Real-time Character Motion Effect Enhancement Based on Fluid Simulation

Tianchen Xu [1], Enhua Wu [1, 2], Mo Chen [1] and 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 of crucial importance, and this is especially true when an artist wants to generate exaggerating effect through figure motion. For a quite long period of time, animators seek the answer by using certain kind of image blending, no matter by the means of hardware or software. In recent years, methods based on 3D geometry of the motion figure with global illumination become gradually in demand, as they could deliver relatively high quality of motion blur effect. However, the computation cost in those methods is always very high, thus real time rendering become quite difficult to achieve.**

In this paper, a real-time motion effect based on 3D geometric approach is proposed, in which a special effect along the motion trajectory based on fluid simulation is combined with the volumetric motion blur. Furthermore, the motion trajectory would be decomposed and multi-pass geometry rendering would be employed to achieve geometry instancing for reuse. In this manner, the redundant calculation of each frame could be avoided, and the limitation of trajectory generation would be broken. In the pipeline, we separate motion tracking and fluid solution, to support various fluid effects flexibly. The scheme we present makes use of GPU geometry shading in parallel, aiming at guaranteeing high efficiency of computation while delivering splendid rendering. As a result, real time rendering including the motion blur effect is achieved.

***Index Terms*—motion blurs, skeletal animation, fluid dynamics, GPU geometric processing.**

I. INTRODUCTION

Motion blur is able to realize various effects in live-action footage: it can concentrate viewers' attention to certain area that the animator wants to emphasize, by blurring its background or surrounding, it can express the extreme speed of a moving object, and it can deliver the idea of vibration of an environment or the dizziness one character feels. With proper design, motion blur can provide motion figure animation with highly convincing details. With the help of GPU computation, the trails of moving objects can be analyzed to provide better continuity and smoothness.

Adding realism to high-speed scenarios is the most common purpose among all the motion blur techniques. Originally, it was realized by merging individual frame with its prior frame, instead of simply inserting data into adjacent frames. In using the most popular tools of DirectX 9 and OpenGL 2.0, the outcomes are generally not satisfying: the moving objects and their background are often mixed together, leading to the result of image quality downgrading, with quite low efficiency. Due to the aforementioned reasons, the employ of motion blur is still limited.

A novel scheme of real-time motion effects based on geometry volume will be presented in our paper, which shows motion blur with well quality of illumination in 3D space, by making use of up-to-date GPU capability. To avoid the duplicated calculation of individual 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: first, related works on the topics concerning our study interest will be briefly introduced in the Section 2. After that, main principles of our method, along with descriptions of the overall algorithms will be given in Section 3. Following with that are some key solutions for implementation and the testing results in varying complexity, which will be presented in Section 4 & 5. Finally, we conclude in Section 6 with some discussions of the future work.

II. RELATED WORK

In the early days, motion blur is often achieved with the help of OpenGL accumulation buffer. Haeberli and Akeley [1] provided in-depth analysis towards the use of this architecture. Using ray tracing to perform the Monte Carlo evaluation of integrals [2] 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 [3] present a post-process 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 that image-based motion blur can only provide the

Manuscript Received on September 1, 2011
E-mail: x-universe@live.com

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. [4]. 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" [5] in DirectX 2007, a geometric approach came into the world with the birth of Geometry Shader in Shader Model 4. Based on this method, Sander et al. (Hong Kong UST, Microsoft Research, and Princeton University) [6] 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. [7] that Monte Carlo sampling [2] 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.

In addition, many of methods above only consider the motion blur effect, which has less-level requirement for long-trajectory motion tracking. However, an important motion effect we want to realize is the effects based on fluid dynamics, thus the requirement of tracking length is much higher than the common motion blur. Therefore, image-based methods have drawbacks on geometric flexibility, and traditional geometry-based tracking has the problem of efficiency for tracking long trajectories.

III. METHOD & 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) [8].

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:

- *Character Data Tracking:* The motion data are generated by hierarchically traversing each bone from root to leaves along the tree structure of the model.
- *Skinning:* The motion data on the skeleton are blended according to the weight in each vertex data structure.
- *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.
- *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.
- *Motion Effect Solution:* Various motion effects are generated based on the constructed trajectory. Here we mainly focus on motion blur and fluid tails.
- *Illumination:* In the Pixel Shader (Fragment Shader), the solved trajectory is rendered for visualization.
- *Integration:* The motion effect is merged into the scene rendered in the frame-buffer by depth analysis.

