GPUwave: An implementation of the split-step Fourier method for the GPU

Steinar H. Gunderson

Institutt for elektronikk og telekommunikasjon
NTNU 2006

Abstract

We present *GPUwave*, a free implementation of the split-step Fourier method utilizing the power of modern graphic cards (also known as Graphics Processing Units, or *GPU*s). We discuss various practical issues in maximizing effiency of the resulting implementation, with an emphasis on the discrete sine transform (DST) and the fast Fourier transform (FFT) in a GPU context. Also, issues related to floating-point accuracy are discussed.

Performance-wise, GPUwave has been found to be up to three orders of magnitude faster than *PEEC*, a comparable CPU-based implementation of the split-step Fourier method, and the accuracy has been found comparable to that of RAM and OASES, two well-known CPU-based solvers of the same problem. We present benchmarks and accuracy comparisons, demonstrating the feasibility of using a GPU-based solver for this class of problems. Finally, future work in the area is briefly discussed, both in the context of improving GPUwave itself, and in considering implementing other algorithms for the GPU.

Contents

Chapter 1

Introduction

1.1 Problem overview

The aim of this project was to simulate underwater sound fields using the power of modern graphics cards. A practical solver of such fields, based on the standard split-step Fourier method, was implemented, aiming at obtain increased efficiency over traditional implementations. Also, the numerical accuracy of the solver was tested by comparing its results against the results of already existing implementations, some of which were based on different methods. Being a GPU-based solver of the wave equation, the name *GPUwave* was chosen for the solver, which has been released under a free license.

The problem of simulating sound wave propagation in water is not new; the wave equation has been known since the 19th century, and since the introduction of the *sonar* in the 1910s, the problem has also been interesting for practical purposes, both military and civilian. Furthermore, as computers gradually became more commonplace in the 1970s, *numerical* solutions (which are required for accurate descriptions of all but the most trivial problems) became feasible, which has further advanced the field.[18].

In the last five years, the *Graphics Processing Unit*, or *GPU* for short[1], has become an increasingly interesting computational tool. The GPU provides large amounts of inexpensive computational power — however, its programming model differs significantly from that of traditional CPU-based computation. Thus, the primary challenge of the project was to adapt a previously known solution method, in this case the split-step Fourier method, to the GPU programming model.

1.2 Structure

The three remaining sections of this presentation are:

- **Background:** For readers not already familiar with the field of GPU programming and/or the split-step Fourier method, chapters 2 and 3 contains brief background informations on these two topics, respectively. Also, in chapter 4, the *fast sine transform* (FST) is introduced and its implementation discussed; while not being new material, it is not frequently presented in textbooks, and as such, it is included here for reference.

[1]The term Visual Processing Unit, or VPU, is also sometimes used, mainly by graphic card manufacturers. It appears to have been meant to distinguish between two different generations of graphic cards at one point, but is now used interchangeably with GPU. We will stick to the usual terminology and use the term "GPU".

- **Methods:** In chapter 5 implementation issues are discussed, particularly with regard to the challenges presented by the GPU programming model. Specifically, issues related to memory packing, efficient implementation of the *fast Fourier transform* (FFT), and numerical accuracy on the GPU are discussed.
- **Results:** In chapter 6, benchmark results are presented, and the accuracy of GPUwave is compared to that of already existing implementations. Finally, the experiences from using the GPU for this task are summarized, and some ideas for further work are outlined.

1.3 Earlier work

On the GPU side, the first to use the GPU for solving differential equations (using finite difference methods) was probably Mark Harris, who implemented a full real-time cloud simulation on the first series of properly programmable GPUs[13]. In the field of ocean acoustics, a ray-tracing scheme has earlier been implemented on the GPU with only limited success[15], citing issues with *read-back* from the GPU to the CPU as the primary performance bottleneck. The author knows of no previous work on implementing the Fourier-step method on the GPU, although the fast Fourier transform, a central part of the algorithm, has been implemented multiple times; see [24] and others.

Chapter 2

The GPU programming model

2.1 Introduction

Providing a full tutorial to the field of general-purpose GPU (usually abbreviated to *GPGPU*) programming is beyond the scope of this project — here, only some key facts of the GPU programming model will be summarized, motivating some of the decisions made in the implementation stage. The reader is referred to [14] for a short tutorial on GPGPU programming, or [12] for a more thorough treatment of the subject. For an introduction to OpenGL in general, the standard reference is [2]; [1] and [29] are also useful as reference material.

The GPU programming model can be summarized as follows:

- Data is primarily stored in *textures*, which are 1D or 2D (or sometimes even 3D) arrays of four-component[1] floating-point vectors.
- Writes to these textures are not done in an explicit fashion; instead, one constructs a *fragment shader*[2] (a small program), which based on a set of input textures and some extra parameters, either constant or varying linearly between the corners of the area calculated, calculates a single four-component output vector which is stored in the output texture.[3]
- Output textures can not simultaneously be used as input textures[4]; in other words, no in-place calculations are allowed.

There is other slightly more esoteric functionality, such as blending, fog, mipmaps and the depth buffer, that can be useful in some GPGPU-related situations, but this project has only used the previously described functionality, and as such these other issues will not be discussed further here.

[1] Formally, vectors on the GPU do not have to be four-component. However, they can not have more than four components, and in most GPU implementations using fewer is as expensive as using all of them.

[2] The term "shader" is somewhat unfortunate, since it indicates that shaders are only used for lighting effects. The OpenGL Architecture Review Board attempted for a while to standardize on the terms "vertex program" and "fragment program", but eventually, with the advent of GLSL, reverted to the more common terminology.

[3] Some manufacturers allow multiple output values to be computed and stored into multiple distinct output textures; however, this functionality is at the time of writing still not properly standardized. Its use for the FFT will be discussed in chapter 5.

[4] Actually, it is sometimes possible, but the results are undefined, and one should not expect the results to be portable across different models, makes or even driver revisions.

The GPU programming model's primary advantage is that most computations are *trivially parallelizable*, which is what enables the GPU to achieve such a high throughput. On the other hand, adapting an algorithm to this model can prove a challenge, and as such often careful evaluation of both methods and implementation strategy is needed.

Both the actual GPUs, the programming model and the corresponding tools have matured significantly in only a few short years. Shaders are currently usually written in a C-like high-level language (such as Microsoft's HLSL, nVidia's Cg, or OpenGL's GLSL), and there is ongoing work to create even more developer-friendly languages[3, 21], with few of the arbitrary limits that characterized the early implementations. Specifically, 32-bit IEEE floating point[5] is widely available, as is full support for loops and branches (although with some performance considerations; see the next section). Still, the field is expected to advance considerably in the time to come, both in terms of the environment itself and actual usage of GPU-related techniques..

2.2 Performance

The GPU programming model is significantly different from the classic CPU programming model; this voids much existing knowledge about algorithm performance, and presents many new challenges. Also, GPUs are quickly rivalling CPUs in implementation complexity, and the performance-aware developer must consider not only very long pipelines, but also several layers of cache, varying cache associativity, extensive SIMD capabilities and many other factors. As the architecture becomes more complex, analyzing the performance of a given algorithm becomes more involved, just as in the modern CPU world — indeed, the days are gone when one could simply count the number of operations to find the most efficient algorithm[10].

To further add to the complexities, the GPU is much more of a "black box" than the CPU is, mainly because the abstraction layer offered is much thicker than what is available in terms of the assembly language of a CPU[6]. As such, finding "hard and fast" rules is difficult, but there are still many useful rules of thumb, including:

- It is better to ask for large areas at a time than many small, mainly since this improves the opportunities for parallelism and puts less work on the host CPU.
- Adding more calculations is slow, but adding more texture fetches is usually even slower. In other words, *memory bandwidth* is often the main bottleneck.
- Switching shaders (or in general, any state change requiring breaking up batches of geometry) frequently is slow.
- Complex functions, such as sine or cosine, are usually slower than multiplications and additions, and can usually not be vectorized like the simpler operations.
- "Ping-ponging", that is, switching an output texture to be used as an input texture, used to be slow (in particular since it needed an explicit copy), but has (especially with improvements to the graphics APIs) improved markedly over the years.

[5] There are some differences with what GPUs implement and the IEEE 754 standard; for instance, there is no support for infinities, NaN or denormals, as these values are often cumbersome to handle efficiently in hardware.

[6] This is expected to improve somewhat as the more GPGPU-specific development environments enter the market.

- If branching, one must be prepared to assume that both parts of the branches are calculated for every fragment. (This does, however, vary from model to model and from make to make. Some GPU architectures support calculating only the taken part of the branch if all fragments in some local group branch the same way, ie. the path taken varies relatively slowly with the pixel coordinates.)
- Clearing buffers are surprisingly slow. (The FFT algorithm discussed in chapter 5 was sped up by over 30% just by removing the clears from the beginning of each step of the algorithm.)
- Pulling data back from the GPU to the CPU is *very* slow, and use of such read-back in performance-critical sections should be avoided wherever possible.[7]

In addition, there are many other more specific microoptimization rules available for specific makes and models of GPUs (see for instance [19]). They will not be discussed here, and as the model has moved to high-level languages, many of them are now hopefully taken care of in the shader compiler layers, relieving the programmer of the burden of considering too many different abstraction layers at the same time.

Even as the programming environments and models are maturing, working with the GPU programming model requires a re-thinking of many of common implementation strategies, and in chapter 5 several such problems will be encountered and considered. However, first a precise mathematical definition of the problem in question and its numerical solution will be needed; thus, we will now turn to a brief review of the mathematical formulation of the split-step Fourier method.

[7] This is said to be somewhat better with asynchronous readback, and with the use of PCI Express cards instead of the earlier AGP cards. [12] claims 18 ms to read a 1024x1024 buffer of unspecified precision from a GeForce 6800 to the CPU over the PCI Express bus.

Chapter 3

The split-step Fourier method

3.1 Environment

The model discussed closely resembles the one described in [25]. We consider a 2D cross-section of the ocean along the r (radial distance from the origin) and z (depth from the sea level) axes. This cross-section (which is actually a quarter-plane, as the region of interest is restricted to $r > 0, z > 0$) has one sediment layer[1] and below that a special *sponge layer* absorbing all waves to make sure nothing is reflected from the bottom of the field.

The relevant quantities in the model are:

- $p(r, z)$, the pressure (which is to be solved for).
- z_s, the depth of the sound source.
- f_0, the frequency of the source. (The parallel nature of the GPU allows solving for many different frequencies in parallel.)
- $D(r)$, the depth at which the sediment layer begins.
- $c(z)$, the sound speed profile (influenced by $D(r)$).[2] The reference sound speed at the source is denoted c_0.
- $\alpha(r, z)$, the dampening (influenced by $D(r)$, and often related to f_0).
- $\rho(r, z)$, the density (influenced by $D(r)$).
- $\tilde{n}^2(r, z)$, the complex index of refraction. $\tilde{n}^2(r, z)$ is computed directly from f_0, $D(r)$, $c(z)$, c_0, $\alpha(r, z)$ and $\rho(r, z)$, and together with the reference wave number $k_0 = \frac{2\pi f_0}{c_0}$ it incorporates *all* the information needed from the model in a single complex number.

It is worth noting that every input parameter, except $\tilde{n}^2(r, z)$ (which is generated "on the fly"), is either a constant or an 1D array. This means that one will usually not need to worry about the access patterns against them, as the total amount of data is small and thus will fit into most caches.

[1] There is nothing in the model that prevents the implementation of multiple sediment layers, but GPUwave supports only one.

[2] The implementation described in [25] supports a sound speed profile that varies with distance; however, in the interest of saving memory bandwidth, this feature was cut from the GPU model; re-adding it would, however, not be difficult if it is deemed necessary.

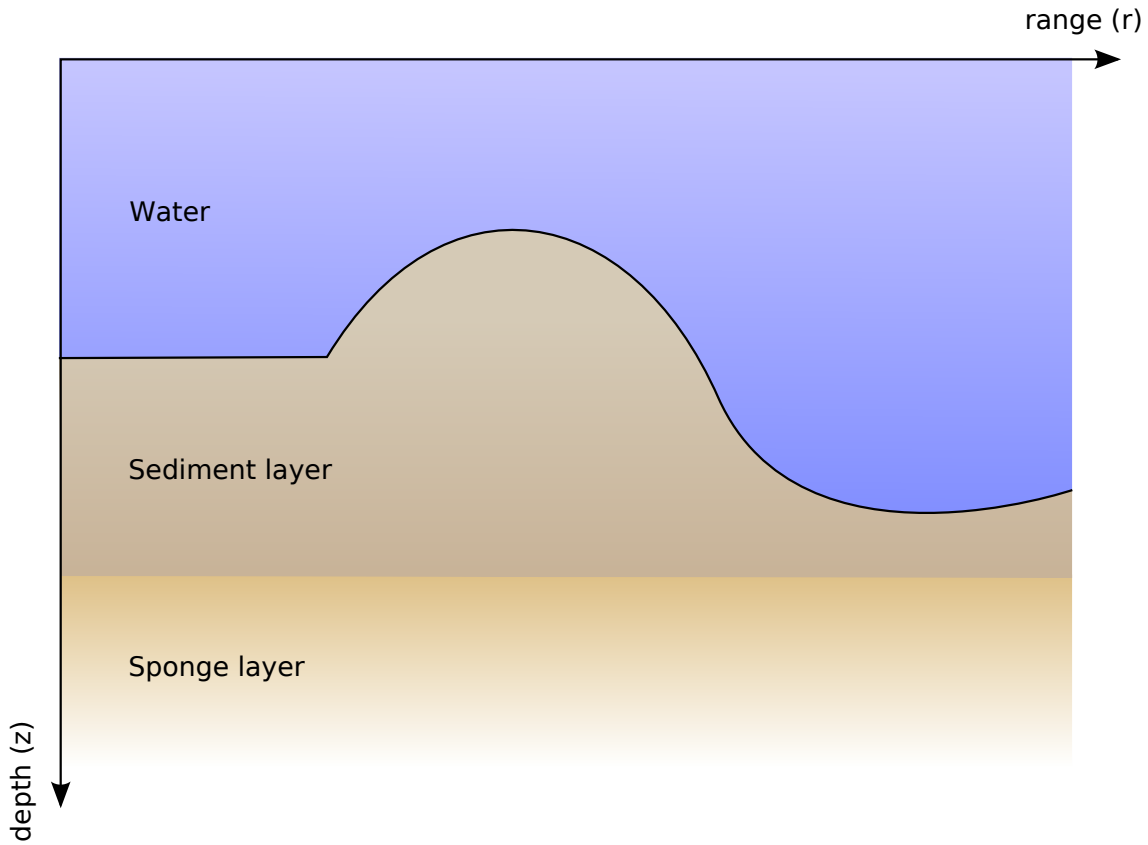

Figure 3.1: The model of the ocean with the water, a sediment layer and the artificial sponge layer.

3.2 Solution outline

Given non-trivial constraints, the problems of this class quickly become too complex for symbolic solving, and numerical methods must be used instead. Several methods have been proposed; the most obvious is direct solution using *finite difference* methods, but also *Padé-step methods* and more recently methods based on ray tracing[16] have been proposed. Here, only a *Fourier-step method* is considered, basing much of the acoustics on [25]; no attempt is made to give a detailed theoretical comparison of this method with other methods.

The derivation of the algorithm will only be outlined in very rough steps; for a complete reference, the reader is referred to [18].

3.2.1 Problem statement

The derivation begins with the Helmholtz equation in cylindrical coordinates:

$$\frac{\partial^2 p}{\partial r^2} + \frac{1}{r}\frac{\partial p}{\partial r} + \frac{\partial^2 p}{\partial z^2} + {k_0}^2 \tilde{n}^2 p = 0 \tag{3.1}$$

It is then assumed that the solution is on the following form:

$$p(r,z) = \psi(r,z) {H_0}^{(1)}(k_0 r) \tag{3.2}$$

where ${H_0}^{(1)}(k_0 r)$ is the *Hankel function*. The problem is thus reduced to finding $\psi(r,z)$, after which one can simply multiply the computed value of $\psi(r,z)$ by the Hankel function to obtain the final answer.

Also assuming that we are in the far field and that the source has a narrow opening angle, this gets reduced to

$$2ik_0\frac{\partial\psi}{\partial r}+\frac{\partial^2\psi}{\partial z^2}+{k_0}^2(\tilde{n}^2-1)\psi=0 \tag{3.3}$$

which is the form used from here on.

3.2.2 Transformation and solving

Keeping $\tilde{n}$ constant and applying the Fourier transform to $\psi(r,z)$ along the z axis gives rise to a new function $\hat{\psi}(r,k_z)$. The Fourier transform of (3.3) is

$$\frac{\partial\hat{\psi}}{\partial r}-\frac{{k_z}^2+{k_0}^2(\tilde{n}^2-1)}{2ik_0}\hat{\psi}=0 \tag{3.4}$$

which is a first order differential equation. Solving (3.4) for $\hat{\psi}$ and applying the inverse Fourier transform (transforming $\hat{\psi}(r,k_z)$ back into $\psi(r,z)$) yields

$$\psi(r,z)=e^{i(\tilde{n}^2-1)\frac{k_0}{2}(r-r_0)}\int_{-\infty}^{\infty}\psi(r_0,z)e^{-ik_z{}^2\frac{1}{2k_0}(r-r_0)}e^{ik_z z}\mathrm{d}k_z \tag{3.5}$$

This equation describes the continuous version of the Fourier-step algorithm — in essence, it describes the relationship between $\psi(r_0,z)$ and $\psi(r,z)$, using the Fourier transform and its inverse to move between the time and frequency domains.

3.2.3 Discretization

A suitable discretization for $\psi(r,z)$ is needed to implement the model numerically, after which inserting $r=r_0+\Delta r$ into (3.5) gives an algorithm than can be implemented directly. The preferred choice is a uniform lattice, selecting suitable Δr and Δz:

$$\begin{aligned} r_m &= m\Delta r, & m=0,1,2,3,...,M-1\\ z_n &= (n+\tfrac{1}{2})\Delta z, & n=0,1,2,3,...,N-1 \end{aligned}$$

Note that for numerical reasons, a mid-point discretization of z has been chosen; in chapter 4, it will be seen that the choice of discretization is important for the numerical implementation of the Fourier transform and its inverse.

In addition, a discretization of $\hat{\psi}(r,k_z)$ is needed. Keeping Δr the same as for $\psi(r,z)$, the discretization for k_z is again rather simple:

$$k_{zl} = (l+\tfrac{1}{2})\Delta k_z, \quad l=0,1,2,3,...,N-1$$

where again a mid-point discretization is favored. (Note that the discretizations of z and k_z use the same amount of steps, as would be expected from the relationship between the variables.)

However, Δk_z can not be chosen arbitrarily; to ensure that the Fourier transform and its inverse are transform pairs also after discretization, it is required that

$$\Delta z\Delta k_z=\frac{2\pi}{N} \tag{3.6}$$

With the choice of parameters in place, (3.5) can be discretized, giving

$$\psi(r_{m+1}, z) = e^A \mathcal{F}^{-1} \left\{ e^B \mathcal{F} \left\{ \psi(r_m, z) \right\} \right\} \tag{3.7}$$

where

$$A = i(\tilde{n}^2(r_m, z) - 1)\frac{k_0}{2}\Delta r \tag{3.8}$$

and

$$B = -ik_z{}^2 \frac{\Delta r}{k_0} \tag{3.9}$$

which then together define the discrete version of the Fourier-step algorithm; given a suitable starting field $\psi(0, z)$, equation (3.7) gives $\psi(\Delta r, z)$, which gives the information needed to calculate $\psi(2\Delta r, z)$, and so on in a recursive fashion. The algorithm has been shown to be unconditionally stable[18], although the accuracy of the final result will of course vary with different discretizations.

3.2.4 The complex index of refraction

As mentioned in section 3.1, the index of refraction $\tilde{n}^2$ will embody most of the information present in the model. The derivation of this quantity starts with the ordinary, real index of refraction:

$$n(r, z) = \frac{c_0}{c(r, z)} \tag{3.10}$$

where c is the speed of sound at a given range of depth, and c_0 is the reference speed of sound, that is, the speed of sound at the source. $c(r, z)$ is normally equal to $c(z)$, the sound speed profile of the water for a given depth, but in the sediment layer, it instead assumes the value c_r, the speed of sound in the sediment (naturally). Mathematically:

$$c(r, z) = \begin{cases} c(z) & ; z < D(r) \\ c_r & ; z \geq D(r) \end{cases} \tag{3.11}$$

where $D(r)$ is, again, the depth.

There will also be *attenuation*, here denoted by $\alpha(r, z)$ (measured in dB/λ and thus dependent on frequency). It is equal to either α_w or α_s, depending on whether the point in question is situated in the water or in the sediment layer, that is:

$$\alpha(r, z) = \begin{cases} \alpha_w & ; z < D(r) \\ \alpha_s & ; z \geq D(r) \end{cases} \tag{3.12}$$

The incorporation of the attenuation to the index of refraction $n(r, z)$ (which is squared for convenience) turns it into a complex quantity $\bar{n}^2(r, z)$, as in

$$\bar{n}^2(r, z) = \left(\frac{c_0}{c(r, z)} \right)^2 (1 + i\frac{\alpha(r, z)}{20\pi \log_{10} e}) \tag{3.13}$$

where the division by the constant $20\pi \log_{10} e \approx 27.29$ is introduced when converting α from dB/λ to Nepers/m.

Lastly, one will also need to take into account the *density* $\rho(r, z)$. Its effect on the index of refraction (now in its final form) is:

$$\tilde{n}^2 = \bar{n}^2(r,z) + \frac{1}{2{k_0}^2}\left[\frac{1}{\rho}\frac{\partial^2\rho}{\partial z^2} - \frac{3}{2\rho^2}\left(\frac{\partial\rho}{\partial z}\right)^2\right] \tag{3.14}$$

From (3.14), it can be seen that ρ must be twice differentiable in the z-direction, so a simple, discontinuous step function (like used for $c(r,z)$ and $\alpha(r,z)$) will not suffice. Instead, the favored approach, as originally introduced in [33], is to introduce a smoothing factor around the step at $D(r)$:

$$\rho(r,z) = \frac{1}{2}(\rho_s + \rho_w) + \frac{1}{2}(\rho_s - \rho_w)\tanh\left(\frac{k_0}{2}(z - D(r))\right) \tag{3.15}$$

The factor $\frac{k_0}{2}$, controlling the length of the smoothing area, has been chosen carefully to make sure this smoothing will not interfere with the rest of the algorithm.

The question soon arises as of how to calculate $\tanh' x$ and $\tanh'' x$ efficiently. GLSL provides automatic calculation of the derivatives of almost any function (since they are needed for other purposes, such as mip-map selection, in graphics applications), but these might be unsuitable due to poor and somewhat undefined numeric properties — for instance, in one particular implementation, the derivatives are calculated by looking at differences within a 2x2 fragment block[19], which means that forward- and backward-difference schemes will be used for every other fragment. Furthermore, GLSL does not provide double derivatives.

A commonly used solution is to calculate $\rho(z)$ for a given r, and then use the finite differences between neighbor values to approximate the needed derivatives. While possible to implement on a GPU, this would also necessitate at least one extra pass, and thus require more memory bandwidth.

Fortunately, since tanh is ultimately composed of exponentials (it is defined as $\tanh x = \frac{e^{2x}-1}{e^{2x}+1}$), it has a very regular structure, and its first and second derivatives can be expressed as:

$$\tanh' x = 1 - \tanh^2 x \tag{3.16}$$

$$\tanh'' x = -2\tanh x(1 - \tanh^2 x) = -2\tanh x \tanh' x \tag{3.17}$$

These formulas can be used to simply and efficiently calculate the needed quantities on-the-fly in the shader with no extra bandwidth cost; to calculate $\tanh x$, $\tanh' x$ *and* $\tanh'' x$ all at the same time, only a single exponential, one division and a few additions and multiplications are needed.

It should be mentioned that since $\tanh x$ is very close to constant outside the central region (of about $|x| < 3$), all its derivatives outside this region could be approximated with zero (and 0 or 1 for the function itself, as appropriate). However, due to the relative (and somewhat unpredictable) cost of a branch in a shader and the efficiency of the calculations, it was not clear if this would be a net win; thus, this was not done. However, x was clamped to $[-20, 20]$ to avoid overflow problems in the exponentials.

This concludes the description of the Fourier-step method. In chapter 5, the practical GPU implementation of this algorithm will be discussed in detail; however, there is still one more piece of mathematical background that will need some consideration, namely that of the discretization of the Fourier transform, and the implementation of the resulting algorithm. Thus, we will now turn our attention to a discussion of the *fast sine transform*.

Chapter 4

The Fast Sine Transform

4.1 Introduction

One of the main computational workhorses of the split-step Fourier algorithm is the Fourier transform and its inverse, discretized and implemented numerically. The most commonly used discretization of the Fourier transform is the *discrete Fourier transform* (DFT), usually implemented as a *fast Fourier transform* (FFT). However, the DFT will not be directly usable in this case, as the DFT corresponds to the case where the given function $p(z)$ can be viewed as a *periodic extension* of the function given by the finite sequence $\{x_k\}$.

This boundary condition is not satisfiable for the problem at hand, as it has a different, natural boundary condition, namely that the pressure must be zero at ocean level; that is, $p(r, 0) = 0$. Thus, a different discretization of the Fourier transform is needed, corresponding to a $p(r, z)$ that is *antisymmetric around* $z = 0$. Given the right such discretization, the boundary condition will automatically be maintained at all times.

For numerical reasons, we do not assume that $x_0 = 0$; storing such a redundant coefficient would not be very useful. A close alternative would be assuming $x_{-1} = 0$ (and the rest of the sequence being antisymmetric around that point), giving rise to the so-called *type-I DST*. However, the type-I DST can have poor numerical properties[10], and this kind of *early discretization* was found to be suboptimal in [25], so it will not be used.

