Poster Proceedings of Pacific Graphics (2008)
T. Igarashi, N. Max, and F. Sillion (Editors)

A Particle-Guided Method for Animating Splashing Stream Water in Real Time

Su-Ian Eugene Lei[1] and Chun-Fa Chang[2]

[1]National Tsing-Hua University, Taiwan
[2]National Taiwan Normal University

Abstract

Realistic water representation in computer graphics involves a number of simulation and rendering techniques. In the case of real-time rendering of water, such calculations need to be simplified in order to achieve an interactive frame-rate. In this work, we present a framework for rendering stream water, with focus on the effects of turbulent flow, splashing over irregular terrain and its interaction with dynamic rigid objects. Unlike similar particle-based methods where each particle in the system is visualized on screen, the particle system in our implementation serves as guidance to the final visual result. Its purpose is to detect the occurrences of splashing and turbulence. Our simulation is fast enough to achieve interactive frame-rate, and produces realistic splashing effects commonly observed in water streams.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism

1. Introduction

The motion and behavior of flowing water is a well-known phenomenon. As such it is challenging to simulate and render the effect realistically since any unnatural behavior is easily discernable to the casual observer.

Many currently employed techniques of visualizing water effects in interactive applications are based on a near-static water surface. And to mimic the effect of local interaction (such as when a bullet penetrates the water surface), they use a simple particle system or an animated texture to represent the splashing of water. These visual effects are detached from the general motion of water body, which means that a splash from a rock thrown into a static pond or a rapidly flowing stream will look visually identical, unless the artist manually specified otherwise. Also water splashes caused by a jagged riverbed and rapid water flow will be difficult to visualize in these systems. While visualizing such effects have attempted before, they are usually animated layers of textures drawn manually by the artist to mimic these effects, and does not physically reflects the feature of the terrain.

Our work strives to increase the realism of visualizing these effects in real-time by unifying them under one particle system. This particle system simulates the flowing of

Figure 1: *Splashing Stream Water. This example shows a teapot set on a rapidly flowing stream. Notice the splash near the handle and spout, and the wake downstream.*

the water stream, detects its interference with rigid objects, and generates appropriate splashing effects. Thus these effects will be visually influenced by the characteristics of the water flow (such as the speed of water current). An implementation using a simple proof-of-concept motion simulation system [CLCC07] was done previously. This work is an extension of that concept.

In this work we provide a hybrid approach that employs a GPU-based guidance particle system and grid-mesh water surface, to simulate a number of effects ranging from tranquil to rapidly flowing water. We focus our work on the interaction between water, terrain and dynamic rigid objects, and the effects of turbulent flow and splashes.

The particle system we are using is inspired by the work of Lutz Latta [Lat04]. We modified the system to allow subparticle generation and state transition. A set of primary particles are used to simulate the general water flow, while subparticles can be spawned from the primary particles to simulate the breakage of water volume rapidly interacting with rigid objects. Unlike its common usage, this particle system we use mainly serves as a reference point of the final visual result. Most implementation of such a particle system visualizes every particle inside the system, treating each individual particle as a basic building element of the entire effect. In our implementation however, the particle is employed as a marker of various events taking place inside the water stream, such as collision, foaming, splashing etc. This is achieved by state transitions (section 3.3) within each particle. Then these events are visualized using the markers as a reference.

Our visualization of splash effect is inspired by surface splatting and metaball rendering techniques. The organic shape of metaball is suitable for portraying fluidic effects such as viscidity of water drops. We also used light attenuation and vertex-texture based displacement mapping to simulate light scattering of deep-water volume, therefore increasing the overall realism.

In our system, the user is required to provide the emitters for the particles. A state machine is used to describe various state of a particle, for example if the particle is about to break into smaller sub-particles. The transition conditions depend on a set of variables including collision, flowing speed etc. These variables can be customized by the user to simulate, for example, different scale of water volume. Te rest of this paper is organized as follows: Section 2 reviews previous works; in section 3 we provide an overview of our algorithm, including how we use the particle system as a reference and the state transitions of particles; section 4 and 5 details the implementation of our algorithm; and in section 6 we provide the results of our work; followed by the conclusion.

