NTNU

Innovation and Creativity

Programming graphic card for fast calculation of sound field in marine acoustics

Olav Haugehåtveit

Master of Science in Electronics

Submission date: June 2006

Supervisor: Jens Martin Hovem, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

The processing capasity in a graphics chip (GPU), which is used in games and animations, is much higher than in a regular CPU. This makes it interesting to use the GPU for relative small and simple tasks, that are repeated in many iterations. A part of an already written Matlab program shall be converted to run on a GPU. This program creates an underwater sound field using ray tracing.

Assignment given: 16. January 2006
Supervisor: Jens Martin Hovem, IET

Preface

This report is a result and documentation of a master assignment spring 2006, in the area of marine acoustics at the Department of Electronics and Telecommunication at the Norwegian University of Science and Technology (NTNU). The master report is written by Olav Haugehåtveit, with professor Jens. M. Hovem as technical supervisor. Trond Runar Hagen with SINTEF ICT Applied Mathematics in Oslo has been assisting with the computer programming.

Trondheim June 14, 2006

Olav Haugehåtveit

Olav Haugehåtveit

Abstract

Commodity computer graphics chips are probably today's most powerful computational hardware one can buy for money. These chips, known generically as Graphics Processing Units or GPUs, has in recent years evolved from afterthought peripherals to modern, powerful programmable processor. Due to the movie and game industry we are where we are to today. One of Intel's co-founder Gordon E. Moore said once that the number of transistors on a single integrated chip was to double every 18 month. So far this seems to be correct for the CPU. However for the GPU the development has gone much faster, and the floating point operations per second has increased enormously.

Due to this rapid evolvement many researchers and scientists has discovered the enormous floating point potential can be taken advantage of, and a numerous applications has been tested such as audio and image algorithms. Also in the area of marine acoustics this has become interesting, where the demand for high computational power is increasing.

This master report investigates how to make a program capable to run on a GPU for calculating an underwater sound field. To do this a graphics chips with programmable vertex and fragment processor is necessary. Programming this will require graphics API like OpenGL, a shading language like GLSL, and a general purpose GPU library like Shallows. A written program in Matlab is the basic for the GPU program. The goal is to reduce calculation time spent to calculate a underwater sound field.

From this the increment from Matlab to GPU was found to be around 40-50 times. However if Matlab was able to calculate the same number of rays as maximum on the GPU, the increment would probably be bigger. Since this study was done on a laptop with nVidia GeForce Go 6600 graphics chip, a higher gain would theoretically be obtainable by a desktop graphics chip.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter explains why GPU are of general interest as a computational resource for developers and scientists. An outline for the rest of the report is introduced. Further reading requires some technical insight in marine acoustics and computer graphics programming.

1.1 Motivation

The main area of applications for graphics processing units (GPUs) has been in the movie and game industry, enabling fast rendering of anti aliased, textured and shaded geometric primitives. As their performance and functionalities have been increasing and now give support for floating point computation in addition to that compilers for high-level programming languages have been released, many new algorithm and application have been suggested. These try to take advantages of the parallelism and vector processing capabilities of the GPUs.

The reason for focusing on the GPU are numerous. One reason is, if you are already planning to visualize your data, you can remove unnecessary data streams from the CPU to the GPU.

The GPU offers a parallel model for programming with internal parallel operations which practically is a broad vector model with varying size on the matrix . The GPU has a much higher performance compared to the CPU, and it gives access to high internal bandwidth. When programming we have to focus on exploiting this internal bandwidth, and the parallel processing capabilities.

Using the GPU as a co-processor can in light of this improve many numerical tasks. The CPU can than be used for other tasks as the GPU is left with much of the computational work. Hopefully this will achieve good performance width low cost.

1.2 Outline

In chapter 2 the theory for ray tracing in marine acoustics is presented, together with the whole PlaneRay model. Chapter 3 contains an explanation of how a GPU operates. Further in chapter 4 the GPU application is explained. Results and discussion follows in chapter 5 and 6. At the end of chapter 6 a discussion on further work is presented.

As appendix some word explanations, GPU program usage, information on used software and hardware and how to set up Visual Studio for GPGPU programming follows. From this report an article about GPU for ray tracing in marine acoustics is written [20]. As the last appendix this article can be found, without its bibliography. However the references in the bibliography used in this report is the same. Full source code can be found at [26].

Chapter 2

Marine Acoustics Theory

This ray tracing of underwater sound field is based on a model called PlaneRay developed at Norwegian University of Science and Technology in Trondheim, Norway, by Jens M. Hovem et al. The whole model is presented here, however only the first part, the initial ray tracing without acoustic intensity and beam displacement, are implemented on the GPU since this is the demanding computational part.

2.1 PlaneRay model

The model PlaneRay is an acoustic propagation model based on ray tracing [6]. A special and essential feature of this model is a unique sorting and interpolation routine for efficient determination of large number of eigenrays, also for range dependent environments. The bottom is modeled with a plane wave reflection coefficient and in principle any number of acoustic or elastic layers can be included. Figure 2.1 shows the general shallow water propagation problem that is addressed in the model.

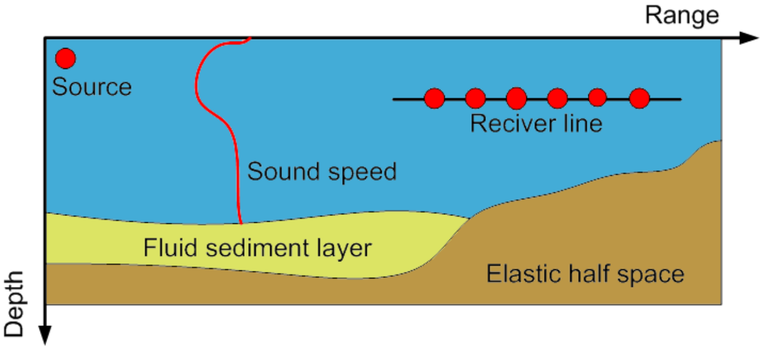


Figure 2.1: The PlaneRay model computes the received field from a source to receivers located on a horizontal line.

The receivers are located on a horizontal line in the water. The bottom can be a fluid sedimentary layer over an elastic half space and both can be range dependent. The sound speed profile can only be a function of depth and not with range. The effect of a layered bottom is included with plane wave reflection coefficient and rays are only traced to the water sediment interface and not into the bottom. The algorithm can be considered as having three stages:

- The initial ray tracing using a large number of rays to map out the entire sound field.
- Sorting and interpolation to determine the trajectorys and the ray history of the eigenrays connecting the source to the receiver.
- Synthesis of the acoustic field in frequency domain by coherently adding the contributions of the eigenrays, and calculation of the full-waveform time response by Fourier transformation.

2.1.1 Initial ray tracing

The input information is the range dependent bathymetry, a sound speed profile and the source location. The initial ray tracing is done by launching a relative large number of rays, with angles selected to cover the entire space between the source location and out to the receiver array. For each ray, the range and the travel time trough out the entire sound field is calculated. When the rays hits the receiver array, some values are recorded, as intersection angle, range and depth, together with location and the angle for reflection from the bottom and surface. All of this information is stored and used in following stages. In the current model the rays are not traced into the bottom and both the sound speed profile and the bathymetry are fixed, i.e the ray tracing is only executed once for each site.

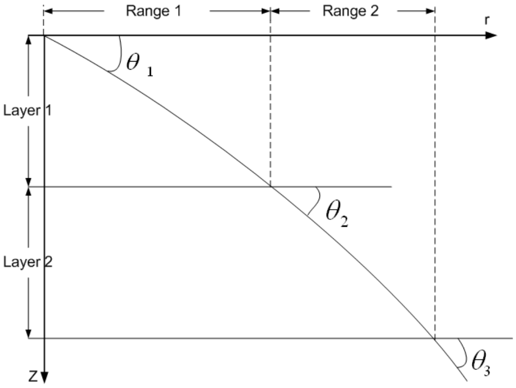


FIGURE 2.2: When the algorithm steps in depth with Δlayer, it calculates a range increment for each step. The sum of all ranges are the total range.

The implementation used in the PlaneRay model is to divide the water column into a large number of layers with the same thickness Δz, as shown in figure 2.2. Within each layer the sound speed profile is approximated with a straight line, and the sound speed profile can be written as

$$c(z) = c_i + g_i(z - z_i), \tag{2.1}$$

where c_i is the sound speed at depth z_i and g_i is the local sound speed gradient. Since the sound speed profile in each of this layers has a constant gradient, the ray in each layer follows a circular arc. The sum of all rays path within each layer is the total distance of the ray. The radius of this curvature is given by the local sound speed gradient,

$$g_i = \frac{c(z_{i+1}) - c(z_i)}{z_{i+1} - z_i}, \tag{2.2}$$

where $c(z_{i+1}) - c(z_i)$ is the sound speed difference from one layer to the next, and $z_{i+1} - z_i$ is the depth difference, and the ray parameter

$$\xi = \frac{cos\theta(z)}{c(z)} = \frac{cos\theta_0}{c_0}. \tag{2.3}$$

Here θ_0 is the initial angle, and c_0 is the initial sound speed at the source. With this the radius is given by

$$R_i(z) = -\frac{1}{\xi g_i(z)}. \tag{2.4}$$

When the ray is traveling from one layer to the next, the range increment is written by

$$r_{i+1} - r_i = -R_i\big(\sin\theta_{i+1} - \sin\theta_i\big), \tag{2.5}$$

which can also be written

$$r_{i+1} - r_i = \frac{1}{\xi g_i}\left[\sqrt{1-\xi^2 c^2(z_{i+1})} - \sqrt{1-\xi^2 c^2(z_i)}\right]. \tag{2.6}$$

The sum of all layers range will be the total distance for the ray. Travel time for the ray is calculated by

$$\tau_{i+1} + \tau_i = \frac{1}{|g_i|}\ln\left(\frac{c(z_{i+1})}{c(z_i)}\frac{1+\sqrt{1-\xi^2 c^2(z_i)}}{1+\sqrt{1-\xi^2 c^2(z_{i+1})}}\right). \tag{2.7}$$

Here given for one layer. The sum of all layers time will be the total travel time for the ray.

The algorithm makes repeated use of equation (2.6) and (2.7), stepping with depth increments Δz in such a way that the new depth z_{i+1} is given by the old depth z_i as

$$z_{i+1} = z_i \pm \Delta z \tag{2.8}$$

The plus sign indicates a ray going downwards and the minus sign, a ray going down upwards. Evidently the sign has to change when the ray strikes the bottom and the surface, and when the ray goes trough a turning point.

The acoustic intensity is calculated by using the principle that the power within a space limited by a pair of rays with initial angular separation of $d\theta_0$ centered on the initial angle θ_0 will remain between the two rays, regardless of the rays' paths. The acoustic intensity as function of horizontal range, $I(r)$ is according to this principle given by

$$\begin{aligned} I(r) &= I_0\frac{r_0^2}{r}\frac{\cos\theta_0}{\sin\theta}\left|\frac{d\theta_0}{dr}\right| \\ &= I_0\left(\frac{r_0^2}{r}\right)\left(\frac{c_0}{c}\right)\frac{\cos\theta}{\sin\theta}\left|\frac{d\theta_0}{dr}\right|. \end{aligned} \tag{2.9}$$

When the water depth varies with distance the ray parameter is no longer constant, but changes with the bottom inclination angle. An incoming ray with angle θ_{in} is reflected to the angle θ_{ref} when the bottom angle is α.

$$\theta_{ref} = \theta_{in} + 2\alpha \tag{2.10}$$

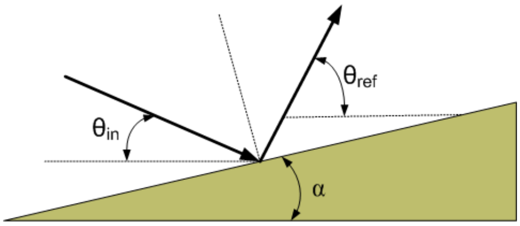


FIGURE 2.3: When the ray hits the bottom, outgoing angle will be different from the incoming one, in accordance to Snell's law.

where θ_{ref} is the outgoing angle and θ_{in} is the incoming angle. Seen in figure 2.3 Consequently the ray parameter has to change to

$$\begin{aligned}\xi_{ref} &= \frac{\cos(\theta_{ref})}{c} = \frac{\cos(\theta_{in} + 2\alpha)}{c} \\ &= \xi_{in}\cos(2\alpha) - \frac{\sqrt{1-\xi_{ref}^2 c^2}}{c}\sin(2\alpha).\end{aligned} \tag{2.11}$$

Beam displacement is also implemented in the model as an option. When the incident grazing angle is lower than the critical angle the ray appears to be displaced a certain distance Δl along the interface. The beam displacement is

$$\Delta l = \frac{\partial \delta}{\partial k} = 2k\frac{\rho_0\rho(\gamma_0^2+\gamma_1^2)}{\gamma_0\gamma_1(\rho_1^2\gamma_0^2+\rho_0^2\gamma_1^2)}. \tag{2.12}$$

Where δ is the phase angle of the reflection coefficient for the interface between the water (0) and the bottom (1) for angles lower than the critical angle

$$\delta = 2tan\left(-i\frac{\rho_0\gamma_1}{\rho_1\gamma_0}\right). \tag{2.13}$$

In equation (2.12) and (2.13), k is the horizontal wave number, γ_0 γ_1 are the vertical wave numbers and ρ_0 ρ_1 are the densities of the water and the bottom medium respectively. The beam displacement of equation (2.13) is a function of frequency and valid only for the half-space model. In the PlaneRay model the beam displacement is introduced in the initial ray tracing for one frequency specified by the user and can therefore only be used for narrow band signals.

