Rapid: An Artist Friendly Particle System

Devon Penney*
PDI/DreamWorks Animation

Nafees Bin Zafar†
Oriental DreamWorks

Figure 1: *Examples of simulations from recent films. Property of DreamWorks Animation*

Abstract

A particle system is a heavily used tool in every effects animation department in the world. In this paper, we re-explore this well understood, and relatively ancient technique. We present our design of Rapid, a modern, and easy to use particle dynamics framework that seamlessly fits into procedural node graphs. We discuss schemes for minimizing sub-frame evaluation of the dataflow graph, including an improved collision model and simple methods for generating smooth and continuous particle emissions. These techniques combined with improved workflow tools enable artists to build production setups faster, and simulate them quicker. The Rapid framework has been used on hundreds of shots at the studio to produce effects like water, magic, debris, spray, dirt, embers, and sparks.

CR Categories: I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Simulation

Keywords: particle simulation, sparse volumes

*devon.m.penney@gmail.com
†nafees.bin.zafar@dreamworks.com

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Request permissions from Permissions@acm.org.
DigiPro '15, August 8, 2015, Los Angeles, California.
© 2015 ACM. ISBN 978-1-4503-3718-2/15/08…$15.00
DOI: http://dx.doi.org/10.1145/2791261.2791265

1 Motivation

Many effects animation systems, such as Houdini, Bifröst, and XSI ICE utilize a node based operator graph architecture. It provides a good balance of proceduralism, and a fast authoring process for users. From the point of view of the software, this design permits efficient definition and execution of the computation. Within this type of framework, our users express a strong preference for an imperative style of operation where each operator executes an action on the incoming data, and produces a modified result for the next operator. Any type of complex simulation system must be architected to live in this environment, and avoid placing restrictions on the idioms used by our artists.

A few years ago we discovered that a simple particle system was unnecessarily difficult to incorporate into our FX setups. We found particle tools, such as those implemented in Houdini's DOPs, have tedious workflows for things like setting up colliders, and also violated the rule of imperative execution. In addition, having simulations live in another context than generic procedural modeling often meant duplicating tool functionality so it could live in both environments. We set out to create a system that would allow easier use of particles, while allowing us to leverage existing procedural point manipulation tools to control simulations.

2 Framework

Our system, called Rapid, is implemented in SideFX's Houdini system. In our mental model for Rapid, we think of dynamics as procedural modeling operations. Thus, Rapid operators act as surface operators (SOPs) that modify particles and colliders. We made an explicit choice in using SOPs to optimize for a majority of use cases and restrict some of the functionality available in Houdini's dynamics context (DOPs) for the sake of ease of use.

2.1 Artist Workflow

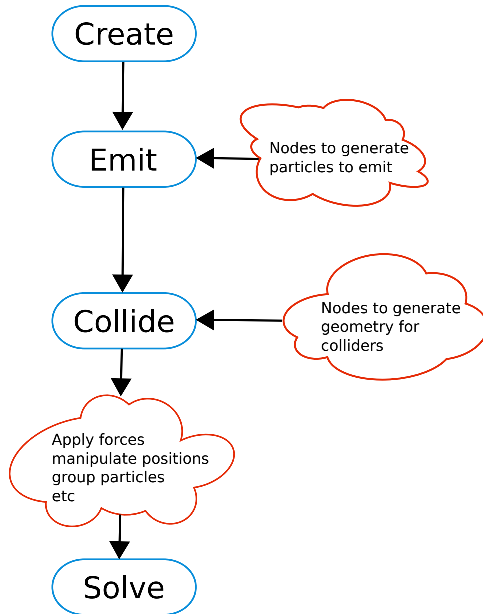

Figure 2: *A basic particle simulation chain.*

Artists set up and run simulations in SOPs, which leverages an extensive pre-existing toolset of point manipulation tools. The basic workflow consists of 4 nodes in a chain, inspired by the workflow used with DreamWorks' Flux gas solver [Vroeijenstijn and Henderson 2011]. See figure 2 for the visual layout of the chain. The nodes are as follows:

Create Creates a simulation context

Emit Emits particles

Collide Adds a collision object

Solve Integrates the particles for 1 frame

We specifically designed this workflow to give the user the impression of composing a "timestep". When the user advances the frame by one the "Solve" node integrates forces and velocities, resolves collisions, and performs time integration of time based attributes such as particle age. An implementation detail particular to Houdini SOPs is that the "Create" node stores a pointer to its data buffer as a "detail attribute", which is often called a "uniform" variable in CG parlance. The output of the solve is posted to the create node on the next frame by modifying this data buffer, thus enabling a feedback loop. Typically, artists utilize many point and level set manipulation tools to control simulations by adding nodes interspersed between the four listed above. Examples are generating forces, modifying collision geometry, and performing extensive group manipulations to do complex behaviors.

2.2 Data Representation

All points and colliders are represented as Houdini geometry point data and OpenVDB grids [Museth 2013]. This seamless integration into the host application allows users to utilize the entire ecosystem of operators available to them, including other solvers. From an engineering standpoint, avoiding a layer of abstraction reduces the overhead of data translation, and utilizes Houdini's copy-on-write data flow to minimize memory overhead. [Johansson 2013] presented a similar Houdini based system which utilizes custom data structures for points. We encourage readers to read his excellent write-up for a full understanding of the pros and cons.

2.3 Dataflow

When we consider the overall complexity of an effects setup, we quickly discover that the simulation control component is a very small portion of this.

A naive implementation requires increasing the number of graph evaluations proportional to the number of solver substeps. One of our primary design constraints was to reduce these expensive graph evaluations. The sub-frame information is necessary for features like birthing particles, and detecting collisions, thus we needed to devise custom motion interpolation schemes.

In order to make the system extensible, we also introduced the concept of "Action Operators" which allow developers to register callbacks that are invoked by the solver at a sub-frame rate. This is useful in cases where we need sub-frame evaluation, such as when developing our particle drag implementation [Yuksel et al. 2014], or custom force fields. This does not handle every imaginable case, however for common cases we noted significant gains in user efficiency.

3 Emission

3.1 Initial Age

Artists frequently express problems with emitting particles in continuous streams, which presents itself as spatial and temporal aliasing in the integrated particle motion. The simplest and most computationally expensive way to address this is by increasing the number of sub-frame timesteps. The root of this issue is that particles are being emitted continuously in time over the course of a frame, thus each particle is alive for a portion of that time. Consequently, the solver must take this fraction into account for time integration of motion. We address this by separating birthed particles from previously emitted ones, and integrating their velocity and position for the correct dt.

Given t_{birth}, the time a particle is born, and L_{birth}, the artist-supplied age for the born particle, we solve a position update for a particle:

$$p_{t+dt} = p_t + v_t * \gamma$$

Where

$$\gamma = \begin{cases} L_{birth} & t \leq t_{birth} \leq t + dt \\ dt & otherwise \end{cases}$$

We typically give artists a choice for the initial age of particles they emit if they do not explicitly specify it. There are three choices:

1. Born at the beginning of the frame
2. Born at the end of the frame
3. Born at a random time within a frame

Many times, artists want particles to be at their birth position at the end of the frame they are born on. For these cases, they would choose 2 from above to ensure that $dt = 0$ for the entire frame.

3.2 Fast Moving Emitters

In addition to integrating emitted particles, fast moving emitters often cause artifacts. The most common solution is to increase solver

steps until it resolves the emission shape's motion. There is a far more efficient technique of smearing the emission shape across the frame, where we use a cubic hermite spline, and its derivative for an inherited velocity.

$$p_{interp} = (1+2t)(1-t)^2 p_{f-1} + \alpha t (1-t)^2 (p_f - p_{f-2}) + t^2 (3-2t) p_f + \alpha t^2 (t-1)(p_{t+1} - p_{t-1}) \quad (1)$$

$$p\prime_{interp} = 6t^3 p_{f-1} + \alpha \left(1 - 4t + 3t^2\right)(p_f - p_{f-2}) - 6(t-1)\, t p_f + \alpha t (3t-2)(p_{t+1} - p_{t-1}) \quad (2)$$

