Modeling Particle Systems in Java 3D

Basic Studies in Natural Sciences
4th Semester, Spring 2009
Christos Karavasileiadis
Stephan O’Bryan
Group 9, House 13.2
Supervisor: Ole Torp Lassen
# Content

1. Abstract........................................................................................................pg.3
2. Introduction....................................................................................................pg.4
3. Target Group..................................................................................................pg.6
   3.1 Terminology..............................................................................................pg.6
4. Problem Formulation......................................................................................pg.7
   4.1. Problem Analysis....................................................................................pg.8
5. Particle Systems............................................................................................pg.10
   5.1. Particle Rendering..................................................................................pg.12
   5.2. Particle Generation................................................................................pg.12
   5.3. Particle Attributes................................................................................pg.14
   5.4. Particle Dynamics................................................................................pg.15
   5.5. Particle Extinction................................................................................pg.16
   5.6. Particle Hierarchy................................................................................pg.16
6. Particle Systems in our Approach.........................................................pg.17
   6.1 Particle Rendering................................................................................pg.17
   6.2 Particle Generation................................................................................pg.17
   6.3 Particle Attributes................................................................................pg.18
   6.4 Particle Dynamics................................................................................pg.19
   6.5 Particle Extinction................................................................................pg.21
   6.6 The Scene..............................................................................................pg.23
7. Discussion/Conclusion................................................................................pg.24
8. Bibliography................................................................................................pg.27
9. Appendix....................................................................................................pg.28
   • Source Code
1. Abstract

In computer graphics, modeling fuzzy objects that have irregular, complex and ill-defined surfaces such as fire, water, snow and smoke etc. is mostly done in a visual simulation technique called ‘Particle System’. The purpose of this project was to obtain basic knowledge of modeling particle systems and examine how visually realistic we can simulate a water fountain in Java 3D with this knowledge. First, we set out some criteria in order to evaluate our model. Then, we identified a number of fundamental features in general particle systems. After that, we spelled out our approach of modeling a particle system for our water fountain and discussed to what degree our animated representation captures the fundamental features. As a result, we managed to simulate the fountain-water quite accurately regarding their behavior and the way they move in a 3D scene, but the degree of realness of other objects in the scene could be extended depending on the application needs.
2. Introduction

Along with the evolution of computer science, the emergence of computer graphics and its applications becomes an essential part of the digital community. At the early stage, computer graphics was seen merely in the fields such as research, video games and applications for computer-aided design (CAD). Nowadays, the developments in computer graphics are great benefits to anyone using a personal computer. The simplest applications of computer graphics can be found in operating systems and applications with graphical user interfaces (GUIs). The main process of computer graphics is modeling and rendering virtual graphics objects with digital computers. The scope of the topics involved in this process extends a wide range of disciplines from mathematics and computer science to psychology and arts. Various sophisticated and ingenious techniques, algorithms and paradigms etc. have been developed along the way to enunciate and resolve the problems of modeling and rendering in computer graphics. Visualization techniques of computer graphics are used for various purposes, ranging from making simple histograms to dynamic 3D animations showing changes of winds or currents over time and forest fire simulations etc. in the same manner as popular computer games and motion pictures. Even those who never use a personal computer still have a big chance to see the results of computer graphics in virtual effect of motion pictures and commercials (on TV or in movies) where parts of scenes or even the whole film might be produced by computer graphics techniques.

One of the most significant computer graphics techniques/methods that is used for simulating “fuzzy” objects such as fire, clouds, snow and water is called ‘particle system’. (It is important to notice that this particle system is merely a visual simulation technique in computer graphics and has nothing to do with physical particles, for instance, water molecules). The appeal of particle systems, which beats any other computer graphics method, is their ability to create realistic natural phenomena in real time [1]. Particle systems have existed since relatively long ago in video games and computer graphics. For example, 2D pixel clouds were used in very early video games in the 1960s to simulate explosions. However, until around 1982, modeling phenomena such as fire, clouds, snow and water have been difficult with the existing methods of computer graphics. Around that time, when working on the film ‘Star Trek II: The Wrath of Khan’, William Reeves was in need of a method to create realistic fire for the Genesis Demo sequence. Reeves noticed that conventional modeling methods for creating objects that have smooth, well-defined and shiny surfaces do not suffice for modeling fuzzy objects like fire. These fuzzy objects have irregular, complex and ill-defined surfaces so that their motions cannot be described by the regular affine transformations [see 3.1] that are common in computer graphics. Reeves wanted to describe these fuzzy objects’ dynamic and fluid changes in shape and appearance. He realized that such objects would be better modeled as a system of particles that behaved within a set of dynamic rules so that he could achieve a chaotic effect. Hence, the particle system was born.

Basically, a particle system is a collection of 2D or 3D points in a plane or a space, respectively. An object can be modeled with particle systems as a cloud of primitive particles that define volume of the object. The representation of particle systems differs from their counterpart standard geometry objects in three basic ways:

“First, an object is represented not by a set of primitive surface elements, such as polygons or patches that define its boundary, but as clouds of primitive particles that define its volume.

Second, a particle system is not a static entity. Its particles change form and move with the passage of time. New particles are "born" and old particles "die."” [3]
Third, objects represented by particle systems are not deterministic—they do not rely only on predetermined path but each particle can have a random element that modifies its behavior. This random element is referred to as a ‘stochastic process’ and is used to generate and change the shape and appearance of an object within a particle system. Stochastic processes make the resulting effect looks organic and natural. [1]

Despite availability of different programming languages for implementing particle systems, there are various reasons for using java 3D as application platform in this project. Regardless of being a relatively new programming language, the popularity of java has grown rapidly because of its unique characteristics and features, such as platform independence, simplicity, and object-oriented programming support. For computer graphics modeling and rendering, java 2D and java 3D provide powerful, natural and object-oriented interfaces.
3. Target Group

We will make introduction to basic terminology of computer graphics in this report. However, the target group of this report is those who have knowledge in computer graphics, especially in Java 3D and basic knowledge in programming, preferably in Java.

3.1 Terminology

In this session, we introduce the technical terms that we employ in this report:

**2D and 3D**: 2D and 3D re abbreviations for 2-Dimensional and 3-Dimensional.

**Affine Transformation**: “Geometric objects go through a transformation stage before being rendered. A general family of geometric transforms commonly used in computer graphics is called affine transforms. An affine transformation preserves lines and parallelism—that is, it is a projective transformation that maps parallel lines to parallel lines. The common affine transforms include: Translation, Rotation, Reflection, Scaling, Shearing.” [2]

**Frame**: A frame is a single rendered image on a screen at any one time (e.g. one millisecond).

**Fuzzy Phenomena (objects)**: Phenomena such as smoke, fire and explosions are called fuzzy because, “due to the absence of straight edges, they appear blurred when rendered: they are composed of particles.”

**Java 3D**: Java 3D is a high-level API that was design by Sun Microsystems for implementing and manipulating 3D scenes. The Java 3D library consists of various classes for implementing shapes, materials, sounds, lights and many more that can all be added to a scene graph and from there be shown on a computer screen.

**Primitive Shapes**: Primitive shapes in computer graphics are no complex shapes such as circles, squares, rectangle etc. By using primitive shapes, complex shapes can be created (e.g. a house).

**Rendering**: Rendering is the creation of the image that has to be represented on the screen with the use of mathematic models.

**Three-dimensional coordinates**: Coordinates in this report always refer to a ‘right-handed coordinate system’, i.e. when one look at thumb, index and middle fingers of his own right hand with the middle finger pointing towards himself, the thumb/the line parallel to the thumb is X-axis, the index/the line parallel to the index is Y-axis and the middle finger/the line parallel to the middle finger is Z-axis. Therefore, when someone views at the screen, he/she is looking from positive end of Z-axis to the negative end.
4. Problem Formulation

The main question of this report is:

How visually realistic can we simulate a water fountain in a computer using Java 3D?