Instead, we begin with the so-called *type-II DST*, the discrete sine transform with odd symmetry around $n = -\frac{1}{2}$ and $n = N - \frac{1}{2}$:

$$X_k = \sum_{n=0}^{N-1} x_n \sin\left[\frac{\pi}{N}\left(n + \frac{1}{2}\right)(k+1)\right] \tag{4.1}$$

We then define

$$S_{2N}^{(2n+1)(k+1)} = \sin\left[\frac{\pi}{2N}(2n+1)(k+1)\right] \tag{4.2}$$

$$C_{2N}^{(2n+1)(k+1)} = \cos\left[\frac{\pi}{2N}(2n+1)(k+1)\right] \tag{4.3}$$

which allows us to write (4.1) as

$$X_k = \sum_{n=0}^{N-1} x_n S_{2N}^{(2n+1)(k+1)} \tag{4.4}$$

Implementing this algorithm directly will give a running time of $O(n^2)$ for calculating $\{X_k\}$; if possible, a running time of $O(n \log n)$, akin to that of the FFT, would be preferable. To this end, the three most commonly discussed methods in the literature will be considered:

- Extension of the input coefficients antisymmetrically around the center, enabling a direct computation by means of an FFT of twice the length.
- Factorization of the DST similar to the derivation of the FFT.
- Pre-processing the input data, rewriting the DST into an FFT of the same length.

Also, the DST and DCT can be viewed as specialized cases of DFTs of *eight* times the length (with appropriate extensions and interleaving with zeros), and implemented efficiently as a regular FFT with computerized removal of redundant or trivial terms.[10] However, this approach does not appear to be easily adjusted to implementation on the GPU, and thus will not be discussed further here.

4.2 Input extension

The most obvious way of implementing the DST (which corresponds to antisymmetric boundary conditions) by means of the DFT (which corresponds to periodic boundary conditions) is simple: Construct a new sequence $\{x'_n\}$ that is antisymmetric, and compute its DFT. In this case, $\{x'_n\}$ would be defined as:

$$x'_n = \begin{cases} -x_{N-n-1} & ; 0 \leq n < N \\ x_{n-N} & ; N \leq n < 2N \end{cases} \tag{4.5}$$

The DFT of such an extended sequence would not be immediately equivalent to the DST; for that, one would need some post-processing[22]. However, in the case of the split-step Fourier method, one can simply adjust the discretization of k_z accordingly, so no such post-processing will be needed. This is the approach implemented in [25].

Even though this method is simple and intuitive, it presents a number of problems:

- As the N-point DST has been implemented by means of a 2N-point DFT, the transform length has been doubled, and the performance suffers — over an N-point DFT of the same size, a factor of about two is lost performance-wise.
- In addition, all calculation in the frequency domain will need to work on the extended sequence, incurring a similar performance loss in the non-DST areas of the algorithm. Furthermore, the storage cost is doubled.
- As the numerical errors grow with the FFT size, doubling the transform length will lead to somewhat worse numerical precision and stability; however, as the typical error of the FFT grows only as $O(\sqrt{\log n})$ [30], this should not be a big problem in practice. It should be mentioned, though, that care must be taken to maintain the antisymmetry for each iteration of the algorithm, as due to rounding errors, the two halves will not be identical after a DFT followed by an IDFT. If this is not done, the boundary condition will not be properly maintained, and the model's accuracy will suffer.

Since half of the output array is in some sense redundant, one could imagine an IFFT algorithm that computed only half of the outputs; however, the computational complexity for an n-point FFT computing k outputs is $O(n \log k)$ [10], so the gains would be minimal.

In [32], the different symmetries of the resulting subsequences when applying the FFT algorithm to such a symmetric sequence are discussed, with possible optimizations. However, as the resulting structure is not very regular, such optimizations do not seem to be immediately useful for implementation on the GPU.

As we have seen, it is entirely possible to use a 2N-point DFT for our uses, but it is far from ideal, and one should search for a better method.

4.3 Factorization of the DST

The derivation roughly follows [34]. We will first need to show an important lemma.

4.3.1 The half-shift property

Given (4.2) and (4.3) above, the following identity holds:

$$\sum_{n=0}^{N-1} x_n S_{2N}^{(2n+1)(k+1)} = \frac{1}{2C_{2N}^{(k+1)}} \sum_{n=0}^{N-1} (x_n + x_{n+1}) S_N^{(n+1)(k+1)} \tag{4.6}$$

Proof. We apply the trigonometric identity $2 \sin a \cos b = \sin(a+b) + \sin(a-b)$. Multiplying the S factor by $2C_{2N}^{(k+1)}$ yields

$$\begin{aligned} 2S_{2N}^{(2n+1)(k+1)} C_{2N}^{(k+1)} &= S_{2N}^{(2n+2)(k+1)} + S_{2N}^{(2n)(k+1)} && (4.7) \\ &= S_N^{(n+1)(k+1)} + S_N^{(n)(k+1)} && (4.8) \end{aligned}$$

which means that the left-hand side of (4.6) can be written as

$$\frac{1}{2C_{2N}^{(k+1)}} \sum_{n=0}^{N-1} x_n \left(S_N^{(n+1)(k+1)} + S_N^{(n)(k+1)} \right) \tag{4.9}$$

or, expanding the braces,

$$\frac{1}{2C_{2N}^{(k+1)}} \left(\sum_{n=0}^{N-1} x_n S_N^{(n+1)(k+1)} + \sum_{n=0}^{N-1} x_n S_N^{(n)(k+1)} \right) \tag{4.10}$$

We now observe that the second term vanishes for $n = 0$, since $S_N^{(0)} = 0$. Furthermore, we are free to add an extra term for $n = N$ to the sum (no matter what x_N is assumed to be), since it will be multiplied by $S_N^{(N)} = 0$. This gives us

$$\frac{1}{2C_{2N}^{(k+1)}} \left(\sum_{n=0}^{N-1} x_n S_N^{(n+1)(k+1)} + \sum_{n=1}^{N} x_n S_N^{(n)(k+1)} \right) \tag{4.11}$$

and a change of index from n to $(n+1)$ in the second sum allows us to join the terms again:

$$\frac{1}{2C_{2N}^{(k+1)}} \sum_{n=0}^{N-1} (x_n + x_{n+1}) S_N^{(n+1)(k+1)} \tag{4.12}$$

■

This shows that it is possible to *half-shift* the coefficients (shifting them by $\frac{1}{2}$) to adjust the sine terms (up to a constant factor); it might not immediately seem like a big achievement, but it simplifies the troublesome sine term containing $(n + \frac{1}{2})$ in the original formulation. (The inverse of the lemma, in that one may rewrite the right-hand side into the left-hand side, is trivial given the forward proof.)

We are now ready to do the actual derivation, following the classical Cooley-Tukey procedure.

4.3.2 Factorization

We split the sum in (4.4) into even and odd parts, a technique known as *decimation in time*. This gives

$$X_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} S_{2N}^{(4n+1)(k+1)} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} S_{2N}^{(4n+3)(k+1)} \tag{4.13}$$

(There is also a symmetry here for $n \geq \frac{N}{2}$ that can and must be exploited for a practical implementation, but we will not bother with the details here, for reasons that will soon be apparent; see [34] for the details.)

Applying the half-shift property (4.6) twice (first directly on the first term, and then in reverse on the second term), we get

$$X_k = \frac{1}{2C_{2N}^{(k+1)}} \sum_{n=0}^{\frac{N}{2}-1} (x_{2n} + x_{2n+2}) S_N^{(2n+1)(k+1)} + 2C_{2N}^{(k+1)} \sum_{n=0}^{\frac{N}{2}-1} (x_{2n-1} + x_{2n+1}) S_N^{(2n+1)(k+1)} \tag{4.14}$$

This is the weighted sum of two $\frac{N}{2}$-point type-II DSTs, which is what we wanted. However, it is not particularly suitable for implementation on a GPU, due to the half-shifts; since simple arithmetic is much cheaper on a GPU than texture lookups, the extra x_n terms will cause an undesirable slowdown.

It is possible to collapse all the half-shifting into $\log_2 n$ addition passes before the actual butterflies take place, as shown in figure 4.3.2 (based on [34]). This reduces the work needed in each pass of the algorithm, but it also means that the number of passes is nearly doubled. In such a scheme, much of the data, especially in the later addition passes, is left untouched; for an in-place transform (such as done on a CPU) this is of course no problem, but a GPU would end up doing much needless copying. Clearly, neither of the implementations of this half-shift seem very attractive.

It should be noted that different DST types vary in their need for half-shifts; if one needs a particular, different DST type, revisiting this approach might provide more fruitful results.

4.4 Data pre-processing

Many textbooks and articles mention that it is possible to re-express a real N-point DST as a real N-point DFT, thus facilitating direct reuse of existing FFT implementations (with some pre- and/or post-processing). However, few give an explicit formula, and even fewer give a formula for *complex* DSTs — indeed, many textbooks, such as [27], define the DST as only taking real inputs. (This is probably because a DST of purely real inputs gives purely real outputs. The DST is equally well defined for complex inputs and outputs, though, as the transform is

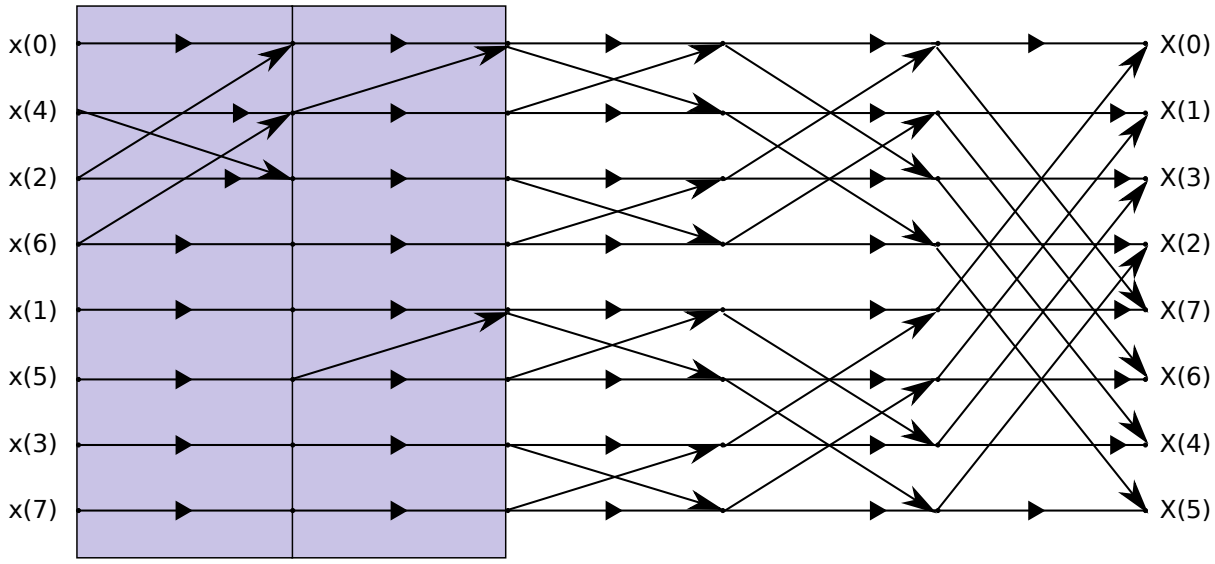

Figure 4.1: The flow diagram for a real 8-point DST, as described in [34] (multiplication constants omitted). The first two addition-only passes are highlighted in blue; the last three passes show the characteristic butterfly structure already known from the FFT.

linear.) Our derivation is based on comments found in the FFTW3[10] source code, which in turn contains a reference to [22].

We will start with the *inverse* of the type-II DST, which is the type-III DST; the forward transform can be obtained by reversing the derived algorithm. The type-III DST is defined as:

$$X_k = \frac{(-1)^k}{2} x_{N-1} + \sum_{n=0}^{N-2} x_n \sin\left(\frac{\pi}{N}(n+1)\left(k+\frac{1}{2}\right)\right) \tag{4.15}$$

The type-III DST corresponds to the boundary conditions that x_n is odd around $n = -1$ and *even* around $n = N - 1$. From this follows that the type-III DST is (up to a constant factor) equivalent with a DFT of the following extended sequence, which is of length $4N$:

$$x'_n = \begin{cases} 0 & ; n = 0 \\ x_{n-1} & ; 0 < n \leq N \\ x_{2N-1-n} & ; N < n \leq 2N-1 \\ 0 & ; n = 2N \\ -x_{n-2N-1} & ; 2N < n \leq 3N \\ -n_{4N-1-n} & ; 3N < n \leq 4N-1 \end{cases} \tag{4.16}$$

A DFT of this sequence will contain exactly the elements of the DST-III, up to a scale factor of $-4i$ (depending on normalization), plus $2N$ elements that are zero and N elements that are simply the antisymmetric extension of the elements we are after. (A formal proof of this will not be given, but it is not difficult to verify numerically.)

Thus, we begin with the DFT of our extended sequence $\{x'_n\}$:

$$X'_k = \sum_{n=0}^{4N-1} x'_n e^{-i\frac{2\pi}{4N}kn} \tag{4.17}$$

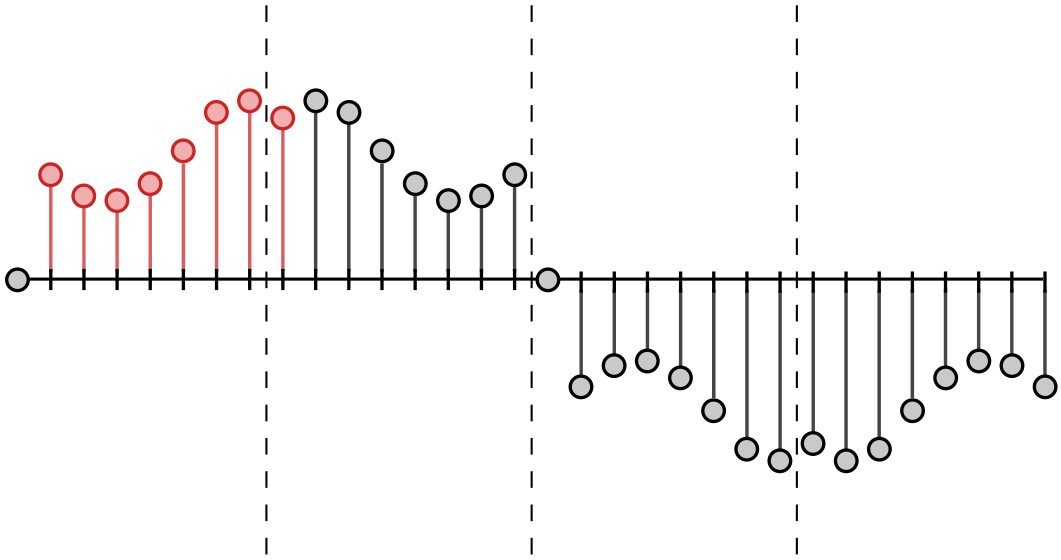

Figure 4.2: The logical extension of an 8-point type-III DST into a 32-point DFT, with the original sequence highlighted in red. The dashed lines show the separation of the 32 points into four parts, corresponding to the four parts of the radix-4 DIF step.

We then apply a *radix-4 decimation in frequency* step, also known as one iteration of the Sande-Tukey algorithm:

$$X'_{4k_1+k_2} = \sum_{n=0}^{N-1} \left(e^{-i\frac{2\pi}{4N}nk_2} \sum_{m=0}^{3} x'_{Nm+n} e^{-i\frac{2\pi}{4}mk_2} \right) e^{-i\frac{2\pi}{N}nk_1} \tag{4.18}$$

Here, k_1 ranges from 0 to $N-1$ and k_2 ranges from 0 to 3. For $k_2 = 0$, the inner sum becomes[1]

$$\begin{aligned} \sum_{m=0}^{3} x'_{Nm+n} &= x'_n + x'_{n+N} + x'_{n+2N} + x'_{n+3N} && (4.19)\\ &= x_{n-1} + x_{N-n-1} + (-x_{n-1}) + (-x_{N-n-1}) && (4.20)\\ &= 0 && (4.21) \end{aligned}$$

Analogously, for $k_2 = 2$, we have:

$$\begin{aligned} \sum_{m=0}^{3} x'_{Nm+n} e^{-i\pi m} &= x'_n - x'_{n+N} + x'_{n+2N} - x'_{n+3N} && (4.22)\\ &= x_{n-1} - x_{N-n-1} + (-x_{n-1}) - (-x_{N-n-1}) && (4.23)\\ &= 0 && (4.24) \end{aligned}$$

This is consistent with the expectation that half of the elements are to be identical to zero.

[1] The intermediate calculations are not correct for $n = 0$, but the end result is zero nevertheless. This also holds for the case of $k_2 = 2$ later in this section.

For $k_2 = 1$, however, we get a slightly more interesting result. For $n = 0$:

$$\sum_{m=0}^{3} x'_{Nm} e^{-i\frac{\pi}{2}m} = x'_n - ix'_{n+N} - x'_{n+2N} + ix'_{n+3N} \quad (4.25)$$

$$= 0 - ix_{N-1} - 0 + i(-x_{N-1}) \quad (4.26)$$

$$= -2ix_{N-1} \quad (4.27)$$

and for $n > 0$:

$$\sum_{m=0}^{3} x'_{Nm+n} e^{-i\frac{\pi}{2}m} = x'_n - ix'_{n+N} - x'_{n+2N} + ix'_{n+3N} \quad (4.28)$$

$$= x_{n-1} - ix_{N-n-1} - (-x_{n-1}) + i(-x_{N-n-1}) \quad (4.29)$$

$$= 2x_{n-1} - 2ix_{N-n-1} \quad (4.30)$$

For $k_2 = 3$, we get exactly the same derivation, except that the factor for the mirrored terms are $+2i$ instead of $-2i$ (which would correspond to complex conjugation if we had real inputs). However, we are not particularly interested in the elements for $k_2 = 3$; as mentioned earlier, the second half of the elements consist of the antisymmetric extension of the first half, which means that the sequence for $k_2 = 1$ *contains all the elements of the DST*, only somewhat reorganized and sometimes negated.

This is a very important point; we can now consider only the elements that correspond to $k_2 = 1$ (let us call this sequence Y), and (4.18) becomes

$$Y_{k_1} = X'_{4k_1+1} = \sum_{n=0}^{N-1} \left(e^{-i\frac{2\pi}{4N}n} y_n \right) e^{-i\frac{2\pi}{N}nk_1} \quad (4.31)$$

where y_n comes from the results of (4.25) and (4.28):

$$y_n = \begin{cases} -2ix_{N-1} & n = 0 \\ 2x_{n-1} - 2ix_{N-n-1} & \text{otherwise} \end{cases} \quad (4.32)$$

But (4.31) is simply the N-point DFT of the sequence $\{e^{-i\frac{2\pi}{4N}n} y_n\}$, which is what we set out to do; via a simple $O(n)$ transformation we have transformed our N-point DFT into an N-point DST.[2] Furthermore, the given transformation has both good numerical properties *and* is suitable for GPU implementation (every term of y_n depends on only two terms from x_n, and the reorganization of the output is a simple shuffle and negation of half the elements).

It should be noted that this is not the only way to do such a transformation; in particular, the given y_n is complex even for real x_n, which means that if the input data had been real, a different algorithm would probably have been preferred (which would enable usage of a real-valued FFT, giving a further speedup of about factor two). If needed, algorithms transforming real input into real output are presented in [32, 8]; in FFTW3, a similar algorithm (plus an additional trick to avoid the use of trigonometric functions) is utilized for its DCT and DST implementations, although complex DSTs and DCTs are not supported.

As previously noted, what has just been derived was an algorithm for the *inverse* transform (nevertheless using a forward DFT to achieve our goal). However, given these steps, the forward transform is trivial:

[2] Usually, a DFT of a sequence multiplied by such an exponential could be rewritten to a DFT of a shifted version of the input sequence. However, in this case we would have to shift by $\frac{1}{4}$, which is not an integer, and thus this property will not help us this time.

- Reverse-shuffle the input data, putting first the even elements (X_0, X_2, X_4 etc...) and then the negative of the reversed, odd elements ($-X_{N-1}$, $-X_{N-3}$, $-X_{N-5}$ etc...), giving Y_n.
- Perform an IDFT (usually realized via the IFFT).
- Multiply each element of the resulting sequence by $e^{i\frac{\pi}{2N}n}$, giving y_n.
- Convert y_n back to x_n by reversing the formula in (4.32) (given below).

The procedure for finding x_n from y_n is:

$$x_n = \begin{cases} \frac{1}{2}iy_0 & n = N-1 \\ \frac{1}{4}(y_{n+1} + iy_{N-n-1}) & \text{otherwise} \end{cases} \tag{4.33}$$

which we observe has just the same numerical properties and ease of GPU-implementation as (4.32).

This completes the description of the pre-processing method. It is readily seen that of the three methods discussed, this method would be the preferred one for GPU implementation; although the pre- and post-processing incurs a non-negligible cost, the implementation complexity is moderate and the performance is good.

Now that all the background material is in place, we will turn to considering the implementation as a whole. It will be seen that even with the basic algorithms in place, there are several choices left to be made for an actual implementation; while many of these are mainly interesting from a performance point of view, some are also important for proper accuracy and operation, and as such must be considered carefully.

Chapter 5

Implementation

5.1 Programming environment

For this project, the C++ programming language was used along with the OpenGL graphics API; both are common choices in the GPGPU community. Similarly, the OpenGL Shading Language (GLSL) was chosen for writing the shaders themselves, as it is relatively simple to use, enjoys high performance and is supported relatively well by all major vendors. Apart from this, no external libraries were used.

GPUwave runs on both Microsoft Windows and GNU/Linux, and should not be difficult to port to other operating systems with adequate driver support (such as Mac OS X); however, the main development was done on GNU/Linux.

On the GPU side, multiple graphics cards were used in testing, attempting to cover most of the major ranges from both ATI and nVidia. This was mainly in the interest of ensuring correct operation and verifying accuracy; the performance results are not intended as realistic benchmarks, especially considering the lack of access to otherwise identical systems with different GPUs.

5.2 Implementation overview

As mentioned earlier, GPUwave runs multiple simulations in parallel, with the single varying parameter being f_0. As will be seen from the benchmark results, there is a considerable speedup over a CPU implementation even for the simulation of a single frequency — however, in many situations, simulations for a range of frequencies are needed, and as such there are two different forms of parallelism present, both the one inherent in each step and the parallelism from running several simulations at the same time.

Apart from the generation of the starting fields (one for each f_0) and similar initialization, the entire program consists of running iterations of the split-step Fourier algorithm over and over again. Essentially, the algorithm consists of five steps:

- Perform a DST on the given input vector.
- Multiply by the e^B (diffraction) term, as given by (3.9).
- Perform an IDST.
- Multiply by the e^A (refraction) term, as given by (3.8).

- Select an appropriate piece of the resulting vector, copy it to more permanent storage (that is, a storage texture — data is not read back to the CPU until at the very end of the algorithm) and display it on-screen.
- Use the output vector from this iteration as input for the next iteration, and repeat the steps above as many times as is required. (This step is skipped when benchmarking the algorithm.)

However, as the DST was to be implemented by means of the FFT, the actual implementation was somewhat more convoluted:

- Shuffle the input row appropriately, as described in section 4.4.
- Perform an inverse FFT.
- Multiply by $e^{i\frac{\pi}{2N}n}$.
- Unravel the data as described in (4.33).
- Multiply by the e^B (diffraction) term, as given by (3.9).
- Ravel (that is, "un-unravel") the data as described in (4.32).
- Multiply by $e^{-i\frac{\pi}{2N}n}$.
- Perform a forward FFT.
- Unshuffle the output row.
- Multiply by the e^A (refraction) term, as given by (3.8).
- Select an appropriate piece of the resulting vector, copy it to more permanent storage and display it on-screen.
- Use the output vector from the current iteration as input for the next iteration, and repeat the steps above as many times as is required.

The observant reader might have noticed that the first four steps do not strictly give the DST-II; the result will be scaled by a $-4i$ term, as described in section 4.4. However, the dispersion step is invariant to such a term, as all it does is modify the phase of the given vector. Thus, as the DST-III reverses this term, it needs no special consideration.

Also, as GPUwave does not perform a normalized FFT, the result of an FFT+IFFT will invariably be scaled by N; this scaling was reversed in the diffraction pass, as this minimized the energy range and thus over-/underflow problems. Scaling at more than one point during each iteration was also attempted, but yielded no significant benefits.

Each of the passes would usually correspond to one pass in the OpenGL implementation, except the FFT and IFFTs, which would take $\log_2 N$ passes each. However, in the interest of efficiency, a few of the passes were collapsed; specifically, the unravel/ravel and the exponential passes were merged. This saves on the number of passes (and thus the amount of data being copied around), at the expense of twice the amount of complex multiplications in the unravel pass. If needed, even more passes (for instance, the unshuffle and refraction passes, or the first

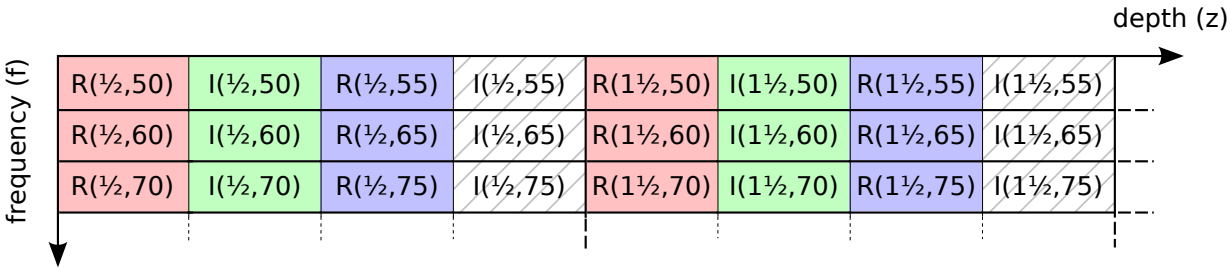

Figure 5.1: The memory layout for the intermediate storage textures, showing the packing of the function $\psi(z, f_0) = R(z, f_0) + iI(z, f_0)$ (r is constant within each such frame). Note that there is room for two such complex values in each RGBA fragment.

pass of the IFFT and the shuffle pass) could have been collapsed, but the gains are somewhat limited, and this option was not pursued further.

