Real-time Character-driven Motion Effects with Particle-based Fluid Simulation

Tianchen Xu* Wen Wu* Enhua Wu*,†
X-Universe@live.com wenwu@umac.mo ehwu@umac.mo
*Faculty of Science and Technology, University of Macau, Macao, China
†State Key Lab of CS, Institute of Software, Chinese Academy of Sciences, Beijing, China

Figure 1: Character animation with fluid motion of splash simulated and rendered at 80-100 FPS

Abstract

This paper presents a novel approach for generating the real-time fluid flow driven by the motion of the character in full 3D space, based on Smoothed-Particle Hydrodynamics method. In the past relevant works, the real-time fluid-character animation could hardly be achieved due to the intensive processing demand on the movement of the character and fluid simulation. In order to handle such interactions, the motion trajectory of the character is first estimated, and then the movement of particles is constrained by the geometric properties of the trajectory. Furthermore, the efficient algorithms of the particle searching and rendering are proposed, by taking advantage of the GPU parallelization. Consequently, both simulation and rendering of 3D liquid effects with realistic character interaction can be performed on a conventional PC in real-time.

Keywords: real-time visual effects, Smoothed-Particle Hydrodynamics, character animation

1 Introduction

In many virtual reality (VR) applications and video games, the pursuit of realistic details and artistic quality becomes increasingly demanding. In retrospect to the past VR systems and video games, most of the visual effects, such as the water splash, the fire and smoke sprayed from weapons, and other special effects of fluid, were implemented by traditional particle systems. Thus, only the approximate and independent particle motion was considered. Since no interaction among particles was taken into consideration, the realistic details were difficult to be conveyed. For the sake of reality, various physical based simulation techniques have been introduced at the cost of more intensive computation in the simulation and rendering. In particular to the effects related to the fluid simulation, they can improve the immersion. However, when compared to other physical-based phenomena like rigid body simulation, the complexity is much higher because of the high dynamics of the fluid.

In this paper, our intention is to simulate liquid coupled with the motion of character, hence to achieve further advanced motion effects with realistic fluid dynamics. There are many applications concerning the requirements of the fluid effect with the precise character motion interaction, such as the artistic performance of martial arts, the stage effects of dancing with water and so on. Therefore, our research is most suitable to convey the special effects of the character-fluid interaction, such as monsters (sprites, slimes,

etc.) with liquid oozing in video games. As commonly known, character animation is essentially a comprehensive simulation process. Despite the existence of many successful cases in computer games, it is still a challenge to simulate the fluid dynamics when interacting with a high-level-of-detail character in real-time. The character motion has to be tracked and estimated in an effective way, and then to be integrated into fluid simulation under the consideration of the interaction. In addition, in order to achieve our real-time goal, better ways of fluid simulation and rendering have to be investigated.

Smoothed-Particle Hydrodynamics (SPH) [1] was chosen as the fluid simulation method in this study. Unlike Eulerian method, SPH does not require to solve the Poisson equations, whereas the methods of particle searching and visualization have to be considered. In the standards of OpenGL 2 and Shading Model 3 (SM3) of DirectX 9, the buffer could be randomly read by sampling texture on the GPU, but it could only be written in atomic threads with ordered access. With the power of CUDA or SM5 of DirectX 11, the buffer could be randomly accessed, which enables the capability of performing some complex array operations on GPU eventually. In this paper, the improved approaches for real-time fluid simulation and rendering are proposed based on the GPU to further speed up the performance of the fluid-character animation.

2 Related work

The primary area related to our work is on the motion effects and SPH. Since the fluid simulation is driven by the character motion in 3D space, the geometry-based methods for motion tracking and trajectory modeling are relevant.

2.1 Motion effects

Schmid *et al.* recently proposed a 4D data structure called Time Aggregate Object to model the motion trajectory by combining the object's geometry at certain instance into a single representation [2]. In their method, the vertices of the edges that define the surfaces were inserted between adjacent time intervals. In this volume-based structure, high-quality motion effects were generated by ray tracing. Hence, the work did not aim at the real-time rendering. In order to improve the geometry-based motion trajectory control, Sander *et al.* developed an efficient method for traversal of the mesh edges using adjacency primitives on GPU [3]. Their method was effective to optimize the motion blur algorithm in the original application by identifying the shared edges to avoid redundant edge extrusion in real-time.

Xu *et al.* developed a fast method for tracking a long trajectory by splitting it into small segments, and constructing the segments on GPU, that optimized the motion-tracking pipeline microscopically [4]. Furthermore, they also realized a screen-space smoke motion effect according to a simplified Navier-Stokes equation on the grid-based method in real-time. It is a primarily successful attempt on generating fluid motion effects, but the rendering method cannot support the realistic illumination. The method is thus deficient in creating vivid 3D perception. The volume-based method is costly and highly depends on the resolution, leading to the trade-off between the quality and efficiency.

