FPGA Assisted Ray Tracing

Rendering large offline scenes

Master of Science Thesis

OLA BÅNGDAHL
MARKUS BILLETER

Department of Computer Engineering
Computer Graphics Research Group
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden, 2007

Abstract

Rendering detailed scenes with ray tracing is a very time consuming task. In this thesis we investigate if ray tracing can be accelerated using *field programmable gate arrays*, FPGAs. Our main focus is on offline rendering as opposed to real time rendering.

We aim to use the FPGA to offload expensive, re-occurring calculations from the CPU – i.e. rather than implementing a complete stand-alone ray tracer on the FPGA, the FPGA will take the role of a co-processor.

Sammanfattning

Att rendera detaljerade scener med “ray tracing” är en tidskrävande uppgift. Detta examensarbete har målet att undersöka om det är möjligt att snabba upp “ray tracing” med hjälp av programmerbara logikkretsar, FPGA:er. Vi riktar oss främst mot stora detaljerade scener och satsar därför inte på realtids rendering.

Vi ämnar att använda FPGA-kretsen som en stödprocessor till ett datorbaserat ray tracing system, där FPGA-kretsen utför tunga återkommande beräkningar med syftet att avlasta huvudprocessorn.

0.1 Preface

The goal of this thesis is to investigate the feasibility of using FPGA-circuits to accelerate ray tracing. The idea of using FPGA-circuits for ray tracing large, offline scenes originally surfaced at a start-up project, called Visionwell, at the School of Entrepreneurship at the Chalmers University of Technology. We were contacted to test the feasibility of this approach by implementing a test platform and performing measurements, in cooperation with the Computer Engineering departement at Chalmers.

We would like to thank the people involved in the Visionwell-project: Peter Wallberg, Maria Lundahl, Mattias Isaksson and Attila Bérces. We also would like to thank Mitrionics for their help and support, especially Fredrik Larsson and Henrik Abelsson. Finally we would like to thank our supervisor, Ulf Assarson at the Computer Engineering departement at the Chalmers University of Technology.

Special thanks to Attila Katona and Henrik Abelsson for providing us with some measurements on bandwidth and transfer overheads on the actual target hardware.

This thesis starts by introducing some important terms and concepts used throughout the remainder of the text. If the reader is already familiar with these terms (e.g. different types of parallellism), it is safe to skip the parts in question.

Finally, our test ray tracer and our co-processor design is presented in detail in the appendices.

Contents

This page is intentionally left empty.

1 Introduction

In this section we present the problem statement and the scope of this thesis. Also, some general terms and concepts are presented for those unfamiliar with ray tracing or field programmable gate arrays. A collection of relevant previous works in the fields covered by this thesis is included towards the end of this chapter.

1.1 Problem description

The main goal of this thesis is to investigate the feasibility of accelerating offline ray tracing using FPGA-circuits.

The original conceptual idea of accelerating offline ray tracing of large scenes was adopted by a project, *Visionwell*, at the Chalmers School of Entrepreneurship[1]. At this stage some specifications and guidelines were imposed on our work by the people partipicating in the project. We further limited the scope of the thesis during initial discussions with our supervisor and people involved in Visionwell.

1.2 Project specifications and scope

During initial investigations of the project by Visionwell contacts were established with Mitrionics who offered support and use of their products. The goal of using the products from Mitironics was to enable rapid development of a prototype.

Therefore, a requirement set by Visionwell was the usage of Mitrionics' products.

Due to the complexity of implementing a complete ray tracing system, we limit ourselves to building a basis for future development, and use this to perform measurments and try to estimate performance.

An other contributing factor to the limitation above are difficulties in performing tests on actual hardware: Mitrionics only supports a subset of available FPGA-circuits and restricts distribution of the actual FPGA compiler. The only way to test on actual target hardware is to send our software to Mitrionics, which is a time consuming task for both parts.

However, as a reference platform we choose the SGI RASC R100 Blade[2], which theoretically would have been available at the Mitrionics lab.

In order to limit the scope of our work, our focus is on the crucial part of ray tracing: ray intersections[3]. A set of limitations on the ray tracing process was thus introduced:

- simplest type of ray tracing [4],[5]
- no support for textures
- no support for reflections or refractions

In order to do initial tests, e.g. verifying various assumptions that we made early in our work, we decided to create a CPU-backend.

Our goal is to study usage of the FPGA as a co-processor, i.e. performing calculations in parallel with the host CPU. Generally, the idea is to perform "complex" operations, e.g. traversing data structures, on the CPU and offload simple, re-occuring calculations to the FPGA.

Since the Mitrionics FPGA development kit paritially supports IEEE 754 single precision floating point numbers[7] we use standard C-floats for representing floating point data. This also provides some degree of portability.

1.3 General Terms and Concepts

1.3.1 Parallellism

On a common CPU tasks are executed *sequentially*, i.e. a sequence of tasks is performed by completing one task at a time. Commonly these tasks are called instructions and may take different amounts of time to complete; for simplicity we will assume that the tasks take equal amount of time to perform. A sample procedure, which consists of M tasks, will thus require

$$t_{\text{procedure}} = M \cdot t_{\text{task}}$$

time to complete. If we wish to perform the procedure several times, the time required will instead be

$$t = N \cdot t_{\text{procedure}} = N \cdot M \cdot t_{\text{task}},$$

assuming we perform the procedure N times. A graphical example is shown in figure 1 where $M = 3$ and $N = 3$.

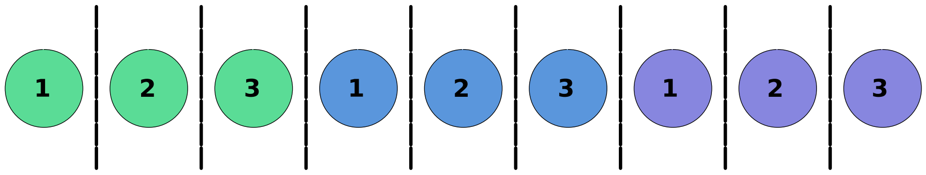

Figure 1: Illustration of sequencial execution; each vertical line indicates a time step, t_{task}. A procedure is three tasks long, and the complete program executes the procedure three times. The numbers indicate which task is executed during the current time step. For clarity each procedure is colored in its own color. The full program completes at $t = 9 \cdot t_{task}$.

In order to reduce the time required to perform all procedures, one can try to complete several procedures at the same time, i.e. in *parallel*. Of course, for this to work, the procedures must be independent from each other. Also one needs additional resources for parallel execution: for example, additional CPU:s. If the requirements are met, the total time required is reduced to

$$t = \text{ceil}\left(\frac{N}{num\_units}\right) \cdot t_{\text{procedure}} = \text{ceil}\left(\frac{N}{num\_units}\right) \cdot M \cdot t_{\text{task}}$$

Here *num_units* describes the number of times the procedure can be executed concurrently. An illustration of parallel execution is shown in figure 2. The parameters used in the example are: $M = 3$, $N = 3$, $num\_units = 3$.

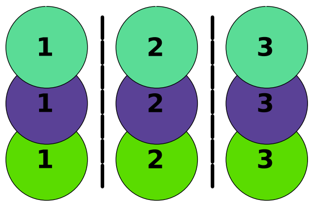

Figure 2: Example of parallel execution; each vertical line indicates a time step, t_{task}. A procedure is three task long, the complete program executes the procedure three times, the same number as the previous example for sequential execution. Again, each procedure is colored in its own color. The program completes at $t = 3 \cdot t_{task}$, as compared to $t = 9 \cdot t_{task}$ for the sequential version.

Another way of accelerating the execution of multiple procedures is by means of *pipelining*. Pipelining allows processing different tasks in different procedures concurrently. Assuming our sample procedure consists of M tasks we can set up M units, each responsible for executing a single task in the procedure: the first unit will execute the first task of the first procedure and send the results to the second unit. While the second unit processes the second task of the first procedure, the first unit can already start working on the first task of the second procedure. The total time required to complete the procedure N times will then be

$$t = (N + M) \cdot t_{\text{task}}$$

Illustration 3 shows an example for the case $N = 3$ and $M = 3$.

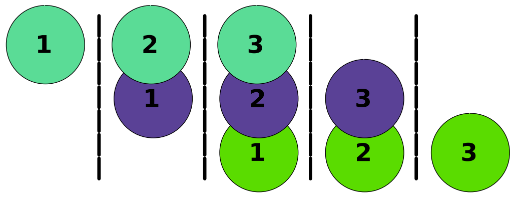

Figure 3: Example showing pipelined execution: each vertical line indicates a time step of length t_{task}. A procedure consists of three tasks, and the complete program executes the procedure three times, as in previous examples. This time the program finishes at $t = 6 \cdot t_{task}$.

Sometimes it is be possible to combine both pipeling and parallelism by executing independent tasks within a single procedure concurently.

While true parallelism reduces the time required by a procedure more than pipelining, it is also expensive: in the case of ordinary computers one usually has to add extra CPU:s.

