Particle Animation and Rendering Using Data Parallel Computation

Karl Sims

Optomystic, 725 N. Highland, Hollywood, CA 90038
Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142

1 Abstract

Techniques are presented that are used to animate and render particle systems with the Connection Machine CM-2, a data parallel supercomputer. A particle behavior language provides an animator with levels of control from kinematic spline motions to physically based simulations. A parallel particle rendering system allows particles of different shapes, sizes, colors and transparencies to be rendered with anti-aliasing, hidden surfaces, and motion-blur. One virtual processor is assigned to each primitive data element: one to each particle, and during the rendering process, one to each pixel-sized particle fragment, and to each pixel. These tools are used to model dynamic phenomena such as wind, snow, water, and fire.

2 Introduction

As computers become a more practical tool for visual expression, modeling and animation systems need to allow for more abstract, high level instructions rather than requiring each object and each motion to be specified individually. Commands such as "make a gust of wind," "drop this object," "grow a tree," or even "make this character walk," are beginning to become feasible [2,6,13,18,21,25,31,32].

Particle systems provide for the creation of complex structure and motion from a relatively brief abstract description. They can be used to produce dynamic and "fuzzy" effects that are difficult to achieve with traditional objects made of surfaces and animated with non-procedural motion [19]. They have previously been used to model fire in the *Genesis Effect* of *Star Trek II* [14], trees and grass such as those shown in *Andre and Wally B.* [11,20], breaking waves [5,15], fireworks [30], and other abstract effects in *Systeme Particulier* [26]. A 2D particle system was used as part of a fluid simulation for Jupiter's surface in *2010* [33]. The flocks and schools in *Breaking the Ice* might also be considered as particle systems where each particle is a complex object [27,22].

The Connection Machine (R) CM-2 is a data parallel computer consisting of between 4K and 64K processors with up to 32K bytes of memory per processor, and floating point hardware [8,9,23,29]. A hypercube connection architecture and special routing hardware allows general communication between processors. A virtual processor mechanism is used to simulate more processors than the physical number so the virtual machine size can vary depending on the number of data elements in the application. For example, if 32K particles were created on a CM-2 with 8K physical processors, a virtual processor set with 4 virtual processors per processor could contain all 32K particles, one per virtual processor. Sets of virtual processors can be configured into n-dimensional grids. For example, a 1D virtual processor set would be used to represent particles, but a 2D virtual processor set would be used to represent the pixels of an image.

Programming a Connection Machine system is similar to programming a single processor except that the thousands of processors all execute the same program at the same time, each on different data. A parallel language called Starlisp is used in this work [10]. Starlisp is a parallel extension of Lisp that allows the same power of combination and abstraction as Lisp, but consists of parallel instructions that operate on parallel variables.

Because of the parallel nature of particle systems, they are well suited for highly parallel computation. A parallel language is convenient to use in building a particle animation language, and conversely this system is a good example of some basic data parallel techniques. Instructions or rules of behavior are described as if addressing a single particle, but they are applied to all particles (or a subset of them) in parallel. The result of the instruction will usually be different for each particle because it uses the state of the particle to determine its effect.

The three main sections of this paper describe a language containing some basic tools for animating particles, a system for rendering 3D particles, and finally, some specific applications that demonstrate how the animation and rendering is used to produce some natural phenomena effects.

3 Particle Animation: A Particle Behavior Language

Although many applications of particle animation will still require their own special software, a general set of tools is used to create a wide variety of effects. Starlisp and Lisp provide an environment that allows existing particle operations to be easily combined into new higher level operations.

Physical simulations can create motion that is much more complex and realistic looking than motion achieved by moving objects along spline curves or through keyframes [1,3,7,12,16,28]. Objects animated kinematically often are not perceived as dynamically correct, whereas objects animated by true physical simulation will look correct. However, ending up with a desired motion by specifying only forces and ac-

celerations can be very difficult. Just as in reality, where it is hard to toss a coin and make sure it lands heads up, or make a legged robot run without falling over, it is difficult to predict the motion resulting from applied forces.

This particle animation system attempts to supply several levels of operations along the spectrum between detailed kinematic control and physically based simulation. The goal is not to strictly obey physics and reality, but to provide a range of tools that allow a variety of effects to be easily created that appear correct.

The equations of motion for a particle in $\Re^3$ with position P, velocity V and an externally exerted acceleration A are:

$$V = V_0 + \int A dt$$

$$P = P_0 + \int V dt$$

Euler's method of integration allows the state of the particle to be updated using a simple approximation of these equations for a small discrete time interval Δt:

$$V' = V + A\Delta t$$

$$P' = P + \frac{V + V'}{2}\Delta t$$

Although other methods of integration are known to be more computationally efficient, Euler's was chosen for its simplicity and is usually sufficient.

A Particle is created by allocating a new CM virtual processor to contain the information about that particle. Particles can either be newly created, or created by duplicating existing particles and copying their state. Particles can be killed and removed from their processors to make those processors available for new particles.

