Efficient Rain Simulation based on Constrained View Frustum

JinGi Im[1]
Dept. of Computer Engineering, Graduate School
Keimyung University, Daegu, Republic of Korea

Mankyu Sung[2]*
Dept. of Game and Mobile Engineering
Keimyung University, Daegu, Republic of Korea

***Abstract*—Realistic real-time rain streaks rendering has been treated as a very difficult problem because of various natural phenomena. Also, for creating and managing many particles in a rain streak, many resources had to be used. This paper propose am efficient real-time rain streaks simulation algorithm by generating view-dependent rain particles, which can express a large amount of rain streaks even with a small number of particles. By creating a 'constrained view frustum' depending on the camera moving in real time, particles are rendered only in that space. Accordingly, particles rendered well even if the camera keep moving or rotating rapidly. And a small number of particles are used, since the simulation is performed in a user-viewed limited space, an effect of simulation many particles can be obtained. This enables very efficient real-time simulation of rain streaks.**

Keywords—View-dependent rendering; realistic real-time simulation; view frustum

I. Introduction

When a digital content is produced for a specific weather from among various weather conditions, the audience or user who encounters the medium can immerge to the content more easily. There are many types of weather conditions such as sunny, cloudy, rainy, and snowy days. There are two main methods of rendering them: off-line rendering and on-line rendering. In the case of movies, since it is a medium that does not communicate with the audience in real-time, so even if it takes a lot of time, it pursues realistic rendering results so that the results naturally melt into the filming scene. On the contrary, in content such as games, even if the factual point is relatively less important, it emphasizes real-time, so it pursues somewhat lower quality rendering results compared to that of movies.

This paper proposes a real-time simulation algorithm for rainy scene. When it rains, complicated and diverse phenomena should be considered, such as droplets, splashes, rainbows, and clouds. Although the graphics hardware has been improved drastically in recent years, real-time simulation of rains is still very difficult problem considering the various conditions and the governing physical law to simulate them. Moreover, considering the physical properties of raindrops or water, the entire simulation process becomes very complicated. Therefore, most of real-time rain simulation focuses only on specific phenomena among various phenomena in rainy weather, and many different technical methods have been proposed to obtain real-time simulation performance by approximating the required parameters. Many of them have presented for modeling realistic raindrops based on the physical properties [1]. This enables realistic rain simulation as well as raindrop modeling. Also, they could present various phenomena including collision detection between raindrops and objects but gave the disadvantage of inefficiency because calculations related to raindrops became heavy due to unnecessary invisible rendering on the screen [2]. To fix it, various studies have been proposed such as defining and rendering only specific region for efficient simulation, but when the camera is moving outside of the region, there are awkward discontinuities where raindrops are not visible or rendered outside of the space [3, 4, 5].

*Corresponding Author

This paper proposes an algorithm that complements the aforementioned shortcomings by mapping rain streaks textures to particles and creating a particle system that depends on the position and FOV (field of view) of the camera. Also present a method of interaction between rain and light sources, which is a simple light scattering technique for changing color of rain streaks. In this work, the particles are rendered even when the camera is moving and rotating. Therefore, it can avoid awkward rendering where rain can be rendered only in certain spaces, which requires only a small number of particles, but appearing to be rendered in very large quantities. This enables more efficient real-time rain-streaks simulation in a 3D space.

II. Related Work

Many research methods have been proposed in realistic rain simulation, and most of them focus on certain parts of the phenomenon of rain. There are two main types of rain simulation: rendering rain streaks with translucent white quads. The other method is to map precomputed rain streak textures to quads. The rain streaks texture rendering model proposed by Garg et al. presents a vibration model for raindrops. The rain streaks texture rendering model proposed by Garg et al. presents a vibration model for raindrops. As a result, they made a database of high-quality renders for many values of the illumination parameters [1]. Then, they used simple image-based algorithms from the depth map, camera parameters, and user input for viewpoints to synthesize the final images. This method showed the performance of 10 sec/frame, which was unsuitable for real-time simulation. This study used their database but implement rain simulation in real-time. Also, for randomness of the rain streaks, hundreds of textures are used to randomly map to the particles.

Weber *et al*. focused on the relationship between raindrops and trees. In their techniques, through-fall simulation is analyzed phenomenologically, and rendering is done based on the amount of water stored in the tree canopy and leaves [2]. Furthermore, in the study of Nanko *et al*., the distribution of dripping through-fall was considered temporally and spatially [6]. The through-fall was largely divided into two different water. The first water was the raindrops in the natural state. This was a free through-falling without hitting anything other objects. The second one was the water splashes stored by hitting the canopy and leaves and then re-appeared in phenomenological and hydrological condition. In their study, through-fall was implemented very similar to the actual phenomenon. But there was no explanation for the calculation of raindrops occurring outside the camera's FOV.

Rousseau et al. proposed a model representing the refraction of light occurring inside the raindrop [7]. For completion, the reflections should also have been considered when designing raindrops. However, since they thought the reflection was negligible enough, they implemented only refraction [8]. In their approach, rain streak textures extracted from the video were modified to match the camera of a scene and then blended into the image to make the artist's intention for the scene more effective. But, real-time performance was not guaranteed. In addition to this, dynamic scenes were not suitable for these methods because the texture must be transformed to fit the resulting screen.

