Rendering Sand
COMP 870 - Advanced Image Synthesis Final Project

Abhinav Golas*
UNC Chapel Hill

Figure 1: *Rendering of 2 sample meshes using the implemented method, Left: Laughing Buddha, Right: My face rendered as sand*

Abstract

Rendering of sand is an interesting problem that has been addressed at various times in the past. Most efforts have been in the direction of rendering moving sand grains, which are handled as particles rendered with motion-blur. More recently, there have also been efforts at rendering slow moving and static sand particles. In such cases, accurate sampling and rendering of particles becomes important.

This project attempts to create a renderer for such sand on the lines of the method sketched out in [Allen et al. 2007]. The project is an attempt to try and recreate a similar quality rendering for 3D meshes in Renderman©. In the process, I also aimed to learn more about Renderman Dynamic Shared Objects (DSOs) and using them to create procedural objects.

*e-mail: golas@cs.unc.edu

1 Introduction

Sand rendering has historically been done by using particle systems with large number of particles. Recently, some work has also been done on simulating sand. There have been 2 primary approaches, one using particle systems with relevant forces on the lines of Smoothed Particle Hydrodynamics (SPH), and the other being grid based fluid simulation. [Bell et al. 2005] details an approach for simulating granular materials using particle systems with pair-wise forces. The simulation is highly accurate, but can only scale to 10K particles at interactive rates. On the other hand, [Zhu and Bridson 2005] approximates sand as a fluid with viscosity, and thus can handle large scale sand simulations, but lacks detail.

Both of these papers follow different approaches to render the simulated sand. [Bell et al. 2005] renders each particle as a pebble, while [Zhu and Bridson 2005] renders sand as a fluid with advected 3D textures to add detail to the render.

Also, recently, a sketch was presented at SIGGRAPH 2007 detailing the approach used to render the villain "SandMan" in Spider-

man 3 [Allen et al. 2007] using Pixar's Renderman renderer [Upstill 1989; Apodaca and Gritz 1999; Pixar 2005]. Their module uses a procedural model to render sand. A DSO generates particles to render at runtime on a given 3D mesh or using a sample set of particles. Using runtime particle generation allows the power to generate close to 480 million particles as needed. The major hurdles that need to be solved in such a scenario are:

- Correct sampling to accurately depict the desired object while ensuring sufficient particle coverage
- Ensuring temporal coherency among particles generated in different frames to prevent popping artifacts

The sketch does not provide details on any of the above, thus solutions need to be found for the problems. It does, however, detail a Level of Detail approach that was used to provide sufficient detail depending on the proximity of the rendered object to the camera. The paper proposes rendering particles as one of points, curves or archived mesh models depending on the distance to the camera, and the pixels occupied in the rendered frame.

2 Rendering Meshes

The basic approach to rendering the mesh as a sand is simple, and follows the following steps:

1. Load the 3D mesh as vertices and triangles, and define a bounding box or a hierarchy of bounding boxes for the mesh
2. Sample particles on the mesh as needed
3. Render sampled particles

The details for each operation and the problems encountered are detailed in this section. For this project, I focused mainly on rendering 3D meshes as sand, as the other possible input, using a set of representative particles is simply one more sampling problem, rather than another separate problem.

2.1 Loading a mesh

I wrote a basic OBJ mesh loader, that loaded the required vertex and triangle mesh information, and generates a bounding box for the entire mesh. This was later expanded to generate multiple bounding boxes as a hierarchy.

2.2 Particle Generation

To generate particles on a triangle, particles are seeded randomly by generating random Barycentric coordinates. The 2 independent barycentric coordinates are sampled from a uniform distribution in the range $[0, 1]$. We also define 2 other parameters, particle radius(r) and particle density(ρ). It is ensured that no 2 particles can intersect, where particles are assumed to be perfect spheres of the same radius. Also, in order to cover the triangle completely, we need to know how many particles are sufficient. [Bell et al. 2005] defines a simple formula for this:

$$nP = ceil\left(\frac{\rho A}{\pi r^2}\right) \tag{1}$$

where nP is the number of particles to be seeded in the triangle, $ceil$ is a function that rounds of any floating point number p to the nearest integer q such that $q > p$. A, the area of the triangle, is calculated by taking the cross product of 2 triangle edges, which is also used to define the triangle normal. ρ acts as a quality control parameter, by allowing fine grained control of the number of particles to be seeded. Since it may not be possible to perform a perfect packing such that no 2 particles intersect, we allow for 1000 iterations of finding a clear spot, after which the last generated location is used. Though this is not the best method to seed particles, it is a fast and inexpensive method to generate particles, without having to perform the density balancing steps detailed in [Bell et al. 2005].

In addition to generating particle positions, we also need suitably jittered normals. To do so, I generate a random normalized 2D vector in the triangle plane n_2, and use a vector sum of the triangle normal n, and a scaled version of n_2, i.e.

$$n_p = n + \alpha \times n_2 \tag{2}$$

where α is a random scaling parameter, and n and n_2 are vectors with norm 1. $\frac{n_p}{\|n_p\|}$ is then the normal of point p. This method ensures that though normals will be jittered, they will never point against the triangle normal, which would result in a culled point.

