Vol. 13, No. 4: 17–33

GPU Rainfall

Pierre Rousseau, Vincent Jolivet, and Djamchid Ghazanfarpour
Institut XLIM

Abstract. Outdoor video games can improve their realism through weather effects simulation. However, simulating rain can yield many problems. We present solutions to address these problems and describe a complete framework to simulate rainfall in a video game. Rendering uses shaders to refract the scene inside the raindrops, based on optical properties. Retinal persistence is also considered. Animation is entirely conducted on graphics hardware, taking into account collisions and wind advection. An interface is presented, which allows the creation of complex wind fields by the user. Videos and shaders are available online.

1. Introduction

Every video game or virtual environment featuring outdoor action can increase its realism using weather effects. Fog rendering enables the reduction of the observable depth of the scene and is widely used in real-time applications thanks to its hardware acceleration. Nevertheless, rain often lacks realism in real-time applications. Little computation time is available for auxiliary effects such as weather rendering. Thus, most of the time, rain is rendered using simple particle systems and static white streak textures.

Recently, real-time rain rendering has become a major topic in natural phenomena simulation. However, most papers introduce heavy simplifying assumptions to maintain execution rates. In [Wang and Wade 04], artist-drawn rain textures scroll on a double cone, thus forbidding interaction between the particles and their environment. In [Yang et al. 04], the drops are distorted in

1086-7651/08 $0.50 per page

image space without considering the laws of optics. The rain scenes presented in [Tatarchuk 06] and [Tariq 07] are set at night, hence removing the need for a refraction model to represent the raindrops.

Building upon work introduced in [Rousseau et al. 06], this paper presents various problems one can be confronted with when trying to simulate rain in a video game and solutions to address them. We describe a complete framework for rainfall simulation in real-time applications. We present solutions to animate rain with an optimized GPU-based particle system and to render them taking into account the physical properties of raindrops (in particular their refraction characteristics and retinal persistence). This paper also introduces techniques to combine particle animation with simple collision detection and primitive-based wind simulation.

2. Rainfall Simulation

Rainfall simulation requires both efficient animation and rendering techniques to achieve realism. We animate raindrops as independent particles and render them taking into account their optical properties.

Animation cannot be efficiently conducted with classical CPU particle systems. The large number of particles involved would imply an unaffordable performance hit in this case. GPU-based particle systems can efficiently tackle this issue; however, some assumptions can be made in the special case of rain animation that allow us to further optimize this technique.

When examined closely, raindrops cannot be satisfyingly represented by simple textures, as most video games do. The refractive nature of raindrops induces the need for an efficient refraction model. Moreover, a specific retinal persistence model is mandatory due to the fast motion and small size of a raindrop.

This section shows how to adapt a basic GPU-based particle system to the needs of rain animation. It then presents the rendering algorithm we use to render realistic raindrops, taking into account retinal persistence.

2.1. Particle Animation

Originally introduced in [Kipfer et al. 04] and [Kolb et al. 04], GPU-based particle animation can be summarized in the following steps:

1. Particle positions are stored in a floating-point texture (used as an arbitrary-data container), inside GPU memory.

2. These positions are updated for each animation step using a fragment shader.

3. The texture is used to obtain the actual positions of the particles when the time comes to display the particles.

2.1.1. Position Texture

This texture is used to store the positions of the particles. Each pixel's red, green, and blue components are associated with a particle's x-, y- and z-coordinates, respectively. The texture uses 32 bits-per-component floating-point format, so that coordinates do not have to be clamped to the [0..1] range. Figure 1 illustrates an example position texture.

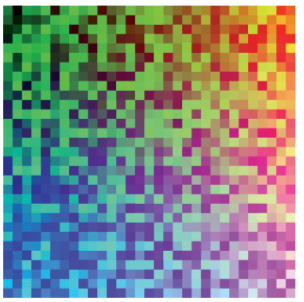

Figure 1. An example position texture.

Due to hardware constraints, a texture cannot be used simultaneously as input and output for the computation. We thus have to use two mirroring textures for position. These textures are used in a *ping-ponging* approach: the texture used as output in a given computation step will be used as input in the following step.

2.1.2. Our "Rain-Box" Model

In a typical rain scene, particles are falling everywhere in the scene; it is obviously useless to animate and render particles that are out of the user's field of view. We thus set a particular constraint on the system, that particles are considered to evolve within a box. This box typically scales along 100 m on each world axis.

