Weather System Utilizing GPU Managed Particles and a Depth Pre-Pass Based Collision Detection Algorithm

Benjamin Skinner
Carleton University

COMP 4905, April 17, 2015
Computer Science Honours Project
Supervisor: Dr. Wilf R. LaLonde, School of Computer Science

Abstract

This project aims to create a weather particle system that supports proper collision detection between falling snow and objects rendered in the scene. Utilizing C++ and OpenGL, various buffers and shaders are used to manage the presented particle systems. Two distinct particle systems are created with this purpose in mind, and the advantages and disadvantages of each are discussed.

Acknowledgements

Thanks to all my friends and family for their support.

Special thanks to my supervisor Dr. Wilf R. LaLonde for his very useful notes on OpenGL functionality.

In-game images taken to illustrate the advantages and disadvantages of various systems of managing precipitation.

Table of Contents

List of Figures

List of Tables

Introduction

Many games have systems for managing weather. Games like The Elder Scrolls V: Skyrim and The Elder Scrolls IV: Oblivion have simplistic particle systems with no collision detection. Other games like World of Warcraft and S.T.A.L.K.E.R: Shadow of Chernobyl appear to have pre-generated depth maps that are used to prevent rain and/or snow from passing through static surfaces. Games with weather systems similar to the two latter games tend to have precipitation always falling in the same direction. This paper proposes an algorithm which both supports collision detection between particles and all objects in the scene and an arbitrary direction of particle movement. This paper specifically deals with falling snow, but similar techniques can be applied to other types of weather as well.

Particle System Overview

Two different particle systems have been implemented as part of this project. Both systems use a depth pre-pass texture to determine which particles should be removed from the scene. An example of such a depth texture is Figures 1 and 2. A depth texture is a texture used as a depth attachment for the temporary framebuffer.

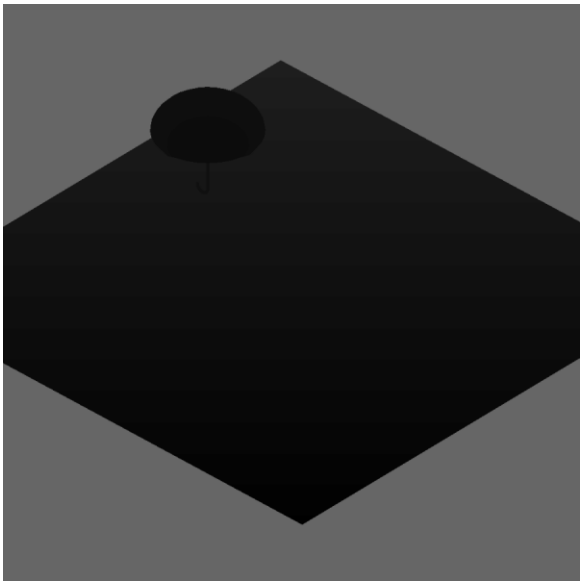

Figure 1: A depth pre-pass of the umbrella object.

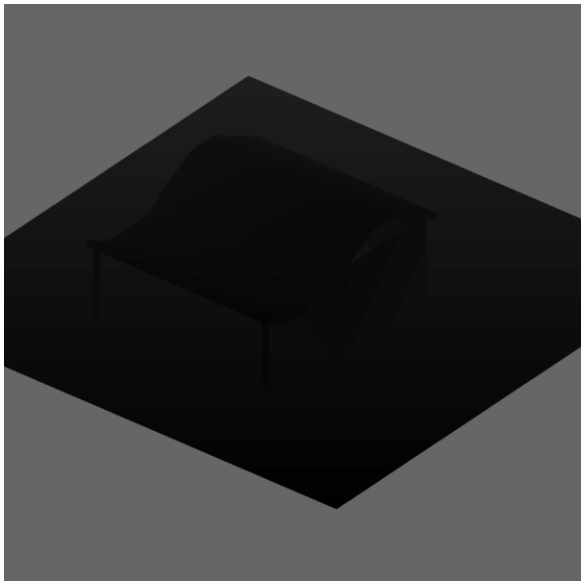

Figure 2: A depth pre-pass of the house object.

The depth pre-pass texture is generated by activating the depth framebuffer and the depthShader, and rendering each object in the scene. The resulting depth texture will inform the particle shader of the camera space depth at which to cull any given particle within the bounds of the texture. The direction

that this is viewed from is the same as the direction that the particles in the system will move. Once the pre-pass is completed, the default framebuffer is activated and the particle system is rendered with the depth texture as an input uniform in one or more of the shaders. This particle system supports the precipitation moving in an arbitrary direction. See Figure 3 for a diagram of how the depth texture works in tandem with the particle system.