Each virtual processor in the CM representing a particle contains a data structure whose elements are the particle's state variables. A particle has both a head position and a tail position. The head position is usually animated and the tail position follows along for motion blurring. The particle structure also contains other variables such as velocity, radius, color, and opacity.

Animation operations on particles can either initialize or alter the position or velocity of particles. In a purely physical simulation, these values would first be initialized, then, for each time interval Δt, the velocity would be altered by external accelerations such as gravity, and finally the position would be updated as shown above. However, for kinematically controlled motion, the position may be set directly, regardless of the previous position or velocity. It is also sometimes useful to set the position relative to the previous position or to alter the velocity in ways other than applying a simple translational acceleration.

Operations used to move particles are divided into four categories: those that set the position, those that set the velocity, those that alter the position or "apply" a velocity, and those that alter the velocity or apply an acceleration. Some examples of each are given in the following sections.

3.1 Position Operations

The positions of particles can be set in the following ways:

- Randomly within a rectangular solid.
- Randomly within a sphere.
- Randomly on the surface of a sphere.
- One particle on each vertex of a polygonal object.
- Randomly on the surface of a polygonal object.
- 3D Transformation from the local coordinates.

These operations are usually performed only once at the beginning of a particle's life, except the 3D transformation operation which is usually performed at every frame if used.

3.2 Velocity Initialization Operations

The velocity is usually initialized only once at the start of a particle's life unless jerky motion is desired. The velocity of particles can be initialized in much the same ways that the position can be initialized. The commonly used velocity initialization operations are:

- In a constant direction.
- Randomly within a rectangular solid.
- Randomly within a sphere.
- Randomly on the surface of a sphere.

3.3 Applied Velocity Operations

An "applied" velocity alters the position parameter of the particle depending on the previous position and affects the apparent velocity, but does not change the velocity parameter of the particle. This allows velocity operations and acceleration operations to act independently on particles without interfering with each other, and can help enable combination of dynamic simulation with other motion. For example, particles falling due to gravity might also be moved randomly from side to side using applied velocity operations.

Operations that are used to alter the position or "apply" a velocity are:

- Translate by a constant.
- Rotate by a constant.
- Scale by a constant.
- Translate Randomly.
- Vortex.

3.3.1 The Vortex

The vortex operation is worth further description. The parameters given to it are: *axis* of rotation, *center*, *magnitude*, and *tightness*.

The positions of particles are rotated about the axis through the center of the vortex by an amount dependent on their distance from the center. Higher tightness causes the angle of rotation, θ, to fade more quickly in the radial direction:

$$\theta = \frac{magnitude}{R^{tightness}}$$

where R is the distance from the center of rotation, and *tightness* is usually between 1.0 and 2.0.

Other options to the vortex operation are useful. A range of influence can cause θ to decrease to zero beyond a certain distance, and a translation along the vortex axis that is also dependent on R can create tornado-like motion.

This operation by no means creates a physical simulation of a vortex, but it is much easier to create specific dynamic fluid-like motions using choreographed vortices than it would be to simulate fluid flow through complex physical equations. There is often a tradeoff between animator control and physically accurate simulations. This vortex operation is an example of sacrificing some physical correctness in favor of animator control, while still allowing realistic looking motions to be achieved.

3.4 Acceleration Operations

Acceleration operations alter the particle's velocity. Forces can be converted to accelerations by: $A = F/m$. "Accelerations" can increment, scale, rotate, or reflect the velocities of particles. These operations usually use other parameters of the particle such as position, velocity, spiral-axis, or mass to find the acceleration and adjust the velocity. They can produce a wide variety of interesting and dynamically correct-looking motions. Some examples of acceleration operations are:

- Constant acceleration (gravity).
- Random acceleration.
- Accelerate towards a point (orbit).
- Accelerate towards a line.
- Accelerate towards the local coordinates.
- Damp
- Undamp
- Spiral.
- Bounce off a plane.
- Bounce off a sphere.

The first five of these are basic translational accelerations where the velocity is simply incremented by the acceleration: $V' = V + A\Delta t$. The acceleration can be constant, random, or may be directed towards a point or a line, and the magnitude of acceleration may depend on the distance of the particle's position from the point or line. For example an acceleration:

$$A = gm_o \frac{O - P}{|O - P|^3}$$

will create inverse-square law orbits where O is the fixed center of the orbit with mass m_o, g is a constant, and P is the particle position. This equation is a form of Newton's $F = gm_1m_2/r^2$. (In this example the acceleration can change very rapidly near O, which, unless Δt is very small, causes errors that fling the particles out of their orbits.)

3.4.1 Damping

A simple approximation to damping is used to simulate effects such as air friction on particles. A deceleration proportional to the current velocity magnitude is applied to the particle in the direction of the current velocity. A damping parameter d typically ranges between 0.0 and 10.0. The damping deceleration is clamped so as not to reduce the velocity below a given threshold.

$$V' = V\ max(\ 1 - d\Delta t\ ,\ min(1.0, \frac{threshold}{|V|}))$$

A more physically accurate model for damping could be implemented with damping forces non-linearly related to velocity, and not necessarily proportional to mass.

