Real-time Motion Effect Enhancement Based on Fluid Dynamics in Figure Animation

Tian-Chen Xu*[1] En-Hua Wu†[1,2] Mo Chen[1] Ming Xie[1]

[1]Faculty of Science and Technology, University of Macau, Macao, China
[2]State Key Lab of CS, Institute of Software, Chinese Academy of Sciences, Beijing, China

Abstract

In fast figure animation, motion blur is often employed to generate fantastic effects of figure motion, for exaggerating the atmosphere one wants to convey. In the previous works for long time, the solution based on certain kind of image blending in terms of hardware or software, was simply adopted. In order to provide high standard of motion blur effect, methods based on 3D geometry of the motion figure with global illumination become gradually in demand. However, the computation cost to meet such demand is very high and it is hard to achieve real time rendering.

In our work, a novel 3D geometric approach of real-time motion effect is proposed. By the approach, a special effect along the motion trajectory based on fluid simulation is combined with the volumetric motion blur. Besides, in order to avoid the redundant calculation of each frame and break the limitation of trajectory generation, we decompose the motion trajectory and employ multi-pass geometry rendering to achieve geometry instancing for reuse. In the pipeline, we separate motion tracking and fluid solution, to support various fluid effects flexibly. As our algorithm is constructed in the context of GPU geometry shading in parallel, high efficiency of computation is guaranteed while realizing gorgeous rendering. As a result, real time rendering including the motion blur effect is achieved.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Radiosity;

Keywords: motion blur, skeletal animation, fluid dynamics, GPU geometric processing

1 Introduction

When the pursuit of realistic detail and artistic quality becomes even stronger in figure animation, motion blur becomes an indispensable part of animation special effect. In various kinds of rapidly moving environments like the extreme speed of a racing car, the vibration of an explosion airstream or just the dizziness one character feels, motion blur can make them all more expressive with convincing particulars. With the help of GPU computation, the trails of moving objects can be analyzed to provide better continuity and smoothness for live-action footage.

Making fast-moving scene more realistic has always been the main purpose of motion blur technique. Originally, it was not realized by simply inserting data into adjacent frames, but by merging individual frame with its prior frame. In using the most popular tools of DirectX 9 and OpenGL 2.0, the outcome did not altogether satisfy the viewers: the moving objects and their background are often mixed together, leading to the result of image quality downgrading. Furthermore, its efficiency is also rather poor. Due to all these reasons, motion blur is not quite as popular.

In our paper, a novel approach of real-time motion effects based on geometry volume is proposed, which presents motion blur with well quality of illumination in 3D space, by taking advantage of modern GPU capability. To prevent from the over-duplicated calculation of each frame, we decompose the trajectory into segments, and employ multi-pass geometry rendering to achieve geometry instancing for reuse. Besides, a special calculation in real time based on fluid dynamics is performed to enhance the post-motion effect.

The structure of our paper is as follows: after briefly discussing related works on the topics concerning our study interest in the Section 2, we show the main principles of our method, with descriptions of the overall algorithms in Section 3. Some key solutions for implementation and the testing results in varying complexity will be presented in Section 4&5. Finally, we conclude in Section 6 with some discussions of the future work.

2 Related Work

In the early days, motion blur is often achieved with the help of OpenGL accumulation buffer. Haeberli and Akeley [1990] provided in-depth analysis towards the use of this architecture. Using ray tracing to perform the Monte Carlo evaluation of integrals [Cook et al. 1984] in the rendering equations, the problems of motion blur, depth of field, and penumbra can be solved. As the accumulation buffer is separated from the normal rendering hardware, the performance of the hardware was unable to achieve the requirement as demanded in this aspect. However, since GPU has come to our sight nowadays, accumulation buffer was gradually abandoned.

Brostow and Essa [2001] present a postprocess approach to simulate motion blur automatically. They first track the motion within the image plane, and then integrate the changing scene as the time elapses. In this manner, a better support for live-action footage and smoothness can be accomplished. The problem is, image-based motion blur can only provide the most basic effect, without sufficient flexibility. Some powerful or special effect such as the fluid dynamics we proposed in this work cannot be integrated with it.

A framework for elliptical weighted average (EWA) surface splatting with time-varying scenes was introduced by Heinzle et al. [2010]. They use a piecewise linear approximation to construct a rendering algorithm for point-based objects. In the context of point-sample geometry, 3D Gaussian kernel rather than 2D Gaussian kernel is employed to unify a spatial and temporal component for motion-blurred images, and the change makes sure the continuity in both space and time.

Since Microsoft released a sample ”Motion Blur 10” in DirectX 2007, a geometric approach came into the world with the birth of

*e-mail: TianchenX@siggraph.org
†e-mail: EHWu@umac.mo

ACM Reference Format Tian-Chen Xu, En-Hua Wu, Mo Chen, and Ming Xie. 2011. Real-time motion effect enhancement based on fluid dynamics in figure animation. In Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry (VRCAI '11). ACM, New York, NY, USA, 307-314. DOI=10.1145/2087756.2087803 http://doi.acm.org/10.1145/2087756.2087803

Geometry Shader in Shader Model 4. Based on this method, Sander et al. (Hong Kong UST, Microsoft Research, and Princeton University) [2008] proposed an efficient method for traversal of mesh edges using adjacency primitives, which was effective to optimize the motion blur algorithm in the original application by identifying the shared edges to avoid redundant edge extrusion. Simultaneously, DX SDK adopted the similar optimization. However, the computation load depending on trajectory length is still not eliminated.

