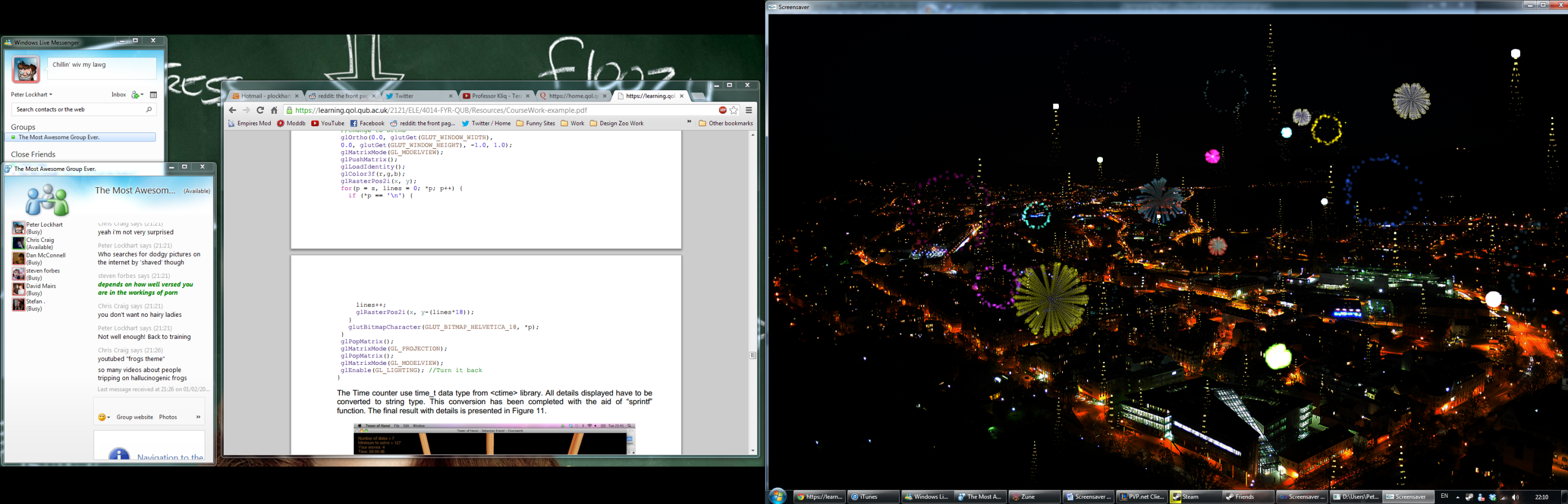


Peter Lockhart

ELE 4014

Screensaver

Stuart Ferguson



Contents

[The Project 3](#_Toc348044662)

[Project Flow 4](#_Toc348044663)

[Init 4](#_Toc348044664)

[Update and Spawning fireworks 4](#_Toc348044665)

[Draw 6](#_Toc348044666)

[The Particle System 6](#_Toc348044667)

[Overview 6](#_Toc348044668)

[Class Variables 7](#_Toc348044669)

[ParticleSystem class 7](#_Toc348044670)

[Particle class 8](#_Toc348044671)

[Class Methods 8](#_Toc348044672)

[ParticleSystem class 8](#_Toc348044673)

[Particle class 12](#_Toc348044674)

[Examples of extensibility 12](#_Toc348044675)

[Fireworks 13](#_Toc348044676)

[Firework stages 13](#_Toc348044677)

[Firework variables 13](#_Toc348044678)

[Firework methods 14](#_Toc348044679)

[Firework modification 16](#_Toc348044680)

[Dazzle Fireworks 17](#_Toc348044681)

[Dazzle Firework variables 17](#_Toc348044682)

[Dazzle Firework methods 17](#_Toc348044683)

[Modelling other effects 19](#_Toc348044684)

[Fire torch 19](#_Toc348044685)

[Bubble blower 19](#_Toc348044686)

[Waterfall 20](#_Toc348044687)

[Testing 20](#_Toc348044688)

[Enhancement 21](#_Toc348044689)

[Evaluation 21](#_Toc348044690)

Not all of the code from the project is pasted in this documentation as some things are simple and not worth explaining. This documentation is to provide an outline of the system and go over in more depth some of the finer parts of the system. This documentation is best used by looking at the appropiate section of source code from the project which is being talked about in the documentation. The code in the project is thoroughly commented and should be looked at to fully analyise the system's quality and operation.

# The Project

For this project I decided to create a particle system so I could model fireworks. Periodically fireworks will launch out from the landscape and explode in a random colour. The particle system I designed is extensible so it can be used to model other systems.

There are 3 main classes involved in modelling the fireworks, each with a series of inheritance to make sure the system is extensible.

ParticleSystem

Firework

DazzleFirework

On overview of the different classes is as follows;

* *ParticleSystem* - This class models a particle system using different properties set by the user
* *Firework* - This class is a child of *ParticleSystem*. It modifies its particle system at different intervals with properties to model a firework going up in the air and then exploding
* *DazzleFirework* - This class is a child of *Firework*. It modifies the normal behaviour of a firework so that there is a dazzle affect around the firework as it explodes.