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

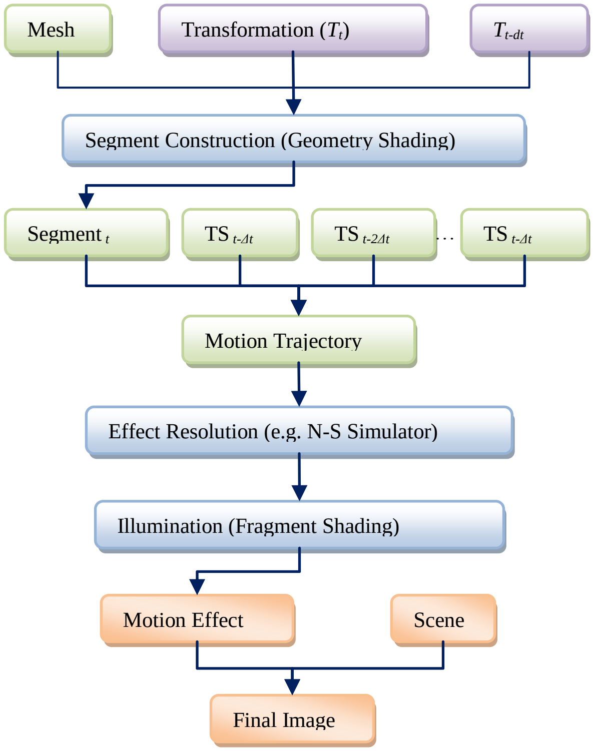

Mesh
Transformation (T_t)
$T_{t\text{-}dt}$
Segment Construction (Geometry Shading)
Segment $_t$
TS $_{t\text{-}\Delta t}$
TS $_{t\text{-}2\Delta t}$
…
TS $_{t\text{-}\Delta t}$
Motion Trajectory
Effect Resolution (e.g. N-S Simulator)
Illumination (Fragment Shading)
Motion Effect
Scene
Final Image

Fig. 1. Pipeline of Split Trajectory Method.

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 Denotation.

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 - Δt) 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 ***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:

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

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 - Δt*) 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) [9], 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 Fig. 2. We can also process the mesh, and make the vertex distribution homogeneous, so that motion particles can be applied to realize some advanced fluid effect.

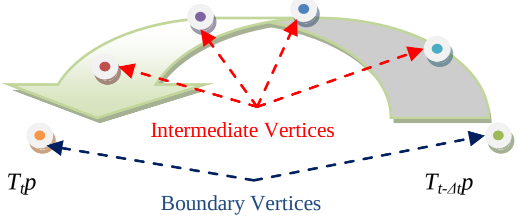


Fig. 2. Motion particle model.

For velocity description, a line based model, namely Motion Line (Fig. 3) is proper for speed-line. Only the vertices generated from the same original vertex are joined. Moreover, in cartoon rendering (Fig. 4 [10]), 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.

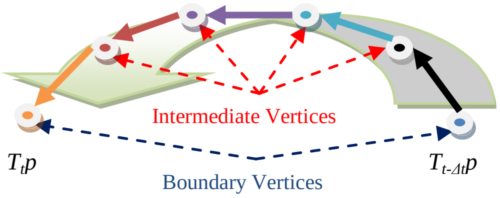


Fig. 3. Motion line model.

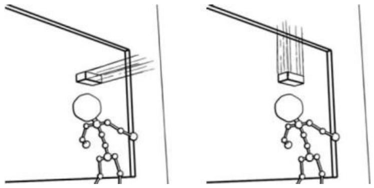


Fig. 4. Speed-lines can describe the direction of motion in cartoon by Maic Masuch.

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. 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.

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 Fig. 5, 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.

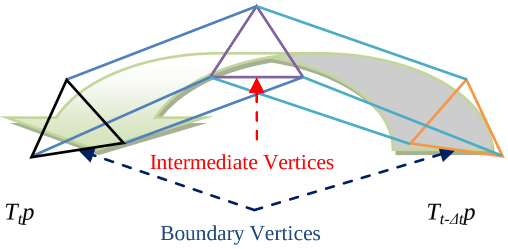


Fig. 5. Motion volume model.

3.2.3. Trajectory Smoothing.

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 [11] 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.

$$Q = \begin{bmatrix} q_{(R,x)} & q_{(R,y)} & q_{(R,z)} & q_{(R,w)} \\ q_{(T,x)} & q_{(T,y)} & q_{(T,z)} & q_{(T,w)} \end{bmatrix} \tag{2}$$

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.