The number of passes will be dependent on the resolution of the discretization of z (which influences the length of the FFT); a typical total number is 20-25 passes per iteration.

5.3 Data packing

As previously mentioned, GPUs generally work on four-component floating-point vectors. Almost all quantities in the given algorithm are complex, and would only need two components (for the real and imaginary parts, respectively); thus, data from two parallel computations could generally be packed into one fragment. This was significantly faster than letting the other half stay empty, as the number of texture fetches and writes were all but halved, and four-component vectors could generally be processed at no extra cost over two-component vectors. There was some increase in code complexity due to the issues of dealing with two different sets of data at the same time, but all in all, this was a very worthwhile trade-off, and it increased the end performance by about 70% in practice[1].

Similarly, in other textures, otherwise unrelated data was packed together wherever it would make sense; for instance, in the FFT support texture, the two source pointers were packed in the same fragment as the complex twiddle factor.

5.4 Implementation of the FFT

Most of the running time of the split-step Fourier algorithm is spent calculating the FFT — in fact, passes one would believe to be very expensive (such as the refraction pass, which also computes $\tilde{n}^2$ as described in section 3.2.4) were only slightly more expensive than each of the FFT passes. As about 75% of all passes (somewhat dependent on the transform length) are FFT passes, this makes FFT is the single most performance-critical section of the algorithm.

Finding optimal or near-optimal FFT implementations, even on the CPU, is a very complex problem[10], and even though the programming model on the GPU is in some respects more

[1] All performance numbers presented in this section are approximate and should be taken with a grain of salt for two reasons; first, they have not been meticulously benchmarked, and second, they might vary between different GPUs.

constrained, there is no reason to believe the performance characteristics to be less convoluted.

A detailed analysis of FFT performance (with regard to different radices, decimation-in-time versus decimation-in-frequency, different memory layouts, etc.) in the context of the GPU is beyond the scope of this project; thus, only the simplest, classic case, that is, a radix-2 decimation-in-time FFT, was considered. The reader is assumed to already have a basic understanding of the FFT; for an introduction, see [28].

All calculations were done in a fragment shader, the procedure being identical between the FFT and IFFT except for the twiddle factors. As a lot of fragments share the same twiddle factor, a scheme using multiple quads (the number of quads varying with the FFT size) for each pass was tested, using a constant (pre-calculated) twiddle factor over each quad.[2] However, it was later discovered that a scheme with a single quad per pass, looking up into a 1D texture storing both the two needed source indices and the twiddle factor was somewhat faster; the possible cause of this is that most GPUs will perform suboptimally when rendering quads that are very thin, as they will normally have to calculate one extra fragment along each axis (due to the need for calculating derivatives).

Yet more testing revealed that using such a "support" texture was usually only faster for the later passes, where there would be many such quads — for the first few passes, the cost of an indirect texture lookup was greater than what is gained from having having fewer quads. (For the passes in-between, both solutions were about as fast.) Thus, a split scheme was implemented, in where the first passes used multiple quads, interpolating source coordinates over the quads, and the later passes used an indirect texture lookup. The specifics of this optimization seem rather hardware-specific as well as dependent on several other factors (such as frame buffer precision or the size of the data set), so the cutoff point was made adjustable in order to facilitate tuning for different hardware — during benchmarking, all possible crossover points were tried and the one giving best performance was chosen. For some hardware at some working sets, this meant never using a support texture; for other hardware and other working sets it meant *always* using a support texture. In mos cases, both strategies ended up being used to some degree, and in some cases the choice did not matter measurably.

The FFT in GPUwave was implemented as a classic breadth-first traversal of the FFT computation tree; first all the sub-FFTs of length 2 were done in one pass, then the sub-FFTs of length 4, then 8, etc.. It was decided on using a rather simple memory layout, requiring no reordering of the inputs nor outputs, and shown in figure 5.2.

Also, a more traditional FFT scheme, with a diagram more akin to that of figure 4.3.2, was implemented (reordering the elements at the beginning of the IFFT and at the end of the FFT), but preliminary testing of the algorithm as a whole (that is, including the DST reordering, diffraction and refraction) showed that markedly worse overall performance (about 15% for a 256-point FFT) than when using the previously mentioned scheme, so all further testing was done using the in-order memory layout. It should be noted that without the reordering (assuming either the diffraction or refraction shaders could somehow work with the shuffled output), the scheme was found to be marginally better than the in-order scheme (giving overall

[2] A word of caution here: Since one will have to send in the twiddle factor as a vertex parameter when not using a texture, it will be subject to interpolation like any other vertex parameter. The precision of this interpolation as of today's GLSL specification is undefined and might be limited, even though the value is constant over the entire quad. GPUwave will automatically detect if this precision is markedly worse than the precision for values from textures, and if so force using a texture for all passes where precision in the twiddle factors might be an issue, even though this means taking a performance hit. That being said, testing has still not found any cards which actually exhibit this behavior for values as small as in this case.

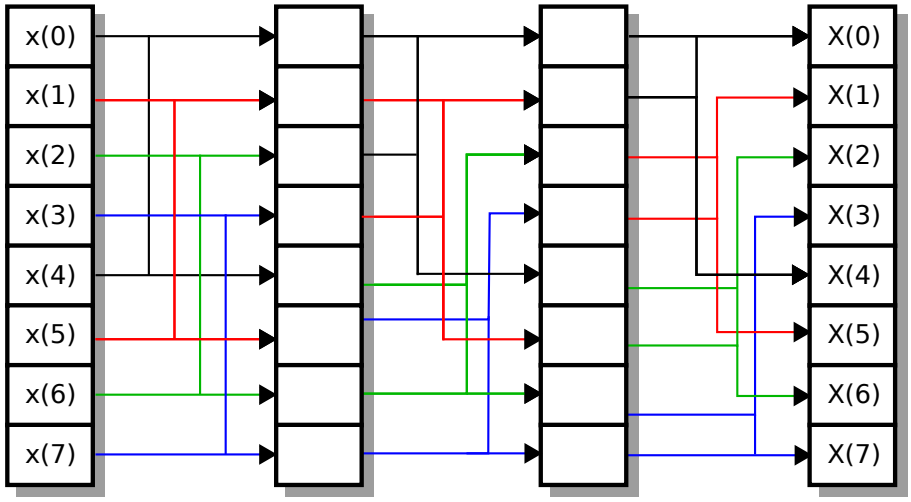

Figure 5.2: The memory layout and flow diagram for one 8-point in-order FFT, as implemented in the fragment programs. In the actual implementation, many such FFTs, all with the same memory layout, were done in parallel.

2-3% speedup, again for a 256-point FFT), so the cost of the reordering was most likely the primary disadvantage.

While experiments showed that it was easy to make these passes perform very *poorly* (for instance, by reversing the bits of the indexes in the outputs of one pass and inputs of the next one, yielding a very irregular memory access pattern), it seemed harder to change the memory layout to get a noticeable performance gain over the used scheme. Again, more research would be beneficial here.

5.4.1 Multiple render targets

As previously mentioned, some OpenGL implementations allow outputting more than one value from a shader, storing them in two or more output textures. (How many textures will vary, but most modern cards with this feature support at least four.) This usage of *multiple render targets* (MRT) can be useful in some cases, in particular those where two or more output fragments share many common calculations or texture fetches.

Not all algorithms adapt well to this kind of optimization — in particular, it is not always beneficial to have the data spread among two or more different textures when it is to be processed further. The most likely target for this kind of optimization seemed to be the FFT; in some sense, it was an ideal match, as the butterfly of the FFT outputs two values using the same inputs ($d_1 + Cd_2$ and $d_1 - Cd_2$ where d_1 and d_2 are the inputs and C is the twiddle factor). Similarly, the chosen memory layout seemed well-suited to an MRT implementation, as the stride between the two outputs of the butterfly was constant, always equal to half the size of the texture.

However, a few complications were encountered; since each pass outputted its output in two halves, the next pass also had to be prepared to take its *input* in two textures. With the given memory layout, both inputs would always come from the same texture, which meant that the workload could be split into two different batches using only one texture each; the

alternative would be introducing a a (possibly expensive) branch in the fragment shader in order to select which texture to sample from. On the other hand, this introduced more complexity into the implementation, and considering the fact that the first and last passes would need special handing (since they have only one input and one output, respectively) together with already present complexity in the FFT implementation (in particular with regard to quads versus support textures), it was clear that a MRT-based implementation of the FFT would be significantly more complex than a non-MRT-based implementation.

A full MRT-based FFT was not implemented; however, informal testing showed that a simple quad-based (that is, with no support texture) implementation of the FFT could gain anywhere between 15-60% over the existing non-MRT FFT. However, the gain varied significantly between GPUs, which suggested either a gap in the efficiency of the MRT implementations between manufacturers, or that the existing implementation hit different bottlenecks on different cards.

Finding the optimal balance for a GPU-based FFT in the presence of MRT/SRT, different radices (radix 4 seems significantly more attractive in the presence of MRT than without), quad-based implementations vs. ones using an indirect texture lookup, different memory layouts, and combinations of these for different passes and sizes is expected to be a very complex problem, and this problem was not pursued further here.

Further work in this area, or even a standard set of GPU routines for FFT, might prove useful — a possible starting point could be [17], in which a GPU-friendly reordering of the FFT calculations with multiple passes and multiple outputs is described. Also, [26] mentions performance gains for the FFT by by-passing the conventional graphics APIs entirely, possibly opening up for entirely different optimization strategies.

5.5 Discrete sine transform

As mentioned in chapter 4, eventually the DST was implemented by converting the input data set into a form that allowed using the standard FFT algorithm. Comparing this method to the most realistic alternative, doubling the transform size, yielded performance gains of about 70%; clearly, it was the best option for a GPU-based transform, even though the pre- and post-processing does not come for free. (One could also flip the coin, taking these results as a testament to the overall efficiency of the FFT; the algorithm is indeed remarkably efficient for what it does.)

Most of this slowdown in the pre- and post-processing came from the ravel/unravel passes; while a detailed analysis of these issues was not performed, it was believed that the primary slowdown stemmed from both passes reading the entire input texture twice, and as such, the slowdown would be less important for the total running time as the transform size (and thus the number of FFT passes) increased.

Also, given that the shuffle and unshuffle passes are inverses of each other, it is possible to make the refraction shader work on shuffled inputs and drop the unshuffling and reshuffling entirely; however, this would have to be undone in the display shader, and the gains are somewhat questionable: Removing the shuffle/unshuffle passes gained a mere 5-6%, and for the refraction shader to handle shuffled inputs, it would need more logic (most likely implemented using an extra indirect texture lookup). Also, the branch change rate would go from very slow to very fast (causing a possible performance hit on some architectures, as previously noted), which was seen as not very desirable. Thus, the DST implementation was kept as-is, with the explicit unshuffling and reshuffling.

5.6 Choice of rendering primitive

Some authors, such as [20], recommend using an oversized triangle instead of a full-screen quad whenever possible, since modern hardware can render this as a single primitive, whereas the quad would need to be split into two different triangles. (Older hardware would usually clip the triangle back into a quad, rendering the optimization moot.) Using a single primitive is then said to enjoy a certain performance advantage, as fewer edge fragments (which are slightly more expensive than other fragments) would have to be rendered — though somewhat dependent on the card in question. (It should also be noted that some hardware have a special primitive for screen-aligned quads, but this opportunity was not considered.)

However, as the algorithm is described, only a few of the passes can actually use this optimization (since not all actually render a single full-screen quad), and we were unable to measure any such performance gain. Thus, again in the interest of simplicity, this change was discarded, and quads are used as the primary primitive in all passes.

5.7 Numerical accuracy

As previously mentioned, the fragment shaders in GPUwave generally work on 32-bit IEEE floating point values (or at least, quantities closely resembling them); support for 64-bit floating point numbers (that is, double-precision floats) would certainly be useful for some classes of GPGPU work, but at the present time, no GPUs support this (although there has been some work in circumventing this limitation[23, 11]).[3]

However, all 32-bit capable hardware is also capable (at least to some extent) of supporting *16-bit* IEEE floating point numbers (again with the usual caveat of correctness in edge cases such as denormals and overflow), also called the "half" format. With only ten mantissa bits, five exponent bits and a sign bit, this format is pushing the limits of what is considered traditionally useful for scientific work; however, it should not immediately be discounted, as it has attractive aspects, also outside computer graphics.

First, and perhaps a bit surprisingly, some GPUs support more features (such as automatic bilinear filtering) when using the half format than when using the full 32-bit format, possibly enabling useful speedups. However, none of these features were used in GPUwave, so this was not applicable — also, this difference in functionality is gradually diminishing as GPUs mature.

However, using the half format can often yield performance gains, both because of reduced memory bandwidth demands, increased cache efficiency and, on some GPU series, faster internal calculations. The GLSL standard did not directly expose such reduced-precision numbers; some GPUs expose such functionality through extensions, but in the interest of greater compatibility, this option was not pursued.[4] However, using the half format for *textures* is still possible; this makes for lower bandwidth demands, and in some cases, it gives the compiler freedom to use reduced precision for some or all of the computations.

In GPUwave, each pass used at least one 1D support texture — some used two. In addition, there were two 2D textures used for intermediate storage between the passes. Testing showed

[3] On a historical note, the very first revisions of GPGPU-capable hardware pretended to operate on 32-bit values, but actually had only 24 bits of precision; see chapter 6 for a discussion on the effects of using reduced precision.

[4] ESSL (a variant of GLSL intended for embedded systems using OpenGL ES) support multiple precisions in the core language; however, the semantics of such mixed-precision work is somewhat ill-defined, in particular with regard to type promotion between these types.

that only three textures of these 20-25 actually benefited performance-wise from being stored in the half format instead of the usual float format:

- Naturally, converting the two intermediate 2D textures would yield a sizable benefit, as they were by far the largest single source of memory bandwidth usage, being used at least once in every single pass. However, reducing the intermediate precision of even a single pass is not something to be taken lightly; as all the passes are serial, errors should be expected to grow at least linearly with ϵ (a common measure of floating-point precision), which is $2^{24-11} = 2^{13}$ times as large in the half format as when using float.

- Also, converting the support texture for the FFT/IFFT passes (containing source coordinates and twiddle factors) yielded a notable performance benefit, most likely since it is used in so many of the passes. Unfortunately, the accuracy of the FFT is very sensitive to reduced accuracy in the twiddle factors[30], so this was also not a viable option.

One such optimization could be safely used: In the innermost two FFT/IFFT passes, the twiddle factors are all either 1, -1, i or -i, which could all be converted to the half format with no precision loss. However, as previously described, the first passes would usually not utilize a texture for storing twiddle factors at all, so this optimization ended up being of little benefit in practice.

Converting the remaining (ie. non-FFT-related) textures gives yielded only a minimal speedup, about 2-3%. This, it can be concluded that reducing floating-point precision was not especially beneficial for increasing performance in this particular case, and 32-bit floating point were used for nearly all calculations.

5.8 Summary

As seen, it is indeed possible to implement the Fourier-step algorithm efficiently on the GPU; however, it calls for somewhat constrained data structuring and careful implementation of each individual step. Furthermore, there should still be be room for additional optimization, even though considerable effort has already been put down in creating an efficient implementation.

GPUwave is available under a free license, both in electronic form and in this paper; see appendix A for code listings and a URL for a machine-readable version.

Chapter 6

Results

Like with any numerical solution to a given problem, there are two main properties of the implementation that will need to be assessed, namely *performance* and *accuracy*. As the primary point of interest would be seeing if the GPU approach is worthwhile or not compared to existing CPU models, it was chosen to compare the relative qualities of GPUwave with those of existing implementations.

For the performance reviews, GPUwave was compared against *PEEC*, a MATLAB-based implementation of the Fourier split-step method as described in [25]. This was a natural choice, as the model implemented in GPUwave matched PEEC's model almost exactly; in fact, the sole difference was that PEEC supported a variable sound speed profile and GPUwave did not. This made it easier to compare "apples to apples", as one could compare two implementations doing the same calculations without having to consider differences stemming from different methods.

However, a comparison with PEEC would be less useful when reviewing the implementation's *accuracy*; even though the Fourier-step method is well known, neither it nor PEEC are particularly widely used as standard references. (PEEC was, however, useful as a reference during debugging, even though some details, such as the discretization chosen, were different.) Thus, GPUwave was instead compared to a more well-known reference implementation for this kind of problem, namely *Range Acoustic Model (RAM)*[4].

6.1 Performance

The relative performance of the GPU and the CPU models is an interesting point; after all, increasing computational efficiency over CPU implementations was the primary motivation for considering a GPU implementation in the first place. However, actually measuring the relative performance of the CPU and the GPU is somewhat involved, so a brief discussion of test methodology will be required.

6.1.1 Measuring CPU performance

As previously mentioned, PEEC was used as the base point of the benchmarks, running on an AMD Athlon 64 3000+ (at 1.8GHz), a mid-range CPU. (It is interesting to note that this CPU has a comparable street price to that of an ATI Radeon X1600 XT, which served as the primary development card for the GPU implementation.) There are a few points that are worth noting about this choice:

	1	2	4	8	16	32	64	128	256	512
Athlon 64 3200+	9.588	—	—	—	—	—	—	—	—	—
GeForce 6200	—	0.408	0.513	0.909	1.846	3.640	7.261	14.420	29.001	57.190
GeForce 6600	—	0.300	0.319	0.530	1.112	2.865	5.517	10.862	21.408	42.592
Radeon X300[1]	—	0.399	0.733	1.572	1.298	2.539	4.927	9.631	19.197	38.170
GeForce 6600GT	—	0.317	0.320	0.319	0.509	1.127	2.086	4.093	8.071	15.997
Radeon 9600 Pro	—	0.835	0.833	0.834	0.829	1.133	2.120	4.085	7.931	15.834
GeForce 6800	—	0.345	0.344	0.346	0.439	1.090	1.996	3.669	7.268	14.438
GeForce Go 7600	—	0.956	0.950	0.953	0.956	1.010	1.731	3.343	6.531	12.919
GeForce 7600GS	—	0.311	0.304	0.297	0.412	0.635	1.123	2.150	4.195	8.355
Radeon X1600 XT	—	0.538	0.537	0.537	0.540	0.622	1.048	1.948	3.767	7.440
GeForce 7600GT	—	0.781	0.784	0.787	0.787	0.772	0.907	1.638	3.172	6.328
GeForce 7900GS	—	0.294	0.294	0.294	0.294	0.392	0.648	1.185	2.267	4.425
GeForce 7800GT	—	0.241	0.235	0.234	0.231	0.344	0.587	1.066	2.018	3.975
Radeon X1900 XTX	—	0.444	0.440	0.440	0.439	0.457	0.492	0.822	1.466	2.758

Table 6.1: Run time against number of frequencies calculated, measured on various cards. All values are in seconds.

- PEEC is written in MATLAB, which is known to be somewhat sub-par with regard to efficiency (even though recent versions include a JIT compiler). A comparable port to a compiled language would definitely be faster; however, the primary inner loop of PEEC's calculations is written exclusively in vector notation, which is usually a good match for MATLAB's calculation model. Thus, while this is something that should be kept in mind, it is believed that the main observed trends will still be representative in the general case.

- All calculations in PEEC are performed in double-precision floating point; GPUwave uses single-precision floating point, for reasons previously explained.

- All times quoted for PEEC are measured in CPU time, and include the setup of the $\tilde{n}^2$ matrix (as this will vary from run to run, and takes a considerable amount of the total running time).

For reference, the 64-bit version of MATLAB 2003b, running on GNU/Linux, was used for all testing.

6.1.2 Measuring GPU performance

For the GPU implementation, the time measured was wall time (CPU time would not be a realistic measure, as the CPU could very well be idle while the GPU works), without any kind of display nor permanent storage. The latter matched that of the CPU benchmark — any real use of the data would need to factor in the extra overhead of actually doing anything useful with the end result. Since the GPU is deeply pipelined, care was taken to wait for all commands to finish on the GPU before sampling the clock.

[1]Both R300-based cards (that is, the Radeon 9600 Pro and the X300) were run with a slightly simplified version of the refraction shader, since the correct version triggered bug in the R300 GLSL compiler, causing a crash. This causes the results to be slightly incorrect for half of the frequencies calculated, but it is not believed that it skews the benchmark results measurably in practice.

	1	2	4	8	16	32	64	128	256	512
Athlon 64 3200+	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
GeForce 6200	23.5	47.0	74.8	84.4	83.1	84.3	84.5	85.1	84.6	85.8
GeForce 6600	32.0	64.0	120.0	144.8	138.0	107.1	111.2	113.0	114.7	115.3
Radeon X300	24.0	48.1	52.4	48.8	118.2	120.8	124.6	127.4	127.9	128.6
GeForce 6600GT	30.2	60.4	120.0	240.1	301.3	272.3	294.2	299.9	304.1	306.9
Radeon 9600 Pro	11.5	23.0	46.0	92.0	185.1	270.9	289.4	300.5	309.5	310.0
GeForce 6800	27.8	55.5	111.4	221.4	349.2	281.5	307.4	334.5	337.7	340.0
GeForce Go 7600	10.0	20.1	40.4	80.5	160.4	303.9	354.5	367.1	375.8	380.0
GeForce 7600GS	30.8	61.6	126.3	258.5	372.0	482.9	546.5	570.7	585.1	587.5
Radeon X1600 XT	17.8	35.6	71.4	142.9	284.2	493.2	585.8	630.0	651.6	659.8
GeForce 7600GT	12.3	24.6	48.9	97.4	194.8	397.3	676.6	749.3	773.8	775.8
GeForce 7900GS	32.6	65.2	130.5	260.8	522.1	782.3	947.3	1035.3	1082.7	1109.3
GeForce 7800GT	39.8	79.6	163.5	327.2	663.0	891.9	1044.7	1151.7	1216.1	1234.9
Radeon X1900 XTX	21.6	43.2	87.1	174.2	349.6	671.4	1247.2	1492.7	1674.5	1779.9

Table 6.2: Relative performance against the CPU for various cards, for various numbers of frequencies calculated in parallel.

In the interest of diversity, GPUwave was benchmarked on several different test setups. Again, it must be stressed that the results should not be interpreted as proper benchmarks of these GPUs against each other (most importantly because they have not been tested in equal environments, and much of the optimization was carried out using results from one specific card from one manufacturer); in essence, what was attempted was to benchmark GPUwave by running it on different GPUs, not benchmarking the GPUs by running GPUwave.

As one might infer from the previous chapter, the present implementation can not be assumed to be optimal; while efficient, there are still many areas that can and should be improved for a production-quality implementation. Also, such new optimization work might apply in different degrees to different GPUs.

All this being said, the numbers should be relatively representative for what can be expected from an algorithm in this class, and could provide some general pointers into GPU efficiency, especially given the relative lack of conclusive GPGPU-specific benchmarks available at the time of writing.

6.1.3 Performance results

A fairly simple test case was run through both PEEC and GPUwave — the performance profile is not particularly sensitive to the individual parameters chosen, so a single case was believed to be indicative of general performance. Due to bugs in the R300 driver set preventing rendering to large textures, the test case was not run at the highest possible resolution, but rather a more medium resolution of 512x512 (that is, using 512 range steps and an FST size of 512).

All benchmark runs were sampled five times and then averaged. The results are presented in table 6.1 for various numbers of frequencies being calculated in parallel.[2]

Obtaining performance results on GeForce FX-class (NV3x) hardware was also intended, but since this would require quite invasive changes in GPUwave, final performance numbers could

[2]PEEC does not support calculating data for multiple frequencies, neither sequentially nor in parallel. Thus, all PEEC results for multiple frequencies were extrapolated from the base result. Similarly, GPUwave always calculates at least two frequencies in parallel, so there are no single-frequency measurements for any of the GPUs.

not be obtained. However, preliminary results indicate that FX5200 (arguably the slowest GPU ever shipped capable of running GPUwave) was about 60% slower than the 6200 for this task. Also, the benchmark was run on GeForce 8800 (G80), but due to driver bugs, no meaningful performance results could be obtained at the time of writing.

Reorganizing the results, filling in the single-frequency GPU measurements with data from the double-frequency measurements, dividing by the measured CPU time for a single frequency and multiplying by the number of frequencies calculated gives the relative speedup over the CPU implementation, as as shown in figure 6.2.

All in all, the performance numbers reinforce the notion that the GPU's strongest area lies in exploiting parallelism, with all cards tested performing markedly better when calculating many frequencies in parallel. (It was assumed that with the smallest working sets, the bottleneck was actually getting the CPU to send control data quickly enough to the GPU. As the tests were not run on identical CPUs, this provides some explanation as of why some of the faster cards performed worse than the otherwise slower cards for small data sets.) Remarkably, the fastest GPU outperformed the reference CPU by more than three orders of magnitude for the largest test case; even with room for optimization in both implementations and faster CPUs available, this is a sizable performance gap.

