CENG477
PROGRESS REPORT

PARTICLE SYSTEMS USING OPENGL
AND APPLICATION AREAS

COURSE INSTRUCTOR: VOLKAN ATALAY
COURSE ASSISTANT: ÖMER ÖNDER TOLA
GROUP NAME: PORTAKAL
GROUP MEMBERS: ELİF BAHAR ERDOĞAN 1202944
                NECİBE NUR KOÇ 1250463
TABLE OF CONTENTS

1 INTRODUCTION .............................................................................................................. 3
2 PARTICLE SYSTEMS ........................................................................................................ 3
   2.1 SIMULATION OF BASIC PARTICLE SYSTEMS ..................................................... 5
   2.2 REQUIREMENTS AND PERFORMANCE SPECIFICATIONS ............................. 6
   2.3 PARTICLES ............................................................................................................... 6
   2.4 EMITTERS ................................................................................................................ 7
   2.5 FORCES .................................................................................................................... 10
3 MANAGING VARIOUS PARTICLE SYSTEMS ........................................................... 11
4 THE ADVANTAGES OF PARTICLE SYSTEMS ....................................................... 12
5 THE DISADVANTAGES PARTICLE SYSTEMS ........................................................ 12
6 EXAMPLES FOR PARTICLE SYSTEMS .................................................................... 13
   6.1 USING PARTICLE SYSTEMS TO MODEL FIRE AND EXPLOSIONS .......... 13
   6.2 USING PARTICLE SYSTEMS TO MODEL TREES ............................................. 14
   6.3 USING PARTICLE SYSTEMS TO MODEL FLOCKING BIRDS ....................... 15
7 CONCLUSION ................................................................................................................ 16
8 REFERENCES ............................................................................................................... 16
1 INTRODUCTION

Generating real-looking images or scenes is one of the main motivates of Computer Graphics. In order to model and render the real scenes, one has to be aware of real life natural actions. Rains, snow, fog, sun, reflection of light, are the first phenomena’s that come to mind first. These physical phenomena’s can be modeled by particle systems. In this report we will basically cover the details of particle systems and their examples. You can also see our fundamental class types and class functions.

2 PARTICLE SYSTEMS

Some objects are difficult to represent as a set of primitives, even taking advantage of transparency and texture mapping techniques. These include objects that have poorly defined or dynamic topologies, or have no solid surface. They are created in some kind of randomization and behaviors of these objects basically depend on some physical rules of the environment that they are present. In terminology, a particle system is a collection of many small particles behaving in same fashion that model an object in computer graphics or a particle system is a large set of simple primitive objects which are processed as a group to represent an object.

Particle systems first came about in 1983, when William T. Reeves published his paper "Particle Systems - A Technique for Modeling a Class of Fuzzy Objects." [1]. Particles were first used in the modeling of fire, in the Genesis sequence from the film Star Trek II: The Wrath of Khan. In the film the particles are represented as independent volumes of light moving and changing in three-dimensional space. Particles are some kind of objects that behaves in same fashion. For this reason, every group of particles has same kind of properties. These properties can be listed as:

I. Shape
   II. Velocity (speed and direction)
   III. Acceleration
   IV. Color
   V. Transparency
   VI. Size
   VII. Position
   VIII. Lifetime

These properties are generally same in groups. Shape and size are the main properties of a particle. A specific particle can be represented in some kind of several primitives like points, lines, rectangles, spheres… and size of a group of particles has to be determined in some random fashion. The aim lies under this idea is to give more realistic scenes. For example, snow particles are all considered to have different shapes and sizes in
nature. For that reason designers and programmers use this following approach to set the properties of the particles.

\[
\text{ParticleAttribute} = \text{AttributeMean} + \text{Rand()} \times \text{AttributeVariance}
\]

Velocity is the main decider that considers how long a particle moves between two consecutive frames. Acceleration and velocity is responsible for the motion of the particle. Transparency component can be used for special purposes in some applications. Position component is directly related to the source that particles born. Source can be any geometric 2D object like a point, a rectangle, a circle, a triangle… Point sources can be used to implement foundations and taps. Rectangle sources are used in weather effect applications. Lifetime is another important property determining when the particle is going to be destroyed.

