Insomniac's Water Rendering System

Mike Day
March 2009

This document describes the water technology developed by Insomniac for our *Resistance 2* title.

Supported features

- view-driven mesh refinement / level-of-detail scheme with geomorphing
- 'ambient' (procedural) waves with dispersion property
- 'interactive' waves with dispersion property (which we believe is a technological first for games)
- real-time procedural normal maps, with distance-based cross-over into geometry
- optical effects including Fresnel reflectance, cloudy water, and refraction
- special clipping technique to render surface nearer than the near plane
- procedural underwater caustics
- simple particle splashes
- time-critical routines all down-coded to assembly

Introduction

Fundamentally, the system simulates waves propagating across a planar surface. Each component wave is a sinusoid, and a detailed height field is generated by summing many such waves. The Fast Fourier Transform (FFT) provides an efficient way to perform the accumulation of the waves. For general background in this technique, see the classic paper by Tessendorf [1]. The paper describes the method used for creating ocean water in the film 'Titanic', where the offline generation of images provides the luxury of using very large arrays in the FFT calculations – 2048 × 2048 grid points, with a grid resolution of 3cm. In real-time applications though, it quickly becomes apparent that the size of FFT array must be severely restricted. Consider also the need to render views in which details smaller than 3cm are visible. Our solution is to combine detail at many different scales. We employ only 32 × 32 FFTs, but many such height fields are summed from long wavelengths down to short wavelengths as required by each particular viewpoint. This is not unlike rendering fractal terrain, the major difference being that in the case of water the detail added at each level of detail (LOD) must be animated in a particular way.

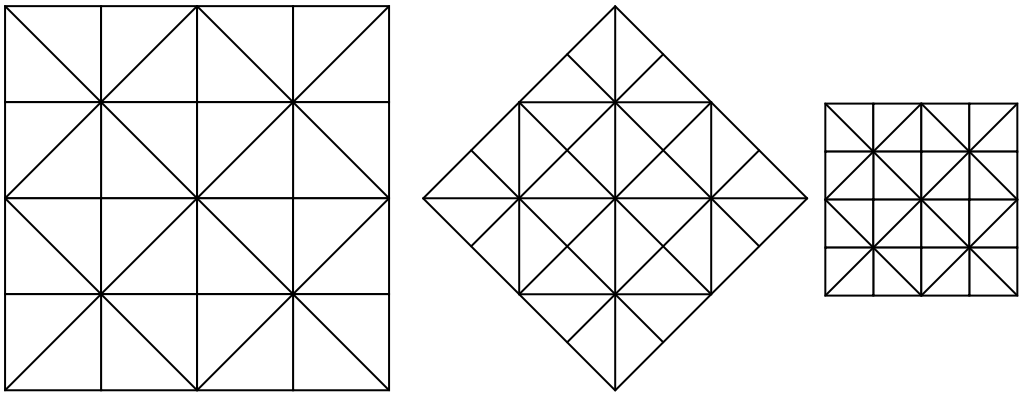
We decided early on to devote much of the development budget to the modelling of interactive waves – waves emitted by local disturbances such as the impact of a projectile or a character wading through the water. The result is that while we feel the system is still lacking in certain features, we believe it demonstrates a technological first in games: interactive waves with the property of *dispersion* (a physically-based dependency

between wavelength and wave velocity). In future revisions we plan to add the features which were sacrificed for the benefit of our new interactivity technique.

Level of detail scheme

The underlying mesh is a world-aligned horizontal grid. Rather than existing at one fixed resolution though, the grid is locally refined in such a way that the apparent diameter of each grid square remains roughly constant. When we look out across a large body of water, we might be looking at part of the water from 1000m away, and a different part of the same body of water just 1m away. In a case like this the distant part will be rendered using grid squares which are very large on an absolute scale (i.e. in world coordinates), and the close-up part using grid squares roughly 1000 times smaller, such that the grid squares of both regions appear to be about the same size on the screen. We would also expect to see all intermediate sizes of grid square tessellating the gap between background and foreground. The tessellation provides the topology of the mesh to be drawn, but each LOD also provides the opportunity to add in the FFT-generated heights appropriate to the LOD's scale.

