Chapter 12. Particle Systems

Three particle systems are developed in this chapter: one where the particles are points, another using lines, and a third using quadrilaterals (quads). Figures 1, 2, and 3 show the three systems in action:

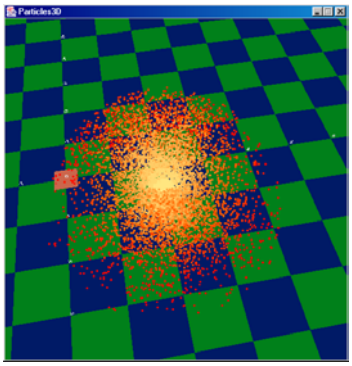

Figure 1. A Particle System of Points.

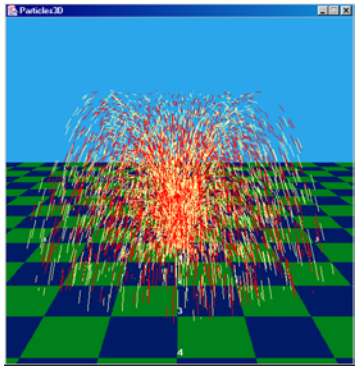

Figure 2. A Particle System of Lines.

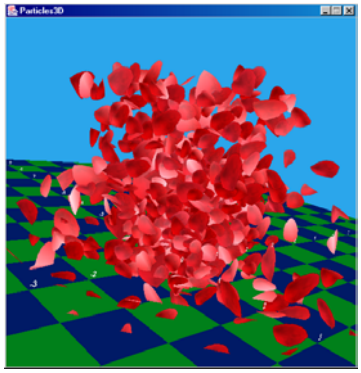

Figure 3. A Particle System of Quads.

The three systems are part of a single application, Particles3D. The code is located in /Code/Particles3D.

The coding illustrates the following techniques:

- the use of BY_REFERENCE geometries for holding information;
- the subclassing of the GeometryUpdater interface to manage shape updating;
- the use of PointArray, LineArray, and QuadArray geometries;
- the application of a single texture to multiple quads;
- the use of textures with transparent elements;
- the blending of texture, colour, and lighting;
- always rotating a shape towards the viewer with OrientedShape3D.

1. What are Particle Systems?

Particle Systems are an important component of many 3D games: when you see sparks flying, smoke swirling, fireworks exploding, snow falling, water shooting, or blood spurting, then it's probably being done with a particle system.

A particle system consists of a large population of individual particles, perhaps tens or hundreds of thousands (although many commercial games use far fewer depending on the effect required). The particle system is in charge of creating and deleting particles, and updating their attributes over time.

A particle is typically rendered as a graphics primitive, such as a point or line. This means that rendering overheads can be reduced, an important consideration when so many particles are involved. With the advent of more powerful graphics cards, simple polygons (e.g. triangles, quads) have also been used, which allows textures and lighting to be introduced.

Particle attributes can be very varied, but typically include position, velocity, forces (e.g. gravity), age, colour/texture, shape, size, and transparency. Attribute updates tend to use time-based equations, but other approaches are possible. For instance, a particle's new position may be a random adjustment of its previous position.

Particle systems often have a generation shape, which specifies a bounding volume in which particles can be created. Generation shapes have been extended to specify bounding volumes for particle updating and aging. For example, if a particle moves outside the space then it will begin to age, and age more quickly as it moves further away.

A central practical issue with particle systems is efficiency, since a system may be made up of so many particles. Efficiency influences how particles are updated, rendered, and reused (i.e. a 'dead' particle may simply be reset to its initial attribute settings, and started again). If particles use texturing, then the texture should be applied to the entire system.

A particle is a passive entity which is acted upon by the particle system by adjusting its attributes. An off-shoot of particle systems are *flocking boids*, which we consider in chapter 13. A boid is more 'intelligent' in the sense that it has its own internal state and behaviour. For instance, a boid may examine the attributes of its nearest neighbours in the flock in order to adjust its velocity to avoid colliding with them.

2. Particle Systems in Java 3D

There are several ways of implementing a particle system in Java 3D, but concerns about efficiency mean that many of them are impractical for large systems.

We represent particles as elements of a GeometryArray. The example in this chapter displays three different types of systems, but they all use subclasses of GeometryArray: a PointArray is employed to store particles as points, a LineArray for particles which are lines, and a QuadArray for quads.

These GeometryArrays are not utilised in the usual manner. The standard approach applies changes to a GeometryArray by *copying* in new coordinates, colours, etc. The problem is that the rapidly changing nature of a particle system, and its size, will necessitate very large amounts of copying, which is just too slow. Instead, the GeometryArray is created with a BY_REFERENCE flag:

```
// a BY_REFERENCE PointArray with numPoints points
PointArray pointParts = new PointArray(numPoints,
            PointArray.COORDINATES | PointArray.COLOR_3 |
            PointArray.BY_REFERENCE );

// allow things to be read and written by reference
pointParts.setCapability(GeometryArray.ALLOW_REF_DATA_READ);
pointParts.setCapability(GeometryArray.ALLOW_REF_DATA_WRITE);
```

This signals that the data which the PointArray manages is not copied into it, instead the PointArray references data structures stored in the user's execution space. pointParts will refer to two data structures: one maintaining the coordinates of the PointArray, the other the colours of those coordinates.

The next step is to create the local data structures which pointParts will utilise. Java 3D v1.3. only encourages references to float arrays.

```
private float[] cs, cols;
  :
cs = new float[numPoints*3];   // to store each (x,y,z) coord
cols = new float[numPoints*3];
// fill in the arrays with coordinates and colours
   :
// store coordinates and colours array refs in PointArray
pointParts.setCoordRefFloat(cs);    // use BY_REFERENCE
pointParts.setColorRefFloat(cols);
```

The restriction to float arrays means that the coordinates must be stored as individual x, y, and z values, which requires a numPoints*3 size array. Similarly, the red-green-blue components of each color must be stored separately.

Once the arrays have been filled, the references are set up with calls to setCoordRefFloat() and setColorRefFloat(). From now on, the program need only change the cs and cols arrays to change the PointArray. There is no need to copy the changes into the PointArray.

pointParts is made the geometry of a scene graph node, such as a Shape3D, with:

```
setGeometry(pointParts);
```

Now Java 3D will render the shape using the data in PointArray, and update the shape when the referenced float arrays are modified.

