

Real-Time Deep Ocean Simulation on Multi-Threaded Architectures

This white paper investigates a technique for real-time simulation of deep ocean waves on multi-processor machines under simulated work loads using threading.

White Paper
David Reagin, Adam Lake

Introduction

Computer graphicists have a long history of attempting to model the real world. When designing immersive experiences, our goal is to design environments that look and feel as compelling as those in the real world. We can trace the origins of these simulations to previous deep thinking by physicists and computational scientists working in the applied sciences. In this article we investigate the simulation of compelling deep ocean waves. To improve the performance of our solution we have a multi-threaded workload to take advantage of dual processor machines. We demonstrate our technique with a real-time demonstration running on a two processor machine and provide an implementation able to run in real-time with integrated graphics solutions such as the Intel® 965 Express Chipset and Mobile Intel® 965 Express Chipset family.

First, we describe a list of previous work. Next, we lay out the mathematics of a summation of sine waves approach used for our implementation. We then give details of our implementation including the mechanisms we used for threading. Source code is provided with the demonstration to be used in your own multi-threaded ocean rendering extensions and implementation.

Previous Work

A number of researchers have investigated water simulation. One of the most successful has been Tessendorf, whose deep ocean water simulations have been used in movies such as Titanic* and Waterworld* [5], [6]. Since then there have been a number of other researchers and developers that have approached the problem with an eye towards real-time simulation. In his book, *Interactive Simulation of Water Surfaces*, Miguel Gomez [2] describes an implicit solution for height fields. In some cases this solution may be preferred, but a major disadvantage is the need to maintain at least two meshes--the previous mesh and the current mesh, in order to calculate the next mesh to be rendered. Another downside is the need to obtain neighbor information to calculate the next position for each vertex. Mark Finch provides an explicit solution that does not require this information in his book, *Effective Water Simulation from Physical Models* [3], and provides a number of other advantages as listed below:

- No neighbor information needed for position updates, making it easy to parallelize.
- Since no neighbor information is needed it is also easy to implement in a vertex shader in situations where a developer is better off doing the water simulation on the graphics subsystem.
- A fully parameterized simulation to give us precise control over our geometry.
- If desirable, normal can be updated based only on local vertex data, again simplifying parallelization in a vertex shader implementation. Alternatively, normal updates can use neighbor information. We compare the two approaches on the CPU in this article.
- Easy to scale and extend: We plan to add features to our water simulation. A parameterized solution makes this easy.
- Algorithm can be multi-purpose: We can use the same approach for the larger, low frequency waves of the surface and normal maps that simulate higher frequency surface waves created by wind.

In his book, *Rendering Ocean Water*, John Isidoro [4] uses a Sum of Sines approach similar to that described in [3]. They present the associated assembly level vertex and pixel shader code for implementation and a walkthrough of the technique in low level DX8 vertex and pixel shader assembly. We present a CPU based algorithm inspired by [3] that can be mapped to HLSL or left on the CPU.

Theory

First, we review a few basic definitions for waves from physics [Giancoli85], [3].

Amplitude: The maximum height of a crest or trough relative to the normal level. The total swing from a crest to a trough is twice the amplitude.

Wavelength: The distance between two successive crests.

Velocity: How fast the wave moves per unit time. ϕ , a phase constant, is used to represent speed where $\phi = \text{velocity} * 2 \pi / \text{wavelength}$.

3.1 Sum of Sines Approach to Wave Generation

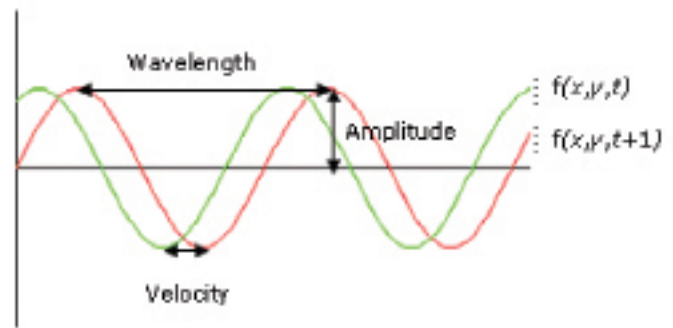


Figure 3-1. Wave Physics

In Figure 3-1, the *wavelength* is the distance between two crests, the *velocity* is the distance a wave moves in one unit time, and the *amplitude* is the distance from the origin to the top of a crest.

