WEST UNIVERSITY OF TIMIŞOARA
FACULTY OF MATHEMATICS AND INFORMATICS
DEPARTMENT OF COMPUTER SCIENCE

SIMPLE API FOR 3D APPLICATIONS

Author:
Lucian F. Ştefănoaica

Scientific Coordinator:
Lector Dr. Marc E. Frîncu

Timişoara, 2015

Contents

Abstract

Computer graphics improves our daily lives by offering computer aided design tools and means of entertainment. Particle systems are what enrich computer graphics by enabling the possibility to render fire, smoke and other interesting natural phenomena using computers. The application programming interface presented in this thesis, alongside implementation related concepts, is actually a small library packed with a couple of graphical effects based on particle systems. The thesis is composed out of four chapters and the content of each chapter is shortly described bellow.

Chapter 1 gives a brief history of computer graphics and particle systems. This chapter also specifies the role that particle systems play in computer graphics and why are they necessary.

Chapter 2 describes some technical details about particle systems. By the end of this chapter a programmer should already have an idea about how to implement such a system in a graphical application.

Chapter 3 presents all the particle based graphical effects in the API and gives implementation details about them. It also gives details about the structure of the API and that of the application which uses the API. Besides this it demonstrates the API's use with a couple of screen-shots.

Chapter 4 discusses some future work directions which should enhance the rendering performance of the API. It also gives a short description of my learning experience.

Abstract

Grafica pe calculator ne îmbunătăţeşte viaţa de zi cu zi oferindu-ne unelte pentru proiectare asistată de calculator si mijloace de divertisment. Sistemele de particule îmbogăţesc grafica pe calculator făcând posibilă redarea de fenomene naturale interesante cum ar fi focul sau fumul. Interfaţa de programare de aplicaţii prezentată in această lucrare de licenţă împreună cu concepte legate de implementare este defapt o mica librărie ce conţine câteva efecte grafice bazate pe sisteme de particule. Lucrarea este structurată in patru capitole si conţinutul fiecarui capitol este descris pe scurt mai jos.

Capitolul 1 oferă un scurt istoric al graficii pe calculator şi al sistemelor de particule. Acest capitol specifică de altfel şi rolul pe care îl joacă sistemele de particule in grafica pe calculator şi de ce sunt acestea necesare.

Capitolul 2 descrie nişte detalii tehnice legate de sisteme de particule. După ce citeşte acest capitol un programator ar trebui sa aibă deja o idee destul de bună despre cum ar trebui implementat un astfel de sistem intr-o aplicaţie grafică.

Capitolul 3 prezintă toate efectele grafice bazate pe sisteme de particule din librărie si oferă detalii legate de implementarea lor. De asemenea mai oferă si detalii legate de structura librăriei si a unei aplicaţii ce foloseşte librăria. Mai oferă şi nişte capturi de ecran ce demonstrează efectele din librărie.

Capitolul 4 discută nişte direcţii de lucru viitoare care ar trebui să îmbunătăţească performanţa de redare a librăriei. Mai oferă şi o scurtă descriere a experienţei mele de învăţare.

Chapter 1

Introduction

1.1 Brief History of Computer Graphics

What is *computer graphics*? The term first appeared in 1960 and it was made-up by William Fetter who was, at the time, a computer graphics researcher for Boeing. The term refers to a subfield of computer science which deals with image data representation and manipulation using computers.

The first computer able to do graphics is the Whirlwind computer. Its development was started in 1945 at MIT by a team of computer scientists led by Jay Forrester. The purpose of this computer was to make aircraft tracking possible on a large oscilloscope screen via a graphical application. Although the aircraft tracking application was the first application in the field of computer graphics it was not interactive. It could only display the real time positions of the tracked aircrafts. The first interactive graphical application was Tennis For Two and it was created by William Higinbotham because he wanted to kill the boredom of the visitors of the Brookhaven National Laboratory.

In 1959 the TX-2 computer emerged. This computer was used by Ivan Sutherland to program the Sketchpad, a tool for creating very precise engineering drawings. The software offered its users the possibility to draw lines and circle arcs. The lines could then be made perfectly parallel or perpendicular in order to fit the users drawing needs. Sketchpad is known to be the first graphical user interface or GUI in short form.

In 1966 Ivan Sutherland made yet another contribution to the field of computer graphics by inventing the Sword of Damocles the first computer

controlled head mounted display. This device displayed two stereoscopic images of the same wire-frame mesh. Two decades later NASA would use his methods in virtual reality research.

