Japanese-Style Toon-Based Water Shader

Oscar Dadfar
School of Computer Science
Carnegie Mellon University
odadfar@andrew.cmu.edu

Joel Welling
Supercomputing Department
Carnegie Mellon University
welling@psc.edu

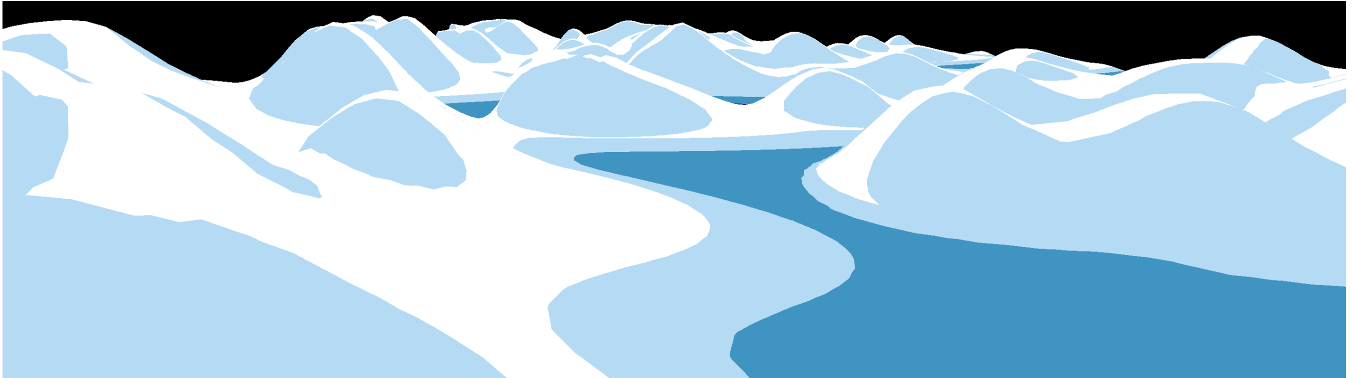

Figure 1. Representation of toon-based water shader inspired by the Japanese water artwork The Great Wave off Kanagawa by artist Katsushika Hokusai. Wave components are independently calculated and superimposed onto the mesh in the vertex shader. The position, normal, and heighmaps attributes are responsible for the shading of each fragment.

Abstract

Modeling water in computer graphics can be a computationally expensive process in order to achieve the realistic motions of water. Water shaders attempt to relieve this expense by running all vertex and fragment updates in parallel on the GPU specialized for simple linear operations. This paper looks at a Non-Photo-Realistic (NPR) representation of water inspired by the famous Japanese water-art painting The Great Wave off Kanagawa by Katsushika Hokusai and attempts to produce a modular toon-based water shader based on the work. Such a system requires vertex position updates based on interpolated spline calculations on the CPU, laplacian height diffusion in the vertex shader, as well as thresholding on position, curvature, and heightmap information in the fragment shader.

1 Introduction

Simulating water effects in computer graphics is a dynamic simulation that has to be able to separate, combine, and take up any shape. Physically-accurate water effects requires knowledge of fluid dynamics, and can be modeled with geometries such as meshes, nurbs, and more-expensively, particle systems.

In recent years, the rise of toon-shading-based graphics inspired many researchers to represent many common BSDF (Bidirectional Scattering Distribution Functions) such a skin, cloth, and even water, in an artistic matter. Toon Shading became a way for artists to create stylized representations of 3D scenes that would mimic their 2D counterpart. Rather than rendering skin as a smooth gradient of colors from light to shadows, toon shaders would instead segment colors into bright lights and hard shadows. These results replicated older 2D animations back when artists filled in many scenes with hard colors rather than soft lighting because gradients were difficult to paint.

1.1 Objective

With toon-based water shading, artists attempt to simulate the overall deformations and color scheme of water, while injecting a bit of their artistic ideas into the shader to make certain colors brighter or other motions more exaggerated. Many of these artistic ideas are inspired by older animation styles from Disney and Studio Ghibli movies, both of which have been praised for their artistic direction. CGI artists use these films as templates to create robust water shaders that can be used in any 3D aspect such that when they are rendered, the 3D shader becomes a 2D animation that looks nearly identical to the original film.