It was found in the recent work by Schmid et al. [2010] that Monte Carlo sampling [Cook et al. 1984] is not very effective if the time of a trace is very short, compared with the motion effect's active period. To provide better solution to the aforementioned scenario, they propose a 4D data structure called TAO (Time Aggregate Object) by combining the object's geometry at certain instance into a single representation. In addition, the vertices of edges that define surfaces are inserted between adjacent time segments. In this structure, the accuracy of trace could scale with the number of TAO intersections. With volume based motion blur prototype, a 3D appeal is generated, and by intensive ray tracing, the resultant images are in high quality. However, the complexity of computation makes it difficult to realize real-time rendering.

3 Method and Principles

3.1 Split Trajectory Method

An important task of motion effect generation is to trace the motion trajectory. In traditional method, it is convenient to construct the whole trajectory, because the length of motion blur is often very short. However, in our purpose, we want to make it available to trace a long trajectory, in order to support fluid effect with fluid simulation, since the lifetime of a particle in fluid simulation often lasts quite long time. Thus, to trace the whole trajectory is a deal of consumption. To make things simple, we only focus on a segment of trajectory, which makes the base of Split Trajectory Method (STM).

In this section, we first give an overview of our approach named Split Trajectory Method employed to generate high quality real-time motion effect with a dramatic efficiency. The primary steps in the working procedure are as follows:

1) ***Figure Data Tracking:*** The motion data are generated by hierarchically traversing each bone from root to leaves along the tree structure of the model.

2) ***Skinning:*** The motion data on the skeleton are blended according to the weight in each vertex data structure.

3) ***Trajectory Segment Construction:*** By sampling and interpolating the vertex data transformed by the world transformation function of adjacent frames in the Geometry Shader, the trajectory segments are exported back to the memory buffer for reuse.

4) ***Trajectory Welding:*** All the trajectory segments are connected by rendering the corresponding segments in sequence of time. The joint of every two segments is sampled at the same time.

5) ***Motion Effect Solution:*** Various motion effects are generated based on the constructed trajectory. Here we mainly focus on motion blur and fluid tails.

6) ***Illumination:*** In the Pixel Shader (Fragment Shader), the solved trajectory is rendered for visualization.

7) ***Integration:*** The motion effect is merged into the scene rendered in the frame-buffer by depth analysis.

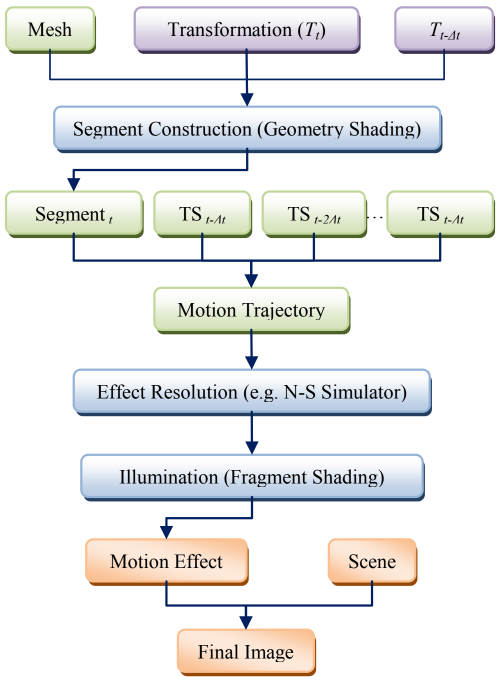


Figure 1: *Pipeline of STM.*

The whole procedure of our approach is shown in Figure 1.

Based on GPU computation, the key stage of our approach is the Geometry Shading, which generates motion trajectory. Conceptually, we split the whole trajectory into a number of segments, named Trajectory Segment (TS), instead of constructing the whole trajectory at once. Such a process reduces the complexity of motion tracking, accelerates the algorithm, and increases the flexibility of the trajectory for GPU computation and the further supports of fluid dynamics.

3.2 Trajectory Structure

3.2.1 Trajectory Representation

As is mentioned in Section 3.1, in order to avoid the complexity of handling the whole trajectory, we split the trajectory into Trajectory Segments, which is the part in the interval between two adjacent frames, and generate only one segment per frame. For the detailed structure, a Trajectory Segment consists of many points, called Trajectory Vertex. We only sample the motion data at the current $(t = 0)$ and the previous $(t - \Delta t)$ frames respectively as the boundary vertices of the segment.

A vertex location in motion state can be represented by a 4D vector (x, y, z, t), where $\vec{p}(x, y, z)$ is the spatial coordinate in modeling space, and t is the corresponding time. For any time t, $P(x, y, z, t)$ is denoted by:

$$P(x, y, z, t) = P(\vec{p}, t) = T(t)\vec{p} \tag{1}$$

where $T(t)$ is the world transformation at time t.

Thus, the position of a trajectory vertex is described by $P(x, y, z, t)$ and calculated. The normal vector is calculated similarly. Then, we define a trajectory vertex in the obtained position and fill the data from the Input Assembler, recording the texture coordinates, color with transparency, etc. As we obtained the two sets of boundary vertices transformed by $T(t)$ and $T(t-\Delta t)$ respectively, a trajectory segment is construct in the interval by joining boundary vertices.

3.2.2 Taxonomy of Trajectory Segment

In traditional applications, there are three basic types of geometry: point, line, and face. Our Trajectory Segments also have three categories of models for different applications.

For particle system, especially for some recently popular particle based simulation, e.g. Smoothed-Particle Hydrodynamics (SPH) [Müller et al. 2003], the trajectory segment can be constructed as points. In this model, vertex generation is the only necessity, and vertex connection is needless. After simulation, the points are extended to for visualization. Thus, we call this category Motion Particle shown in Figure 2.