2.2 SPH simulation

SPH has been successfully deployed in Computer Graphics to simulate fluid in real-time, since it was first introduced by Müller *et al.* [5]. Methods to improve the performance including adaptive sampling [6] and predictive-corrective pressure computation [7], were fully implemented on CPU. In [8], the fluid simulation by SPH was implemented on GPU, but the neighbor particle searching was still computed on CPU. Later, many GPU-based methods for the neighbor particle searching have been proposed. Kipfer *et al.* [9] built a particle engine, and used a 2D texture to locate the potential particles within the smoothing radius approximately. Harada *et al.* [10] proposed a method based on bucket sort to search the neighbor particles in a 3D grid on GPU using the pixel/fragment shader in graphics pipeline. The limitation is that the size of each bucket is limited to 4 (RGBA). In [11], a method of KD-Tree on GPU was proposed. This method, however, suffers from the high cost of memory access, so that the overall performance is slowed down.

Broad-phase collision detection with CUDA was studied in [12]. In their study, the particle

position and particle ID were hashed and then sorted (a good alternative is bitonic sort). Subsequently, the parallel threads were dispatched to determine the start and end addresses of each bucket, so that the particle buffer was partitioned with the buckets. The particles were sorted finally. The method works in the complexity of $\Theta(NlogN)$, suitable for fully GPU computation both in CUDA and DirectX with some optimization. However, the bit width of data for hashing is limited. Developed along the similar line of approach, the GPU-based neighbor search method on binary search can be found in [13][14]. In [15], two types of atomic operations in CUDA have been proposed. In our paper, a neighbor search algorithm without hashing and full sorting is implemented on GPU (see Section 3.3 for details). Therefore, the computation efficiency can be improved without any space overhead.

2.3 Particle rendering

In Müller's work [5], where SPH was first introduced into Computer Graphics for real-time fluid simulation, the examples of particle rendering via Marching Cube algorithm [16] and point splatting method were both successfully tested. Later, in Van Kooten's study, splatting was implemented on GPU to efficiently visualize the metaballs [17]. Both Marching Cube and splatting can obtain the real-time rendering result for fluid simulation of certain scale. However, these two methods are still time-consuming when the entire scene is complex. The Marching Cube algorithm has to extract the iso-surface of the specified scalar field by intensive computations for each voxel. Splatting has a high requirement of the particle number, typically at least 10,000 to 100,000, for rendering. Usually, in the real-time VR systems and video games, it is not necessary to render the scene in such high resolutions. Therefore, Van der Laan *et al.* proposed a screen space method, in which the depth buffer was utilized to compute the normal vectors in the image space for illumination [18]. The particles were only rendered as the billboards, which greatly reduced the geometric computation. In this paper, an improved screen space method is introduced to enhance the rendering quality in Section 4.3.

3 Method and principles

In order to simulate the fluid dynamics driven by the character motion, the overall work-flow consists of three steps:

- Track the current and past motion states after skinning the character with skeletal animation and estimate the coming motion state.
- Emit the particles from the surface of the skinned character with the initial dynamic information such as the velocity driven by the estimated motion.
- Simulate the fluid dynamics using the SPH as well as processing the further fluid-character coupling.

In the following subsections, the methods in each step are described in detail.

3.1 Emitter distribution uniformization

To facilitate the fluid - character interaction, the character should be represented in both mesh and particle form. In our method, the particles of the character surface are defined as the emitters. A simple method is to assign the original vertices of the mesh as the emitters. However, that could not guarantee the emitters are uniformly distributed because the vertex layout is often nonuniform (as the example shown in Figure 2), causing an uncontrollable emission state.

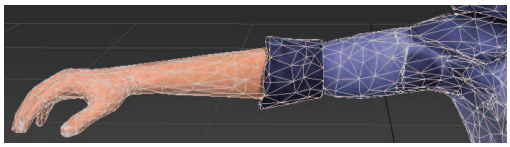


Figure 2: Usually, the vertices of an arbitrary mesh are not uniformly distributed.

In order to obtain an uniform distributed emitters, a simple but efficient method is utilized based on the rasterization of the graphics pipeline as well as the texture coordinates (UV) of each vertex. The reason to use the texture coordinates is that, in most cases, UV coordinates are isotropic.

In the rectangle texture image, the area which has the information of the character texture is defined as the valid area. First, a binary UV mask of the valid texture area (the area in white as shown in Figure 3) was created in the preprocessing step. An index buffer was created to

hold the indices (UV value) of the pixels in the valid area by traversing the UV mask. Then, a position buffer was populated by rendering the position vector of each pixel on the character surface into UV space according to the UV atlas.

The emitters are originally at each vertex of the character surface. During the emission, a UV value of an emitter is got from the index buffer, and then the position information is obtained by sampling the position buffer accordingly.

