JOGL-ES Chapter 3. A Particle System

A particle system is a collection of simple elements (particles) moving through 3D space, their behavior governed by physical properties, such as gravity, friction, their mass, age, and proximity to other objects.

Particle systems hit the big time in the movie *Star Trek II: The Wrath of Khan* (or *Khaaaaaannnn*), where they were used to create a planet-engulfing wall of fire. Since then, they've become commonplace in games, employed for explosions, smoke, fog, blood, snow, clouds, laser beams, and many other effects.

The particle system shown in Figure 1 generates an unending fountain of fireballs which gradually burn out and disappear.

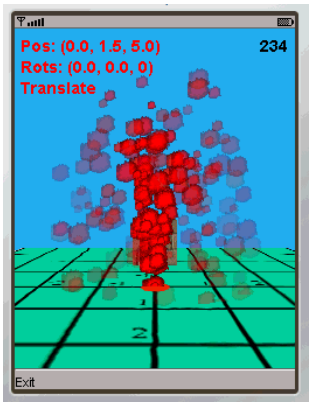

Figure 1. The Fountain of Fireballs.

The MIDlet, ParticlesES, utilizes the OES_point_sprite and OES_point_size_array core extensions in OpenGL ES 1.1, which allows each particle to be represented by a point sprite.

The particles come in different sizes, and are a mix of red and a fireball texture. Over time, each particle becomes more transparent, and eventually disappears. The 'death' of a particle is triggered either by it reaching a certain age or when it drops through the floor. The particle isn't dead for long – it's immediately reinitialized and ejected from the fountain's starting position on the floor.

Figure 2 shows the fountain from a different viewpoint, illustrating that point sprites behave like *billboards*, automatically turning to face the viewer without the need for programmer intervention.

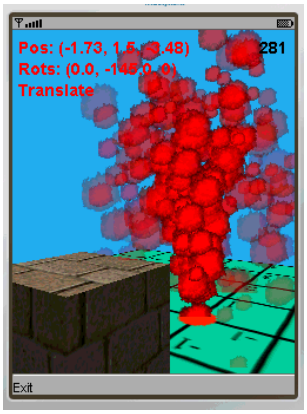

Figure 2. The Fountain Viewed from a Different Position

The sizes of the particles are adjusted depending on how close the camera is to the fountain's floor position. The particles are bigger in Figure 2 than in Figure 1 because the camera is nearer the fountain. This behavior requires code in the Particles class since, by default, a point sprite's size doesn't vary with the camera distance.

A slightly tricky form of alpha blending is employed when rendering the particles, to ensure they blend correctly, and that other objects in the scene (e.g. the textured cube) are drawn properly.

1. An Overview of ParticlesES

Figure 3 shows the class diagrams for the ParticlesES application; only public methods are shown.