One of the most inspirational works of water art not in film has been *The Great Wave off Kanagawa* by Japanese artist Katsushika Hokusai. Hokusai uses a toon-shading style similar to other films previously discussed where he deforms waves to exaggerate motion while using a color pallet of hard solids to differentiate between the various levels of the water depth and curvature. This paper attempts to analyze the different features of *The Great Wave off Kanagawa* in a

Figure 2. *The Great Wave off Kanagawa* by Japanese artist Katsushika Hokusai in the early 1830's.

toon-shading context in an effort to create a new toon-shader inspired by the iconic work of Japanese history.

2 Related Works

Researchers Yu. et al. pioneered the toon-water computer animation field with their first publication on real-time cartoon water animation in 2007 [3]. Their approach draws inspiration from many Disney 2D classics in order to create a 2D-based 3D representation of water. By segmenting water into different components, they were able to individually capture the physical properties of water jets, stream riffs, and droplet.

As animation slowly started moving towards the GPU, Park and Choi were able to write a GPU-executable water shader that could account for reflections and refractions based on the viewing incident angle [4]. Their model provides faster parallelized results from GPU computations, and also calculates the Gaussian curvature of the triangulated mesh in order to detect peaks and valleys in order to determine depth-based shading.

Researchers Chandra and Sivaswamy analyzed first through fourth order derivatives in order to assist with peak and valley detection [5]. Using 1D curvature analysis as the basis, they located specific properties the function's derivatives must have in order to locate where the derivative of curvature vanishes. Such mathematical tools are useful for identifying peaks and valleys in smoother mesh instances where the curvature change is very gradual.

Toon-based water shading also has applications in Non-Photo-Realistic (NPR) video processing. Zhang and his team used optical flow of pixels to determine water movements of low/high variance [6]. With this information, their approach draws brush strokes through low variance flows on an NPR background of a video sequence to make the motions look like Chinese paintings. Such an application is useful for

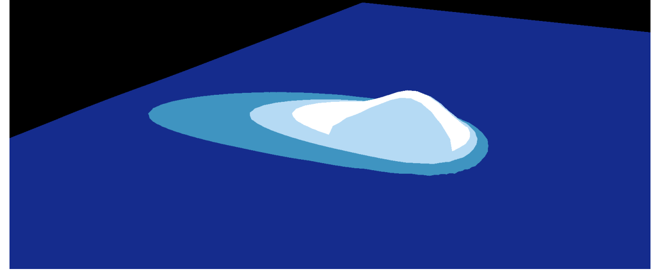

Figure 3. vertex displacement using interpolated spline bursts. At each timestep, the vertex shader uses laplacian diffusion to spread out the wave while also updating the height of nearby vertices.

conditional-based water-shaders that rely on an input image sequence to drive the NPR graphics.

3 Methods

The objective was to develop a modular web-based water shader algorithm that took advantage of the GPU for accelerated computation while also being easily portable over to other applications. The Three.JS JavaScript library was chosen because of its fast compilation and execution using WebGL and for easy viewing over multiple web browsers [2].

3.1 Single Wave Component

Using a triangulated mesh grid, the system injects displacements into the mesh at specific vertex locations. The Laplacian from vertex neighbors is calculated in the vertex shader to smooth out vertex positions at each time-step.

$$\triangle h(x,y) = 2(\frac{dh}{dx} + \frac{dh}{dy}) - 4\frac{dh}{dt}$$

The Laplacian $\triangle h(x,y)$ is computed using the two spatial derivatives and the temporal derivative. For our case of using a discrete mesh, we use finite differentiation to capture the spatial component of the Laplacian below.

$$L_s^t(x,y) = \frac{1}{4}(h^t_{[x+1,y]} + h^t_{[x,y+1]} + h^t_{[x-1,y]} + h^t_{[x,y-1]})$$

In the discrete spatial Laplacian equation above, $h^t_{[x,y]}$ represents the height of the vertex at location $[x,y]$ at time t. The spatial Laplacian can then be used in the 2D Scalar Wave equation to update the height diffusion after each timestep [1].

$$h^{t+1}_{[x,y]} = c_1 * (2L_s^t(x,y) - h^{t-1}) + c_2 * h^t_{[x,y]}$$

The constants c_1 and c_2 are used to regulate sampling between the current and previous height timesteps. These values should be bounded by $c_1 + c_2 \leq 1.0$ to prevent the height

field from diverging. There is an evident tradeoff in sampling between these two values, as too large a sampling from the previous height timesteps (c_1) causes the waves to propagate out of control and diverge, while too large a sampling from the current height timesteps (c_2) makes height diffusing too slow. The implementation presented in this paper uses $c_1 = 0.3$ and $c_2 = 0.65$. These values were intentionally configured to sum to less than 1.0 so that the wave components can diffuse down to a height of 0 over time.

