Calculation of underwater sound fields with Graphic 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

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 [10], and tracing light rays [8] [7] 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 [11], 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.

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) [2] or Medwin and Clay (1998) [1]. 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

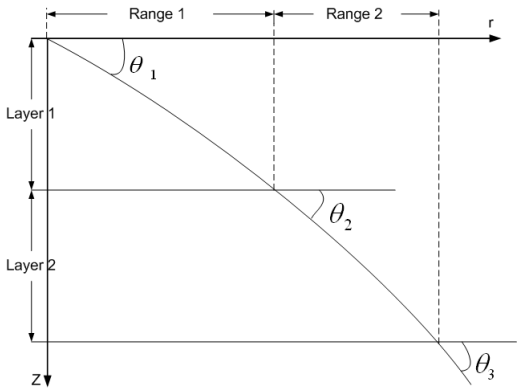

FIGURE 1: The algorithm step with $\Delta layer$ and calculate a range increment for every step. The sum of all layers range will give the total range for the ray.

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

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

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,

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

The ray parameter is defined as:

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

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

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

which also can also be written in the form

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

The local sound speed gradient is approximated by

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

The travel time increment is

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

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.

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

Consequently, the ray parameter has to change to

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

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

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

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,

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

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

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

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.

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

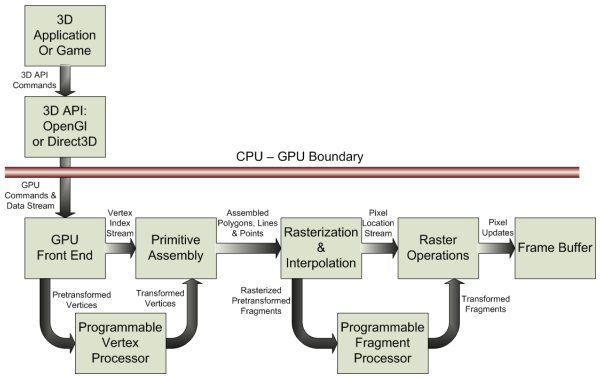

FIGURE 2: The graphics pipeline processing stages used to perform rendering to the frame buffer. By programming the vertex and fragment processor to do other things than it was intended for, we can calculate the sound field. Figure are from [14].

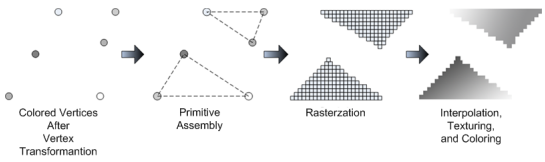

FIGURE 3: In the graphics pipeline stages the initial data are colored before making the primitives. The primitives can be triangles or lines. By rasterizing and interpolating the primitives, a visible object is made. Figure are from [14].

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 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.

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

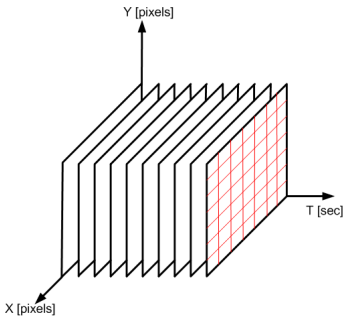

FIGURE 4: For every frame the program creates, the pixel grid size determines the number of rays to be generated.

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.

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.

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

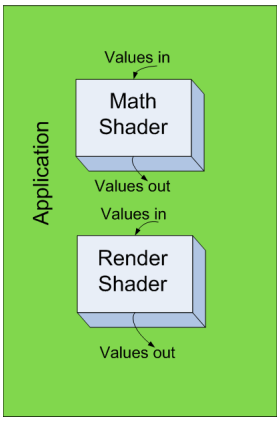

FIGURE 5: After a step in the ray-tracing algorithm is done, the results are written to a render target for input to next iteration. For visualization of the result, it is also sent to a render shader. The rays path will than be displayed on the screen.

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.

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.

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.

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

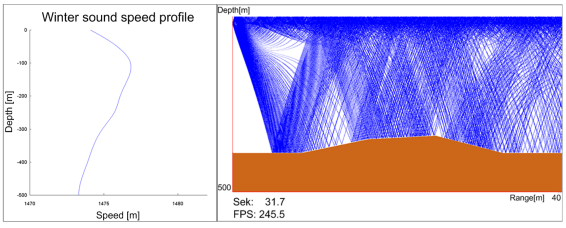

FIGURE 6: With the source at 20 meter of depth using a winter sound speed profile the rays has a concentration near the water surface. Here the layer thickness are 0.5 meter.

measurements at certain depths with relative large spacing. The first step is therefore to interpolate the measured sound speed points to even spacing

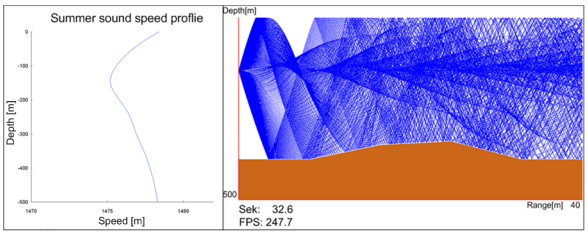

FIGURE 7: With the source at 150 meter of depth and summer sound speed profile, the rays has a concentration around depth of source, resulting in a sound tunnel. Layer thickness are 0.5 meter.

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.

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

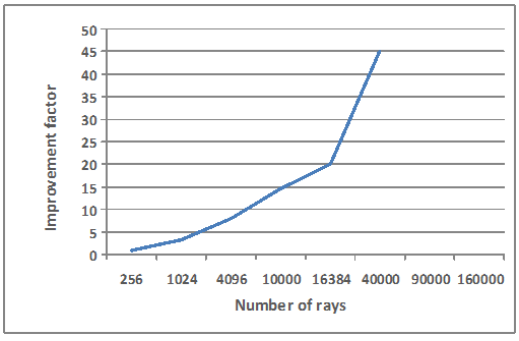

FIGURE 8: The GPU has increased the calculation time for high density of rays, however for small number of rays Matlab outperform the GPU.

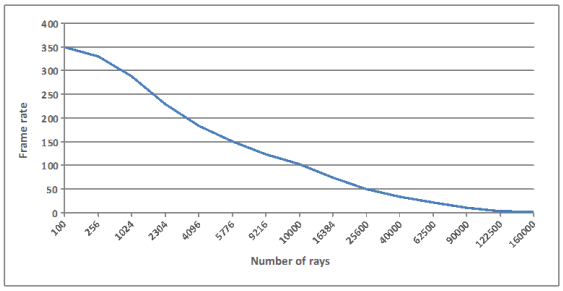

FIGURE 9: The GPU has high frame rate when small number of rays are calculated, and this is decreasing with number of rays.

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 [15], depending on the application.

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

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 [12] 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 [13] using 550 GFLOPS.

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

References

[1] MEDWIN, H. AND CLAY, S. C. *Fundamentals of acoustical oceanography.* Academic press, Boston, 1998.

[2] 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.

[3] OPENGL ARCHITECTURE REVIEW BOARD. *OpenGL Programming Guide, Fourth Edition*, Addison-Wesley, 2004.

[4] OPENGL ARCHITECTURE REVIEW BOARD. *OpenGL Reference Manual, Fourth Edition*, Addison-Wesley, 2004.

[2]CRAY X-MP/24

[3]ATI X1900

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

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

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

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

[9] 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.

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

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

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

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

[14] Fernando, K. and Kilgard, M. J. *The Cg tutorial. The definitive guide to programmable real-time graphics* Addison-Wesley, Boston, 2003.

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