Computer Graphics: An Introduction

By Michael Berg

I. Introduction

Particle systems are used in all forms of media as a powerful tool to replicate real world phenomena (see figure I), or create an effect that otherwise doesn't occur in our natural world (see figure II). In a way, particle systems are powerful tools to capture the imagination and entrance the mind, with its random shape and infinite variation, and it should be that computers could generate something so beautiful, and creates an interesting intersection between opposites, logic and art. But this powerful tool was born out of purpose, as a solution to a problem, and grew into a solid core of concepts with many variations and is a widely explored research area, both visual and optimization capacities.

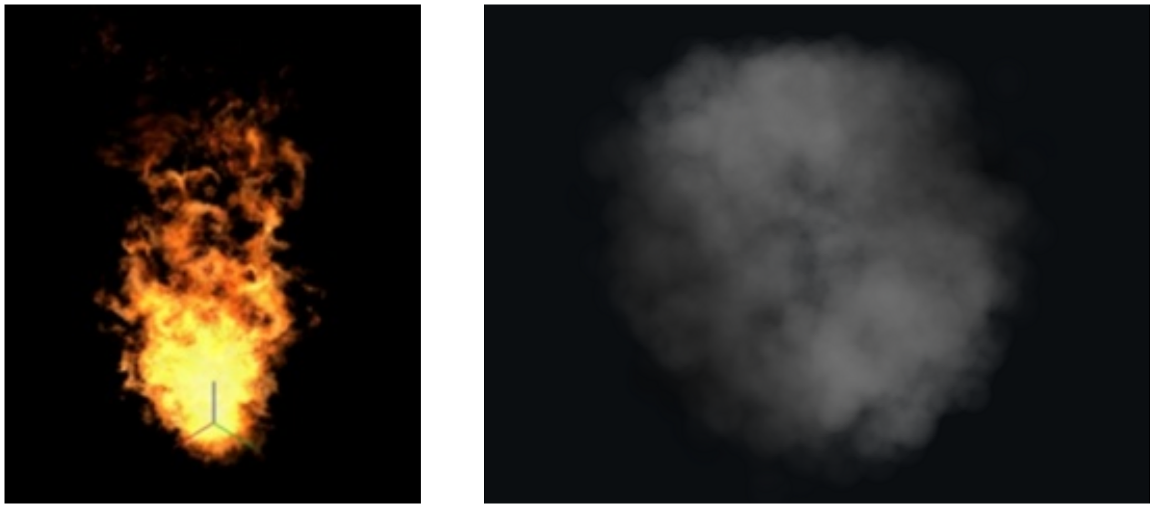


Figure I – Replicating real world phenomena with Particle Systems

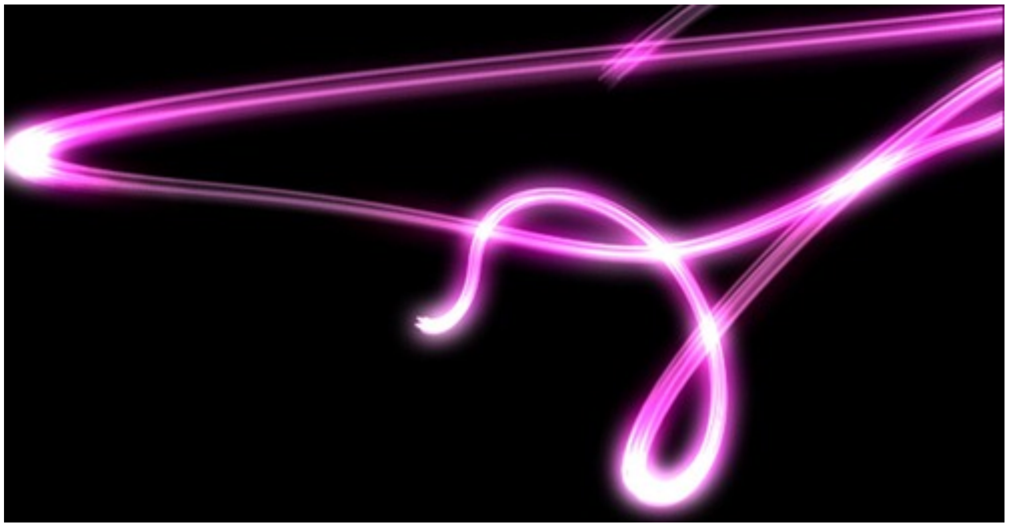


Figure II – Simulated phenomena that don’t occur in our natural world

II. Particle System Terminology

As the name implies, “[a] particle system is a collection of a number of individual elements or particles” (Video Games Technologies 5). But what exactly is a particle, or rather what can it be? A particle in its simplest form is merely a single pixel on the screen, but a particle can also be a line or even a polygon (see figure III). Each particle has several attributes that control its characteristics:

- position
- velocity
- life span
- size
- color
- owner
- weight, acceleration, etc

The position data member stores the x, y and z coordinates of a given particle. Velocity is the speed with which the particle is moving. Life span is the amount of time the particle stays a part of the particle system before being deleted from the particle system. Size gives the particle its volume and subsequently how much screen space it takes up. Color is the color the particle has and what Red, Green, Blue value it is given on the screen. Owner is an identifier that tells which particle system a given particle belongs to if multiple particle systems exist within a given scene. There are other such attributes that can be added to particles, such as weight and acceleration to simulate physics and gravity. Particles can also be textured, to make simple points or polygons resemble a particular effect or phenomena, such as smoke (see figure I).