Undamping is used to cause acceleration instead of deceleration in the direction of the current velocity ($d < 0$), and can be performed on particles below a threshold value to smoothly accelerate them to some minimum speed.

3.4.2 The Spiral

Spiral motions contribute to many different effects such as swirling fire or twirling snowflakes. Each particle to be spiraled is given a spiral axis which can be initialized using the same set of methods for initializing velocities shown above. The spiral operation causes the velocity vector to be rotated

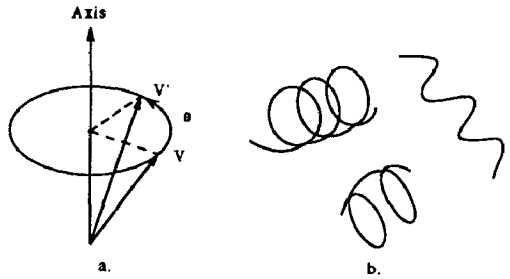

Figure 1: Spiral.

about the spiral axis. For a given spiral speed, s, the velocity is rotated by an angle $\theta = s\Delta t$, for each time interval Δt. [See figure 1a].

A particle can move in a variety of helix shaped paths depending on the relative angle of its velocity to its spiral axis [figure 1b]. If they are perpendicular the particle will remain within a circle. If they are parallel there will be no effect. Notice that when particles are spiraling, they move in the general direction of plus or minus their spiral axes.

3.4.3 The Bounce

Particles can be bounced off primitive shapes made of planes and spheres. A simple bounce with no energy loss could be performed by just reflecting the velocity of particles that have passed beyond the boundary of the surface by the normal N:

$$V' = V - 2(V \cdot N)N$$

This method allows particles to penetrate the surface for at least one iteration, and the effective position they bounce from is usually slightly below the surface unless many iterations are calculated per frame.

A more complete bouncing method considers friction and resilience of the particle and sets the new position and velocity as if the bounce occured exactly on the surface. All other operations are performed before any bounce operations, and the positions are updated by the new velocities. Then, particles are tested and bounced off any surfaces that they have penetrated. If a particle has penetrated a surface a bounced-flag is set, and the velocity is broken into its normal and tangential components, V_n and V_t:

$$V_n = (V \cdot N)N$$

$$V_t = V - V_n$$

[See figure 2.] A friction parameter, μ, reduces the tangential component, and the normal component is reflected and scaled by a resilience parameter, ϵ, (both can range from 0.0 to 1.0):

$$V' = (1 - \mu)V_t - \epsilon V_n$$

If it is desirable to prevent particles from getting entirely stopped by friction, it is necessary to provide a velocity magnitude value below which friction has no effect.

Particles have both head and tail positions, P_h and P_t, to be used for motion blur. They are both flipped about the surface to account for the bounce. If S is any point on the surface:

$$P_h' = P_h - 2[(P_h - S) \cdot N]N$$

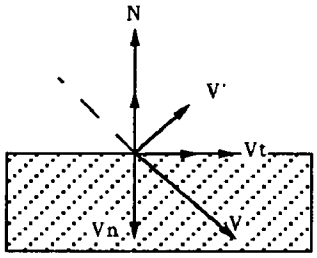

Figure 2: Bounce.

$$P_t' = P_t - 2[(P_t - S) \cdot N]N$$

The tail is also flipped because the particle renderer can not motion blur particles with kinks in their motion. Finally the tail is pulled up to the point of contact on the surface so it doesn't hang below the surface.

Bouncing off spheres is performed in the same way as bouncing off planes, except N and S are calculated for the closest point on the surface of the sphere to the particle. If C is the center of a sphere of radius r:

$$N = \frac{P_h - C}{|P_h - C|}$$

$$S = C + rN$$

More accurate physical models for bouncing could probably be implemented, but this method is sufficient for creating reasonable bouncing effects.

3.5 Particle Animation Summary

Particles have state variables in addition to position and velocity that are used by some animation operations but not by others. For example: type, age, mass, spiral-axis, color, opacity, and size can be used. Other spare slots exist for information such as initial velocity, a color to fade to, or an age to die at.

A valuable component of this particle animation system is a particle preview capability. Particles can be animated as shown above, and viewed with a quick vector display at near real time speeds. Fast turnaround time for particle motion experimentation is very helpful.

An outline of an animation loop for creating particle motion is as follows:

```
Create particles.
Initialize particles.
For each frame:
    Set tails to previous heads.
    For each simulation time increment:
        Select subset of particles.
            Perform operations.
        Update positions using velocities.
        Select subset of particles.
            Perform bounce operations.
    Adjust tails for motion blur shutter speed.
    Render or preview.
```

4 Particle Rendering

This section describes a data parallel method used to render large numbers of anti-aliased, motion blurred particles of variable sizes, colors, and transparencies.