1.3.2 Field programmable gate arrays

The *field programmable gate array*, FPGA, is a semiconductor device with programmable logic. The actual chip consists of an array of cells with programmable gates and interconnection logic: the gates can be programmed to implement any logic operations on a fixed number of inputs. The interconnection logic is used to connect the inputs and outputs of these gates.

Modern FPGA-chips also contain dedicated memory and multiplication units.

The strength of FPGA-circuits is that they can be reprogrammed many times after manufacture, unlike *application specific integrated circuits*, ASICs, whose functionality is determined during manufacturing. Commonly, FPGAs are programmed using hardware description languages (HDL): the two most prominent ones are *VHDL*[8] and *Verilog*[9]. Working with a HDL is quicker and requires less knowledge of electrical engineering than designing ASICs.

HDLs are still relatively low-level, compared to "common" programming languages like C or C++. Some higher-level languages which can be used to program FPGAs have been developed, e.g. Handel-C[10] and Mitrion-C[11].

For the development of our prototype ray tracer we were required to use Mitrion-C, which is introduced below.

1.3.3 Ray tracing

Ray tracing is a well-known method of rendering 3D-scenes within the field of computer graphics.

The basic algorithm can be summarized as follows:

1. Select a viewing position and an image plane
2. Create a ray originating from the viewing position and piercing a fragment ($\approx$ pixel) in the image plane
3. Trace along the ray until it hits an object in the scene or exits the scene
4. If no objects were hit, set the color of the fragment in step 2 to zero, otherwise calculate a color (*shade*) depending on the intersection point.
5. Repeat from step 1 until all fragments in the image plane have been colored.

Usually, the shading step involves many additional calculations: for instance finding whether the point in question is lit or shadowed.

Many introductionary texts to ray tracing exist: e.g. [5] or [6].

1.4 Mitrion-C

Mitrion-C is a high level language[1] used to program FPGA-circuits. The language has a syntax similar to C, thus requiring less knowledge of hardware design compared to programming with a HDL. Mitrion-C is developed to allow easy expression of parallelism, and can therefore utilize the parallel nature of FPGA-circuits.

Mitrion-C further simplifies development by providing built-in functions for memory access, synchronization and floating point arithmetic. In contrast, if

[1] "High level" compared to common HDL languages.

one were to develop directly in a HDL, it is likely one would have to implement memory controllers, synchronization objects and floating point logic by themselves or aquire relevant "IP-blocks" which provide the required functions.

One major difference of Mitrion-C compared to regular C/C++ is the lack of language-level support of pointers and references. Therefore it is cumbersome to implement most data structures ordinarily found in C/C++. Hardware description languages commonly do not have a concept of pointers; memory banks which can be addressed by "pointers" can however be implemented in a HDL.

In order to control and communicate with the Mitrion-C applications running on a FPGA from a PC ("host") a hardware abstraction layer, HAL, is provided through Mithal[12]. Mithal provides a C/C++ interface for programming and running the FPGA. Functions for sending and reciving data are also provided.

Finally, a Mitrion-C simulator is supplied with the Mitrion development suite. The simulator can be used to test and analyze Mitrion-C applications on a ordinary PC, reducing the need for potentially expensive FPGA-chips during initial development. The simulator application is work in progress; for instance only incomplete information about memory accesses and synchronization is provided.

An especially useful feature of the simulator is its ability to display data-flow and dependecies in applications, see picture 13 in appendix B.

1.5 Previous Work

General ray tracing Ray tracing is a well known subject, first used for visible surface detection [4]. Modern ray tracing with support for reflections, refractions and shadowing was later introduced by [13]. Good introductions to ray tracing can be found at e.g. [5] [6].

Current reasearch in ray tracing often involves introducing data structures in order to improve performance: a popular choice of spatial data structures are Kd-trees[14] [15]. Other possibilities include e.g. octrees[16] and grids[17] [18], our choice of spatial data structure.

A method for accelerating large scenes is described in [19] [20], where rays are kept in memory and objects are processed sequentially in order to improve memory locality: this is similar to our approach where rays are collected in memory and grouped according to spatial location. Other attempts at accelerating ray tracing include exploitation of ray coherence[21] [22] [23], which is outside the project scope.

Hardware assisted ray tracing As specialized hardware has become more accessible, there have been attempts at accelerating ray tracing using this hardware. There are two main approaches: reusing existing hardware and developing specialized solutions.

In [24] an approach similar to ours is used: "the ray engine" mainly accelerates ray-triangle intersections. However, in comparison to our work, GPUs were used as the hardware platform, rather than FPGAs.

FPGAs are often used as an experimental platform for development towards specialized solutions (e.g. ASICs) [25]. Also, often the goal of this research is to apply ray tracing to real time rendering [26] [27]. In constrast, our aim is to accelerate offline rendering of large static scenes.

A common problem with ray tracing is memory bandwidth [28]; one solution is the send data with less precision [25]. However, for simplicity and comparability we have opted to use full 32-bit precision floats.

2 Implementation

This chapter describes our implementation of the test ray tracer. Here we describe a high level view of our implementation: the basic design and its different parts. Implementation specific details like building and running the test system are described in appendix A. Appendix B is dedicated to our implementation of the FPGA co-processor application, and finally, information about the data formats used are documented in appendix C.

An analysis of our implementation in found towards the end of this chapter.

2.1 Overview

Figure 4 illustrates a coarse overview of the prototype system. The main components on both host and FPGA are shown; these components will be described in more detail later in this chapter.

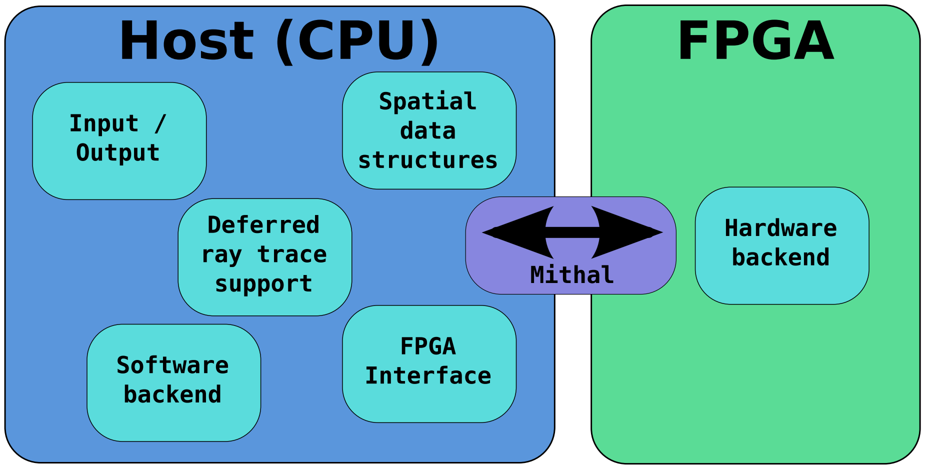

Figure 4: Symbolic overview of the test ray tracing system.

2.2 Spatial data structures

Our implementation uses a hierachial approach with two tiers of grids[17]. The grid traversal algorithm used here is described in [29].

The tier one grid, the *scene grid*, essentially acts as a scene graph, sorting all objects in the scene spatially. The scene grid contains references to meshes[2]: each unit element, *voxel*, in the grid stores references to *mesh instances* that intersect with the voxel.

Each "mesh instance" contains a reference to the actual mesh data for this instance, and an associated transform, which can be used to transform from/to world-space to/from a space local to the mesh.

The second tier of grids is applied at mesh-level: each mesh is subdivided into a grid during preprocessing. At this level, each voxel contains a list of triangles that are contained by the voxel or intersect with the voxel. We refer to the second tier of grids often as *mesh grids*. A two dimensional schematic of this type of hierarchial grids is shown in figure 5.

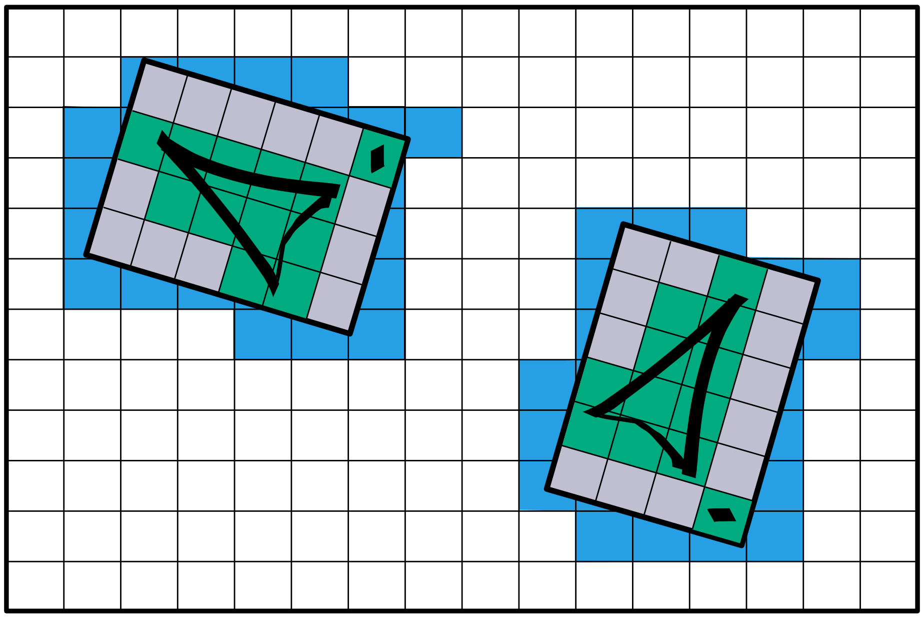

Figure 5: Coarse schematic of the hierarchial grid data structure. The large outer grid in the figure represents the scene grid. Voxels in the scene grid that intersect with objects placed in the scene are shown in a light blue. Due to the use of mesh instances, a single mesh can be placed twice in the scene grid, with different transforms. The preprocessed mesh grids are shown as oriented boxes with a gray background. Voxels in the second tier mesh grids that actually contain parts of the mesh are colored turquoise.

