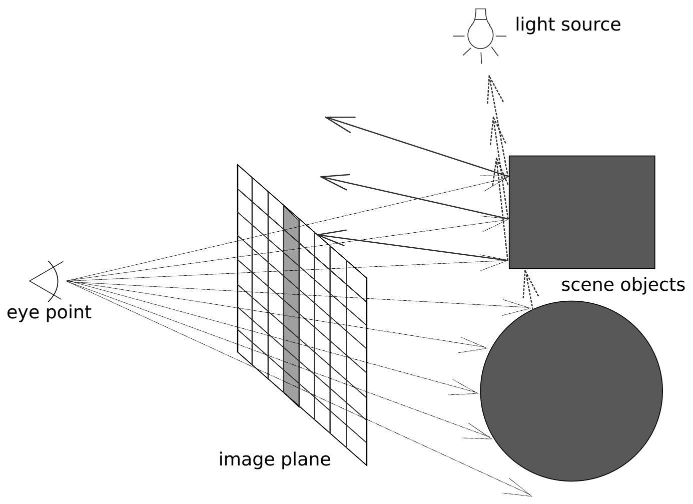

Figure 1.2: Recursive ray tracing, with highlighted reflection rays and shadow rays (dotted)

The most recent improvement of Ray Tracing algorithms in terms of realism was 'Distributed Ray Tracing', introduced by Robert Cook in 1984[3]. This method extended Recursive Ray Tracing by creating several reflection and refraction rays, and sampling them using probability methods. Distributed Ray tracing is also used to simulate motion blur or depth of field, by shooting multiple primary rays per pixel, with different timeframe or focal point. Multiple secondary rays produce realistic effects like glossy reflection, diffraction or smooth shadows from area lights, but also significantly increase the computational complexity.

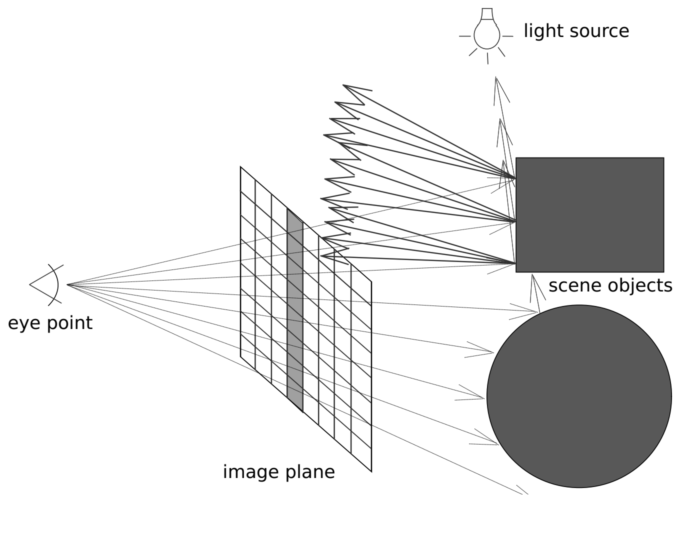

Figure 1.3: Distributed ray tracing, with highlighted multiple reflection rays

[4]shaded without casting recursive rays

1.2 Performance of ray tracing

While Ray Tracing can generate very realistic imagery, it comes at a price. Measurements show that up to 95% of time is spent on calculating ray-triangle intersection[2], which makes it a perfect target for improvements.

There are two independent directions of research, one aims to improve the speed of a single intersection test, while the other aims to reduce the number of triangles each ray has to be tested against in order to find the closest intersection point.

Methods of improving speed of intersection tests mostly consist of using vector instructions to do several tests at the same time, or intersect the scene with a higher order primitive instead of a ray. Such methods are often highly platform-dependent and will not be discussed in detail in this thesis, instead we will discuss ways of reducing the number of intersection tests needed.

Hybrid techniques

Some parts of ray tracing pipeline can be approximated by methods used in classic rasterization pipeline, which often have significant positive performance impact. However, these methods come at the cost of image quality, which makes them unsuitable for general ray tracers, but useful in specific applications.

Examples of such techniques would be using depth buffers instead of tracing shadow rays, or vista-buffering, a method of using perspective projection to determine intersection candidates for primary rays.

Spatial subdivision

The general idea of spatial subdivision is to divide space into non-overlapping cells, and to traverse the ray through this structure, intersecting only with triangles present in relevant cells. Disadvantage of spatial subdivision is creation of duplicate entries for triangles overlapping multiple cells.

Some well-known examples are uniform grids, octrees and kD-trees, discussed later.

Mailboxing

Mailboxing is based on a concept of intersection memory. Spatial subdivision techniques create neighbor cells containing the same triangles, which cause the algorithm to repeat some intersection tests, gaining no new information. Mailboxing tries to prevent this by remembering a certain number of triangles the ray has already been intersected with in a ring buffer-like data structure. This, however, adds some computational and memory overhead, and has not seen widespread adoption.