Provided with the core particle and firework classes there are the following helper classes:

* *CoordSystems* contains Vector3 which represents a point in 3D space. It also contains methods to help create a random vector within a range.
* *ColourSystems* contains RGBA which represents a Red-Green-Blue colour with an Alpha component. It also contains methods to help create random colours and to interpolate between 2 colours.

To load graphics there are the classes *Texture* and *TGALoader*.

* *Texture* parses the filename of a texture and finds out what file type it is. Using that it would call the appropriate loader and parse the information, then create an OpenGL texture from it. Currently only .tga is supported by the system is easily extensible to allow multiple formats.
* *TGALoader* is a class taken from the internet which parses a .tga file and its alpha channel, producing a byte array of information.

For debugging purposes I created a camera class *Camera*. Using the keys WASD you can move the position of the camera and using the ARROW keys you can change its rotation.

# Project Flow

In this section I will detail how a typical screensaver will run.

The core stages of the project are:

* **Init** graphic settings and load textures
* **Update** and **Spawn** fireworks depending on properties
* **Draw** background and active fireworks

## Init

In the project at the top of *ScreenSaver*.cpp are the variables that dictate how often fireworks spawn and the max amount you can have on the screen.

The screensaver window and textures are loaded in the 'int main()' method.

I set the window to be the size of the screen. When initialising the glut display mode I enable double buffering to improve performance of the application as well as setting the colour mode to Red-Green-Blue-Alpha.

glutInitWindowSize(GetSystemMetrics(SM\_CXSCREEN), GetSystemMetrics(SM\_CYSCREEN));

glutInitDisplayMode(GLUT\_RGBA | GLUT\_DOUBLE);

I use 2 textures to display the screensaver. I have 1 small ball that is used to represent a particle, which is reused in all of the created fireworks. The 2nd texture is a picture of a city landscape which I display in the background.

Finally, I register the main game loop. The update function will call the draw function after updating all the game objects.

## Update and Spawning fireworks

At the start of the update the time is calculated between update calls. This will allow for more smooth movements of fireworks if the computer lags at certain bits due to the operating system.

//work out the length of time since the program started

currentTime = glutGet(GLUT\_ELAPSED\_TIME);

//subtract the current time by the last recorded time to get the change

float dt = currentTime - previousTime;

//convert to seconds

dt /= 1000;

//store the current time as the previous time for the next update call

previousTime = currentTime;

*Updating*fireworks takes place in the method 'void Update(int i)'. Here all of the current fireworks are looped through.

for (int i = 0; i < (int)\_fireworks.size(); i++) {

Firework \* loopedFirework = \_fireworks[i];

Here its Update() method is called which will in turn update its particle system. If it is found that a firework has finished then it is removed from the list of active fireworks and deleted from memory.

During the *Updating* if it is found that there are less active fireworks than what there is allowed, then we roll for a chance that a new firework is to be spawned. The system generates a random number between 0 and 100 and compares it to the firework spawn chance, and if lower it then **Spawns** a new firework.

Spawning a firework takes place in the method 'void SpawnFirework()'.

Here a new firework object is created, where it has a chance to be a Dazzling firework. In the method I change some of the properties of the firework so it behaves the way I would like it to.

For example, I make it start in a random position so it looks like it is launching from within the cities landscape.

//place it at a random position around the point (-3, -3, -7)

Vector3 pos = Vector3(-3, -3, -7);

Vector3 randOffset;

randOffset.Random(5);

//giving the variation in position added weight on the x axis so the fireworks spread out more

randOffset.x \*= 1.5f;

pos = pos + randOffset;

newFirework->Origin = pos;

I also set the firework to explode with a random colour and make it so it launches with a trail.

//make it a random colour

RGBA randColour;

randColour.Random();

randColour.a = 1;

newFirework->ExplodeColour = randColour;

//other poperties I like

newFirework->ShouldEmitTrail = true;

I also have a 1 in 3 chance that thenewly created firework will use a different blending function, which helps bring variety to the scene

//make it a rare chance that some of the fireworks have different settings

if ((int) rand() % 3 == 1) {

newFirework->BlendFactor = GL\_ONE\_MINUS\_SRC\_COLOR;

newFirework->TrailLifeModifier = 0.6;

newFirework->ExplodeSpeed = 0.65f;

}

Once the firework has been created, it is added to the list of currently active fireworks.

After all the updating has been done, I schedule the next update to be performed at a target 60 frames per second. After scheduling the next update, the scene is drawn.

glutTimerFunc(0.16, Update, 1);

glutPostRedisplay();

## Draw

The entire project's drawing calls are done in the 'void RenderScene()' method.

Most of the drawing functionality of the project is separated off into particle system's draw method. The only noteworthy thing here is the switch between Orthographic and Perspective views.

