FLUCOME 2007
Tallahassee, Florida

# WEB-BASED TSUNAMI VISUALIZATION

**Chris Janik[1] Mike Bailey[2] Dylan Keon[3] Cherri Pancake[4] Harry Yeh[5]**

## ABSTRACT

The Indonesian tsunami in December 2004 awakened the world to the need to better understand and predict tsunami wave behavior using simulations. In order to understand such an immense amount of data, a visual approach was desired. This paper describes a visualizer for tsunami wave simulations. The *Tsunami Visualization and Animation Tool* (TVAT) works with multiresolution wave height and bathymetry depth files. It animates the wave simulation, and allows the user to probe the waves to see a plot of wave heights in time or along a transect in space. TVAT is written in Java using Java OpenGL (JOGL) so that it can be run standalone or as a web applet. It also uses GPU programming to speed the display. The methods developed and lessons learned from this project can be applied to many other large-dataset visualization projects.

**Keywords: visualization, fluid dynamics, tsunami**

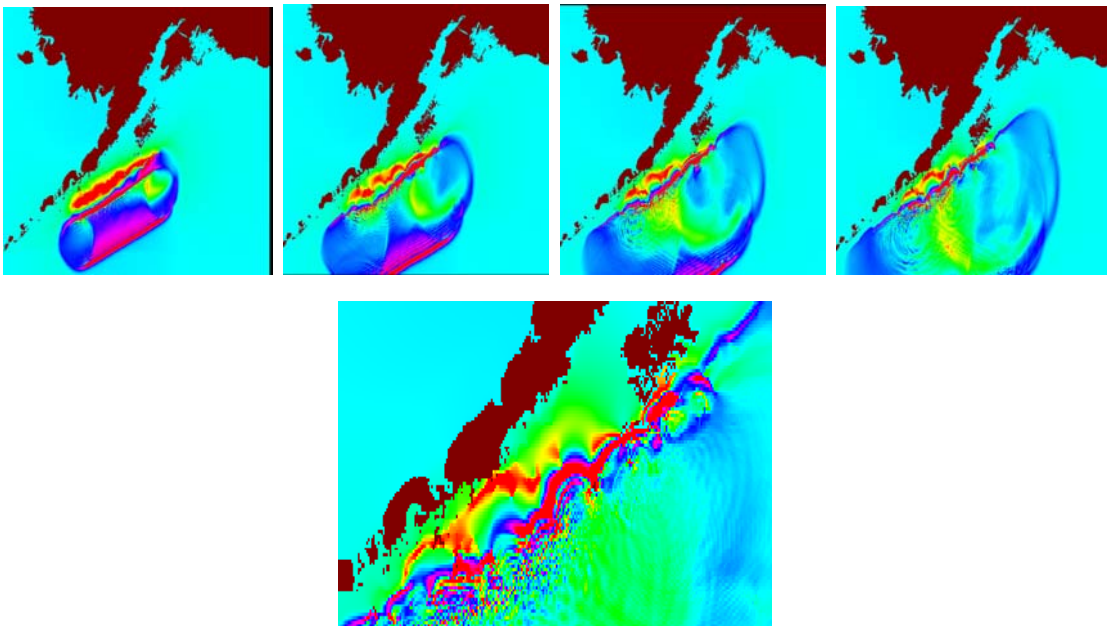

**Figure 1: Sample Animation Frames from the Tsunami Viewer**

[1] Computer Science, Oregon State University
[2] Corresponding author: Computer Science, Oregon State University, email: mjb@cs.oregonstate.edu, fax: 541-737-1300
[3] Computer Science, Oregon State University
[4] Computer Science, Oregon State University
[5] Computer Science, Oregon State University

## INTRODUCTION

Following the 2004 Indian Ocean Tsunami, simulating tsunamis to better understand them has become a pressing issue. In order to do this, larger initial data sets are required, which produce subsequently larger output. The Northwest Alliance for Computational Science and Engineering (NACSE) at Oregon State University, in cooperation with the Artic Region Supercomputer Center (ARSC), has created one such simulation package. In order to understand these immense data sets a visual approach was desired. This project had very ambitious plans, such as requiring cross-platform compatibility, web-based distribution, and most importantly, real time visualization and interaction with the data set.

## PRIOR WORK

Computer graphics displays of ocean waves have appeared a number of times over the past few years. Procedural models were made popular by Max (1981), Fournier and Reeves (1986), and Mastin et al (1987) in the 1980s. More recently, Takahashi et al (2003) used particle systems, Enright et al (2002) used velocity extrapolation, Hinsinger et al (2002) used adaptive methods, and Thon et al (2000) used turbulence functions. The goal of these efforts has been to create plausible-looking ocean waves for animation systems. As the graphics-oriented publication titles show, these were largely undertaken for computer graphics applications in entertainment. Instead, this project has focused on visualizing and querying voluminous quantitative wave data from supercomputer simulations.

