Real-time simulation of water surface

Vladimir Belyaev

Applied Math. Department,

St. Petersburg State Polytechnical University, St. Petersburg, Russia

vladimir@d-inter.ru

Abstract

The paper is devoted to efficient algorithms for computer simulation of water surface with waves. The water surface is a usual component of virtual environment in trainers/simulators and games and often should look as highly realistic as possible, which leads to sophisticated algorithms. On the other hand, these algorithms are aimed at real time mode, and their high speed is crucial. This paper embraces a number of techniques covering all three steps of water simulation: the surface generation, optical effects imitation and building a polygon mesh. Some of the used techniques are well known; we have adapted them to real time requirements. Some new algorithms were developed. There are presented results of implementation of these three steps altogether: on up-to-date personal computers we can calculate and render the surface 30 times per sec and spend at most 8 ms for each frame.

Keywords: real-time rendering, water surface simulation.

1. INTRODUCTION

Today there are a lot of non real-time tools for creating different types of wavy water surface and even more tools for rendering this surface. At the same time there are not many techniques to provide real-time applications with simulated water of realistic looking visual quality. Of course, major problem is performance restrictions. For example, we had to spend for the simulation calculations not more than 25% of overall processor time and to reach rendering speed 30 frames per second and higher. So our goal was to develop fast enough simulation that gives us realistic visual results and fits into performance restrictions. Our target platform is PIII-733, with GeForce 3 Ti.

Logic of water surface simulation brings us to the idea of three main blocks that form entire simulation task:

- simulation of the water surface itself (getting height field on each frame),
- optic effects (such as reflection, refraction) simulation,
- rendering technique, i.e. the way of forming triangle mesh for visualization of the surface.

In this paper we are discussing methods for simulating wavy, but not stormy water surface in areas away from the shore with reflection and refraction only.

2. PREVIOUS WORK

Water surface simulation itself (here we mean the surface simulation only, i.e. without rendering) is actually very popular direction of research. There are many approaches to the problem. Some of them are based on solving Navier-Stokes equations in 2D [2] and 3D [4]. Some of them employ simplified water models: small amplitude waves [4] and shallow water waves [7, 10]. For example, in the paper [8] a shallow water waves model is used to create waves near the shore. There is a method based on the idea of approximating water volume with columns connected by pipes [1]. It is used to produce animation of splashing fluids. Fournier and Reeves developed a simple method of representing water surface using Gerstner model [5]. The mentioned methods are either highly performance consuming or give (when using too coarse grid) unrealistic results. One of the latest methods is based on using fast Fourier transformation (FFT) to produce tilable height map [11]. This method is rather scalable and so can be used not only for high-quality water animation for video, but in real-time applications also. This is why we have chosen it as a core of our water surface simulation method.

Common approach for the optic effects simulation is ray-tracing or reverse ray-tracing [13]. These methods are widely used in commercial products like Blue Moon Rendering Tools™[1] or 3DS MAX™. One of the models of light scattering in water volume and water color dependence on different parameters is described in [9]. Some real-time methods of realistic simulation are described in [6]. It seems that nowadays it is the most advanced work on real-time optic effects simulation, so we used and evolved some ideas from it.

Actually, in the current papers there is not much information about way of creating triangle mesh for water surface rendering. This is obviously because realistic water surface simulation is still rarely used in realtime applications where highly optimized geometry is really needed. Most realtime applications with the live water are actually only demos like NVidia TLWater demo[2] or Chen's demo[3], in which water surface is bounded by walls of the pool. These demo applications always use uniform grid, which is inapplicable for large water surfaces. Standard continuous LODs

[1] http://www.exluna.com/products/bmrt/

[2] http://developer.nvidia.com/view.asp?IO=demo_pool

[3] http://www.unet.univie.ac.at/~a9104678/ChengineOrg/Pages/Home.htm

techniques like ROAM [3, 12] or adaptive quad-tree [14] are too complex and performance consuming to be used as method of water surface tessellation in real-time.

3. BUILDING WATER SURFACE

As it was said earlier the core of our water simulation is based on Tessendorf's method described in [11]. In short, this method is based on water surface representation given by the following expression:

$$h(\vec{x},t) = \mathrm{Re} \int\limits_{-\infty}^{\infty} \int\limits_{-\infty}^{\infty} (\tilde{h}(\vec{k}) e^{i\omega t} + \tilde{h}(-\vec{k}) e^{-i\omega t}) e^{i\vec{k}\vec{x}} dk_x dk_z \qquad (1)$$

where $\vec{k} = (k_x, k_z)$, $k = \|\vec{k}\|$, $\omega = \sqrt{gk}$ and $\tilde{h}(\vec{k})$ - spectrum that defines surface configuration.

