The Design of an API for Particle Systems

David K. McAllister

Department of Computer Science
University of North Carolina at Chapel Hill

Abstract

Simulation of dynamic particle systems has been used in computer animation for several years and has more recently been used in real time simulation and video games. Much research has explored ways to compute and render particle systems, but relatively little research has discussed suitable application programmer interfaces to the particle systems. What constructs and abstractions are useful for specifying particle effects? How can the API be made general so that unforeseen effects can be created? What set of functionality should be provided that can be implemented on either the application CPU or within specialized graphics hardware?

The result of our research is the Particle System Application Programmer Interface (API). This is a C++ function library specification that allows applications to simulate the dynamics of particles. The API was created for easily adding a variety of particle-based effects to interactive graphics applications such as games and virtual environments. The API is also well suited for non-interactive graphics systems such as modeling and rendering packages for computer animation. It is not intended for scientific simulation, although principles of Newtonian physics have been used to implement the particle dynamics where applicable, and the accuracy of the numeric integration is scalable.

1 Introduction

As the computer graphics field becomes more sophisticated and the demand for realism, quality and interaction increases – both in computer generated effects, and in video games and other simulations, it becomes ever more necessary to encapsulate and abstract functional components of these applications. Also, computer graphics hardware is changing very rapidly, so application programmer interfaces (APIs) are necessary to provide a consistent, portable interface to the hardware implementations of any given functionality. There are several examples: OpenGL [Neider 1993] and Direct3D abstract the low-level operations of real-time rendering. OpenGL Optimizer and DirectModel encapsulate scene graph functionality. Object physics and collision detection libraries such as the Ipion Virtual Physics Engine and Telekinesys Research's Omniate provide encapsulated object dynamics. Game engines such as id Software's Quake Engine and Numerical Design Ltd.'s NetImmerse have become an important component for encapsulating the representation and dynamics of a video game's virtual world, and often integrate the functionality of other APIs. We propose to extend this list by researching an API to dynamic particle system simulation. The two motivations for particle system API research are the same as those for the above-mentioned APIs. First, we desire to encapsulate the functionality so that developers do not need to implement it repeatedly and so that the application can be easily ported to any platform that has the API. Second, we desire to provide an interface to the functionality so that it may be implemented within specialized graphics hardware, without affecting the application's portability.

Within the Particle System API, a particle is computed as a simple point in three-space with a fixed set of attributes such as position, velocity and color. The responsibility of the library is to compute values of these attributes for each particle over time. The API also includes a few helper functions for rapidly rendering the particles using some implementation dependent method, although these functions will usually be ignored in favor of application-dependent rendering methods.

The API specification is centered on the *particle group*, which is a set of particles that are acted upon as a group by the same forces. Several particle groups may exist at once, but all API functions apply to the *current particle group*. Particle groups are acted upon using *action* functions that perform fairly low-level operations on a group of particles, often simulating physical forces. Examples of *actions* include gravity, bouncing, color fading, etc. A complete effect is created using a series of actions. Actions may be compiled into *action lists*, which are similar to OpenGL display lists. Action lists encapsulate particle behavior into higher-order effects and also enable implementation-specific optimizations.

The source code for the UNIX and Windows implementations of the Particle System API and its user manual are publicly available at [McAllister 1999] and have been used by many programmers. The API is a key component in a number of publicly available software packages ranging from video games to virtual environment and animation authoring systems. This paper does not seek to teach the use of the API but instead focuses on solutions to the problems encountered while designing an interface to particle system functionality.

2 Previous Work

Dynamic particle systems for computer graphics were first conceived of by [Reeves 1983]. His specification of what attributes a particle has and how particle systems are computed prevail through most subsequent research. Reeves' particles are generated within a spherical, rectangular, or circular *generation shape*. Their initial velocity is tightly coupled to their initial position. The other particle attributes like color are generated stochastically with a mean and a tolerance, similar to the **PDBox** domain that we will present. The dynamics of the original particle systems were quite simple. The only force af-

fecting particles after their creation was gravity, yielding parabolic paths for all particles. The great complexity achieved by these particle systems stemmed mainly from complex, stochastic initial values, not from complex dynamics. Reeves' follow-on paper, [Reeves 1985], includes more sophisticated particle motion for grass and introduces constrained particles for modeling solid objects like grass and trees.

[Reynolds 1987] introduces *Boids*, which represent individual birds or other animals that flock or school in a realistic-appearing manner. Boids can be seen as extending particle systems by adding a full coordinate frame to each particle and modeling more complex particle motion, especially motion resulting from interaction with other particles. This is necessary to model flocking behavior. Obstacle avoidance, surveyed in [Reynolds 1988], allows boids or particles to steer to avoid obstacles. We use a selection of these methods to implement obstacle avoidance.

The Connection Machine was used in [Sims 1990] to render particle system animations in a data parallel system, with quite compelling results. The paper set forth a suite of simple operations to be performed on particles, which could be combined to create the complete effect. We directly adopt many of these operations in our work. The interface to these operations is not discussed. The paper mentions that re-implementing the particle animation code on a serial computer would be interesting future work that has not been done because of the unique parallel nature of the implementation. In part we are fulfilling this future work. This also illustrates the need for a particle system API so that the animation and rendering application would be more portable.

