Interactive Underwater Acoustic Simulator with Ray Tracing

Christopher Kitras, Ashton Palacios
Department of Electrical and Computer Engineering, Brigham Young University, Provo, UT, USA
Email: {chkitras, apal6981}@byu.edu

I. Background

Underwater communication is immensely more difficult than terrestrial communication. In terrestrial communication, higher frequencies of electromagnetic radiation can be used to encode information in higher data rates. The data rate underwater is lower because the medium dictates using sound as the carrier frequency, in the range of kilohertz to hundreds of kilohertz. Using sound waves in water comes with additional difficulties. These difficulties arise from the properties of the water column not being homogeneous. There can be different temperatures, salinity, and amounts of particulates at varying levels in the water column, among other factors. All of these factors change the speed of sound at the different layers. Having differing speeds of sound causes haves to bend and refract leading to destructive interference. To design waveforms and algorithms around these difficulties to communicate underwater, mathematical models are used to simulate underwater wave propagation.

Many different mathematical models have been developed over the past fifty years to model how sound waves move through underwater environments. A popular one that is still used today is the BELLHOP algorithm. The BELLHOP algorithm is a beam tracing algorithm that approximates the underwater pressure fields caused by acoustic propagation that is defined by many variables. Many of these variables can be manipulated by a user such as the altimetry, bathymetry, sound profile speed, and the receiver and transmitter locations. These parameters, and many more, are manipulated via configuration files that are loaded by the program at run time. The configuration files need to be updated by hand if any parameters want to be changed. The manual editing of these files makes using the BELLHOP model cumbersome and slow. Figure 1 is from the BELLHOP Manual and User's Guide and shows the many features that are wrapped up in the model.

Originally, the algorithm was implemented in the popular math modeling language FORTRAN [1] and has been given Python bindings [2] to make it more easily integrated into rapidly developed projects/scripts. The Python bindings has lead to a few projects to visualize the output from the BELLHOP model. The already compiled model is run and Python does the visualization. This is currently the state of the art. Files need to be manually adjusted and run through the model before they can be visualized. Recently, more optimized versions of this algorithm and its behavior have been implemented in both C++ and CUDA [3]. Our project aims at using the optimized C++ and CUDA implementation to make an interactive GUI to update the configurations and visualize the outputs in real-time.

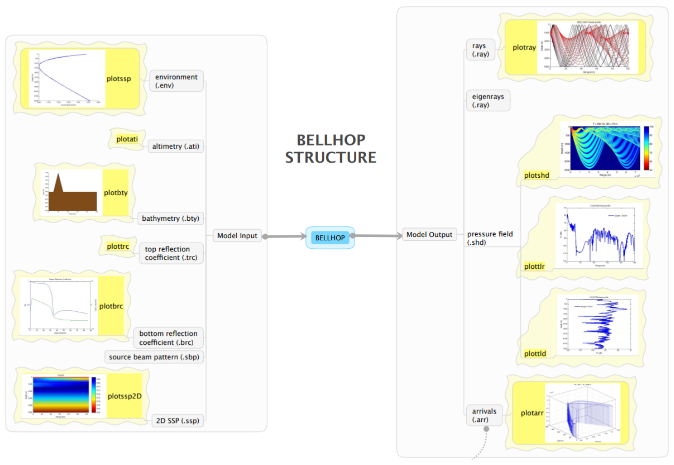


Fig. 1: Features of the BELLHOP model.

II. Project Goals

The goal of the project is to create an interactive GUI using OpenGL to run the BELLHOP continuously and simulate underwater communication. Our research lab is starting a new project involving underwater robotic peer-to-peer networks. To test our developed algorithms and protocols we need a simulator that can easily update parameters. Our semester project is the start of this endeavor.

To stay within the scope of the semester project, we focus on the ray and eigenray broadcast functionality and updating the speed of sound in the water column. Implementing the broadcast functions allows us to visually see how the waves are propagating through our environment. This functionality allows the optimal placement of robots in an environment with the best communication signal strength. In addition to visually seeing the rays, we also can the speed of sound at different levels of the water column. Many factors of underwater environments cause different speeds of sound at different layers. These changes cause the acoustic waves to bounce and behave differently at the different layers. This also can inform users in optimally placing robots in an environment. Our final project goal was to create an OpenGL application integrated within the C++/CUDA implementation of the BELLHOP model. OpenGL has the capability of running on a CPU or GPU. This allows us to

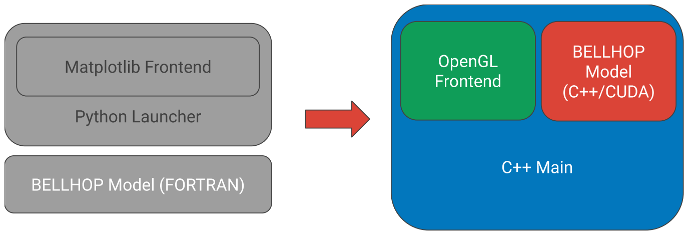


Fig. 2: Comparison of old BELLHOP visualization and our solution

not only benchmark the BELLHOP implementation but also our interactive GUI.

