CHALMERS

Sphere Tracing GPU

An evaluation of the Sphere Tracing algorithm and a GPU designed to run it

Bachelor's thesis in Computer Science and Engineering

Jesper Åberg
Björn Strömberg
André Perzon
Chi Thong Luong
Jon Johnsson
Elias Forsberg

CHALMERS UNIVERSITY OF TECHNOLOGY
University of Gothenburg
Department of Computer Science and Engineering
Gothenburg, Sweden, July 2017

Bachelor of Science Thesis

Sphere Tracing GPU

An evaluation of the Sphere Tracing algorithm and a GPU designed to run it

Jesper Åberg

Björn Strömberg

André Perzon

Chi Thong Luong

Jon Johnsson

Elias Forsberg

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden, July 2017

Sphere Tracing GPU
An evaluation of the Sphere Tracing algorithm and a GPU designed to run it

Jesper Åberg

Björn Strömberg

André Perzon

Chi Thong Luong

Jon Johnsson

Elias Forsberg

Supervisor: Miquel Pericas, Computer Science and Engineering
Examiner: Arne Linde, Computer Science and Engineering

Bachelor's Thesis 2017:12
Department of Computer Science and Engineering
DATX02-17-12
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2017

Abstract

This thesis documents our experiences and conclusions during the design of a graphics processing unit (GPU), specifically designed and optimized for executing the Sphere Tracing algorithm, which is a variation of the standard Ray Tracing algorithm. The GPU is written in the functional hardware description language CλaSH. A shader was also implemented in GLSL to enable algorithm research.

Real-time rendering performance has always been a significant factor for 3D graphics cards. However, using the Sphere Tracing algorithm on conventional 3D graphics cards is slow because they are designed and built for polygon based rendering. We assessed potential performance improvements using our GLSL shader. The results indicate that it is possible to increase the performance of the Sphere Tracing algorithm.

Keywords: sphere tracing, ray marching, ray tracing, GPU, real-time rendering.

Sammanfattning

Denna uppsats dokumenterar våra upplevelser och slutsatser under utvecklingen av en grafikprocessor (GPU) som vi designat och optimerat för att exekvera renderingsalgoritmen Sphere Tracing, en typ av Ray Tracing. GPU:n är skriven i det funktionella hårdvarubeskrivningsspråket CλaSH. För att möjliggöra algoritmstudier implementerades en shader i GLSL.

Hastigheten på realtidsrenderingen hos 3D-grafikkort har alltid varit en viktigt faktor. Dock, är det långsamt att använda Sphere Tracing algoritmen för att rendera grafik på konventionella grafikkort eftersom de är designade och byggda för polygonbaserad rendering. Vi undersökte potentiella prestandaförbättringar för Sphere Tracing algoritmen i vår GLSL shader. Resultaten indikerar att det är möjligt att förbättra prestandan för Sphere Tracing algoritmen.

Nyckelord: sphere tracing, ray marching, ray tracing, realtidsrendering.

Contents

1

Introduction

During the late '80s and through the '90s many types of real time rendering algorithms were developed and tested [10]. This was possible because they ran in software on the CPU (Central Processing Unit), and could thus be easily altered and enhanced. Photo realistic rendering could be achieved using a method called Ray Tracing, but this is slow for real time graphics [22]. One version of this algorithm is a method called Sphere Tracing. This method could be used to render 3D fractals [10], but it was still too slow to render complex geometry in real-time [10]. Better graphics rendering (fast real-time graphics) was a key selling point among competing computer systems, so to improve rendering speed, the computer industry moved towards dedicated hardware based graphics rendering [12]. These systems were almost exclusively using polygon-based rendering, leading to hardware unsuitable for Sphere Tracing . Today, 3D graphics cards have become more programmable and the last 10 years have seen a small resurgence in the use of Sphere Tracing rendering [20]. Simple scenes can now be rendered using Sphere Tracing in real-time using state-of-the-art consumer 3D graphics cards. However, the full potential of the algorithm is not achievable on current GPUs due to the underlying architecture being designed for polygon rendering [12]. This project examines the performance potentials of graphics cards based on the Sphere Tracing algorithm.

1.1 Note on shaders

Throughout the report we will use the word *shader*. A shader is a historical term that usually refer to the code running on the GPU as opposed to on the CPU. It is called a *shader* because initially only the color and illumination attributes were programmable, where as today's GPUs are more flexible.

1.2 Project goals

The main goal of this project is to design and implement a basic GPU architecture that is designed to efficiently execute the Sphere Tracing algorithm. In striving towards efficiency, we examine Sphere Tracing in order to find possible optimiza-

tions, both algorithmic and hardware based. For testing some of the optimizations' performance, a software shader is to be designed.

1.3 Scope

Writing a GPU for the first time, even a smaller one, and doing it well is a large undertaking for a single bachelors project. The project is therefore done on a best-effort basis where we try to implement the novel parts of the design, while leaving out generally well researched systems such as cache and memory management. When creating software shaders, very simple scenes has been prioritized over complex ones. This, in order to have more time for optimizations and testing.

2

State of the Art

2.1 Real Time Graphics Rendering on Current Hardware

Sphere Tracing is currently being used to visualize complex data such as fractals [8], but there is no hardware designed to run it efficiently. Today's top of the line consumer 3D graphics cards are instead made for real time rendering of polygons [12]. They achieve high performance by executing rasterization and pixel color calculation in parallel between pixels. To make this efficient they use what is known as lockstepping, which means a group of calculation units that are always operating on the same instruction, but with differing input data. This enables all of the cooperating cores to use only a single instruction memory and bus. This reduces the area usage of the design, making it possible to add more cores per chip and thus achieve greater performance. Pixels rendered by polygon based methods can usually be queued for calculation in such a way that pixels belonging to the same object in the scene can be grouped for computation. This is possible because the GPU gets its graphics drawing instructions as a list of polygons, and consecutive polygons in this list mostly belong to the same 3D object. The drawback of a group of lockstepped cores; that all of the cores have to execute the same instruction at any given time, is greatly reduced since the color calculation for adjacent pixels of a certain 3D object will have the same set of instructions.

Using this hardware for Sphere Tracing, as is currently being done, increases the overhead caused by this lockstepped design significantly. Order of pixel rendering in a polygon renderer is done by object and then by constituent polygons. Pixels can not be grouped by what object in the scene they belong to as easily in a Sphere Tracer, because Sphere Tracers do not have polygons, and pixel order is usually based on the output image pixel order. Even if more grouping was introduced to a Sphere Tracer, the pixels in these groups differ much more in their instruction execution path, since the algorithm contains a loop that often varies greatly in the number of steps until completion. As described above, when the current instruction differs across cores in a group, some cores can not execute and have to wait until the remaining cores arrive at the instruction that they are waiting to execute.

2.2 Hobbyists and Academia

Despite the limitations of contemporary graphics processors, computer art hobbyists are using Sphere Tracing to render stunning visuals on consumer PC's in real time. They showcase the possibilities of the algorithm by reducing scene complexity and instead rendering using techniques that are commonly found in non real-time Ray Tracers. Examples of such are true reflections and refractions, spacial repetition, object morphing and 3D fractals [20]. Inspired by research papers such as John C. Hart's 1996 paper [9], their success encouraged some to do academic research of their own, which led to new papers being written. A good example of this is the 2014 paper "Enhanced Sphere Tracing" [15].

2.3 Industry

There are as of today no big commercial applications using Sphere Tracing that we know of, but Ray Tracing algorithms have long been in use in multiple computer graphics domains. One example is film making [7], where a comparatively large amount of time taken to render a scene can be acceptable since it can be computed ahead of time and saved as individual images, rather than producing the images in real-time, which is needed for interactive visuals. Thus the computer graphics in movies can be much more photorealistic than interactively rendered computer graphics. An example of this would be the Ray Tracing engine RenderMan developed by Disney Pixar which they use in some of their movie productions [7].

2.4 Hardware Design Methods

The design of Integrated Circuitry in the hardware industry has since the early '90s primarily been done in hardware description languages (HDLs) [6], where the operation of a chip is described in a style similar to regular imperative programming languages. This descriptive code can then be compiled into a list of components and connections that constitute the blueprint for that specific circuit. The most prevalent of these languages are VHDL and Verilog [6]. While being a powerful aid in circuit design, compared to a more block diagram-based workflow [16], these languages can still be quite cumbersome to work with. They are verbose and require a fair amount of boilerplate code. This makes it more difficult to understand, follow, and also write code that performs complex tasks, since it can be more difficult to see the greater patterns in interconnecting code.

This has resulted in Functional HDLs (FHDLs), as defined by Christiaan Baaij in [CλaSH: From Haskell To Hardware] "Functional hardware description languages are a class of hardware description languages that emphasize on the ability to express higher level structural properties, such a parameterization and regularity. Due

to such features as higher-order functions and polymorphism, parameterization in functional hardware description languages is more natural than the parameterization support found in the more traditional hardware description languages, like VHDL and Verilog" [3]

HDLs have been around since the late '70s [6], but in recent times they have become more mature [6]. Two notable modern FHDLs are *Lava* and CλaSH [3,5], which are both implemented in Haskell. This allows the same high-level interactive simulation of the program that normal Haskell programs enjoy. This means that instead of simulating the underlying circuit directly, which is more time consuming, the design can be tested repeatedly at a faster pace, allowing faster development. This high level simulation can also be done in an interpreter enabling easy and rapid testing of code. These are called read-eval-print-loop interpreters [11].

3

Sphere Tracing

The name Sphere Tracing [9] is based on the fact that the algorithm uses distance bounding spheres to incrementally advance along a ray in 3D space. The method of advancing along rays in general is called Ray Marching which in turn is a subset of Ray Tracing [25]. Ray Tracing, is a way to wholly or partially render the world through rays, cast from the eye of the observer into the scene [22]. Sphere Tracing has been around since at least as early as the late eighties [10] and Ray Tracing since the sixties [1].

Using Sphere Tracing, scenes can be rendered with lighting that is very close to how real light behaves. It also allows building more complex objects from simpler primitives using constructive solid geometry, explained later in section 3.1.1. These objects can be repeated and distorted at a low cost to create higly detailed vast scenes. For an example of some interesting scenes rendered using Sphere Tracing, see appendix B.

In this chapter the Sphere Tracing algorithm is discussed in detail with both definitions and examples. The explanation is based on the the very succinct explanation in [15]. That paper also expands upon the original algorithm in some innovative ways. For a more in-depth description of the original algorithm see [9].

3.1 Sphere Tracing

The Sphere Tracing algorithm uses Signed Distance Functions to represent the geometry of a scene. If f is a signed distance function:

$$f : \mathbb{R}^3 \mapsto \mathbb{R}$$

then f returns, for all points in $\mathbb{R}^3$, the signed distance to the closest point on the implicit surface $\mathrm{f}^{-1}(0)$. Signed in this context refers to the fact that the distance should be negative when measured from the inside of the object represented by the surface.

A common example would be the signed distance function, f, of a sphere centered at the origin with a radius of one: $f(\vec{v}) = |\vec{v}| - 1$. We can here see that a point

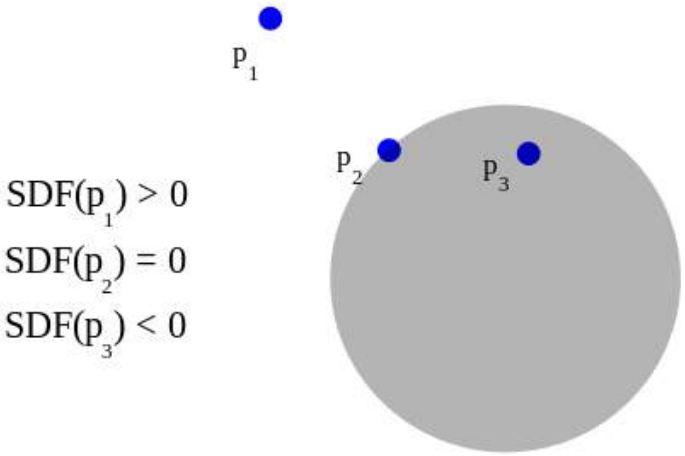


Figure 3.1: Signed Distance Function of a sphere, sampled at three points

$\vec{v}$ exactly on the surface would evaluate to $1 - 1 = 0$, any point outside of the sphere would evaluate to some positive distance and any point inside the sphere will result in a negative distance. It is, as will be discussed later, possible to generalize these distance functions to both unsigned distance functions and different kinds of distance bounds and approximations.

