Programming for Graphics

Particle System implementation report

Hannes Ricklefs
d1132332@bmth.ac.uk
University of Bournemouth

ABSTRACT

This report is structured into three different sections. The first section gives a brief introduction into particle systems. The second provides a description of each of the components that combine the particle system. The third gives a brief summary of the report and the main focus during implementation of the particle system.

Keywords

Particle System, Particles, Emitter, Solver, Data Structures

1. INTRODUCTION

Since their introduction in 1983 particle systems have become one of the fundamental parts of computer animation [1]. The pioneer in this area is William T. Reeves, in his initial paper he describes the use of particle systems to model “fuzzy” objects such as fire and clouds. Since then particle systems have been used in scientific simulations, feature films [1,8] and games [10] to model almost any kind of natural phenomena including smoke [3,6,8], fire [1,3,4,8], water [4,6,9], explosions [1,5,7], plant growth [2] and galaxies [3].

2. PARTICLE SYSTEM

2.1 Overview

As described by Reeves [1] a particle system is defined by a collection of particles that evolve over time. The evolution is determined through applying certain rules to the particles: creation of new particles, changing their attributes during their lifetime and finally the destruction (“death”) of old particles. The main essence of a particle system is that all of the afore mentioned rules are executed automatically. In general, particle systems are constructed using the following components: particles, emitters and solvers [1,2,3,4,5,10,11,12,13,14].

The implementation evolves its particles according to these five steps [1,10,11].

1) New particles are generated and injected into the current system.
2) Each new particle is assigned its individual attributes.
3) Any particles that have exceeded their lifetime are extinguished.
4) The current particles are moved according to their scripts.
5) The current particles are rendered.

The main aim of the initial design was to have a particle system that is highly Object Orientated for all the components in the system. The implemented components are **Particle System Manager**, **Particle System, Emitter, Particle, Solver, Force, CollisionObject**, and **Data Structure**.

2.2 Particle System Manager

In his article J. Burg mentions a Particle System Manager, which controls all the various particle systems [5]. As the

initial design supports multiple particle systems a manager class becomes important. The manager class is in charge of creating, removing, updating, and rendering all of its particle systems. In addition functions for interaction have been added in order to reset, pause and restart the particle system manager. The initial design proposed that the particle system manager renders according to a GraphicsLib::ThreadTimer this has been moved into the main class and the rendering and evolving is done through specifying the frame rate per second.

2.3 Particle System

The particle system class is responsible for holding all the initialization values for its solver, particles, and emitter. One aspect of the inital design was to be able to combine particle systems in such a way that a particle system itself can include other particle systems in order to create images as in Figure.1 [1]. Because of time limitations this feature has not been tested jet.

Figure.1 from [1]

Furthermore, the particle system is able to change over time, which will allow effects like a glowing cigarette particle that turn into smoke. Therefore the particle system class needs to hold all the relevant attributes, solvers and emitters to make the changes to its particles. The effects that have been implemented are changing of size and change of colour over time as well as updating the Particles position. Evermore, the particle system counts the number of alive particles to determine whether a system is ready to be deleted or not.

The initial class diagram in Appendix A shows two subclasses of the particle system class: Behavior Particle System and Effect Particle System. The Behavior Particle System enables interaction between particles in order to model flocking behavior [15,16]. However as seen in the final class diagram in Appendix B these because of time restrictions these have not been implemented.

The demon program shows to possibilities of using the Particle System to create fire and smoke which could be made into a subclass of Particle System Class to provide a Fire or Smoke Effects Particle System. This would allow other developers to use the particle system as an application programmer interface (API) [13]. In addition further method for pausing and restarting the Particle System have been added. To be able to draw Particles with Textures in a efficient way the Particle System has a Texture attribute which is checked for in the draw method in order to do the Texture binding once for all Particles.

2.4 Emitter

For a particle system to evolve and change its volume over time new particles need to be added to the system. This is the main work of the emitter, who generates new particles by means of controlled stochastic processes [1,11]. For each frame new particles are randomly emitted into the system according to some user specified distribution [11]. The actual number of new Particles to be emitted is specified in the Particle System Class it emits new Particles according to the following function

(Eq. 1) NParts = RandomPosNum(varNumParticles)

Where RandomPosNum(_) returns a random number between 0.0 and +varNumParticles,

In the proposed design all particle behaviour is assigned from the emitter: **Emitting direction, Initial velocity, Initial Position, Color, Flare Colour, Size, Flare Size and Texture Map.** All the initial values will be set at the beginning of a particle's life.