Ray tracing against this structure is performed through the following steps:

```
ray = get current ray;
scene_grid = get tier one grid from scene graph;

do
{
   scene_voxel = traverse_to_next_voxel( ray, scene_grid );
```

[2]A *mesh* refers to the data that describes a 3D model. In our case, meshes only contain vertices (points in a 3D space) and normals (vectors which describe surface orientation). The vertices are connected to build triangles that define the surface of the 3D model in question.

```
   for each( mesh_instance in scene_voxel )
   {
      mesh_ray = mesh_instance.world_mesh_transform * ray;

      do
      {
         mesh_voxel = traverse_to_next_voxel( mesh_ray,
            mesh_instance.mesh_grid );

         hit = ray_trace( mesh_ray, mesh_voxel.raw_mesh );

         if( hit )
            process_hit();
      }
      while( voxels left in mesh_voxel );
   }
}
while( voxels left in scene_grid );
```

By using the same spatial data structure for all levels in the scene hierarchy, we can reuse large parts of code. The scene grid is constructed on the fly, whereas meshes are preprocessed offline. In both cases there are some problems with objects (meshes or triangles respectively) intersecting with several voxels. Two possible solutions exist for this: either one can subdivide the objects to fit in each voxel exactly, or simply add the original object to all voxels it intersects with. The second approach requires some extra checks during traversal to avoid sorting errors. We choose to use the second approach.

2.3 Input / Output

The input/output module is responsible for loading data into the data structures used internally in the application, and after successful ray tracing it should store the results.

Input to the ray tracing application consist of

- a scene definition
- a view definition
- at least one object definition
- raw mesh data

The scene definition is a XML file which contains information about object placement and lights. The scene definition is described in appendix C.2. Along with the scene definition, a view definition must be specified which contains information about viewpoint and output, see appendix C.1.

Objects consist of two parts: the object definition and the raw data. The object definition is a XML file with references to the meshes which build the object. Information required to reproduce a precalculated grid data structure is also contained. The raw data contains the actual vertices and normals of the meshes. A reference of the object format can be found in appendix C.3.

Two external utilities exist for pre- and postprocessing input and output data: `meshconv` and `fbconf`. The `meshconv` application is used to preprocess meshes from *.3ds*-model files into the internally used grid representation. The `meshconv` application is described in more detail in appendix C.4.

Output images consist of a very simple format where each color channel is described by single precision floating point numbers. The `fbconv` utility can be used to convert these into *.tga* image files. A detailed description can be found in appendix C.5.

2.4 Deferred ray trace support

In order to increase efficiency in the communication with FPGA one wishes to process large batches of data during each run of the FPGA application. With larger batches there is more data to process on the FPGA, over which delays introduced by moving data to the FPGA and activating the FPGA can be amortized.

The deferred ray trace module therefore creates collections of rays and intersecting voxels, by traversing rays through the spatial data structures until a node containing an actual mesh is hit. Once a number of rays intersect with a single mesh voxel, the raw mesh of that voxel can be queued for processing, along with the rays currently intersecting with the voxel. Figures 6 and 7 demonstrates these steps.

During development it became apparent that batch sizes still had large variations; in order to work around this, batches are further collected into *tasks*. A task consists of several batches; the FPGA application is capable of processing a task in a single run. A overview of both batches and tasks is shown in figure 8.

2.5 FPGA Interface

The FPGA interface is responsible for the acutal communication with the FPGA. Data structures produced by the deferred ray tracing module need to be packed into a buffer that can be sent to the FPGA. Later, the data recieved from the FPGA need to be decoded and interpreted.

Actual communication with the FPGA is facilitated by the *Mithal* API [31]. By using Mithal it is possible to use the same interface for both the target FPGA and the simulator provided by Mitrionics.

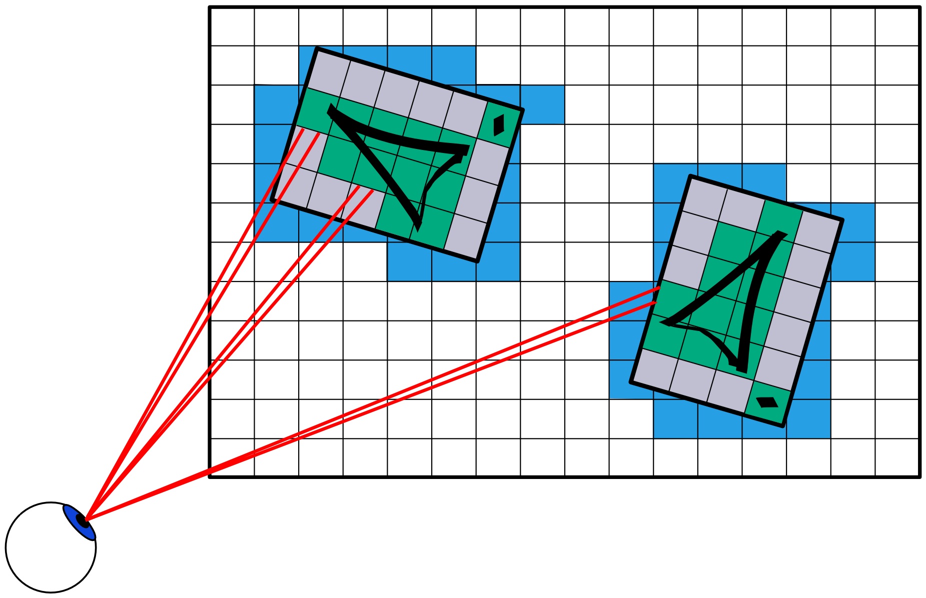

Figure 6: Rays are initially cast into the spatial data structure. The intersection with the first non-empty voxel is found, and the rays are associated with that voxel.

2.6 Software backend(s)

Two software backends exist: one which performs ray tracing directly on the data structures in the scene, without using the deferred tracing module. This backend was used mostly for debugging and profiling purposes.

The second backend uses deferred ray tracing for gathering statistics on batches which would have been generated during a real ray tracing pass.

2.7 FPGA Implementation

The hardware backend for the test ray tracer system is implemented on the FPGA; its sole purpose is to perform ray-triangle intersection tests in order to find the closest intersection. The FPGA implementation can be roughly partitioned into three parts:

1. buffering rays from a single batch into local high-speed memory
2. performing intersection calculations for each combination of rays and trianlges in the batch
3. finding the closest intersection point for each ray

These three parts are preformed for each batch in the current task.

During the first stage, ray data is read from the FPGA-circuits dedicated SRAM and buffered in local high-speed memory banks. This extra step was introduced in order to avoid reading data from the (slower) SRAM several times.

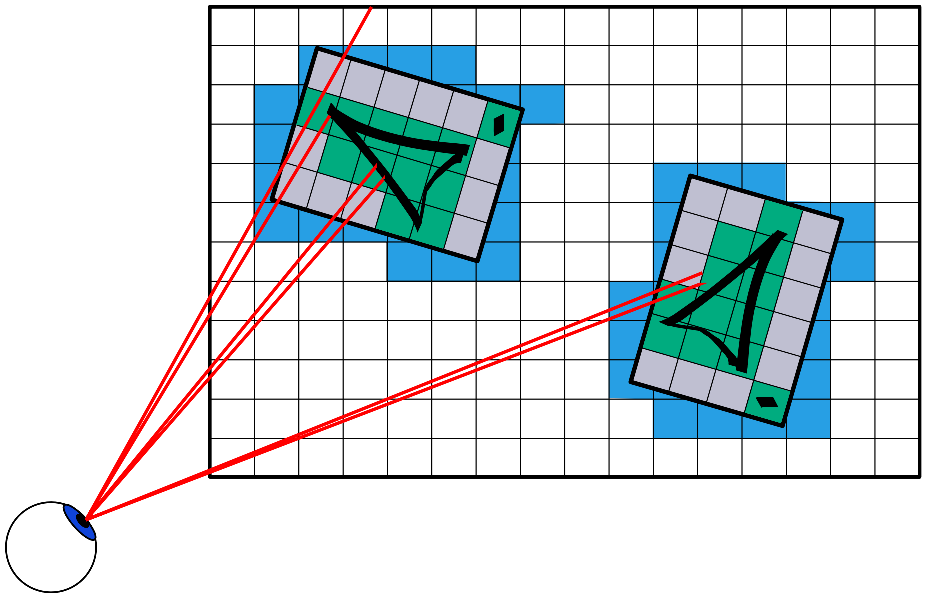

Figure 7: Once a batch is processed, actual intersection tests are preformed. Unless a hit is registered, the next non-empy voxel along the ray is found and the process repeated until a ray either hits or leaves the scene.