2. Previous Work

Much effort was applied in order to capture the complexity of water flow convincingly. Most of the approaches involved certain levels of abstraction and simplification, since in the strictest sense, a truly accurate simulation model involves interactions between water molecules. This is especially true in the case of real-time rendering, where further simplification is required to reduce the number of calculations to achieve interactive frame-rate. In many cases developers focuses on one particular type of water behavior in their simulation, such as deep-water [LSJ01], shallow streams [KW06], waterfalls [RG07] etc.

Even some widely accepted simulation models, such as the computationally expensive Navier-Stokes Equation [FM96], simplifies the environment by assuming continuous incompressible homogeneous fluids, thus not applicable in very small scales or extreme conditions when the mixtures of discrete molecules, suspended particles and dissolved gases are needed to be taken into account. Smooth Particle Hydrodynamics (SPH) [MCG03] works by dividing fluid into a set of discrete particles, and the water surface is reconstructed from the density field accumulated by said particles. However SPH has a bottleneck in its neighbor search; since for each particle, we have to search all its neighbors and compute their interacting force.

The visualization of the simulated fluid, usually done by surface reconstruction, is also a non-trivial issue. Kipfer et al. [KW06] propose a real-time algorithm utilizing SPH and use a carpet-covering method to visualize the surface of a streaming water-body, with some loss of fine details. All in all, a simulation structure fully based upon SPH can usually only utilize a small amount of particles, thus a water-body limited in scale, if it were to achieve an interactive frame-rate.

To increase the level of realism without dramatically increasing the number of particles and calculation, a number of approaches use techniques similar to level-of-detail in polygonal geometry rendering. Iwasaki et al. [IODN06] uses off-line point-based rendering and divides primary particles into sub-particles to simulate the effects of water splash. Takahashi et al. [TFK*03] used a state machine to generate fine details such as splashes and foam. Some aspects of our work, most importantly the particle state transition and particle sub-division are inspired by their approach.

With the recent technical progress in graphics hardware, it is possible to generate and maintain a particle system with particles numbered in the millions. Lutz Latta [Lat04] proposed a GPU based massive particle system. Kipfer et al. [KSW04] used the graphics hardware to sort particles spatially, thus incorporating inter-particle collision into their system, creating a large-scale particle flow.

Various developers try to simulate the visual effect of interactions between water and rigid object. Marcelo et al.

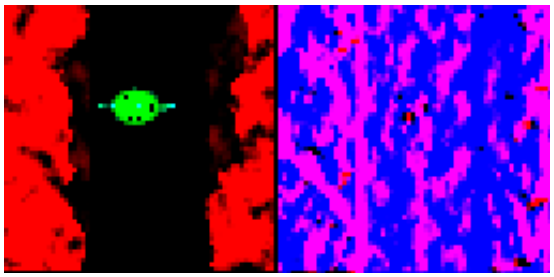

Figure 2: *(Left) Height map. The green shape is the teapot. This is the height field that we are using for our examples in Figure 1. (Right) Flow field derived from height field.*

[MFC06] presented a column-based and height field approach to simulate water flow over irregular terrains in GPU. However, their work did not incorporate the effects of turbulent flows. The recently released Crysis [Sou08], which uses the in-house developed CryENGINE2, utilizes a number of special effects to create realistic deep-water (ocean). The engine renders shoreline using procedural textures and vertex animated static mesh, to simulate the effect of foaming. Splashing effects, such as bullet penetrating the water surface, are represented using traditional animated billboard texture and simple particle system. Splashing do not interfere the basic water structure, which is a screen-space tessellated grid mesh. Using their provided development tool, a group of users [Hav08] has successfully utilized its particle system to simulate a particle-based liquid flow, similar to Uberflow, with streaming and pooling effects. However the particle system does not possess advanced visual effects beyond alpha-blending, thus lacking the optical effects one would have expected in a water simulation.

