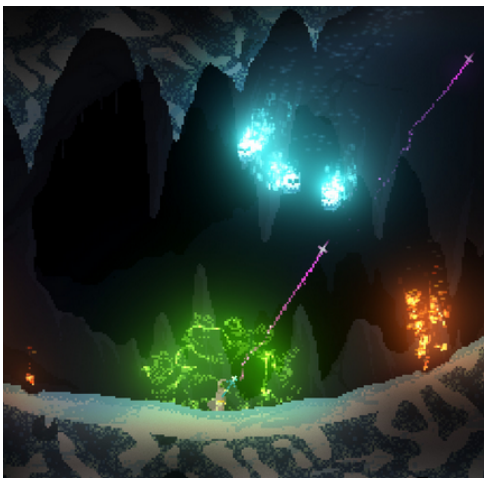
CSC417: Exploring the Tech of Noita

Howard Chen - chenhsu6

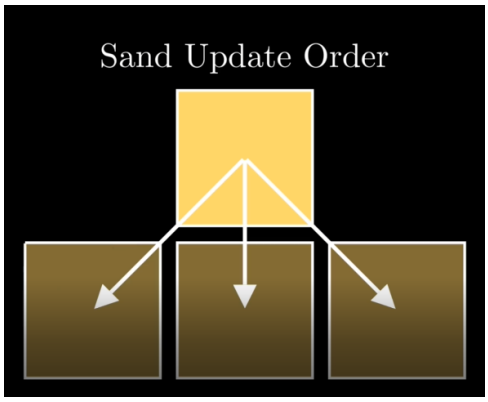
Contents

1 Abstract

In this paper, I discuss the technology and physics that powers the popular Roguelike game, Noita. Noita is a procedurally generated pixel-based game where every pixel is simulated in one of three different types of simulations.



The three different types of simulations are a Falling Sand, a Rigid Body, and a Particle Simulation. In this paper, I am going to explore the first two of these simulations, alongside various optimizations to improve the performance to run in real time.



2 Falling Sand

The physics of Noita is simulated using a singly-buffered grid. Like other cellular automatas, the grid is updated on a tick-by-tick basis and pixel states are dependent on its surrounding pixels in the previous tick. In particular, the falling sand simulation treats each grid "pixel" as one of many particle types, updated by a simple set of rules. For example, the movement of the sand particle can move it to one of the three downwards directions. If there is a solid particle blocking its movement, it simply stays where it is.

Water updates similarly, with the exception of checking its direct left and right neighbours as well. This update order gives rise to emergent behaviours such spreading out until it is level with the horizon.

The third main mechanic is the fire spreading. On update, there is a small chance (based on the material's `flammability` attribute) of catching on fire if one of its 8 neighbours is on fire. The fire particle's movement is deter-

mined by the material that caught fire, so oil will still flow, but wood remains static. After the fire burns out, the pixel is set to `air`

With these three simple mechanics, we can create a variety of different materials. Oil is water but flammable (its lower density has the added effect of floating on water), wood, cotton, and fuse are static non-moving materials with varying degrees of flammability, gunpowder is flammable sand, smoke is inverted water, and acid is like fire but with zero burntime (that gives the effect of eating away at solid materials).

However, because the simulation is singly buffered (to save on memory space) and we use a uni-directional grid update order, the movement of some particles will prioritize a certain direction. Utilizing a multi-directional alternating grid update order, we can eliminate this undesired behaviour and create a symmetric physics simulation.

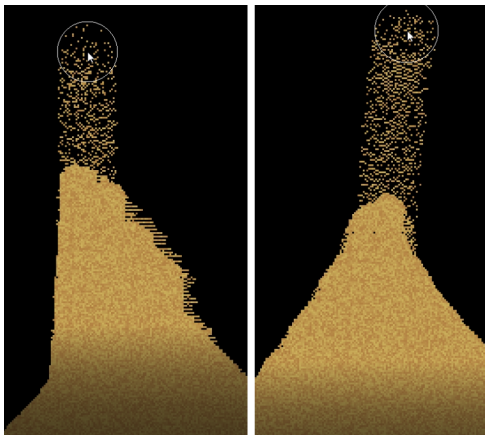


Figure 1: Left: Artifacting of uni-directional grid update order, Right: Altering update orders

3 Rigid Bodies

The rigid body simulation of Noita uses the rigid body solver, Box2D, with a translation layer. This translation layer translates between the Falling Sand and Rigid Body simulations, so that grid particles can influence the movements of rigid bodies, by creating a triangulated mesh from particle data.

To create this triangulated mesh, we use three algorithms: Marching Squares, Douglas-Peucker, and Ear Clipping Triangulation.

The first of these, the Marching Squares algorithm, creates counter clockwise contours around the boundaries of solid particles, and clockwise contours around holes. However, the Marching Squares algorithm has a side effect of creating too many contour segments (such as co-linear segments) with most of the segments not contributing to the overall rigid body simulation on a macro scale.

To eliminate these unnecessary segments, I use the Douglas-Peucker contour approximation algorithm. Essentially, this algorithm removes vertices that fall within a threshold ϵ of our approximation.

Then, using an ear clipping triangulation algorithm, we can convert the approximated contours into a triangulated mesh to feed it into Box2D. Us-

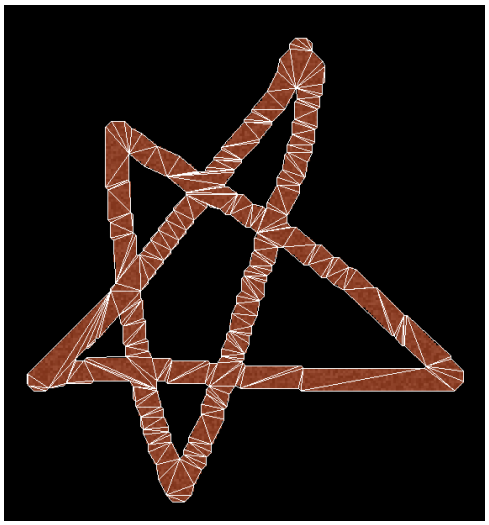


Figure 2: Meshifying particle data

ing this translation layer, it is possible to create any type rigid body object and let it interact with our falling sand simulation.

4 Optimizations

The rigid body translation layer is quite expensive to compute. This is because, without optimizations, we need to run the marching squares algorithm on every single pixel. However, an optimization we can implement is to only triangulate the pixel data in sections where there are rigid bodies. We can find these sections by querying Box2D with the corresponding Axis-Aligned Bounding Box. Skipping chunks where there aren't any rigid bodies yields a significant speedup.

Multithreading is also another option that we can implement. We can split the grid into 16 by 16 pixel chunks, and update these chunks in a checkerboard pattern. This is so that we can ensure particles can move 8 pixels in any direction without encountering a race condition. Using this chunk system also allows us to skip the updating of chunks where they weren't marked "dirty" by a previous tick.