Very soon after, in 1970 actually, the field of computer graphics was upgraded by Henri Gouraud, Jim Blinn and Bui Tuong Phong. The first added the Gouraud shading model and the other two added the Blinn-Phong shading model. In 1978 Jim Blinn also added bump mapping to the computer graphics field.

1.2 Brief History of Particle Systems

The term **particle systems** was coined in 1982 by William T. Reeves who was a computer graphics researcher for Lucasfilm Ltd., a film production company known for films like the Star Trek and Star Wars franchises. But what is a particle system? Here's the original definition:

"A particle system is a collection of many minute particles that together represent a fuzzy object. Over a period of time, particles are generated into a system, move and change from within the system, and die from the system."

–William T. Reeves–acm Transactions On Graphics–April 1983–Vol. 2, No. 2

At Lucasfilm Ltd. Reeves helped to develop the wall of fire effect which occurred every time the Genesis Device was used in the film Star Trek II: The Wrath of Khan. Though this is not the first time when particle systems have been used in the computer industry, this particular effect helped to coin down the term **particle systems**.

The first particle systems were implemented in the earliest video games where this concept was used to portray exploding spaceships which left behind many little dots or lines. For example "Spacewar!" which appeared in 1962 and "Asteroid" which appeared later in 1978 had such effects implemented. Since their dawn till the very present particle systems have been used to implement various interesting phenomena such as fire, smoke or waterfalls. In the absence of particle systems these phenomena would be very difficult to implement if not impossible.

1.3 The Necessity of Particle Systems

An alternate solution to the problem of modeling objects like stormy clouds, explosions or smoke would be to use surface-based representations (the ones which are used to model objects like cars or spaceships). It is easy to notice though, that it would not be a very natural approach of the problem to try to model an object which has no smooth surfaces, perhaps a cloud of smoke, with a series of triangles where each triangle represents a surface which actually does not exist on the real object. One could easily implement such a smoke cloud using triangles to represent its margins but the results would not be very realistic. If a large number of very small triangles were to be used then the smoke cloud would look better, in fact the smaller the triangles and the larger the count of the triangles the better the quality of the image. But there is a downside to this approach.

Let us imagine what would happen to the previously mentioned smoke cloud if a sudden wind gust were to hit it. The smoke cloud would most certainly move in the direction of the wind. Only with the surface-based representation it would move in the direction of the wind as a whole just like a very large balloon would. The representation of the smoke cloud would not be very realistic because in real life when a smoke cloud gets hit by a lateral wind force some pieces of it go with the wind while others go in completely different directions or move slightly from their initial position. Those pieces of the smoke cloud are actually *particles* of smoke. So if particle systems are used to "implement" smoke clouds in real life, why not use this concept as well, in the implementation of a smoke cloud which would only exist inside a graphical application? Instead of representing only the surface of the smoke cloud using medium to large sized triangles one should represent the whole volume of the smoke cloud using many small sized triangles. This way, not only the visual aspects of the smoke cloud would be improved but also a great amount of flexibility would be added to it. This added flexibility will make the object behave realistically, in other words it will make the smoke cloud react like a real smoke cloud would react not like a balloon which is the case of the surface-based representation. This added flexibility represents the advantage of particle systems over surface-based representations.

1.4 Reasons To Study Particle Systems

Particle systems represent a fairly important role in computer graphics because without them the problem of rendering natural phenomena like fire, smoke, waterfalls, fog and others would not be solved correctly. There may be other techniques which can be used to treat this problem out there, but one should reckon that using particle systems is the most natural approach to this problem.

I have always been curious of what stands behind those nice fiery graphical effects which appear in most computer games. So naturally, I wanted to learn about them. Programming particle systems represents a good way to learn graphics programming, and not only that. Particle systems are a rather complex concept because their implementation requires notions of mathematics and physics besides the obvious notions of computer science. One of the reasons why I accepted to study this concept is because I wanted to acquire experience in interdisciplinary areas and consolidate my computer graphics knowledge. With this experience I can further improve my professional career by writing graphic applications for both desktop and mobile devices.