With the recent release of DirectX 10, Nvidia released a number of demos to present a wide range of new effects utilizing its GeForce 8 hardware. In particular, the demo ąğCascadesąĺ [RG07] implements a waterfall scene using a GPU based particle system. It uses the Geometry Shaderąęs variable output to control the spawning and removal of particles. The water droplets and stream are basically billboards, aligned to camera and geometry normal respectively. The water stream (sliding water, as called in the demo) also renders itself into a 3D texture, to combine with the surface texture creating a wet rock effect. A number of visual effects such as mist, sine wave wobbling, and specularity are implemented to improve realism. Our specularity rendering of water droplets and blending between stages is similar to their approach.

3. Overview

Our work primarily focuses on visualizing splash and turbulent flow in streaming water, which were not effectively realized in real-time by previous implementations. We use a GPU-based particle system to provide a reference of the visualization. We are using a set of hidden primary particles to track the river flow, and use a state machine to decide whether splashes and turbulent flow would occur. If so, these hidden primary particles will spawn sub-particles to visualize such effects.

To visualize streaming water such as a river, first we need a representation of the terrain, which serves as the riverbed and the immediate surroundings. Since the water will eventually flow through this terrain, we use a 2D height field to represent it because we can then employ simple image processing algorithms with GPU to acquire certain data, such as the flow vectors and velocities.

Although we use particle sub-division to more efficiently simulate the splashing behavior, in some cases these sub-particles are still too large to realistically represent micro particles at extreme breakage. A second or further level of sub-division would be inefficient to implement. Therefore we employ the technique of particle animation, which is a set of textures that visually depicts the sequence of particle breakage.

Thus, in our implementation, three sets of data are required to be pre-computed procedurally, or specified by user input.

- The terrain data represented by a height field.
- A set of animation sequence, the transition condition for navigating this sequence, and the movement behavior of sub-particles for each particle state.
- The position of the emitters.

We elaborate them in the following sections.

3.1. Height Field and Flow Field

Our terrain data is represented by a 2D height field. We then use the height field to approximate the 2D flow velocity. This flow field controls the general direction of the particles. The flow field is created by first calculating the gradient of the height field. Then we apply a Gaussian filter on the gradient map. The reason behind the Gaussian filter is that we would like to extend the range of the flow around the terrain. Note that we need to invert the gradient field since the flow direction is generally the inverse of the gradient. Then we add a rough velocity into the flow field, which represent the average river speed and direction. The result is our approximated flow field:

$$G = gauss\_filter(-\nabla(height\_field))$$
$$flow\_velocity = G \cdot flow\_scale + rough\_velocity$$

When we include dynamic rigid objects in our calculation, we only consider the case that the water surface is intersected between the front and back face of dynamic objects; which means that the object is neither above nor under water. The flow field is then approximated from the height field of the

front face of the dynamic object. We use a weighted sum to combine the flow fields of terrain and dynamic objects:

$$flowfield_{final} = w_1 \cdot flowfield_{objects} + w_2 \cdot flowfield_{terrain}$$

In addition, we define a threshold ϕ, which categorize the current field position as a *rapid_flow* or *slow_flow*. This will be used as a variable in our state machine which will be described in later sections.

$$\begin{aligned} &if(\ \ |flow\_velocity| \ > \phi) \\ &\quad flow\_fieldstate \ = rapid\_flow \\ &else \\ &\quad flow\_fieldstate \ = slow\_flow \end{aligned}$$