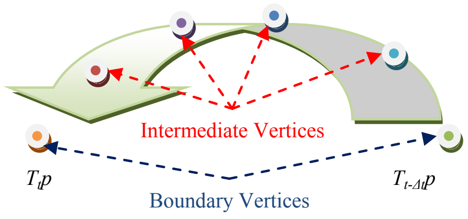


Figure 2: *Motion Particle Model.*

For velocity description, a line based model, namely Motion Line (Figure 3) is proper for speed-line. Only the vertices generated from the same original vertex are joined. Moreover, in cartoon rendering, speed-line is a typical abstract object, and in order to simulate the different sizes of brush footprints, lines can be extended to tubes in the next passes of rendering for more complex shading.

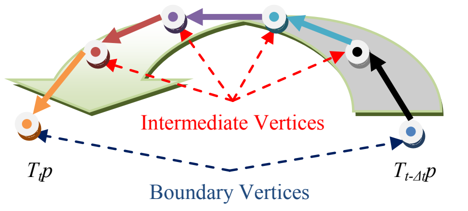


Figure 3: *Motion Line Model.*

As motion blur is often presented in the photo-realistic rendering of animation, it is not suitable only drawing several lines like speed-lines in cartoon, i.e. what is visualized should be full-filled geometry. Motion Particle is not a proper alternative either, in respect that sparsely particles cause hole-style noise in the final rendered image, while densely distributed ones cost more vertex buffer and computation.

Here we mainly use motion volume as an important structure to build motion blur and other high-density structure for more complex effects. The illumination model can also be enforced on the volume surfaces correctly in 3D space. Thus, we choose motion volume as the prototype of motion blur, even for advanced fluid effect.

Different from motion particle and motion line, for motion volume model, vertex data are input as triangle into the Geometry Shader. As is shown in Figure 4, the three vertices of the input triangle constitute a set of boundary vertices. Then, we interpolate between the two sets of boundary vertices, and obtain the intermediate vertices. Thus, each set of trajectory vertices encloses the cross-section of the volume.

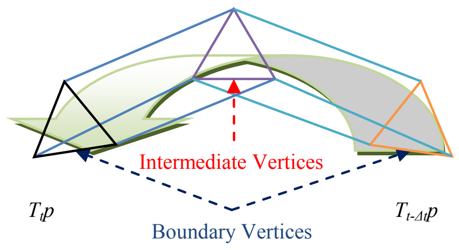


Figure 4: *Motion Volume Model.*

3.2.3 Smoothing Strategy

For real-time animation, the motion effect is significant as the object is in high-speed motion. In such a situation, sampling from each frame is sufficient to construct a good-looking motion effect for human vision. However, for a higher demand, a smoothing strategy is necessary. Hence, we utilize interpolation to generate new joints. Since the trajectory is split, the trajectory segments are independent, i.e. the adjacent data cannot be shared to provide at least four points for traditional cubic curve fitting. Therefore, we considered a strategy depending on the skinning algorithm.

In traditional skinning algorithm, linear interpolation is directly employed on the world transformation matrices. However, some movements of joint area should not be linear. When rotation transformations occur, linear operation on vertex cannot present the curvilinear movement. Nevertheless, we can utilize Dual Quaternion Linear Blending [Kavan et al. 2007] to solve such problems. A dual quaternion is a pair of quaternions shown as following, in which the first row denotes the rotation transformation and the second row denotes the translation transformation.

$$Q_t = [\vec{q}_{(R)}, \vec{q}_{(T)}]^T = \begin{bmatrix} q_{(R,x)} & q_{(R,y)} & q_{(R,z)} & q_{(R,w)} \\ q_{(T,x)} & q_{(T,y)} & q_{(T,z)} & q_{(T,w)} \end{bmatrix} \quad (2)$$

Since the transformation of skinning is already represented in quaternion form, for our smoothing method, we just directly employ linear interpolation between the dual quaternions, and then transform the original vertices with the interpolated dual quaternion $Q_{t,\tau}$, to obtain the intermediate trajectory vertices. The formula below presents the linear quaternion interpolation.

$$Q_t(\tau) = \begin{cases} (1-\tau)Q_t - \tau Q_{t-\Delta t}, & \text{if } \vec{q}_{t(R)} \cdot \vec{q}_{t-\Delta t(R)} < 0 \\ (1-\tau)Q_t + \tau Q_{t-\Delta t}, & \text{otherwise} \end{cases} \quad (3)$$

where τ is the interpolation parameter from time t to $t-\Delta t$, denoted in interval $[0,1]$. Thus, q_t and $q_{t-\Delta t}$ are the world transformation at time t and at $t-\Delta t$ respectively, both in quaternion form.

3.3 Fluid Dynamics Support

One novel feature of our approach is supporting motion effect using fluid dynamics. Once the trajectory segment is constructed, the basic sketch of motion effect is instantiated as 3D mesh. As the trajectory segments constitute the prototype of motion effect, for further effect in visualization, we need solve the effect from initial trajectory to self-solved motion variety to express enhanced effect and fantasy. Our work emphasizes the effect with fluid trajectory. In this section, we will describe the techniques during solving and visualizing the fluid effect.

3.3.1 Physical Model

We use the fundamental physical model of Navier-Stokes (N-S) Equation [Pnueli and Gutfinger 1996] for fluid dynamics simulation:

$$\rho\left(\frac{\partial\vec{u}}{\partial t}+\vec{u}\cdot\nabla\vec{u}\right) = -\nabla p+\mu\nabla^2\vec{u}+\vec{F} \quad (4)$$

$$\nabla\cdot\vec{u} = 0 \quad (5)$$

where $\vec{u}$, p, ρ and $\vec{F}$ are velocity, pressure, density, and external force respectively. All the data values are measured in unit volume.