An improved version called hashed mailboxing tries to lower the overhead of consulting the mailbox by using a small hash table instead of a ring buffer, providing computation overhead independent on mailbox size. Despite this, the overhead still proves too big for general use[4]. However, techniques intersecting higher order primitives consisting of multiple rays with the scene do see performance improvements.

Scene subdivision

Scene subdivision techniques try to divide objects in the scene into groups, and quickly eliminate the whole groups as intersection candidates. As opposed to spatial subdivision, scene subdivision structures only have one entry for each triangle, but can overlap in space, adding some complexity to construction and intersection algorithms.

An example of scene subdivision techniques are bounding volume hierarchies, which will be discussed later

1.2.1 Uniform grids

Uniform grid is a simple spatial subdivision technique, creating a three dimensional grid of uniformly sized cells. This approach is very easy to implement, and very fast to build and therefore well suited for dynamic scenes.

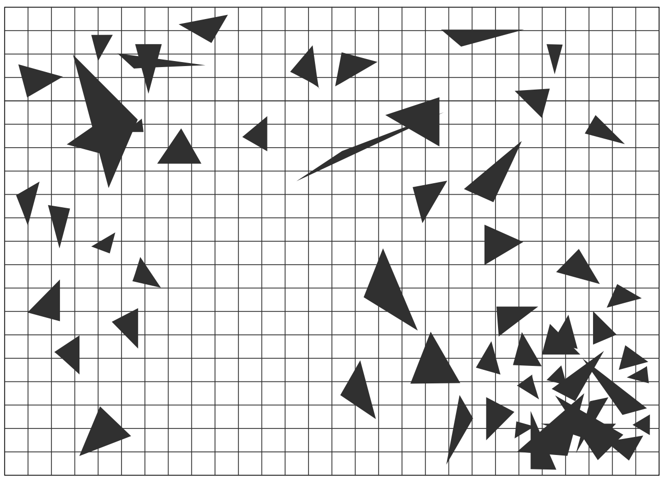

Figure 1.4: Uniform grid example

As a non-hierarchical approach, Uniform Grids suffer from teapot-in-a-stadium problem, which occurs in scenes with small complex geometry in one place. In such case, most of the cells with the large low-detailed scene (stadium) will be empty, while the cell containing the complex geometry (teapot) will have a long list of triangles, which greatly decreases performance

A simple way to create an Uniform Grid in compact memory with O(n) complexity[5] is explained in Algorithm 1.

Simple traversal algorithm for uniform grids is given in Algorithm 2

One of posible improvements of this concept is sparse grids, which include the distance to nearest non-empty cell in each cell to improve traversal speed for scenes with lots of open space, Another such improvement is hierarchical grids, which embed new uniform grid into some cells, creating partially hierarchical structure, which increases scale differences needed for teapot-in-a-stadium problem to manifest. Nevertheless, it does not eliminate it entirely. Another grid-based acceleration structure is a perspective grid, which partitions a perspective projection of space and simplifies the traversal algorithm. However, this is only

Algorithm 1 Fast construction of compact uniform grid

```
This algorithm requires one fixed array of length Cell Count+1, referred to as
$Cells$, and a second array of triangle indexes, referred to as $Tris$, allocated
later.

for all $t \in$ Triangles do
  for all $c \in$ Cells intersecting $t$ do
    $Cells\,[c] += 1$
  end for
end for
for $c \in 1$ ... Cell Count do
  $Cells\,[c] += Cells\,[c-1]$
end for
$Tris$ = Allocate memory for $Cells$[Cell Count]
for all $t \in$ Triangles do
  for all $c \in$ Cells intersecting $t$ do
    $Cells\,[c] -= 1$
    $Tris\,[Cells\,[c]] = t$
  end for
end for

Triangles in a given cell C are now found in $Tris\,[Cells\,[C]\,...Cells\,[C+1]-1]$
```

Algorithm 2 Traversal algorithm for uniform grid

```
Ray is in form $origin\ +\ t\ *\ direction$

set $t_x$,$t_y$,$t_z$ to $direction\ /\ cellsize$ {this represents $t$ increments between cells
in each direction}
set $d_x$,$d_y$ and $d_z$ according to the origin point {this represents $t$ increment
required to move to next cell in each direction}
intersect ray with all triangles in the current cell
while intersection not found do
  find minimal $d_i$ and subtract it from $d_x$,$d_y$ and $d_z$
  set the now zero $d_i$ to corresponding $t_i$, move one cell in the corresponding
  direction
end while
select intersection with minimal $t$ in the current cell {shadow rays do not need
to find the closest intersection}
```

usable for primary rays, one would need a regular grid or another acceleration structure for secondary rays.

1.2.2 Octrees

Octrees try to solve the teapot-in-a-stadium problem by adding adaptive hierarchy, while still retaining relatively simple traversal and construction algorithms.

Generally, octrees are tree structures where each internal node has eight children. In Ray tracing, octrees are visualized as cells splitting into eight by halving each axis.