If we define a ray $r(s) = \vec{d} \cdot s + \vec{o}$ where $\vec{d}$ is the direction of the ray and $\vec{o}$ the origin, then $f \circ r(s) = 0$ means that the ray intersects the surface described by f at exactly distance s from the view point origin. Sampling every signed distance function in a given scene and returning the smallest value yields a function known as a distance field. Finding the surface can be done by marching point by point from the origin along the ray like below:

$$p_{i+1} = p_i + \vec{d} \cdot f(p_i)$$

This is repeated until $f(p_i) \leq \varepsilon$ for a given precision limit ε. $f(p_i)$ is the furthest possible march distance the ray can march while still not overshooting any potential surfaces at iteration i. The direction to the closest surface point is never known, thus $f(p_i)$ can be interpreted as a spherical bound, giving the algorithm it's name. The Sphere Trace is usually performed for each pixel of the screen, almost simulating the light rays entering the lens of an eye or camera in reverse.

3.1.1 Generalized Distance Functions

It is also possible to generalize signed distance functions into signed distance bounds: we define signed distance bounds such that if f is the signed distance function for a given object, g is a signed distance bound to the same object iff $\forall \vec{v}.|g(\vec{v})| \leq$

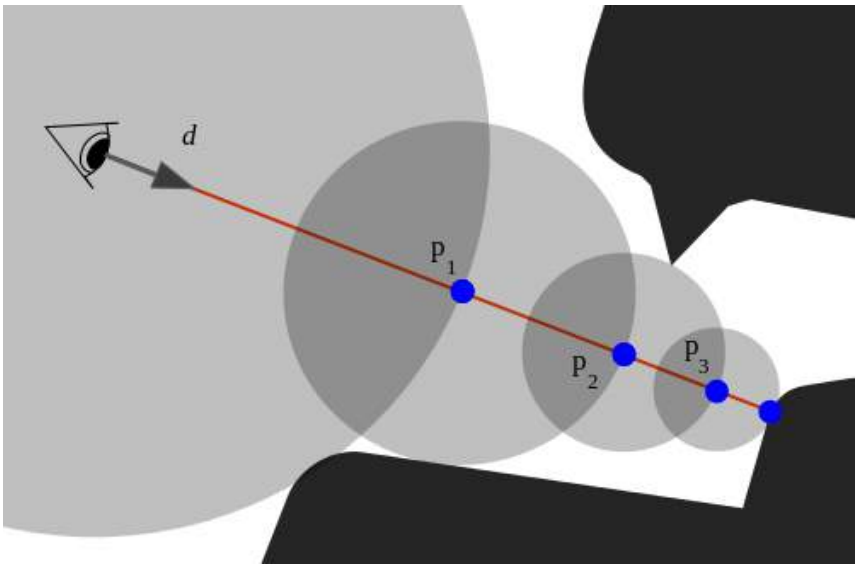


Figure 3.2: Finding the intersection of an implicitly defined surface and a ray can be done by marching point by point from the origin along the ray as show here.

$|f(\vec{v})|$. The Sphere Tracing algorithm continues to function correctly (*That is, it will not march past any object without detecting it*) when signed distance functions are exchanged with signed distance bounds. The reason for this is simple: given a position $\vec{p}$, a Sphere Tracer using signed distance bounds will march forward $g(\vec{p})$ distance units, whereas a Sphere Tracer using signed distance functions will march forward at least as far or further, since $f(\vec{p}) \geq g(\vec{p})$.

Using signed distance bounds, it is easy to construct complex objects from simple primitives using constructive solid geometry. That is, constructing new objects using unions, intersections, and complements: Let f and g be signed distance bounds of a and b respectivley. $\min(g, f)$ is then a signed distance bound for the union of a and b, $\max(g, f)$ is a signed distance bound for the intersection of a and b, and $-f$ is a signed distance bound for the complementary shape of a [9]. The union of signed distance bounds relates nicely to how Sphere Tracers handle multiple objects in scenes. By taking the minimum of all distance functions it is in effect taking the union of all ojects in the scene.

Signed distance bounds for objects are not unique for a given object, and not all signed distance bounds are useful for Sphere Tracing. Indeed, $f(\vec{v}) = 0$ is a signed distance bound for every object that exists, but a Sphere Tracer would stop tracing immediately upon evaluating this function, and no useful object would be rendered. A Sphere Tracer can still work well though, as long as the distance bounds are 'close enough' to the signed distance function of the object they represent [20]. Being close enough is a fuzzy constraint, but generally the length of the gradient should be as close to 1 as possible for good results [13].

It is of course possible to relax this concept further, and use distance approximation functions. These can still work together with a Sphere Tracer if they are 'close enough' to the signed distance function for the object they are meant to represent,

but correctness can be harder to achieve. Distance approximation functions allow significantly greater flexibility when designing objects however, making it possible to add e.g. sinusoidal deformations [13, 19].

Another relaxation that can be considered is using unsigned distance functions. This might reduce the complexity of the distance function, but it also reduces the stability of the Sphere Tracing algorithm and invalidates some optimizations that are possible for their signed counterparts, such as predictive overstepping [15].

3.1.2 Reflections, refractions and shading

Once a point on a surface for a given pixel has been located, reflections, light, shadows and refractions can be calculated. A lot of these depend on the surface normal which can be approximated with the partial derivatives around the surface point given some small value δ. This is the normalized gradient $\vec{g}$ of that point.

$$\vec{g} = \vec{x} \cdot \frac{\mathrm{SDF}(x+\delta, y, z)}{\delta} + \vec{y} \cdot \frac{\mathrm{SDF}(x, y+\delta, z)}{\delta} + \vec{z} \cdot \frac{\mathrm{SDF}(x, y, z+\delta)}{\delta}$$

$$\vec{n} = \frac{\vec{g}}{|\vec{g}|}$$

A simple way to illuminate the scene could for example be to use Phong Lighting [18]. But any number of alternative algorithms could be used instead to determine the final color. Shadows can be determined by further Sphere Tracing out towards the sources of light from the surface to check for obstructions. Same for reflections and refractions where further marching in proportionate angles to the angle of incidence can determine what objects are reflected on the surface.

4

Implementation

In order to achieve the goals set in this project, two separate pieces of software and one piece of hardware were written. The software programs are two different implementations of a reference shader. The first of these was written in GLSL [14] for traditional graphics hardware, and the second is written in an assembly language we designed for the Sphere Tracing processor. Both of these shaders are described in section 4.1. The hardware is comprised of a multi-core programmable parallel processing unit designed for the Sphere Tracing algorithm, detailed in section 4.2.

4.1 Shaders

The shaders written uses the theory described in chapter 3, iterating over a ray in their main loop. The GLSL shader has more optimizations, but lacks the ray grouping optimization. The optimizations are described separately after the shaders.

4.1.1 GLSL Based

This shader is designed to run on conventional graphics hardware. It was created in the high-level shading language GLSL (short for OpenGL Shading Language). It executes its main-loop for each pixel the shader is rendered on, using the screen-coordinates as input. The Bounding Sphere and Orthogonal Culling optimizations were implemented and used in this shader.

Before the loop is run Orthogonal Culling is executed, reducing the number of objects in the distance field. Orthogonal Culling can cull individual objects or Bounding Spheres. The Sphere Tracing loop is a straight-forward implementation of the algorithm.

After a hit is detected different things can happen depending on the material assigned to the object. If the object is reflective, then the normal of the surface that was hit is calculated and used to reflect the ray. If the object is colored the pixel will simply be set according to the object. Color can also be calculated from object or ray coordinates.

4.1.2 Assembler Based

In this shader the Ray Grouping optimization was implemented. This shader was written in an assembly language we designed for our GPU. When this shader starts executing, a single thread is initially created. This thread will create a single render thread and one new version of itself in order to create more render threads until it has created a render thread for each pixel (or group if Ray Grouping is used). This will happen continuously during execution. This will render all pixels as concurrently as the GPU has cores, minimizing core stalls and therefore maximizing GPU usage.

When a render thread is created it immediately starts setting up the ray. The screen coordinates are calculated from the thread id, this is essential for calculating the ray direction. Camera position and a point to orient the camera against are predefined.

After screen coordinates are calculated a direction vector for the ray is calculated. The Sphere Tracing loop is a straight-forward implementation of the algorithm. If a hit is detected a color will be assigned to the corresponding pixel.

4.1.3 Bounding Spheres Optimization

In order to decrease the number of distance calculations that has to be performed in each march step, objects can be grouped together in larger enclosing bounding spheres. When the distance field is evaluated only the bounding sphere will be considered and none of the objects enclosed by it. If the bounding sphere is hit by the ray, it will be removed and all the contained objects will be introduced to the distance field.

A Bounding Sphere is defined by its location, radius and enclosed objects. The location is the mean position of all the enclosed objects. The radius is made large enough to encompass all of it's objects.

In our implementation a Bounding Sphere is simply an object in the scene. Instead of returning a color or redefining the ray (such as reflecting objects do) when hit it will return the smallest distance of the enclosed objects. Our implementation does allow for nested Bounding Spheres but this was not tested.

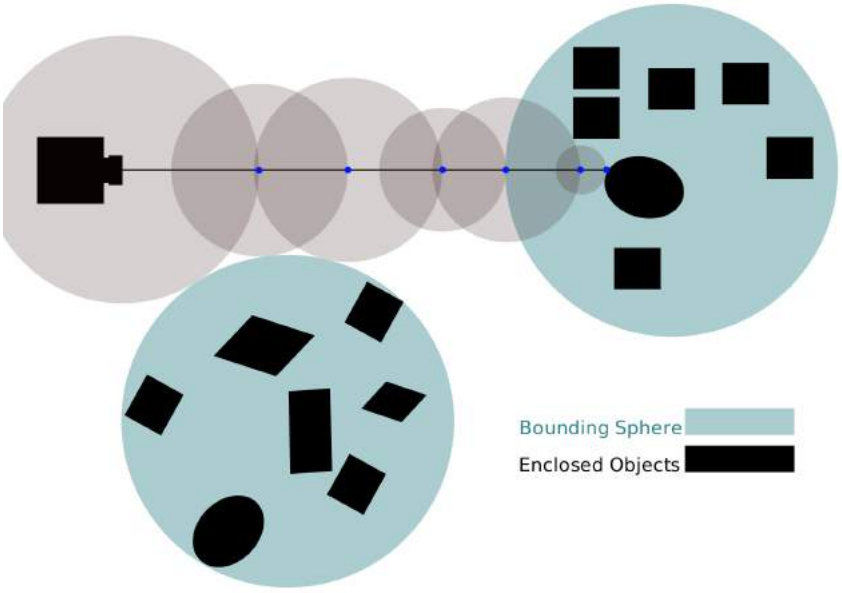


Figure 4.1: Bounding spheres, notice that the distance to individual objects is not calculated until the ray is inside a sphere.

4.1.4 Orthogonal Culling Optimization

A bottleneck of Sphere Tracing is that the distance to a lot of objects has to be calculated once for every march step. Performance can be increased by reducing the number of objects that the distance has to be calculated to, without reducing the actual number of objects in the scene. To do this the GPU needs to calculate which object a ray might hit during the execution of the Sphere Tracing algorithm. This information is traditionally calculated on the CPU and then passed to the GPU (a.k.a Frustum culling [2]).