3.2 Static Wave Modeling

Let's start from the basics. This way, if your application does not need some of these parameters you can eliminate them and follow the steps below to derive your procedural wave geometry. First, since we want our waves to have a periodic, controllable parameterization we chose a sine wave:

$$f(x) = \sin(x)$$

For our simulations we find it desirable to stay in a normalized space where all values produced are between 0 and 1 for the height of each sine wave. This way, we find it easier to think about what we need to do to position the water simulation in the world. Therefore, we need to shift our sine wave such that it produces no negative values:

$$f(x) = \sin(x) + 1$$

While this has shifted us up so that our values are positive, we can see that we will be producing values larger than 1.0. Therefore, we will scale the results so that it fits into our 0..1 height domain:

$$f(x) = \frac{\sin(x) + 1}{2}$$

Many times, particularly in deep ocean simulation, these large scale sinusoidal movements are desirable. However, it is often the case that we want to have steeper swells in our ocean, for example to signify an approaching storm. To simulate this effect, we add an exponent, steepness, to our simulation framework:

$$f(x) = \left(\frac{\sin(x) + 1}{2} \right)^{\text{steepness}}$$

Next, we want to be able to adjust the height of our wave, also known as the amplitude. To accomplish this we will introduce a scale factor into our simulation:

$$f(x) = \text{amplitude} \times \left(\frac{\sin(x) + 1}{2} \right)^{\text{steepness}}$$

3.3 Dynamic Wave Modeling

We now have a wave with a sinusoidal pattern that we can control steepness and amplitude. However, we would like to have greater control over the surface. For example, we would like to take into account the speed and direction of the wave as well as the wavelength.

Since we are simulating a 2 dimensional height field we need to consider the movement in each direction. To accomplish this we project the x, y position onto a wave direction vector using a dot product. For simplicity we assume the direction vector is parallel to the flat surface and therefore has no z component. Recall the result of a dot product between two vectors is a scalar value we denote as S:

$$S = \text{Dir}(x, y) \bullet \text{Pos}(X, Y)$$

Next, we want to take into account the frequency of the wave. We know from physics that the wavelength relates to frequency as frequency = 2*π/wavelength. Therefore we can use the wavelength as input and generate the frequency by this function. Since we want this to influence the periodicity of our values delivered to our sin function, we incorporate this into our function:

$$S = \text{Dir}(x, y) \bullet \text{Pos}(X, Y) \times \text{frequency}$$

We have a wave that takes into account direction and wavelength to determine position, but does not actually move across the surface.

The final variability we introduce is to vary the velocity of the wave. Again, we know from physics that the phase constant φ is related to velocity by the equation:

We enhance our equation as follows, where t is time:

$$\phi = \text{velocity} \times \text{frequency} = \text{velocity} \times (2\pi / \text{wavelength})$$

In summary, we now have a function that takes into account wave-

length, amplitude, velocity, direction, position, steepness, and amplitude:

$$f(x, y, t) = \text{amplitude} \times \left(\frac{\sin(S) + 1}{2} \right)^{\text{steepness}}$$

Where

$$S = \text{Dir}(x, y) \bullet \text{Pos}(X, Y) \times \text{frequency} + t \times \phi$$

$$\phi = \text{velocity} \times \text{frequency} = \text{velocity} \times (2\pi / \text{wavelength})$$

3.4 Wave Composition

A single wave function would be adequate if our simulation is a simple case. However, we are going to want to have a greater degree of variability to simulate a true deep ocean surface. Observing an ocean surface you notice that there are multiple waves coming from multiple directions that interfere with one another at any given point to create peaks, troughs, etc. that vibrate vibrantly. To simulate this we will take into account several waves by summing their positions at any point in our simulation. We have chosen to limit the number of sine waves to 4, and found that this provides an adequate amount of variability. To simulate the height of a position (x,y) we have the equation:

$$\text{height}(x, y, t) = \sum_{i=1}^4 f_i(x, y, t)$$

3.5 Surface Normals

For shading of the surface it is important to know the surface normal. Assuming we were using a tessellated surface we could update the position of each vertex using Equation 3 above then recalculate normals for the surface by re-computing face normals then averaging these together for each vertex. However, there is an alternative that needs consideration, an explicit solution presented in [3]. We can take the derivative in the x and y direction to determine the rate of change of the surface normal. We refer to these as the binormal and tangent vectors respectively. Previously we showed that a given position (x,y) has a surface height based on the function Position(x,y,t) = (x,y,f(x,y,t)). The binormal and tangent for a height field (assuming for simplicity that the height field is oriented along an x-y grid) are:

$$Binormal(x, y) = \left(\frac{\partial x}{\partial x}, \frac{\partial y}{\partial x}, \frac{\partial}{\partial x} (f(x, y, t)) \right)$$

simplifies to

$$Binormal(x, y) = \left(1, 0, \frac{\partial}{\partial x} (f(x, y, t)) \right) \text{ and}$$

$$Tangent(x, y) = \left(\frac{\partial x}{\partial y}, \frac{\partial y}{\partial y}, \frac{\partial}{\partial y} (f(x, y, t)) \right)$$

simplifies to

$$Tangent(x, y) = \left(0, 1, \frac{\partial}{\partial y} (f(x, y, t)) \right)$$

The cross product is therefore:

$$Normal(x, y) = Binormal(x, y) \times Tangent(x, y)$$

$$Normal(x, y) = \left(-\frac{\partial}{\partial x} f(x, y, t), -\frac{\partial}{\partial y} f(x, y, t), 1 \right)$$

Now, we need to compute the derivative of f(x,y,t) and sum them together for each sine wave we are composing for the final position. To accomplish this we will differentiate f(x,y,t) with respect to x and y for each wave composing our geometry simulation:

$$\frac{\partial}{\partial x} f(x, y, t) = \frac{1}{2} \times steepness \times Dir.x \times frequency \times amplitude \times \left(\frac{\sin(S)+1}{2} \right)^{steepness-1} \times \cos(S)$$

Where

$$S = Dir(x, y) \cdot Pos(X, Y) \times frequency + t \times \phi$$

Differentiation with respect to y follows similarly:

$$\frac{\partial}{\partial y} f(x, y, t) = \frac{1}{2} \times steepness \times Dir.y \times frequency \times amplitude \times \left(\frac{\sin(S)+1}{2} \right)^{steepness-1} \times \cos(S)$$

For the final surface normal we compute each component as follows and normalize the result:

$$Normal_{final}(x, y) = \left(-\sum_{i=0}^n \frac{\partial}{\partial x} f(x, y, t), -\sum_{i=0}^n \frac{\partial}{\partial y} f(x, y, t), 1 \right)$$

3.6 Multi-threading

In the past many game engine designers have used threading in their games. Typically this is for functionality that maps well to threading on the task level. These architectural decisions were often made not so much to increase performance as they were to simplify coding. In this case we would like to explore using multiple threads to increase performance. Here, we limit ourselves to the simplest case of two threads: one to handle the initialization, rendering, and other aspects of a game engine, and one thread for water wave simulation.

At a very high level a game can be broken into three tasks: initialization, world update, and rendering. We are going to focus on threading the world update of our workload since this has to take place each frame and will provide us the most benefit. Figure 3-2 has a diagram showing how the workload will be partitioned. Thread A manages the game initialization and rendering, thread B handles the vertex position and normal generation for our water simulation.

Next, we need to think about the implications of threading a graphics application. Since threading of DirectX can greatly decrease the performance of an application we want to avoid this. The reason DirectX actually slows down is due to the thread safe version of DirectX only permitting one thread to enter the API at any one time. In some cases this may be the right decision but for our water simulation we decided to keep all rendering in one thread.

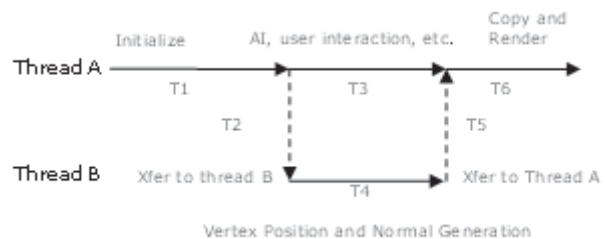


Figure 3-2. Two Threaded Simulation

A simple diagram of a two threaded simulation and how we load balanced our water simulation for each iteration of the render loop. TN represents the time it takes to do the operation in each thread.

Thread A is our main thread that controls the initialization, AI, user interaction, rendering, and shutdown sequence. Thread B will do our simulation. In this case thread A would just sit idle because we only have the water to simulate. One way to think about this is that the work given to thread A and thread B should be balanced such that neither thread sits idle waiting for the other thread to complete, or at the very least that this time be kept to a minimum. This load balancing technique generalizes as we increase the number of threads used for simulation: to obtain maximum benefit from threading, spread the workload as evenly as possible across the available threads.

Implementation

Our first implementation was inspired by [2]. However, the need for neighbor information does not make it amenable to a parallel implementation[1]. Also, the lack of parameters to control the surface was not what we were looking for. Modeling the wave surface as an elastic membrane forces us into an ‘add energy then let it go’ way of thinking, when really we want a repeatable, rolling wave simulation. Therefore, we favored the implementation described in [3]. This implementation has a number of advantages described in Chapter 2.