As an aside, the original goal we had was to exploit the beam-tracing functionality of the algorithm and create an implementation that would run on the RTX cores of an NVIDIA graphics card. This would have been done through the OptiX API [4] which provides access to these specific cores. Upon closer review of the OptiX library, we determined that implementation of the BELLHOP algorithm would not be a feasible goal without substantial knowledge of not only the OptiX API but also a deep understanding of how the BELLHOP model deals with the underwater physics propagating acoustic waves.

III. Approach

Our project is split into two portions: the BELLHOP algorithm and the OpenGL GUI. The BELLHOP portion of the project involved understanding the implementation and how to integrate it into our project. The project has dozens of files that are intractably intertwined. The project outlines an input parameter struct and an output parameter struct. Through much investigation, trial, and error, we discovered how to adjust the run mode and speed-of-sound profile. We can effectively tell the algorithm to switch between normal and eigenray runs. We can also modify the speed-of-sound profile for the environment on the fly.

Our biggest optimization from the BELLHOP computation standpoint is that we don't simply use a precompiled binary with manipulated input configuration files. We are integrating our code directly into the BELLHOP implementation. This saves time by avoiding file I/O overhead that would be used to update and process the input and outputs of the algorithm. Integrating our code into the C++/CUDA project was no easy feat. We had to deeply explore, through trial and error, how to manipulate the parameters without losing performance or accuracy. While injecting our code was challenging, the most difficult integration for this portion of the project was updating the CMake files to correctly compile and link the OpenGL libraries. The C++/CUDA project uses complex CMake extensively to compile the project, but we were able to make the necessary adjustments to compile and link correctly. Injecting our code instead of using a precompiled binary enables more BELLHOP computations to be performed pushing our project towards running in real time.

The BELLHOP implementation can be compiled for CPU or GPU usage. It is structured in such a way that a simple change in the top-level CMake file allows compiling for either architecture. We liked this capability because it allows us to benchmark the algorithm's speed on the respective platforms. Being able to benchmark, however, pushed us into transferring some of the ray memory to the host computer for visualization. Transferring the data enabled a flexible code base that was easier to implement and compile for evaluation. In practice, this project will fully be realized on the GPU. This will enable us to keep the ray information on the GPU and allow OpenGL to access the GPU device the memory directly. Although there was a slight speed decrease because of this compromise, this compromise allowed us to finish the semester project in a timely fashion with decent results.

Creating the GUI in OpenGL was no small feat either. Instead of relying on frameworks that are built on top of OpenGL like SFML [5] or SDL [6], we rely on raw OpenGL calls with minor input help from GLFW [7] to handle interaction and window events. We also integrated a lightweight, widget library meant for OpenGL called ImGUI [8] which provided the necessary sliders, buttons, and input fields that allow users to enter in simulation-specific parameters. Keeping the rendering logic as close to OpenGL as possible allows us to avoid unnecessary overhead when depicting the current simulation status. It also allows us to further inject ourselves into the rendering pipeline more directly for future iterations of this work. Finally, we use an OpenGL configuration script that creates the correct set of environmental parameters that allows OpenGL to run optimally on our system called GLAD [9].

Relying on OpenGL calls directly comes with its drawbacks. Since everything is as low-level to the rendering engine as possible, we do not reference abstracted objects such as a square, circle, line, etc. For every object, we are responsible for the following:

1) Creating and compiling a shader that controls the transformation and color of the proposed drawing.
2) Creating and populating a Vertex Array Object (VAO) and Vertex Buffer Object (VBO) which keep track of an object's vertices in host and OpenGL memory.
3) Indicate how the vertices are to be connected (i.e. `GL_LINE_LOOP`, `GL_LINE_STRIP`, and a myriad of other options).
4) Delete all the previous items from memory.

While this may not seem like a lot to keep track of, this overhead in programming balloons exponentially the more that is (potentially) drawn to the screen.

