WHITE PAPER

Particle System

Realtime smoke rendering with OpenGL® ES 2.0

Roberto Lopez Mendez July 2013

INTRODUCTION

In Computer Graphics the term “particle system” refers to a technique that makes use of a large number of very small sprites or other graphic objects to simulate chaotic systems, natural phenomena and other process which are otherwise very hard to reproduce with conventional rendering techniques. Usually phenomena such as fire, explosions, smoke, fog, snow, dust, etc, are graphically simulated using particle systems. Typically in a particle system there are three stages: particle creation, simulation and rendering.

Particle creation is governed by what is referred as an emitter. The emitter is the source of particles and its location in the 3D space defines where particles are generated and their properties or behaviour parameters. Parameters can include any relevant property or magnitude such as colour, mass, initial velocity vector, particle lifetime, etc. Usually these parameters are random numbers in a given interval.

Once particles are created we need to update their position in the space as time passes as well as any other time-dependent parameter, velocity, acceleration, colour, etc. This is what is known as simulation and where physics’ Laws of Motion are considered to take into account the influence of any external force such as gravity, wind, friction, etc. Usually collision detection of particles with other relevant objects is considered but not the collision between the particles themselves as it can be really computationally expensive. At this stage all particles are also checked to see if they have exceeded their lifetime, in which case they are removed from the simulation.

Finally we need to render each particle after the update is complete. Typically each particle has a partially transparent associated texture (in many cases animated for better realism) mapped onto a plane that is always oriented to the camera, known as sprite. Nevertheless particles can be also rendered simply as a point. Particle simulation and rendering is performed in every frame of animation.

This whitepaper describes how to implement a simple particle system to simulate smoke.

THE EMITTER

The emitter is the source of particles responsible for creating particles with a given initial set of properties. It is important to define correctly the emitter. A suitable emitter will simplify the work on the simulation and rendering stages and will contribute decisively to achieve a better final result in terms of realism. Particles could be emitted just from a simple point or from a surface or a volume or from along a line or curve. Emitters could be classified as isotropic or anisotropic whether the particles are emitted equally in all directions or not.

Particle properties are initialized by the emitter and could be as many as needed to describe the particle behaviour and achieve a suitable simulation. A basic set of particle properties include position, velocity vector, colour and lifetime. The last parameter is very important as it defines how long a given particle will exist.

In the example an emitter of disc shape is implemented in the class DiscEmitter. By default the emitter creates particles within a region defined by a circle of unitary diameter with a velocity vector in the direction of positive Z axis (up) with an azimuth angle in the interval [0, π/10]. Other particle parameters are lifetime and delay. A delay parameter is used to avoid all particles being emitted at the same time when the program starts.

Particle generation is performed by requesting to DiscEmitter object for a particle. Particle parameters are created using pseudo random numbers obtained by means of rand() function.

```
void DiscEmitter::getParticle(Particle &part)
{
      /* Generate  a random number in the interval [0,1]. */
      float rad = (float)rand()/((float)RAND_MAX);

      /* Generate  a random number in the interval [0,1]. */
      float polarAngle = (float)rand()/((float)RAND_MAX);
      polarAngle *= 2 * M_PI;

      part.initPos.x = discRadius * rad * cos(polarAngle);
      part.initPos.y = discRadius * rad * sin(polarAngle);
      part.initPos.z = 0.0f;

      /* Generate  a random number in the interval [0,1]. */
      float azimuthAngle = (float)rand()/((float)RAND_MAX);
      azimuthAngle *= maxEmissionAngle;

      part.initVel.x = sin(azimuthAngle) * cos(polarAngle);
      part.initVel.y = sin(azimuthAngle) * sin(polarAngle);
      part.initVel.z = cos(azimuthAngle);

      part.lifetime = (float)rand()/((float)RAND_MAX);

      /* Generate  a random number in the interval [0,1]. */
      part.delay = (float)rand()/((float)RAND_MAX);

}
```

THE SIMULATION

In the simulation stage particle parameters are updated in time according to the rules governing the particle behaviour. We need to update the particle state before it is rendered and the most suitable place to do this is in the vertex shader. In this way per particle calculations are parallelized. The below code describes how particle parameters are updated in the vertex shader.

```
attribute vec3 a_v3Position;
attribute vec3 a_v3Velocity;
attribute vec3 a_v3ParticleTimes;
uniform mat4 mvp;
uniform vec3 u_v3gravity;
uniform vec3 u_v3colour;

varying float v_ageFactor;
varying vec3 v_v3colour;

void main()
{
      vec3 newPos;
      float ageFactor;
      float delay       = a_v3ParticleTimes.x;
      float lifetime    = a_v3ParticleTimes.y;
      float age         = a_v3ParticleTimes.z;

      if( age  > delay )
      {
            /* Particle motion equation. */
            newPos = a_v3Position + a_v3Velocity * (age - delay) +
                     0.5 * u_v3gravity * (age - delay) * (age - delay);

            ageFactor = 1.0 - ((age - delay) / lifetime);
            ageFactor = clamp(ageFactor, 0.0, 1.0);
            /* The older the particle the smaller its size. */
            gl_PointSize = ageFactor * 250.0;
      }
      else
      {
             newPos = a_v3Position;
             /* If null size particle will not be drawn. */
             gl_PointSize = 0.0;
             ageFactor = 0.0;
      }

      /* Initializing varying. */
      v_ageFactor = ageFactor;
      v_v3colour = u_v3colour;

      /* Particle position. */
      gl_Position.xyz = newPos;
      gl_Position.w = 1.0;

      /* Apply matrix transformation. */
      gl_Position = mvp * gl_Position;
}
```

The lines above the main function are simple variable declarations. We have three variables of attribute type which define the particle's coordinates, velocity and a vector containing different time variables. Other three variables of uniform type are passed to the shader from the application code.