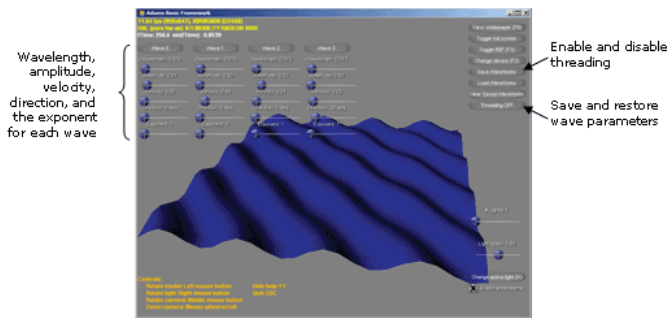


Figure 4-1. Deep Ocean Wave Simulation

In Figure 4-1, we present our implementation. In the upper left one can see the controls for each of the sine waves that control the surface properties. On the right, a button that allows us to switch between threaded and non-threaded implementations, adjust how the normals are calculated and save parameterizations for future recall. Our demo is adapted from the BasicHLSL demo from [11].

4.1 User Interface

One feature of the work in [3] is the ability to have full control over all the surface parameters. Our implementation features the ability to control in real time all surface parameters: amplitude, velocity, direction, and the exponent from Equation 2. Additionally, these parameters can be saved and recalled for later simulations. To compare the threading and non threaded versions there is a button on the right hand side of the GUI. The controls for the waves can be removed so as not to block the view of the simulation.

4.2 C++ Implementation of Sum of Sines with Exponent

Next, we present the actual method used for our multi-threaded CPU implementation of the sum of sines approach. This is adapted directly from Equation 2

```
void CSinWaterMesh::TakeStepSumOfWavesWithExp( float t,
                                                int numOfWavesToSum )
{
    for( int i=0; i<m_iNumRows; i++ )
    {
        for( int j=0; j<m_iNumCols; j++ )
        {
            for( int k=0; k<numOfWavesToSum; k++ )
            {
                CVector3 posVect;
                float dotresult = 0.0f;
                float phase_constant = 0.0f;
                float final = 0.0f;
                posVect.Init( m_pVB[i*m_iNumCols+j].x,
                            m_pVB[i*m_iNumCols+j].y,
                            0.0f );
                if( m_bSumWave[k] )
                {
                    dotresult = m_direction[k].Dot( &posVect );
                    dotresult *= ( 2*(float)MYPPI ) / m_wavelength[k];
                    phase_constant = t*
                        ( m_speed[k]*2*(float)MYPPI / m_wavelength[k] );
                    final = ( dotresult + phase_constant );
                    final = ( sin(final) + 1.0f ) / 2.0f;
                    final = m_amplitude[k] * pow( final, m_kexp[k] );
                }
                else
                {
                    final = 0.0f;
                }

                if( k!=0 )
                {
                    m_pVB[i*m_iNumCols+j].z += final;
                }
                else
                {
                    // The first wave calculated will overwrite the
                    // summation from the last frame.
                    m_pVB[i*m_iNumCols+j].z = final;
                }
            }
        }
    }
}
```

4.3 Normal Calculation

Equation 4 presents an explicit calculation for normal generation and was the approach we expected to find best. However, we found it much faster to calculate vertex normals by averaging the face normals. The key to speeding up this implementation was to know who the neighbors were for each vertex without having to search per frame. To do this we pre-calculate a neighbor list for each vertex. This is not possible with a DX9 GPU based implementation because we do not have access to neighbor data, but on the CPU this is much faster than the calculation using the derivative. Therefore, if using a GPU Equation 4 is still the best way to do normal calculations, but for the CPU a traditional averaging of face normals is faster, including the time, of course, to calculate the new face normals.

4.4 Multi-threading

For the first implementation, the goal was to get a multi-threaded version of the demo to run and calculate updated surface normals and position correctly. The simplest way to do this was to create a function which wrapped the mesh update function inside a thread and create a new thread whenever the mesh needed to be updated. So, once per frame, a new thread would be launched to compute the new height values for the mesh. It worked, but the performance of this implementation was not good.