The first step in the modeling is to apply the algorithm described above to a relative large number of rays spanning the whole range of initial angles that are relevant for the actual studies. For each ray the trajectory, travel time and the transmission loss are calculated and stored in the computer disk together with the ray history in terms of number, angle and locations of bottom and surface reflection and turning point. Since the sound speed profile and the bathymetry are supposed to be fixed and not changed, this ray tracing calculation is only done once for each site.

2.1.2 Sorting and interpolation

The next step is to determine the eigenrays and their trajectories and the approach used in PlaneRay is based on interpolation on the results of the initial ray tracing. However, the

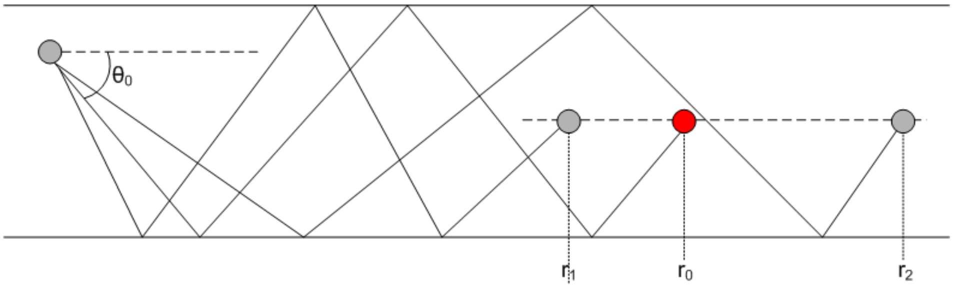


FIGURE 2.4: The eigenrays to receiver at range r_0 is found by interpolating between the two rays arriving at the same receiver depth at range r_1 and r_2.

TABLE 2.1: Dimensions for creating correct printing area.

Class	Bottom hits	Surface hits	Ray
Class 0	0	0	Direct ray
Class 1	n-1	n	Positive and negative start angle
Class 2	n	n	Positive start angle
Class 3	n	n	Negative start angle
Class 4	n	n-1	Positive and negative start angle

interpolation has to be done on rays that have same type of ray history. This consideration is illustrated in figure 2.4. The figure shows three ray paths from the source to reach three receivers at the same depth but different range. All the three rays have one reflection from surface and two reflections from the bottom. The two rays intersecting the receiver depth at range r_1 and r_2 are the two rays from the initial ray tracing and the desired ray is the one with start angle θ_0 reaching the target at range r_0. Notice that all the rays have the same number of reflections from the surface (one), and the bottom (two). Therefore the relation between initial angle θ_0 and receiver range can be expected to follow a reasonable smooth curve amendable to interpolation. In the case of a constant sound speed there will be 5 classes of arrivals and these are shown in Table 2.1. With a depth dependent $c(z)$ there will be additional classes in order to include upper and lower turning point. The user is required to supply a selection table of classes to be included in the synthesis of the complete time and frequency response.

2.1.3 Synthesis of the sound field

The received sound field is synthesized by coherently adding the contributions of the eigenrays. No rays are traced into the bottom and a layered bottom is described entirely by plane ray reflection coefficients. The reflection coefficient between the water and the sediment layer, r_{01} is given as

$$r_{01} = \frac{Z_{p1} - Z_{p0}}{Z_{p1} + Z_{p0}}, \tag{2.14}$$

and r_{12} is the reflection coefficient between the sediment layer and the solid half space,

$$r_{12} = \frac{Z_{p1} cos^2 2\theta_{s2} + Z_{s2} sin^2 2\theta_{s2} - Z_{p2}}{Z_{p1} cos^2 2\theta_{s2} + Z_{s2} sin^2 2\theta_{s2} + Z_{p2}}. \tag{2.15}$$

In equation 2.14 and 2.15 Z_{ki} is the acoustic impedance for the compressional (k=p) and shear (k=s) waves in water column (i=0), sediment layer (i=1) and solid half-space (i=2), respectively. θ_{s2} is the transmitted grazing angle for the shear wave in the solid half-space.

Chapter 3

GPU Theory

3.1 GPU

GPU stands for **G**raphics **P**rosessing **U**nit, and is a dedicated graphics rendering device for a personal computer or game console. Modern GPUs are very efficient at manipulating and displaying computer graphics, and their highly-parallel structure makes them more effective than typical CPUs for a range of complex algorithms. A GPU implements a number of graphics primitive operations in a way that makes them render much faster than drawing directly to the screen with the CPU.

3.1.1 History

Modern GPUs are descended from the monolithic graphic chips of the late 1970s and 1980s. In the late 1980s and early 1990s, high-speed, general-purpose microprocessors became popular for implementing high-end GPUs [13]. Several (very expensive) graphics boards for PCs and computer workstations used digital signal processor chips to implement fast drawing functions, and many laser printers where shipped with a post script raster image processor. By the early 1990s, the rise of Microsoft Windows sparked a surge of interest in high-speed, high-resolution 2D bitmapped graphics (which had previously been the domain of Unix workstations and the Apple Macintosh). For the PC market, the dominance of Windows meant PC graphics vendors could now focus development effort on a single programming interface, GDI.

Throughout the 1990s, 2D Graphics User Interface (GUI) acceleration continued to evolve. As manufacturing capabilities improved, so did the level of integration of graphics chips. Video acceleration became popular as standards when VCD and DVD arrived, and the Internet grew in popularity and speed. In the mid-1990s, computer CPUs were becoming powerful enough to handle real-time 3D graphics. Graphics chip manufacturers scrambled to be the first to offer hardware 3D acceleration to their product line-ups. However, as manufacturing technology again progressed, video, 2D GUI acceleration, and 3D functionality were all integrated into one chip. Out of this the two major graphics vendors nVidia and ATI grew up.

With the advent of APIs like DirectX 8.0 and OpenGL, GPUs added programmable shading to their capabilities. Each pixel could now be processed by a short program that could include additional image textures as inputs, and each geometric vertex could likewise be processed by a short program before it was projected onto the screen. nVidia also held the crown for being the first to market with a chip capable of programmable shading, the GeForce 3 (NV20).

Modern GPUs use most of their transistors to do calculations related to 3D computer graphics. They were initially used to accelerate the memory-intensive work of texture mapping and rendering polygons, later adding units to accelerate geometric calculations such as translating vertices into different coordinate systems.

Recent developments in GPUs include support for programmable shaders which can manipulate vertices and textures with many of the same operations supported by CPUs, oversampling and interpolation techniques to reduce aliasing, and very high-precision color spaces. Because most of these computations involve matrix and vector operations, engineers and scientists have increasingly studied the use of GPUs for non-graphical calculations. In figure 3.1 the increase in computational power for last four years are shown. As can be seen there are enormous floating point power in the GPU chip compared to a CPU. Because all these applications exceed an actual GPU's usage target, a new term, General Purpose Graphics Processing Unit (GPGPU) is usually employed to describe them. While GPGPUs are the same chips as GPUs, there is increased pressure on manufacturers from ”GPGPU users” to improve hardware design, usually focusing on adding more flexibility to the programming model.

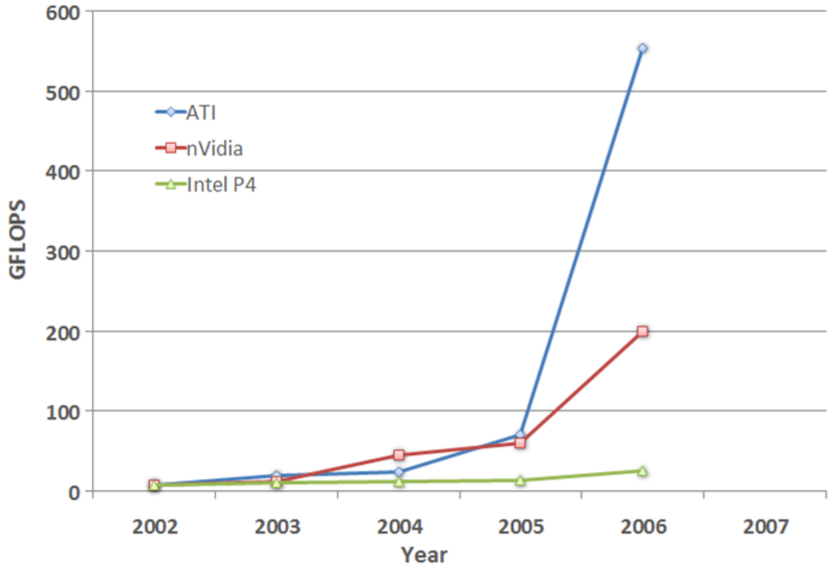


FIGURE 3.1: In the past years the GPU has outgrown the CPU when it comes to floating point performance. With the two manufactures ATI and nVidia constantly competing, the performance is rapidly growing.

3.2 GPU pipeline

The GPU uses a pipeline architecture to process multiple fragments in parallel. This means that it can do a lot more operations at same time, compared to the CPU. In figure 3.2 the GPU pipeline are shown.

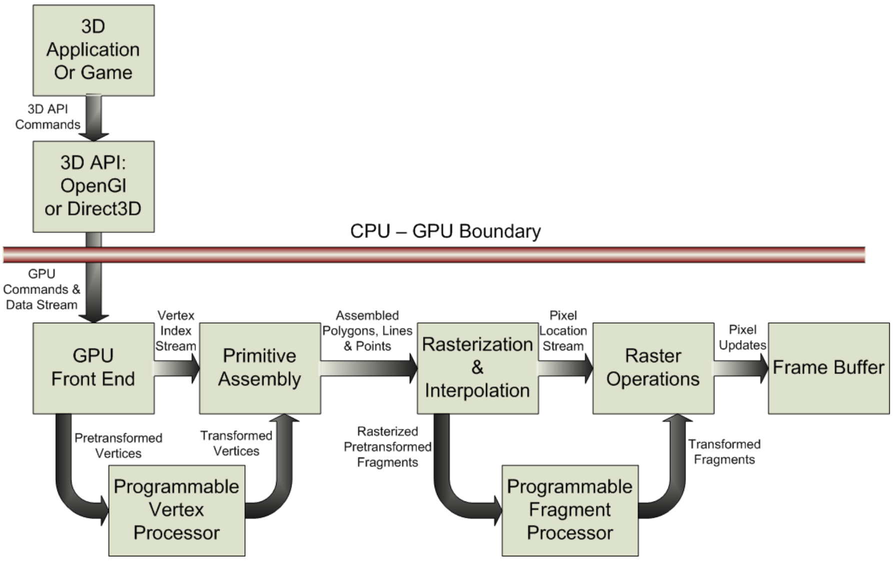


FIGURE 3.2: The Programmable Graphics Pipeline acts as a stream computer. On every element in the stream, the fragment processor computes in accordance with instructions given in the shader file.

3.2.1 Processing stages

In principal concern the GPU is split up in two processing stage, vertex and fragment, in the pipeline. For each stage a program containing instruction will be applied, called shader. The shader in the first stage will transform the initial data to a primitive, which is than rasterized before sent to the next stage. Here all pixel are computed in accordance to instructions given in the shader file, like color and depth. This is shown in figure 3.3. The final result are sent to the frame buffer for visualizing on the screen.

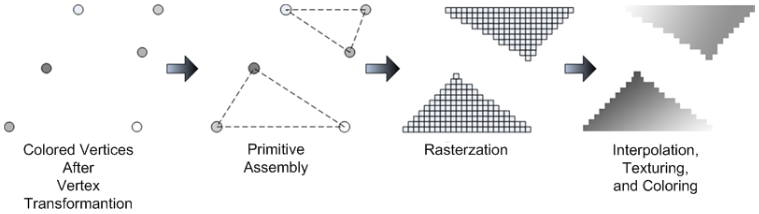

Colored Vertices After Vertex Transformantion
Primitive Assembly
Rasterzation
Interpolation, Texturing, and Coloring

FIGURE 3.3: When initial data are sent trough the pipeline, they are transformed to a visible object sent to the frame buffer. Stages consists of making pixels of the primitives before coloring.