There has been a small, yet growing, body of visualization work using Java-OpenGL. Kenneth Russell (2007) from Sun Microsystems developed a terrain visualization application using Java, JOGL, and Memory Mapped I/O. The purpose of their application was to visualize large terrain data sets in real time. The *JCanyon* program that they created showed that this combination of technologies can indeed perform well. Since our application had similar requirements, these technologies appeared to be useful in our case as well.

## PROGRAM DESIGN

The design requirements caused us to select a unique tool set for implementing the project. We chose Java as the implementation language due to its cross-platform nature, and, when combined with Java Web Start Technology [JavaWebStart (2007)], offered a web-based means of distribution. This allowed the application to be distributed in a web-based manner, where researchers could access it anywhere across the Internet. In addition, this web based distribution allows us to update the application without end users ever needing to take part in, or even be aware of, the process.

The real-time requirements led us to select the Java OpenGL (JOGL) API [JSR231 (2007)] as a means of accessing hardware accelerated OpenGL rendering. Within the OpenGL context we chose to use hardware-accelerated texturing to store individual time slices, and use fragment shaders to apply different color representations in real time. With the use of fragment shaders, multiple color schemes were available, allowing the user to switch between them as their requirements change, or if a different color scheme is more informative.

We also wanted to be able to capture a video of the animated tsunami. For this task, the Java Media Framework (JMF) supplied methods that enabled us to capture an AVI file of the animation in a background thread while the animation proceeds at a real time rate.

When visualizing data sets that are many gigabytes in size, it is usually impractical to store the data in physical memory. To solve this problem, we used the java.nio.MappedByteBuffer [MmapA (2007), MmapB (2007)] class to perform memory mapping. Using this class we loaded each time slice into memory just prior to its use. This allowed us to keep memory consumption to a minimum while still maintaining performance. Additionally, the memory mapped I/O offered another benefit. When a time slice is used again shortly after being loaded, the disk I/O operations would not need to be performed the

second time due to the data being cached in memory. This yields a large gain in performance when viewing a subset of a much larger data set.

## USER INTERFACE

Cross platform compatibility was one of our primary requirements, and as a result the user interface needed to be familiar to users on all systems. This can be seen in figures 2 and 3, where Java creates a familiar interface for two of the most popular operating systems.

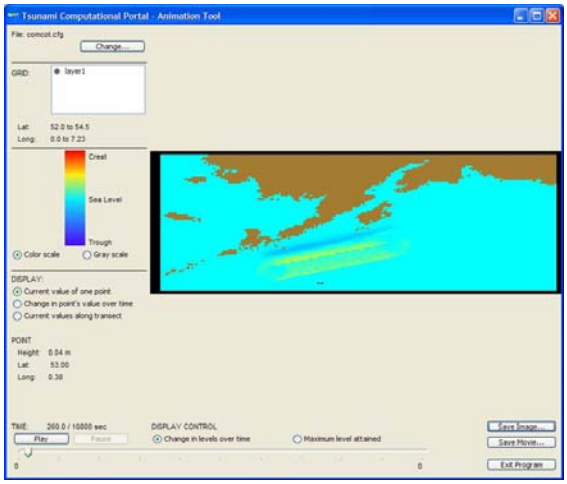


**Figure 2: Windows OS look and feel**

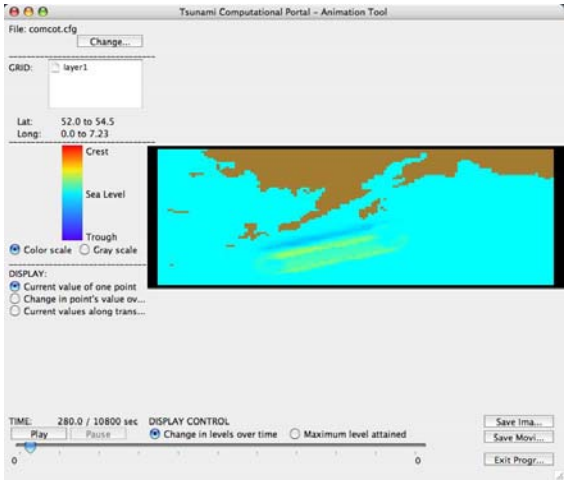


**Figure 3: Mac OS look and feel**

The program layout was broken into areas of common usage, as shown in figure 4.