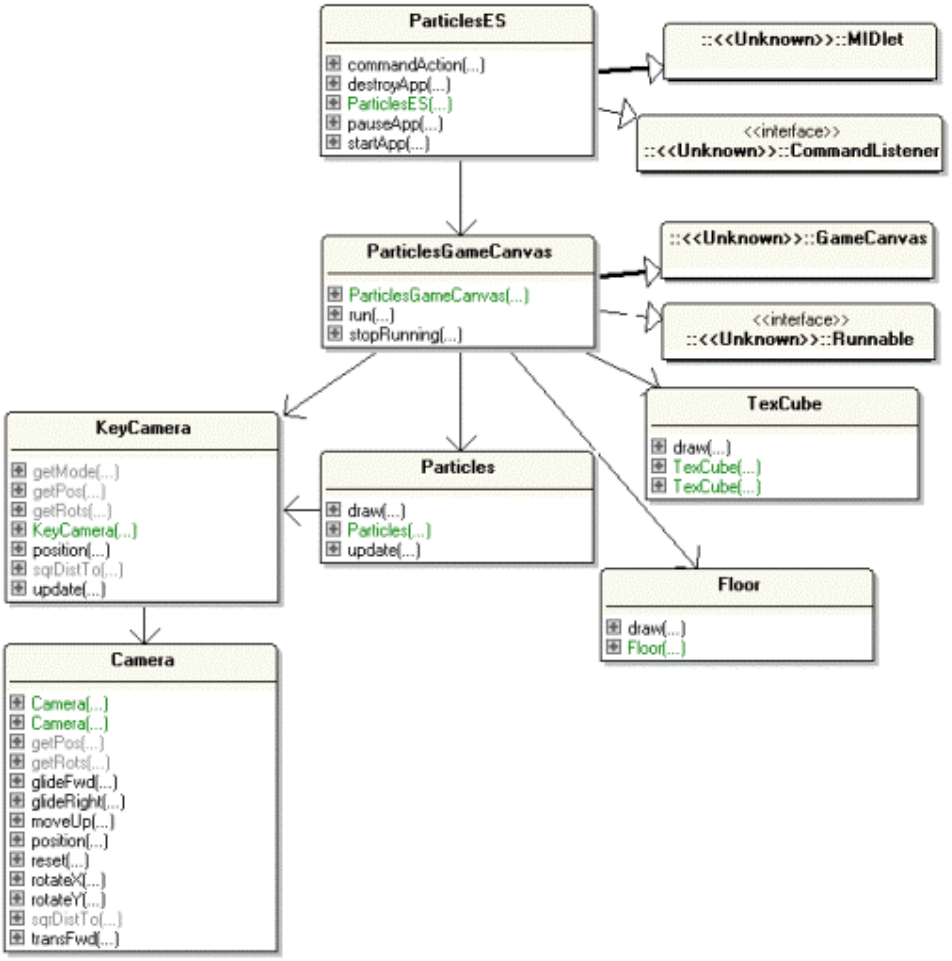

Figure 3. Class Diagrams for ParticlesES.

ParticlesES is the top-level MIDlet, and very similar to our earlier examples. ParticlesGameCanvas is also quite familiar – it draws the scene at the required frame rate.

Key presses are converted into camera movements with the help of KeyCamera and Camera. These classes are virtually unchanged from before, but have a new sqrDistTo() method to calculate the squared distance between the camera and fountain.

TexCube is employed to draw the textured cube, and Floor is in charge of the floor, both unaltered from earlier chapters.

Particles, the only completely new class, manages the creation, updating, and rendering of the particle system.

2. Creating the Fountain in the Scene

The particle system is instantiated in ParticlesGameCanvas.createScenery(), along with the floor and the cube:

```
// globals
private GL11 gl;       // for calling JOGL-ES

private KeyCamera keyCamera;
private Floor floor;
private Particles particles;
private TexCube texCube;


private void createScenery()
{
  floor = new Floor(gl, "/bigGrid.png", 8);   // 8 by 8 size

  particles = new Particles(gl, 0.0f, 0.0f, keyCamera);
              // positioned at the origin on the floor

  texCube = new TexCube(gl, "/brick2.png", 0, 0.51f, -2.0f, 0.5f);
                                          // (x, y, z), scale
} // end of createScenery()
```

The Particles() constructor requires a reference to the camera, so keyCamera must be created before createScenery() is called.

Particles()'s arguments include the fountain's (x, z) floor position, which is the origin in this example. The Particles class fixes the y- axis position to be 0, so the fountain rests on the floor.

The OpenGL ES reference, gl, passed to the scenery objects is an instance of the GL11 class. In earlier examples, I used GL10, but GL11 is required here for the point sprites. GL11 inherits all the functionality of GL10, so there's no need to alter the GL10 method calls in the game canvas, floor, or cube.

2.1. Updating the Fountain

ParticlesGameCanvas updates the particle system in its run() method via a call to Particles.update():

```
// in run() in ParticlesGameCanvas
keyCamera.update( getKeyStates() );
particles.update();
drawScene();
```

It's important to update the camera before updating the particle system, since particle sizes are calculated based on the current camera position.

2.2. Rendering the Fountain

The scene is rendered by ParticlesGameCanvas.drawSceneGL(), in the same way as earlier examples. The code fragment below shows how the floor, cube, and particles are drawn with their draw() methods:

```
// drawSceneGL() in ParticlesGameCanvas
floor.draw();
texCube.draw();
particles.draw();
```

Particles.draw() must be called last, after the other objects have been rendered. This ensures that the particles will be drawn at the correct position with respect to the other objects in the scene. I'll explain this in more detail when I discuss the implementation of Particles.draw().

3. The Camera's Squared Distance

The only new feature of the Camera and KeyCamera classes is sqrDistTo().

KeyCamera.sqrDistTo() simply passes its input arguments on to the Camera instance, and returns the result:

```
// in the KeyCamera class
public double sqrDistTo(double x, double z)
{  return camera.sqrDistTo(x, z);  }
```

The Camera class computes the squared distance between the supplied (x, z) coordinate and the camera's current (x, z) position.

```
// in the Camera class
// globals
private double xCamPos, yCamPos, zCamPos;
              // current (x,y,z) camera position

public double sqrDistTo(double x, double z)
// return squared distance between (x, z) and camera's (x, z) posn
{
  double xDiff = xCamPos - x;
  double zDiff = zCamPos - z;
  return  (xDiff*xDiff) + (zDiff*zDiff);
}  // end of sqrDistTo()
```

sqrDistTo() is called by the Particles object during each update, so it's important that it be fast. That's why the camera's y- coordinate isn't utilized, and also why the distance is squared. A squared value avoids the cost of a square root calculation, and also means that the method's result is always positive.

4. Defining the Particles

Different particle systems can possess a wide range of attributes, such as position, velocity, acceleration, mass, color, translucency, shininess, age, and even health (e.g. an ill particle is more likely to die).

The most important attributes for my fountain particles are their initial velocities in the x-, y-, and z- directions, which are used to calculate their (x, y, z) position over time. Each particle has an age (which starts at 0 when it's emitted from the floor), a size, and color. A particle will be red initially, and use a fireball texture, but gradually becomes transparent as it gets older. A particle's size is randomly chosen at initialization time, and varies at run time depending on the camera's distance from the fountain.

