Real-time Rendering of Splashing Stream Water

Jun-Wei Chang Su Ian Eugene Lei
Department of Computer Science, National Tsing Hua University

Chun-Fa Chang
National Taiwan Normal University[1]

Yu-Jung Cheng
Institute for Information Industry

Abstract

We present a real-time method to simulate and render the turbulent flow and splash of stream water over irregular terrain and with dynamic rigid objects. In order to achieve real-time frame rates, we adopt a GPU-based particle system to perform the simulation, which can simulate interaction between water and dynamic rigid object in real time. We use 2D metaballs with billboards to represent the 3D water surface. Our billboard rendering method does not possess the traditional clipping artifact. Although our proposed method is not a direct physical simulation, the results are empirically plausible and efficient, and especially suitable for interactive graphics application such as computer games.

1. Introduction

Traditionally flowing water such as rivers, streams, are represented using grid mesh and texture mapping. Such method could not easily adapt to interactive change of terrain (landslide, objects passing on water surface etc). We propose a method based on particle systems that can realistically present water flowing over a dynamic terrain. Our method can produce realistic splash effects with interactive rigid objects, such as a dropping rock, and achieving a high frame-rate.

We design a system that focuses primarily on the turbulent flow and splash of water streams. Our particle system is based on the work by Lutz Latta [6] and was modified to allow the sub-particle generation and the state transition. We use a set of primary particles to simulate the general flowing water, while sub-particles may be spawned from the primary particles to portray splash effects.

Our visualization of the water effect is inspired by surface splatting and metaball rendering techniques. The organic look of metaballs is suitable for creating fluidic effects such as viscidity of water drops. However 3D metaballs are too computationally expensive for our system where thousands of particles are presented at the same time. Therefore our method uses a multi-layered 2D metaball rendering to create the effect of the water surface.

2. Previous work

There have been many methods to simulate realistic dynamic motion of fluids. Matthias et al. [1] proposed an interactive method based on Smoothed Particle Hydrodynamics (SPH) to simulate small amount of fluid with free surfaces. Kipfer et al. [2] propose a real-time approach based on SPH simulation and a "carpet-covering" method to reconstruct the surface of river, with some loss of fine details. Takahashi et al. [3] describe a method for modeling and rendering dynamic behavior of fluids with splashes and foam, using a state machine to generate the fine details. Maes et al. [4] present a column-based and height field approach to simulate water flow over irregular terrains in GPU. However, their work is not able to exhibit the phenomena of turbulent flows.

Iwasaki et al. [5] propose an off-line point-based rendering approach to divide particles into sub-particles to represent the effects of splash. Lutz Latta [6] and Kipfer et al. [7] introduced a full GPU implementation using fragment shaders of simulation and rendering of large-scale particle system. They present a general framework for GPU-based particle system and describe how to utilize the resources on the current graphic hardware.

3. Simulation

There are three sets of data that needed to be precomputed or provided by the user. The first is the terrain data that is represented by a height field. The second is the precomputation of the animation sequence, the transition condition, and sub-particle

[1] Department of Computer Science and Information Engineering

movements for each particle state. The final one is the position of the emitters. Our simulation in a single time step can be divided into the following phases:

- 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

3.1 Height field

Since our goal is to simulate the interaction between the stream water and rigid bodies like terrain and rocks, we have to pack these rigid bodies' geometry data into textures. For simplification, we assume that the terrain and other rigid bodies can be represented by a single layer height field. We can render the height field texture by setting a camera perpendicular to the terrain. In this way we can capture the height field texture of the terrain and the dynamic rigid bodies. In addition, if we need to distinguish the terrain and other rigid body objects, we can render the terrain into R channel, and other objects into other color channels. On the other hand, we also save the depth of the river in the height field texture.

3.2 2D flow field approximation

We are using the height field to approximate the 2D flow velocity field. We compute the gradient of the height field, and apply a Gaussian filter upon it. Note that we have to inverse the gradient field because the flow direction is the inverse of the gradient. We also need to add a rough velocity into the flow field to approximate the average river speed. Furthermore, when we compute the state transition in the later simulation phase, the flow field state is required. So we also determine the flow field type here. There are two flow field types: rapid_flow and slow_flow. They are determined by a flow velocity threshold φ, which is controllable by user.