Limiting the evolution of particles or fluid flow to a box is commonplace in computer graphics; however, in our context, this box must be able to adapt itself to the user's motion, so that most particles are always inside the user's field of view.

We locate the box so that the user is always on an edge of the box; it moves depending on the view direction of the user, so that he is always looking

toward the center of the box. Coordinates in the position texture are expressed relative to the center of this box. This implies that the positions have to be corrected when the user moves or changes his view direction, to enable the particles to fall in world space.

In an initialization step, the rain box is filled with homogeneously distributed particles. Every time a particle leaves the box on any of its sides (due to its own motion or user movement), it is relocated on the opposite side; this avoids having to deal with particle birth and death. Particles die when they leave the box, and they are immediately re-spawned.

2.2. Particle Rendering

Combined with the efficient hardware animation model presented above, our method aims at the complementary goal of realistically rendering the rain particles. This section introduces the physical properties of raindrops on which we built our model and outlines the rendering model we use. We refer the reader to [Rousseau et al. 06] for full detail.

2.2.1. Physical Properties of Raindrops

The typical shape of a raindrop can be described as roughly spherical and flattened at the bottom. This shape results essentially from an equilibrium between surface tension (which yields a spherical shape) and aerodynamic pressure (which tends to flatten the drop). Equation (1) (introduced in [Chuang and Beard 90]) accurately models this shape by distorting a regular sphere of radius a:

$$r(\theta) = a\left(1 + \sum_{n=0}^{10} C_n \cos(n\theta)\right), \tag{1}$$

where θ is the polar elevation of the estimated point and C_n is a set of coefficients (which can be found at the address listed at the end of the paper).

In raindrops, refraction is strongly dominant over reflection. What is seen in a raindrop is a flipped and distorted view of the scene behind the drop.

Considering the refractive indexes of water and air, one can easily calculate that the maximum deviation a ray of light can suffer through a spherical raindrop is 82.5°. We can thus state that the "field of view" of a raindrop is 165°.

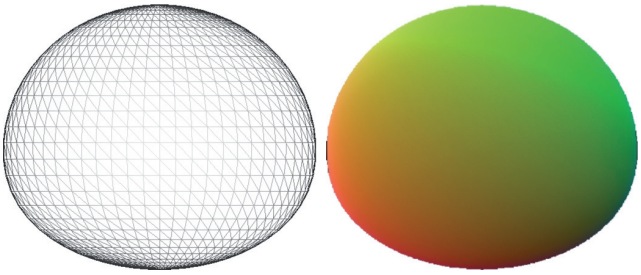

Figure 2. Left: 3D shape of a 3 mm wide raindrop. Right: corresponding refraction mask.

2.2.2. Rendering Algorithm

The key idea of our algorithm is to map a wide-angle texture (representing the "field of view" of a drop) onto the particles, based on precomputed refraction directions.

For each frame, the scene is rendered to a texture, using a wide-angle virtual camera located at the observer's position. This camera uses a 165° FOVy, (thus capturing the "field of view" of a raindrop).

Refraction masks are computed in a preprocessing step for various typical drop diameters using Equation (1) and are stored into textures (as illustrated in Figure 2). Each pixel of a mask indicates the direction toward which light is refracted after passing through the drop (spatial directions being encoded as colors). These masks are used at runtime to obtain the shape of the drop and the refraction direction for each pixel.

The pixel shader in charge of rendering the drops uses the refraction direction given by the mask to select the pixel from the wide-angle capture that will be mapped onto each location of the rendered drop. In a first step, we locate in this texture the pixel that would be seen with no refraction deviation. This location is then shifted (in image space) by the refraction direction (taken from the mask) to get the pixel seen through the raindrop.

Full details about this algorithm can be read in [Rousseau et al. 06], where we also introduce an extension to handle illumination from point light sources, suitable for dim-light scenes. All the shaders used, as well as the precomputed masks, are available online at the address listed at the end of this paper.

2.2.3. Retinal Persistence

In spite of its solid physical grounding, this algorithm lacks visual realism. This is due to the retinal persistence phenomenon. The human eye is used to seeing rain *streaks*, blurred and vertically stretched.