Every particle has a head and a tail, and the following parameters are passed from the particle animation system to the renderer for both the head and the tail of each particle:

- position (x, y, z)
- radius
- color (r, g, b)
- opacity

[See figure 3a.]

Motion blur is accomplished by linearly blurring each particle independently. The animation system sets the head and tail appropriately for the desired shutter speed. The renderer produces a blurred streak for each particle such that all the parameters above are interpolated between the head and the tail.

Alternatively, the ability of the radius, color, and opacity to vary between the head and tail of particles, can allow some variety in particle shape, such as might be used to approximate comets, sparks, or water droplets. [See figure 7b.]

Particles occupy an area in which the opacity falls off from 1.0 at the center to 0.0 at the extremes. The function that determines the falloff of the opacity can vary, and is used to perform out-of-focus or blurry effects as well as spatial anti-aliasing. Linear or Gaussian shapes are usually used.

The rendering system first transforms the particles' head and tail positions and radii into screen coordinates. Then it dices the particles into fragments (in two stages) such that for each particle there is one fragment for every pixel that it will affect. These fragments containing color, opacity, and depth are then sorted by depth and the final pixel colors are calculated. This method has similarities to a simple a-buffer polygon rendering algorithm [4], but does not use coverage masks and does not perform texturing or lighting.

An overview of the rendering process is given below:

```
Update particles (animation).
Transform to screen coordinates.
For each horizontal patch:
    Determine effective patch height.
    Dice particles into spans.
    For each vertical patch:
        Determine effective patch width.
        Dice spans into fragments.
        Sort fragments by pixel and depth.
        Perform hidden surface calculation.
        Send colors to pixels.
        Add background color.
        Output pixels.
```

4.1 Dicing into Fragments

The particle animation operations described above only require a single data type, the particle, to exist in a virtual processor set within the CM. The rendering system, however, generates multiple data types as it proceeds. Particle spans, particle fragments, and pixels each are created in virtual processor sets with one data element per virtual processor.

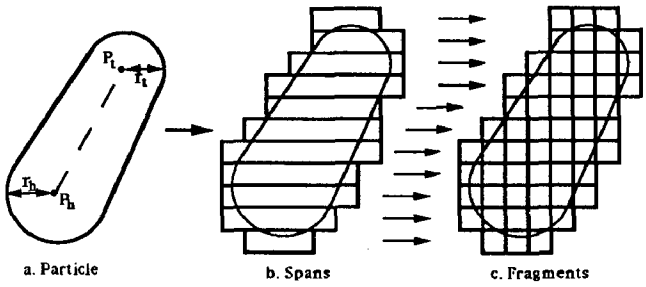

Figure 3: Particle Dicing.

First, each particle processor determines the number of scan lines that the particle will occupy and the particle is diced into spans. Multiple span processors are allocated for each particle processor, and the particle information is sent to them.

Then, each span processor similarly determines the number of fragments it will occupy, allocates fragment processors and sends the particle information to them.

When particles are diced into fragments, there are often more total fragments than will fit into Connection Machine memory. To compensate for this, the image is rendered in subsections or patches. The size of each patch is adjusted such that all the fragments in that patch will fit into memory at once.

The patch height is chosen before allocating span processors such that the number of span processors will not exceed a limit. Likewise, for that horizontal section of the image, patch widths are chosen such that the number of fragment processors will not exceed a limit for each patch.

It is also desirable to fill as many fragment processors as possible up to the limit since empty processors sit idle while the others compute. The processor usage "efficiency" for a given patch size is the total number of data elements (spans or fragments) within the patch divided by the maximum number permitted. The false position method [17] is used to search for patch widths and heights that result in an efficient use of span and fragment processors.

After the fragment processors are set for a given patch size, the color, opacity and depth of that fragment are calculated by finding the closest point to the fragment on the line between the head and tail of the particle. The particle radii, depths, colors and opacities are interpolated between the head and tail to give values at that point. The final fragment opacity is set as a function of the interpolated opacity, the interpolated radius, and the distance from the center of the particle.

Spatial anti-aliasing of particles is performed by ramping the opacity to zero near the edges of the particle. Radii below one pixel are clamped to one pixel and the opacity is lowered to compensate to prevent aliasing due to sub-pixel sized particles. Coverage-masks or multiple samples per pixel were not used because unlike polygons whose edges often touch each other to make a continuous surface, the edges of particles usually do not line up.

4.2 Sorting Fragments and Calculating Transparency

After the color, opacity and depth are calculated for all the fragments in the image patch being rendered, the fragments are reordered within the CM to make all the fragments covering the same pixel adjacent to each other. They are also sorted within the pixel groups by depth.

The parallel operator *scan* operates on values in 1D arrays of processors. It allows each processor to receive the sum (or product) of all the values in the preceding processors. A segmented scan can be performed on groups of processors to prevent the results from spreading beyond the group. Local sums and products can be efficiently calculated for groups of processors of variable sizes.