For our simulation, we do not consider the viscosity temporarily. Then (4)&(5) can be simplified in vector form as (6), where a is the acceleration in unit volume.

$$\frac{D\vec{u}}{Dt} = -\frac{\nabla p}{\rho}+\vec{a}_{external} \quad (6)$$

In order to solve the N-S equation, the process is divided into several steps:

1) ***Advection:*** we apply Semi-Lagrangian Advection for stable fluids [Stam 1999] to realize the particle motion according to the velocity. In GPU simulation, since we use fixed cells to represent particles, it is necessary to trace the data from the cell before advection.

2) ***Poisson Pressure:*** the pressure term can be transferred into the form of (7), which is a linear system. Then we have to solve the system and compute the pressure p.

3) ***External force:*** the acceleration affected by external force is pushed on the velocity. This fact depends on the motion trajectory, which will be discussed in Section 3.3.2.

4) ***Projection:*** since the pressure field has been solved, the acceleration provided by pressure difference is calculated by (8), and the new velocity now can be updated.

$$\nabla^2 p = \frac{\rho}{\Delta t}\nabla\cdot\vec{u} \quad (7)$$

$$\vec{a}_{pressure} = -\frac{\nabla p}{\rho} \quad (8)$$

3.3.2 Effect Solution & Interaction

We enforce the physical model discussed above into the effect solution system. As is shown in Figure 5, the process is iterative according to the current simulation time.

For the external data transmission to the solution system, we substitute the property of the segment as the impulse of fluid simulation system. The external force is calculated according to the segment of Motion Volume, and the pigment of the fluid is sampled by the color source of the segment. Instead of expressing the external force/accleration term in (4)&(6), we add the velocity and pigment directly as impulse. The impulse for velocity is expressed by (9)-(11).

$$\vec{v}_{prev} = \frac{T_t(\tau,\vec{p})-T_t(\tau+\delta\tau,\vec{p})}{\delta\tau} \quad (9)$$

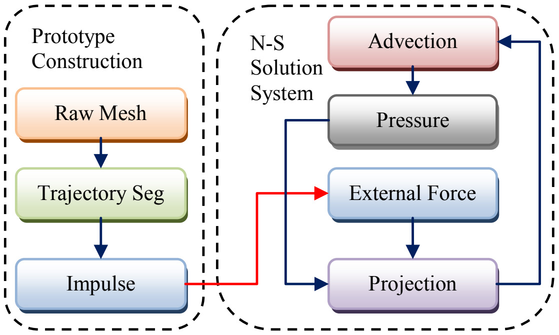


Figure 5: *The architecture of effect solution (N-S solution).*

$$\vec{v}_{post} = \frac{T_t(\tau-\delta\tau,\vec{p})-T_t(\tau,\vec{p})}{\delta\tau} \quad (10)$$

$$\vec{v} = k\frac{\vec{v}_{prev}+\vec{v}_{post}}{2} \quad (11)$$

where τ is the interpolation parameter from time t to $t-\Delta t$, and k is the coefficient for deducting energy loss. $\vec{p}$ and $T_t(\tau,\vec{p})$ denotes the original position of vertex and the world transformation function of τ at time t respectively.

4 Implementation

Our system is implemented in DirectX 10. For the construction of the geometric prototype of the motion effects, our Split Trajectory Method is deployed. Then, for advanced effect, we fetch the trajectory segment as the impulse, solve N-S equation with GPU to simulate the fluid dynamics, and shade the effect finally.

4.1 GPU Acceleration

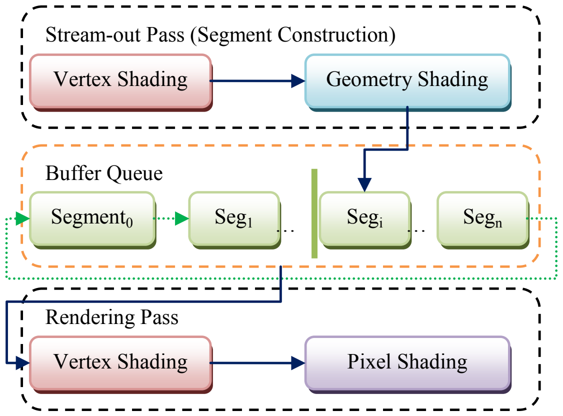


Figure 6: *Split Trajectory Method by GPU multi-pass rendering.*

In our real-time rendering pipeline in DirectX 10, we make use of multi-pass rendering. Especially, Geometry Shading is recommended to burden the computation for trajectory segment construction. Besides, we also have a function accompanied by the Geometry Shader, which is named Stream Output (named Transformed Feedback in OpenGL 3.0). Stream-out is able to store the primitives generated by the Geometry Shader back to the memory buffer.

With our Split Trajectory Method, first, a queue of buffers is prepared for fetching the generated segments. Then, we construct a trajectory segment in the Geometry Shader, and then stream-out the segment to a memory buffer, which is dequeued for the segment out of lifetime, and enqueued for the new segment. This method does not only accelerate the trajectory construction especially with skinning, but also break the limitation of the Geometry Shading. Figure 6 simply expresses the rendering pipeline for our implementation.

