Real-time 3D Sand Simulation

By Jon Castro and Samuel Chernick

1 - ABSTRACT

In game production, game engines are typically used to simulate two dimensional and three dimensional environments. These game engines allow specific objects to be affected by physical properties such as gravity, while other objects are simply drawn to the screen. Particle engines are a special type of game engine that apply physics to all objects drawn on the screen. This allows for highly interactable and malleable environments. In this paper we will discuss implementation and evaluation of two dimensional and three dimensional particle engines.

2 - INTRODUCTION

Falling sand is a genre of video game that uses a 2D particle engine. Falling sand games allow users to place various types of sand particles that can interact with each other in unique ways. This can lead to emergent behaviours and complex interactions between the sand particles. **[1]** Typically in these games players have no objective. A player can simply place sand particles to observe their behaviour or make sculptures or other art out of sand.

In 2019, a new type of sand game was released. Noita is a rpg game that runs in a 2D particle engine. This game innovates on the sandbox style gameplay of typical falling sand games by adding a controllable character. In addition to this, the particle engine has been expanded to allow for rigidbodies, animations, and a variety of other gameplay functions. **[2]**

Noita showed that a 2D particle engine could be run in real-time to create more complex games. In this paper we discuss the possibility of adding a third dimension to a particle engine. Adding a third dimension can exponentially increase the computational load, so it is possible that a 3D particle engine is not yet computationally viable.

3 - RELATED WORK

Despite his focus on realism, Sims details a useful method for storing particle data, animating particles, and using parallel computing with particles.**[4]** Sims explains that particles can be rendered simply by storing a position and color, and optionally a radius and opacity. From this particles can be drawn either as stationary points, or as animated particles by using another position as a tail to render a line. He also describes the proper order to update and render particles.

In his Game Developers' Conference talk, Purho shows his method for implementing a 2D particle engine. He explains the basic particle update behaviour and some common problems when implementing a particle engine. He also goes into acceleration structures and techniques for increasing the performance of a two dimensional game inside his engine.

4 - IMPLEMENTATION

The implementation of the project can be split into two parts. The 2D implementation and the 3D implementation. The 2D implementation is the standard sand algorithm described by Petri Purho. The 3D implementation is an extended version of Purho's implementation that adds an additional dimension as well as an integration with OpenGL.

4.1 - 2D Implementation

Initially, two classes needed to be created for the simulation. One class, the particle class, stores information about the individual particles. The other class, the particle buffer class, stores, manages, and renders all particles that have been spawned.

The particle class stored data for each pixel, or particle, that was drawn on the screen. Most commonly, position and color are used to determine where to draw and what to draw. This class is also structured to allow polymorphism. This lets us quickly create different particle types that all have the same base functionality. Each particle type can have its own update function that changes the behaviour each frame. The generic update function for a standard sand particle has the particle move down if possible, and if not, it moves down and left. If it can't move down and left, it tries to move down and right. If that too is blocked, the particle does nothing.

The particle buffer class is more of a management tool that stores all the particles. The buffer is stored as a 2D array of particles, where empty particles are used to represent blank spaces. When calling a function like render or update, the buffer checks each particle, including the empty ones. One issue with this approach is the performance. A much faster method is to use a particle vector that only stores active particles. This was our original implementation, but this may cause issues when updating large amounts of particles with varied behaviours. It is safer to hold a buffer and check from the bottom row to the top row. This prevents particles from potentially not moving when they should be allowed to.

4.2 - 3D implementation

For the 3D extension of the particle class, the same overall idea from the 2D version was kept while changing the buffer to a three dimensional array, adding a z-axis. The other main change was the logic of the particles as they fell. Rather than just going down, left, or right; forwards, backwards, and diagonally are now options to take. This didn't have any noticeable impact on individual particle performance since it's just more cases to check, but the addition of another axis caused the buffer to be exponentially larger. The growth in buffer size also caused a significant decrease in performance. This is due to the program checking every space in the buffer to determine if there is a particle there and if it can move. Something like a buffer with a large z axis could significantly slow down the program despite most of the cells being empty.