A particle dies either when it passes through the floor, or when its age reaches a predefined maximum. The particle is reinitialized with a new velocity and size, and fired out of the floor again.

4.1. Velocities and Positions

The velocity components up the y-axis and across the XZ plane are shown in Figure 4.

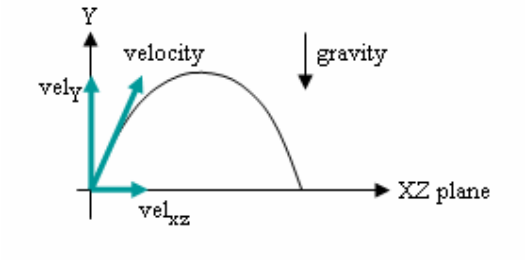

Figure 4. A Particle's Velocity Components.

Each particle follows a parabolic trajectory depending on the initial velocities (vel_y and vel_{xz} in Figure 4) and gravity. Many other physical factors could be added in, such as mass and air resistance. My particles can't collide, and don't bounce off the floor or cube.

Using standard Newtonian physics, the vertical distance traveled at time t is:

$$dist_y = vel_y - 1/2\ g\ t^2$$

The Particles class utilizes the current age of the particle as the t value; g stands for gravity.

The distance traveled over the XZ plane at time t is:

$$dist_{XZ} = vel_{XZ}\ t$$

The particles can shoot off in different directions over the floor (over the XZ plane), so vel_{XZ} is split into x- and z- component velocities to allow the x- and z- distances to be calculated.

Figure 5 shows how vel_{XZ} can be separated into vel_X and vel_Y, by employing a randomly chosen angle (rad) .

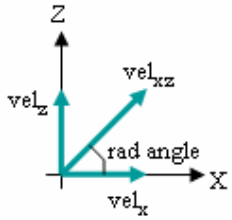

Figure 5. A Particle's Velocity Components over the XZ Plane.

The component velocities according to Figure 5 are:

$vel_x = vel_{XZ} \cos(rad)$ and $vel_z = vel_{XZ} \sin(rad)$

The component distances traveled at time t are then:

$dist_x = vel_x \, t$ and $dist_z = vel_z \, t$

4.2. Data Structures for Representing the Particles

A particle's attributes are its x-, y-, and z- axis velocities, its age, position, color, and size. Each of these attributes is represented by its own array in the Particles class, which holds the attribute values for all the particles:

```
// globals in the Particles class
private float[] velX, velY, velZ;
                          // velocities in x-, y-, z- directions
private float[] age;
private float[] posns;    // (x,y,z) positions
private byte[] cols;      // colors (RGBA)
private float[] origSizes, currSizes;   // sizes
```

The Particles class isn't very object-oriented; a better OO design would be to utilize a Particle class (no 's') for all of a particle's attributes (and its behavior), and store an array of Particle objects in the Particles class. Multiple arrays are employed instead for rendering time efficiency – point sprites can be quickly initialized when all the particles data is collected in arrays rather than spread over Particle objects.

It may seem inconsistent that the x-, y-, and z- axes velocities are held in three arrays, while the (x, y, z) positions are stored in one. The reason is the same as before – the point sprites can be updated much faster if all the position information is in one place. Since the point sprites don't use the velocity values directly, those values are easier to understand if separated into three arrays.

The particles are rendered by GL11.glDrawArrays() which utilizes buffers initialized with the particles' positions, colors, textures, and point sizes. These are defined globally as:

```
// globals in the Particles class
private FloatBuffer posnsBuf;   // (x,y,z) positions
```

```
private ByteBuffer colsBuf;     // colors (RGBA)
private FloatBuffer sizesBuf;   // point sizes (floats)
private ByteBuffer texBuf;      // RGBA texture image
```

An obvious question is why are the positions, colors, and sizes stored twice, both in arrays and in buffers? Once again the answer is efficiency – it's faster to update arrays than byte buffers. The arrays data is modified first, particle by particle, and then only single array copies are needed to adjust the buffers.

There's no texture array corresponding to the texture buffer, texBuf. An array isn't needed since a particle's texture doesn't change at run time. Also, all the particles use the same fireball image, so texBuf only holds a single texture.

5. Constructing the Particles

The Particles constructor stores the fountain's (x, z) position on the floor, camera details, and delegates the particles set-up to loadTexture() and initParticles().

```
// globals
private GL11 gl;
  /* use GL11 so OES_point_sprite and OES_point_size_array
     are available */
private Random rand;
private float xStart, zStart;  // fountain source

// camera-related
private KeyCamera keyCamera;
private double initialSqrDist;
private float sqrDistFrac;

private int texNames[];   // for the texture name


public Particles(GL11 gl, float x, float z, KeyCamera kc)
{
  this.gl = gl;
  xStart = x;
  zStart = z;
  keyCamera = kc;

  rand = new Random();

  // set initial squared distance and fraction
  initialSqrDist = keyCamera.sqrDistTo((double)xStart,
                                       (double)zStart);
  sqrDistFrac = 1.0f;

  // load the particles texture
  loadTexture(TEX_FNM);

  // generate a texture name
  texNames = new int[1];
  gl.glGenTextures(1, texNames, 0);

  initParticles();
```



```
}  // end of Particles()
```

A camera reference is needed so that Particles can calculate its proximity to the fountain when resizing its point sprites. The camera's initial squared distance (initialSqrDist) is retrieved, and the camera's current distance from the fountain is stored as a fraction of that initial value (in sqrDistFrac). The fraction is always positive because of the squared distances, which simplifies matters when scaling the point sizes.