4.2 Figure Animation

In our work, the character is animated with skeletal animation. In order to enhance the skinning quality, as is mentioned in Section 3.2.3, we choose the dual quaternion algorithm for skinning. After loading two sets of matrices at the current frame time t and the previous frame time $t - \tau t$, we transform them into dual quaternion, and then skin the quaternion using (12) for weight blending and (13)-(15) for a fast vertex transformation [Kavan et al. 2008].

$$Q_t = \sum_{i=0}^{n} k_i w_i q_i, \quad k = \begin{cases} -1, & \text{if } q_{0(R)} \cdot q_{i(R)} < 0 \\ 1, & \text{otherwise} \end{cases} \tag{12}$$

$$\begin{aligned} R(\vec{p}) &= \vec{p} + 2[\vec{q}_{R,v} \times (\vec{q}_{R,v} \times \vec{p} + q_{R,w}\vec{p})] && (13) \\ T &= 2(q_{R,w}\vec{q}_{T,v} - q_{T,w}\vec{q}_{R,v} + \vec{q}_{R,v} \times \vec{q}_{T,v}) && (14) \\ \vec{p}_{world} &= R(\vec{p}) + T && (15) \end{aligned}$$

In (12), q_i and w_i denote the quaternion and the weight of the bone indexed by the input vertex. In DirectX 10, a vertex structure can hold four indices for corresponding bones, i.e. $n = 4$. After $Q_t(\vec{q}_v, q_w)$ is obtained, in which vector $\vec{q}_v(q_x, q_y, q_z)$ is the first three columns of Q, we calculate the final transformation function $\vec{p}_{world}$ for skinning.

4.3 Trajectory Segment Construction

We construct the trajectory segments in the Geometry Shader, while different types of segments are constructed similarly but varying a little, as expressed in the pseudo programs.

1) ***Motion Particle:***
Algorithm 1 shows the Motion Particle construction in GPU Geometry Shader. First, the original vertex data is cloned to the sample. Then, we interpolate the position of the vertex. In skinning function $Skin(V, t_i)$ and $skinN(V, t_i)$, V is the input vertex and t_i is the flag to identify the transformation of the current frame (cur) and the previous frame ($prev$). After the transparency is calculated by the attenuation function, we finally append the vertex to the geometry stream. In Motion Particle model, the primitive topology of input assembler is POINTLIST and the output stream format is point<S>.

Algorithm 1 Motion Particle: $GS\_MPMain(V, stream)$

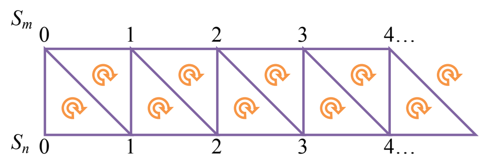
```
1: for each Sample $S$ at time parameter $\tau$ from 0 to 1 do
2:     $S \leftarrow V$   // Clone data from vertex $V$
3:     $S.postion \leftarrow (1-\tau)Skin(V, cur) + \tau * Skin(V, pre)$
4:     $S.color.\alpha \leftarrow Attentuation(\tau)$
5:     $stream.Append(S)$
6: end for
```

2) ***Motion Line:***
In Motion Line model, almost all the codes are as the same as Motion Particle model, except that the primitive topology is set to LINELIST and that the output stream format is line<S>.

3) ***Motion Volume:***
In order to construct the segment in Motion Volume model, we first calculate the trajectory vertices as similarly as Motion Particle (Algorithm 2). All the vertices are sorted into three arrays $S_k[]$ according to the original vertex $V[k]$ during generation. After vertex construction, the vertices are appended to output stream in order to generate triangle faces. Care must be taken that the order of vertex output is significant. Thus, we simply derive the organization of indices following from the example below (Figure 7). In addition, for Motion Volume, the primitive topology of input vertex is TRIANGLELIST and the stream format is triangle<S>. Algorithm 3 presents the process in GPU Geometry Shader.

Algorithm 2 Motion Volume: $XtrudeVertex(V, \tau)$

```
1: $S \leftarrow V$
2: $S.postion \leftarrow (1-\tau)Skin(V, cur) + \tau * Skin(V, pre)$
3: $S.normal \leftarrow (1-\tau)SkinN(V, cur) + \tau * SkinN(V, pre)$
4: $S.color.\alpha \leftarrow Attentuation(\tau)$
```



Index sequence: $S_n[0]\ S_m[0]\ S_n[1]$, $S_m[1]\ S_n[1]\ S_m[0]$,
$S_n[1]\ S_m[1]\ S_n[2]$, $S_m[2]\ S_n[2]\ S_m[1]$,
…
$S_n[i]\ S_m[i]\ S_n[i+1]$, $S_m[i+1]\ S_n[i+1]\ S_m[i]$

Figure 7: *Export triangles to geometry output stream.*

Algorithm 3 Motion Volume: $GS\_MVMain(V, stream)$

```
1: for each vertex $V[k] \in$ triangle $V[3]$ do
2:     for each vertex sample $S_k[i] \in$ array $S_k$ at step $\tau$ do
3:         $S_k[i] \leftarrow XtrudeVertex(V[k], \tau)$
4:     end for
5: end for
6: for each edge $\overline{V[m][n]} \in$ triangle $V[3]$ do
7:     for $i = 1 \rightarrow$ length of segment do
8:         $stream.Append(S_n[i])$
9:         $stream.Append(S_m[i])$
10:        $stream.Append(S_n[i+1])$
11:        $stream.Append(S_m[i+1])$
12:        $stream.Append(S_n[i+1])$
13:        $stream.Append(S_m[i])$
14:    end for
15: end for
```

4.4 Effect Solution & Illumination

In this section, we will discuss the implementation of shading the two types of motion effects separately.

