Real-time Rain Simulation in Cartoon Style

Zhong-Xin Feng[1], Min Tang[1], Jin-Xiang Dong[1], Shang-Ching Chou[2]
[1]*College of Computer Science and Technology, Zhejiang University, Hangzhou, P.R.China, 310027*
[2]*Department of Computer Science, Wichita State University, Wichita, KS, USA, 67260-0083*
feng_zx@hotmail.com tang_m@zju.edu.cn djx@zju.edu.cn chou@cs.wichita.edu

Abstract

An efficient method for simulating cartoon style rain in 3D environment is proposed here. By taking advantage of the parallelism and programmability of GPUs (Graphic Processing Units), real-time interaction can be achieved. Splashing of raindrop is simulated using collision detection, series of stylized textures and rotations of point sprites. To simulate wind-driven raining effect, the motion of particles can be freely controlled based on Newtonian dynamics. We can also control the size of raindrops dynamically by using different textures or changing the size of point sprites. Many experiments have been done in 3D scenes with different complexity and GPU-based stylized rendering. The experimental results demonstrated the efficiency of our method for real-time rain simulation in cartoon style with complex geometries of 3D scenes.

Keywords: rain simulation, cartoon rendering, particle systems, collision detection, GPU

1. Introduction

Cartoon animation of natural phenomena is an important research topic of computer graphics, and of these phenomena, rain is perhaps most frequently seen. Its presence in games and simulators etc. significantly enhances attractiveness of generated scenes. Currently, the performance of GPUs is progressing faster than general purpose CPUs. The main reason is an architecture that combines stream processing and SIMD processing. To speed up the animation and fulfill the requirement for real-time rendering, GPUs could be used.

Based on techniques known from traditional CPU-based rain simulation approaches, we develop a new method that utilizes the graphics hardware acceleration. Collision detection of raindrops is based on the graphics hardware acceleration, and some effects, such as splashing, wind-driven motion and size of raindrops, are specially emphasized and dynamically controlled in our method. GPU-based stylized rendering is also applied to our rain simulation.

2. Related work

Much attention has been paid to the simulation of natural phenomena in the computer graphics community for many years, and there is much work related to this field including photorealistic rendering and non-photorealistic rendering (NPR) of these phenomena. Since graphics hardware has developed very fast, real time rendering of natural phenomena has been a hot topic. Here some closely related work has been introduced.

Several approaches to render and model water have been proposed yet and a few of them deal with the phenomenon of water droplets flow. Jonsson [1] proposed a new model to simulate the water droplets flow on structured surfaces using bump maps. Sato et al. [2] proposed a method for real-time animation of water droplets running down on a glass plate using graphics hardware. Taking into account depth of field effects, it is possible to change the focal point interactively depending on a point in the center of the scene being observed.

Besides, several papers dealt with rain simulation. A common approach to the rain simulation is to build a particle system [3]. Adding artificial rain to a video is also an interesting task. In order to derive a fast and simple algorithm for rain simulation, Starik et al. [4] investigated visual properties of rainfall in videos, in terms of time and space, then derived visual properties of the rain "strokes" in the video space and use these strokes to modify the video to give a realistic impression of rain. However, there is much work that can be done to simulate other effects of rain, such as splashing of the raindrops when they reach solid objects etc.

Recent advances in computer graphics hardware and algorithms have also introduced increasing interests in non-photo realistic rendering and cartoon animation. There are many papers about how to produce real-time NPR effects, such as hatched line drawings [5], feature edges drawings [6], stylized haloed outlines drawings [7], cartoon, etc., based on graphics hardware [8]. Rain simulation in cartoon style has also been paid attention to.

Collision detection of raindrops is a pivotal algorithm in our rain simulation. Kipfer et al. [9] have presented a system for real-time animation and rendering of large particle sets using GPU computation and memory objects in OpenGL. Collision detection of large numbers of particles based on GPU has been done.