[Leech 1993] describes an interactive particle system virtual environment toolkit. Wearing a head-mounted display, the user has access to a suite of widgets that each represent a particle system operation, most of which correspond directly to the operations defined by [Sims 1990]. These widgets can be interactively adjusted and placed in the environment to see results of particle system special effects. The main focus of the paper was the immersive, graphical interaction system. The paper also seems to be the first in the literature to use arbitrarily combinable operations like those of Sims in a serial computer.

Video games have started to use real-time particle systems for special effects. Quake III Arena by id Software has particle sparks and particle smoke trails. These particle systems seem to contain between one hundred and five hundred particles and are rendered either as point primitives or textured screen-aligned polygons. Their behavior mostly consists of gravity and a color fade, with complex initial velocities, such as spirals.

Most commercial animation packages such as Maya and 3D Studio Max now include a particle system animation package. Since these are usually rendered offline the developers are free to include more computationally intensive particle operations than those that must run in real time. The major area of innovation for these implementations seems to be the method given to the animator to constrain and script the particle dynamics. This is less of an issue in the real-time domain since there is no animator, although our API can be used for tight control of particle systems.

3 Design Goals

The Particle System API was designed with specific goals and constraints. We list these roughly in order of their importance to the design.

Run-time Efficiency – The primary purpose of the API is to enable real-time applications to include dynamic particle systems. This requires that the particle dynamics be computed efficiently so that the CPU has enough time per frame to perform the rest of the application's computation.

Flexibility – The API should be specifically designed to allow the user to create many effects not envisioned by the API designers. For this reason, the API consists of simple building blocks such as *Gravity* and *Bounce*, rather than complete effects such as *Flock* or *Fountain*.

Independence of Parameters – Since the API will consist of simple building blocks, the design space for implementing a particular effect can be quite large. It should be clear which parameter of which action in a particular effect should be modified for a particular visual result. Also, the parameters to the action functions should be in terms that do not depend on a particular external measuring system and the units of parameters should be transparent to the API.

Scalable Simulation Quality – The numerical accuracy of the simulation must be scalable and modifiable by the application. This is because we desire the same API to be usable for production-quality offline animation and for real-time special effects in already CPU-intensive video games. Also, some action functions, such as orbiting a gravity source, are much less numerically stable than others, like velocity damping. The application programmer should be able to specify different accuracy needs for different effects.

Hardware Abstraction – The API must truly be an abstraction of the particle system functionality. The application should work identically whether the API is implemented entirely as a library linked into the application, or as a thin layer that merely communicates the particle dynamics instructions to the graphics hardware or some other device.

Scalability – Particle functions should be callable from multiple threads and multiple processors in order to increase the total rate of particle simulation. The applicable API calls should be reentrant.

Easy to Learn – The OpenGL graphics API is broadly considered to have an elegant, easy to learn structure [Sepúlveda 1998; Silicon Graphics 1998]. For example, most OpenGL calls for defining geometry take a quadruple of numbers that can be specified using any data type and the user can choose whether to specify two, three, or all four parameters, with consistent defaults chosen for the unspecified parameters. This self-consistency allows users' knowledge of one API call to extend to many other calls, and greatly reduces the need to refer to documentation. We desire a similar elegance for particle system function calls. We also want the order of the function calls to matter as little as possible to avoid programming pitfalls.

4 API Description

A particle within the Particle System API is an object with a set of attributes very similar to those of Reeves' original particle systems [Reeves 1983]: *position, velocity, color, alpha, size, age, secondary position*, and *secondary velocity*. All attributes are three-vectors, except a*lpha* and *age*, which are scalars. The secondary position is normally a destination position and is rarely used. The secondary velocity normally stores the velocity from the previous time step for computing particle orientations from their instantaneous curvature. The three-vector *size* is only a rendering attribute, like color. Its use is completely application-dependent. For particle dynamics purposes, particles are a unit point-mass.

Like OpenGL, the API uses C-like function calls that are not members of a class. For the sake of elegance, a context handle is not passed into the functions. Section 4.6 discusses implications of not passing a context. API function names take the form **pFunctionName**. Most calls are defined with default values for the lesser-used arguments to simplify the application developer's coding in the common case. This is the only trait that makes the API C++-specific.

The API includes four kinds of functions: calls that operate on and manage *particle groups*, calls that set the current state of the library, *action* calls that act on *particle groups*, and calls that create and operate on *action lists*. We will discuss each of these in turn. Refer to the appendix for a complete list of API calls.

4.1 Particle Groups

All particles exist within a *particle group*, which is a set of particles that are acted upon by the same forces. Several particle groups may exist at once, but all API functions apply to the *current particle group*. All actions apply to every particle in the current particle group. A particle group is first created using **pGenParticleGroups**, which will return the identifying number of the generated particle group. **pCurrentGroup** switches which group is current. Particle groups are initially empty and **pSource** emits new particles into the group. The maximum number of particles in the group is specified using **pSetMaxParticles**. When a particle group reaches its maximum size, attempts to add more particles are ignored.