The intersection tests are then performed by reading a triangle from SRAM and testing all rays against this triangle. Once all triangles have been processed, the closest intersection for each ray has to be found. Data about this intersection is then written back to the SRAM memory, where it is accessible by the host applicaiton.

The following pseudocode discribes this procedure.

```
ray_buffer = buffer_all_rays();
intersections = {};

do
{
   triangle = read_triangle();

   for each ray in ray_buffer
   {
      isect = intersect( ray, triangle );
      add isect to intersections;
   }
} while( unprocessed tringles left );

for each ray
{
   result =
      find_closes_intersection( intersections );
```

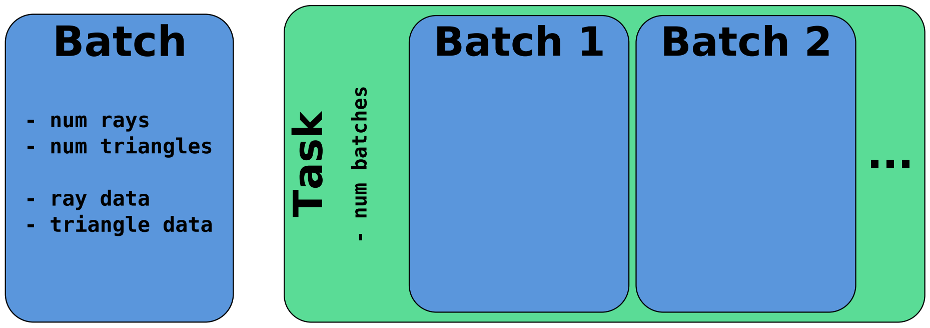

Figure 8: Left: overview of a single batch. A batch can contain both multiple rays and multiple triangles. Right: overview of a task. Batches are packed further by collecting a number of batches into a task, which then can be sent to the FPGA for processing.

```
   write( result );
}
```

Note that the above procedure is heavily pipelined and parallelized. For instance, the innermost loop is pipelined, with the added possibility of executing it in parallel, see figure 9. However due to space constraints on the FPGA we have been forced to use a single pipeline only.

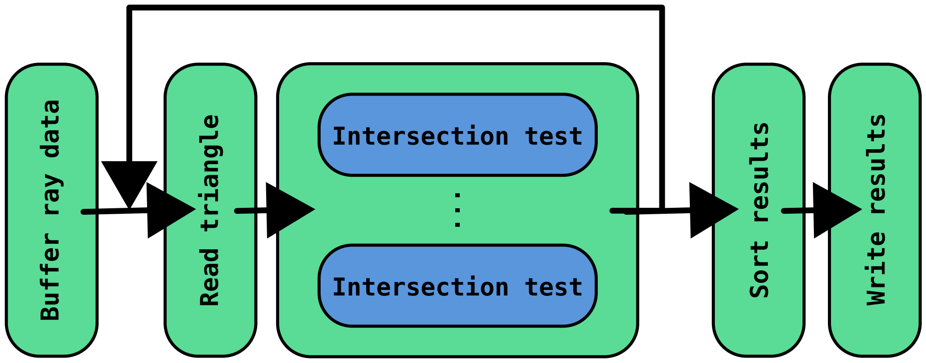

Figure 9: Flowchart overview of the FPGA backend. Note the possibility for several ray-triangle intersection tests in parallel.

The algorithm to find intersections between rays and triangles is presented in [32] [25]. If an intersection is found, the algorithm returns the distance to the intersection along the ray, and the barycentric coordinates, u and v, describing intersection location on the triangle.

2.8 Analysis

Despite the limited functionality of our software there are numerous tests one could perform. The tests we have choosen to run fall mainly into one of the following three categories:

- verification of assumptions

Table 1: Data on the reference scenes

	"Colony Ship"	"Dragons"
Meshes	12	3
Instances of each Mesh	3	2
Total vertices	388035	109662
Total triangles	689322	217716
Lights	2	4
Target vertices/voxel	10	10
Scene definition	colony-10.scene	dragon-10.scene
View definition	colony.view	dragon.view

- gathering statistics using the CPU-implementation
- gathering statistics using the FPGA-simulator software

2.8.1 Test setup

Tests performed on the CPU-implementation involved ray tracing one of two reference scenes. Unless otherwise noted, all tests use the following settings:

- Framebuffer resolution: 800×600 pixels
- Multisampling: 4x (i.e 16 samples per pixel)
- Simple shading with shadows

Data on the two reference scenes is presented in table 1. Figures 10 and 11 show renderings of the scenes, using the settings described here.

Various numbers of target vertices per voxel were tested, and 10 turned out to yield good results. One should note that the algorithm performing the subdivision into grids is far from perfect[3], causing non-empty voxel to contain up to ten times as many triangles on average.

2.8.2 Preliminary tests

The purpose of the preliminary tests is to verify some assumptions we have made: e.g. the main assumption in this thesis is that the most time-consuming task in ray tracing is performing ray-intersection tests.

Table 2 summarizes the time spent in the various parts of our algorithm.

2.8.3 Statistics for deferred tracing

In order to estimate the workload transferred from the CPU to the FPGA, we measured the number of batches created during deferred tracing, and average numbers of rays and triangles in each batch. This data is summarized in table 3. Note that, due to resource usage, only the primary rays generated with 2x multisampling are included in these statistics.

[3]It assumes that the triangles are distributed uniformly inside the bounding box volume of the meshes, which is seldom the case.

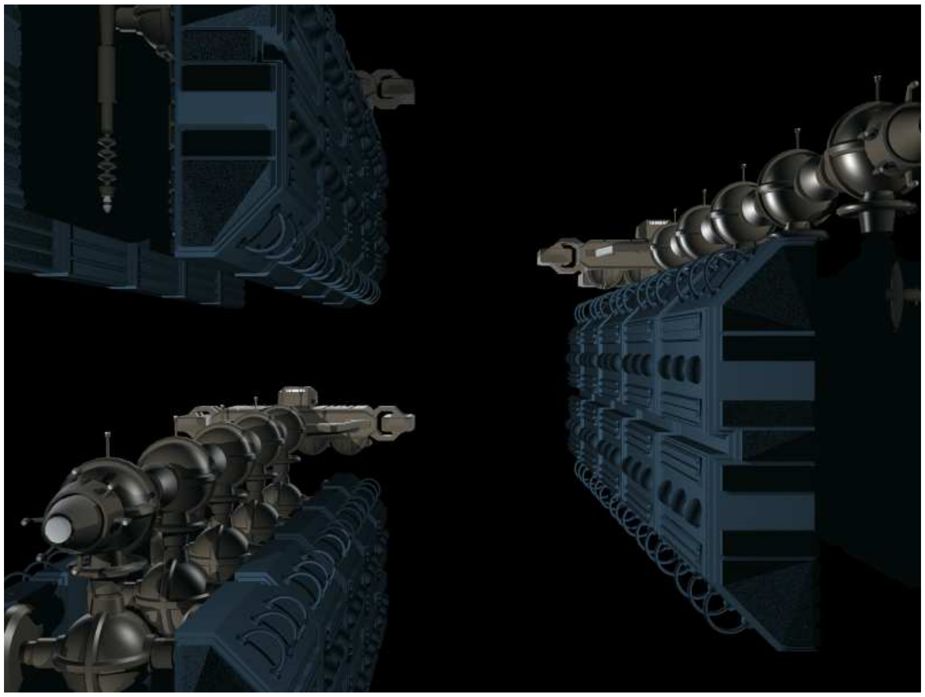

Figure 10: Rendering of the scene "Colony Ship"

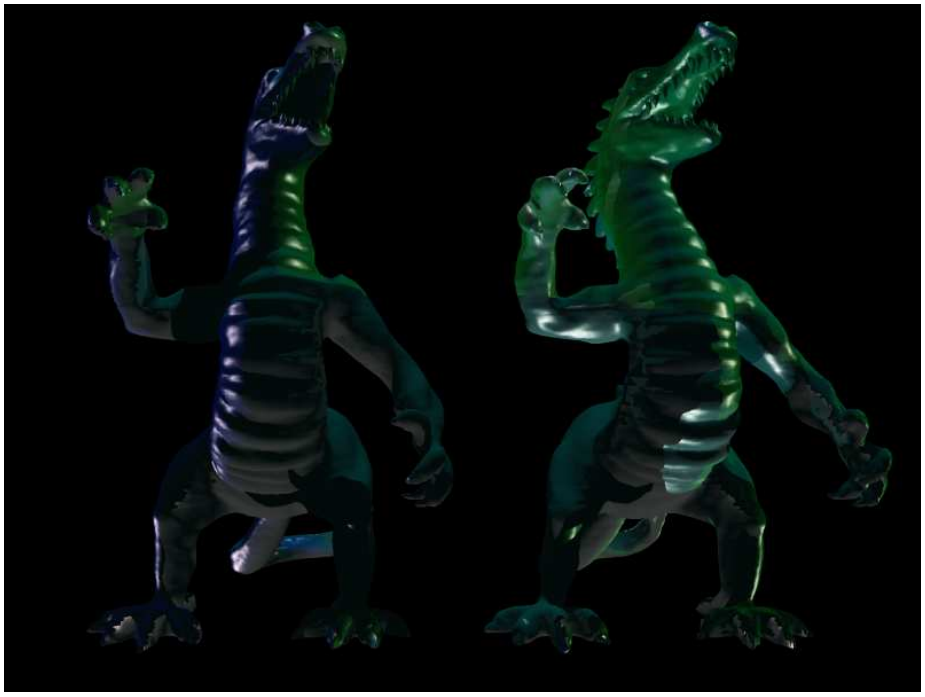

Figure 11: Rendering of the scene "Dragons"

Table 2: Profiling data for preliminary tests. All numbers are given in percent of the total execution time. The group "Other" includes other tasks, like loading data, shading and profiler overhead.