Orthogonal Culling works by going through all the objects in the scene and calculating what objects a ray might hit. This is done for each ray before the Sphere Tracing loop begins. Orthogonal projection is used to tell whether an object is in front of the ray or not. By projecting the center of the bounding sphere of the object onto the ray, the line's closest point to the object is known. Then, the distance between the line and the sphere can be calculated using the radius of the bounding sphere. If this distance is less than zero, the ray will intersect the sphere. This way the number of objects that has to be evaluated at each march step is reduced.

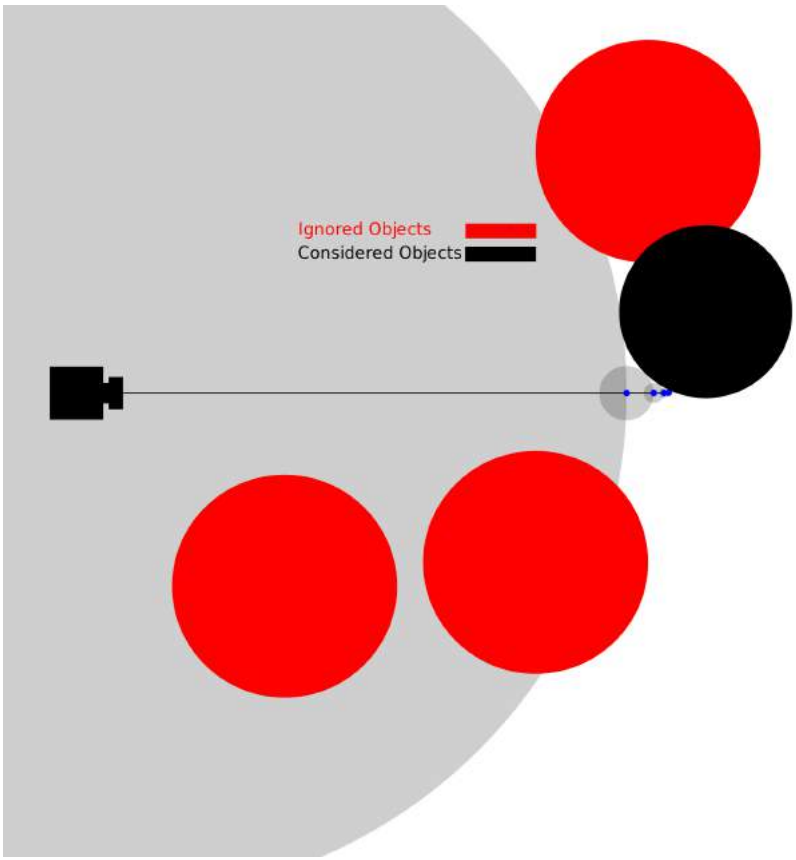


Figure 4.2: Note that the objects that are not on the path of the ray is not considered at every step.

4.1.5 Ray Grouping Optimization

Ray Grouping works by grouping adjacent rays together and marching them all at once in a single step. Each group of rays marches along its center pixels' ray in the same way that is done in normal Sphere Tracing. For each step, the smallest distance to the scene is compared to the total size of the group, to check if any ray might collide with the geometry. If any ray of the group might be inside the geometry, the group will be split into four new groups. Each new group then repeats these steps again individually along its new center. This is repeated either until a group has stepped far enough into the scene to discard the group entirely or until there is only one ray left in the group, and that ray has hit an object. The closer the groups are to a target, the more likely it is for a higher amount of sub-groups to be created due to the decreased distance to the geometry, making less pixels fit inside the group. When a group is far from an object in the scene it will likely not need to split and can therefore reduce the number of computations. An illustration of the reduced number of rays is shown in figure 4.3.

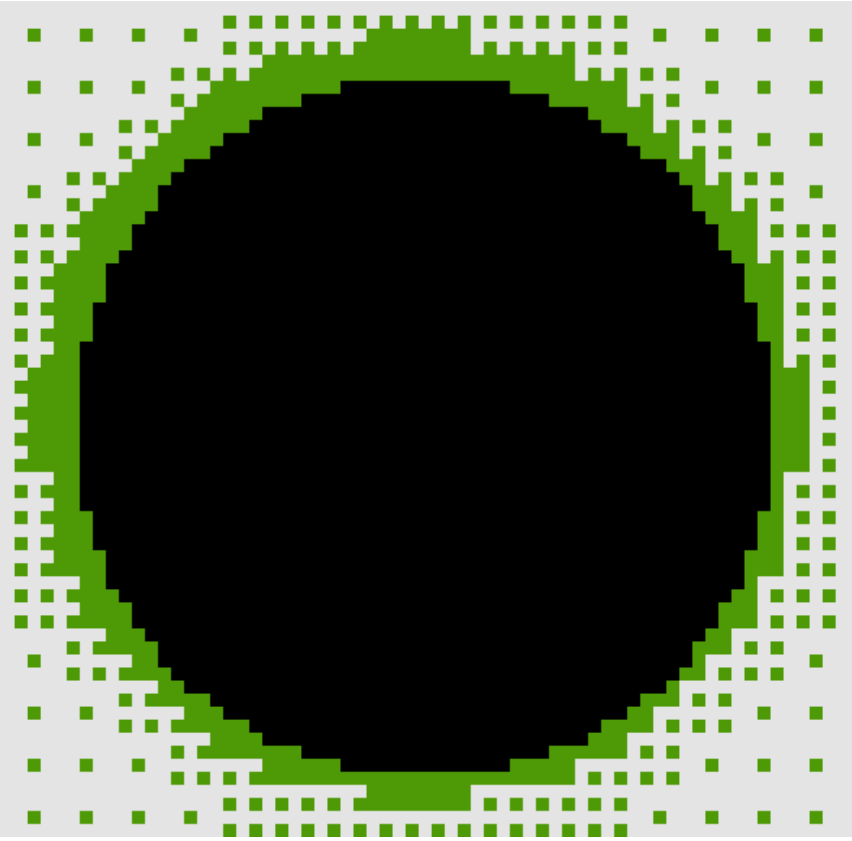


Figure 4.3: A simulated 64 × 64 display. Black pixels are rays that hit an object. Green pixels are groups that were discarded because they did not hit any objects. Note that the groups are larger further from the sphere. White pixels did not need to be calculated because of the ray grouping.

4.2 Processor Unit

The properties of the ray marching algorithm as described in chapter 3 led to the decision of designing a GPU architecture that does not utilize multi-ALU SIMD with lockstepping to the same extent as that of traditional graphics processors. We instead went for a solution with slightly simpler independent cores. This is similar to the reasoning employed by [26] for a Ray Tracing hardware design.

The hardware described in this section was written in CλaSH. CλaSHis based on the functional programming language Haskell and is therefore a functional hardware description language, or FHDL.

4.2.1 Architecture

The GPU is designed to be able to handle a form of threading natively, where all computations are performed by a thread running on a hardware core. A thread

in this context is very light-weight: only an instruction pointer along with some mutable registers are stored.

All threads that are not running are sent to a storage manager that communicates with all cores. Threads are then dispatched to the cores as they become available for execution. The cores are also able to spawn new threads by sending their data to the storage manager, or by terminating their current thread. This is done either by simply dropping them or by sending them to the frame buffer, where the register content will be used to determine the color of a pixel on the screen.

The storage in the storage manager on the GPU is implemented as a double-ended queue. This makes it possible for threads to have some coarse-grained control over execution priority, which is needed in order for many types of programs to be able to run with a bounded number of threads at any given time. During execution the storage manager selects one core that is ready to send a thread, if any core is ready to do so, and adds its thread to the storage. If any core is ready to receive a thread, one is taken from the storage and sent to that core. If more than one core is ready, one is chosen arbitrarily.

Each core has an instruction memory and a data memory. The instruction memory controls logic for execution order and the DFU. The data and instruction memories are immutable and mirrored across all cores in their entirety. The control logic is responsible for feeding the DFU with instructions and data. It also updates the thread registers when a new thread is attached, sending thread registers to the storage manager when new threads are spawned and requesting a new thread whenever the previously executing one terminates.

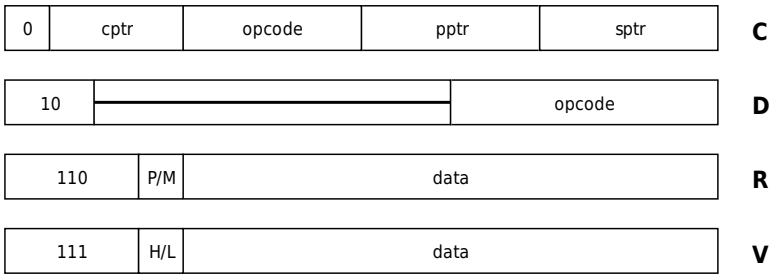
The DFU contains a stack which all arithmetic instructions operate on, as well as an ALU for carrying out calculations. It also keeps track of all the thread registers and updates them when such instructions are issued. The DFU contains very little extra control logic and is only able to receive and execute instructions, it does not keep track of any instruction pointers or when threads are attached or dropped, this is instead handled by the core control logic.

4.2.2 Instruction Set

A representation for distance functions was also designed. It is implemented as a reverse polish notation inspired stack based instruction set. There are a total of four instruction types with different instruction encodings. Their layout in memory is shown in figure 4.4. The instructions are explained in more detail in appendix A. The type of instructions are:

C-type instructions are for control flow, that is, instructions that affect the global queue or frame buffer or terminate the current thread. C-type instructions can be conditionally executed based on the condition encoded in the two highest bits in the opcode, equal to zero or not equal to zero.

Figure 4.4: Bit encoding for the different instruction types. All instruction types have a fixed width 16-bit encoding.



D-type instructions encode arithmetic- and vector operations that work on the stack. These operate only on the internal stack in the core and not on any of the other registers. They can take up to six arguments and return up to three return values. D-type instructions can be split up in chunks using the `next` instruction, where the last computed result will be accumulated using the min-function automatically. This is useful for encoding distance functions.

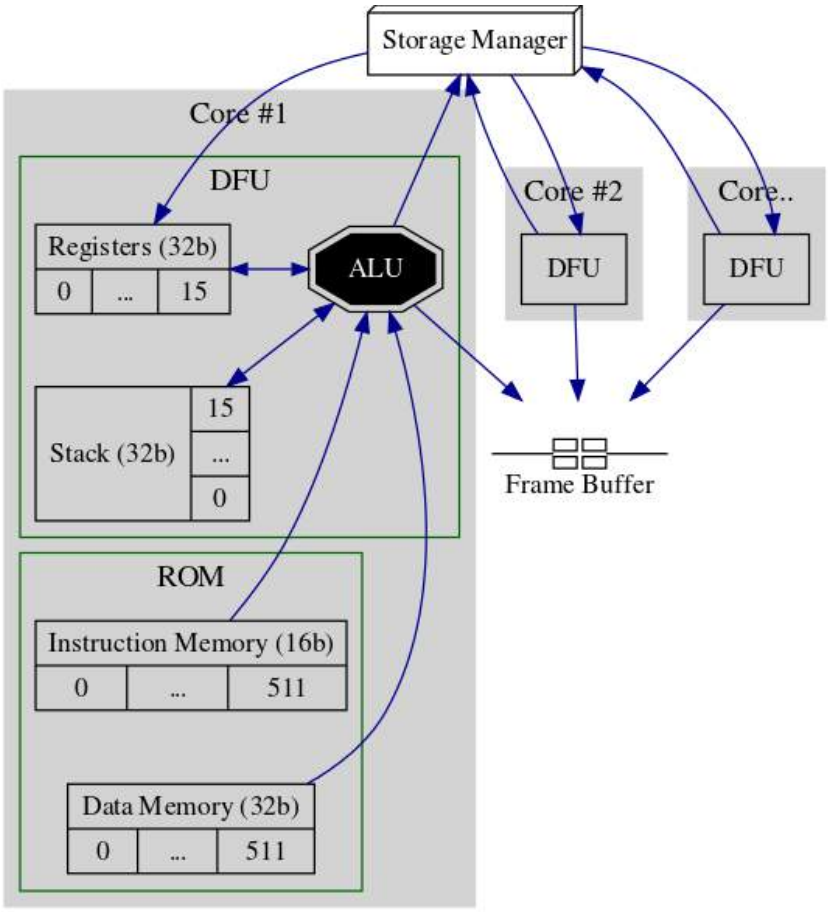
V-type instructions deal with memory accesses. The `p` bit encodes whether this access is for pack (mutable register) or read-only memory.