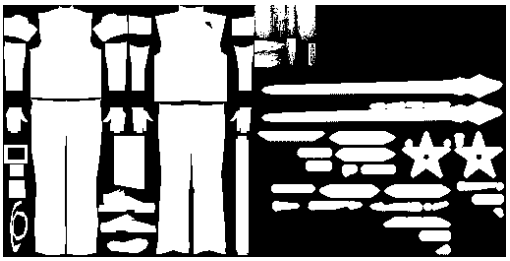


Figure 3: The UV mask for position buffer

The one-to-one mapping is an important condition of this method. Otherwise, the different emitters may be mapped onto the same pixel. The case may occur when artists assign the symmetric meshes with the same UV atlas in order to save the texture space. One solution is to re-map the UV by some effective UV unwrapping algorithms, such as Least Squares Conformal Maps (LSCM) [19] or Angle Based Flattening (ABF/ABF++) [20].

3.2 Motion tracking & state estimation

Once the emitters are identified, the fluid particles can be emitted on the basis of the character motion states. The emission position and initial velocity for each particle to be emitted are determined by the following two steps.

Firstly, the motion and the corresponding positions in the current frame at time t, and the historical frames at time $(t-\Delta t)$, $(t-2\Delta t)$, and $(t-3\Delta t)$ are tracked and recorded for all vertices of the character mesh. The positions of the vertex $\vec{A}_{t-n\Delta t}$ at time $(t-n\Delta t),\ (n=0,1,2,3)$ can be obtained by:

$$\vec{A}_{t-n\Delta t} = T(t-n\Delta t)\vec{A}, \quad n=0,1,2,3 \qquad (1)$$

where $T(t-n\Delta t)$ is the transformation matrix at time $(t-n\Delta t)$. $\vec{A}$ is the original position of the mesh vertex without skinning.

As mentioned in the previous section, after the vertex transformation, the vertices are rasterized into the UV space. Hence, the emitter position $\vec{P}_{t-n\Delta t}$ at time $(t-n\Delta t)$ can be determined correspondingly after the rasterization. Then, the tangent vectors of each emitter at time $(t-\Delta t)$ and $(t-2\Delta t)$ can be computed by:

$$\begin{cases} \vec{m}_{t-\Delta t} & = k(\vec{P}_t - \vec{P}_{t-2\Delta t}) \\ \vec{m}_{t-2\Delta t} & = k(\vec{P}_{t-\Delta t} - \vec{P}_{t-3\Delta t}) \end{cases} \qquad (2)$$

where k is a tension parameter to scale the norm of the tangent vector. In our implementation, $k = \frac{1}{2}$ was adopted.

Secondly, the trend of the current motion can be predicted from the cue of the historical motion states. Assuming a constant acceleration of the character motion, the tangent vector of each emitter at time t is estimated by the tangent vectors at time $(t-2\Delta t)$ and $(t-\Delta t)$:

$$\vec{m}_t = \vec{m}_{t-\Delta t} + (\vec{m}_{t-\Delta t} - \vec{m}_{t-2\Delta t}) \qquad (3)$$

The fluid particles that start appearing in the frame at time t are actually emitted in the time interval $[t-\Delta t, t]$, that is between the current and previous time frames asynchronously. At time t, the particles appear approximately along the motion of the trajectory geometrically bounded by the emitter positions at $(t-\Delta t)$ and t. The Catmull-Rom (cubic Hermite) spline was utilized to carry out the trajectory interpolation fitting from the previous motion state to the current one:

$$\vec{P}(\tau) = H_{00}\vec{P}_{t-\Delta t} + H_{10}\vec{m}_{t-\Delta t} + H_{01}\vec{P}_t + H_{11}\vec{m}_t$$
$$\text{where } H = \begin{bmatrix} 2\tau^3 - 3\tau^2 + 1 & -2\tau^3 + 3\tau^2 \\ \tau^3 - 2\tau^2 + 1 & \tau^3 - \tau^2 \end{bmatrix},\ \tau \in [0,1] \qquad (4)$$

For the emission velocity $\vec{v}_{impulse}$ of a fluid particle, its magnitude is approximately calculated by the length of the trajectory segment during the interval Δt, and its direction is along the direction of the tangent vector at the corresponding location on the trajectory curve:

$$\vec{v}_{impulse} = \|\vec{P}\| \cdot \frac{H'_{00}\vec{P}_{t-\Delta t} + H'_{10}\vec{m}_{t-\Delta t} + H'_{01}\vec{P}_t + H'_{11}\vec{m}_t}{\|H'_{00}\vec{P}_{t-\Delta t} + H'_{10}\vec{m}_{t-\Delta t} + H'_{01}\vec{P}_t + H'_{11}\vec{m}_t\|}$$
$$\text{where} \begin{cases} \|\vec{P}\| & = \dfrac{1}{\Delta t}\displaystyle\int_0^1 \|\vec{P}(\tau)\| d\tau \\ H' & = \begin{bmatrix} 6\tau^2 - 6\tau & -6\tau^2 + 6\tau \\ 3\tau^2 - 4\tau & 3\tau^2 - 2\tau \end{bmatrix},\ \tau \in [0,1] \end{cases} \qquad (5)$$

where $\|\vec{P}\|$ is the length of the trajectory segment during Δt, and H' is the derivative of H in Eqn. (4). The integral can be implemented by Simpson's rule. Thus, the impulse from the character motion to the fluid field is calculated by Eqn. (5).