Tariq further simplified Garg's rain textures and map them to the quads [3]. This processing was done on the GPU using DirectX, and each particle was rendered using a geometric shader at each frame over times. Tariq also made lights glow to show more realistic simulation of rain under the lights, but the relationship with the light source and rains was not described mathematically, therefore rain was not rendered properly as the camera moved or rotated.

Puig-Centelles *et al*. proposed a new real-time rain simulation technique in which a rain area was defined as an ellipse and all rain simulation was limited to a semi-cylindrical sub-volume [4]. Due to observer's movement, the update of new particle position was forced to inside the sub-volume, while their density was adapted to reduce the number of particles needed. Furthermore, they separated the close rain and far rain and added a transition area in between for a natural and realistic change [9]. The switch was made depending on whether the observer is in a rain area or not. If the camera is located in the rainy area from a very long distance, awkward scene could be rendered, with certain area raining and others not raining.

Unlike the above studies, the model proposed in this paper checks the position of the camera and then create a rain space for the camera in which particles are generated inside. This ensures that even if the camera keep moving or rotating rapidly, it makes illusion that rain is rendering over the entire scene. Other benefit is that can use only small number of particles to create heavy raining, which improves the rendering performance. On the top of that, this study presents a model for the interaction between particles and the light source, which can represent the light scattering effect.

Fig. 1 is a captured scene of the proposed model. The four red lines indicate the volume of view-frustum, and the intersection point where four lines meet together represents the actual camera position. In the Fig. 1, particles are generated only in the frustum of the FOV and when the camera rotates or moves, the location where the particles are generated is also moved according to the camera.

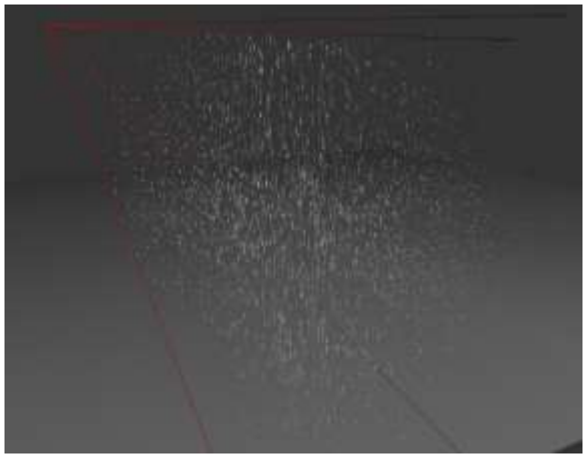


Fig. 1. Particles Rendered only in the Constrained Area.

III. Algorithm

In this study, we propose a real-time rain simulation method that creates a constrained rain space depending on the current camera so that particles representing the rain streaks are generated only in the camera frustum. Because of this, even when a small number of particles are used, suggested algorithm is able to synthesize seemingly heavy rain for users.

This chapter describes the detailed algorithm of the proposed model. The overall algorithm is briefly described in Fig. 2. Each procedure is as follows:

1) *Scene configuration:* Compose the overall scene such as background and model loading(street lamp, plane).

2) *Particles initialize and configuration:* In this model, the KTX(Khronos Texture) format is applied. More details in in Sec. 3.D, and the particle system was constructed using transform feedback from openGL [10, 11].

3) *Create constrained rain area:* Under the perspective projection, to make area for the rain fall, this method compute the 8 vertex positions of the truncated pyramid shape of the view frustum and the normal vectors of 6 faces. At the same time, it creates a virtual sphere in the frustum for enforcing a constraint to limit the area for generating particles.

4) *Set the initial particle positions in the rain area:* Set the initial positions of the particles in a constrained rain area. Particles are updated in position from top to bottom(-y).

5) *Set the rain streak colors by calculating the scattering of particles with the light source:* Depending on the relative position between particles and light soruce, the method of handling scattering from particles is different. In this paper, only spotlight is considered among the types of light sources.

6) *Real-time simulation:* Particles are created and rendered in constrained rain area. Calculating the rain space is handled by the CPU, but scattering and calculation of particle is handled by the shader.

