Erosion of Point Clouds with Smoothed Particle Hydrodynamics

Alec Bernardi, Emilee Reichenbach

Abstract

Traditional representations of geometry for use in computer graphics can be restrictive in applications with highly dynamic surfaces due to the high degree of connectivity information that they require. To address this shortcoming, forays into the realm of point-based rendering have shown that connectivity information is not always necessary to produce convincing results. To explore the potential of point primitives in computer graphics, we have integrated point clouds derived from triangular meshes into a particle-based fluid simulation that is rendered with the point-splatting technique.

1 Introduction

Models in computer graphics are usually either static geometry or animated characters. However, it is often the case that the geometry in a scene must change dynamically to produce a desired artistic effect. One of these effects in particular is the dissolution of a model into particles acting under the effects of a fluid simulation. Consider an ice cube melting in the sun, a sand dune eroding in the wind, or your favorite superhero ceasing to exist. In these examples, the models themselves deform over time as their volume transitions from traditional geometry into particles governed by a fluid simulation. To facilitate the continuous deformation of an arbitrary mesh, consider sampling its initial surface to generate a sufficiently dense point cloud. Assuming that these points can be rendered as a surface efficiently, it would be less troublesome to identify and deform individual points, instead of regions on a mesh, that are affected by the erosive forces of the fluid simulation. New particles can then be introduced to the fluid simulation on the deformation site to represent the eroded volume.

1.1 Overview

In an effort to implement it, this behavior was split into several distinct pieces. The first task is to take a traditional triangle mesh and convert it to a set of points that are uniformly distributed across its surface. The next component is a fluid simulation capable of physics-based interactions with such a point cloud. Lastly, the points representing both the static geometry and the fluid must be rendered.

1.2 Related Work

The purpose of the project is to deform an object, which can easily become difficult when using a mesh, as constantly updating, creating a new mesh, or keeping track of the mesh quality each time step is expensive [Selim et al. 2016]. Therefore using a point cloud is the chosen solution to avoid the hassles of traditional mesh deformation.

To actually deform the point cloud data, we need a fluid simulation. After learning how the Marker and Cell (MAC) approach works in a previous homework, the fluid representation used in this paper is Smoothed Particle Hydrodynamics.

A convenient and simple way to render points is by splatting. The data from the point clouds and fluid simulation lend themselves to being rendered via point splatting. These splats are then shaded through the use of Phong shading [Botsch et al. 2003].

2 Mesh to Point Cloud Conversion

The main component of this project is the fluid simulation, which is being used to dissolve or erode objects of arbitrary shapes and sizes. This is not easily done with meshes, especially if the mesh becomes invalid during the simulation. Converting triangular meshes to point cloud data provides a more intuitive way to deform the object and takes away the expensive checks necessary to ensure a mesh is valid.

Algorithm 1 Weighted Random Selection

```
1: procedure WeightedRandSelection(n, triAreas)
2:     p = rand() % 100
3:     c = 0
4:     for i = 0, triAreas do
5:         c += triAreas[i]
6:         if p ≤ c then
7:             return i
8:     return (rand() % (n + 1))
```

To convert a three-dimensional mesh, given in a .OBJ file format, to point cloud data, instead of performing a uniform random sampling of the mesh, a weighted random sampling is done. This ensures a relatively even spread of points over the entire surface of the mesh. Each triangle of the mesh is given a probability of being chosen to take a sample from, which is proportional to the ratio of its area to the entire mesh. Calculating the areas of each triangle in the mesh is done by computing half the cross product of the 3 points, shown below.

$$S = \frac{|AB \times AC|}{2}$$

For each triangle, its area is divided by the sum of areas for all triangles, giving each triangle a probability proportional to its area. An overview of the algorithm used to perform a weighted random selection is shown in Algorithm 1, which is called as many times as input samples.

Once a triangle is selected, to generate a random point inside of it, barycentric coordinates are utilized to aid in the process. These allow a point on a triangular surface to be expressed with the following equations.

$$P = uA + vB + wC$$

$$\text{where, } u + v + w = 1$$

A, B, and C are the vertices of the triangle and u, v, and w are the barycentric coordinates. The barycentric coordinates add up to 1 because they are normalized, so if any of the three are less than 0 or greater than 1, the point given by the previous equation is outside of the triangle.