After the emission velocity is determined, it can be considered as the external force contribution to the Navier-Stokes equation [21] for fluid simulation. The external forces in the fluid field mainly come from the motion of the character. During the coupling of the character and the fluid simulation, the emitters on the character surface are responsible to generate the boundary condition applicable to the fluid simulation. Therefore, when updating the particle positions of the character in the advection pass, the positions of these emitters are updated by the method introduced in Section 3.1 rather than calculated by the SPH equation.

3.3 Neighbor search on GPU

Except for the physical computation and rendering, the bottleneck of the most particle-based simulation is to search the adjacent particles within the smoothing radius h. A straightforward way is to arrange the particles in advance using the bucket sort. However, it is still hard to predict how much space necessary for each cell (bucket), leading to a waste of space when overestimated. Another choice is to sort the particles within an exactly tightened memory space, but the cost of the full sorting is expensive. In the following, a fast and novel indexing method is introduced to realize the bucket sort with dynamic space allocation on GPU:

1) For each cell, the number of particles within the smoothing radius is firstly counted and stored in the x component of a buffer, called as the index buffer. Meanwhile, the relative (local) position of each particle in the cell is recorded in another buffer, called as the offset buffer. In Figure 4, the buffers allocated for a 16-cell bucket sort are shown as an example. The x component of the index buffer after the first step is shown in Figure 4 (a). In the example, the 6th cell has 4 particles. The relative positions of the particles in each cell are kept in the offset buffer as shown in Figure 4 (b).

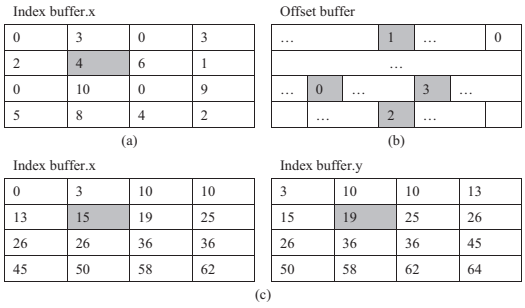


Figure 4: The buffers allocated for neighbor search on GPU. (a) The index buffer after the first step. The number of particles in the 6th cell is 4. (b) The offset buffer. The relative positions of the particles in the 6th cell are shaded in grey. (c) The index buffer after the second step.

2) For the particle indices, the start and end addresses of each cell are computed by Eqn. (6) according to the particle number per cell, and then stored into the x and y components of the index buffer, respectively, as shown in Figure 4 (c).

$$i^j_{end} = \begin{cases} a_j, & j = 0 \\ i^{j-1}_{end} + a_j, & j > 0 \end{cases} \qquad (6)$$
$$i^j_{start} = i^j_{end} - a_j$$

where i^j is a particle address belonging to cell j, and a^j is the number of particles in cell j.

Therefore, the particles can be sorted according to the start address of each cell and the offsets. When the particles in the smoothing radius h are accessed, the particle information in the cell and its neighbor cells are easily obtained. Thus all particles of the specified cell are available for traversal.

4 Implementations

Our method was implemented upon the framework of DirectX 11.

4.1 Particle emission along trajectory

Firstly, the character was skinned in a high quality using the Dual-Quaternion Algorithm [22] to shape a motion state (one pose). Unlike the Linear Matrix Blending method, which is often

used for skinning, the Dual-Quaternion Algorithm can avoid the twisted flaw. Then, the character can be regarded as a static mesh after all the vertices were stored from the vertex shaders into the vertex buffers via Stream-Output (also called Transformed-Feedback in OpenGL). We kept four buffers to fetch the skinned data for the computation introduced in Section 3.2, and updated them by the method for the producer-consumer problem every frame during the real-time processing.