This representation is derived from model of small amplitude waves in the case of absence of waves with cylindrical symmetry [10]. Following Tessendorf, we transform this representation for 2^q x 2^q ($2^q = N$) grid and get

$$h(\vec{x},t) = \sum_{n=-N/2}^{N/2-1} \sum_{m=-N/2}^{N/2-1} (\tilde{h}(\vec{k}) e^{i\omega t} + \tilde{h}(-\vec{k}) e^{-i\omega t}) e^{i\vec{k}\vec{x}} \qquad (2)$$

where $\vec{k} = \left(\frac{2\pi n}{S}, \frac{2\pi m}{S}\right)$ - wave vector, $k = \|\vec{k}\|$, $\omega = \sqrt{gk}$, S – geometrical size of the grid.

It can be proved (by determining $h(\vec{x},0)$ и $\frac{\partial}{\partial t} h(\vec{x},0)$, then using Fourier inversion) that

$$\tilde{h}(\vec{k}) = \tilde{h}^*(-\vec{k}). \qquad (3)$$

Using this equation and changing indices limits we can rewrite (1) as

$$h(p - \tfrac{N}{2}, q - \tfrac{N}{2}, t) = 2(-1)^{p+q} * \qquad (4)$$

$$* \mathrm{Re} \sum_{n=0}^{N-1} \sum_{m=0}^{N-1} (-1)^{n+m} H(n - \tfrac{N}{2}, m - \tfrac{N}{2}, t) e^{i\frac{2\pi}{N}(np+mq)}$$

So, if we know $H(u,v,t)$ – the waves spectrum, we can use FFT to get h values in all grid points at once. Note that according (4) resulting height map is repeatable (tilable).

Oceanographic researches show that (4) describes wind-driven waves in the open ocean close enough. They also show that $\tilde{h}(\vec{k})$ coefficients are Gaussian distributed random numbers with zero mean value and standard deviation depending on $\vec{k}$. Deviation spectrum is well known for different waves kinds. There are several analytical semi-empirical models for this spectrum. Tessendorf used Phillips spectrum for wind-driven waves:

$$P_h(\vec{k}) = A \frac{1}{k^4} e^{-1/(kL)^2} \left|\vec{k} \cdot \vec{w}\right|^2 \qquad (5)$$

where $L = \|\vec{w}\|^2 / g$ is the largest possible wave arising from continuous wind with the speed w and A is just a scale coefficient defining waves height.

We modified this spectrum for better control under waves shape:

$$P_h(\vec{k}) = \begin{cases} A \frac{1}{k^4} e^{-(kL)^{-2}} \left|\vec{k} \cdot \vec{w}\right|^{\gamma(\vec{k}\cdot\vec{w})} * \\ \qquad * e^{-k^2 l^2}, k > k_*, \\ 0, \; k \le k_* \end{cases} \qquad (6)$$

where $\gamma(t) = \begin{cases} \gamma_+, t \ge 0 \\ \gamma_-, t < 0 \end{cases}$.

Parameters k_* and l allow us to cut too long and too short waves, parameters γ_+, γ_- along with w allow us to control oblongness of waves. Making γ_+ and γ_- different leads to water with waves moving in a certain direction.

Knowing P_h we can write down $\tilde{h}(\vec{k})$:

$$\tilde{h}(\vec{k}) = \frac{1}{\sqrt{2}} (\xi_r + i\xi_i) \sqrt{P_h(\vec{k})}, \qquad (7)$$

where ξ_r and ξ_i are ordinary independent draws from a Gaussian random number generator, with zero mean value and the standard deviation equal to 1.

According to the model of small amplitude waves $\omega = \sqrt{gk}$ (in the case of constant depth d, $\omega = \sqrt{gk \tanh(kd)}$ (see [10] for details). Now we have everything to calculate H matrix for FFT.

$$H(\vec{k},t) = \frac{1}{\sqrt{2}} (\xi_r + i\xi_i) \sqrt{P_h(\vec{k})} e^{i\omega t} \qquad (8)$$

Actually, the height map is not all that we need. We also need a normal to the surface at each grid point. In [11] and [6] it is proposed to use another FFT to get normals matrix. But it is still needed to fit into performance restrictions so we just used finite difference method to get the normals. The result of such optimization is visually acceptable.

Since resulting surface is the sum of the number of sine waves, which tops can not be sharp as it is in a real life (see Fig. 1), we need to modify the grid to sharp them. The corresponding method is described in [11] and [6]. In short, every grid point is displaced according to the formula:

$$\vec{x} = \vec{x}_0 + \lambda \vec{D}(\vec{x}_0, t), \qquad (9)$$

where λ is a parameter that controls degree of grid modification.

Offset D is given by the formula:

$$\vec{D}(\vec{x},t) = -\sum_{n=-N/2}^{N/2-1} \sum_{m=-N/2}^{N/2-1} i \frac{\vec{k}}{k} H(\vec{x},t) e^{i\vec{k}\vec{x}}. \qquad (10)$$

But this method has great disadvantage: it needs one more FFT. Instead of it we use simpler way for calculation of the offset:

$$\vec{D}_{xy} = \frac{S}{N}(M_{x+1y} - M_{xy}, M_{xy+1} - M_{xy}), \qquad (11)$$

where M is the height map.