	"Colony Ship"	"Dragons"
Grid traversal, primary rays	18.1%	9.5%
Intersection tests, primary rays	28.1%	13.2%
Grid traversal, shadow rays	14.9%	21.6%
Intersection tests, shadow rays	19.5%	40.0%
Other	19.4%	15.7%

Table 3: Statistics on batches generated by the deferred ray tracing algorithm for primary rays only. Each voxel hit by at least one ray will generate at least one batch.

	"Colony Ship"	"Dragons"
voxels hit by primary rays	2840	928
average triangles per batch	47.3	55.3
average rays per batch	863.6	898.2
maximal triangles per batch	587	235
maximal rays per batch	>10000	3522
largest batch	60099 bytes	21267 bytes

2.8.4 Statistics for the FPGA routines

Statistics for the FPGA routines were gathered using the FPGA simulator software (due to lack of access to the actual hardware). The FPGA simulator does not provide data on overheads incurred by transfering data from/to the FPGA; we were able to get some information on this from Attila Katona at Evopro and Henrik Abelsson at Mitrionics.

Figure 12 shows plots over the number of steps required for various operations. A step in the simulator roughly corresponds to a single cycle on the corresponding target FPGA, however it is important to remember that overheads from memory accesses are **not** included in the measurements[4].

Our tests show that the number of steps required by the FPGA grow linearly with the number of triangles and/or rays; redistributing rays and triangles between batches does not seem to affect total execution time significantly.

According to the simulation software, our program will be able to execute at 100MHz on the target FPGA; correspondece with Henrik Abelsson indicated that transferring a buffer of 8MB will take approximately 12ms.

A test run on the simulation software[5] with a single batch of 875 rays and 50 triangles each gave us the following results: 43750 intersection tests take 199281 steps on the FPGA. Using this number we can perform the following estimate, which also attempts to take memory overheads into accound:

- An 8MB buffer can hold data for 16.1M intersection tests

[4]A severe limitation in the simulation software...

[5]During this test run it became painfully obvious that attempting to ray trace a complete scene using the simulation software would not be feasible due to overheads from the simulation: processing the data from a single typical voxel took over 20 minutes. For a complete scene, we would have to run simulations with at least 900 voxels/batches.

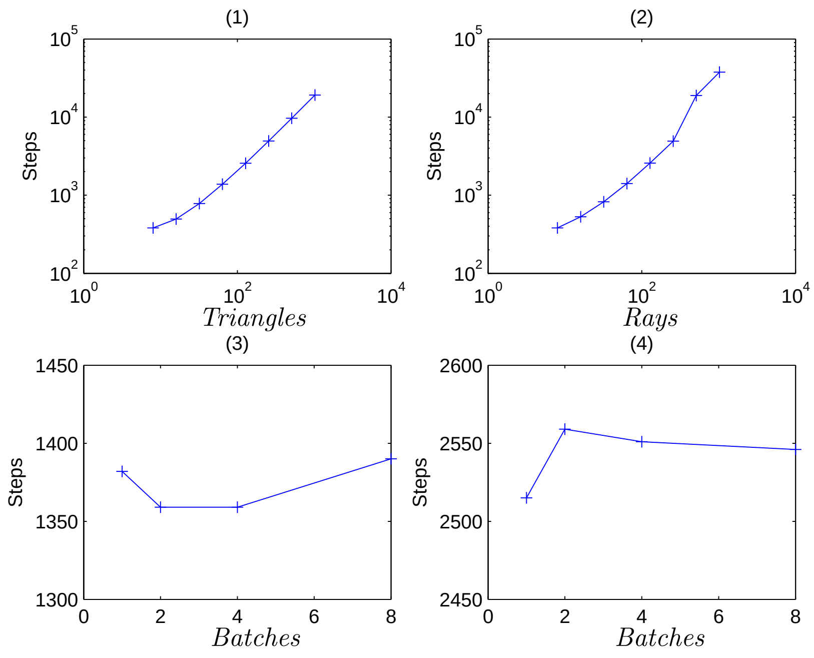

Figure 12: Numbers of steps required by various operations: (1) steps performed for a varying amount of trianges, while keeping the number of rays and voxels constant (four and one respectively). (2) steps performed for a varying amount of rays, while keeping the number of triangles and voxels constant (eight and one respectively). (3) & (4) steps performed for varying the number of batches, but keeping the total number of intersection tests constant. Triangles and rays are distributed differently over the voxels.

- Performing 16.1M intersections tests will take approximately 73.3M steps, or 0.73 seconds.
- The overhead of transfering the data and running the program amounts to 12 ms.

I.e. we can perform approximately 21.6M intersection tests per second[6].

3 Discussion

In this part we inspect various aspects of our system and discuss some of the weak points and explore possible improvements.

3.1 Bottlenecks

Ray tracing large scenes potentially requires large amounts of data to be processed. In our case, this data has to be moved to and from the host system to the FPGA; in order to actually accelerate ray tracing, the FPGA has to not only outperform the CPU, but also overcome the overhead of transferring data. For offloading calculations to be advantageous one has to fullfill the following requirement:

$$time_{\text{cpu}} > time_{\text{fpga}} + overhead_{\text{send}} + overhead_{\text{recive}}$$

The data sent to the FPGA consists of six rays, triangles and some control-data[7]. Each ray requires six floats and each triangle nine floats. Four floats are returned for each ray: distance to hit, the u and v coordinates and a index identifying the triangle in the current batch.

A bottleneck is introduced by the way Mithal[8] handles data transfers and buffers; the amount of data transfered each run is static[9]. For example, our program will send more data to the FPGA than is returned from the FPGA, however as the complete buffers are transferred in each transaction, large amounts of "scrap-data" will be sent from the FPGA to the host system.

The following example will show a case where 200 rays and 1000 triangles are sent to the FPGA for processing, i.e. $n_{rays} = 200$ and $n_{triangles} = 1000$:

$$\begin{aligned} size_{\text{input}} &= (n_{\text{rays}} \cdot 6) + (n_{\text{triangles}} \cdot 9) = 10200\,\text{floats} = 40800\,\text{bytes} \\ size_{\text{output}} &= (numRays \cdot 4) = 800\,\text{floats} = 3200\,\text{bytes} \end{aligned}$$

This means that we send $size_{\text{output}} - size_{\text{input}} = 37600\,\text{bytes}$ of unused data back from the FPGA. Sending more rays than triangles per batch would reduce this problem somewhat.

[6]Compared to approximately 12M intersection tests per second on a 1.66GHz Core Duo processor using the same method. Note that the method is not optimized to take advantage of SSE vectorization, however it assumes that the data is arranged in a linear manner in memory.

[7]The amount of control-data is however neglible assuming batches are relatively large.

[8]Mithal is in turn limited by RASClib, which is the underlying communication library.

[9]In other words, the size of the buffers is determined during compile-time and can not be changed inbetween runs.

As can be seen in table 2, a non-trivial amount of time is spent traversing the grid structure. On one hand, one could try to move the grid traversal to the FPGA, but this would only move the problem from the CPU to the FPGA. Currently, the various grids contain many empty voxels, a side effect from the relatively fine subdivision. Using other, more adaptive, data structures, e.g. octrees and kd-trees, could minimize this problem.

Moving grid traversal has some other positive properties, e.g. the technique described in [19] could be applied directly in order to minimize the amount of data transferred to the FPGA. This, however, requires that the models are "small enough" to completly fit into the memory on the FPGA platform.

3.2 Future work and improvements

There are many possible improvements and during this thesis many ideas were discussed. Generally, the next step would be to actually implement a hardware backend, preferably using actual FPGA hardware. However, some data might be gleaned from implementing a backend targeting GPUs instead, assuming that access to a FPGA remains elusive.