The u and v are randomly generated, through the `rand()` function on a scale of $[0, 1]$, then w is found with $w = 1 - (u + v)$. The new point is then computed by using the equation above. Each point is paired with a normal for the triangle it was generated from.

As a way to visualize the point cloud data, Gmsh, a mesh generator and post-processing program, is used to see the points generated from the original mesh. Figures 1 and 2 show the Stanford bunny and Utah teapot, given as input meshes, for a variety of sample sizes.

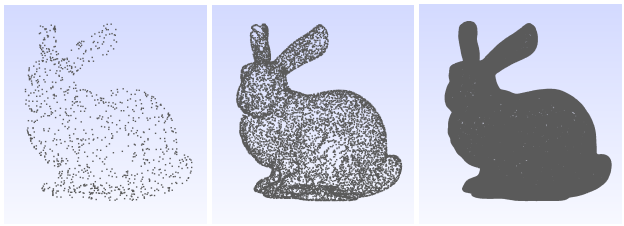

Figure 1: *The three images show a range of number of surface samples taken from an input mesh of the Stanford bunny. From left to right the number of samples is,* 10^3*,* 10^4*, and* 10^5*.*

Figure 2: *The three images show a range of number of surface samples taken from an input mesh of the Utah teapot. From left to right the number of samples is,* 5×10^3*,* 5×10^4*, and* 5×10^5*.*

3 Fluid Simulation

The fluid body is simulated as a set of points that represent the particle field. These points are used to approximate the properties of the fluid and respond to the resulting forces. As with most computer simulations, the forces are applied at discrete time steps, and then each particle's properties are updated accordingly. The following description details the process of updating the particles in this discrete fashion.

For each time step, the new positions of the particles are integrated using the second-order Velocity Verlet integration technique [Swope et al. 1982]. Each particle maintains a position, velocity, and acceleration, denoted as $\vec{r_i}$, $\vec{v_i}$, and $\vec{a_i}$, respectively. The following equations are used to update such a particle after a time-step of Δt.

$$\vec{r_i}' = \vec{r_i} + \vec{v_i} \cdot \Delta t + \frac{\vec{a_i} \cdot \Delta t}{2}$$

As $\vec{a_i}'$ denotes the acceleration of the particle at its new position $\vec{r_i}'$, each particle must have its position updated before computing the net force acting upon it. This force can then be used to update the acceleration of the particle, and consequently compute its new velocity.

A formulation of smoothed particle hydrodynamics closely following the work of Müller et al. [2003] is used to compute the forces on each particle. The net force on each particle is the sum of forces due to pressure, viscosity, surface tension, and gravity. To compute these, however, the density and pressure of the fluid at each particle must first be determined. Fortunately, the Lagrangian formulation of the fluid simulation allows such quantities to be approximated as sums across neighboring particles weighted by a normalized, radially symmetric kernel function with a compact support radius of h. In this fashion, the density at each particle is computed as follows, given that particle j has mass m_j.

$$\vec{v_i}' = \vec{v_i} + \frac{\vec{a_i} + \vec{a_i}'}{2} \Delta t$$

With this density, the pressure at the particle can be computed using an equation of state. In this case, one resembling the ideal gas law is used. Two constants are used to stabilize the gradient of the pressure field, which is used to compute the force due to pressure. These constants are the rest pressure p_0 and the rest density ρ_0, and can be chosen arbitrarily. Meanwhile, the gas constant k depends on the temperature of the simulated fluid.

$$\begin{aligned} \rho_i &= \sum_j m_j W_{poly6}\left(\vec{r_i} - \vec{r_j}\right) \\ &= \frac{315}{64\pi h^9} \sum_j m_j \left(h^2 - |\vec{r_i} - \vec{r_j}|^2\right)^3 \end{aligned}$$

$$p_i = k(\rho_i - \rho_0) - p_0$$

The force due to pressure is computed with the following equation. This differs from the work of Müller et al. [2003], which uses a different formulation that did not yield acceptable results when implemented.