6.2 Accuracy

6.2.1 Comparison against RAM

For accuracy testing, the results from GPUwave were compared to those from Range Acoustic Model (RAM). The two solvers were given similar input conditions, summarized as

- Range: $r = [0, 10]$ km
- Sound speed profile: $c(z) = (1490 + 0.04z)$ m/s
- Source depth: $z_s = 25$ m
- Receiver depth: $z_r = 75$ m
- Sediment layer start: $D(r) = (100 + 0.01r)$ m (that is, ranging from 100 to 200 m)
- Sediment dampening: $\alpha_s = 0.5$ dB/λ
- Sediment sound speed: $c_r = 1700$ m/s
- Sediment density: $\rho_s = 1500$ kg/m^2
- Source frequencies: $f_0 = \{25, 50, 100, 200\}$ Hz

The results for the four different f_0 used are presented in figures 6.1 through 6.4. As is readily seen, the results vary somewhat between the two implementations; more so for higher frequencies. This is most likely because RAM uses somewhat different solution methods compared to GPUwave; specifically, RAM uses a *split-step Padé method*[6, 5], which yields greater accuracy than the split-step Fourier method at comparable efficiency. A more detailed comparison between different algorithms for solving parabolic wave equations is available in [7].

The graphs are not free of systemic bias along either axis. There are two notable explanations for this:

- Loss graphs are by convention measured in dB loss compared to the strength of the field at $r = 1$m. However, neither models are accurate in the near field, so the values close to $r = 0$ will be rather inaccurate. (GPUwave often does not even have a non-zero value at this point; thus, when calculating the reference value, the contribution from $\psi(r, z)$ is ignored, leaving only the contribution from the Hankel function, which is ${H_0}^{(1)}(k_0)$ at that point.)
- The Fourier-step model requires as parameter the speed of sound at the source (the "reference speed of sound"), c_0, which is not always readily available. Different estimates of c_0 will correspond to slightly different scaling of the graph along the r axis; no attempt has been made to find the c_0 that matches optimally with the data from RAM. Also, the choice of Δr influences the graph in similar ways, which means that if Δr is high, it might be beneficial to adjust c_0 accordingly to compensate.

Even with these variations, it is believed that the results are comparable between the implementations, even though the lack of double precision inherently makes any GPU implementation somewhat less accurate than a comparable CPU implementation.

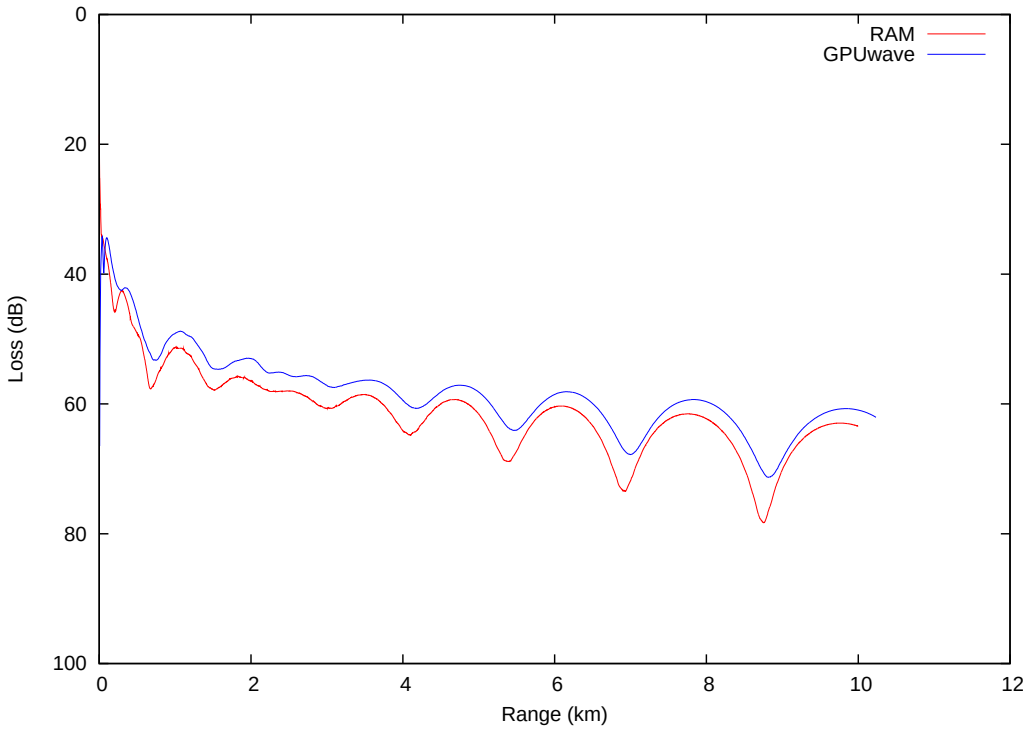

Figure 6.1: Accuracy comparison between GPUwave and RAM at 25Hz.

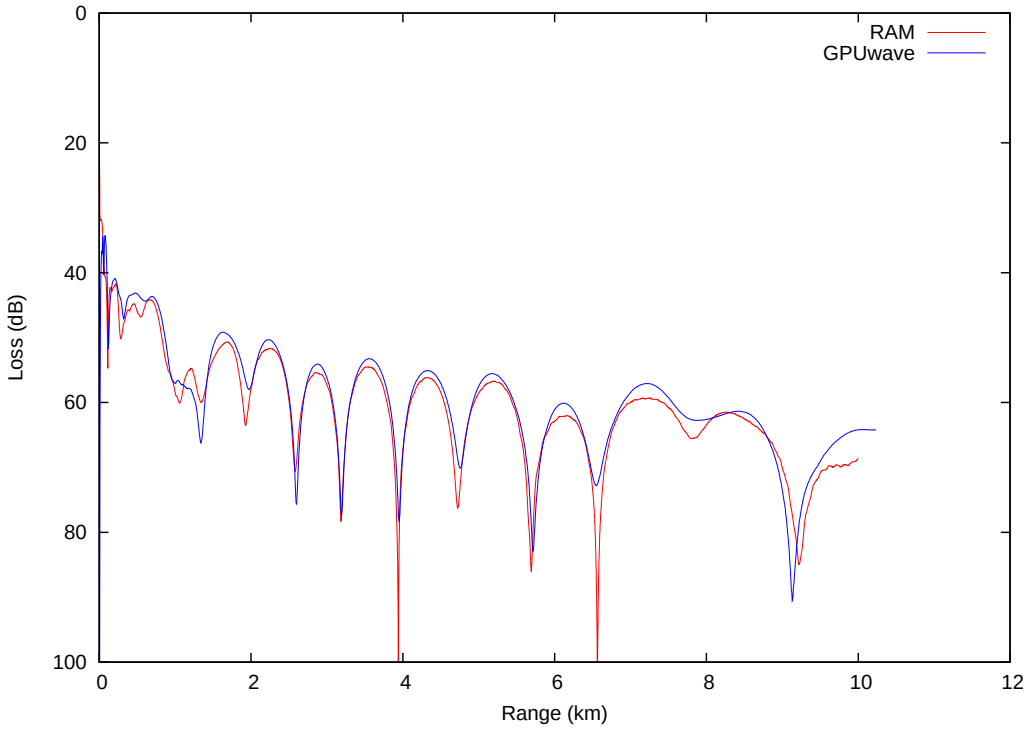

Figure 6.2: Accuracy comparison between GPUwave and RAM at 50Hz.

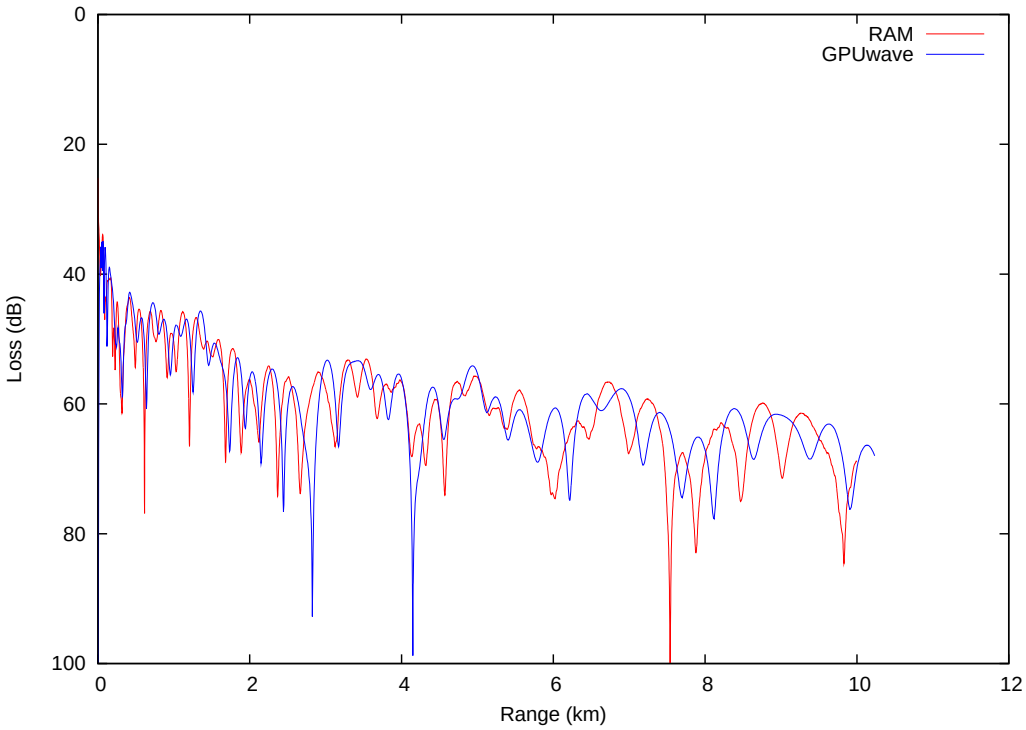

Figure 6.3: Accuracy comparison between GPUwave and RAM at 100Hz.

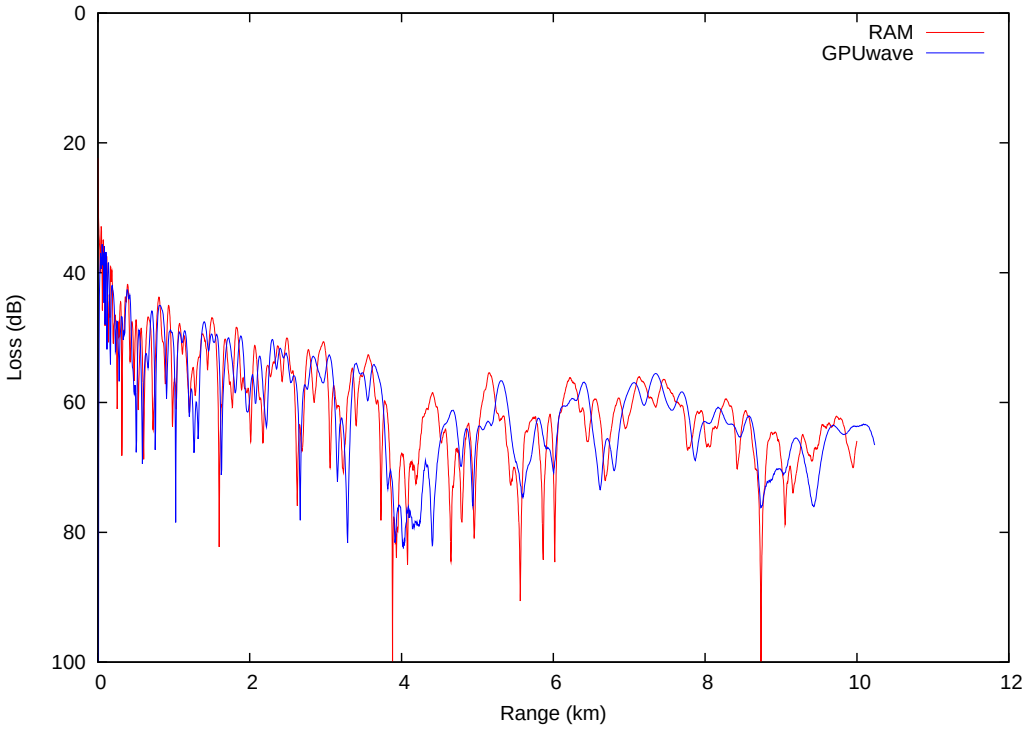

Figure 6.4: Accuracy comparison between GPUwave and RAM at 200Hz.

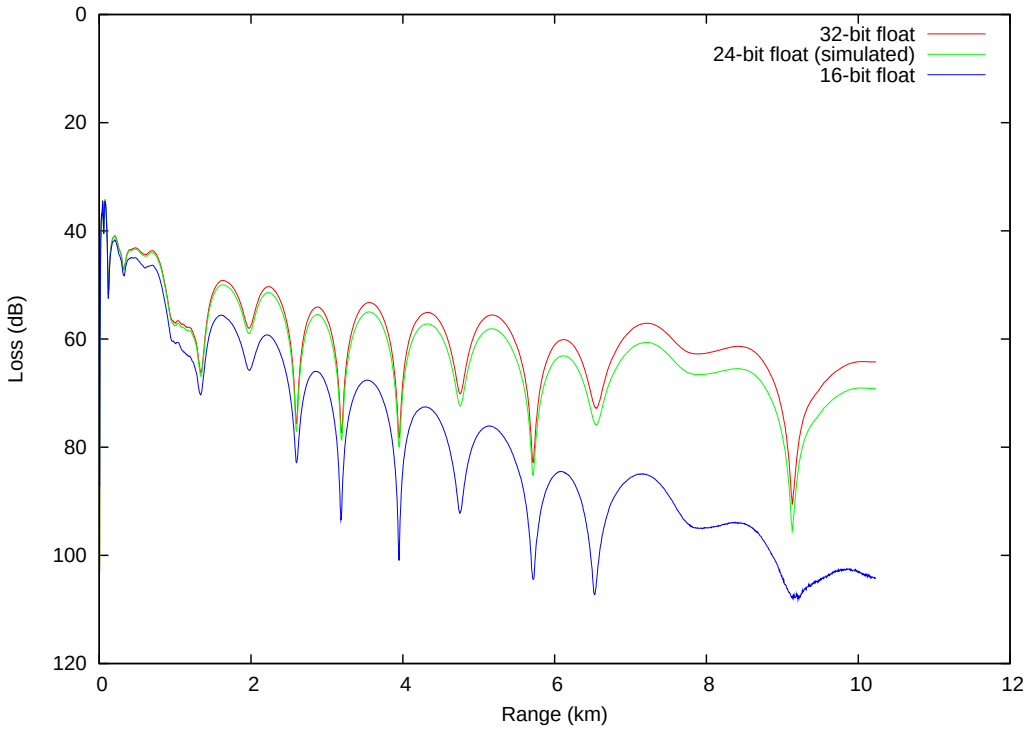

Figure 6.5: Accuracy comparison between different GPU formats, at 50Hz.

6.2.2 Floating-point precision

As it is desirable to study the effects of using reduced floating-point precision, the 50Hz test case was repeated using three different levels of precision on the GPU. For the 16-bit tests, all textures were set to the half format, thus forcing 16-bit intermediate precision at all steps. For the 24-bit tests, the most natural choice would be running the tests on an R300-based card, which does not support full 32-bit floating-point internally — unfortunately, bugs in the R300 driver prevented a full-resolution test on these cards. Instead, a special truncation shader was used, rounding away the lowest eight bits of the mantissa of each value between every pass. This allows simulation of reduced precision even when running in full 32-bit mode — while it does not give bit-for-bit identical results compared to a real R300, lower-resolution tests showed that the simulated and real results were indistinguishable by visual inspection, and as such the simulated values were accepted as a reasonable substitute. The results are shown in figure 6.5.

As can be seen from the graph, in the lower-precision runs energy is gradually lost as the algorithm progresses, dependent on the number of steps; this means that a smaller Δr will not always give a more accurate result, as N (the number of steps) will be larger. This is, in general, a general problem with the parabolic approximation; for instance, in RAM, special energy-conserving measures are employed[4].

Even though the difference between the results from the 32- and 16-bit versions is large at this Δr, the difference between the 32- and 24-bit versions is relatively small. This suggests the 32-bit results are not significantly different from those of a hypothetical 64-bit implementation, which in turn indicates that the lack of 64-bit precision is not a critical point.

6.2.3 Comparison against OASES

Finally, the results of GPUwave were compared to those of *Ocean Acoustics and Seismic Exploration Synthesis* (OASES)[31], another well-known model which uses the technique of *wavenumber integration*[18]. OASES does not support varying $D(r)$, so a different test case was used, summarized as

- Range: $r = [0, 20]$ km
- Sound speed profile: $c(z) = 1500$ m/s
- Source depth: $z_s = 25$ m
- Receiver depth: $z_r = 75$ m
- Sediment layer start: $D(r) = 100$ m
- Sediment dampening: $\alpha_s = 0.5$ dB/λ
- Sediment sound speed: $c_r = 1700$ m/s
- Sediment density: $\rho_s = 1500$ kg/m^2
- Source frequencies: $f_0 = \{25, 50\}$ Hz

The results are shown in figures 6.6 and 6.7. These test cases also show systemic bias; figures 6.8 and 6.9 show results from the same test cases, only corrected for systemic bias (by adjusting c_0 and the estimate of $p(1, z_r)$). After correction, the results are clearly comparable.

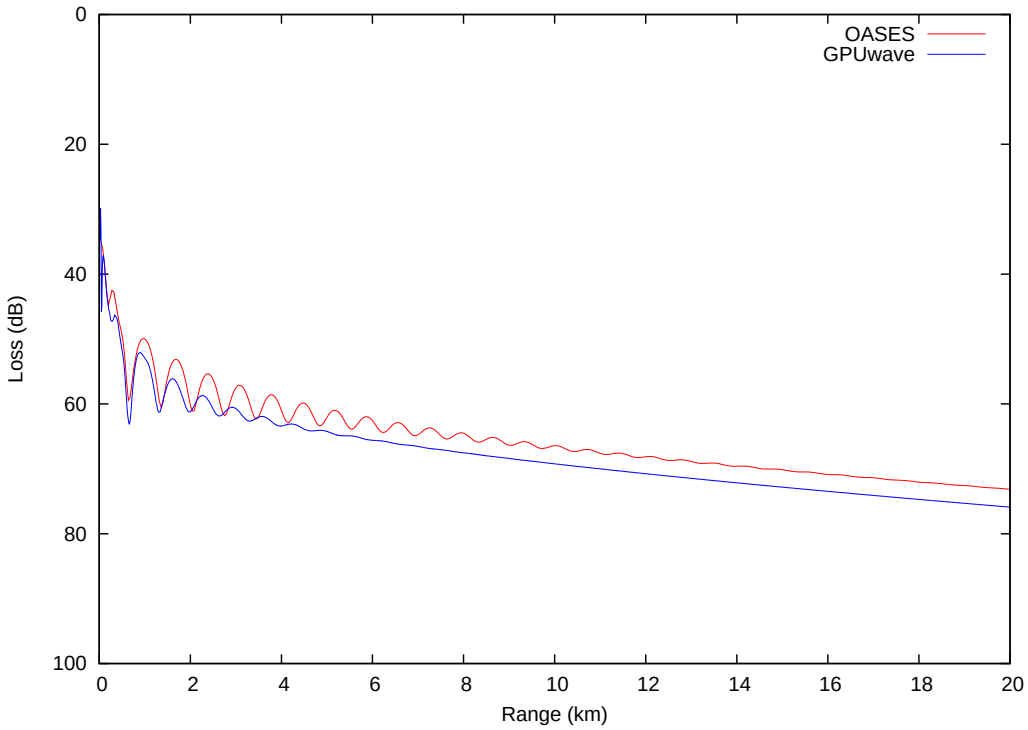

Figure 6.6: Accuracy comparison between GPUwave and OASES at 25Hz.

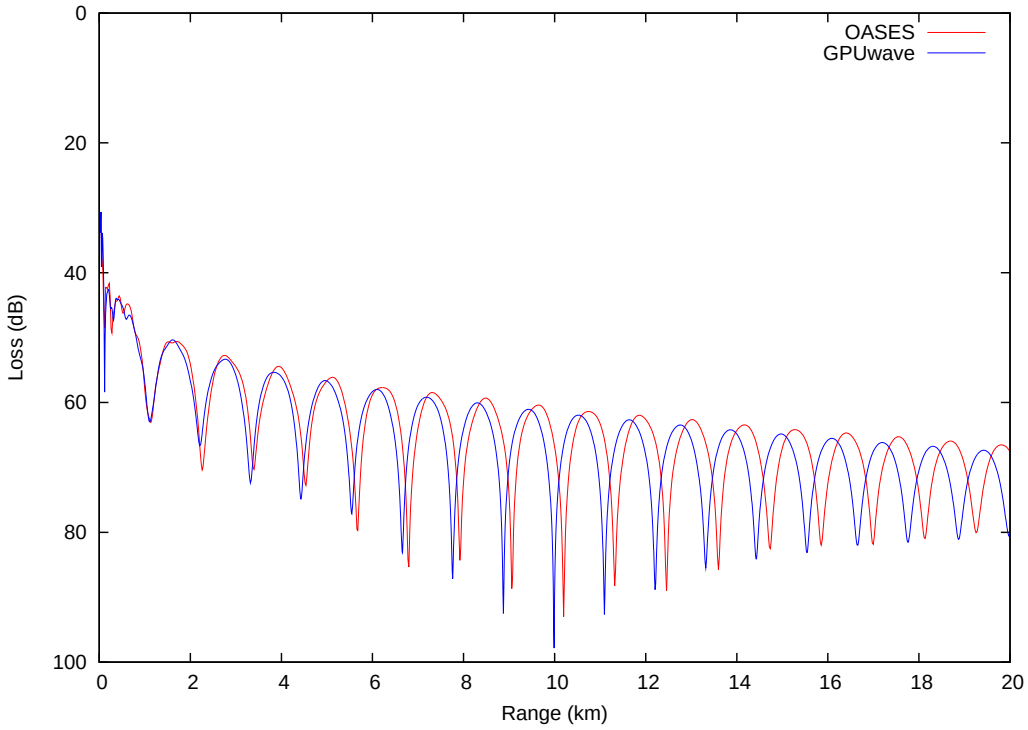

Figure 6.7: Accuracy comparison between GPUwave and OASES at 50Hz.

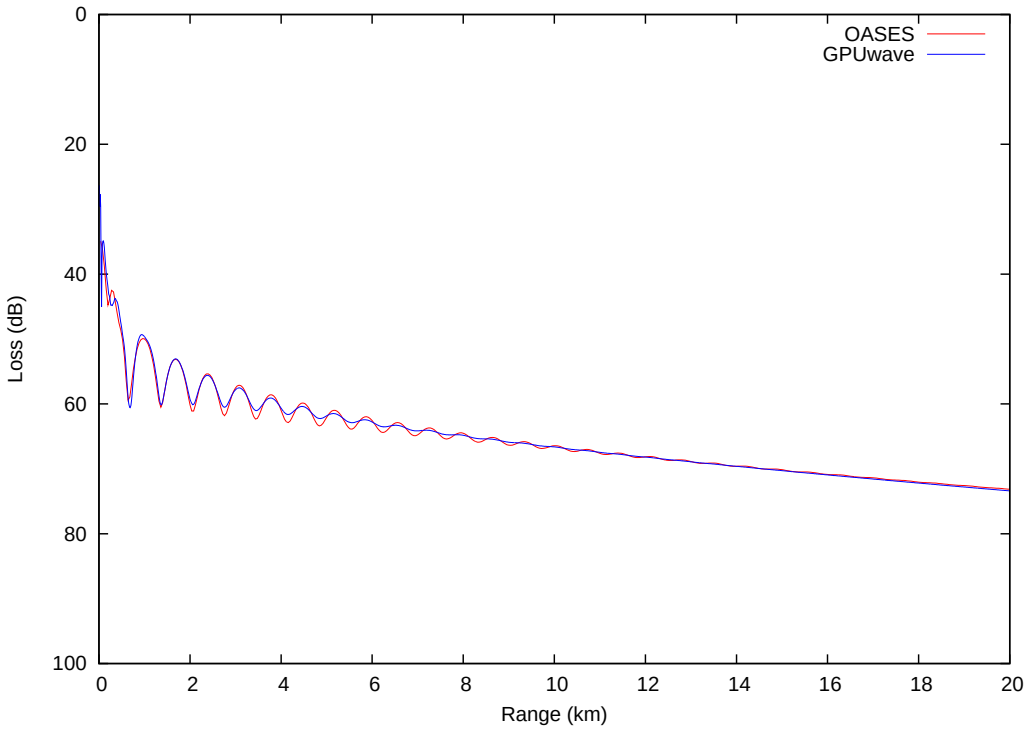

Figure 6.8: Bias-corrected accuracy comparison between GPUwave and OASES at 25Hz.