3.2. Particle Motion

Basically, we adopt the Euler integration scheme to update particles from time t to $(t+dt)$:

$$\begin{aligned} v(t+dt) &= v(t) \cdot scale_{particle} + v_{flow}(p(t)) \cdot scale_{flow} + \frac{F}{m}dt \\ p(t+dt) &= p(t) + v(t+dt) \cdot dt \end{aligned}$$

where v is the velocity, p is the position, vflow is the 2D flow field texture computed; $scale_{particle}$ and $scale_{flow}$ are two scalars which can be adjusted by users manually. F represents the external force such as the gravity. In order to maintain the simulation efficiency, we do not model the inter-particle collision here. Collision with obstacles and the dampening of water velocity are important effects of streaming water. A water droplet has its velocity dampened when it breaks off from the main water flow, and slowed down by friction with air. In this work we use an intuitive method to simulate these phenomena. If the particle collides with the terrain, the velocity of it will be reflected against the normal of the terrain or dynamic object. And if the particle penetrates a user defined highest depth (a pseudo water surface), we will damp the particle velocity. To perform the collision detection, we need the predicted position of the particle:

$$p_{predict} = p_0 + V_0 \cdot dt$$

where p_0, v_0 are the position and the velocity of the particle in the current frame. Then we have to test if the predicted position of the particle is inside the terrain, the dynamics objects or the water body. Assume that the y direction is the up direction of the world space:

$if\ (p_{predict}).y < terrain.height$
  collide with the terrain
  reflect the velocity
$elseif\ object.bottom\_height < (p_{predict}).y < object.top\_height$
  collide with the dynamic object
  reflect the velocity
$elseif\ terrain.height < (p_{predict}).y < water.height$
  collide with water
  damp the velocity :

$$v.y = v.y \cdot damp\_factor$$

All the values that represent the terrain height, the front and back face of dynamic objects, and the water surface can be accessed directly from the height field. When a collision occurs, the velocity and the position after collision can be computed as follows:

$$\begin{aligned} v_{reflect} &= v_t + (-v_n) = (v - v_n) + (-v_n) \\ &= v - 2v_n \end{aligned}$$

where n is the normal of the collision position and can be computed from the height field texture of the terrain and the dynamic objects.

3.3. State Transition

From the last section, we can determine the collision condition of each particle. A collision condition is asserted when the particle collides with the terrain, with a dynamic object, or penetrates the pseudo water surface. In addition, we have the flow field condition (rapid or slow). We use these two variables to decide how the particleąęs state changes.

The state transition serves two roles. It navigates the predefined animation sequence, which visually describe the lifecycle of a particle. In addition is plays an important role for generating sub-particles. The transition condition can be customized by the user to incorporate a larger range of animation choice, or suppress/encourage certain effects such as foaming.

4. Implemention

Our simulation in a single time step can be divided into the following phases, the underlying concepts of which are described in the last section.

- Render the terrain and the rigid bodies into the height field
- 2D flow field approximation
- Emit particles and sub-particles
- Particle motion simulation
- Collision and dampening
- State transition
- Transfer the position texture to vertices

In this section, we briefly explain several implementation details for our GPU based particle simulation.

4.1. References

In our implementation, we need to allocate the attribute textures for the *height field*, *flow field*, *position texture*, *velocity*, *current state*, and *state transition*.

If the user provides a geometry of the riverbed terrain instead of a 2D height field texture, we can render the height field texture by setting a camera perpendicular to the terrain. We also use this camera to render the dynamic object, which

is clipped against the pseudo water surface (so that only the parts that intersects with the water surface is accounted for). In our implementation, we store the height field of the terrain in the R channel of a height texture, and use the G channel to store the dynamic object. The creation of the flow field is described in section 3.1, and we store it in a separate texture.

The positions, velocities and states of all active particles are stored in 2D floating-point textures. Each pixel of these textures records the attributes of one particle. Since we cannot read and write the attribute textures in a single cycle under current rendering pipeline, we need to use double-buffering.

The state texture stores the state, age, lifespan and size of each particle in the RGBA channels respectively. The state is closely related to the transition texture. We perform a texture lookup in the transition texture to determine if a state transition occurs. The age and lifespan will be used to decide if a particle is alive at the current iteration. The size of the particle determines its visual proportion.