A basic octree construction algorithm would begin by creating one cell containing the entire scene, and then finding any cell with too many triangles and splitting it

However, due to added construction complexity compared to uniform grids and poor performance compared to kD-trees, octrees are not widely used in modern ray tracers

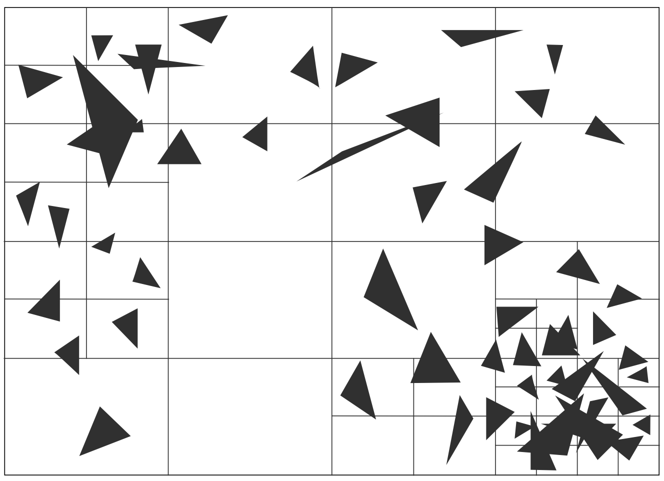

Figure 1.5: Octree example

1.2.3 kD-trees

kD-trees are a popular kind of binary space partitioning trees, which means that every internal node has two children. In case of kD-tree, each cell can be split by a plane perpendicular to an axis at any point of the cell.

Most, if not all, construction algorithms use some form of Surface Area Heuristic (or SAH for short) to aproximate the cost of intersecting a ray with this cell. A simple SAH weighs the cost of intersection tests with the bounding box and all triangles contained within the cell[5] against the surface area of the cell [6].

It is easy to see that SAH is linear and partially continuous, with discontinuities in triangle begin/end points. This means that any candidate for minimal

[5]all triangles are assumed to be of similar size

[6]with an assumption that the rays are uniformly distributed over the cells surface, which doesn't hold for primary rays, but still provides good estimates

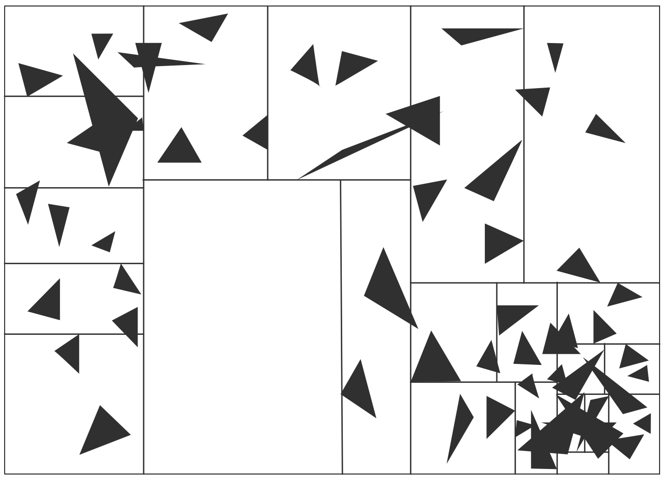

Figure 1.6: kD-tree example

SAH is a triangle begin/end point. Furthermore SAH can be easily calculated by one pass over sorted vertex data, because the number of triangles in each cell always changes by one.

kD-trees show superior performance to octrees, with known construction algorithms with O($n \log n$) complexity with very good results [6], such as Algorithm 3

Algorithm 3 Fast construction of kD-tree

```
create lists $S_x$, $S_y$ and $S_z$ by sorting all triangle begin/end points in x,y and z coordinate respectively

evaluate SAH on all points in $S_x$,$S_y$ and $S_z$
if minimal SAH < cell SAH then
   place the split plane on minimal SAH value
   split $S_x$ into $S_{x1}$ and $S_{x2}$ for child cells, according to the split plane position
   repair $S_{x1}$ and $S_{x2}$ by adding begin/end coordinates for triangles present in both lists
   do the same for $S_y$ and $S_z$ {note that this keeps the lists sorted}
   recurse for each child cell
end if

with reasonably low number of triangles intersection the split plane this finished in $O(n \log n)$ time
```

Due to the hierarchical nature of kD-trees, traversal algorithm is best explained in recursive way, as in Algorithm 4

1.2.4 Bounding Volume Hierarchies

Bounding Volume Hierarchies (BVHs for short) are a scene subdivision structure, which creates geometrically simple pseudo-objects (mostly axis-aligned bounding

Algorithm 4 Traversal algorithm for kD-tree

```
if cell is leaf then
    intersect with all triangles
    if intersection found then
        select closest intersection and finish
    end if
else
    intersect ray with split plane
    if intersection is after ray leaves the cell then
        recurse into first cell
    else if intersection is before ray enters the cell then
        recurse into second cell
    else
        recurse into first cell
        recurse into second cell
    end if
end if
```

boxes, but some researchers work with higher order volumes) to exclude large amount of geometry from intersection testing. This is done in a hierarchical fashion, creating bounding volumes over bounding volumes etc. creating a tree-like structure. As opposed to kD-trees, BVHs tend to have a lot of empty space, with no leaf volume (and therefore no triangles to test), but also contain space which is encompassed by several leaf volumes, forcing rays going through this area to be tested against triangles in all these volumes (no triangles are shared though).