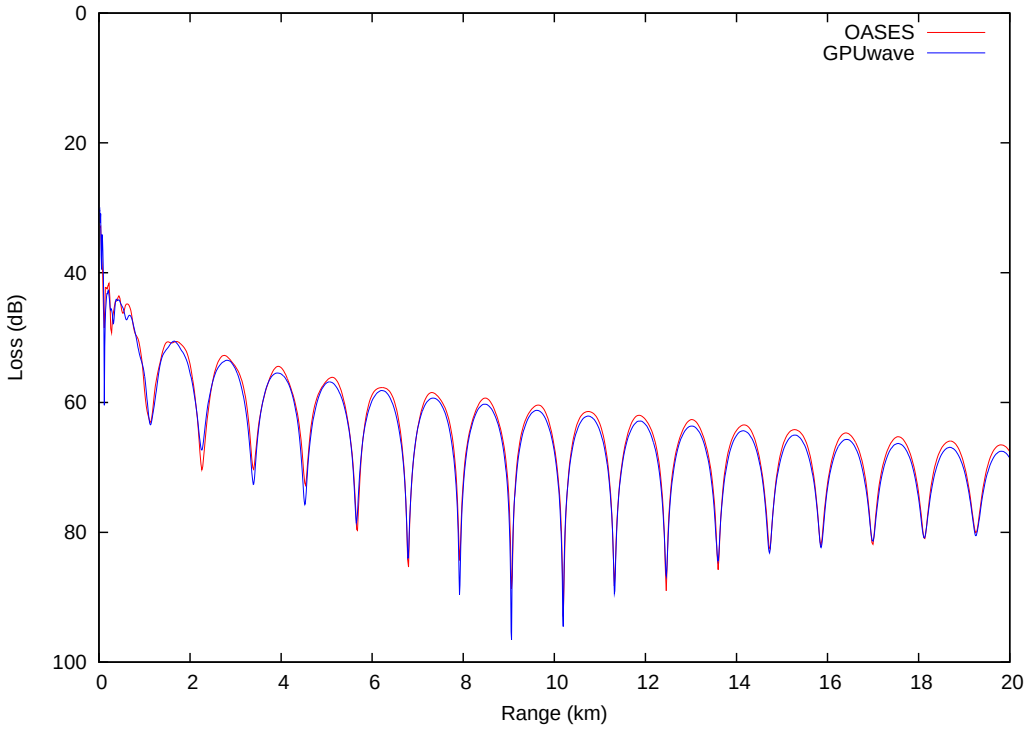

Figure 6.9: Bias-corrected accuracy comparison between GPUwave and OASES at 50Hz.

6.3 Conclusion

Based on our findings, it can be concluded that the GPU is highly suited to the task of computing underwater sound fields, especially with regard to efficiency. In particular, the GPU is well-suited to algorithms with a high degree of inherent parallelism, with the fastest cards outperforming the reference CPU implementation by more than three orders of magnitude on the largest test cases. Even though somewhat higher precision could be desired, the results are comparable to that of existing CPU implementations, and should be usable for many applications.

6.4 Future work

There are several interesting areas of future research. As previously noted, GPUwave itself could be optimized further, for instance by utilizing support for multiple render targets (MRT) in the FFT. Also, there are several extra features that could be implemented — most notably support for emulating double precision, which would be useful for simulations with many steps or otherwise high resolution.

Another possible area of research would be investigating opportunities for GPU implementations of more sophisticated algorithms than the split-step Fourier method (such as a split-step Padé method, as used by RAM) — this might result in both more efficient and accurate implementations than what is currently provided by GPUwave. It has, however, not been actively researched how viable such a GPU implementation would be.

Finally, both major GPU vendors have announced GPGPU-specific interfaces for their most recent series of GPUs[26, 9]. It is assumed that these interfaces (although proprietary) will provide a major boost in both efficiency and flexibility of GPU programming, as they provide a much thinner abstraction layer around the GPU, relieving the programmer of the work of "fitting triangular pegs in round holes" that is currently needed. Hopefully, eventually more general libraries for common routines (such as the FFT) will also appear, facilitating shorter development time for GPGPU programming in general and finally bringing it from its infancy into a mature, well-established field.

Bibliography

[1] OpenGL Architecture Review Board. *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.4*. Addison Wesley, 2004. Commonly known as "The Blue Book".

[2] OpenGL Architecture Review Board. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2*. Addison Wesley, 2005. Commonly known as "The Red Book".

[3] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *SIGGRAPH*, 2004.

[4] Michael D. Collins. User's guide for RAM versions 1.0 and 1.0p.

[5] Michael D. Collins. A split-step Padé solution for the parabolic equation method. *Journal of the Acoustical Society of America*, 93(4), April 1993.

[6] Michael D. Collins. Generalization of the split-step Padé solution. *Journal of the Acoustical Society of America*, 96(1), July 1994.

[7] Michael D. Collins, R.J. Cederberg, David B. King, and Stanley A. Chin-Bing. Comparison of algorithms for solving parabolic wave equations. *Journal of the Acoustical Society of America*, 100(1), July 1996.

[8] J. W. Cooley, P. A. Lewis, and P. D. Welch. The Fast Fourier Transform algorithm: Programming considerations in the calculation of sine, cosine and Laplace transforms. *Journal of Sound and Vibration*, 12(3):315–337, July 1970.

[9] NVIDIA Corporation. NVIDIA unveils CUDA™ — the GPU computing revolution begins. Press release, November 2006. Available online at `http://www.nvidia.com/object/IO_37226.html`.

[10] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, February 2005.

[11] Dominik Göddeke, Robert Strzodka, and Stefan Turek. Accelerating double precision FEM simulations with GPUs. In *ASIM*, September 2005.

[12] Mark Harris and David Luebke. Course notes for the SIGGRAPH 2005 GPGPU course. In *SIGGRAPH*, 2005. Available online at `http://www.gpgpu.org/s2005/`.

[13] Mark J. Harris. *Real-Time Cloud Simulation and Rendering.* PhD thesis, University of North Carolina, 2003.

[14] Mark J. Harris. GPGPU: Beyond graphics. In *Game Developers' Conference*, 2004.

[15] Olav Haugehåtveit. Programming graphic card for fast calculation of sound field in marine acoustics. Master's thesis, NTNU, June 2006.

[16] Jens Martin Hovem and Hefeng Dong. A forward model for geoacoustic inversion based on ray tracing and plane-wave refection coefficients. In *Proc. of the 2nd Workshop on Acoustic Inversion methods and Experiments for Assessment of the Shallow Water Environment*, 2004.

[17] Thomas Jansen, Bartosz von Rymon-Lipinski, Nils Hanssen, and Erwin Keeve. Fourier volume rendering on the GPU using a split-stream-FFT. *Proceedings of the Vision, Modeling and Visualization workshop*, pages 395–403, 2004.

[18] Finn B. Jensen, William A. Kuperman, Michael B. Porter, and Henrik Schmidt. *Computational Ocean Acoustics.* AIP Press, 1994.

[19] Emmett Kilgariff and Randima Fernando. The GeForce 6 series GPU architecture. In Matt Pharr, editor, *GPU Gems 2*, chapter 30, pages 471–491. Addison Wesley, March 2005.

[20] Aaron Lefohn, Ian Buck, Patrick McCormick, John Owens, Timothy Purcell, and Robert Strzodka. General purpose computation on graphics hardware. In *IEEE Visualization*, 2005.

[21] Aaron Lefohn, Joe M. Kniss, Robert Strzodka, Shubhabrata Sengupta, and John D. Owens. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics*, 25(1):60–99, January 2006. Available online at `http://graphics.cs.ucdavis.edu/~lefohn/work/glift/`.

[22] John Makhoul. A fast cosine transform in one and two dimensions. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(1):27–34, February 1980.

[23] Jeremy Meredith, David Bremer, Lawrence Flath, John Johnson, Holger Jones, Sheila Vaiyda, and Randall Frank. The GAIA project: Evaluation of GPU-based programming environments for knowledge discovery. In *HPEC*, 2004.

[24] Kenneth Moreland and Edward Angel. The FFT on a GPU. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119, 2003.

[25] Andreas Morland. Beregning av akustiske lydfelt med den parabolske likningen. Master's thesis, NTNU, June 2004.

[26] M. Peercy, M. Segal, and D. Derstmann. A performance-oriented data parallel virtual machine for GPUs. In *ACM SIGGRAPH sketches.* ACM Press, 2006.

[27] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, 1992.

[28] John G. Proakis and Dimitris G. Manolakis. *Digital Signal Processing.* Prentice-Hall, third edition, 1996.

[29] Randi J. Rost. *OpenGL Shading Language*. Addison Wesley, 2006.

[30] James C. Schatzman. Accuracy of the discrete Fourier transform and the fast Fourier transform. *SIAM Journal on Scientific Computing*, 17(5):1150–1166, 1996.

[31] Henrik Schmidt. OASES user guide and reference manual.

[32] Paul N. Swarztrauber. Symmetric FFTs. *Math. Comput.*, 47(175):323–346, 1986.

[33] Fred D. Tappert. The parabolic approximation method. In J. Ehlers, K. Hepp, R. Kippenhan, H.A. Weidenmüller, and J. Zittartz, editors, *Lecture Notes in Physics*, volume 70, pages 224–245. Springer-Verlag, 1977.

[34] Y.C. Yao and C.-Y. Hsu. New approach for fast sine transform. *Electronics Letters*, 28(15), July 1992.

Appendix A

GPUwave source code

This appendix contains all code for our reference implementation, licensed under the GNU General Public License, version 2. It is also available in electronic form from `http://gpuwave.sesse.net/`.

A.1 The GNU General Public License

The following is the text of the GNU General Public Licence, under the terms of which this software is distrubuted.

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

A.1.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

A.1.2 Terms and conditions for copying, distribution and modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

(c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the

operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. **Because the Program is licensed free of charge, there is no warranty for the Program, to the extent permitted by applicable law. except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the Program is with you. Should the Program prove defective, you assume the cost of all necessary servicing, repair or correction.**

12. **In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the Program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.**

END OF TERMS AND CONDITIONS

A.1.3 Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) 19yy  <name of author>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