In order to represent the effects of external sources on particles, one has to update these particles in every frame. Once you create a group or groups of particles, then you can create whatever you want by just changing these attributes of the particles. Block diagram of the particle system is shown below.

![Block diagram](image_url)

**Block diagram**

There are many applications including natural phenomena like rain, snow, running water, smoke, fire, explosions etc. In rendering rain and snow this particle system is extensively used. Raindrops and snow particles are designed as group of particles and by arranging attributes of these particles, one can easily generate real-looking snow and rain effects.

In rest of the particle systems we are going to talk about simulation of particle systems, particle system’s advantages and disadvantages and some examples (how they are done with particle systems).
2.1 SIMULATION OF BASIC PARTICLE SYSTEMS

A particle system is basically just a collection of 3D points in space. Unlike spheres, cubes, etc., particles belonging to the system are not static. This means they are dynamic. They go through a complete life cycle. Particles are born, change over time, and then die off. By adjusting the parameters that influence this cycle, we can create different types of effects.

Another key point regarding particle systems is that they are chaotic. That is, instead of having a complete predetermined path, each particle can have a random element that modifies its behavior. It is this random element that makes the effect look very natural.

Here you see the way of implementing the particle systems:

Particle Systems

```
Large Number of Elements
↓
Simplify Assumptions for Rendering and Motion calculations
↓
• No particle-particle collisions
• No shadow casting (except in aggregate sense)
• Shadow casting on the environment
  • No light reflection
  (each particle modeled as point light sources)
```

A particle system is mainly composed of particles, emitters, and forces that act on it. The particle system is connected to an Ordinary Differential Equation (ODE) Solver.

Before we examine each of these subjects in detail, we should consider the requirements and performance specifications.
2.2 REQUIREMENTS AND PERFORMANCE SPECIFICATIONS

When designing a particle system, one of the first things to keep in mind is that particle systems greatly increase the number of visible polygons per frame. Each particle probably needs four vertices and two triangles. Thus, with 2,000 visible snowflake particles in a scene, we are adding 4,000 visible triangles for the snow alone. And since most particles move, we cannot pre-calculate the vertex buffer, so the vertex buffers will probably need to be changed every frame.

To enable this, we should try to perform as few memory operations as possible. Thus, if a particle dies after some period of time, we do not delete it from the memory. Instead, we set a flag that marks it as dead. When all particles are tagged as “dead”, we delete the entire particle system. If we want to reconstruct the system or just add a new particle to a system, we should automatically initialize the particle with its default settings/properties set up according to the system to which it belongs.

Each particle system behaves in a unique manner. For example, blood splats are rendered differently than smoke is displayed. Smoke particles always face the active camera, whereas blood splats are mapped (and maybe clipped) onto the plane of the polygon that the splat collides with.

When creating a particle system, it is important to consider all of the possible parameters that we may want to affect in the system at any time, and build that flexibility into our system. If we consider a smoke system, we might want to change the wind direction vector so that a car moving closely past a smoke system makes the smoke particles respond to the wind generated by the passing car.

At this point we may have realized that each of these systems (blood splat, smoke, sparks, etc.) is very specific to certain tasks. But what if we want to control the particles within a system in a way not supported by the formulae in the system? To support that kind of flexibility, we need to create a "manual" particle system as well; one that allows us to update all particle attributes every frame.

The last feature we might consider is the ability to link particle systems within the hierarchy of our engine. Perhaps at some point we'll want to link different dynamic systems in one frame. We will consider these kinds of cases in the second part of the report.