I initially set the view to Orthographic so I can draw a background landscape to the scene in 'void DrawBackground()', then I switch to Perspective to draw the fireworks.

I loop through all the active fireworks and call their draw methods.

//draw the fireworks

for (int i = 0; i < (int)\_fireworks.size(); i++) {

Firework \* loopedFirework = \_fireworks[i];

loopedFirework->Draw();

}

# The Particle System

## Overview

The class *ParticleSystem* manages, creates and draws particles for the project. It has a number of different properties and functionalities available so it can be used to model a wide range of environments.

The system relies on 3 methods to run in a project:

* The **constructor methods** which loads the default values for a system and the texture
* The **Update** method which moves the particles and spawns new ones depending on the situation
* The **Draw** method which uses blend functions and the positions of particles to draw them on the screen

Individual *Particles* are managed by the *ParticleSystem* it was spawned from. Its properties are set by the *ParticleSystem* when it spawns a particle, where they are affected by the properties defined by the user.

## Class Variables

### ParticleSystem class

There are a series of properties that are accessible outside of the class that help define how that particle system should behave. After changing some of these properties the next time particles are spawned they will take into account these new properties. Changing these properties will not change currently existing particles.

Vector3 Origin; //Position all particles will start off in the scene

Vector3 OriginVariance; //The variance in the start position of the particles. A variance of eg (0, 1, 0) means the particle can randomly spawn anywhere between 1 above and 1 below the Origin.

float ParticleLife; //The amount of time a particle lasts for in seconds

float ParticleLifeVariance; //The variance in the amount of time a particle is alive for. This ranges the life +/- by the variance

float ParticleSize; //The size a particle should be on the screen

float StartScale; //The scale the particle should start off at

float EndScale; // The scale the particle should end at

float StartScaleVariance; //The variance in the scale of the particle at the start

float EndScaleVariance; //The variance in the scale of the particle at the end

RGBA StartColour; //The colour particles start off with

RGBA StartColourVariance; //The variation in colour particles start off with

RGBA EndColourVariance; //The variation in colour particles end with

RGBA EndColour; //The colour particles end on. You can make it so the end colour of the particles is the same as the start colour by using RGBA(-1, -1, -1, x). Similarly, you can make the end alpha channel the same by using RGBA(x, y, z, -1). These can be combined together to make completely identical RGAB(-1, -1, -1, -1). This is useful for systems with widely varying variance variables.

float Angle; //The angle which the particle should move out from, in degrees. 0 is up and 90 is right etc.

float AngleVariance; //The variance in angle, in degrees

float MoveSpeed; //The speed at which a particle moves at

float MoveVariance; //The variance in a particle's speed

Vector3 Gravity; //The adjustment that is applied to the particle's move vector every step

float MaxParticles; //The max amount of particles that can be active at once

float EmissionRate; //The time that needs to expire before a particle can be spawned. 0 means particles spawn instantly

bool ShouldRegenParticles; //Flag for whether the system should replace particles that expire

GLenum BlendFactor; //how RGBA blending factors are computed when drawing

There are a few private variables in the system which just store what texture to use when drawing the particles; a list of the currently active particles and a count of the length of time since a particle was spawned.

### Particle class

There are only a few properties in the *Particle* class that need to be set to fully model what the *ParticleSystem* is required of.

Vector3 Position; //The position the particle is at in the scene

float Age; //The length of time this particle has been active for

float MaxAge; //The max length of time this particle can be alive for

float Scale; //The current scale of the particle

float EndScale; //The target end scale of the particle

RGBA Colour; //The particles colour

RGBA EndColour; //The colour this particle will finish at

Vector3 MoveVector; //The movement a particle will make per frame

bool IsFake; //Flag for whether this particle is part of a trail. This flag is important as the parenting ParticleSystem will ignore it for certain things, discussed later

There are 2 private variables which store the starting scale and colour of the particle. This is used in the particle's update method where it uses the original and the ending variables for a specific property to interpolate between the 2 values, depending on how long it has been alive for.

## Class Methods

### ParticleSystem class

Most of the methods in this class are handled by the system itself through the Update() method.

#### void ParticleSystem::Update(float dt)

This method loops through all the active particles and calls their update method. If it comes across a particle that has passed its age then it is removed.

//loop through all the active particles

for (int i = 0; i < (int)\_activeParticles.size(); i++) {

Particle \* loopedParticle = \_activeParticles[i];

//if the particle is older than what it should be..

if (loopedParticle->Age > loopedParticle->MaxAge) {

//remove the particle

RemoveParticle(loopedParticle);

//reduce the iterator because we've just deleted a particle and don't want to skip over the next in the loop

i--;

continue;

}

//updating them

\_activeParticles[i]->Update(dt);

//affect the particle's velocity by the system's gravity

\_activeParticles[i]->MoveVector = \_activeParticles[i]->MoveVector + (Gravity \* dt);

}

It also replenishes particles if the system allows regeneration and the emission time has been passed.