Vertex Processor

Vertex transformation is the first processing stage in the graphics hardware pipeline. The transformation will preform a sequence of math operations on each vertex, like texture coordinate generation and transformation, lighting and color material applications [18]. Programs that are intended to run on this processor are called vertex shaders or vertex programs. Vertex shaders can be used to specify a completely general sequence of operations to be applied to each vertex. On the other hand, they can not perform graphics operations that require knowledge of other vertices at a time or that require topological knowledge, e.g. perspective division, back face culling and depth range. The processor operates on one vertex at a time, and continue inside the Vertex Program Loop until it terminates. Its design is focused on the functionality needed to transform and light a single vertex. The output of the vertex processor is sent through subsequent stages of processing before the fragment processor performs its operations. The vertex processor writes to memory through multiple stages, as can been seen in figure 3.2

Fragment Processor

The fragment processor operates on fragment values and their associated data, and is intended to perform operations on inputs from the rasterization stage. This includes operations on interpolated values, texture access and texture application, fog, color sum and point size [18]. Programs that run on this processor are called fragment shaders or fragment programs. Fragment shaders can be used to specify a completely general sequence of per-fragment operations to be applied to each fragment passing through the processor. Every fragment is invisible to all the others, so the fragment shaders can not perform graphics operations that require knowledge of other fragments. This programmable unit can only write to the frame buffer. It does not have read capability. However, it does have the capability of texture lookup. In GPGPU the fragment processor has to be exploited, because it writes directly to the frame buffer and has the most processor within. 48 in the newest from manufacture ATI[1].

3.3 Graphic Programming Languages

Efficient programming the GPU is essential to take advantage of the calculation speed. This is done with programming language like C/C++ and GLSL. Also librarys like OpenGL, Boost and Shallows can be used.

3.3.1 OpenGL and Direct3D

OpenGL stands for **Open G**raphics **L**ibary, and is one of two APIs for GPUs [1]. Microsoft's Direct 3D is the second one, and is designed explicitly for Windows platform. However this makes it especially popular among game developers. OpenGL is a cross platform interface, and is compatible with operating systems like Linux, Unix and Windows, The use is widely in CAD, virtual reality, scientific visualization and video game development. Both APIs uses C/C++ language for implementation.

3.3.2 BrookGPU

Brook for GPUs is a compiler and runtime implementation of the Brook stream program language for modern graphics hardware, and is a research project in Stanford University Graphics lab in USA. The idea is to develop a programming language which is not meant for graphic tasks, but rather for general purpose programming on the GPU. It utilizes the parallel processing possibility as one important parameter to optimize the code. Another name for it is C for streams because GPU is a streaming processor. BrookGPU can be found at [17].

[1] ATI X1900XTX

3.3.3 Shading Languages

GLSL

OpenGL Shading Language (GLSL) is a high level shading language based on the C programming language, and developed by OpenGL ARB as an extension to OpenGL. It is not operating system specific, and works on Windows, Linux and Unix. The OpenGL Shading Language specification defines 22 basic data types, some are the same as used in the C programming language, while others are specific to graphics processing [3].

Cg

C for Graphics (Cg), C for graphics is made for programming GPUs by nVidia. It has the ability to compile a written code to be optimized for the GPU. Cg is based on C and much of the syntax is alike. The idea behind Cg is to make the programmer focus on the idea, not on the hardware implementation [18]. Cg programs are portable by the fact that they can run on every operating systems like GLSL.

HLSL

The High Level Shader Language (HLSL) is a shader language developed by Microsoft for use with Direct3D, and is very similar to Cg. HLSL allows expensive graphical computations to be done on the graphics card, thus freeing up the CPU for other purposes. However it runs only on Windows operating system [18].

3.3.4 GPGPU library

Shallows

Shallows is a GPGPU programming library developed by SINTEF ICT Applied Mathematics in Oslo and are a C++ library designed to make GPGPU programming easier and safer. The aim is to reduce the time spent on writing and debugging OpenGL related C/C++ code, so the developers of GPGPU applications can concentrate on implementing the algorithms instead. Shallows can be found at [15].

3.4 Heat Equation, an Example

The heat equation is a partial differential equation (PDE), and it show how the graphics pipeline can exploit its power in a GPGPU application. The 2D heat equation can be written

$$\frac{\partial^2 u}{\partial t^2} = \nabla^2 u. \tag{3.1}$$

To numerically solve such equations, one need to discretize the equation use approximations to derivates. When using a forward difference scheme, the equation reduces to

$$u_{i,j}^{n+1} = 2u_{i,j}^{n} - u_{i,j}^{n-1} + \frac{k}{h^2}\left(u_{i+1,j}^{n} + u_{i-1,j}^{n}u_{i,j+1}^{n} + u_{i,j-1}^{1} + 4u_{i,j}^{n}\right). \tag{3.2}$$

For the 2D heat equation this typically boils down to a grid point at time step t+1 being a weighted sum of it's 4 neighbors and itself at time step t. Here, we let a pixel correspond to

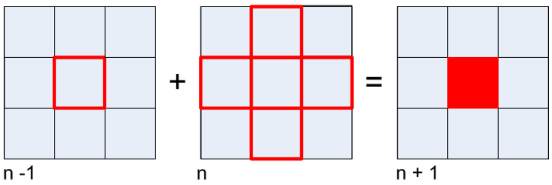


Figure 3.4: The solution of the heat equation is a weighted sum of it's 4 neighbors and it self at time step t

a grid point, and the neighboring pixels are thus the neighboring grid points. This is shown in figure 3.4.

The code in listing 3.1 is a program written in GLSL and solves the heat equation using equation 3.2. For every loop the program runs the values for the neighboring pixels are read. This five values are summed together, and the result are sent to the frame buffer, which is a render target. The render target then stores the values in the texture array, for use as input to next calculating. This is known as ping-ponging. As seen in listing 3.1 the texture containing the values are searched from using *sampler2D*. For visualizing the result, a specific shader is used. See listing 3.2. Here the values are only passed trough and plotted on the screen. To get the right position on the screen the *ModelViewProjectionMatrix* are used. The

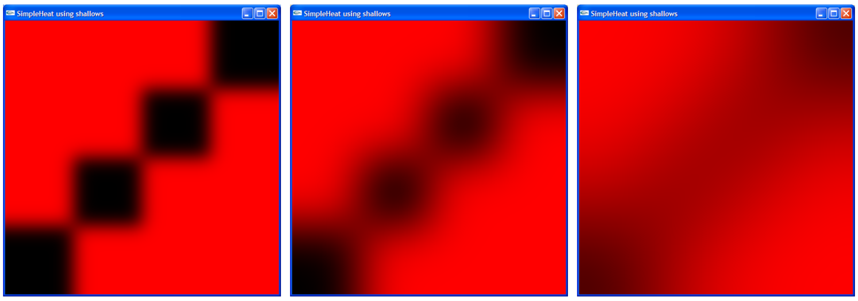

SimpleHeat using shallows
SimpleHeat using shallows
SimpleHeat using shallows

Figure 3.5: The heat equation shows a melting of the colors as a result of the pixels are summed together

final result are a melting of the red and black color. Shown in figure 3.5.

Listing 3.1: When calculating the heat equation the color value for the neighboring pixels are added together, and the color values are written to the frame buffer.

```
[Vertex shader]

void main(void)
{
        texCoord = gl_MultiTexCoord0;
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

[Fragment shader]

uniform sampler2D texture;
uniform float h;

void main ()
{
  const float r = 0.23; // Stability criterion. r = eps*dT/dX*dY

  // Read the neighboring texture values
  vec4 u =    texture2D(texture, gl_TexCoord[0].xy);
  vec4 u_pj = texture2D(texture, gl_TexCoord[0].xy + vec2(h,0.0));
  vec4 u_mj = texture2D(texture, gl_TexCoord[0].xy - vec2(h,0.0));
  vec4 u_ip = texture2D(texture, gl_TexCoord[0].xy + vec2(0.0,h));
  vec4 u_im = texture2D(texture, gl_TexCoord[0].xy - vec2(0.0,h));

  // Sum them together
  vec4 result = (1.0-4.0*r)*u + r*(u_pj + u_mj + u_ip + u_im);

  // Output the result to the framebuffer
  gl_FragColor = result;
}
```

Listing 3.2: Visualizing the result is a simply texture lookup, after placing the pixel in space.

```
[Vertex shader]

void main ()
{
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  gl_TexCoord[0]=gl_MultiTexCoord0;
}

[Fragment shader]

uniform sampler2D texture;

void main ()
{
  vec4 col=texture2D(texture, gl_TexCoord[0].xy);
  gl_FragColor = col;
}
```

Chapter 4

GPGPU Program

The GPU is a streaming processor, and for every element in the stream the same calculations are done. The stream is ran over and over again, and it continues until the user interferes or other data is set as input. Calculating a sound field with PlaneRay consist of many iterations of same mathematics, however the input values change for every iteration. Setting the initial data as start values for the stream, the shader will calculate the ray path. If one ray is taught of as one pixel, the pixel gird size will give the number of rays. Doing this on the stream once, one frame is generated. Usually the graphics pipeline makes frames for visualizing moving objects. Using this characteristic property to the GPGPU program, we will get the rays paths. From figure 4.1 the frames are created as the time goes by. Number of frames that are created are determined by the pixel grid size. With a large amount of pixels the frame rate will become low due to more calculations within each frame.

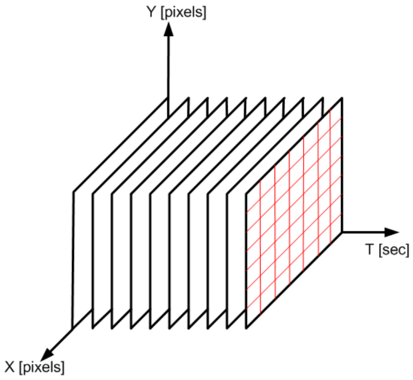


Figure 4.1: For every loop the program does, a frame consisting of pixel is created with distinct values for every pixels. When the program creates 10 frames per second with a pixel grid of 7x7, 490 different calculation are done.

Each frame is stored in the frame buffer for temporarily storage, so the values can be visualized and set as input for next iteration. Looping of values for use in next iteration is called ping-ponging and is common in GPGPU applications . How the program works in one loop are shown in figure 4.2

The frame buffer is the GPU's memory, and usually this memory is overwritten for every loop for visualization purposes. When storing the wanted values they must be written to the CPU for permanent storage, since the GPU overwrites the memory for every iteration. This

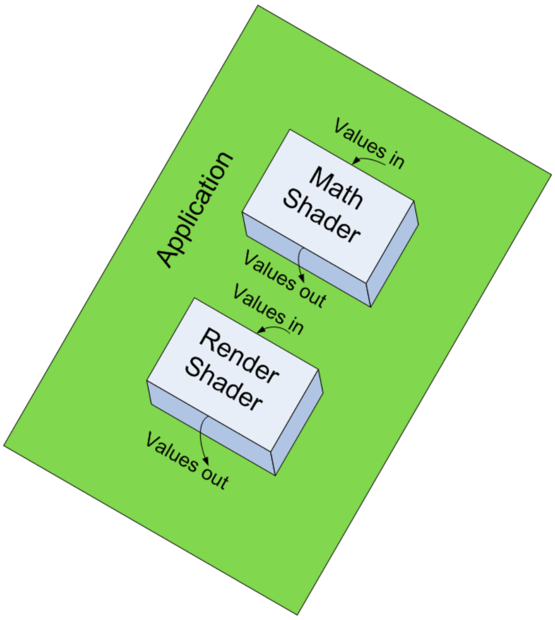


Figure 4.2: Two shaders are used to calculate and visualize the sound field. Initial values are stored in an array for texture lookup in the math shader. The result are visualized to screen and set as input to next program loop.

read back will slow down the overall calculation speed, because the program must write the calculated value to the CPU before proceeding. However using asynchronous read back this bottleneck can be minimized.

4.1 Initial Settings

Initial setting for the GPU pipeline is the first main thing to do. Here all the setting for keyboard, window size and view space are done. This is the same as setting up a regular GPU application. As can be seen from Listing 4.1 the display function are empty. It is here the calculations are taking place. Setting up the initial setting with GLUT, which is a utility library for OpenGL, is an easy way to go. The code in Listing 4.1 uses this library. Using other librarys like Qt and WIN32 the command window will not be shown, only the main window. However in this application GLUT is sufficient.

Further on the sound speed profile needs to be implemented. This is done by reading a text file containing any kind of sound speed profiles into an array on the CPU. Getting this data to the GPU is done by copying the array to a texture on the GPU. By the help of Shallows this is done in one sentence. A selection of five different sound speed profiles for the program are shown in figure 4.3.