R-type instructions load wide immediate values into the DFU. They are currently only used for assigning distance function IDs.

4.2.3 Execution Model

The processor contains a global storage structure that is shared among all cores by a single storage manager. All threads that are ready for execution are held in this storage until a core is ready to start executing them.

Figure 4.5: GPU Data flow diagram



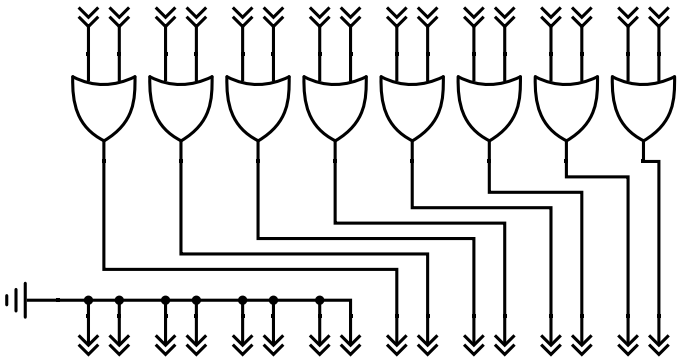
Each shader thread has 16 mutable registers that are automatically loaded into the core when execution starts. These registers are saved when execution pauses and copied whenever a thread spawns a new child. The instruction `setval` can be used to change the value of a register, and the instruction `pack` reads the value of a register.

All calculations are performed with intermediate values stored on a stack, with the ability to move any values between the stack and any of the registers using the `setval` and `pack` instructions.

There are three main instructions for control flow: `pushq`, `pushf`, and `drop`. Threads can spawn new threads at any time using the instruction `pushq`: this will cause the core to make a copy of all of the registers and push it onto the global queue, where another (or the same) core may later access them and start executing from the instruction pointer register (`r0`).

Threads can also terminate execution in two ways. Executing the instruction `drop` will terminate the thread and discard all registers. Executing `pushf` will terminate the thread and discard all values except the pixel pointer (`r1`) and color register (`r2`), which will be sent to the frame buffer in order to be displayed on the screen.

Figure 4.6: A simple square root approximator



This set of instructions for control flow might initially seem to be neither useful nor easy to implement but it is powerful enough for all branching that is needed in our Sphere Tracer while being restrictive enough to make instruction memory accesses very predictable.

The instructions `next` and `accum` are used to accelerate finding the minimum distance for a set of distance functions using a built-in accumulator. Because of this, distance functions can be written without any regard for how the shader that called them works. This very common operation being automated thus reduces the size of each distance function slightly.

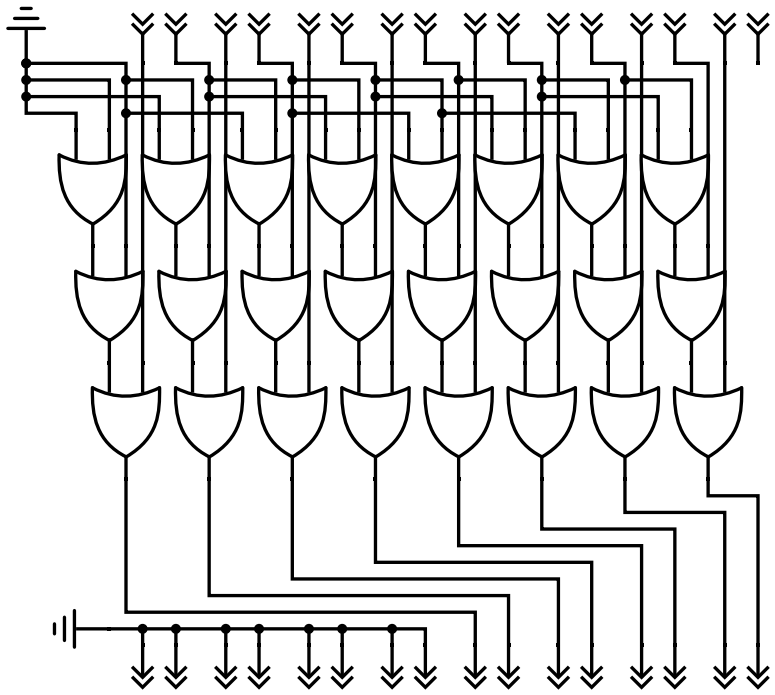
4.2.4 Square Roots

Calculating distances is an integral part of the algorithm, so a good square root implementation was considered important for achieving good performance. Several different approaches for this were considered.

Firstly, using an iterative method like Goldschmidt's or the Babylonian method require an initial starting value which is usually generated using a look-up table. We tested a different and very fast method for generating an initial rough guess of the square root of any number without the need for a look-up table. A 16-bit version of this circuit is shown in figure 4.6. The basic working principle is based on the fact that the square root of a number has roughly half as many digits as the original number. An improved but slightly more expensive version is showed in figure 4.7. In this version with slightly improved accuracy the placement of the extra wires is based on the bit pattern of $\sqrt{2}$.

In order to improve upon this method both in terms of accuracy and stability another design was considered. The input was both rounded up and down to values where the simple square root approximator was very accurate and the result was then linearly interpolated between these roots, as seen in the circuit in figure 4.8. This

Figure 4.7: Simple square root approximator with minor improvement. More layers of or-gates can be added with increased shifts for slight increases in accuracy.



linear interpolation requires only a multiplication and a couple of additions, but choosing upper and lower seed values is a more complex problem. Choosing the two geometrically closest powers of two was tested because the or-gate square root circuit is very accurate for powers of two, but this is not the only possible choice, as there are many points for which the simple square root approximator is very accurate. We did not look further into choosing different upper and lower bounds for the lerp-approximator.

Another algorithm for calculating square roots is the *Shifting nth root algorithm*, also known as *Dijkstras square root algorithm* [17], which calculates the square root one digit at a time. A `c` implementation of this algorithm is shown in figure 4.1. Initially, this algorithm does not seem like a good candidate for a high performance hardware implementation, but several important optimizations are possible when unrolling it combinatorially. Most of the additions can be already reduced to bitwise logic operations when computing it in software, as shown in figure 4.2. In a combinatorial hardware implementation, these bitwise operations can be removed entirely by simply moving wires around yielding a logic cost of 0. The only remaining operations is therefore the subtraction on line 6, and the conditional assignments of `num` and `res` on lines 9 and 10. The cost of the subtraction can be significantly reduced because `res` approximates the result one bit every step, meaning most bits of `num` and `res` are zeroed at any given iteration. This means that on average, only 25% of the subtraction logic is needed. This turned out to result in a design very similar to the PARSQRT implementation by [17].

Figure 4.8: Simple square root approximator used together with linear interpolation. This version includes an extra layer of or-gates with a shift of 4 and is more accurate for powers of 2. Note that the gates in this diagram seem to be ordered in two extra layers. This is, however, only a visual artifact in this image. Adding more layers with shifts of 6, 10, 14, 26 etc increases accuracy further but with rapidly diminishing returns.

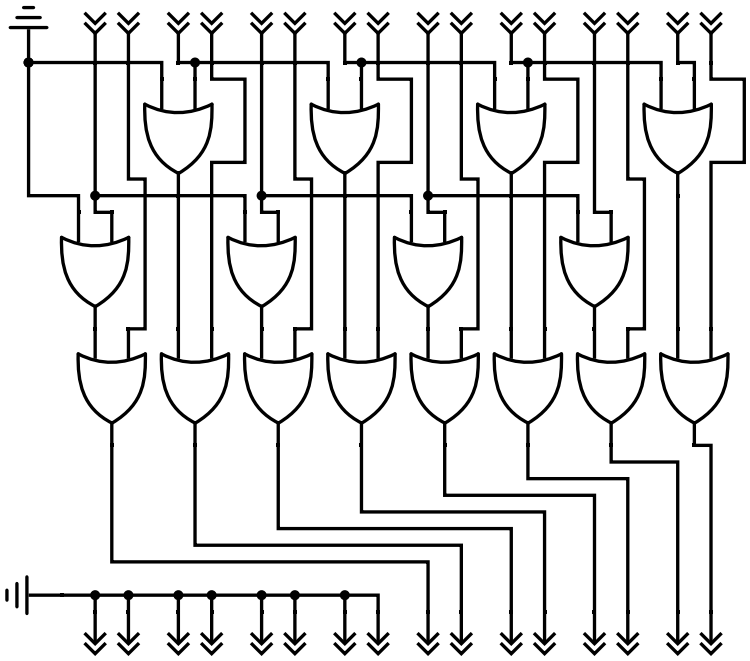


Table 4.1: A `C` implementation of the shifting nth root algorithm. The shifting nth root algorithm approximates the square root one bit at a time, and is as accurate as the precision allows. All calculated bits are stored in the variable `res`.

```
int isqrt(int num) {
    int res = 0;
    int bit = 1 << 30;

    while(bit != 0) {
        if(num >= res + bit) {
            num -= res + bit;
            res = (res >> 1) + bit;
        } else res >>= 1;
        bit >>= 2;
    }
    return res;
}
```

Table 4.2: A `C` implementation of the shifting nth root algorithm. This code example is slightly longer, but relatively expensive additions have been replaced by bitwise ors. The bit-shift of `res` has been moved, removing the else-statement, and the comparison has been rewritten to require less logic when implemented in hardware.

```
int isqrt(int num) {
    int res = 0;
    int bit = 1 << 30;

    while(bit != 0) {
        int tmp = num - (res | bit);
        res >>= 1;
        if(tmp >= 0) {
            num = tmp;
            res |= bit;
        }
        bit >>= 2;
    }
    return res;
}
```

5

Results

In this chapter the results of the GPU design choices and the algorithmic optimizations and their effects are presented.

5.1 GLSL Shader

Our GLSL shader can render simple scenes in real-time but performance degrades rapidly as the number of objects rendered increase. Scenes can easily be made to look more complex than they are. For example by using mod fields, where the modulo operation is used on an object to creates a field with multiple copies of the original object next to each other. The hardware (Geforce GTX 1060M) that the shader was tested on was able to render 20 reflective spheres in real time in 1920x1080 resolution using our performance enhancing algorithm. Examples of scenes rendered using our GLSL shader can be seen in appendix B.

5.2 GPU

The GPU has been tested by simulating it in CλaSH. In the simulation a program is executed on the GPU, which renders a scene using Sphere Tracing.

It spawns threads progressively to fill the screen data buffer with pixels. The simulated execution of our test programs ran exactly as intended, with any number of cores. The design has also been put through a synthesizing tool for FPGAs, which creates a net list of the components and connections that constitute the GPU. This also works as intended without any unexpected problems, but the actual operation of the GPU on an FPGA has not yet been verified.

5.3 Square roots

The accuracy of the different square root approximation and calculation methods are shown in figures 5.1, 5.2, 5.3, 5.4, 5.5, and 5.6. The shifting nth root algorithm is used as a reference in all figures because it is always bit-accurate for square roots. All graphs show integer values only but the absolute and relative error comparisons are against real-valued square roots.