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

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

3.3 Principles of Fluid Simulation

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 structure. 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 [12] for fluid dynamics simulation:

$$\rho\left(\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u}\right) = -\nabla p + \mu \nabla^2 \vec{u} + \vec{F} \tag{4}$$
$$\nabla \cdot \vec{u} = 0$$

where ***u***, *p*, *ρ* and ***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) can be simplified in vector form as (5), where ***a*** is the acceleration in unit volume.

$$\frac{D\vec{u}}{Dt} = -\frac{\nabla p}{\rho} + \vec{a}_{external} \tag{5}$$

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

- *Advection:* we apply Semi-Lagrangian Advection for stable fluids [14] 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.
- *Poisson Pressure:* the pressure term can be transferred into the form of (6), which is a linear system. Then we have to solve the system and compute the pressure *p*.
-

$$\nabla^2 p = -\frac{\rho}{\Delta t}\nabla \cdot \Delta \vec{u} = \frac{\rho}{\Delta t}\nabla \cdot \vec{u}_{t-\Delta t} \tag{6}$$

- *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.
- *Projection:* since the pressure field has been solved, the acceleration provided by pressure difference is calculated by (7), and the new velocity now can be updated.
-

$$\vec{a}_{press} = -\frac{\nabla p}{\rho} \tag{7}$$

3.3.2. Fluid – Character Interaction.

We enforce the physical model discussed above into the effect solution system. As is shown in Fig. 6, the process is iterative according to the current simulation time.

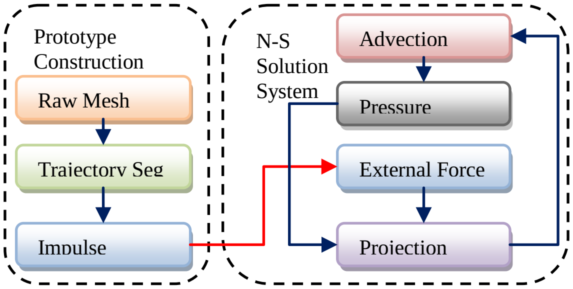


Fig. 6. Fluid simulation cycle.

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/acceleration term in (4) & (6), we add the velocity and pigment directly as impulse. The impulse for velocity is expressed by (9).

$$\begin{aligned} \vec{v}_{prev} &= \frac{T_t\left(\tau,\vec{p}\right)-T_t\left(\tau+\Delta\tau,\vec{p}\right)}{\Delta\tau} \\ \vec{v}_{post} &= \frac{T_t\left(\tau-\Delta\tau,\vec{p}\right)-T_t\left(\tau,\vec{p}\right)}{\Delta\tau} \\ \vec{v} &= k\,\frac{\vec{v}_{prev}+\vec{v}_{post}}{2} \end{aligned} \tag{9}$$

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

3.3.3. Fluid Initialization Problem. Except for the physical information, the source of pigments is another significant material we must provide. Here, we use motion particle model to emit particles as voxels. However, as in many meshes, the vertices are not equally distributed. Only applying the motion particle generation makes sparse results. Here, one choice is making use of the UV information. Since UV coordinates in many models are well distributed basically, we can render the vertex information into UV space as an information buffer.

However, a condition must be satisfied that the UV element is a unique parameterization of the mesh; otherwise, the distinct vertices will be record into the same pixel, leading to data loss. Nevertheless, in many applications, for some symmetrical mesh, artists may assign shared area of UV atlas to save texture size. One solution is to remapping UV by some effective UV unwrapping algorithm, such as Least Squares Conformal Maps (LSCM) [14] and Angle Based Flattening (ABF) [15], but it greatly increases the computation of preprocessing.

For such intractable cases, we also proposed a simple process to reallocate particles without rendering vertices in the global UV space. We emulate rasterization in triangle space with GPU Geometry Shader (DX10/11) or Compute Shader (DX11), due to the consideration of efficiency even for the offline preprocessing. As is shown in Fig. 7, for an arbitrary triangle primitive, we conceptually transform it into UV space. Our purpose is to select out the area inside the triangle (back points), compute the positions in modeling space, and generate vertices. To reduce the calculation of texel search, we first restrict the area of UV using (10).

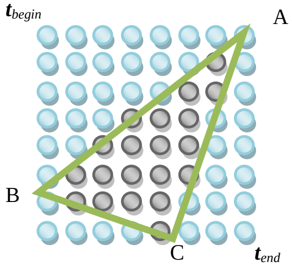


Fig. 7. An arbitrary triangle in UV space.

$$\begin{aligned} \vec{t}_{begin} &= \min\left(t_A, t_B, t_C\right) \\ \vec{t}_{end} &= \max\left(t_A, t_B, t_C\right) \end{aligned} \tag{10}$$

where $\boldsymbol{t}_{begin}$ and $\boldsymbol{t}_{end}$ are the vector to define the boundary rectangle of a triangle in UV space. t_A, t_B, and t_C are the texture coordinates of vertex A, B and C respectively.