Points are scattered on the emission mesh at frame f. Mesh vertex positions on the frames, p_f, p_{f+1}, p_{f-1}, and p_{f-2} are interpolated onto the particles based on the scattered position. The particle position is then placed on the emitter's motion path according to evaluating p_{interp} and assigning an inherited velocity based on $p\prime_{interp}$. To do this, each point is given a t on the interval $[0, 1]$ where 0 corresponds with $p_{interp} = p_{f-1}$, and 1 gives $p_{interp} = p_f$.

4 Collisions

We use narrow-band level sets, implemented in OpenVDB, in to represent collision objects. This use of level sets has been shown to be efficient, and reliable [Guendelman et al. 2003], [Allen et al. 2007]. The tree structure used by OpenVDB gives us hierarchical collision detection. Though it is trivial, for the sake of completeness we will describe our physically inspired collision response model [Witkin 2001]. Given a particle with velocity $\mathbf{v}_p$ and collision object with velocity $\mathbf{v}_c$, the relative speed in the normal direction between the two objects is:

$$\triangle v_n = \mathbf{n} \cdot (\mathbf{v}_p - \mathbf{v}_c)$$
$$\mathbf{n} = \frac{\nabla \phi}{||\nabla \phi||}$$

Where ϕ is the level set function, and $\mathbf{n}$ is the normal of the level set at the point of collision. A particle is considered to be colliding with an object if the particle is inside the object, and the two bodies are not separating. i.e. $sign(\triangle v_n) =< 0$.

The instantaneous "incoming" velocity of the particle can be decomposed as the sum of the non-separating normal component and the remainder:

$$\mathbf{v} = \mathbf{v}_n + \mathbf{v}_t$$
$$\mathbf{v}_n = (\mathbf{v} \cdot \mathbf{n})\mathbf{n}$$
$$\mathbf{v}_t = \mathbf{v} - \mathbf{v}_n$$

In our model the normal velocity is reflected across the collision plane. The collision event is considered to be frictionless, therefore the tangential velocities are unaffected. The ratio between the "outgoing" velocity and the incoming velocity is determined by the coefficient of restitution, ϵ.

$$\mathbf{v}_n^+ = -\epsilon \mathbf{v}_n \quad (3)$$
$$\mathbf{v}_t^+ = \mathbf{v}_t$$

The negative sign in (3) indicates geometric reflection. The superscript, $+$, indicates outgoing velocity.

Applying these relations we can derive an expression for the outgoing velocity in terms of the incoming velocities.

$$\begin{aligned} \mathbf{v}^+ &= \mathbf{v}_n^+ + \mathbf{v}_t^+ \\ &= -\epsilon \mathbf{v}_n + \mathbf{v}_t \\ &= -\epsilon (\mathbf{v} \cdot \mathbf{n})\mathbf{n} + \mathbf{v} - (\mathbf{v} \cdot \mathbf{n})\mathbf{n} \\ \mathbf{v}^+ &= \mathbf{v} - (1+\epsilon)(\mathbf{v} \cdot \mathbf{n})\mathbf{n} \quad (4) \end{aligned}$$

This last equation (4) is the total velocity of the particle after the collision.

4.1 Basic Collision Resolution

The most simple way to resolve collisions between particles and level sets is to project particles along the level set gradient when they are inside of a collider:

$$\mathbf{p} = \mathbf{p} + \mathbf{n} * \phi$$

Note that when ϕ is larger than the narrow-band half width, we cannot get an accurate $\mathbf{n}$ since the particle is inside a sparse region of the level set. In these cases, the particle is updated by backtracking along the velocity to the previous position.