2.3 PARTICLES

Let’s begin with the particles and their features. It is obvious that we need to know the position of the particle. Storing the previous position of the particle can also be a good idea since we need to make antialiasing. We need to know the direction in which the particle is currently traveling. Current speed is also another important feature. Previous color can also be stored as well as current color in the case of antialiasing. Mass can be an important feature, for example, if we want to see the effect of gravitational force at each particle separately. The last information we need is the life count. This is briefly the number of frames that the corresponding particle will exist before dying.
These are just some fundamental attributes of the particle. If it is required, it would be very easy to add new attributes such as transparency.

Considering this information, we can construct the data structure for the particle such as

```c
struct tParticle
{
    tParticle *prev,*next; // LINK
    tVector pos; // CURRENT POSITION VECTOR
    tVector prevPos; // PREVIOUS POSITION VECTOR
    tVector dir; // CURRENT DIRECTION WITH SPEED
    int life; // HOW LONG IT WILL LAST
    float m; // MASS
    tColor color; // CURRENT COLOR OF PARTICLE
};
```

Here tVector is another data type, which is constructed to hold the position information as well as tColor is the vector constructed to hold color information.

### 2.4 Emitters

The particle emitter is the entity, which is responsible for creating the particles in the system. The emitter controls the number of particles and general direction in which they should be emitted as well as all the other global settings. Stochastic processes are also set here. For example, emitNumber is the average number of particles that should be emitted each frame. The emitVariance is the random number of particles either added or subtracted from base emitNumber. By adjusting these two values, we can change the effect from a constant to a more random flow. The formula for calculating how many particles to emit each frame is simply:

\[
\text{ParticleCount} = \text{emitNumber} + (\text{emitVariance} \times \text{Random\_number})
\]

where Random\_number is between –1.0 and 1.0.

This technique can also be used to vary the color, direction, speed and the life span of a particle.

Now, we should consider the directions in which the particles should be emitted. Two angles of rotation about the origin are generally enough for basic or less complex particle systems. Those two angles are rotation about the x-axis (ψ pitch) and rotation about the y-axis (ϑ yaw). These angles are varied by a random value and then converted for each particle.

Here is the conversion process for generating this direction vector. We will use the 3D rotation technique, which I was thought in the CEng 477 course.
When these two matrices are combined into a single rotation matrix, we get the following:

\[
\begin{bmatrix}
\cos(\psi) & \sin(\psi) & -\sin(\psi) \\
\sin(\psi) & \cos(\psi) & \cos(\psi) \\
-\sin(\psi) & \cos(\psi) & \cos(\psi)
\end{bmatrix}
\]

From this matrix we can easily calculate the new x y z values of the position of the particles.

\[
X = -\sin(\theta) \cdot \cos(\psi)
\]
\[
Y = \sin(\psi)
\]
\[
Z = \cos(\psi) \cdot \cos(\theta)
\]

Finally, this particle motion vector is multiplied by the speed scalar, which is also randomly modified.

By considering this information, we can construct the structure of emitter as follows:

```c
struct tEmitter
{
    long id;   // EMITTER ID
    long flags;   // EMITTER FLAGS

    // rotation to direction
    tVector pos;   // XYZ POSITION
    float yaw, yawVar;  // YAW AND VARIATION
    float pitch, pitchVar;  // PITCH AND VARIATION
    float speed, speedVar; // SPEED AND VARIATION

    // Particle
    tParticle *particle;    // NULL TERMINATED LINKED LIST
    int totalParticles;    // TOTAL EMITTED AT ANY TIME
    int particleCount;    // TOTAL EMITTED RIGHT NOW
    int emitsPerFrame, emitVar;  // EMITS PER FRAME AND VARIATION
    int life, lifeVar;  // LIFE COUNT AND VARIATION
    tColor Color, ColorVar;  // CURRENT COLOR OF PARTICLE

    // Forces
    tVector force;    // GLOBAL GRAVITY, WIND, ETC.
};
```
Variables ending with –Var are necessary to calculate the current values of the attributes. They are calculated as the emitNumber described before.