For the second implementation, the on-demand thread model was replaced with a thread pool model. In our implementation, we only need a second thread to help the main rendering thread so we only have one thread in our *thread pool*. The idea behind a thread pool is to create the threads at startup and have them available when needed by the main thread. This eliminates the penalties in starting up and shutting down threads every time one is needed. The disadvantage of a thread pool is resources allocated to threads when they are not running. Additionally, depending upon how a thread pool is implemented, there may be threads that are needlessly taking CPU cycles in idle wait loops. To compensate for this one can use a strategy of periodic polling or use an OS synchronization object. The idea is to not be stuck in a spin loop consuming CPU resources; instead just periodically poll to see if we have the data we need to run. The idea was to reduce the overhead associated with the second thread by creating only one thread and removing that cost from the render loop. We simulate a real game engine workload by putting an additional workload in thread A to perform while thread B is computing the mesh and normals. This workload can take a variable amount of time and is meant to represent the other aspects of the water simulation that runs independent of the result of the water wave solver. Examples would be user inter-

action, other physics calculations such as collision detection, AI, etc. If one were running a water simulation the work could be partitioned at the *task level* where we place the work of surface calculation in one thread and normal generation in another. Another alternative would be to perform *loop level* decomposition and do a portion of the grid on one thread, a portion of the grid on another thread, and have a bit of overhead where the grids are stitched together if neighbor information is needed.

Thread Creation

To create a thread we use the function `__beginthreadex(...)`. We chose this implementation based on the tradeoffs between the win32 functions and C run-time implementations of threading discussed in [1]. Basically, `__beginthreadex(...)` has less problems and is more reliable with the same functionality as `CreateThread(...)`. Indeed, this was verified by our own experimentation. The documentation in Microsoft Visual Studio.Net 2005* states that using `CreateThread(...)` with the C run time can cause small memory leaks when the thread calls

```
#include <process.h>
HANDLE hThreadHandle; //unsigned long
DWORD dwThreadId;
.
.
hThreadHandle = (HANDLE) _beginthreadex(void *security,
    unsigned stack_size,
    unsigned (__stdcall *) (void *),
    void *arg,
    unsigned initflag,
    unsigned *threadaddr,
);
```

The parameters to the call `__beginthreadex(...)` are as follows: the first parameter is to a security attributes structure. Setting this parameter to NULL means the thread gets a default security level. The second is the stack size. If this parameter is set to 0 the stack size will be set to the same as the current thread. The next parameter is the address of the user function that the new thread will call when it begins executing. The three remaining parameters are as follows: `arg` is a value passed to the new thread, `initflag` is an additional flag to control the state of the thread on thread creation, and finally an address to write the thread identifier. For our application you can see the call in the code:

```
hThreadHandle = (HANDLE) _beginthreadex( NULL,
    0,
    LaunchTakeStepThread,
    (void*)&g_time,
    0,
    (unsigned int*)&dwThreadId );
```

Thread Execution

Now that the thread has been created, it waits using the `sleep()` function until its told to work. The `sleep()` function is passed the sleep time in milliseconds. Passing it a value of 0 causes it to give up its current time slice and wait until it is called again. When we are ready to have the thread begin execution, the master thread will tell the helper thread by incrementing a shared variable. The shared variable is a location in memory that both threads share. The master thread increments the value to tell the helper thread it has consumed the values and is ready to receive the next mesh. The helper thread computes the next set of data, sets this value, and goes to sleep until awakened again by the master. This is known as a producer/consumer relationship.

The body of the helper thread, the producer:

```
while(we are not exiting the thread)
{
    TakeStep();
    Time += Time_Increment;
    bStepComplete = true;
    while(bStepComplete && !bExitThread)
    {
        Sleep(0);
    }
}
```

The master thread consumes the mesh produced by the helper thread and tells the thread to go ahead and compute the next mesh.

```
while(1)
{
    while(!bStepComplete)
    {
        // Wait for the grid update to finish
        Sleep(0);
    }

    //Copy the vertex info to the "real" VB.
    g_pGrid->CopyVBToRenderVB();

    //Allow the other thread to compute a new set of vertices
    g_pGrid->ResetStepComplete(); //resets bStepComplete
}
```

`ResetStepDone()` functionality depends on the methodology for mutual exclusion. For the critical section code we must enter the critical section, set a value, and leave the critical section. For interlocking, we will use an interlocked decrement. Each of these is considered in Section 4.5.

Thread Deletion

Using the implementation described above, the helper thread is automatically deleted when the function we started with `_beginthreadex(...)` reaches the end. In some cases thread deletion would require `_endthreadex(...)`, however this was not necessary for our implementation.