4.2 Actions

Actions are the functions in the API that modify the attributes of particles in the current particle group. Most actions are meant to simulate physical forces such as **pGravity**. Others have no physical counterpart such as **pRestore**, which computes a parametric quadratic path for each particle to some specified position. The API currently has 27 action functions, and future versions of the API can easily add more actions.

Action functions are meant to be building blocks for special effects, not finished special effects themselves. Most particle effects consist of between three and eight actions. Each action performs a pass over all particles in the particle group. This may be slightly slower than a single pass for a hard-coded series of actions, but separate passes allow the action functions to be building blocks that can be combined in any desired order. Multiple passes over the particles are feasible because several thousand particles can fit in a current L2 cache.

Action functions are normally called while or just before the application renders the scene each frame. The following pseudocode outlines this task.

```
for each particle group j
    pCurrentGroup(j)        // Context is group j.
    for each time step per rendered frame
        pSource(…)          // Emit new particles
        other actions…
        pMove()             // Update positions
    end for
    pDrawGroup(…)           // Draw particles
end for
other drawing…
```

The **pMove** action adds the fraction *dt* of each particle's current velocity to its current position, and then increments its age by *dt*, thus completing the time step for the particle. Section 4.5 discusses the mathematics of this process in greater detail. Once particles have been moved to their new location the particles are rendered. To render a particle group with each particle being a primitive (such as a point) the application calls **pDrawGroupp**. For each particle to be rendered as a model stored in a display list the application calls **pDrawGroupl**. Alternatively, calling **pGetParticles** will return the particle data to the application for any desired rendering or processing.

As an example of the ability to rapidly and simply define interesting particle effects with the API, in figure 1 we give the C++ code for creating a simple water fountain. A still image resulting from running the code is shown in figure 2.

One of our principal goals was for the API calls to be homogeneous. Whenever sensible, we desire multiple action functions to take the same parameters in the same order with the same default values. For example, many functions specify a target value for some particle attribute and asymptotically approach that value over time. These functions are **pTargetColor**, **pTargetSize**, **pTargetVelocity**, and **pAvoid** (steer to avoid a physical obstacle). All these functions take the blending parameter as a single float called *scale*. This allows users to learn the behavior of *scale* once and apply that knowledge to all similar functions.

Many action functions require a falloff of the effect of the action based on the particle's distance from some influence. We describe our method of specifying this falloff, which is another example of the API's homogeneity. Some functions that use the falloff are **pOrbitPoint** (accelerate toward a point gravity source), **pMatchVelocity** (match each particle's velocity to those of its neighbors), and **pVortex**. The force of these actions and their falloffs are specified using three floats: *magnitude, epsilon,* and *max_radius*. We use a modified inverse-square falloff as a function of range, *f(r)*, given by

$$f_{m,\boldsymbol{e}}(r) = \begin{cases} 0 & r \ge r_{\max} \\ \dfrac{m}{r^2 + \boldsymbol{e}} & r < r_{\max} \end{cases}$$

Epsilon serves to dampen the inverse square so that *f(r)* does not approach infinity as *r* approaches zero. An *epsilon* near in

pVelocityD(**PDCylinder**, 0.01, 0.0, 0.35, 0.01, 0.0, 0.37, 0.021, 0.019);
Choose particle velocities in a ring, like a nozzle.

pColorD(1.0, **PDLine**, 0.8, 0.9, 1.0, 1.0, 1.0, 1.0);
Particle colors are between light blue and white.

pSource(100, **PDLine**, 0.0, 0.0, 0.401, 0.0, 0.0, 0.405);
Generate particles along a small line in the nozzle.

pGravity(0.0, 0.0, -0.01);
Accelerate particles downward.

pBounce(0.0, 0.35, 0, **PDDisc**, 0, 0, 0, 0, 0, 1, 5);
Bounce particles off a horizontal disc of radius 5.

pSink(false, **PDPlane**, 0,0,-3, 0,0,1);
Kill particles below horizontal plane.

pMove();
Step particles forward in time.

Figure 1: *The particle system code used to simulate a water fountain.*

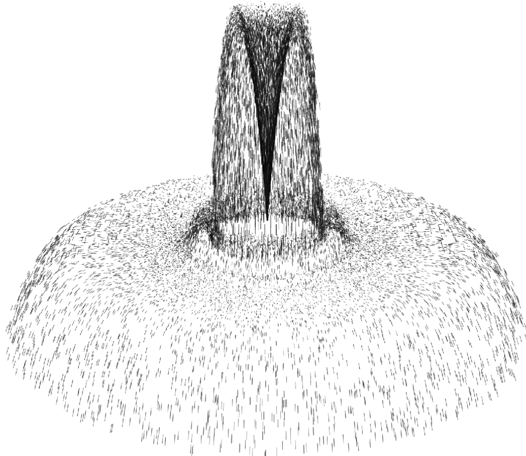

Figure 2: *A fountain created with the code in figure 1.*

size to *magnitude* can yield a much slower falloff and gives the action a wider neighborhood. Figure 3 shows an example of *f(r). max_radius* clamps the influence to zero beyond a given range. This can provide sharper falloffs than are possible with just *magnitude* and *epsilon* and can also increase efficiency in some implementations.