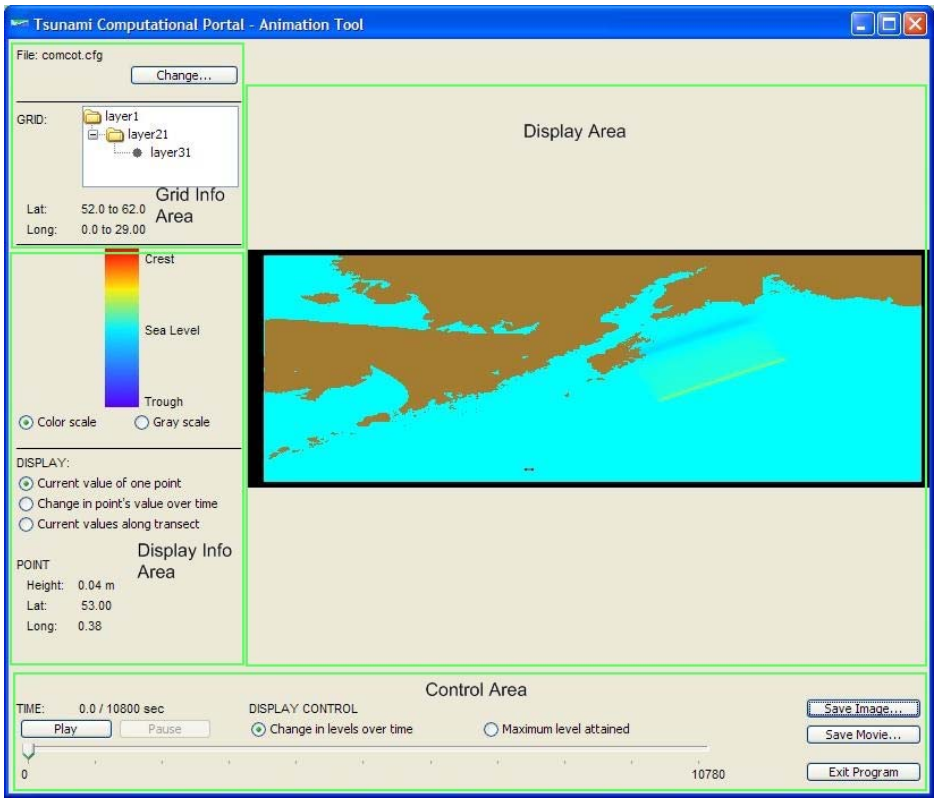


**Figure 4: Application layout**

The first area is the grid information area, shown in figure 5, where information regarding the open data set, the hierarchy of nested grids, and the latitude and longitude bounds of the currently active grid. By clicking with the mouse in the hierarchical view the user is able to select which grid is visible in the display area.

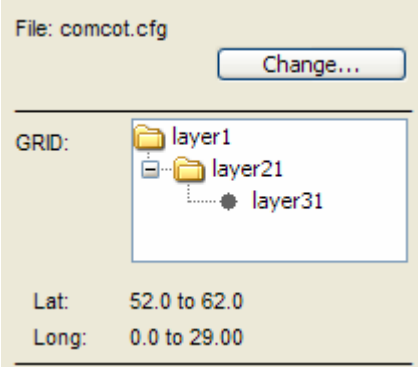


**Figure 5: Grid info area**

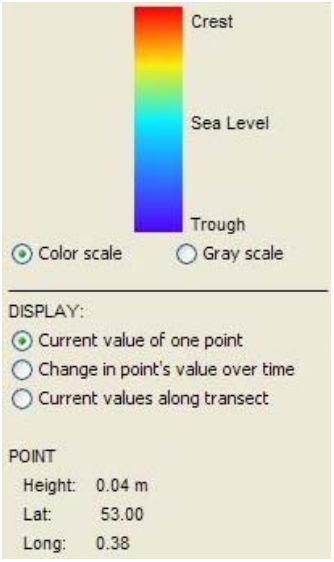


**Figure 6: Display info area**

Next is the display information area, shown above in figure 6. In this area, the user is first presented with a legend illustrating how the visualization's color mapping corresponds to the sea level. Below the legend and visualization portion is a radio button to select different methods to probe the data.

The first two probe methods are implemented with a simple point and click interface accessed through clicking in the display area to directly query the sea level at that point. The last probe method is the wave height along a transect. This method also relies on navigation in the display area but instead of a simple point and click, the user instead clicks and drags to set the beginning and end points of the transect. While the user is dragging the path of the transect is displayed in real time to better illustrate to the user what their end result will be. In addition to the display info area, information from two of those probe modes will be graphed for ease of understanding. The first of which is "Change in point's value over time." option. This probe mode displays the sea level height over the time of the simulation run, as shown in figure 7.

The second probe mode that generates a graph for visual inspection is "Current values along transect", as shown in figure 8. This mode is the one in which the user clicks and drags to set the beginning and end points of the transect, to further aid the user in making the connection between the drawn transect and the graph, the same green and red markers are used to indicate the start and end points both on the graph but also on the transect the user selected in the display area.