Particle systems can control many aspects of the particles it is made up of, and the systems themselves have many attributes:

- Particle List
- Postion
- Emission Rate
- Forces
- Current State
- Color delta, weight delta, etc

The particle list is the collection of particles generated by a single particle system. The position is the position of the emitter, the point where particles are created. Emission rate is the rate at which new particles are created into the scene. Forces are any specific velocity or acceleration that should be applied to any particles generated by the particle system. Current state is a variable that determines if the particle system is emitting particles or not. Other attributes can be characteristics that change the characteristics of the particles created over time, for example changing color, weight or acceleration over time. It merits mentioning that some of the attributes of a particle system can be inherited by the particles it generates, but others, such as emitter position and emission rate, cannot be inherited by particles.

One of the most notable characteristics of particle systems is their ability to be unique each and every time they are generated through the use of various randomizing functions. If these randomizing functions are implemented for a particle system, the result is "[a] particle system [that] is not deterministic, its shape and form not completely specified" (Video Games Technology 5). This is one of the main advantages of using a particle system, to use one asset that is infinitely varied, to give the viewer a new and fresh experience, in the case of a simulation or video game.

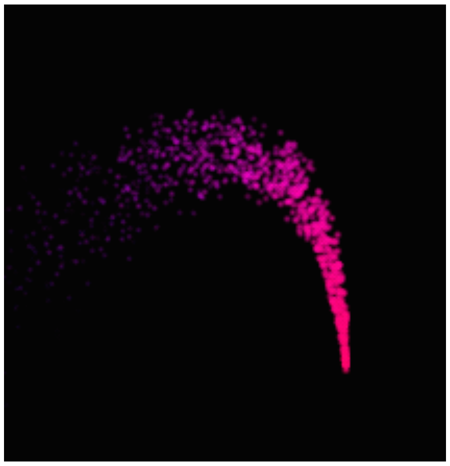
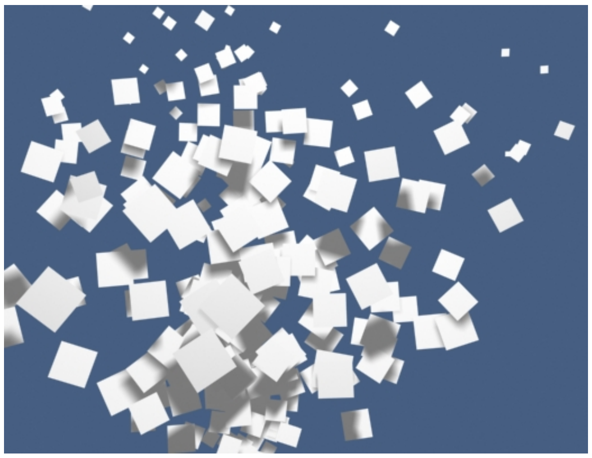
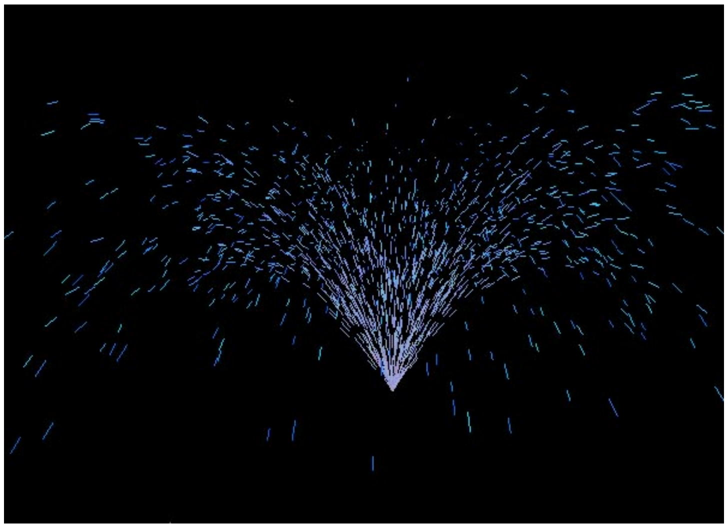




Figure III – Point, Polygon and Line Particle Systems

III. History

Necessity is the mother of invention, so the saying goes, and particle systems were initially conceived to fill a gap in computer graphics that didn't have a set or widely accepted way of modeling a class of fuzzy objects. A programmer by the name William Reeves came up with the idea while working on the special effects for the movie Star Trek: the Wrath of Khan.

> "The sequence depicts the transformation of a dead, moonlike planet into a warm, earthlike planet by an experimental device called the Genesis bomb. In a computer-simulated demonstration, the bomb hits the planet's surface and an expanding wall of fire spreads out from the point of impact to eventually 'cleanse' the entire planet…The wall-of-fire element in the Genesis Demo was generated using a two-level h9ierarchy of particle systems. The top-level system was centered at the impact point of the genesis bomb. It generated particles which were themselves particle systems" (Reeves 365).