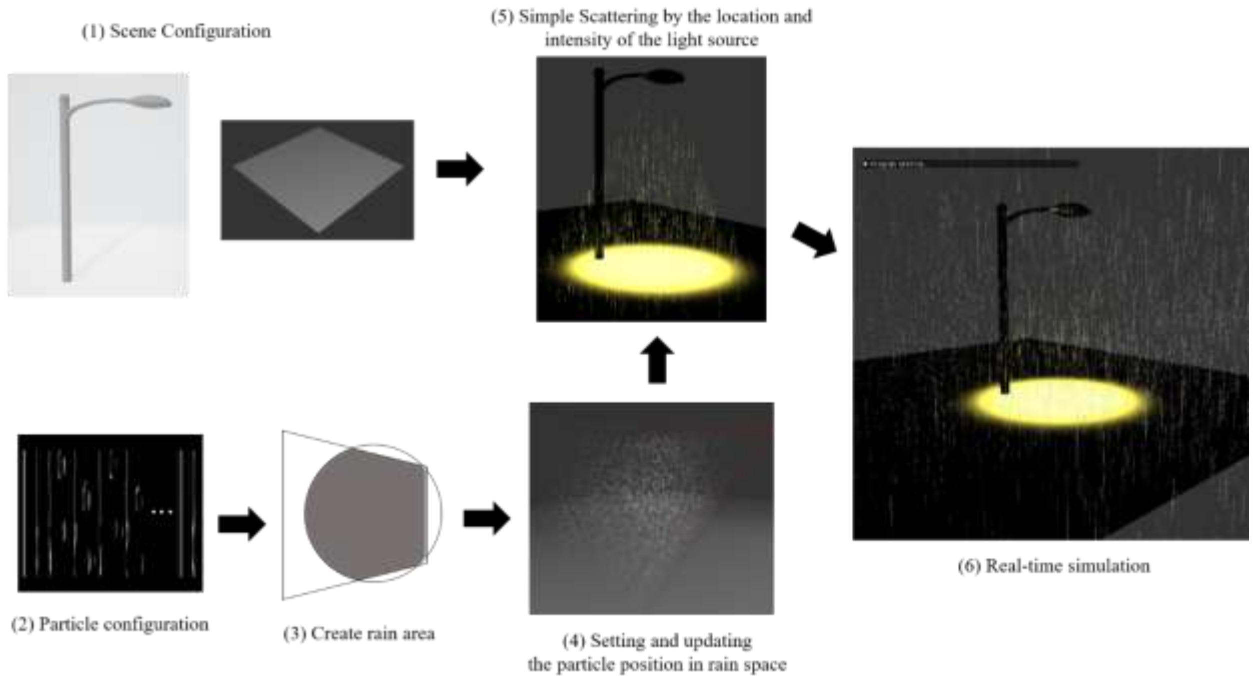

(1) Scene Configuration
(5) Simple Scattering by the location and intensity of the light source
(6) Real-time simulation
(2) Particle configuration
(3) Create rain area
(4) Setting and updating the particle position in rain space

Fig. 2. Algorithm Overview of Proposed Model.

First, explain the frustum and rain area, apply texture to particles generated only in space. After that, this paper discusses methods for light scattering in this section.

A. *Create view-dependent Rain Area*

To create the rain area from the camera's frustum, this method first need to calculate the height, width, center point of the far and near planes of the frustum. If the distance from the camera to the far plane increases, the size of the frustum increases as well. Therefore, the density of the particle created in the frustum is very low. In this case, it is different from what we wanted because algorithm has to make lots of particles to increase the density. Also, particles generated near to the far plane are rendered too small or almost invisible. Therefore, the distance from the camera to the far plane for a particle should be set as small as possible. In this paper, it was set to 50.0. After that, we need to find the positions of the 8 vertices that make up the frustum, which can be got from following formula:

$$f_{lr} = C_f - \{\vec{U} * (h_f * 0.5)\} + \{\vec{R} * (w_f * 0.5)\} \tag{1}$$

$$f_{ll} = C_f - \{\vec{U} * (h_f * 0.5)\} - \{\vec{R} * (w_f * 0.5)\}$$

$$f_{ur} = C_f + \{\vec{U} * (h_f * 0.5)\} + \{\vec{R} * (w_f * 0.5)\}$$

$$f_{ul} = C_f + \{\vec{U} * (h_f * 0.5)\} - \{\vec{R} * (w_f * 0.5)\}$$

$$n_{lr} = C_n - \{\vec{U} * (h_n * 0.5)\} + \{\vec{R} * (w_n * 0.5)\}$$

$$n_{ll} = C_n - \{\vec{U} * (h_n * 0.5)\} - \{\vec{R} * (w_n * 0.5)\}$$

$$n_{ur} = C_n + \{\vec{U} * (h_n * 0.5)\} + \{\vec{R} * (w_n * 0.5)\}$$

$$n_{ul} = C_n + \{\vec{U} * (h_n * 0.5)\} - \{\vec{R} * (w_n * 0.5)\}$$

where, C is the center point of far and near face, h, w is the height and width of each face. $\vec{U}$ is up vector of camera, and $\vec{V}$ is direction of camera. Also, $\vec{R}$ is a cross product and normalized vector of $\vec{U}$ and $\vec{V}$. Each vertex is described in Fig. 3.

Using the 8 vertices, this paper can find the normalized vectors for the four faces of the frustum, excluding the near and far plane. This is given subsequent formula:

$$\widehat{RP} = (n_{lr} - f_{lr}) \times (n_{lr} - n_{ur}) \tag{2}$$

$$\widehat{LP} = (n_{ll} - f_{ll}) \times (n_{ll} - n_{ul})$$

$$\widehat{BP} = (n_{ll} - n_{lr}) \times (n_{ll} - f_{ll})$$

$$\widehat{TP} = (n_{ul} - f_{ul}) \times (n_{ul} - n_{ur})$$

Where $\widehat{RP}$, $\widehat{LP}$, $\widehat{BP}$, $\widehat{TP}$ is a normal vector for the right, left, bottom and top faces, respectively, and each vector is normalized. Each face is also described in Fig. 3.