In the following sections, we will introduce our work in detail on how to achieve real-time rain simulation based on GPU. The remaining of this paper is organized as follows. Section 3 describes the particle systems of the rain simulation. Collision detection of raindrops based on GPU is described in Section 4. In Section 5, we show how to render raindrops and sprays in cartoon style and achieve GPU-based stylized rendering of 3D scenes. The implementation of our method and the experimental results are given in Section 6. The conclusions and the future work of our approach are the subjects of Section 7.

3. Particle systems of rain simulation

In our rain simulation, the particle systems include the particle subsystem of raindrops before reaching solid objects and the particle subsystem of raindrops splashing after collision. Attributes of the first kind of particle and the second kind of particle are shown as follow:

```
Struct particle1 {
        p,          /// current position
        v,          /// current velocity
        a,          /// current acceleration
        alpha,    /// raindrop transparency
        size,      /// raindrop size
        T, …      /// life cycle
};
Struct particle2 {
        p,          /// current position
        n,          /// normal vector of object surface
        w,          /// raindrop weight
        alpha,   /// raindrop transparency
        angle,  /// the angle between the v and n vector
        texID, /// texture ID in different time step of
                     splashing
        T,…    /// life cycle
};
```

The state is updated on a per time step basis, where a single time step is comprised of the following events: emission, collision-free motion of particles, collision response, rendering of collision-free raindrop particles and raindrop-spray particles after splashing.

During collision-free motion of raindrops, we consider wind and gravity as factors that influence the motion. In the current implementation, each particle is first streamed by its initialized velocity in stochastic down direction during time interval *dt,* and the start position is stochastic in certain space. The simplification of forces and accelerations acting on the raindrop is shown in Figure 1, and the displacement is computed using an Euler scheme to numerically integrate quantities based on Newtonian dynamics.

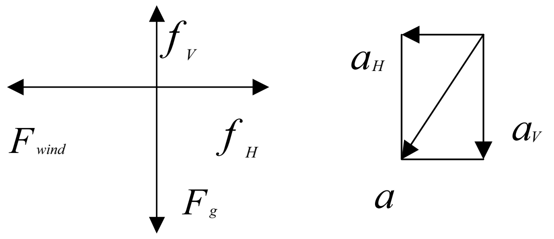

Figure 1: **The forces and accelerations acting on the raindrop**

In the Figure 1, F_{wind} is the external wind force, f_H is the resistance of air on the horizontal, F_g is the gravity, f_V is the resistance of air on the vertical, a_H is the acceleration acting on the horizontal, a_V is the acceleration acting on the vertical, and a is the final acceleration generated from composition of forces based on the formula of Newtonian dynamics:

$$F = ma. \tag{1}$$

Actually the resistance of air is varied base on the formula:

$$f = kv^2. \tag{2}$$

where k is a constant, but in our simulation, f_V is a constant and f_H is omitted for simplification of computation (avoid application of calculus) and reduction of time since it does not make much effect in the test results comparing with realistic rain video. The direction and force of wind on the horizontal can be randomly generated or dynamically controlled in our model.

Collision detection of raindrops is done in the first particle subsystem, and positions and normals of collision are calculated. It is specially presented later in Section 4. After collision of raindrop, the particle enters into the sticked state and fades out of life cycle. At the same time, the particle that simulating splashing of raindrop is created and the information of the sticked particle in the first particle subsystem is sent to it for rendering of raindrop-spray.

Generally, the better utilization of particles is important for the efficiency of the system. In order to reduce the time of creation and initialization of new particles, the particles out of life cycle are reused. Each particle contains the attribute of its current state, such as drop, collision, sticked, and death. When rendering raindrops, we remove the particles in the state of death in time from the active list and push them into the free list. When new particles are needed to be created and the free list is not empty, the particles in the free list are pushed into the active list and are reused.