This simplified method produces a bit worse result, but it is still acceptable visually. Parameter λ should be chosen carefully to avoid self-intersection of water surface.

Figure 1. Photo picture of Neva river waves.

It is important to note that normals in the grid points should be recalculated after this modification. For a grid quad with the height h_{ij} at ij-th point, point offsets δx_{ij} and δz_{ij} and grid step s (using expansion in Taylor series) we can write down a simple system of linear equations from which we can get:

$$\Delta = \begin{vmatrix} s + \delta x_{i+1\,j} - \delta x_{i+1\,j} & \delta z_{i+1\,j} - \delta z_{i+1\,j} \\ \delta x_{i\,j+1} - \delta x_{i\,j+1} & s + \delta z_{i\,j+1} - \delta z_{i\,j+1} \end{vmatrix}$$

$$\frac{\partial h}{\partial x} = \frac{1}{\Delta}\begin{vmatrix} h_{i+1\,j} - h_{i\,j} & \delta z_{i+1\,j} - \delta z_{i+1\,j} \\ h_{i\,j+1} - h_{i\,j} & s + \delta z_{i\,j+1} - \delta z_{i\,j+1} \end{vmatrix} \qquad (12)$$

$$\frac{\partial h}{\partial z} = \frac{1}{\Delta}\begin{vmatrix} s + \delta x_{i+1\,j} - \delta x_{i+1\,j} & h_{i+1\,j} - h_{i\,j} \\ \delta x_{i\,j+1} - \delta x_{i\,j+1} & h_{i\,j+1} - h_{i\,j} \end{vmatrix}$$

So, the new normal is:

$$\vec{N} = \left(-\frac{\partial h}{\partial x}, 1, -\frac{\partial h}{\partial z}\right) \qquad (13)$$

4. OPTICS EFFECTS IMITATION

In this work we present two "models" for imitating optics effects. They can be called "reflection and refraction" and "sky reflection".

4.1. "Reflection and refraction"

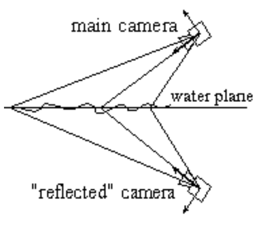

Figure 2. "Reflected" camera

The first model is briefly described in [6]. The base idea is clear: the whole scene (except water) is rendered into main camera forming "refraction" texture. Then camera is reflected as shown on Fig. 2. Rendering into this camera produces "reflection" texture. Then using projection mapping, the textures are mapped onto the water surface and mixed, using Fresnel coefficient. This technique works fine for a plane water. For oscillating water we need to take into account the normal change to get the correct result. It can be done in such way: in every grid point we calculate the reflection ray and intersect it in with the plane raised on h_r above the water plane; intersection point is then projected into camera to form texture coordinate for the grid point. This method is illustrated by Figure 3. Reflection vector is given by the formula:

$$\vec{r} = \vec{e} - 2(\vec{n}\cdot\vec{e})\vec{n}, \quad \vec{e} = \frac{\vec{p}-\vec{c}}{\|\vec{p}-\vec{c}\|}, \qquad (14)$$

where $\vec{p}$ - grid point position, $\vec{n}$ - grid point normal, $\vec{c}$ - camera position.

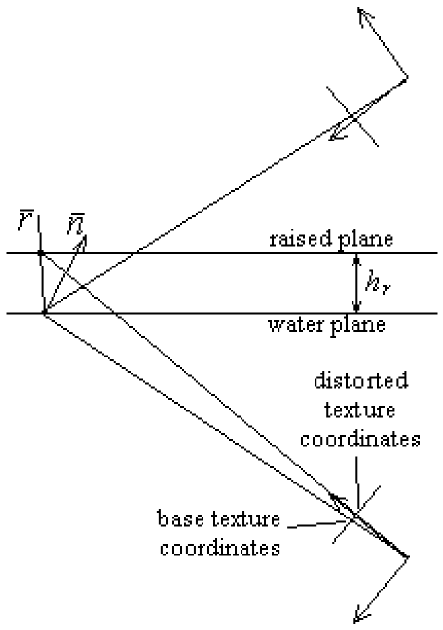

Figure 3. The way of "reflection" texture mapping

This method actually is nothing more than good imitation of reflection.

Texture coordinates for "refraction" are calculated in the same way except for three things: plane is lowered, refraction vector is used and intersection point is projected to main camera. Refraction vector formula is more complex than reflection one:

$$\begin{aligned}\vec{t}(\vec{p},\vec{n}) &= n_{12}^{-1}\vec{e} - \alpha(\tau)\vec{n}, \\ \alpha(\tau) &= n_{12}^{-1}\tau + \sqrt{1 - n_{12}^{-2}\left(1-\tau^2\right)}\end{aligned}, \qquad (15)$$

where $\tau = \vec{e}\cdot\vec{n}$, $n_{12}^{-1} = n_1/n_2$ and n_1, n_2 – indices of refraction.