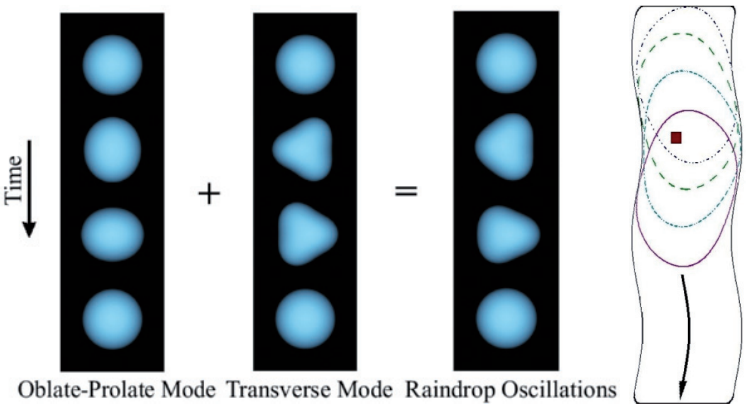

Figure 3. Left: raindrop oscillations [Garg and Nayar 06]. Right: the rendering process used in our method. The red pixel receives refractive contributions from four sample drops.

"Classical" motion-blur techniques (relying on postprocessing) are inappropriate in this case, as they suppose images of an object overlap from frame to frame; this is not the case with small, fast-moving objects such as raindrops. To take retinal persistence into account, we thus have to alter the shape and appearance of the falling particles. We present a new way to efficiently handle retinal persistence in the case of rain, taking into account raindrop oscillations.

To render retinal-persistence–affected streaks, we first need to modify the shape of the particles and stretch them vertically. Drop oscillations are then taken into account in the rendering process. These oscillations (illustrated in Figure 3 (left)) are described by Equation (2) (originally introduced in [Frohn and Roth 00]):

$$r(t,\theta,\phi) = r_0 \left(1 + \sum_{n,m} A_{n,m} * \sin(\omega_n * t) * P_{n,m}(\theta) * \cos(m\phi)\right), \qquad (2)$$

where

- r_0 is the radius before distortion;
- $A_{n,m}$ is the amplitude of the spherical harmonic mode (n, m);
- $P_{n,m}(\theta)$ is the Legendre function that describes the dependence of the shape on the angle θ for the mode (n, m);
- ω_n is the oscillation frequency.

Equation (2) can be simplified based on the following assumptions: (n, m) evaluation can be restricted to two modes, $(n = 2, m = 0)$ and $(n = 3, m = 1)$, which are highly predominant in a raindrop's oscillation. Comparing simulated and measured data, [Garg and Nayar 06] demonstrated that $(A_{2,0}, A_{3,1}) = (0.1, 0.1)$ and $(A_{2,0}, A_{3,1}) = (0.2, 0.1)$ produce realistic results.

The pixel shader used in our method to render these streaks uses Equation (2) to outline the raindrops. The outline is computed in the horizontal plane, hence the θ parameter can be fixed to $\pi/2$.

The parameter ϕ is the angle in the horizontal plane along which the radius is computed. Any value will yield physically plausible results. We defined this parameter to be dependent on the horizontal coordinates of the drop, allowing the various drops observable onscreen to adopt different phases. The only requirement is to evaluate oscillating radii simultaneously on the left and right outlines of the drop, using $\phi_{\text{right}} = \phi_{\text{left}} + \pi$.

With these simplifying assumptions, the outline of the drop can be efficiently computed. This outline is then regularly sampled to obtain a few sample drop positions (as illustrated in Figure 3 (right)). For each pixel inside this outline, refraction is computed for each sample drop, based on the technique described in Section 2.2.2. These refraction contributions are then averaged to obtain the color of the pixel. Our experience shows that five to ten samples are sufficient for a satisfying visual impression.

This method has a moderate impact on the overall application performance and does bring a strong improvement in the rain perception. Although refraction is no longer perceivable in retinal-persistence–affected streaks, drops rendered with this technique emphasize the dominant colors of the scene.

3. Wind-Field Generation

Up to this point of the algorithm, our application allows a realistic rendering of raindrops, but the animation still lacks realism, as no external force is applied to the particles. We thus want to include wind advection in our simulation.

In computer graphics, Navier–Stokes and Lattice–Boltzmann methods are often the preferred ways of tackling fluid simulation problems. However, open-field wind simulation requires large-scale grids, which are impractical with these methods in the context of real-time applications such as video games.

We chose to adopt a primitive-based procedural modeling technique, such as presented in [Wejchert and Haumann 91] and [Perbet and Cani 01]. This technique can be conducted at low cost and is based on the combination of simple primitives to create a complex wind flow inside a 3D grid.