In order to illustrate our solution better, we will attempt to simulate an existing water fountain. Here is a picture of the chosen fountain:

![Image of the fountain](image)

Fig.1. This fountain’s location is Mozarts Plads, Copenhagen SV. Already from this picture we can identify a number of features regarding the fountain:

- The physical structure of pool is a circular torus with a cone in the middle from where the water is emitted.
- The water is emitted vertically upwards and then it is spread to the side.
- When the emitted water has reached the surface of the pool water, ripples are created from this interaction.
- There are reflections on the surface of the pool water.
4.1. Problem Analysis

In this session we will attempt to describe how we are going to approach our solution to the problem formulation. It might be a good idea for someone who does not have knowledge about particle systems in computer graphics to read the chapter Particle Systems In General in this report before reading the Problem Analysis. It will help the reader to get familiar with this particle system technique and also give a better understanding on our decisions.

As mentioned in the problem formulation, the problem that will be discussed throughout this report is how visually realistic we can simulate a water fountain in a computer. So the first question that is raised here is; what is visually realistic? Actually, this question is one of the most abstract questions that can be asked, since every different person has a different perception of visual realism. An interesting research about this matter was made by a Microsoft’s research group [7], where they showed to a group of people a variety of photographs and computer generated images, and ask each of them to say whether they think if those pictures where from the real world or made in a computer. But as mentioned before a matter like defining visual realism is debatable, so they set up a number of factors that they found most relevant to define visual realism [7]:

1. Shadow Softness
2. Surface Smoothness
3. Number of Objects in the Scene
4. Variety of Object Shapes
5. Number of Light Sources

However, we have to set up different ones since we want to focus mostly on how visually realistic the dynamic part of the fountain looks. The evaluation criteria we have chosen are:

1. Water movement (Applying physics laws i.e. kinematics and forces).
2. Interaction of running water with the water surface.

Having these as our criteria, we can already state that we will not focus on making an accurate model of the fountain’s surroundings, but we will attempt to seek the best solution for the actual fountain-water.

The physical body of the water fountain in a 3D scene can be created easily by using some primitive geometry shapes and computer graphic techniques with the assist of Java 3D libraries. The actual difficult part of the problem is the representation of the running water. This is where we put our focus in this report.

In order to make our decisions regarding problem formulation, we will first make some observations about the fountain we will simulate:

- The water has whitish color and it is transparent.
- The water does not have a whole shape but it consists of smaller parts.
- The water is pumped vertically (along positive Y-axis in Java 3D) upwards.
- Water drops also move horizontally (on XZ-plane in Java 3D) depending on the wind direction.
- Due to gravity, water drops fall down on the surface of the pool.
• The combination of the previous three is the reason that the running water has a parabolic movement.
• When the pumped water lands on the surface of the pool, some ripples are visible.
• Last and most important, the water drops’ movement are random and different from each other. Some drops fall down faster than other, while other drops do not manage to get that far from the fountain compared to others.

We can understand from the previous observations, that this random movement of water can not be described very good with affine transformations, obviously because affine transformations are deterministic. That is the reason we need the particle system technique to describe objects such this.

Now the next step is to decide what attributes we will assign to particl
• Color: White
• Transparency: A transparency rate will be assign to each particle.
• Shape: the particles will be represented as points in the 3D space.
• Size: We will rather use a large number of small particles to represent the water drops instead of using a few big ones.
• Velocity: The velocities will be different and random for each particle on the X, Y and Z-axis. Though a positive speed on the Y-axis will be the reason that our particles will move upwards. The speeds on the X and Z-axis will decide the direction that the particles will move horizontally.
• Forces: Gravity will apply on the Y-axis and forces that represent air can be applied on the X and Z-axis.

One thing that we will not put effort is the representation of the ripples on the surface of the pool. In order to achieve this effect we will have to make our particles to interact with their environment (in this situation is the pool). But we will not include this effect due to the lack of time and as our focus is on the fountain-water rather than the water in the pool.
5. Particle Systems in General

In this session, we will spell out basic requirements for particle systems according to William T. Reeves [4].

With particle systems, we are able to model fuzzy phenomena such as fire, smoke, explosion, blood and many more. What these fuzzy phenomena have in common is that surfaces are complex, ill defined and blurry. We can say that fuzzy objects’ appearance is random.

A particle system consists of hundreds of thousands of particles that are basically primitive shapes in 3 dimensions (one can also model a particle system in 2 dimensions, then particles are 2 dimensional shapes). These particles have different attributes such as age, position, color, velocity, forces, size etc. Depending on the phenomena that one has to model, different attributes can be assigned to the particles. Actually, what makes particle systems “special”, compared to other techniques for modeling fuzzy phenomena, is the fact that each particle has an age, or better, a lifetime. Particles are born and particles die. During this lifetime, their attributes change.

When it comes to modeling fuzzy objects, particle systems surpass other classical surface-oriented in three ways [3]:

- Since particles are primitive 3D points, the computation time that is needed to process a complex image is less than using other techniques such as ‘affine transformation’. Motion-blur is also easier with particles.
- Random numbers control a particle system. Acquiring a high-level detailed model does not require a great deal of human design and time.
- The form of the particles changes over their lifetime. Complex dynamics of this form is easier to represent with a particle system than other static techniques.

An important fact about particle systems is that the motion of each particle cannot be described by simple affine transformations. Instead, the particle’s motion is described by complex equations or differential equations. Generally, we can use different models that have been introduced in other sciences. This is valid since particles’ motions depend on their attributes (e.g. velocity and forces such as gravity) and these attributes are not static. So describing a particle’s motion with an affine transformation will look rather unrealistic.

Another important issue is about the computation of each frame on an animated particle system (in other words, when particles are in motion). An efficient particle system will be one that does not use a lot of computer resources. In general, large particle systems (with millions or billions of particles) can be rather “heavy” for even the newest hardware since the GPU (Graphics Processing Unit) has to process every single particle. That is also the reason why particles die. In his paper, Reeves describes a few steps that have to be taken in order to compute the frames in an animated particle system as follow:
- New particles have to be generated.
- Attributes have to be assigned to each particle (It is important to remember that each particle has unique attributes. Take as an example, the position of each particle—it differs in all alive-particles).
- Particles that surmount their lifetime vanish (From the computer screen and from the computer’s memory).
- The alive-particles have motion according to their attributes
- The image of the alive-particles is rendered.

In the rest of this chapter, we will address these steps one by one in details.

One of the most important issues with particle systems is ‘efficiency’ since a particle may consists of a huge amount of individual particles up to hundreds of thousands. Particle rendering, updating and reusing (e.g., a dead particle may be sent back to its initial position and reset to be born again) are influenced by efficiency.
5.1. Particle Rendering

Unlike a full-fledged 3D shape, such as a sphere or a cylinder, a particle is typically rendered as a graphics primitive, such as a point or line. However, particle systems have started to deploy basic geometry objects (such as triangles and rectangles) with the emergence of more powerful graphic cards. The advantage is that basic geometry objects allow textures and lighting to be assigned.

5.2. Particle Generation

Particle systems use controlled stochastic processes to generate particles. A stochastic process determines the number of particles that enter the system during each interval of time (at a given frame). As the number of particles generated strongly influences the density of the fuzzy object, it is a most central and practical issue in a particle system.

Unless the designer wants a fixed number of particles in a system, there are two ways that the designer of a particle system can choose to control the number of new particles:

“In the first method, the designer controls the mean number of particles generated at a frame and its variance. The actual number of particles generated at frame $f$ is

$$N_{\text{Parts}}_f = \text{Mean}_f + \text{Rand} \times \text{Var}_f$$

where $\text{Rand}$ is a procedure returning a uniformly distributed random number between -1.0 and +1.0, $\text{Mean}_f$ is the mean number of particles and $\text{Var}_f$ is its variance.