$$\begin{aligned} \vec{f_p} &= \sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}\right) \Delta W_{spiky}\left(\vec{r_i} - \vec{r_j}\right) \\ &= -\frac{45}{\pi h^6} \sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}\right) \frac{\left(h - |\vec{r_i} - \vec{r}|\right)^2}{|\vec{r_i} - \vec{r}|} \end{aligned}$$

The force due to viscosity, on the other hand, is unaltered, and can be computed as follows, given the viscosity constant of the fluid, μ.

$$\begin{aligned} \vec{f_v} &= \mu \sum_j m_j \frac{\vec{v_j} - \vec{v_i}}{\rho_i} \Delta^2 W_{viscosity}\left(\vec{r_i} - \vec{r_j}\right) \\ &= \frac{45}{\pi h^6} \mu \sum_j \frac{m_j}{\rho_j} \left(\vec{v_j} - \vec{v_i}\right) \left(h - |\vec{r_i} - \vec{r_j}|\right) \end{aligned}$$

To compute the force due to surface tension, the surface normal $\vec{n_i}$ must be determined. This is computed by taking the gradient of what the literature refers to as the color field. If the length of the resultant normal is less than some threshold, then it is considered inside the fluid and subsequently set to zero. This increases stability by avoiding division by near-zero values, while also preserving the idea that surface tension only has noticeable effects on particles close to the surface of the fluid.

$$\begin{aligned} \vec{n_i} &= \sum_j \frac{m_j}{\rho_j} \Delta W_{poly6}\left(\vec{r_i} - \vec{r_j}\right) \\ &= -\frac{945}{32\pi h^9} \sum_j \frac{m_j}{\rho_j} \left(h^2 - |\vec{r_i} - \vec{r_j}|^2\right)^2 \end{aligned}$$

$$\begin{aligned} \vec{f_t} &= -\sigma \sum_j \frac{m_j}{\rho_j} \Delta^2 W_{poly6}\left(\vec{r_i} - \vec{r_j}\right) \\ &= \frac{945}{32\pi h^9} \sigma \sum_j \frac{m_j}{\rho_j} \left(|\vec{r_i} - \vec{r_j}|^2 - h^2\right) \left(7|\vec{r_i} - \vec{r_j}|^2 - 3h^2\right) \end{aligned}$$

The force due to gravity is relatively trivial to compute. Given that $\vec{g}$ is the acceleration due to gravity, the following equation will apply the uniform acceleration to each particle as a force.

$$\vec{f_g} = \rho_i \vec{g}$$

After computing all forces acting upon each particle, the new accelerations can be determined with the following equation.

$$\vec{a_i}' = \frac{\vec{f_p} + \vec{f_v} + \vec{f_t} + \vec{f_g}}{\rho_i}$$

Subsequently, the velocity of the particle can be updated according to the aforementioned integration method, and the entire process begins again to capture the next state of the following time-step in the simulation.

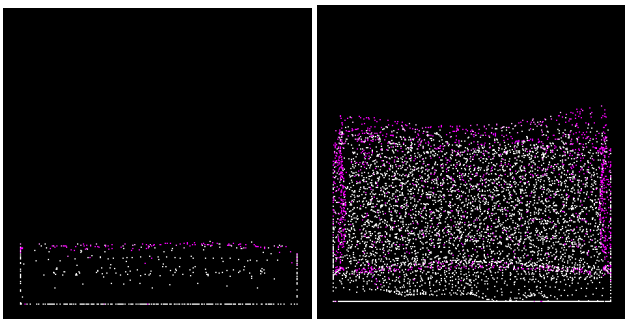

Figure 3: *On the left, a 2D view of the fluid simulation. On the right is a 3D view. Particles with density less than the fluid's rest density are marked in magenta.*

4 Point Splatting

The technique of point splatting has been quite successful at rendering points on the surface of an object with no connectivity information [Botsch et al. 2005]. This is readily applicable to particle-based fluid simulation, where the only readily available information about the fluid is the location of each particle, as well as the surface normal in that location.