3.1. Wind Primitives

The wind field is designed by the user through a combination of various primitives to model a complex wind flow. Primitives are set so that they are solutions of the Navier–Stokes equations, and their linear combination is also a solution. Each primitive is defined by a set of simple parameters: strength, base position, axis. Available primitives are the following:

- *Source.* Wind blows in all directions from this point.
- *Sink.* Wind concentrates from all directions towards this point.
- *Vortex.* Wind spins around an axis.
- *Uniform wind.* Wind is constant in orientation and strength in every location.
- *Tornado.* This is easily obtained though the combination of a sink on the base, a source on the top, and a vortex.

When a primitive is added, the relative information is transmitted through a matrix (used as a built-in data container) to a shader which computes the impact of the primitive on each point of the 3D grid and adds the contributions of the different primitives. Each matrix contains the mandatory information relative to each primitive, such as its type, strength, orientation, and base point. Here is an example sink primitive matrix:

$$\begin{pmatrix} \text{Type} & \text{Strength} & 0 \\ \text{Center}.x & \text{Center}.y & \text{Center}.z \\ \text{Up}.x & \text{Up}.y & \text{Up}.z \end{pmatrix}.$$

The shader that generates the wind texture thus receives an array of matrices from the interface and iterates through the array to add the contribution of each primitive to each grid cell. The influence of a given primitive on a given grid cell is evaluated with the following formulas (expressed in cylindrical coordinates):

- *Source.*
$$V_r = \frac{a}{2\pi r}, \qquad V_\theta = 0, \qquad V_z = 0,$$
where a (positive) is the strength of the source.
- *Sink.* Same equation as source, with $a < 0$.
- *Vortex.*
$$V_r = 0, \qquad V_\theta = \frac{b}{2\pi r}, \qquad V_z = 0,$$
where b is the strength of the vortex.
- *Uniform wind.* Same strength and direction in every grid cell.

3.2. Wind-Field Texture

To interact with the particle position update shader, we chose to store our 3D wind field in a texture, each pixel corresponding to a cell of the 3D grid. Wind directions (evaluated from the primitives) are encoded on red, green, and blue components.

Representing the wind in a texture naturally leads to the idea that it can be easily modified using a pixel shader. Since 3D textures cannot be used as render targets on current hardware, we use the *flat 3D texture* concept introduced in [Harris et al. 03]. Each slice of a conceptually 3D texture is tiled in a 2D texture; a $16 \times 16 \times 16$ 3D texture can thus be stored as a 64×64 2D texture. This simple conversion enables us to animate our 3D wind texture through a pixel shader. We can thus conceive our texture as volumetric, though internally storing it as two-dimensional.

In our implementation, a $64 \times 64 \times 64$ wind texture proves to be sufficient and produces no visual artifacts (with trilinear texture interpolation of the wind texture in the pixel shader updating the particles' positions).

3.3. Wind Design Interface

Parameters of the various primitives are not fully intuitive, and hand-assembling those primitives might prove tedious. It is, however, easy to overcome this difficulty by using a simple design interface such as the one illustrated in Figure 4. In this interface, primitives are represented by easily identifiable 3D models, illustrating the shape that the primitive will give to

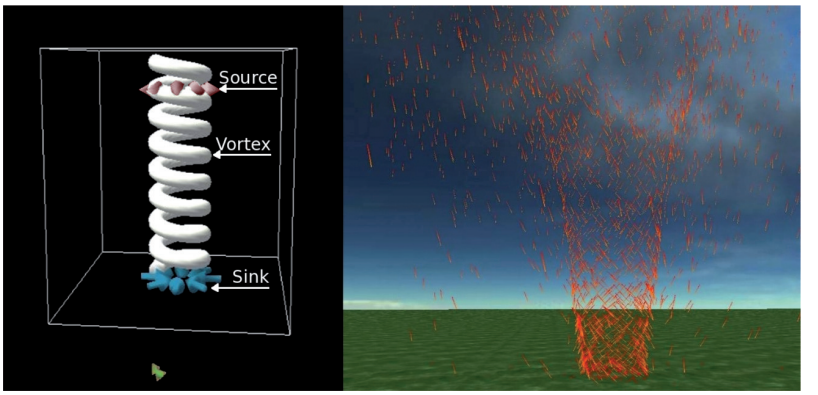

Figure 4. Left: wind design interface. Right: impact of the wind field on the particles.

the wind flow; the user can thus visually construct the wind flow and discover its impact on the particle's trajectories in real time (as illustrated in Figure 4 (right)).