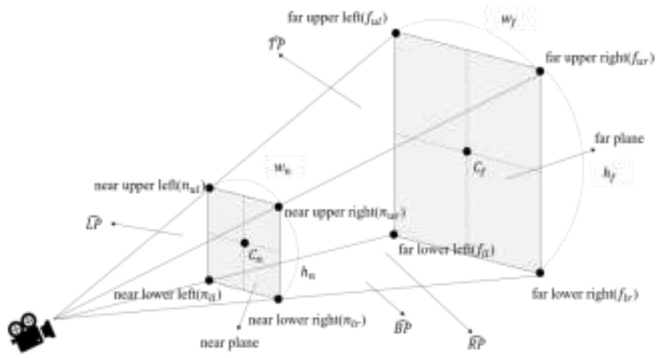


Fig. 3. Vertex and Faces of the view Frustum.

B. Virtual Sphere Setting

To constrained create space where particles will be generated, we need to create a virtual sphere that overlaps the frustum. So, we need to set the center point $C_{vs} = (x_{vs}, y_{vs}, z_{vs})$ and radius r_{vs} of the sphere. The C_{vs} of the sphere is the midpoint of the distance between the near and far faces, so that the sphere and the frustum overlap as much as possible.

As a result, as much space as possible can be defined as rain space on the frustum. Since the C_{vs} the midpoint between the near and far face, the radius of virtual sphere is defined as half the distance between two faces.

Now, algorithm need to decide the initial position $p_{par} = (x_{par}, y_{par}, z_{par})$ where a particle is created. Particle must be created inside a constrained rain space where the frustum and virtual sphere overlapped. To find this location, we first use the equation for converting from the spherical coordinate system to Cartesian coordinate system to create particles in sphere. Before this, we should get the θ and ϕ for coordinate system conversion. θ means azimuthal angle of spherical coordinate and ϕ means polar angle of spherical coordinate. This is shown in as follow:

$$\theta = 2 * PI * randmom\ seed(0,1) \tag{3}$$

$$\phi = \arccos(2 * randmom\ seed(0,1) - 1)$$

Where $\mathrm{random\ seed(a,b)}$ is a function that return on random number between a to b. And now, we can get the $\mathrm{p_{par}}$ and equation is as follow:

$$x_{par} = random\ seed(-r_{vs}, r_{vs}) \sin(\phi)\cos(\theta) + x_{vs} \tag{4}$$

$$y_{par} = random\ seed(-r_{vs}, r_{vs}) \sin(\phi)\sin(\theta) + y_{vs}$$

$$z_{par} = random\ seed(-r_{vs}, r_{vs}) \cos(\phi) + z_{vs}$$

Because of coordinate system conversion, the particles take the form of a virtual sphere. In other words, the particle is randomly set the initial position inside the virtual sphere.

So, now we need to limit the position where the particles are generated to the space where the frustum and the sphere overlap, that is, the rain area. First, select 2 of the 8 vertices of the frustum. Then, two normal vector value are got with the position of the particle as the starting point and the selected vertex as the endpoint. This normal vector is a normal vector later value to determine whether the current position of the particle is inside or outside the frustum. It is obtained as follow:

$$\widehat{PAR\_N}_{ll} = p_{par} - n_{ll} \tag{5}$$

$$\widehat{PAR\_N}_{ur} = p_{par} - n_{ur}$$

In this paper, algorithm used nll (near lower left) and nur (near upper right) vertex.

Using results of (5), calculate the dot product of two normal vectors in (5) and the four faces normal vector of the frustum besides near and far faces. Note that NLL of $\widehat{PAR\_N}_{ll}$ is the bottom left vertex of near face. So, this normal vector should be calculated with $\widehat{BP}$ and $\widehat{LP}$ vectors. Similarly, the normal vector $\widehat{PAR\_N}_{ur}$ works $\widehat{TP}$ and $\widehat{RP}$. This is shown in (5) as follow:

$$n_{ur}RP = \widehat{PAR\_N}_{ur} \cdot \widehat{RP} \tag{6}$$

$$n_{ll}LP = \widehat{PAR\_N}_{ll} \cdot \widehat{LP}$$

$$n_{ur}TP = \widehat{PAR\_N}_{ll} \cdot \widehat{BP}$$

$$n_{ll}BP = \widehat{PAR\_N}_{ur} \cdot \widehat{TP}$$

By checking whether the 4 scalar values resulting from (5) are greater than or less than 0 or not, it is possible to know that the position of the particle is inside of the frustum. If the particles are generated inside the frustum as we wish, the algorithm keeps the position unadjusted and only updates y_{par}. Conversely, if it is created outside of the frustum, then the position of the particle is moved before rendering so that it is created inside the frustum.

C. Spotlight Scattering

In order to render the rain more realistically under the various light condition, we propose a simple light scattering model between particles and lights. In our approach, we consider only spotlight because it is the type of light that affects the rain streak color significantly. For example, a spotlight such as streetlight can be found easily in real life. Other lights such as direction light and point lights are hardly seen in rainy days. Thus, this paper did not consider those lights in our study.

Fig. 4 shows three different cases when the rain streaks interact with light source. When we calculate the light scattering, the position of the particle must be decided as follows:

- Particles are located above the light source.
- Particles are under the light source but are not affected by the light.
- Particles are under the light source and are affected by the light.

Since the range that the spotlight affects has a shape of a cone, we consider particles that are inside the cone and ignore all other particles outside. To improve the physical accuracy, both 1) and 2) cases must be considered because lights may be reflected from other objects or raindrops, but since this is very insignificant and unnoticeable by the human eye, so those cases are not considered in this paper.