When the vertex buffers with the skinned positions were populated, the particles were emitted according to the process in Section 3.2. The pseudo code of the particle emission algorithm is shown as follows:

Algorithm 1 $Emit(tid)$ // tid is the thread ID

if $Particle[tid].lifetime > LIFE\_LIMIT$ **then**
  $i \leftarrow random(0, sizeof(ValidArea(Position_t)))$
  // $Position_t$ is the position at t in the position buffer
  $\{A, B, C, \alpha, \beta\} \leftarrow GetEmitterData(i)$
  for $n \leftarrow 0$ to 3 **do**
    $\vec{p}_{t-n\Delta t} \leftarrow Position_{t-n\Delta t}$
    /* $Position_{t-n\Delta t}$ is the position at $(t - n\Delta t)$ in the position buffer */
  end for
  $\vec{m}_{t-\Delta t} \leftarrow 0.5(p_t - p_{t-2\Delta t})$
  $\vec{m}_{t-2\Delta t} \leftarrow 0.5(p_{t-\Delta t} - p_{t-3\Delta t})$
  $\vec{m}_t \leftarrow 2m_{t-\Delta t} - m_{t-2\Delta t}$
  $\tau \leftarrow random(0, 1)$
  $l \leftarrow random(0, LIFE\_LIMIT\ /\ 2)$
  $Particles[tid].SetHermite(\vec{p}_{t-\Delta t}, \vec{m}_{t-\Delta t}, \vec{p}_t, \vec{m}_t)$
  $Particles[tid].position \leftarrow Eqn4(\tau)$
  $Particles[tid].velocity \leftarrow Eqn5(\tau)$
  $Particles[tid].lifetime \leftarrow l$
else
  $Particles[tid].lifetime++$
end if

4.2 Simulation pipeline on GPU

It was observed that the random access was only necessary for the density and force update. Hence, the buffers allocated for the fluid simulation are shown in Table 1.

Table 1: Buffers allocated for SPH

Buffer	Data Type	Quantity
Position	float3 (3×32 bits)	4
Particle	structured type*	2
Density	32-bit float	1
Force	float3	1

*Containing the position (float3), velocity (float3), and lifetime (UINT, 32-bit unsigned integer)

The fluid simulation pipeline shown in Algorithm 2 was implemented using the Compute Shaders in DirectX11.

Algorithm 2 Fluid simulation pipeline in each time step

$Particle[i] \leftarrow Emit(Position)$
$Particle'[i] \leftarrow Arrange(Particle[j])$
// $Particle'$ is the replicated buffer to store sorted particles
$Density[i] \leftarrow ComputeDenisty(Particle')$
$Force[i] \leftarrow ComputeForce(Density, Particle')$
$Particle[i] \leftarrow Integrate(Force[i], Const)$
/* $Const$ represents the constraints, such as the boundary, and the external forces */

The operations in each pass are described as follows in detail:

1) ***Emission:*** the particles are emitted into the simulation field by Algorithm 1.
2) ***Indexing:*** the particles are indexed into the position order.
3) ***Density computation:*** the density of the fluid is computed by Eqn. (7) [5].
4) ***The computation of the internal force contribution:*** the accelerations of each particle contributed by the pressure and viscosity are calculated by Eqn. (8) [5].
5) ***Integration:*** the total accelerations are calculated by summing all internal and external force contribution. Then, the velocities are updated accordingly. Consequently, the new particle positions are obtained. If the particles are marked as the boundary, their new positions are updated by tracing the corresponding emitter positions.

$$\rho_i = \frac{315}{64\pi h^9} \sum_j m_j (h - \|\vec{r}_i - \vec{r}_j\|^2)^3 \tag{7}$$

$$\begin{cases} \vec{a}_{press} &= \frac{45}{\pi h^6} \sum_j m_j [\frac{p_i + p_j}{2\rho_i \rho_j} (h - r)^2 \frac{\vec{r}_i - \vec{r}_j}{r}] \\ \vec{a}_{visc} &= \frac{45}{\pi h^6} \sum_j \mu m_j \frac{\vec{u}_j - \vec{u}_i}{\rho_i \rho_j} (h - r) \end{cases}$$
$$\text{where } r = \|\vec{r}_i - \vec{r}_j\| \tag{8}$$

4.3 Rendering

An improved screen space fluid rendering method was proposed in order to reduce the rendering cost. The basic steps of the screen space method are listed in the following [18]:

1) The particles are rendered as the billboards. The depth values are decreased based on the ellipsoid surface equation in pixel shading, and recorded into the z-buffer with the z-testing based culling by the hardware.
2) The values in the z-buffer are blurred to make the metaball-shaped particles.

3) The normal vectors are computed by taking the gradient of the depth field in the viewing space (unprojected from the z-buffer). Then the illumination is implemented with the refraction map resolved from the frame-buffer.

In Van der Laan's work, the particle size was uniform, thus causing the unnatural rendering effect especially at the boundary of the splash. The free particles in these areas should have been sparsely distributed rather than forming a heavy cluster. In order to solve this problem, the density computed in the simulation stage has been introduced to control the size of the billboard. Considering the billboard size increased moderately with the density for cluster cohesion, we used the exponential function kd^p as an approximation, where d, k and p denote the density, a scalar proportional to the average particle size, and an exponent between 0 and 1, respectively. In our implementation, $k = 0.014$ and $p = 0.6$ were adopted by testing as shown in Figure 5 (c) and (d).