4.5 Mutual Exclusion

As mentioned in the Thread Execution section, our simulation is in essence a producer/consumer relationship where the helper thread is the producer and the main thread is the consumer of water meshes. In a typical producer/consumer model, the producer puts completed data items into a storage space, where the consumer then removes and consumes those data items. The size of this queue limits how far "ahead" of the consumer the producer can go. We decided to pay the synchronization cost every frame, and not allow the producer to proceed on the next frame's data until the current frame had been consumed. The reason for this is that in many games, the next frame relies on data from the current frame, such as AI or physics calculations, or player input.

In our implementation, synchronization is done by both threads polling a state variable, `m_bStepDone`, which reports whether the mesh has been updated since the last time a mesh was consumed. To protect this state variable, we tried two methods of synchronized access described in Aaron Cohen's book – *Win32 Multithreaded Programming: interlocked accesses and critical sections* [1]: interlocked accesses and critical sections. Choosing between the two is application dependent. Interlocking tends to be easiest and best when using a shared variable. Critical sections are more general and can be used anywhere in the code to wrap sections of code or data at a larger granularity. The example that comes with the article permits either implementation to be compiled. They are described as being the fastest synchronization primitives available in Windows.

Interlocked accesses are done through a set of functions which map directly to atomic CPU instructions for read-modify-write scenarios. These atomic operations prevent situations where the variables get into an incorrect state due to read/write ordering issues between the threads.

There are only three functions for interlocking:

`InterlockedIncrement()`, `InterlockedDecrement()`, and `InterlockedExchange()`.

The `InterlockedIncrement` and `InterlockedDecrement` each allow an increment or decrement of a long value respectively. The `InterlockedExchange` permits a swap of one value with another in a single atomic operation.

Critical sections are synchronization primitives which can be used to protect code or data through the principle of mutual exclusion. Before executing the protected code or accessing the protected memory, a thread must acquire the critical section. If the critical section is not available, which occurs when another thread has already acquired the critical section, the thread is blocked until the critical section becomes

available. Note that several different resources, code and/or data, may be protected by a single instance of a critical section, depending on the needs of the application.

Visual Studio Properties

Be sure to set the properties in Visual Studio properly. Specifically, the compiler flags for VC++, Project->Properties->Configuration Properties->C/C++->Code Generation->Runtime Library->select appropriate multi-threaded library. Figure 4-2 shows where this is located on the Property Pages in the Runtime Library section.

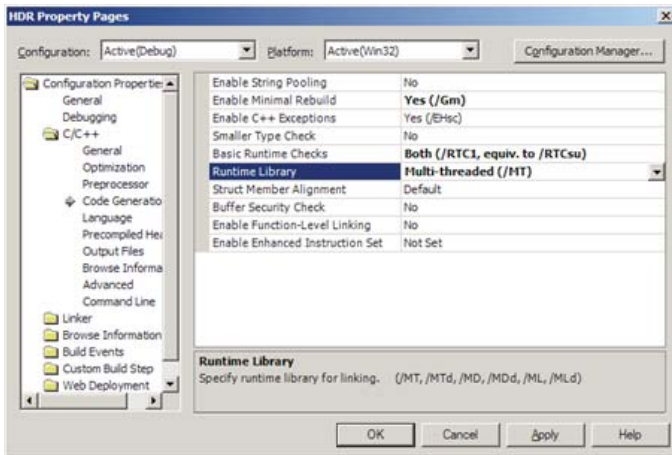


Figure 4-2. Property Pages Location in the Runtime Library Section

5.1 Code Optimization

After developing the workload, we used VTune™ Performance Analyzer for performance analysis. VTune analyzer showed that most of the time the helper thread was executing a function, LookupTriIndex() to lookup triangle indices for every vertex 6 times, to determine which triangle normals to average to compute the vertex normal. LookupTriIndex() is itself a linear function, so the entire normal calculation process was O(n²). The normal calculation process needed to be redone. We had two options: speed up the normal lookup process or calculate the normals directly using the partial derivatives of the wave equations. For the sake of comparison, we implemented both of these options, and left the original algorithm in as well.