Referring to Float Arrays

The Java 3D distribution comes with many demo examples, which are copied to the subdirectory <JAVAHOME>/demo/java3d when Java 3D is installed. There are several examples using BY_REFERENCE geometry, the most relevant one for us being the GeometryReferenceByTest application.

A study of the code will reveal that it appears possible to set up references to Point3f and Color3f arrays with the methods:

```
setCoordRef3f();
setColorRef3f();
```

However, these methods, and similar ones for textures and normals, are deprecated in Java v.1.3. Unfortunately, the demo has not been updated. The reason for the deprecation is to reduce the work required of Java 3D to maintain the references.

Synchronization Problems

Another issue is *when* the program should update the float arrays. The wrong answer is 'whenever it wants', because this may lead to synchronization problems. Java 3D will periodically access the arrays to use their information for shape rendering, and problems may occur if this examination is intertwined with the arrays being changed. The nasty aspect of synchronization bugs are their time-dependency, which makes them hard to detect during testing.

Synchronization worries are avoided by using the GeometryUpdater interface to update the geometry:

```
public class PointsUpdater implements GeometryUpdater
{         :
  public void updateData(Geometry geo)
  {  PointArray pa = (PointArray) geo;
     float[] cs = pa.getCoordRefFloat();    // use BY_REFERENCE
     float[] cols = pa.getColorRefFloat();
      // update the cs and cols float arrays
  }
}
```

Java 3D passes a GeometryArray reference to the updateDate() method when it's safe to carry out changes. The reference must be cast to the right type, and then the getCoordRefFloat() and getColorRefFloat() methods are used to return references to the required float arrays. The arrays can be safely modified, and the changes will be seen by Java 3D when it next renders the shape.

A GeometryUpdater object is set up like so:

```
PointsUpdater updater = new PointsUpdater(...);
       :
pointParts.updateData(updater);  // request update of geometry
```

The call to updateData() in the GeometryArray is *not* the GeometryUpdater method. It is a method with the same name, which requests that Java 3D carry out an update. The method's single argument is the GeometryUpdater object, which Java 3D will execute when an update can be safely performed.

The Inner Class Coding Style

A particle system will consist of three classes:

- The Particle System class which holds the BY_REFERENCE geometry (e.g. a PointArray, LineArray, or QuadArray), the float arrays holding the referenced coordinates, colours, and so on. The class will also contain the attribute initialisation code.
- A GeometryUpdater implementation which will carry out an update of the particle system by changing various attributes in the particles. In implementation terms, this means accessing and changing the particle system's float arrays.
- A Behavior class which will be triggered periodically, and then call the geometry's updateData() method, thereby requesting an update.

The required functionality shows a need for a lot of shared data between the classes. Consequently, our particle systems will use inner classes to implement the Behaviour and GeometryUpdater classes, both located inside the particle systems class. An added benefit of this strategy is that the inner classes will be hidden from the user of the particle system.

The coding style is illustrated in Figure 4, which shows a simplified UML diagram for the PointParticles class, which manages the particle system made of points.

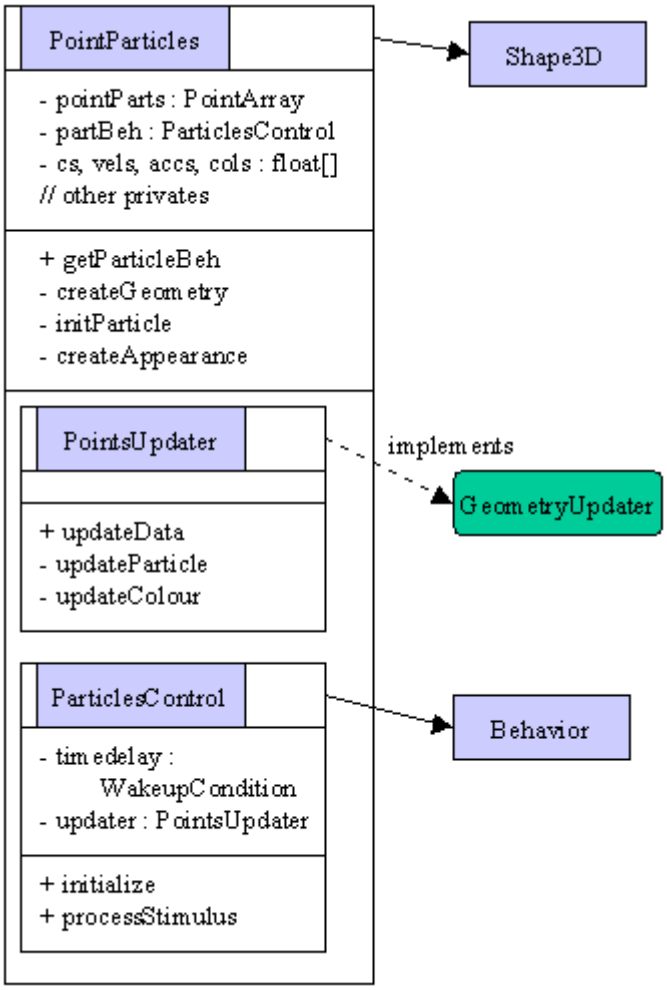

Figure 4. Point Particle Systems Class Structure.

The details of these classes will be explained in later sections.

The other two particle systems in the Particles3D application have the same basic structure.

3. UML Diagrams for Particles3D

Figure 5 shows the UML diagrams for the classes in the Particles3D program. Only the class names are shown.

