I HATE EVERYONE

Julia Wang '17

OVERVIEW

I Hate Everyone (IHE) is a cheeky 2D particle system game in which you attempt to breed a deadly strain of bacteria to destroy the world. For this project, I built version 1.0 of IHE, which comes in 2 modes: a regular gameplay mode and a sandbox mode. On top of the base functionality given in the sandbox mode, the gameplay mode includes storyline developments, as well as time and resource constraints. **Disclaimer**: IHE does not condone mass genocide.

IHE's internal physics are quite simplistic, so I'm not breaking any new ground. There are certainly other 2D games out there based on particle systems. However, in my approach I attempted to give the system some more interest with several factors: 1) different types of particle behaviors and interactions, 2) biologically inspired growth and evolution heuristics, 3) UI animations for more fluid gameplay, and 4) an evil backstory. As the game is purely intended for entertainment, I think this approach works well!

IMPLEMENTATION

Overview

IHE runs completely from the client. The graphics make use of the following open-source libraries: **Velocity.js** for fluid UI animations that run faster and have a lot more extensibility than jQuery, and **PIXI.js** as an HTML5 canvas wrapper.

Although we have been using Three.js in class, I decided a 2D canvas wrapper was probably more suited to the project. PIXI.js does not provide any physics or vector classes; I implemented movement and collision detection myself.

Particle System

For the game to come together, the particle system needed to be able to handle complex collision behavior, as well as particle death. I made several implementation decisions to enable this:

1) For physics, I represented position as a 2D vector. Rather than do the same for velocity, I decided to use a combination of direction and speed. This allowed me to make each bacterium look more organic, as I could add a random turning velocity that made their paths arc naturally.
2) Although I drew sprites with complex shapes, I decided that the best way to implement collisions was to represent particles universally as circles. Different types of particles have different radii, from which I calculate collisions. Because individual particles are so small, this makes a negligible visual difference and makes the implementation much simpler.
3) Bacteria and powerups are spawned from separate engines, which keep track of their respective particles and update their positions.
4) The petri dish object handles collisions with itself, while the game engine calls collision handlers between bacteria and powerups. The bacterium and powerup objects each have collision handlers in their prototypes, allowing each particle to have its own interaction behavior.

Performance

Bacteria and powerups are rendered in separate containers. Bacteria, which are all rendered from the same sprite, are rendered in PIXI's performance-optimized but limited-functionality container. Powerups require different sprites and have short lifetimes, so they are rendered in a higher-functionality container.

Memory is managed in a pretty naïve fashion, but it seems to do the trick. Originally I simply marked particles dead after killing them, but this slowed the game immensely over time. To remedy this, on each frame I remove dead particles from the particle arrays of each engine. This way the game runs consistently at 50-60 fps.

User Interface

UI elements are rendered off the canvas, mainly because I am a lot more familiar with animating and interacting with non-canvas DOM elements. Velocity.js makes registering custom UI effects quite simple. For example, the following snippet is how I animate a bouncing effect:

```
Velocity.RegisterEffect('transition.bouncyIn', {

    defaultDuration: 1000,

    calls: [

        [{ opacity: [1, 0], translateY: [0, -1000] }, 0.60, {easing: "easeOutCirc" }],

        [{ translateY: -25 }, 0.15],
        [{ translateY: 0 }, 0.25]]
});
```

Although this kind of 2D animation is much more trivial than the animation techniques we went over in the course lectures, I think it’s still an important component of the look and feel of a graphics-based application.

GAME LOGIC

I wanted to make the bacteria more lifelike than just any 2D particle, so I took inspiration from natural heuristics to model living organisms. Each bacterium is assigned a random lifespan based on an exponential distribution, as well as a random growth rate, mutation rate, resistance, and infectivity based on uniform distributions.

When a bacterium undergoes cell division, it splits into two bacteria with slight variations in resistance and infectivity. This variation is based on the mutation rate. The powerups in the game also rely heavily on randomness:

- **Agar** increases a bacterium’s remaining life, up to its maximum lifespan.
- **Penicillin** kills a bacterium with (75% - resistance) probability.
- **Streptomycin** kills a bacterium with (85% - resistance) probability.
- **Ceftobiprole** kills a bacterium with (99% - resistance) probability.
- **Plasmids** have a 50/50 chance of changing the bacterium’s infectivity or lysing the bacterium to release more plasmids.
- **Mutagens** have a chance of killing the bacterium, increasing the bacterium’s mutation rate, creating a benign tumor (large mass of bacteria), or causing cancer (high growth, infectivity, and mutation).

LESSONS LEARNED

Probably the most instructive part of the project was building the particles and the physics engine from scratch. In the course assignments, the boilerplate code was written for us, so I was unfamiliar with working with the actual HTML canvas. The assignments also took care of code structure and basic performance and memory usage concerns, which I had to tinker with a lot on my own.

One pain point I experienced was with using the wrong library. When I was starting out I tried to use Paper.js, which claimed to be good for particle systems. Paper does draw primitives very nicely, but that's because it's a vector graphics engine, not an animation engine! When I tried to draw more than 200 particles the frame rate plummeted to near zero. In porting my code to PIXI.js, I had to give up Paper's handy functions for changing fill color and stroke of primitives on the fly. However, this was definitely worth the massive performance boost from PIXI's sprite-based architecture and support for WebGL hardware acceleration.