//if the system is allowed to regenerate particles...

if (ShouldRegenParticles == true) {

//reduce the emission count by the dt so we know when we can next spawn a particle

if (\_emissionCountdown > 0)

\_emissionCountdown -= dt;

//else if the emission countdown has been passed then we can spawn more

else

ReplenishParticles(dt);

}

#### void ParticleSystem::ReplenishParticles(float dt)

This method replenishes particles, spawning new particles while the number of active particles is below the max amount of particles allowed. The method first works out how many particles it can spawn in the period since the last update calculated with the emision rate of the system.

For an over exaggerated example, consider a particle system with an emission rate of a particle every 0.1 seconds. If the project takes 1 second to update every frame, then the time since the last update is 1 second. This means that in the space of that 1 second since the last update, the system should spawn 1/0.1 = 10 particles. If an emission rate is 0 then particles can be spawned instantly.

The method continues spawning the calculated amount of particles until it spawns them all or hits the max amount of particles the system can have. After doing so, the emission timer is reset so particles cannot be spawned until the set amount of time has expired again

int numParticlesToSpawn;

//work out the number of particles that can be spawned in the length of time it took since the last update if the system is spawning faster than the systems FPS

if (EmissionRate < dt)

numParticlesToSpawn = dt / EmissionRate;

else

numParticlesToSpawn = 1;

//if the system can spawn more particles and the cooldown for spawning particles has been passed, then spawn a new particle

while ((numParticlesToSpawn > 0 || EmissionRate == 0) && \_totalActiveParticles < MaxParticles) {

//decrement the num of particles to spawn count

numParticlesToSpawn--;

SpawnParticle();

}

//reset the emission count

\_emissionCountdown = EmissionRate;

#### void ParticleSystem::SpawnParticle()

This method spawns a particle according to the system's properties. Using the properties discussed above a particle is created and its properties set according to the values calculated in the method. Most of the code is self explanatory and easy. A few sections to highlight are as follows.

newParticle->Scale = (StartScale - StartScaleVariance) + StartScaleVariance \* ((float)rand() / ((float)RAND\_MAX / 2));

This section and ones similar to it make it that the particle scale will be in **the range of StartScale +/- StartScaleVariance**. This is done by subtracting the StartScale by the variance so the values are initially

**- StartScaleVariance** but then increment it by **a random value of up to double of the** **StartScaleVariance** to make the final value a random number in the range **StartScale +/- StartScaleVariance**.

//working out position with variance

Vector3 randomOffset;

randomOffset.Random(2);

randomOffset.z = 0;

newParticle->Position = (Origin - OriginVariance) + (OriginVariance \* randomOffset);

This section of code varies the position of the particle system. It takes the Origin of the system and subtracts it by the variance in the position of origin of a particle.

For a system of Origin (0, 0, 0) this makes the starting values **-OriginVariance**. By incrementing this by **a random value up to double of OriginVariance** this brings the range of potential values  **-OriginVariance** to **+OriginVariance**

//work out the direction for the particle by adjusting the starting angle

float randomAngle = (Angle - AngleVariance) + AngleVariance \* ((float)rand() / ((float)RAND\_MAX / 2));

//convert to radians

float radians = (3.141592 \* (randomAngle + 90)) / 180;

//convert radian angle to direction vector

newParticle->MoveVector = Vector3(-cosf(radians), sinf(radians), 0);

float randMoveSpeed = (MoveSpeed - MoveVariance) + MoveVariance \* ((float)rand() / ((float)RAND\_MAX / 2));

newParticle->MoveVector = newParticle->MoveVector \* randMoveSpeed;

Like the above sections this creates a random angle using the system's starting angle and its angle variance. After calculating the angle in degrees it is converted into radians. From this angle in radians a movement vector is created.

This is done by taking the angle and resolving the vertical and horizontal components and storing them into a 3D vector with no Z component (ie a 2D vector). This vector is then multiplied by a calculated movement speed for the particle.

#### void ParticleSystem::RemoveParticle(Particle \* particleToRemove)

This method removes a given particle from the particle system and then deletes it from memory. It firstly finds the index of the particle in question then removes it from the list of active particles.

//get the index of the particle in the vector

int pos = std::find(\_activeParticles.begin(), \_activeParticles.end(), particleToRemove) - \_activeParticles.begin();

//remove it from the vector

\_activeParticles.erase(\_activeParticles.begin() + pos);

Next it decrements the count of the total active particles in the system. Trail particles are not treated as proper particles in the system and so do not count towards the total count of particles.

//don't count the particle as removed if it was a trail

if (particleToRemove->IsFake == false)

\_totalActiveParticles--;

#### void ParticleSystem::Draw()

This method draws all of the particles in the particle system according to each particle's properties. The following OpenGL settings are used along with the chosen blend function the user has chosen for the system.

glDisable(GL\_DEPTH\_TEST);

glEnable(GL\_BLEND);

glBlendFunc(GL\_SRC\_ALPHA, BlendFactor);

glHint(GL\_PERSPECTIVE\_CORRECTION\_HINT, GL\_NICEST);

glHint(GL\_POINT\_SMOOTH\_HINT, GL\_NICEST);

Once the routines have been set up then we bind that particle system's texture using glBindTexture() ready for drawing.