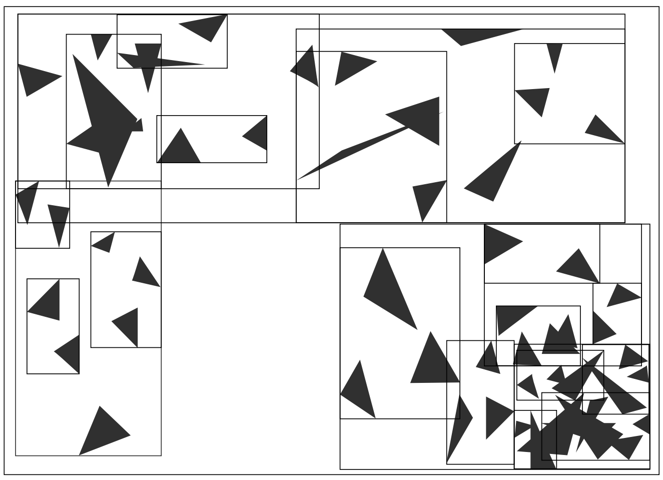

Figure 1.7: Bounding volume hierarchy using axis-aligned bounding boxes

there are several BVH generation techniques, with top-down techniques (splitting space into groups, similar to kD-tree construction) being faster, and bottom-up (grouping objects and smaller groups until all geometry is in a single group) creating superior results.

2. GPU Computing

GPU computing or GPGPU refers to the usage of advanced graphics hardware to perform non-graphics related calculations. This is done by running programs called kernels on GPU shader units in a parallel manner.

2.1 History of GPU Computing

Historically, users were able to adjust some options regarding the rendering pipeline to change the behavior of texturing and shading, but this was very limited. As speed of such hardware increased, users became interested in greater flexibility for fragment shading.

In 2000, Microsoft published version 8 of their DirectX API, which included support for programmable fragment and vertex shaders [1]. At first, these shaders were very limited (in maximum instruction size, floating point precision etc), but with each new revision of DirectX these limitations loosened.

Starting with GeForce 8 and Radeon HD2000 series (introduced in 2006/7 respectively), instruction sets for fragment and vertex shaders became so similar that manufacturers began using a unified shader design, where one hardware unit could run both fragment and vertex shaders.

in 2007, nVidia unveiled its CUDA GPGPU technology, designed to run arbitrary calculations on GeForce 8 or newer GPU [2].

in 2008, Apple created initial version of OpenCL, submitting it to Khronos Group[3] for review and publication.

2.2 CUDA

CUDA is the first and currently most widespread GPGPU technology. The language for writing kernels is a modified version of C, with some notable exceptions [4] and additions for parallelism and disjoint memory spaces

CUDA-compatible hardware includes any nVidia-manufactured GPU from G80 generation (GeForce 8) onward. Hardware feature differences are specified using 'Compute Capability'[5] or CC in short. Differences between Compute Capability levels include double precision support and performance (introduced in CC 1.3, speed improved in CC 2.0), memory access capabilities, cache size, atomics and more.

Current versions of CUDA toolkit combined with a CC 2.0/2.1 GPU supports a subset of C++ objective functionality, with more functionality being added in every new version of CUDA toolkit.

[1]First hardware with support for this new feature was GeForce 3 by nVidia and Radeon 8500 by ATi, both available in 2001

[2]ATi introduced their 'Close to Metal' technology as a part Stream GPGPU toolkit at the same time, but it has not seen widespread adoption and was later replaced by OpenCL

[3]maintainer of OpenGL and other open standards

[4]no function pointers, no recursion, no double precision floats on most hardware

[5]Current GPUs have Compute Capability 2.0/2.1

Kernel execution is done in groups called warps, which are executed on a group of shader units called SM, which shares registers, instruction decoder unit, texture access units and SFUs[6]. In this architecture instructions are issued to all threads on a SM at once. While threads can mask instructions to perform different calculations, they have to wait in loops until all threads in a warp are finished before proceeding. Furthermore instructions of all conditional blocks that at least one of the threads takes have to be issued. Due to this fact, calculations that diverge greatly suffer a serious performance penalty, and the aim is to use as little conditional blocks as possible. Currently all GPUs can run 32 threads on a single SM at the same time, but using larger warps is desirable, to avoid latencies from memory access

The memory model of CUDA is also different from standard C, having three disjoint memory spaces, global, shared and local. Local memory represents registers allocated by the running thread, and is used to keep most of temporary variables during computation, as it has very low access latency, but is very limited in size. Shared memory is also stored in registers, but is shared between threads in a warp. Because of not all threads are running at the same time, all read/write access to shared memory should be protected by barrier instructions to prevent race-conditions. Global memory represents RAM on the card, and it is significantly larger than local memory but also much slower and has large access latency. This is where kernel input and output data generally is. This is also the only memory space that can be accessed on the host code (running on the CPU). All local and shared memory is allocated statically, global memory is allocated in host code and until recently, there was no support for dynamic memory allocation. GPUs with CC 2.0/2.1 have an ability to dynamically allocate global memory in kernel code with cooperation of the GPU driver, local and shared memory is still allocated statically.