The transition texture is essentially a descriptor of our finite state machine. In our implementation we use a 1D floating-point texture. And for the sake of simplicity, only one transition leads out from each state. Each entry records the transition condition, which consists of transition switch (if this state is enabled), collision condition, flow field state, and the transition target state. These values will be stored in the RGBA channels of the texture.

In our implementation, we need to read back the state texture from GPU to CPU when some primary particles are spawning sub-particles, under the NV_vertex_program3 extension in OpenGL (or Shader Model 3.0 in DirectX). We will describe the method of emitting sub-particles in the following subsection.

4.2. Particles and Sub-Particles Emission

Particles are allocated or removed as the simulation goes on. Unfortunately, since allocation problems are serial by nature, they cannot be handled efficiently on the GPU. We still need to utilize the CPU in order to allocate new particles in the simulation. A simple solution is to store available indices in a stack. Once the index of a particle is determined, the particle attribute will be updated into the attribute textures.

The death of a particle is decided by its birth time and lifespan. To collect the dead particles efficiently, we use a priority queue that is sorted by the particleąęs death time on the CPU side. If the current time is larger than the supposed time of death of a particle, its index will be freed and pushed to the indices stack.

Under the NV_vertex_program3 extension, it is difficult to divide particles into smaller sub-particles entirely using GPU. We would need to allocate sub-particles in CPU and pass their attributes to the GPU. To generate sub-particles, we first record the initial attributes of sub-particles for each particle state into a linked-list. The initial attributes can be precomputed or specified by users. By reading back the state texture of particles from GPU to CPU, we can determine which particles will spawn sub-particles. Then the sub-particles recorded in the linked-list will be emitted by the CPU. Ultimately, these sub-particles are recorded to the same attribute textures describing the primary particles.

In our implementation, we treat the splash emitter as a kind of particle. When a splash emitter collides with objects, the splash emitter will emit sub-particles, which we would call splash particles.

4.3. Transfer the Position Data to Vertices

In our implementation, we use vertex texture fetch (VTF) to transfer the position texture data into vertex data using graphics hardware. We can cache the vertex data in graphics hardware in advance, and then use VTF to translate these cached vertices to their respective positions. The particles can be rendered as point sprites, billboards, or reconstructed as a polygonal mesh by Marching cube etc. In our implementation we choose to render the particles as billboards, therefore we would cache four vertices per particle. Since the particle size is recorded in the state texture, we can determine the size of the billboard dynamically with VTF.

5. Rendering

In the visualization stage, we first construct the billboards of particles in GPU then render them with a number of post-processing techniques. In our hybrid approach, we use a grid mesh to represent the water surface. The particles and water surfaces are then combined into the final result of our rendering.

We choose to use billboards to represent our metaball effect, since it is computationally less costly than reconstructing the entire splash surface. While there are works that significantly reduces the cost of full polygonal iso-surface reconstruction [RB08], our implementation needs to visualize a significantly higher amount of particles. A fully polygonal approach is also unnecessary since we focus our work on visualizing splashes, where individual particles often occupies no more than a few pixels. To use polygons to reconstruct a discernable spherical surface would be impractical.

5.1. Billboard Construction

There are two types of billboards in our implementation. One of them aligns towards the camera so they always face the user, the other aligned parallelly to the water flow. Since the splashing droplets will eventually fall back into the water, we first render the splash texture on a camera-aligned billboard. Then for each iteration, we turn the billboard gradually until it is parallel to the water surface. The billboards

parallel to the water surface represents the turbulent wake of the water stream.

5.2. Splash Rendering

There are several optical effects that are necessary when we create the look of water, such as reflection, refraction and Fresnel. Both reflection and Fresnel are related to the incident viewing angle and normal vector of the water surface, and refraction is related to the depth of the water body. Therefore our implementation will concentrate on acquiring the normal vector and depth information from our particle data. In our method, all particles needed to be traversed and rendered once into a rendering target. Then we use a set of pixel shaders operating upon this texture to create the effect we need.