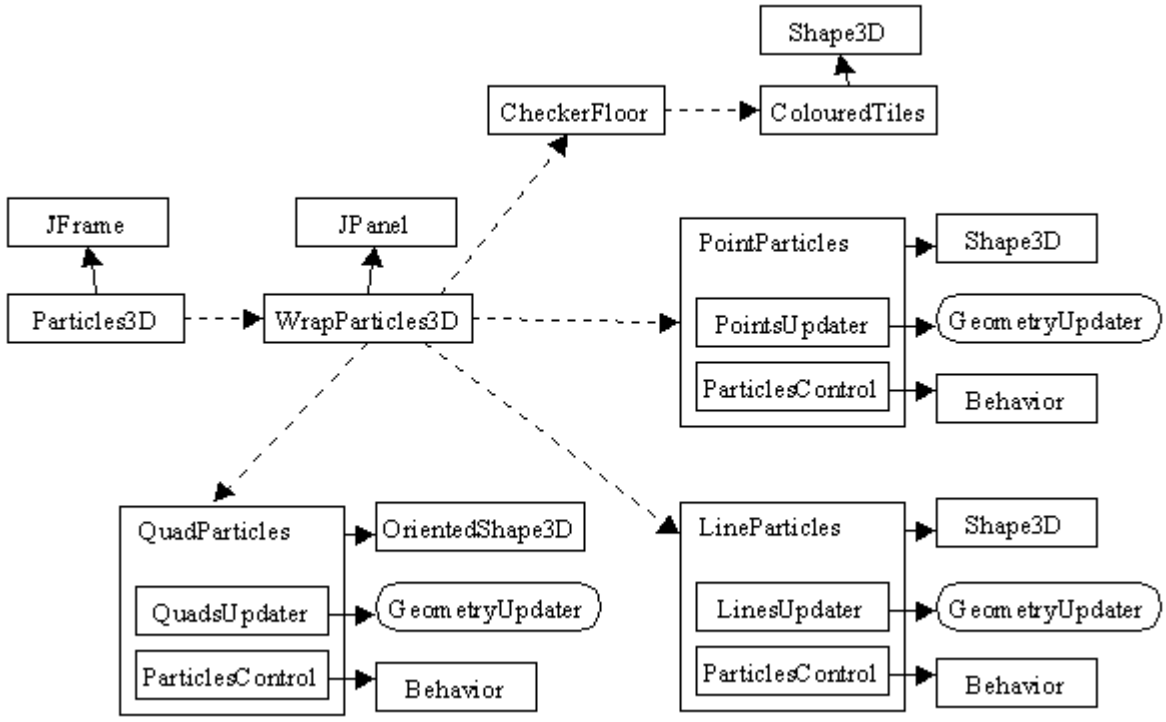

Figure 5. The Particles3D Classes.

Particles3D is the top-level JFrame for the application.

WrapParticles3D creates the 3D world, and is similar to the earlier 'Wrap' classes in that it creates the checkered floor, and sets up lighting. WrapParticles3D invokes one of the three particle systems (PointParticles, LineParticles, or QuadParticles) depending on user input.

The PointParticles and LineParticles particle systems are subclasses of Shape3D, which allows them to be added to the 3D scene easily. QuadParticles is a subclass of OrientedShape3D, to allow its on-screen representation to always rotate towards the viewer.

The UML for the particle system classes show that they all use the inner classes approach, with a GeometryUpdater and Behaviour.

CheckerFloor and ColouredTiles are the same classes as in previous examples.

The WrapParticles3D Class

The WrapParticles3D object is passed two integers from the command line: the number of points to be used when creating a particle system, and an integer between 1 and 3 which selects a particular system. The selection is done inside the createSceneGraph() method:


```
switch(fountainChoice) {
      case 1: addPointsFountain(numParts); break;
      case 2: addLinesFountain(numParts); break;
      case 3: addQuadFountain(numParts); break;
      default: break;   // do nothing
    }
```

The three particle systems all render variants of a fountain, which explains the prevalence of the word "fountain".

The three 'addFountain' methods are very similar, with addPointsFountain() the longest:

```
  private void addPointsFountain(int numParts)
  {
    PointParticles ptsFountain = new PointParticles(numParts, 20);
    // 20ms time delay between updates

    // move particles start position to (2,0,1)
    TransformGroup posnTG = new TransformGroup();
    Transform3D trans = new Transform3D();
    trans.setTranslation( new Vector3d(2.0f, 0.0f, 1.0f) );
    posnTG.setTransform(trans);
    posnTG.addChild(ptsFountain);
    sceneBG.addChild( posnTG );

    // timed behaviour to animate the fountain
    Behavior partBeh = ptsFountain.getParticleBeh();
    partBeh.setSchedulingBounds( bounds );
    sceneBG.addChild(partBeh);
  }
```

The particle system (together with its GeometryUpdater and Behavior objects) is created by the PointParticles() constructor which supplies the number of points to use, and the time delay between each update.

The middle part of the addPointFountain() shows that it is straight forward to move the system, as a single Shape3D entity, to a new position. By default, the systems all start at the origin.

Although the Behavior object is created inside PointParticles, it still needs to be attached to the scene graph and given a bounding volume. This is done in the last part of addPointFountain(), and requires a public getParticleBeh() method to return a reference to the behaviour.

4. The PointsParticles Class

PointsParticles creates a fountain of points, which are coloured yellow initially, and then gradually turn red. The particles are emitted from the origin, and travel in parabolas of various velocities in any direction across the XZ plane, and upwards along the y-axis. The only force applied to the particles is gravity (we assume that a particle has unit mass, so F=A). When a particle drops below the XZ plane it is reused by resetting its attributes to their initial settings.

A particle has four attributes:

- its (x,y,z) location;

- its velocity (expressed in x-, y-, z- directional components);
- its acceleration (also expressed as three components);
- its colour (as three floats for its Red-Green-Blue parts).

The UML diagram for PointParticles is given in Figure 4. The attributes are represented by the float arrays cs, vels, accs, and cols. If the user starts the PointParticles system with numPoints particles, then these arrays will sized at numPoints*3 to accommodate all the necessary data.

The PointParticles() constructor initialises the PointArray, as already outlined:

```
PointArray pointParts = new PointArray(numPoints,
            PointArray.COORDINATES | PointArray.COLOR_3 |
            PointArray.BY_REFERENCE );

// allow things to be read and written by reference
pointParts.setCapability(GeometryArray.ALLOW_REF_DATA_READ);
pointParts.setCapability(GeometryArray.ALLOW_REF_DATA_WRITE);
```

The constructor also creates the GeometryUpdater and Behaviour objects:

```
PointsUpdater updater = new PointsUpdater();
partBeh = new PartclesControl(delay, updater);
```

partBeh is a global so that it can be returned by getParticleBeh():

```
public Behavior getParticleBeh()
{ return partBeh;  }
```

The constructor calls createGeometry() to initialise the Shape3D's geometry, and createAppearance() for its appearance.