4. Collision detection of raindrops

In our implementation of collision detection, we make use of the extensions of OpenGL for visibility queries (occlusion queries) [10]. We use GL_NV_OCCLUSION_QUERY to detect collision. It returns the number of visible pixels, and provides an interface to issue multiple queries at once before asking for the result of any one, so applications can now overlap the time it takes for the queries to return with other work increasing the parallelism between CPU and GPU (in Figure 2).

```
(a). Calculate the viewing frustum when the camera
     has moved in some way in the raining scene.
(b). Disable depth/color buffers.
(c). For each particle:
     Generate $i^{th}$ occlusion query for $i^{th}$ particle if it is
     not sticked and visible in the viewing frustum.
     Begin $i^{th}$ occlusion query;
     Render $i^{th}$ particle's bounding box;
     End occlusion query.
(d). Enable depth/color buffers.
(e). Do other CPU computation while queries are
     being made.
(f). For each particle:
     Get pixel count of $i^{th}$ occlusion query.
     If (count < DEFINE_COUNT), Collision is true,
     then set the state of $i^{th}$ particle to sticked;
     Delete occlusion query.
```

Figure 2: **The algorithm of collision detection using GL_NV_OCCLUSION_QUERY**

Unfortunately, when rendering particles during occlusion queries, we cannot get good test results using GL_POINT. So we utilize *gluSolidSphere* (of very small radius) to render each particle's bounding box. The shortcoming of this processing is the increasing time of rendering during occlusion queries. In order to reduce the time of queries, detection of frustum clipping is utilized to omit the particles out of viewing frustum. In the following, the algorithm of collision detection is described in Figure 2. (Please note that we cannot create *n* queries together because the number of particles is unknown in advance.)

During occlusion queries, the current postion is calculated as the collision position when collision of a paticle is true. However, there is a special problem that should be solved. If the raindrop is moving fast enough to enter and exit the collision geometry in one time step (although one time step is very short), the collision detection will miss the event and the collision position will be unknown. In order to get precise collision position, the iterative binary searching algorithm is utilized. The processing includes three steps:

```
(a). Get $P_{cur}$ (the current position of particle after exit
     the collision geometry) and $P_{old}$ (the previous
     position before enter it);
(b). Calculate the midpoint of $P_{cur}$ and $P_{old}$, then set
     $P_{next}$ = midpoint, and test if the collision of $P_{next}$ is
     true;
(c). If the collision of $P_{next}$ is true,
         Set $P_{cur} = P_{next}$, $P_{next} = P_{old}$,
       Else
         Set $P_{old} = P_{next}$.
     Repeat (b) and (c) until find precise collision
     position according to a certain precision.
```

After collision detection, we the get normal of the collision position from depth map that has been created using shaders. The depth map generated by the pixel shader is a screen_space texture (the values of normals stored in depth map as RGB values have been transformed from the range of [-1,1] to the range of [0,1]). The processing to calculate the normal of the collision position includes two steps: Firstly, according the transform matrix that is used in the genertion of the depth map, transform the coordinate of the collision position from the world coordinate system to screen coordinate system. Secondly, determine the texture coordinate of the collision position in the depth map, then get the RGB value of the texture coordinate and transform the value from the range of [0,1] to the range of [-1,1], and the result is the normal at the collision position we need for rendering of raindrop-spray.

5. Stylized rendering of raindrop and splashing

In our simulation, we use stylized textures to cue blurring effect of fallen raindrops. The rendering of collision-free raindrops is implemented by selecting different textures according to the size of raindrops.

We use three kinds of textures (in Figure 3) to express the light, middle and heavy raindrops and select them dynamically in animation. In Figure 3, we show how to bind the textures onto collision-free raindrops where $V_{raindrop}$ is the current velocity of raindrop, V_{center} is the current position of raindrop, V_{up} is the up vector of the eye space, and V_{right} is the right vector of the eye space. Build a square around V_{center} based on the V_{up} and V_{right}. This will guarantee that the square will be orthogonal to the view. Make sure that the direction of raindrop is relevant to $V_{raindrop}$ while binding the textures.

