

Real-Time Special Effects for a Computer Game Using Particle Systems

Markus Hadwiger
e9425555@stud1.tuwien.ac.at

Abstract

An implementation of a particle system as part of the rendering pipeline of a 3-D computer game is described. Due to the interactive nature of such a computer game one of the most important issues in this implementation is the real-time rendering capability of the created special effects. A short summary of the game's rendering pipeline is given, restrictions on special effects without a particle system's being integrated are described and used to show where tremendous improvements are possible with the deployment of a particle system. The fundamental changes of the original rendering pipeline necessary to accommodate the particle system are considered and looked at in detail. As an example of a special effect made possible by the introduction of a particle system the calculation and rendering of a lightning stroke is presented. Due to the nature of the application the emphasis is not on physically correct evaluation of a lightning beam but instead on visual appearance, i.e., the aim is not so much to make it look real, but to make it look like a typical special effect.

1 Introduction

In the Spring of 1996 I started work on a computer-game called Parsec [1] for the computer graphics 2 and 3 laboratory course taking place in the summer term of 1996. Together with Andreas Varga I developed a first version of the 3-D rendering pipeline and the game engine as well as all the tools necessary, but due to the time schedule and the complexity of the task I never got around to implementing some impressive special effects.

What I am going to present in this paper is the first step towards the integration of a complete and quite powerful particle system into the already existing rendering pipeline, although limited by the criterion that it must not compromise the interactivity of the game, which is to say the game should still run with at least 20 frames per second on standard desktop PCs. I am going to look at some problems that posed and how I solved them. As a particular example for a number of special effects the particle system makes possible I am going to explain how I implemented a lightning stroke which is used as a 'extra weapon' during gameplay. I will also touch briefly on some other effects I implemented up to now using the new particle system as a basis.

2 Original Rendering Pipeline

Originally, every object in Parsec was represented and rendered as a b-rep model. The visibility determination of these objects is done using object-local BSP trees on the object level and simple depth-sorting on the inter-object level. Hence, every object is preprocessed by a BSP compiler [2] and the generated BSP trees are used to render object after object in order of decreasing z coordinate¹. The use of BSP trees made the rendering pipeline very effective in terms of rendering speed and ease of implementation, but also had severe implications on the types of effects possible.

¹ painter's algorithm; viewer looking along the positive z-axis

Basically, particles² could be implemented as 2-D bitmaps without extending the underlying rendering pipeline in any way and simply depend on the depth sort to generate the correct drawing order of objects and particles together most of the time.

Unfortunately, the first problem with this approach is that special effects where particles are contained in a non-convex region of a 3-D object - where depth sorted drawing order will fail most of the time - are very important. For instance, the lightning beam originates very closely to an object and - if it hits another object - will also terminate very closely to an object.

Secondly, the goal when using a particle system is most certainly to be able to render literally thousands of particles which renders depth sorting unfeasible for the determination of a correct drawing order.

Thus, a particle system could not easily be integrated with the already existing rendering pipeline.

3 Modified Rendering Pipeline

An easy way for a rendering pipeline needing to accommodate a particle system is simply to use a z-buffer for all necessary visibility determination purposes. There are two problems with this approach, however.

Firstly, a z-buffer is costly in terms of memory, which is a problem increasing with resolution³ and an especially crucial issue on rendering hardware that perhaps doesn't support z-buffering for the desired resolution due to lack of display memory.

Secondly, the use of a software z-buffer slows down the software implementation of a texture mapper dramatically, especially on an architecture like the x86 where the number of registers is very low and conditional branching in inner loops very costly.

I am therefore currently using a hybrid approach, using a z-buffer for the drawing of particles, and BSP-trees and depth-sorting for the drawing of 3-D objects. Naturally, these two components can't be entirely independent of one another. Whenever polygons are rendered into the screen buffer, they also fill the z-buffer accordingly. The difference to the typical use of a z-buffer is that it is never checked during rasterization of polygons, hence no conditional branching is necessary in inner loops. Actually, the z-buffer fill proceeds independently of the texture mapper, thus keeping register usage to a minimum. Since Parsec does not support any rendering hardware in its current version the z-buffer's memory usage isn't so much an issue as is the run-time efficiency of the rasterization routines. Nevertheless, only a 16 bit z-buffer is employed, which actually contains 1/z values instead of z values, both making linear interpolation possible and correct, and increasing the relative depth-precision of objects closer to the camera as opposed to objects farther away.