Our interface allows the user to freely move, rotate, and alter the strength of each primitive added to the system. Each modification of the primitive set triggers an update of the primitive matrices, which are then transmitted to the shader in charge of creating the wind texture.

Depending on the needs of the video game, is is straightforward to couple this interface with a key-framing system, e.g., to get animated wind patterns associated with different regions of a game level and reinforce gameplay.

4. Collision Response

To further enhance the realism of our simulation, we must avoid having drops falling through objects; a collision detection model is hence necessary. We are dealing with a very specific case of collisions here, where particles' fall directions are close to each other, with a dominant vertical component.

This consideration leads us to the idea that collisions between particles can safely be ignored, as well as collisions between particles and vertical surfaces (such as walls); in such a case, particles can be considered to be absorbed by the surface and re-spawned at another location.

The approach we use extends that of [Tecchia and Chrysanthou 00]; an orthographic camera pointing vertically downward is positioned on top of the rain box, with its view frustum adjusted to the borders of the box. This camera renders all the objects in the scene to a "collision texture" after applying a specific shader to each object. The shader represents the normal of the observed object with the RGB components of the collision texture. The alpha component represents the altitude of the object (in world coordinates),

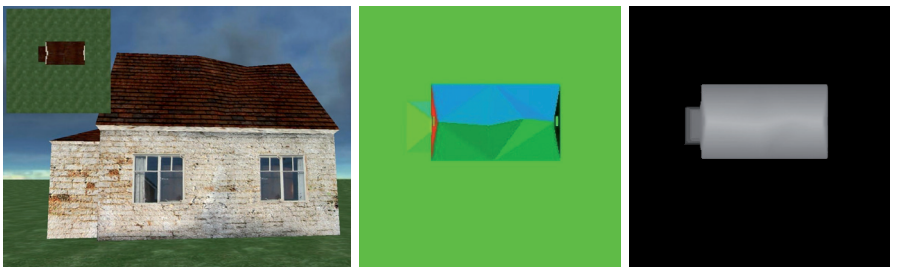

Figure 5. Left: a small house model seen from the collision texture. Center: colors correspond to the normals. Right: height is encoded in the alpha channel.

as illustrated in Figure 5. Alpha is thus used as a height map, while RGB components can be used to determine bouncing directions.

The various parameters of the collision texture obviously depend on the size of the rain box. For improved precision (especially regarding altitude encoding), we recommend using 32 bits per channel floating-point textures. In our application, a 128×128 collision texture gives satisfying results for typical boxes (spanning over 100 meters along each axis), without noticeably impacting the overall performance of the simulation. For scenes with strong geometry variations and large dimensions of the rain box, larger collision texture resolutions can be used. The main advantage of this technique is that it enables us to detect collisions between the particles and moving objects (such as vehicles, characters, etc.).

5. Interactions

We have described algorithms to animate and render rain particles, to design a wind field, and to detect collisions between particles and objects in the scene. This section shows how to make these algorithms work together. We first present how the wind and collision textures influence the particles, then we summarize the framework into one algorithm.

5.1. Influencing Particles

5.1.1. Velocity Texture for Complex Motion

We use a "velocity texture" to update the position texture. It defines the velocity for each particle and provides a link between the position texture on one side and the wind and collision texture on the other side. With this texture, complex motion patterns (such as bounces) can be easily achieved. Each pixel of the velocity texture stores the fall direction of a particle in the RGB components. Knowing the current velocity, the previous position, and the time elapsed since the previous frame was rendered, evaluation of the new position of a particle is straightforward.

The alpha channel can be used as a Boolean indicator of the type of particle: this provides an easy way to use a different rendering model for free-falling drops and bouncing particles.

Like the position texture, the velocity texture uses a mirror ping-ponging texture. In our implementation, both position and velocity texture are updated from the same pixel shader (i.e., only one pass required for two textures) using the *multiple render targets* technique (also referred to as *multiple draw buffers* [Everitt 05]).

5.1.2. Wind and Collisions Influence on Particles

The ultimate goal of the wind-field and collision textures is to alter the motion of particles; our approach makes this easy. The wind-field texture straightforwardly impacts the velocity texture, which is simply updated by averaging the velocity at the previous time step and the wind direction .