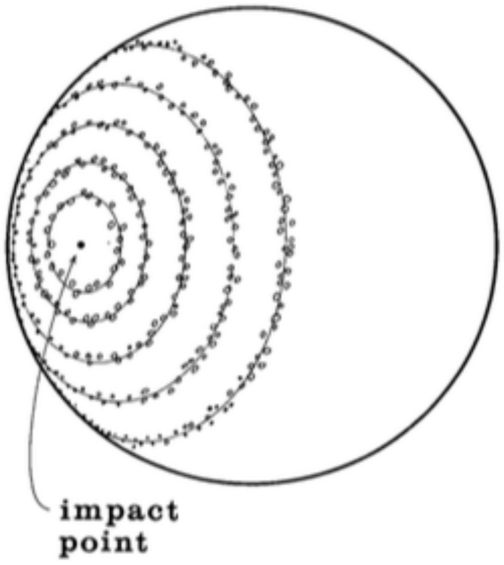


Figure IV – Diagram of Genesis Bomb Particle System

Reeves goes to explain that the first level particle system has a set of attributes that its child particle systems inherit. For instance, each ring of fire is a bunch of particle systems that belongs to the particle system at the point of impact. Each particle system also has a distance from the point of impact particle system that is passed to its child particle systems to determine how many particles that should be generated. The child particle systems were all random,

> [their] average color and the rates at which the colors changed were inherited from the parent particle system, but varied stochastically. The initial mean velocity, generation circle radius, ejection angle, mean particle size, mean lifetime, mean particle generation rate, and mean particle transparency parameters were also based on their partner's parameters, but varied stochastically. Varying the mean velocity parameter caused the explosions to be of different heights"(Reeves 366).

And the color of the particles varied depending on the particle's distance from the emitter. "When many particles covered a pixel, as was the case near the center and base of each explosion, the red component was quickly clamped at full intensity and the green component increased to a point where the resulting color was orange and even yellow. Thus, the heart of the explosion had a hot yellow-orange glow which faded off to shades of red elsewhere" (Reeves 367). The end product of Reeves' work is awe inspiring to say the least (see Figure V), and started a new trend in computer graphics that is heavily used today, and considered a cornerstone of effects rendering.

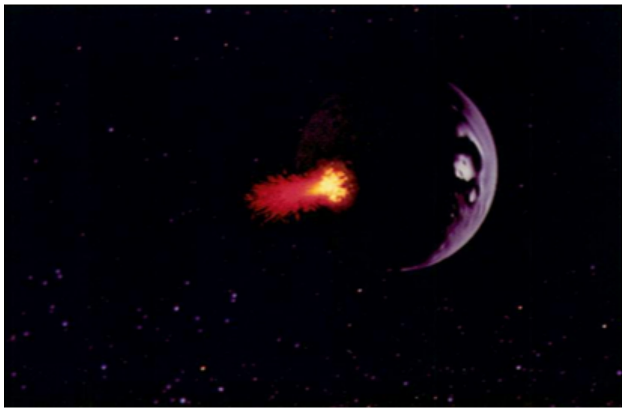


Figure Va – Genesis bomb's initial impact point

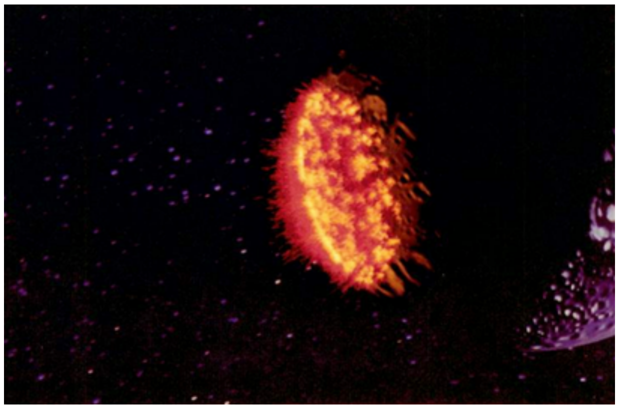


Figure Vb – Rings expanding from the impact point

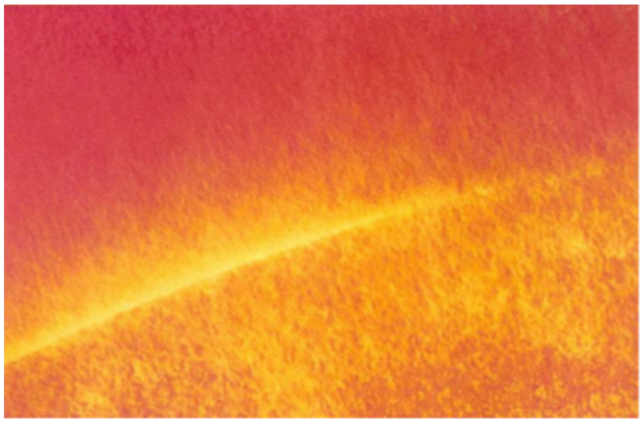


Figure Vc – Coloring of the explosion particle systems

IV. Technical Specifications (Simulation and Rendering)

Particle system implementation can vary widely given all the different parameters that can be set both for the system and the individual particles. But every implementation breaks down into two phases: the simulation stage and the rendering stage.

At the beginning of the simulation stage, a "number of new particles that must be created is calculated based on spawning rates and the interval between updates, and each of them is spawned in a specific position in 3D space based on the emitter's position and spawning area specified. Each of the particle's parameters (i.e. velocity, color, etc.) is initialized according to the emitter's parameters" (Wikipedia). Typical implementations for spawning rates would be a parameter that tells the max number of particles a system can have at anyone time and how many can be spawned at each update. This simple implementation makes sure the particle system doesn't get too big or unexpectedly slow down the platform that is rendering it. The particle system's emitter usually sits at the same position as the coordinates of the particle system, as a nonvisible point on the screen most times.