The sudden rise in height in localized regions of the mesh that create each wave component is a result of calculating a burst location (wave center) for each component. Spline curves were used to interpolate burst locations in the mesh. Each spline is defined by 4 control points, and a duration parameter d defining how long it takes to traverse the path. At each time, the burst interpolated position is calculated as a function of its normalized time $\hat{t}$.

$$\hat{t} = \frac{t \pmod d}{d}$$

This produces a burst target (x_t, y_t) that is then passed into the vertex shader. The shader uses Euclidian distance from each vertex point (u, v) to the burst target before calculating the cosine of the distance to create a 3D cosine-wave distribution over the mesh.

$$h^t(\vec{u}) = cos(clamp(||\vec{i} * [\vec{u} - \vec{x_t}]||_2, 0, \pi) + 1.0)$$

In the above equation, $\vec{u}$ is the vertex point, $\vec{x_t}$ is the burst location, and $\vec{i}$ is the influence vector that weights each component differently. As an example, for longer waves across the x-axis, $\vec{i}$ can have a larger x-component relative to its y-component. The L2 norm of the resulting vector is taken and clamped between 0 and π before generating a 3D cosine wave component.

The fragment shader thresholds over the height of each fragment in order to determine one of 4 colors to use (white foam, light blue, mid blue, dark blue). In the provided code, these heights are set as [5.0, 3.0, 1.0, 0.0], though these are subject to change for different average wave heights.

To provide a more robust means of foam formation on each wave component, the normal at each component is considered in the fragment shading process. For regions with large normals components in the z-direction (flat regions) above a certain height threshold, these regions are where the derivative of the wave change sign, indicating high curvature. Areas with large normal z-components are then also shaded a white foam color.

3.2 System of Wave Components

Combining multiple wave components in order to produce a system of waves requires duplicating the number of streams responsible for producing offset bursts in the mesh. For

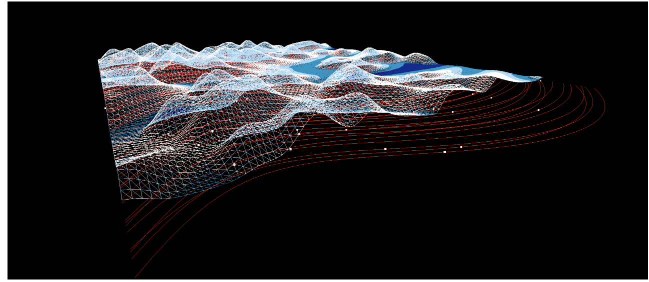

Figure 4. compositing multiple singular waves into one scene, superimposing overlapping wave. Multiple wave locations are calculated from multiple spline locations before being passed into the GPU.

greater modularity, the user is able to define a guide spline that can be used in computing a system of N splines. For each spline, some gaussian noise is added to each parameter to disrupt the uniform spread of splines and encourage overlap of wave components in the overall system. At each time t, N burst locations are computed and sent to the vertex shader to process.

For each vertex, the cosine of its Euclidean distance is computed and accumulated against all burst locations. This allows burst locations in the same region to superimpose as with real water.

$$h^t(\vec{u}) = \sum_{n=1}^{N} cos(clamp(p_n * ||\vec{i} * [\vec{u} - \vec{x_{tn}}]||_2, 0, \pi) + 1.0)$$

The p_n coefficient defines the peak height of each wave. Higher waves will have a larger height and radius over the

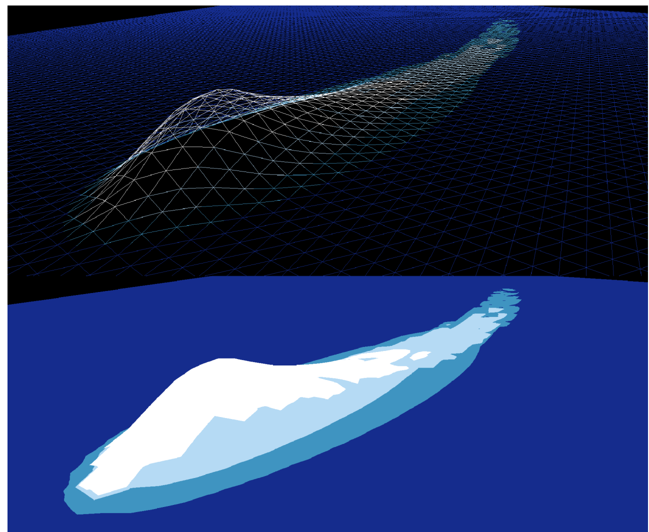

Figure 5. wave composed from thresholding on the z-component of the heightmap responsible for holding temporal information on wave component height properties.