Looping through each of the particles in the active particle list, we store the view matrix and pop it each time. With each particle we translate it to its position in the world and apply scale and colour functions according to that looped particle's properties before drawing.

//translate to particle position

glTranslatef(loopedParticle.Position.x, loopedParticle.Position.y, loopedParticle.Position.z);

//scale according to particle size

glScalef(loopedParticle.Scale, loopedParticle.Scale, loopedParticle.Scale);

//set the colour according to the particle

glColor4f(loopedParticle.Colour.r, loopedParticle.Colour.g, loopedParticle.Colour.b, loopedParticle.Colour.a);

//draw the particle

glBegin(GL\_QUADS);

glTexCoord2d(0, 0);

glVertex3f(-ParticleSize, -ParticleSize, 0);

glTexCoord2d(1, 0);

glVertex3f(ParticleSize, -ParticleSize, 0);

glTexCoord2d(1, 1);

glVertex3f(ParticleSize, ParticleSize, 0);

glTexCoord2d(0, 1);

glVertex3f(-ParticleSize, ParticleSize, 0);

glEnd();

One that is in place the entire particle texture is drawn and the OpenGL colour/scale/translations are reset so other drawing commands from outside the particle system will not be affected.

glColor4f(1, 1, 1, 1);

glScalef(1, 1, 1);

glPopMatrix();

### Particle class

#### void Particle::Update(float dt)

The update method is very simple. It updates the position, colour, scale and age of a particle by adjusting the properties using its age.

For example the colour and scale of a particle are linearly interpolated to using the stored Start and End variables for each. The particle's age compared to the max age is used to work out what the value should be.

Age += dt;

Scale = \_startScale + (EndScale - \_startScale) \* (Age / MaxAge);

Position = Position + (MoveVector \* dt);

Colour = RGBA(\_startColour, EndColour, (Age / MaxAge));

Where the interpolating function is:

//Interpolates between 2 RGBAs

RGBA(RGBA start, RGBA end, float time) {

r = start.r + (end.r - start.r) \* time;

g = start.g + (end.g - start.g) \* time;

b = start.b + (end.b - start.b) \* time;

a = start.a + (end.a - start.a) \* time;

}

# Examples of extensibility

As already mentioned the particle system can be easily extended to add additional functionality. In this section I will explain how I used the *ParticleSystem* class and extended it to make something that looks like a firework.

Because the variables and methods in *ParticleSystem* are virtual they can be overridden by child classes to extend the functionality.

## Fireworks

The firework class is a child of *ParticleSystem* and so has access to all the same variables which can be changed like a normal particle system. A firework changes how the particle system works depending on what the firework is currently doing. To best model the firework I decided to have 2 important stages; **Launching** and **Exploding** which both drastically change how the particle system operates.

D:\Temp Stuff\launching.png

### Firework stages

A firework has 4 stages:

* **kReady** - the default value for when the firework is created. Not active
* **kLaunching** - when the firework is launching in the air
* **kPrepExplode** - for when the firework is about to explode
* **kFinished** - when the firework has finished exploding and there are no particles left

The use of these stages allows for different particle system behaviors to be set which will be explained later when illustrating how a typical firework works in the system.

### Firework variables

The firework class comes with several properties of its own that allow for easy definition of the main properties you want particles to have with the various stages of a firework. The firework class does not have its own unique additions to the particle system; its variables are merely used as shortcuts to changing the particle system's properties for future stages in the firework. The other properties of the particle system are still accessible and can be changed for the various stages regardless. It is these properties in the firework class that are the main ones that I anticipate will be the only ones a user will be concerned about as shortcuts.

//These LaunchVariables will move the origin of the particle system

float LaunchSpeed; //The speed which the launched firework moves at

Vector3 LaunchVector; //The direction the firework moves when its launching

float SparkSpeed; //The speed which the sparks move at when the rocket is launching

float SparkAngle; //The direction the sparks moves at when its rocket is launching, degrees

float SparkAngleVar; //The variance in angle in degrees of the sparks when the rocket is launching

RGBA ExplodeColour; //The colour of the firework when it explodes

float ExplodeParticleCount; //The number of particles that will be spawned on explosion

float ExplodeSpeed; //The speed the particles will move away from the center of the explosion

float ExplodeSpeedVar; //The variance in the speed which the firework explodes out at

Vector3 ExplodeGravity; //The gravity acting on particles from an exploded firework

bool ShouldEmitTrail; //Flag for whether the firework should have a trail or not when exploding

int UpdatesPerTrail; //The number of updates that take place before a trail is extended. Higher values mean better performance

float TrailLifeModifier; //The modifier to the length of a particle in the trail

### Firework methods

The firework class uses an override of *ParticleSystem*'s Update() method to apply additional functionality to the system by calling some of the other *Firework* specific methods, then going on to call *ParticleSystem*'s update function afterward.