We consider that no wind blows inside objects; the collision texture is thus used by the shader in charge of creating the wind texture. If a grid cell is inside an object, it isn't influenced by the wind primitives and is set to a null wind. As the wind texture is trilinearly interpolated in the position-computing shader, there is a gradual wind cutoff for grid cells close to objects' borders.

The collision texture is also used in the position-computing shader itself. The position of each particle is compared with the altitude of the highest object sharing the same horizontal coordinates (taken from the collision texture). When a collision is detected, the velocity texture can be used to make the particle bounce off the impacted surface (based on the normal). It can also help by changing the rendering mode of the particle to a low-motion quasi-spherical refractive drop, a precomputed animated splash sprite (as done in [Tatarchuk 06]), or any suitable specific model.

5.2. Complete Algorithm

Depending on the target hardware, several ways exist to transfer data from the position texture to actual geometry:

- For low-end hardware, *Über-buffers* can be used, as presented in [Kipfer et al. 04]. This technique consists of alternatively interpreting a single buffer as pixel data or geometry data, thus enabling us to render a texture that can subsequently be considered as a vertex array.

- For systems supporting Shader Model 3.0, we presented in [Rousseau et al. 06] how to use *vertex texture fetch* for our purpose. The principal idea of this technique is to feed a static vertex array containing dummy positions to a vertex shader that reads the actual particle positions directly from the position texture.

- For higher-end systems, the preferred method is to use *pixel buffer objects* together with *geometry shaders*. This way, positions can be efficiently rendered to a vertex buffer by a pixel shader, and particles can be expanded from one position (the particle's center) to the four positions of a billboard's corners in a geometry shader. This technique is definitely the most efficient available at the time of writing but requires hardware compliant with Shader Model 4.0 specifications.

We advise the reader to wisely consider which hardware the implementation is to be deployed upon before choosing from the three aforementioned techniques. In particular, graphics cards predating the advent of the GeForce 8 series cannot handle the last of these techniques.

Algorithm 1. (Rainfall simulation.)

```
Update collision texture;
Update wind texture;
Compute next velocity;
Compute next position;
Compare position with altitude from collision texture;
if below then
    if far below then
        Re-spawn the particle (absorbed by a wall)
    else
        // slightly below : particle bounces
        Compute bouncing direction using current direction and normal
        to the surface. Use this new direction to determine next velocity
        and position
    end
end
if particle left the box then
    Relocate it on the opposite side to that from which it left the box
end
Write velocity and position to the textures;
Render particles
```

Algorithm 1 illustrates the sequence of events involved in each animation timestep of our framework.

To further increase the realism of the animation, the particles can be inclined in the vertex shader according to their current trajectories (extracted from the velocity texture). This insures consistency from frame to frame, as particles slide along their trajectories.

6. Performance and Examples

Table 1 presents the performance of our application, measured on a test scene taken from an existing video game for raindrops of radius $r = 1$ mm and an image resolution of 1024×768. Frame rates are measured on a 2 GHz Intel Core 2 Duo laptop with an NVIDIA GeForce 8600M GT graphics card. The first column of the results is achieved with a static drop texture, requiring

	Simplified rendering	Refraction	Streaks
0 particles	490 fps		
5,000 particles	185 fps	168 fps	142 fps
10,000 particles	174 fps	154 fps	118 fps
20,000 particles	160 fps	133 fps	86 fps
40,000 particles	139 fps	108 fps	59 fps

Table 1. Performances.

no rendering overhead; this enables us to estimate the cost of the animation process alone. The other two columns present results using the refraction model and the retinal persistence model described in Section 2.2. In our experience, 10,000 to 20,000 drops are sufficient to provide a visually convincing rain impression.

Radius	1 sample	5 samples	10 samples	20 samples	50 samples
0.5 mm	152 fps	143 fps	128 fps	112 fps	77 fps
1 mm	148 fps	133 fps	114 fps	93 fps	55 fps
2 mm	141 fps	119 fps	94 fps	72 fps	38 fps
5 mm	127 fps	95 fps	67 fps	48 fps	22 fps

Table 2. Impact of the retinal persistence simulation (10,000 particles).

The retinal persistence technique we present does have an impact on the overall performance, as it is sample-based. Table 2 outlines this impact in various situations (using 10,000 particles). Our experience shows that five to ten drop samples per pixel provide visually satisfying results, and drops smaller than 2 mm wide are more realistic than larger drops.