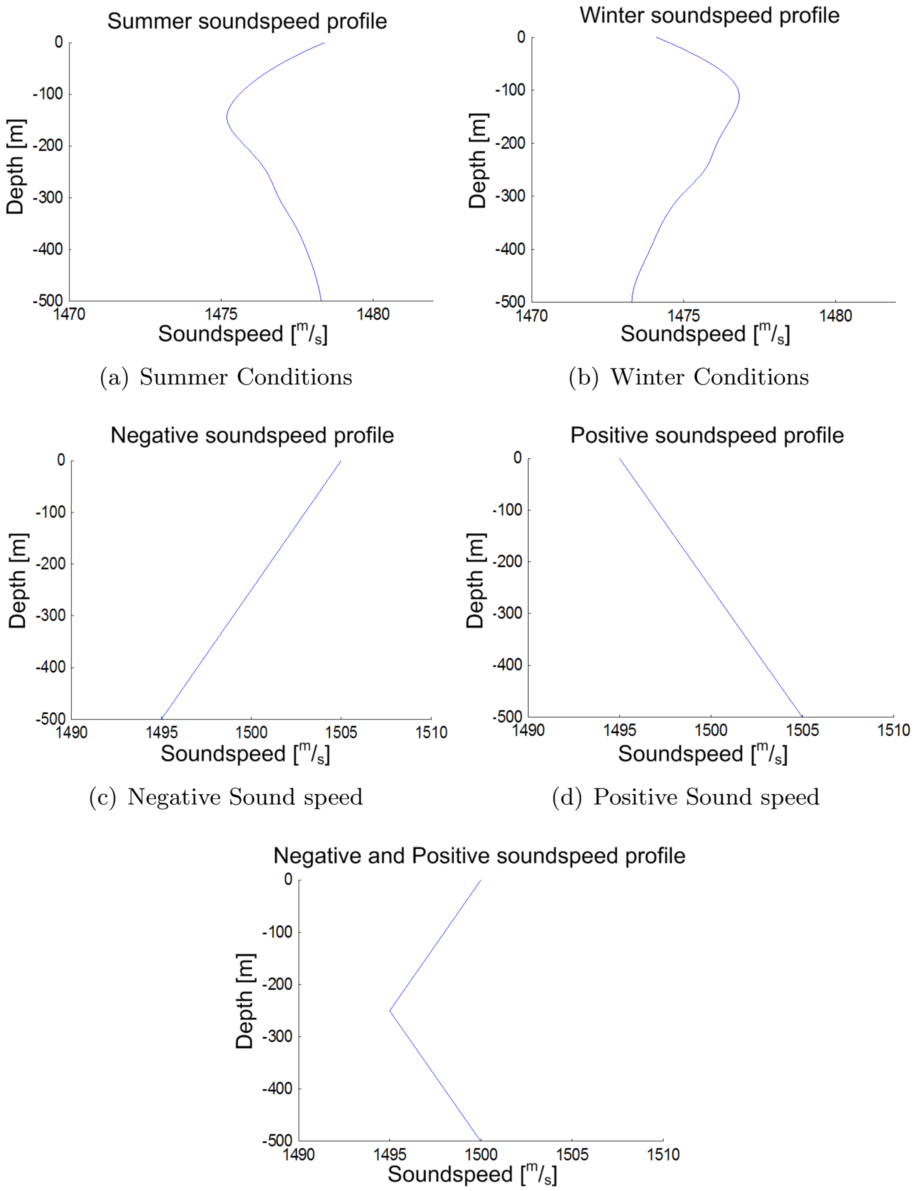
For layers other than one meter an interpolation routine using

$$\Delta = \max[c(z_i), c(z_{i+1})] - \min[c(z_i), c(z_{i+1})] \tag{4.1}$$

is ran, where σ is the absolute difference between two layers. This value are split on wanted

Listing 4.1: By setting the initial settings with GLUT, the code is much less than with e.g Qt or MFC.

```
#include <iostream>
#include <GL/glut.h>
// These two globals store the current window width and height.
int window_width;
int window_height;

/* This function is called whenever a key is pressed, for this application
 * the only possibility is to quit. */
void keyboard(unsigned char key, int winx, int winy){
  switch(key) {
    case 'q':
      exit( EXIT_SUCCESS );}}

/* This function loads identity OpenGL matrices, and only adjusts
   the viewport. Since we will end up with different sizes of the
   computational grid and visualization window is is nice to to
   separate the callback and the function doing the main work. */
void reshape(int width, int height){
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glViewport(0,0,width,height);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();}

/* This will be registered as the reshape callback. */
void reshape_callback(int width, int height){
   window_width=width;
   window_height=height;
   reshape(window_width, window_height);}

/* Our main display loop, empty as of yet. */
void display() {};

/* Initialize glut, create a window and start rendering. */
int main(int argc, char** argv){
   glutInit(&argc, argv);
   glutInitWindowPosition(100, 100);
   glutInitWindowSize(512, 512);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
   int windowID = glutCreateWindow("SimpleHeat using shallows");
   glutSetWindow(windowID);
   glutReshapeFunc(reshape_callback);
   glutKeyboardFunc(keyboard);
   glutDisplayFunc(display);
   glutMainLoop();}
```



(a) Summer Conditions

(b) Winter Conditions

(c) Negative Sound speed

(d) Positive Sound speed

(e) Positive and Negative Sound speed

FIGURE 4.3: Summer, Winter, Negative, Positive and Positive/Negative sound speed profiles

number of layers

$$\eta = \frac{\Delta}{numberoflayers}, \tag{4.2}$$

and the new values are added to the lowest of the two sound speed values. The interpolation routine used is linear, and only used when more values of the sound speed profile than the initial 512 are needed. With this the interpolation routine needs only 512 values as input for all layer thickness. Other initial settings like source depth, bottom depth, receiver depth, bottom contour and start angle are set in the source code and stored in an array for later to be transfered to texture on the GPU.

The GPU needs somewhere to store the data after calculation, and usually this is done in a render target. A render target is generic name for data storage on the GPU, such as a frame buffer or texture. Since the calculation needs to be visualized, a on- and off-screen frame buffer is needed. This two are set in the beginning of the program, along with four render targets for swapping data in and out of the shader.

GPU does all calculations in parallel, because it usually generates motion frames. A GPGPU application can take advantage of this by using a grid made of out of pixels to execute a number of calculation. From figure 4.1 a 7x7 calculation grid is shown, which in this case will result in 49 rays, since each pixel is taught of as one ray. Each pixel computes in accordance to instructions written in a shader file. Which the instructions are the same for all pixels, however the values are different.

4.2 Shaders

As mentioned the shader is a small program within the main program which does the mathematics calculations and rendering. Here shaders has been written in GLSL, in preference to Cg and HLSL. As in Listing 4.2 of the mathematics shader program, the shader contains instructions for both the vertex and fragment part in the pipeline. In this code the vertices are only passed trough to the fragment processor. By writing *uniform sampler2D texSetValues* the shader knows it shall use values within this texture when calculating. In the texture lies values to be used in calculation in the form of a four-component vector. Meaning four different values can be sent to each pixel from one texture. Current program uses two four-component vectors in the ping-ponging, meaning eight values for each pixel are written to the earlier set off-screen frambuffer after each calculation. When finding the values belonging to each pixel the *texture2D* command is used. Here the first four values in the texture are set to pixel number one, next four values to pixel number two, and so on. When it comes to the sound speed profile the values are not changed when calculating, however the texture are used only as a look up texture. Instructions used are the formulas 2.1 till 2.7 and 2.10 from chapter two. The results of calculation are stored in two four-component vectors using swizzle operators. In the bottom of Listings 4.2 this four-component vectors are written to the frame buffer for storage.

4.3 GPGPU work flow

For making the shader files and initial settings working together a work flow is needed. In Listings 4.3 the work flow starts with creating textures from arrays, which shallows does with

Listing 4.2: For each pixel in the grid, this program code is applied. For every iteration the program fetches the values for calculation from *texSetValues and texInit2*. This values are the result from last calculation. The other textures are only lookup textures, and remains the same trough out the simulation.

```
[Vertex shader]
varying vec4 texCoord;
void main(void){
  texCoord = gl_MultiTexCoord0;
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;}

[Fragment shader]
varying vec4 texCoord;
uniform sampler1D texSSPCurr;
uniform sampler2D texSetValues;
uniform sampler2D texInit;
uniform sampler2D texInit2;

void main(void){
  vec4 texStart = texture2D(texInit, texCoord.xy);
  vec4 texCurr  = texture2D(texSetValues, texCoord.xy);
  vec4 spCurr   = texture1D(texSSPCurr, texCurr.x/(512));
  vec4 texCurr2 = texture2D(texInit2, texCoord.xy);
  ............
  ''Formulas from chapter 2''
  ............
  vec4 result;
  result.x = texCurr.x + texCurr.z;
  result.y = texCurr.y + rangeinc;
  result.z = texCurr.z;
  result.w = texCurr.w + traveltime;

  vec4 result2;
  result2.x = rayparam;
  result2.y = Bottom.x;
  result2.z = Bottom.y;
  result2.w = spCurr.x;

  //The results are sent to frame buffer
  gl_FragData[0] = result;
  gl_FragData[1] = result2;}
```

a simple command. Using boost shared pointer gives the program an easy control over the texture, which in this case is a 4 component vector. The newly created textures now contains the initial values earlier set in the program. This initial values are the earlier mentioned input to the shader.

The results are stored in a given frame buffer, here called offs_fb. For visualization another frame buffer is used, here called ons_fb. Which shader to use is given by a simple command. Reshaping the calculation grid must be done so the wanted number of rays are calculated, done with the reshape function. At the end of Listing 4.3 the output render target are swapped with an other render target so the calculated values are set as input to next iteration.

Visualizing the result, the other shader pass the calculated values to the screen, along with a visualizing of the bottom and a time- and frame-counter. This will show how fast the program calculates. From figure 4.1, 49 rays are calculated, done as one frame. When the program makes 350 frames per second (fps), $49x350 = 17150$ different ray calculations are done in one second.

At the end all results in the frame buffer must be stored in an array on the CPU. The reason for storing this at the CPU is because the frame buffer in GPU is overwritten for every calculation. When not clearing this, the visualization of the results will be the rays path on the screen. However moving the window will create the path to be wiped out by other random data. So for safe storage the results needs to be sent to the CPU. Doing this by not interfering with the speed of the calculation, asynchronous read back must be used. This means that the read back of data is done simultaneously while the calculations executes. The values from the read back process are stored in a dimensionless vector, with range values for bottom, surface and receiver when ray hits the respectively place, and at the end written to text file. Also some additional information is then added in the vector like start angle, intersection angle and travel time. All this values shall be used in later stages in the PlaneRay model.

Listing 4.3: The work flow for the GPGPU program is set in the display function. Here all the array to texture and swapping of render targets are set. Also the work flow in and out from the shader file is set. Telling it what texture to use and where to store the calculated result.

```
boost::shared_ptr<Texture1D> init_texSSPCurr = shallows::utils:
                      :createFloatTexture1D(512, 4, SSPCurr);
boost::shared_ptr<Texture1D> init_texSSPNext = shallows::utils:
                      :createFloatTexture1D(512, 4, SSPNext);
boost::shared_ptr<Texture1D> init_texBottom = shallows::utils:
                      :createFloatTexture1D(512, 4, bottom);
boost::shared_ptr<Texture2D> init_texInit = shallows::utils:
                      :createFloatTexture2D(BUFF_DIM, BUFF_DIM, 4, init);
boost::shared_ptr<Texture2D> init_texInit2 = shallows::utils:
                      :createFloatTexture2D(BUFF_DIM, BUFF_DIM, 4, init2);
boost::shared_ptr<Texture2D> init_texSetValues = shallows::utils:
                      :createFloatTexture2D(BUFF_DIM, BUFF_DIM, 4, setvalues);

/*************** Off-Screen Rendering ***********************************/
offs_fb.reset(new OffScreenBuffer(BUFF_DIM, BUFF_DIM));
rt[0] = offs_fb->createRenderTexture2D();
rt[1] = offs_fb->createRenderTexture2D();
rt[2] = offs_fb->createRenderTexture2D();
rt[3] = offs_fb->createRenderTexture2D();
curr1 = rt[0];
next1 = rt[1];
curr2 = rt[2];
next2 = rt[3];
RayShader.reset(new GLProgram);
RayShader->readFile("rayEquations.glshader");
RayShader->setFrameBuffer(offs_fb);

/************** On-Screen Rendering *************************************/
ons_fb.reset(new OnScreenBuffer);
ons_rt.reset(new OnScreenRenderTarget(GL_BACK_LEFT));
RenderShader.reset(new GLProgram);
RenderShader->useNormalizedTexCoords();
RenderShader->readFile("rayRenderer.glshader");
RenderShader->setFrameBuffer(ons_fb);
RenderShader->setOutputTarget(0, ons_rt);
RenderShader->setInputTexture("result", rt[0]->getTexture());

/*************** Initial values *****************************************/
InitProg.reset(new GLProgram);
InitProg->useNormalizedTexCoords();
InitProg->readFile("rayEquations.glshader");
InitProg->setFrameBuffer(offs_fb);
InitProg->setInputTexture("texSSPCurr", init_texSSPCurr);
InitProg->setInputTexture("texSSPNext", init_texSSPNext);
InitProg->setInputTexture("texBottom", init_texBottom);
InitProg->setInputTexture("texInit", init_texInit);
InitProg->setInputTexture("texInit2", init_texInit2);
InitProg->setInputTexture("texSetValues", init_texSetValues);

reshape(BUFF_DIM, BUFF_DIM);
InitProg->setOutputTarget(0, curr1);
InitProg->run();
InitProg->setOutputTarget(0, next1);
InitProg->run();
InitProg->setOutputTarget(1, curr2);
InitProg->run();
InitProg->setOutputTarget(1, next2);
InitProg->run();
```

Chapter 5

Results

The GPGPU program calculates a underwater sound field, with 40 km in range and up to 500 meter depth. The bottom can be set as terrain contour, which will make the rays change direction in accordance to Snell's law. All the parameters that are necessary to adjust for numerous sound fields are implemented, such as source depth, number of rays, receiver depth and start angle for every ray. However to make all this changes the source code has to be modify and compiled for every time. See appendix B for more information.