3.3 Emit particles and sub-particles

The particles will be allocated or deleted in the simulation. Unfortunately, the allocation problems are serial by nature, so they cannot be handled efficiently on the GPU. We still need CPU to allocate the new particles in the simulation. One simple way is to save available indices in a stack. Another complex way is to use a priority queue that always returns a smallest

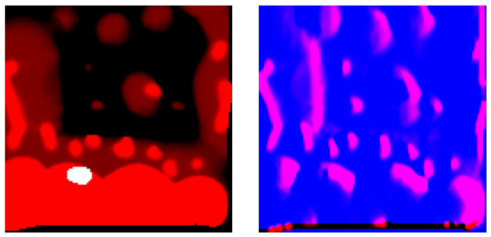
Figure 1: (Left) Height map. The white spot is a dynamic object. This is the height field that we are using for our examples in Figure 3. (Right) Flow field derived from height field.

available index. Once the index of a particle is determined, the particle attribute will be rendered to the textures as a pixel.

In NV_vertex_program3 extension, it is hard to split the particles into smaller sub-particles entirely in GPU; we still need to allocate sub-particles in CPU and pass these sub-particle attributes to 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 given 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 in CPU. In the end, these sub-particles are rendered to the attribute textures which will be described in the following subsection.

3.3.1 Attribute textures.

Before our particle system simulation, we have to allocate the following attribute textures:

a. position texture, velocity texture, state texture

b. transition texture

The positions, velocities, and states of all active particles are saved in the 2D floating-point textures. Each pixel of these textures records the attributes of each particle. Since we cannot read and write the attributes of particles at the same time, the double buffer mechanism is required.

The state texture saves the state, the age, the lifespan, and the size of each particle into RGBA channels respectively. The state is closely related to the transition texture. We can look up the transition texture (which will be described later) to determine if a state transition occurs. The age and the lifespan will be used to decide if a particle is alive and do the interpolation which depends on time. The size of the particle is also packed into the state texture to determine the particle size dynamically.

To describe a variety of particle behaviors, we pack a deterministic finite state machine into a 1D transition texture. And for simplification, only one transition is allowed for each state. Each entry records the transition condition, which consists of the collision state, the flow field state, and the transition target state. These three states will be packed into the RGB channels respectively.

3.4 Particle motion simulation

We adopt the Euler integration scheme to update particles:

$$\mathbf{v}(t+dt)=\mathbf{v}(t)\cdot scale_{particle}+\mathbf{v}_{flow}(\mathbf{p}(t))\cdot scale_{flow}+\frac{\mathbf{F}}{m}dt$$

$$\mathbf{p}(t+dt)=\mathbf{p}(t)+\mathbf{v}(t+dt)\cdot dt$$

where v_{flow} is the 2D texture computed in Section 3.2, and $scale_{particle}$ and $scale_{flow}$ parameters 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.

3.5 Collision and dampening

The collision with obstacles and the dampening of water are important effects of stream water.

Here we attempt to use an intuitive method to mimic these phenomena. If the particle collides with the terrain, the velocity of the particle will be reflected against the terrain normal. And if the particle penetrates the highest water depth, which is defined by users, we will damp the particle velocity.

3.6 State transition

The state transition plays an important role for generating the sub-particles. To determine if a state transition occurs, we can compute the collision state and flow field state from the current particle in the pixel shader, and compare the two states with the transition texture. If the transition condition is satisfied, the state transition occurs, and the current particle state will be set to the next state recorded in the transition texture. Then by reading back the state texture of particles from GPU to CPU, we can generate sub-particles as the method described in the section 3.3.

3.7 Transfer the position data to vertices

In our implementation, we use the vertex texture fetch (VTF) method to transfer the position texture data into the vertex data using graphic hardware. We can cache the vertex data in the graphic hardware in advance, and then use the VTF to translate these cached vertices to the appropriate locations.