In the second method, the number of new particles depends on the screen size of the object. The model designer controls the mean number of particles generated per unit of screen area and its variance. The particle system can determine the view parameters at a particular frame, calculate the approximate screen area that it covers, and set the number of new particles accordingly. The corresponding equation is

$$N_{\text{Parts}}_f = \left(\text{Mean}_s + \text{Rand} \times \text{Var}_s\right) \times \text{Screen Area}$$

where $\text{Mean}_s$ is the mean per screen area, $\text{Var}_s$ is its variance and $\text{Screen Area}$ is the particle system’s screen area. This method controls the level of detail of the particle system and, therefore, the time required to render its image. For example, there is no need to generate 100,000 particles in an object that covers 4 pixels on the screen.” [5]

A simple linear function for the mean number of particles in a given frame $f$ is [5]:

$$\text{Mean}_f = \text{Initial}_f + \text{Delta}_f \times (f - f_0)$$

(or)

$$\text{Mean}_s = \text{Initial}_s + \text{Delta}_s \times (f - f_0)$$

where $f$ is the current frame, $f_0$ is the first frame during which the particle system is alive, $\text{Initial}_f$ is the mean number of particles at this first frame and $\text{Delta}_f$ is its rate of change. The variance controls, $\text{Var}_f$ and $\text{Var}_s$ are currently constant over all frames.
However, it is possible to add more sophisticated quadratic, cubic, or perhaps even stochastic variations in both the mean and variance parameters. The designer specifies $f_0$ and either the parameters $\text{InitialMeanParts}$, $\text{DeltaMeanParts}$ and $\text{VarParts}$, or the parameters $\text{InitialMeanParts}_{sa}$, $\text{DeltaMeanParts}_{sa}$, and $\text{VarParts}_{sa}$ in order to control the particle generation.
5.3. Particle Attributes

Depending on the effect required, the attributes of a particle might vary from system to system. However, typical attributes include initial position, initial velocity (both speed and direction), initial size, initial color/texture, initial transparency, shape, age/lifetime and forces (e.g., gravity). Despite validity of other approaches (for instance, a particle’s new position is dependent of its previous position), updating particle attributes usually involves time-based equations. Attributes of a particle are changed by the particle system.

The origin of a particle system is defined by its position in a three-dimensional space. Particle systems often have a so-called ‘generation shape’, which defines a bounding region about its origin where particles are placed. Furthermore, there is usually a point/space within the bounding region where particles are created. For example, particles for a water fountain are created at a small point/space near the fountain’s base. Particle updating and aging also take place within the bounding region. For instance, if a particle moves outside the point/space where it was born, then it begins to age more and more as it moves further away. The aging could affect the particle’s color and/or size, depending on the application’s needs. Furthermore, a particle dies/disappears if it moves outside the bounding region. A generation shape could be for instance, a sphere of radius $r$, a circle of radius $r$ in the XY-plane or a rectangle of height $h$ and width $w$ in the XZ-plane etc. The generation shape also defines the initial direction in which new particles move. For example, particles move outward away from the origin of the particle system in a spherical generation shape (see Figure 2). Particles move upward from XY-plane or XZ-plane in the cases of circular and rectangular shapes, respectively, but they are allowed to vary from the vertical according to another parameter called “ejection” angle. Ejection angle is the angle that restricts the scope of the initial direction of the particle's movement (see Figure 3).

Fig.2. Typical particle system with spherical generation shape. (Figure taken from William T. Reeves: Particle Systems A Technique for Modeling a Class of Fuzzy Objects. Computer Graphics, Volume 17, Number 3, July 1983, p.363 http://www.lri.fr/~mbl/ENS/IG2/devoir2/files/docs/fuzzyParticles.pdf). The particles are born at the origin of the particle system and they move outward away from the origin of the particle system.
An average color (average Red, Green and Blue values) is assigned to a particle system to determine a particle’s initial color and maximum deviation from that color.

Size and transparency of a particle are also determined by mean values and maximum variations akin to the above equation for the initial speed.

The shape of each particle is specified by a parameter in its particle system. The shapes are for instance, spherical, rectangular and streaked spherical.

The number of attributes may be plenty, depending on the application’s needs.

5.4. Particle Dynamics

Each particle in a particle system may change over time in color, transparency and size as it moves in three-dimensional space. A particle can be moved from one frame to the next by adding its velocity vector to its position vector. An acceleration factor is used in order to modify the velocity of each particle. This acceleration parameter is also used for simulating gravity and cause particles to move in parabolic arcs rather than in straight lines.

The changes in color, transparency and size of a particle over time can be controlled by the rate-of-color-change, rate-of-transparency-change and rate-of-size-change parameters accordingly. These rates of change may be global for all particles or this parameter could be stochastic too.
5.5. Particle Extinction

As mentioned before, one attribute that can be assigned to a particle is age. The age of particle is given in frames. So, whenever a frame in our animation is computed, the age increments or decrements (depending always on the way someone wants to program) and when it reaches a fixed limit, the particle will die. But regarding particle extinction, other techniques can be used as well in order to “kill” a particle. For example particles can be programmed to die whenever their speed is below a chosen limit or whenever they move at a certain area.

5.6. Particle Hierarchy

We can use particle hierarchies [4] in order to represent complicated fuzzy objects. A particle hierarchy is basically a particle system whose particles are particle systems themselves. This way we can achieve the control of a group of systems only by controlling the parent system. The children systems can inherit the attributes from the parent system and basically every equation regarding the change of velocity and so on.

Since this mechanism is not used in our report we will not go in more details. We only chose to hint what may be achieved using particle systems. For more details, see William T. Reeves’ paper [4].
6. Particle Systems in our Approach

We use Java 3D for implementing our water fountain and inspire from KGP book [6]. In Java 3D, particles can be represented as elements of a Java 3D GeometryArray [9]. For example, particles which are points can be stored in PointArray, particles which are lines can be stored in a LineArray and a QuadArray for quad particles (i.e. particles that textures can be assigned). For more for information about the GeometryArray, see [9].

6.1 Particle Rendering

In our water fountain model, water particles are rendered as points and they are represented as elements of a Java 3D PointArray, which is a subclass of GeometryArray.

In the Particles class, the GeometryArray is created with a BY_REFERENCE flag so that the PointArray refers to data structures stored in the user’s execution space:

```java
PointArray pointParticles = new PointArray(numPoints, PointArray.COORDINATES | PointArray.COLOR_3 | PointArray.BY_REFERENCE);
// allow the data of the array to be read and written.
pointParticles.setCapability(GeometryArray.ALLOW_REF_DATA_WRITE);
pointParticles.setCapability(GeometryArray.ALLOW_REF_DATA_READ);
```

pointParticles is reference to two data structures namely, the coordinates of the PointArray and the colors of the points at those coordinates.

The Particles() constructor creates the GeometryUpdater object to update the geometry and the Behavior objects to trigger the updates:

```java
PUpdater updater = new PUpdater();
partBehav = new ParticlesControl(delay, updater);
```

The createGeometry() and createAppearance() methods initialize the Shape3D’s geometry and appearance, respectively.

6.2 Particle Generation

Our particle system does not use stochastic process to determine the number of particles that enter the system at each frame; instead the number of particles is constant, i.e. there is same number of particles at every frame. The particles are created at a given point, which is the centre of the fountain base.

The number of particles and (x, y, z) coordinate of the particle generating point are determined in the WaterFountain class. In the Particles class, the Particles() constructor assigns the particles as points and these points are stored in a PointArray called PointParticles:

```java
int numPoints = numParticles;
PointArray pointParticles = new PointArray(numPoints, PointArray.COORDINATES | PointArray.COLOR_3 | PointArray.BY_REFERENCE);
```
6.3 Particle Attributes

Each particle in our particle system has—position, velocity, acceleration, gravity, age, color, size and transparency attributes.