4.4.1 Motion Blur & Speed Lines

Motion blur and speed-line do not concern physical simulation. Therefore, we skip solving the physical system of the effect, and illuminate the motion effect directly in one pass. In the previous pass, we have computed the position and the normal vector in world space, set the transparency by attenuation function, and copied the information of color and texture coordinate from the original vertex. In a new pass, after rasterization, the final color of the output pixels is computed in the Pixel Shader. During pixel shading, we enforce Phong Illumination Model [1975].

4.4.2 Fluid Effect

We implement the smoke effect with figure motion in our experiment, and a complex process of fluid solution is necessary for smoke. Here 16bit-float textures are used in to denote each vector field to compute the signed values directly. The detailed resource allocation is shown in Table 1.

Buffer	Format	Number of sets
Pigment	R8G8B8A8	2
Depth	R32F / D32F	2
Impulse	R16G16B16F	1
Velocity	R16G16B16F	2
Pressure	R16F	2

Table 1: *Buffer allocation for fluid simulation*

Then, we follow the steps below to implement the smoke effect in Pixel Shader:

1) ***Segment output:*** we output the pigment and the initial velocity according to the status of the constructed trajectory segment at once in the same rendering pass. The pigment and the velocity are accumulated to the pigment buffer and velocity buffer in corresponding format respectively.

2) ***Velocity advection:*** Using Semi-Lagrangian Advection as (16), we sample the velocity buffer, and trace the last position according to the obtained velocity. Then we sample the velocity buffer again with the source position. After that, the velocity in current voxel / pixel is updated with the velocity in the source position.

$$\vec{u}(\vec{x}(t+\Delta t), t+\Delta t) = \vec{u}(\vec{x}(t+\Delta t) - \vec{u}(x(t), t), t) \quad (16)$$

where $\vec{u}(\vec{x}, t)$ denotes the velocity of the particle located in position $\vec{x}$ at time t, and $\vec{x}(t)$ represents the position at time t.

3) ***Pigment advection:*** the pigments are updated as similar as the previous step, but operate on the pigment buffer. Besides, an attenuation function is employed on pigments simultaneously as well.

4) ***Divergence calculation:*** the divergence is calculated for solving pressure by sampling the adjacent texels.

5) ***Poisson Pressure:*** the pressure must be discretized from (7) as is mentioned in Section 3.3.1, so that we deploy Jacobi Iteration (17) [Golub and Van 1996] to solve the linear system by 7-8 cycles.

$$p^{\phi+1}_{i,j,k} = \frac{\sum_{i',j',k'}^{|i'-i|+|j'-j|+|k'-k|\leq 1} p^{\phi}_{i',j',k'} - \nabla\cdot\vec{u}}{8} \quad (17)$$

6) ***Projection:*** we transform the pressure differential of adjacent pressure into acceleration, and update the velocity buffer.

Moreover, we recommend off-screen rendering [Lorach 2007] for fluid tails in order to eliminate aliasing and hide the flaw of 3D transparent calculation. Hence, we first illuminate the fluid effect into a texture, namely fluid buffer. In order to obtain a smooth combination, in the Pixel Shader we merge the fluid buffer to frame-buffer with alpha blending by depth value as is shown in (18), instead of general depth testing.

$$z_{fade} = saturate\left(1 - \frac{D_{fluid} - D_{scene}}{b}\right) \quad (18)$$

where z_{fade} denotes the attenuation factor of alpha, D_{fluid} and D_{scene} are the depth of fluid buffer and frame-buffer respectively, b represent the buffer value. We clamp the result into interval $[0, 1]$ with $saturate()$. Finally, we multiply the alpha value by the value z_{fade} as the output value of pixel shading.

5 Results

In this section, we evaluate our method, and show the results of individual motion effects and an integrated demo with complex constructed scene obtained by our method. Our testing machine is equipped with Graphic Card Nvidia® Geforce GT240M and Intel® Core™2 Duo CPU P7450. We also prepare an attach video to animate our result.

5.1 Complexity Analysis & Evaluation

Our Split Trajectory Method is not only a geometric approach to generate motion effects with high quality, but also accelerate the construction of trajectory by utilizing the GPU parallel computation and pipeline design. In order to evaluate how much it speed up the motion effect by splitting the trajectory, we compare our result to the demo without trajectory splitting from DirectX SDK [Microsoft 2007]. In the demo, we keep all things by Microsoft, and add our approach into the codes (Figure 8).

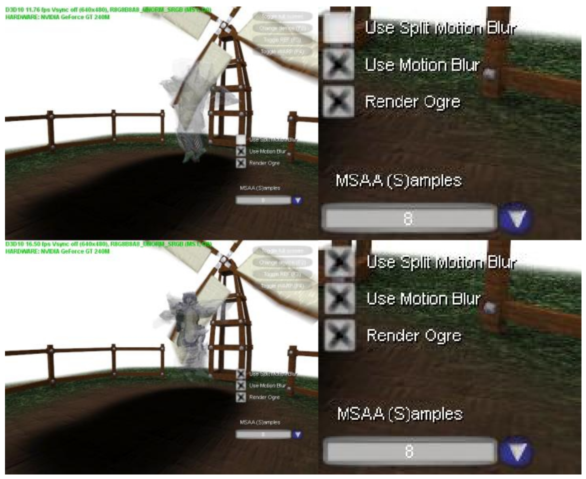


Figure 8: *DirectX SDK Sample Modification. Original sample (up, 11.76fps) and our STM (down, 16.50fps)..*

We also do the statistics for the skinning frequency, the number of vertices output from the Geometry Shader, and the rendering speed in the contradistinctive demo, depending on the trajectory lifetime (length of tracking).