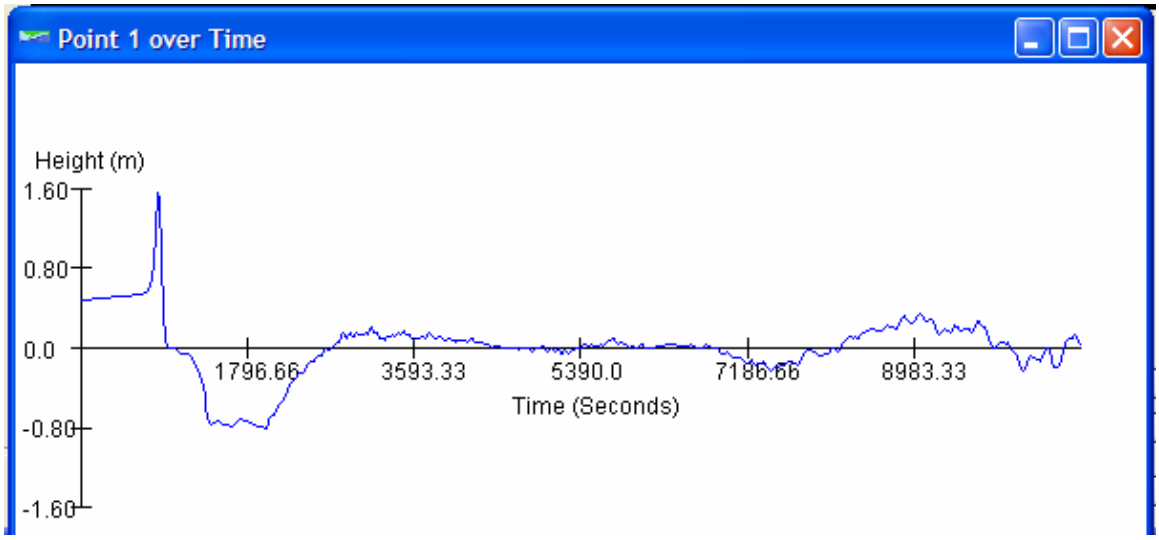


**Figure 7: Change in point's value over time**

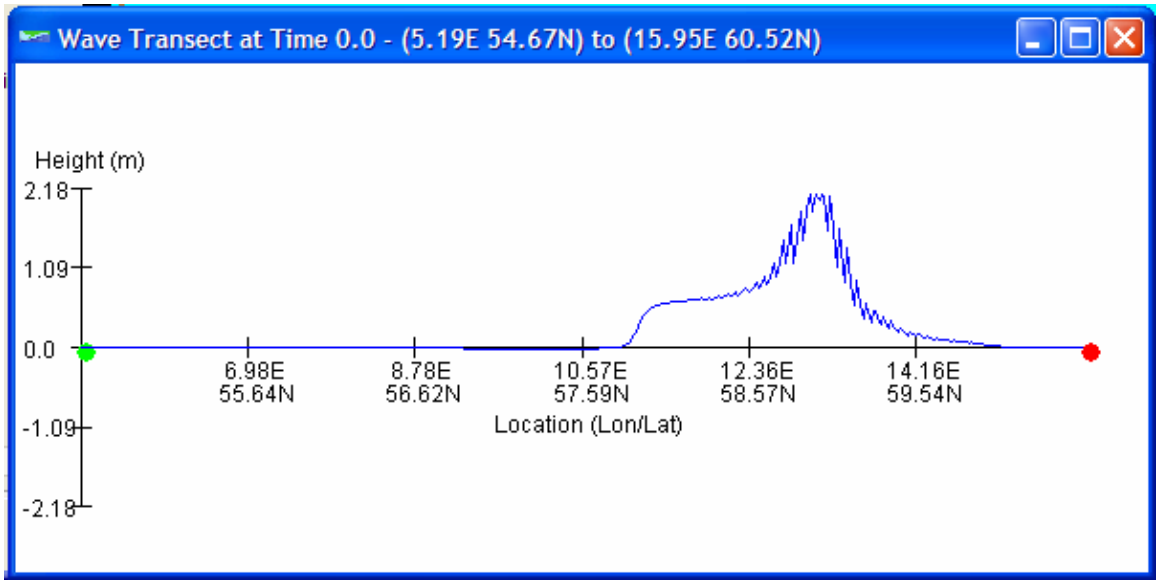


**Figure 8: Current values along transect**

The control area, shown in figure 9, allows the user to play back the visualization animation, as well as capture stills and animations for further distribution. Play back of and pausing the animation is controlled through a pair of buttons, while seek forward and reverse is handled by dragging the slider in the desired direction. There is also functionality to make a screen capture of the application with the click of a button. There is similar functionality for capturing the animation to an AVI file, where there is a one-click capture.