#### void Firework:: LaunchWithDuration(float timeToExplosion)

This is the method that kickstarts the firework's sequence. A firework can only be launched when its state is kReady. Upon method call, variables for the particle system are set according to the Spark variables, as well as some default variable values.

#### void Firework::ExplodePrep()

This method puts the firework to the stage before exploding. This changes the particle system to some default values and ones defined through the Explode variables. This means that when the next phase is switched to these changes will be applied to the particles that are spawned in the explosion.

By default there is a slight downward gravity to the particles and any colour can be spawned.

#### void Firework::Explode()

This method represents the stage of the firework when it explodes out fully coloured. This is done by simply spawning the number of particles defined by the ExplodeParticleCount. The particles spawned will spawn with whatever settings the particle system currently has. Unmodified these will be the ones set by the method ExplodePrep(), but other variables may be modified as detailed later.

#### void Firework::Update(float dt)

In this method the state of the firework is updated and the particle system's behavior is adjusted accordingly. For example, if the firework is launching then the particle system is moved according to the launch vector. If the firework has been launching for long enough then it will move on to the explode preparation phase.

if (State == kLaunching) {

\_timeSinceLaunch += dt;

//move the origin of the particle system along the launch vector

Origin = Origin + (LaunchVector \* LaunchSpeed \* dt);

//If the firework has been launching for the target amount of time, then explode it

if (\_timeSinceLaunch > \_launchDuration)

ExplodePrep();

}

If the firework is prepared to explode then it moves to the exploding phase and calls the Explode() method which is detailed later.

If the firework is exploding then it checks to see if all the particles have been used up. If so that means the firework has finished exploding and so should move to the finished state.

The firework has an option to allow particle trails when exploding. This is handled in the firework's Update() method.

//loop through all of the active particles that aren't trails, if the firework is exploding

if (ShouldEmitTrail == true && State == kExploding) {

if (\_tailUpdatesCount == UpdatesPerTrail) {

\_tailUpdatesCount = 0;

for (int i = 0; i < (int)\_activeParticles.size(); i++) {

Particle \* loopedParticle = \_activeParticles[i];

// if not a trail then it is eligable to have a trail behind it

if (loopedParticle->IsFake == false) {

// create a new trail particle at its current position except with no velocity

ParticleSystem::SpawnParticle();

Particle \* newParticle = \_activeParticles.back();

//decrement the active particle count so its as if this particle doesn't exist (we don't want it to be counted since its a trail)

\_totalActiveParticles--;

//make it a trail

newParticle->IsFake = true;

//copy its settings from the particle it is being created from, but make sure it doesn't change colour or move

newParticle->Age = 0;

newParticle->MaxAge = (loopedParticle->MaxAge \* TrailLifeModifier) - loopedParticle->Age;

newParticle->Colour = loopedParticle->Colour;

newParticle->EndColour = loopedParticle->EndColour;

newParticle->EndScale = loopedParticle->EndScale;

newParticle->MoveVector = Vector3(0, 0, 0);

newParticle->Position = loopedParticle->Position;

newParticle->Scale = loopedParticle->Scale;

}

}

}

else

\_tailUpdatesCount++;

}

What the piece of code does is it loops through all the particles in the system and checks to see if it is a trail or not. Trail particles are considered fake particles and are not counted (it would be silly to have trails of trails). If one of the looped particles is an actual particle then it should get a trail. A trail is nothing more than a fake particle which takes on the properties of its predecessor. A modification is applied to the age of the trail particle based upon the TrailLifeModifier.

Trails may only be applied after a certain amount of updates have been called to save on performance. These are defined by the UpdatesPerTrail variable.

#### bool Firework::HasFinishedExplosion()

#### This method checks to see if the firework has finished exploding. If a firework has 0 active particles then it is considered to have finished exploding.

### Firework modification

The firework properties each change different properties of the particle system during their respective stages. For example, when the firework is launched in the 'void Firework::LaunchWithDuration(float timeToExplosion)' method:

//load values for launching. The particles spawned will represent the sparks

Angle = SparkAngle;

AngleVariance = SparkAngleVar;

Gravity = Vector3(0, -0.01f, 0);

MoveSpeed = SparkSpeed;

StartScale = 0.5f;

EndScale = 0.1f;

As you can see, it merely just applies values to the parented particle system itself. If you wanted further modifications to be applied to a launching firework you can simply edit the other particle system properties as follows:

//launch the firework

newFirework->LaunchWithDuration(launchTime);

newFirework->EmissionRate = 0.001f;

Similarly properties can be changed for when the firework is about to explode. Because of the way the firework operates, you can only change the properties of the particle system in the **kPrepExplode** stage. After that stage particle regeneration is turned off and all the particles will have already spawned, so your changes will not be applied to them. That's not to say however you could turn regeneration back on- though it wouldn't look like a firework anymore!