mesh. Each stream is randomly assigned a peak value during declaration, and the GPU saves the list of peak values associated with each burst location when calculating the updated height values of each vertex.

3.3 Rift Water Components

To achieve the trailing effect of the water component, the system needs to record previous height information for each fragment. The current heightmap in the vertex shader is configured as a 3-vector, but only uses the first two components to store the previous and current height used in calculating Laplacian diffusion. The third value in the vector can be used to store temporal information indicating how long ago a burst location passed through the current fragment.

$$min(h^t(\vec{u})) = min_{n \in N}(||\vec{u} - \vec{x_{tn}}||_2)$$

Where $min(h^t(\vec{u}))$ stores the min distance at each vertex position to the nearest burst location. On each update in the vertex shader, if the vertex coordinate is within some threshold distance d of any burst location, then it is given a rift value $\bar{r}$ inversely proportional to its distance from the nearest burst location that is stored in the z-component of the heightmap. This ensures that more depletion information is stored near the peak of wave components.

$$r_t = r_{t-1} + k * (d - min(h^t(\vec{u}))) \text{ if } (min(h^t(\vec{u})) < d)$$

Where r_t is defined in terms of r_{t-1} so that rift values can accumulate over time. To prevent rift values from going to infinity, at each timestep, the vertex shader divides any current r_t by some value $(1.0 + \epsilon)$ so that over time, r_t will deplete to 0. The result is a rift of white foam that trails behind the water component as it moves. This same effect can also be applied to other colors by using different thresholds on r_t

Rift values are also modular in that they accumulate in the presence of multiple nearby bursts. This is supported by the idea that two waves superimposed will produce a larger wave, which as a direct result will have a larger trailing rift. As such, the rift values in the vertex shader are accumulated if there are multiple non-zero values computed in order to produce a longer-lasting rift at that instance.

4 Results

The following results in Figures 6-8 were run in Three.JS in the Chrome browser. Performance hit 60 FPS using an 2.3GHz Intel i5 processor and 1536 MB Intel Iris Plus Graphics for 100 wave components.

5 Discussion

Given extra time, the shader could be improved to more-closely parallel different effects in the original Great Wave off Kanagawa painting while also increasing performance across different devices.

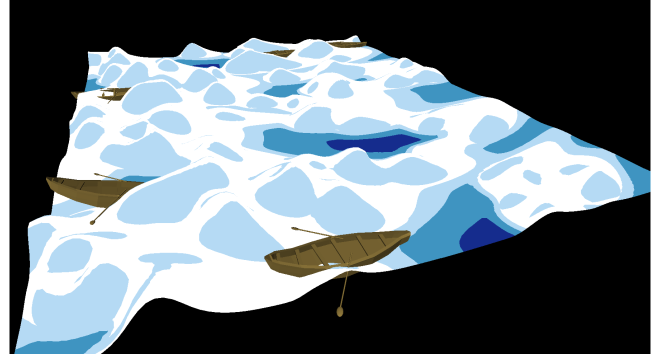

Figure 6. smooth wave composition with integrated boat dynamics.

5.1 Shader Capabilities

Compared to the original Great Wave off Kanagawa painting, the demonstrated model fails to capture any sense of line drifts present to the sides of the waves. Some proposed ideas to integrate this feature included constructing a texture map of lines that could be passed in to the fragment shader to be sampled at a skewed angle relative to the normal and height of the fragment (fragments with lower heights would sample from an even more skewed direction). This approach was not robust since turns in the wave would segment the skewed sampling, causing non-continuous line segments shaded onto the wave.