To mix “reflection” and “refraction” textures we need to calculate Fresnel coefficient at every grid point. Its formula is also complex enough:

$$F(k) = \frac{(g-k)^2}{2(g+k)^2}\left(1+\frac{(k(g+k)-1)^2}{(k(g-k)+1)^2}\right), \qquad (16)$$

where $k = -\tau,\ g = \sqrt{\left(\frac{n_2}{n_1}\right)^2 + k^2 - 1}$.

(We have to notice that formula for Fresnel coefficient given in [6] is wrong. A mistake is in expression for value g.) Since this formula is very expensive to compute per vertex we use its approximation proposed in [6]. We approximate it by the function:

$$G(k) = (1+k)^{-\alpha}, \qquad (17)$$

where α is coefficient corresponding to relative refraction index. Examples of correspondence are given in Table 1.

n_2/n_1	α
1.1	10
1.2	8
1.33	7
1.5	6

Table 1. Examples of relative refraction index and alpha correspondence

To implement this method we need to code two additional renders of the whole scene, code one vertex shader (containing reflection and refraction vectors calculation, code for finding intersection points, projection to cameras and Fresnel coefficient evaluation) and one pixel shader (mixing 2 textures according per-pixel interpolated Fresnel coefficient). But that is only the beginning. If somebody codes this algorithm “as is”, he will surely encounter two problems.

First, there are texture misses. Water with waves reflects a bit more of the scene than the plane water. It also refracts more of the underwater scene than visible in main camera. This idea is illustrated on Fig. 4.

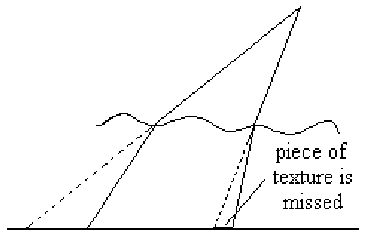

Figure 4. Texture miss for the refraction case.

To cover this texture misses we increase FOV (field of view) for the cameras: a little for reflection camera and on 10-20% for main camera when rendering refraction scene. Also to avoid artifacts if texture miss happens we use *clamp* texture addressing. Fig. 5 shows how camera FOV adjustment affects the picture.

The second problem is more difficult. Let us examine reflection case: the problem appears when underwater objects are rendered into “reflection” texture (see Fig. 6). To avoid this we just need to skip underwater objects while rendering into “reflection” texture. The same situation is for refraction: we must skip objects, which are above water surface. Objects crossing the surface should be clipped. Clipping can be performed by *user clip plane* technique (on GeForce 2) or *texkill* pixel shader instruction (on GeForce 3 or later). In the second case, all objects that intersect water surface should be rendered through vertex shader that copies vertex height above the surface to texture coordinate which is used in pixel shader to mask the pixel that is under/above water.

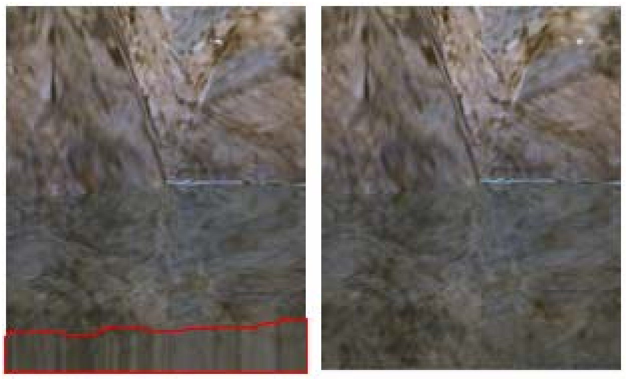

Figure 5. Scene that shows FOV artefacts at refraction (FOV is not adjusted (left); FOV increased 20% (right)). Water surface is the same in both cases.

We must also say some words about texture resolution. Of course, the best case is if the texture resolution matches output resolution. But it might be too expensive. Experiments showed that everything looks fine enough with the texture resolution 1.5 times less then output one. Fig. 11 shows the result of this approach.

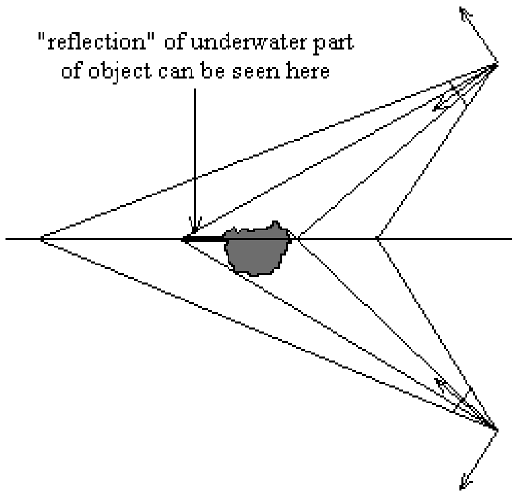

Figure 6. Underwater part of the object is rendered into reflection texture. So it has reflection – impossible situation in real life.