CUDA kernels are compiled into low-level bytecode called PTX, which is compiled by the runtime in the driver for the specific GPU.

2.3 OpenCL

OpenCL was designed by Apple in pursuit of enabling user to develop GPGPU applications without worrying about his target platform. The OpenCL kernel language is very similar to CUDA, with some differences in keywords ('private' memory instead of 'local', 'local' instead of 'shared', 'workgroups' instead of 'warps' etc), while the runtime characteristics are somewhat similar to OpenGL, having a set of core functionality and a set of vendor extensions. Currently, the feature set of OpenCL is somewhat lacking, as there is no C++ support, no dynamic allocation in kernel code, and no support for most features introduced in CUDA CC 2.0/2.1.

Due to runtime platform detection and selection, OpenCL mostly uses kernels distributed in source mode, and compiles them at runtime to suit the specified platform and device [7].

The main advantage of OpenCL is openness and ability to run on several

[6]special function units for transcendental functions

[7]Binary kernels are supported, but cross-device portability is not guaranteed

different platforms, with the latter currently not working as originally intended. [8]. Some of the disadvantages are a too general approach, which causes the lack of features in comparison with CUDA, and current state of compilers/drivers, which often contain bugs and have non-ideal optimization routines.

The current revision of OpenCL is 1.1, which adds support for 3-component vector types [9], moves some atomic operations into the core instead of being an extension, and guarantees OpenCL thread-safety from host code, along with some less significant changes. This revision is supported by AMD Stream SDK version 2.2 and newer. Unfortunately, nVidia tries to push its CUDA technology, and their support of OpenCL is somewhat lacking. nVidia drivers version 195 and newer support OpenCL 1.0, while a pre-release driver with support for OpenCL 1.1 is available to registered developers[10].

[8]The same OpenCL kernel can run on both nVidia and AMD/ATi hardware, but due to architecture differences the performance would not be optimal

[9]even though these are considered as 4-component in memory

[10]version 258.19, released in June 2010

3. The clTracer program

clTracer is a modular ray tracing system, using OpenCL technology to utilize the computing power of GPUs.

The aim of clTracer is rendering flythrough-like scenes, where the scene remains constant while the camera changes positions. The input is a custom configuration file, which references a single scene geometry file, several lights and camera settings for all the requested frames. An interactive mode is also available, where the configuration file contains camera settings for the first frame, and the user is able to dynamically change the position and orientation of the camera. In addition to the configuration and geometry file, a material file with a list of textures and the material shaders have to be supplied.

Material shaders are called directly, and thus have to be programmed in OpenCL. However, most of the language specifics are prepared in advanced, so basic knowledge of C is sufficient to create shaders.

Implementation details like function interfaces or code examples are not included here, but are available in the source tree under 'doc' subdirectory. This is because those can change in future, confusing the potential user of clTracer after reading this thesis.

3.1 General design

clTracer codebase consists of two major parts, the control program and OpenCL kernels, both made of several components.

The overall design strives to be highly modular, enabling improvements or alterations of components without affecting large amount of scattered pieces, and thus improving maintainability.

The Central part of the design is the shared type definition, which guarantees type compatibility between the control program and OpenCL kernels. Any changes to types require recompilation of the control program, because OpenCL kernels are runtime-compiled and automatically use the latest type structure, which could cause invalid memory access. This also requires the use of some preprocessor macros, in case of types with same name have different size.

3.2 Control program design

The control program consists of a grid construction component, an OpenCL component, a configuration parser component, a scene loading component, a texture and material loading component, an image output component, an interactive control component and a control component. Some of these components are simple, while some will be explained into detail.

Most of the code is written in C99 with GNU extensions[1], only the scene loading component uses C++. This language was chosen for its low overhead, direct interface with OpenCL and the possibility of code-sharing with kernels

[1]anonymous unions, used to mimic OpenCL vector type access

3.2.1 The Simple components

The simplest of the components is the image output, which is used in non-interactive mode to save the result to a file, and is basically a wrapper around DevIL library. This component contained some custom image manipulation code in the past, but has since been swapped for the current DevIL wrapper.

Another quite simple component is the configuration parser. This component isn't based on a stack machine design like parsers generated by lex/yacc and similar, but rather works by simply trimming spaces of a line and trying to match a keyword, and then parsing the rest of the line using C scanf function. While a stack machine based approach is more powerful than this design, it was deemed to be an overkill for short and simple configuration files.

The grid construction component is also in need of no explanation, as it uses the fast compact grid building algorithm listed in section 1.2.1 as Algorithm 1

The control component is responsible for interacting with all the other components - it takes care of calling everything in the correct order, and makes sure that all memory gets cleaned.