Within each pixel group *scan with multiply* is applied to the fragment's transparencies from front to back to determine the total pixel contribution of each fragment. Then, *scan with add* is applied to each color component and the opacity, each scaled by the fragment's pixel contribution. The last processor in each pixel group receives the final pixel color and opacity.

The pixel colors are then sent to processors in the 2D image patch virtual processor set. Each virtual processor representing one pixel of the image patch being rendered receives the final color and opacity from the fragments covering it, if any. Background color is added if necessary, and the patch is finally output to a frame buffer or file.

4.3 Mixing with Other Data Types

A recent addition to this system allows particles to be mixed with other renderable data types such as polygons. The fragments of both particles and polygons are depth sorted together and rendered simultaneously. This permits any number of layers of particles and polygonal objects to move in front of and behind each other with correct hidden surfaces. This can be preferable to rendering the different data types separately and then compositing the images together afterwards.

5 Results

The animated film *Particle Dreams* [24] was created entirely using the animation and rendering tools described above. It contains orbiting fire, an explosion, a snow storm, a crashing head, and a waterfall. These tools are also being used in a commercial production environment to create "burning logos," galaxies, and other effects.

5.1 Snow and Wind

A snow storm was created using white snowflake particles, spirals, and vortices. Some snowflakes were created and dropped above the field of view at each iteration. They were given an initial velocity and spiral axis straight down but with some random variation, and were bounced off the plane of the ground with zero bounce and high friction, so that once they hit, they stuck. [Figure 6a.]

Gravity and air friction were not considered because the air friction damping and gravity would have canceled out at a steady critical velocity.

Gusts of wind were made by moving pairs of vortices across the field of view. A gust procedure was built from two vortex operations so that gusts could be moved between given start and end positions. Gusts were choreographed and tested until the desired swirling effects were achieved.

Finally, "splat" shapes were created by duplicating particles into several particles when they hit a vertical plane.

Figure 4: Water.

5.2 Falling Water

A waterfall was simulated by applying gravity to blue particles and bouncing them off obstacles made of planes and spheres.

Some water droplet particles were created on each iteration randomly within an area at the top of the waterfall. When particles flowed over the last edge at the bottom of the waterfall they were recycled back to the top of the waterfall. Around 60K particles were used for this animation.

Splashes were achieved by placing spherical rocks of different sizes in the path of the flow. The droplets were bounced off the rocks with friction and resilience that varied randomly within a range. When a bounce was detected, the particles were turned from blue to white and then faded slowly back to blue as they fell to the next rock. The variety of blue to white particles gave the waterfall a sparkling quality without any actual lighting calculations.

Motion blur was exaggerated in this sequence: the shutter speed was slightly more than the entire frame duration to give the flow a smoother look.

5.3 Fire

Fire simulation is a more complex effect that can be created using these tools. Extensions to the utilities described above allow arbitrary polygonal objects to be burned.

First, a large number of particles are created with their initial positions located on the surface of the object. This is done by triangulating the polygons of the object and creating some particles randomly within each triangle. To give an even distribution, the number created in each triangle is proportional to the triangle's area. The particles' initial velocities and spiral axes are set to directions between the object's surface normal and the up-most surface tangent vector to cause the fire particles to hug the surface somewhat before curling up.

Second, several groupings of the particles are created, and parameters of color and motion are set to be the same or nearly the same within the groups. Particles are grouped in small regions with similar colors, so different regions of the surface emit different colored flames as if some regions are hotter than others. The particles are also grouped into flickers. Each particle in a flicker group is given a similar spiral axes, initial velocity, start time, and life duration, but with slight variation, so that each flicker had some coherency and was perceived as a unit, rather than each particle being independently random.

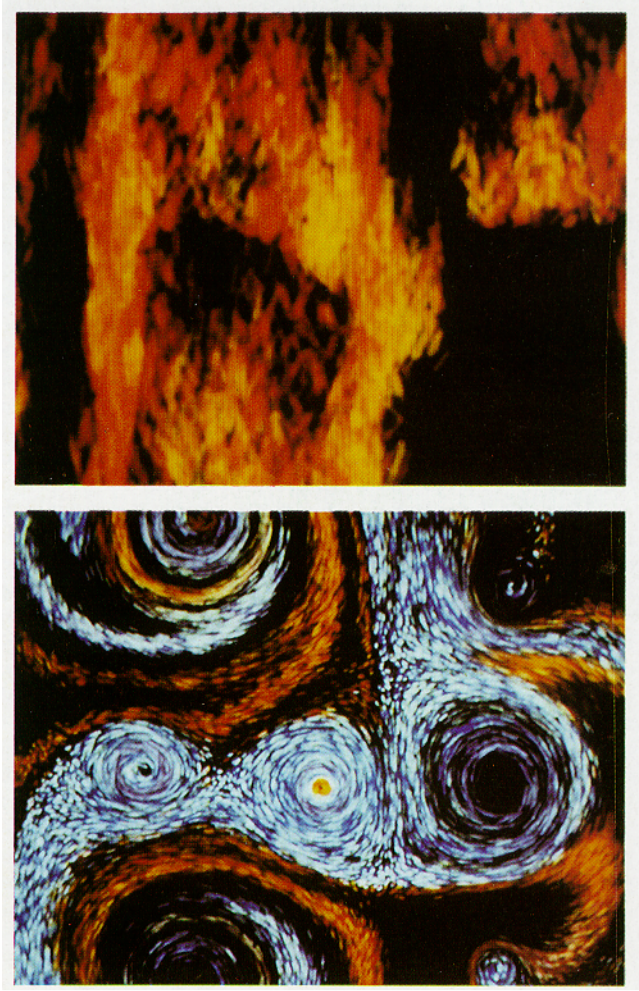

Figure 5: (a) Burning Letters, (b) Vortex Field.