Similarly to Scan-line Rendering algorithm [16], with the boundary defined, we scan the area row by row with a user-given step to calculate the position of each vertex in modeling space, so that the number of vertices generated is guaranteed to be controllable. We employ bilinear interpolation to obtain the position of the vertex attached on the surface of a triangle face. The algorithm is demonstrated in Fig. 8.

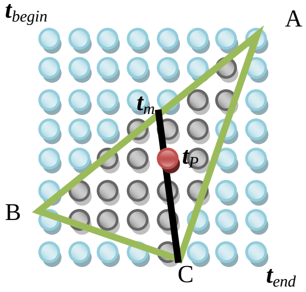


Fig. 8. Construct the bilinear interpolation equation.

For mapping the position of a certain vertex P in modeling space (red point) from the texture coordination, we construct a set of parameter equations shown as following.

$$\begin{aligned} \vec{t}_m &= \left(1-\alpha\right)\vec{t}_A + \alpha\vec{t}_B \\ \vec{t}_p &= \left(1-\beta\right)\vec{t}_m + \beta\vec{t}_C \end{aligned} \tag{11}$$

where α and β are the interpolation parameter respectively, and $\boldsymbol{t}_p$ is the texture coordinate of the corresponding vertex, generated during scanning from $\boldsymbol{t}_{begin}$ and $\boldsymbol{t}_{end}$. Besides, $\boldsymbol{t}_m$ is the intermediate value. All the UV vectors are in form $\boldsymbol{t}_i(u, v)$.

We solve the equation, thus obtaining the parameter α and β:

$$\begin{aligned} \alpha &= \frac{\left(u_p-u_A\right)v_C+\left(u_C-u_p\right)v_A+\left(u_A-u_C\right)v_p}{\left(u_B-u_A\right)v_C+\left(u_A-u_C\right)v_B+\left(u_C-u_p\right)v_A+\left(u_A-u_B\right)v_p} \\ \beta &= \frac{\left(u_A-u_p\right)v_B+\left(u_p-u_B\right)v_A+\left(u_B-u_A\right)v_p}{\left(u_B-u_A\right)v_C+\left(u_A-u_C\right)v_B+\left(u_C-u_B\right)v_A} \end{aligned} \tag{12}$$

where u_i and v_i are the components of vector $\boldsymbol{t}_i(u, v)$.

Care must be taken that for the area inside the triangle, the value of α and β are distributed in the interval [0, 1]. Vice versa, for either parameter out of the range [0, 1], it indicates that the vertex is located outside the triangle (blue points), which should be discarded.

Finally we calculate the position $\boldsymbol{p}(x, y, z)$ in modeling space by substitute the solved α and β into the bilinear interpolation equation (13) on the positions of Vertex A, B and C, which enclose the triangle.

$$\begin{aligned} \vec{p}_m &= \left(1-\alpha\right)\vec{p}_A + \alpha\,\vec{p}_B \\ \vec{p}_p &= \left(1-\beta\right)\vec{p}_m + \beta\,\vec{p}_C \end{aligned} \tag{13}$$

where $\boldsymbol{p}_A$, $\boldsymbol{p}_B$, and $\boldsymbol{p}_C$ are the position of vertex A, B and C in modeling space respectively. All the position vectors are in form $\boldsymbol{p}_i(x, y, z)$.

Therefore, using the algorithm above, a set of vertices (black points) is generated, which satisfies isotropic distribution on each triangle face. Then, we can emit the particles, and rendering them into pigment buffer, so that the rendering material of fluid effect is prepared.

IV. 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

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 de-queued for the segment out of lifetime, and en-queued 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. Fig. 9 simply expresses the rendering pipeline for our implementation.

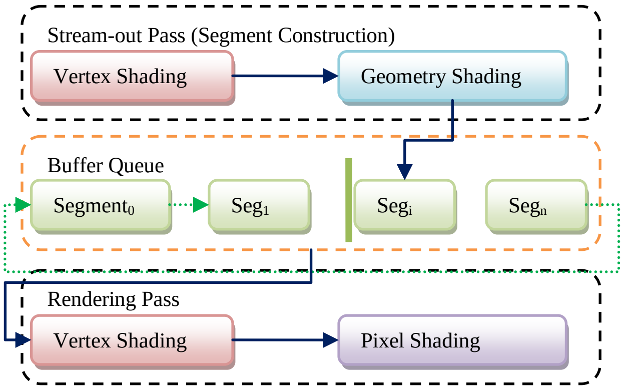


Fig. 9. Split Trajectory Method by GPU multi-pass rendering.