5.1. Loading the Texture

All the particles use the same fireball texture, shown in Figure 6.

Figure 6. The Fireball Texture.

The edges of the 64-by-64 pixel image are transparent, and its center is white.

loadTexture() performs almost the same tasks as the version seen in earlier chapters. It loads the image stored in the specified file into a byte buffer as before, but now includes the image's alpha channel.

```
// globals
private ByteBuffer texBuf;       // RGBA texture image
private int imWidth, imHeight;   // dimensions of the texture image


private void loadTexture(String fnm)
{
  try {
    Image image = Image.createImage(fnm);
    imWidth = image.getWidth();
    imHeight = image.getHeight();
    int pixels[] = new int[imWidth];   // for a row of pixels

    texBuf = ByteBuffer.allocateDirect(imWidth * imHeight * 4);
                // make space for RGB and A

    for (int i = imHeight-1; i >= 0; i--) {
      image.getRGB(pixels, 0, imWidth, 0, i, imWidth, 1);
                                          // get a pixel row
      for (int j = 0; j < imWidth; j++) {
        // store each pixel in the row as four bytes in RGBA order
        texBuf.put((byte) (pixels[j] >> 16 & 0xff));  // red
        texBuf.put((byte) (pixels[j] >> 8 & 0xff));   // green
        texBuf.put((byte) (pixels[j] >> 0 & 0xff));   // blue
        texBuf.put((byte) (pixels[j] >> 24 & 0xff));  // alpha
      }
    }
    texBuf.rewind();
    System.out.println("Loaded texture from " + fnm);
  }
```



```
  catch (Exception e)
  {  System.out.println("Error loading texture from " + fnm); }
}  // end of loadTexture()
```

The image is read in reverse row sequence to flip it vertically in the buffer. This orders the texture correctly for OpenGL's coordinate system, where (0, 0) is the lower left corner.

Each ARGB pixel is written into the buffer as four bytes in RGBA order (i.e. the alpha changes position). This format is required by GL11.glTexImage2D() when it prepares the texture for rendering.

5.2. Particle Initialization

initParticles() spends most of its time creating correctly sizing global arrays and buffers. The job of filling the arrays with values is delegated to reset(), and those values are copied across to the buffers via fast array copying at the end of initParticles().

```
// globals
private static final int NUM_PARTS = 250;    // number of particles


private float[] velX, velY, velZ;
                           // velocities in x-, y-, z- directions
private float[] age;
private float[] posns;     // (x,y,z) positions
private byte[] cols;       // colors (RGBA)
private float[] origSizes, currSizes;   // sizes

private FloatBuffer posnsBuf;   // (x,y,z) positions
private ByteBuffer colsBuf;     // colors (RGBA)
private FloatBuffer sizesBuf;   // sizes


private void initParticles()
{
  velX = new float[NUM_PARTS];
  velY = new float[NUM_PARTS];
  velZ = new float[NUM_PARTS];
  age = new float[NUM_PARTS];

  // create positions array and buffer
  posns = new float[NUM_PARTS * 3];
  ByteBuffer bb = ByteBuffer.allocateDirect(NUM_PARTS * 3*4);
            // each position is a 3-float (x,y,z) coord
  posnsBuf = bb.asFloatBuffer();

  // create colors array and buffer
  cols = new byte[NUM_PARTS * 4];
  colsBuf = ByteBuffer.allocateDirect(NUM_PARTS * 4);
             // each color is a 4-byte RGBA value

  // create point sizes arrays and buffer
  origSizes = new float[NUM_PARTS];
  currSizes = new float[NUM_PARTS];
  ByteBuffer sb = ByteBuffer.allocateDirect(NUM_PARTS * 4);
  sizesBuf = sb.asFloatBuffer();
```



```
  getPointSizesInfo();

  // initialize the particles
  for(int i=0; i< NUM_PARTS; i++)
    reset(i);

  // fill the particle buffers
  posnsBuf.put(posns).rewind();
  colsBuf.put(cols).rewind();
  sizesBuf.put(currSizes).rewind();
}  // end of initParticles()
```

posns[] stores the (x, y, z) coordinates for all the particles, so must be sized at 3*NUM_PARTS. A particle's colors consist of four values (the red, green, blue, and alpha channels), so the color array needs to be 4*NUM_PARTS large.

The buffers use different types – posnsBuf and sizesBuf hold floats (for the positions and sizes respectively), while colsBuf gets by with bytes since a color can only range between 0 and 255.