An example of how you would change the properties for a firework before it explodes is in the Update() method in the main project file (Screensaver.cpp). When you are looping through all active fireworks to update them you can also check to see which ones are about to explode, and apply any changes there for when the firework is about to explode out.

void Update(int i) {

//update the fireworks

for (int i = 0; i < (int)\_fireworks.size(); i++) {

Firework \* loopedFirework = \_fireworks[i];

//Change the size of a particle for explosion

if (loopedFirework->State == kPrepExplode) {

loopedFirework->ParticleSize = 2;

}

...

}

}

This change will mean when the firework explodes next update each of the newly generated particles for the explosion will start off with size 2.

## Dazzle Fireworks

To further demonstrate the extensibility of objects using the particle system I created a dazzling firework. A dazzling firework is one which has rapid short bursts of light appearing around the firework when it is exploding.

To achieve this I override some of the virtual methods that are part of *Firework*.

D:\Temp Stuff\launching.png

### Dazzle Firework variables

Because the extension upon *Firework* is relatively small there are only a few variables to get the desired effect.

int MaxDazzleParticles; //The max number of dazzle particles that can be active at a time

float DazzleLifeSpan; //The length of time a dazzle particle will last for

float MaxDazzleTime; //The max amount of time the firework can dazzle for before it stops

Vector3 DazzlePosVar; //The position variance dazzle particles spawn at

RGBA DazzleColour; //The colour which the dazzle particles will spawn with

### Dazzle Firework methods

The class relies on overriding methods in both *Firework* and *DazzleFirework* to get a dazzling effect.

#### void DazzleFirework::Update(float dt)

*DazzleFirework* does not interfere with the overall functionality of *Firework*. What it does is it keeps track of how long a firework has been exploding for. This is used later in the ReplenishParticles() method when used for a dazzling effect.

#### void DazzleFirework::Explode()

This method overrides the *Firework* method. It works by first exploding the parented *Firework.*

//explode the normal firework first

Firework::Explode();

Next it sets new values for the particle system's properties, based off default values and the property variables of *DazzleParticle*.

//now set up the dazzle system

ShouldRegenParticles = true;

ParticleLife = DazzleLifeSpan;

OriginVariance = DazzlePosVar;

StartColour = DazzleColour;

StartColourVariance = RGBA(0, 0, 0, 0);

EndColour = RGBA(-1, -1, -1, 0);

EndColourVariance = RGBA(0, 0, 0, 0);

Gravity = Vector3(0, 0, 0);

StartScale = 0.75f;

EndScale = 0;

Finally the MaxParticles of parenting particle system is incremented. This is to make enough 'room' for new particles to be allowed to spawn. This is done by taking the amount of particles that were spawned when the firework explodes and add them to the amount of dazzle particles that can spawn.

//increase the max amount of particles possible to allow for dazzle particles to spawn

MaxParticles = ExplodeParticleCount + MaxDazzleParticles;

#### void DazzleFirework::ReplenishParticles(float dt)

This method overrides *ParticleSystem*'s ReplenishParticles() method. There are 3 situations that need to be considered when replenishing particles in a dazzle firework.

Firstly if the firework has finished exploding, it shouldn't be allowed to regenerate particles.

//Don't allow replenishment when the firework has finished

if (State == kFinished)

return;

Secondly if the particle isn't exploding (i.e. if it is launching), then normal replenishment can take place.

//allow normal replenishing when not exploding

if (State != kExploding)

Firework::ReplenishParticles(dt);

Finally if the firework is exploding then it needs to check to see if it can still spawn dazzle particles. Currently the condition is that a dazzle particle cannot be spawned if the firework has been exploding past a certain period of time. If this period of time has not been surpassed then replenishment of particles can take place (which will be dazzle particles).

else if (State == kExploding) {

//if exploding, only allow dazzle replenishment if it hasn't passed the max time

if (\_timeSinceDazzleStarted < MaxDazzleTime)

Firework::ReplenishParticles(dt);

}

## Modelling other effects

Due to the flexibility of the system you can easily model other effects using either just the base particle system or by extending the particle system class similar to how I have created the fireworks.

### Fire torch



ParticleSystem \* torch = new ParticleSystem(\_particleTexture, 10000);

torch->Origin = Vector3(0, 0.5f, -1);

torch->ShouldRegenParticles = true;

torch->StartColour = RGBA(1, 1, 0.01, 1);

torch->EndColour = RGBA(-1, -1, -1, 0);

torch->OriginVariance = Vector3(0.02, 0.0f, 0);

torch->MoveSpeed = 0.05f;

torch->MoveVariance = 0.04f;

torch->Angle = 0;

torch->AngleVariance = 30;

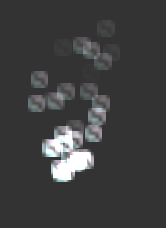
torch->EmissionRate = 0.0001f;

torch->ParticleLife = 2;

torch->ParticleLifeVariance = 3;

torch->Gravity = Vector3(0, 0.01f, 0);

### Bubble blower



For this effect I changed the particle texture I was loading which had hints of pink and blue in it.