5.1 GPGPU program results

A selection of sound fields from the GPU program is shown in figure 5.1 to 5.5. In figure 5.3 to 5.5 the layer thickness are varied, respectively 1 meter, 0.5 meter and 0.1 meter. By varying the layer thickness the accuracy should increase, because there are more values in the sound speed profile to calculate from. However by doing this the overall calculation speed decreases. A trade off for accuracy and speed has to be made. When using layer thickness 0.5 meter the accuracy has not been compromising the calculation time in a larger scale. From figure 5.5 the layer thickness is 0.1 meter, which makes the computation time much higher than for figure 5.3 and 5.4. However the frame rate is almost the same.

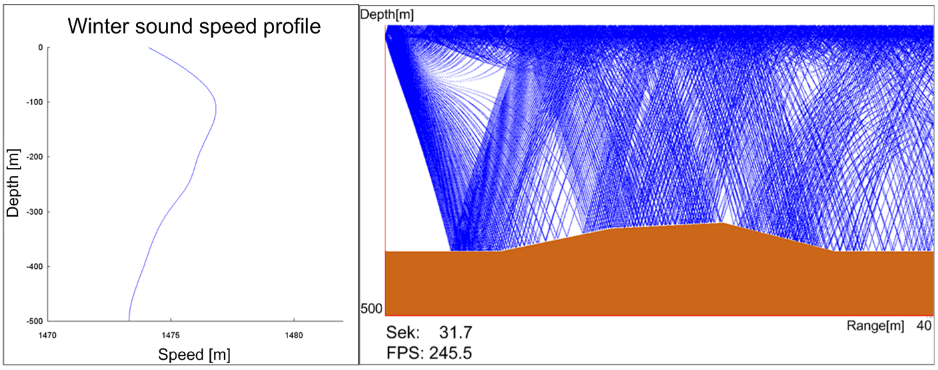


Figure 5.1: With the source at 20 meter of depth using a winter sound speed profile the rays has a concentration near the water surface. Here the layer thickness is 0.5 meter.

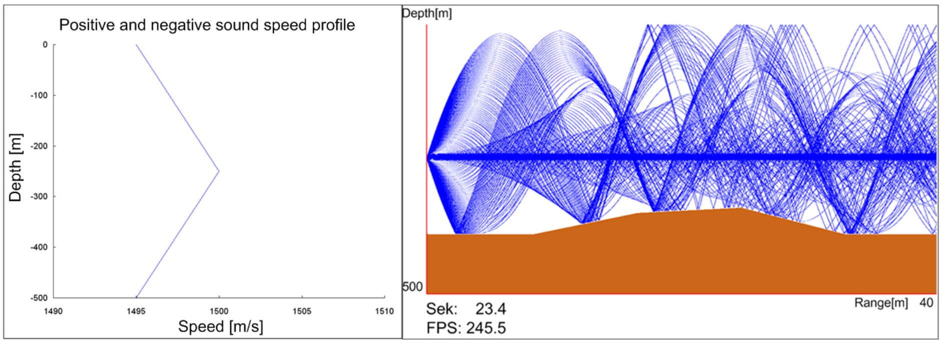


FIGURE 5.2: With the source at 256 meter of depth using a positive and negative sound speed profile we get a sound tunnel, because the source is set where the sound speed profile turns from positive to negative. Here the layer thickness is 0.5 meter.

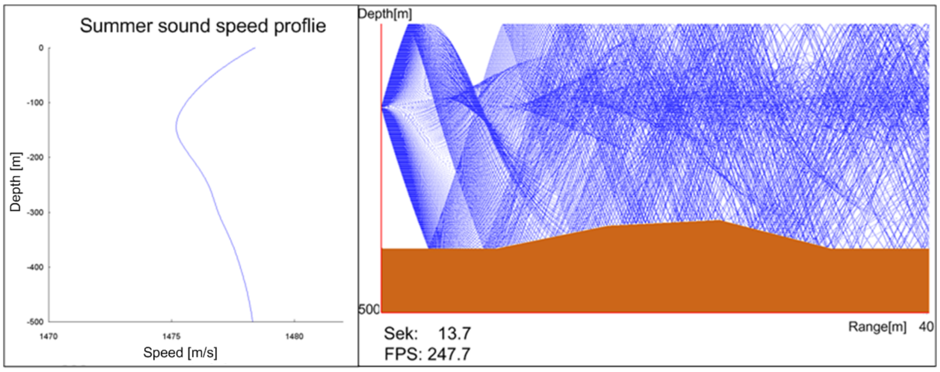


FIGURE 5.3: With the source at 150 meter of depth using a summer sound speed profile the rays has a concentration around depth of source. Since the source is set at the turning point from negative to positive sound speed, a sound tunnel will be made. Here the layer thickness is 1.0 meter.

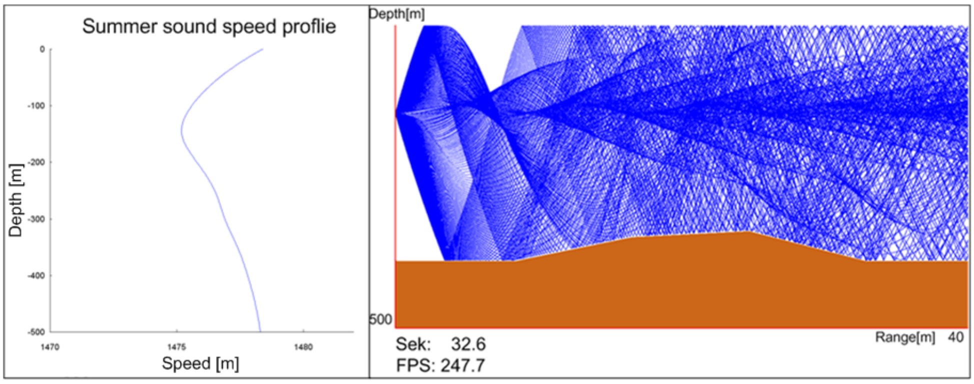


FIGURE 5.4: The sound speed profile here are same as the figure above, only the layer thickness is 0.5 meter.

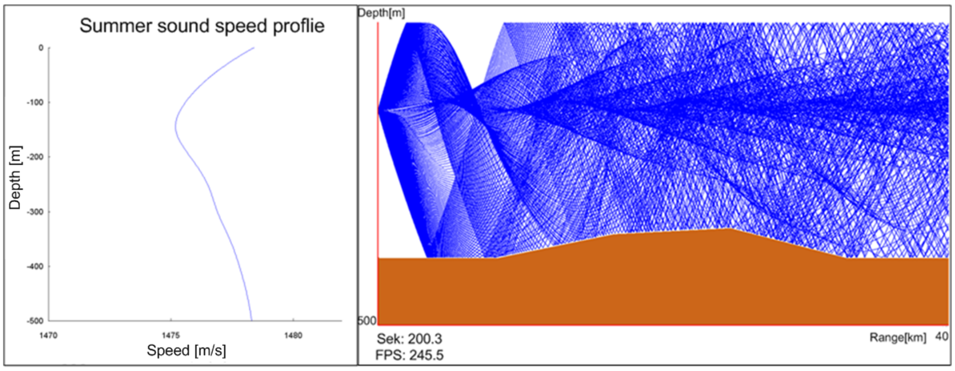


Figure 5.5: The sound speed profile here are the same as the two figures above, only the layer thickness is 0.1 meter.

5.2 Matlab comparison

From earlier studies [19] Matlab is said to be 2-4 times slower than a pure CPU code. If the already written Matlab code is optimal, we can make an assumption and say that an optimal CPU code is twice the speed of Matlab. From table 5.1 and 5.2 the improvement will than be halved.

Table 5.1: By timing the GPU application and the Matlab application the computational increment can be found. Some values are estimated by the fact that the increment for Matlab in seconds is approximated by a factor of four for twice the number of rays.

Number of rays	**FPS**	**Matlab [sec.]**	**GPU[sec.]**	**Improvement**
100	350	3.91	14.5	0.27
256	330	9.96	15.0	0.66
1024	288	40.85	17.2	2.38
2304	230	93.98	20.5	4.58
4096	184	172.83	25.7	6.73
9216	124	504.15	30.2	16.70
16384	75	1163.20	58.2	19.99
40000	34	5412.30	120.1	45.06
90000	11	_26149.70*_	253.8	_103.03_
160000	2.8	_86595.80*_	438.7	_197.40_

*Estimated values

The estimated values in table 5.1 and 5.2 are estimates for 90000 and 160000 rays in Matlab. Due to hardware limitations, this amount of rays is not possible to calculate on current hardware. The computation time is than estimated by the fact that four times the the ray, the time increases by an approximated factor of 4 in Table 5.1 for small number of rays. If this assumption is correct, the maximum gain by converting the initial ray tracing from Matlab to a GPU is 200 times by using current hardware. By this the improvement from a CPU implementation will of course be less than comparing to Matlab. A timed gain of 22 times, with the estimate of 100 times when maximum number of rays are simulated.

By the fact that this program is ran on a laptop graphics chip, the performance from a

TABLE 5.2: Saying the CPU is twice the speed of Matlab, the GPU improvement will become smaller.

Number of rays	**estimated CPU [sec.]**	**GPU[sec.]**	**Improvement**
100	1.955	14.5	0.13
256	4.98	15.0	0.33
1024	20.43	17.2	1.19
2304	46.99	20.5	2.29
4096	86.42	25.7	3.36
9216	251.08	30.2	8.32
16384	581.60	58.2	9.99
40000	2706.15	120.1	22.53
90000	*13074.85**	253.8	*51.52*
160000	*43297.90**	438.7	*98.70*

*Estimated values

desktop graphics card could theoretically be higher[1].

[1] e.g nVidia 7950GX2 or ATI X1900XTX

Chapter 6

Discussion

The GPU program is timed to be somewhat faster than Matlab, and are increasingly faster when more rays are calculated. However, when small number of rays are calculated, Matlab is faster. A reason for this can be that the GPU code is not optimized, and it also use computational effect on rendering the bottom contour.

The GPU hardware[1] used in this study support a calculation grid of 410x410, which is more than Matlab can do, due to hardware limitations. In summary the result this GPU program have in measured processing time is about 45 times the calculation speed than using Matlab. In figure 6.1 the improvement when calculating up to 40000 rays was is shown. Compared with the measured processing time using Matlab, the estimates are almost 200 times higher with maximum number of rays.

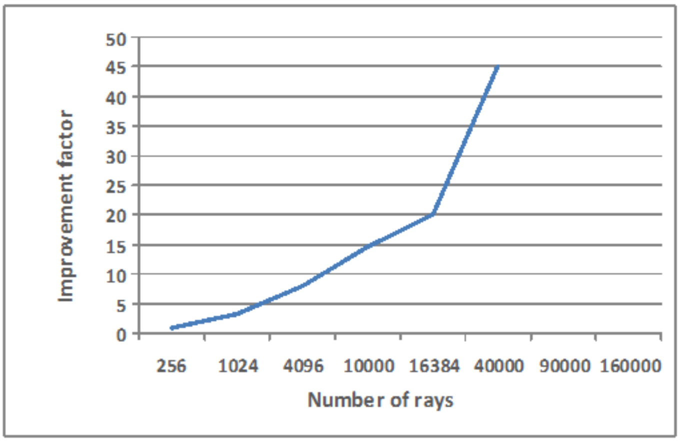


FIGURE 6.1: The GPU has increased the calculation time for high density of rays, however for small amount of rays Matlab outperform the GPU.

In practice more rays than 2000 rays are not necessary, but increasing the number of layers is more interesting. Doubling the number of layers will double time, so from table 5.1, 2304 ray will take 41 seconds to calculate.