4.3 Action Lists

Actions may be compiled into *action lists*, which encapsulate all the operations required to produce a particular effect. Not only do action lists provide abstraction of effects but they also provide an interface to programmability within graphics hardware, if the particle system computations are being performed in such hardware, as on PixelFlow [Eyles 1997]. This is akin to the problem of defining an API for programmable shading on graphics hardware and our simple solution may have some applicability to that problem as well.

Action lists are generated using **pGenActionLists** followed by **pNewActionList**. Then all subsequent action and state change calls are stored in the action list instead of being executed immediately. Finally, **pEndActionList** finishes the list and returns the API to normal execute mode. **pCallActionList** executes the calls within action list as if they were called explicitly at that point. An exception is that the scope of state changes made within the action list ends when the list returns, so that action lists to be executed on different processors do not require synchronization of state afterward.

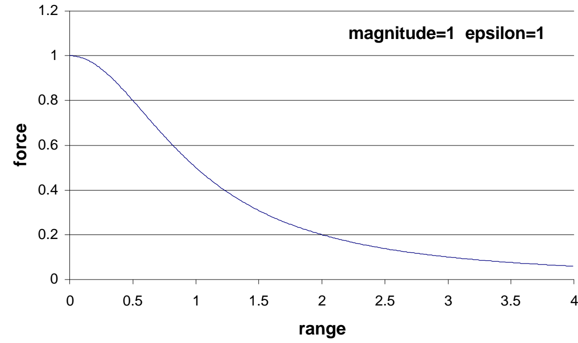

Figure 3: *Graph of damped inverse-square falloff function.*

Action lists can improve performance in two ways. First, they reduce the communication between the application and the hardware that computes the particle dynamics. Second, action lists allow certain combinations of actions to be recognized and replaced by optimized, hard-coded routines. For example, an implementation of the API could recognize the sequence of actions in figure 1 and call an optimized fountain function that does a single pass over all particles.

4.4 Attributes and Domains

One of the API's most important actions is **pSource**, which creates new particles. These particles must be given a color, velocity, size and initial age. To avoid placing all of these values in a single call to **pSource** and to increase the API's flexibility we store the current color, velocity, size, etc. of particles yet to be created as API state. This is similar to calling **glNormal**, **glColor**, or **glTexCoord** before creating a vertex with **glVertex** in OpenGL. The functions **pColor**, **pSize**, **pStartingAge** and **pVelocity** perform similar functions for particles, which are then created using **pVertex** or **pSource**.

Many particle effects require the particles' initial attribute values to vary, which the above state setting calls do not do. To handle this and other issues, we introduce the concept of a *domain.* Domains provide a uniform mechanism for specifying a region of three-space to the API. The function **pColorD** takes a domain that specifies a region of color space from which to stochastically choose the color of each new particle. Likewise, **pVelocityD** specifies a region of vector space from which to choose the velocity of each particle. Figure 4 illustrates randomly chosen velocity vectors within a disc domain.

Domains are used as parameters to some actions and state functions. For example, **pSource** creates particles with positions randomly chosen within a domain, **pSink** kills particles that enter or leave a domain, **pAvoid** specifies a domain of space for the particle to steer away from, and **pBounce** causes particles to bounce off the surface of a domain.

Domains come in a variety of shapes. Most domains define a 3D volume: **PDSphere**, **PDPlane** (which represents the half-

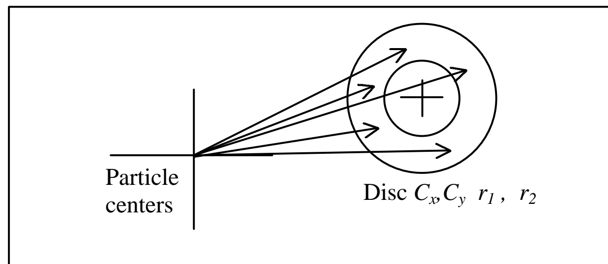

Figure 4: *Velocities chosen from a* **PDDisc** *domain.*

space bounded by the plane), **PDBox**, **PDBlob**, **PDCylinder**, and **PDCone**. Other domains are of lower dimension: **PDTriangle**, **PDRectangle**, **PDDisc**, **PDLine**, and **PDPoint**.

Nine or fewer floats can specify any domain. For example, **PDTriangle** has three 3D points, **PDCylinder** has an axis vector, a point on the axis, and scalars for length and an inner and outer radius. All API calls using a domain take an enumerated domain type having one of the values listed above, followed by nine floats with default values so that simpler domains can be specified more succinctly.