Typically most programmers use libraries. Why? Because it is faster. Most problems, large or small, can be divided into subproblems. If one divides a larger problem into a set of ten smaller subproblems he would then have to gradually solve each and every one of the subproblems in order to solve the larger problem. Let us say that every subproblem in the set takes about two to three days to solve, this means that the whole set of subproblems should take, in the worst case, about a month to solve. What if somebody else has already solved half of the subproblems in the set and has packaged the solutions in a library? Then that library would make any programmer who would face the initial larger problem twice as productive because that somebody else has already done half of the work. Java OpenGL (JOGL) is such a library. It offers all of the high level OpenGL function calls necessary to write graphic applications in Java. Although this library is highly comprehensive it does not offer easy ways to implement particle system based graphical effects meaning that, if someone would want to implement a fiery particle system he would have to do so from scratch. So another reason why I accepted to write the simple graphical API presented in this thesis is to extend JOGL. The simple graphical API is completely dependent on JOGL and it offers some simple function calls which will render particle systems of different shapes, sizes, colors or textures.

1.5 Thesis Structure

The thesis is structured in four chapters and the content of each chapter is shortly described bellow.

Chapter 1 gives a brief history of computer graphics and particle systems. After that, this chapter specifies the necessity of particle systems and the role that is played by them in computer graphics. This chapter also presents some reasons why I chose this topic.

Chapter 2 offers technical details about particle systems. It talks about how to implement such systems and how to organize the code in order to achieve the desired results. By the end of this chapter a programmer should already have an idea about how to implement a particle system in a graphic application.

Chapter 3 presents all the particle based graphical effects in the API and gives implementation details about them. It also gives details about the structure of the API and that of the application which uses the API. Besides this it demonstrates the API's use with a couple of screen-shots.

Chapter 4 discusses some future work directions which should enhance the rendering performance of the API. Work directions regarding new effects which can be added to the API later are also mentioned. Besides this it gives a short description of my learning experience.

Chapter 2

Particle Systems

2.1 The Basic Structure of Particle Systems

From an object-oriented programming point of view a **particle system** is a simple container object which manages all the **particles** inside it. At first the container is completely empty. The container then starts adding a constant or variable number of particles per frame to itself. This container has an interface which can be used by programmers to control the particles inside it and also some properties of the container itself. But how does one design and implement such a container?

Here are three short steps which can be carried out in order to obtain an object-oriented model of the problem:

1. In order to highlight some of the base components of an object-oriented system write down a detailed description of the problem which the system is supposed to solve.

2. Iterate through the description and identify all the nouns, adjectives and verbs. The nouns represent potential class names, the verbs represent potential class attributes and the verbs represent potential class methods.

3. Assign attributes and methods to classes.

Following is a description of a particle based graphical effect.

Let us assume that we are asked to implement multiple particle effects using particle systems. Some particles may react to external forces in a different manner than others. For example some particles can be affected by gravity while others can not. Other particles can have a very high mass making their trajectory only slightly modifiable by an external wind force. Every particle should have a trajectory weather it is influenced by an external force or not.

Some particle systems may emit more than one type of particles. For example a particle system which simulates the explosion of a nuclear bomb should emit both lightweight and heavy particles. The heavy particles will fall down to the ground while the lightweight particles will form a cloud of fire, smoke and dust. The fire will burn out but the smoke and dust will get carried away by external wind forces if they exist.

A few basic components can be easily identified in the above problem description. This components are actually classes and they can be named as follows:

- Particle
- FireParticle
- SmokeParticle
- DustParticle
- ParticleSystem
- FireSystem
- SmokeSystem
- DustSystem

No matter how high is the difference between a particle system and another one there will still be state and behavior which is common to all particle systems. The same statement is valid for the Particle classes as well. The Particle class should be a class which the other particle classes (FireParticle, SmokeParticle, etc...) can use to inherit common functionality from. The ParticleSystem class should be the class which holds the functionality common to all the particle system classes (FireSystem, SmokeSystem, etc...). But this is just a simple class structure which can be used to model particle

systems. What is the actual mechanism which makes particle systems work? These steps are executed whenever most particle systems get rendered on a screen:

1. New particles are created in order to be added to the particle system which is initially completely empty.
2. All the attributes of the newly created particles are initialized.
3. The list of particles contained by the particle systems is scanned for dead particles in order to remove them from the list.
4. All the particles which still exist in the system will be moved to their new position. Each new position is computed with respect to the old position of the particle and attributes like direction, speed and acceleration.
5. Finally an image of the particle system is rendered on the screen.

Each step is described in a more detailed manner in the following subsections.