When simulating the sound field a constant frame rate on 60 FPS was obtained for lower number of rays than 25000. The reason for constant frame rate was due to vertical synchronization was enabled [25]. Disabling this feature made a maximum frame rate of 350 frames

[1] nVidia GeForce Go 6600

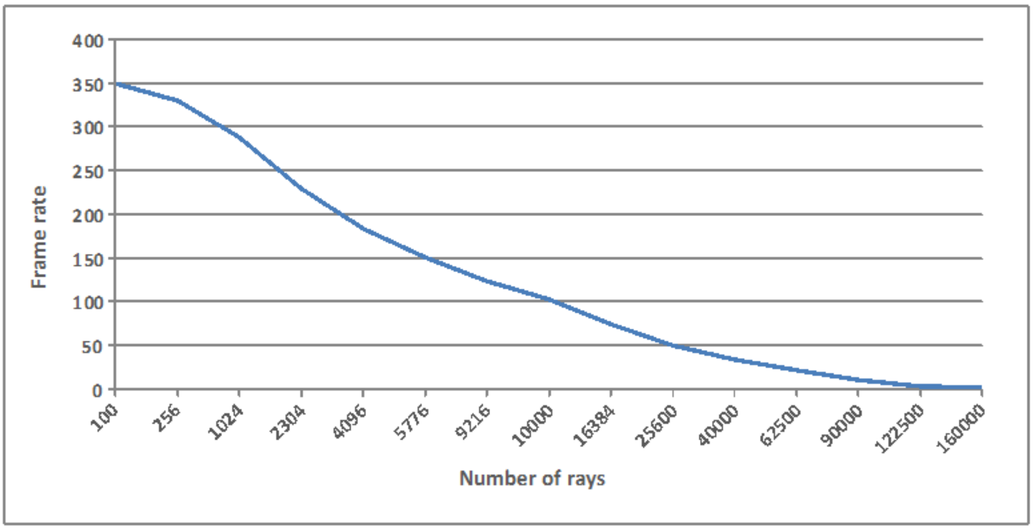


FIGURE 6.2: The GPU has a high frame rate when small number of rays are calculated, and increasingly higher with more rays.

per second when small number of rays where calculated. The frame rate for all different numbers of rays are shown in figure 6.2.

6.1 Further Work

General computation on a GPU is principally interesting because of the high floating point processing power. To make use of the full potential in computing power, this GPU code has to be optimized. Making the number of texture lookups in the shader file less is one thing to achieve high processing power. Another important factor is to use arithmetic operators instead of *if, else, for* and *while* loops in the shader file. Also the asynchronous read back earlier discussed, which is not implemented, will increase the time.

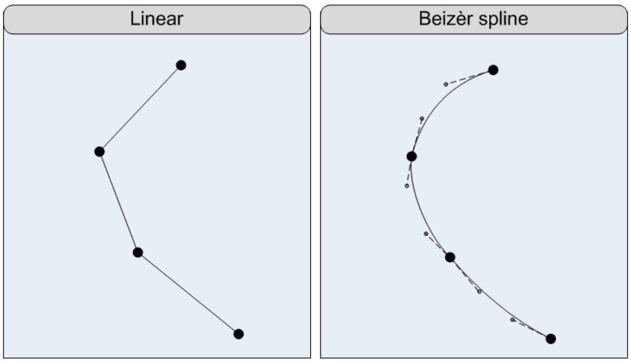
In the program code 512 values are read from a text file being the sound speed profile, for a layer thickness of 1 meter. When the layer thickness is more than 1 meter, more sound speed values are needed, so an interpolation routine between the layers are used. This interpolation routine is linear, however to obtain a more accurate result a Beizèr spline interpolation could be used. From figure 6.3 the Beizèr curve is smoother than the linear curve. Using this will give more realistic results. The Beizèr spline curve is given by

$$c(t) = \sum_{i=0}^{n} P_i B_{i,n}(t) \tag{6.1}$$

where

$$B_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i} \tag{6.2}$$

is the Bernstein polynomial. However since the sound speed profile in the PlaneRay model is assumed to be linear between two layers, this will only make a difference in current GPGPU program when a layer thickness less than 1 meter is used.



(a) Linear interpolation (b) Beizèr spline interpolation

Figure 6.3: Using Beizèr spline interpolation instead of linear interpolation the sound speed will become more accurate when the layer thickness is less than 1 meter.

Making it easier for the user to change the program parameter will make the program more surveyable for simulation. By using Qt, MFC or WIN32 a good GUI (graphics user interface) can be made.

6.1.1 GPU advantage

In a study from MIT [21] a nonlinear inversion method based on the parabolic equation (PE) method was developed and tested using simulated data. This method was proposed in 1992, and for the current supercomputer[2] at that time, it took for a given problem using this method, 1.3 hours to calculate at 250 MFLOPS. Using todays state of the art graphics chip[3] this can theoretically [14] be improved by a factor of 2200 times, calculating at 550 GFLOPS.

[2] Cray X-MP/24

[3] ATI X1900XTX

Bibliography

[1] OpenGL Architecture Review Board. *OpenGL Programming Guide, Fourth Edition*, Addison-Wesley, Boston, 2004.

[2] OpenGL Architecture Review Board. *OpenGL Reference Manual, Fourth Edition*, Addison-Wesley, Boston, 2004.

[3] Rost, R. J. *OpenGL Shading Language, First Edition*, Addison-Wesley, Boston, 2004.

[4] Edited by Pharr, M. *GPUGems2 Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley, 2005.

[5] Hovem, J. M. *Marine Acoustics, The Physics of Sound in Underwater Environment, Part I*, NTNU, Trondheim, 2005 and ARL, Texas, 2005.

[6] Hovem, J. M. *A forward model for geoacoustic inversion based on ray tracing and plane wave reflection coefficients*, NTNU, Trondheim, 2006.

[7] Medwin, H. and Clay, S. C. *Fundamentals of acoustical oceanography.* Academic press, Boston, 1998.

[8] Jensen, F. B., Kauperman, W. A., Porter, M. B. and Schmidt, H. *Computational ocean acoustics.* AIP Press, American Institute of Physics, Woodbury, New York City, 1994.

[9] Carr, N. A., Hall, J. D. and Hart, J. C. *The Ray Engine*, University of Illnois, 2002.

[10] Purcell, T. J., Buck, I., Mark, W. R. and Hanrahan, P. *Ray Tracing on Programmable Graphics Hardware*, Stanford University

[11] Wright, R. S. Jr and Sweet, M. *OpenGL SuperBible, Second Edition* Waite Group Press, Boston, 1999.

[12] Angel, E. *Interactive Computer Graphics, A Top Down Approach Using OpenGL, Fourth Edition* Addison-Wesley, Boston, 2006.

[13] Graphics prcessing unit. *http://en.wikipedia.org/wiki/Graphics_processing_unit*

[14] Floating point operations per second. *http://en.wikipedia.org/wiki/Teraflop*

[15] Shallows GPGPU library. *http://shallows.sourceforge.net/*

[16] Boost C++ library *http://boost.org/*

[17] Brooks for GPU *http://graphics.stanford.edu/projects/brookgpu/index.html*

[18] Kilgard, F. *The Cg tutorial. The definitive guide to programmable real-time graphics.* Addison-Wesley, Boston, 2003.

[19] Comparing Matlab to C/C++. *http://www.stats.uwo.ca/faculty/aim/epubs/MatrixInverseTiming/default.htm*

[20] Haugehåtveit, O., Hovem. J. M. and Hagen. T. R. *Calculation of underwater sound fields with Graphic Processing Unit (GPU)* Norwegian University of Science and Technology, Trondheim, and SINTEF ICT Applied Mathematics, Oslo, 2006.

[21] Collins, M. D., Kuperman, W. A. and Schmidt, H. *Nonlinear inversion for ocean-bottom properties* Naval Reasarch Laboratory, Massachusetts Institute of Technology, 1992.

[22] Weiskopf, D., Schafhitzel, T. and Ertl, T. *GPU-Based Nonlinear Ray Tracing*, Institute of Visualization and Interactive Systems, University of Stuttgart, Eurographics 2004.

[23] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E. and Purcell, T. J. *A Survey of General-Purpose Computation on Graphics Hardware*, The Eurographics Association, 2005.

[24] Jedrzejewski, M. and Marasek, K. *Computation of room acoustics using programmable video hardware*, Polish Japanese Institute of Information Technology, 2004.

[25] Vertical synchronization. *http://en.wikipedia.org/wiki/Vertical_sync*

[26] Haugehåtveit, O. The use of GPU in underwater sound field calculation. http://folk.ntnu.no/haugehat/. Trondheim, 2006.

Appendix A

Some explanations

Word definitions

- **GPGPU** = General Purpose Graphics Processing Unit. A concept to use the GPU for other tasks than it was intended for, such as scientific and mathematical computation.
- **vertex** = a vertex is a point in 3D space with a particular location, usually given in terms of its x, y, and z coordinates. It is one of the fundamental structures in polygonal modeling.
- **vertices** = plural of vertex, defines the edges of lines, triangles and squares.
- **fragment** = pixel.
- **pixel** = pixel is one of the many tiny dots that make up the representation of a picture in a computer's memory. The color for a pixel is composed out of red, green and blue.
- **render** = draw the primitives, in GPGPU it does the calculation.
- **texture** = array on the CPU, used to store variables.
- **ping-ponging** = storing the calculated values in one texture, and read them back to the input texture for next calculation.
- **frame buffer** = storage for data
- **off-screen frame buffer** = storage for not visualized data.
- **on-screen frame buffer** = storage for visualized data, this is what we see on the screen.
- **render target** = on- and off-screen frame buffer, and texture buffer.
- **shader** = the code within a shader is done on each pixels in the fragment processor, and there are also instructions on how the vertex processor should treat the initial data.
- **state machine** = GPU is a state machine, which means that when a condition is set, it will not change until the condition gets another value.

- **VRAM** = Video Random Access Memory which is a high-speed, high-density temporary storage for graphics memory. Build out of DRAM.

- **V-sync** = Generally video displays are refreshed sequentially and on older CRT based displays, a short delay is required between updating the lowest horizontal line of the display and returning to refresh the highest. This delay, which is preserved by more modern display equipment, gives an opportunity in computer graphics to alter the contents of a frame buffer without visible graphical errors such as partially redrawn graphics or page tearing. Must be turned off for faster calculation.

- **GUI** = Graphical User Interface. Gives the user an easier way to control the program.

Appendix B

Program usage

Since the program does not have a GUI (Graphics User Interface), every change has to be written in the source code before compiling the program. Here the start angle, source depth, layer thickness, receiver depth, sound speed profile and bottom contour can be chosen. Beginning from the top in the source code the pixel grid is first to define.

```
const unsigned int BUFF_DIM = 10;
```

The pixel grid here is 10x10 and will result in 100 rays. Next in the source code to be given by the user is the sound speed profile to be used.

```
InFile.open("SoundSpeedProfile2.dat", ios::in);
```

Here the text file *SoundSpeedProfile2.dat* is given as input which is the summer profile in chapter 4. Further the layer thickness is given when altering *deltaStep*. Here set to 1 meter. Start angle follows in *Degrees*. Here set from 5° till $-5°$. Further the depth and range for the source is given.

```
float deltaStep = 1.0f;
float Degrees = 5.0f;
float delta = (Degrees*2)/float(BUFF_DIM*BUFF_DIM);
float temp = 0.0f;
for(int i=0; i<=BUFF_DIM*BUFF_DIM; i++)
{
 setvalues[i*4] = 100.0f;       // Depth for source.
 setvalues[i*4+1] = 0.0f;       // Range for source.
 setvalues[i*4+2] = deltaStep;  // Direction of the ray
 setvalues[i*4+3] = 0.0f;       // Travel time, always starts at zero
}

for(int j=0; j<=BUFF_DIM*BUFF_DIM; j++)
{
 init[j*4] = (Degrees - temp)*Pi/180;
 if((Degrees - temp)<0.0)
  setvalues[j*4+2] = -deltaStep;
 temp += delta;
}
```

The rest of the above code part is setting initial values like start angle to coordinate with number of rays. The array *setvalues* are an input to the shader file. Below the bottom is modeled, with the parameter y being the bottom depth. All the rest is bottom contour, split up in five steps, where the user can make the bottom rise or fall.

```
float x = 0.0f;
float y = 200.0f;
float delta2 = 0.0f;
float delta3 = 0.0f;
float rangeStep = 0.0f;
float angle = 0.0f;
float norm = 1.0f;

for(int z=0; z<512; z++)
{
 bottom[z*4] = (x + rangeStep)/norm;
 bottom[z*4+1] = (y + delta2)/norm;

if (z <= 102)
{
 delta2 += 0.0;
 angle = atanf((102.0*0.0)/(39.0*102.0));
}
else if ( z <= 204 )
{
 delta2 += 0.0;
 angle = atanf((102.0*0.0)/(39.0*102.0));
}
else if ( z <= 306)
{
 delta2 += 0.0;
 angle = atanf((102.0*0.0)/(39.0*102.0));
}
else if (z <= 408)
{
 delta2 += 0.0;
 angle = atanf((102.0*0.0)/(39.0*102.0));
}
else
{
 delta2 += 0.0;
 angle = atanf((104.0*0.0)/(39.0*104.0));
}
 bottom[z*4+2] = angle;
 bottom[z*4+3] = 50.0f;
 delta3 += 0.001;
 rangeStep += 39.0;
}
```

Further down in the source code in the function *display* the frame buffer can be cleared by using

```
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

If it is preferred to change the read back length, the number 10000 below must be set at a number between 0 and 40000 since the visualized sound field is within this range. Also the receiver depth is set here.