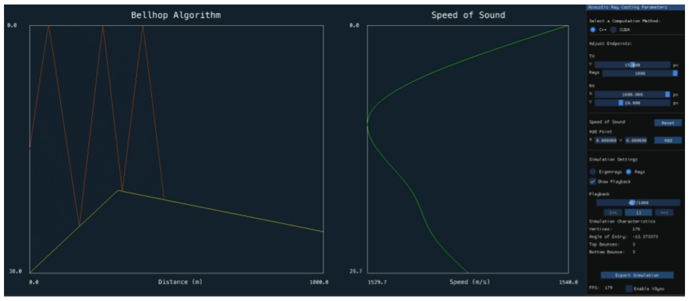
Once we could reliably draw shapes to the screen at any color and position, we were able to render the waves represented in the BELLHOP algorithm. We then created a second viewport to visualize the speed of sound spline that compliments every simulation. The algorithm to take a few points and render a spline of n points was provided by an already existing solution [10]. All of the vertices for these waves and splines are provided for by the BELLHOP algorithm which runs once every frame.

TABLE I: Ray Test

# of Rays	100	200	300	400	500	600	700	800	900	1000
Computation Speed Up	0.80	0.63	1.01	0.84	1.33	1.27	2.32	1.55	1.55	1.63
Graphics Speed Up	3.17	2.69	4.7	3.23	6.7	4.58	8.17	5.79	5.47	5.16

Clearing the old frame's previous objects and drawing the new objects once per frame provides the real-time experience of seeing the BELLHOP model react instantaneously.

The final and most annoying hurdle to cross in rendering the drawings to the screen is rendering text. There is no native OpenGL support for rendering text and creating a renderer by hand from scratch seemed pretty unreasonable for the scope of the project. So in the true spirit of all things open-source, we use a text renderer that is the product of a Learn OpenGL tutorial series [11]. Once this was all set up, we could render the labels for the axes of our simulator, leaving us with a tool as seen in Figure 3.



Select a Computation Method: C++ CUDA

Adjust Endpoints:

TX
Y 15.000 px
Rays 1000

RX
X 1000.000 px
Y 10.000 px

Speed of Sound
Add Point
X 0.000000 Y 0.000000

Simulation Settings
Eigenrays Rays
Show Playback

Playback
417/1000

Simulation Characteristics
Vertices: 176
Angle of Entry: -13.373373
Top Bounces: 3
Bottom Bounce: 3

FPS: 179 Enable VSync

Fig. 3: BELLHOP OpenGL simulator screenshot.

IV. Analysis and Results

To evaluate both portions of our project we used the normal ray functionality. This allowed us to explicitly control how many rays were being computed by the BELLHOP model and how many rays were being drawn by our OpenGL application. We compiled a binary to run both the BELLHOP model and graphics solely on the CPU and another that utilized the GPU. We performed two tests using these setups. These tests are meant to characterize the speed profiles of running on either a CPU or GPU and how much time is spent doing each task in rendering a single frame.

The first test compares the run times of the computation and graphics between CPU and GPU architectures. We selected 100-1000 rays in 100-ray increments to show this comparison. The results are found in Fig. 4 and Table I. These results show that there was a speed-up for both the BELLHOP computation and our OpenGL application when utilizing the GPU. The only scenario when the GPU did not cause a speed-up is in the first 100-400 ray tests. We attribute this to the memory overhead of transferring memory back and forth between the host machine and the GPU. The C++/CUDA project documentation claims significant speedup between the C++ and CUDA implementations does not manifest until tens of thousands of rays are computed. Running with tens of thousands of rays would most likely crash our OpenGL application and render the test useless. We instead focused on a reasonable amount of rays that can feasibly be drawn coherently.

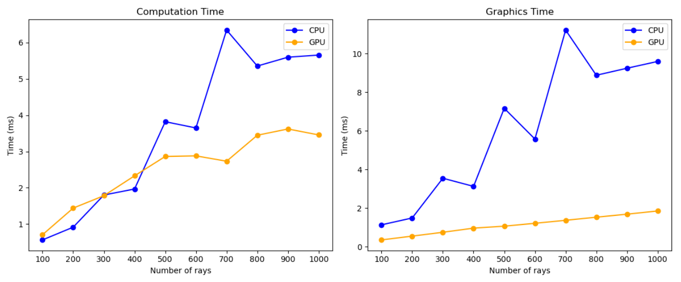


Fig. 4: Performance comparison of BELLHOP execution and OpenGL rendering between the CPU and GPU.