Here we try to avoid the frequency of using skinning algorithm, due to the consumption of skinning. Figure 9 indicates that with the previous method, the frequency depends on the trajectory lifetime, while in our algorithm, to construct trajectory segment each frame, we only skin the character with constant frequency.

The number of vertices output by the Geometry Shader is limited due to the current standards of GPU. In our testing GPU, the output

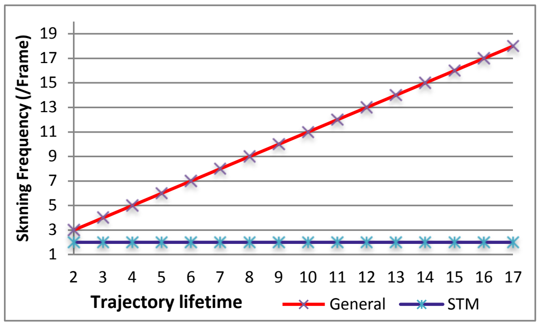


Figure 9: *Frequency of using skinning per frame.*

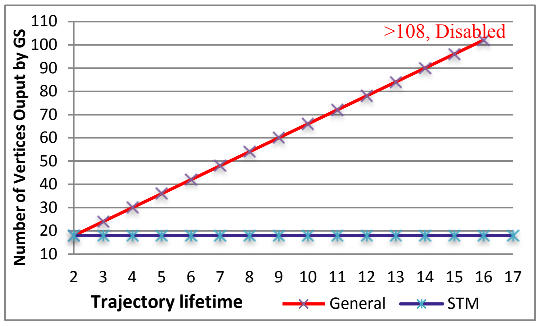


Figure 10: *Number of Vertices output by the Geometry Shader. It is disabled to generate trajectory more than 108 vertices, due to the limitation of the Geometry Shader on our GPU.*

vertex count multiplied by the total number of scalar component of output data cannot be greater than 1024. From Figure 10, it is obvious that when the trajectory lifetime increases greater than 16, it is out of limitation and disabled without STM, while our method is lifetime-independent. Therefore, our method can generate a long trajectory, thus making it possible to support fluid simulation.

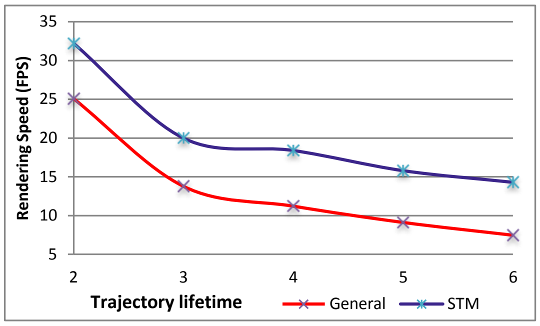


Figure 11: *Rendering speed (FPS, character only).*

Finally, the FPS is counted for both demos (Figure 11), in which we only focus on the character animation, excluding other objects. It is persuasive that our approach indeed faster than the general trajectory generation method using the Geometry Shader, due to the fact that we reduce the frequency of using highly-consumptive algorithm.

5.2 Examples

We first present the individual samples of our result. Here we show the speed-line and motion blur tested on a running character as shown in Figure 12. To express the effect obviously, a mild attenuation function is applied to the motion volume, thus the results are exaggerated. It is obvious that the motion effects are abundant in spatial perception and flexible to satisfy the different demands of clarity for different effects.

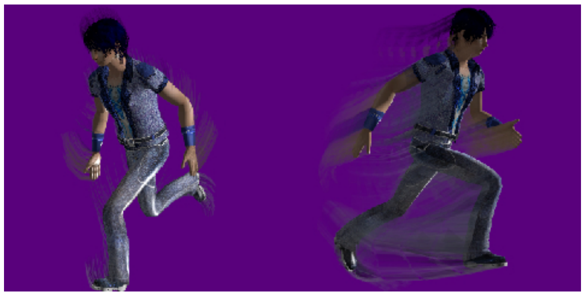


Figure 12: *Results of speed-lines (left) and motion blur (right).*

What's more, we modeled an integrated scene with our motion effects in a complex scale. The integrated example simulates a situation with many techniques commonly used in current games, including shadow mapping, dynamic water reflection, HDR etc. The whole scene including trees and bamboos are all modeled in 3D, with a relatively complex environment. Figure 13 shows an image of our demo result, picked up from a demo video given in the attached file.

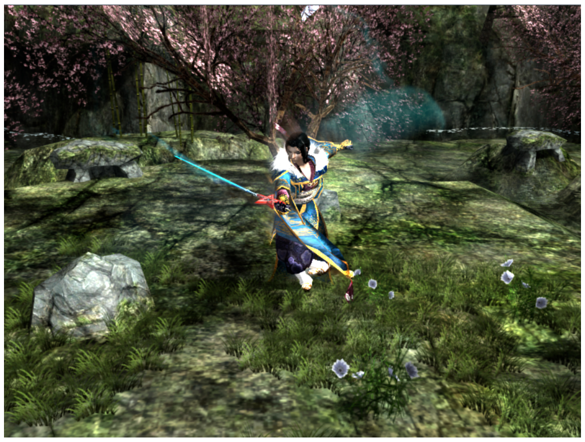


Figure 13: *Result of integrated scene.*

The model statistics is as given in the Table 2. Testing with such environment shows that the algorithm and method proposed is feasible for a general gaming environment in real time operations.