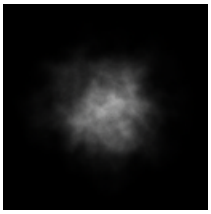
The outputs of the vertex shader are called varyings. Varyings are the available way to pass data from vertex shader to fragment shader. As we need to pass ageFactor and the particle base colour to the fragment shader we declare v_ageFactor and v_v3colour as varying type and fill them in the shader code.

The vertex shader checks first if the particle's age is greater than its delay time, i.e. if the particle is flying. In this case the particle position is updated using the formula for the uniformly accelerated motion. Additionally the age factor is calculated. The edge factor is used to control the value of gl_PointSize. The variable gl_PointSize is a built in variable that indicates the size in pixels of the point to be rasterized. The older the particle the smaller its size.

The variable glPosition is also a built in four vector component variable indicating the homogeneous vertex coordinate (x, y, z, w) and must be always filled by the vertex shader.

THE RENDERING

Once a particle's parameters have been updated in the simulation stage we are ready to draw the particle. This final operation is performed in the fragment shader. The below code implements the fragment shader used to draw the particles. Instead of drawing a simple point we will use point sprites. The point sprite primitive supported by OpenGL ES is GL_POINTS. A point sprite is a screen-aligned quad specified as a position and radius.



The position describes the center of the square and the radius is used to calculate the four coordinates of the quad that defines the sprite. The image on the left is used to texture the sprite quad.

The first line of the code sets the default precision qualifier. The next three lines are a simple declaration for varyings received from the vertex shader to be used in the shader code.

The program calculates the factor alphaFactor according to the value of the age factor. When the particle is very young or very old it is more transparent.

In the next step the value of gl_PointCoord is used to draw a textured point sprite where the transparency value is modulated by the previous calculated alphaFactor. gl_PointCoord is a vec2 built-in variable available only inside the fragment shader when the primitive being rendered is a point sprite. It takes values from 0.0 to 1.0 as we move from left to right and from bottom to top. The code illustrates how gl_poinCoords are used as a texture coordinate to draw a texture point sprite.

The final fragment color gl_FragColor is also modulated by the base colour passed from the vertex shader.

```
precision mediump float;

varying float v_ageFactor;
varying vec3 v_v3colour;
uniform sampler2D s_texture;

void main()
{
      vec4 texColour;
      vec2 texCoords;
      float alphaFactor;

      texCoords = vec2(gl_PointCoord.x, gl_PointCoord.y);

      /* Define an alpha modulation factor depending on
         the particle's size. */
      if(v_ageFactor  <= 0.5)
      {
            alphaFactor = 0.14 * v_ageFactor;
      }
      else
      {
            alphaFactor = -0.14 * v_ageFactor;
      }

      vec4 baseColour = vec4(v_v3colour.x, v_v3colour.y, v_v3colour.z,
                             1.0);
      texColour = texture2D(s_texture, texCoords);

      /* Modulate alpha component. */
      texColour.a = texColour.r * alphaFactor;


      gl_FragColor = vec4(texColour.r, texColour.r, texColour.r,
                          texColour.a) * baseColour;

}
```

THE APPLICATION

The application is responsible for initializing the window; EGL and OpenGL ES graphic subsystem (see the main function). In the `setupGraphics` function the texture is loaded by calling the function `initialiseTextureFromRawGreyscaleAlphaFile`. The texture is two bytes per pixel image. First byte encodes the gray scale colour and the second one the transparency. This is the reason why when the texture is specified by means of glTexImage2D the internal texture format is set to `GL_LUMINANCE_ALPHA`. Usually in particle system simulations based on CPU particles are kept in a

list. When the particle's age is greater than its lifetime the particle is removed from the list. New particles are added to the list as they are created by the emitter. In our case the application holds particles in the particle data array `particleData` where the number of particles is constant. Every frame the function `updateParticleAge` increases each particle's age but when it is greater than a particle's lifetime the particle's age is reset. In this way particles are reused over and over and we can keep constant the size of the particle data array.

Finally, in the `renderFrame` function once particles' age is updated the application sends particle data to the graphics pipeline, binds the texture, sets the sampler to point to the 0^{th} texture unit and draws the particles by calling the `glDrawArrays` function indicating the GL_POINTS primitive type.

Figure 1. Screen capture of the application running on Linux Arndale.