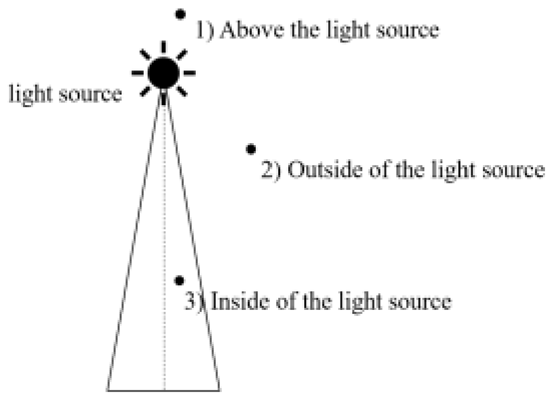

1) Above the light source
light source
2) Outside of the light source
3) Inside of the light source

Fig. 4. Conditions between Particle and Light Source.

1) Particles above the light source: Equation (7) is the equation for calculating a scalar value UD which is the dot product of the normal vector $\widehat{PL}$, which is the normalized vector from the particle to light source, and normal vector $\widehat{D}$ representing the direction of the light source.

$$ab_{light} = \widehat{PL} \cdot \widehat{D} \tag{7}$$

The ab_{light} is the value that determines whether the particle's current position is above or below the light source. If this value is higher than the height of spotlight, it means that the position of the particle is above the light source. So, the particle is not affected at all. Therefore, there is no change in particles at this case.

2) Particles under the light source: When the current particle position is under the light source, there are two cases as shown 2) and 3) in Fig. 4. Most spotlights have a cone shape. A cone is a collection of smaller or lager circles based on a point on an axis. In other words, it can be seen as a collection of circles that gradually getting smaller from the radius of the base. If the particles are in circles, they are scattered under the influence of light source. On the contrary, if particles are outside the circles, they are not affected. For this, the radius r_{circle} of the cone at the current position of the particle along the axis can be obtained using $updown$, which is the result of (7), and h_{cone}, which is height of the cone. It can be expressed as the following (8), where $base\ radius$ is the base radius of the cone:

$$r_{circle} = \left(\frac{ab_{light}}{h_{cone}}\right) * (r_{cone}) \tag{8}$$

And we can get the orthogonal distance $dist_{ortho}$ from the axis of cone to the p_{par}. $dist_{ortho}$, along with r_{circle}, is an important to know whether a particle is inside or outside the cone. To obtain $dist_{ortho}$ is expressed as (9), where $length()$ is a function that return the size of a vector as a parameter.

$$dist_{ortho} = length\{(p_{par} - h_{cone}) - ab_{light} * \widehat{D}\} \tag{9}$$

Now, we can compare r_{circle} and $dist_{ortho}$ to determine whether the p_{par} is inside or outside the cone. If r_{circle} is a larger than $dist_{ortho}$, the particle is inside the cone, which is the case as 3) in Fig. 4. And this case, the color of the particle becomes the same as the color of the light source. Also, because it is affected by light, the color of the particle appears more clearer as the intensity of the light increases.

Contrary, $dist_{ortho}$ is larger, it is the same as 2) in Fig. 4. This case, the particle does not change. The process can be expressed as the following (10) and the contents of each variable expressed in the Fig. 5.

$$if\,(r_{circle} \geq dist_{ortho}) \tag{10}$$

$$color_{par} = color_{tex} \times color_{light} \times intensity_{light}$$

$$else\ if\,(r_{circle} < dist_{ortho})$$

$$color_{par} = color_{tex} \times color_{bg} \times \alpha$$

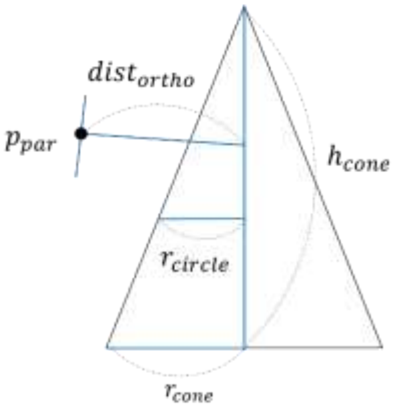


Fig. 5. Structure of Cone (Spotlight Area).

D. Texture Mapping

Garg and Nayer released their rain streaks textures as a PNG files [12]. Since this study used many textures, these files were put into one KTX file invented by the Khronos group, and then the Texture Array was used in OpenGL graphics API [13]. When we use the Texture Array, each texture corresponds to a single layer of the array. Therefore, when initializing particles, many particles are created, and they are allocated a layer for each particle. The condition for assigning a layer is random.

IV. Experiment

The proposed view-dependent rain model calculates the camera position and various parameters continuously. The experiment compares the performance of proposed model with other models after generating random numbers with a seed number. The CPU for the computer in which the experiment was conducted is Intel i7-8700, and the memory size is DDR4 16Gb * 2, a total of 32Gb. Also, the graphics card uses GTX GeForce 1080ti. All experiments were conducted in the same environment.

This study compared our proposed method with two existing models that Creus and Patow-Tariq proposed. Both models used Garg's rain streaks textures in a same way as the proposed model [3, 5]. Although the details of each algorithm may be different, it is enough to compare their FPS because three models used same rain textures. Two other models and proposed model in this paper were tested in the same environment. The changes of FPS according to the number of particles for three models are shown Fig. 7.