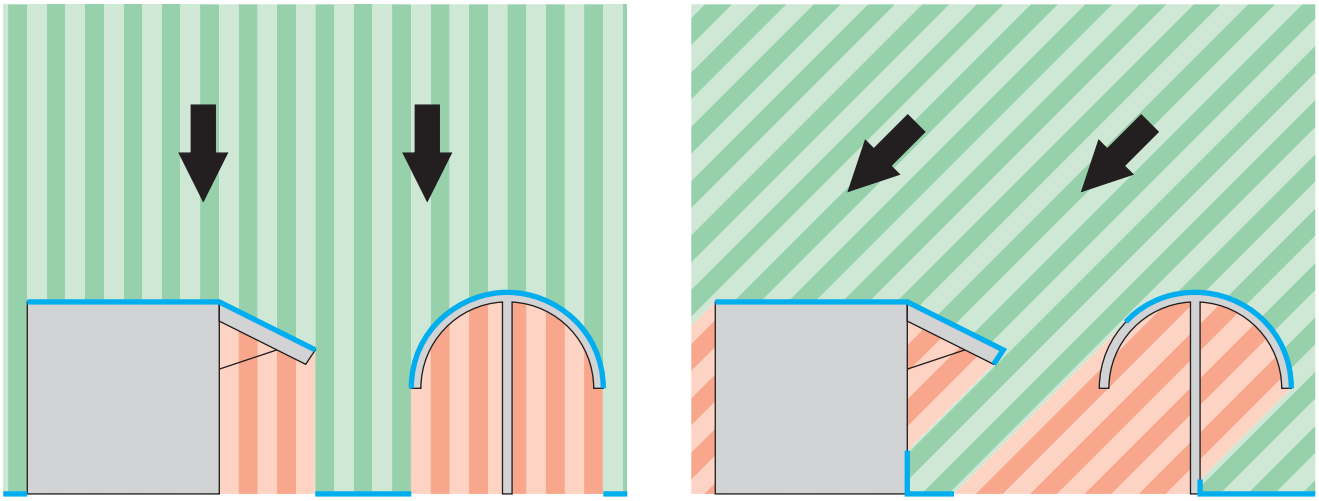

Figure 3: Depth texture diagram. Green shows valid particle locations, red show invalid particle locations. Arrows indicate particle direction. Blue line represents the depth values in the texture.

Two different particle systems have been implemented with this depth based culling method in mind. One method uses a buffer of vertices allocated when the particle system is initialized and translates and removes these vertices while rendering with a shader. This method is referred to as the Static Buffer method. The second method uses textures to store particle position data, and updates and renders the particles using shaders. This method is referred to as the Dynamic Buffer method.

The Static Buffer is a simple rendering method that allows for fast, efficient display of a large number of particles. Figure 4 describes how the Static Buffer is rendered. See Page 4 for more information about the Static Buffer. This particle system only needs to be initialized once, and only needs to be rendered with a supplied depth texture. This particle system however always has a set amount of particles in the simulation, so the amount of snow cannot be changed after the particle system is initialized. Particle sway is easily managed in this system, as the swaying itself does not impact the position of which the particles will be removed.

Static Buffer Rendering

```
0.    activate depth frame buffer (with depth texture attached);
1.    activate depth shader;
2.    set view matrix from particle spawner;
3.    set projection matrix from particle spawner;
4.    for each object o in the scene
5.        o.render();
6.    activate default frame buffer;
7.    activate default shader;
8.    set view matrix from particle spawner;
9.    set projection matrix from particle spawner;
10.   for each object o in the scene
11.       o.render();
12.   activate static particle shader;
13.   attach depth texture to static particle shader;
14.   render particles;
```

Figure 4: Pseudocode showing how the Static Buffer method of rendering particles is performed.

The Dynamic Buffer method is similar to the Static Buffer method, but an additional update function is called and different shaders are used to render the Dynamic Buffer. Figure 5 describes how the Dynamic Buffer is rendered. See Page 5 for more information about the Dynamic Buffer.

This particle system is managed by it's own update and render function. Individual particles are managed entirely within shaders executing on the GPU.

Dynamic Buffer Rendering

```
0.    update particles;
1.    activate depth frame buffer (with depth texture attached);
2.    activate depth shader;
3.    set view matrix from particle spawner;
4.    set projection matrix from particle spawner;
5.    for each object o in the scene
6.        o.render();
7.    activate default frame buffer;
8.    activate default shader;
9.    set view matrix from particle spawner;
10.   set projection matrix from particle spawner;
11.   for each object o in the scene
12.       o.render();
13.   activate dynamic particle shader;
14.   attach depth texture to dynamic particle shader;
15.   render particles;
```

Figure 5: Pseudocode showing how the Dynamic Buffer method of rendering particles is performed.

Static Buffer Particle System

The static buffer initializes a single buffer of vertices, and manipulates each vertex with a vertex shader such that each particle in the buffer appears to be moving.

The buffer is initialized with the creation of the particle system, and is never modified. Vertices are randomized within the limits of the particle system, and this leads to all particles being randomly distributed within a rectangular structure. See Figure 6 for an example of such a buffer.