**Figure 9: Control area**

In addition to controlling the animation and capturing the data to alternative formats, the control area also allows the user to switch to viewing an additional data set created through the computation portal, the maximum levels attainted over the course of the simulation. Since this presents a static image that cannot be animated, the controls involving animation are grayed out and disabled as illustrated in figure 10. This is done to avoid confusion about when the controls will work.

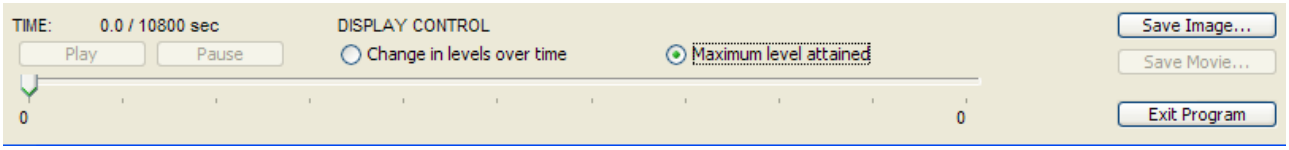


**Figure 10: Control area, animation tools disabled**

## USING OPENGL TO ACCELERATE RENDERING

OpenGL [Shreiner et al (2007)] is a cross-platform API for producing 3D graphics. Special graphics processing unit (GPU) hardware has been developed to accelerate OpenGL and has become widely available in consumer-level desktop and laptop computers.

Simply accelerating OpenGL by itself was not enough to be of use in this application. Additional features were necessary to make a GPU truly useful in a more general scenario. The first of these was the ability to store floating-point values in textures [OpenGLARBb (2007), OpenGLARBc (2007)]. The second was support for non-power of two textures [OpenGLARBa (2007)]. The third was the ability to write programs for GPUs in a high level language such as OpenGL's GLSL [Rost (2006)].

### Java OpenGL Binding (JSR-231)

Despite OpenGL being a cross-platform API, it requires system level access, which poses a problem for using it in Java. To solve this problem several libraries and bindings have been created by various groups. The most important were the Lightweight Java Game Library, LWJGL (2007) and the Java OpenGL Binding (JOGL) [JOGL (2007)]. Eventually JOGL was selected by Sun to provide the basis for an official Java binding to OpenGL, which is now known as JSR-231 [JSR231 (2007)]. Due to this official status, we chose to use the JSR-231 binding.

### Floating-Point Textures

Textures in 3D APIs, such as OpenGL, are effectively 2D arrays that the GPU hardware is able to manipulate quickly and efficiently. In most applications using OpenGL a texture consists of a width by height number of 8-bit tuples, which usually represent the red, green, and blue color components. This color mapping is then applied to a geometric surface. Many other formats are available, depending on application requirements, and this can include single values, two values, or even up to four values. Additionally higher precisions than 8-bits per component can be used as well.

For our application we required floating-point values to be stored in a texture. This ability is offered by the ATI_texture_float [OpenGLARBb (2007)] and ARB_texture_float extensions [OpenGLARBc (2007)]. These extensions expose the ability to store data in standard single precision IEEE format, or if memory space is a concern a half precision format is available as well. For visualization purposes in our application, the half precision format provided sufficient precision and allowed bandwidth and memory usage to be cut in half.

Another by-product of using floating-point textures is that the actual data can be passed to the visualization with minimal interaction from the CPU. Previously floating-point data needed to be quantized before being placed into one of OpenGL's integer based formats. This not only leads to precision loss, but also takes up precious CPU cycles when done with large data sets.

**Non-Power of Two Textures**

Previously textures in most 3D APIs, OpenGL included, required that a texture's dimensions both be powers of two. This behavior is not desirable because it forces the application to either resample the data to be a power of two, or make the texture the next largest power of two, place the data in it as usual, then fill the excess space with some sort of null data to be ignored. Both of these approaches use substantial CPU time and the latter requires a larger texture be transferred to and stored by the GPU. The worst case scenarios is if the data is only marginally larger than the next smallest power of two in both dimensions, as this larger textures can be up to four times larger than the original data set.

A better approach is to just load the original data, and this is facilitated through the ARB_texture_non_power_of_two [OpenGLARBc (2007)] extension as well as the promotion of this extension into OpenGL 2.0. This extension relaxes the power of two requirements for texture dimensions, which means that the two previously discussed work-arounds are no longer necessary.

When non-power of two texture support is combined with floating-point texture support, the data to be visualized can be uploaded directly to the GPU as soon as it is available, whether it is being loaded from disk, memory, or as the result of a computation.

**GLSL Shaders**

So far this paper has discussed new features such as floating-point textures and non-power of two textures in the context of getting data to the GPU with minimal CPU involvement. The reason that we want minimal CPU usage is because GPUs have gained programmable processors of their own, called shaders [Rost (2006)]. Without going into great detail of the architecture of a GPU, it can now be thought of as a highly parallel CPU with tens of cores and hundreds of threads, all of which culminates in very high performance for a data parallel work [NVIDIA (2007)].