2.1.1 The Particle Spawning Step

This is the first step in the rendering process of a particle system. This is when new particles are born. The number of particles generated per frame can contribute to the visual quality of a particle system. For example a real burning flame will not always have the same intensity meaning that it will not generate a constant number of fire particles. Using some kind of stochastic process to compute the number of particles generated per frame may make a simulated flame look more real, but for simplicity one can easily set the number of particles generated per frame to be constant.

One good example of a stochastic method which can be used to compute the number of particles generated per frame is to use a mean and a variance counter. The mean counter represents the average number of particles generated throughout a sequence of frames and the variance counter represents the largest possible difference between the average and the actual number of particles generated per one frame. The following formula shows exactly how this is supposed to work:

$$n = m + r \cdot v,$$

where n is the number of particles generated per frame, m is the mean, v is the variance and r is a random number between -1 and 1.

Depending on the number of particles generated per frame, the memory consumption of particle systems can be quite high. Fortunately there is a method which can sometimes reduce their memory consumption. This method requires the inclusion of the screen area covered by the particle system in the equation. In other words if a particle system is very far away from the camera then it will appear to be very small thus a smaller number of particles will not affect the quality of the rendered image. There is no need to use $100,000$ particles to render a particle system which takes only 1 to 2 squared centimeters of the screen. The formula looks as follows:

$$n = (m + r \cdot v) \cdot s,$$

where n is the number of particles generated, m is the average number of particles generated over a period of time, v is the variance, r is a random number between -1 and 1 and s represents the screen area covered by the particle system.

2.1.2 The Particle Initialization Step

While from a physics point of view a particle can be anything from a photon to a few water molecules or a even star, from an object oriented programming point of view a particle is a simple object which has a state and a behavior. The attributes of a particle represent its state and the methods of that particle represent a way to control its state. In this step all the particle attributes are initialized. Particles can have a very wide variety of different attributes but the most basic ones are listed bellow.

- position
- speed
- acceleration
- size
- color
- lifetime
- texture

The position attribute should be implemented as a two-dimensional or a three-dimensional vector variable depending on the coordinate system. This variable is usually initialized with the origin of the particle system. In order to move a particle to a new position one would have to execute various mathematical vector operations on this variable.

The speed attribute should also be implemented as a vector and it should be used together with the acceleration attribute in order to modify a particle's position. This formula shows how to do this:

$$\vec{n} = \vec{o} + (\vec{s} + \vec{a}),$$

where $\vec{n}$ is the new position vector, $\vec{o}$ is the old position vector, $\vec{s}$ is the speed vector and $\vec{a}$ is the acceleration vector. The magnitude of the speed vector can remain constant throughout the whole lifetime of a particle while the magnitude of its acceleration vector can get random values from a predefined range in order to give the particle a more complex movement.

From a conceptual point of view a particle is a simple point in space. The downside to implementing particles as points is that points can not be

textured. This is why particles are usually implemented with billboards. Billboards offer the possibility of obtaining textured particle effects. In the case of a square billboard the size attribute is actually the distance between the center of the billboard and one of its edges whereas in the case of a point implementation the size attribute represents the surface area covered by the point on the screen.

The color attribute can be used to control the color tone of a particle. In other words even if a particle has a texture with blue accents by changing this attribute one can make its texture to have reddish accents.

The lifetime attribute is used to decide when a particle is dead and needs to be removed from the particle system but this attribute can also be used to control a particle's opacity. A particle's opacity usually has values between 0.0 and 1.0. When a particle's opacity is at it's lowest bound the particle is completely transparent and when its opacity is at its highest bound the particle is completely opaque. If we make the particle's lifetime start with 1.0 and decrease at each frame with a constant or variable amount we can control both the duration and the transparency of a particle with a single variable.

The texture attribute controls the texture of a particle and it can be implemented with a single texture variable or with a texture array variable. When it is implemented with an array a particle instance will be more dynamic because it will have multiple different textures throughout its lifetime.

2.1.3 The Dead Particle Removal Step

In this step the list of particles managed by the particle system is scanned for dead particles. Whenever a dead particle is found it is quickly removed from the system. The scanning can be done starting from the beginning of the list or from the end.

When the scan is done starting from the beginning of the list a small glitch occurs at each pass. Whenever a particle is removed from the list, all the following particles in the list are moved to left with one position, this causes the pass to skip the verification of one particle. The skipped particle will be verified at the next pass but not having a particle removed when it is expected for it to be removed is not what we want.