The **PDBlob** domain probabilistically defines a fuzzy region of space by specifying a center and a standard deviation of the gaussian probability density. This probability density is used both for choosing a random point in the domain and for choosing whether a given point is said to be within the domain. The **PDBlob** domain is very important to the API's simulation quality because many natural phenomena have been shown to follow a normal distribution [Ross 1976]. Specifying particles with colors, positions, or velocities taken from a normal distribution gives natural-looking results, free of distracting discontinuities at the boundaries. For this same reason, the **pStartingAge** state call takes an optional standard deviation argument so that initial particle ages are taken from a normal distribution. This way not all particles created at a given time will die at the same upon a call to **pKillOld** because of their normally distributed age range.

4.5 Particle Simulation

Since many action functions simulate physical forces, the acceleration of each particle must be computed using Newton's Law, **F = ma**. We treat all particles as having unit mass so we can implement actions directly using accelerations and yet think of them as forces.

Particle action functions specify the first or second derivative of some particle attribute over time. The method of numeric integration of this function affects the physical accuracy (for effects meant to simulate the physical world) and visual quality of the resulting simulation.

The purpose of a particle system simulation is to compute each particle's attributes – position, size, color, etc. – as they vary over time. Each particle attribute is a function of time, albeit a rather complicated function that is difficult or impossible to express in closed form. This function is instead expressed as a differential equation – the particle actions specify the *change* to the particle's attributes at a given time, rather than specifying their *value*. For example, the **pGravity** action specifies that each particle's velocity vector undergo a constant acceleration. The task of computing each particle's position given this differential equation is a numeric integration problem.

Many numeric integration methods exist, but few of them are suitable to real-time particle system simulation for computer graphics. Most methods adapt the step size to the function itself (the path of each particle), but giving each particle its own step size requires much more computation and storage. This would also cause each attribute of a particle to require a different, conflicting step size. Also, some action functions are random and some yield discontinuous attribute values over time, causing adaptive and predictive methods to fail. We use Euler's method of numeric integration with a user-specified step size *dt* to adjust accuracy. Euler's method simply steps the attribute value by the derivative (the action function in our case) evaluated over the interval *dt*. Using a smaller *dt* requires more integration steps per unit time but increases accuracy by treating the attribute function as piecewise-linear over shorter intervals.

The API is independent of the time unit – no physical constants are hard-coded into the API. The most common time unit is a single frame time. Although the value of *dt* can be anything, it is most often a whole number of time steps per unit time so that attribute values are computed exactly at each frame time, rather than interpolated from nearby integration step times to the rendered frame times.

4.6 Parallelization

The API has been implemented both on the parallel geometry processors (GPs) of PixelFlow and on UNIX and Win32 systems. In the PixelFlow implementation, the API library is a thin layer that passes the calls to the GPs. The core of the particle system implementation runs on a subset of the GPs, with each GP being responsible for a subset of the particle groups, or for a subset of the particles in a particle group[1]. The particles themselves never exist on the host, only in the GP, and are rendered directly by that GP. This makes for a very fast implementation, free of communication bottlenecks.

Our UNIX/Win32 implementation works on both uniprocessor and multiprocessor shared memory systems. The simulation of large particle systems is easy to parallelize on these systems by using multiple threads. However, the API and the program must both be carefully written to avoid race conditions. Any thread can execute actions on a particle group, but typically only the thread that created the particle group does so. On SGI Origin systems this ensures that the computation is performed on the processor that contains that particle's memory.

In most implementations of OpenGL it is forbidden for multiple threads to be executing an OpenGL call concurrently. The fundamental reason for this is that OpenGL calls do not receive a context parameter. The current context is a global variable within the library, and is thus not thread safe. We wish to keep the elegance of not passing a context parameter, but make the Particle System API thread safe so that it can run on large SMP machines. We solved this problem by having each API call find its calling thread's current context within a hash table keyed off a thread identification number. This number is

[1] This prevents inter-particle forces, so **pFollow**, **pGravitate**, **pMatchVelocity**, and **pGetParticles** are not supported.

not passed to the API, but is determined using an implementation-dependent call such as *getpid*.

5 Results

5.1 Ease of Use

The Particle System API has been publicly available on the Internet since July 1998 and has been downloaded by over a thousand people. Although the extent of its use by the public is impossible to determine, it has been used in a number of different applications of which we are aware. Within the Computer Science Department at UNC, the Particle System API was used to create an immersive Fountain Construction System and to make water flowing from a faucet interact with the shape of a person's moving hand.

Flow [Allen 1999] is a publicly available animation scripting system for Linux and IRIX that uses the Particle System API to implement particle dynamics. *Flow* is similar to many other commercial computer animation systems. It allows all animations to be edited and previewed online in an OpenGL window, with all particle system effects simulated in real-time. Flow synthesizes RenderMan RIB files that are rendered by Blue Moon Rendering Tools or Photorealistic Renderman.