A certain amount of displacement is also added to each particle along the triangle normal, to add another level of detail.

2.3 Rendering particles

[Allen et al. 2007] denotes 4 possible options of rendering particles. Since we do not need extreme close-ups of the models, using expensive primitives like curves or archived meshes was avoided. In addition, obtaining mesh model of rocks or pebbles proved to be a tough task.

When using points, the sketch described using blocks of 4 points with jittered normals, which would provide the impression of a small grain. We employ this method of generating particles. In the previous section, we described the method for generating particle normals by generating a vector n_2 in the plane of the triangle. We can use the same generated vector, to generate normals for the 4 points, by simply flipping the signs of the components along 1 or both of the basis vectors of the triangle plane. I.e. if:

$$n_2 = n_{2_x}\hat{\imath} + n_{2_y}\hat{\jmath} \tag{3}$$

then use a modified normal jitter:

$$n_2^* = \pm n_{2_x}\hat{\imath} + \pm n_{2_y}\hat{\jmath} \tag{4}$$

for each of the 4 particles. In addition, we add a displacement along the jittered normal so that the 4 particles do not intersect, thereby avoiding spurious popping.

3 Rendering point clouds

As an experiment, I tried rendering sand point clouds I generated using my implementation of [Zhu and Bridson 2005]. This poses a different challenge as compared to meshes, as the sampling issue is somewhat simplified. The bigger challenge is to ensure that any extra generated particles are coherent across frames. For this purpose, I tried to sample new points randomly in a region around the simulated particle. To ensure coherency across frames, I simply re-seeded the random number generator with the same seed at each frame. Being pseudo-random, RNGs will provide the same set of random number sequences for each frame if the seed is kept constant. In spite of being very simple, this method provides compelling results, as can be seen in the images shown in Figure 2.

4 Results and Implementation details

The system was implemented in C++ while utilizing Renderman DSOs to render the particles as needed. For both mesh examples,

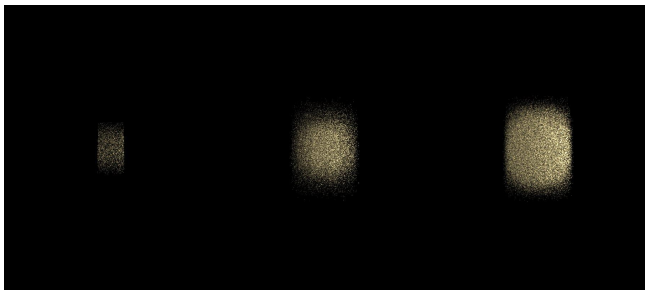

Figure 2: *Cubical region of sand with point multipliers 1, 10, and 50*

particles of the order of 5-9 million particles were generated. The time taken to generate particles was between 20-30 seconds on single threaded execution, on a Core 2 processor running at 2.4Ghz. Rendering time to generate images at resolution 600×800 was 3-4 seconds per frame. To keep temporal coherency, particles are preserved across frames, and we do not perform a true on-demand generation method. However, since we are only dealing with static examples and a simulated sand example is not taken, this is sufficient.

For the simulated example, 10K particles were simulated, and the examples shown had 1, 10, and 50 particles per simulated point. Generation times varied from 2 seconds to 10 seconds, with rendering times remaining under 1 second.

An experiment to perform on demand generation for triangle blocks was undertaken, by utilizing a flat bounding box hierarchy, but that instead lead to an increase in runtime due to the increase in number of function calls. The number of particles we are working with, do not seem to warrant such an overhead. As a result, that approach was discarded, and all particles are generated only once. This does render the DSO approach an overkill, but I would like to extend the renderer in the future to use rendering LOD and runtime particle generation in the near future. This would be useful, and required to render sand from the grid based simulations, which involve dynamic scenes with changing topology.

I would also like to note and interesting digression. During execution, I ran into an interesting problem where the jittered normals were approaching 0. This was tracked down to the problem of triangles being too small in the mesh, thus the edge vectors were of very small magnitude. When their cross product was taken, the magnitude was relatively accurate and usable, but the direction information in the normal was lost at floating point precision. To work around this, the triangle normal generation was done separately, after normalizing the edge vectors, while the area calculation is done with the original vectors.

References

ALLEN, C., BLOOM, D., COHEN, J. M., AND TREWEEK, L. 2007. Rendering tons of sand. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*, ACM, New York, NY, USA, 27.

APODACA, A. A., AND GRITZ, L. 1999. *Advanced RenderMan: Creating CGI for Motion Picture*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

BELL, N., YU, Y., AND MUCHA, P. J. 2005. Particle-based simulation of granular materials. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, ACM, New York, NY, USA, 77–86.

PIXAR. 2005. The renderman interface, version 3.2.1.

UPSTILL, S. 1989. *RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

ZHU, Y., AND BRIDSON, R. 2005. Animating sand as a fluid. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, ACM, New York, NY, USA, 965–972.