Solving the texture problem also led to another problem that the line thicknesses were constant along the wave, while in the original painting, the line segments decreased in thickness until they vanished. A proposed method for resolving this was to use the height of each fragment to determine how close together samples in the texture would be drawn. Larger heights would correspond to closer samples being drawn, leading to an increase in line thickness, but also an increase in average line distance, which then becomes a new problem. An attempted resolve to this was to have multiple

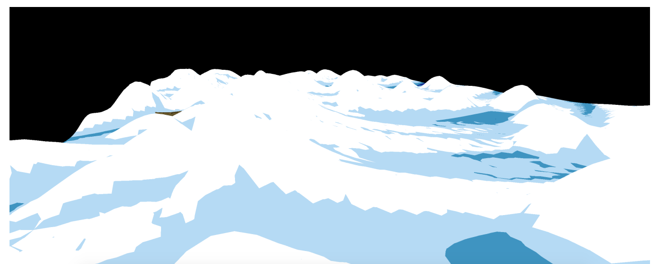

Figure 7. side view of multiple rift wave components.

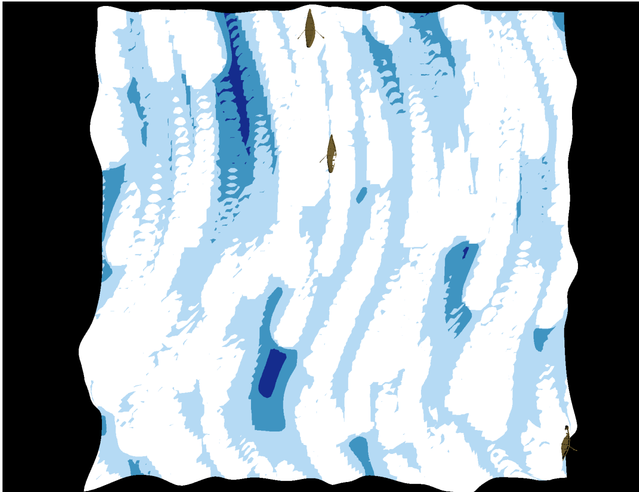

Figure 8. top view of rift water group.

textures with different line thicknesses, but then interpolating continuously between different thicknesses would prove to be very difficult.

The biggest issue with sampling drifts from a texture is that the drifts do not move. No information is carried from the shaders about the velocities of each burst, so the texture sampler has no way of adjusting for speedups or direction changes in wave components. A future area of research may be in how to pass this information down, and how to configure the texture sampler to use this information on creating consistent samples that track to the movements of each wave component.

5.2 Performance

While significant performance was achieved using the specs listed above, other devices such as a 2.6GHz Intel i7 processor with Intel Skylake GT2 graphics achieve an average performance of 58 fps, dipping below the 60 fps threshold. Some of the more expensive computations occur on both the CPU and GPU side when working with a large number of wave components. For each vertex in the vertex shader, the distance between it and every other burst is computed in linear time relative to the number of wave components attempting to be rasterized. The reference images were produced using 100 wave components, yet the performance can dip drastically when increasing the number of components by orders of magnitude for very large scenes. An alternative speedup would be to use pre-processing on the CPU to construct a bonding-box hierarchy with amortized logarithmic construction and search relative to the number of wave components.

Another expensive computation lies in the CPU when retrieving the burst locations to pass into the shaders. Computing all location is linear in time, but can be sped up using multithreading since each burst location can be retrieved independently.

6 Conclusion

We presented a novel Japanese-inspired water shading algorithm that is dynamic relative to itself and other objects on the water. This shader has potential applications in NPR-style video game, animations, and anything else with a toon-shading art direction. While there were some limitations in implementing certain details from the original painting, the overall water shader is computationally simple enough to display in realtime, and able to be ported into any graphics or animation software.

References

[1] 2008. 2D Water. (2008).

[2] 2019. Three-JS GPU Water. (2019).

[3] Haiying Chen-Cheng Yao Jinhui Yu, Xinan Jiang. 2007. Real-Time Cartoon Water Animation. *Computer Animation Virtual Worlds* (2007), 405–414.

[4] Byungkuk Choi-Junyong Noh Mi You, Jinho Park. 2009. Cartoon Animation Style Rendering of Water. *ISVC* (2009), 67–78.

[5] Jayanthi Sivaswamy Siva Chandra. 2006. An Analysis of Curvature Based Ridge and Valley Detection. *IEEE* (2006), 737–740.

[6] Yi-Fei Zhang Shi-Min Hu Ralph R. Martin Song-Hai Zhang, Tao Chen. [n. d.]. Video-Based Running Water Animation in Chinese Painting Style. ([n. d.]).