# Document and Critique of the Algorithms Used

Contents

[Document and Critique of the Algorithms Used 1](#_Toc251333453)

[Contents 1](#_Toc251333454)

[1 - Particle System 2](#_Toc251333455)

[1.1 – How it Works 2](#_Toc251333456)

[1.2 – Critique 2](#_Toc251333457)

[2 – Phong Shader 3](#_Toc251333458)

[2.1 Phong Shader Problems 3](#_Toc251333459)

[3 - Shadows 3](#_Toc251333460)

[3.1 – How it Works 3](#_Toc251333461)

[3.2 – Critique 3](#_Toc251333462)

## 1 - Particle System

### – How it Works

The Particle System is split into two distinct classes, the ParticleEmitter class, and the Particle class.  
When an instance of a ParticleEmitter is created it is passed in parameters to define the particles it will create and their subsequent effects, such as lifetime, shape, and direction for example.  
When the particles are created via the Particle.Create method they have their IsAlive flag set to true.  
Then in the ParticleEmitter.Update method the particles are split into two separate lists, one list holds the alive particles (which at the first iteration will be full), the second list holds all the dead particles (which at the first iteration will be empty).  
This update method iterates through the aliveParticles list and checks to see which particles have exceeded their set lifetime, if the particle has exceeded its set lifetime it is moved to the deadParticles list and removed from the aliveParticles list.  
Whilst this is happening there are also two integers that contain a value that points to the end of the aliveParticles list, and the deadParticles list these are updated accordingly.  
Next the Update method compares the ParticleRateOfGeneration value against the PointerToEndOfDeadParticles, if the ParticleRateOfGeneration is bigger in value the ParticleRateOfGeneration is set to PointerToEndOfDeadParticles (because there isn’t enough dead particles to meet the desired ParticleRateOfGeneration).  
Else, there are enough dead particles to meet the desired ParticleRateOfGeneration.  
Once the current ParticleRateOfGeneration has been determined for that current frame then that number of particles is removed from the deadParticles list, reinitialising the particle and placing them at the end of the aliveParticles list.  
The final step in the ParticleEmitter.Update function is to iterate over the list of aliveParticles again, calling the Particle.Update method on them all.  
The Draw function for the ParticleEmitter is simple, for each particle in the aliveParticles class, the Particle.Draw method is called on it.

### 1.2 – Critique

One of the key issues with Particle Systems is efficiency; this is because potentially thousands of particles can be drawn/updated per frame. This is why efficiency was always kept in mind when designing and implementing the algorithms used for the Particle System.  
This is why OpenGL’s immediate mode was avoided (glBegin/glEnd) and vertex arrays (glDrawElements) was utilised for the drawing of the particles. Another good efficiency idea was implemented for the maintaining of the particles algorithm. This was achieved by setting the memory of the vectors that hold the particles (alive and dead) using the STL resize command. Doing this meant that I wouldn’t be penalising myself efficiency wise because I could avoid dynamically resizing the particle vectors when I moved one element from one vector to another. Another positive thing with this design of the algorithm is that by maintaining two lists, one for alive particles and one for dead ones, I could easily avoid checking all particles when testing to see if the particle was still alive, because I could check the alive particles list and avoid checking against all the dead particles. One more positive point about the algorithm implemented is that it uses two maintained integers to keep track of where the last active/dead particles are in their respective list. This is so no pointless iterations are done past the last active/dead particle vectors.

## 2 – Phong Shader

### 2.1 Phong Shader Problems

Please see the implementation issues document in the portfolio folder for more information.

## 3 - Shadows

### 3.1 – How it Works

Wall shadows are implemented, not shadows cast onto other objects aren’t.  
It works by creating the number of shadow matrices for each light, e.g. 8 lights equal 8 shadow matrices. Then the position of 6 planes of the cubes are taken (this is what the shadows will be projected onto) and are stored in a vector. Then a vector of Vector4f is created containing 8 (one for each light source) Vector4f which represents each position of each of the light sources. This information is passed to the BuildShadowMatrix function which builds the shadow matrices by calculating the dot product between the plane and the light’s position. Finally in the draw method of the main game loop a loop executes (numberOfLights\*numberOfPlanes) and another loop is nested in this drawing all of the shapes, then if the stencil test fails a shadow is drawn (projected onto the correct plane).

### 3.2 – Critique

A good thing about the algorithm is that it works for correctly producing wall shadows for each object.  
The major criticism of the algorithm implemented is that it is incredibly inefficient; this is because for each number of light sources multiplied by the number of planes every shape must be drawn. This is so the correct shadow could be projected onto more than one plane due to multiple light sources.