4.2. “Sky reflection”

This method can be used in case of invisible or nearly invisible bottom, for the open water (or the water that does not reflect anything except sky). It is based on per-pixel cubic environment mapping. So we need bump and cubic environment texture. Environment texture should contain sky (or other things to be reflected). Bump is generated from *N* x *N* normals matrix. That is actually a simple operation: lock bump texture, convert normals to RGB values using formula (for RGB bump texture):

$$\begin{cases} R = (x+1)*127 \\ G = (z+1)*127, x, y, z \in [-1,1] \\ B = (y+1)*127 \end{cases} \quad (18)$$

and then unlock the texture. Tiling of the bump texture should be chosen carefully because too frequent tiling is noticeable and looks unrealistic, while too small tiling coefficient leads to pixelization of the bump texture that looks bad.

Now, on GeForce 3, it is impossible to calculate Fresnel coefficient per-pixel, so we still calculate Fresnel coefficient (via approximation) in vertex shader. But in this method we have no refraction texture to mix it with reflection. There are two options: to use constant color as a color of light coming from the volume of water or use Fresnel coefficient as opacity value and to draw some bottom picture.

To implement per-pixel cubic environment mapping we must write pixel shader using instructions *texm3x3pad* and *texm3x3vspec* (see DirectX 8.0 help for details). Therefore we need to calculate texture space coordinate system in the vertex shader. We already have one (y) vector – normal. If the mesh is axis aligned we can reconstruct two other vectors from the normal by formulas:

$$\begin{cases} \vec{x} = \left(n_y, -n_x, 0\right) \\ \vec{z} = \left(0, -n_z, n_y\right) \end{cases} \quad (19)$$

and then normalize them (or better: calculate only *x*-vector and get *z*-vector via cross product). Result of this method is shown on Fig. 12.

This method can be modified for platforms that have feature-poor graphics hardware of high performance. Well-known example of such platform is Sony Playstation® 2. Its high performance Graphics Synthesizer can deal with only one texture and it doesn't support features like cubic environment mapping. But we can simplify this method to fit into such hardware. We just get rid of bump and replace cubic environment mapping with spherical. The result is worse but it still looks like real water surface.

5. RENDERING TECHNIQUE

Rendering technique is mostly depending on the way of constructing mesh representing water surface. During the development of construction method four requirements appeared. By the first requirement, final triangular representation of water surface must allow us fast culling of invisible (out of the camera frustum) triangles. Of course, no calculations should be performed for vertices of the invisible triangles. The second requirement appeared because of artefacts in the final image when using translucency. The problem is in triangle rendering order (it is very important when dealing with translucent objects and z-buffer). When the triangle order was back-to-front, the distant triangles were seen through near ones, thus creating those artefacts. So, in a resulting mesh all triangles should be sorted front-to-back.

To produce water surface of acceptable quality the grid step should be small enough: 10-40 cm. So, we can not use an uniform grid and need some kind of LODs. So the third requirement says: mesh building algorithm must provide enough details near the camera while overall triangles number should be not very big (actually, we used 5000 – 15000 triangles).

The fourth requirement is simple: water surface mesh should be easily stripifiable. That is for improving rendering speed (especially on the platforms like Sony Playstation® 2).

5.1. 3D grid solution

One of possible solutions is a static mesh: a mesh with fixed structure is created on XZ plane. This mesh is moving together with camera, so every time there is water around the camera. Structure and topology of the mesh is not changed during the movement. But in case of rigid binding of mesh to camera one more problem appears. We call this effect "movement of a rabbit inside a boa". Actually, water surface is called "rabbit", while "boa" is mesh. This effect at great camera speed can be seen as jerky color changes in mesh vertices. The reason can be explained in such way: water movement and its color changing depends only on time, but when we move mesh together with camera, change of height and color in mesh vertex become dependent on the camera speed. So when the speed is small, the effect is unnoticeable, and when the speed increases, it looks like we move some static surface under the cloth (that reminds of gulped rabbit moving inside boa). If the speed increases more, the whole mesh begins to jerk.

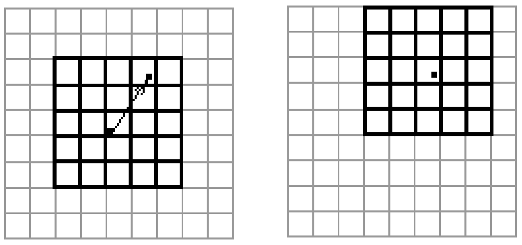

Figure 7. Example of mesh movement to preserve camera's position projection in central square. Gray lines – possible mesh positions, black lines – mesh itself

To avoid this problem the movement of mesh can be quantized. E.g., if mesh is a grid with a constant step, we move mesh only when XZ projection of camera position leaves the central square. We move the mesh to keep the projection in the central square (see Fig. 7). The mesh is an uniform grid, so we can notice its movement only at the border. So, if we hide the border (e.g. by some flat polygons lasting from the grid border to horizon) the movement will be nearly unnoticeable.