Having continous access to the hardware would also simplify further development, as one would no longer have to make educated guesses on which parts of the process might be viable to move to hardware.

Many of the features and improvements described here were not included as they fall outside the scope of the project or we were unable to implement them in the given time frame.

3.2.1 Software Implementation

As described in chapter 2, we currently employ the simplest ray tracing algorithm; alternative choices of ray tracing algorithms are possible, depending on the goal: visual improvements can be achieved by using e.g. photon mapping [33]. Rendering could possible be further accelerated by using some of the techniques described in chapter 1.5, e.g. exploiting coherence in ray-space [21] [23].

We belive that our work can be applied, with some additional work, to most ray tracing techniques.

Further improvements may be gained by using different spatial data structures, e.g. kd-trees[14] or octrees[16]. Our approach limits possible choices of data structures somewhat, as we require the possibility of associating many rays to many triangles in order to build batches which can be efficiently transfered to and processed on "external" hardware.

Also, a problematic part of the software implementation is the asynchronous nature of the communication between the FPGA and the software program. The Mithal API provides no facilities for polling the status of the FPGA, rather one can only choose to wait for calculations to finish, which leads to pipeline stalls: the CPU is forced to wait for the FPGA and vice versa. As a result, less time is spent with both units performing calculations in parallel, which is suboptimal.

Unfortunately, we have not found any FPGA communications API which allows us to poll the state of the FPGA; a workaround would involve using multiple threads in the software part.

3.2.2 FPGA Implementation

The following list presents some improvements that we have discussed during development:

- A further improvement of the throughput on the FPGA could be achieved by adding more intersection pipelines.
- Better use of onchip resources. Our current solution uses the onchip local high speed memory for caching rays suboptimally: instead of creating a dedicated area for each batch, one single large area could be reused for all batches.
- Optimize the Mitrion-C code in order to reduce the onchip space. The free space can then be used for other tasks, or possible for further parallelization of the current program, for instance by adding more pipelines.
- During our thesis, the Mitrion-C system has been under development concurrently and new functions and features have been added. Some of these new functions have helped us greatly. Reviewing every new release of Mitrion-C and Mithal would be a good way to ensure that the implemenation stays optimal.
- Re-design the co-processor in a hardware description language, for example VHDL. The downside would be the work effort involved; the benefits would be that we could choose the hardware platform more freely, and the control over details, which are hidden when a high level programming language is used, would be regained.
- Naive optimizations of the FPGA platform include using lower precision data. FPGAs are not limited to fixed size variables, but the programmer can choose precision (in terms of bits) relatively freely. Using lower precision data will not only reduce the bandwith requirements, but also free up resources on the FPGA-circuit, such as dedicated multipliers.

  Reducing precision has the sideeffect that the host program becomes slightly more complex, i.e. rather than being able to send the raw data to the FPGA, all data must be converted to a non-native format (e.g. 24-bit floating point numbers, as used in [25]).
- Data could be transferred using ping-pong buffers, i.e. using two communication buffers concurrently, where one buffer is filled by the host, while the other buffer is processed by the hardware in parallel. This would reduce potential pipe-line stalls.
- Implement more functionality on the FPGA, e.g. move grid-traversal on a per-mesh level to the FPGA. While possible, this would require more space on the FPGA-circuit and may also consume more memory.

- A special property of FPGAs is the ability to fully reprogram them on the fly. One could take advantage of this by writing several different programs, each specialized for a different stage of the rendering process, and reprogram the FPGA on demand. A possible improvement could include writing a special program for casting primary rays. As primary rays can be calculated easily, it is not necessary to transfer any ray data to the FPGA: the rays could be generated by the FPGA during run time. Another special program could be written for shadow rays. The test for shadow rays involves only checking if a light source is occluded, i.e. the trace can be aborted as soon as a intersection is detected.

3.2.3 Hardware platform

This section deals with possible further development on the hardware side. Since we spent least energy on this subject in the thesis, it is also the area with most headroom for improvements.

- Investigate other supported platforms[10] for optimizing the data throughput in the system.
- Utilizing more resources on the hardware platform such as large banks of DDR-DRAM[11] on the SGI RASC platform [34].
- Something that falls completly outside the scope of the original idea would be to implement a backend that runs on GPUs. This could used to analytic tool for evaluating the concept or even a complement to the FPGA platform.
- Develop application specific hardware. This could be a new specific FPGA platform or even creating a ASIC to replace the FPGA.

4 Conclusions

In this thesis we have built a test platform with the goal to evaluate the feasibility of accelerating ray tracing using FPGAs. A large part of the development time was spent getting acclimatized to the Mitrion platform, a choice of platform over which we initially had little control. Also, this choice limited our possibilities of testing the program on actual hardware, which would have been advantageous during development.

Development of this test platform has given us insights into the subject at hand, e.g. approaches that might turn out to be feasible when attempting acceleration of ray tracing using external hardware in general, not necessary only FPGAs. Much of this information is documented in the chapter 3.2 “Future Work”, along with possible improvements of the software side of things.

Our investigations show that, with some additional work in the future, FPGA-accelerated ray tracing is possible. However, with the current trend of more

[10]E.g other platforms supported by Mitrionics if one were to keep developing in Mitrion-C.
[11]Instead of the SRAM currently provided.

powerful and general GPU:s and stream processing units[12], and the relative cost of FPGAs compared to these technologies, using FPGAs for this task might not be viable.

Future work could directly expand on the framework presented here, albeit some major parts remain unsolved (unimplemented).

[12]An example here is the relatively recent announcement by AMD[35] to develop CPUs with an integrated GPU.

References

[1] Chalmers School of Enterpreneurship, *http://www.enterpreneur.chalmers.se*, December 17, 2007

[2] SGI, "SGI Reconfigurable Application Specific Computing: Accelerating Production Workflows", 2006

[3] HyperGraph: Accelerating Ray Tracing, *http://www.siggraph.org/education/materials/ ...*
... HyperGraph/raytrace/rtaccel.htm, December 17, 2007

[4] A. Appel, "Some techniques for shading machine renderings of solids", *Spring Joint Computer Conf*, pp. 37-48, 1968

[5] E. Haines, T. Moller, "Real-Time Rendering (2nd Ed)", *Natick: A.K. Peters Ltd*, 2002

[6] HyperGraph: Ray Tracing, *http://www.siggraph.org/education/materials/HyperGraph/toc.htm*, December 17, 2007

[7] *IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985)*

[8] VASG: VHDL Analysis and Standardization Group, *http://www.eda.org/vhdl-2000x*, December 17, 2007

[9] IEEE System Verilog WG, *http://www.eda.org/sv-ieee1800*, December 17, 2007

[10] I.Page, M. Spivey, "How to Program in Handel", Technical Report, *Oxford Univerity Computing Laboratory*, 1993

[11] Mitrionics, *http://www.mitrionics.com*, December 17, 2007

[12] Mitrion SDK, *http://www.mitrionics.com/default.asp?pId=23*, December 17, 2007

[13] T. Whitted, "An improved illumination model for shaded display", *Proc. 6th annual Conference on Computer Graphics and Interactive Techniques*, pp. 14, Aug. 1979

[14] J. Bentley, "Multidimensional binary search trees used for associative searching", *Communications of the ACM*, pp. 509-517, Sept. 1975

[15] I. Wald, V.Havran, "On building fast kd-trees for ray tracing, and on doing that in O(N log N)", *Proceedings of the 2006 IEEE Symposium Ray Tracing*, pp. 61-69, 2006

[16] A. Glassner, "Space subdivision for fast ray tracing", *IEEE Computer Graphics and Applications, Vol4, Number 10*, pp. 15-22, Oct. 1984

[17] A. Fujimoto, K.Iwata, "Accelerated Ray Tracing", *Proc. CG Tokyo '85*, pp. 41-65, 1985

[18] D. Jevans, B. Wyvill, “Adaptive Voxel Subdivision for Ray tracing”, *Proc. Graphics Interface ’89*, pp. 164-172, June 1989

[19] K. Nakamaru, Y. Ohno, “Breadth-first ray tracing utilizing uniform spatial subdivision”, *IEEE Transactions on Visualization and Computer Graphics 3*, pp. 316-328, Oct. 1997

[20] K. Nakamaru, Y. Ohno, “Enhanced breadth-first ray tracing”, *Journal of Graphics Tools 6*, pp. 13-28, 2001

[21] J. Amantides, “Ray tracing with cones”, *Proc. 11th annual Conference on Computer Graphics and Interactive Techniques*, pp. 129-135, 1984

[22] A. Reshetov, A. Soupikov, J. Hurley, “Multi-Level Ray Tracing Algorithm”, *ACM SIGGRAPH 2005 Papers*, pp. 1176-1185, 2005

[23] I. Wald, T. Ize, A. Kensler, A. Knoll, S Parker, “Ray Tracing Animated Scenes using Coherent Grid Traversal”, *ACM SIGGRAPH 2006 Papers*, pp. 485-493, 2006

[24] N. Carr, J. Hall, J. Hart, “The Ray Engine”, *Proc. ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pp. 37-46, Sep. 2002