Our first step is to render all particles using view-aligned billboards. The billboard texture is a spherical gradient texture representing a water drop. The billboards are rendered into different layers based on their Z value. For the ease of our experiments, we use the RGB channel of a single texture as the three separate layers. The particles do not need to be sorted since we are simply using alpha blending to accumulate the depth. These layers represent the depth of the water body. Then we use a pixel shader to perform Gaussian blur on the layers. While the billboards are gradient textures, we need to filter the resulting image to eliminate the blending artifact in order to create the metaball effect.

On the next step we create two maps: the normal map and a metaball ąğmaskąĺ. The normal map is created by applying linear gradient function on the filtered depth layers, with the sum of all layers representing the total depth on a pixel.

The metaball mask is created by applying a threshold function on the filtered depth layers. This threshold represents the boundary of the water body. We use a threshold of 0.5 in our experiments. The metaball mask and the filtered depth layers represent the surface shape of the water splash. The filtering of the depth layers also eliminated traditional clipping artifacts present in billboard rendering. Since the metaball mask is obtained from the filtered layers, such artifacts are no longer visible in the final result.

Next, we use the normal map, filtered depth layers and metaball mask to create the necessary optical effects. In our experiments, we implemented screen-based refraction effect and specular lighting. The screen-based refraction is not physically accurate, since it is the rendered background being distorted using the normal vector. However it serves its purpose well in our application. We use the Blinn-Phong model to compute the specular highlight of splash surfaces.

To visualize the volume color of the splash, such as whitening due to microscopic bubbles being formed within, and mist created by tiny droplets; we simply use the filtered depth map as the volume color. The composition of specular highlight and volume color would then be the final result of the visualization of water splashes and turbulence.

5.3. River Surface Rendering

As we have stated that it is impractical and inefficient to render the entire water body using particles, we use a more traditional approach to render the water surface using a grid mesh.

The grid mesh is distorted using a normal map, which represents a generalized rippling water surface. We can animate this normal map using procedural wave functions [Kry05], or simply shifting the texture coordinates continuously to create an illusion of flowing water. With the normal map, we can easily create optical effects such as refraction, reflection and Fresnel.

When the light transmits within translucent volume, the energy of the light will be gradually absorbed by the participating media. According to Beerąęs Law, the approximation of the light attenuation in homogenous liquid is:

$$T = 10^{c'l}$$

Where c is the multiplication of the concentration of the liquid and its absorption constant, and l is the length of the light path. We use the water depth as l in our implementation. When the water flow collides with obstacles, a splash is generated and bubbles will form underwater. We illustrate this effect by first blurring the filtered depth map, and then use a noise texture to distort it. We render this texture masked by the water surface grid mesh, to create the illusion that this is an underwater phenomenon.

Since the surface grid mesh is flat and the normal map cannot represent a very wide range of variation of the water surface, the particles with certain states are used to displace the grid mesh vertices. We render these particles as point sprites into a displacement texture with a camera orthogonal to the terrain. Then using vertex shader, the y component of the grid mesh vertex will be displaced by VTF.

The final visualization is the composition of resulting images from 5.2 and 5.3. First, we render the z-value of the terrain into the z-buffer, and then render the splashes into a screen texture with z-test on. Finally we use alpha blending to blend the splashes and other objects in screen space.

6. Results and Conclusion

All the experiments shown here ran on an Intel Pentium D at 2.8GHz processor and 1GB of memory, and an NVIDIA GeForce 7900GS graphics card with 256MB of memory. OpenGL and NVIDIA Cg shading language were used for all graphics operations. The attribute textures for recording the particles had the size of 128ąÑ128, and the viewport resolution was set to 512ąÑ512 pixels. In our demo, the frame

rates can achieve 120 frames per second. We update particles and sub-particles in the same pixel shader, and use the position texture to translate the cached vertices to appropriate locations by VTF method. Nevertheless, the method that generates particles and sub-particles simultaneously may use up the texture storage very quickly, even if we only spawn sub-particles once. But this problem will be solved soon in the GeForce 8 series with the geometry shader power. On the other hand, the 2D metaball surface reconstruction is very efficient.