Figure 6. Left: large drops with our refraction model. Right: streaks blown in a tornado.

Figure 7. A rainy night scene.

Figures 6 and 7 present example scenes rendered using our method. The left part of Figure 6 illustrates our refraction technique, in front of a photograph, and the right part shows streaks advected by a hand-designed tornado. Figure 7 illustrates a night scene, with light rain streaks blown in a wind coming from the left of the image.

Acknowledgments. 3D models in the test scenes are extracted (with kind permission) from the game *Secret of Time*. Figure 3 (left) is reprinted with kind permission from Kshitiz Garg and Shree K. Nayar. This work is partially supported by the GameTools project of the European Union (contract number 004363).

References

[Chuang and Beard 90] C. Chuang and K.V. Beard. "A Numerical Model for the Equilibrium Shape of Electrified Raindrops." *J. Atmos. Sci.* 47:11 (1990), 1374–1389.

[Everitt 05] C. Everitt. "OpenGL 2.0 and New Extensions." In *Proceedings of Game Developers Conference*, 2005.

[Frohn and Roth 00] A. Frohn and N. Roth. *Dynamics of Droplets*. Berlin-Heidelberg: Springer, 2000.

[Garg and Nayar 06] K. Garg and S.K. Nayar. "Photorealistic Rendering of Rain Streaks." *Proc. SIGGRAPH '06, Transactions on Graphics* 25:3 (2006), 996–1002.

[Harris et al. 03] M. J. Harris, W. V. Baxter, T. Scheuermann, and A. Lastra. "Simulation of Cloud Dynamics on Graphics Hardware." In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pp. 92–101. New York: ACM Press, 2003.

[Kipfer et al. 04] P. Kipfer, M. Segal, and R. Westermann. "UberFlow: A GPU-Based Particle Engine." In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pp. 115–122. New York: ACM Press, 2004.

[Kolb et al. 04] A. Kolb, L. Latta, and C. Rezk-Salama. "Hardware-Based Simulation and Collision Detection for Large Particle Systems." In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pp. 123–131. New York: ACM Press, 2004.

[Perbet and Cani 01] F. Perbet and M. P. Cani. "Animating Prairies in Real-Time." In *ACM-SIGGRAPH Symposium on Interactive 3D Graphics (I3D)*, pp. 103–110. New York: ACM Press, 2001.

[Rousseau et al. 06] P. Rousseau, V. Jolivet, and D. Ghazanfarpour. "Realistic Real-Time Rain Rendering." *Computers & Graphics* 30:4 (2006), 507–51. Special issue on Natural Phenomena Simulation.

[Tariq 07] S. Tariq. "Rain." Technical report, NVIDIA, 2007.

[Tatarchuk 06] N. Tatarchuk. "Artist-Directable Real-Time Rain Rendering in City Environments." In *Proceedings of Game Developers Conference*, 2006.

[Tecchia and Chrysanthou 00] F. Tecchia and Y. Chrysanthou. "Real-Time Visualisation of Densely Populated Urban Environments: A Simple and Fast Algorithm for Collision Detection." In *Proceedings of the Eurographics Workshop on Rendering Techniques*, pp. 83–88. Berlin-Heidelberg: Springer, 2000.

[Wang and Wade 04] N. Wang and B. Wade. "Rendering Falling Rain and Snow." In *ACM SIGGRAPH Technical Sketches Program*, 2004.

[Wejchert and Haumann 91] J. Wejchert and D. Haumann. "Animation Aerodynamics." *Proc. SIGGRAPH '91, Computer Graphics* 25:4 (1991) 19–22.

[Yang et al. 04] Y. Yang, C. Zhu, and H. Zhang. "Real-Time Simulation: Water Droplets on Glass Windows." *Computing in Science and Eng.* 6:4 (2004), 69–73.

Web Information:

Additional material available at http://jgt.akpeters.com/papers/RousseauEtAl08/.

Pierre Rousseau, Institut XLIM, 83 rue d'Isle, 87000 Limoges, France
(rousseau@msi.unilim.fr)

Vincent Jolivet, Institut XLIM, 83 rue d'Isle, 87000 Limoges, France
(vincent.jolivet@xlim.fr)

Djamchid Ghazanfarpour, Institut XLIM, 83 rue d'Isle, 87000 Limoges, France
(djamchid.ghazanfarpour@xlim.fr)

Received April 11, 2008; accepted November 20, 2008.