In order to avoid this glitch one should iterate through the list starting from the end of the list. This way when a particle is removed the subsequent particles are moved in front of the verification index not in the back of it.

There may be times when we are unable to iterate backwards through a list. This is where **Iterator**s come in handy. Most programming languages (Java is one of these languages) offer these objects. When one uses an Iterator to scan a list of objects he does not need to worry about glitches like skipping an object because this matter is handled by the Iterator itself.

What are the conditions which need to be satisfied by a particle in order for it to be dead? One method of determining this is to keep track of a lifetime value which is supposed to drop with each rendered frame.

The distance between a particle's current position and the origin of the particle system can also be used to determine when a particle leaves a certain region surrounding the origin of the system. If that distance exceeds a certain value, the particle will be removed from the particle system.

2.1.4 The Particle Update Step

In this step all the particles from a particle system are updated. Usually a particle update means moving it to a new position in space. But other particle attributes can also be affected by the update. Making the update affect other attributes will make the particle more dynamic.

The texture attribute is one of the attributes which can be updated. When it comes to texturing a particle one can use a single texture or a texture atlas (a whole range of textures). The texture update may mean to change the current texture with a new texture from the texture atlas. Thus giving a fire particle effect a more realistic look because in real life fire particles have variable shapes and sizes.

Let us assume that one wants to implement a particle effect that resembles a rainbow. This means that a particle from that particular particle system must morph through all the colors of the rainbow. This change of color can and should be done here in the update step.

Another particle attribute that can be updated is the size of the particle. Updating the size of a particle means to make it grow or shrink throughout its lifetime. So a particle may start really small at the origin of the particle system and become really large by the time it dies or vice versa.

2.1.5 The Particle Display Step

A particle can be implemented in more than one way. For example the particles which formed the Genesis Effect from Star Trek II: The Wrath of Khan were implemented with light sources. The light sources combined themselves with each other in an additive manner regarding their color and transparency values. This technique had the advantage of removing the hidden surface problem because the particles did not obscure each other but simply added more light to a certain pixel.

A second method would be to use billboards to represent particles. A simple rectangle made out of two simple triangles can be a billboard. In fact any two-dimensional element rendered in a there-dimensional environment can be a billboard. The main difference between billboards and simple triangles is that a billboard always rotates itself around one or more axes in order to face the camera. There are advantages and disadvantages to using billboards instead of light sources.

One advantage is that billboards can be textured with more than one texture over their lifetime. A disadvantage is that billboards may need to be decreasingly sorted according to their distance from the camera. In other words the billboards which are further from the camera need to be rendered before the billboards which are closer to the camera otherwise the blending effect will be spoiled.

2.2 Billboards

Particles can be displayed as simple colored points or as fully textured billboards. The obvious advantage of billboards is that they can be textured offering programmers the possibility to implement a very wide variety of textured particle based graphical effects.

A billboard is usually implemented with a quad made out of two triangles welded together because graphic cards render triangles faster. So a particle can be a simple textured quad. Only there is a problem with plain old quads.

Let us assume that a graphics programmer implements a graphic application which relies on particle systems and offers its users the possibility to change the camera position at will. If our programmer uses simple quads to render the particles the user can move the camera to positions from where he would only see a part of a specific particle or no particles at all (this can happen only when the back-face culling option is enabled and the camera is placed exactly in the back of the particles).

Imagine that we would like to see the particle effects from any position possible so having them visible only from some positions is unacceptable. We could disable the back-face culling option but then we would encounter other problems.

Firstly we would have to texture all particles on both sides which, besides the performance downsides, can be a little difficult. The performance downsides of this approach are memory consumption and more calls to the underlying graphics API.

And secondly there will still be positions from where the user will not be able to see the particles correctly. A particle watched from the sides will look like a vertical line. This might not be the desired result for most of us so this approach is a no-go. However there is an advantage to this approach, one can use two different textures for each of the sides of the particle. This might be useful at some point.

The simplest solution to this problem is to make the quads rotate in such a manner that they will always face the camera. This rotation is what differentiates a billboard from a simple quad and fortunately, there are a few techniques which can be used to achieve it.

When it comes to billboard rotations the most basic and most used are the cylindrical and the spherical rotations. One can invent other different billboard rotation types according to his or her needs but the two basic rotation types solve most of the problems related to billboards.