We used a number of testing sets in our experiments, to demonstrate how the splash and the turbulent wake of the water stream are influenced by the terrain features of the riverbed, and their interaction with dynamic rigid objects. Our system shows that realistic visualization of recognizable effects of a river stream is possible with minimum input from the artist.

References

[CLCC07] CHANG J.-W., LEI S. I. E., CHANG C.-F., CHENG Y.-J.: Real-time rendering of splashing stream water. In *IIH-MSP '07: Proceedings of the Third International Conference on International Information Hiding and Multimedia Signal Processing (IIH-MSP 2007)* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 337–340.

[FM96] FOSTER N., METAXAS D.: Realistic animation of liquids. *Graph. Models Image Process. 58*, 5 (1996), 471–483.

[Hav08] Realtime water physics by havoksage. *http://www.crymod.com/thread.php?threadid=14945* (2008).

[IODN06] IWASAKI K., ONO K., DOBASHI Y., NISHITA T.: Point-based rendering of water surfaces with splashes simulated by particle-based simulation. *CDROM of Proc. Nicograph International* (June 2006).

[Kry05] KRYACHKO Y.: *Using Vertex Texture Displacement for Realistic Water Rendering*. Addison-Wesley, 2005.

[KSW04] KIPFER P., SEGAL M., WESTERMANN R.: Uberflow: a gpu-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2004), ACM, pp. 115–122.

[KW06] KIPFER P., WESTERMANN R.: Realistic and interactive simulation of rivers. In *GI '06: Proceedings of Graphics Interface 2006* (Toronto, Ont., Canada, Canada, 2006), Canadian Information Processing Society, pp. 41–48.

[Lat04] LATTA L.: Building a million-particle system. *Article of Gamasutra* (July 2004).

[LSJ01] LASSE STAFF JENSEN R. G.: Deep-water animation and rendering. *Article of Gamasutra* (Sept. 2001).

[MCG03] MÜLLER M., CHARYPAR D., GROSS M.: Particle-based fluid simulation for interactive applications. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation* (Aire-la-Ville, Switzerland, Switzerland, 2003), Eurographics Association, pp. 154–159.

[MFC06] MAES M. M., FUJIMOTO T., CHIBA N.: Efficient animation of water flow on irregular terrains. In *GRAPHITE '06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia* (New York, NY, USA, 2006), ACM, pp. 107–115.

[RB08] ROSENBERG I. D., BIRDWELL K.: Real-time particle isosurface extraction. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2008), ACM, pp. 35–43.

[RG07] RYAN GEISS M. T.: Nvidia demo team secrets - cascades. *Presentation on Game Developers Conference 2007* (2007).

[Sou08] SOUSA T.: Crysis next gen effects. *Presentation on Game Developers Conference 2008* (2008).

[TFK*03] TAKAHASHI T., FUJII H., KUNIMATSU A., HIWADA K., SAITO T., TANAKA K., UEKI H.: Realistic animation of fluid with splash and foam. *Comput. Graph. Forum 22*, 3 (2003), 391–400.