In addition, due to the blurring operation with a fixed radius in the image space, some particles that were close in the xy-plane but far away from each other in the z-direction were wrongly blurred as well. Another problem is that the fixed radius in the image space (projected) results in the inconsistency of the particle blending measured in the 3D viewing space (unprojected). To tackle these problems, the blurring radius r_{blur} in the image space needs to be dynamically controlled based on the depth value unprojected from the z-buffer, as computed in Eqn. (9).

$$r_{blur} = \frac{m_{00} r P_w w}{2} \quad \text{where } \vec{P} = M_p^{-1} \vec{P}_p \tag{9}$$

where $\vec{P}_p$ denotes the pixel position in the normalized projected space, which can be derived by the screen position and the depth value from the z-buffer. M_p and m_{00} represent the perspective projection matrix and its first entry, respectively. Besides, $\vec{P}(P_x, P_y, P_z, P_w)$, r, and w are the unprojected pixel position, the blurring radius set in the viewing space, and the width of the viewport measured in pixels, respectively.

The comparison of the rendering results under different conditions is shown in Figure 5.

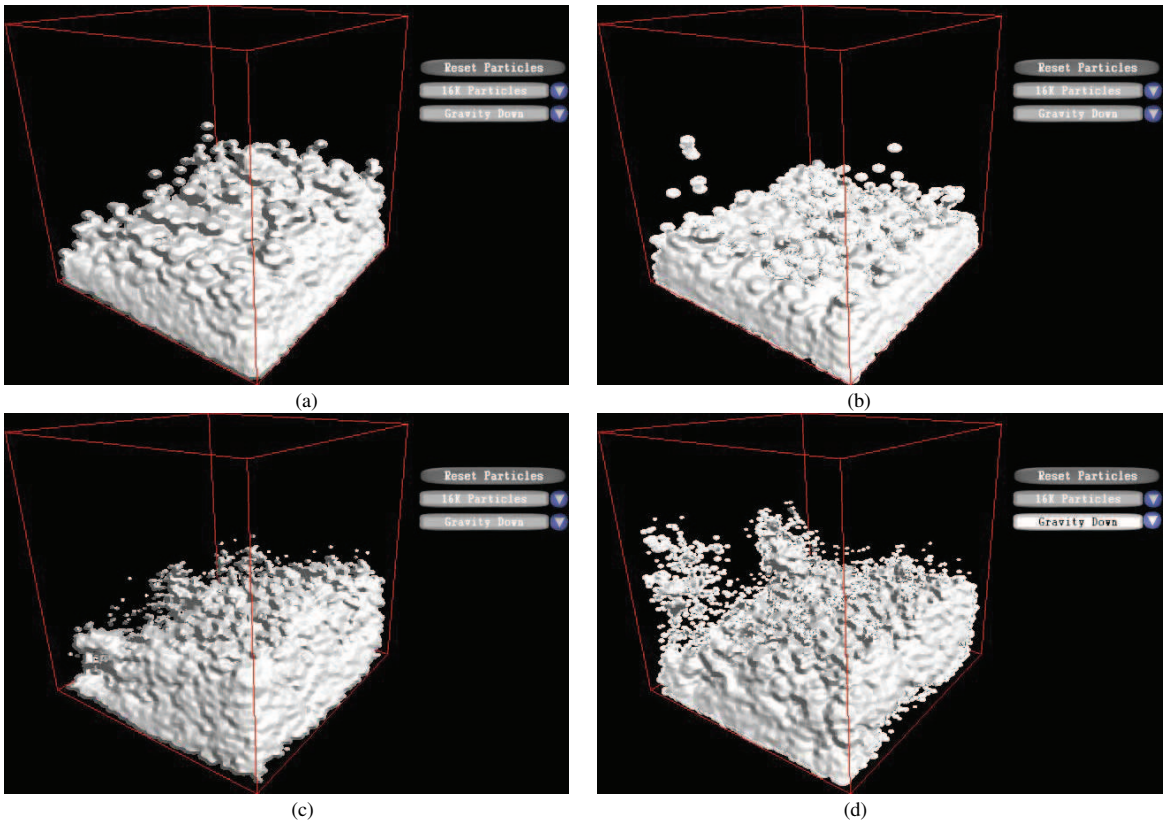


Figure 5: Rendering results (a) without any enhancement; (b) with dynamic blurring radius; (c) with dynamic billboard size of $0.014d^{0.6}$ applied; (d) with dynamic blurring radius and dynamic billboard size of $0.014d^{0.6}$ applied.

5 Results

In this section, our method is evaluated. A water dance scene was created for testing. The testing machine is equipped with Intel® Core™ i7-2600K CPU and Nvidia® GTX590 graphic card.