The rotations of a billboard can be towards any object or point in the scene not just towards the camera. However rotating a billboard towards a certain point is not the only option. There is a technique which can be used to fake true billboarding.

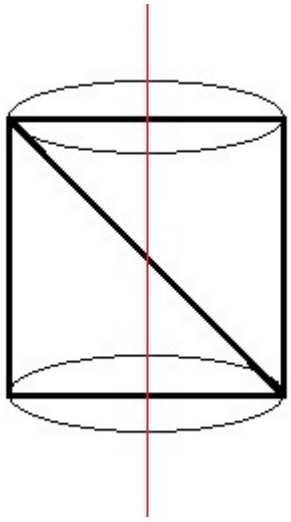
Let us take the simple case when a billboard has to rotate itself towards the camera. Instead of making it rotate exactly towards the point where the camera lies we can make it rotate towards a plane perpendicular to the cameras looking direction vector. This technique is sometimes called fake billboarding because it is an approximation of what a true billboard does. Tough it is just an approximation it might be perfect for some applications and sometimes easier to implement.

The two basic billboard rotation mechanisms (the cylindrical and spherical rotation types) do true billboarding, meaning that both mechanisms make the quad rotate itself exactly towards the point where the camera lies.

2.2.1 The Cylindrical Rotation Mechanism

When using this mechanism the rotation of the billboard is restricted by one axis. Figure 2.1 shows a billboard composed from two triangles. The red vertical line which intersects it represents its rotation axis.

Figure 2.1: Rotation axis of a billboard



Notice that the billboard is a vertical slice of a cylinder. The slice goes exactly through the central axis of the cylinder. Actually the rotation axis of the billboard and the axis of the cylinder are one and the same.

If we would rotate the billboard from Figure 2.1 fast enough it would appear to look exactly like a cylinder. This is why the billboards which are implemented with this techniques are sometimes referred to as cylindric billboards.

2.2.2 How To Implement Particles as Quads

Implementing a particle as a simple quad is fairly easy. Let us start with the center of the particle. This point is supposed to be used to control the movement of the particle. When one changes the coordinates of this point the coordinates of the corners of the particle will also change. This is because the values of the corner coordinates depend on the value of the particle center coordinates.

With these coordinates one can compute the corner coordinates of the particle but he can also move the particle by changing this coordinate.

2.2.3 Other Applicabilities of Billboards

Although billboards are useful for implementing textured particle systems they can also be used to reduce the triangle count of other three-dimensional objects.

Let us take as an example a tree. If a programmer were to render a tree using triangles he would have to face a trade-off between detail and rendering speed. In other words if he would use a small number of triangles the rendering speed will be high but the detail level of the tree will be fairly low thus making the tree look unrealistic.

On the other hand if our programmer were to use a very high triangle count the detail level of the tree will be high but the rendering speed would be lower. Actually the difference will not be that noticeable when rendering a single tree but when one renders a whole forest the difference will be greater.

Fortunately we have modern hardware which can handle rendering scenes with a high polygon count very well. But for the times when we do not have modern hardware or we simply want to optimize the rendering speed of a graphic application for no reason we have **billboards**.

Billboards can be easily used to render a forest. In the rendering process of the forest we can use a billboard (one made out of two triangles not a few thousands) for each tree in the forest. This will increase the rendering speed of the graphic application without loosing that much on the detail side of the application.

Billboards are perfect for the task of reducing the polygon count of graphic applications because a tree can be represented with only two triangles and they do not break the tree illusion because they always rotate their face towards the camera.

Chapter 3

Application

Particle systems can have many different movement mechanisms and the best way to truly understand them is through design and implementation. This is what I did, I designed four different particle systems and packaged their implementations in a simple API for later use. This API (API is the acronym for "Aplication Programming Interface") is designed with extensibility in mind meaning that new particle based graphical effects can be easily added to the next versions of the API.

The following particle based graphical effects are offered by the current version of the API:

- the conic flame effect.
- the cylindrical flame effect.
- the conic flame effect (reversed).
- the fountain effect.

In order to obtain the conic flame effect one must use a particle system in which all the particles are spawned exactly at the tip of an imaginary cone and move towards the base of it.

To obtain the cylindrical flame effect one must use a particle system in which all the particles are spawned at one end of an imaginary cylinder and move towards the other end of it.

The reversed conic flame effect is basically a conic flame effect in which the particles are spawned at the base of an imaginary cone and move towards the tip of it.