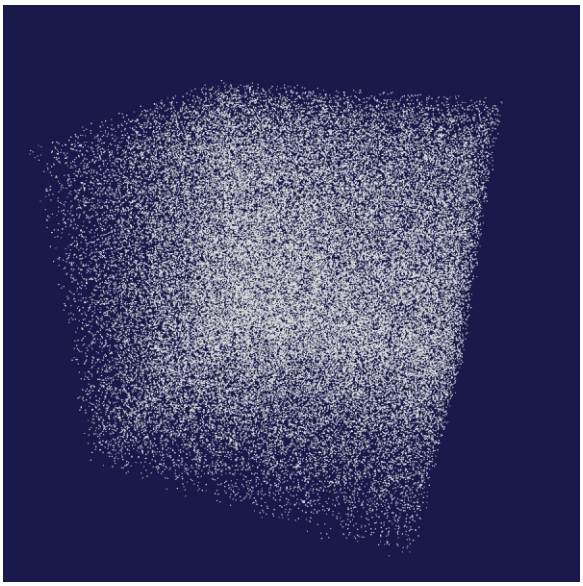

Figure 6: An example of a static buffer used in the StaticBufferParticleSystem.

Vertices are moved linearly along the z-axis based off of the total elapsed time of the application and the given speed of the particles. The position of the vertices is then clipped to stay within the boundaries of the particle system by taking the position modulo spawner Height. An X and Y offset is then calculated based on the position unique to each vertex and total elapsed time of the application, resulting in the particles following a swaying motion. A transformation matrix is applied to the buffer when rendering such that all movement along the z axis will result in movement following the direction of the particle system.

```
vec3 worldPos = vertex + vec3(0.0f, 0.0f, time*speed*g);
worldPos.z = -mod(worldPos.z, spawnerBounds.y);

float offsetX = a*sin((time+b*mod(vertex.x*c+vertex.y*d, e))*f);
float offsetY = a*sin((time+b*mod(vertex.z*c+vertex.y*d, e))*f);
worldPos += vec3(offsetX, offsetY, 0.0f);
gl_Position = worldViewProj * vec4(worldPos, 1.0f);
```

Figure 7: The shader code responsible for giving the vertices their apparent motion. Values a, b, c, d, e, f and g are arbitrary large values.

Vertices with resulting positions below the depth texture are sent to the position (0, 0, -1, 0) in camera space. This position is behind the camera, and these vertices are removed by the

GPU. The texture is accessed by using the textureLod() function, which allows for texture lookup within the vertex shader. The x and y positions of the transformed vertex position are used to calculate the uv coordinates used in the textureLod() lookup. Particle size is modified by setting gl_PointSize based on the distance from the particle to the camera, making close particles larger appear 3-dimensional.

The fragment shader simply outputs the colour of the particle. The particle system is drawn by calling the OpenGL function glDrawArrays(), with the mode set to GL_POINTS.

This particle system only needs to be initialized once, and only needs to be rendered with a supplied depth texture. This particle system however always has a set amount of particles in the simulation, so the amount of snow cannot be changed after the particle system is initialized. Particle sway is easily managed in this system, as the swaying itself does not impact the position of which the particles will be culled. This allows the particles to sway without extra particles being culled because of this sway.

Dynamic Buffer Particle System

The dynamic buffer stores particle positions within a double buffered texture setup. Particle position data is updated using the DynamicBufferUpdate shader and rendered using the DynamicBufferRender shader.

When a particle either travels too far or travels past the supplied depth texture, the particle is “killed”. The program kills a given particle by setting its position to (0,0,0). Every frame, all dead particles have a random chance to be “reborn”, or initialized to random x, y and z values. This algorithm is described by the Particle Creation section on page 7, and the Random Number Generator section on page 9. When a particle is reborn, its position is randomly set around the particle spawn location. Given a spawn rate and a particle speed, the system will tend towards a certain number of particles at any given time.

To update the particles, one texture (texture A) is sent to the update shader as an input, and the frame buffer corresponding to the other texture (texture B) is bound as the active frame buffer. The system then renders a quad facing the camera, with an orthographic projection. The projection has the same dimensions as the dimensions of the textures, such that an instance of the pixel shader is evoked

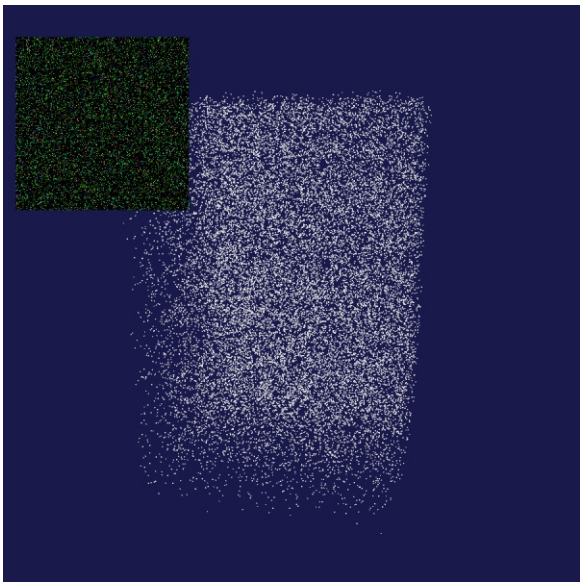

Figure 8: An example of the Dynamic Buffer Particle System. The storage texture is displayed in the top left.

for each pixel, or particle, in the system of textures. Data is used as input from texture A to write updated data into texture B. For the next frame, texture B is used for input and texture A is written to. This allows for up-to-date information to always be stored in the active texture.