4.3 - Rendering

Initially, we decided to use the Viper engine to render the particles in real time. However we had to pivot away from that engine as it ended up being far more cumbersome than what was needed. It had a considerable slow down when many objects needed to be rendered, effectively capping how efficient we could be. We then transitioned to an OpenGL renderer to complete the rest of the project. The renderer works by keeping a float pointer pointing to an array. These floats could be broken down into groupings of eight and represented the position and color of the particle being rendered as vectors of size four. The first four floats represented a particle's x, y, and z values and 1 while the second four floats represented the RGB and alpha values of the particle. After the particles in the particle buffer were updated in a frame, the positional information of each particle would be entered into the point data array so that when the renderer was called, it would render each particle as a point in space.

4.4 - Parallelization

To further increase the simulation speed we parallelized functions in the particle buffer. Most functions that involve looping over the particles in the buffer can be done in a massively parallel fashion. Although a large number of threads can be used, it may not actually provide a massive increase in performance. According to Amdahl's law, there will come a point where adding processors and threads will cease to increase the speed of a program. This is because a program needs to be specifically optimized for parallel usage in order to receive significant speedup from parallelization. **[3]**

5 - RESULTS

Fig. 1: A 20 by 20 buffer of sand. Particles are spawning in at the top of the screen and flowing downward each iteration.

Initially, we rendered two dimensional ASCII representations of the particle buffer. This was done to test the known implementation explained by Purho. Once we knew the systems were functional we could simply expand the particle to the third dimension.

Fig. 2: initial rendering with incorrect position corrections.

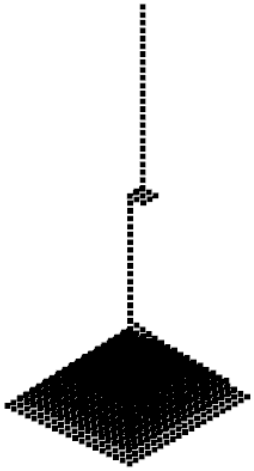

Fig. 3: Functional render of particles in real time. The 3x3 platform is a type of particle that doesn't move.

After successfully testing our algorithm in an ASCII representation and getting good results, we moved on to 3d rendering in OpenGL. Initially, we encountered lots of issues with sizing and display since the rendering system we used worked on a zero to one position scale, while our particle buffer worked on an integer scale. The size of the particle was also an issue, because, depending on how many particles we were using, they would end up being rendered with space in between them or inside each other. We also found that the 3d nature meant that a lack of outlines and shadows made it difficult to see what was happening at times. Eventually we decided to have a small amount of space between particles to allow for a more understandable visual. We also found that a lack of randomization for when particles determined where they would move led to very repetitive looking results which can be seen in Fig. 3 with the single constant stream of particles in the center.

6 - ANALYSIS

Parallelization tests were run on the two dimensional version of the particle simulation. After parallelizing the particle buffer functions, a

simple timer was used to determine how many seconds the simulation took to run sixty iterations. The simulation was run on a windows computer with an i7-7700HQ CPU and a GTX 1050 GPU.

```
Iteration (0):0
Iteration (60):6.016
Iteration (120):12.246
Iteration (180):18.288
Iteration (240):24.196
Iteration (300):30.107
Iteration (360):36.013
Iteration (420):41.938
Iteration (480):47.905
```

Fig. 4: The time to run (seconds) 60 iterations with 1 million particles in two dimensions with no threading.

```
Iteration (0):0
Iteration (60):2.564
Iteration (120):4.921
Iteration (180):7.259
Iteration (240):9.582
Iteration (300):11.943
Iteration (360):14.302
Iteration (420):16.665
Iteration (480):19.087
Iteration (540):21.967
```

Fig. 5: The time to run (seconds) 60 iterations with 1 million particles in two dimensions with four threads.

Using standard library C++ threads we observed a significant performance gain. Using just four threads while updating the particle buffer resulted in a more than 100% increase in efficiency. While adding more threads may further increase performance, massive parallelization without a thread system such as CUDA could actually plateau or even lower the performance.

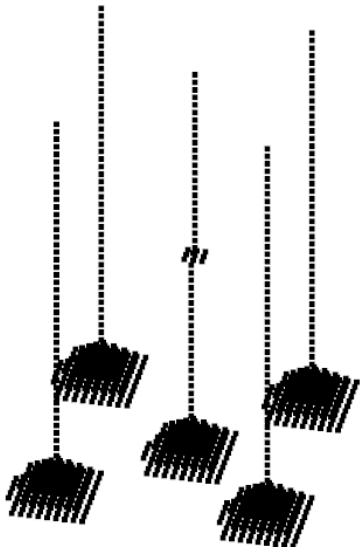

Fig. 6: Initial start of rendering for separate sources and multiple particles being generated per frame.