After all visible polygons are drawn in depth- and BSP-order, respectively, the z-buffer contains a correct 'z-footprint' of the scene which can be used to render any additional objects or effects that have to use z-buffer rendering for some reason. All particles are drawn into the screen buffer at this time, taking advantage of the fact that checking of previous z values needs not be done concurrently with the interpolation of new z values - due to the fact that the particle's bitmap has constant z over its entire surface.

² When I refer to the term particle in context of the rendering pipeline it denotes just a single location in 3-space where a bitmap is placed by the renderer. The dimensions of the bitmap are scaled accordingly to the particle's z-distance to the viewer.

³ Parsec currently supports resolutions from 320x200 up to 1280x1024, if supported by the graphics hardware and reported correctly by an installed VBE 2.0 compliant driver. Standard resolutions are 640x480 and 800x600, respectively - which is to say the frame rate is satisfying on mid-range desktop PCs in these modes.

4 Particles

Until now, no detailed description of the particles themselves or their capabilities has been given. This section is going to remedy this.

4.1 Attributes

Various attributes control the behavior of a particle during its lifespan. The attributes used are an adaptation of the ones used in [3].

1. *Position*: Every particle has a current position in 3-space which is updated every frame to reflect the particle's new position according to its animation type.
2. *Lifetime*: The lifetime of a particle controls its automatic destruction; every particle is created at a particular instant and dies automatically after its lifespan is spent.
3. *Type*: A very important property is the general particle type and its assigned animation style. The general type controls the interrelationship of particles, whether they are completely autonomous after creation or share attributes with other particles, for instance if they are part of the same 'particle-object'⁴ and therefore have to be destroyed simultaneously. Particle objects can be attached to spacecrafts, which implies that all transformations applied to the spacecraft are applied to all attached particles as well. This is used to visualize different kinds of protective shields, for instance.

Other properties are the corresponding bitmap and its scaling information and some more attributes needed for rendering.

4.2 Animation

The animation type and related attributes like velocity, etc. determine which kind of animation is applied to the particle. I implemented the following animation types:

1. linear motion
A fixed velocity vector representing both speed and directional information is added to the particle's position every frame.
2. spherical motion
The particle is part of a cluster of particles comprising a sphere. Different types of motion along the sphere's surface are possible:
 - Stochastic motion reevaluates the position of all particles on the sphere randomly each frame.
 - Rotational motion rotates all particles around the center of the sphere with fixed pitch, yaw, and roll angles per frame.Additional animation can be applied to the sphere as a whole, namely a pulsating sphere where the radius is a sine function over time, a contracting sphere which expands itself at the beginning of its lifespan and contracts itself automatically before dying, and an exploding sphere where the radius is increased constantly until the particles are destroyed.

⁴ A powerful feature of the particle system is the possibility to create objects consisting solely of particles. Particles comprising the same object must behave in a consistent manner. For instance, they must all be animated in a homogeneous way, e.g. rotate around the same origin.

3. lightning stroke

The particle is part of a lightning stroke. Currently two lightning beams originate from the spacecraft's front if activated. Particles that are part of a lightning beam have to be handled differently than all other particles, their position is reevaluated each frame according to the scheme presented in 5.2.

4.3 Particle Clusters

Particles are grouped into clusters to ease maintenance, viewing frustum culling, and express interrelationships. Shared attributes are only stored for the cluster as a whole, e.g. the radius of a particle sphere. Clusters can be attached to spaceships by inserting them into a list of attached clusters maintained for each existing ship. If the ship is destroyed, all attached particle clusters have to be destroyed as well.

All particle clusters are maintained in a doubly linked list. The renderer traverses this list each frame and performs culling against the viewing frustum for entire particle clusters, clipping for individual particles, and draws all visible particles afterwards using the z-buffer.