A.2 Code listings

A.2.1 gpuwave.cpp

```
#define _CRT_SECURE_NO_DEPRECATE
#define _USE_MATH_DEFINES

#include <stdio.h>
#include <string.h>
#include <getopt.h>
#include <unistd.h>
#include <math.h>
#include <stdlib.h>
#include "glwindow.h"
#include <GL/glu.h>
#include <algorithm>
#include <string>

#ifdef __linux__
#include <GL/glext.h>
#include <time.h>
#include <sys/time.h>
#else
#include "glext.h"
#endif

#include "shader.h"

/* environment setup */
static double alpha_water = 0.004;              // dB/km
static double alpha_sediment = 0.5000;          // dB/lambda
static double alpha_sponge = 0.01;
static double c0 = 1491.00;
static double c_sediment = 1700;
static double rho_water = 1000.0;
static double rho_sediment = 1500.0;
static double max_depth = 512.0;
static double max_range = 10240.0;
static double sponge_layer_start = 400.0;
static double zs = 25.0;      // center of initial source
static double zr = 75.0;      // receiver line
static unsigned num_freqs = 2;
static unsigned z_levels = 9;
static unsigned z_steps = (1 << z_levels);
static unsigned range_levels = 9;
static unsigned range_steps = (1 << range_levels);
```

```
static unsigned fft_cutoff = 128;   // at what size to begin using the
   twiddle support texture
static unsigned truncate_bits = 0;
static double delta_z = max_depth / z_steps;
static double delta_r = max_range / range_steps;
static unsigned window_width = 1024;
static unsigned window_height = 512;
static double delta_kz = 2.0 * M_PI / (2.0 * z_steps * delta_z);
static double z_nudge = 0.5;     // 0.0 = early discretization of z,
   1.0 = late discretization
static double kz_nudge = 0.5;    // 0.0 = early discretization of k_z,
   1.0 = late discretization
static bool no_display = false;
static bool enable_r300_hack = false;
static bool use_greene = false;
static GLuint precision_format = GL_RGBA_FLOAT32_ATI;

static std::string depth_filename = "data/depth.txt";
static std::string ssp_filename = "data/ssp.txt";
static std::string f0_filename = "data/f0.txt";
static std::string results_filename = "results.txt";
static std::string log_filename = "log.txt";

/* corresponding options */
const static struct option longopts[] = {
        { "alpha-water", required_argument, NULL, 'a' },
        { "alpha-sediment", required_argument, NULL, 'A' },
        { "sound-speed-source", required_argument, NULL, 'c' },
        { "sound-speed-sediment", required_argument, NULL, 'C' },
        { "density-water", required_argument, NULL, 'e' },
        { "density-sediment", required_argument, NULL, 'E' },
        { "max-depth", required_argument, NULL, 'Z' },
        { "max-range", required_argument, NULL, 'R' },
        { "depth-res-order", required_argument, NULL, 'z' },
        { "range-res-order", required_argument, NULL, 'r' },
        { "sponge-layer-start", required_argument, NULL, 's' },
        { "source-depth", required_argument, NULL, 'd' },
        { "receiver-depth", required_argument, NULL, 'D' },
        { "num-frequencies", required_argument, NULL, 'n' },
        { "fft-cutoff", required_argument, NULL, 'f' },
        { "window-width", required_argument, NULL, 'w' },
        { "window-height", required_argument, NULL, 'h' },
        { "z-nudge", required_argument, NULL, 'u' },
        { "kz-nudge", required_argument, NULL, 'U' },
        { "greene-source", no_argument, NULL, 'g' },
        { "depth-file", required_argument, NULL, 'i' },
        { "ssp-file", required_argument, NULL, 'j' },
        { "f0-file", required_argument, NULL, 'k' },
        { "result-file", required_argument, NULL, 'l' },
        { "log-file", required_argument, NULL, 'm' },
        { "force-16-bit", no_argument, NULL, '1' },
        { "truncate-bits", no_argument, NULL, 't' },
        { "r300-workaround", no_argument, NULL, '3' },
```

```
        { "no-display", no_argument, NULL, 'q' },
        { "help", no_argument, NULL, 'h' },
        { NULL, 0, NULL, 0 }
};

float *start_field, *depth, *ssp, *f0, *support1, *support2, *
   delta_r_div_double_k0;
GLuint fb, fb_result;
GLuint depth_tex, ssp_tex, support1_tex, support2_tex,
   delta_r_div_double_k0_tex;
GLuint result_tex, initial_tex, rtt_tex1, rtt_tex2;
GLuint *twiddle_tex, *twiddle_inv_tex;
GLuint shuffle_tex, ravel_tex, unshuffle_tex, unravel_tex,
   exp_forward_tex, exp_inverse_tex;

Shader *fft_shader, *fft_constant_twiddle_shader, *display_shader, *
   refraction_shader, *diffraction_shader, *identity_shader, *
   truncate_shader, *identity_texcoord_shader;
Shader *shuffle_shader, *ravel_postmultiply_shader, *
   ravel_premultiply_shader;

struct fft_constant_data {
        double from0, to0;   // destination
        double from1, to1;   // source 1
        double from2, to2;   // source 2
        double twiddle_real, twiddle_imag;
};
std::vector<fft_constant_data> *twiddle_quads, *twiddle_inv_quads;

#ifdef __linux__
extern "C" void (*glXGetProcAddressARB(const GLubyte *procName))();
#endif

PFNGLGENFRAMEBUFFERSEXTPROC glGenFramebuffersEXT;
PFNGLGENRENDERBUFFERSEXTPROC glGenRenderbuffersEXT;
PFNGLBINDFRAMEBUFFEREXTPROC glBindFramebufferEXT;
PFNGLFRAMEBUFFERTEXTURE2DEXTPROC glFramebufferTexture2DEXT;
PFNGLBINDRENDERBUFFEREXTPROC glBindRenderbufferEXT;
PFNGLRENDERBUFFERSTORAGEEXTPROC glRenderbufferStorageEXT;
PFNGLFRAMEBUFFERRENDERBUFFEREXTPROC glFramebufferRenderbufferEXT;
PFNGLCHECKFRAMEBUFFERSTATUSEXTPROC glCheckFramebufferStatusEXT;
PFNGLCREATEPROGRAMOBJECTARBPROC glCreateProgramObjectARB;
PFNGLCREATESHADEROBJECTARBPROC glCreateShaderObjectARB;
PFNGLSHADERSOURCEARBPROC glShaderSourceARB;
PFNGLCOMPILESHADERARBPROC glCompileShaderARB;
PFNGLGETOBJECTPARAMETERIVARBPROC glGetObjectParameterivARB;
PFNGLGETINFOLOGARBPROC glGetInfoLogARB;
PFNGLATTACHOBJECTARBPROC glAttachObjectARB;
PFNGLDELETEOBJECTARBPROC glDeleteObjectARB;
PFNGLLINKPROGRAMARBPROC glLinkProgramARB;
PFNGLUSEPROGRAMOBJECTARBPROC glUseProgramObjectARB;
PFNGLGETUNIFORMLOCATIONARBPROC glGetUniformLocationARB;
PFNGLUNIFORM1IARBPROC glUniform1iARB;
```

```
PFNGLUNIFORM1FARBPROC glUniform1fARB;
PFNGLUNIFORM2FARBPROC glUniform2fARB;
PFNGLGETATTRIBLOCATIONARBPROC glGetAttribLocationARB;
PFNGLVERTEXATTRIB1FARBPROC glVertexAttrib1fARB;

#if WIN32 || !defined(GL_ARB_multitexture)
PFNGLMULTITEXCOORD1FARBPROC glMultiTexCoord1fARB;
PFNGLMULTITEXCOORD2FARBPROC glMultiTexCoord2fARB;
PFNGLMULTITEXCOORD3FARBPROC glMultiTexCoord3fARB;
PFNGLACTIVETEXTUREARBPROC glActiveTextureARB;
#endif

#define check_error() { GLenum ret = glGetError(); if (ret != 0) {
   printf("GL␣error␣at␣" __FILE__ ":%u:␣0x%04x\n", __LINE__, ret); exit
   (1); } }

void simulate_truncation(GLuint tex, unsigned b);

#ifdef WIN32
int ffs(long i)
{
        return int(log2(float(i))) + 1;
}
#endif

void * get_proc_address(const char *proc)
{
#ifdef __linux__
        void *ret = (void *)glXGetProcAddressARB((const GLubyte *)proc)
           ;
#else
        void *ret = (void *)wglGetProcAddress(proc);
#endif
        if (ret == NULL) {
                fprintf(stderr, "Error:␣Couldn't␣find␣'%s'\n", proc);
        }

        return ret;
}

GLuint make_texture_1d(GLuint w, GLenum internal_format, GLenum format,
    GLfloat *data)
{
        GLuint tex;

        glGenTextures(1, &tex);
        glBindTexture(GL_TEXTURE_1D, tex);
        glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER,
           GL_NEAREST);
        glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER,
           GL_NEAREST);
        glTexImage1D(GL_TEXTURE_1D, 0, internal_format, w, 0, format,
           GL_FLOAT, data);
```

```
        check_error();

        return tex;
}

GLuint make_texture_2d(GLuint w, GLuint h, GLenum internal_format,
   GLenum format, GLfloat *data)
{
        GLuint tex;

        glGenTextures(1, &tex);
        glBindTexture(GL_TEXTURE_2D, tex);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
           GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
           GL_NEAREST);
        glTexImage2D(GL_TEXTURE_2D, 0, internal_format, w, h, 0, format
           , GL_FLOAT, data);

        return tex;
}

/*
 * Writes 4/3 to the framebuffer in two different ways and measures
 * the precision; 4/3 is selected because its mantissa has every other
 * bit 1 and 0 in IEEE 754 notation, so it should catch truncation
 * errors quite quickly.
 */
void check_precision()
{
        float tmp[4];
        double test_double = 4.0/3.0;
        double error_texcoord, error_texture;

        glViewport(0, 0, z_steps, num_freqs/2);
        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
           GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, rtt_tex2, 0);

        // check texcoord precision
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0.0, 1.0, 0.0, 1.0, 0.0, 1.0);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        {
                Shader::Binding sb(*identity_texcoord_shader);
                glBegin(GL_QUADS);

                glTexCoord1d(test_double);
                glVertex2f(0.0, 0.0);
```

```
                glVertex2f(1.0, 0.0);
                glVertex2f(1.0, 1.0);
                glVertex2f(0.0, 1.0);

                glEnd();
        }

        glReadPixels(0, 0, 1, 1, GL_RGBA, GL_FLOAT, tmp);
        error_texcoord = fabs(double(tmp[0]) - test_double);

        // check texture precision
        tmp[0] = test_double;
        GLuint test_tex = make_texture_2d(1, 1, precision_format,
            GL_RGBA, tmp);

        {
                Shader::Binding sb(*identity_shader);
                sb.bind_texture_2d("tex", test_tex);
                glBegin(GL_QUADS);

                glTexCoord1f(0.0);
                glVertex2f(0.0, 0.0);

                glTexCoord1f(1.0);
                glVertex2f(1.0, 0.0);;

                glTexCoord1f(1.0);
                glVertex2f(1.0, 1.0);

                glTexCoord1f(0.0);
                glVertex2f(0.0, 1.0);

                glEnd();
        }

        glReadPixels(0, 0, 1, 1, GL_RGBA, GL_FLOAT, tmp);
        error_texture = fabs(double(tmp[0]) - test_double);

        if (error_texture > 1e-07 && precision_format ==
            GL_RGBA_FLOAT32_ATI) {
                printf("WARNING:␣Card␣does␣not␣have␣full␣32-bit␣
                    precision.\n");
                printf("␣␣␣␣␣␣␣␣␣␣Expect␣suboptimal␣accuracy␣in␣results
                    .\n");
        }
        if (error_texcoord/error_texture > 16.0) {
                printf("Note:␣Card␣has␣reduced␣texture␣coordinate␣
                    precision.\n");
                printf("␣␣␣␣␣␣␣Switching␣FFT␣strategy␣to␣compensate.\n")
                    ;
                fft_cutoff = 8;
        }
}
```

```
void swap_buffers()
{
        if (truncate_bits > 0) {
                std::swap(rtt_tex1, rtt_tex2);
                glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                   GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, rtt_tex2,
                   0);
                simulate_truncation(rtt_tex1, truncate_bits);
        }

        std::swap(rtt_tex1, rtt_tex2);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
           GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, rtt_tex2, 0);
}

void disp(GLuint tex, unsigned x)
{
        Shader::Binding sb(*display_shader);
        glViewport(0, 0, window_width, window_height);

        if (x == 0)
                glClear(GL_COLOR_BUFFER_BIT);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0.0, window_width, window_height, 0.0, 0.0, 1.0);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        glEnable(GL_TEXTURE_2D);
        glBindTexture(GL_TEXTURE_2D, tex);

        glBegin(GL_QUADS);

        glTexCoord2f(0.0, 1.0);
        glVertex2f(0.0, 0.0);

        glTexCoord2f(1.0, 1.0);
        glVertex2f(window_width, 0.0);

        glTexCoord2f(1.0, 0.0);
        glVertex2f(window_width, window_height);

        glTexCoord2f(0.0, 0.0);
        glVertex2f(0.0, window_height);

        glEnd();
}

/*
 * Change this function and/or the appropriate shader, plus read_back()
```

```
     if you
 * want to store different data (say, data from more than the first
    frequency, or
 * a loss vs. range/frequency plot).
 */
void store(GLuint tex, unsigned x)
{
        Shader::Binding sb(*identity_shader);

        if (x == 0)
                glClear(GL_COLOR_BUFFER_BIT);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();

        glViewport(0, 0, range_steps, z_steps);
        glOrtho(0.0, range_steps, z_steps, 0.0, 0.0, 1.0);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        glEnable(GL_TEXTURE_2D);
        glBindTexture(GL_TEXTURE_2D, tex);

        glBegin(GL_QUADS);

        glTexCoord2f(0.0, 0.0);
        glVertex2f(x, 0.0);

        glTexCoord2f(0.0, 1.0/double(num_freqs/2));
        glVertex2f(x + 1.0, 0.0);

        glTexCoord2f(1.0, 1.0/double(num_freqs/2));
        glVertex2f(x + 1.0, double(z_steps));

        glTexCoord2f(1.0, 0.0);
        glVertex2f(x, double(z_steps));

        glEnd();
}


void identity(GLuint tex)
{
        Shader::Binding sb(*identity_shader);
        sb.bind_texture_2d("tex", initial_tex);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0.0, 1.0, 0.0, 1.0, 0.0, 1.0);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
```

```
        check_error();

        glBegin(GL_QUADS);

        glTexCoord2f(0.0, 0.0);
        glVertex2f(0.0, 0.0);

        glTexCoord2f(1.0, 0.0);
        glVertex2f(1.0, 0.0);

        glTexCoord2f(1.0, 1.0);
        glVertex2f(1.0, 1.0);

        glTexCoord2f(0.0, 1.0);
        glVertex2f(0.0, 1.0);

        glEnd();
}

void simulate_truncation(GLuint tex, unsigned b)
{
        Shader::Binding sb(*truncate_shader);
        sb.bind_texture_2d("tex", tex);
        sb.set_uniform("scalefac", float((1 << b) - 1));

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0.0, 1.0, 0.0, 1.0, 0.0, 1.0);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        check_error();

        glBegin(GL_QUADS);

        glTexCoord2f(0.0, 0.0);
        glVertex2f(0.0, 0.0);

        glTexCoord2f(1.0, 0.0);
        glVertex2f(1.0, 0.0);

        glTexCoord2f(1.0, 1.0);
        glVertex2f(1.0, 1.0);

        glTexCoord2f(0.0, 1.0);
        glVertex2f(0.0, 1.0);

        glEnd();
}

void refraction(GLuint tex, unsigned frameno)
```

```
{
        Shader::Binding sb(*refraction_shader);
        sb.bind_texture_2d("tex", tex);
        sb.bind_texture_1d("support1", support1_tex);
        sb.bind_texture_1d("support2", support2_tex);
        sb.bind_texture_1d("depth", depth_tex);
        sb.bind_texture_1d("ssp", ssp_tex);

        sb.set_uniform("delta_r", delta_r);
        sb.set_uniform("r", frameno * delta_r);
        sb.set_uniform("r_coord", frameno / double(range_steps));
        sb.set_uniform("alpha_sponge", alpha_sponge);
        sb.set_uniform("rho_average", 0.5 * (rho_sediment + rho_water))
           ;
        sb.set_uniform("half_rho_difference", 0.5 * (rho_sediment -
           rho_water));
        sb.set_uniform("c_sediment", c_sediment);
        sb.set_uniform("c0_div_c_sediment_m1", c0 / c_sediment - 1.0);
        sb.set_uniform("c0", c0);
        sb.set_uniform("sponge_layer_start", sponge_layer_start);
        sb.set_uniform("sponge_layer_end", max_depth);
        sb.set_uniform("three_inv_sponge_len", 3.0 / (max_depth -
           sponge_layer_start));

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0.0, 1.0, 0.0, 1.0, 0.0, 1.0);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        glBegin(GL_QUADS);

        glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0, 0.0);
        glMultiTexCoord1fARB(GL_TEXTURE1_ARB, 0.0);
        glMultiTexCoord2fARB(GL_TEXTURE2_ARB, z_nudge, 0.0);
        glVertex2f(0.0, 0.0);

        glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0, 0.0);
        glMultiTexCoord1fARB(GL_TEXTURE1_ARB, 0.0);
        glMultiTexCoord2fARB(GL_TEXTURE2_ARB, max_depth + z_nudge, 1.0)
           ;
        glVertex2f(1.0, 0.0);

        glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0, 1.0);
        glMultiTexCoord1fARB(GL_TEXTURE1_ARB, 1.0);
        glMultiTexCoord2fARB(GL_TEXTURE2_ARB, max_depth + z_nudge, 1.0)
           ;
        glVertex2f(1.0, 1.0);

        glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0, 1.0);
        glMultiTexCoord1fARB(GL_TEXTURE1_ARB, 1.0);
        glMultiTexCoord2fARB(GL_TEXTURE2_ARB, z_nudge, 0.0);
```

```
        glVertex2f(0.0, 1.0);

        glEnd();
}

void shuffle(GLuint tex, GLuint shuffle_data_tex)
{
        Shader::Binding sb(*shuffle_shader);
        sb.bind_texture_2d("tex", tex);
        sb.bind_texture_1d("shuffle_data", shuffle_data_tex);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0.0, 1.0, 0.0, 1.0, 0.0, 1.0);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        glBegin(GL_QUADS);

        glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0, 0.0);
        glVertex2f(0.0, 0.0);

        glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0, 0.0);
        glVertex2f(1.0, 0.0);

        glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0, 1.0);
        glVertex2f(1.0, 1.0);

        glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0, 1.0);
        glVertex2f(0.0, 1.0);

        glEnd();
}

void ravel(Shader &s, GLuint tex, GLuint rtex, GLuint etex)
{
        Shader::Binding sb(s);
        sb.bind_texture_2d("tex", tex);
        sb.bind_texture_1d("ravel_data_tex", rtex);
        sb.bind_texture_1d("exp_data_tex", etex);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0.0, 1.0, 0.0, 1.0, 0.0, 1.0);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        glBegin(GL_QUADS);

        glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0, 0.0);
        glVertex2f(0.0, 0.0);
```

```
        glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0, 0.0);
        glVertex2f(1.0, 0.0);

        glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0, 1.0);
        glVertex2f(1.0, 1.0);

        glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0, 1.0);
        glVertex2f(0.0, 1.0);

        glEnd();
}


void diffraction(GLuint tex)
{
        Shader::Binding sb(*diffraction_shader);
        sb.bind_texture_2d("tex", tex);
        sb.bind_texture_1d("delta_r_div_double_k0_tex",
           delta_r_div_double_k0_tex);
        sb.set_uniform("scalefac", 1.0 / double(z_steps));

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0.0, 1.0, 0.0, 1.0, 0.0, 1.0);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        glBegin(GL_QUADS);

        glMultiTexCoord3fARB(GL_TEXTURE0_ARB, 0.0, 0.0, kz_nudge *
           delta_kz);
        glMultiTexCoord1fARB(GL_TEXTURE1_ARB, 0.0);
        glVertex2f(0.0, 0.0);

        glMultiTexCoord3fARB(GL_TEXTURE0_ARB, 1.0, 0.0, (z_steps +
           kz_nudge) * delta_kz);
        glMultiTexCoord1fARB(GL_TEXTURE1_ARB, 0.0);
        glVertex2f(1.0, 0.0);

        glMultiTexCoord3fARB(GL_TEXTURE0_ARB, 1.0, 1.0, (z_steps +
           kz_nudge) * delta_kz);
        glMultiTexCoord1fARB(GL_TEXTURE1_ARB, 1.0);
        glVertex2f(1.0, 1.0);

        glMultiTexCoord3fARB(GL_TEXTURE0_ARB, 0.0, 1.0, kz_nudge *
           delta_kz);
        glMultiTexCoord1fARB(GL_TEXTURE1_ARB, 1.0);
        glVertex2f(0.0, 1.0);

        glEnd();
}
```

```
GLuint make_twiddle_texture(unsigned fft_size, float mulfac, bool
    inverse)
{
        float tmp[z_steps * 4];
        float fft_stride = double(z_steps) / fft_size;

        for (unsigned element = 0; element < z_steps; ++element) {
                float k = floor(element / fft_stride);     // element #
                    in this fft
                float offset = element - k * fft_stride;   // FFT
                   number
                float twiddle_real, twiddle_imag;

                if (k < 0.5 * fft_size) {
                        twiddle_real = cos(mulfac * k);
                        twiddle_imag = sin(mulfac * k);
                } else {
                        k -= 0.5 * fft_size;
                        twiddle_real = -cos(mulfac * k);
                        twiddle_imag = -sin(mulfac * k);
                }

                float base;
                base = k * fft_stride * 2.0 + offset;

                tmp[element * 4 + 0] = base / double(z_steps);
                tmp[element * 4 + 1] = (fft_stride + base) / double(
                   z_steps);
                tmp[element * 4 + 2] = twiddle_real;
                tmp[element * 4 + 3] = twiddle_imag;
        }

        // convert this back to constant_sized quads
        std::vector<fft_constant_data> &q = (inverse) ?
           twiddle_inv_quads[ffs(fft_size) - 2] : twiddle_quads[ffs(
           fft_size) - 2];
        for (unsigned element = 0; element < z_steps; element +=
           unsigned(fft_stride)) {
                fft_constant_data fd;

                fd.from0 = element;
                fd.to0 = (element + fft_stride);

                fd.from1 = 2 * element / double(z_steps);
                fd.to1 = (2 * element + fft_stride) / double(z_steps);

                fd.from2 = (2 * element + fft_stride) / double(z_steps)
                   ;
                fd.to2 = (2 * element + 2 * fft_stride) / double(
                   z_steps);

                fd.twiddle_real = tmp[element * 4 + 2];
```

```
                fd.twiddle_imag = tmp[element * 4 + 3];

                q.push_back(fd);
        }

        return make_texture_1d(z_steps, (fft_size <= 4) ?
           GL_RGBA_FLOAT16_ATI : precision_format, GL_RGBA, tmp);
}

// ravel/unravel textures are packed as [x1, x2, f1, f2], and the ravel
// shader computes f1 * x[x1] + i f2 x[x2]
GLuint make_ravel_texture(unsigned fft_size)
{
        float tmp[z_steps * 4];

        // y[0] = -2i x[N-1]
        tmp[0] = (fft_size - 1) / float(fft_size);
        tmp[1] = (fft_size - 1) / float(fft_size);
        tmp[2] = 0.0;
        tmp[3] = -2.0;

        // y[n] = 2x[n-1] - 2i x[N-n-1]      (n > 0)
        for (unsigned i = 1; i < unsigned(fft_size); ++i) {
                tmp[4*i + 0] = float(i - 1) / float(fft_size);
                tmp[4*i + 1] = (fft_size - i - 1) / float(fft_size);
                tmp[4*i + 2] = 2.0;
                tmp[4*i + 3] = -2.0;
        }

        return make_texture_1d(z_steps, precision_format, GL_RGBA, tmp)
           ;
}

GLuint make_unravel_texture(unsigned fft_size)
{
        float tmp[z_steps * 4];

        // x[n] = 1/4 ( y[n+1] + i y[N-n-1])      (n < N-1)
        for (unsigned i = 0; i < fft_size - 1; ++i) {
                tmp[4*i + 0] = (i + 1) / float(fft_size);
                tmp[4*i + 1] = (fft_size - i - 1) / float(fft_size);
                tmp[4*i + 2] = 0.25;
                tmp[4*i + 3] = 0.25;
        }

        // x[N-1] = 1/2 i y[0]
        tmp[4*(fft_size-1) + 0] = 0.0;
        tmp[4*(fft_size-1) + 1] = 0.0;
        tmp[4*(fft_size-1) + 2] = 0.0;
        tmp[4*(fft_size-1) + 3] = 0.5;

        return make_texture_1d(z_steps, precision_format, GL_RGBA, tmp)
           ;
```

```
}

// shuffle/unshuffle textures are packed as [x1, f1, 0, 0], and the
   shuffle
// shader computes f1 * x[x1]
GLuint make_shuffle_texture(unsigned fft_size)
{
        float tmp[z_steps * 4];

        for (unsigned i = 0; i < fft_size/2; ++i) {
                tmp[4*i + 0] = (i * 2) / float(fft_size);
                tmp[4*i + 1] = 1.0;
                tmp[4*i + 2] = 0.0;
                tmp[4*i + 3] = 0.0;
        }
        for (unsigned i = unsigned(fft_size)/2, j = 0; i < unsigned(
           fft_size); ++i, ++j) {
                tmp[4*i + 0] = (fft_size - j * 2 - 1) / float(fft_size)
                   ;
                tmp[4*i + 1] = -1.0;
                tmp[4*i + 2] = 0.0;
                tmp[4*i + 3] = 0.0;
        }

        return make_texture_1d(z_steps, precision_format, GL_RGBA, tmp)
           ;
}

GLuint make_unshuffle_texture(unsigned fft_size)
{
        float tmp[z_steps * 4];

        for (unsigned i = 0; i < fft_size/2; ++i) {
                unsigned k = 2*i;

                tmp[4*k + 0] = i / float(fft_size);
                tmp[4*k + 1] = 1.0;
                tmp[4*k + 2] = 0.0;
                tmp[4*k + 3] = 0.0;
        }
        for (unsigned i = fft_size/2, j = 0; i < fft_size; ++i, ++j) {
                unsigned k = fft_size - 2*j - 1;

                tmp[4*k + 0] = i / float(fft_size);
                tmp[4*k + 1] = -1.0;
                tmp[4*k + 2] = 0.0;
                tmp[4*k + 3] = 0.0;
        }

        return make_texture_1d(z_steps, precision_format, GL_RGBA, tmp)
           ;
}
```

```
GLuint make_exp_texture(unsigned fft_size, bool inverse)
{
        float tmp[z_steps * 4];

        for (unsigned i = 0; i < fft_size; ++i) {
                double f = float(i)*M_PI/(2.0*fft_size);

                tmp[4*i + 0] = cos(f);
                tmp[4*i + 1] = (inverse ? -sin(f) : sin(f));
                tmp[4*i + 2] = 0.0;
                tmp[4*i + 3] = 0.0;
        }

        return make_texture_1d(z_steps, precision_format, GL_RGBA, tmp)
           ;
}

void precompute_twiddle()
{
        for (unsigned s = 2; s <= z_steps; s *= 2) {
                float mulfac = -2.0 * M_PI / double(s);
                twiddle_tex[ffs(s) - 2] = make_twiddle_texture(s,
                   mulfac, false);
        }
        for (unsigned s = 2; s <= z_steps; s *= 2) {
                float mulfac = 2.0 * M_PI / double(s);
                twiddle_inv_tex[ffs(s) - 2] = make_twiddle_texture(s,
                   mulfac, true);
        }
}

void fft(GLuint tex, GLuint size, bool inverse)
{
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0.0, double(z_steps), 0.0, double(num_freqs/2), 0.0,
           1.0);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        Shader::Binding sb(*fft_shader);
        sb.bind_texture_2d("tex", tex);
        if (inverse)
                sb.bind_texture_1d("twiddle_tex", twiddle_inv_tex[ffs(
                   size)-2]);
        else
                sb.bind_texture_1d("twiddle_tex", twiddle_tex[ffs(size)
                   -2]);

        glBegin(GL_QUADS);

        glTexCoord2f(0.0, 0.0);
```

```
        glVertex2f(0.0, 0.0);

        glTexCoord2f(1.0, 0.0);
        glVertex2f(double(z_steps), 0.0);

        glTexCoord2f(1.0, 1.0);
        glVertex2f(double(z_steps), double(num_freqs/2));

        glTexCoord2f(0.0, 1.0);
        glVertex2f(0.0, double(num_freqs/2));

        glEnd();
}

void fft_constant_twiddle(GLuint tex, GLuint size, bool inverse)
{
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0.0, double(z_steps), 0.0, double(num_freqs/2), 0.0,
           1.0);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        Shader::Binding sb(*fft_constant_twiddle_shader);
        sb.bind_texture_2d("tex", tex);

        std::vector<fft_constant_data> &q = (inverse) ?
           twiddle_inv_quads[ffs(size) - 2] : twiddle_quads[ffs(size) -
            2];

        glBegin(GL_QUADS);

        for (std::vector<fft_constant_data>::const_iterator i = q.begin
           (); i != q.end(); ++i) {
                glMultiTexCoord2fARB(GL_TEXTURE0_ARB, i->from1, 0.0);
                glMultiTexCoord2fARB(GL_TEXTURE1_ARB, i->from2, 0.0);
                glMultiTexCoord2fARB(GL_TEXTURE2_ARB, i->twiddle_real,
                   i->twiddle_imag);
                glVertex2f(i->from0, 0.0);

                glMultiTexCoord2fARB(GL_TEXTURE0_ARB, i->to1, 0.0);
                glMultiTexCoord2fARB(GL_TEXTURE1_ARB, i->to2, 0.0);
                glMultiTexCoord2fARB(GL_TEXTURE2_ARB, i->twiddle_real,
                   i->twiddle_imag);
                glVertex2f(i->to0, 0.0);

                glMultiTexCoord2fARB(GL_TEXTURE0_ARB, i->to1, 1.0);
                glMultiTexCoord2fARB(GL_TEXTURE1_ARB, i->to2, 1.0);
                glMultiTexCoord2fARB(GL_TEXTURE2_ARB, i->twiddle_real,
                   i->twiddle_imag);
                glVertex2f(i->to0, double(num_freqs/2));
```

```
                glMultiTexCoord2fARB(GL_TEXTURE0_ARB, i->from1, 1.0);
                glMultiTexCoord2fARB(GL_TEXTURE1_ARB, i->from2, 1.0);
                glMultiTexCoord2fARB(GL_TEXTURE2_ARB, i->twiddle_real,
                    i->twiddle_imag);
                glVertex2f(i->from0, double(num_freqs/2));
        }

        glEnd();
}

GLint gen_rtt(GLenum format, GLuint width, GLuint height)
{
        GLuint rtt_tex;
        glGenTextures(1, &rtt_tex);

        glBindTexture(GL_TEXTURE_2D, rtt_tex);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
            GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
            GL_NEAREST);
        glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0,
            GL_RGBA, GL_FLOAT, NULL);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
            GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, rtt_tex, 0);

        check_error();
        return rtt_tex;
}

void read_depth(const char *filename, float *dst)
{
        FILE *f = fopen(filename, "r");
        if (f == NULL) {
                perror(filename);
                exit(1);
        }

        unsigned w;
        if (fscanf(f, "%u", &w) != 1) {
                fprintf(stderr, "short␣read␣in␣depth␣data\n");
                exit(1);
        }

        memset(dst, 0, range_steps * 4 * sizeof(float));

        for (unsigned x = 0; x < w && x < range_steps; ++x) {
                double d;
                if (fscanf(f, "%lf", &d) != 1) {
                        fprintf(stderr, "short␣read␣in␣depth␣data\n");
                        exit(1);
                }

                dst[x * 4] = d;
```

```
        }

        fclose(f);
}

void read_f0(const char *filename, float *dst1, float *dst2)
{
        FILE *f = fopen(filename, "r");
        if (f == NULL) {
                perror(filename);
                exit(1);
        }

        unsigned w;
        if (fscanf(f, "%u", &w) != 1) {
                fprintf(stderr, "short read in f0 data\n");
                exit(1);
        }

        memset(dst1, 0, num_freqs * 2 * sizeof(float));
        memset(dst2, 0, num_freqs * 2 * sizeof(float));
        memset(delta_r_div_double_k0, 0, num_freqs * 2 * sizeof(float))
           ;

        for (unsigned x = 0; x < w; ++x) {
                double g;

                if (fscanf(f, "%lf", &g) != 1) {
                        fprintf(stderr, "short read in f0 data\n");
                        exit(1);
                }

                if (x >= num_freqs)
                        break;

                double k0 = 2.0 * M_PI * g / c0;

                f0[x] = g;
                dst1[x * 2 + 0] = (20.0 * M_LOG10E * alpha_water * ( c0
                    / ( g * 1000.0 ) )) / 27.29;
                dst1[x * 2 + 1] = alpha_sediment / 27.29;
                dst2[x * 2 + 0] = 0.5 * k0;
                dst2[x * 2 + 1] = 0.5 / (k0*k0);

                delta_r_div_double_k0[x * 2 + 0] = delta_r / (2.0*k0);
        }

        fclose(f);

}

// will neccessarily be incomplete (doesn't count the refraction layer)
// but that's OK; the other values will be ignored by the shader
```

```
void read_ssp(const char *filename, float *dst)
{
        FILE *f = fopen(filename, "r");
        if (f == NULL) {
                perror(filename);
                exit(1);
        }

        unsigned h;
        if (fscanf(f, "%u", &h) != 1) {
                fprintf(stderr, "short␣read␣in␣SSP␣data\n");
                exit(1);
        }

        memset(dst, 0, z_steps * 4 * sizeof(float));

        for (unsigned z = 0; z < h && z < z_steps; ++z) {
                double c;
                if (fscanf(f, "%lf", &c) != 1) {
                        fprintf(stderr, "short␣read␣in␣SSP␣data\n");
                        exit(1);
                }

                dst[z * 4] = c;
        }

        fclose(f);
}


void load_opengl_extensions()
{
        // FBO extension
        glGenFramebuffersEXT = (PFNGLGENFRAMEBUFFERSEXTPROC)
           get_proc_address("glGenFramebuffersEXT");
        glGenRenderbuffersEXT = (PFNGLGENRENDERBUFFERSEXTPROC)
           get_proc_address("glGenRenderbuffersEXT");
        glBindFramebufferEXT = (PFNGLBINDFRAMEBUFFEREXTPROC)
           get_proc_address("glBindFramebufferEXT");
        glFramebufferTexture2DEXT = (PFNGLFRAMEBUFFERTEXTURE2DEXTPROC)
           get_proc_address("glFramebufferTexture2DEXT");
        glBindRenderbufferEXT = (PFNGLBINDRENDERBUFFEREXTPROC)
           get_proc_address("glBindRenderbufferEXT");
        glRenderbufferStorageEXT = (PFNGLRENDERBUFFERSTORAGEEXTPROC)
           get_proc_address("glRenderbufferStorageEXT");
        glFramebufferRenderbufferEXT = (
           PFNGLFRAMEBUFFERRENDERBUFFEREXTPROC)get_proc_address("
           glFramebufferRenderbufferEXT");
        glCheckFramebufferStatusEXT = (
           PFNGLCHECKFRAMEBUFFERSTATUSEXTPROC)get_proc_address("
           glCheckFramebufferStatusEXT");

        // shader extension
```

```
        glCreateProgramObjectARB = (PFNGLCREATEPROGRAMOBJECTARBPROC)
           get_proc_address("glCreateProgramObjectARB");
        glCreateShaderObjectARB = (PFNGLCREATESHADEROBJECTARBPROC)
           get_proc_address("glCreateShaderObjectARB");
        glShaderSourceARB = (PFNGLSHADERSOURCEARBPROC)get_proc_address(
           "glShaderSourceARB");
        glCompileShaderARB = (PFNGLCOMPILESHADERARBPROC)
           get_proc_address("glCompileShaderARB");
        glGetObjectParameterivARB = (PFNGLGETOBJECTPARAMETERIVARBPROC)
           get_proc_address("glGetObjectParameterivARB");
        glGetInfoLogARB = (PFNGLGETINFOLOGARBPROC)get_proc_address("
           glGetInfoLogARB");
        glAttachObjectARB = (PFNGLATTACHOBJECTARBPROC)get_proc_address(
           "glAttachObjectARB");
        glDeleteObjectARB = (PFNGLDELETEOBJECTARBPROC)get_proc_address(
           "glDeleteObjectARB");
        glLinkProgramARB = (PFNGLLINKPROGRAMARBPROC)get_proc_address("
           glLinkProgramARB");
        glUseProgramObjectARB = (PFNGLUSEPROGRAMOBJECTARBPROC)
           get_proc_address("glUseProgramObjectARB");
        glGetUniformLocationARB = (PFNGLGETUNIFORMLOCATIONARBPROC)
           get_proc_address("glGetUniformLocationARB");
        glUniform1iARB = (PFNGLUNIFORM1IARBPROC)get_proc_address("
           glUniform1iARB");
        glUniform1fARB = (PFNGLUNIFORM1FARBPROC)get_proc_address("
           glUniform1fARB");
        glUniform2fARB = (PFNGLUNIFORM2FARBPROC)get_proc_address("
           glUniform2fARB");
        glGetAttribLocationARB = (PFNGLGETATTRIBLOCATIONARBPROC)
           get_proc_address("glGetAttribLocationARB");
        glVertexAttrib1fARB = (PFNGLVERTEXATTRIB1FARBPROC)
           get_proc_address("glVertexAttrib1fARB");

#if WIN32 || !defined(GL_ARB_multitexture)
        glMultiTexCoord1fARB = (PFNGLMULTITEXCOORD1FARBPROC)
           get_proc_address("glMultiTexCoord1fARB");
        glMultiTexCoord2fARB = (PFNGLMULTITEXCOORD2FARBPROC)
           get_proc_address("glMultiTexCoord2fARB");
        glMultiTexCoord3fARB = (PFNGLMULTITEXCOORD3FARBPROC)
           get_proc_address("glMultiTexCoord3fARB");
        glActiveTextureARB = (PFNGLACTIVETEXTUREARBPROC)
           get_proc_address("glActiveTextureARB");
#endif

        check_error();
}

void generate_gaussian(float *start_field, float *freqs)
{
        for (unsigned f = 0; f < num_freqs; f += 2) {
                double k0_1 = 2.0 * M_PI * freqs[f] / c0;
                double k0_2 = 2.0 * M_PI * freqs[f+1] / c0;
```

```
                for (unsigned z = 0; z < z_steps; ++z) {
                        float zf = float(z) * max_depth / double(
                           z_steps);

                        start_field[(z + (f/2) * z_steps) * 4 + 0] =
                           sqrt(k0_1) * exp(-(k0_1 * k0_1 * .5) * (zf -
                            zs) * (zf - zs));
                        start_field[(z + (f/2) * z_steps) * 4 + 1] =
                           0.0;
                        start_field[(z + (f/2) * z_steps) * 4 + 2] =
                           sqrt(k0_2) * exp(-(k0_2 * k0_2 * .5) * (zf -
                            zs) * (zf - zs));
                        start_field[(z + (f/2) * z_steps) * 4 + 3] =
                           0.0;
                }
        }
}

void generate_greene(float *start_field, float *freqs)
{
        for (unsigned f = 0; f < num_freqs; f += 2) {
                double k0_1 = 2.0 * M_PI * freqs[f] / c0;
                double k0_2 = 2.0 * M_PI * freqs[f+1] / c0;

                for (unsigned z = 0; z < z_steps; ++z) {
                        float zf = float(z) * max_depth / double(
                           z_steps);

                        start_field[(z + (f/2) * z_steps) * 4 + 0] =
                           sqrt(k0_1) * (1.4467 - 0.4201*k0_1*k0_1 * (
                           zf - zs) * (zf - zs))*exp(-(k0_1*(zf-zs)*(zf
                           -zs))/3.0512);
                        start_field[(z + (f/2) * z_steps) * 4 + 1] =
                           0.0;
                        start_field[(z + (f/2) * z_steps) * 4 + 2] =
                           sqrt(k0_2) * (1.4467 - 0.4201*k0_2*k0_2 * (
                           zf - zs) * (zf - zs))*exp(-(k0_2*(zf-zs)*(zf
                           -zs))/3.0512);
                        start_field[(z + (f/2) * z_steps) * 4 + 3] =
                           0.0;
                }
        }
}

double abs_hankel0(double x)
{
        double real = j0(x), imag = y0(x);
        return sqrt(real * real + imag * imag);
}

void read_back()
{
        double k0 = 2.0 * M_PI * f0[0] / c0;
```

```
        unsigned z = (z_steps-1) - unsigned((zr - z_nudge) / delta_z +
           0.5);
        float tmp[range_steps * 4];

        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb_result);

        glReadPixels(0, z, range_steps, 1, GL_RGBA, GL_FLOAT, tmp);
        check_error();

        // All dB results are expected relative to r = 1m, but our
           results
        // are not correct that close to the origin. Let us just use
        // the Hankel function at 1m as base.
        float p1v = abs_hankel0(k0);

        FILE *f = fopen(results_filename.c_str(), "w");
        for (unsigned i = 0; i < range_steps; ++i) {
                float r = i * delta_r;
                float v = sqrt(tmp[i * 4] * tmp[i * 4] + tmp[i * 4 + 1]
                    * tmp[i * 4 + 1]) * abs_hankel0(k0 * r);

                fprintf(f, "%f␣%f\n", r, -20.0*log10(v / p1v));
        }
        fclose(f);
}

/*
 * The (tab-separated) columns in the log file are:
 *
 * - Number of z-levels
 * - Number of range levels
 * - Number of frequencies
 * - FFT cutoff point
 * - Benchmark mode (no display) (0/1)
 * - Forced 16-bit precision (0/1)
 * - Truncate mode active (0/1)
 * - R300 hack (0/1)
 * - Time spent (seconds)
 * - Operating system (Win32/UNIX)
 * - OpenGL vendor
 * - OpenGL renderer
 * - OpenGL version
 */
void write_log(double elapsed)
{
        FILE *log = fopen(log_filename.c_str(), "a");

        fprintf(log, "%u\t%u\t%u\t%u\t%u\t%u\t%u\t%u\t%u\t%f\t",
                z_levels, range_levels, num_freqs, fft_cutoff,
                no_display, (precision_format == GL_RGBA_FLOAT16_ATI),
                (truncate_bits > 0), enable_r300_hack, elapsed);
#ifdef WIN32
        fprintf(log, "Win32\t");
```

```
#else
        fprintf(log, "UNIX\t");
#endif

        fprintf(log, "%s\t", glGetString(GL_VENDOR));
        fprintf(log, "%s\t", glGetString(GL_RENDERER));
        fprintf(log, "%s\n", glGetString(GL_VERSION));

        fclose(log);
}

void parse_options(int argc, char **argv)
{
        int option_index = 0;
        for ( ;; ) {

                int c = getopt_long(argc, argv, "a:A:c:C:e:E:Z:R:z:r:s:
                   d:D:n:f:x:y:u:U:gi:j:k:l:m:q13t:h", longopts, &
                   option_index);

                switch (c) {
                case 'a':
                        alpha_water = atof(optarg);
                        break;
                case 'A':
                        alpha_sediment = atof(optarg);
                        break;
                case 'c':
                        c0 = atof(optarg);
                        break;
                case 'C':
                        c_sediment = atof(optarg);
                        break;
                case 'e':
                        rho_water = atof(optarg);
                        break;
                case 'E':
                        rho_sediment = atof(optarg);
                        break;
                case 'Z':
                        max_depth = atof(optarg);
                        break;
                case 'R':
                        max_range = atof(optarg);
                        break;
                case 'z':
                        z_levels = atoi(optarg);
                        break;
                case 'r':
                        range_levels = atoi(optarg);
                        break;
                case 's':
                        sponge_layer_start = atof(optarg);
```

```
            break;
        case 'd':
            zs = atof(optarg);
            break;
        case 'D':
            zr = atof(optarg);
            break;
        case 'n':
            num_freqs = atoi(optarg);
            if (num_freqs <= 0 || num_freqs % 2 != 0) {
                fprintf(stderr, "ERROR: Number of
                    frequencies must be a positive, even
                    number\n");
                exit(1);
            }
            break;
        case 'f':
            fft_cutoff = atoi(optarg);
            break;
        case 'x':
            window_width = atoi(optarg);
            break;
        case 'y':
            window_height = atoi(optarg);
            break;
        case 'u':
            z_nudge = atof(optarg);
            break;
        case 'U':
            kz_nudge = atof(optarg);
            break;
        case 'g':
            use_greene = true;
            break;
        case 'q':
            no_display = true;
            break;
        case 'i':
            depth_filename = optarg;
            break;
        case 'j':
            ssp_filename = optarg;
            break;
        case 'k':
            f0_filename = optarg;
            break;
        case 'l':
            results_filename = optarg;
            break;
        case 'm':
            log_filename = optarg;
            break;
        case '1':
```

```
            precision_format = GL_RGBA_FLOAT16_ATI;
            break;
        case '3':
            enable_r300_hack = true;
            break;
        case 't':
            truncate_bits = atoi(optarg);
            break;
        case -1:                    // end of argument list
            return;
        default:                    // --help, or invalid option
            fprintf(stderr, "Usage: gpuwave [OPTION]...\n\n
                ");
            fprintf(stderr, "  -a, --alpha-water=VAL       
                set dampening in water (dB/km), default 
                %f\n", alpha_water);
            fprintf(stderr, "  -A, --alpha-sediment=VAL    
                set dampening in sediment (dB/lambda), 
                default %f\n", alpha_sediment);
            fprintf(stderr, "  -c, --sound-speed-source=VAL
                set sound speed at the source (m/s), 
                default %f\n", c0);
            fprintf(stderr, "  -C, --sound-speed-sediment=
                VAL  set sound speed in sediment (m/s), 
                default %f\n", c_sediment);
            fprintf(stderr, "  -e, --density-water=VAL     
                set density of water (kg/m^3), default %
                f\n", rho_water);
            fprintf(stderr, "  -E, --density-sediment=VAL  
                set density of sediment (kg/m^3), 
                default %f\n", rho_sediment);
            fprintf(stderr, "  -Z, --max-depth=VAL         
                set bottom of simulation area (m), 
                default %f\n", max_depth);
            fprintf(stderr, "  -R, --max-range=VAL         
                set range of simulation area (m), 
                default %f\n", max_range);
            fprintf(stderr, "  -z, --depth-res-order=VAL   
                depth resolution order (depth = 2^z), 
                default %u\n", z_levels);
            fprintf(stderr, "  -r, --range-res-order=VAL   
                range resolution order (range = 2^r), 
                default %u\n", range_levels);
            fprintf(stderr, "  -s, --sponge-layer-start=VAL
                start depth of sponge layer (m), default
                %f\n", sponge_layer_start);
            fprintf(stderr, "  -d, --source-depth=VAL      
                depth of source (m), default %f\n", zs);
            fprintf(stderr, "  -D, --receiver-depth=VAL    
                depth of receiver (m), default %f\n", zr
                );
            fprintf(stderr, "  -n, --num-frequencies=VAL   
                number of frequencies to calculate for, 
```

```
                    default␣%u\n", num_freqs);
                fprintf(stderr, "␣␣-f,␣--fft-cutoff=VAL␣␣␣␣␣␣␣␣
                    ␣␣␣␣FFT␣size␣strategy␣crossover␣point,␣
                    default␣%u\n", fft_cutoff);
                fprintf(stderr, "␣␣-x,␣--window-width=VAL␣␣␣␣␣␣
                    ␣␣␣␣width␣of␣display␣window,␣default␣%u\n",
                    window_width);
                fprintf(stderr, "␣␣-y,␣--window-height=VAL␣␣␣␣␣
                    ␣␣␣␣height␣of␣display␣window,␣default␣%u\n",
                     window_height);
                fprintf(stderr, "␣␣-u,␣--z-nudge=VAL␣␣␣␣␣␣␣␣␣␣␣
                    ␣␣␣␣z␣discretization␣adjustment␣[0..1],␣
                    default␣%f\n", z_nudge);
                fprintf(stderr, "␣␣-U,␣--kz-nudge=VAL␣␣␣␣␣␣␣␣␣␣
                    ␣␣␣␣k_z␣discretization␣adjustment␣[0..1],␣
                    default␣%f\n", kz_nudge);
                fprintf(stderr, "␣␣-g,␣--greene-source␣␣␣␣␣␣␣␣␣
                    ␣␣␣␣Use␣Greene's␣source␣instead␣of␣Gaussian␣
                    source\n");
                fprintf(stderr, "\n");
                fprintf(stderr, "␣␣-i,␣--depth-file=FILENAME␣␣␣␣
                    ␣␣␣␣file␣containing␣depth␣data␣(default␣%s)\
                    n", depth_filename.c_str());
                fprintf(stderr, "␣␣-j,␣--ssp-file=FILENAME␣␣␣␣␣␣
                    ␣␣␣␣file␣containing␣the␣sound␣speed␣profile␣
                    (default␣%s)\n", ssp_filename.c_str());
                fprintf(stderr, "␣␣-k,␣--f0-file=FILENAME␣␣␣␣␣␣␣
                    ␣␣␣␣file␣containing␣source␣frequencies␣(
                    default␣%s)\n", f0_filename.c_str());
                fprintf(stderr, "␣␣-l,␣--results-file=FILENAME␣
                    ␣␣␣␣file␣to␣write␣results␣to␣(default␣%s)\n"
                    , results_filename.c_str());
                fprintf(stderr, "␣␣-m,␣--log-file=FILENAME␣␣␣␣␣␣
                    ␣␣␣␣file␣to␣write␣performance␣log␣to␣(
                    default␣%s)\n", log_filename.c_str());
                fprintf(stderr, "\n");
                fprintf(stderr, "␣␣-q,␣--no-display␣␣␣␣␣␣␣␣␣␣␣␣
                    ␣␣␣␣turn␣off␣display␣(for␣benchmarking)\n");
                fprintf(stderr, "␣␣-1,␣--force-16-bit␣␣␣␣␣␣␣␣␣␣
                    ␣␣␣␣force␣16-bit␣float␣precision\n");
                fprintf(stderr, "␣␣-3,␣--r300-workaround␣␣␣␣␣␣␣␣
                    ␣␣␣␣hack␣refraction␣shader␣for␣R300␣(
                    benchmarking␣only!)\n");
                fprintf(stderr, "␣␣-t,␣--truncate-bits=NUM␣␣␣␣␣␣
                    ␣␣␣␣simulate␣N␣bits␣of␣mantissa␣truncation␣(
                    default␣%u)\n", truncate_bits);
                fprintf(stderr, "␣␣-h,␣--help␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣
                    ␣␣␣␣display␣this␣help␣and␣exit\n");
                exit(1);
            }
        }
}
```

```
int main(int argc, char **argv)
{
        parse_options(argc, argv);
        GLWindow *win = new GLWindow("Underwater␣acoustics␣simulation",
            window_width, window_height, 32, false);

        load_opengl_extensions();

        // these may have changed due to options
        z_steps = 1 << z_levels;
        range_steps = 1 << range_levels;
        delta_z = max_depth / z_steps;
        delta_r = max_range / range_steps;
        delta_kz = 2.0 * M_PI / (2.0 * z_steps * delta_z);

        // allocate memory
        start_field = new float[z_steps * num_freqs/2 * 4];
        depth = new float[range_steps * 4];
        ssp = new float[z_steps * 4];
        f0 = new float[num_freqs];
        support1 = new float[num_freqs/2 * 4];
        support2 = new float[num_freqs/2 * 4];
        delta_r_div_double_k0 = new float[num_freqs/2 * 4];
        twiddle_tex = new GLuint[z_levels];
        twiddle_inv_tex = new GLuint[z_levels];
        twiddle_quads = new std::vector<fft_constant_data> [z_levels];
        twiddle_inv_quads = new std::vector<fft_constant_data> [
           z_levels];

        // load shaders
        fft_shader = new Shader("fft-v.glsl", "fft-f.glsl");
        fft_constant_twiddle_shader = new Shader("fftc-v.glsl", "fftc-f
           .glsl");
        display_shader = new Shader("identity-v.glsl", "display-f.glsl"
           );
        refraction_shader = new Shader("refraction-v.glsl", "refraction
           -f.glsl", enable_r300_hack);
        diffraction_shader = new Shader("diffraction-v.glsl", "
           diffraction-f.glsl");
        identity_shader = new Shader("identity-v.glsl", "identity-f.
           glsl");
        truncate_shader = new Shader("identity-v.glsl", "truncate-f.
           glsl");
        identity_texcoord_shader = new Shader("identity-v.glsl", "
           identity-texcoord-f.glsl");
        shuffle_shader = new Shader("shuffle-v.glsl", "shuffle-f.glsl")
           ;
        ravel_postmultiply_shader = new Shader("ravel-v.glsl", "ravel-
           postmultiply-f.glsl");
        ravel_premultiply_shader = new Shader("ravel-v.glsl", "ravel-
           premultiply-f.glsl");

        // generate our framebuffers
```

```
glGenFramebuffersEXT(1, &fb);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
rtt_tex1 = gen_rtt(precision_format, z_steps, num_freqs/2);
rtt_tex2 = gen_rtt(precision_format, z_steps, num_freqs/2);

glGenFramebuffersEXT(1, &fb_result);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb_result);
result_tex = gen_rtt(precision_format, range_steps, z_steps);

check_error();

{
        GLenum status = glCheckFramebufferStatusEXT(
           GL_FRAMEBUFFER_EXT);
        switch(status) {
        case GL_FRAMEBUFFER_COMPLETE_EXT:
                break;
        case GL_FRAMEBUFFER_UNSUPPORTED_EXT:
                printf("configuration unsupported\n");
                exit(1);
        default:
                printf("unknown status %x\n", status);
                exit(1);
        }
}

// depth (in the ocean sense, not in the OpenGL sense) texture
read_depth(depth_filename.c_str(), depth);
depth_tex = make_texture_1d(range_steps, precision_format,
   GL_RGBA, depth);

// sound speed profile texture
read_ssp(ssp_filename.c_str(), ssp);
ssp_tex = make_texture_1d(z_steps, precision_format, GL_RGBA,
   ssp);

// f0 data
read_f0(f0_filename.c_str(), support1, support2);
support1_tex = make_texture_1d(num_freqs/2, precision_format,
   GL_RGBA, support1);
support2_tex = make_texture_1d(num_freqs/2, precision_format,
   GL_RGBA, support2);
delta_r_div_double_k0_tex = make_texture_1d(num_freqs/2,
   precision_format, GL_RGBA, delta_r_div_double_k0);

// start field texture,
if (use_greene)
        generate_greene(start_field, f0);
else
        generate_gaussian(start_field, f0);
initial_tex = make_texture_2d(z_steps, num_freqs/2,
   precision_format, GL_RGBA, start_field);
```

```
        precompute_twiddle();
        ravel_tex = make_ravel_texture(z_steps);
        unravel_tex = make_unravel_texture(z_steps);
        shuffle_tex = make_shuffle_texture(z_steps);
        unshuffle_tex = make_unshuffle_texture(z_steps);
        exp_forward_tex = make_exp_texture(z_steps, false);
        exp_inverse_tex = make_exp_texture(z_steps, true);

        glDisable(GL_DEPTH_TEST);
        glDisable(GL_BLEND);

        check_precision();

        // input the starting field
        glViewport(0, 0, z_steps, num_freqs/2);
        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
           GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, rtt_tex2, 0);

        identity(initial_tex);
        swap_buffers();
        check_error();

#if WIN32
        DWORD start, end;
        start = timeGetTime();
#else
        struct timeval start, end;
        gettimeofday(&start, NULL);
#endif

        while (!win->is_done()) {
                static unsigned frameno = 0;

                glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
                glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                   GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, rtt_tex2,
                   0);
                glViewport(0, 0, z_steps, num_freqs/2);

                // convert DST to IFFT (preprocessing)
                shuffle(rtt_tex1, shuffle_tex);
                swap_buffers();

                // inverse FFT
                for (unsigned s = 2; s <= z_steps; s *= 2) {
                        if (s < fft_cutoff)
                                fft_constant_twiddle(rtt_tex1, s, true)
                                   ;
                        else
                                fft(rtt_tex1, s, true);
                        swap_buffers();
                }
```

```
                // and postprocessing (forward exp, then unravel) to
                   finish the DST
                ravel(*ravel_premultiply_shader, rtt_tex1, unravel_tex,
                    exp_forward_tex);
                swap_buffers();

                // multiply by diffraction term
                diffraction(rtt_tex1);
                swap_buffers();

                // convert IDST to FFT (preprocessing: ravel, then
                   inverse exp)
                ravel(*ravel_postmultiply_shader, rtt_tex1, ravel_tex,
                   exp_inverse_tex);
                swap_buffers();

                // forward FFT
                for (unsigned s = 2; s <= z_steps; s *= 2) {
                        if (s < fft_cutoff)
                                fft_constant_twiddle(rtt_tex1, s, false
                                   );
                        else
                                fft(rtt_tex1, s, false);
                        swap_buffers();
                }

                // and finally unshuffle to finish the IDST
                shuffle(rtt_tex1, unshuffle_tex);
                swap_buffers();

                refraction(rtt_tex1, frameno);
                swap_buffers();

                if (!no_display) {
                        // store
                        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,
                           fb_result);
                        store(rtt_tex1, frameno);
                        check_error();

                        // and display on screen
                        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
                        disp(result_tex, frameno);

                        win->flip();
                }

                ++frameno;
                if (frameno == range_steps)
                        break;
        }
```

```
        double elapsed;
#if WIN32
        end = timeGetTime();
        elapsed = double(end - start) * 1e-3;
#else
        gettimeofday(&end, NULL);
        elapsed = double(end.tv_sec - start.tv_sec) +
                double(end.tv_usec - start.tv_usec) * 1e-6;
#endif

        glFinish();
        printf("Simulation took %.3f seconds.\n", elapsed);

        write_log(elapsed);
        read_back();

        delete win;
}
```