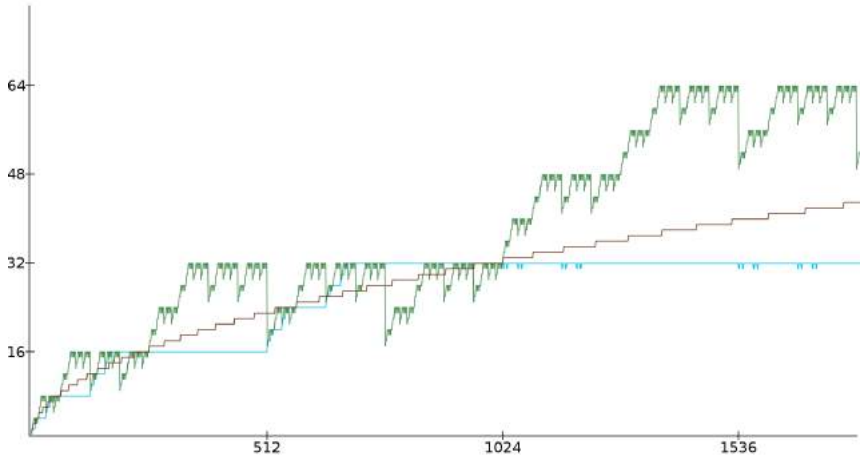


Figure 5.1: Value from the simple square root approximator (green), the improved version (blue), and the shifting nth root algorithm (red).

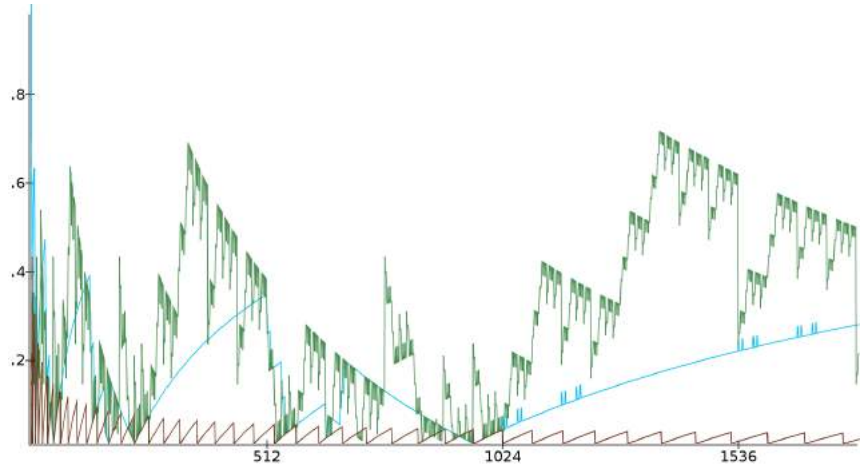


Figure 5.2: Relative error for the simple square root approximator (green), the improved version (blue), and the shifting nth root algorithm (red).

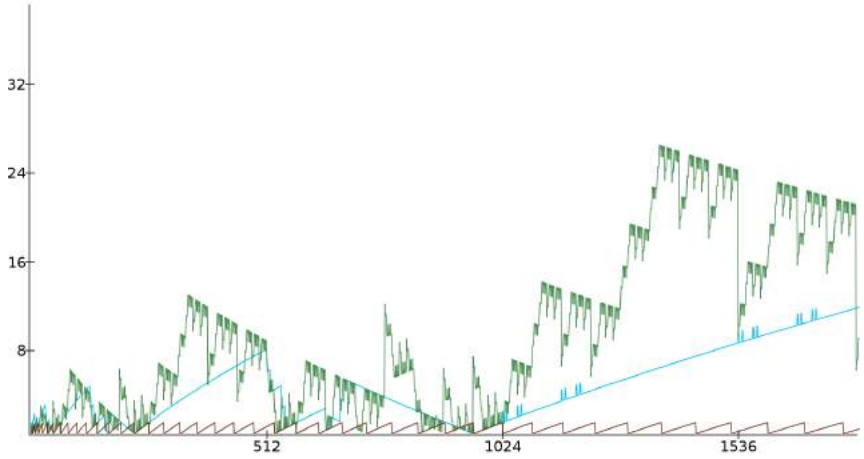


Figure 5.3: Absolute error for the simple square root approximator (green), the improved version (blue), and the shifting nth root algorithm (red).

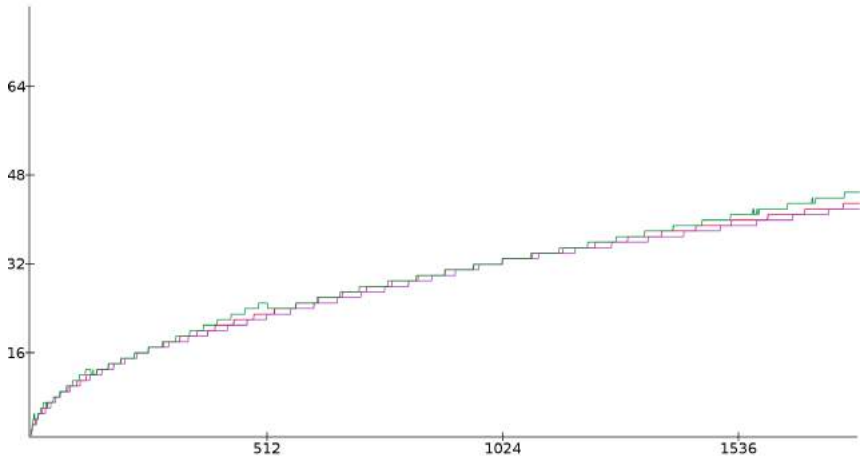


Figure 5.4: Value from lerp-approximator (purple), simple square root approximator with one step of the Babylonian method (green), and the shifting nth root algorithm (red). In these graphs, they all operate on integers. The shifting nth root is exact for integer square roots.

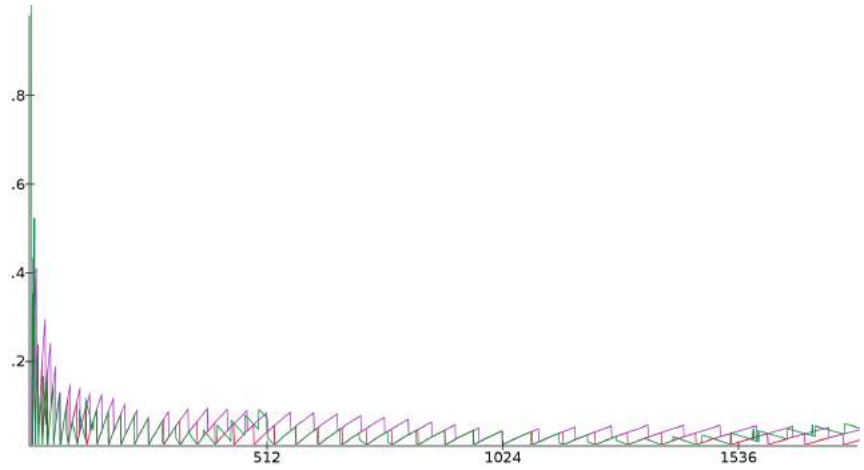


Figure 5.5: Relative error for the lerp-approximator (purple), simple square root approximator with one step of the Babylonian method (green), and the shifting nth root algorithm (red).

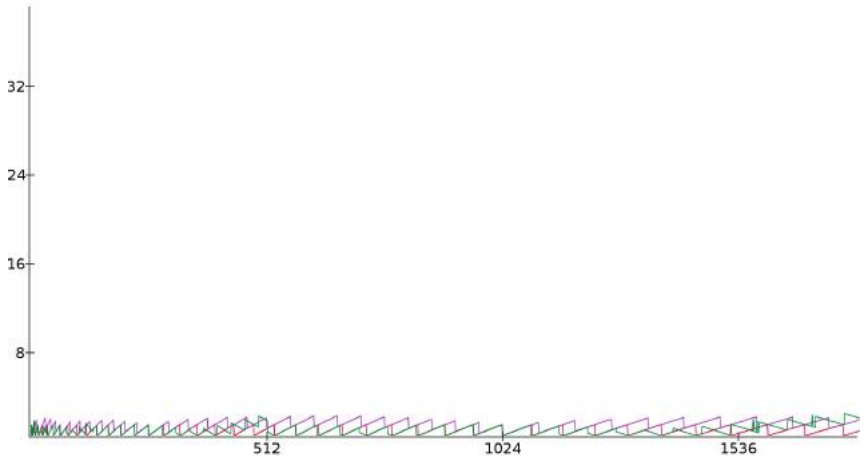


Figure 5.6: Absolute error for the lerp-approximator (purple), simple square root approximator with one step of the Babylonian method (green), and the shifting nth root algorithm (red).

5.4 Optimizations

During this project, a number of optimizations were discussed and developed. Implemented optimizations are explained below and the theoretical ones are presented in chapter 6.6 and 6.4. Some of these are based on earlier work, while others are believed to be quite unique optimizations which have not been discussed for sphere tracing previously. All implemented optimizations that affects the algorithm were implemented in the software shader, not on our own GPU.

For all tests performed, FPS (Frames Per Second) is used to measure performance.

Objects	Optimized	Unoptimized
1	600	350
5	430	180
10	290	98
15	85	13
20	58	9
25	40	6
30	29	4
35	7	3
40	6	2
45	4	1.5

Table 5.1: Frames generated per second using the GLSL shader with and without the orthogonal culling optimization.

FPS is simply how many times the GPU is able to render the scene per second. The time it takes to render the scene once is equal to one divided by the FPS.

5.4.1 Orthogonal culling

Tests to examine the performance gain of the Orthogonal Culling optimization were performed by putting an increasing number of solid-colored spheres in a plane in front of the camera. Because of this, the spheres are not obstructed by other spheres, making this the best possible scenario for the optimization. In all scenarios tested there was an increase in performance when using the optimization, but the increase varied with the number of objects and how they were set up.

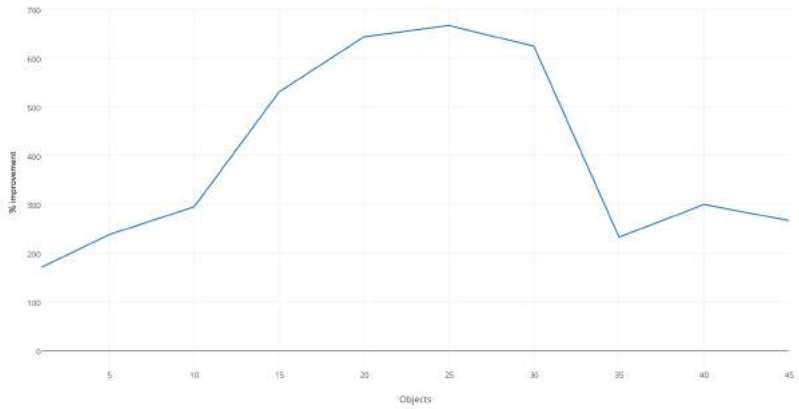


Figure 5.7: The performance difference of the optimized and unoptimized version, displayed in percentage.

5.4.2 Bounding spheres

Bounding spheres were implemented and tested. There was a clear gain in performance in some cases, but the results are situational. The performance gain depends on number of objects bound, the spacing between them, how well the bounding sphere fits the objects, etc.

To test the performance of this method, based on number of enclosed objects, a scene with an successively increasing amount spheres was rendered and the FPS measured. For reference an identical scene was rendered but without the optimization running.

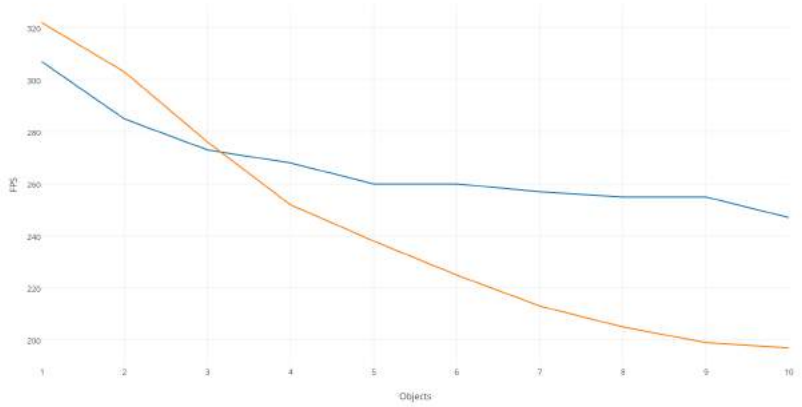


Figure 5.8: Results from Bounding Sphere tests. Blue is with Bounding Spheres, red is without optimization. X-axis being the number of objects enclosed by a single Bounding Sphere, Y-axis showing the frames per second(FPS).