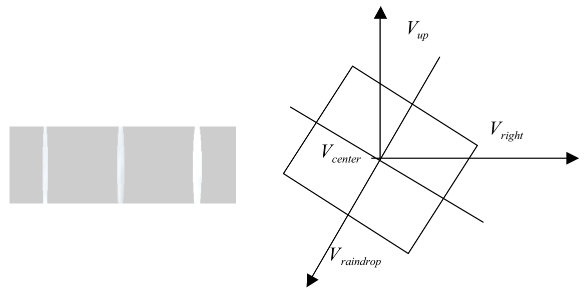

Figure 3: **The textures for light, middle and heavy raindrops (from left to right), and how to bind them onto raindrops**

To our knowledge, splashing of raindrops in rain simulation has not been simulated in real time to achieve a convincing result in past. In our method, we use a series of stylized textures and rotations of point sprites to simulate splashing of raindrops.

After collision detection, the collision positions and normals of particles are known. To achieve 3D effect of splashing of raindrops, we use GL_POINT_SPRITE_NV and call *glPointParameterfARB* to set the parameters of point sprites.

Because the texture binded on point sprite is always parallel with screen and cannot be freely rotated, we rotate the point sprite using the pixel shader. Firstly, calculate the angle between V_{up} (the up vector of the eye space) and the normal of collision position, then project the angle onto screen and use the result as the rotated angle. Secondly, send the rotated angle, raindrop-spray texture, and destination alpha value of the texture to the pixel shader as uniform parameters, and rotate the texture coordinate based on following formula in Figure 4.

$$(x', y') = (x, y)\begin{pmatrix} \cos a & \sin a \\ -\sin a & \cos a \end{pmatrix}$$

(x, y) is the original coordinate,
(x', y') is the rotated coordinate,
a is the rotated angle

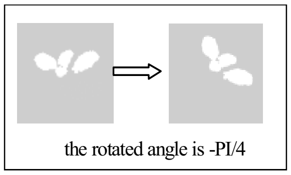

Figure 4: **Rotation of texture coordinate (left). Comparison of point sprite before rotation and after rotation (right)**

In order to represent the diversity of raindrop-sprays, the sprays are classified into three patterns according to the angle between the velocity and the collision normal, and different series of textures are automatically used for different kind of spray. We use six different textures (in Figure 5) to simulate splashing of a raindrop in its life cycle, and reduce the destination alpha value of textures slowly until it is out of its life cycle. It appears as if raindrops fade out gradually in the course of splashing. The size and the shape of sprays can be also dynamically controlled by selecting the relevant series of textures according to the size of raindrops in animation.

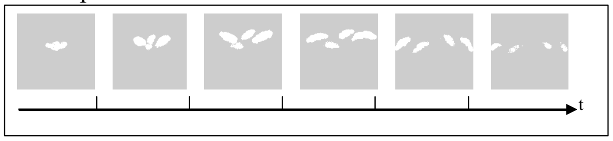

Figure 5: **Six different textures to simulate splashing of a raindrop in its life cycle (The time interval is equal)**

Since there are many papers about NPR based on graphics hardware, the GPU-based stylized rendering of 3D scenes for rain simulation do not need to be specially described. In our rain simulation, all of the NPR effects are implemented using vertex and pixel shaders, and we can choose different effects dynamically to apply to 3D scenes. Because the rendering of raindrops and sprays mentioned above has achieved cartoon effect and the effect look naturally, the GPU-based NPR effects will not be applied to it. The test results are shown in Section 6.

6. Implementation and results

We have implemented our method described above using VC++ 6.0 and OpenGL on an NVidia GeForce 6800 LE GPU. Expect some shaders for hatching are written using assemble language, all of other shaders are written using the OpenGL shading language. The depth map used in collision detection of raindrops is created using the p-buffer and floating-point textures with the rectangle texture target.