In this application we used what is known as *fragment shaders* on the GPU to perform the color mapping for the visualization, instead of on the CPU. Because the color of each pixel is independent of its neighbor, this is inherently a data parallel problem, one in which using the GPU offers a substantial speedup.

Additionally, because the fragment shaders work on the data in its original format and we avoided having to pre-compute color values, the color mapping of the visualization can be changed in real time simply by switching which fragment shader program is active. At one point we had implemented upwards of a dozen different color mappings, but for the production version we settled on two mappings, rainbow and grayscale, shown below in figure 11.

There are other benefits to keeping the data in its original format and operating on it in shader programs. Computations that are more advanced than the color mapping can also be performed at the shader level, whether it is further analysis of the data or part of the simulation that generates the data.

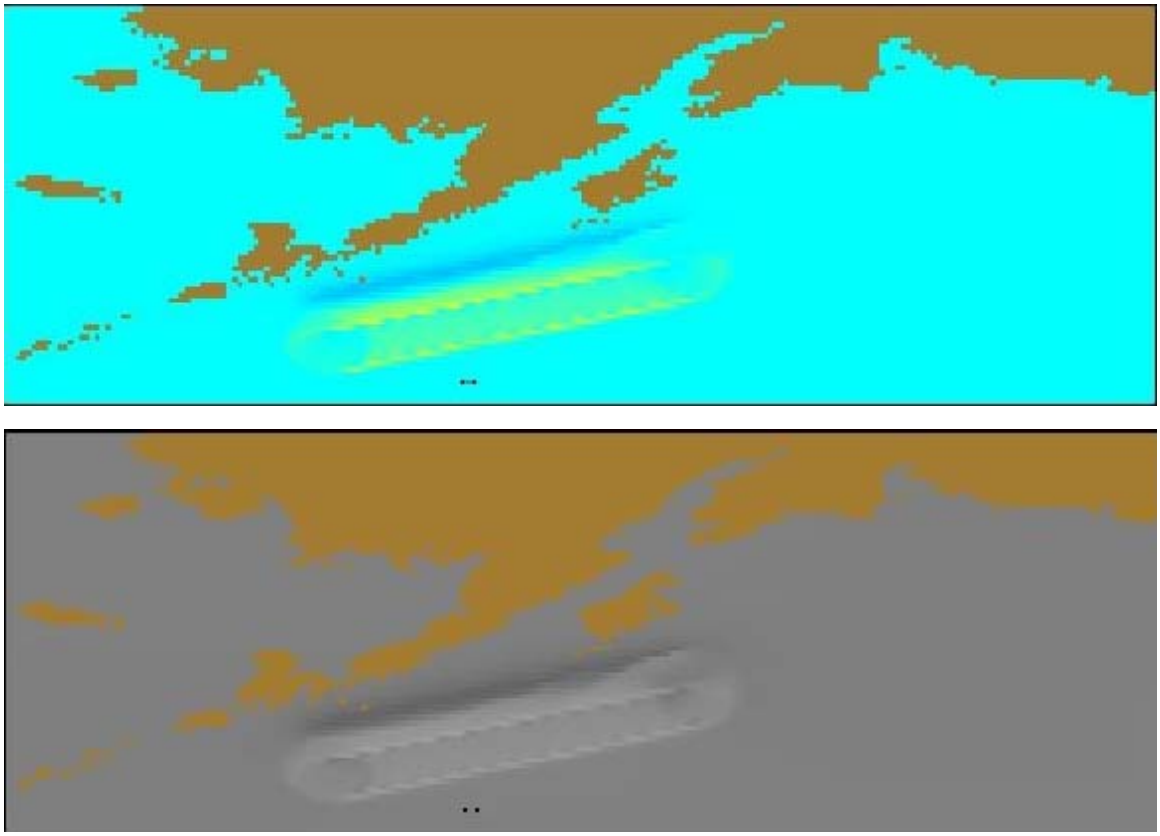

**Figure 11: Rainbow and Grayscale Color Mappings**

## MEMORY MAPPING

One of the most difficult challenges we faced was dealing with multi-gigabyte data sets, because it is impractical to store all the data in physical memory. To solve this problem we used the java.nio.MappedByteBuffer [JAVA (2007)] class to create a memory-mapped file. Using this class, a whole file or sections of a file could be mapped to virtual memory where it was then treated by the application as if it were in physical memory. With this technique, the data needed for the current frame was only loaded into physical memory just before it was used.

Because the data file was mapped to a section of virtual memory, the operating system became responsible for managing how long it remained in physical memory [Java (2007), MmapA (2007), MmapB (2007)]. In general, it remained in physical memory until that physical memory was needed by something else. This means that if a user was viewing a section of data that was recently viewed, performance would be substantially better because no disk IO operations would occur.