To render the particles, the position must be retrieved from the storage texture. To do this, a custom vertex is used. Whereas a vertex normally contains position data, the vertices used to render the dynamic buffer system contain only uv coordinates. A buffer of these uv coordinate vertices is created alongside the creation of the particle system. This allows for an instance of the render vertex shader to be evoked once for each pixel, or particle, in the storage texture. The position data in the storage texture is retrieved with a textureLod() lookup. If the position data in the storage texture reads (0, 0, 0), indicating a dead particle, the render vertex shader sends the camera space position vector to (0, 0, -1, 0) which is behind the camera. Otherwise, the position is transformed like any other ordinary vertex. As with the Static Buffer method, snow particles are given an apparent swaying motion in the render vertex shader. Instead of unique initial position, the unique uv coordinates of each vertex are used to randomize the particle sway. Particle size is modified by setting gl_PointSize based on the distance from the particle to the camera, making close particles appear larger, and as a result more 3-dimensional. The render fragment shader simply outputs the colour of the particle. The particle system is drawn by calling the OpenGL function glDrawArrays(), with the mode set to GL_POINTS.

This particle system only needs to be initialized once in the application, and then it will manage itself with update() and render() function calls. Unlike other particle systems, there does not need to be any work done by the CPU to specifically insert new particles into the buffer. Particles are created, updated and removed all in shaders executing on the GPU. This particle system also allows for a variable amount of particles in the simulation at any given point in time, determined by maximum particle count, particle spawn rate and the speed of the particles.

Dynamic Buffer Particle Creation

To find the approximation for a percentage chance for a dead particle to be reborn, the known relative chances of different time-per-frame spawn rates was used. Given a set period of time, the percent chance of a particle being reborn should be the same throughout said time span. Alternatively, the percent chance of a particular particle not spawning should be equal given the same time interval. By comparing the chance of a particle to not spawn in a given time-per-frame and the chance of the same particle to not spawn in half of this given time-per-frame, the recurrence in Figure 9 is found. In Figure 9 *x* is the chance to spawn at an arbitrary time-per-frame and *y* is the equivalent chance to spawn at half the time-per-frame that corresponds to *x*. Given the same time period, the event corresponding to *y* will occur exactly twice for every equivalent event occurring for *x*.

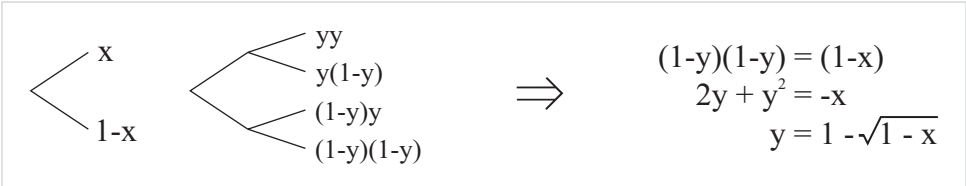

Figure 9: Calculation of the recurrence used to approximate the creation chance of a given particle given an arbitrary elapsed time.

Using this recurrence, data points were generated by repeatedly halving the time per frame, resulting in the values given by Table 1.

Chance/Frame	0.1	0.051317	0.025996	0.013084	0.006563	0.003287	0.001645	0.000823	. . .	3E-15
TPF (s)	1.0	0.5	0.25	0.125	0.0625	0.03125	0.015625	0.007813	. . .	2.84E-14

Table 1: Values obtained using the recurrence given in Figure 9.

When plotted on a graph, these data points give the trend line shown in Figure 11 and Figure 12. The trend line calculated in Figures 11 and 12 are of a quadratic form. For the purposes of this program, a similar linear approximation was used instead for the sake of simplicity. This led to the equation found in DynamicBufferUpdate.frag, as shown in Figure 10. This equation is called for every dead pixel on every frame. Each dead pixel has a chance to be reborn, governed by the elapsedTime and the spawnRate. If the random number is less than the right hand side of the equation, a the dead particle is initialized randomly.

```
return ((seed % 100000000u) < uint(0.5f+100.0f*spawnRate*elapsedTime));
```

Figure 10: The determining equation on whether a particle is reborn or not. Seed is a randomized value, spawnRate is an arbitrary value and elapsedTime is 1 / frame rate.