3.2.2 The Scene loading component

The scene loading component is responsible for loading the triangle mesh from a file. Wavefront OBJ was chosen as an input format, mainly due to its simplicity and support in wide range of editors.

OBJ is a simple text-based format, with lines either adding new vertex coordinates (separately for space, normal and texture) to the scene database, or creating convex polygons by referencing coordinate index triplets. While the format allows for incomplete triplets[2], this component expects full index triplets [3]. Additionally convex polygons are not very usable objects, and are thus interpreted as triangle fan and split into separate triangles.

No NURBS geometry is supported by the parser, and will be ignored if present in the file. The material format of OBJ is not used, and is instead replaced by a custom one more suited for the use-case [4].

The parser is not based on a grammar-generated state machine, but rather has a much simpler design. After trimming any leading whitespaces, the program takes advantage of the hierarchical meaning in most keywords in the format, and goes through a few switch statements directly to the code used to parse the specific line. This means the code is still easily readable and fast, as it only passes the line once. On the other hand, the code does not recognize an invalid input file if the incorrect part is not used to determine the type of line being parsed..

Due to the OBJs stream-like format with back references, this component is written in C++ to take advantage of templates for automatically resizing data structures.

[2]space coordinates are required, texture and normal coordinates may be omitted

[3]There is no sensible default value - zero would break normalization, anything else would be wrong when used, therefore such polygon is ignored

[4]the material file states the name of material, followed by the number of image textures used, followed by texture filenames. The actual material shader is provided elsewhere

3.2.3 The Interactive control component

This component handles the interactive mode, and is the only place with platform-specific code [5], currently with support only for GLX [6], on which the code is developed

This code uses GLUT[7] to create an output window, handle input and copy output from OpenCL memory onto screen.

The most straightforward way of achieving this goal is to use OpenGL-OpenCL texture sharing, which should allow an OpenCL kernel to write directly into the OpenGL texture. This, proved to be non-functional for some reason, and a workaround had to be found.

The cleanest way to do this was found in the AMD Stream SDK examples[8], and it is to employ a pixel buffer as a middle step. This trick uses the pixel buffer as a simple byte array - the image is converted from its internal floating point representation into 8bit per channel RGBA and saved into the pixel buffer. The next step is to synchronize with OpenGL and hand over the pixel buffer, and to copy its contents into a prepared texture. This texture is then displayed using orthogonal projection.

Due to high complexity of ray tracing, the image is recalculated only if movement is detected, and not on every screen refresh.

3.2.4 The Texture and material loading component

The Texture and material loading component is responsible for loading the texture images and for generating material shader kernels.

The texture loading part uses information from the scene loading component to load appropriate files using DevIL image library and to fill texture information structures, which are to be used by the OpenCL component

Due to restrictions posed on usage of textures in OpenCL[9], the material loading part of this component has to dynamically create the shading kernel based on materials used in the scene, and also make sure that textures get passed to the correct material shaders.

The format of the shading kernel will be detailed in section 3.3.4.

3.2.5 The OpenCL component

The OpenCL component is responsible for most of GPU-related operations.

This component loads sources for all kernels and compiles them, also is responsible for allocation (and setting, if necessary) of all OpenCL memory buffers, and for running the kernels in the correct order.

Because there may be several reflection/refraction rays coming from a single primary ray[10], the ray memory buffers may need to be extended. To keep the

[5] there is no platform-independent way of getting the current OpenGL context, which is required for memory sharing between OpenGL and OpenCL

[6] OpenGL on X11 on UNIX-like systems

[7] OpenGL Utility Toolkit

[8] and also many places on the web

[9] Image types cannot be used to define a type, or within structures or arrays[7]

[10] shadow rays are cast separately, at rate one per intersection per light source

overhead from reallocation small while conserving memory, this component will at least double the size of each buffer when reallocation is necessary, and will not reallocate when smaller number of items is requested.

3.3 OpenCL kernels

Due to the non-recursive nature of OpenCL, the standard recursive ray tracing algorithms do no really work, and instead have to be transformed into iteration-based algorithms. This has some implications for the shading process, and makes it a bit harder to implement hierarchical acceleration structures.

The revision of OpenCL used is 1.1, which is the highest at the time of writing. For more information, please see section 2.3.

A simplified Dataflow diagram is listed as Figure 3.1, the counting kernel missing from the diagram is run in parallel with the lighting kernel.