This technique did introduce an interesting problem in that, in the right situations, the disk drive became the performance bottleneck. There was no easy solution to this, as CPUs' and GPUs' processing capacities have continued to grow in accordance with Moore's law, but memory capacity and disk drive speed have not. This means that datasets will likely increase to a size where it is neither practical to store the data solely in main memory nor stream it from the disk drive. For our application, the disk drive became the bottleneck when each individual time slice was too large, while the number of time slices did not have an immediate affect on performance. If there are too many time slices, the entire file might not be able to be memory-mapped at once, and remapping or falling back to other IO methods would likely result in a performance loss.

## JAVA MEDIA FRAMEWORK

While these tools allow a researcher to gain a better understanding of tsunamis, they are still limited in their ability to make this information available to others. Others may not have adequate hardware to run the application, and, probably more frequently, the datasets may simply be too large to easily transfer to someone who does not need that level of detail. This is why the application has built-in support for capturing the animation to an AVI animation file.

To accomplish this, we used the Java Media Framework (JMF), and based our solution on the JpegImagesToMovie.java example [Sun Developer network (2007)]. We then made our animation encoding object run in its own thread; separate from that of the visualization thread that generated the images. The goal behind this was to allow the visualization to continue to run while the capture and encoding took place. While there was some performance loss on a single-core machine, dual-core is becoming commonplace and allows for each thread to occupy its own CPU, resulting in improved performance. Because this portion of the application is multi-threaded, special care was taken to ensure that the encoding thread accepts the images in the correct order and also waits for the animation thread to provide it with an image. To relieve us from having to implement a synchronizing mechanism, we chose to use the LinkedBlockingQueue data structure from the java.util.concurrent package [Java (2007)] to implement our producer-consumer model. While any data structure that implemented the BlockingQueue interface would have sufficed, the linked list structure seemed the most logical due to our lack of removing from the middle of the list but also that it offers a higher throughput than array-based queues [Java (2007)].

## JAVA WEB START

After deciding to use Java, it made sense to use Java Web Start to fulfill the web-based distribution requirement. Java Web Start allows for any Java application to be distributed to any user who has a network connection and Java 1.4 or greater installed [Sun Microsystems (2007), JavaWebStart (2007)]. While the application is stored locally after having been run the first time, when it is launched again, Java Web Start checks the server for an updated version before executing. If the application has been updated, Web Start then downloads and executes that version. Otherwise, the locally-cached copy is executed. This happens transparently to the user and ensures that they always have the most up-to-date version of the application. Java Web Start also allows the developer to specify the version of Java required, and if the user's machine does not have it, Web Start will download and install it for them. This allows multiple applications with different requirements to all run seamlessly with each other.

Java Web Start, by default, offers a similar level of security that a Java Applet has because it runs in a restricted environment. However, one characteristic that differentiates a Web Start application from an Applet is that there are a couple of techniques that allows the application to be granted greater access to the system. One such technique is the use of the JNLP API [JNLP (2007)], which allows basic system I/O operations, such as file, printer, or network access, to be performed in a secure fashion. An alternative for when the JNLP API is inadequate is to digitally sign the application and grant it full permission through Web Start. In keeping with requiring security the user is now prompted to give permission to this application, and if they deny permission the application is not executed. In our application there did not exist a method in the JNLP API that was equivalent to the memory mapping offered through the java.nio classes. As a result our application required the digitally signed approach.

## CONCLUSIONS AND FUTURE WORK

While this application was designed specifically for simulated tsunami visualization, the methods we described are applicable to most any visualization application that requires real time interaction and large data sets.

Using advanced 3D graphics hardware with support for non-power-of-two textures, floating point textures, and programmable shaders, has allowed us to avoid preprocessing, and as a result keep performance high. When these techniques were combined with memory-mapped file I/O the working data sets were kept small and memory usage kept to a minimum. Furthermore, memory mapped file I/O

allowed us to work with very large datasets without being affected by the size of main memory. It made it so that there was virtually no limit to the number and complexity of time slices in a data set.

Finally, the selection of the Java programming language allowed the application to be run on all the major desktop platforms available. This also opened up the ability to make use of Java Web Start to distribute the application over the Internet. This allows researchers access to the up-to-date application anywhere in the world on nearly any machine, so long as it has access to the Internet.

In the future, problems involving the disk performance bottleneck might require the simulation to be adapted to run on a GPU. The reasoning behind this is that it takes the disk drive, CPU, and memory bus out of the picture altogether. The data and the visualization would be created at the same time. Similar projects have already begun utilizing the GPU in this fashion, although for different reasons [Folding (2007), GPGPU (2007)].