A.2.2 shader.h

```
#ifndef _SHADER_H
#define _SHADER_H 1

#include <GL/gl.h>
#include <vector>
#include <string>
#include <algorithm>

#ifdef WIN32
#include "glext.h"
#endif

/*
 * A utility class to encapsulate many of the common operations
 * we will need to use in our shaders.
 *
 * Shader::Binding is a kind of cleaning-up class; just create one
 * when you want to use the shader, and when the binding object
 * goes out of scope, it will take care to unbind textures as
 * neccessary.
 */

class Shader
{
public:
        class Binding;

        Shader(const std::string &vs_filename, const std::string &
           fs_filename, bool enable_r300_hack = false);
        ~Shader();

private:
```

```
        void compile_shader(GLuint obj, const std::string &filename);
        void check_linked();

        GLuint prog;
        bool enable_r300_hack;
};

class Shader::Binding
{
        friend class Shader;

private:
        const Shader &s;
        unsigned current_texunit;
        std::vector<std::pair<GLenum, GLenum> > active_bindings;

public:
        Binding(Shader &s);
        ~Binding();
        void bind_texture_1d(const std::string &name, GLuint tex);
        void bind_texture_2d(const std::string &name, GLuint tex);
        void set_uniform(const std::string &name, GLfloat x);   //
           would be overloaded if we needed it
};

#endif /* !defined(_SHADER_H) */
```

A.2.3 shader.cpp

```
#include "shader.h"

extern PFNGLCREATEPROGRAMOBJECTARBPROC glCreateProgramObjectARB;
extern PFNGLCREATESHADEROBJECTARBPROC glCreateShaderObjectARB;
extern PFNGLSHADERSOURCEARBPROC glShaderSourceARB;
extern PFNGLCOMPILESHADERARBPROC glCompileShaderARB;
extern PFNGLGETOBJECTPARAMETERIVARBPROC glGetObjectParameterivARB;
extern PFNGLGETINFOLOGARBPROC glGetInfoLogARB;
extern PFNGLATTACHOBJECTARBPROC glAttachObjectARB;
extern PFNGLDELETEOBJECTARBPROC glDeleteObjectARB;
extern PFNGLLINKPROGRAMARBPROC glLinkProgramARB;
extern PFNGLUSEPROGRAMOBJECTARBPROC glUseProgramObjectARB;
extern PFNGLGETUNIFORMLOCATIONARBPROC glGetUniformLocationARB;
extern PFNGLUNIFORM1IARBPROC glUniform1iARB;
extern PFNGLUNIFORM1FARBPROC glUniform1fARB;
extern PFNGLUNIFORM2FARBPROC glUniform2fARB;
extern PFNGLGETATTRIBLOCATIONARBPROC glGetAttribLocationARB;
extern PFNGLVERTEXATTRIB1FARBPROC glVertexAttrib1fARB;

#if WIN32 || !defined(GL_ARB_multitexture)
extern PFNGLACTIVETEXTUREARBPROC glActiveTextureARB;
#endif

Shader::Shader(const std::string &vs_filename, const std::string &
```

```
    fs_filename, bool enable_r300_hack)
        : enable_r300_hack(enable_r300_hack)
{
        prog = glCreateProgramObjectARB();
        GLuint vs = glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
        GLuint fs = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);

        compile_shader(vs, "shaders/" + vs_filename);
        compile_shader(fs, "shaders/" + fs_filename);
        glAttachObjectARB(prog, vs);
        glAttachObjectARB(prog, fs);
        glDeleteObjectARB(vs);
        glDeleteObjectARB(fs);

        glLinkProgramARB(prog);
        check_linked();
}

Shader::~Shader()
{
//      glDeleteProgramObjectARB(prog);
}

void Shader::compile_shader(GLuint obj, const std::string &filename)
{
        FILE *file = fopen(filename.c_str(), "r");
        if (file == NULL) {
                perror(filename.c_str());
                exit(1);
        }

        GLcharARB *buf = (GLcharARB *)malloc(16384);
        GLint len = fread(buf, 1, 16384, file);

        char define_str[256];
        sprintf(define_str, "#define␣R300_BENCHMARK_WORKAROUND␣%u\n",
            enable_r300_hack);

        const GLcharARB* ptrs[2] = { define_str, buf };
        const GLint lens[2] = { strlen(define_str), len };
        glShaderSourceARB(obj, 2, ptrs, lens);

        glCompileShaderARB(obj);
        free(buf);

        GLint compiled = GL_FALSE;
        GLint maxLength, length;
        glGetObjectParameterivARB(obj, GL_OBJECT_COMPILE_STATUS_ARB, &
            compiled);
        glGetObjectParameterivARB(obj, GL_OBJECT_INFO_LOG_LENGTH_ARB, &
            maxLength);
        GLcharARB *pInfoLog = (GLcharARB *) malloc(maxLength * sizeof(
            GLcharARB));
```

```
        glGetInfoLogARB(obj, maxLength, &length, pInfoLog);

        if (!compiled) {
                printf("Compile␣of␣'%s'␣failed:␣%s\n", filename.c_str()
                    , pInfoLog);
                exit(1);
        }
        free(pInfoLog);
}

void Shader::check_linked()
{
        GLint linked = GL_FALSE;
        GLint maxLength, length;
        glGetObjectParameterivARB(prog, GL_OBJECT_LINK_STATUS_ARB, &
            linked);
        glGetObjectParameterivARB(prog, GL_OBJECT_INFO_LOG_LENGTH_ARB,
            &maxLength);
        GLcharARB *pInfoLog = (GLcharARB *) malloc(maxLength * sizeof(
            GLcharARB));
        glGetInfoLogARB(prog, maxLength, &length, pInfoLog);

        if (!linked) {
                printf("%s\n", pInfoLog);
                exit(1);
        }
        free(pInfoLog);
}

Shader::Binding::Binding(Shader &s)
        : s(s), current_texunit(0)
{
        glUseProgramObjectARB(s.prog);
}

Shader::Binding::~Binding()
{
        for (std::vector<std::pair<GLenum, GLenum> >::const_iterator i
            = active_bindings.begin(); i != active_bindings.end(); ++i)
            {
                glActiveTextureARB(i->second);
                glDisable(i->first);
        }
        glActiveTextureARB(GL_TEXTURE0_ARB);
}

void Shader::Binding::bind_texture_1d(const std::string &name, GLuint
    tex)
{
        glActiveTextureARB(GL_TEXTURE0_ARB + current_texunit);
        glEnable(GL_TEXTURE_1D);
        glBindTexture(GL_TEXTURE_1D, tex);
        glUniform1iARB(glGetUniformLocationARB(s.prog, name.c_str()),
```

```
            current_texunit);

        active_bindings.push_back(std::make_pair(GL_TEXTURE_1D,
            GL_TEXTURE0_ARB + current_texunit));
        ++current_texunit;
}

void Shader::Binding::bind_texture_2d(const std::string &name, GLuint
    tex)
{
        glActiveTextureARB(GL_TEXTURE0_ARB + current_texunit);
        glEnable(GL_TEXTURE_2D);
        glBindTexture(GL_TEXTURE_2D, tex);
        glUniform1iARB(glGetUniformLocationARB(s.prog, name.c_str()),
            current_texunit);

        active_bindings.push_back(std::make_pair(GL_TEXTURE_2D,
            GL_TEXTURE0_ARB + current_texunit));
        ++current_texunit;
}

void Shader::Binding::set_uniform(const std::string &name, GLfloat x)
{
        glUniform1fARB(glGetUniformLocationARB(s.prog, name.c_str()), x
            );
}
```

A.2.4 glwindow.h

```
#ifndef _GLWINDOW_H
#define _GLWINDOW_H

#ifdef WIN32
#include <windows.h>
#endif

#ifdef __linux__
#include <GL/glx.h>
#include <X11/extensions/xf86vmode.h>
#include <X11/keysym.h>
#endif

#include <GL/gl.h>
#include <GL/glu.h>

#define         GLX_SAMPLE_BUFFERS_ARB                100000
#define         GLX_SAMPLES_ARB                       100001

class WGL_RTT;
class PBuffer_RTT;
class Framebuffer_RTT;

class GLWindow {
```

```
public:
        GLWindow(char *title, int width, int height, int bpp, bool
           fullscreen);
        ~GLWindow();
        void resize(int x, int y, int w, int h);
        void flip();
        bool is_done();

#if __linux__
        Display *dpy;
#endif

protected:
#ifdef WIN32
        HDC hDC;
        HGLRC hRC;
        HWND hWnd;
        HINSTANCE hInstance;
#endif
#ifdef __linux__
        int screen;
        Window win;
        GLXContext ctx;
        XSetWindowAttributes attr;
        Bool fs;
        XF86VidModeModeInfo deskMode;
#endif

        char *title;
        bool fullscreen;
        int x, y;
        unsigned int width, height;
        unsigned int bpp;
        int zbuffer;
        bool done;
        void initGL();
};

#endif
```