> At each update, all existing particles are checked to see if they have exceeded their lifetime, in which case they are removed from the simulation. Otherwise, the particles' position and other characteristics are advanced based on a physical simulation, which can be as simple as translating their current position, or as complicated as performing physically accurate trajectory calculations which take into account external forces (Wikipedia)

Particles stored in a particle system are typically stored as a doubly linked list data structure, so it is easy to traverse the list to check if a particle has expired, update a parameter, etc.

After the simulation stage is the rendering stage, where "each particle is rendered, usually in the form of a textured billboarded quad…However, that is not necessary; a particle may be rendered as single pixel in small resolution/limited processing power environments" (Wikipedia). A textured, billboarded quad is a 2d shape with 4 sides, rotated to face the camera and textured to appear as something else (such as smoke or fire). But there are even more, high power solutions such as "3D mesh objects [which] can 'stand in' for the particles-a snowstorm might consist of a single 3d snowflake mesh being duplicated and rotated to match the positions of thousands or millions of particles" (Wikipedia). This stage can be the most costly part of an entire particle system in terms of computing power, especially for particles that are meshes with many vertices and/or textured with high-resolution textures. Great care must be taken with how particle systems are implemented when complicated particles are involved.

V. Current Challenges and Research

As the capabilities of computer hardware grow, so does the demand for more detailed visual effects for video games and movies, and more detailed simulations for research and development of innumerable projects. Thus, particle systems with particle numbers ranging in the hundreds of thousands are required to meet this demand, but optimizing particle systems with so many members has been a challenge faced by researchers today, and has been tackled by the researchers at Shanghai University. They have approached the problem of optimizing large-scale particle systems with “optional integration algorithms based on CUDA (Compute Unified Device Architecture) for both graphic and scientific simulation” (Li 572).

This optimizing and speedup of calculation time is packaged into an “API library for particle systems [that] has backends that support both CPU and GPU (via CUDA) as execution engines” (Li 572). This library assumes a similar particle system described in Technical Specifications (section IV), and uses the capabilities of the CUDA language to distribute the spawning, updating, despawning and all relevant calculations between the CPU and the GPU. This method of distribution can be further optimized by using “optional different integration algorithms in updating particle attributes. According to the need for accuracy and stability in the application” (Li 574). CUDA truly brings out the computing power of a NVIDIA GPU when It comes to the amount of calculations and memory needed to simulate large scale particle systems by parallelizing many of these simple calculations on particles to rapidly increase performance by splitting particles into particle groups, “which [are] a set of particles that are acted upon by the same forces. Several particle groups may exist at the same time, but all API functions are only applied to the active particle group. Action are the functions in the API that modify the attributes of particles in the active particle group" (Li 573).

It turns out that the research group at Shanghai University found success with optimizing and parallelizing calculations for large-scale particle systems, reporting up to "27 times speedup over CPU (taking into account all host and GPU data transfers), and reach the goal of simulating a real-time million-particle system in conditions of using Euler or Verlet integration" (Li 576). This breakthrough will tremendously help researchers and developers in the future when simulating large scale celestial phenomena, data visualization and visual effects.

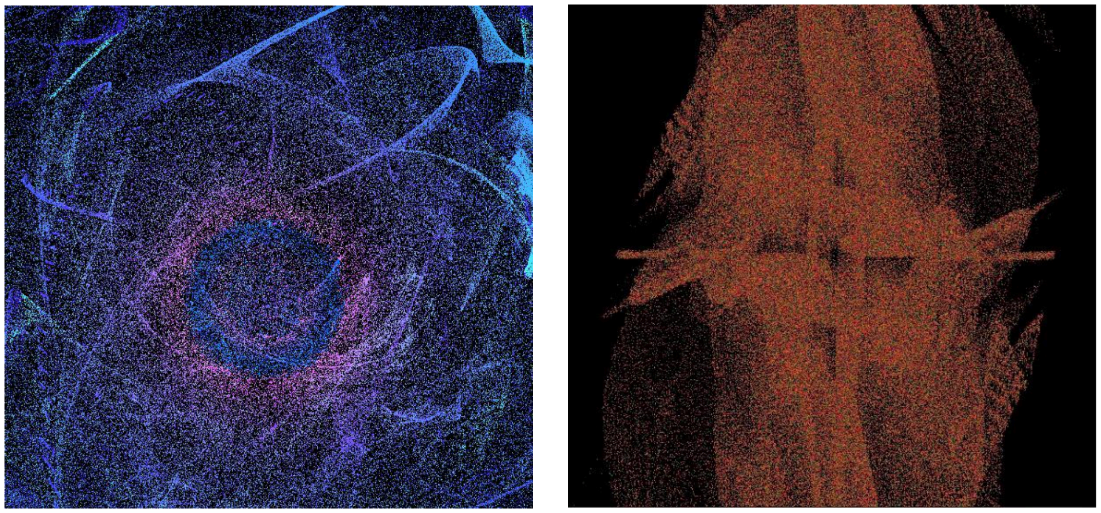
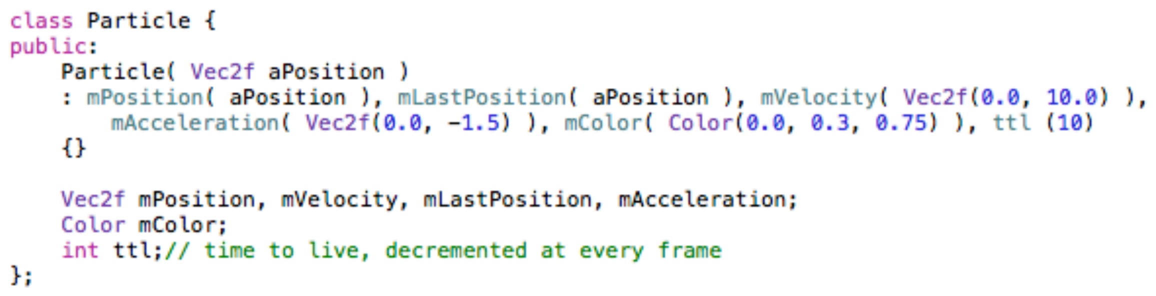