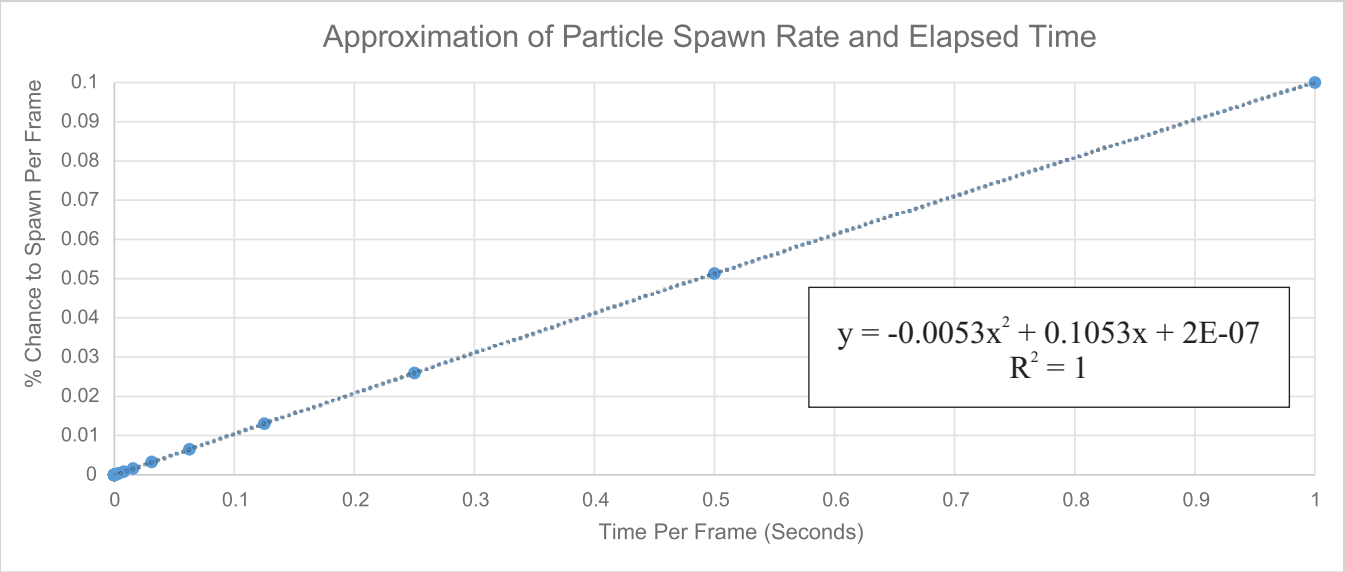

Figure 11: Approximation for particle creation rate at a given Time Per Frame

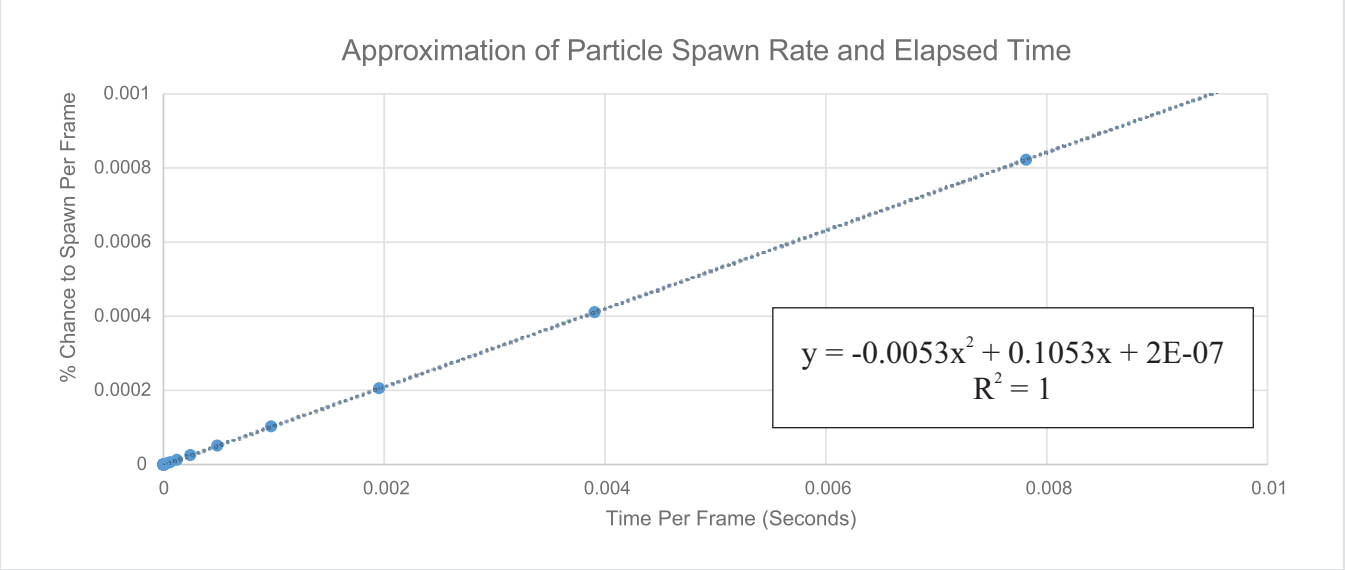

Figure 12: Approximation for particle creation rate at a given Time Per Frame, with a smaller range

Random Number Generator

In a typical random number function, the seed that is used to generate consecutive numbers is modified with each execution of the random number function. In this way, consecutive calls of the random function generate seemingly random numbers, as they are each given a different starting seed. When working in a pixel shader, this sequential calling of the random function is not possible, as all instances of a given pixel shader execute in parallel with one another. The only difference between the pixels in a given shader is the attributes from the vertices that said pixel is associated with. In DynamicBufferUpdate, the only attribute unique to each pixel is its corresponding uv coordinate.