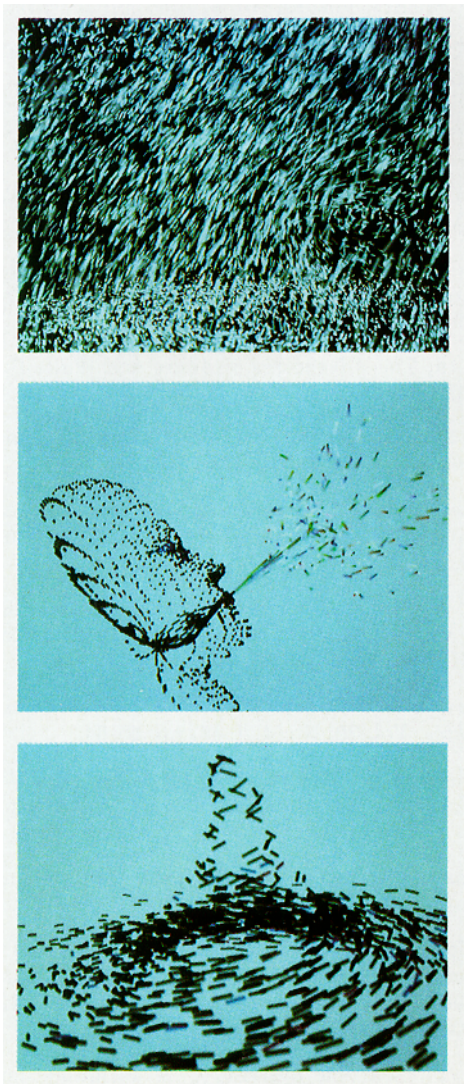

Figure 6
(a) Snowstorm with vortex gust.
(b) Self Breathing Head.
(c) Inverted Tornado.

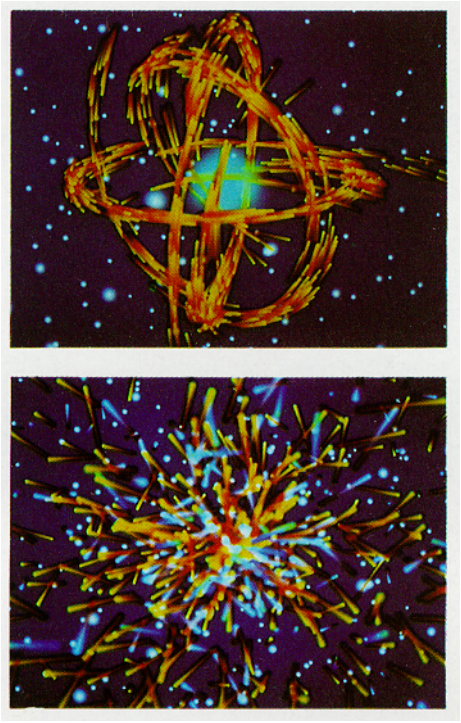

Figure 7: *Left*
(a) Orbiting Fire.
(b) Explosion.

Figure 8: Waterfall.

The fire particles leave the surface and spiral upward while changing color. After they fade and die, they are recreated again at the initial position on the surface to start another cycle. The spiral axis slowly rotates to prevent duplicate motion, and the flickers have slightly different frequencies to create a pseudo-random rhythm that natural fire can have.

6 Conclusion

Some general tools for animating and rendering particle systems are implemented that permit both kinematic and dynamic control of particles. They are used to create effects that would probably be difficult to achieve using traditional techniques, but there are still many potential additions to this set of particle system utilities.

Future work in this area might include operations that cause particles to influence each other: N-body types of simulations might be used for galaxy simulations, more natural fluid motion, or collision avoidance. In the current implementation particles ignore each other and only follow global rules, sometimes resulting in interpenetration.

More efficient collision detection of surfaces would be beneficial. Currently every particle is tested against every surface element. The ability to create procedural motion for more complex objects (other than particles) including rigid body dynamic simulations would also be desirable.

It would be interesting to compare the parallel speed of particle rendering with that of a serial computer. This was not done because of the unique parallel software implementation. Rendering speed is approximately proportional to the number of processors, and inversely proportional to the number and sizes of the particles. Frame times commonly vary from several seconds to several minutes.