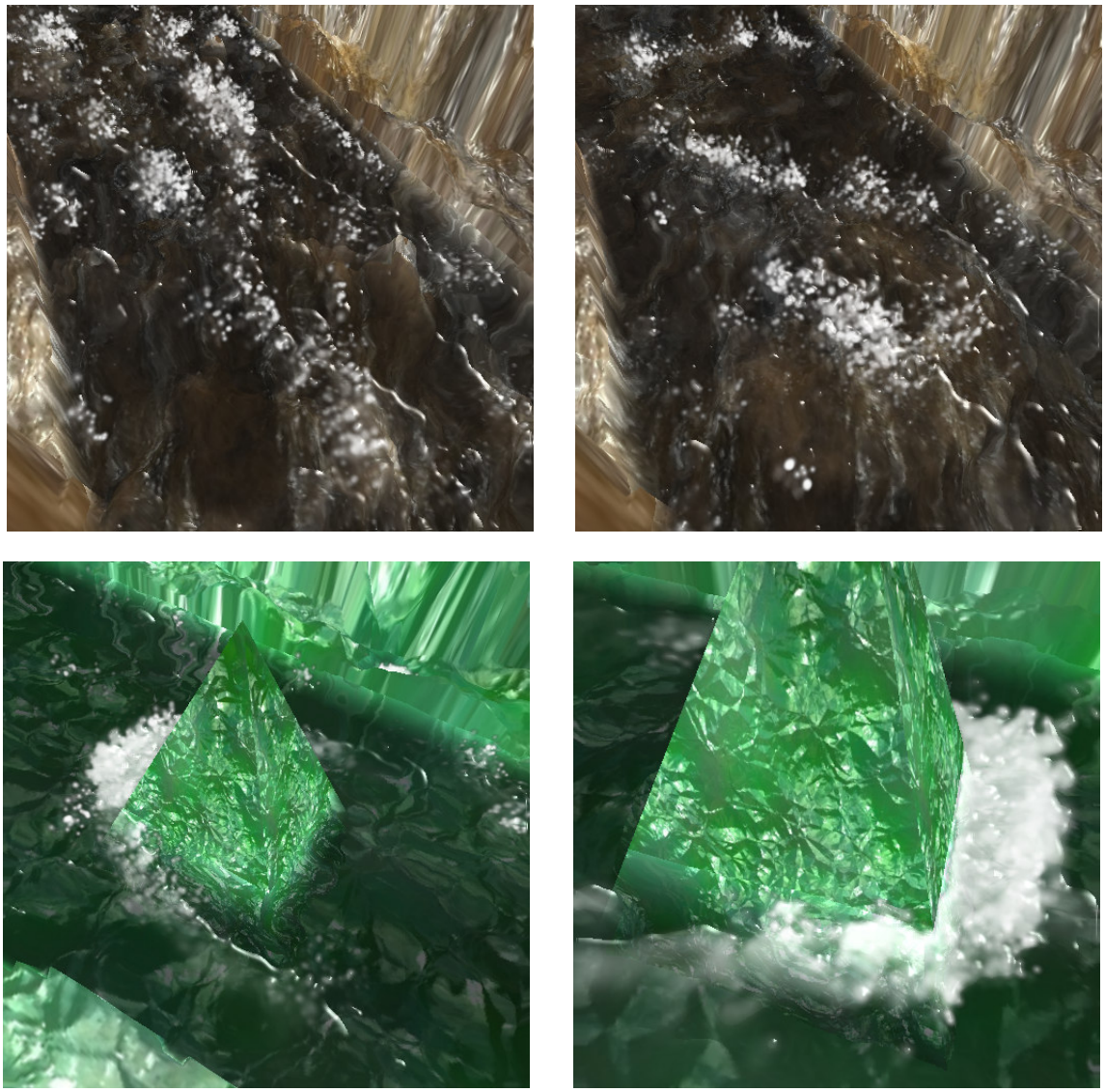

Figure 3: *Various examples of our algorithm. Only the terrain is different between these cases, and the water stream featured different characteristics with no additional input required.* ***Top-left:****Water stream with a jagged riverbed.* ***Top-right:****This riverbed features an S shape rock formation underwater. This example shows how underwater terrain feature influences the surface wake.* ***Bottom-left:****A relatively smooth riverbed, with a pyramid-shaped obstacle.* ***Bottom-right:*** *Details of the splash and foam. Notice that there is no clipping artifact.*