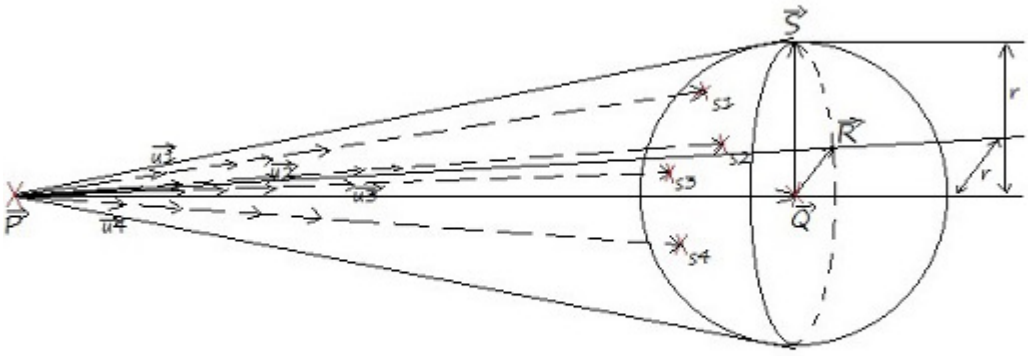
The fountain effect is based on the conic flame effect meaning that the particle system behind it has a movement mechanism which resembles the movement mechanism of the particle system behind the conic flame effect. The difference is that the particles from the particle system which does the fountain effect are affected by gravity. In other words they move in the shape of a cone but when the particles reach the base of the cone their trajectory gets curved by a gravity force.

The following sections will present and explain the design sketches of the four particle based graphical effects.

3.1 The Cone Shaped Particle System

In order for the particle system to look like a cone (it will look roughly like a cone, it will actually have the resemblance of a sprayed flame) all the particles must be spawned exactly at the tip of an imaginary cone and must move towards random points which reside inside an imaginary sphere (the points can also lie on the sphere) which is placed exactly at the base of the cone.

Figure 3.1: Cone shaped particle system sketch



The sketch from Figure 3.1 shows how to conceive a cone shaped particle system. All the descriptions of the notations from the sketch are given bellow.

- $\vec{P}$ represents the initial position vector, all the particles will have their position attribute initialized with this vector.
- $\vec{Q}$ represents the position vector of the center of the sphere. This vector is used to compute the destination position vectors which will lie inside or on the sphere.
- $\vec{s_1}, \vec{s_2}, \vec{s_3}$ and $\vec{s_4}$ represent the destination position vectors which lie inside or on the sphere.

- $\vec{u_1}, \vec{u_2}, \vec{u_3}$ and $\vec{u_4}$ represent series of direction vectors. These vectors are used to move each particle from the system in the particle update step. These vectors can be obtained by either normalizing the destination position vectors ($\vec{s_1}, \vec{s_2}$, ...) or by dividing them with a constant meant to scale them down.

- r represents the size of the radius of the sphere. This radius is also the radius of the circle which forms the base of the cone. The value of this radius is used to compute the destination position vectors ($\vec{s_1}, \vec{s_2}$, ...).

For every frame that gets rendered on the screen the following steps are executed:

1. Generate a number of particles.

2. Initialize their position attributes with $\vec{P}$ (the position vector which points to the tip of the cone)

3. Generate a set $S = \{\vec{s_1}, \vec{s_2}, \vec{s_3}, \vec{s_4}, ...\}$ of random destination position vectors for each spawned particle which will lie inside or on the sphere with the center in $\vec{Q}$.

4. Use all the previously generated position vectors (from the set S) to generate a set $U = \{\vec{u_1}, \vec{u_2}, \vec{u_3}, \vec{u_4}, ...\}$ of direction vectors for each spawned particle.

5. Update the position attribute of each particle from the particle system by simply adding their corresponding direction vector (each particle will have its own $\vec{u}$ from the previously generated set U) to their current position vector.