Figure 5.8 shows that in this particular scene a performance gain is achieved when more than three objects are placed inside a Bounding Sphere. When 10 objects is in the Bounding Sphere we have a 25% increase in FPS. If fewer than 4 objects are enclosed, a performance decrease is observed.

5.4.3 Ray Grouping

The ray grouping optimization was implemented and performance tests were done, the results of which can be seen in table 5.2. These tests rendered a scene containing three spheres at different distances on a 64 by 64 as well as a 128 by 128 pixel display. Ray grouping lowered the execution time when the maximum group size was set to 2, but did not decrease significantly further when increasing the group size to 4 or 8. When rendering to a 128 by 128 pixel display, the relative performance improved slightly more than for the smaller display.

Group size	64 by 64	128 by 128
1	128k	512k
2	84k	328k
4	80k	312k
8	82k	313k

Table 5.2: Number of clock cycles until frame rendering was complete for different group and display sizes. Note that a group size of one is equivalent to not using ray grouping.

6

Discussion

This chapter will discuss our result, some not yet implemented optimizations, and what future work is possible from this point in time.

6.1 Development Environment

In this section we describe some project specific tools and our experience using them.

6.1.1 CλaSH

As described in section 2.4, the ability to simulate the function of the GPU directly in a REPL [11] was essential for us being able to complete a working GPU in the time span available. The way CλaSH handles state is quite elegant and effective once one has learned the general pattern. Lists however, are too abstract for synthesis so CλaSH uses its own vector type instead. Most regular data functions are implemented, but all such vectors must have a fixed size. This makes some operations such as combining two vectors rather cumbersome and one has to work around this.

6.1.2 Synthclipse

This is an IDE for developing shaders, using GLSL or JSX. It has features for rapid recompilation and testing, easy resource loading and Uniform Controls. The Uniform Controls are a collection of user interface elements such as sliders and color pickers, which can be saved into Presets. These allowed us to navigate our scene and modify its parameters on the fly, enabling visual examination and debugging of our algorithms.

6.2 Software Shader

The optimizations that were implemented surprised with the enormous performance increase they gave. There are many optimizations that are yet to be implemented and tested and it's hard to estimate the potential achievable performance of a shader on conventional GPU. The only thing that is certain is that there are massive performance potentials of a shader on conventional hardware. To develop a GLSL shader for algorithmic development and testing was a great decision as it helped us understand the algorithm and how to improve it.

GLSL has some very attractive features, it was easy to learn because the syntax has much in common with C++. This makes the time it takes to develop new features and test them short. The purpose of the software shader was to study the algorithm, implement and test optimizations and develop new features. For the purposes of this project GLSL was a perfect fit.

6.3 GPU

The project originally intended to create a simple single-core GPU that could fully render a scene given enough time, and then add components as time allowed. Every design decision one makes clarifies what further needs to be done, and reveals previously unseen flaws in the design that must be resolved. In the end we implemented the core and a storage manager which enabled multiple cores running in parallel.

The storage manager turned out to need to be more complex than we originally anticipated. In its first implementation, the storage in the storage manager was a simple FIFO queue. This worked well for very simple shader programs, but we found that it was not possible to write a full Sphere Tracer using this storage type. This was not due to any complexity inherent in the sphere tracing algorithm, but rather a consequence of how the GPU requires the shader programs to keep track of all pixels they need to render. This is solved by having the shader generate a new thread for every pixel it needs to render, and then running the actual pixel shader code on that thread. This fails when threads need to execute many jumps or loops, which are also solved by spawning new threads. Whenever a thread executed a jump, the pixel spawning thread took priority because of the FIFO storage, and a new pixel thread would be spawned. After many jumps, the storage in the storage manager would fill up entirely, and all cores would stall waiting for the storage manager allowing them to send it more threads. This was solved by changing the internal storage to a double-ended queue, giving the threads some control over execution priority.

If we were to continue working on this project another workflow would be adopted when developing the GPU. More detailed planning is required in the early stages and empiric testing of different implementations is required along the entire development to reach the best possible solution.

6.4 Optimizations

6.4.1 Bounding spheres

If used the right way this technique can give great performance boosts. The testing showed an increase in gain as the number of objects increased inside the sphere, which is promising.

The test was in this case simpler than the issue. The results depend on more factors than we can analyze in our time frame, the test only handles the number of objects inside a fixed sphere. To be used in a optimal way the objects should be bound in a way similar to the one described in the subsection implementation4.1 but more adaptive, enclosing objects depending how they are grouped per frame. It should also not try to compute the sphere definition exactly but rather approximate it for increased performance.

6.4.2 Orthogonal culling

The performance gain from this optimization was surprisingly high. However great the performance gain, this is still far from the optimal implementation of Orthogonal culling. It currently performs Orthogonal Culling on all the objects in the scene, per pixel. Instead, culling should be done once per object and calculate which rays an object intersects, this way fewer calculations would have to be performed and optimally this should be performed by the CPU.

This method could work together with the Bounding Sphere optimization. Instead of projecting single objects onto the ray, bounding spheres could be culled, decreasing the number of orthogonal projections that has to be made. This was tested and an increase in performance was observed.

6.4.3 Ray grouping

Ray grouping improved performance by lowering the total number of marching steps by decreasing the total number of initial rays. The improvements seemed to be more significant for higher resolutions, which is reasonable, given that the cost at each step for a given group is not dependent on the number of rays in the group. This could be very useful when rendering on high-resolution displays.

We had initially expected the performance improvements to be greater than those achieved. There is, however, a significant amount of overhead incurred from splitting groups of rays into smaller groups, which is the likely cause of both the decreased improvements as well as the diminishing returns when increasing the group size to a significant portion of the display size. In addition to this, we know that our

implementation is slightly too eager to split groups; improving this could reduce execution time further.

Ray grouping was also the only optimization that was implemented on the GPU. The hardware support for threads and the ability to create new threads dynamically worked very well with this optimization. This optimization also reduces the amount of rays that have similar execution flows and might therefore be less effective on traditional lockstepped architectures, although it should probably still be possible to implement ray grouping and improve performance on those GPUs.

6.5 Square roots

The simple square root approximator and its improved version are extremely simple circuits, in fact they are significantly smaller than most other common operations performed on numbers in hardware. Of course, they are far from accurate, but we find it interesting that it is possible to achieve a bounded relative error from the true square root for any number of input bits with these small circuits. When combined with linear interpolation or iterative methods like the Babylonian method the accuracy of these approximations increases significantly.

The shifting nth root algorithm is the only bit-accurate method tested, and it performs well, albeit slower than the approximations. If non-exact results are acceptable, which they often are to some extent when doing sphere tracing, the number of steps in the shifting nth root algorithm could be reduced, which would both increase speed and decrease area usage. The lerp-based and iterative methods might still be able to reuse common components such as adders and multipliers however, and might therefore be preferable.

For the GPU, we have used the shifting nth root algorithm in order to not have to consider possible accuracy problems when debugging shaders, but all methods are implemented and ready to be switched out at any time.

6.6 Future work

6.6.1 GPU-CPU cooperation

In the current implementation everything is performed on the GPU, including object transformations and culling. Currently if objects in the scene are supposed to move they have to be moved using mathematical functions, such as sinus. These functions are then evaluated for each march step. This is essentially wasted computing power and could be performed on the CPU as is standard in modern graphics engines.

6.6.2 Plane-ray intersection

Infinite planes currently require a lot of computing power to render. When the camera is oriented so that the field of view is along an infinite plane some rays will travel parallel to the plane. These rays will march equally long steps until their max number of steps has been taken or the max range has been reached, without hitting anything. This will cause some rays to draw computing power without ever being able to hit the plane anyway. Some rays that should hit the plane will fail to do so because of to many steps taken, making it look like the plane has an edge because after a certain range it is no longer being rendered.

One potential problem is that because no distance field evaluations are performed, it might be hard to perform Boolean operations or mathematical deformations on planes rendered this way. Another is that every time we add more features that aren't part of the regular Sphere Tracing, the complexity of the program increases. Thus, the advantages of the individual optimizations must be weighed against the overall performance loss they incur.

6.6.3 Overstepping

Bounding spheres technique is somewhat similar to the normal sphere tracing. The difference is that when you march you march to the edge of the MDS and then you march to the next MDS edge. With overstepping technique you march a small distance further outside the MDS edge. You then compare the original MDS with the new MDS if these two spheres overlap in any way we can march that little bit further. By marching that little bit further, a decrease in the number of times marched is achieved, giving an increase in performance. This has previously been discussed by [15]. If the two MDS's do not overlap it steps back to the edge of the first MDS, not overstepping, then continuing like before.

6.7 Future impact on industry

The GPU is still needs to be developed further in order to definitively show if the algorithm has potential one way or another. Further prototypes need to be built with greater rigor before that can happen. If we however assume that somewhere down the line it turns out to be a success, then there are generally three ways in which we could foresee that happening.

First up is by breaking through in the games industry. Granted that it probably will not be able to outperform modern graphic cards at their own game even after improving the card as much as possible. But the up-and-coming VR technology faces different challenges, usually rendering smaller scenes with fewer objects and capturing actual "3D" images to display to the user. Both of these factors favors

Sphere Tracing since it inherently produces perfect 3D objects and works much faster in scenes with shorter render distance.

Another possibility is that the GPU turns out to be too slow for rendering games in real time, but runs other Ray Tracing algorithms commonly used in movie production faster that modern graphic cards. This is not too unlikely since the calculation of the rays in both cases follow many similar patterns in bouncing and reflecting etc. Since movies render in render farms where like server farms, lots of copies of the same hardware works in tandem. Meaning that even a small improvement over current graphic cards might be amplified to a significant gain in computational power.

Finally, we designed our GPU similar to a CPU but with more cores like a GPU except with no lockstepping. If this proves to be feasible for a full ASIC (Application-specific integrated circuit), then this type of processing unit would be better at parallel processing than a regular CPU, and it would perform better than a regular GPU at recursive and conditionally branching algorithms. This lends itself well to computations such as fractals, population simulation and artificial intelligence. It might also fit classes of computation not yet thought of, such in the example of Folding@Home [4]. They created software, deployed all over the world, to run on Graphics cards in home computers, when the computer was idle. So instead of wasting precious computing power, these GPUs, adept at parallel computations, were repurposed to work on the complicated problem of protein folding. In effect acting as a global supercomputer to aid medical research. Given the uniqueness of our GPU it might find a similar novel use in the future.

7

Conclusions

The goal of the project was to design and create a GPU for the Sphere Tracing algorithm and try to improve its performance. A working GPU was designed together with two shaders, one written in our own assembly language running on the GPU and one written in GLSL running on a traditional consumer graphics card. We also simulated the GPU, rendering to a small virtual screen. The GPU executes as intended and, if programmed correctly, can Sphere Trace complete scenes.

For the GLSL shader we describe and tried three optimizations, Bounding Spheres, Orthogonal Culling and Ray Grouping. Bounding Spheres decrease the number of calculations made in each march by enclosing several objects into one bounding sphere. Orthogonal Culling is used to tell whether a object is in front of the ray or not, decreasing the number of objects needed to calculate the signed distance function. Ray Grouping decreases the number of of steps needed to march by grouping rays together into blocks. All of them improved performance. We believe they can be improved further by offloading parts of the computation to the CPU.

For the hardware optimization, calculating square roots was considered important and methods for calculating them efficiently where investigated. We found a number of fast approximations that are bounded in their relative error. We also looked at possible optimizations for an exact integer square root algorithm.

Bibliography

[1] A. Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference on - AFIPS '68 (Spring)*, page 37, New York, New York, USA, 1968. ACM Press. URL: http://portal.acm.org/citation.cfm?doid=1468075.1468082, doi:10.1145/1468075.1468082.