We can render particles as point sprites, billboards, or triangle mesh, etc. Here we choose to render particles as billboards. Thus we need to cache four vertices per particle. Since we recorded the particle size in the state texture, we can determine the billboard size dynamically in the vertex shader with VTF.

4. 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 "mask". 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.

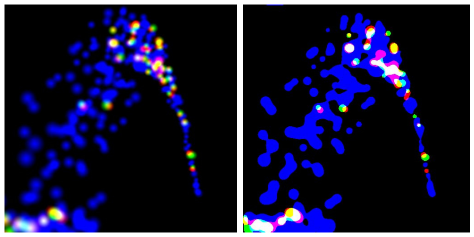

Figure 2. (Left) Filtered depth map, with RGB channels representing different layers. (Right) Metaball mask.

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 water surface.

In the final step, we use the normal map, filtered depth layers and metaball mask to create the necessary optical effects. In our experiments, we implemented cube-map based refraction effect and specular lighting. Refraction is implemented by cube-map reference using the refract vector, calculated using normal vector and eye vector. Reflection can be implemented similarly.

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.

5. 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 128×128, and the viewport resolution was set to 512×512 pixels. In our demo, the frame rates can achieve about 30 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 sub-particles generation in CPU is the bottleneck of our simulation. 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. However since we did not fully reconstruct the water surface, some artifacts may be observed when we pause the system.

Regardless, our system demonstrates the splash and the turbulent flow of river in real time. We also show that the turbulent flow can interact with the dynamic objects and the riverbed terrain.

Acknowledgements

This work is supported in part by NSC Grant 95-2221-E-007-183 and by Institute for Information Industry.

References

[1] Matthias Muller, David Charypar, and Markus Gross, "*Particle-based fluid simulation for interactive applications*", In Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer animation, pages 154-159. Eurographics Association, 2003.

[2] Peter Kipfer, Rüdiger Westermann: *Realistic and interactive simulation of rivers*. Graphics Interface 2006: 41-48

[3] T. Takahashi, H. Fujii, A. Kunimatsu, K. Hiwada, T. Saito, K. Tanaka, and H. Ueki 2003. *Realistic animation of fluid with splash and foam*. Computer Graphics Forum 22, 3, 391--401.

[4] M.M.Maes, T.Fujimoto, N.Chiba, *Efficient Animation of Water Flow on Irregular Terrains*, Graphite2006, pp.107-115, 2006.11

[5] K. Iwasaki, K. Ono, Y. Dobashi, T. Nishita, "*Point-based Rendering of Water Surfaces with Splashes Simulated by Particle-based Simulation,*" CDROM of Proc. Nicograph International, 2006-6.

[6] Lutz Latta, Building a Million-Particle System, Article of Gamasutra , July 28, 2004.

[7] Peter Kipfer and Mark Segal and Rüdiger Westermann, *UberFlow: a GPU-based particle engine*, HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware.

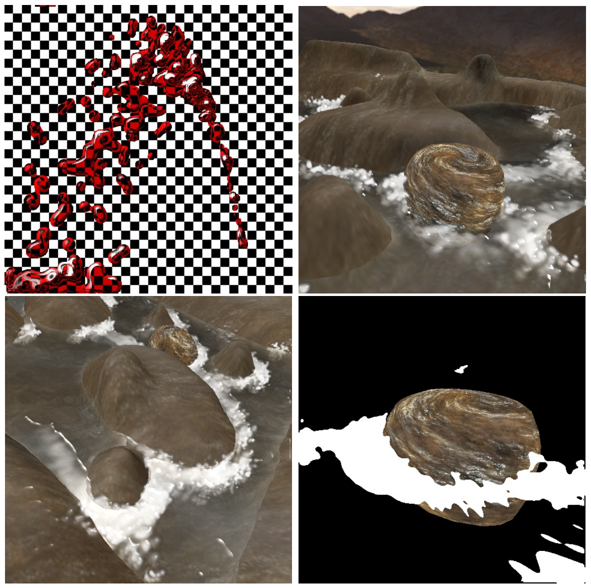

Figure 3: (Top left) Splash rendering result from Figure 2. (Top right, lower left) Water stream rendering from different angles. (Lower right) Note that clipping artifact is not visible in our method.