[25] J. Fender, J. Rose, “A High-Speed Ray Tracing Engine Built on a Field-Programmable System”, *IEEE International Conf. On Field-Programmable Technology*, pp. 188-195, Dec. 2003

[26] S. Woop, J. Schmittler, P. Slusallek, “RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing”, *ACM Transactions On Graphics*, pp. 434-444, 2005

[27] J. Schmittler, S. Woop, D. Wagner, W. Paul, P. Slusallek, “Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip”, *Pro. ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pp. 95-106, 2004

[28] C. Benthin, I. Wald, M. Scherbaum, H. Friedrich, “Ray Tracing on the CELL Processor”, *Proc. IEEE Symposium on Interative Ray Tracing*, pp. 15-23, 2006

[29] J. Amanatides, A. Woo, “A Fast Voxel Traversal Algorithm for Ray Tracing”, *Proc. Eurographics ’87*, pp. 3-10, 1987

[30] T. Purcell, “Ray Tracing on a Stream Processor”, *Ph.D dissertation, Standford University*, March 2004

[31] Mitrionics, “The Mitrion Host Abstraction Layer API”, *http://www.mitrionics.com/*, December 17, 2007

[32] T. Möller, B. Trumbore, “Fast, Minimum Storage Ray-Triangle Intersection”, *Journal of Graphics Tools*, pp. 21-28, 1997

[33] H. Jensen, “Global Illumination using Photon Maps”, *Proc. Eurographics Workshop on Rendering Techniques ’96*, pp. 21-30, 1996

[34] SGI, *http://www.sgi.com/pdfs/3920.pdf*, December 17, 2007

[35] Advanced Micro Devices, *http://www.amd.com*, December 17, 2007

A The test ray tracer

This section describes the prototype implementation in detail. Also included is a short section explaining how to run and build the application.

A.1 Building the system

Building the software developed during this thesis requires a number of external tools and libraries (which are not included in the distribution). Also, development was mainly performed in GNU/Linux systems, and it is thus assumed that the target platform is a GNU/Linux compatible system[13].

The following (freely available, open-source) libraries and tools are required:

- SCons, *http://www.scons.org*
- Python, *http://www.python.org* (required by SCons)
- Boost, *http://www.boost.org*
- GCC, *http://gcc.gnu.org*, preferably version 4.0 or newer

The following external libraries and sources are included in the distribution:

- TinyXML, *http://www.grinninglizard.com/tinyxml/*
- TinyXPath, *http://tinyxpath.sourceforge.net/*
- Lib3DS, *http://lib3ds.sourceforge.net/*
- flux-base, flux-math and flux-tests, developed by one of the authors of this thesis
- pcube, received from our supervisor

The MitrionSDK is required for tests involving the MitC-code on the FPGA simulator. The limited MitrionSDK is avaialable on
`https://secure.mitrion.com/mitrion-sdk-pe/`,
requiring acceptance of an EULA.

A script, `build.sh`, is included in the top-level directory of the distribution. This script will attempt to build all required modules and applications.

The source code can be retrieved from the web:
`http://www.dd.chalmers.se/~billeter/permanent/rt.tar.bz2`

[13]Very few platform dependent functions are used, so porting the software to a different OS should be possible with minimal work. There is a good chance that the software will run without modifications in any POSIX-compatible environment.

A.2 Running the reference ray tracer, `rt`

The reference ray tracer is located in the `source/app_rt/bld-release` directory (assuming the user choose to perform a "release" build). The application can be run as

```
  > ./rt -s scene.xml -v view.xml
```

where `scene.xml` is the path to the scene-definition file, and `view.xml` is the corresponding view-definition. See appendices C.2 and C.1 for information on these file formats.

A.3 Implementation details

Internally, the software is divided into several modules, each responsible for different tasks:

`profile`: a simple module that provides profiling functions and macros. The aim is to provide timing data with little overhead and to use an API that is as non-invasive as possible.

`scene_primitives`: "primitive" (i.e. basic) objects used by higher level modules. Examples include representation of transforms, material data, meshes and intersection routines.

`scene_graph`: representation of a scene, with the associated data structures (i.e. the grid data structure and various methods for traversing the grids).

`scene_load`: functions for loading external resources, which include mesh data, scene and view specifications. See appendix C.

`scene_draw`: code related to drawing things which includes a representation of the framebuffer, shading functions and a view class.

`ancient`: ancient code which, unfortunately, is still used by some utility applications which nobody had time to rewrite.

The following applications are included:

- `meshconv`: convert *.3ds* meshes to the interal format
- `fbconf`: convert output from the ray tracer to *.tga* images
- `rt`: the software ray tracer
- `rt_def`: deferred ray tracer; currently only useful for gathering statistics
- `fpga`: FPGA test applications. Requires an installation of the Mitrion-SDK.

Additionally, some test applications are included. These can be used to verify functionality provided by some modules; however the tests are by no means exhaustive.

B The FPGA co-processor implementation

This section delves deeper into the details of the co-processor. The following two sections cover the FPGA-Host communication and internals of the implementation.

B.1 The FPGA-Host interface

The exchange of data between the host and co-processor is managed by Mitrions hardware abstraction layer Mithal. The procedure for setting up and running the co-processor can be described by the following steps:

1. Allocate and initialize (program) the FPGA
2. Allocate and fill a data buffer
3. Start co-processor, which will access the data via DMA
4. Wait for co-processor to finish
5. Access data buffer for results
6. Repeat from step 2 until all data is processed

As explained in chapter 2.4, input to the co-processor is sent in *tasks*; each task consists of one or several batches. Due to the fact that scences vary in size, we have kept the sizes of batches and tasks dynamic, which also requires us to send additional meta-data describing the sizes of each batch and the number of batches involved in the current task. These sizes are sent in a header consisting of a single *task control world* and a *batch control world* for each batch.

The task control word states the number of batches and the total number of rays. Each batch control word contains information on how many *ray*- and *triangle bundles* the batch contains and offsets describing where to find the ray and triangle data in SRAM.

The interface to the FPGA decvice consists of a eighty megabytes SRAM memory[14]; the memory is accessible by two 128 bit busses. I.e. every cycle the co-processor can access eight 32 bit floaing point numbers. Since triangles are represented by nine floats and rays by six, loading data at the highest possible speed becomes somewhat convoluted.

B.2 Data Flow

This section will explain in more detail how the data passes through the system. Figure 13 is the representation of the data flow given by the Mitrion FPGA simulator.

[14]Though, it is apparently not possible to use the full SRAM using the MitrionSDK

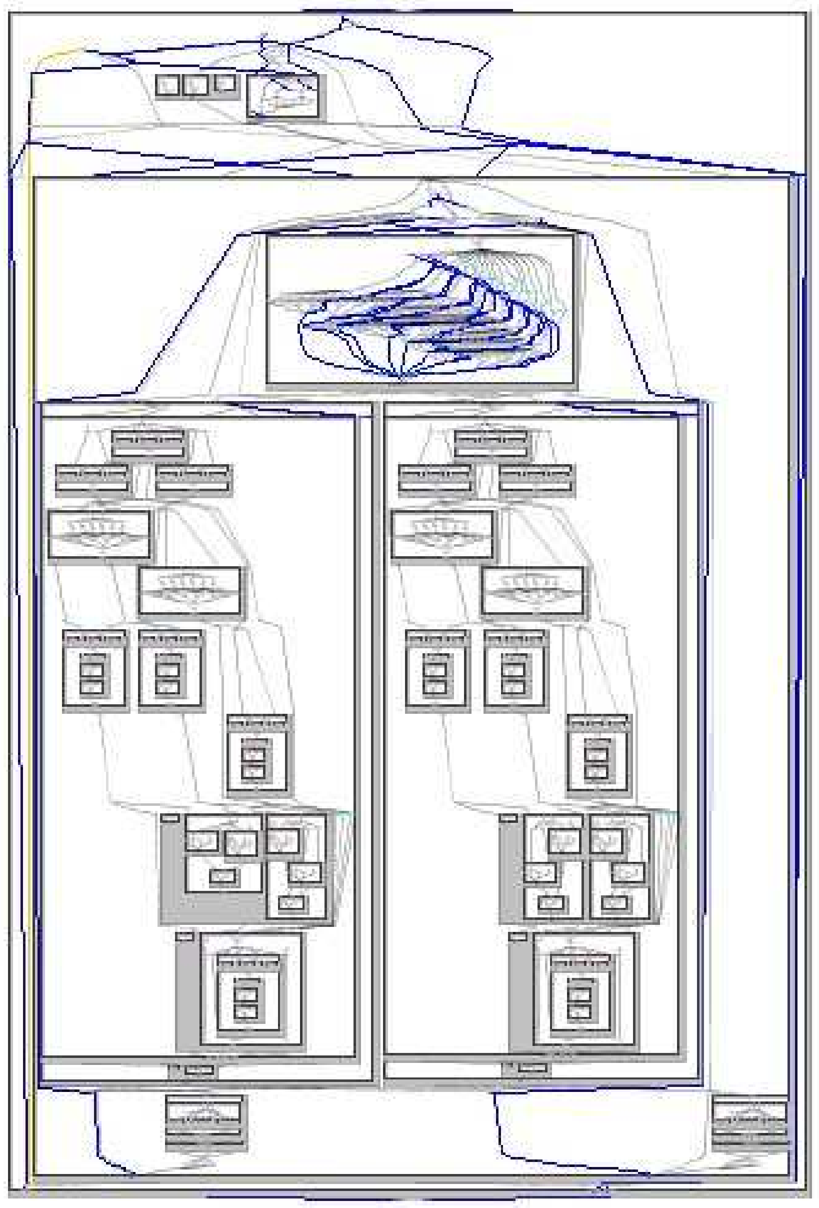

Figure 13: Data flow and program structure as displayed by the Mitrion FPGA simulator software.