[2] U. Assarsson and T. Moller. Optimized view frustum culling algorithms for bounding boxes. *Journal of Graphics Tools*, 5(1):9–22, 2000. URL: http://dx.doi.org/10.1080/10867651.2000.10487517, doi:10.1080/10867651.2000.10487517.

[3] C. Baaij. *CλasH : from Haskell to hardware*. PhD thesis, University of Twente, dec 2009. URL: http://essay.utwente.nl/59482/.

[4] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande. Folding@home: Lessons from eight years of volunteer distributed computing. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8. IEEE, may 2009. URL: http://ieeexplore.ieee.org/document/5160922/, doi:10.1109/IPDPS.2009.5160922.

[5] P. Bjesse, K. Claessen, M. Sheeran, S. Singh, P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming - ICFP '98*, volume 34, pages 174–184, New York, New York, USA, 1998. ACM Press. ISBN 1581130244. URL: http://portal.acm.org/citation.cfm?doid=289423.289440, doi:10.1145/289423.289440.

[6] G. Chen. A Short Historical Survey of Functional Hardware Languages. *International Scholarly Research Network ISRN Electronics*, 11, 2012. doi:10.5402/2012/271836.

[7] P. H. Christensen, J. Fong, D. M. Laur, and D. Batali. Ray Tracing for the Movie 'Cars'. *IEEE Symposium on Interactive Ray Tracing 2006*, 2006.

[8] J. Granskog. CUDA Ray Marching, 2017. URL: http://granskog.xyz/blog/2017/1/11/cuda-ray-marching.

[9] J. C. Hart. Sphere tracing: a geometric method for the antialiased ray tracing of

implicit surfaces. *The Visual Computer*, 12(10):527–545, 1996. doi:10.1007/s003710050084.

[10] J. C. Hart, D. J. Sandin, and L. H. Kauffman. Ray tracing deterministic 3-D fractals. *ACM SIGGRAPH Computer Graphics*, 23(3):289–296, jul 1989. URL: http://portal.acm.org/citation.cfm?doid=74334.74363, doi:10.1145/74334.74363.

[11] T. Hey and G. Pápay. *The Computing Universe: A Journey through a Revolution*. 2014. ISBN 978-1-31612322-5.

[12] M. Houston. CS448S – Spring 2010 – Beyond Programmable Shading GPU Architecture CS448S – Spring 2010 – Beyond Programmable Shading, 2010. URL: https://graphics.stanford.edu/wikis/cs448s-10/FrontPage?action=AttachFile&do=get&target=CS448s-10-03.pdf.

[13] B. Keinert, J. Korndörfer, and U. Ganse. hg_sdf. URL: http://mercury.sexy/hg_sdf/.

[14] J. Kessenich, D. Baldwin, and R. Rost. The OpenGL® Shading Language. 2016. URL: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.50.pdf.

[15] J. Korndörfer, M. Stamminger, and B. Keinert. Enhanced Sphere Tracing. *STAG: Smart Tools & Apps for Graphics*, page 8, 2014. URL: http://erleuchtet.org/~cupe/permanent/enhanced_sphere_tracing.pdf, doi:10.2312/stag.20141233.

[16] R. Lauwereins and J. Madsen. *Design, Automation, and Test in Europe*. 2008. ISBN 9781402064883.

[17] Y. Li and W. Chu. Implementation of Single Precision Floating Point Square Root on FPGAs. *FCCM '97 Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, pages 226–, 1997.

[18] B. T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, jun 1975. URL: http://portal.acm.org/citation.cfm?doid=360825.360839, doi:10.1145/360825.360839.

[19] I. Quilez. modeling with distance functions. URL: http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm.

[20] I. Quilez. Modeling with Signed Distance Fields. URL: http://www.iquilezles.org/www/articles/sdfmodeling/sdfmodeling.htm.

[21] I. Quilez. Rendering Worlds with Two Triangles with raytracing on the GPU in 4096 bytes, 2008. URL: http://www.iquilezles.org/www/material/nvscene2008/rwwtt.pdf.

[22] P. Shirley, M. Ashikhmin, and S. Marschner. *Fundamentals of Computer Graph-*

ics. 2005. ISBN 9781568812694.

[23] I. te Raa. *Recursive functional hardware descriptions using CλaSH*. PhD thesis, University of Twente, 2015. URL: http://essay.utwente.nl/68804/.

[24] M. T. Tommiska. Area-efficient implementation of a fast square root algorithm. Technical report, Helsinki University of Technology, Laboratory of Signal Processing and Computer Technology, 2000. doi:10.1109/ICCDCS.2000.869869.

[25] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, jun 1980. URL: http://portal.acm.org/citation.cfm?doid=358876.358882, doi:10.1145/358876.358882.

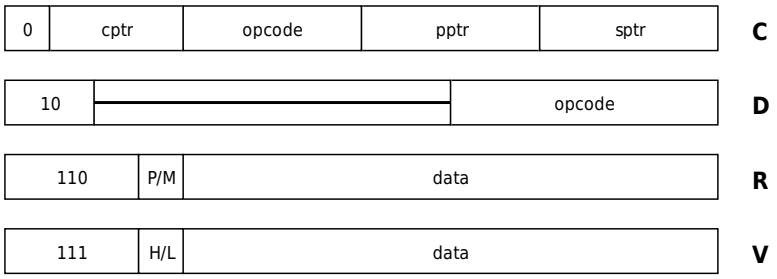
[26] S. Woop, J. Schmittler, and P. Slusallek. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. In *ACM SIGGRAPH 2005 Papers on - SIGGRAPH '05*, page 434, New York, New York, USA, 2005. ACM Press. URL: http://portal.acm.org/citation.cfm?doid=1186822.1073211, doi:10.1145/1186822.1073211.

A

GPU Instruction Listing

In this appendix, all instructions of the GPU are listed and explained. On the left hand side of each instruction explanation is a diagram showing the changes each instruction applies to the stack. Instruction encodings for all instructions are shown in figure A.1.

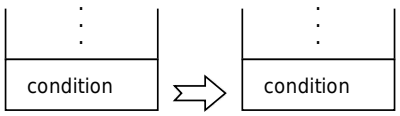
Figure A.1: Bit encoding for the different instruction types. All instruction types have a fixed width 16-bit encoding.



A.1 C-type Instructions

pushf

`cond pushf` sends information from the tread registers to a pixel in the frame buffer, if `cond` is true for the bottom value of the stack. Any previous information associated with that pixel is overwritten.



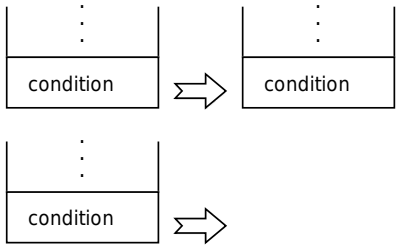
Thread register `1` is used by the frame buffer as a pixel id. Pixel ids are sequential, the pixel with the highest id corresponds to the lower right corner on the screen, and pixel id zero corresponds to the top left pixel.

Thread register `2` is used by the frame buffer to determine pixel color. The color information must be stored as 24-bit RGB color in the highest 24 bits of the register. The lowest 8 bits of register `2` are not used.

`pushf` does not affect the stack.

pushs

`cond pushs` duplicates the current thread and adds it to the storage manager with a high execution priority, if `cond` is true for the bottom value of the stack.



The argument is *not* removed from the stack for the parent thread.

Thread register `0` is used to determine where the copied thread will start execution when it is sent to a core.

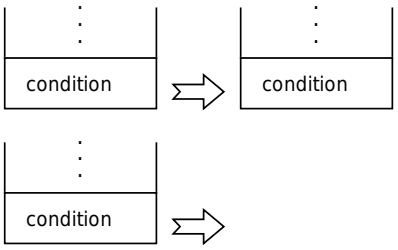
`pushs` does not use any of the values on the stack.

No values are returned to the stack of the parent thread.

All values of the stack are discarded for the child thread.

pushq

`cond pushq` duplicates the current thread and adds it to the storage manager with a low execution priority, if `cond` is true for the bottom value of the stack.



The argument is *not* removed from the stack for the parent thread.

Thread register `0` is used to determine where the copied thread will start execution when it is sent to a core.

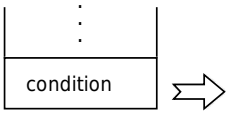
`pushq` does not use any of the values on the stack.

No values are returned to the stack of the parent thread.

All values of the stack are discarded for the child thread.

drop

`cond drop` discards the current thread, if `cond` is true for the bottom value of the stack.



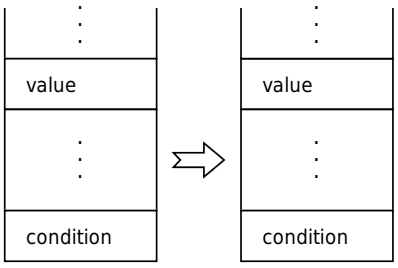
The argument is removed from the stack.

All thread registers are discarded.

All stack values are discarded.

setval

`cond setval pptr sptr` sets the value of thread register `pptr` to the value on the stack at location `sptr`, if `cond` is true for the bottom value of the stack.



The arguments are *not* removed from the stack.

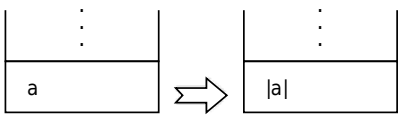
The value of thread register `pptr` is changed.

No arguments are returned to the stack.

A.2 D-type Instructions

abs

`abs` computes the absolute value of a fixed-point or integer value on the stack.

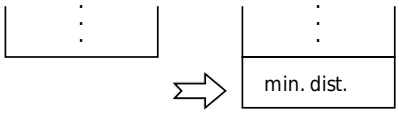


The argument is removed from the stack.

The result is stored at the bottom of the stack.

acc

`acc` returns the accumulated minimum distance updated with the `next` instruction, stored in the core

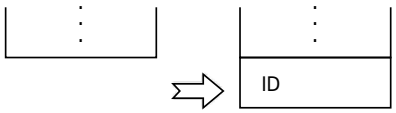


No arguments are removed from the stack. The accumulated minimum is not affected.

The result is stored at the bottom of the stack.

accid

`accid` returns the ID associated with the accumulated minimum distance updated with the `next` instruction, stored in the core



No arguments are removed from the stack. The accumulated minimum is not affected.

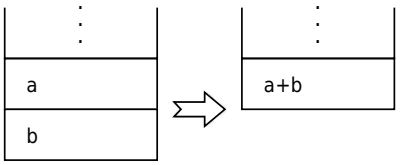
The result is stored at the bottom of the stack.

`add`

`add` adds two fixed-point or integer values on the stack.

Both arguments are removed from the stack.

The result is stored at the bottom of the stack.

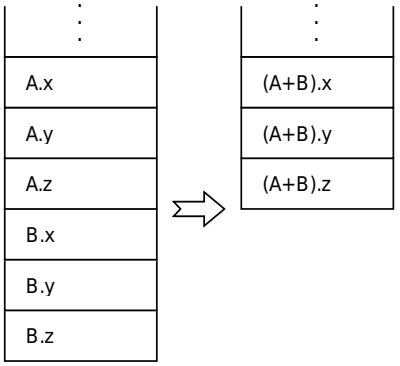


`addv`

`addv` adds two three-dimensional fixed-point or integer vectors on the stack. Vectors are stored on the stack as three consecutive values with the x-coordinate at the top and the z-coordinate at the bottom.

All arguments are removed from the stack.

The result is stored at the bottom of the stack.

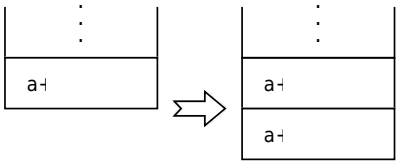


`copy`

`copy` duplicates the bottom value of the stack.

The argument is *not* removed from the stack.

The result is stored at the bottom of the stack.