The proposed model in this paper, as the number of particles increased, decrease its framerate compared to the other two models. However, in the proposed model, even when a small number of particles was used, since particles were generated only in rain space within the camera's field of view, they were seemingly more than the actual number.

When we compare our result with two other method, the visual results are obvious as shown in Fig. 6. Although all three models have a fixed number of particles of 10,000, the proposed model looked to render larger number of particles than other models. This means that even with a small number of particles, we can express a large number of particles.

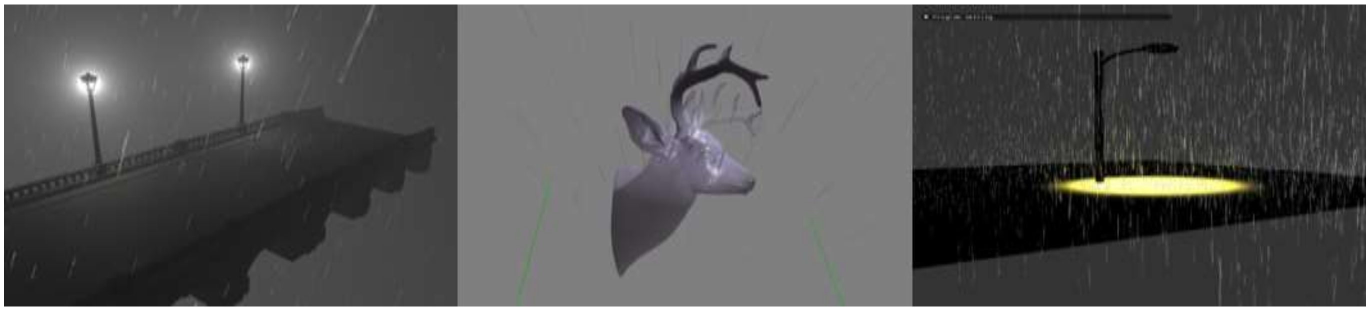


Fig. 6. (a) The Proposed Model (b) Tariq's Model (c) Crues and Patow 's Model. Three Models have the Same Number of Particles.

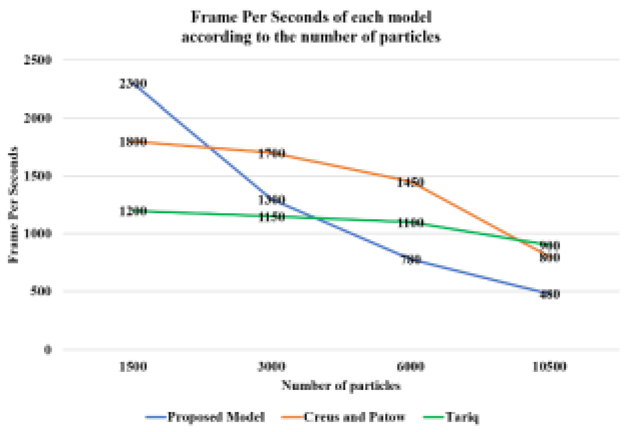


Fig. 7. Graph of Frame Change according to Particle Number.

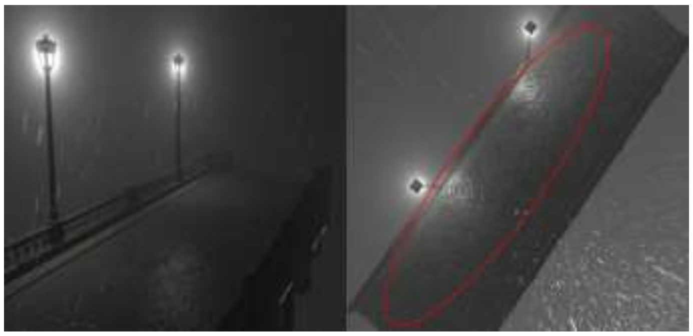
Tariq's model showed very stable performance in terms of FPS even when we increase the number of particles. In addition, a very realistic simulation result was obtained because the glow effect of the light source was considered as shown in Fig. 8(a). However, when the camera was continuously moving, at some point, particles were disappeared as shown in red circle Fig. 8(b).

The algorithm proposed by Creus and Patow, on the other hand, the FPS drops relatively in stable manner as the number of particles increases. Although not shown in the graph in Fig. 7, even when the number of particles was exceeded 10,500, real time performance was still maintained. However, as shown in Fig. 9, there were empty space in the environment where no rain was rendered when the camera is moving around. In addition, particles are keep generating and collisions are still checked even when the camera is not looking at, which degrades the overall performance.

The proposed model in this paper, as shown in the graph of Fig. 6, the FPS looks to drop higher than other two algorithms. This was caused by heavy computation on updating the position of constrained rain space, the frustum, the virtual sphere, and the particle position.

However, as shown in Fig. 10, because our algorithm makes the constrained rain space depend on the camera, even if there are a lot of changes in the camera, particles are still generating in front of the camera. This improves the visual quality of simulation.