B.2.1 Rays

To be able to maximize the troughput in the FPGA we buffer all rays in local high speed memory. There are several reasons for this:

- Rays are reused for several triangles; caching them more locally reduces the bandwidth from the SRAM.
- Storing the rays in a separate memory frees up memory space for storing intermediate results. This removes the risk of writing over important data, wich can be a problem in parallel programming.
- Since the local memory is configurable, the memory word can be set to the exact size of ray, thus fetching a ray takes exactly one cycle.

B.2.2 Triangles

Triangles are read one bundle (e.g. in batches of eight) at a time. Each triangle in the bundle is then stored in local registers together with its calculated index in the batch.

B.2.3 Intersection and Sorting

For each triangle in a read bundle, intersection tests with each ray will be performed. Misses are discarded; hits will be compared to previous results, of which the closer hit is kept.

C Data formats and utilities

This appendix contains more detailed references about the data formats and utility applications presented in the thesis.

C.1 The View Definition

The view definition file is used to define the "view" of the scene, i.e the current viewport and framebuffer settings. A sample view definition is included below:

```
<?xml version="1.0" ?>

<view>
  <buffer width="800" height="600" />
  <camera fov="70.0" near="4" multisample="2">
    <transform type="translate" x="0" y="0" z="-480.0" />
  </camera>

  <output file="dragon.hdfb" />
</view>
```

Most fields are self-explanatory, possible with the exception of the `multisample` setting: `multisample` currently supports three different settings, 0, 1 and 2.

- `multisample="0"` Disable multisampling (one sample per pixel)
- `multisample="1"` 2x multisampling (four samples per pixel)
- `multisample="2"` 4x multisampling (16 samples per pixel)

Several elements of the type `transform` can be added to the `camera` node; the transform-elements are evaluated in order. Three types of transforms are supported:

- `type="translate"` translation, requires `x`, `y` and `z` attributes.
- `type="rotate"` rotation, requires `x`, `y`, `z` and `degrees` attributes.
- `type="invert"` invert the current transform

C.2 The Scene Definition

Contents of a scene are defined in a scene definition file. A scene definition includes some global parameters (e.g. resolution of the tier one grid), references to the objects used in the scene, and one or several instances of each objects with associated transforms. Finally one or several light sources can be defined.

A sample scene definition is included below:

```
<?xml version="1.0" ?>

<scene>
  <!-- SETTINGS -->
  <grid i="10" j="10" k="20" />

  <!-- RESOURCES -->
  <resource rel="object" id="dragon"
    source="../../data/s_10/dragon.object" />

  <!-- INSTANCES -->
  <instance rsrc="dragon">
    <transform type="rotate" x="1" y="0" z="0" degrees="180.0" />
    <transform type="translate" x="0.0" y="-180.0" z="0.0" />
  </instance>

  <!-- LIGHTS -->
  <light>
    <position x="400.0" y="0.0" z="0.0" />
    <color name="ambient" r="0.2" g="0.2" b="0.2" />
    <color name="diffuse" r="0.0" g="1.0" b="0.0" />
    <color name="specular" r="4.0" g="4.0" b="4.0" />
  </light>
  <light>
    <position x="-400.0" y="0.0" z="0.0" />
    <color name="ambient" r="0.2" g="0.2" b="0.0" />
    <color name="diffuse" r="1.0" g="0.0" b="0.0" />
    <color name="specular" r="4.0" g="4.0" b="4.0" />
  </light>
</scene>
```

Again, most elements are self-explanatory. As described in the view definition (C.1), several transforms can be chained in each `instance` node; the transforms are then applied in order to the instance. The same three types of transforms are supported in the scene definition as in the view definition.

C.3 The Object Definition

Object definitions are XML-files created by the `meshconv` utility (described in appedix C.4), and do not need to be edited or created manually. A sample object definition is shown below, slightly edited, as the original file is rather lengthy (> 3000 lines):

```
<?xml version="1.0"?>

<object name="example">
  <!-- MATERIALS -->
  <material name="example:material">
    <color name="ambient" r="0.200000" g="0.200000"
      b="0.200000" />
    <color name="diffuse" r="0.439216" g="0.352941"
      b="0.247059" />
    <color name="specular" r="0.000000" g="0.000000"
      b="0.000000" />
    <color name="emission" r="0.000000" g="0.000000"
      b="0.000000" />
  </material>

  <!-- MESHES -->
  <mesh name="example:OBJ01" type="grid">
    <!-- Stats:
      * voxels: 4608 (3813 empty)
      * triangles: 1 min, 235 max, 11 average
      * 54677 triangles total
    -->
    <blob vertices="164031" src="example-OBJ01.blob" />
    <bounds type="aabb" rel="origin" x="-71.860001"
      y="197.419998" z="-45.199997" />
    <bounds type="aabb" rel="extent" x="146.130005"
      y="164.500015" z="145.810028" />
    <bounds type="sphere" rel="origin" x="3.983251"
      y="271.453979" z="19.567768" />
    <bounds type="sphere" rel="radius" val="107.350876" />
    <grid i="16" j="18" k="16">
      (       0       0) (       0       0) (       0       0)
      <!-- snip -->
      ( 164031       0) ( 164031       0) ( 164031       0)
      ( 164031       0) ( 164031       0) ( 164031       0)
    </grid>
  </mesh>

  <!-- snip -->

  <!-- INSTANCES -->
  <instance mesh="example:OBJ01" material="example:material" />
</object> <!-- EOF -->
```

The object definitions consists of three parts: material definitions, mesh information and mesh instances. Material information is relatively straight-forward, each material can have ambient, diffuse, specular and emissive color properties.

Each mesh contains the following information:

- `type`: currently only meshes of the type `grid` are supported.
- `blob`: each mesh links to an external binary blob, which contains the actual data (i.e vertices and normals).
- `bounds`: a bounding sphere and box is provided with each mesh (the bounding box is required by the grid data structure).
- `grid`: information about the grid data structure, i.e. number of subdivisions and vertices contained in each voxel (i.e. index of the first vertex in a given voxel, and the number of vertices in the voxel).

Each mesh must be instanciated once and may be associated with a materal (earlier versions supported specifing transforms with each instance, but this was removed later).

C.4 The `meshconv` Utility Application

The `meshconv` utility converts 3D data from the common *.3ds* model format to the internally used `.object` format. At the same time, the application preprocesses the meshes in the model by subdividing them into grids.

Subdivision into grids requires the following calculations:

- Find axis aligned bounding box (AABB) containg the mesh
- Guess a suitable subdivision
- Insert every triangle into all grid voxels it intersects with

Finding the minimal AABB which containts the mesh is trivial. The AABB is then used to guess a suitable subdivision under the assumptions that

1. All triangles are uniformly distributed in the AABB
2. It is advantageous to have voxels with sides of approximately equal lengths.

Additiontally, a target ratio of triangles per voxel is specified by the user.

The forumla used to guess a suitable subdision is as follows:

$$\begin{pmatrix}\text{subdiv}_i\\ \text{subdiv}_j\\ \text{subdiv}_k\end{pmatrix} = \text{ceil}\left[\left(\frac{\text{numTriangles}}{\text{targetTrianglesPerVoxel}}\right)^{\frac{1}{3}} \cdot \frac{3}{x+y+z}\begin{pmatrix}x\\ y\\ z\end{pmatrix}\right]$$

where x, y and z describe the size of the AABB containing the mesh.

Usually, meshes do not fullfill the first assumption which often results in a significantly higher number of triangles per voxel than specified by the user. The `meshconv` utility runs in interactive mode per default, and allows the user to manually correct the subdivision configuration, which can be used to compensate for this fact.

C.5 The `fbconv` Utility Application

Rendered images are saved in a very simple format, usually with the extension `.hdfb`. These rendered images can not be displayed directly in any image viewing program, rather the `fbconv` utility is used to convert these images to the *.tga* format.

As the rendered images contain color values which are not limited to the range $[0, 1]$, conversion is performed in three steps:

- find maximum color values
- scale color values to the range $[0, 1]$
- output *.tga* image

Usage of the `fbconv` utility is as follows:

```
$ ./fbconv in.hdfb out.tga
```

where `in.hdfb` is an output rendering from the ray tracing application, and `out.tga` is the output *.tga*-file.