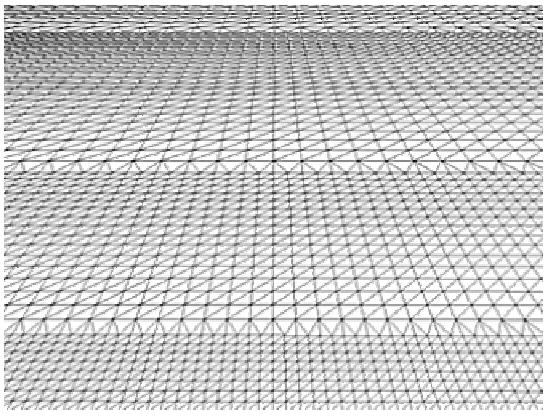

Figure 8. Example of the mesh (camera is looking 45° down**).**

To meet the requirements we build mesh of blocks and solve invisible triangles problem by culling blocks by camera frustum (the first requirement). Blocks must be numbered in a "spiral" order (inner blocks first). If we construct whole block as one stripe with front-to-back triangle order, the second and fourth requirement will be met. To meet the third requirement we use blocks with different steps and add "connection" blocks that have interstep translation triangle line. The result is shown on Fig. 8.

5.2. 2D grid solution

Other possible solution is 2D mesh. If we prohibit the camera tilt we can create uniform axis-aligned 2D mesh in the screen space as *one* strip. Then we just need to find upper and lower borders of the mesh in output window and to evaluate height and normal for each mesh vertex. Raytracing method (shooting a ray through 2D vertex, intersecting it with the water plane, indexing into height and normal map) proved itself inefficient. The trouble is in indexing into matrix. If we do not blend four neighboring matrix elements we will get some sort of pixelization, if we do we will lose performance. Solution is in using render-to-texture. The final algorithm is as follows:

1. On create we build a static 2D grid (M x N cells) that covers whole output window.
2. On frame update we project water plane into screen space and determine water borders in screen space.
3. Then we code height & normal map into RGB texture (8-bit per color channel). Let us call it *"matrix"* texture. We lose some precision here, but it is nearly unnoticeable.
4. We map "matrix" texture onto the water plane and render the plane into main camera with *"grid"* texture's (M+1) x (N+1) region (2D grid is M x N cells) as target. "Grid" texture is also RGB texture with 8-bit per channel.
5. Then we lock "grid" texture for read and decode it into 2D mesh (note that "grid" texture region completely matches 2D grid).
6. At each used vertex we calculate reflection vector, then texture coordinates, then "project" height (note that we have got the height in the world space and have to transform it into the screen space). This operation can be highly optimized due to the nature of the grid: z value is constant per line and camera-to-point normalized vector (used for calculating reflection vector) can be precalculated.
7. Then we just render selected (by the water borders in the screen space) block grid lines.

This method has several disadvantages. The first and the most important is that here we can not fight against "rabbit inside boa" effect. We just increase number of cells to compensate it and use special methods to hide jerking at distance. These methods are very unnatural: amplitude of height and normal oscillation is decreased (the farther, the more). Also mipmaps are added for the "matrix" texture. Other disadvantages are: prohibition of the camera tilt, restriction on the camera attack angle and performance dependence on the camera attack angle (the greater angle - the more triangles we see - less fps we get). The main advantages are high speed and (it sounds strange but it is true) simplicity of coding in comparison with the previous method of building 3D mesh. Textures are small enough (lock time is small), world-to-screen transformations are done by graphics processor (works in parallel), some calculations may be precomputed and the whole mesh is one strip only.

6. RESULTS AND PERFORMANCE

6.1. Water surface building

The most performance expensive part of water patch calculation algorithm is FFT due to its computational complexity O(n^2log$n$). We found that the largest but acceptable height map has the size of 64x64. On our target platform (PIII-733) FFT for 64 x 64 matrix takes 0.89[4] ms (4.13 ms for 128 x 128 matrix – unacceptable for real-time). Performance measurements for the entire method are given in the Table 2.

Name	Time, ms
Matrix H filling	0.577
FFT	0.886
Normals calculation	0.332
Shearing (chopping) and normals recalculation	0.838
Overall	2.633

Table 2. Performance measurements of water surface building algorithm.

6.2. "Optical effects"

Here we will discuss the external part of "optical effects" imitation – the part performed out of water surface rendering.