Figure VI – Million Particle System Demos using

VI. Implementation

I've utilized the Cinder libraries for C++ to implement my particle system. Cinder abstracts away the lower level functionality of openGL into various function calls, so as to focus on overall visual aesthetic of the program. This library is open source and the code can be free edited to fit specific needs.

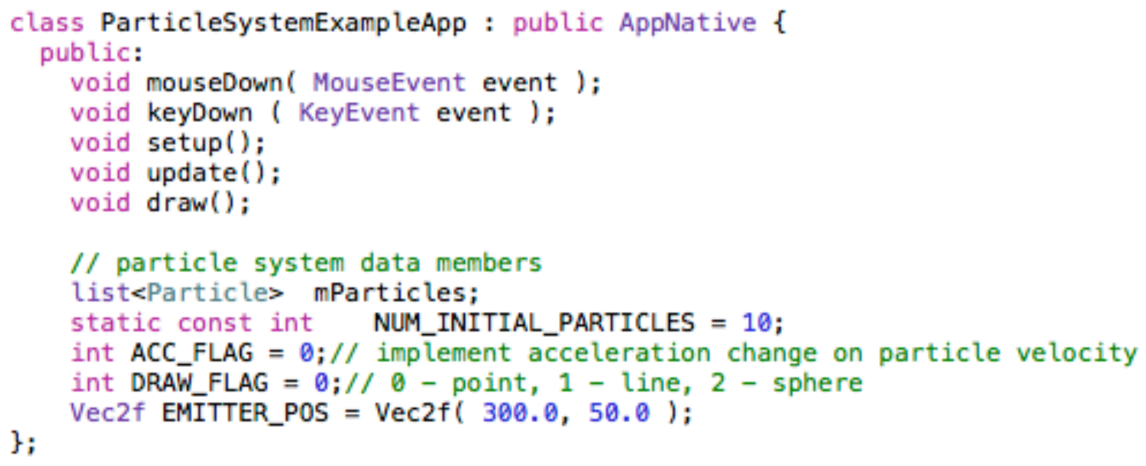
With Cinder, I've implemented a simple particle system simulating randomized particles that are instantiated with various variables (see figure VII) that affect the particle's place on the screen, its appearance and time on the screen before it disappears



```
class Particle {
public:
    Particle( Vec2f aPosition )
    : mPosition( aPosition ), mLastPosition( aPosition ), mVelocity( Vec2f(0.0, 10.0) ),
        mAcceleration( Vec2f(0.0, -1.5) ), mColor( Color(0.0, 0.3, 0.75) ), ttl (10)
    {}

    Vec2f mPosition, mVelocity, mLastPosition, mAcceleration;
    Color mColor;
    int ttl;// time to live, decremented at every frame
};
```

Figure VII – Implementation of Particle

This class is instantiated numerous times to create the various on screen elements that comprise the particle system. The particle system data members are used to store particle information in a doubly linked list, and various integers for things like how many particles start in the system on instantiation, to other things like if a particle's acceleration is applied to its velocity during the update method of the particle system (see figure VIII).