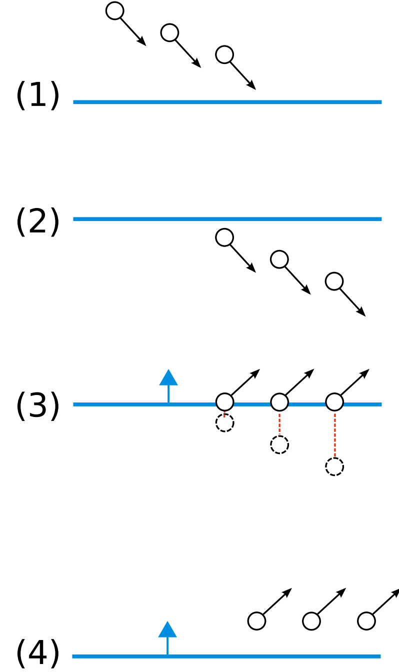

Figure 3: *Particle collisions with projecting along the gradient.* ***(1)*** *shows the particles at $t = 0$.* ***(2)*** *shows the particles at $t = 1$ before collision resolution.* ***(3)*** *has particles projected along the collision gradient to the surface at $t = 1$.* ***(4)*** *At $t = 2$, particles are integrated and stepping occurs after the bounce.*

4.2 Improved Collision Resolution

The method from section 4.1 falls apart when many particles that are in close proximity to each other collide with a surface in 1 solver step (figure 3). The result is that all colliding particles are projected to the surface of the level set, and then stepping artifacts occur. Rather than reduce the timestep, one can estimate the collision position of each particle, then integrate them post-collision depending

on how long they tunneled under the surface of the object. This method is shown in figure 4.

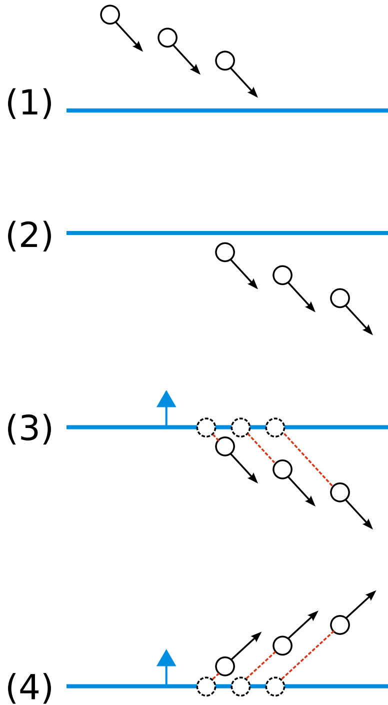

Figure 4: *Improved particle collisions.* ***(1)*** *shows the particles at* $t = 0$*.* ***(2)*** *shows the particles at* $t = 1$ *before collision resolution.* ***(3)*** *has particles backtracked along their velocity to their estimated collision location on the surface.* ***(4)*** *At* $t = 1$*, collision resolution ends with particles being integrated for their penetration time.*

4.3 Radius for Collisions

In the basic implementation, particle to level set collisions only consider the particle center, and ignore the radius (figure 5). There is a computationally cheap extension of our collision model to allow for per-particle radius. The distance to surface, ϕ, is calculated by querying the level set at the particle center, then the depth the particle penetrates the surface is estimated as $d = r - \phi$ where r is the particle radius.

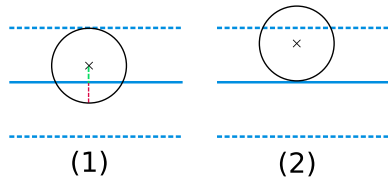

Figure 5: *Example of a particle with radius colliding against a level set.* ***(1)*** *shows the particle penetrating the level set. The dashed blue is the level set narrow-band width around the zero crossing in solid blue. The penetration depth (red dashed line) is estimated as described.* ***(2)*** *shows contact resolution.*

The level set narrow-band half width must be at least as big as the largest particle being collided. Thus, it must be expanded accordingly to get correct results. See figure 6 for an illustration of this issue.