Name	Vertices	Faces	FPS
Character	20625	14397	-
Scene	13483	10521	-
Sword	83	161	-
Total	34191	20579	22.4 - 30.7

Table 2: *Original model statistics for the integrated scene*

Figure 14 displays some details of our result. It is clear that the vivid motion blur is partly hidden and partly visible surrounded by the atmosphere of smoke. Here we mainly focus on the smoke. As is shown, since the motion and the propagation of smoke are computed obeying the physical model (N-S equation), the perception of diffuse and the spinning vortices are sufficiently expressed accompanied with the motion of the character.

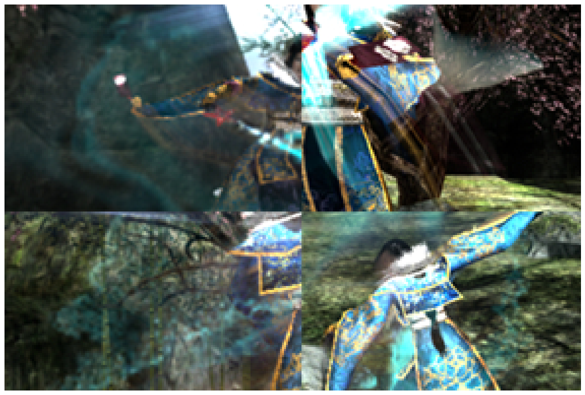


Figure 14: *Zoomed details of motion blur with smoke.*

6 Conclusion

In this paper, we present an approach to generate the motion effects for real-time figure animations. In order to provide motion effects, we explored methods of GPU computation and multi-pass geometry rendering to reuse the trajectory segment data, thus accelerating the rendering process and making the motion effects with long trajectories possible, especially for skeletal animation with complex skinning algorithm. Our approach is based on geometric shading, which presents more spaciousness than that using image processing. We implemented some commonly used motion effects based on the geometric structure, making use of GPU computation for full rendering. Meanwhile, we especially have enhanced the motion effect to support fluid dynamics. Our method can be easily incorporated into many existing real-time animation systems and simplified physical simulation systems, yielding the benefits of both efficient rendering and realistic result production.

Our future work is to enforce our method to much larger scale scenes with optimization. Another necessary improvement is to enhance the fluid effect with more precise simulation. Besides, we also intend to have our approach to support other physical simulation method in Larangian space, such as SPH and Lattice Boltzmann Method, for reducing the complication of Poisson solution and acquiring more advanced fluid effect and physical results.

Acknowledgements

The authors would like to thank the associate editor and all the reviewers for their constructive comments to improve the manuscript, and the work is supported by the Science and Technology Fundamental 973 program of China (2009CB320802), NSFC (60833007) and the grant of University of Macau.

References

BROSTOW, G. J., AND ESSA, I. 2001. Image-based motion blur for stop motion animation. In *Proceedings of SIGGRAPH 2001*, ACM Press / ACM SIGGRAPH, New York, E. Fiume, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 561–566.

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. In *Proceedings of SIGGRAPH 1984*, ACM Press / ACM SIGGRAPH, New York, H. Christiansen, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 137–145.

GOLUB, G. H., AND VAN, L. 1996. *Matrix Computations*. Johns Hopkins University Press.

HAEBERLI, P., AND AKELEY, K. 1990. The accumulation buffer: hardware support for high-quality rendering. In *Proceedings of SIGGRAPH 1990*, ACM Press / ACM SIGGRAPH, New York, Computer Graphics Proceedings, Annual Conference Series, ACM, 309–318.

HEINZLE, S., WOLF, J., KANAMORI, Y., WEYRICH, T., NISHITA, T., AND GROSS, M. 2010. Motion blur for ewa surface splatting. In *Proceedings of Eurographics 2010*, Eurographics Association and Blackwell Publishing Ltd., Computer Graphics Proceedings, Annual Conference Series, Eurographics Association, 733–742.

KAVAN, L., COLLINS, S., ŽRA, J., AND O'SULLIVAN, C. 2007. Skinning with dual quaternions. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, ACM Press / ACM SIGGRAPH, New York, Computer Graphics Proceedings, Annual Conference Series, ACM, 39–46.

KAVAN, L., COLLINS, S., ŽÁRA, J., AND O'SULLIVAN, C. 2008. Geometric skinning with approximate dual quaternion blending. *ACM Transactions on Graphics 27*, 4:105 (Oct.).

LORACH, T., 2007. Soft particles. NVIDIA DirectX 10 SDK, Jan.

MICROSOFT, 2007. Motionblur10. Microsoft DirectX SDK.

MÜLLER, M., CHARYPAR, D., AND GROSS, M. 2003. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH / Eurographics symposium on Computer animation*, ACM Press / ACM SIGGRAPH, New York, Computer Graphics Proceedings, Annual Conference Series, ACM, 154–159.

PHONG, B. T. 1975. Illumination for computer generated pictures. *ACM Transactions on Graphics 18*, 6 (June), 311–317.

PNUELI, D., AND GUTFINGER, C. 1996. *Fluid Mechanics*. Cambridge University Press.

SANDER, P. V., NEHAB, D., CHLAMTAC, E., AND HOPPE, H. 2008. Efficient traversal of mesh edges using adjacency primitives. *ACM Transactions on Graphics 27*, 5:144 (Dec.).

SCHMID, J., SUMNER, R. W., BOWLES, H., AND GROSS, M. 2010. Programmable motion effects. *ACM Transactions on Graphics 29*, 4:57 (July).

STAM, J. 1999. Stable fluids. In *Proceedings of SIGGRAPH 1999*, ACM Press / ACM SIGGRAPH, New York, Computer Graphics Proceedings, Annual Conference Series, ACM, 121–128.