Since data parallel computers have potential for growth in both the speed of processors and the number of processors, they should become more powerful and more available in the future. Techniques that permit computer animation of complex structure and motion automatically and can utilize data parallelism, such as those presented here, may soon be more frequently used.

7 Acknowledgments

Thanks to Lew Tucker for continuing support. Thanks to all the folks at Whitney/Demos Productions for a unique learning experience. Thanks to Thinking Machines Corporation for building Connection Machines computers and being generous. Thanks to Jim Salem, Brewster Kahle, Gary Oberbrunner, and Peter Schroeder for discussions and encouragement. Thanks to J.P.Masser, Jeff Mincy, and Cliff Lasser for Starlisp and its support. Thanks to Arlene Chung and Debbie Mahe for layout and figures. And finally, thanks to John Whitney Jr., Jerry Weil, and Optomystic for the environment to put this work to use.

References

[1] Armstrong, W., Green, M., "The Dynamics of Articulated Rigid Bodies for Purposes of Animation," *Proceedings Graphics Interface '85*, pp. 407-415.

[2] Amburn, P., Grant, E., Whitted, T., "Managing Geometric Complexity with Enhanced Procedural Methods," *Computer Graphics*, Vol. 20, No. 4, August 1986.

[3] Barr, A., Barzel, R., "A Modeling System Based on Dynamic Constraints," *Computer Graphics*, Vol. 22, No. 4, 1988, p. 179.

[4] Carpenter, L.C., "The A-buffer, an Anti-aliased Hidden Surface Method," *Computer Graphics*, Vol. 18, No. 3, 1984.

[5] Fournier, A., Reeves, W., "A Simple Model of Ocean Waves," *Computer Graphics*, Vol. 20, No. 4, 1986, pp. 75-84.

[6] Girard, M., Maciejewski, A., "Computational Modeling for the Computer Animation of Legged Figures," *Computer Graphics*, Vol. 19, No. 3, 1985, pp 263-270.

[7] Hahn, J. K., "Realistic Animation of Rigid Bodies" *Computer Graphics*, Vol. 22, No. 4, 1988, p. 299.

[8] Hillis, W. D., *The Connection Machine*, MIT Press, 1985.

[9] Hillis, W. D., "The Connection Machine," *Scientific American*, Vol. 255, No. 6, June 1987.

[10] Lasser, C., Massar, J.P., Mincy, J., Dayton, L., "Starlisp Reference Manual," Thinking Machines Corporation, 1988

[11] Lucasfilm Ltd, *The Adventures of Andre and Wally B.*, (film), August 1984.

[12] Miller, G., "The Motion of Snakes and Worms" *Computer Graphics*, Vol. 22, No. 4, 1988, p. 169.

[13] Oppenheimer, P. "Real time design and animation of fractal plants and trees. *Computer Graphics*, Vol. 20, No. 4, 1986, pp 55-64.

[14] Paramount, *Star Trek II: The Wrath of Kahn*, Genesis Demo, also in SIGGRAPH Video Review 1982, ACM SIGGRAPH, New York.

[15] Peachy, Darwyn R., "Modeling Waves and Surf," *Computer Graphics*, Vol. 20, No. 4, 1986, pp. 65-84.

[16] Platt, J., Barr, A., "Constraint Methods for Flexible Models," *Computer Graphics*, Vol. 22, No. 4, 1988, p. 279.

[17] Press, Flannery, Teukolsky, and Vetterling, *Numerical Recipes*, Cambridge University Press, 1986, p. 248.

[18] Prusinkiewicz, P., Lindenmayer, A., and Hanan, J., "Developmental Models of Herbaceous Plants for Computer Imagery Purposes," *Computer Graphics*, Vol. 22 No. 4, 1988, pp. 141-150.

[19] Reeves, W. T., "Particle Systems — A Technique for Modeling a Class of Fuzzy Objects," *ACM Transactions on Graphics*, Vol. 2, No. 2, April 1983, reprinted in Computer Graphics 1983, pp. 359-376.

[20] Reeves, W. T., and Blau, R. Approximate and probabilistic algorithms for shading and rendering structured particle systems. *Computer Graphics*, Vol. 19, No. 3, 1985, pp 313-322.

[21] Reffye, P., Edelin, C., Francon J., Jaeger, M., Puech, C. "Plant Models Faithful to Botanical Structure and Development," *Computer Graphics* Vol. 22, No. 4, 1988, pp 151-158.

[22] Reynolds, Craig W., "Flocks, Herds and Schools: A Distributed Behavioral Model," *Computer Graphics*, Vol. 21, No. 4, July 1987, pp 25-34.

[23] Simon, H.D., *Scientific Applications of the Connection Machine*, World Scientific Publishing Co., 1988.

[24] Sims, K., *Particle Dreams*, SIGGRAPH Video Review 1988, ACM SIGGRAPH, New York.

[25] Smith, A. R., "Plants, Fractals, and Formal Languages," *Computer Graphics*, Vol. 18, No. 3, pp. 1-10, July 1984.