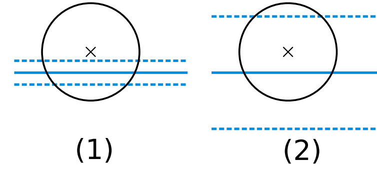

Figure 6: ***(1)*** *shows the collision event is not captured because the narrow-band is too small and does not overlap the particle center.* ***(2)*** *fixes the problem by increasing the narrow-band width.*

4.4 Fast Moving Colliders

When fast moving colliders are interacting with particles, the most obvious solution is to reduce the timestep until the collider motion is resolved. This results in many geometry to level set conversions, when simpler solutions can often be useful. One workaround is to use rigidly deforming objects as colliders in a fashion similar to DOPs in Houdini. By deriving the transformation matrices describing the body's motion, one can transform the rest frame's level set to the current solver time on each substep.

5 Constraints

Simulating granular materials requires modeling particle to particle collisions, so we decided to use position based dynamics [Macklin et al. 2014]. This framework has the benefit of easily being able to model a variety of constraint types, such as distance, pin, and clumping constraints. Each constraint type is modeled as a separate node that fits into the Rapid framework. These nodes provide the solver with a definition of the constraint function, as well as the group of particles it acts on. By having a generic interface for writing constraints, it is easy for even novice C++ programmers to write their own and have it plug into the system without changing the core solver. All existing nodes, such as colliders, emitters, and forces, behave as expected when constraints are used. Figure 7 shows a test of the constraint system.

5.1 Particle to Particle Collisions

We use a grid to accelerate particle to particle collision detection. However, we make use of sparse grids by utilizing OpenVDB Point Index Grids, which store a list of particle indices that are contained inside a voxel. The voxel width for the grid is sized according to the user-specified collision radius, R, such that $v = 2 * R$. By using sparse grids, we have quick access to potential collisions using iterators, and far better memory usage than dense grids.

5.2 Arbitrary Radius Particle to Particle Collisions

The above method works well for constant sized particles, but cannot work when the size varies from particle to particle. In order to accomplish this, we employ multiple Point Index Grids of different sizes. We want to limit the number of grids due to potential for excessive voxel evaluations to find neighbors, and we also want particles to fit into grids as tightly as possible to avoid iterating over too many potential colliders. Thus, we use a simple greedy approach to segment the particles based on their radii. This method is useful for cases where you might have several clusters of particles (ie: small, medium, and large), each of which has a bit of size variation. This type of distribution is common with debris simulations where we would want to use particle collisions instead of rigid body dynamics. Note that we often cap the total number of grids since user data can be unpredictable and the above algorithm will produce an un-

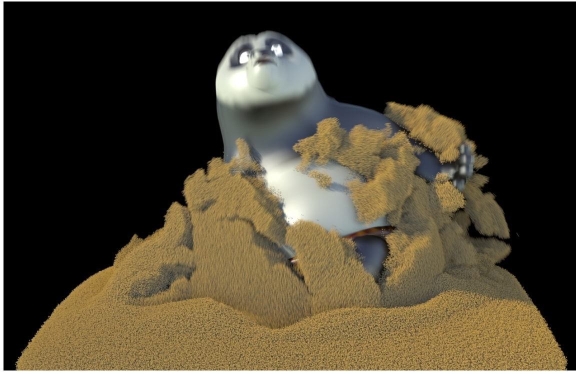

Figure 7: *Example of a character emerging from a pile of clumpy sand.*

Algorithm 1 Determine Collision Grids

$P \leftarrow sortRadiiDescending$ (array of particles)
$currGridRadius = radius\,(P[0])$
$currgrid \leftarrow createGrid\,(2 * currGridRadius)$
$gridArray.add\,(currGrid)$
for all p in P **do**
  if $currGridRadius - radius\,(p) > maxDifference$ **then**
    $currGridRadius = radius\,(p)$
    $currgrid \leftarrow createGrid\,(2 * currGridRadius)$
    $gridArray.add\,(currGrid)$
  end if
end for

bounded number of grids, depending on $maxDifference$. Also, we can calculate an error value that represents how well our sparse set of grids represent the distribution of radii. This works under the observation that an ideal scenario exists with zero error when each particle has a grid sized exactly according to its radius. Mathematically, the error is:

$$error = \sum_{p_i} voxelSize(i)^3 - \frac{4}{3}\pi r_i^3 \tag{5}$$

Where p_i is particle i, and r_i is its radius. Also $voxelSize$ returns the size of the grid that contains particle i.