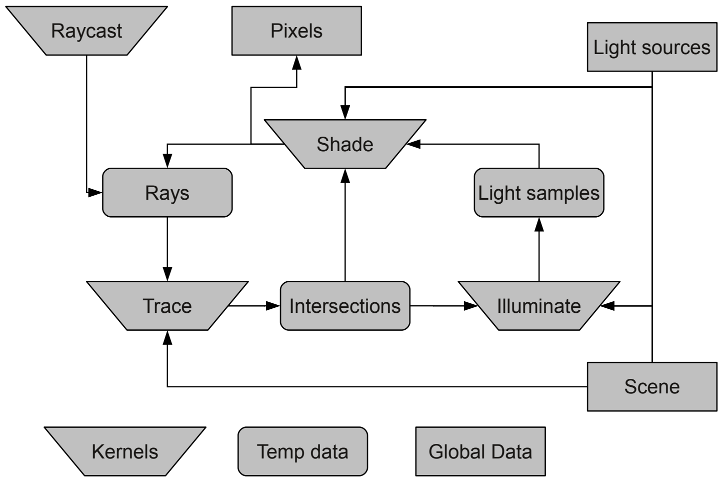

Figure 3.1: Dataflow between OpenCL kernels

Due to OpenCL not having a stack or dynamically allocated memory, association between rays is kept in a structure called image reconstruction information, described in section 3.3.4

3.3.1 Misc kernels

Misc kernels are contained in the 'tools.cl' file, and are run in order to prepare the scene. These kernels normalize direction vectors for lights and normal vectors for triangles.

These kernels also include the ray casting kernel, which is used to generate primary rays for the scene. The algorithm used for this creates a side vector, perpendicular to both the up vector and the eye direction vector, then it creates a new up vector perpendicular to the side vector and eyedirection vector. The last step of the algorithm is to scale the side vector and the new up vector to suit the desired view angle, and to generate primary rays by taking the eye direction vector and adding fractions of the side and the up vectors corresponding to the coordinates of the desired pixel.

The last kernel considered in this group is used to copy the rendered image to the OpenGL buffer, as explained in section 3.2.3.

3.3.2 Ray tracing kernel

The ray tracing kernel is used to find intersection between the scene and the rays. On hit, this kernel records the closest triangle and barycentric coordinates of the intersection point for use in shading kernel. When the ray doesn't hit any triangle, a negative triangle index is used, and barycentric coordinates are not valid.

This kernel examines the supplied acceleration structure and selects the intersection algorithm at runtime, currently by using a simple brute-force algorithm or a uniform grid traversal algorithm.

Brute-force algorithm

This algorithm is based on basic principles of ray tracing, intersecting each ray with each triangle and picking the intersection point closest to the ray origin. The performance of this algorithm is very bad, nevertheless it is included as a reference algorithm that produces correct results

Grid traversal algorithm

This algorithm uses a uniform grid structure to lower the required number of intersection tests, and is very similar to Algorithm 2. While the performance of this algorithm is not in the real-time domain, it is considered fast enough to use, enabling easy development of other components of the program.

3.3.3 Lighting kernel

The lighting kernel is responsible for evaluating light samples at each intersection point. For each light source it evaluates the potential light sample, and if necessary it traces a shadow ray.

The currently supported light types are ambient light (constant light in the whole scene, doesn't cast a ray), directional light (light with a uniform direction and intensity, like sunlight), omni light (omni directional light with a single point of origin), spot light (light with a given point of origin, a direction vector, and an angle of the light cone).

If a shadow ray is needed, an early-exit version of the trace kernel is called, performing the same algorithm selection as the tracing kernel. Light samples from all light sources are made available to the shading shading kernel for each intersection point.

3.3.4 Shading and counting kernels

These two kernels get runtime generated by material loading component to suit the materials used in the scene.

The counting kernel works around OpenCL not allowing to allocate memory from kernel code. Instead, this kernel runs a limited material shader called the counting shader, which returns the number of secondary rays from the given intersection, and uses atomic instructions to sum the rays and pass the number to the control program, which then reallocates buffers if necessary.

The shading kernels task is to evaluate the pixel color and determine secondary ray properties at each intersection point. The material shaders are required to cast exactly as many secondary rays as the corresponding counting shader allocated. Due to iterative manner of the shading process, any change to the pixel is done before casting secondary rays, which also means that each run of shading kernel directly accesses the image.

Any ray-related information needed by the shading process has to be passed forward along with the ray as image reconstruction information. Normally, this structure contains the number of the original pixel, the importance of the ray and a seed for a random number generator (see 3.3.5), but it can also contain any additional information needed by the shading kernel to determine what operation to apply. Examples of such would be the ray spectrum, if one used clTracer as a spectral renderer[11], the depth of the secondary ray, if one wanted to use it as a ray termination criterion, or any data one wanted to access in process of shading the intersection point of the ray.

Every triangle in the scene has a single associated material shader, and the environment has one additional special material shader, which gets called when the ray doesn't hit any triangle. Because the environment shader is implicit, it cannot use any textures, and because no intersection was found, no triangle is available. The only useful attribute for the environment shader is the direction of the ray, and anything passed in image reconstruction information.

3.3.5 Helper tools

The helper tools are small algorithms kept in a single place and used in several kernels. These algorithms include the ray-triangle intersection test or linear and spherical interpolations.

Two important parts of there tools are pixel locking support and random number generation.

Locking support

To properly handle multiple rays coming from one pixel, access serialization must be implemented. In current version, this is done by putting a spinlock-based structure on every pixel, and having the shading kernel use it whenever there is a possibility of multiple rays accessing one pixel. This, of course has some overhead, and should be replaced by atomic functions once atomic float addition becomes possible in future OpenCL versions.