[26] Studio Base 2, *Systeme Particulier*, Chesnais, Alain, SIGGRAPH Video Review 1987, ACM SIGGRAPH, New York.

[27] Symbolics, *Stanly and Stella in Breaking the Ice*, SIGGRAPH Video Review 1987, ACM SIGGRAPH, New York.

[28] Terzopoulos, D., Fleischer, K., "Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture," *Computer Graphics*, Vol. 22, No. 4, 1988, p. 269.

[29] Thinking Machines Corporation, *Connection Machine Model CM-2 Technical Summary*, technical report, May 1989.

[30] Weil, J., A T & T Bell Labs, *Boom Boom Boom*, SIGGRAPH Video review 1987, ACM SIGGRAPH, New York.

[31] Wilhelms, J. Barsky, B., "Using Dynamic Analysis for the Animation of Articulated Bodies Such as Humans and Robots," *Proceedings Graphics Interface '85*, pp. 97-104.

[32] Wilhelms, J., Moore, M., "Collision Detection and Response for Computer Animation," *Computer Graphics*, Vol. 22, No. 4, 1988, p. 289.

[33] Yaeger, L., Upson, C., "Combining Physical and Visual Simulation – Creation of the Planet Jupiter for the Film 2010," *Computer Graphics*, Vol. 20, No. 4, 1986, pp 85-93.

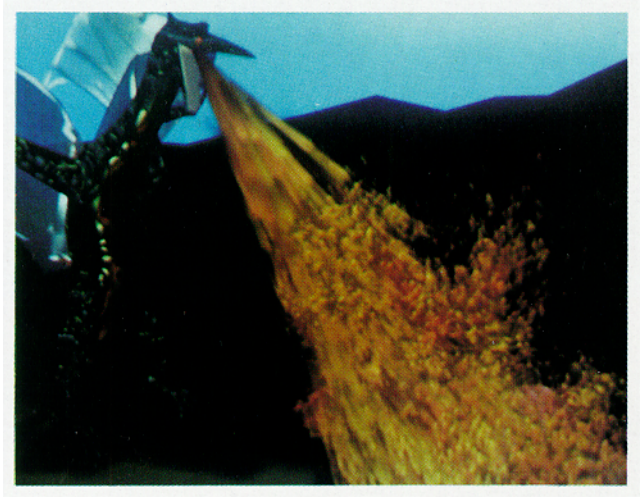

Figure 9: Fire Breathing Dragon. Fire was simulated with particle systems. Dragon by Jerry Weil, Optomystic.

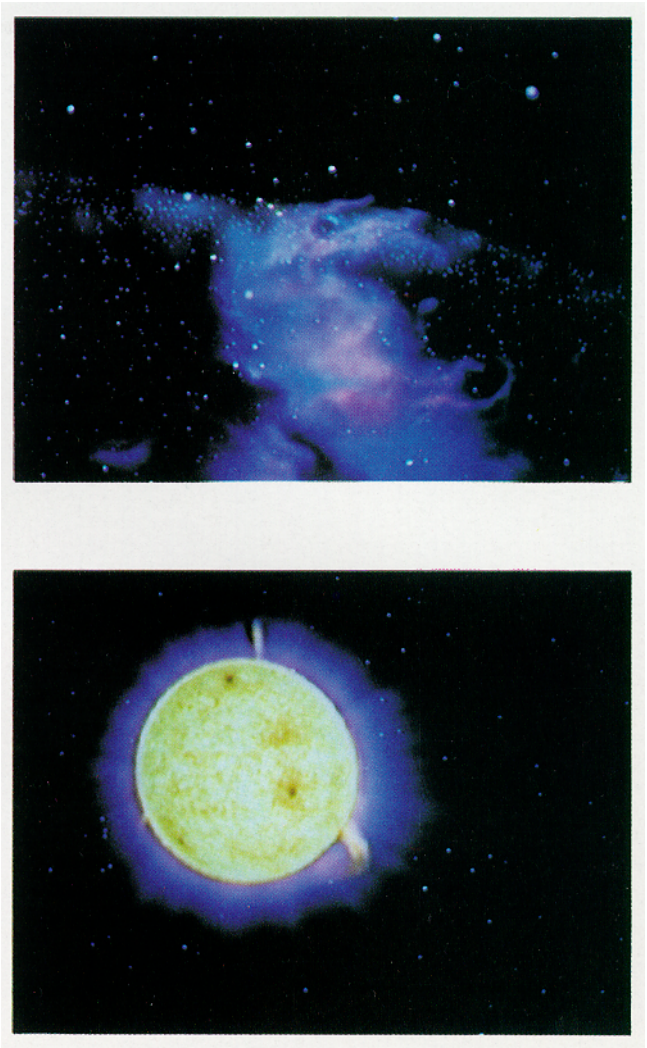

Figure 10: (a) Nebula, (b) Solar flyby. Created by Jerry Weil, Optomytic, for *Earth Day Special 1990*. (Nebula also contains surfaces with color and opacity texture mapping.)