The nature of the grid refinement scheme follows the classic method of bisecting right-angled triangles (see, for example, the ROAM-based terrain rendering algorithm [2]). The diagram below shows the relationship between 3 consecutive LODs in our water surface.



Note that

(1) the mesh for each LOD is obtained by adding grid points on the hypotenuses of the triangles in the parent LOD
(2) the dimensions of each LOD are smaller than those of the parent LOD by a factor of $\sqrt{2}$
(3) odd LODs are rotated at 45° with respect to the coordinate axes.

An early decision was made to add geometric at both even and odd LODs so as to obtain a higher-quality surface with more detail and smoother transitions between levels. With hindsight though, this is probably a decision we would reverse since the additional quality gained did not justify the more complex implementation. For example, normal mapping (described later) became much more problematic.

Our system supports 40 LOD's, ranging in size from 128m per grid square down to a fraction of a millimetre per grid square. The dimensions for the lower end may seem a little extreme, but these come into play when the camera passes through the water surface, which will be described later.

The highest LOD captured in any given region of the screen undergoes a cross-fading procedure to fade it in smoothly with distance. This process, sometimes referred to as geomorphing, masks the LOD transitions which would otherwise cause popping as the view shifts and as the waves animate. The result is a significant improvement in overall rendering quality.

Ambient Waves

We use the term 'ambient waves' to describe the fully procedural waves summed by FFT. Each wave component is an infinitely repeating sinusoid, which propagates at constant speed in a constant direction, maintaining a constant amplitude. Each LOD comprises 1024 such waves, whose wavenumbers correspond to the points of a 32×32 grid.

The direction of propagation of each wave is fixed by its position in the 32×32 grid of frequencies. The speed of propagation is also fixed by the dispersion relation described below. The amplitude is derived from a pseudo-random wave spectrum, which takes wavelength into account so that waves of shorter wavelength are also proportionally smaller in amplitude.

The time-dependent state of each wave component consists purely of its phase, stored as an angle. Updating each phase angle is trivial – add an increment, calculated as the product of the time step and a precomputed angular velocity. Precomputing the angular velocities allows the desired dispersion relation to be implemented with no run-time overhead:

$$\omega^2 = gk + \frac{\sigma k^3}{\rho}$$

with ω the angular velocity, g acceleration due to gravity, σ the surface tension, ρ the fluid density, and k the wavenumber of the wave component. This encapsulates both deep-water gravity waves and small-scale surface tension waves (though the latter can only be seen in very close-up views).

It is not necessary though to store and update the wave phases for all 1024 ambient wave components of each of the 40 LODs. To see this, consider a smaller-scale example – 8×8 wave components in each LOD. Tabulating these components as they appear in frequency space, we'll write out the square of the wavenumber (k^2) for each component:

0	1	4	9	16	9	4	1
1	2	5	10	17	10	5	2
4	5	8	13	20	13	8	5
9	10	13	18	25	18	13	10
16	17	20	25	32	25	20	17
9	10	13	18	25	18	13	10
4	5	8	13	20	13	8	5
1	2	5	10	17	10	5	2

The DC component appears at the upper-left with a k^2 of 0; stepping across the first row we see perfect squares because the corresponding wave directions are parallel to the *x*-axis and thus have a z-component of 0, and the maximum occurs after stepping halfway across the grid with the component at the Nyquist frequency. The same pattern occurs down the left edge, corresponding to waves moving in the *z*-direction. All other values in the table are pairwise sums of these values.

Now let's list the distinct values for k^2:

0, 1, 2, 4, 5, 8, 9, 10, 13, 16, 17, 18, 20, 25, 32

and we find there are only 15 in the list – significantly fewer than the 64 components in the table, the reduction being due to the symmetries in the table. With 1024 components in the table we find there are only 135 distinct entries, roughly one-eighth of the original dataset due to the 8-fold symmetry of the table. This permits an 8-fold reduction in the size of persistent data, and the cost of reconstructing the 'palette' of complex values from angular arguments is correspondingly reduced. The only additional cost imposed by this method is in reconstructing the full table from the much smaller palette of complex values, but this is a trade-off well worth making as the net result is faster and has a much smaller memory footprint.