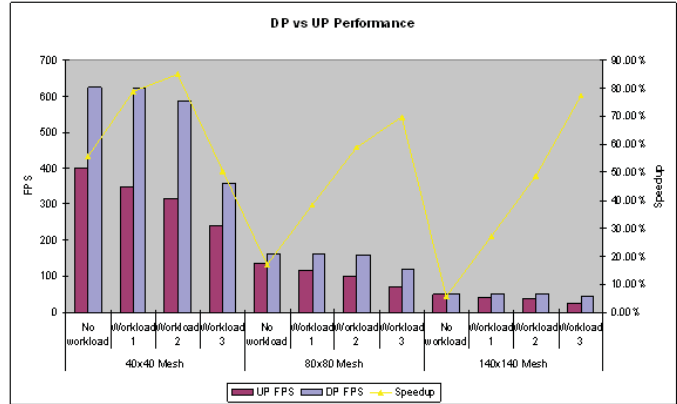


Figure 5-1. Single Processor vs. Dual Processor Performance

Figure 5-1 displays the frames per second with 3 different mesh sizes. A single processor (UP) and a dual processor (DP) implementation are compared. The workloads increase in the amount of time they take to execute and are the simulations of the other aspects of a game engine. For example: AI, collision detection, etc.

5.2 Performance Analysis

We have several dimensions of performance to discuss. To validate our implementation we created 3 workloads to simulate 3 different scenarios, each increasing in time that processor one takes to complete its work and request the results from thread 2. In the single processor case, performance decreases as the simulated workload is increased. This is expected: there's more work to be done, and one CPU resource to perform that work. Note that in the dual processor cases, there is little or no absolute performance (FPS) difference whether there is no workload, workload 1 or workload 2. Another way of saying this is that on a DP machine, workload 2 is essentially free in a properly load balanced situation. But for all 3 meshes, workload 3 does cause the frame rate to drop on a dual processor system. This shows that the simulated workload on the primary thread now takes longer than the mesh and normal generation on the helper thread. Therefore, performance is now bound by the simulated workload rather than by the geometry calculations.

5.3 Load Balancing

Looking at relative performance, we see that with no simulated workload for the primary thread, the benefit of two processors gets less as the mesh size increases. With workloads 1 and 2, for both larger meshes, we know that absolute performance did not change, thus relative performance is better with the increased workload. One may wonder how it is possible that absolute performance could decrease, yet relative performance increase as shown by workload 3 on both larger meshes. This indicates that workload 3 is “closer” to the ideal balance than workload 2 for those mesh sizes. Presumably, the

ideal workload balance would be found between workload 2 and 3. Looking at the 40x40 mesh, the ideal workload balance appears to be between workload 1 and 2. The key result was as expected: *multithreaded performance is best when each thread has a large workload relative to the thread overhead and the workloads are well-balanced with respect to each other.*

Future Work

While we have completed our work on deep ocean wave geometry, there are still a number of issues for us to tackle for a truly compelling ocean water simulation. First, we would like to compute normal maps using the same basis as for our deep ocean waves. This will improve our simulation and better reflect the higher frequency wave components generated by wind on the top of the surface—this will not be true geometry but have all the costs and benefits of normal maps in other situations. There are a number of things we can do to improve the lighting and shading of the surface. Most importantly, the calculation of lighting taking into accounts the reflection and refraction vectors. We would also like to incorporate some of the techniques presented in [8] to improve this lighting with High Dynamic Range Imaging techniques. We would also like to research the simulation of foam on the water surface.

As for threading, there are also a few issues left to explore. For example, we would like to compare this implementation to one in which we do loop level decomposition. Additionally, we are interested in exploring how well this implementation scales with additional threading and doing implementations with some of the work with threading on the CPU and some of the work on the GPU, as well as using an OS level synchronization object for synchronization of threads.