## REFERENCES


Douglas Enright, Stephen Marschner, and Ronald Fedkiw (2002), "Animation and Rendering of Complex Water Surfaces" *ACM Transactions on Graphics*, Volume 21, Number 3 (2002), pp. 736-744.

Folding (2007), <http://folding.stanford.edu/>

Alain Fournier and William Reeves (1986), "A Simple Model of Ocean Waves", *Computer Graphics* (Proceedings of SIGGRAPH 86), Volume 20, Number 4, pp. 75-84.

GPGPU (2007), <http://www.gpgpu.org/>

Damien Hinsinger, Fabrice Neyret, and Marie-Paule (2002), "Interactive Animation of Ocean Waves", *ACM SIGGRAPH Symposium on Computer Animation*, 2002, pp. 161-166.

Java (2007), "Overview (Java 2 Platform SE 5.0)", JavaTM 2 Platform Standard Edition 5.0 API Specification. 2004. Sun Microsystems, Inc., <http://java.sun.com/j2se/1.5.0/docs/api/index.html>.

JavaWebStart (2007), "Java Web Start Overview", Sun Microsystems, Inc. < http://java.sun.com/developer/technicalArticles/WebServices/JWS_2/JWS_White_Paper.pdf >.

JNLP (2007), <http://java.sun.com/products/javawebstart/docs/javadoc/index.html>

JOGL (2007), "java.net The Source for Java Technology Collaboration", *Java bindings for OpenGL*. <https://jogl.dev.java.net/>.

JSR231 (2007), "The Java Community Process Program", *JSR 231: Java Binding for the OpenGL API.* <http://jcp.org/en/jsr/detail?id=231>.

LWJGL (2007), "lwjgl.org – Home of the Lightweight Java Game Library", *LWJGL Lightweight Java Game Library*. < http://www.lwjgl.org/ >.

G.A. Mastin, P. A. Watterberg, J. F. Mareda (1987), "Fourier Synthesis of Ocean Scenes", *IEEE Computer Graphics and Applications*, Volume 7, Number 3 (1987), pp. 16-23.

Nelson. Max (1981), "Vectorized Procedural Models for Natural Terrain: Waves and Islands in the Sunset", *Computer Graphics* (Proceedings of SIGGRAPH 81), Volume 15, Number 3, pp. 317-324.

MmapA (2007), “mmap”, The Open Group Base Specifications Issue 6. <http://www.opengroup.org/onlinepubs/000095399/functions/mmap.html>.

MmapB (2007), “Managing Memory-Mapped Files in Win32”, Microsoft Developer Network Technology Group. <http://msdn2.microsoft.com/en-us/library/ms810613.aspx>.

NVIDIA (2007), “Beyond3D”, *NVIDIA G80: Architecture and GPU Analysis*. <http://www.beyond3d.com/content/reviews/1/>.

OpenGLARBa (2007), “ARB_texture_non_power_of_two”, 14 May. 2004. OpenGL ARB. 5 May 2007 <http://oss.sgi.com/projects/ogl-sample/registry/ARB/texture_non_power_of_two.txt>.

OpenGLARBb (2007), “ATI_texture_float”, 04 Dec. 2002. OpenGL ARB. 5 May 2007 <http://oss.sgi.com/projects/ogl-sample/registry/ATI/texture_float.txt>.

OpenGLARBc (2007), “ARB_texture_float”, 22 Oct. 2004. OpenGL ARB. 5 May 2007 <http://oss.sgi.com/projects/ogl-sample/registry/ARB/texture_float.txt>.

Randi Rost (2006), *OpenGL Shading Language*, Second Edition, Addison-Wesley.

Kenneth Russell (2007), “JCanyon: Grand Canyon for Java”, 5 Mar 2007 <http://java.sun.com/products/jfc/tsc/articles/jcanyon/>.

Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis (2006), *The OpenGL Programming Guide*, Fifth Edition, Addison-Wesley.

Sun Developer Network (2007), “Generating a Movie File from a List of (JPEG) Images”, <http://java.sun.com/products/java-media/jmf/2.1.1/solutions/JpegImagesToMovie.html>.

Sun Microsystems (2007), “Java Web Start Technology”, <http://java.sun.com/products/javawebstart/>.

Tsunemi Takahashi, Hiroko Fujii, Atsushi Kunimatsu, Kazuhiro Hiwada, Takahiro Saito, Ken Tanaka, and Heihachi Ueki (2003), “Realistic Animation of Fluid with Splash and Foam”, *Computer Graphics Forum*, Volume 22, Number 3, (2003), pp. 391-400.

Sebastien Thon, Jean-Michel Dischler, and Djamchid Ghazanfarpour (2000), “Ocean Waves Synthesis using a Spectrum-Based Turbulence Function”, *Computer Graphics International 2000*, pp. 65-74.