4.2 Character 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 - Δt, we transform them into dual quaternion, and then skin the quaternion using (14) for weight blending and (15) for a fast vertex transformation [14].

$$Q = \sum_{i=0}^{n-1} k_i w_i q_i \quad k_i = \begin{cases} -1 & \left(q_{0(R)} \bullet q_{i(R)} < 0\right) \\ 1 & \left(otherwise\right) \end{cases} \tag{14}$$

$$\begin{aligned} R(\vec{p}) &= \vec{p} + 2\left(\vec{q}_{R,v} \times \left(\vec{q}_{R,v} \times \vec{p} + q_{R,w}\vec{p}\right)\right) \\ T &= 2\ \left(\mathrm{q}_{\mathrm{R,w}}\ \vec{\mathrm{q}}_{\mathrm{T,v}} \text{-} \mathrm{q}_{\mathrm{T,w}}\vec{\mathrm{q}}_{\mathrm{R,v}}\ + \vec{\mathrm{q}}_{\mathrm{R,v}} \times \vec{\mathrm{q}}_{\mathrm{T,v}}\ \right) \\ \vec{p}_{world} &= R(\vec{p}) + T) \end{aligned} \tag{15}$$

Since the animation data of many models exported from the modeling software is represented in the matrices, we have to transfer the matrices into dual quaternion. As many transformation concerns the scaling, thus we follow Xu's suggestion [18], so that the whole skinning workflow is below:

- Split the world transformation matrix into translation, rotation and scaling matrix.
- Transfer the translation & rotation matrix into quaternions; extract the scaling matrix into scaling vector $\boldsymbol{S}(x, y, z)$
- Rotation quaternion is stored in the first row of DQ; translation quaternion is stored in the second row; scaling vector $\boldsymbol{S}$ is stored in the third row.
- Blend DQ using (14), and blend scaling vector using k = 1 in (14).
- Transform vertex position using scaling factor $\boldsymbol{S}$.
- Transform normal using 1/$\boldsymbol{S}$.
- Transform scaled vertex using (15).

4.3 Trajectory Segment Generation

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.

a) *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) for position and SkinN(V, t_i) for the normal vectors, V is the input vertex and t_i is the flag to identify the transformation of the current frame (cur) and the previous frame (pre). 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*)

```
for each Sample $S$ at time parameter $\tau$ from 0 to 1 do
    $S \leftarrow V$ // Clone data from vertex V
    $S.postion \leftarrow (1 - \tau)\mathrm{Skin}(V, cur) + \tau\mathrm{Skin}(V, pre)$
    $S.color.\alpha \leftarrow \mathrm{Attentuation}(\tau)$
    $stream.\mathrm{Append}(S)$
end for
```

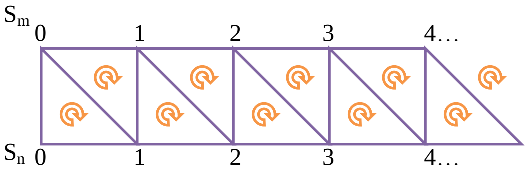
2) *Motion Line:* In Motion Line model, almost all the codes

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 (Fig. 10). 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: ExtrudeVertex(V, τ)

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



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

Fig. 10. Export extruded triangles to geometry output stream.

Algorithm 3 Motion Volume: GS_MVMain(V, *stream*)

```
for each vertex $V[k]$ $\in$ triangle $V[3]$ do
    for each vertex sample $S_k[i]$ $\in$ array $S_k$ at $\tau$ do
        $S_k[i] \leftarrow$ ExtrudeVertex($V[k]$, $\tau$)
    end for
end for
for each edge $V[m][n]$ $\in$ triangle $V[3]$ do
    for $i \leftarrow 1 \rightarrow$ length of segment do
        stream.Append($S_n[i]$)
        stream.Append($S_m[i]$)
        stream.Append($S_n[i+1]$)
        stream.Append($S_m[i+1]$)
        stream.Append($S_n[i+1]$)
        stream.Append($S_m[i]$)
    end for
end for
```

4.4 Simulation & Rendering

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

4.4.1. Motion Blurs & 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 [19].

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.