About the Authors

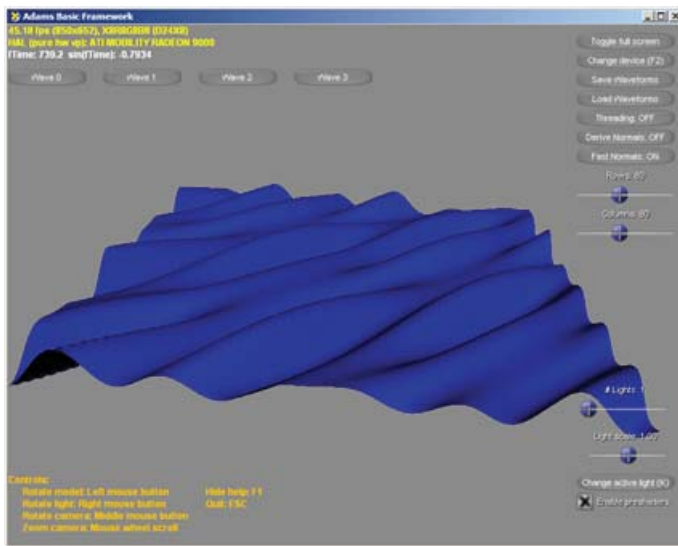
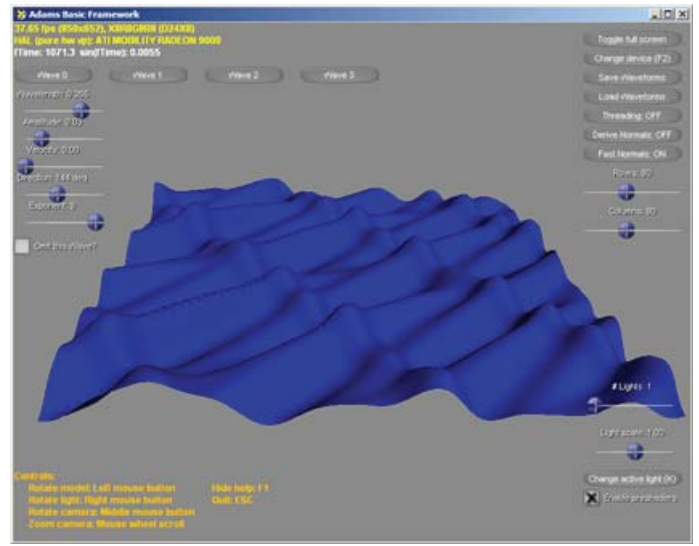
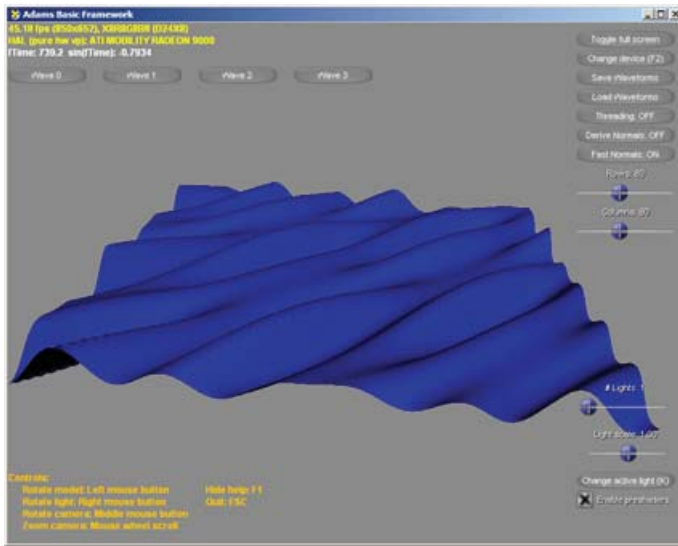
Adam Lake is a Sr. Software Engineer in the Software solutions group leading The Modern Game Technologies Project specializing in next generation computer graphics algorithms and architectures.
Read More

David Reagin is a software engineer at Intel Corporation, where he validated microprocessor designs for 7 years before pursuing his interest as a graphics developer and evangelist. He holds a B.S. in Computer Science from Georgia tech.

References

- [Cohen98]** Aaron Cohen and Mike Woodring. Win32 Multithreaded Programming. O'Reilly and Associates. 1998.
- [Gomez00]** Miguel Gomez. Interactive Simulation of Water Surfaces. Game Programming Gems 1. Edited by Mark Deloura. Pages 187-194.
- [Finch04]** Mark Finch. Effective Water Simulation from Physical Models. GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics. Edited by Randima Fernando. Pages 5-29. 2004.
- [Isidoro02]** John Isidoro, Alex Vlachos, and Chris Brennan. Rendering Ocean Water. Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks. Pages 347-356. 2002.
- [Tessendorf01]** Simulating Ocean Water. SIGGRAPH2001 Course Notes. 2001.
- [IMDB04]** Credits for Titanic. <http://us.imdb.com/title/tt0120338/>.
- [Vterrain04]** Web site: <http://www.vterrain.org/Water/>. Index for Water Simulation. 2004.
- [Lake04]** Adam Lake and Cody Northrop. Real-Time High Dynamic Range Environment Mapping.
- [QuickMath04]** <http://www.quickmath.com/>. November 4, 2004.
- [Giancoli85]** Douglas Giancoli. Physics, Principles with Application, 2nd edition. Prentice Hall, Inc. 1985.
- [MSSDK04]** Microsoft Corporation DirectX 9.0 SDK Summer 2004 Update. <http://www.microsoft.com/downloads/search.aspx?displaylang=en&categoryid=2>. August 2004.

Additional Images



INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

Copyright © 2008 Intel Corporation. All rights reserved. Intel, and the Intel logo, and Xeon are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

Printed in USA

 Please Recycle