Fig. 11 shows that the color and location of light source are fixed, and the number of particles is different. The case of (a) and (b), rendered particles are small, but it seems more than actual number. In the case of (c), rendered particles are 10,500 and it gives a feeling that it is raining quite a bit. In (d), the number of particles is the highest, 49500, and it shows that seems like it is raining a lot.



(a) Before Camera Moving. (b) After Camera Moving.

Fig. 8. Tariq's Rendering Results.

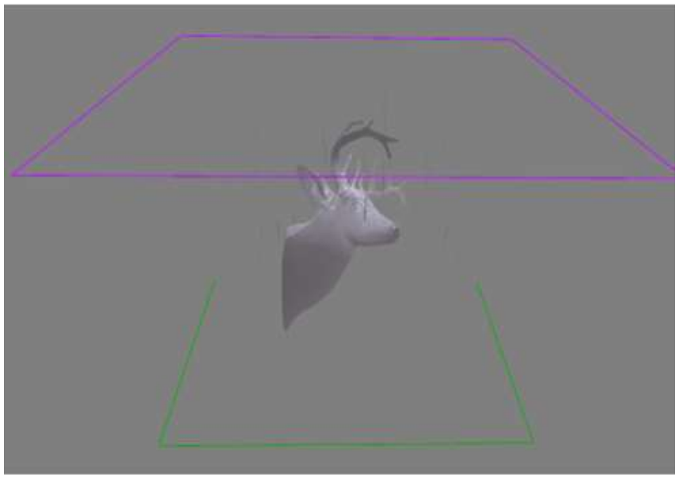
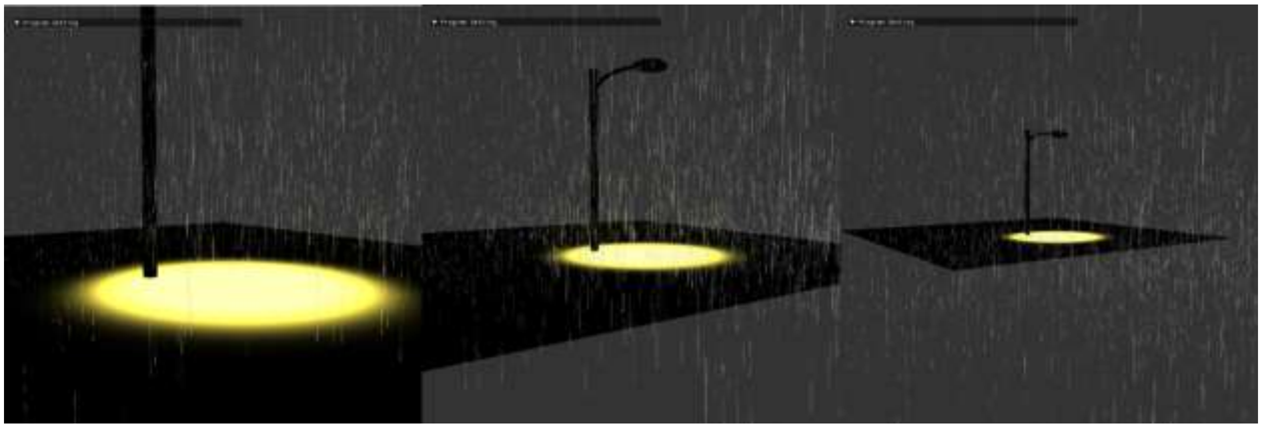
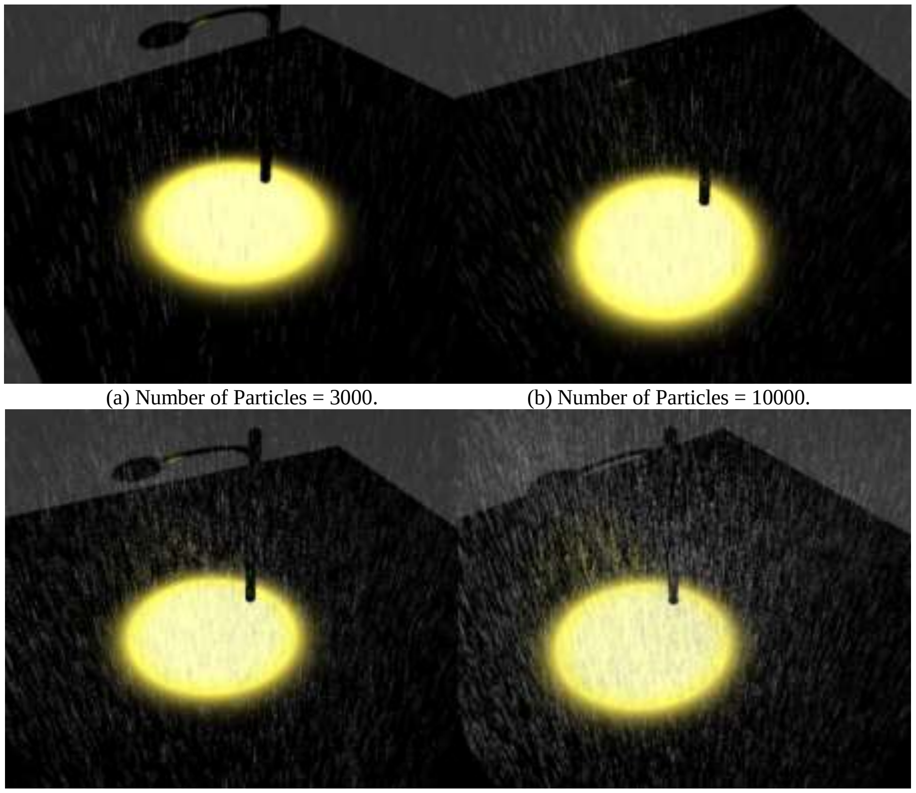


Fig. 9. Creus and Patow's Rendering Result.