TABLE 1: BUFFER ALLOCATION, FOR SOME BUFFERS, TWO SETS ARE NECESSARY FOR READ/WRITE SWAPPING

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

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.

$$\vec{u}\left(\vec{x}_{t+\Delta t}, t+\Delta t\right) = \vec{u}\left(\vec{x}_{t+\Delta t} - \vec{u}\left(\vec{x}_t, t\right), t\right) \tag{16}$$

where $\boldsymbol{u}(\boldsymbol{x}, t)$ denotes the velocity of the particle located in position $\boldsymbol{x}$ at time t, and $\boldsymbol{x}_t$ 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 (6) as is mentioned in Section 3.3.1, so that we deploy Jacobi Iteration (17) [20] to solve the linear system by 7-8 cycles.

$$p_{i,j,k}^{\phi+1} = \frac{\sum_{i',j',k'}^{|i'-i|+|j'-j|+|k'-k|\le 1} p_{i',j',k'}^{\phi} - \nabla\cdot\vec{u}}{8} \tag{17}$$

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

Moreover, we recommend off-screen rendering [21] 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.

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

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.

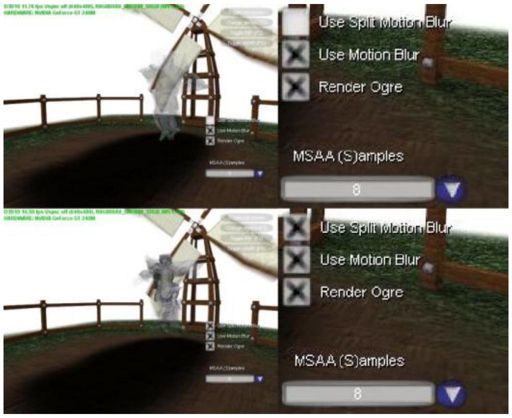
V. RESULTS

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® CoreTM2 Duo CPU P7450. We also prepare an attach video to animate our result.

5.1 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 (

Fig. 11).



□ Use Split Motion Blur
☒ Use Motion Blur
☒ Render Ogre
MSAA (S)amples
8

☒ Use Split Motion Blur
☒ Use Motion Blur
☒ Render Ogre
MSAA (S)amples
8

Fig. 11. DirectX SDK Sample Modification. Original sample (up, 11.76fps) and our STM (down, 16.50fps) (Color Plate 11).

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. Fig. 12 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.

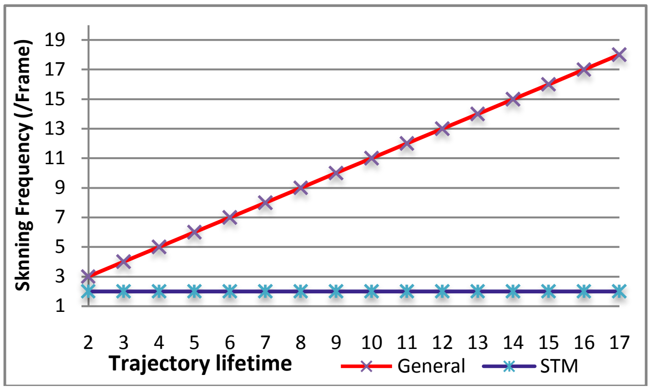


Fig. 12. Frequency of enforcing skinning per frame.

The number of vertices output by the Geometry Shader is limited due to the current standards of GPU. In our testing GPU, the output vertex count multiplied by the total number of scalar component of output data cannot be greater than 1024. From

Fig. 13, 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.

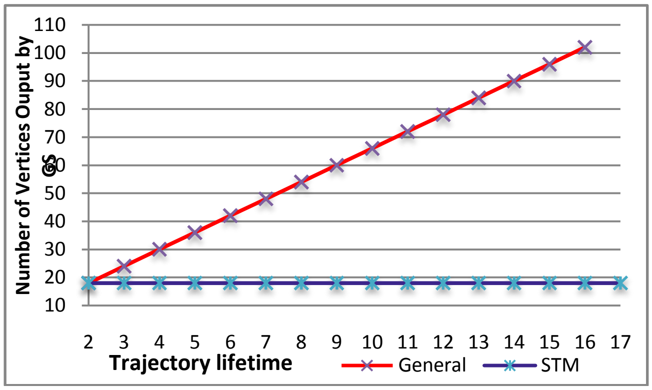


Fig. 13. Number of Vertices output by the Geometry Shader. It is disabled to generate trajectory more than 108 vertices, due to the limitation of the Geometry Shader on our GPU.

Finally, the FPS is counted for both demos (

Fig. 14), 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.

In a word, the Split Trajectory Method accelerates the pipeline macroscopically. It reduces the generation burden from the Geometry Shader, and avoids the calls of complex algorithms. It breaks the limitation of stream output, so it is made to be possible to construct a trajectory lasting for long time in preparation for the fluid simulation.