The Stage 3 Research Group at Carnegie Mellon University has included the Particle System API in its *Alice* virtual worlds authoring system [Conway 2000]. *Alice* allows its users to create virtual worlds by loading models and images in standard formats and then using the Python scripting language [Beazley 1999] to add behaviors to them and integrate them into a virtual world. The Particle System API was integrated into *Alice* such that every API function becomes a Python function in *Alice*. *Alice* was used in a *Building Virtual Worlds* course at CMU that contained a mix of Computer Science, Art, Architecture, and Drama undergraduates. Approximately one fourth of the 65 projects submitted by the students included particle system special effects – usually rain, snow or fountains, as well as some explosions. We feel that this gives some indication of the ease of use of the Particle System API.

5.2 Flexibility

Jason Pratt, who integrated the Particle System API into *Alice*, has created three virtual worlds that each include a particle effect that we did not envision the API being capable of. These effects are ocean waves, a rubber sheet model, and a simulated snow globe with the snow swirling around in response to user motion. The fact that the API succeeded in simulating effects far different from those we imagined helps indicate our achievement of the flexibility design goal.

5.3 Performance

See the accompanying video for an illustration of the practical performance of the existing implementations of the API. All but the animation from *Flow* were recorded in real time. To measure numerical performance of the API we simulated 20,000 points orbiting two gravity sources, with one bounce plane and velocity dampening, with particles emitted using a normal distribution. Particles were rendered using aliased GL_POINTS in a 512x512 window. A a 500 MHz Pentium III processor with an nVidia GeForce 256 achieved 24.9 fps. Our 300 MHz R12000 SGI InfiniteReality2 reached 32.9 fps.

6 Future Work

Particle systems are beginning to come of age in the interactive computer graphics world. Many video games use particle systems and still more are in development. As these developers have approached us they always ask the question of whether they will be able to *get the effect that they want*. Occasionally they can't. The API needs user-definable particle attributes. For example, we have been asked for separate diffuse and specular colors. Rather than creating many special attributes we desire the API to be extensible in this regard while still conforming to its original design goals.

Related to this, particle attributes should be treated more generically. For example, rather than having a **pTargetColor** for fading particle colors and a **pTargetSize** for interpolating size, we would like to have a generic attribute fade function. Most actions currently affect the particle's position. Making the computation itself orthogonal to which attribute it is being performed on would increase the API's flexibility.

Another interesting area of future work is to implement the API in graphics hardware on additional systems. The PixelFlow implementation benefited greatly from having the particles computed on the same processor that then rendered them. We suspect that the same would be true in PC graphics cards that have a geometry processor. The advantages of this are that AGP bus bandwidth, a major bottleneck in PC 3D graphics, is greatly reduced since the particles only exist within the graphics card; and also the CPU is then free for other computation.

7 Acknowledgements

We wish to thank Jon Leech of Silicon Graphics, Inc. for example code and the *domain* concept. We appreciate the loan of graphics cards from nVidia and Evans and Sutherland. The PixelFlow video is courtesy of Hewlett-Packard Corp. This work was supported by a fellowship from Integrated Device Technologies. Thanks to Jason Pratt of CMU for the **PDTriangle** domain and for help with the *Alice* demo video. Thanks to Mark Allen, the creator of *Flow,* for helpful discussion, motivation, and really making particle systems look good.

8 Appendix – List of API Functions

8.1 Particle Group Functions

void **pCopyGroup**(int *p_group_num*, int *index* = 0, int *copy_count* = P_MAXINT)

Copy particles from the specified group into the current group.

void **pCurrentGroup**(int *p_group_num*)

Change which group is current.

void **pDeleteParticleGroups**(int *p_group_num*, int *p_group_count*)

Delete one or more consecutive particle groups.

void **pDrawGroupl**(int *dlist*, bool *const_size* = false, bool *const_color* = false, bool *const_rotation* = false)

Draw each particle as a model using OpenGL display lists.

void **pDrawGroupp**(int *primitive*, bool *const_size* = false,

bool const_color = false)

Draw a particle group using OpenGL primitives.

int **pGenParticleGroups**(int *p_group_count* = 1, int *max_particles* = 0)

Create particle groups, each with max_particles allocated.

int **pGetGroupCount**()

Returns the number of particles existing in the current group.

int **pGetParticles**(int *index*, int *count*, float **position* = NULL, float **color* = NULL, float **vel* = NULL, float **size* = NULL, float **age* = NULL)

Copy particles from the current group to application memory.

int **pSetMaxParticles**(int *max_count*)

Change the maximum number of particles in the current group.

8.2 Attribute State Functions

void **pColor**(float *red*, float *green*, float *blue*, float *alpha* = 1.0)

Specify the color of new particles.

void **pColorD**(float *alpha*, **PDomainEnum** *dtype*, float *a0* = 0.0, float *a1* = 0.0, float *a2* = 0.0, float *a3* = 0.0, float *a4* = 0.0, float *a5* = 0.0, float *a6* = 0.0, float *a7* = 0.0, float *a8* = 0.0)

Specify the color domain of new particles.

void **pSize**(float *size_x*, float *size_y* = 0.0, float *size_z* = 0.0)

Specify the size of new particles.

void **pSizeD**(**PDomainEnum** *dtype*, float *a0* = 0.0, float *a1* = 0.0, float *a2* = 0.0, float *a3* = 0.0, float *a4* = 0.0, float *a5* = 0.0, float *a6* = 0.0, float *a7* = 0.0, float *a8* = 0.0)