**The Functions:**

Now that we know what attributes are needed in the particle system base class, we can start thinking about what functions are needed.

To avoid many costly memory allocations, all particles are created in a common particle pool. We probably implement this as a **linked list**. When a particle is emitted, it’s removed from the common pool and added to the emitter’s particle list. While this limits the total number of particles we can have in the scene, it also speeds things up a bunch. By making the particle bi-directional linked, it’s easy to remove a particle when it dies.

Many particle systems will need their own unique **constructors**, forcing us to create a virtual constructor and destructor within the base class. In the constructor of the base class, we are planning to include these attributes:

- The number of particles we initially want to have in this particle system.
- The position of the particle system itself.
- The system type (its ID).

New features will be added during the implementation part. But we try to decide on the important and trivial ones at this step.

Each type of particle system **updates** particle attributes in a different way, so we need to have a virtual update function.

Once a particle is born, it is handled by the particle system. For each cycle of the simulation, each particle is updated. First, it is checked to see if it is died. If it has, the particle is removed from the emitter and returned to the global particle pool. During this process, global forces apply to the direction vector, and also the color can be modified.

This update function performs the following tasks:

- Updates all particle positions and other attributes, such as color, speed, etc.
- Counts the number of living particles. It returns FALSE if there are no living particles, and returns TRUE if particles are still alive. The return value can be used to determine whether a system is ready to be deleted or not.

A particle system is simply a collection of points, and so it can be **rendered** as just that, a set of colored 3D points. We can also calculate a polygon around the point so that it always faces the camera like a billboard. Then apply any texture you like to the polygon. By scaling the polygon with the distance from the camera, we can create perspective. Another option is to draw a 3D object of any type at the position of the particle.
2.5 FORCES

All particles are essentially alike. In contrast, the objects that give rise to forces are heterogeneous. As a matter of implementation, we would like to make it easy to extend the set of force-producing objects without modifying the basic particle system model. We accomplish this by having the particle system maintain a list of force objects, each of which has access to any or all particles, and each of which “knows” how to apply its own forces.

Forces can be grouped into three broad categories:

- **Unary forces**, such as gravity and drag that act independently on each particle either exerting a constant force, or one that depends on one or more of particle position, particle velocity, and time.
- **N-ary forces**, such as springs, that applies forces to a fixed set of particles.
- Forces of **spatial interaction**, such as attraction and repulsion, which may act on any or all pairs of particles, depending on their positions.

Each of these raises somewhat different implementation issues.

2.5.1 Unary forces

**Gravity.** Global earth gravity (as opposed to particle-particle attraction) is trivial to implement. The gravitational force on each particle is $\mathbf{f} = m \mathbf{g}$, where $\mathbf{g}$ is a constant vector (presumably pointing down) whose magnitude is the gravitational constant. If all particles are to feel the same gravity, which they need not in a simulation, then gravitational force is applied simply by traversing the particle system’s particle list, and adding the appropriate force into each particle’s force accumulator. Gravity is basic enough that it could reasonably be wired into the particle system, rather than maintaining a distinct “gravity object.”

**Viscous Drag.** Ideal viscous drag has the form $\mathbf{f} = \alpha \mathbf{v}$, where the constant $\alpha$ is called the **coefficient of drag**. The effect of drag is to resist motion, making a particle gradually come to rest in the absence of other influences. It is highly recommended that at least a small amount of drag be applied to each particle, if only to enhance numerical stability. Excessive drag, however, makes it appear that the particles are floating in molasses. Like gravity, drag can be implemented as a wired-in special case.