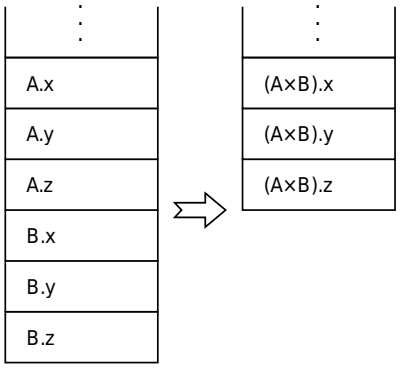


`cross`

`cross` computes the cross product of two three-dimensional fixed-point or integer vectors on the stack. Vectors are stored on the stack as three consecutive values with the x-coordinate at the top and the z-coordinate at the bottom.

All arguments are removed from the stack.

The result is stored at the bottom of the stack.

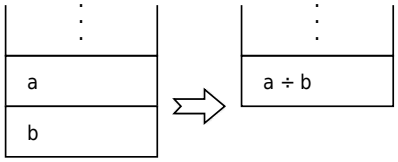


`div`

`div` divides two fixed-point or integer values on the stack. The result is undefined if the divisor is zero.

Both arguments are removed from the stack.

The result is stored at the bottom of the stack.

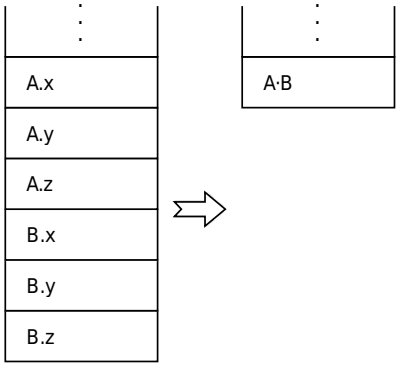


`dot`

`dot` computes the dot product of two three-dimensional fixed-point or integer vectors on the stack. Vectors are stored on the stack as three consecutive values with the x-coordinate at the top and the z-coordinate at the bottom

All arguments are removed from the stack.

The result is stored at the bottom of the stack.

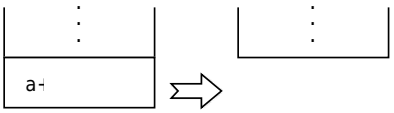


`dropv`

`dropv` removes the bottom argument from the stack.

The argument is removed from the stack.

No result.

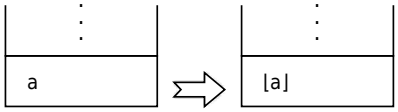


`floor`

`floor` rounds a fixed-point value from the stack down to the closest whole number. Alternatively, `floor` rounds an integer down to its closest multiple of 2^{16}.

The argument is removed from the stack.

The result is stored at the bottom of the stack.

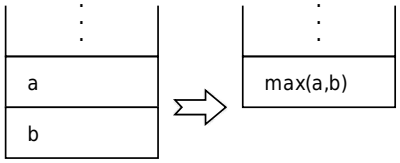


`max`

`max` computes the maximum of two fixed-point or integer values on the stack.

Both arguments are removed from the stack.

The result is stored at the bottom of the stack.

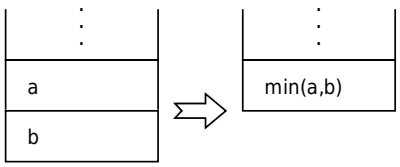


`min`

`min` computes the minimum of two fixed-point or integer values on the stack.

Both arguments are removed from the stack.

The result is stored at the bottom of the stack.

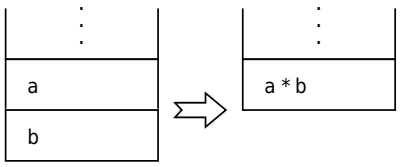


`mul`

`mul` multiplies two fixed-point or integer values on the stack.

Both arguments are removed from the stack.

The result is stored at the bottom of the stack.

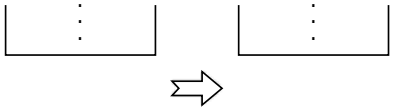


`nop`

`nop` does nothing.

No arguments.

No results.

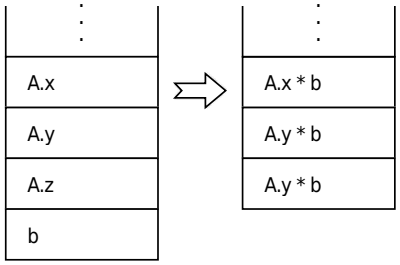


`scale`

`scale` multiplies all components of a fixed-point or integer vector on the stack with a single scalar factor on the stack. Vectors are stored on the stack as three consecutive values with the x-coordinate at the top and the z-coordinate at the bottom. The scalar factor must be stored below the vector on the stack.

All arguments are removed from the stack.

The result is stored at the bottom of the stack.

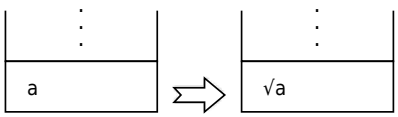


`sqrt`

`sqrt` takes the square root an *unsigned* fixed-point value on the stack.

The argument is removed from the stack.

The result is stored at the bottom of the stack.

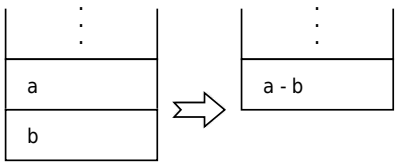


sub

`sub` subtracts two fixed-point or integer values on the stack.

Both arguments are removed from the stack.

The result is stored at the bottom of the stack.

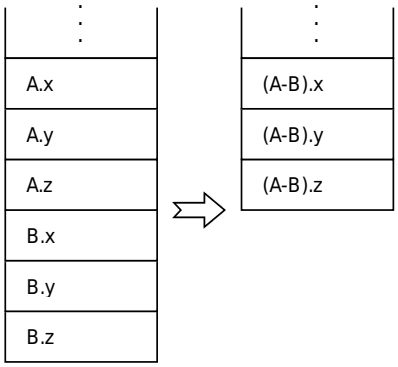


subv

`subv` subtracts two three-dimensional fixed-point or integer vectors on the stack. Vectors are stored on the stack as three consecutive values with the x-coordinate at the top and the z-coordinate at the bottom.

All arguments are removed from the stack.

The result is stored at the bottom of the stack.

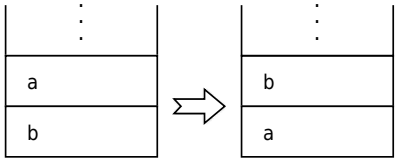


swap

`swap` swaps two values on the stack.

Both arguments are removed from the stack.

The results are stored at the bottom of the stack.



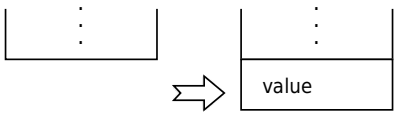
A.3 R-type Instructions

pack

`pack pptr` loads the value from thread register `pptr` to the bottom of the stack.

No arguments are taken from the stack.

The result is stored at the bottom of the stack.

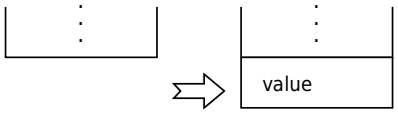


val

`val mptr` loads the value from immutable memory location `mptr` to the bottom of the stack.

No arguments are taken from the stack.

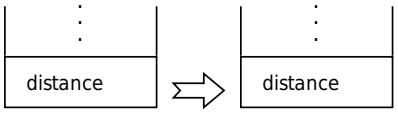
The result is stored at the bottom of the stack.



A.4 `V`-type Instructions

`next`

`next id` updates the accumulated minimum distance register with the bottom value of the stack as the distance, and `id` as the ID.

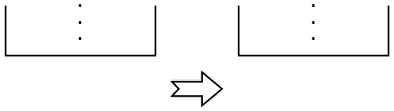


The argument is *not* removed from the stack.

No results are generated on the stack. The accumulated minimum distance register is updated.

`nextclr`

`nextclr` clears the minimum distance register, setting the distance to the largest possible value.

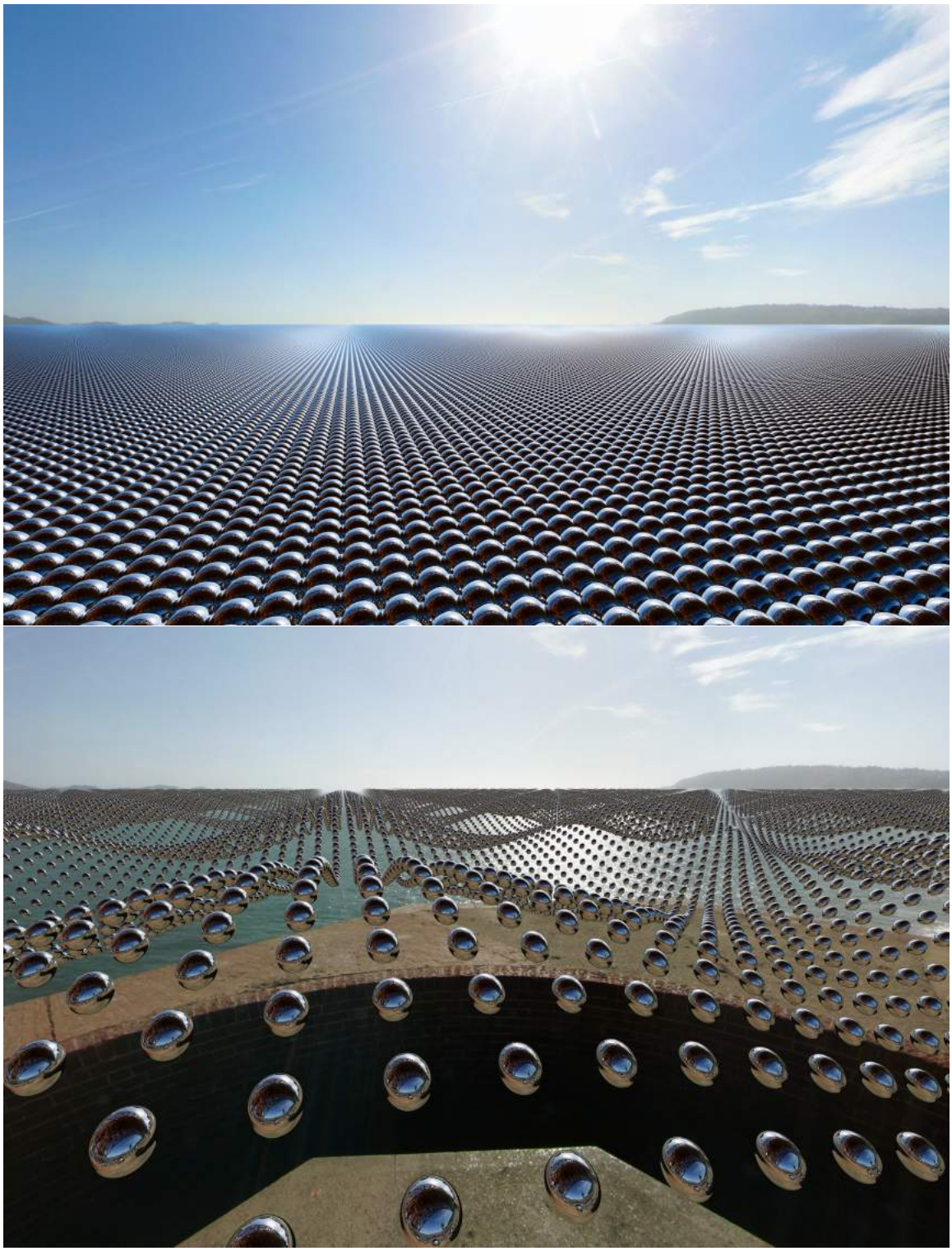


No arguments are taken from the stack.

No results are generated on the stack. The accumulated minimum distance register is updated.

B

Shader Graphics Showcase



C

Source Code

The source code for this project can be found at: https://github.com/hexointed/DATX02-17-12