A.2.5 glwindow.cpp

```
#include <stdio.h>

#ifdef WIN32
#include <windows.h>
#endif

#ifdef __linux__
#include <unistd.h>
#include <GL/glx.h>
#include <X11/extensions/xf86vmode.h>
#include <X11/keysym.h>
```

```
#endif

#include <GL/gl.h>
#include <GL/glu.h>
#include <stdexcept>

#include "glwindow.h"

#ifdef WIN32
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
#endif

GLWindow *curr_win;

#define XASPECT 4.0f
#define YASPECT 3.0f

void GLWindow::resize(int x, int y, int w, int h)
{
        /* Prevent division by zero */
        if (h == 0) {
                h = 1;
        }

        float aspect = (float)w / (float)h;
        if (aspect > XASPECT / YASPECT) {
                int new_w = (int)((float)h * XASPECT / YASPECT);
                x += (w - new_w) / 2;
                w = new_w;
        } else if (aspect < XASPECT / YASPECT) {
                int new_h = (int)((float)w * YASPECT / XASPECT);
                y += (h - new_h) / 2;
                h = new_h;
        }

        glViewport(x, y, w, h);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(53.0f, (GLfloat)w / (GLfloat)h, 1.0f, 500.0f);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

#ifdef __linux__
//      XClearWindow(this->dpy, this->win);
#endif
}

GLWindow::GLWindow(char *title, int width, int height, int bpp, bool
   fullscreen)
{
        curr_win = this;
#ifdef WIN32
```

```
        GLuint PixelFormat;
        WNDCLASS wc;
        DWORD dwExStyle;
        DWORD dwStyle;
        DEVMODE dmScreenSettings;

        memset(&dmScreenSettings, 0, sizeof(dmScreenSettings));
        dmScreenSettings.dmSize = sizeof(dmScreenSettings);
        dmScreenSettings.dmPelsWidth = width;
        dmScreenSettings.dmPelsHeight = height;
        dmScreenSettings.dmBitsPerPel = bpp;
        dmScreenSettings.dmFields = DM_BITSPERPEL | DM_PELSWIDTH |
           DM_PELSHEIGHT;

        RECT WindowRect;
        WindowRect.left = (long)0;
        WindowRect.right = (long)width;
        WindowRect.top = (long)0;
        WindowRect.bottom = (long)height;

#endif /* WIN32 */
#ifdef __linux__
        XVisualInfo *vi;
        int dpyWidth = 0, dpyHeight = 0;
        int i;
        XF86VidModeModeInfo **modes;
        int modeNum;
        int bestMode;
        Atom wmDelete;
        Window winDummy;
        unsigned int borderDummy;

        static int attrList[] = {
                GLX_RGBA,
                GLX_RED_SIZE, 1,
                GLX_GREEN_SIZE, 1,
                GLX_BLUE_SIZE, 1,
                GLX_DOUBLEBUFFER,
                None
        };
#endif /* __linux__ */

        this->x = 0;
        this->y = 0;
        this->width = width;
        this->height = height;
        this->bpp = bpp;
        this->fullscreen = fullscreen;
        this->zbuffer = zbuffer;
        this->done = 0;

#ifdef WIN32
        this->hInstance = GetModuleHandle(NULL);
```

```
        wc.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
        wc.lpfnWndProc = WndProc;
        wc.cbClsExtra = 0;
        wc.cbWndExtra = 0;
        wc.hInstance = this->hInstance;
        wc.hIcon = NULL;
        wc.hCursor = NULL;
        wc.hbrBackground = NULL;
        wc.lpszMenuName = NULL;
        wc.lpszClassName = "Excess-OGL";

        if( !RegisterClass(&wc) ) throw new std::runtime_error("Couldn'
           t␣register␣Window␣Class");

#endif /* WIN32 */
#ifdef __linux__
        /* set best mode to current */
        bestMode = 0;

        /* get a connection */
        this->dpy = XOpenDisplay(0);
        this->screen = DefaultScreen(this->dpy);

        if (fullscreen) {
                XF86VidModeGetAllModeLines(this->dpy, this->screen, &
                   modeNum, &modes);

                /* save desktop-resolution before switching modes */
                this->deskMode = *modes[0];

                /* look for mode with requested resolution */
                for (i = 0; i < modeNum; i++) {
                        if ((modes[i]->hdisplay == width) && (modes[i
                           ]->vdisplay == height)) {
                                bestMode = i;
                        }
                }

                /* if we don't have it, bomb out */
                if (bestMode == 0 && (modes[0]->hdisplay != width ||
                   modes[0]->vdisplay != height)) {
                        throw new std::runtime_error("Couldn't␣set␣
                           requested␣screen␣mode.");
                }
        }

        /* get an appropriate visual */
        vi = glXChooseVisual(this->dpy, this->screen, attrList);
        if (vi == NULL) {
                throw new std::runtime_error("Couldn't␣get␣double-
                   buffered␣visual");
        }
```

```
        /* create a GLX context */
        this->ctx = glXCreateContext(this->dpy, vi, NULL, GL_TRUE);

        /* create a color map (umm, needed?) */
        Colormap cmap = XCreateColormap(this->dpy, RootWindow(this->dpy
            , vi->screen),
                vi->visual, AllocNone);
        this->attr.colormap = cmap;

        /* make a blank cursor */
        {
                static char data[1] = {0};
                Cursor cursor;
                Pixmap blank;
                XColor dummy;

                blank = XCreateBitmapFromData(this->dpy, RootWindow(
                    this->dpy, vi->screen), data, 1, 1);
                if (blank == None)
                        throw new std::runtime_error("Out␣of␣memory!");
                cursor = XCreatePixmapCursor(this->dpy, blank, blank, &
                    dummy, &dummy, 0, 0);
                XFreePixmap(this->dpy, blank);
                this->attr.cursor = cursor;
        }

        this->attr.border_pixel = 0;
#endif /* __linux__ */

        /* change screen mode */
        if (fullscreen) {
#ifdef WIN32
                if (ChangeDisplaySettings(&dmScreenSettings,
                    CDS_FULLSCREEN) != DISP_CHANGE_SUCCESSFUL) {
                        throw new std::runtime_error("Couldn't␣set␣
                            requested␣screen␣mode.");
                }
#endif /* WIN32 */
#ifdef __linux__
                XF86VidModeSwitchToMode(this->dpy, this->screen, modes[
                    bestMode]);
                XF86VidModeSetViewPort(this->dpy, this->screen, 0, 0);
                dpyWidth = modes[bestMode]->hdisplay;
                dpyHeight = modes[bestMode]->vdisplay;
                XFree(modes);
#endif /* __linux__ */
        }

        /* create the window */
#ifdef WIN32
        if (fullscreen) {
                dwExStyle = WS_EX_APPWINDOW;
                dwStyle = WS_POPUP;
```

```
                ShowCursor(FALSE);
        } else {
                dwExStyle = WS_EX_APPWINDOW | WS_EX_WINDOWEDGE;
                dwStyle = WS_OVERLAPPEDWINDOW;
        }

        AdjustWindowRectEx(&WindowRect, dwStyle, FALSE, dwExStyle);

        if (!(hWnd = CreateWindowEx(dwExStyle,
                                    "Excess-OGL",
                                    title,
                                    dwStyle | WS_CLIPSIBLINGS |
                                       WS_CLIPCHILDREN,
                                    0, 0,
                                    WindowRect.right - WindowRect.left,
                                    WindowRect.bottom - WindowRect.top,
                                    NULL,
                                    NULL,
                                    this->hInstance,
                                    NULL))) {
                delete this;
                throw new std::runtime_error("Could not change
                   screenmode");
        }
#endif
#ifdef __linux__
        this->attr.background_pixel = 0;

        if (fullscreen) {
                /* create a fullscreen window */
                this->attr.override_redirect = True;
                this->attr.event_mask = KeyPressMask | ButtonPressMask
                   |
                                        StructureNotifyMask;

                this->win = XCreateWindow(this->dpy, RootWindow(this->
                   dpy, vi->screen),
                        0, 0, dpyWidth, dpyHeight, 0, vi->depth,
                           InputOutput, vi->visual,
                        CWColormap | CWCursor | CWEventMask |
                           CWOverrideRedirect,
                        &this->attr);
                XWarpPointer(this->dpy, None, this->win, 0, 0, 0, 0, 0,
                    0);
                XMapRaised(this->dpy, this->win);
                XGrabKeyboard(this->dpy, this->win, True, GrabModeAsync
                   ,
                        GrabModeAsync, CurrentTime);
                XGrabPointer(this->dpy, this->win, True,
                   ButtonPressMask,
                        GrabModeAsync, GrabModeAsync, this->win, None,
                           CurrentTime);
        } else {
```

```
                /* create a window in window mode*/
                this->attr.event_mask = KeyPressMask | ButtonPressMask
                    |
                        StructureNotifyMask;
                this->win = XCreateWindow(this->dpy, RootWindow(this->
                    dpy, vi->screen),
                        0, 0, width, height, 0, vi->depth, InputOutput,
                            vi->visual,
                        CWColormap | CWBorderPixel | CWEventMask, &this
                            ->attr);

                /* only set window title and handle wm_delete_events if
                     in windowed mode */
                wmDelete = XInternAtom(this->dpy, "WM_DELETE_WINDOW",
                    True);
                XSetWMProtocols(this->dpy, this->win, &wmDelete, 1);
                XSetStandardProperties(this->dpy, this->win, title,
                        title, None, NULL, 0, NULL);
                XMapRaised(this->dpy, this->win);
        }
#endif /* __linux__ */

#ifdef WIN32
        static PIXELFORMATDESCRIPTOR pfd = {
                sizeof(PIXELFORMATDESCRIPTOR),
                1,
                PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
                    PFD_DOUBLEBUFFER,
                PFD_TYPE_RGBA,
                bpp,
                0, 0, 0, 0, 0, 0,
                0,
                0,
                0,
                0, 0, 0, 0,
                zbuffer,
                8,
                0,
                PFD_MAIN_PLANE,
                0,
                0, 0, 0
        };

        hDC = GetDC(hWnd);
        PixelFormat = ChoosePixelFormat(hDC, &pfd);
        if (PixelFormat == 0) {
                throw new std::runtime_error("Could␣not␣find␣a␣usable␣
                    pixelformat");
        }
        SetPixelFormat(hDC, PixelFormat, &pfd);
        hRC = wglCreateContext(hDC);
        wglMakeCurrent(hDC, hRC);
        ShowWindow(hWnd, SW_SHOW);
```

```
        SetForegroundWindow(hWnd);
        SetFocus(hWnd);

        SetPriorityClass(GetCurrentProcess(), HIGH_PRIORITY_CLASS);
#endif /* WIN32 */
#ifdef __linux__
        /* connect the glx-context to the window */
        glXMakeCurrent(this->dpy, this->win, this->ctx);
        XClearWindow(this->dpy, this->win);
        XGetGeometry(this->dpy, this->win, &winDummy, &this->x, &this->
           y,
                &this->width, &this->height, &borderDummy, &this->bpp);
        if (!glXIsDirect(this->dpy, this->ctx)) {
                throw new std::runtime_error("No␣direct␣rendering␣(
                   hardware␣acceleration)␣available!");
        }

//      nice(-7);
#endif /* __linux__ */

        this->resize(0, 0, this->width, this->height);
}

GLWindow::~GLWindow()
{
#ifdef __linux__
        if (this->ctx) {
                if (!glXMakeCurrent(this->dpy, None, NULL)) {
                        throw new std::runtime_error("Could␣not␣release
                           ␣drawing␣context.");
                }
                glXDestroyContext(this->dpy, this->ctx);
                this->ctx = NULL;
        }
#endif

        if (fullscreen) {
#ifdef __linux__
                XF86VidModeSwitchToMode(this->dpy, this->screen, &this
                   ->deskMode);
                XF86VidModeSetViewPort(this->dpy, this->screen, 0, 0);
#endif
#ifdef WIN32
                ChangeDisplaySettings(NULL,0);
                ShowCursor(TRUE);
#endif
        }

#ifdef __linux__
        XCloseDisplay(this->dpy);
#endif

#ifdef WIN32
```

```
        if (hRC) {
                wglMakeCurrent(NULL, NULL);
                wglDeleteContext(hRC);
                hRC = NULL;
        }

        if (hDC != NULL && ReleaseDC(hWnd, hDC)) hDC = NULL;
        if (hWnd != NULL && DestroyWindow(hWnd)) hWnd = NULL;
        UnregisterClass("Excess-OGL", hInstance);
#endif
}

void GLWindow::flip()
{
#ifdef WIN32
        MSG msg;
        if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
                if (msg.message == WM_QUIT) {
                        this->done = TRUE;
                } else {
                        TranslateMessage(&msg);
                        DispatchMessage(&msg);
                }
        }
        SwapBuffers(this->hDC);
#endif
#ifdef __linux__
        glXSwapBuffers(this->dpy, this->win);
#endif
}

bool GLWindow::is_done()
{
        return this->done;
}

#ifdef WIN32
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM
   lParam)
{
        switch (uMsg) {
        case WM_SYSCOMMAND:
                switch (wParam) {
                case SC_SCREENSAVE:
                case SC_MONITORPOWER:
                        return 0;
                }
                break;

        case WM_CLOSE:
                PostQuitMessage(0);
                return 0;
```

```
        case WM_KEYUP:
                if (wParam == VK_ESCAPE)
                        PostQuitMessage(0);
                return 0;

        case WM_SIZE:
                curr_win->resize(0, 0, LOWORD(lParam), HIWORD(lParam));
                return 0;
        }

        return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
#endif //WIN32
```

A.2.6 shaders/diffraction-v.glsl

```
varying vec2 tc;
varying float k_z;
varying float f0_coord;

void main(void)
{
        gl_Position = ftransform();
        tc = gl_MultiTexCoord0.xy;
        k_z = gl_MultiTexCoord0.z;
        f0_coord = gl_MultiTexCoord1.x;
}
```

A.2.7 shaders/diffraction-f.glsl

```
uniform sampler2D tex;
uniform sampler1D delta_r_div_double_k0_tex;
uniform float scalefac; // compensates for the FFT scaling properties
varying vec2 tc;
varying float k_z;
varying float f0_coord;

vec2 complex_multiply(in vec2 a, in vec2 b)
{
        return a.x * b + a.y * vec2(-b.y, b.x);
}

// computes exp(-zi)
vec2 exp_m(float z)
{
        return vec2(cos(z), -sin(z));
}

void main(void)
{
        vec4 color = texture2D(tex, tc);
        vec2 drk0 = texture1D(delta_r_div_double_k0_tex, f0_coord).xz;
        gl_FragColor.rg = scalefac * complex_multiply(exp_m(k_z * k_z *
            drk0.x), color.rg);
```

```
        gl_FragColor.ba = scalefac * complex_multiply(exp_m(k_z * k_z *
            drk0.y), color.ba);
}
```

A.2.8 shaders/display-f.glsl

```
uniform sampler2D tex;

void main(void)
{
        vec4 color = texture2D(tex, gl_TexCoord[0].st);
        float luma = length(color.xy) * 10.0;

        if (luma <= 0.0) {
                gl_FragColor = vec4(0.0, 0.0, 1.0, 1.0);
        } else if (luma <= 1.0) {
                gl_FragColor = vec4(0.0, luma, 1.0, 1.0);
        } else if (luma <= 2.0) {
                gl_FragColor = vec4(0.0, 1.0, 2.0 - luma, 1.0);
        } else if (luma <= 3.0) {
                gl_FragColor = vec4(luma - 2.0, 1.0, 0.0, 1.0);
        } else if (luma <= 4.0) {
                gl_FragColor = vec4(1.0, 4.0 - luma, 0.0, 1.0);
        } else {
                gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
        }
}
```

A.2.9 shaders/fft-v.glsl

```
void main(void)
{
        gl_Position = ftransform();
        gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

A.2.10 shaders/fft-f.glsl

```
uniform sampler2D tex;
uniform sampler1D twiddle_tex;

void main(void)
{
        vec4 support = texture1D(twiddle_tex, gl_TexCoord[0].s);
        vec4 c1 = texture2D(tex, vec2(support.x, gl_TexCoord[0].t));
        vec4 c2 = texture2D(tex, vec2(support.y, gl_TexCoord[0].t));
        gl_FragColor = c1 + support.z * c2 + support.w * vec4(-c2.y, c2
            .x, -c2.w, c2.z);
}
```

A.2.11 shaders/fftc-v.glsl

```
varying vec2 s1_coord, s2_coord;
varying float twiddle_real, twiddle_imag;
```

```
void main(void)
{
        gl_Position = ftransform();
        s1_coord = gl_MultiTexCoord0.xy;
        s2_coord = gl_MultiTexCoord1.xy;
        twiddle_real = gl_MultiTexCoord2.x;
        twiddle_imag = gl_MultiTexCoord2.y;
}
```

A.2.12 shaders/fftc-f.glsl

```
uniform sampler2D tex;
varying vec2 s1_coord, s2_coord;
varying float twiddle_real, twiddle_imag;

void main(void)
{
        vec4 c1 = texture2D(tex, s1_coord);
        vec4 c2 = texture2D(tex, s2_coord);
        gl_FragColor = c1 + twiddle_real * c2 + twiddle_imag * vec4(-c2
           .y, c2.x, -c2.w, c2.z);
}
```

A.2.13 shaders/identity-v.glsl

```
void main(void)
{
        gl_Position = ftransform();
        gl_TexCoord[0] = gl_MultiTexCoord0;
        gl_TexCoord[1] = gl_MultiTexCoord1;
}
```

A.2.14 shaders/identity-texcoord-f.glsl

```
void main(void)
{
        gl_FragColor = gl_TexCoord[0];
}
```

A.2.15 shaders/identity-f.glsl

```
uniform sampler2D tex;

void main(void)
{
        vec4 color = texture2D(tex, gl_TexCoord[0].st);
        gl_FragColor = color;
}
```

A.2.16 shaders/ravel-v.glsl

```
varying vec2 tex_coord;
```

```
void main(void)
{
        gl_Position = ftransform();
        tex_coord = gl_MultiTexCoord0.xy;
}
```

A.2.17 shaders/ravel-premultiply-f.glsl

```
uniform sampler2D tex;
uniform sampler1D ravel_data_tex;
uniform sampler1D exp_data_tex;
varying vec2 tex_coord;

/* from [a,b,c,d] and [e,f], computes (a + bi)(e + fi) and (c + di)(e +
    fi) */
vec4 complex_multiply(in vec4 a, in vec2 b)
{
        return b.x * a + b.y * vec4(-a.y, a.x, -a.w, a.z);
}

void main(void)
{
        vec4 dd = texture1D(ravel_data_tex, tex_coord.x);

        // we do some extra work here, but it should be better than an
           extra pass
        vec4 eda = texture1D(exp_data_tex, dd.x);
        vec4 edb = texture1D(exp_data_tex, dd.y);

        vec2 ta = vec2(dd.x, tex_coord.y);
        vec2 tb = vec2(dd.y, tex_coord.y);

        // compute (f1 * x[x1]) and (f2 * x[x2])
        vec4 p1 = dd.z * complex_multiply(texture2D(tex, ta), eda.xy);
        vec4 p2 = dd.w * complex_multiply(texture2D(tex, tb), edb.xy);

        // compute (f1 * x[x1]) + i (f2 x[x2])
        gl_FragColor = p1 + vec4(-p2.y, p2.x, -p2.w, p2.z);
}
```

A.2.18 shaders/ravel-postmultiply-f.glsl

```
uniform sampler2D tex;
uniform sampler1D ravel_data_tex;
uniform sampler1D exp_data_tex;
varying vec2 tex_coord;

/* from [a,b,c,d] and [e,f], computes (a + bi)(e + fi) and (c + di)(e +
    fi) */
vec4 complex_multiply(in vec4 a, in vec2 b)
{
        return b.x * a + b.y * vec4(-a.y, a.x, -a.w, a.z);
}
```

```
void main(void)
{
        vec4 dd = texture1D(ravel_data_tex, tex_coord.x);
        vec4 ed = texture1D(exp_data_tex, tex_coord.x);

        vec2 ta = vec2(dd.x, tex_coord.y);
        vec2 tb = vec2(dd.y, tex_coord.y);

        // compute (f1 * x[x1]) and (f2 * x[x2])
        vec4 p1 = dd.z * texture2D(tex, ta);
        vec4 p2 = dd.w * texture2D(tex, tb);

        // compute (f1 * x[x1]) + i (f2 x[x2])
        vec4 f = p1 + vec4(-p2.y, p2.x, -p2.w, p2.z);

        // and postmultiply in the exponential
        gl_FragColor = complex_multiply(f, ed.xy);
}
```

A.2.19 shaders/refraction-v.glsl

```
varying vec2 tex_coord;
varying float f0_coord;
varying float z;
varying float z_coord;

void main(void)
{
        gl_Position = ftransform();
        tex_coord = gl_MultiTexCoord0.xy;
        f0_coord = gl_MultiTexCoord1.x;
        z = gl_MultiTexCoord2.x;
        z_coord = gl_MultiTexCoord2.y;
}
```

A.2.20 shaders/refraction-f.glsl

```
uniform sampler2D tex;
varying vec2 tex_coord;

uniform sampler1D depth;
uniform sampler1D ssp;              // sound speed profile
uniform sampler1D support1;         // [alpha_water,alpha_sediment] for
    both f0
uniform sampler1D support2;         // [k0,half_inv_k0_squared]     for
    both f0
uniform float alpha_sponge;
uniform float c_sediment;
uniform float c0_div_c_sediment_m1; // (c0 / c_sediment) - 1
uniform float rho_average;          // 1/2 (rho_sediment + rho_water)
uniform float half_rho_difference;  // 1/2 (rho_sediment - rho_water)
uniform float c0;
uniform float r;
uniform float r_coord;
```

```
uniform float sponge_layer_start;
uniform float sponge_layer_end;
uniform float delta_r;
uniform float three_inv_sponge_len;   // 3.0 / (sponge_layer_end -
    sponge_layer_start)
varying float z;
varying float z_coord;
varying float f0_coord;

vec2 complex_multiply(in vec2 a, in vec2 b)
{
        return a.x * b.xy + a.y * vec2(-b.y, b.x);
}

// computes exp(i(x + yi))
vec2 exp_p(vec2 z)
{
        float e = exp(-z.y);
        return e * vec2(cos(z.x), sin(z.x));
}

void main(void)
{
        vec4 color = texture2D(tex, tex_coord);
        vec4 n2;
        vec4 supp1 = texture1D(support1, f0_coord);
        vec4 supp2 = texture1D(support2, f0_coord);

        if (z >= sponge_layer_start) {
                float temp = (z - sponge_layer_start) *
                    three_inv_sponge_len;

                n2.xz = vec2(c0_div_c_sediment_m1);
                n2.yw = vec2(alpha_sponge * exp(temp*temp));
        } else {
                vec2 alpha;
                float c = texture1D(ssp, z_coord).x;
                float D = texture1D(depth, r_coord).x;

                //
                // The compiler will normally change this kind of
                    construct
                // from branches to conditional writes -- no need to
                    get ugly
                // with mix() and step().
                //
                if (z >= D) {
                        c = c_sediment;
                        alpha = supp1.yw;
                } else {
                        alpha = supp1.xz;
                }
```

```
                // density
                vec2 tanh_arg_double = 2.0 * supp2.xz * (z - D);

                // this is needed to keep the exp() from growing
                   without bounds --
                // needed on some cards
                tanh_arg_double = clamp(tanh_arg_double, -20.0, 20.0);

                vec2 th;
                {
                        float e2x = exp(tanh_arg_double.x);         //
                           exp(2x)
                        vec2 temp = vec2(e2x) + vec2(-1.0, 1.0);    //
                           exp(2x)-1, exp(2x)+1
                        th.x = temp.x / temp.y;                     //
                           tanh(x)
                }
                {
#if R300_BENCHMARK_WORKAROUND
                        // use this version for the R300 -- it gives
                           wrong answers, but the
                        // computational workload is comparable, and
                           the correct version
                        // segfaults the GLSL compiler
                        float e2x = exp(tanh_arg_double.x);         //
                           exp(2x)
#else
                        float e2x = exp(tanh_arg_double.y);         //
                           exp(2x)
#endif
                        vec2 temp = vec2(e2x) + vec2(-1.0, 1.0);    //
                           exp(2x)-1, exp(2x)+1
                        th.y = temp.x / temp.y;                     //
                           tanh(x)
                }
                vec2 th_dx = 1.0 - th*th;               // 1 - tanh(x)^2
                vec2 th_dxdx = -2.0 * th * th_dx;       // -2 tanh(x) (1
                   - tanh(x)^2)

                // note that (tanh(A*x))' = A*tanh(A*x), thus the extra
                    muls by supp2.xz
                vec2 rho = vec2(rho_average) + vec2(half_rho_difference
                   ) * th;
                vec2 rho_dx = supp2.xz * vec2(half_rho_difference) *
                   th_dx;
                vec2 rho_dxdx = supp2.xz * supp2.xz * vec2(
                   half_rho_difference) * th_dxdx;
                vec2 rho_inv = 1.0 / rho;

                // now finally calculate n^2
                float n = c0 / c;
                n = n * n;
```

```
                n2.xz = n + supp2.yw * rho_inv * (rho_dxdx - 1.5 *
                    rho_inv * rho_dx * rho_dx) - 1.0;
                n2.yw = n * alpha;
        }

        vec4 A = n2 * delta_r * supp2.xxzz;
        gl_FragColor.rg = complex_multiply(exp_p(A.xy), color.rg);
        gl_FragColor.ba = complex_multiply(exp_p(A.zw), color.ba);
}
```

A.2.21 shaders/shuffle-v.glsl

```
varying vec2 tex_coord;

void main(void)
{
        gl_Position = ftransform();
        tex_coord = gl_MultiTexCoord0.xy;
}
```

A.2.22 shaders/shuffle-f.glsl

```
uniform sampler2D tex;
uniform sampler1D shuffle_data;
varying vec2 tex_coord;

void main(void)
{
        vec4 sd = texture1D(shuffle_data, tex_coord.x);

        vec2 s = vec2(sd.x, tex_coord.y);
        gl_FragColor = sd.y * texture2D(tex, s);
}
```

A.2.23 shaders/truncate-f.glsl

```
/*
 * Simulates loss of precision in the mantissa. Strictly speaking, a
    GLSL compiler
 * _could_ attempt to optimize the "+x-x" part away (it isn't
    particularily
 * well defined), but it should not happen in practice, and it's easy
    to detect.
 */

uniform sampler2D tex;
uniform float scalefac;    // 2^x - 1, where x is the number of bits to
    lose

void main(void)
{
        vec4 color = texture2D(tex, gl_TexCoord[0].st);
        vec4 x = color * scalefac;
        gl_FragColor = color + x - x;
```

```
}
```