2.5.2 N-ary forces

In a basic mass-and-spring simulation, the springs are the structural elements that hold everything together. The spring forces between a pair of particles at positions $\mathbf{a}$ and $\mathbf{b}$ are

$$
\mathbf{f}_a = -\left[ k_s (|\mathbf{l}| - r) + k_d \frac{\mathbf{l} \cdot \mathbf{l}}{|\mathbf{l}|} \right] \frac{\mathbf{l}}{|\mathbf{l}|}, \quad \mathbf{f}_b = -\mathbf{f}_a.
$$
where \( f_a \) and \( f_b \) are the forces on \( a \) and \( b \), respectively, \( \mathbf{l} D a \circ b \), \( r \) is the rest length, \( k_s \) is a spring constant, and \( k_d \) is a damping constant. \( \mathbf{l}I \), the time derivative of \( \mathbf{l} \), is just \( v_a \circ v_b \), the difference between the two particles’ velocities.

In equation given above, the spring force magnitude is proportional to the difference between the actual length and the rest length, while the damping force magnitude is proportional to \( a \)’s and \( b \)’s speed of approach. Equal and opposite forces act on each particle, along the line that joins them. The spring damping differs from global drag in that it acts symmetrically on the two particles, having no effect on the motion of their common center of mass. Later, we will learn a general procedure for deriving this kind of force expression. A damped spring can be implemented straightforwardly as a structure that points to the pair of particles it connects. The code that applies the forces according to equation (given above) fetches the positions and velocities from the two particle structures, performs its calculations, and sums the results into the particles’ force accumulators.

### 2.5.3 Spatial Interaction Forces

A spring applies forces to a fixed pair of particles. In contrast, spatial interaction forces may act on any pair (or n-tuple) of particles. For local interaction forces, particles begin to interact when they come close, and stop when they move apart. Spatially interacting particles have been used as approximate models for fluid behavior, and large-scale particle simulations are widely used in physics. A complication in large-scale spatial interaction simulations is that the force calculation is \( O(n^2) \) in the number of particles. If the interactions are local, efficiency may be improved through the use of spatial buckets.

### 3 MANAGING VARIOUS PARTICLE SYSTEMS

We may also link the particle systems. By this way, we can have more than one system at the same time in the same frame. For his kind of cases we need a system manager to control all of our various particle systems. We can think this as a class.

We need to add particle systems easily. We also don’t want to keep track of all the systems to see if all of the particles died so we can release them from memory. That's what the manager class is for. The manager will automatically update and render systems when needed, and remove dead systems.

We basically decide which functions are really necessary for such a class. The functions described below are just the most essential ones. We can, in the following stages of the project, add or remove these functions

- Initialize = initializes the particle manager
- AddSystem = adds a specified particle system to the manager.
- RemoveSystem = when it is decided that all the particles which belong to that system are died, this functions removes that system from the manager.
- Update = updates all active particles systems and removes all system which died after the update.
- Render = renders all active and visible systems
Shutdown = removes all allocated systems and by this way shuts down the manager
CheckExistence = checks whether a given particle system will exist in the particle manager.

4 THE ADVANTAGES OF PARTICLE SYSTEMS

The two main advantages of particle systems are that they can generate large amounts of
detail and model objects with a dynamic form. According to Reeves a particle is a lot
simpler than most geometric primitives. If this is so, then it is possible to render more
particles, than geometric primitives, in a given amount of time. Main problem at this
point is producing a more detailed representation of an object. Particle systems are also
"procedural" and stochastic, so very little effort is needed to model and render complex
objects, In other words less computational power needed for rendering particle systems.
The characteristics and position of particles in a particle system can be changed to allow
it to represent dynamic objects, and the level of detail can be easily adjusted. That is,
attributes of the particles can be adjusted in certain amount of time.
Since particle systems are able to produce a large amount of irregular three-dimensional
detail with very little effort, they are the more appealing method for modeling natural
phenomena. Unfortunately, because they produce so much irregular detail, exact "visible
surface" and shading calculations become infeasible [1].

5 THE DISADVANTAGES PARTICLE SYSTEMS

Reeves’ initial paper on particle systems ignores visible surface and shading problems.
The particles in the paper represent fire, modeled as individual light sources. When
particles overlap in a pixel, their colors are simply added together. Shading is easy, due to
each particle being an independent light source. Unfortunately, particle systems are
sometimes used to model objects that are more complex than fire. The colors of the
particles representing these objects cannot simply be added together, and the objects
require a more complex shading model [1].