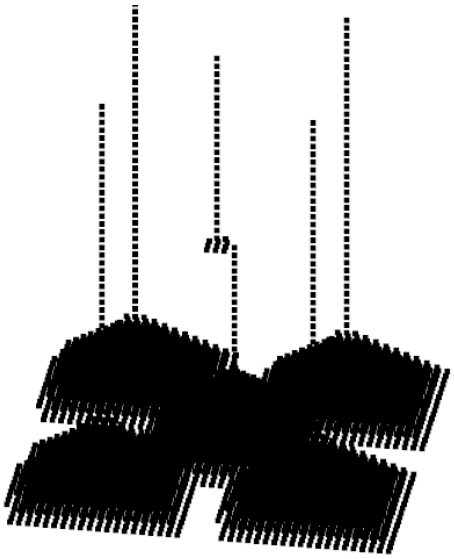

Fig. 7: multiple particle generation after some time.

The next set of tests were run on a windows computer with an i5-6600k CPU and a GTX 1060 6GB GPU. The outcome in Fig. 3 was the result of a 50x50x50 particle buffer and could be observed in real time with fluid camera movement and no noticeable slowdown upon shifting the camera. The next test, in Fig. 6 and Fig. 7, was the same scene but with four more

particles being generated every frame. The actual performance difference was not noticeable and the program suffered little, if any, slowdown. However, when the particle buffer was increased to a size of 200x200x200 particles, the program wouldn't even render a single frame, let alone in real time. The limitation in the amount of particles derives far more from the number of items to check in the buffer rather than the number of particles that need to be rendered.

7- LIMITATIONS

During testing we were limited on the diversity of our tests. Between the two of us, we only had two computers to actually test the simulation on. Testing on an older machine would likely show a significant slowdown and would receive little performance increase from threading.

We were also limited by using one of the homework assignments as a framework for the 3D rendering. Using the homework made it difficult to package the simulation. Transferring the build was also troublesome since dependencies would typically throw errors. Using a more open source engine that allowed for simpler rendering could have increased performances as well.

8 - CONCLUSION

The final results of our program produced a rather favorable outcome for our initial goal. The program can easily render as many as 125,000 particles at once with no noticeable slowdown. However, there do exist hard limits based on the initial size of the particle buffer and at high enough numbers the program will cease to operate. On the other hand, the actual number of particles being displayed on the screen at a time doesn't change the overall performance of the program based on the small differences found from our experiments, Fig. 3 and Fig. 7. With 3D integration of the parallelization code, the overall limit of the buffer size should be increased based on our findings in the 2D version of the algorithm.

9 - WORK DISTRIBUTION

The work on this project was distributed between Jon and Sam. The 2D implementation and parallelization was handled by Jon, while the 3D implementation and rendering was handled by Sam. Initially, Jon created the particle class and particle buffer. This was then handed off to Sam to render in OpenGL and adjust to fit in 3D. Jon then proceeded to work on parallelization for the 2D implementation.

10 - FUTURE WORK

The parallelization portion of the project could be improved significantly. Using CUDA to handle threading at a much larger scale could lead to significant performance gains. Standard game engine optimizations like culling could be applied to increase rendering performance. Implementing an acceleration structure or other ways to prevent updating all the particles each frame could also lead to a performance gain. One method could be to add all particles in the camera's view to a buffer. This secondary buffer would be updated while all other particles would schedule updates to be run asynchronously or just simply not update if they are far enough away.

REFERENCES

[1] Bittker, Max. *Making Sandspiel*, maxbittker.com/making-sandspiel.

[2] Purho, Petri. *Exploring the Tech and Design of Noita*. Game Developers' Conference, 2 Jan. 2020.

[3] Hill, Mark D, and Michael R Marty. *Amdahl's Law in the Multicore Era*. 2008, *Amdahl's Law in the Multicore Era*, www.cs.rpi.edu/~chrisc/COURSES/PARALLEL/SPRING-2020/papers/hill-multicore.pdf.

[4] Sims, Karl. “Particle Animation and Rendering Using Data Parallel Computation.” Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '90, 1990, doi:10.1145/97879.97923.