"Reflection and refraction" model is very performance consuming due to two additional scene renders. So instead of $N_{above}+N_{intersect}$ triangles in scene (N_{above} – number of triangles in objects that are completely above water, $N_{intersect}$ and N_{below} are defined in the same way) we have $2N_{above}+3N_{intersect}+ N_{below}$ triangles. For average scene that means 2.5-3 times increasing number of triangles to render.

To be more concrete let us discuss a real scene. Scene image shown on Fig. 11 is a screenshot of the game prototype. Visible

[4] We used Intel® Math Kernel Library™ for FFT because it is optimized for PIII SIMD instructions.

landscape at this point has about 15000 triangles. Visible water consists of 7100 triangles. On our platform this scene is viewed at 75 FPS (game physics is off).

“Sky reflection” (see an example on Fig. 12) model needs the bump texture update on each frame. The texture also should have mipmaps, because to avoid noise effect in the distance we use mipmap filtering. On our platform building and uploading to video hardware 64x64 texture with mipmaps till 8x8 costs 0.96 ms.

6.3. Grids

Real 3D grid used in water skiing game prototype (maximum camera height 5 meters) has 7100 visible triangles, 4 LOD levels, smallest step 0.2 meters and grid side 90 meters. This grid was used to get results shown on Fig. 11 and Fig. 12. GPU render cost for this configuration in case of bumped water is about 1.59 ms. All per vertex calculations are in vertex shader so CPU cost is rather small: about 1 ms for grid updating and blocks visibility determination.

2D grid is best used on PlayStation 2. We found that 8000 triangles is enough. Overall performance of this method on PS2 is greater that in 3D grid case (with nearly the same quality) because all texture transfers are performed in parallel and rendering of 2D grid is faster.

Results for 2D and 3D grids on PS2 can be compared on Fig. 9.

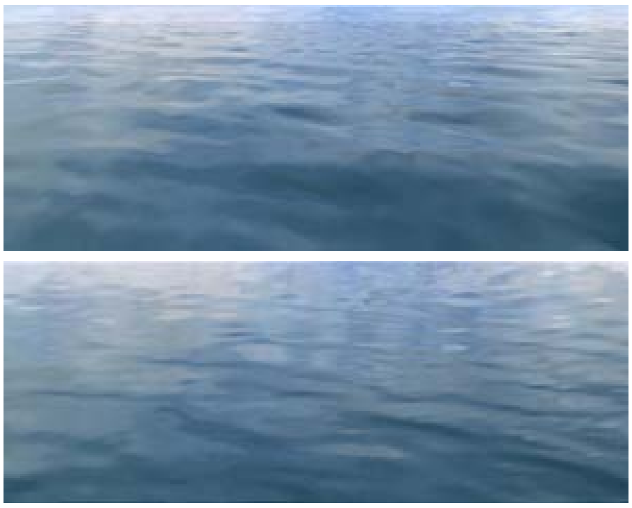

Figure 9. Water surface rendering results using 3D grid (upper) and 2D grid (lower). Note that 2D grid smoothes water at the distance.

7. CONCLUSION AND FUTURE WORK

We have presented the methods for all 3 steps of real-time water surface simulation. These methods can be used as a meccano to build simulation for the different purposes. Fig. 10 illustrates possible combinations of different methods.

Although this work presents ready-to-use simulation, it covers only a small number of possible ways of water interaction with the rest world. There are several directions of our further research: waves interacting with the shore, breaking waves, foam and spray, color of water depending on angle of view, light transporting model and caustics.

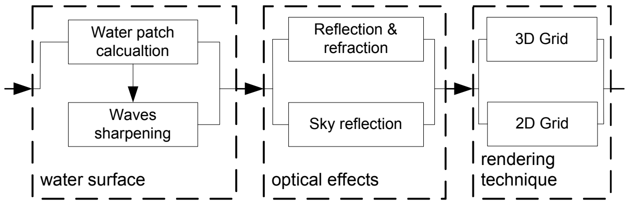

Figure 10. Chart of possible methods combination. E.g. you can walk such path: (water patch calculation) → (sky reflection) → (3D grid).

8. REFERENCES

[1] O’Brien, J.F., and Hodgins, J.K. “Dynamic Simulation of Splashing Fluids” Proceedings of the Computer Animation'95 (CA '95)

[2] Jim X. Chen. Physically-based modelling and real-time simulation of fluids. PhD in the Department of Computer Science of University of Central Florida, 1995.

[3] Duchaineau, M., Wolinsky, M., Sigetti, D.E, Miller, M.C., Aldrich, C., Mineev-Wienstein, M.B. ROAMing terrain: Real-Time Optimally Adapting Meshes. Proceedings of the 8th IEEE Visualization '97 Conference.

[4] Foster, N. and Metaxas, D. Realistic Animation of Liquids. Proceedings GI '96, pp. 204-212

[5] Fournier, A., and Reeves, W.T. “A Simple Model of Ocean Waves”, Proceedings of SIGGRAPH '86, Volume 20, Number 4, pp. 75-84.