The Particle System's Geometry and Appearance

createGeometry() declares the float arrays, initialises them, then sets up references to the coordinate and colour float arrays for the PointArray:

```
private void createGeometry()
{ cs = new float[numPoints*3];   // to store each (x,y,z)
  vels = new float[numPoints*3];
  accs = new float[numPoints*3];
  cols = new float[numPoints*3];

  // step in 3's == one (x,y,z) coord
  for(int i=0; i < numPoints*3; i=i+3)
    initParticle(i);

  // store the coordinates and colours in the PointArray
  pointParts.setCoordRefFloat(cs);    // use BY_REFERENCE
  pointParts.setColorRefFloat(cols);

  setGeometry(pointParts);
}
```


pointParts is only set to refer to the cs and cols arrays since these contain the position and colour data required for each point. GeometryArrays may also be assigned normals and texture coordinates, as will be seen in the QuadParticles class.

initParticles() is called in steps of three since each iteration is initialising one point, which is equvalent to three values in the float arrays.

```
private void initParticle(int i)
{ cs[i] = 0.0f; cs[i+1] = 0.0f; cs[i+2] = 0.0f;
      // (x,y,z) at origin
  // random velocity in XZ plane with combined vector XZ_VELOCITY
  double xvel = Math.random()*XZ_VELOCITY;
  double zvel = Math.sqrt((XZ_VELOCITY*XZ_VELOCITY) - (xvel*xvel));
  vels[i] = (float)((Math.random()<0.5) ? -xvel : xvel);  // x vel
  vels[i+2] = (float)((Math.random()<0.5) ? -zvel : zvel);// z vel
  vels[i+1] = (float)(Math.random() * Y_VELOCITY);  // y vel

  // unchanging accelerations, downwards in y direction
  accs[i] = 0.0f; accs[i+1] = -GRAVITY; accs[i+2] = 0.0f;

  // initial particle colour is yellow
  cols[i] = yellow.x;  cols[i+1] = yellow.y; cols[i+2] = yellow.z;
}
```

The method initialises the cs[], vels[], accs[], and cols[] arrays.

The x-axis velocity is randomly set between -XZ_VELOCITY and XZ_VELOCITY, and the z-axis velocity is assigned the value that makes the magnitude of the combined XZ vector equal XZ_VELOCITY. This means that particles can travel in any direction across the XZ plane, but they all have the same speed.

The only acceleration is a constant – gravity down the y-axis. By including accelerations in the x- and z- directions, forces such as air resistance could be simulated.

createAppearance() increases the point size of the particles:

```
private void createAppearance()
{ Appearance app = new Appearance();
  PointAttributes pa = new PointAttributes();
  pa.setPointSize( POINTSIZE );  // may cause bugs
  app.setPointAttributes(pa);
  setAppearance(app);
}
```

Point size adjustment, point anti-aliasing, line size adjustment, and line anti-aliasing are poorly supported in Java 3D because of weaknesses in the underlying graphics libraries and/or drivers. Currently, OpenGL and OpenGL-compatible graphics cards cope pretty well, but DirectX-based system often crash.

PointsUpdater

The PointsUpdater class utilises updateData() in a different way than outlined earlier.

```
  public void updateData(Geometry geo)
  { // GeometryArray ga = (GeometryArray) geo;
    // float cds[] = ga.getCoordRefFloat();
```



```
    // step in 3's == one (x,y,z) coord
    for(int i=0; i < numPoints*3; i=i+3) {
      if (cs[i+1] < 0.0f)    // particle dropped below y-axis
        initParticle(i);     // re-initialise it
      else       // update the particle
        updateParticle(i);
    }
  }  // end of updateData()
```

The commented out lines indicate that no use is made of the Geometry input argument. Instead the float arrays (cs[], vels[], accs[], and cols[]), which are global, are accessed directly.

It should be remembered that updateData()'s primary purpose is to be called by Java 3D *when* it is safe to modify the arrays. It does not matter where the array references come from.

updateData() implements particle reuse by detecting when a particle has dropped below the y-axis, and then re-initializing it by calling PointParticles' initParticle() method. This shows the advantage of using an inner class and global float arrays.

Updating Particles

Underpinning the motion of the particles is Newton's second law, F = ma, which reduces to F = a by assuming unit mass. In other words, the particles move in the presence of constant acceleration, which is gravity for our examples.

It is possible to obtain velocity and distance equations from this basic assumption by using Euler's integration algorithm.

The acceleration equation can be written as:

$$d\,vel\ /\ dt\ =\ a$$

or $$d\,vel = a\ dt$$

Using Euler's method, we obtain the velocity equation:

$$vel(t+dt) = vel(t) + a\ dt \qquad \text{// equation 1}$$

Integrating again:

$$dist(t+dt) = dist(t) + vel(t)\ dt + 1/2\ a\ dt^2 \qquad \text{// equation 2}$$

The equations can be separated into their x-, y-, and z- components. For example:

$$vel_x(t+dt) = vel_x(t) + a_x\ dt$$

$$dist_x(t+dt) = dist_x(t) + vel_x(t)\ dt + 1/2\ a_x\ dt^2$$

These equations are embedded in the updateParticle() method, where ($dist_x$, $dist_y$, $dist_z$) are cs[i] – cs[i+2] and (vel_x, vel_y, vel_z) are vels[i] – vels[i+2].

```
private void updateParticle(int i)
{ cs[i] += vels[i] * TIMESTEP +
            0.5 * accs[i] * TIMESTEP * TIMESTEP;     // x coord
  cs[i+1] += vels[i+1] * TIMESTEP +
             0.5 * accs[i+1] * TIMESTEP * TIMESTEP;  // y coord
  cs[i+2] += vels[i+2] * TIMESTEP +
             0.5 * accs[i+2] * TIMESTEP * TIMESTEP;  // z coord
```



```
    vels[i] += accs[i] * TIMESTEP;      // x vel
    vels[i+1] += accs[i+1] * TIMESTEP;  // y vel
    vels[i+2] += accs[i+2] * TIMESTEP;  // z vel

    updateColour(i);
  } // end of updateParticle()
```

The small time step, dt, has been fixed as the constant TIMESTEP (0.05f).

updateColor() reduces the green and blue components of a point's colour. Over time, these will drop to 0, leaving only red.

```
  private void updateColour(int i)
  { cols[i+1] = cols[i+1] - FADE_INCR;   // green part
    if (cols[i+1] < 0.0f)
      cols[i+1] = 0.0f;
    cols[i+2] = cols[i+2] - FADE_INCR;   // blue part
    if (cols[i+2] < 0.0f)
      cols[i+2] = 0.0f;
  }
```

Triggering an Update

The ParticlesControl behavior requests an update to the PointArray every few milliseconds.

```
public class PartclesControl extends Behavior
{ private WakeupCondition timedelay;
  private PointsUpdater updater;

  public PartclesControl(int delay, PointsUpdater updt)
  {  timedelay = new WakeupOnElapsedTime(delay);
     updater = updt;
  }

  public void initialize( )
  { wakeupOn( timedelay );  }

  public void processStimulus(Enumeration criteria)
  { pointParts.updateData(updater);  // request update of geometry
    wakeupOn( timedelay );
  }
}
```

This behaviour is virtually the same in each of the particle system classes: only the types of the GeometryArray and GeometryUpdater arguments change.