5.1 Performance & Evaluation

Firstly, the efficiency of our neighbor search for the SPH algorithm was tested. The overall SPH execution time with our neighbor search method was compared with that of "FluidCS11" in Direct 11 SDK. The neighbor search method in FluidCS11 is based upon hashing and bitonic sorting [12]. The comparison result is shown in Figure 6.

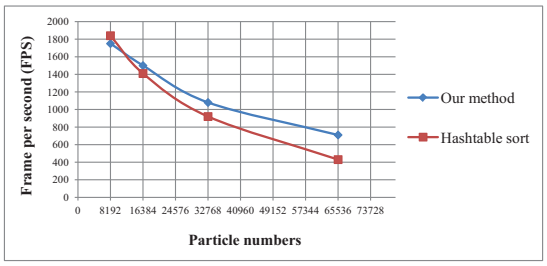


Figure 6: Comparison of the simulation speed in FPS

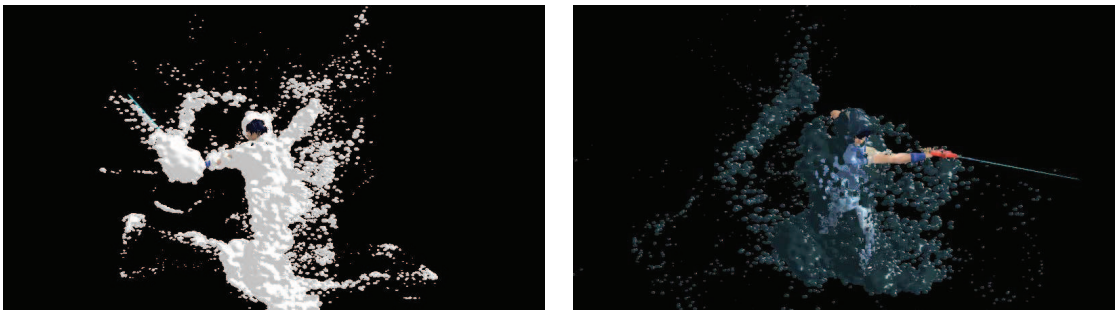
Our method has two advantages over the hashtable sorting method. Firstly, the time complexity of the neighbor search method based on the hashtable sorting is $\Omega(NlogN)$ on average, but $\Theta(N^2)$ in the worst case. Therefore, it is seriously influenced by the number of particles. Our method based on the scan algorithm has the time complexity of $\Theta(N)$. Therefore, as the number of the particles increases, the advantage of our method becomes more obvious.

Moreover, the space used in our method is relatively low as well. The hashtable sorting method needs two more buffers to record the hashed value and the particle data during the sorting, respectively. In our algorithm, a small buffer is required to record the offset, and the index buffer is utilized to keep the number of particles per cell and the particle addresses. Furthermore, the word length for hashing must be long enough (usually the 64-bit integer for the 3D case) in order to index all particles and cells. However, the current GPU can only provide the 32-bit integer representation, leading to the difficulty of coding. Our method can avoid such problem because only particles need to be indexed.

5.2 Water Dance Example

The real-time water dance (see Figure 7 and Figure 1) in a complex 3D environment was created by our proposed method. The scene consists of the character, water, and 3D complex models of environment, such as trees and ancient

Figure 7: The water dance with fluid motion effects in a complex 3D environment



(a) (b)

Figure 8: Fluid motion effects of the sword dance with (a) illumination; (b) illumination and the refraction mapping

Table 2: The frame rates and the corresponding model scales of the water dance simulation

Simulation Experiments		Number of Water particles	Total number of vertices	Total number of faces	Average FPS
Case 1	Character and water simulation only	8192	44710	36197	181
		16384	77478	34367	95
		32768	143014	50771	54
Case 2	Character and water simulation with 3D environment	8192	58192	46718	132
		16384	90961	44888	65
		32768	156497	61292	43

buildings. The character model in our experiments is 11,942 vertices and 18,003 faces. The frame rates with different water scales of the water dance simulation are shown in Table 2. Figure 8 shows a sword dance with fluid motion effects without the 3D environment.

In the water dance example, the fluid interacts with the character in motion with a high quality skinning. The motion effects depend on both the dynamics of the character motion and the physical properties of fluid. The dynamics of splash shown in our example is the result of collision with the character and the ground as the boundary. Furthermore, our improved screen-space rendering method is viewing independent since it is based on the dynamic blurring adjustment. Thus, the rendering quality remains stable, wherever the view point is located.

6 Conclusions

In this paper, an effective approach to realizing the real-time fluid motion effects driven by the character has been proposed. The particles were successfully generated based on the motion trajectory of the character. The details of the fluid splashing effect around the character and the fluid interacting with the character motion have been vividly simulated. The techniques of the motion tracking, data extraction from the motion states, fast fluid simulation with SPH, and the fluid rendering have been presented. The animation can be run in real-time with the plausible rendering result. Hence, it has reached our goal to present a fantastically artistic effect. It is also a good attempt for simulating the fluid-character interaction.