Specify the size domain of new particles.

void **pStartingAge**(float *age*, float *stdev* = 0.0)

Specify the initial age of new particles.

void **pTimeStep**(float *new_dt*)

Specify the time step length.

void **pVelocity**(float *x*, float *y*, float *z*)

Specify the initial velocity of new particles.

void **pVelocityD**(**PDomainEnum** *dtype*, float *a0* = 0.0, float *a1* = 0.0, float *a2* = 0.0, float *a3* = 0.0, float *a4* = 0.0, float *a5* = 0.0, float *a6* = 0.0, float *a7* = 0.0, float *a8* = 0.0)

Specify the initial velocity domain of new particles.

void **pVertexB**(float *x*, float *y*, float *z*)

Specify the secondary position of new particles.

void **pVertexBD**(**PDomainEnum** *dtype*, float *a0* = 0.0, float *a1* = 0.0, float *a2* = 0.0, float *a3* = 0.0, float *a4* = 0.0, float *a5* = 0.0, float *a6* = 0.0, float *a7* = 0.0, float *a8* = 0.0)

Specify the secondary position domain of new particles.

void **pVertexBTracks**(bool *do_copy*)

Specify how the secondary position of each new particle is chosen.

8.3 Action Functions

void **pAvoid**(float *magnitude*, float *epsilon*, float *look_ahead*, **PDomainEnum** *dtype*, float *a0* = 0.0, float *a1* = 0.0, float

void **pBounce**(float *friction*, float *resilience*, float *cutoff*, **PDomainEnum** *dtype*, float *a0* = 0.0, float *a1* = 0.0, float *a2* = 0.0, float *a3* = 0.0, float *a4* = 0.0, float *a5* = 0.0, float *a6* = 0.0, float *a7* = 0.0, float *a8* = 0.0)

Bounce particles off a domain of space.

void **pCopyVertexB**(bool *copy_pos* = true, bool *copy_vel* = false)

Set the secondary position from current position.

void **pDamping**(float *damping_x*, float *damping_y*, float *damping_z*, float *vlow* = 0.0, float *vhigh* = P_MAXFLOAT)

Dampen particle velocities, with threshold.

void **pExplosion**(float *center_x*, float *center_y*, float *center_z*, float *velocity*, float *magnitude*, float *stdev*, float *epsilon* = P_EPS, float *age* = 0.0)

An Explosion.

void **pFollow**(float *magnitude* = 1.0, float *epsilon* = P_EPS, float *max_radius* = P_MAXFLOAT)

Accelerate toward the next particle in the group.

void **pGravitate**(float *magnitude* = 1.0, float *epsilon* = P_EPS, float *max_radius* = P_MAXFLOAT)

Accelerate each particle toward each other particle.

void **pGravity**(float dir_x, float dir_y, float dir_z)

Accelerate particles in the given direction.

void **pJet**(float *center_x*, float *center_y*, float *center_z*, float *magnitude* = 1.0, float *epsilon* = P_EPS, float *max_radius* = P_MAXFLOAT)

Accelerate particles that are near the center of the jet.

void **pKillOld**(float *age_limit*, bool *kill_less_than* = false)

Remove old particles.

void **pMatchVelocity**(float *magnitude* = 1.0, float *epsilon* = P_EPS, float *max_radius* = P_MAXFLOAT)

Modify each particle's velocity to be similar to that of its neighbors.

void **pMove**()

Move particle positions based on velocities.

void **pOrbitLine**(float *p_x*, float *p_y*, float *p_z*, float *axis_x*, float *axis_y*, float *axis_z*, float *magnitude* = 1.0, float *epsilon* = P_EPS, float *max_radius* = P_MAXFLOAT)

Accelerate toward the closest point on the given line.

void **pOrbitPoint**(float *center_x*, float *center_y*, float *center_z*, float *magnitude* = 1.0, float *epsilon* = P_EPS, float *max_radius* = P_MAXFLOAT)

Accelerate toward the given center point.

void **pRandomAccel**(**PDomainEnum** *dtype*, float *a0* = 0.0, float *a1* = 0.0, float *a2* = 0.0, float *a3* = 0.0, float *a4* = 0.0, float *a5* = 0.0, float *a6* = 0.0, float *a7* = 0.0, float *a8* = 0.0)

Accelerate particles in random directions.

void **pRandomDisplace**(**PDomainEnum** *dtype*, float *a0* = 0.0, float *a1* = 0.0, float *a2* = 0.0, float *a3* = 0.0, float *a4* = 0.0, float *a5* = 0.0, float *a6* = 0.0, float *a7* = 0.0, float *a8* =

0.0)

Immediately replace position with a position from the domain.

void **pRandomVelocity**(**PDomainEnum** *dtype*, float *a0* = 0.0, float *a1* = 0.0, float *a2* = 0.0, float *a3* = 0.0, float *a4* = 0.0, float *a5* = 0.0, float *a6* = 0.0, float *a7* = 0.0, float *a8* = 0.0)