Random numbers

Random numbers are required for distributed ray tracing and for some procedural materials, and as such need to be included by clTracer. Due to OpenCL

[11]in which case the format of the pixel data, the light source data and light sample data would probably need to be changed

not having random number generator routines[12], a custom solution had to be implemented.

clTracer uses the image reconstruction information of each ray to keep its unique random seed. Those seeds are first generated by the control program (see 3.2.5), and then distributed during the shading process. To ensure the uniqueness of each sequence, two random number generators are used - one generates a floating point number in range (-1,1), and the other generates a new random seed, but doesn't change the original seed. A new random seed is necessary when creating multiple rays from one intersection point, so that the new rays don't share the same random number sequence. For this to work, it is necessary to use two different generators for this purpose.

Currently, both generators are implemented as linear congruent generators, due to their good randomness/instruction ratio, and their small seed size. Thanks to the modularity of the clTracer design, using other random generators is straightforward, and would only require to change the actual generator code, and possibly the data structure used as a seed.

[12]OpenGL has a random number function, but it either returns a constant zero (on nVidia cards), or a value is generated and supplied by the driver

Conclusion

clTracer is a usable GPGPU ray tracing system, with large amount of flexibility in shader programming and reasonable performance, licensed under GPL open-source license.

That being said, nVidia has released their GPGPU ray tracing framework called OptiX, which is even more flexible than clTracer, while achieving higher performance, probably due to use of hierarchical acceleration structures. The only downside of OptiX is that it is CUDA based and will only run on nVidia GPUs, and also the full source is not publicly available.

Results

The performance results of clTracer are somewhat underwhelming, reaching real-time frame rate only for simple scenes without secondary rays or for small resolution, even on high-end hardware. Profiling shows that most of the time is spent in kernels that do intersection tests and acceleration structure traversal, and that these kernels have a high number of divergent branches. This is an inherent limitation of GPU programming, where groups of threads share instruction units. Attempts to use this coherency were made, but explicit thread synchronization proved to have too large an overhead.

Furthermore, uniform grid is not the best acceleration structure speed-wise, but it has simple control flow, with should have minimize the number of divergent branches. Neither kD-trees nor BVHs were tested or implemented in clTracer as of yet.

Further work

clTracer is far from finished, with several issues to improve upon. One of these points is performance, which can be helped either by improving intersection tests or implementing new acceleration structures or even decreasing the control overhead. Another infinished issue is creating a library for common shader operations, including common light models, world-to-local and local-to-world coordinate transformations, noise functions and more such helper algorithms.

The most recent version of clTracer can be found at `http://repo.or.cz/w/cltracer.git`

References

[1] Arthur Appel. Some Techniques for shading Machine Renderings of Solids. AFIPS '68 (Spring Joint Computer Conference) Proceedings of the April 30–May 2, 1968, pages 37-45

[2] Turner Whitted. An improved illumination model for shaded display. Communications of the ACM CACM, Volume 23 (6), 1980, pages 343-349

[3] Robert L Cook, Thomas Porter, Loren Carpenter Distributed Ray Tracing. SIGGRAPH '84 Proceedings of the 11th annual conference on Computer graphics and interactive techniques, 1984, pages 137-145

[4] Vlastimil Havran. Mailboxing, yea or nay?. Ray Tracing News, 2002, volume 15(1), page 1

[5] Ares Lagae, Philip Dutré. Compact, Fast and Robust Grids for Ray Tracing. Eurographics Symposium on Rendering 2008, 2008, volume 27(4), pages 1235-1244

[6] Ingo Wald, Vlastimil Havran. On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N). Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, 2006, pages 61-69

[7] Khronos OpenCL Working Group. The OpenCL Specification, version 1.1.

List of Figures

List of Abbreviations

API Application Programming Interface

Definition of ways to use an external programming library

BVH Bounding Volume Hierarchy

Method of grouping objects used for improving ray tracing performance

CPU Central Processing Unit

The main computation component of every computer

CUDA Compute Unified Device Architecture

Name for nVidia GPU architecure, and also for their GPGPU technology

GPGPU General Purpose GPU

Concept of using GPU as a parallel co-processor for non-graphical calculations, or a related technology

GPU Graphics Processing Unit

Hardware specialized in rendering graphics

PTX Parallel Thread Execution

Low-level bytecode produced by CUDA compiler, interpreted by nVidia GPUs

RGBA Red Green Blue Alpha

Format of storing image color information with transparency

SAH Surface Area Heuristic

An explicit formula used to approximate the cost of entering a cell in kD-tree structure

SDK Software Development Kit

A set of libraries, documents and code examples, giving a programmer access to some technology

SM Shader Multiprocessor

Single unit of hardware in nVidia GPU architecture, with several execution units and a shared instruction unit

Attachments

CD contents

thesis.pdf Electronic version of this thesis

src/ TeX sources for this thesis

cltracer/ Source code for clTracer

- **cl/** Source code for clTracer OpenCL kernels
- **doc/** Technical documentation of clTracer