Our future work will focus on more complex interactive simulations of real-time animations, involving in two-way fluid-character coupling and multiple fluids blending with character interaction.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the manuscript. The work was supported by the grants (MYRG202(Y1-L4)/FST11/WEH, MYRG150(Y1-L2)/FST11/WW) and the studentships of University of Macau, the National Fundamental Research Grant 973 Program (2009CB320802), and NSFC (61272326).

References

[1] J. J. Monaghan. Smoothed Particle Hydrodynamics. *Annual Review of Astronomy and Astrophysics*, 30(1):543–574, 1992.

[2] J. Schmid, R. W. Sumner, H. Bowles, and M. Gross. Programmable motion effects. In *ACM SIGGRAPH'10 papers*, pages 57:1–57:9, 2010.

[3] P. V. Sander, D. Nehab, E. Chlamtac, and H. Hoppe. Efficient traversal of mesh edges using adjacency primitives. In *ACM SIGGRAPH Asia'08 papers*, pages 144:1–144:9, 2008.

[4] T.-C. Xu, E.-H. Wu, M. Chen, and M. Xie. Real-time motion effect enhancement based on fluid dynamics in figure animation. In *Proc. ACM VRCAI'11*, pages 307–314, 2011.

[5] M. Müller, D. Charypar, and M. Gross. Particle-based fluid simulation for interactive applications. In *Proc. ACM SIGGRAPH/Eurographics SCA'03*, pages 154–159, 2003.

[6] B. Adams, M. Pauly, R. Keiser, and L. J. Guibas. Adaptively sampled particle fluids. In *ACM SIGGRAPH'07 papers*, 2007.

[7] B. Solenthaler and R. Pajarola. Predictive-corrective incompressible SPH. In *ACM SIGGRAPH'09 papers*, pages 40:1–40:6, 2009.

[8] T. Amada, M. Imura, Y. Yasumuro, Y. Manabe, and K. Chihara. Particle-based fluid simulation on GPU. In *GP2 Workshop Proc.*, 2004.

[9] P. Kipfer, M. Segal, and R. Westermann. Uberflow: a GPU-based particle engine. In *Proc. ACM SIGGRAPH/Eurographics HWWS'04*, pages 115–122, 2004.

[10] T. Harada, S. Koshizuka, and Y. Kawaguchi. Smoothed Particle Hydrodynamics on GPUs. In *Proc. CGI'07*, pages 63–70, 2007.

[11] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time KD-tree construction on graphics hardware. In *ACM SIGGRAPH Asia'08 papers*, pages 126:1–126:11, 2008.

[12] S. Le Grand. Broad-phase collision detection with CUDA. In *GPU Gems 3*. NVIDIA, 2007.

[13] X. Zhao, F. Li, and S. Zhan. A new gpu-based neighbor search algorithm for fluid simulations. In *2nd International Workshop on Database Technology and Applications (DBTA)*, pages 1–4, 2010.

[14] S. Bayraktar, U. Güdükbay, and B. Özgüç. GPU-based neighbor-search algorithm for particle simulations. *J. Graphics, GPU, & Game Tools*, pages 31–42, 2009.

[15] P. Goswami, P. Schlegel, B. Solenthaler, and R. Pajarola. Interactive SPH simulation and rendering on the GPU. In *Proc. ACM SIGGRAPH/Eurographics SCA'10*, pages 55–64, 2010.

[16] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM SIGGRAPH Comput. Graph.*, 21(4):163–169, August 1987.

[17] K. van Kooten, G. van den Bergen, and A. Telea. Point-based visualization of metaballs on a GPU. In *GPU Gems 3*. NVIDIA, 2007.

[18] W. J. van der Laan, S. Green, and M. Sainz. Screen space fluid rendering with curvature flow. In *Proc. ACM I3D'09*, pages 91–98, 2009.

[19] B. Lévy, S. Petitjean, N. Ray, and J. Maillot. Least squares conformal maps for automatic texture atlas generation. In *Proc. ACM SIGGRAPH'02*, pages 362–371, 2002.

[20] A. Sheffer, B. Lévy, M. Mogilnitsky, and A. Bogomyakov. Abf++: fast and robust angle based flattening. *ACM Trans. Graph.*, 24(2):311–330, 2005.

[21] D. Pnueli and C. Gutfinger. *Fluid Mechanics*. Cambridge University Press, 1997.

[22] L. Kavan, S. Collins, J. Žára, and C. O'Sullivan. Geometric skinning with approximate dual quaternion blending. *ACM Trans. Graph.*, 27(4):105:1–105:23, 2008.