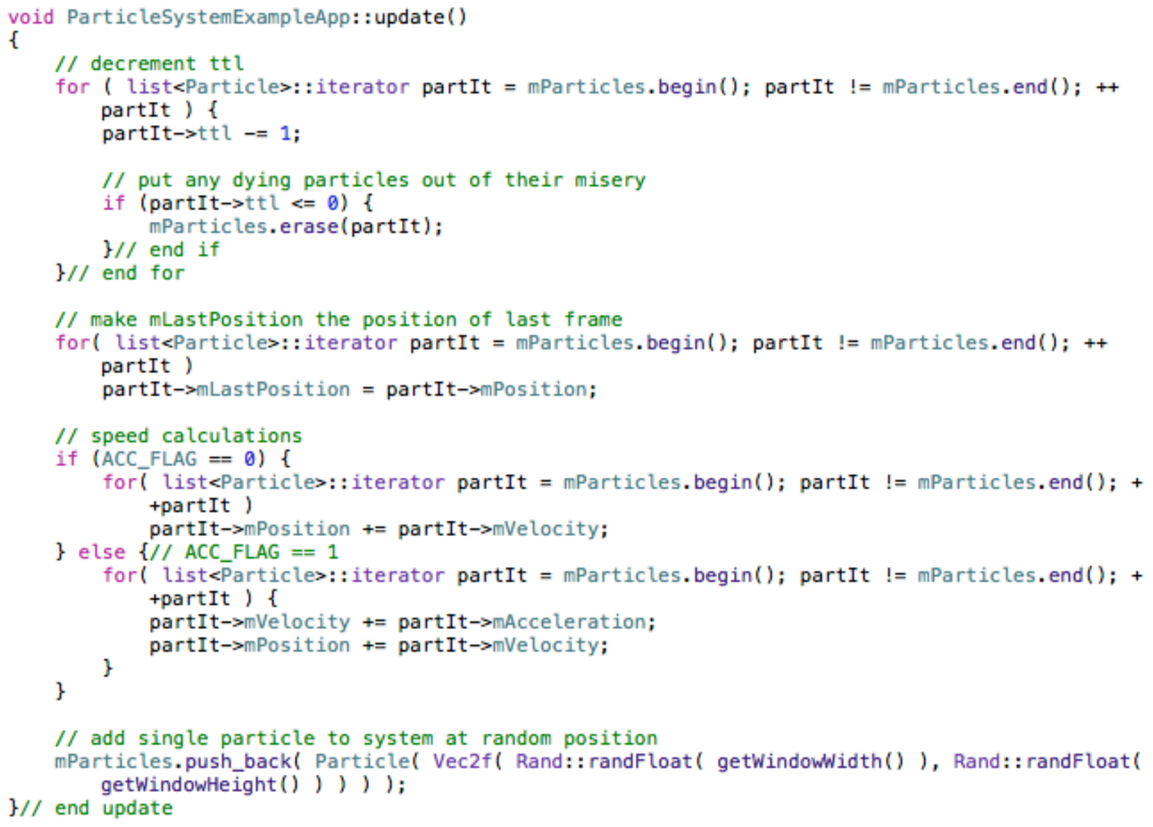


```
class ParticleSystemExampleApp : public AppNative {
  public:
    void mouseDown( MouseEvent event );
    void keyDown ( KeyEvent event );
    void setup();
    void update();
    void draw();

    // particle system data members
    list<Particle>  mParticles;
    static const int    NUM_INITIAL_PARTICLES = 10;
    int ACC_FLAG = 0;// implement acceleration change on particle velocity
    int DRAW_FLAG = 0;// 0 - point, 1 - line, 2 - sphere
    Vec2f EMITTER_POS = Vec2f( 300.0, 50.0 );
};
```

Figure VIII – Implementation of Particle System

The program starts with the setup method being called, which pushes back a number of particles to the particle system equal to NUM_INITIAL_PARTICLES (which in this case equals 10) at a random position on the screen. From there, the program starts running, and the update and draw methods are called every frame refresh.

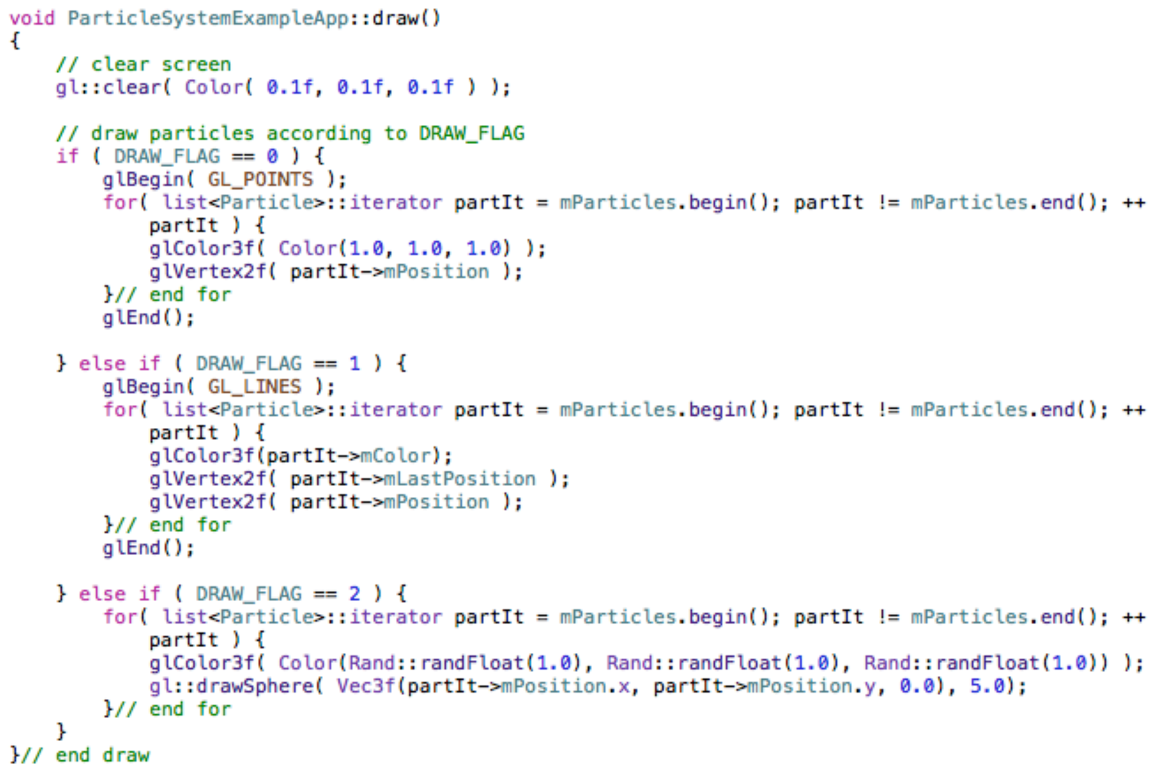
The update method serves 4 main purposes in relation to updating particles (see figure IX). First, decrement the particles' Time to Live (ttl), or how long before the particle is removed from the system, and removes any particles from the system who's ttl has reached 0. Next, all the remaining Particle's last position (mLastPosition) are updated to its current position (mPosition). Then, each particle's velocity (mVelocity) is incremented by the particle's acceleration (mAcceleration) if the ACC_FLAG is 1, or mVelocity is unaffected if ACC_FLAG is 0, and is then added to mPosition. Lastly, a new particle is pushed back to the system at a random position in the window.



```
void ParticleSystemExampleApp::update()
{
    // decrement ttl
    for ( list<Particle>::iterator partIt = mParticles.begin(); partIt != mParticles.end(); ++
        partIt ) {
        partIt->ttl -= 1;

        // put any dying particles out of their misery
        if (partIt->ttl <= 0) {
            mParticles.erase(partIt);
        }// end if
    }// end for

    // make mLastPosition the position of last frame
    for( list<Particle>::iterator partIt = mParticles.begin(); partIt != mParticles.end(); ++
        partIt )
        partIt->mLastPosition = partIt->mPosition;

    // speed calculations
    if (ACC_FLAG == 0) {
        for( list<Particle>::iterator partIt = mParticles.begin(); partIt != mParticles.end(); +
            +partIt )
            partIt->mPosition += partIt->mVelocity;
    } else {// ACC_FLAG == 1
        for( list<Particle>::iterator partIt = mParticles.begin(); partIt != mParticles.end(); +
            +partIt ) {
            partIt->mVelocity += partIt->mAcceleration;
            partIt->mPosition += partIt->mVelocity;
        }
    }

    // add single particle to system at random position
    mParticles.push_back( Particle( Vec2f( Rand::randFloat( getWindowWidth() ), Rand::randFloat(
        getWindowHeight() ) ) ) );
}// end update
```