The second test we ran was to check what percentage of rendering a frame (including the BELLHOP calculations) is affected by running on a CPU vs. a GPU and whether or not we are using the C++ implementation or the CUDA implementation. To keep the test relatively simple, we decided to run the CPU test with the C++ BELLHOP code and the GPU test with the CUDA code. This would ensure that the CPU solution would be both calculating and rendering our code with host hardware and that the GPU would use its hardware in the CUDA version for both calculations and OpenGL rendering (as verified by checking `nvidia-smi`). We used the same initial environment for both of the runs and rendered 1000 normal waves, averaging the time it took to perform the calculations in BELLHOP and the time it took to change the vertices for drawing, drawing on the screen, and swapping the buffers.

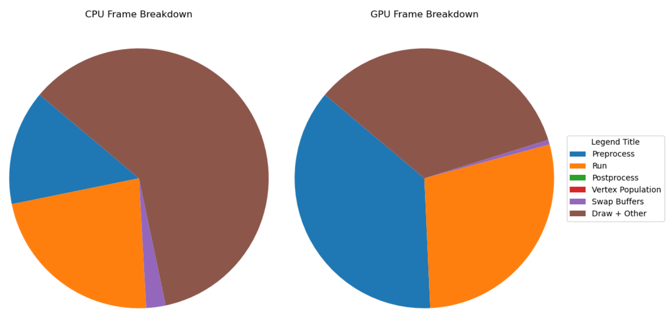


Fig. 5: Percentage of frame rendering cycle breakdown on CPU and GPU.

As apparent in Figure 5, when running the CPU-only version

TABLE II: Frame Breakdown

	Preprocessing	Run	Postprocessing	Vertex Population	Swap Buffers	Draw + Misc.
CPU time (ms/%)	2.401 **14.3%**	3.81 **22.69%**	0.0004 **0.0023%**	0.0014 **.0083%**	0.41 **2.44%**	10.17 **60.56%**
GPU time (ms/%)	1.96 **39.74%**	1.13 **22.91%**	0.00015 **.003%**	0.00052 **.01%**	0.031 **.63%**	1.81 **36.7%**

of the test, we can see that the majority of the time taken to render the frame is from the drawing (+ other since we also include the time it takes to print out values to `stdout`) with a small yet significant sliver coming from swapping the buffers. The other two categories come from BELLHOP calculations and the rest of the contributing factors are insignificant. Conversely, when looking at the GPU-aided code, we can see that there is a much more even distribution of what contributes to the render time. We see that Draw + Misc. time was reduced significantly. Upon closer analysis from Table II, we see that the top categories for rendering across both tests are Preprocessing, Run, and Draw + Misc. However, in the CPU test, over half of the frame render time was dedicated to drawing whereas the second test it is under the Preprocessing time! Furthermore (and as expected) the total render time has decreased significantly from CPU time to GPU.

V. Conclusion and Future Work

For this project, we created an OpenGL application that directly modifies and visualizes the Bellhop model parameters and outputs. We directly inject our code into the Bellhop algorithm and do not simply modify configuration files and call a precompiled binary. By doing so we can simulate and visualize hundreds to thousands of acoustic waves in real time. We measured the speed up of running both the model and GUI application on a GPU vs a CPU. We found that using a GPU is computationally advantageous in most cases, especially for the graphics. This project is the first step of many of creating a working underwater acoustic communications simulator for a research project in our lab. We plan on modifying the C++/CUDA Bellhop project further to keep the ray memory solely in the GPU and not transfer any data back to the CPU. We also plan on supporting more of the features the Bellhop model can perform. This project has pushed our understanding of both modern C++ and OpenGL which gives a great footing to continue this and future projects.

References

[1] May 2023. [Online]. Available: http://oalib.hlsresearch.com/AcousticsToolbox/
[2] A. NUS, "arlpy," 2023. [Online]. Available: https://github.com/org-arl/arlpy
[3] M. P. L. at Scripps Oceanography, "bellhopcxx / bellhopcuda," https://github.com/A-New-Bellhope/bellhopcuda, 2013.
[4] [Online]. Available: https://developer.nvidia.com/rtx/ray-tracing/optix
[5] L. Gomila. [Online]. Available: https://www.sfml-dev.org/
[6] [Online]. Available: https://www.libsdl.org/
[7] [Online]. Available: https://www.glfw.org/
[8] O. Cornut. [Online]. Available: https://github.com/ocornut/imgui
[9] D. Herberth. [Online]. Available: https://glad.dav1d.de/
[10] T. Kluge. [Online]. Available: https://github.com/ttk592/spline/tree/master
[11] J. de Vries. [Online]. Available: https://learnopengl.com/code_viewer_gh.php?code=includes/learnopengl/shader.h