There are two particle sizes arrays: origSizes[] and currSizes[]. As their names suggest, origSizes[] holds the original sizes for the particles, while currSizes[] contains the current sizes. The original sizes have to be retained since a particle's current size is calculated using its original size and the camera's fractional squared distance from the fountain.

5.3. Getting Point Size Information

getPointSizesInfo() retrieves the minimum and maximum point sizes supported by the device, and uses them to set a default point size for the particles. An upper bound for the maximum point size is also enforced.

```
// globals
private static final float MAX_POINT_SIZE = 50.0f;

private float minPointSize, maxPointSize, pointSize;


private void getPointSizesInfo()
{
  float[] range = new float[2];
  gl.glGetFloatv(GL11.GL_ALIASED_POINT_SIZE_RANGE, range, 0);
  minPointSize = range[0];
  maxPointSize = (MAX_POINT_SIZE < range[1]) ?
                                    MAX_POINT_SIZE : range[1];
                              // limit the maximum point size
  // store a default point size
  pointSize = (maxPointSize + minPointSize)/4.0f;
}  // end of getPointSizesInfo()
```

The minimum and maximum point sizes are retrieved in a float array by calling GL11.glGetFloatv() with the GL11.GL_ALIASED_POINT_SIZE_RANGE attribute. This isn't the usual way of accessing the device's point sizes in OpenGL applications. Typically the minimum is assumed to be 1.0f, and the maximum is retrieved with:


```
float[] maxSize = new float[1];
gl.glGetFloatv(GL11.GL_POINT_SIZE_MAX, maxSize, 0);
System.out.println("Max point size: " + maxSize[0]);
```

When I tried using this code in getPointSizesInfo(), the result was:

```
GLConfiguration: glGetNumParams called with pname=33063
Max point size: 0.0
```

33063 is the integer value for GL11.GL_POINT_SIZE_MAX, which doesn't appear to be supported by GL11.glGetFloatv() in the WTK 2.5.

5.4. Assigning a Particle its Initial Values

reset() assigns values to the i^{th} particle in the system by initializing various global arrays in the Particles class. reset() is called in two situations: when a particle is first created, and when it's reinitialized after dieing.

The values assigned to a particle are: x-, y-, and z- axis velocities, an age (0), a starting position (the base of the fountain), a color (red), and a point size. The velocities and size are generated semi-randomly.

```
// globals
private float xStart, zStart;     // fountain source

private float[] posns;
private byte[] cols;
private float[] origSizes, currSizes;


private void reset(int i)
// assign new values to particle i
{
  initVels(i);
  age[i] = 0;

  // set (x,y,z) starting position (the fountain's base)
  posns[i*3] = xStart;
  posns[i*3+1] = 0;    // always start on the floor
  posns[i*3+2] = zStart;

  // set RGBA colors between 0 and 255
  cols[i*4] = (byte) 255;    // bright red
  cols[i*4+1] = (byte) 0;    // no green
  cols[i*4+2] = (byte) 0;    // no blue
  cols[i*4+3] = (byte) 200;  // alpha (starts fairly opaque)

  // initialize point size
  origSizes[i] = genPointSize();
  currSizes[i] = scalePointSize( origSizes[i] );
}  // end of reset()
```

The i value passed into reset() is used as an index into the global arrays.

Creating Velocities

Three velocities are required along the x-, y-, and z- axes.

```
// globals
private static final float VEL_XZ0 = 0.7f;
private static final float VEL_Y0 = 9.81f;
private float[] velX, velY, velZ;


private void initVels(int i)
{
  float velXZ = genVelocity(VEL_XZ0);   // XZ plane velocity

  double rad = Math.toRadians(rand.nextInt(360));
                            // a random angle between 0-360
  float velX0 = (float)(velXZ * Math.cos(rad));   // vel along x-axis
  float velZ0 = (float)(velXZ * Math.sin(rad));   // vel along z-axis

  velX[i] = velX0;
  velY[i] = genVelocity(VEL_Y0);   // vertical velocity
  velZ[i] = velZ0;
}  // end of initVels()
```

Two velocities are generated semi-randomly by calling genVelocity(), one for the XZ plane, the other up the y-axis. The XZ value is divided into x- and z- components by randomly generating an angle and using trigonometry derived from Figure 5.

genVelocity() generates a velocity within ±VEL_PERC percent of the supplied velocity.

```
// global
private static final float VEL_PERC = 0.2f;
                        // percentage variation (+/- 20%)

private float genVelocity(float startVel)
{
  float velVar = startVel * rand.nextFloat()*VEL_PERC;
  float vel = (rand.nextInt(2) != 0) ? (startVel + velVar) :
                                       (startVel - velVar);
  return vel;
}  // end of genVelocity()
```