Immediately replace velocity with a velocity from the domain.

void **pRestore**(float *time_left*)

Over time, restore particles to their secondary positions.

void **pSink**(bool kill_inside, **PDomainEnum** *dtype*, float *a0* = 0.0, float *a1* = 0.0, float *a2* = 0.0, float *a3* = 0.0, float *a4* = 0.0, float *a5* = 0.0, float *a6* = 0.0, float *a7* = 0.0, float *a8* = 0.0)

Kill particles with positions on wrong side of the specified domain.

void **pSinkVelocity**(bool kill_inside, **PDomainEnum** *dtype*, float *a0* = 0.0, float *a1* = 0.0, float *a2* = 0.0, float *a3* = 0.0, float *a4* = 0.0, float *a5* = 0.0, float *a6* = 0.0, float *a7* = 0.0, float *a8* = 0.0)

Kill particles with velocities on wrong side of the specified domain.

void **pSource**(float *particle_rate*, **PDomainEnum** *dtype*, float *a0* = 0.0, float *a1* = 0.0, float *a2* = 0.0, float *a3* = 0.0, float *a4* = 0.0, float *a5* = 0.0, float *a6* = 0.0, float *a7* = 0.0, float *a8* = 0.0)

Add particles in the specified domain.

void **pSpeedLimit**(float *min_speed*, float *max_speed* = P_MAXFLOAT)

Clamp each particle's speed to the given min and max.

void **pTargetColor**(float *color_x*, float *color_y*, float *color_z*, float *alpha*, float *scale*)

Change color of all particles toward the specified color.

void **pTargetSize**(float *size_x*, float *size_y*, float *size_z*, float *scale_x* = 0.0, float *scale_y* = 0.0, float *scale_z* = 0.0)

Change sizes of all particles toward the specified size.

void **pTargetVelocity**(float *vel_x*, float *vel_y*, float *vel_z*, float *scale*)

Change velocity of all particles toward the specified velocity.

void **pVertex**(float *x*, float *y*, float *z*)

Add a single particle at the specified location.

void **pVortex**(float *center_x*, float *center_y*, float *center_z*, float *axis_x*, float *axis_y*, float *axis_z*, float *magnitude* = 1.0, float *epsilon* = P_EPS, float *max_radius* = P_MAXFLOAT)

Swirl particles around a vortex.

8.4 Action List Functions

void **pCallActionList**(int *action_list_num*)

Apply the specified action list to the current particle group.

void **pDeleteActionLists**(int *action_list_num*, int *action_list_count* = 1)

Delete one or more consecutive action lists.

void pEndActionList()

End the creation of a new action list.

int **pGenActionLists**(int *action_list_count* = 1)

Generate a block of empty action lists.

void **pNewActionList**(int *action_list_num*)

Begin the creation of the specified action list.

9 Bibliography

Allen, M. B. Flow - a particle animation application. *http://www.dnai.com/~mba/software/flow/*, 1999.

Beazley, D. Python Essential Reference, New Riders Publishing, 1999.

Conway, M., S. Audia, et al. "Alice: Lessons Learned from Building a 3D System for Novices". *Proc. of CHI 2000*, The Hague, The Netherlands, April 2000, 2000.

Eyles, J., S. Molnar, et al. "PixelFlow: The Realization". *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, Los Angeles, CA, August, 1997.

Leech, J. P. and R. M. Taylor. "Interactive Modeling Using Particle Systems". *Proc. of 2nd Conference on Discrete Element Methods*, MIT, 1993.

McAllister, D. K. Particle System API Home Page. *http://www.cs.unc.edu/~davemc/Particle*, 1999.

Neider, J., T. Davis, et al. OpenGL Programming Guide, Addison Wesley, 1993.

Reeves, W. T. "Particle Systems - A Technique for Modeling A Class of Fuzzy Objects". *Proc. of SIGGRAPH '83*, Detroit, Michigan, July, 1983.

Reeves, W. T. and R. Blau. "Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems". *Proc. of SIGGRAPH '85*, San Francisco, California, July, 1985.

Reynolds, C. W. "Flocks, Herds, and Schools: A Distributed Behavioral Model". *Proc. of SIGGRAPH '87*, Anaheim, California, 1987.

Reynolds, C. W. "Not Bumping Into Things - Developments in Physically-Based Modeling course notes". *Course Notes of SIGGRAPH '88*, Atlanta, Georgia, August, 1988.

Ross, S. A First Course in Probability. New York, New York, MacMillan College Publishing, 1976.

Sepúlveda, M. A. What is OpenGL? *http://mercury.chem.pitt.edu/~tiho/LinuxFocus/English/January1998/article2.html*, 1998.

Silicon Graphics, I. OpenGL: The Industry's Foundation for High-Performancs Graphics. *http://www.opengl.org/Downloads/OpenGL_Datasheet.pdf*, 1998.

Sims, K. "Particle Animation and Rendering Using Data Parallel Computation". *Proc. of SIGGRAPH '90*, Dallas, Texas, August, 1990.