With this paletting scheme in place for ambient wave components, the update stage simply consists of rotating each of the 135 phase angles of each LOD by a frame-rate-dependent incremental angle. The rendering stage requires the 32×32 height maps to be

constructed for the currently visible LODs. Construction of a height map involves generating the 32×32 grid of points in frequency space, using both the compressed palette of phase angles and a procedural wave spectrum, performing an inverse FFT to generate complex heights, and discarding the imaginary components. (Actually this process could be improved upon by 'two-for-one' FFT techniques which avoid wasting half the transformed values, which we plan to add in a future revision.)

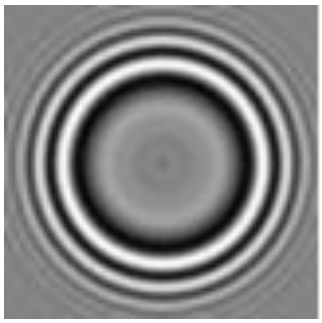
Interactive Waves

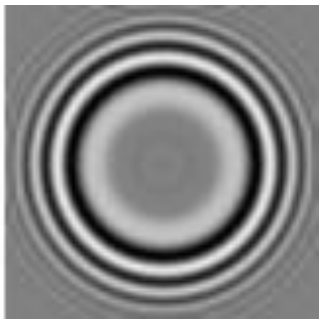
'Interactive waves' is the term we use to describe waves emitted by local disturbances in the water.

A commonly-used technique in games for modelling interactive water waves is to perform a numerical solution of the wave equation using finite differences (see, for example, [3]). This is cheap but effective, and served the games world well for a generation or two of consoles. However, now that we use higher-quality methods for procedural (non-interactive) waves, simply adding this cheaply simulated variety of interactive waves reveals the inconsistency in quality, and this is visible in some contemporary games. The problem is simple – the high-quality procedural waves are dispersive since they use spectral methods, where the cheap interactive waves are not. Built into the wave equation is the constant *c*, the speed of propagation of waves, and in the cheap simulation all interactive waves will move at this speed. However, since the source of interactive waves can cause oscillations at arbitrary frequencies, a whole spectrum of waves ought to be generated, and each component would ideally be moving at its own speed according to the dispersion relation. Our goal was to model interactive waves with a quality comparable to ambient waves; in other words, we sought to build the dispersion relation into the interactive wave simulation too.

We solve this problem by applying Fourier-based methods to interactive waves, so that wave spectra can be represented. One might initially notice an apparent contradiction – Fourier techniques, which use only periodic basis functions, being used to model disturbances which are purely local and hence non-periodic in nature. Indeed the Tessendorf paper raises exactly this objection. Actually though, Fourier transforms can be applied in a special way, somewhat differently from the way we used them in the ambient wave treatment.

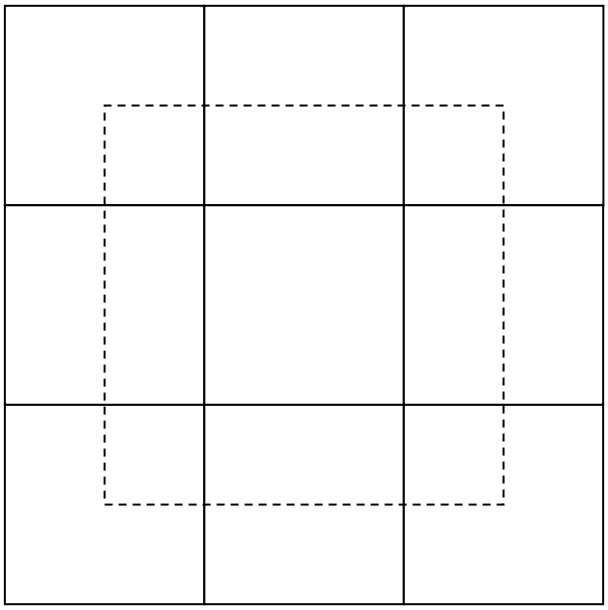
A paper by J. Loviscach [4] presents the idea that waves arising from arbitrary disturbances can be propagated over a given time step Δt by appealing to the Huygens-Fresnel principle [5]. Under this method, one convolves the complex height field with a kernel which exactly duplicates a snapshot of the waves given off by a point disturbance after time Δt. The real and imaginary parts of this kernel look like this:





For our purposes, it's not possible to directly apply the methods of this paper, because (1) the time steps modelled in the paper are prohibitively long, ranging from 0.2s in the simulation of raindrops up to 4s in the simulation of a ship's wake; and (2) because the time step is baked into the convolution kernel making it hard to support simulations at multiple scales and also hard to adjust to a variable frame rate. We also appear to be faced at the outset with the following conundrum: if we were to apply the method directly, but with a time step of $1/60^{th}$ of a second (or $1/30^{th}$), and a typical in-game grid resolution (say, 0.1m intervals) surely all the circular ripples will remain bunched up inside the central sample point and our convolution kernel will consist of a single central spike? We have a solution to this, which will be explained once the other key ingredient has been introduced.

The other aspect to such a convolution-based approach is efficiently performing the convolution itself. We'll typically have not only a large number of data points in our interactive simulation, but they will also be strewn about the water surface and not conveniently arranged in a square array. To perform the convolution we use a variant of a method from signal processing known as *overlap-save* [6], closely related to a possibly better-known algorithm, *overlap-add* [7]. A good reference for background on both algorithms is [8]. Either algorithm could be used, but overlap-save seems to be more suitable in the context of spu processing. At the heart of both algorithms is the notion that while a cyclic convolution can be computed efficiently using an FFT and its inverse (together with point-wise multiplication in the frequency domain), in order to compute a *linear* convolution (which we need in this case) the FFT/IFFT must be deployed in a special way. In the 1D case, the overlap-save approach involves breaking a long input signal into short overlapping segments and performing an FFT and IFFT on each segment, discarding some data 'polluted' by the cyclic wrap-around nature of the FFT and appending the remaining valid data to the output. In 2D this can be used to process square tiles taken from a larger data set. In the water system we modify standard 2D overlap-save by centring each valid output region symmetrically within the overlap region as follows:



The nine solid squares represent neighbouring tiles (not all of which may be explicitly stored, in the case where only some of them contain non-zero activity), with the central tile the one being currently processed, and the dashed region shows the area transformed by FFT/IFFT. Data must be gathered from portions of up to 9 neighbouring tiles to form the input, substituting zeros where a tile is not explicitly represented. In our water system, the interactive tile size is 16×16, and the dashed area is 32×32 which conveniently allows reuse of the optimized 32×32 FFT routines from the ambient wave system.

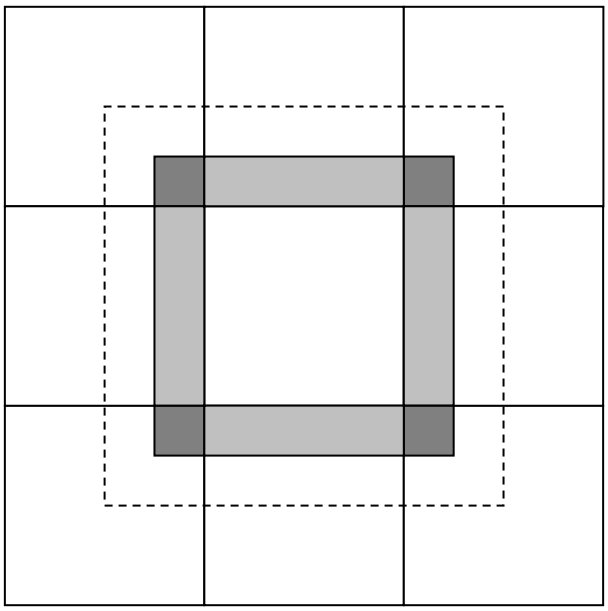
This brings us to the question of what we do to the data in the frequency domain – between the FFT and IFFT stages. We are trying to convolve with a particular function, and in the frequency domain this corresponds to complex multiplication. But by what? Each multiplicand represents the complex phase of a particular spectral component. So the multiplier is merely the change of phase required to advance that component so that its wave propagates at the speed dictated by the dispersion relation. In the frequency domain then, the multipliers are just incremental rotations, and the process here is exactly the same as that for updating the palette of ambient waves described previously and reuses much of the same code.