5. The LineParticles Class

The LineParticles class implements a particle system made up of yellow and red lines, which shoot out from the origin with parabolic trajectories. The effect, as seen in figure 2, is something like a firework. The thickness of the lines is increased slightly, and anti-aliasing is switched on. When a line has completely dropped below the y-axis, it is re-initialised, which means the firework never runs out!

The main difference from PointParticles is the extra code required to initialise the lines and update them inside the float arrays. Six values in a float array are necessary to represent a single line: three values for each of the end-points.

The LineParticles constructor creates a LineArray object using BY_REFERENCE geometry for its coordinates and colour:

```
lineParts = new LineArray(numPoints,  LineArray.COORDINATES |
        LineArray.COLOR_3 | LineArray.BY_REFERENCE );

lineParts.setCapability(GeometryArray.ALLOW_REF_DATA_READ);
lineParts.setCapability(GeometryArray.ALLOW_REF_DATA_WRITE);
```

Initializing the Particles

The changes start with the initialisation of the float arrays inside createGeometry():

```
// step in 6's == two (x,y,z) coords == one line
for(int i=0; i < numPoints*3; i=i+6)
  initTwoParticles(i);
```

initTwoParticles() initialises 6 floats (two point; one line). It assigns the same position and velocity to both points, and then calls updateParticle() to update one of the point's position and velocity. This specifies a line with a point which is one update ahead of the other point. This means that, as the particle system is updated, each line will follow a smooth path since one point is following the other.

The colour of the line is set to red or yellow by fixing the colours of both points.

```
private void initTwoParticles(int i)
{ cs[i] = 0.0f; cs[i+1] = 0.0f; cs[i+2] = 0.0f;   // origin

  // random velocity in XZ plane with combined vector XZ_VELOCITY
  double xvel = Math.random()*XZ_VELOCITY;
  double zvel = Math.sqrt((XZ_VELOCITY*XZ_VELOCITY) - (xvel*xvel));
  vels[i] = (float)((Math.random()<0.5) ? -xvel : xvel);   // x vel
  vels[i+2] = (float)((Math.random()<0.5) ? -zvel : zvel); // z vel
  vels[i+1] = (float)(Math.random() * Y_VELOCITY); // y vel

  // unchanging accelerations, downwards in y direction
  accs[i] = 0.0f; accs[i+1] = -GRAVITY; accs[i+2] = 0.0f;

  // next particle starts the same, but is one update advanced
  cs[i+3] = cs[i]; cs[i+4] = cs[i+1]; cs[i+5] = cs[i+2];
  vels[i+3] =vels[i]; vels[i+4] = vels[i+1]; vels[i+5] = vels[i+2];
  accs[i+3] =accs[i]; accs[i+4] = accs[i+1]; accs[i+5] = accs[i+2];
  updateParticle(i+3);

  // set initial colours for the first particle
  Color3f col = (Math.random() < 0.5) ? yellow : red;
  cols[i] = col.x;   cols[i+1] = col.y; cols[i+2] = col.z;
  // the next particle has the same colours
  cols[i+3] = col.x; cols[i+4] = col.y; cols[i+5] = col.z;
}
```

initTwoParticles() is similar to the initParticles() method in PointParticles because they both set up a parabolic trajectory for their particles.

The updateParticles() method is the same as the one in PointParticles, but is now located in the particle systems class (LineParticles) rather than in GeometryUpdater.

Particle Appearance

createAppearance() adjusts the line width, and switches on anti-aliasing. As mentioned previously, these features may cause DirectX-based system, or machines with old graphics cards, to respond strangely or even crash.

```
private void createAppearance()
{ Appearance app = new Appearance();
  LineAttributes la = new LineAttributes();
  la.setLineWidth( LINEWIDTH );    // may cause bugs
  la.setLineAntialiasingEnable(true);
  app.setLineAttributes(la);
  setAppearance(app);
}
```

Updating the Particle System

LinesUpdater implements the GeometryUpdater interface, and specifies updateData(). The method ignores the Geometry argument, instead using the global float arrays directly. It makes use of the initTwoParticles() and updateParticle() methods in LineParticles.

```
public void updateData(Geometry geo)
{ // step in 6's == two (x,y,z) coords == one line
  for(int i=0; i < numPoints*3; i=i+6) {
    if ((cs[i+1] < 0.0f) && (cs[i+4] < 0.0f))
    // both particles in the line have dropped below the y-axis
      initTwoParticles(i);   // re-initialise them
    else {       // update the two particles
      updateParticle(i);
      updateParticle(i+3);
    }
  }
}
```

6. The QuadParticles Class

Although many particle systems can be modeled with points and lines, moving to quadrilaterals (quads) combined with textures allows many more interesting effects.

The texture can contain extra surface detail, and can be partially transparent in order to break up the regularity of the quad shape.

A quad can be assigned a normal and a Material node component to allow it to be affected by lighting in the scene.

The only danger with these additional features is that they may slow down rendering by too much. For example, we want to map the texture to each quad (each particle), but do not want to use more than one QuadArray and one Texture2D object.

The effect we are after is suitably gory: a fountain of blood corpuscles gushing up from the origin. Each 'blood' particle should be roughly spherical, and should oscillate slightly as it travels through the air.

Figure 3 shows the QuadParticles system in action. If the user viewpoint is rotated around the fountain, the particles seem to be rounded on all sides (see Figure 6).