Later research by Reeves attempts to fix the problem of shading. Trees and grass are
generated using a probabilistic shading model. Because a tree consists of millions of
independent particles, it is difficult and time consuming to shade each individual particle
accurately, calculating whether it is in shadow. Instead Reeves determines the probability
of a particle being in shadow, and then uses a random number to decide whether or not to
render the particle if it were in shadow [3].
6 EXAMPLES FOR PARTICLE SYSTEMS

6.1 USING PARTICLE SYSTEMS TO MODEL FIRE AND EXPLOSIONS

Reeves initially used the term "particle system" to describe a method he used to create a sequence of images for the move Star Trek II: The Wrath of Khan. The effect he was trying to create was that of a bomb exploding on the surface of a planet and fire spreading out from the point of impact to eventually engulf the planet. Each particle in this system was a single point in space. The fire was represented by thousands of these individual points. Typical polygonal rendering methods create objects with straight edges, but representing the fire by thousands of points gives the fire a "fuzzy" shape. Reeves calls an object made up of particles a fuzzy object with the attributes position, velocity (speed and direction), color, lifetime, age, shape, size, transparency.

Particle Life Cycle

Each particle goes through three distinct phases in the particle system: generation, dynamics, and death. These phases are:

- **Generation**

  Particles in the system are generated randomly within a predetermined location of the fuzzy object. This space is termed the generation shape of the fuzzy object, and this generation shape may change over time. Each of the above mentioned attribute is given an initial value. These initial values may be fixed or may be determined by a stochastic process.

- **Particle Dynamics**

  The attributes of each of the particles may vary over time. For example, the color of a particle in an explosion may get darker as it gets further from the center of the explosion, indicating that it is cooling off. In general, each of the particle attributes can be specified by a parametric equation with time as the parameter. Particle attributes can be functions of both time and other particle attributes. For example, particle position is going to be dependent on previous particle position and velocity as well as time.

- **Extinction**

  Each particle has two attributes dealing with length of existence: age and lifetime. Age is the time that the particle has been alive (measured in frames). This value is always initialized to 0 when the particle is created. Lifetime is the maximum
amount of time that the particle can live (measured in frames). When the particle age matches its lifetime, it is destroyed. In addition there may be other criteria for terminating a particle prematurely:

- Running out of bounds - If a particle moves out of the viewing area and will not reenter it, then there is no reason to keep the particle active.
- Hitting the ground - It may be assumed that particles that run into the ground burn out and can no longer be seen.
- Some attribute reaches a threshold - For example, if the particle color is so close to black that it will not contribute any color to the final image, then it can be safely destroyed.

Rendering

When rendering this system of thousands of particles, some assumptions have to be made to simplify the process. First, each particle is rendered to a small graphical primitive (blob). Particles that map to the same pixels in the image are additive - the color of a pixel is simply the sum of the color values of all the particles that map to it. Because of this assumption, no hidden surface algorithms are needed to render the image, the particles are simply rendered in order. Effects like temporal ant-aliasing (motion blur) are made simple by the particle system process. The position and velocity are known for each particle. By rendering a particle as a streak, motion blur can be achieved.

6.2 USING PARTICLE SYSTEMS TO MODEL TREES

Reeves used the particle system approach to model trees in the animated short movie The Adventures of Andre and Wally B [2]. Each tree was created by using a particle system, and the position of the trees within the forest was also controlled by a particle system. The major enhancement over the previously mentioned particle system is the rendering algorithm used. In The Adventures of Andre and Wally B, a more traditional rendering method is used, and the previous simplification of rendering the particles in order and making them additive is not used. In addition, the generation shape of the particle system is not fixed, but is dependent on the current state. For example, leaves may only be created near the ends of branches.