These incremental rotations will generally be small, and the full array of them represents the Fourier transform of the ‘spike’ shaped convolution kernel which appeared to present a problem earlier. In reality, all the same information is there in the spike kernel too – encoded as tiny variations in all the other grid points. But we never need to directly encode this kernel since we only apply the phase shifts in the frequency domain.

The method described thus far represents the bulk of the number crunching, but on top of this there is a system to manage the creation and deletion of tiles – creation as waves propagate into undisturbed areas and also when new disturbances are added, and deletion once activity in a tile decays sufficiently. In the current system, this management is made difficult by the choice to single-buffer the height map data for the sake of conserving memory. A much lighter set of routines could be used if the data were double-buffered.

Another possible improvement would be to adjust the size of the FFT according to the LOD's frequency band. In the inter-frame time interval, waves in low LODs (long waves) propagate less far across the interactive tile they inhabit compared to waves in high LODs (short waves), and could get away with using a smaller size of FFT – a size intermediate between 16×16 and 32×32. In choosing the FFT size one must ensure that a sufficiently small fraction of the energy in the original signal propagates into the 'polluted' region at the edges. Too small a size for the FFT and we'd be chopping off too much important information at the edges.

Related to this is the issue of when and how to spawn new tiles as ripples propagate off the edges of tiles into unmapped territory. To capture the waves moving off the edge of the tile we slightly abuse the overlap-save algorithm. Consider again the arrangement of neighbouring tiles covered by each FFT:



Shown in shades of grey is a strip of width 4 grid points lying immediately outside the tile being processed. Although this strip forms part of the discarded region, under special circumstances parts of it may be used as the initial height map data for one or more newly spawned tiles. The idea here is that if there is currently no neighbour in a given direction (i.e. all heights are zero), the shaded region in that direction will not be significantly

polluted by neighbouring height data after the convolution has been applied. There may be pollution from information wrapped around from the *other* side, but we're banking on this pollution not having traversed more than halfway across the 8-wide discarded zone – i.e. not having significantly entered the grey zone. Under these conditions we can use the grey zone to initialize a new neighbouring tile (filling the rest of the new tile with zeros). To test whether to spawn a neighbour, we can simply examine the heights in the grey zone to determine either the absolute maximum or the sum-of-squares, and only spawn if a threshold is exceeded. A corresponding technique is used on existing tiles to determine whether their energy has fallen below a certain threshold, in which case they are deleted.

As with ambient waves, multiple LODs of interactive waves are modelled. When a new disturbance is added to the water surface, an assessment is made of the physical scale of the disturbance event to determine which LODs should have their height maps modified (spawning new tiles where necessary). In this way, bullets can generate waves with short wavelength, and a large explosion can give off longer (and larger) waves. The propagation algorithm is run separately for each LOD which has any interactive tiles; no transfer of signals takes place between LODs.

As a final note on this topic, we can legitimately claim that the physics code for the entire water system consists of a single line of code – the one which, given a wavenumber, uses the dispersion relation to compute an angular frequency.

Normal mapping

The refined mesh captures geometric detail only down to a certain resolution, and a typical triangle is still tens of pixels across. In order to capture finer detail, down to a scale on the order of a single pixel, we generate on-the-fly procedural normal maps. The detail encoded in a given normal map consists of a summation of the next few LODs of height map that we *would* be adding geometrically if we viewed the surface from correspondingly closer. We sum LODs in consecutive groups of 5 on spu (i.e. levels 0-4, levels 1-5, levels 2-6, etc) to yield 128×128 normal maps, and in the pixel shader we combine two of the summed maps (e.g. levels 10-14, and levels 15-19). This provides 10 LODs of resolution beyond the geometrically captured detail, and allows us to achieve pixel-level detail. While the technique is simple in principle, it is somewhat complicated by the need to rotate alternate LODs by 45 degrees as they are accumulated on spu, and also by the single-use nature of normal maps generated for interactive waves.