The basic idea behind point splatting is to render circular discs, called splats, tangent to the surface at each sample point. Using the standard primitive `GL_POINTS` in OpenGL, each uploaded vertex is rendered as a filled quad of a given size. Furthermore, if `GL_POINT_SPRITE` is enabled, then the fragment shader is supplied the UV coordinates of the fragment within the filled quad. This functionality allows each fragment to determine whether it lies on the splat.

While splats are circular discs, they exist on a plane defined by its center and normal. So, unless the camera views the splat along the normal, then it will appear as an ellipse when projected onto the image plane. To determine which fragments fall within this ellipse, first consider the projection of a single point onto an arbitrary plane. For a point $\vec{p}$ and plane defined by the point $\vec{c}$ and unit normal $\vec{n}$, the projection $\vec{p_*}$ of $\vec{p}$ onto the plane can be found by subtracting the component of the difference between $\vec{p}$ and any point on the plane that is orthogonal to the plane itself. Since the point $\vec{c}$ is defined to be on the plane, the following equation holds.

$$\vec{p_*} = \vec{p} - \hat{n}(\vec{c} - \vec{p} \cdot \hat{n})$$

Once an image space fragment is projected onto the same plane as the splat, a simple distance check can be used to determine if it falls within the splat radius. Fragments that do not pass this test are discarded. Those that do, however, may be colored and rendered to the framebuffer. Phong shading was implemented in order to provide visual cues like shadows and specular highlights to better emphasize the shape of the rendered surface.

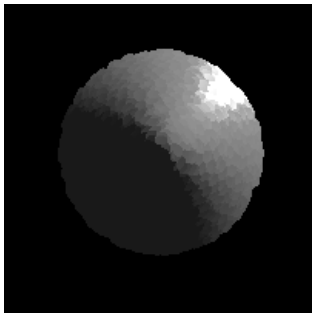

Figure 4: *Points on a sphere visualized with point splatting. Note the small elliptical shapes that comprise the surface.*

5 Discussion of Results

After implementing each of the pieces outlined in the previous sections, models can be converted into point clouds, imported into the fluid simulation as an initial state for particles, and then visualized with splatting as the fluid simulation progresses.

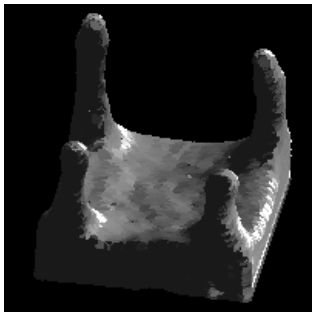

Figure 5: *A drop of viscous liquid after colliding with the floor plane and sending columns of liquid upwards.*

5.1 Performance

After tweaking parameters to adjust for numerical stability, the simulation is capable of interactive frame-rates thanks to load distribution over all available cores. This degree of parallelization is possible due to the highly parallel nature of particle-based approaches for fluid simulation. Further performance benefits were achieved by using an efficient approach to finding the nearest neighbors for each particle, inspired by the work of Bayraktar et. al [2009].

Num. of Particles	Approx. FPS	h	Δt
2^{15} (32k)	6	$\frac{1}{20}$	0.002
2^{14} (16k)	12	$\frac{1}{16}$	0.002
2^{13} (8k)	30	$\frac{1}{16}$	0.0001

Table 1: *Performance for fluid simulation on Intel Core i7-6700HQ.*

Even with such promising performance metrics, the current implementation is heavily CPU-bound. The only task given to the GPU is the point splatting, which involves hardly any work, since there are relatively few splats, and drawing splats is innately quite efficient. As a result, offloading more of the particle simulation onto

the GPU via compute shaders would further increase performance, and perhaps attain interactive frame-rates with even more particles.

5.2 Limitations

While there were plans to gradually introduce particles to the fluid simulation by eroding volumes from the original mesh, the mechanism to actually do so remains unimplemented, mostly due to time constraints. Even so, the point clouds can be used as the initial state of the fluid simulation, to a limited effect.