[6] Jensen, L. Deep-Water Animation and Rendering. Gamasutra, September 26, 2001. http://www.gamasutra.com/gdce/jensen/jensen_01.htm

[7] Kass, M., and Miller, G. Rapid, stable fluid dynamics for computer graphics. Proceedings of SIGGRAPH '90, in Computer Graphics, Volume 24. Number 3, pp.49-57.

[8] Layton, A.T. Numerically Efficient and Stable Algorithm for Animating Water Waves. The Visual Computer, Vol. 18, No. 1, pp. 41-53, 2002.

[9] Premoze. S, and Ashikhmin, M. “Rendering Natural Waves”, Computer Graphics Forum, v. 20, no. 4 (2001), pp. 189-200

[10] J. J. Stoker. Water waves. The Mathematical Theory with Applications. Interscience Publishers Inc., New York. 1958

[11] Tessendorf, J. Simulating Ocean Water. SIGGRAPH 2001 Course notes. http://home1.gte.net/tssndrf/index.html.

[12] Turner, B. Real-Time Dynamic Level of Detail Terrain Rendering with ROAM. Gamasutra, April 03, 2000, http://www.gamasutra.com/features/20000403/turner_01.htm

[13] Watt, A. and Watt, M. Advanced Animation and Rendering Techniques. Theory and Practice.ACM Press, 1992.

[14] Ulrich, T. Continuous LOD Terrain Meshing Using Adaptive Quadtrees. Gamasutra, February 28, 2000, http://www.gamasutra.com/features/20000228/ulrich_01.htm

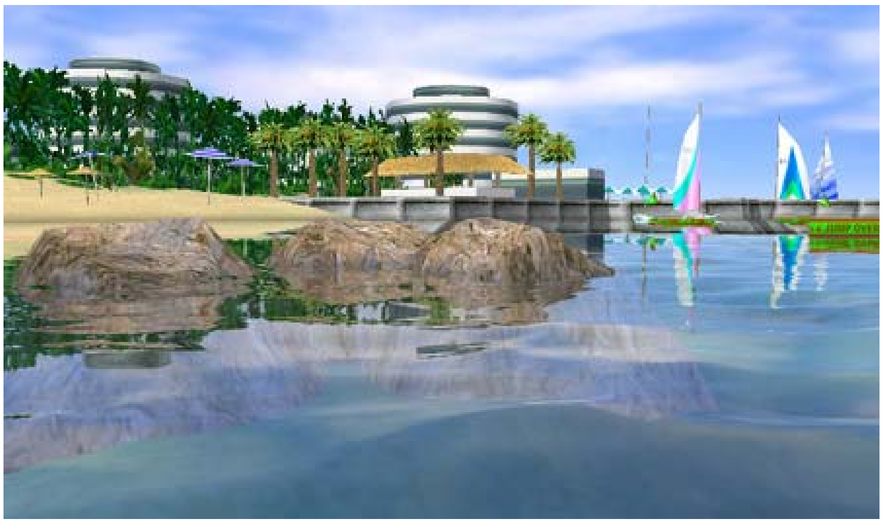

Figure 11. "Reflection and refraction" method result (resolution of reflection and refraction textures – 512x256)

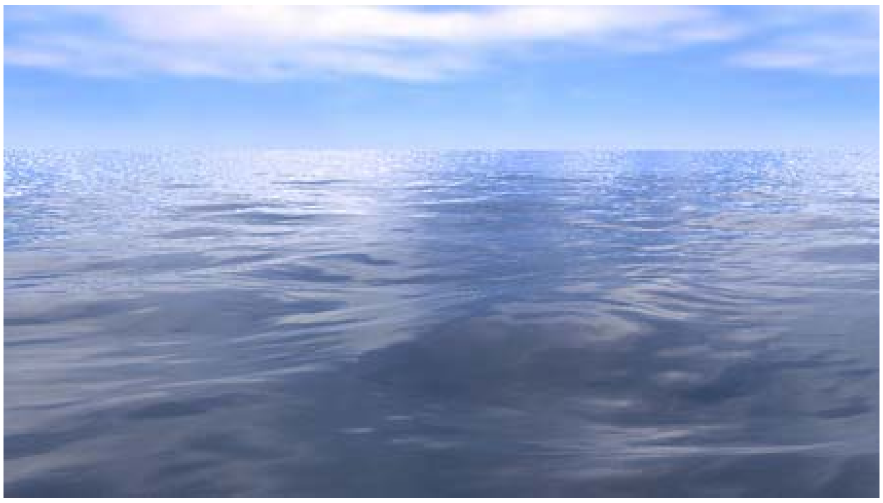

Figure 12. "Sky reflection" with environment reflection bump

About the author

Vladimir Belyaev is a Ph.D. student at Applied Math. Department,
St. Petersburg State Polytechnical University, St. Petersburg, Russia
His contact email is vladimir@d-inter.ru.