The class diagram in Appendix A and Appendix B shows that the Emitter class is an abstract class. In order to have the most flexibility when creating new particles the following Emitter types are implemented:

1) PointEmitter: Particles are emitted from point in 3D space. The Emitter has a sweep angel for allowing the Particles to be emitted in a spherical motion
2) PlanarEmitter: Particles are emitted from a plane. The Emitter takes two dimension vectors and randomly selects points between these two vectors as the initial position.

In addition to the original design the author added a method called rebirth, which takes a pointer to a Particle and reinitialises its values. This is needed for the dead pool as described in section 2.7.

2.5 Particles

The particles determine the motion, volume, color and general appearance of the particle system. Typical Attributes according to [1,5,11] are: **position, velocity, color, transparency,** and **lifetime**. In [5] J. Burg adds two attributes **oldPosition, texture** and **dead**. OldPosition enables to stretch a particle between its old and new position, which can be useful when creating sparks. Texture enables particles to have a texture assigned to them. The attribute dead is particular effect introduced by

J. Burgs article. He states that instead of deleting and creating new particles, a dead particle should be flagged and then reinitialized when new particles are created. This reduces the amount of memory access and is hence faster.

In [4] Sims adds jet another attribute called **bounce,** which specifies what happens to a particle when colliding with other particles or objects in the scene.

The class diagram in Appendix A and Appendix B shows that the particle class has subclasses. The implemented subclasses are:

1) PointParticle: A Particle is a sphere in space, which has a radius.

   ImgParticle: A ImgParticle contains a Texture which is projected on to a quad which can be changed in size to increase the size. Additionally special test are made to enable Billboarding.

2) StrikeParticle: A StrikeParticle stores the current and old position. When its draw function is called it draws a line between these in order to enable effects like sparks and strikes.

2.6 Solver and Forces

In order to create any of the effects mentioned in section 1 the particles need to be animated. Although many effects need specialized algorithms, this particle system is designed to incorporate new algorithms very easily. By specifying a Solver interface newly created algorithms can be "plugged" into the particle system. The literature presents all kinds of different solvers from basic linear solvers to advanced solvers using Navier-Stokes equations [6,8,9]. In [3,4,14] the authors are introducing Forces into the movement of the particles. To give more flexibility this design separates the forces into classes in order for the Solver to choose one or many correct forces. In [14] A. Witkin groups the forces into three broad categories, which form the three subclasses of the Force class:

1. Unary forces, such as gravity and drag, that act independently on each particle, either exerting a constant force, or one that depends on one or more particle positions, particle velocity, and time.
2. n-ary forces, such as springs, that apply forces to a fixed set of particles.
3. Forces of spatial interaction, such as attraction and repulsion, which may act on any or all pairs of particles, depending on their positions.

All the Forces need to have a Force vector that affects the Particles motion according to a specified attenuation. If the attenuation is set to 1 the Force is completely applies. The implemented Forces are:

1. RandomForce: Applies the specified Force vector randomly to the given velocity of the Particle.
2. GravityForce: Applies the specified Force vector including the magnitude to the Particles velocity vector.
3. UniformForce: Applies the specified Force vector directly to the Particles velocity vector

The class diagram in Appendix A. shows the Solvers the author is proposing to implement. Appendix B shows the Solvers the Author implemented:

1. StraightSolver: Solves particle movement according to a linear equation. Particles fly through space at constant time.
2. Gravity Solver: Solves particle movement according using gravity. Particles fly through space and the direction and speed varies according to the laws of gravity.
3. Navier-Stokes Solver: This is the most advanced Solver. In order for the Solver to work properly a 3D data structure is needed [6,18,19].

Appendix B shows the Solvers the Author implemented:

1. StraightSolver: When no Force is added to the Solver the StraightSolver just adds the current velocity to its self. If however Forces are added to the StraightSolver it adds these in turn to affect the Particles motion.

2.7 Data Structure

Through the ever-increasing speed of processing power in computer hardware, particle systems are reaching a stage at which they can be used to create effects close to reality in real-time [3]. In the special effects industry this is still not achieved [8]. In order to create satisfying results in real-time, the number of particles used within a particle system needs to be within thousands [5]. Therefore an efficient data structure is needed to store the particles.