Creating Point Sizes

genPointSize() generates a point size using a coding style similar to genVelocity().

```
// globals
private static final float POINT_SIZE_PERC = 0.5f;
                        // percentage variation (+/- 50%)
private float pointSize;


private float genPointSize()
// generate pointSize +/- POINT_SIZE_PERC percent
{
  float psVar = pointSize * rand.nextFloat()*POINT_SIZE_PERC;
```



```
  float size = (rand.nextInt(2) != 0) ? (pointSize + psVar) :
                                        (pointSize - psVar);
  return size;
}  // end of genPointSize()
```

scalePointSize() scales the supplied size using the camera's fractional squared distance from the fountain.

```
// globals
private float sqrDistFrac;
   /* current distance of camera from fountain as a fraction
      of the initial squared distance */
private int minPointSize, maxPointSize;


private float scalePointSize(float sz)
{
  float ptSize = sz/sqrDistFrac;
  if (ptSize < minPointSize)
    ptSize = minPointSize;
  else if (ptSize > maxPointSize)
    ptSize = maxPointSize;
  return  ptSize;
}  // end of scalePointSize()
```

6. Updating the Particles

ParticlesGameCanvas repeatedly calls Particles.update() as it updates the scene.

update() increases the age of each particle and then either resets the particle's attributes, or updates its position, color, and size values. Once all the particles have been updated, the modified array values are copied over to the relevant buffers.

```
// globals
private static final float GRAVITY = 9.81f;
         // for the parabolic vertical distance equation

private static final float MAX_AGE = 2.5f;
private static final float AGE_INCR = 0.1f;


public void update()
{
  updateSqrDistFrac();

  // update each particle
  for(int i=0; i< NUM_PARTS; i++) {
    age[i] += AGE_INCR;

    // reset particle if below the floor (y < 0) or too old
    if ((posns[i*3+1] < 0) || (age[i] > MAX_AGE))
      reset(i);
    else {
      // vertical position = (v0*t) - (0.5*g*t*t)
      // use age as the time value
      float vertPos = (velY[i] * age[i]) -
                           (0.5f*GRAVITY * age[i]*age[i]);
```



```
      posns[i*3] = xStart + (velX[i] * age[i]);    // x
      posns[i*3+1] = vertPos/2.0f;        // y with scaling factor
      posns[i*3+2] = zStart + (velZ[i] * age[i]);  // z

      // reduce alpha based on age (a particle fades away over time)
      cols[i*4+3] = (byte)(255 - (int)(age[i]/MAX_AGE * 255));

      // modify point size
      currSizes[i] = scalePointSize(origSizes[i]);
    }
  }

  // update the particles buffers
  posnsBuf.put(posns).rewind();
  colsBuf.put(cols).rewind();
  sizesBuf.put(currSizes).rewind();
}  // end of update()
```

The if-test in update() resets a particle in two situations:

```
if ((posns[i*3+1] < 0) || (age[i] > MAX_AGE))
  reset(i);
```

The i^{th} particle is reset if it's dropped below the floor (i.e. its y- axis value is less than 0), or if the particle is too old. posns[i*3+1] denotes the y-axis position for the i^{th} particle, and age[i] is its age.

A particle's vertical position is updated using the parabolic $dist_y$ equation described earlier (see Figure 4). The calculated position (vertPos) is divided by two to improve the look of the simulation, by making the particles stay closer to the floor. Horizontal movement uses the $dist_x$ and $dist_z$ equations described above (see Figure 5).

A particle's color is made up of red, green, blue and alpha channels. For example, the red component of the i^{th} particle is cols[i*4], and its alpha value is cols[i*4+3]. update() reduces the alpha value, making the sprite more transparent.

The point size is scaled with scalePointSize() which uses the particle's original size in origSizes[i] and the camera's fractional squared distance from the fountain. The fraction is updated at the start of update() via a call to updateSqrDistFrac():

```
// globals
private double initialSqrDist;
private float sqrDistFrac;


private void updateSqrDistFrac()
{
  double sqrDist = keyCamera.sqrDistTo((double)xStart,
                                       (double)zStart);
                  // get current squared distance from camera
  sqrDistFrac = (float) (sqrDist/initialSqrDist);
} // end of updateSqrDistFrac()
```


At the end of update(), the values in the posns[], cols[], and currSizes[] arrays are copied over to their corresponding buffers. Array coping is generally faster than modifying the buffers incrementally for each particle.