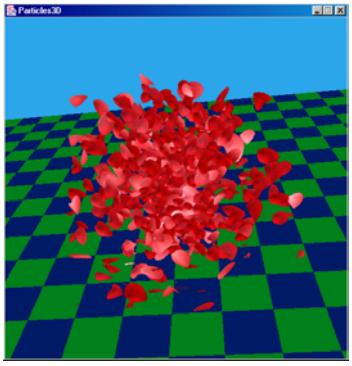

Figure 6. A Fountain of Blood from the Back.

It would be a mistake to try to represent the particles using GeometryArray meshes: the number of vertices required for a reasonable blood cell would severely restrict the total number of particles that could be created. Instead the effect is achieved by trickery: the particle system is placed inside an OrientedShape3D node rather than a Shape3D.

OrientedShape3D nodes automatically face towards the viewer, and can be set to rotate about a particular point or axis. Since the particle system is rooted at the origin, it makes most sense to rotate the system about the y-axis.

QuadParticles is made a subclass of OrientedShape3D, rather than Shape3D, and its constructor specifies the rotation axis:

```
// rotate about the y-axis to follow the viewer
setAlignmentAxis( 0.0f, 1.0f, 0.0f);
```

This means that the illusion of blood globules starts to breaks down if the viewpoint is looking down towards the XZ plane, as seen in Figure 7.

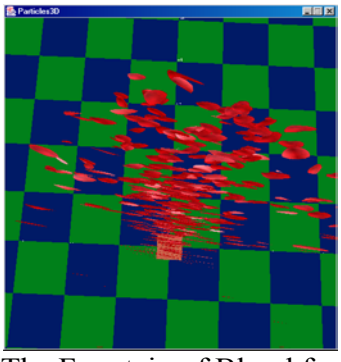

Figure 7. The Fountain of Blood from Above.

Specifying the Geometry

The QuadArray requires information about coordinates, textures, and normals:


```
// BY_REFERENCE QuadArray
quadParts = new QuadArray(numPoints,
              GeometryArray.COORDINATES |
              GeometryArray.TEXTURE_COORDINATE_2 |
              GeometryArray.NORMALS  |
              GeometryArray.BY_REFERENCE );

// the referenced data can be read and written
quadParts.setCapability(GeometryArray.ALLOW_REF_DATA_READ);
quadParts.setCapability(GeometryArray.ALLOW_REF_DATA_WRITE);
```

The use of BY_REFERENCE means that there must be float arrays for the coordinates, velocities, and accelerations (as before), and for normals and texture coordinates:

```
private float[] cs, vels, accs, norms;
private float[] tcoords;
    :
cs = new float[numPoints*3];   // to store each (x,y,z)
vels = new float[numPoints*3];
accs = new float[numPoints*3];
norms = new float[numPoints*3];
tcoords = new float[numPoints*2];
```

Each vertex in the QuadArray (there are numPoints of them) requires a texture coordinate (s,t), where s and t have float values in the range 0 to 1 (see chapter 9 for the first use of Texture2D).

As of Java 3D 1.3, the use of a TexCoord2f array to store the texture coordinates of a BY_REFERENCE geometry is no longer encouraged; a float array should be employed. So instead of storing numPoints TexCoord2f objects, numPoints*2 floats are added to the tcoords[] array.

Initialising Particle Movement

A particle is a single quad, which is the same as 4 vertices or 12 floats. The consequence is that much of the particle initialisation and updating code utilises loops which make steps of 12 through the float arrays: the creation/updating of a single quad involves 12 floats at a time.

createGeometry() calls initQuadParticle() to initialise each quad:

```
for(int i=0; i < numPoints*3; i=i+12)
  initQuadParticle(i);
// refer to the coordinates in the QuadArray
quadParts.setCoordRefFloat(cs);
```

initQuadParticles() is similar in style to the initialisation methods in PointParticles and LineParticles. The idea is to use the same initial velocity and acceleration for all the vertices, to make the parts of the quad move in the same manner.

The position of a quad is determined by setting its four points to have the values stored in the globals p1, p2, p3, p4:

```
private float[] p1 = {-QUAD_LEN/2, 0.0f, 0.0f};
private float[] p2 = {QUAD_LEN/2, 0.0f, 0.0f};
private float[] p3 = {QUAD_LEN/2, QUAD_LEN, 0.0f};
```



```
private float[] p4 = {-QUAD_LEN/2, QUAD_LEN, 0.0f};
```

The order of these starting points is important: they specify the quad in a counter-clockwise order starting from the bottom-left point. The points together define a quad of sides QUAD_LEN which is facing along the positive z-axis, resting on the XZ plane, and centered at the origin. This is shown graphically in Figure 8.

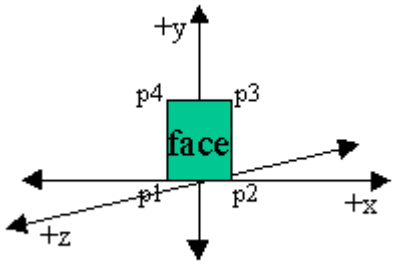

Figure 8. Initial Quad Position.

The QuadArray will contain quads which all begin at this position and orientation.

Initialising Particle Texture Coordinates

The aim is to add a texture to each particle (each quad) in the QuadArray, and to use only one Texture2D object. This is possible by mapping the four vertices of each quad to the same (s,t) texels. Later when the Texture2D is set in the Appearance node component it will be applied to all the quads in the QuadArray individually. The mapping of quad coords to texels is represented in figure 9.

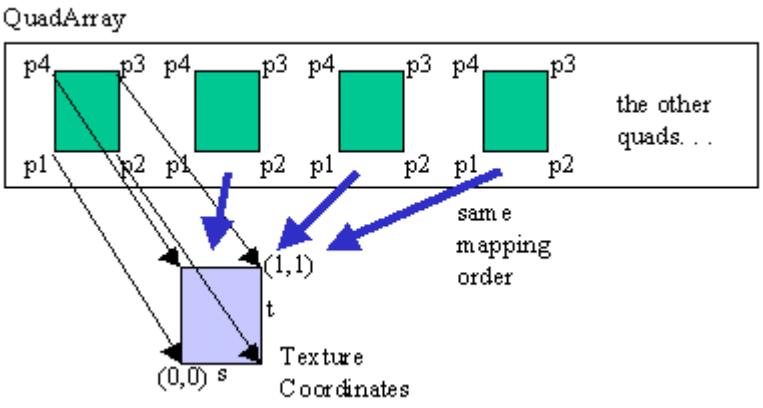

Figure 9. Mapping Quads to the same Texels.