Tree Particle System Rendering Technique

The particles of the trees are small circles used to represent leaves and lines used to represent branches. A more traditional rendering technique is used that is based on the painter's algorithm. Because of the large number of particles, standard lighting and shadow calculations are computationally prohibitive, so more reasonable solutions must be made.

- **Self Shadowing** - Particles on the outside of the tree are more likely to be directly illuminated then points on the inside of the tree due to the self shadowing of the tree. To model this, a lighting model is used where a cone is used as the bounding
volume for the tree. The distance from the point to the outside of the bounding
volume along the line between the point and the light source is used as a
parameter to the lighting equation. The lighting equation has a maximum value at
the outer edge of the bounding volume facing the light, and decreases
exponentially inside the tree. The previously mentioned distance parameter is
used as the exponent in this equation.

- **Ambient Lighting** - Ambient lighting is based on an equation similar to the self
shadowing equation. Since ambient light is not based on direct illumination from
a light source the distance metric is made from the point being illuminated
horizontally out to the edge of the bounding volume.

- **External Shadowing** - In addition to shadows cast by the tree on itself, shadows
cast from neighboring trees must be considered. This is accomplished by tracing a
line from the top of each neighboring tree to the light source. When a particle is
being rendered it is checked to see if it is above or below this line. Points above
this line are not shadowed, and points below this line have a chance of being in
shadow that increases in proportion to the distance from the line.

For the same reasons that shadows could not be perfectly generated for particles within
the trees, the shadows that the trees cast on the ground must also be approximated. The
way this was done was by rendering an image of the trees from a viewpoint on the
ground. This image was then texture mapped to the ground as a "shadow map." Points in
the image that correspond to trees were rendered as though they were in shadow.

The actual rendering algorithm used to draw the trees was a modified painter's algorithm.
The first assumption used by this algorithm is that the bounding volumes of no two trees
intersect, so each tree could be rendered independently. This cuts down on the number of
particles that have to be loaded simultaneously. In addition, instead of simply depth
sorting the particles, they are placed in a number of depth sorted bins. Inside each bin the
particles are simply rendered in order. This saves time because an exhaustive sort of all
particles does not have to be made. Errors caused by rendering particles in order inside
the bin are masked by the large number of particles.

**6.3 USING PARTICLE SYSTEMS TO MODEL FLOCKING BIRDS**

The third example of using particle systems was used by Reynolds to model the behavior
of birds moving in a flock [3]. In this particle system the particles are used to represent
"boids" (short for bird-object). This use of a particle system has a few differences from
what was used by Reeves:

- Each boid is an entire polygonal object rather than a graphical primitive
- Each boid has a local coordinate system.
- There are a fixed number of boids - they are not created or destroyed.
- Traditional rendering methods can be used because there are a small number of
  boids.
• Boids behavior is dependent on external as well as internal state. In other words, a boid reacts to what other boids are doing around it.

**Boid Behavior Model**

Reynolds used observation of real flocks and research of flock behavior to come up with three primary needs of a boid.

- **Collision avoidance** - The boid does not wish to collide with other boids or obstacles.
- **Velocity matching** - Each boid attempts to go the same speed and direction as neighboring boids.
- **Flock centering** - Each boid attempts to stay close to nearby flockmates.

The boid's movement is made by combining the impulses that are generated from these three needs. If each need produces a vector that represents the direction and speed it thinks the boid should move in, it may not be adequate to simply average these vectors. In the worst case, these three vectors may be pointing in completely divergent directions, and the net movement would be zero.

**7 CONCLUSION**

The paper presented a brief survey of the different approaches of Particle System Simulation used in most of the applications. A characterization of the fundamental methods has been given. We also try to compare the different algorithms to achieve the best performance and also try to determine the implementation difficulties. Then we decide on the class types and the fundamental functions that these classes have. Finally we search the most important and commonly used applications of the particle systems in order to easily decide on our own application. We get very beneficial information about the applications and also decide the main points of our application.

**8 REFERENCES**