Whenever complex or efficient storage requirements are needed, the literature refers to Octrees [6,12,18,19]. Octrees are a hierarchical construct for spatially managing large amounts of three-dimensional data [12,17]. Their construction is well documented and researched [12,17,18]. The usual approach is to start with a three-dimensional cube and continually subdivide it into eight cubes until either a specified cube size is reached or there are no more objects that could be subdivided. See figure 2.

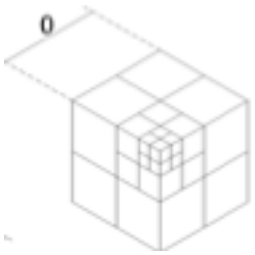


Figure 2 from [19]

When implementing an Octree data structure for particle systems, additional aspects need to taken into consideration. The main aspects are finding neighbors on the faces, edges and vertices of the corners, the constant subdividing of the sub-cubes, and changing particles from one cube to another.

The main benefit of Octrees is the ability to store extra information within nodes that can be applied when solving the animation of the particles using the Navier-Stokes solver.

The class diagram in Appendix A. shows additional data structures the author proposes:

1. Grid: A three-dimensional grid structure with a gird size of 30 * 30 * 30.
2. List: A data structure that uses a list container class from the standard C++ library. Here some special sorting algorithm (bubble or quick sort) is going to store the particles in an order relating to their position in the particle space. Another approach would be to cluster particles according to their location, as stated in section 2.5.
3. Octree: The aforementioned data structure.

However the author only had time to implement one DataStructure called VectorDS. This DataStructure uses the standard vector container class to store the Particles.

It is the data structures responsibility to do collision detection between the particles and objects from within the scene. Additionally, the data structure is responsible for creating a pool of deleted “dead” particles that can be reused when new particles are created.

2.8 CollisionObjects

During the Implementation the author realized the need for detecting collision between the Particles and Object within the scene. To generate this in a Object Orientated design the author decided to generate Collision Objects which have to have a collision method. This method takes a Particle Pointer and does internal tests whether collision has occurred. If it did it changes the Particles velocity accordingly. Furthermore the CollisionObjects include methods for drawing themselves and methods for translation, scaling and rotation.

The implemented CollisionObjects are:

1. CollisionSphere: Does simple SphereSphere collision tests. When collision occurs the velocity of the Particle is simply reversed.
2. CollisionPlane: Does simple SpherePlane collision detection. This Class is not completed jet.

3. SUMMARY

This report outlined the components the author included in his original design and show the implement components. His main focus is on the extendibility of any of the components through Object Orientated programming. Furthermore it was the authors aim to create the main skeleton for all the Superclasses to lay down the fundamental building blocks for the Particle System. It has to be mentioned that the author mainly wanted to have all the methods defined that are needed for the Particle System it is therefore possible to find methods that are still not implemented. They are there to show what the author has planed for future development.

The outline of the algorithm for evolving a particle system is as follows:

Create particles

Initialize particles

For each frame:

For each simulation time increment:

Select particles

Perform operations

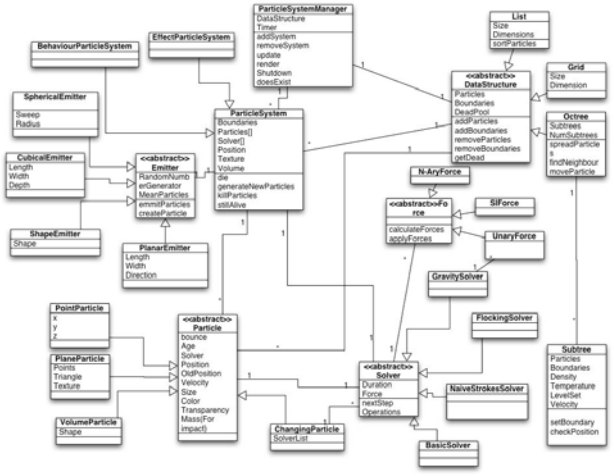
Update position and other attributes

Evolve age remove dead particles

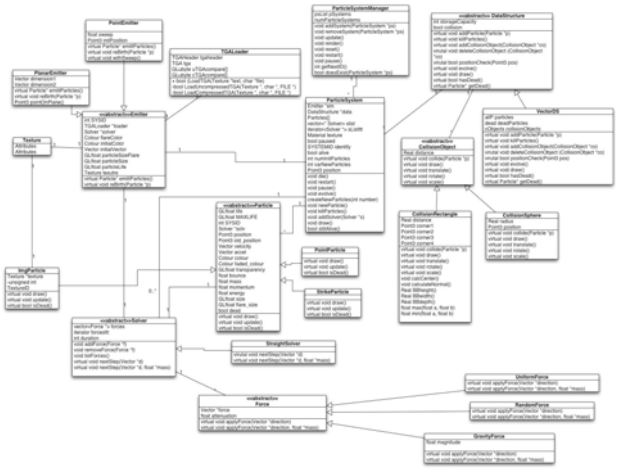
Render

The resulting demonstrations of the particle system are include: fire, smoke, and collision detection.

4. APPENDIX A



5. APPENDIX B



6. REFERENCES

[1] REEVES, W. T., 1983. Particle Systems – A Technique for Modeling a Class of Fuzzy Objects, *ACM Transaction on Graphics*, 2 (2), 359-367.

[2] REEVES, W. T., AND BLAU R., 1985. Approximate and Probabilistic Algorithm for Shading and Rendering Structured Particle Systems, *Computer Graphics*, 19(3), 313-322.

[3] ILMONEN, T., AND KONTKANEN J., 2003. The Second Order Particle System, *Journal of WSCG*, 11(1)

[4] SIMS, K., 1990. Particle Animation and Rendering Using Data Parallel Computation, *Computer Graphics*, 24(4), 405-413.

[5] BURG, J., 2000. Building an Advanced Particle System, *GAME DEVELOPER*, 44-50.

[6] FEDKIW, R., GIBOU, F., AND LOSASSO, F., 2004. Simulating Water and Smoke with an Octree Data Structure, *SIGGRAPH 2004 ACM TOG*, 23, 457-462.

[7] FELDMAN, E. B., O`BRIAN F. J., AND ARIKAN, O., 2003. Animating Suspended Particle Explosions, *ACM SIGGRAPH 2003, San Diego, CA, July 27-31*, 1-8.

[8] FEDKIW, R., ENRIGHT, D., AND NGUYEN, D., 2003. Simulation and Animation of Fire and Other Natural Phenomena in the Visual Effects Industry, *Western States Section, Combustion Institute, Fall Meeting, UCLA, 2003.*

[9] MÜLLER, M., CHARYPAR, D., AND GROSS, M., 2003. Particle-Based Fluid Simulation for Interactive Applications, *Eurographics/SIGGRAPH Symposium on Computer Animation (2003).*

[10] WATT, A. AND POLICARPO F., 2001. *3D GAMES Real-time Rendering and Software Technology*, New York: ADDISON-WESLEY

[11] PARENT, R., 2002. *Computer Animation Algorithms and Techniques,* London:Morgen-Kaufmann.

[12] FOLEY D. J., VAN DAM, A., FEINER, K. S., AND HUGES, F. F., 1997. *Computer Graphics PRINCIPLES AND PRACTICE*, 2nd ed. New York: Addison-Wesley.

[13] MCALLISTER, K. D., 2000. The Design of an API for Particle Systems, *Technical report, University of North Carolina*, 2000.

[14] WITKIN, A., 1999. Physically Based Modeling Principles and Practice – Particle Systems, *SIGGRAPH-99 Course Notes.*

[15] REYNOLDS, W. C., 1987. Flocks, Herds, and Schools: A Distributed Behavioral Model, *Computer Graphics*, 24(4), 25-34.

[16] LOREK, H. AND WHITE, M., 1993. Parallel Bird Flocking Simulation,

[17] DELOURA, M., 2001. *GAME PROGRAMMING Gems 2*, Hingham: CHARLES RIVER MEDIA, INC.

[18] VEMURI, C. B., CAO, Y., AND CHEN, L., 1998. Fast Collision Detection Algorithms with Applications to Particle Flow, *Computer Graphics forum*, 17(2), 121-134.

[19] TISNOVSKY, P., AND HEROUT, A., 2002. Adaptive Algorithm for Vector Field Interpolation Based on Octree Structure, *International Conference on Computer Graphics and Interactive Techniques Proceedings of the 18th spring conference on Computer graphics*, 151-156