(a) Camera Zoom in. (b) Camera Zoom Out. (c) Camera Zoom Out More.

Fig. 10. Proposed Model Rendering Results as Camera Zoom In and Out.



(a) Number of Particles = 3000. (b) Number of Particles = 10000.

(c) Number of Particles = 20000. (b) Number of Particles = 40000.

Fig. 11. Rendering Results of Proposed Model according to the Number of Particles.

V. CONCLUSION

As seen in previous experiment chapter, the proposed algorithm is somewhat inferior to other algorithms in terms of performance. However, in other researches, when the camera position is changing, the particle positions are rarely moving along the camera. Therefore, a very large number of particles are required and should be managed, thereby can waste the computer hardware resources. This study, however, creates a camera-dependent rain space that allows particles to be rendered only where the camera is rendered. In addition, it is possible to obtain the effect of making a large amount of rain falling even with a small number of particles.

VI. FUTURE WORK

Some limitations remain in our method, though. Particle system made with transform feedback is not intuitive to manage individual particles. Compute shader or GPGPU such as CUDA would provide much more flexibility in managing GPU threads [14, 15].

Another limitation is the way of using rain streaks textures. In our implementation, the textures did not choose according to the particular angle of light and camera conditions, although the texture database does have a lot of textures according to such parameters. Instead, this study randomly assigned one texture layer to one particle. Suggested algorithm ignored them because it turned out that it did not make a big different in terms of visual quality, though physical accuracy may be downgraded.

As future works, we have a plan to use GPGPU APIs to solve the problem of particle system and heavy computation [15, 16]. This allows us to take advantage of the flexibility of the GPU and improve the performance. Also, particle systems will be more intuitive and easier to manage. In addition, next study will consider the angle of light and camera conditions when chose the streak textures. We believe that rain simulation will be more physically accurated, realistic and effeicent.

ACKNOWLEDGMENT

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(2018R1D1A1B07048414 and 2021R1A2C1012 316).

REFERENCES

[1] K. Garg, S. K. Nayer, "Photorealistic Rendering of Rain Streaks," ACM Transactions on Graphics, vol. 25, no. 3, pp. 996-1002, 2006.

[2] Y. Weber, V. Jolivet, G. Gilet, K. Nanko, and D. Ghazanfarpour, "A phenomenological Model for Throughfall Rendering in Real-time," Eurographics Symposium on Rendering, vol. 35, pp. 13-23, 2016.

[3] S. Tarik, "Rain," Nvidia White Paper, 2007.

[4] A. Puig-Centelles, O. Ripolles, and M. Chover, "Creation Control of Rain in Virtual Environments," The Visual Computer, Vol. 25, no. 11, pp.1037-1052, 2009.

[5] C. Creus, G. A. Patow, "R4: Realistic Rain Rendering in Realtime," Computers & Graphics, Vol. 37, pp. 33-40, 2013.

[6] K. Nanko, Y. Onda, A. Ito, and H. Moriwaki, "Spatial Variability of Throughfall under a Single Tree: Experimental Study of Rainfall Amount, Raindrops, and Kinetic Energy," Agricultural and Forest Meteorology, 151, pp. 1173-1182, 2011.

[7] P. Rousseau, V. Jolivet, and D. Ghazanfarpour "Realistic Real-time Rain Rendering," Computer & Graphics, Vol. 30(4), pp. 507-518, 2006.

[8] L. Wang, Z. Lin, T. Fang, X. Yang, X. Yu, and S. B. Kang, "Real-Time Rendering of Realistic Rain," ACM SIGGARPH Sketches, pp. 156.

[9] A. Puig-Centelles, O. Ripolles, and M. Chover, "Creation Control of Rain in Virtual Environments," The Visual Computer, Vol. 25, no. 11, pp.1037-1052, 2009.

[10] W. T. Reeves, "Particle System – a Technique for Modeling a Class of Fuzzy Objects," ACM Transactions on Graphics, vol. 2, No. 2, pp. 91-108, 1983.

[11] Transform Feedback, Available online: https://www.khronos.org/opengl/wiki/Transform_Feedback (accessed on September 20, 2020).

[12] Rain Streaks Database, Available online: https://www1.cs.columbia.edu/CAVE/databases/rain_streak_db/rain_streak.php (accessed on August 10, 2020).

[13] OpenGL Array Texture, Availble online: https://www.khronos.org/opengl/wiki/Array_Texture (accessed on August 10, 2020).

[14] OpenGL Compute Shader, Availble online: https://www.khronos.org/opengl/wiki/Compute_Shader (accessed on December 10, 2020).

[15] CUDA Toolkit, Availble onlie: https://developer.nvidia.com/cuda-toolkit (accessed on December 10, 2020).

[16] OpenCL, Availble online: https://www.khronos.org/opencl (accessed on December 10, 2020).