Experiments were made by many test examples, and the test results demonstrated the efficiency of our method for 3D rain simulation implemented on programmable graphics hardware. For an AMD Athlon(tm) XP 2500+ 1.83GHz processor with an NVidia GeForce 6800 LE GPU and 512M main memory, the performance varies with the quantity of geometries in raining scenes, but typically runs between 10 and 45 frames per second (in Table 1). The performance of our simulation also varies with the particle system complexity (in Table 2). Some frames in the real-time rain simulation (all raindrops are rendered in cartoon style) are as shown in Figure 6. Please get more frames (higher resolution pictures) and animation sequences (avi files) from the webs: http://www.cs.wichita.edu/~tang/rain/raining.html. (Please note that the frames and animation sequences on the webs are created using an NVidia GeForce FX 5200 Ultra GPU, and collision detection of raindrops is implemented using the GL_OCCLUSION_TEST_HP extension.)

7. Conclusions and future work

In summary, we have demonstrated an interactive system for real-time rain simulation in cartoon style using particle systems and GPU-based stylized rendering. The effects of wind-driven raining and splashing of raindrops have been emphasized. Some improvement has been done to reduce the time of creation and initialization of new particles. The simplification of computation based on Newtonian dynamics during collision-free motion of raindrops and reduction of time has also been achieved. Splashing of raindrops is simulated using collision detection, a series of stylized textures and rotations of point sprites and taking advantage of the parallelism and programmability of GPUs. The size and the shape of raindrops or sprays can be controlled dynamically by using different textures. Many experiments have been done in 3D scenes with different complexity and GPU-based stylized rendering, and the results demonstrate the efficiency of our method.

There is room for improvement, however. We will attempt to use the GPU-based particle engine [9] for animation, collision and rendering of raindrops in our future work if there is an efficient method for simulation of splashing of raindrops based on the engine. With the development of the general-purpose computation on graphics hardware (GPGPU), the computation for the motion of wind-driven raindrops based on more complex dynamics model can be done on GPUs. The accumulation of raindrops on surfaces of objects after collision will be implemented using pattern-based procedural textures [11] based on the graphics hardware. More interesting GPU-based cartoon effects will be applied to the 3D scenes. We will also apply our rain simulation to 3D cartoon games with more complex geometries.

Acknowledgements

This work was supported in part by the NSF Grant CCR-0201253 and Chinese national 973 project, project number: 2002CB312106.

References

[1] Malin Jonsson, “Animation of Water Droplet Flow on Structured Surfaces”, *Linkoping Electronic Conference Proceedings,* SIGRAD, 2002, ISSN 1650-3686. http://www.ep.liu.se/ecp/007/003/.

[2] Tomoya Sato, Y. Dobashi, T. Yamamoto, “A Method for Real-Time Rendering of Water Droplets Taking into Account Interactive Depth of Field Effects”, *Proc. IWEC*, 2002, pp. 110-117.

[3] Tabuchi Yoshihiko, Kusanoto Kensuke, and Tadamura Katsumi, “A method for rendering realistic rain-fall animation with motion of view”, *IPSJ SIGNotes Computer Graphics and cad Abstract,* 2001,No.105 – 005.

[4] Sonia Starik, Michael Werman, “Simulation of Rain in Videos”, *The 3rd international workshop on texture analysis and synthesis,* 17 October 2003, Nice, France, pp. 95-100, ISBN 1-904410-11-1, 17.

[5] Emil Praun, Hugues Hoppe, Matthew Webb, Adam Finkelstein, “Real-Time Hatching”, *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, August 2001, p. 581.

[6] Morgan McGuire, John F. Hughes, “Hardware-Determined Feature Edges”, *Proceedings of Third International Symposium on Non Photorealistic Animation and Rendering (NPAR 2004, Annecy, France, June 7-9, 2004),* ACM Press, New York, 2004, pages 35-44.

[7] Jorn Loviscach, “Stylized Haloed Outlines on the GPU”, In *SIGGRAPH 2004 Poster,* New York, 2004. ACM Press.

[8] Ramesh Raskar, “Hardware Support for Non-photorealistic Rendering”, *2001 SIGGRAPH / Eurographics Workshop on Graphics Hardware (August 2001),* 2001, pp. 41-47.