Figure IX – Particle System Update Method

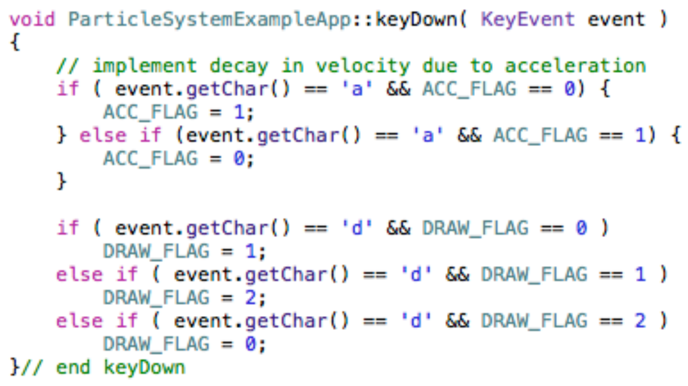
After the Update method finishes executing, the draw method is called. The draw method iterates through the doubly linked list, and draws the particles a certain way according to the DRAW_FLAG: 0 for points, 1 for lines, 2 for spheres, and starts at 0. This is to demonstrate that particles can be any sort of geometry, from the simple to the increasingly complex (see figure X).



```
void ParticleSystemExampleApp::draw()
{
    // clear screen
    gl::clear( Color( 0.1f, 0.1f, 0.1f ) );

    // draw particles according to DRAW_FLAG
    if ( DRAW_FLAG == 0 ) {
        glBegin( GL_POINTS );
        for( list<Particle>::iterator partIt = mParticles.begin(); partIt != mParticles.end(); ++
            partIt ) {
            glColor3f( Color(1.0, 1.0, 1.0) );
            glVertex2f( partIt->mPosition );
        }// end for
        glEnd();

    } else if ( DRAW_FLAG == 1 ) {
        glBegin( GL_LINES );
        for( list<Particle>::iterator partIt = mParticles.begin(); partIt != mParticles.end(); ++
            partIt ) {
            glColor3f(partIt->mColor);
            glVertex2f( partIt->mLastPosition );
            glVertex2f( partIt->mPosition );
        }// end for
        glEnd();

    } else if ( DRAW_FLAG == 2 ) {
        for( list<Particle>::iterator partIt = mParticles.begin(); partIt != mParticles.end(); ++
            partIt ) {
            glColor3f( Color(Rand::randFloat(1.0), Rand::randFloat(1.0), Rand::randFloat(1.0)) );
            gl::drawSphere( Vec3f(partIt->mPosition.x, partIt->mPosition.y, 0.0), 5.0);
        }// end for
    }
}// end draw
```

Figure X – Particle System Draw Method

The particle system also has a helper method for handling key events called keyDown, that cycles through values for ACC_FLAG and DRAW_FLAG. Pressing the 'a' key will cycle through the values 0 and 1 for ACC_FLAG, to implement decaying velocity on the particles, and 0, 1 and 2 for DRAW_FLAG, to draw the particles as dots, lines or spheres (see figure XI).



```
void ParticleSystemExampleApp::keyDown( KeyEvent event )
{
    // implement decay in velocity due to acceleration
    if ( event.getChar() == 'a' && ACC_FLAG == 0) {
        ACC_FLAG = 1;
    } else if (event.getChar() == 'a' && ACC_FLAG == 1) {
        ACC_FLAG = 0;
    }

    if ( event.getChar() == 'd' && DRAW_FLAG == 0 )
        DRAW_FLAG = 1;
    else if ( event.getChar() == 'd' && DRAW_FLAG == 1 )
        DRAW_FLAG = 2;
    else if ( event.getChar() == 'd' && DRAW_FLAG == 2 )
        DRAW_FLAG = 0;
}// end keyDown
```

Figure XI – Particle System keyDown Method

This small-scale implementation of a Particle System highlights the versatility and customizability of particle systems, showing velocity decay due to acceleration, the instantiation and destruction of particles and the variety of ways particles can be drawn. Figure XII shows a few screen captures of a demo, highlighting the random placement of particles, the different ways they are drawn, and the ensuing recognizable visual patterns they make.