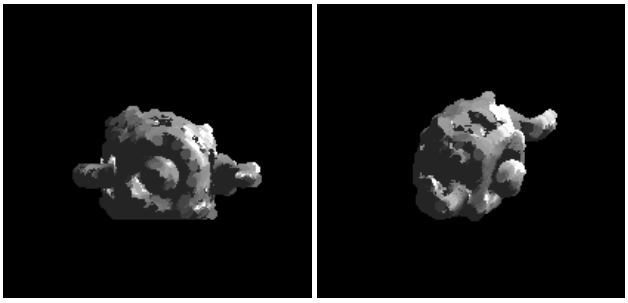

Figure 6: *The Utah Teapot converted to fluid and colliding with the floor plane.*

While the performance of the simulation is promising, there is a fair amount of parameter tuning that must be done to achieve fast and stable results. The kernel radius h must be large enough to ensure well-approximated behavior, while also being small enough to keep the average number of neighbors for each particle low. Moreover, while the fluid properties can be assigned to more-or-less realistic values, low-viscosity fluids like water have much lower stability than highly-viscous fluids.

5.3 Future Work

There are many routes to take for future work. Improvement of the program performance can be done through moving the fluid simulation calculations onto the GPU and performing all the calculations in parallel using compute shaders. Furthermore, the numerical stability of the algorithm could be improved in order to allow for smaller time-steps, liquids with much lower viscosity, and overall improve the accuracy of the simulation.

In an attempt to make each of the point splats more cohesive, splat filtering can be used to create a soft boundary on each splat to reduce aliasing [Botsch et al. 2003, 2005] This would decrease the visual distinction between splats. Splat quality can be further improved by using multipass rendering to interpolate splat normals across the surface, resulting in more even shading, and even less distinction between actual splats. Lastly, more splats could be generated by using the fluid simulation state to interpolate more surface particles exclusively for rendering.

While the current implementation of mesh conversion only samples surface points, the idea of sampling meshes volumetrically arises as a potential improvement. This would improve upon the ability to use point clouds as initial states for the fluid simulation, which expects a volume of fluid, and not just the surface. One of the difficulties in importing the point clouds is determining the mass and volume of each particle. Accuracy in this regard is imperative for stable simulation. Volumetric sampling would not only provide proper initial state, but also provide a method of estimating the volume of the point cloud before importing it.

6 References

BAYRAKTAR, S., GUDUKBAY, U., AND ZG, B. 2009. Gpu-based neighbor-search algorithm for particle simulations. *J. Graphics, GPU, and Game Tools 14* (01), 31–42.

BOTSCH, M., AND KOBBELT, L. 2003. High-quality point-based rendering on modern gpus. 335– 343.

BOTSCH, M., SPERNAT, M., AND KOBBELT, L. 2004. Phong splatting. vol. 2004, 25–32.

BOTSCH, M., SORKINE-HORNUNG, A., ZWICKER, M., AND KOBBELT, L. 2005. High-quality surface splatting on today's gpus. 17– 141.

EFRAIMIDIS, P., AND SPIRAKIS, P. 2008. *Weighted Random Sampling*. Springer US, Boston, MA, 1–99.

HARADA, T., KOSHIZUKA, S., AND KAWAGUCHI, Y. 2007. Smoothed particle hydrodynamics on gpus. *Computer Graphics International* (01).

HUANG, C., ZHU, J., SUN, H., AND WU, E. 2013. Efficient fluids simulation and rendering on gpu. 25–30.

MACKLIN, M., AND MLLER, M. 2013. Position based fluids. *ACM Transactions on Graphics 32* (07), 104:1–104:12.

MACKLIN, M., MLLER, M., CHENTANEZ, N., AND KIM, T. 2014. Unified particle physics for real-time applications. *ACM Transactions on Graphics 33* (07), 1–12.

MLLER, M., CHARYPAR, D., AND GROSS, M. 2003. Particle-based fluid simulation for interactive applications. vol. 2003, 154–159.

NEHAB, D., AND SHILANE, P. 2004. Stratified point sampling of 3d models.

SCHUSTER, R. 2007. Algorithms and data structures of fluids in computer graphics.

SELIM, M., AND KOOMULLIL, R. 2016. Mesh deformation approaches a survey. *Journal of Physical Mathematics 7* (06).