7. Drawing the Particles

ParticlesGameCanvas renders the particles by calling Particles.draw(). draw() enables numerous OpenGL ES features, calls GL11.glDrawArrays(), and finishes by disabling the features. The required capabilities include alpha blending, texturing, coloring, point sprite functionality, and the enabling of the buffers for the particles' positions, colors, and sizes.

```
public void draw()
{
  // avoid problem with overlapping alpha blends
  gl.glDepthMask(false);
       // protect the depth buffer by making it read-only

  gl.glEnable(GL11.GL_BLEND);   // enable color and texture blending
  gl.glBlendFunc(GL11.GL_SRC_ALPHA, GL11.GL_ONE_MINUS_SRC_ALPHA);
  gl.glEnable(GL11.GL_TEXTURE_2D);
  gl.glEnable(GL11.GL_COLOR_MATERIAL);

  gl.glEnable(GL11.GL_POINT_SPRITE_OES);  // use point sprites

  // generate point sprite texture coords
  gl.glTexEnvx(GL11.GL_POINT_SPRITE_OES,
                 GL11.GL_COORD_REPLACE_OES, GL11.GL_TRUE);
  setTexture();

  // enable the buffers for rendering
  gl.glEnableClientState(GL11.GL_VERTEX_ARRAY);
  gl.glEnableClientState(GL11.GL_COLOR_ARRAY);
  gl.glEnableClientState(GL11.GL_POINT_SIZE_ARRAY_OES);

  // load the buffers
  gl.glVertexPointer(3, GL11.GL_FLOAT, 0, posnsBuf);
  gl.glColorPointer(4, GL11.GL_UNSIGNED_BYTE, 0, colsBuf);
  gl.glPointSizePointerOES(GL11.GL_FLOAT, 0, sizesBuf);

  gl.glDrawArrays(GL11.GL_POINTS, 0, NUM_PARTS);  // draw points

  // disable the buffers at the end of rendering
  gl.glDisableClientState(GL11.GL_VERTEX_ARRAY);
  gl.glDisableClientState(GL11.GL_COLOR_ARRAY);
  gl.glDisableClientState(GL11.GL_POINT_SIZE_ARRAY_OES);

  // switch off point sprites
  gl.glTexEnvx(GL11.GL_POINT_SPRITE_OES,
              GL11.GL_COORD_REPLACE_OES, GL11.GL_FALSE);
  gl.glDisable(GL11.GL_POINT_SPRITE_OES);

  // restore other states
  gl.glDisable(GL11.GL_COLOR_MATERIAL);
  gl.glDisable(GL11.GL_TEXTURE_2D);
  gl.glDisable(GL11.GL_BLEND);
```



```
  gl.glDepthMask(true);
}  // end of draw()
```

A particle's appearance is a mix of red and a transparent texture, which requires alpha blending. Unfortunately, alpha blending can cause errors at render time because it may modify the depth buffer incorrectly. (The depth buffer specifies the rendering order for objects relative to the current camera position.)

One solution is to sort the particles' buffer points before calling GL11.glDrawArrays() to guarantee that they'll be drawn in a particular order. That can be expensive because whenever the camera moves, a re-sort may be necessary.

A simpler solution is to call GL.glDepthMask(false), as in Particles.draw(). It prevents the points from changing the depth buffer, thereby avoiding potential rendering errors. For this approach to work, all the other objects in the scene must be rendered prior to the point sprites, to ensure the depth buffer contains all the necessary ordering details so the sprites can be drawn correctly. A drawback is that the points may not be drawn in the right order relative to each other, but this usually doesn't matter for the kind of effects that utilize particles (e.g. smoke, fire, clouds).

The required drawing order for the scene is implemented by drawSceneGL() in ParticlesGameCanvas, as shown in the code fragment:

```
// drawSceneGL() in ParticlesGameCanvas
floor.draw();
texCube.draw();
particles.draw();
```

The particles are drawn last, after the floor and the cube.

Texturing

An advantage of representing particles with point sprites is that texture coordinate generation can be handled by the system. The call in Partcles.draw():

```
gl.glTexEnvx(GL11.GL_POINT_SPRITE_OES,
                  GL11.GL_COORD_REPLACE_OES, GL11.GL_TRUE);
```

requests that the points' texture coordinates be generated automatically.

The setTexture() method specifies the texture to be used at rendering time, and is similar to the same-named function used in earlier examples:

```
// global
private int texNames[];   // for the texture name


private void setTexture()
// set the texture used by all the particles at render time
{
  gl.glBindTexture(GL11.GL_TEXTURE_2D, texNames[0]);
                                    // use the texture name
```



```
  // specify the RGBA texture for the currently bound tex name
  gl.glTexImage2D(GL11.GL_TEXTURE_2D, 0, GL11.GL_RGBA,
                  imWidth, imHeight, 0,
                  GL11.GL_RGBA, GL11.GL_UNSIGNED_BYTE, texBuf);

  // set the minification/magnification techniques
  gl.glTexParameterx(GL11.GL_TEXTURE_2D,
                     GL11.GL_TEXTURE_MIN_FILTER, GL11.GL_LINEAR);
  gl.glTexParameterx(GL11.GL_TEXTURE_2D,
                     GL11.GL_TEXTURE_MAG_FILTER, GL11.GL_LINEAR);
} // end of setTexture()
```

This method differs from earlier versions by employing the GL11.GL_RGBA texture format in GL11.glTexImage2D() (previously it utilized GL_RGB). The change ensures that the texture buffer's alpha byte will be correctly interpreted.

8. Specifying Point Sizes

The draw() method enables three buffers for GL11.glDrawArray(): posnsBuf holds the particles' coordinates, colsBuf has their colors, and sizesBuf contains the point sizes. In this way, every particle can have its own position, color, and size.

However, if the required particle system only utilizes same-sized point sprites then it's possible to do without a sizes buffer. Instead, a single point size can be specified with a call to GL11.glPointSize(). For example, the following call sets the size of all the sprites to be 15.0f:

```
gl.glPointSize(15.0f);
```

This operation should be located in Particles.update(), before the execution of gl.glDrawArray(). Buffers will now only be required for the particles' positions and colors; the point size buffer and arrays can be discarded.