[9] Peter Kipfer, Mark Segal, and Rudiger Westermannn, “UberFlow: A GPU-Based Particle Engine”, *In Graphics Hardware 2004*, Aug 2004, pp. 115-122.

[10] Naga Govindaraju, Ming C. Lin and Dinesh Manocha, “Fast and Reliable Collision Culling using Graphics Processors”, *Proceedings of ACM VRST 2004*, http://www.cs.unc.edu/Research/ProjectSummaries/RCullide04.pdf.

[11] Sylvain Lefebvre, Fabrice Neyret, “Pattern Based Procedural Textures”, *Proceedings of ACM SIGGRAPH 2003 Symposium on Interactive 3D Graphics,* 2003, pp. 203-212.

Table 1: **Running Times (Frames per Second) (statistical average) for 3D scenes with different complexity and NPR effects**

Model	#Of points	#Of triangles	Number of particles released per second	Fps for scenes with different NPR effects (viewport size 512 x 512)				
				Gooch	Gooch & Outlines	Cartoon	Cartoon & Outlines	Outlines
Car	6,387	9,907	10,000	32.27	31.97	33.41	32.78	32.50
House1	8,726	14,240	10,000	31.70	31.49	32.97	32.36	31.95
House2	952	1,187	10,000	32.58	32.11	34.26	33.50	32.78
Table	19,016	37,372	10,000	31.60	31.40	32.84	32.58	31.97
Park1	5,744	10,308	10,000	32.29	32.01	33.04	32.71	32.55
Park2	25,817	48,154	10,000	31.49	31.31	32.76	32.35	31.92
Ground	4,601	8,180	10,000	32.12	31.97	32.91	32.55	32.20

Table 2: **Running Times (Frames per Second) (statistical average) for 3D scenes with Cartoon effect and different particle system complexity**

Model	Particle system complexity (number of particles released per second)				
	6,000	8,000	10,000	15,000	20,000
Car	45.96	39.44	33.41	22.09	11.33
House1	45.12	38.93	32.97	21.60	11.21
House2	46.29	40.91	34.26	22.45	12.08
Table	44.97	38.03	32.84	21.80	11.21
Park1	46.59	38.98	33.04	22.39	11.41
Ground	46.01	39.58	32.91	21.70	11.20

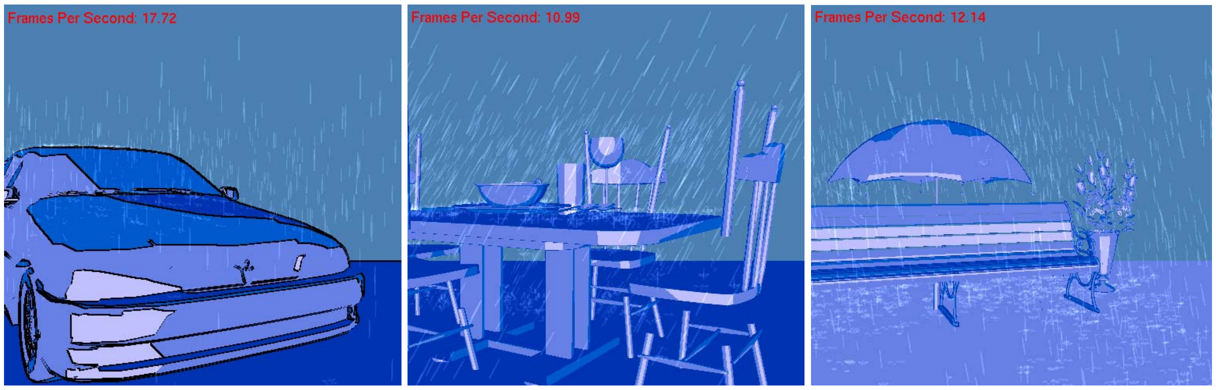

Figure 6: **Cartoon & outlines effect (the car model) (15,000 particles released per second) (left) Cartoon & wind-driven effect (the table model) (20,000 particles released per second) (middle) Cartoon & wind-driven effect (the park2 model) (18,000 particles released per second) (right)**