In **Particles** class, float arrays for the position, velocity, acceleration, color and age attributes are created because Java 3D only supports reference to float arrays:

```java
private float[] coord, vels, accs, colors, age;
```

In `createGeometry()` method, the number of particles (points) is multiply with 3 because the position, velocity, acceleration attributes have \((x, y, z)\) values and the color attributes have \((R, G, B)\) value. They are stored in the float arrays as:

```java
coord = new float[numPoints*3]; // to store each \((x, y, z)\) coordinate
vels = new float[numPoints*3];
accs = new float[numPoints*3];
colors = new float[numPoints*3]; // to store each \((R, G, B)\) value
```

// store the coordinates and colours in the PointArray
pointParticles.setCoordRefFloat(coord); // use BY_REFERENCE
pointParticles.setColorRefFloat(colors);
setGeometry(pointParticles);

And the age attributes are stored as:

```java
age = new float[numPoints];
```

The gravity, size and transparency attributes are initialized as:

```java
private static final float GRAVITY = 9.8f;
private final static int POINTSIZE = 3;
private static final float transAlpha = 0.5f;
```

When `setparticle()` method is called, the position, velocity, acceleration, color, age attributes of each particle is determined. \(Y\_VELOCITY\) is used for particles moving upwards along Y-axis and \(XZ\_VELOCITY\) is for particles moving on XZ-plane. The \(-GRAVITY\) is used as each particle’s acceleration on positive Y-axis.

In `createAppearance()` method, the size and transparency attributes are set to the appearance:

```java
PointAttributes pa = new PointAttributes();
pa.setPointSize(POINTSIZE); // sets the size of the particles
```

```java
TransparencyAttributes ta = new TransparencyAttributes();
ta.setTransparency(transAlpha);
```
6.4 Particle Dynamics

For every new frame, we need some of the particles’ attributes to be changed. The attributes that are going to be changed depend mostly on the position of the particle. For start, we should say that color is constant and age decrements by one on each frame. So the interesting part is regarding the position, speed and acceleration of each particle.

Observing the real-life water fountain, we have noticed that each particle is pumped upwards but at a certain height, where its velocity reaches zero, it starts falling down with a parabolic movement instead of falling down vertically. This happens because “old” particles, which have a small velocity, interact with the “new” particles that are moving vertically upwards with bigger velocity, so the old ones are casted to the side. The problem here was that since we are not implementing interactions between particles, we could not represent this effect realistically. What we have done instead is to give to the particles speeds on the X and Z – axis whenever the particles’ velocity on Y - axis reaches 0. When a particle needs to be updated, the following method is called:

```java
public void updateParticles(int i)
{
    //Change of velocity and position according to accelearated motion physics.
    //It calculates the position of each particle every TIMES
    coord[i] += vels[i] * TIMESTEP + 0.5 * accs[i] * TIMESTEP * TIMESTEP;  // x coord
    coord[i+1] += (vels[i+1] * TIMESTEP + 0.5 * accs[i+1] * TIMESTEP * TIMESTEP);  // y coord
    coord[i+2] += (vels[i+2] * TIMESTEP + 0.5 * accs[i+2] * TIMESTEP * TIMESTEP);  // z coord

    // calculate new velocities for every TIMESTEP.
    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
}
```

where coord is the position, vels is the velocity, accs is the acceleration and TIMESTEP is change in time. The i represents the X- axis. For the Y- axis we write the same equations with i+1 and for the Z- axis with i+2. In case the reader is unaware, these functions are based on the kinematic theory in physics. We have taken these functions from Andrew Davison’s KGP book example classes [6].

The particles start with initial velocities and acceleration (gravity) on the Y- axis and no velocities on the X and Z - axis. Now, there are two things that might happen. The first is that if there is no acceleration on the X and Z- axis, then the particles move only upwards vertically, and in order to describe the “old particles casted to the side” phenomenon, we call the following function when the Y-velocity gets to zero:

```java
public void setXZ(int i)
{
    // we set random velocities with random signs.
    double xvel = Math.random() * XZ_VELOCITY;
    double zvel = Math.sqrt((XZ_VELOCITY * XZ_VELOCITY) - (xvel*xvel));  //Using Pythagoream theorem
}
```
```java
float j = (float)Math.random();
// x velocity
if (j > 0.5)
    { vels[i] = (float)xvel; }
else if (j < 0.4)
    { vels[i] = -(float)xvel; }
else { vels[i] = 0; }

float k = (float)((Math.random()));
// z velocity
if (k > 0.5)
    { vels[i+2] = (float)zvel; }
else if (k < 0.4)
    { vels[i+2] = -(float)zvel; }
else if (k > 0.4 && k < 0.5)
    { vels[i+2] = ((float)xvel); }
else { vels[i+2] = 0; }

// y velocity after the point where the particles get the X,Z velocities.
vels[i+1] = (float)Math.random() * 0.5f; //because they are pushed so they ought to get some small velocity at least

What happens here is that for each particle, a random X and Z velocity is calculated. And then using a random number, we decide whether these velocities should be negative or positive. Be aware that the sign of the velocities on the X or Z-axis decide the direction on the horizontal plane while the sign of the velocity on the Y-axis decide whether the particle should move upwards or downwards. We have gravity on the Y-axis as well (Negative gravity means that the gravity is opposite to the way that the particle moves on the vertically. Since our particles move upwards, a negative gravity takes them to the ground), which means that at a certain point the Y-velocity of particles will go to 0 and then become negative. This way we get the parabolic movement.

The second thing that can happen is that there are accelerations on the X and Z speed applied from the beginning. These accelerations basically represent air. If these happens, there is no need for calling the setXZ() method, because the particles are casted to the side from the air. But since gravity applies on the vertical movement, the particles fall down.
6.5 Particle Extinction.

When a particle is initialized, it gets a certain age. This age is decrementing by 1 every new frame. When the age become less than 1, then the particle dies and it is reinitialized. The reason that we “kill” the particle when its age is less than 1 instead of just becoming zero is because; we initiate a random age for each particle, which is basically a float number. So we might find ourselves in a situation where the particle has the age of f.x. 1,02, which means at the next frame its age won’t become 0 but 0,2 and then at the next frame it will become -0,8 and so on. This means that the age will never become 0 or it will become negative (this is not possible in real-life). So killing the particle when its life is below 1, will give us the right amount of life.

The next thing regarding the extinction of a particle, is to see how long in real-time, the particle live. In our program we initialize the age of the particle as:

\[
\text{age[i/3]} = (\text{float})\text{Math.random()} \times \text{life};
\]

where \(\text{life}\) is equal to 700 in our implementation. The \(\text{Math.random()}\) is basically a number that is greater or equal to 0.0 and less than 1.0. So the age of a particle is greater or equal to 0 frames or smaller than 700 frames. In our class, a new frame is generated every 1ms (delay variable in our class), which means that in overall a particle can leave between 0 and 700ms (0,7 sec). A general equation that can describe this is:

\[
\text{ParticleRealTimeLife} = \text{ParticlesAgeInFrames} \times \text{NewFramePeriod}
\]

The method where we check if the particle should be updated or reinitialized, is the following:

```java
public void updateData(Geometry geometry) {
    for (int i=0; i < numPoints*3 ; i=i+3) {
        if (coord[i+1] <= (startY) && direction[i/3] == true && age[i/3] > 1) {
            age[i/3] = age[i/3] - 1;
        } else if (age[i/3] < 1){setParticle(i);}

        else if (vels[i+1] <= 0f && direction[i/3] == false && coord[i+1] >= 0.2f && accs[i] == 0f && accs[i+2] == 0f) {
            setXZ(i);
            age[i/3] = age[i/3] - 1;
            direction[i/3] = true;
            updateParticles(i);
        }

        else if ((vels[i+1] <= 0f && direction[i/3] == false && coord[i+1] >= 0.2f) && (accs[i] != 0f || accs[i+2] != 0f)) {
            age[i/3] = age[i/3] - 1;
            direction[i/3] = true;
            updateParticles(i);
        }

        else
```
Notice that if a particle reaches the starting position on the Y-axis, it stays a bit inactive. We chose to perform such an action to illustrate that the particles stay a bit on the surface of the water and then disappear after they reach an age smaller than 1. In the other cases, if the age is bigger than 1, then the particles are updated.
6.6 Java 3D Scene.

The particle system can be used in a scene as a transform group. In our example, we create our scene in the class called WaterFountain. Our system is initialized in the following way:

```java
BoundingSphere bound = new BoundingSphere(new Point3d(0,0,0), BOUNDSIZE);

Particles ptsFountain = new Particles(3000, 0f, 0.05f, -0.035f, 0.2f, 3.0f);

TransformGroup posnTG = new TransformGroup();
Transform3D trans = new Transform3D(); // Transformation for the water fountain

posnTG.setTransform(trans);
posnTG.addChild(ptsFountain);
theScene.addChild(posnTG); // Adds the main transform group to the scene

// timed behaviour to animate the fountain
Behavior partBeh = ptsFountain.getParticleBeh();
partBeh.setSchedulingBounds(bound);
theScene.addChild(partBeh);
```

The constructor of the particle system is:

```java
public Particles(int numParticles, float sx, float sy, float sz, float vxz, float vy)
```

where numParticles is the number of particles we want to use in the system, sx-sy-sz represent the initial position from where the particles are emitted in the 3D coordinate system and finally vxz and vy are the velocities on the XZ – plane and Y- axis respectively.

The last thing we would like to address to the reader is that a behaviour object should be created and assigned to the particle system; otherwise the system will not be animated:

```java
Behavior partBeh = ptsFountain.getParticleBeh();
partBeh.setSchedulingBounds(bound);
```
7. Discussion/Conclusion

Implementing the particle system that was described in the session Particle Systems In Our Approach will give results that are shown in the following pictures:

Fig.4. The result of implementing the particle system described in the session Particle Systems In Our Approach in this report. In these 3 pictures, there is *no air in the system*. Notice the movement of the particles, which is correct, but there is an important factor missing: Ripples on the water surface.
Fig. 5. The result of implementing the particle system described in the session Particle Systems In Our Approach in this report. In these 3 pictures, there is **air in the system**

Now we have to discuss whether we fulfilled the criteria we set up for visual realism and what is our answer to the problem formulation. To begin with, we will address again our criteria and discuss for one by one our approach. Our criteria were:

1. Water movement (Applying physics laws i.e. kinematics and forces).
2. Interaction of running water with the water surface.

Starting from 1, we can say that the physics laws that are applied to the system give each particle a movement that can represent the movement of a water drop in real life almost 100% realistically. But we say almost, because there is an important fact to discuss— When testing the particle system in a large computer screen and after zooming in, we found out that there are gaps between the particles’ movement on Y-axis. The reason for this is that whenever the new position of a particle was calculated, the particle was shown there at the next frame. So for example, if we assume that the particle was in position $Y=1$ and then the next position calculated was $Y=2$, then the particle jumps from 1 to 2, so there is a visible gap. The reason that this happens is because no position in between these two positions was calculated. A way to fix this problem is simply to use smaller TIMESTEP, but this will end up in getting a slow water movement. But as mentioned before, this effect with the gaps between the particles is visible only on higher resolution screens and thus we did not put so much effort in perfecting it. In reality, water drops of course would have passed through all points on the Y-axis without doing the jump, but with a continuous movement. The rest we can say about physics laws in our system is that by adding random values to accs[i] and
accs[i+2], we get an effect of blowing air. The sign on these accelerations decide the direction of the air and it is actually a nice feature for our system.

Regarding 2, we have already mentioned throughout the entire report that we discarded it from the beginning. This was for various reasons, but we can address the main one; lack of time and knowledge. We researched and found out that in order to create ripples on the water surface, we need to learn the whole theory about dynamic surfaces, and still, to model realistic ripple movements would require much more time than we had. Therefore we decided that we would only focus on our main problem, mainly the implementation and theory of particle systems.

At last about 3, we have to break our answer in two parts: The visual aspects of the particles and the visual aspects of the particle system’s surrounding. Starting from the latter, we fail to find out how to use anti-aliasing in Java 3D and also we had a lack of knowledge in using shades in a scene. So someone can easily understand that the surroundings are made in a computer. Regarding the particles’ appearance, we used points to represent the water drops. These points are anti-aliased so they look very smooth and they have a whitish transparent color so it can be understood that they represent water. Though, Java 3D does not offer so many possibilities for creating photo-realistic scenes, so even if our implementation was better, it still would not look perfect.

Finally, our conclusion and answer to the problem formulation is that using Java 3D, we can simulate fuzzy phenomena such as the water fountain quite accurately regarding their behavior and the way they move in a scene, but it lacks some aspects that could make our generated scene to look sharper (e.g. A explicit way for the scenes anti-aliasing).
8. Bibliography


   Computer Graphics, Volume 17, Number 3, July 1983, p.359  

   Computer Graphics, Volume 17, Number 3, July 1983, p.359- 375  


[7] Paul Rademacher, Jed Lengyel, Edward Cutrell & Turner Whitted:  
   Measuring the Perception of Visual Realism in Images.  
   University of North Carolina at Chapel Hill, Microsoft Research  


[9] http://download.java.net/media/java3d/javadoc/1.3.2/
9. Appendix
   . Source code

Particels Class
/**
The construction of this class was inspired from the book Killer Game Programming in JAVA by Andrew Davison. Refer to Chapter 21.
* * Authors:Christos Karavasileiadis (dika@ruc.dk)
* Stephan O'Bryan (mhaldj@ruc.dk)
* * Date: 3rd June, 2009
**/

import java.util.Enumeration;
import javax.media.j3d.Appearance;
import javax.media.j3d.Behavior;
import javax.media.j3d.Geometry;
import javax.media.j3d.GeometryArray;
import javax.media.j3d.GeometryUpdater;
import javax.media.j3d.PointArray;
import javax.media.j3d.PointAttributes;
import javax.media.j3d.Shape3D;
import javax.media.j3d.TransparencyAttributes;
import javax.media.j3d.WakeupCondition;
import javax.media.j3d.WakeupOnElapsedTime;

public class Particles extends Shape3D
{
    //instance fields
    private final static int POINTSIZE = 3;
    private static final float GRAVITY = 9.8f;
    private static final float TIMESTEP = 0.001f;
    private static final float XZ_VELOCITY;
    private static final float Y_VELOCITY;

    private PointArray pointParticles; // Geometry holding the coords and colors
    private ParticlesControl partBehav; // The Behaviour triggering the updates

    private float[] coord, vels, accs, colors, age; // We must use floats because Java 3D 1.3 only supports reference to float arrays.
    private boolean[] direction;
    private int numPoints;

    private float startX,startY,startZ;
    private static final int delay = 1;
    private static final float transAlpha = 0.5f;
    private static final float life = 700f;

    // Constructor
public Particles(int numParticles, float sx, float sy, float sz, float vxz, float vy) {
    numPoints = numParticles;
    this.startX = sx;
    this.startY = sy;
    this.startZ = sz;
    XZ_VELOCITY = vxz;
    Y_VELOCITY = vy;
    // creates a PointArray with a number of points to be created and uses the BY_REFERENCE.
    // This way we only have to update the user's arrays of coordinate etc.
    // Then the Point array will fetch automatically the values from the referenced arrays.
    pointParticles = new PointArray(numPoints, PointArray.COORDINATES|PointArray.COLOR_3|PointArray.BY_REFERENCE);
    // We also have to allow the data of the array to be read and written.
    pointParticles.setCapability(GeometryArray.ALLOW_REF_DATA_WRITE);
    pointParticles.setCapability(GeometryArray.ALLOW_REF_DATA_READ);
    PUpdater updater = new PUpdater();
    partBehav = new ParticlesControl(delay, updater); //a new frame every delay (3 ms)
    createGeometry();
    createAppearance();
}

public Behavior getParticleBeh() {
    return partBehav;
}

public void createGeometry() {
    // Here we create the referenced arrays. One for coordinates and one for color.
    // Notice that since coordinates and colors take 3 values, (X,Y,Z) and (R,G,B) respectively,
    // we have to create arrays with multiple of 3 elements.
    // Another important fact is that referenced arrays must be floats !!!
    coord = new float[numPoints*3];
    colors = new float[numPoints*3];
    // Creation for arrays regarding velocity, aceleration, age and direction.
    // As the use of 3 first arrays is quite obvious, the direction has a very vital role in the way this fountain is implemented.
    // The value of the direction for each particle is decided whenever the particle has fulfilled certain conditions.
    // Notice that all the arrays except the age and direction have size numPoints*3. This is because we need to store three values for
    // each particle, regarding their speed, aceleration and position on the X,Y,Z-axis and also three values for the RGB standard
    // But each particle has only one age value and direction.
    vels = new float[numPoints*3];
accs = new float[numPoints*3];
age = new float[numPoints];
direction = new boolean[numPoints];

for (int i=0; i< numPoints*3; i=i+3)
{  setParticle(i);}

// store the coordinates and colours in the PointArray
pointParticles.setCoordRefFloat(coord); // use BY_REFERENCE
pointParticles.setColorRefFloat(colors);

setGeometry(pointParticles);

//*********************************************************
// This method "gives birth" to particles. Inspired from KGP in JAVA by Andrew Davison, chapter 21.
// Every particle here gets its initial attributes. These attributes change over time as the particle is updated.
// When a particle dies, then it is reinitialized.
public void setParticle(int i)
{
    coord[i] = startX; coord[i+1] = startY; coord[i+2] = startZ;  // (x,y,z) coordinates for the starting point.
age[i/3] = (float)Math.random() * life; // age of each particle
direction[i/3] = false;

    vels[i] = 0; // x velocity
    vels[i+2] = 0; // z velocity

    //Accelerations: (wind) on x, z and gravity on y for each particle. If both accs[i] and accs[i+2] are 0 then there is no wind in our system.
    // A value different than zero on the X or Z acceleration means that wind is applied.
    accs[i] = 0f;//((float)Math.random() * 3f;
    accs[i+1] = -GRAVITY;
    accs[i+2] =0f; //-((float)Math.random() * 3f;

    float rand = (float)Math.random();

    // y velocity
    if ((rand > 0.5f && rand < 1.0f) && (accs[i] !=0 || accs[i+2] !=0) )
    {  vels[i+1] = rand * Y_VELOCITY;}
    else if (accs[i] == 0 && accs[i+2] == 0)
    {vels[i+1] = Y_VELOCITY;}
    else {setParticle(i);}

    //Color for each particle
    colors[i] = 0.8f;
    colors[i+1] = 0.8f;
colors[i+2] = 1f;
}
//****************************************************************************** END of setParticle(int i)
******************************************************************************

// In the case that there is no wind in the system, when the velocity of the particle on the Y-axis get to 0, it gets velocity on the X,Z axis.
// This way we achieve the parabolic movement of the particles.
public void setXZ(int i)
{
    // we set random velocities with random signs.
    double xvel = Math.random() * XZ_VELOCITY;
    double zvel = Math.sqrt((XZ_VELOCITY * XZ_VELOCITY) - (xvel*xvel)); //Using Pythagoream theorem
    float j = (float)Math.random(); // x velocity
    if (j > 0.5)
        { vel[i] = (float)xvel; }
    else if (j < 0.4)
        { vel[i] = -((float)xvel); }