The texels should be specified in an anti-clockwise order starting at the bottom left, so they have the same order as the quad vertices, otherwise the texture image will appear distorted.

The mapping is done in createGeometry():

```
for(int i=0; i < numPoints*2; i=i+8) {
   tcoords[i]   = 0.0f; tcoords[i+1] = 0.0f;  // for 1 vertex
```



```
      tcoords[i+2] = 1.0f; tcoords[i+3] = 0.0f;
      tcoords[i+4] = 1.0f; tcoords[i+5] = 1.0f;
      tcoords[i+6] = 0.0f; tcoords[i+7] = 1.0f;
    }
  quadParts.setTexCoordRefFloat(0, tcoords);   // use BY_REFERENCE
```

tcoords is filled with a repeating sequence of (s,t) texels (in float format). When this is applied to the QuadArray, Java 3D will map the vertices to the texels.

The code is a little difficult to understand because the cs[] array (holding the vertices) is bigger than tcoords[] (holding the texels). The reason for the size difference is that a vertex is stored as three floats, while a texel only needs two.

Initialising Particle Normals

For lighting and Material node components to work, normals must be set for the points of each quad.

Our desired effect is a fountain of globules pumping out in different directions, which can be enhanced by assigning random normals to the quads. This causes each one to reflect light in a different way, giving the impression of surface variations. In fact, the normals will not be completely random: the z-direction should be positive so that light is bounced back towards the viewer, but the z- and y- components can be positive or negative. A further restriction is that the normal vector must be unit length (i.e. be normalized).

The code in createGeometry() which does this:

```
Vector3f norm = new Vector3f();
for(int i=0; i < numPoints*3; i=i+3) {
  randomNormal(norm);
  norms[i]=norm.x; norms[i+1]=norm.y; norms[i+2]=norm.z;
}
quadParts.setNormalRefFloat(norms);   // use BY_REFERENCE
```

and

```
private void randomNormal(Vector3f v)
{ float z = (float) Math.random();     // between 0-1
  float x = (float)(Math.random()*2.0 - 1.0);   // -1 to 1
  float y = (float)(Math.random()*2.0 - 1.0);   // -1 to 1
  v.set(x,y,z);
  v.normalize();
}
```

Particle Appearance

createAppearance() carries out four tasks:

- it switches on transparency blending so that the transparent parts of a texture will be invisible inside Java 3D;
- it turns on texture modulation so that the texture and material colours will be displayed together;
- it loads the texture;

- it sets the material to a bloody red.

The code:

```
private void createAppearance()
{ Appearance app = new Appearance();

  // blended transparency so texture can be irregular
  TransparencyAttributes tra = new TransparencyAttributes();
  tra.setTransparencyMode( TransparencyAttributes.BLENDED );
  app.setTransparencyAttributes( tra );

  // mix the texture and the material colour
  TextureAttributes ta = new TextureAttributes();
  ta.setTextureMode(TextureAttributes.MODULATE);
  app.setTextureAttributes(ta);

  // load and set the texture
  System.out.println("Loading textures from " + TEX_FNM);
  TextureLoader loader = new TextureLoader(TEX_FNM, null);
  Texture2D texture = (Texture2D) loader.getTexture();
  app.setTexture(texture);

  // set the material: bloody
  Material mat = new Material(darkRed, black, red, white, 20.f);
  mat.setLightingEnable(true);
  app.setMaterial(mat);

  setAppearance(app);
}
```

TEX_FNM is the file smoke.gif, which is shown in Figure 10. Its background is transparent (i.e. its alpha is 0.0).

Figure 10. The smoke.gif Texture.

createAppearance() does not switch off polygon culling, which means that the back face of each quad will be invisible (the default action). Will this spoil the illusion when the viewer moves round the back of the particle system? No, because the system is inside an OrientedShape3D node, and so will rotate to keep its front pointing at the user.

Permitting back face culling also improves overall speed since Java 3D doesn't need to render half of the quad.

Updating the Particles

QuadsUpdater is similar to previous GeometryUpdater implementations, but works on groups of four points or 12 floats at a time. When a quad has completely dropped below the XZ plane then it is re-initialised: the blood never stops flowing!

updateDate()'s code:

```
public void updateData(Geometry geo)
{  // step in 12's == 4 (x,y,z) coords == one quad
  for(int i=0; i < numPoints*3; i=i+12)
     updateQuadParticle(i);
}

private void updateQuadParticle(int i)
{ if ((cs[i+1] < 0.0f) && (cs[i+4] < 0.0f) &&
      (cs[i+7] < 0.0f) && (cs[i+10] < 0.0f))
    // all of the quad has dropped below the y-axis
    initQuadParticle(i);
  else {
    updateParticle(i);  // all points in a quad change the same
    updateParticle(i+3);
    updateParticle(i+6);
    updateParticle(i+9);
  }
}
```

updateParticle() uses the same position and velocity equations as for the point and line-based particle systems, but with a slight modification to the position calculation. After the new coordinates for a point have been stored in cs[i], cs[i+1], and cs[i+2], they are adjusted slightly:

```
cs[i] = perturbate(cs[i], DELTA);
cs[i+1] = perturbate(cs[i+1], DELTA);
cs[i+1] = perturbate(cs[i+1], DELTA);
```

perturbate() adds a random number in the range -DELTA to DELTA to the coordinate (DELTA is set to be 0.05f).

This has the effect of slightly distorting the quad as it moves through the scene, which causes the texture to twist as well.

Figure 11 shows the particle system with the perturbate() calls commented out: each quad looks like a perfect sphere, which is unconvincing.

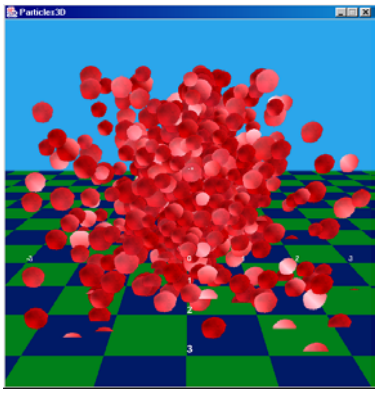

Figure 11. Particles without Perturbation.