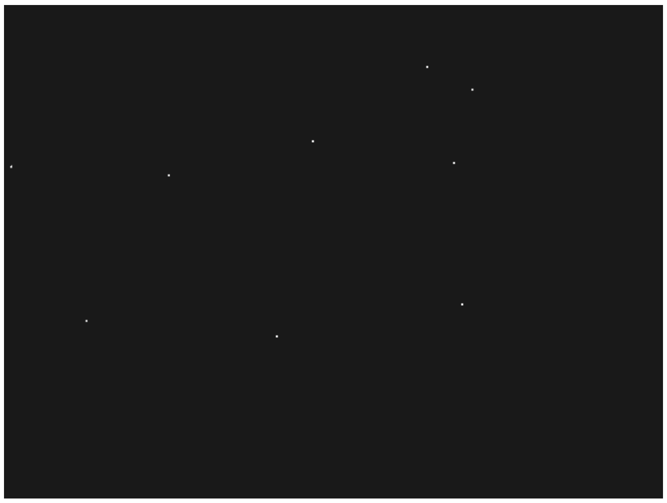


Figure XIIa – Particles as dots simulating a snowing effect

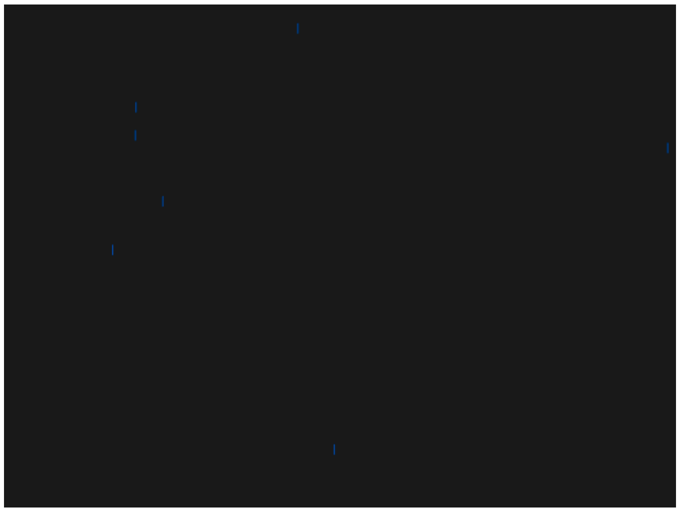


Figure XIIb – Particles as lines simulating a raining effect

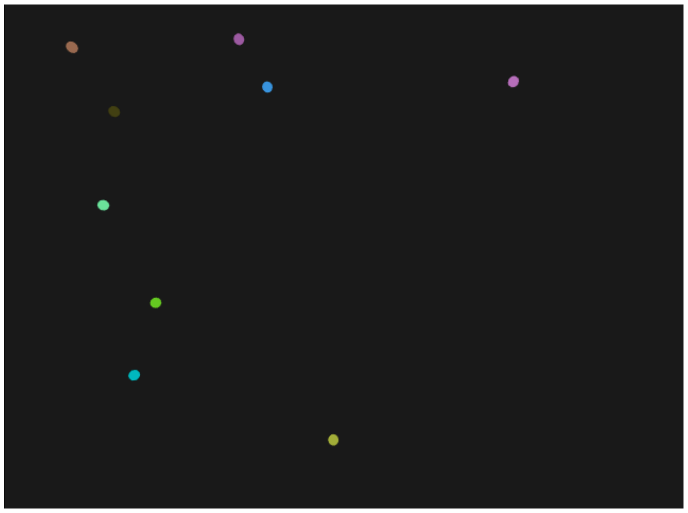


Figure XIIc – Particles as spheres

VII. Conclusion

Throughout this paper, the idea of particle systems has been thoroughly explored, from its beginning as a movie effect, to the core concepts behind particle systems, a solution to the growing performance issues with using particle systems for video games and other visual media, and a simple implementation to reinforce the core concepts explained. Particle systems an important and versatile aspect of computer graphics that enables a whole new world of study and implementation that will continue to grow as the field of computer graphics grows.

VIII. Bibliography

Reeves, William. "Particle Systems - Technique for Modeling a Class of Fuzzy Objects."*ACM Transactions on Graphics* 2.2 (1983): 359-76. Print.

Li, Xiangfei, Xuzhi Wang, Wanggen Wan, Xiaoqiang Zhu, and Xiaoqiang Yu. "Parallel Simulation of Large-Scale Universal Particle Systems Using CUDA." *2013 IEEE 11th International Conference on Dependable, Autonomic and Secure Computing* (2013): 572-77. Print.

Video Game Technologies 6931: MSc in Computer Science and Engineering. 1 - 20. Print.