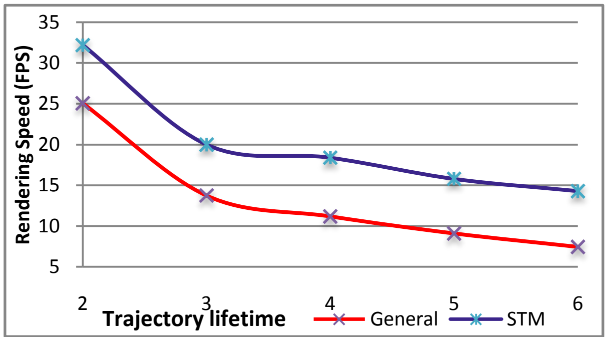


Fig. 14. Rendering speed (FPS, character only).

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

Fig. 15. 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. In the image of (a), the lines are clear and brief, which is appropriate for both static cartoon and animation. The image (b) shows that the motion blur is an area of blurring effect, which possess high density of pixels, thus the prototype Motion Volume is very suitable for it.

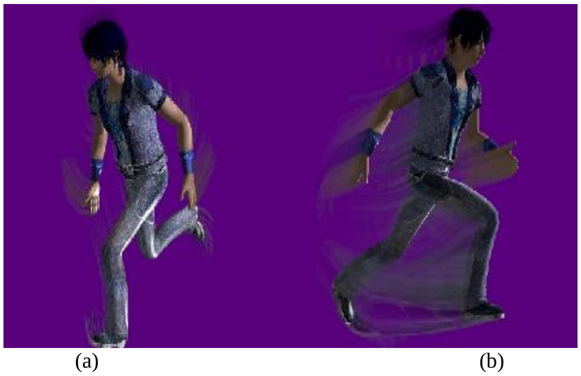


Fig. 15. Results of speed-lines (a) and motion blur (b).

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.

Fig. 16 shows an image of our demo result, picked up from a demo video given in the attached file. Besides, 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.

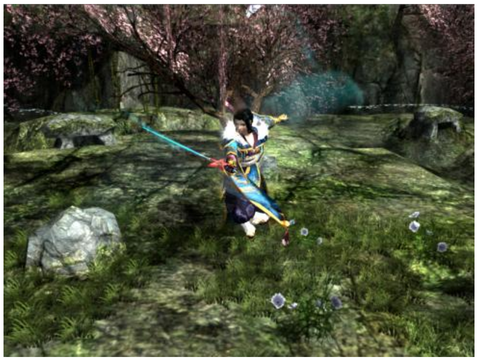


Fig. 16. Result of integrated scene(Color Plate 12).

TABLE 2: STATISTICAL TABLE FOR DATA AMOUNT IN THE SCENE OF THE DEMO

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

Furthermore, Fig. 17 specifies 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.

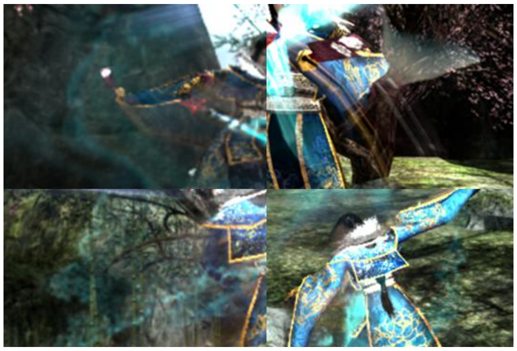


Fig. 17. Zoomed details of motion blurs with smoke.

VI. 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. Besides, more types of effects driven by physical simulation can be realized, such as water splash, fire, and explosion. We expect to attempt the interaction among different fluid effects, character motions, and environments in the scene (terrains)

ACKNOWLEDGEMENT

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

[1] P. Haeberli, and K. Akeley. The accumulation buffer: hardware support for high-quality rendering, in *Proc. SIGGRAPH 1990, ACM Press/ACM SIGGRAPH*, New York, pp. 309–318, 1990.

[2] R. L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing, in *Proc. SIGGRAPH 1984*, ACM Press/ACM SIGGRAPH, New York, pp. 137–145, 1984.

[3] G. J. Brostow, and I. Essa. Image-based motion blur for stop motion animation, in *Proc of SIGGRAPH 2001*, ACM Press/ACM SIGGRAPH, New York, pp. 561–566, 2001.

[4] S. Heinzle, J. Wolf, Y. Kanamori, T. Weyrich, T. Nishita, and M. Gross. Motion blur for ewa surface splatting, in *Proc. Eurographics 2010*, Eurographics Association, pp. 733–742, 2010.