6. Remove all the "dead" particles from the particle system.

7. Render the current image of the particle system.

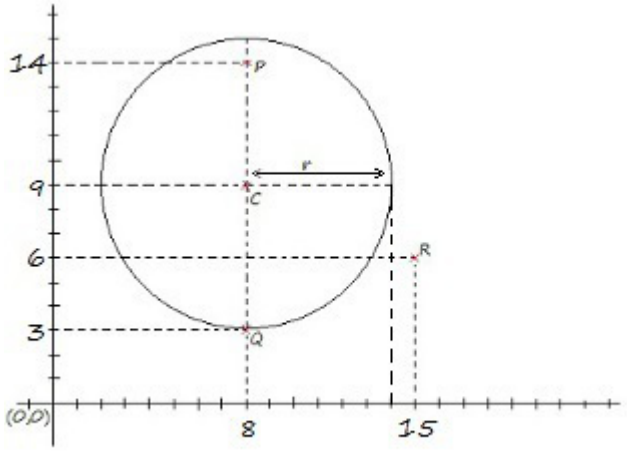
The greatest problem in the rendering process of a cone shaped particle system is how to generate destination position vectors that lie inside or on the surface of an sphere.

The equation of a circle is:

$$(x-a)^2 + (y-b)^2 = r^2,$$

where a and b represent the two-dimensional coordinates of the center of the circle, r represents the radius of the circle and x and y represent the coordinates of all the points on the circle.

Figure 3.2: Testing the borders of a circle



For example the equation of the circle from Figure 3.2 which has its center positioned in $C\big(8,9\big)$ and its radius $r = 6$ will look like this:

$$(x-8)^2 + (y-9)^2 = 36$$

Our concern here is how to generate points that are inside or on the circle with the previously mentioned equation. What we can do is to take three random points, one inside the circle, one exactly on the circle and one outside the circle substitute their coordinates in the equation and see how they verify the equation. Let us take for example $P(8, 14)$, $Q(8, 3)$ and $R(15, 9)$. As you can notice from Figure 3.2 P is inside the circle, Q is exactly on the circle and R is outside the circle.

For $P(8, 14)$ we get:

$$\begin{aligned}(8-8)^2 + (14-9)^2 &= 36\\ 0^2 + 5^2 &= 36\\ 25 &= 36\ (F)\end{aligned}$$

For $Q(8, 3)$ we get:

$$\begin{aligned}(8-8)^2 + (3-9)^2 &= 36\\ 0^2 + (-6)^2 &= 36\\ 36 &= 36\ (T)\end{aligned}$$

And for $R(15, 9)$ we get:

$$\begin{aligned}(15-8)^2 + (9-9)^2 &= 36\\ 7^2 + 0^2 &= 36\\ 49 &= 36\ (F)\end{aligned}$$

As you can see in the case of $P(8, 14)$ (the point inside the circle) the left side of the equation is smaller than the right side. In the case of $Q(8, 3)$ (the point exactly on the circle) the left side of the equation is equal with the right side and in the case of $R(15, 9)$ (the point outside the circle) the left side of the equation is greater than the right side.

From this we can deduce that in order for a point to be inside or on a circle its coordinates must verify the following in-equation:

$$(x_{point} - x_{circle})^2 + (y_{point} - y_{circle})^2 \leq r^2,$$

where x_{point}, y_{point} are the coordinates of the point, x_{circle}, y_{circle} are the coordinates of the center of the circle and r is the radius of the circle.

If we apply a square root to the both sides of the in-equation we obtain the following:

$$\sqrt{(x_{point} - x_{circle})^2 + (y_{point} - y_{circle})^2} \leq r$$

We can observe that the left side of the in-equation is actually the formula for computing the distance between two points in space. What this in-equation actually says is that the distance between the center of the circle and any point inside or on the circle must be at most equal with the radius of the circle. This means that we can use the following routine to generate every point inside or on the circle:

1. Generate a pair of random values, let us call them a and b, from the interval $\big[-r, r\big]$.
2. Compute the coordinates of the randomly generated point (which will never be outside the circle) like this:
   $(x_{point}, y_{point}) = (x_{circle}, y_{circle}) + (a, b)$

I used a similar mechanism in the generation of destination position vectors for the cone shaped particle system. Only for the cone shaped particle system I used a sphere instead of a circle. I adapted the previously mentioned routine in order to fit the following in-equation:

$$\sqrt{(x_{point} - x_{sphere})^2 + (y_{point} - y_{sphere})^2 + (z_{point} - z_{sphere})^2} \leq r$$

3.2 The Cylindrical Flame Effect

This effect can be achieved through the use of a particle system which spawns particles inside one imaginary sphere and moves them towards another imaginary sphere. The rapidly moving particles will form the cylindrical flame effect.

Chapter 4

Conclusion

Test...