    else { vel[i] = 0; }

    float k = (float)((Math.random())); // z velocity
    if (k > 0.5)
        { vel[i+2] = (float)zvel; }
    else if (k < 0.4)
        { vel[i+2] = -((float)zvel); }
    else if (k > 0.4 && k < 0.5)
        { vel[i+2] = ((float)xvel); }
    else { vel[i+2] = 0; }
    // y velocity after the point where the particles get the X,Z velocities.
    vel[i+1] = (float)Math.random() * 0.5f; //because they are pushed so they ought to get some small velocity at least
}
//****************************************************************************** end of setXZ(int i)
******************************************************************************

public void createAppearance()
{
    Appearance app = new Appearance();
    //set attributes to the appearance. Size of point and transparency.
    PointAttributes pa = new PointAttributes();
    pa.setPointSize(POINTSIZE);// sets the size of the particles
    pa.setPointAntialiasingEnable(true); // turns anti-aliasing on!
    app.setPointAttributes(pa);
TransparencyAttributes ta = new TransparencyAttributes();
ta.setTransparencyMode(TransparencyAttributes.BLENDED);
ta.setTransparency(transAlpha);
app.setTransparencyAttributes(ta);
setAppearance(app);

// ------------------------ PUpdater inner class ------------------------

public class PUpdater implements GeometryUpdater {

    public void updateData(Geometry geometry) {
        // Here we check whether a particle should be updated, die or reinitialized.
        // the direction decides whether the particle's Y-velocity has dropped below 0 for the
        // first time.
        // This way we ensure that the conditions we have set up, are applied correctly.
        for (int i = 0; i < numPoints*3 ; i = i+3) {
            if (coord[i+1] <= startY && direction[i/3] == true && age[i/3] > 1) {
                age[i/3] = age[i/3] - 1;
            } else if (age[i/3] < 1) {
                setParticle(i);
            } else if (vels[i+1] <= 0f && direction[i/3] == false && coord[i+1] >= 0.2f && accs[i] == 0f && accs[i+2] == 0f) {
                setXZ(i);
                age[i/3] = age[i/3] - 1;
                direction[i/3] = true;
                updateParticles(i);
            } else if ( (vels[i+1] <= 0f && direction[i/3] == false && coord[i+1] >= 0.2f) && (accs[i] != 0f || accs[i+2] != 0f) ) {
                age[i/3] = age[i/3] - 1;
                direction[i/3] = true;
                updateParticles(i);
            } else {
                age[i/3] = age[i/3] - 1;
                updateParticles(i);
            }
        }
    }

} //******************************END of updateData(Geometry geometry)
//Here we update the position of the particle according to our laws.
//The information that is produced is fetched from the updateData().

public void updateParticles(int i)
{
    //Change of velocity and position according to acceleratred motions in physics s1 = s0 + v0*t + 1/2(a*t*t)
    // Taken from KGP, chapter 21 example classes.
    //It calculates the position of each particle every TIMESTEP.
    coord[i] += vels[i] * TIMESTEP + 0.5 * accs[i] * TIMESTEP * TIMESTEP; // x coord
    coord[i+1] += vels[i+1] * TIMESTEP + 0.5 * accs[i+1] * TIMESTEP * TIMESTEP; // y coord
    coord[i+2] += vels[i+2] * TIMESTEP + 0.5 * accs[i+2] * TIMESTEP * TIMESTEP; // z coord
    // calculate new velocities for every TIMESTEP.
    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
}
// end of updateParticles(int i)

//end of PointsUpdater inner class

// taken from the example classes used in Killer Game Programming in Java
// by Andrew Davison, chapter 21.
public class ParticlesControl extends Behavior
    // Requests an update every timedelay in ms by using the updater object.
    // It is important in order to get real-time changes to have our delay to be equal with the
    // So if every frame will be calculate with f.x. 50 ms delay, then the new velocities and position of
    // for a TIMESTEP of 50 ms.
{
    private WakeupCondition timedelay;
    private PUpdater updater;

    public ParticlesControl(int delay, PUpdater updt)
    {
        timedelay = new WakeupOnElapsedTime(delay);
        updater = updt;
    }

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

**public void** processStimulus(Enumeration criteria)
{
    pointParticles.updateData(updaterr); // request an update of the geometry
    wakeUpOn( timedelay );
}

//------------------------------- END of ParticlesControl inner class -------------------------------

}  

//'''''''''''''''''''''''''''''''' end of Particles class ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
import java.util.Enumeration;
import java.util.Hashtable;
import javax.media.j3d.AmbientLight;
import javax.media.j3d.Appearance;
import javax.media.j3d.Background;
import javax.media.j3d.Behavior;
import javax.media.j3d.BoundingSphere;
import javax.media.j3d.BranchGroup;
import javax.media.j3d.Canvas3D;
import javax.media.j3d.DirectionalLight;
import javax.media.j3d.ImageComponent2D;
import javax.media.j3d.Material;
import javax.media.j3d.PointLight;
import javax.media.j3d.Shape3D;
import javax.media.j3d.SpotLight;
import javax.media.j3d.Texture2D;
import javax.media.j3d.Transform3D;
import javax.media.j3d.TransformGroup;
import javax.media.j3d.TransparencyAttributes;
import javax.swing.JFrame;
import javax.vecmath.Color3f;
import javax.vecmath.Point3d;
import javax.vecmath.Point3f;
import javax.vecmath.Vector3d;
import javax.vecmath.Vector3f;
import com.sun.j3d.loaders.Scene;
import com.sun.j3d.loaders.objectfile.ObjectFile;
import com.sun.j3d.utils.behaviors.vp.OrbitBehavior;
import com.sun.j3d.utils.image.TextureLoader;
import com.sun.j3d.utils.universe.SimpleUniverse;
import com.sun.j3d.utils.universe.ViewingPlatform;

public class WaterFountain extends JFrame {
    public Canvas3D myCanvas3D; //The canvas to be drawn upon.
    private BoundingSphere bound;
    private static final int BOUNDSIZE = 100;
    private static final Point3d USERPOSN = new Point3d(0, 4, 4.5); // initial user position
    private SimpleUniverse simpUniv;
public WaterFountain()
{
    //Mechanism for closing the window and ending the program.
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Default settings for the viewer parameters.
    myCanvas3D = new Canvas3D(SimpleUniverse.getPreferredConfiguration());

    //Construct the SimpleUniverse:
    //First generate it using the Canvas.
    simpUniv = new SimpleUniverse(myCanvas3D);

    //Default position of the viewer.
    simpUniv.getViewer().setNominalViewingTransform();

    //The scene is generated in this method.
    createSceneGraph(simpUniv);

    //Add some light to the scene.
    addLight(simpUniv);

    initUserPosition(); // Set the user's initial viewpoint

    //The following three lines enable navigation through the scene using the mouse.
    OrbitBehavior ob = new OrbitBehavior(myCanvas3D);
    ob.setSchedulingBounds(new BoundingSphere(new Point3d(0,0,0),Double.MAX_VALUE));
    simpUniv.getViewer().setViewPlatformBehavior(ob);

    //Show the canvas/window.
    setTitle("Particles");
    setSize(700,700);
    getContentPane().add("Center", myCanvas3D);
    setVisible(true);
}

public static void setToMyDefaultAppearance(Appearance app, Color3f col)
{
    app.setMaterial(new Material(col,col,col,col,150.0f));
}

//In this method, the objects for the scene are generated and added to
//the SimpleUniverse.
public void createSceneGraph(SimpleUniverse su)
{
    //*** Generate (the root of) the scenegraph. ***
    BranchGroup theScene = new BranchGroup();
    bound = new BoundingSphere(new Point3d(0,0,0), BOUNDSIZE);

    Particles ptsFountain = new Particles(3000, 0f, 0.05f, -0.035f, 0.2f, 3.0f);
TransformGroup posnTG = new TransformGroup(); // The main transform group for the water fountain
Transform3D trans = new Transform3D(); // Transformation for the water fountain
posnTG.setTransform(trans);
posnTG.addChild(ptsFountain);
theScene.addChild(posnTG); //Adds the main transform group to the scene

// timed behaviour to animate the fountain
Behavior partBeh = ptsFountain.getParticleBeh();
partBeh.setSchedulingBounds(bound);
theScene.addChild(partBeh);

/*********************** MAKING THE POOL & WATER
*******************************************************************/
ObjectFile f = new ObjectFile(ObjectFile.RESIZE);
Scene s = null;
try {
    s = f.load("Fountaintorus3.obj");
} catch (Exception e) {
    System.out.println("File loading failed:" + e);
}
Transform3D tfObject1 = new Transform3D(); //Transformation for the loaded object.
    tfObject1.setTranslation(new Vector3f());
    Transform3D zRotation = new Transform3D();
    tfObject1.mul(zRotation);
    TransformGroup tgObject1 = new TransformGroup(tfObject1);
    //Transform group for the loaded object.
    tgObject1.addChild(s.getSceneGroup());

    //In the following way, the names of the parts of the object can be obtained. The names are printed.
    Hashtable namedObjects = s.getNamedObjects();
    Enumerationenumer = namedObjects.keys();
    String name;
    while (enumerator.hasMoreElements()) {
        name = (String) enumerator.nextElement();
        System.out.println("Name: " + name);
    }

    //************************ Pool torus
*****************************************************************/
Color3f col = new Color3f(0.3f,0.3f,0.3f); //color for the pool torus
Appearance torusApp = new Appearance();
setToMyDefaultAppearance(torusApp,col);
Shape3D partOfTheObject = (Shape3D)
namedObjects.get("null_object");
partOfTheObject.setAppearance(torusApp);
posnTG.addChild(tgObject1); // Adds the pool & water to the main transform group

/********************* Water in the pool **********************************
int tMode = TransparencyAttributes.BLENDED;
float tValue = 0.5f; // Transparency value
TransparencyAttributes ta = new
TransparencyAttributes(tMode,tValue);
Appearance textureApp1 = new Appearance();
textureApp1.setTransparencyAttributes(ta);

//Load the water image for the texture.
TextureLoader textureLoad1 = new TextureLoader("water-
surface.jpg",null);

//Generate a (scaled) image of the texture. Height and width must be
//powers of 2.
ImageComponent2D textureIm1 = textureLoad1.getScaledImage(1024, 1024);

//Generate the texture.
Texture2D texture1 = new
Texture2D(Texture2D.BASE_LEVEL,Texture2D.RGB,
    textureIm1.getWidth(),
    textureIm1.getHeight());
texture1.setImage(0,textureIm1);
textureApp1.setTexture(texture1);
Shape3D partOfTheObject1 = (Shape3D) namedObjects.get("tube");
partOfTheObject1.setAppearance(textureApp1);

WATER ******************************************
************************** END OF MAKING THE POOL & WATER ****************************

************************* MAKING THE GROUND ***************************************
ObjectFile f2 = new ObjectFile(ObjectFile.RESIZE);
Scene s2 = null;
try
{
    s2 = f2.load("terrain.obj");
}
catch (Exception e)
{
    System.out.println("File loading failed:");
}