5 Example Special Effect: Lightning Stroke

As an example of a special effect made possible by the particle system, I am going to explain how I implemented the visualization of a lightning stroke.

5.1 Basics

As in [4] the emphasis is not on physically correct simulation of a real lightning stroke, but on visual appearance. I furthermore generously adapted the algorithm given in this work to the application at hand. The biggest difference to the algorithm presented in [4], is that in Parsec lightning beams have to be able to originate from a specific position in (outer-)space in order to be used as a weapon and there is no ground zero they can hit. Also, there are no branches off the main channel. So, besides visual appearances, they have nothing in common with physical lightning anymore. Furthermore, real lightning does not alter its path once the downward and upward leader meet and the first return stroke is triggered⁵. To make the lightning beams look more like a special effect, however, the channel is reevaluated every n frames, thus giving them more the look of arcs of light that can be seen when arc welding.

The work in [4] uses particles to evaluate the lightning channels and connects them via shaded and 'glowing' lines afterwards. In Parsec, however, I am simply evaluating the position of the particles comprising the main channel and using a brightly colored, jagged bitmap for the rendering of these particles.

If the weapon is triggered, two jagged beams of light progress in the direction of the positive z axis. Whenever a collision with a spaceship is detected, the progression of the beam is stalled and the spacecraft's protective shield - also visualized using particles - is activated. At the point of impact particles are created to visualize the collision between lightning beam and protective shield.

⁵ A negative stream of electrons - the stepped downward leader - propagates from the cloud towards ground zero, to be met by the positively charged upward leader. This process determines the form of the lightning channel, which is subsequently illuminated by multiple return strokes. However, the form of the channel never changes during the discharge. For the details of a physical lightning stroke see [4].

5.2 Algorithm

Each lightning beam starts with a seed segment, which basically determines the general direction of the channel and its basic segment length. The progression of the channel is iteratively evaluated by choosing a random deviation within some maximum limit from the original seed segment. These segments - vectors from one particle in the channel to its successor - are also randomly varied in length. Starting with the seed segment, segments are evaluated and concatenated until some maximum channel length limit is reached. There the lightning beam terminates - in reality it would terminate when hitting ground zero, but, as we are in outer space, it has to terminate automatically when a certain limit is reached. Apart from the seed segment, all segments are reevaluated every n frames to give the effect the impression of a sizzling beam of light.

[4] also discusses the implications of the pseudo-random number generator on the generation of branches off the main channel. However, as the particles aren't connected by lines in Parsec, the use of branching just confused the visual appearance of the lightning beam as a whole. Particles in secondary channels obstructed the main channel too much. Therefore, only main channels without branches are generated.

6 Potential Improvements

There is lots of room for potential improvements. Foremost of all, the collision detection for single particles, if performed at all, is done using a simple brute force method. The clustering of particles can be used as a kind of hierarchical grouping to reject multiple particles simultaneously, but in practice this is far from sufficient, especially with highly spatially distributed particle clusters. An octree would probably be a feasible approach to solve the execution-time problems of collision detection with a large number of particles in each frame. Also, besides lightning, various kinds of particle spheres, and some kinds of linear motion particles, there are no special effects implemented as yet. The addition of a particle system to the original rendering pipeline makes a huge number of special effects possible, thus yielding a powerful tool to improve gameplay and visual appearance via additional devices, weapons, and effects.

References

- [1] Markus Hadwiger and Andreas Varga, Parsec - a 3-D Network Space-Fight Simulator, Version 0.8b, available from <http://cg.tuwien.ac.at/courses/CG23/HallOfFame.html>
- [2] Markus Hadwiger, MakeBSP, Version 1.26a, available from <ftp://cg.tuwien.ac.at/pub/cg23/makebsp.zip>
- [3] William T. Reeves. Particle Systems - A Technique for Modeling a Class of Fuzzy Objects. In *acm Transactions On Graphics*, 2(2), April 1983
- [4] Todd Reed and Brian Wyvill. Visual Simulation of Lightning. In *Proceedings of SIGGRAPH '94*, pp. 359-363, 1994