6 Discussion

We have benefited from having a robust collision model, and accurate particle birthing mechanisms. The tradeoff is that our system cannot to handle arbitrary geometric primitives for collisions. Given the availability of high quality levelset generation tools, we found this constraint to have a negligible impact in artistic flexibility. The benefit is that artists like the simplicity and flexibility of this system over using a more generic system like POPs or Maya Particles.

While we have worked extensively on reducing the dependence on substepping, there are cases where it is necessary, such as with fast moving deforming colliders where the previously mentioned rigid collision model will not work. Also, some emission and sub-frame forces need to do sub-frame graph evaluations, which does not fit in our action operator framework. Thus, for such cases, the entire graph needs to be evaluated at a sub-frame rate. In general, many shots have made extensive use of the system without this type of sub-frame graph evaluation.

The past 15 years have seen an explosion (sometimes literally) in volume based techniques for modeling, simulation, and rendering [Wrenninge et al. 2010]. It was our opinion that some of the production uses of volumetric methods were due to the tools being newer, and better integrated, and not because they were better. In the world of modern visual effects, particle systems are extremely powerful, and not to be taken for granted. Creating a streamlined interface led to an increase in use, since artists can set up complex effects quickly with Rapid. The open structure of the node graph, and wide range of supporting tools have lead to many creative and impressive effects.

Acknowledgements

The authors would like to thank Kyle Maxwell, David Hill, Jaideep Khadilkar, Mark Carlson, and Yongning Zhu for their contributions to Rapid. We would also like to thank Ron Henderson, and Jonathan Gibbs for their support of the Effects Production Development Team. Finally we would like to thank every past and present member of the Effects Animation department at PDI/DreamWorks and DreamWorks Animation for inspiring us.

References

ALLEN, C., COHEN, J. M., BLOOM, D., FERREIRA, D. P., HASEGAWA, S., AND MCMAHON, C. 2007. Levelsets in Production: Spider-man 3. In *ACM SIGGRAPH 2007 Sketches*, ACM, New York, NY, USA, SIGGRAPH '07.

GUENDELMAN, E., BRIDSON, R., AND FEDKIW, R. 2003. Nonconvex rigid bodies with stacking. *SIGGRAPH '03: SIGGRAPH 2003 Papers* (July).

JOHANSSON, J. 2013. OpenVDB Course: OpenVDB Adoption at Digital Domain. In *ACM SIGGRAPH 2013 Courses*, ACM, New York, NY, USA, SIGGRAPH '13, 19:1–19:1.

MACKLIN, M., MÜLLER, M., CHENTANEZ, N., AND KIM, T.-Y. 2014. Unified Particle Physics for Real-Time Applications. *ACM Transactions on Graphics (TOG) 33*, 4, 153.

MUSETH, K. 2013. VDB: High-resolution sparse volumes with dynamic topology. *ACM Transactions on Graphics (TOG) 32*, 3, 27.

VROEIJENSTIJN, K., AND HENDERSON, R. D. 2011. Simulating massive dust in megamind. In *ACM SIGGRAPH 2011 Talks*, ACM, New York, NY, USA, SIGGRAPH '11, 67:1–67:1.

WITKIN, A. 2001. Physically Based Modeling Constrained Dynamics. *ACM SIGGRAPH 2001 Course Notes*.

WRENNINGE, M., BIN ZAFAR, N., CLIFFORD, J., GRAHAM, G., PENNEY, D., KONTKANEN, J., TESSENDORF, J., , AND CLINTON, A. 2010. Volumetric Methods in Visual Effects. In *ACM SIGGRAPH 2010 Courses*, ACM, New York, NY, USA, SIGGRAPH '10.

YUKSEL, C., MAXWELL, K., AND PETERSON, S. 2014. Shaping particle simulations with interaction forces. In *ACM SIGGRAPH 2014 Talks*, ACM, New York, NY, USA, SIGGRAPH '14, 42:1–42:1.