```
int receiverDepth = 50;

for (int i=0; i < BUFF_DIM*BUFF_DIM; i++)
{
 if( readback3[i*4+1] < 10000)
 {
  if ( (readback3[i*4+2] + readback4[i*4+2]) == 0 )
 {
  //Depth for hit (bottom_label / surface_label /
    turning_below_label / turning_above_label)
  save[i].push_back(readback3[i*4]);
  //Range to hit from source (bottom_range / surface_range)
  save[i].push_back(readback3[i*4+1]);
  //Angle for bottom when hit (bottom_angle)
  save[i].push_back(init[i*4]-bottom[i*4+2]);
 }
 if( readback3[i*4] == reciverDepth )
 {
  //Reciver depth
  save[i].push_back(readback3[i*4]);
  //Range when reciver is hit N time (eig_range)
  save[i].push_back(readback3[i*4+1]);
  //Travel time for ray when N hit (eig_time)
  save[i].push_back(readback3[i*4+3]);
  //Start angle for ray N(eig_theta)
  save[i].push_back(init[i*4]);
  //Angle for ray to reciver when N hit (eig_angle)
  save[i].push_back(readback3[i*4+2]*acos(init2[i*4]*SSPCurr[reciverDepth*4]));
 }
 }
}
```

Appendix C

Development tools

Hardware

- Acer TravelMate 4652LMi
- Intel Pentium M 740 (2MB L2 cache, 1.73GHz, 533MHz FSB)
- 512MB DDR2 SDRAM
- NVIDIA GeForce Go 6600 with 64MB VRAM

Software

- Microsoft Visual Studio 2005
- Boost C/C++ Library
- OpenGL 2.0 library with GLUT
- Shallows GPGPU library

Appendix D

Using Visual Studio

Since the program is developed in Windows environment, Visual Studio is used. Setting up the developer program for GPGPU programming is done by includeing Shallows GPGPU library and Boost C++ library. Shallows relay on the Boost C++ library to work properly, so installing Boost it than the first thing to do after installing the developer program.

D.1 Boost

From the boost home page [16] the library can be downloaded. After downloading the library it needs to be compiled with the developer tool. This is done with the

```
bjam "-VC_8_0=gcc"
```

command from the command window. Further instructions can be found under 'Getting Started' in the home page. In environment variables the path to the compilers directory must be added if not present. After compiling the librays default directory is

```
C:\Boost
```

, and here the library objects are placed. A link to this must be set in the program properties in Visual Studio before compiling Shallows.

D.2 Shallows

Before starting GPGPU programming, Shallows needs to be compiled, and resulting .dll file must be placed under the

```
C:\Windows\system32
```

catalogue. The output files must be included when a GPGPU program is made. This affects the files in the