The initial seed of each pixel is determined by using a combination of the unique uv value of each vertex, multiplied by large values, and the total elapsed time that the program has run for. Together these are used to provide a unique random offset between each pixel. This generates a seed unique to each vertex, which allows the random() function to generate unique results for each vertex. The seed then undergoes several bit shift operations as dictated by the Tiny Encryption Algorithm (TEA). See Figure 13 for this randomization function.

```
uint random(uint seed)
{
	uint sum, delta;
	uint result = 0u;
	for (int n = 0; n < 3; n++) {
		sum = (delta + sum) % a;
		result = ( ((seed << b) + c) ^ (seed + sum) ^ ((seed >> d) + e) + result) % f;
	}
	return result;
}
```

Figure 13: The random function derived from the Tiny Encryption Algorithm.
Values sum and delta are initialized to arbitrary large integer values.
Values a, b, c, d, e and f are arbitrary large unsigned integer values.

As detailed in *GPU Random Numbers via the Tiny Encryption Algorithm* by Fahad Zafar, Marc Olano and Aaron Curtis, the Tiny Encryption Algorithm can be used to generate pseudo random numbers given unique seeds. See Figure 15 in Appendix A for an example output of the random particle creation. The seed and random functions are used to determine when a given particle will be reborn, and where the given particle will be placed within the range of the particle spawner.

Run Time Comparisons

The Static Buffer is quicker to render than the Dynamic Buffer is to update and render. See Figure 14 for a graphical depiction of the average time-per-frame of the Static Buffer and the Dynamic Buffer for a given number of particles in the simulation. See Table 2 for the data values used in Figure 14. Testing was performed on a AMD A4-4355M APU @ 1.90GHz, at a resolution of 1280 x 720 pixels.

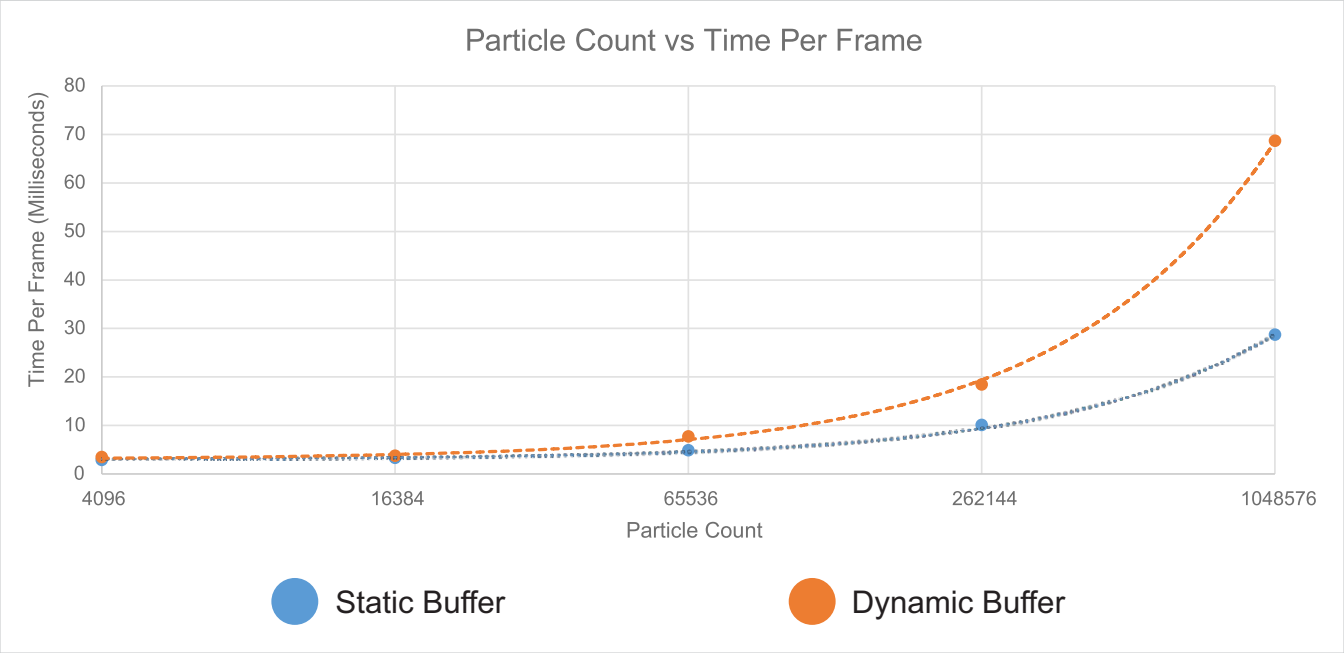

Figure 14: Particle Count vs Time Per Frame of the Static Buffer and Dynamic Buffer systems. Note that the x-axis follow a logarithmic scale. Time Per Frame increases linearly with Particle Count.

Particle Count	4096	16384	65536	262144	1048576
Static Buffer Time Per Frame (ms)	2.93	3.34	4.87	10.10	28.73
Dynamic Buffer Time Per Frame (ms)	3.52	3.78	7.77	18.46	68.68

Table 2: Data for Figure 14 of the Static Buffer and Dynamic Buffer systems.

It is clear from Figure 14 that the Dynamic Buffer performs worse than the Static Buffer. It is also clear that as the number of particles in the simulation increases, the performance of the Dynamic Buffer decreases at faster rate than the performance of the Static Buffer.