Ideally, normal maps would be cross-faded in the same way as the geometric detail. However it was found without a cross-fade the artifacts are quite well-hidden and we chose to avoid the additional texture look-up needed to cross-fade.

Optical effects

With only a small amount of our development budget (and rendering budget) available for optical effects, we opted to use simple but effective techniques to create the illusion of reflection and refraction.

The reflected image is a blend of artist-created cube map, and a downsampled copy of the frame buffer data. The cube map provides a cheap form of environment mapping, and blending in an amount of the frame buffer image provides a low-cost substitute for a more optically correct reflected image. Waves perturbing the downsampled frame buffer image cause colours to bleed from scene geometry and colour the water beneath them, giving a vague sense that they are being reflected. This is an extremely crude but fairly effective technique.

The refracted view makes use of the captured frame buffer to distort the image visible through (semi-)transparent water. Only deviations from vertical normals are used in the perturbation – a perfectly calm body of water will not create any distortion. Of course this is physically incorrect, since water and air have different refractive indices, but provided there are some waves present the illusion is fairly convincing given the simplicity of the technique. The refracted view also makes use of the captured depth buffer information to determine an amount of fogging to apply, which provides a realistic 'murkiness' to the water which can be easily controlled by colour and density parameters.

Fresnel reflectance is applied in the pixel shader.

Super-near clipping

Under normal circumstances, when the camera is moved through the surface of a 3D model, near-plane clipping comes into effect and a gap appears between the camera and the clipped surface. In most games, the camera is controlled by code which prevents this from ever happening, so that the illusion of solid surfaces is maintained. However, if we wish to be able to move the camera from above water to underwater then we inevitably run up against this clipping problem with the water surface. To solve this problem we introduce a technique we call *super-near clipping*.

Under this technique, the refinement process which generates mesh vertices does not terminate when refinement reaches the near plane: instead we continue to generate more vertices in the extreme foreground between the near plane and the camera, at ever-higher LODs as the mesh moves closer to the camera, with the refinement continuing down to sub-millimetre scales if necessary. This extra geometry must clearly be rendered in a special way though, because if it passes through the same pipeline as ordinary geometry it will be clipped away by the rendering hardware since it lies outside the usual view frustum.

To render the 'super-near' geometry, we hijack the closest one millimeter of the ordinary depth-buffer range, assuming a typical near-plane distance of 10 centimetres – i.e. the

region of depth starting at 0.1m from the camera and ending at 0.101m. Due to the non-linear distribution of depth buffer values, this millimetre-thick band actually contains about 16 bits of resolution when using a 24-bit depth buffer, which is more than enough for rendering the super-near water geometry. The extra geometry is simply remapped into this range by a special vertex program, compressing it into the millimetre-thick band so that it lies fully inside the view volume and yet remains fully self-consistent with respect to depth.

The only penalty of using super-near clipping is that super-near water geometry will not depth-sort correctly with any conventionally-rendered geometry which comes up against the near plane. However, as we discussed, the camera control code ensures that conventional geometry is not generally viewed from such close distances, so this is never a problem in practice.

References

[1] Jerry Tessendorf, *Simulating Ocean Water*.
http://www.finelightvisualtechnology.com/docs/coursenotes2004.pdf
[2] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, Mark B. Mineev-Weins, *ROAMing Terrain: Real-time Optimally Adapting Meshes*. https://graphics.llnl.gov/ROAM/roam.pdf
[3] Miguel Gomez, *Interactive Simulation of Water Surfaces*, Game Programming Gems 1, pp.187-94. http://www.gameprogramminggems.com/
[4] J. Loviscach, *A Convolution-Based Algorithm for Animated Water Waves.*
http://www.j3l7h.de/publications/convolution_web.pdf
[5] http://en.wikipedia.org/wiki/Huygens-Fresnel_principle
[6] http://en.wikipedia.org/wiki/Overlap-save_method
[7] http://en.wikipedia.org/wiki/Overlap-add_method
[8] http://en.wikipedia.org/wiki/Circular_convolution