```
include
```

and

```
lib
```

catalogue.

Appendix E

Ray tracing on GPU article

Calculation of underwater sound fields with Graphics Processing Unit (GPU)

Olav Haugehåtveit[1], **Jens M. Hovem**[1], **Trond R. Hagen**[2]

haugehat@stud.ntnu.no, hovem@iet.ntnu.no, Trond.R.Hagen@sintef.no

Norwegian University of Science and Technology, Trondheim, Norway[1]

SINTEF ICT, Department of Applied Mathematics, Oslo, Norway[2]

Abstract
This paper reports the initial results of using a modern graphic board to calculate sound propagation in the ocean. A ray tracing algorithm has been implemented on a Graphics Processing Unit (GPU) and used to calculate the trajectories of a large number of rays in an ocean where the sound speed varies with depth and the bathymetry varies with horizontal range. The algorithm and the implementation are described and some examples of tracing rays out to several kilometers are presented. With the GPU implementation we have achieved a reduction in computation time of the order 100 compared with a conventional CPU implementation while retaining the same numerical accuracy.

Keywords: Programmable graphics processor, Underwater acoustics, Ray tracing

E.1 Introduction

In the last few years, modern graphics cards have developed extremely rapidly in terms of processing speed, memory size, and most significantly, programmability. So far, this development has largely been driven by the mass-market demand for faster and more realistic computer games and multi-media entertainment. But several research groups are now realizing that this graphics hardware can also be used to dramatically speed up many conventional numerical methods of importance in scientific computing. An example of scientific computation where this method may become very useful is the modeling of sound propagation in ocean.

In this paper we report the initial results of a study to use a Graphics Processing Unit (GPU) for acoustic ray tracing in the ocean. The motivation for this work was to obtain experience in programming a GPU and evaluate the implications and the gain in computation time by using programmable GPU for modeling of wave propagation phenomena.

Mathematical/numerical modeling of the acoustic propagation in the oceans is an important issue in underwater acoustics and required in many applications in order assess the performance of acoustic equipments such as echo sounders, sonar, and communications systems. In particular, fast and versatile propagation modals are required in model-based estimation of oceanographic parameters of the water and geo acoustic parameters of the sea floor where the acoustic propagation model is required to be run many times with different environmental parameters. Ray acoustics studies and ray tracing calculations are the simplest means for assessment of sound propagation in the sea. This is essentially a high-frequency approximation of the solution of the wave equation, applicable to frequencies so high that the signal wavelength is considerably smaller than the characteristic distance of variation in sound speed. According to ray acoustics, the sound follows rays that are normal to surfaces with the same phase. When generated from a point source in a medium with constant sound speed, the phase fronts form surfaces that are concentric circles, and the sound follows straight paths that spread out from the sound source. If the speed of sound is not constant, the sound rays will follow curved paths rather than straight ones. The computational technique known as ray tracing is a method used to calculate the coordinates of the sound rays emanating from the source.

The sound speed in the ocean varies with the oceanic condition, in particular with the temperature and the salinity of the waters. Diurnal and seasonal changes in these conditions may therefore have strong impact on the propagation conditions. In general, the sound speed will vary both with depth and with range, but for many applications we may neglect variations with range and only consider the depth dependence of the sound speed in addition to the effect of range dependent water depth or bathymetry. As indicated before, long computation times can often be a concern and limitation in application of ocean acoustics models, also for models based on ray tracing. Therefore, the new programmable Graphics Processing Unit (GPU) offers new possibilities in implementation that are considerably faster than a CPU based implementation, by doing the calculation in parallel. The idea of using GPU for ray tracing is not new or original, earlier studies considered tracing sound rays in rooms [24], and tracing light rays [10] [9] in visualization applications for more realistic lighting. In theses articles the sound speed is assumed constant and the rays are straight lines. We have found one article that considers so called nonlinear ray tracing [22], for tracing light rays in space with varying wave speed After first presenting the simple ray tracing algorithm used in this study, we will describe the essential features of the GPU implementation. We will then present some results of rays in an ocean with depth dependent sound speed with range dependent bathymetry and compare the computation time with a CPU implementation. Finally we will discuss and conclude.

E.2 A simple ray-tracing algorithm

In this section we will give a short description of the ray tracing algorithm used in the present work, more information of theory can be found in text books such as the book by Jensen et al. (1994) [8] or Medwin and Clay (1998) [7]. In the implementation used here, the water column is divided into a large number of layers with the same thickness Δz as shown in figure 1. Within each layer, the sound speed is approximated with a straight line so that, in the layer $z_i < z < z_i + 1$, the sound speed is taken to be

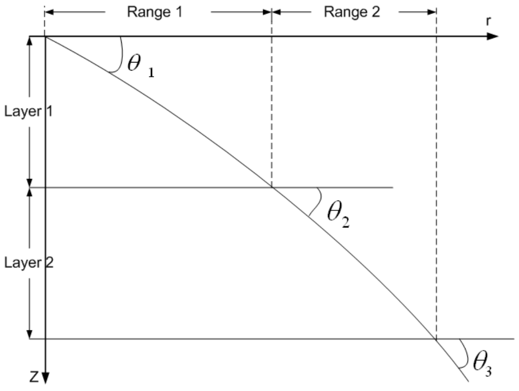


FIGURE E.1: The algorithm step with $\Delta layer$ and calculate a range increment for every step. The sum of all layers range will give the total range for the ray.

$$c(z) = c_i + g_i(z - z_i), \tag{E.1}$$

where c_i is the sound speed at depth z_i, and the sound speed gradient in the segment is g_i. Since the sound speed in each of these layers has a constant gradient, the ray in each layer follows a circular arc; the arc's radius of curvature $R_i(z)$ is given by the local sound speed gradient $g_i(z)$ and the ray parameter,

$$R_i(z) = -\frac{1}{\xi g_i(z)}. \tag{E.2}$$

The ray parameter is defined as:

$$\xi = \frac{cos\theta_s}{c(z_s)}. \tag{E.3}$$

where θ_s is the initial angle of the ray's trajectory at the source depth z_s and the sound speed is $c(z_s)$. After traveling through the layer from z_i to $z_i + 1$; the ray's range increment is

$$r_{i+1} - r_i = -R_i\big(\sin\theta_{i+1} - \sin\theta_i\big), \tag{E.4}$$

which also can also be written in the form

$$r_{i+1} - r_i = \frac{1}{\xi g_i}\left[\sqrt{1-\xi^2 c^2(z_{i+1})} - \sqrt{1-\xi^2 c^2(z_i)}\right]. \tag{E.5}$$

The local sound speed gradient is approximated by

$$g_i = \frac{c(z_{i+1}) - c(z_i)}{z_{i+1} - z_i}. \tag{E.6}$$

The travel time increment is

$$\tau_{i+1} + \tau_i = \frac{1}{|g_i|}\ln\left(\frac{c(z_{i+1})}{c(z_i)}\frac{1+\sqrt{1-\xi^2c^2(z_i)}}{1+\sqrt{1-\xi^2c^2(z_{i+1})}}\right) \tag{E.7}$$

When the water depth varies with distance the ray parameter is no longer constant, but changes with the bottom inclination angle. An incoming ray with angle θ_{in} is reflected to the angle θ_{ref} when the bottom angle is α.

$$\theta_{ref} = \theta_{in} + 2\alpha \tag{E.8}$$

Consequently, the ray parameter has to change to

$$\begin{aligned}\xi_{ref} = \frac{\cos(\theta_{ref})}{c} &= \frac{\cos(\theta_{in}+2\alpha)}{c} \\ &= \xi_{in}\cos(2\alpha) - \frac{\sqrt{1-\xi_{ref}^2c^2}}{c}\sin(2\alpha).\end{aligned} \tag{E.9}$$

The algorithm makes repeated use of equation (5) and (7), stepping with depth increments Δz in such a way that the new depth z_{i+1} is given by the old depth z_i as

$$z_{i+1} = z_i \pm \Delta z \tag{E.10}$$

The plus sign indicates a ray going downwards and the minus sign, a ray going upwards. Evidently the sign has to change when the ray strikes the bottom and the surface, and when the ray goes trough a turning point. The layer thickness, or depth increment Δz and the number of depth points N_z,

$$N_z = \frac{max(waterdepth)}{\Delta z}. \tag{E.11}$$

The algorithm presented so far gives the trajectory and travel time for a single ray with initial angle $\Delta\theta$ departing from a given source depth z_s. By tracing a large number of rays with different initial angles, we obtain a visualization of the complete sound field with shadow zones, with particular low sound intensity, and convergence zones with high intensity. This visualization is useful in itself but for a more detailed study requires further processing steps consisting of finding all the rays that connects the source with a given receiver location, the so called eigenrays, and thereafter adding the contributions of all eigenrays taking into account their travel times and the amplitudes. The amplitudes are computed from the calculation of the acoustic intensity which again is calculated by using the principle that the power within a space limited by a pair of rays with initial angular separation of $d\theta_0$ centered on the initial angle θ_0 will remain between the two rays, regardless of the rays' paths. The acoustic intensity as function of horizontal range, $I(r)$ is according to this principle given by

$$I(r) = I_0 \frac{r_0^2}{r} \frac{cos\theta_0}{sin\theta} \left| \frac{d\theta_0}{dr} \right|. \tag{E.12}$$

These two processing steps are not carried out in the current GPU implementation, but by another program using the result of the ray tracing calculation. This means that, in addition to the visualization of the ray coverage, we need to store a number of generated rays during the trajectory calculations, such as the eigenrays to any position in space with their travel times and amplitudes.

E.3 Description of a Graphics Processing Unit

A Graphics Processing Unit is a dedicated graphics rendering device for a personal computer or game console. Modern GPUs are very efficient at manipulating and displaying computer graphics, and their parallel structure makes them more effective than typical CPUs for a range of complex algorithms. A GPU implements a number of graphics primitive operations in a way that makes them render much faster than drawing directly to the screen with the CPU. The GPU uses a pipeline architecture to process multiple fragments in parallel. This means that it can do a lot more operations at same time, compared to the CPU. From figure 2 the vertex and fragment processing stages is the programmable part of the pipeline. Making an object visible is shown in figure 3, where the initial data are processed in multiple stages. For every initial data that is pushed through the pipeline the vertex and fragment stages will transform the data in accordance to the instructions given. This instruction are written in a file for each stage, beginning with the vertex processor, which will make primitives out of vertex data. The vertex and fragment processor is taught of as one unit, however they consist of several processors. Since rasterization and interpolation are more demanding than primitive assembly, the fragment processor consist

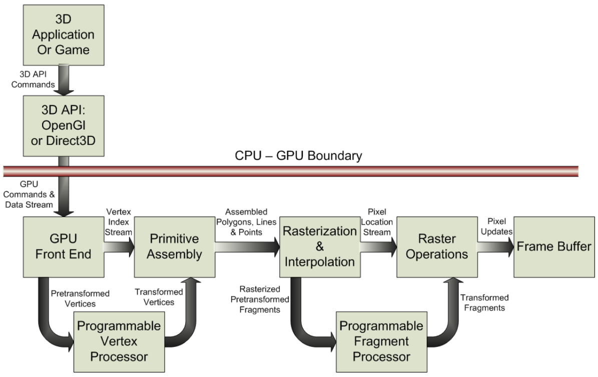


Figure E.2: The graphics pipeline processing stages used to perform rendering to the frame buffer. By programming the vertex and fragment processor to do other things than it was intended for, we can calculate the sound field. Figure are from [18].

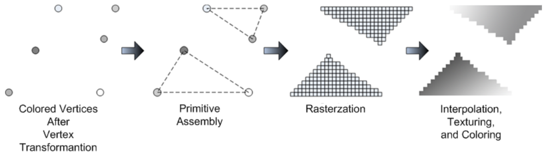

Colored Vertices After Vertex Transformantion
Primitive Assembly
Rasterzation
Interpolation, Texturing, and Coloring

Figure E.3: In the graphics pipeline stages the initial data are colored before making the primitives. The primitives can be triangles or lines. By rasterizing and interpolating the primitives, a visible object is made. Figure are from [18].

of more processors. The result sent to the frame buffer is than displayed on the screen as e.g a character in a computer game.

E.4 Implementation

Calculating a sound field with PlaneRay are done in steps, using the result as input to next iteration of the algorithm. Transferring this to the GPU can be done by writing the algorithm in a shader file, which executes the instruction on every element going through the pipeline. Thinking of one pixel as one ray will make the shader calculate one step for the ray tracing algorithm as one frame is generated. As the time steps forward, a number of frames will be created depending on the pixel grid size, shown in figure 4. This will make the ray path through the ocean. Calculating a step for all the rays in parallel, each pixel needs different initial values. This can be made possible with texture lookup. In figure 5 the program runs trough one step, calculating a number of rays from the pixel grid size. The values are different for each pixel, which will make the output values also different. Visualizing the rays paths, the range values are written to the screen for every iteration. The result are also set as input to next iteration by swapping render targets, described as ping-ponging. This property are common in GPGPU applications.

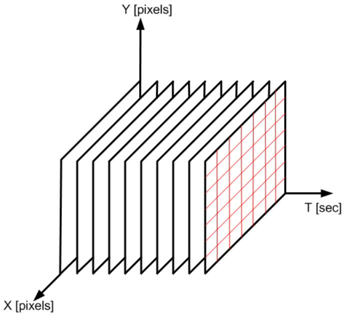


FIGURE E.4: For every frame the program creates, the pixel grid size determines the number of rays to be generated.

The frame buffer is the GPUs memory, and usually this memory is overwritten for every loop for visualization purposes. When storing the wanted values they must be written to the CPU for permanent storage, since the GPU overwrites the memory for every iteration. This read back will slow down the overall calculation speed, because the program must write the calculated value to the CPU before proceeding. However using asynchronous read back this bottleneck can be minimized.

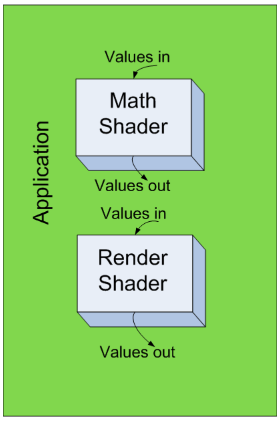


FIGURE E.5: After a step in the ray-tracing algorithm is done, the results are written to a render target for input to next iteration. For visualization of the result, it is also sent to a render shader. The rays path will than be displayed on the screen.

E.4.1 Initial Settings

The first step is to set up the settings for the keyboard, window size and view space for the data. This is the same as setting up a regular GPU application. The next step is to read in the sound speed profile from a text file into an array on the CPU. This data is then copied as a texture on the GPU. By the help of Shallows this is done in one sentence. With current program follows five different profiles to choose from, however any profile can be used. The GPU needs somewhere to store the data after calculation, and usually this is done in a render target. A render target is generic name for data storage on the GPU, such as a frame buffer or texture. Since the calculation needs to be visualized, an on- and off-screen frame buffer is needed. This two are set in the beginning of the program, along with four render targets for swapping data in and out of the shader. GPU does all calculations in parallel, and creates a frame for ever iteration of the code. A GPGPU application can take advantage of this by using a grid made of out of pixels to execute a number of calculation. For a calculation grid out of 7x7, will in current application result in 49 rays, since each pixel is taught of as one ray. Each pixel is computed accordingly to the instructions given in the shader file. The instructions are the same for all pixels, however the input values are different.

E.4.2 Shaders

A shader is a small program within the main program which does the mathematics calculations and rendering. The shaders contains instructions for both the vertex and fragment part in the pipeline. In current code the vertices are only passed trough to the fragment processor. In a texture lies values to be used in calculation in the form of a four-component vector. Meaning four different values can be sent to each pixel from one texture. Current program uses two four-component vectors in the ping-ponging, meaning eight values for each pixel are written to the earlier set off-screen frame buffer after each calculation. When finding what values belonging to each pixel the *texture2D* command is used. Here the first four values in the texture are set to pixel number one, next four values to pixel number two, and so on. When it comes to the sound speed profile the values are not changed when calculating, however the data is stored in a constant texture. Instructions used are the formulas from initial Ray Tracing section. The results of the calculation are stored in two four-component vectors using swizzle operators.

E.4.3 GPGPU workflow

For making the shader files and initial settings working together a work flow is needed. The work flow starts with creating textures from arrays. Using boost shared pointer gives the program an easy control over the texture, which in this case is a 4 component vector. The newly created textures now contains the initial values earlier set in the program. This initial values are the earlier mentioned input to the shader.

The results are stored in a given frame buffer, and for visualization another frame buffer is used. Which shader to use is given by a simple command. Reshaping the calculation grid must be done so the wanted number of rays are calculated, done with the reshape function. At the end, the output render target are swapped with an other render target so the calculated values are set as input to next iteration.

By visualizing the result of the other shader pass the calculated values are shown on the screen along with a visualizing of the bottom and a time- and frame-counter. This gives an impression of how fast the program calculates. From figure 4, 49 rays are calculated, done as one frame. When the program makes 350 frames per second (fps), $49x350 = 17150$ different ray calculations are done in one second.

At the end all results in the frame buffer must be stored in an array on the CPU. The reason this has to be stored at the CPU is because the frame buffer in GPU is overwritten for every calculation. When not clearing this, the visualization of the results will be the rays path on the screen. Since the frame buffer is over written for every calculation, the data in the window will be not be permanent. Clearing the frame buffer will make the visualized path to be erased. So for safe storage the results needs to be sent to the CPU. Doing this by not interfering with the speed of the calculation, asynchronous read back must be used. This means that the read back of data is done simultaneously while the calculations executes. The values from the read back process are stored in a dimensionless vector, with range values for bottom depth, surface depth and receiver depth when ray hits the respectively place. Also some additional information is than added in the vector like start angle, intersection angle and travel time. All this values shall be used in other stages in the PlaneRay model.

E.5 Examples of calculated sound fields

In the following we present two examples of sound field calculated by the ray tracing program described above. The two examples are typical for sound propagation under summer and winter conditions at particular location in the Norwegian sea where the water depth varies from 400 to 350 meter. In practice the sound speed values are obtained by measurements at certain depths with relative large spacing. The first step is therefore to interpolate the measured sound speed points to even spacing Δz, Equation (10). The values of Δz is important for the accuracy of the calculation, in the examples we have used the value of $\Delta z = 0.5$ meter, the total number of layers is therefore 800. In both cases the rays have angles between 5° and $-5°$. The sound fields for winter and summer conditions are shown in Figure 6 and 7 respectively. In both cases, the interpolated sound speed profiles are shown to the left and the ray tracing to the right.

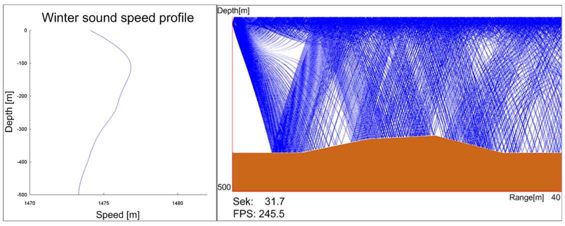


FIGURE E.6: With the source at 20 meter of depth using a winter sound speed profile the rays has a concentration near the water surface. Here the layer thickness are 0.5 meter.

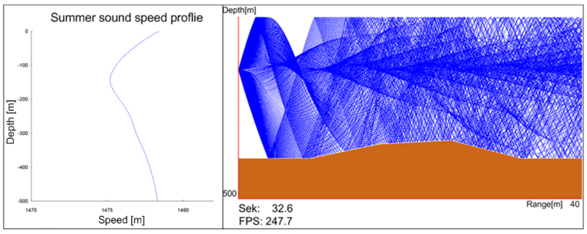


FIGURE E.7: With the source at 150 meter of depth and summer sound speed profile, the rays has a concentration around depth of source, resulting in a sound tunnel. Layer thickness are 0.5 meter.

E.6 Discussion and evaluation

Comparing this GPU application to what the same Matlab application performs, the computational time difference is increasing with number of rays. As one can see from figure 8 the implementation in GPU is 45 times faster than the implementation in Matlab when calculating a 200x200 grid, which results in 40000 rays. With more rays the current[1] hardware has limitations, and Matlab is than not suited for calculation. From figure 9 one can see how the frame rate decreases with number of rays. However the GPU can handle grid size up to 410x410, but than the frame rate is low. Theoretically the GPU is 200 times faster than Matlab when 168100 rays are computed. This number is an approximate from the fact that when four times the rays, the Matlab calculation time is four times higher. This applies for small number of rays, and the estimate are done by assuming this characteristic property resume also with large number of rays. Since Matlab runs on the CPU, it is not optimal CPU code. An application in Matlab will not calculate as fast as a straight forward CPU application. Estimating CPU vs Matlab the CPU is 2-4 times faster [19], depending on the application.

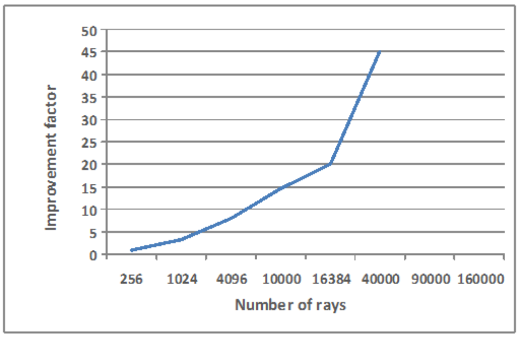


FIGURE E.8: The GPU has increased the calculation time for high density of rays, however for small number of rays Matlab outperform the GPU.

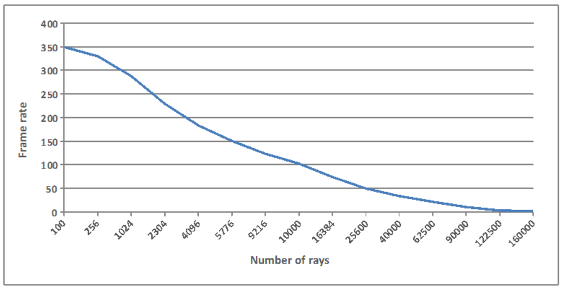


FIGURE E.9: The GPU has high frame rate when small number of rays are calculated, and this is decreasing with number of rays.

E.6.1 Bottlenecks

One bottleneck is the read back of data, which come into being when values from the calculation are to be stored at the CPU. Since the CPU gets the values from the GPUs frame buffer, the GPU has to wait for the values to be stored at the CPU. In the current program the read back is only done up to 10 km, however this can be extended.

Another destructive speed effect for the program is the use of *if, else, for* and *while* loops in the shader file. To much texture lookup will also create the same effect.

[1] Intel Centrino M 740 processor with 512 MB DDR2 and nVidia GeForce Go6600 with 64 MB VRAM

E.6.2 Further Work

As mentioned earlier, effectively programming the shader files will probably increase the speed of the application. An improvement for the read back of data is doing it asynchronously, getting the values to the CPU by using a pixel buffer. This will make the GPU and CPU write and read from the same memory. Since this is done at the same time, it will have a positive effect on the computational speed.

Instead of using *if* and *else* sentences, the use of arithmetic operators are more efficient. Also a Beizèr spline interpolation of the sound speed values between two point, would make the ray tracing more accurate. As of today the interpolation is linear. This interpolation routine is only an increase of data when the layer thickness is less than 1 meter in this application, so the acoustic model will not be affected by this. Only the result.

In 1992 a method for nonlinear inversion for ocean-bottom properties [21] was calculated on a supercomputer at that time[2], at 250 MFLOPS it took 1.3 hours to calculate. State of the art graphics hardware today[3] could theoretically do this in 2.11 seconds [14] using 550 GFLOPS.

Starting with this article, calculations methods in marine acoustics can take advantage of this speedup.

[2] CRAY X-MP/24

[3] ATI X1900