ParticleSystem \* bubble = new ParticleSystem("Bubble.tga", 30);

bubble->Origin = Vector3(0, 0.5f, -1);

bubble->ShouldRegenParticles = true;

bubble->StartColour = RGBA(1, 1, 1, 1);

bubble->EndColour = RGBA(-1, -1, -1, 0);

bubble->OriginVariance = Vector3(0.04, 0.0f, 0.2);

bubble->MoveSpeed = 0.05f;

bubble->MoveVariance = 0.04f;

bubble->Angle = 0;

bubble->AngleVariance = 30;

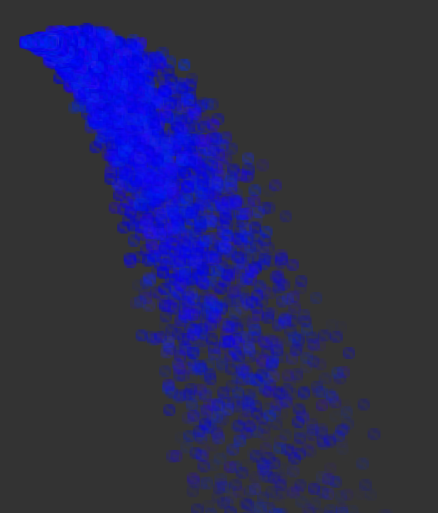
bubble->EmissionRate = 0.1;

bubble->ParticleLife = 4;

bubble->ParticleLifeVariance = 1;

bubble->Gravity = Vector3(0, 0.01f, 0);

### Waterfall



To achieve the waterfall effect I most notably changed the blend function used in the particle system.

ParticleSystem \* waterfall = new ParticleSystem(\_particleTexture, 4000);

waterfall->Origin = Vector3(0, 1.5f, -1);

waterfall->ShouldRegenParticles = true;

waterfall->StartColour = RGBA(0, 0, 1, 1);

waterfall->StartColourVariance = RGBA(0.2, 0.2, 0.1, 0);

waterfall->EndColour = RGBA(-1, -1, -1, 0);

waterfall->OriginVariance = Vector3(0.04, 0.0f, 0.2);

waterfall->MoveSpeed = 0.15f;

waterfall->MoveVariance = 0.05f;

waterfall->Angle = 90;

waterfall->AngleVariance = 30;

waterfall->EmissionRate = 0.0005;

waterfall->ParticleLife = 4;

waterfall->ParticleLifeVariance = 4;

waterfall->Gravity = Vector3(0, -0.1f, 0);

waterfall->BlendFactor = GL\_ONE\_MINUS\_SRC\_ALPHA;

# Testing

To make sure the particle system functioned I used unit testing to make sure each method in the system worked as expected. By testing them in isolation I could be sure they would interact.

To test the fireworks all I needed to do was just observe. Because it was running off the *ParticleSystem* class I could be certain its functions will still operate. All I needed to test was if the right colours were being applied to the particles upon explosion.

I tested the screen saver on a different computer with a different resolution to make sure the system worked accordingly.

# Enhancement

There are a few enhancements that I would like to do in further developments of the particle system.

Currently you can only set 1 colour for the system to spawn. You can of course vary it by the colour variance but I believe it would be good to also specify a list of colours that the system should only be allowed to spawn. This could be implemented by sub-classing *ParticleSystem* and overriding it's SpawnParticle method in your new class. Before calling the parent SpawnParticle method you would change the system's StartColour to a RGBA from your sub-class's list of valid colours.

Another worthwhile addition to the particle system would be to add particle acceleration to a particle's move speed. For example you could make it that a particle should speed up or slow down by a certain amount each update. This would have been good for the fireworks display as upon watching videos of firework explosions the individual particles tend to slow down to near a standstill at the end of their life.

The addition of an interface would be useful for toying with a particular particle system in the scene and helping to quickly create new effects instead of the trial and error approach that entering values in by code and restarting the application tends to produce.

The screensaver itself could do with a few improvements. I believe that if the background of the screensaver changed periodically it would be a nice addition and help break it up.

# Evaluation

Upon completion I am very pleased with the look of the screensaver. The randomly generated colours are pleasant to look at with the background. Even with high values of fireworks spawning (100 or so) the screensaver looks really nice.

In my humble opinion I believe I have created a highly extensible and robust particle system for use in an OpenGL project. Given time I could expand the base classes provided with *ParticleSystem* to allow for different variations of systems (for example ones with multiple origins or ones with lists of colours to spawn). Regardless, on its own I have demonstrated how widely *ParticleSystem* can be adapted to provide quite complex sequences. Because the application has been written in C++ with OpenGL the project is highly portable to Mac OSX, iOS and Android devices.