9. Distance Attenuation and the Fade Threshold

Distance attenuation and a fade threshold for the point sprites can be set up with GL11.glPointParameterfv().

The method's GL11.GL_POINT_DISTANCE_ATTENUATION parameter specifies how the distance from the camera to the sprites affects their size. The attenuation equation, $1/ (a + b*d + c*d^2)$, utilizes three values (a, b, and c) supplied by the programmer, and the distance to the camera (d). For example, when (a, b, c) are (0.0, 1.0, 0.0), the equation becomes 1/d, also known as *linear attenuation*.

The equation can be set up in Particles.draw() with:

```
float[] atten = {0.0f, 1.0f, 0.0f};  // a, b, and c values
gl.glPointParameterfv(GL11.GL_POINT_DISTANCE_ATTENUATION, atten, 0);
```

Unfortunately, this code fragment doesn't work in WTK 2.5, which is why I implemented my own distance attenuation solution. My approach is more flexible

than the built-in mechanism since it's possible to influence the attenuation with other attributes, such as particle age.

A fade threshold for the sprites is set with the GL11.GL_POINT_FADE_THRESHOLD_SIZE parameter in GL11.glPointParameterfv(). The associated value is the point size at which sprites begin to fade. The following code sets the threshold size to be 5.0f:

```
gl.glPointParameterf(GL11.GL_POINT_FADE_THRESHOLD_SIZE, 5.0f);
```

Unfortunately, I couldn't get this to work in WTK 2.5 either, so I implemented my own fading code which reduces a particle's alpha value depending on its age.

10. Other Ways of Implementing Particle Systems

Broadly speaking there are three ways of implementing particle systems in OpenGL – as textured quads (or triangle strips), with point sprites, or by using vertex and fragment shaders. Shaders can be combined with quads, strips, or sprites.

The textured quad (or triangle strip) approach has been employed since the early days of OpenGL, and many examples can be found online and in textbooks. For example, lesson 19 at the Nehe OpenGL site (http://nehe.gamedev.net/) shows how to use triangle strips. A JOGL version of the tutorial is located at http://pepijn.fab4.be/?page_id=34.

Point sprites were first added to OpenGL as an extension, and promoted to a core feature in OpenGL 2.0. Most graphics cards now support them, perhaps with some restrictions (e.g. a limit on the number of point sizes).

The main advantage of point sprites is the reduction in the complexity of the particles' geometry (from four points down to one for each particle), which boosts rendering performance. On the downside is that it's harder to manipulate the point's texture (e.g. rotate it) and clipping can cause a large point to suddenly disappear even when much of it is still visible on screen (the clipping is determined by the location of the point's center).

The TyphoonLabs OpenGL ES tutorials (http://www.typhoonlabs.com) includes a point sprite example as Tutorial 6. It covers similar ground to my example, but implements slightly different particle properties, and doesn't vary the particles' sizes. It's written in C++, not JOGL-ES.

OpenGL allows points to be drawn inside a glBegin()/glEnd() block using the GL_POINT geometry (an approach *not* supported in OpenGL ES). An example by Kevin Harris can be found at http://www.codesampler.com/oglsrc/oglsrc_6.htm#ogl_particle_system.

As graphics cards have got more powerful, it's become possible to program them using vertex and fragment shaders which override their default vertex and fragment processing. Vertex shaders permit vertex attributes such as lighting, texturing, and normals to be controlled, while fragment shaders modify pixel-level elements such as color and texturing. Currently, shaders aren't a part of JOGL-ES, but will be in version 2.0.

It's possible to offload most of the particles' velocity, position, color and point size calculations to a vertex shader. The result is termed a *stateless* particle system since any varying data that can't be calculated at run time is stored outside the shader, not on the graphics card. The advantage of this approach is efficiency – systems of over 100,000 particles are possible. An example shader for a "confetti cannon" is given in Chapter 13 of *OpenGL Shading Language*, by Randi J. Rost, Addison-Wesley, 2nd ed. It combines its vertex shader (and a very simple fragment shader) with point sprites. The code can be found at http://developer.3dlabs.com/downloads/glslexamples/index.htm.

A *state-preserving* particle system utilizes a new shader approach, which stores data such as the particles' current positions and velocities as 2D textures. Particle information, encoded as textures, can be loaded into the vertex and fragment shaders *and* updated on the graphics card. This requires graphics card functionality which isn't widely available as yet, but the payoff is the ability to create systems with a million particles. "Building a Million Particle System" by Lutz Latta is an accessible paper on this new kind of particle system. It can be found at http://www.2ld.de/gdc2004/, along with example code that requires an NVIDIA GeforceFX 5xxx (or higher) graphics card.

Most OpenGL programmers don't create their particle systems from scratch, but instead use a particle systems API. A very popular one, developed by David K. McAllister, can be found at http://www.particleSystems.org/.