```
Transform3D tfObject2 = new Transform3D(); //Transformation for the loaded object.

tfObject2.setTranslation(new Vector3f(0f, -0.05f, 0f));
Transform3D zScale1 = new Transform3D();
zScale1.setScale(2.5);
tfObject2.mul(zScale1);
TransformGroup tgObject2 = new TransformGroup(tfObject2);

//Transform group for the loaded object.
tgObject2.addChild(s2.getSceneGroup());

/************************************* The ground******************************************
Hashtable namedObjects1 = s2.getNamedObjects();
Enumeration enumerr = namedObjects1.keys();
String nm;
while (enumerr.hasMoreElements())
{
  nm = (String) enumerr.nextElement();
  System.out.println("Name: "+nm);
}

//Load the image for the texture.
TextureLoader textureLoad2 = new TextureLoader("Sand2M.jpg", null);

//Generate a (scaled) image of the texture. Height and width must be powers of 2.
ImageComponent2D textureIm2 = textureLoad2.getScaledImage(1024, 1024);

//Generate the texture.
Texture2D texture2 = new Texture2D(Texture2D.BASE_LEVEL, Texture2D.RGB,
  textureIm2.getWidth(),
  textureIm2.getHeight());
texture2.setImage(0, textureIm2);

//An Appearance which will use the texture.
Appearance textureApp2 = new Appearance();
textureApp2.setTexture(texture2);

Shape3D partOfTheObject2 = (Shape3D) namedObjects1.get("plane");
partOfTheObject2.setAppearance(textureApp2);

posnTG.addChild(tgObject2); // Adds the ground tg to the main transform group

************************************************************************ END OF MAKING THE GROUND************************************************************************

************************************************************************ MAKING THE TREE 1************************************************************************
ObjectFile f3 = new ObjectFile(ObjectFile.RESIZE);
Scene s3 = null;
try {
  s3 = f3.load("Tree1.obj");
} catch (Exception e) {
  System.out.println("File loading failed:" + e);
}

Transform3D tfTree1 = new Transform3D(); //Transformation for tree1
tfTree1.setTranslation(new Vector3f(2f,0.45f,-1f));
Transform3D zScale2 = new Transform3D();
zScale2.setScale(0.5);
tfTree1.mul(zScale2);
TransformGroup tgTree1 = new TransformGroup(tfTree1); //Transform group for tree1
tgTree1.addChild(s3.getSceneGroup());
posnTG.addChild(tgTree1); // Adds the tree1 tg to the main transform group

ObjectFile f4 = new ObjectFile(ObjectFile.RESIZE);
Scene s4 = null;
try {
  s4 = f4.load("treerough.obj");
} catch (Exception e) {
  System.out.println("File loading failed:" + e);
}

Transform3D tfTree2 = new Transform3D(); //Transformation for tree2
tfTree2.setTranslation(new Vector3f(-2f,0.45f,0f));
Transform3D zScale3 = new Transform3D();
zScale3.setScale(0.5);
tfTree2.mul(zScale3);
TransformGroup tgTree2 = new TransformGroup(tfTree2); //Transform group for tree2
tgTree2.addChild(s4.getSceneGroup());
posnTG.addChild(tgTree2); // Adds the tree2 tg to the main transform group
ObjectFile f5 = new ObjectFile(ObjectFile.RESIZE);
Scene s5 = null;
try {
    s5 = f5.load("Tree1.obj");
} catch (Exception e) {
    System.out.println("File loading failed:" + e);
}

Transform3D tfTree3 = new Transform3D();
    //Transformation for tree3
    tfTree3.setTranslation(new Vector3f(2f,0.45f,1f));
    Transform3D zScale4 = new Transform3D();
    zScale4.setScale(0.5);
    tfTree3.mul(zScale4);
    TransformGroup tgTree3 = new TransformGroup(tfTree3);
    tgTree3.addChild(s5.getSceneGroup());
    //Transform group for tree3

posnTG.addChild(tgTree3); // Adds the tree3 tg to the main transform group

ObjectFile f6 = new ObjectFile(ObjectFile.RESIZE);
Scene s6 = null;
try {
    s6 = f6.load("bench.obj");
} catch (Exception e) {
    System.out.println("File loading failed:" + e);
}

Transform3D tfBench = new Transform3D();
    //Transformation for the bench
    tfBench.rotY(-Math.PI/2);
    tfBench.setTranslation(new Vector3f(2f,0.1f,0f));
    Transform3D zScale5 = new Transform3D();
    zScale5.setScale(0.3);
Transform3D z6Rot = new Transform3D();
z6Rot.rotY(1);

tfBench.mul(zScale5);
TransformGroup tgBench = new TransformGroup(tfBench);

//Transform group for the bench
tgBench.addChild(s6.getSceneGroup());
posnTG.addChild(tgBench); // Adds the bench tg to the main transform group

//************************* END OF MAKING THE BENCH
****************************************************************************************
//::::::::::::::::::::::::: END OF OBJ FILES

//************* Background
*****************
************************
Background bg = new Background(new Color3f(0.8f,0.8f,0.8f));
BoundingSphere bound = new BoundingSphere(new Point3d(0.0,0.0,0.0),Double.MAX_VALUE);
bg.setApplicationBounds(bound);
theScene.addChild(bg);

su.getViewer().getView().setSceneAntialiasingEnable(true);
theScene.compile();

su.getViewer().getView().setSceneAntialiasingEnable(true);

//Add everything to the universe.
su.addBranchGraph(theScene);

} //:::::::::::::::::::::::: END OF createSceneGraph(SimpleUniverse su)

//The different light sources are added to the scene here.
public void addLight(SimpleUniverse su)
{
   BranchGroup bgLight = new BranchGroup();

   BoundingSphere bounds = new BoundingSphere(new Point3d(0.0,0.0,0.0), Double.MAX_VALUE);

   //Light no. 1: directional light.
   Color3f lightColour1 = new Color3f(0.8f, 0.8f, 0.8f);
   Vector3f lightDir1 = new Vector3f(0.0f, 4.0f, -1.0f);
   DirectionalLight light1 = new DirectionalLight(lightColour1, lightDir1);
   light1.setInfluencingBounds(bounds);
   bgLight.addChild(light1);

   //Light no. 2: a point light.
   Color3f lightColour2 = new Color3f(0.3f, 0.3f, 0.3f);
PointLight light2 = new PointLight(lightColour2, 
    new Point3f(1.0f, 1.0f, 1.0f), 
    new Point3f(0.2f, 0.01f, 0.01f));
light2.setInfluencingBounds(bounds);
bgLight.addChild(light2);

//Light no. 3: a spotlight.
Color3f lightColour3 = new Color3f(0.3f, 0.3f, 0.3f);
SpotLight light3 = new SpotLight(lightColour3, 
    new Point3f(0.0f, 0.0f, 1.0f), 
    new Point3f(0.1f, 0.1f, 0.01f), 
    new Vector3f(0.0f, 0.0f, -1.0f), 
    (float) (Math.PI/20), 
    0.0f);
light3.setInfluencingBounds(bounds);
bgLight.addChild(light3);

//Light no. 4: ambient light.
Color3f lightColour4 = new Color3f(0.3f, 0.3f, 0.3f);
AmbientLight light4 = new AmbientLight(lightColour4);
light4.setInfluencingBounds(bounds);
bgLight.addChild(light4);
su.addBranchGraph(bgLight);
}

// This method is taken from KGP, chapter 21 examples
private void initUserPosition()
// Set the user's initial viewpoint using lookAt()
{
    ViewingPlatform vp = simpUniv.getViewingPlatform();
    TransformGroup steerTG = vp.getViewPlatformTransform();
    Transform3D t3d = new Transform3D();
    steerTG.getTransform(t3d);
    steerTG.setTransform(t3d);
    // args are: viewer posn, where looking, up direction
    t3d.lookAt(USERPOSN, new Point3d(0, 0, 0), new Vector3d(0, 1, 0));
    t3d.invert();
    steerTG.setTransform(t3d);
} // end of initUserPosition()

public static void main(String[] args)
{
    WaterFountain wF = new WaterFountain();
}