7. Performance Results

Many of the coding techniques used in the Particles3D examples have been aimed at increasing performance. The obvious question is how fast are these particle systems? Table 1 gives average frames/sec (FPS) for the PointParticles, LineParticles, and QuadParticles classes, initialised with increasing numbers of points.

No. of Points	1,000	5,000	10,000	20,000	50,000	100,000
Point Particles	19 (50)	19 (35)	19 (24)	12 (14)	7 (7)	4 (4)
Line Particles	19 (50)	19 (45)	19 (35)	19 (24)	11 (11)	6 (6)
Quad Particles	19 (44)	14 (17)	9 (9)	6 (6)	2 (2)	1 (1)

Table 1. Average FPS for the Particle System with Varying Numbers of Points.

The averages were for five or more runs of the application. The hardware used was an Pentium IV 1800 MHz, 256 MB RAM, and a NVIDIA RIVA TNT 2 Model 64 display card. The window size was 512 x 512 pixels.

The first number in a cell is the FPS reported by FRAPS when the application was left to run for a minute or so. The number in brackets is the reported FPS when the mouse was moved rapidly inside the application window – this shows the response rate of the application independent of the particle system.

The results for PointParticles and LineParticles were slightly improved if point size and line size were unchanged and anti-aliasing was not switched on.

The number of points for each column must be divided by two to obtain the number of lines in LineParticles, and by four to get the number of quads in QuadParticles.

If we believe that a FPS of about 15 is reasonable, then the particle systems are capable of dealing with many thousands of points. About 15,000 points in the PointParticles system, about 15,000 lines (30,000 points) in LineParticles, and 1,000 quads (4,000 points) in QuadParticles. These numbers should only be considered approximations, but are pleasing nevertheless.

8. More Examples

There are several other particle system examples using BY_REFERENCE geometries and GeometryUpdater.

Chapter 5 of the Java 3D tutorial ("Interaction and Animation") has a section on the GeometryUpdater interface, and illustrates it with a particle system for a water fountain using a LineArray. It is quite similar to our LineArray example but uses a different algorithm for updating the positions of the lines. It also utilises the Geometry argument of the updateDate() method rather than accessing global float arrays.

Sun's Java 3D Demo page (http://java.sun.com/products/java-media/3D/demos/) contains two examples by Jeff White showing the use of lines and points to create a swirl of particles that change colour.

A Java 3D package for writing particle systems is available at j3d.org (http://code.j3d.org/). It utilises TriangleArrays or QuadArrays, and supports attributes for position, colour, texture, aging, wind movement, and forces. It supports collision detection through picking, and allows bounding boxes to be set to limit movement. It was written by Daniel Selman (who wrote the "Java 3D Programming" textbook), and the package is currently an alpha version.

9. Other Java 3D Approaches

The particle systems in this chapter were coded with BY_REFERENCE geometries and GeometryUpdater, but what other ways are there of writing particle systems?

One possibility is to represent each particle by a separate TransformGroup and Shape3D pair, and have the particle system move the shapes by adjusting the TransformGroups. This coding style is utilised in an example by Peter Palombi and Chris Buckalew which is part of Buckalew's CSc 474 Computer Graphics course (http://www.csc.calpoly.edu/~buckalew/474Lab5-W03.html). It has the advantage of being simple to understand, but isn't scaleable to large numbers of particles.

Another techniques is to store all the particles in a GeometryArray as before, but do the updates directly without the involvement of a GeometryUpdater. Synchronization problems can be avoided by detaching the Shape3D from the scene graph before the updates are made, and reattaching it afterwards. The detachment of the shape removes it from the rendering cycle, so synchronization problems disappear. There seems little advantage to this approach since the overheads of removing and restoring scene graph nodes are quite large.

If the application uses mixed mode or immediate mode rendering, then the programmer gains control of when rendering is carried out, and so can avoid synchronization problems.

The excellent Yaarq demo by Wolfgang Kienreich (http://www.ascendancy.at/downloads/yaarq.zip) is a mixed mode Java3D example which shows off a range of advanced techniques such as bump mapping, reflection mapping, overlays, and particle systems. It utilises a BY_REFERENCE TriangleArray and a GeometryUpdater. The particle attributes include size, position, direction, gravity, friction, alpha values for transparency, and a texture applied to all the particles.

There are three particle systems available at the WolfShade Web site (http://www.wolfshade.com/technical/3d_code.htm), coded in retained, mixed, and immediate modes. They don't use BY_REFERENCE geometries for storing the particles.

10. Non-Java 3D Approaches

The examples in this chapter show that the basic framework for a particle system is fairly standard, and that variations only really appear when we decide on a particle's attributes and how they change over time. Ideas for these can be gained by looking at implementations written in other languages.

Virtually every games programming book seems to have a particle system example. For instance, "OpenGL Game Programming" by Kevin Hawkins and Dave Astle, Prima Pub., 2001, implements a snow storm effect, which is quite easily translated from C++.

A very popular particle system API was created by David McAllister (downloadable at http://www.cs.unc.edu/~davemc/Particle/). Its C++ source code is available, as well as excellent documentation. It uses the interesting notions of actions and action lists. Actions are low-level operations for modifying particles, including gravity, bouncing, orbiting, swirling, heading towards a point, matching velocity with another particle, and avoiding other particles. Action lists are groups of actions which together make more complex effects. Many of the actions employ Euler's integration method, which underpins the workings of the parabolas in our examples. The API has been ported to various UNIXes, Windows OSes, and to parallel machines. The API would make an excellent model for a Java 3D-based package.

Most graphics software, such as Maya and 3D Studio Max, have particle system animation packages, that can be used for testing effects.

The two papers which introduced particle systems are by William T. Reeves:

> Reeves, W.T. 1983. Particle Systems – a Technique for Modeling a Class of Fuzzy Objects, *Computer Graphics*, 17 (3), p.359-376.
>
> Reeves, W.T. and Blau, R. 1985. Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems, *Computer Graphics*, 19 (3), p.313-322.

The first paper was written after Reeves's work on the movie "Star Trek II: The Wrath of Khan", where he created a wall of fire that engulfed a planet with a particle system made of points. The second paper came out of the graphics work for the animated cartoon "The Adventures of Andrew and Wally B" where the particles were small circles and lines used to model tree, branches, leaves, and grass.

GameDev has a small sub-section on particle systems in their "Special Effects" section (http://www.gamedev.net/reference/list.asp?categoryid=72).