[5] Microsoft. Motionblur10, Microsoft DirectX SDK, 2007.

[6] P. V. Sander, D. Nehab, E. Chlamtac, and H. Hoppe. Efficient traversal of mesh edges using adjacency primitives, *ACM Transactions on Graphics* (*Proc. SIGGRAPH Asia 2008),* vol. 27, no. 5, article 144, 2008.

[7] J. Schmid, R. W. Sumner, H. Bowles, and M. Gross. Programmable motion effects, *ACM Transactions on Graphics (Proc. SIGGRAPH 2010)*, vol. 29, no. 4, article 57, 2010.

[8] 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 of SIGGRAPH VRCAI 2011,* ACM Press, pp.307-314, 2011.

[9] M. M üller, D. Charypar, and M. Gross. Particlebased fluid simulation for interactive applications, in *Proc. 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, ACM Press/ACM SIGGRAPH, New York, pp. 154–159, 2003.

[10] T. Strothotte, and Schlechtweg. *Non-Photorealistic Computer Graphics: Modeling, Rendering and Animation*, Morgan Kaufmann, 2002

[11] L. Kavan, S. Collins, J. Zara, and C. O'Sullivan. Skinning with dual quaternions, in *Proc. 2007 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM Press, New York, pp. 39–46, 2007.

[12] D. Pnueli, and C. Gutfinger. *Fluid Mechanics*, Cambridge University Press, 1996

[13] J. Stam. Stable fluids, in *Proc. SIGGRAPH 1999*, ACM Press/ACM SIGGRAPH, New York, pp121-128, 1999.

[14] B. L évy, S. Petitjean, N. Ray, and J. Maillot. Least Squares Conformal Maps for Automatic Texture Atlas Generation, *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002)*, vol. 21, pp362-371, 2002.

[15] A. Sheffer, and E. De Sturler. Parameterization of Fac-eted Surfaces for Meshing using Angle Based Flattening, *Engi-neering with Computers*, vol. 17, no. 3 pp.326-337, 2001.

[16] C. Wylie, G. W. Romney, D. C. Evans, and A. Erdahl. Halftone Perspective Drawings by Computer. In *Proc. AFIPS FJCC*, vol. 31, pp.49-58, 1967.

[17] L. Kavan, S. Collins, J. Zara, and C. O'Sullivan. Geometric skinning with approximate dual quaternion blending, in *ACM Transactions on Graphics (TOG)*, vol. 27, No. 4, Art. 105, ACM Press, New York, 2008.

[18] T. C. Xu, M. Chen, M. Xie, and E. H. Wu. A Skinning Method in Real-time Skeletal Character Animation, *The International Journal of Virtual Reality*, IPI Press, vol. 10. no. 3. pp.25-31 , 2011.

[19] B. T. Phong. Illumination for computer generated pictures, in *Communications of ACM 18*, vol. 6, ACM Press, New York, pp. 311–317, 1975.

[20] G. H. Golub, and L. Van. *Matrix Computations*, Johns Hopkins University Press, 1996.

[21] T. Lorach. *Soft particles*, NVIDIA DirectX 10 SDK, Jan. 2007.

Tianchen Xu was born in Ningbo, China, in 1988. He received the B.S. degree in Software Engineering from University of Macau in summer of 2011. He was the research assistant in the Software Engineering Lab and Motion Capture Lab of University of Macau. Now he is working for the project on figure motion and simulations in the Motion Capture Lab of University of Macau supervised by Prof. Enhua Wu. His research interests include Real-time Animation & Rendering, Virtual Reality, and NPR.

Prof. Enhua Wu undergraduated from Tsinghua University in 1970, Beijing and received his Ph.D degree in 1984 from Department of Computer Science, University of Manchester, UK. He is now a full professor both in the University of Macau, Macao and the State Key Lab of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing. His main interests are Realistic Image Synthesis, Scientific Visualization and Virtual Reality.

Mo Chen was born in Fujian, China, in 1989. He received the B.S. degree in Software Engineering from University of Macau in summer of 2011. He has worked for the project on figure motion effects in the in the Motion Capture Lab of University of Macau supervised by Prof. Enhua Wu. His research interests mainly include Virtual Reality, Distributed System, and knowledge management.

Ming Xie was born in 1988. She received the B.S. degree in Software Engineering in summer of 2011 from University of Macau. She has worked for the project on figure motion effects in the Motion Capture Lab of University of Macau supervised by Prof. Enhua Wu. Her research interests focus on Computer Graphics, Web Design & Development, and marketing.