Conclusions

The Static Buffer is an efficient method of displaying a constant snowing effect. The Static Buffer outperforms the Dynamic Buffer at any given number of particles, and if a changing number of particles in the simulation is not required the Static Buffer seems to be the more effective choice. It is also worth noting that if contact between the snow and dynamic objects is not needed, then the depth texture need only be generated whenever the direction of the snow changes to save computation time.

The Dynamic Buffer is an interesting particle system that, for the most part, manages itself. The Dynamic Buffer also manages to avoid wasting time having the CPU manually inserting new particles into the buffer, which in OpenGL would require some manipulations of the buffer. This in turn manages to reduce the overhead costs of the algorithm. Additionally, as particle movement and particle removal conditions can be freely changed by using different update and render shaders, this particle system can be used for a wide variety of visual particle effects.

References

Zafar, Olano and Curtis (2010) GPU Random Numbers via the Tiny Encryption Algorithm, pp 4
http://www.csee.umbc.edu/~olano/papers/GPUTEA.pdf

Appendix A

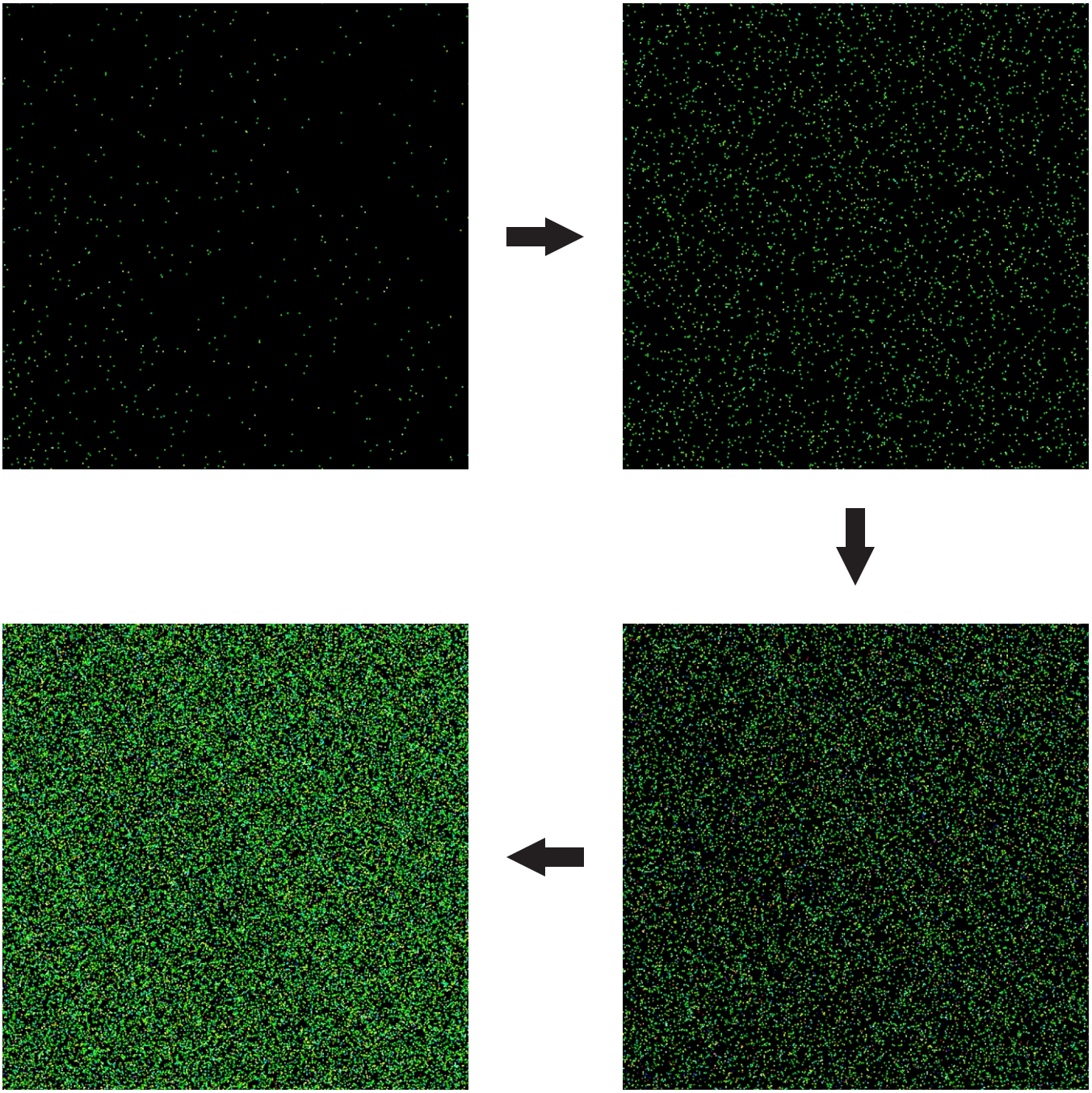

Figure 15: An example result of the pseudo random particle creation algorithm.

Appendix B

Figure 16: Snow falling through a solid surface in The Elder Scrolls IV: Oblivion.

Figure 17: Rain falling through a solid surface in The Elder Scrolls IV: Oblivion.

Figure 18: Snow falling through a solid surface in The Elder Scrolls V: Skyrim

Figure 19: Rain falling through a solid surface in The Elder Scrolls V: Skyrim

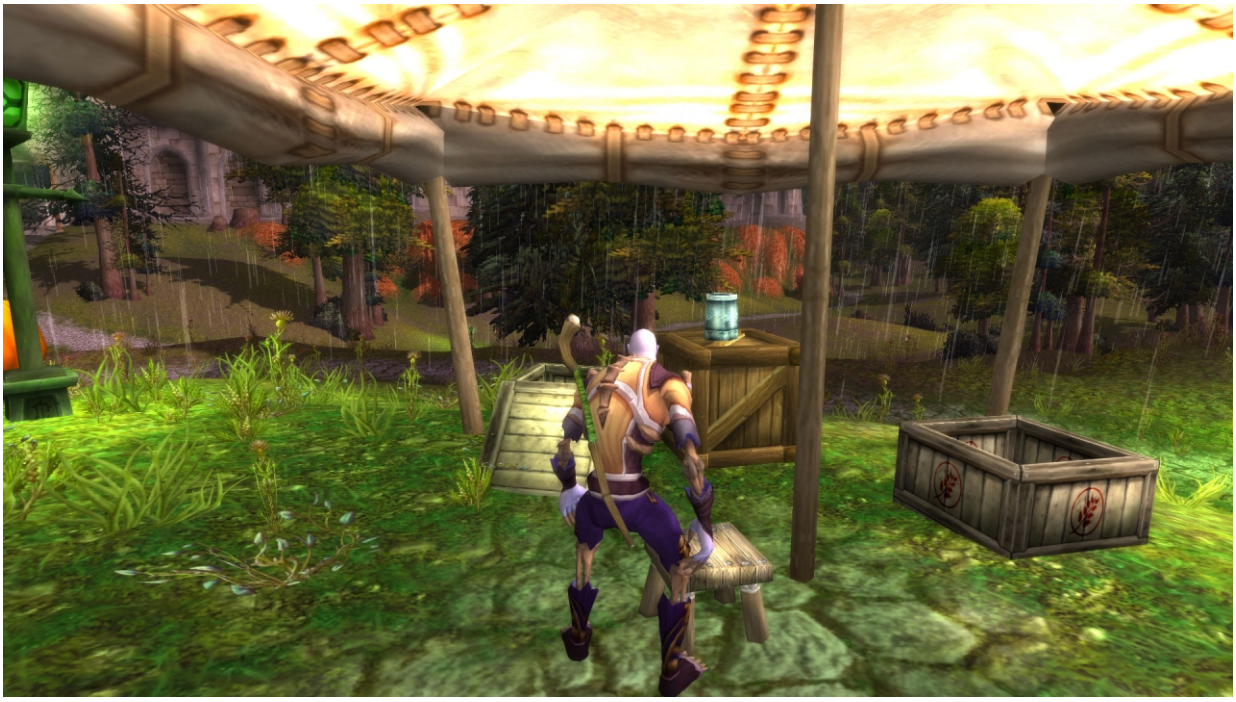

Figure 20: Rain not falling through a solid surface in World of Warcraft

Figure 21: Rain falling through a moving solid surface in World of Warcraft

Figure 22: Snow not falling through a tent in World of Warcraft

Example Videos showcasing various weather systems in motion

SoC Warehouse: http://www.youtube.com/watch?v=Db_bh-c31gU
SoC Tunnel: http://www.youtube.com/watch?v=u1Z74i5_Q0U
SoC Tube: http://www.youtube.com/watch?v=ApmJyYbfmfk
SoC Tree: http://www.youtube.com/watch?v=c92wosNGpWs
SoC Train: http://www.youtube.com/watch?v=3kUunOSqEhA
SoC Bus Stop: http://www.youtube.com/watch?v=yjLlp14w6Qg
SoC Bus: http://www.youtube.com/watch?v=pxRC3lCJLfA
SoC Bunker: http://www.youtube.com/watch?v=LVGUu6rhRbI
TESV Snow Moving: http://www.youtube.com/watch?v=YuJeTja-EEM
TESV Snow Collision: http://www.youtube.com/watch?v=M05cVTFeBds
TESV Rain Moving: http://www.youtube.com/watch?v=43UcHRwt_Q0
TESV Rain Collision: http://www.youtube.com/watch?v=hv-wvxvD-6I
TESIV Snow Moving: http://www.youtube.com/watch?v=lYElQ-H6x5w
TESIV Snow Collision: http://www.youtube.com/watch?v=Gcydlwdvy_E
TESIV Rain Moving: http://www.youtube.com/watch?v=WFgpdiOuHBw
TESIV Rain Collision: http://www.youtube.com/watch?v=RanpazYV3qs
WoW Rain Movement: http://www.youtube.com/watch?v=28jrDRIuJbE
WoW Rain Tent: http://www.youtube.com/watch?v=pk6CjmTCA00
WoW Rain Zepplin: http://www.youtube.com/watch?v=RiAWqs4RYQU
WoW Snow: http://www.youtube.com/watch?v=trH4VUBszok