James Brayton

Andrew Hoyle

Nic Young

**Zombie Apocalypse**

Goal: Exploring the modeling and simulating of a zombie apocalypse programmed in Racket using Dr. Doug Williams inference engine and animated-canvas library.

Logic Rules - Andrew

Normally, a graphics program would be created using a continuous loop that juggles logic functions, an event handler, and a display function - think video game framework. Our group decided instead to make use of predicate logic search procedures to drive the zombie and people "actors". It was a challenge to get the actors to work in sync with one another through just the predicate rules. This was done with a combination of correct ordering of rules' priorities, and by implementing a "timer" within the rule space for every actor to reference.

The following rules were created in descending priority to run both the generation, movement, and interaction of the actors:

1. achieved goal check
   1. time limit reached / zombies win / people win / one left standing
2. random wall generation
3. actor attributes generation
   1. strengths and speeds given a random Gaussian distribution
4. battle distance check
   1. their strengths are compared, a stronger person destroys the zombie, a stronger zombie turns a person into a zombie
5. zombie-sees-person distance check
   1. zombie charges toward person
6. random walk
   1. no more zombies see any people
   2. pick random direction and use speed multiplier
7. draw dead actors (those recently bitten)
8. time increment / redraw buidlings
   1. all actors have taken a turn, now increment the game timer to give everybody a new turn

When the interaction rules made distance checks, they also had to make sure that the characters had the same "time-slice" values to ensure that everybody just got one move per time increment. This prevented a strong actor from wiping out all of the nearby ones he out-muscled. A timer data member was necessary also because the search engine just dwelled on one actor the whole time while the others remained "frozen."

Representing The Actors - Andrew

Every actor was randomly generated and represented as a fact in the rule space with the following characteristics:

('actor label coord-i coord-j ID time-slice strength speed)

label : 'zombie | 'person

Every time a movement was made, the i and j coordinates were updated by the inference engine, and the time-slice was incremented to represent that the actor's turn had been taken. If a zombie was killed, the fact would simply be retracted from the logic space. If a person was bitten, their label would be changed to 'zombie, and their time-slice would be incremented by a large value. This value would essentially freeze the new zombie in place because only actors in the current time slice are allowed to interact. Here's a drawing rule that looks for these fresh zombies using the timer. The binding to ?t-a is only made if the actor's time-slice is in the future:

(define-rule (draw-dead zombie-game-rules)

(?timer <- (time ?t))

(?actor <- (actor ?label ?i ?j ?ID (?t-a (> ?t-a ?t)) ?str ?sp))

==>

;; Nic's canvas

(let\* ((dc (send canvas get-dc))

(width (send canvas get-width))

(height (send canvas get-height)))

;; dead are grey

(send dc set-brush (make-color 0 0 0 .5) 'solid)

(send dc draw-ellipse (- ?i 5) (- ?j 5) 10 10)))

The other larger challenge was to remember all of the trig functions necessary to point the zombies toward any people they saw. It was a nice geometry-refresher to remember the importance of arctangent used with sine and cosine to derive the Cartesian values.

Animation - Nic

The animations involved make use of Dr. Williams animated-canvas library. The actors in the simulation, the zombies and humans, are represented as ellipse either colored red and blue respectively. The shades of color that they are actually colored depends on the strength of the actor itself, which as explained earlier, is assigned randomly with a Gaussian distribution. The darker either the shade of red or blue means the stronger the actor is in relation to the other actors. The actual individual drawing of the actors as they change position is done using a drawing context. As each rule in the inference engine is executed and an actor's position is changed this is sent to the canvas via the drawing context but not made visible to the user yet. The animated-canvas library actually operates with two canvases. One canvas is the one that is actually shown to the user, while the second canvas acts as a buffer and holds all the changes made to the canvas. When all of the rules have been executed the final rule we have actually steps the time forward. This is where the actual swapping of the two canvases happens. Once the canvas is swapped the new buffer canvas is then cleared and any new drawing changes will be made to this one. Using the animated-canvas library allows for smoother animations and better performance. At the time of this writing the random wall generation and drawing code is working and the actors cannot traverse through the walls. However, there are a few bugs that we have yet to work out. The main ones are that an actor can still spawn with in a wall which is obviously not ideal and the edge detection code between the actors and the wall could use some work.

[ Maybe James's part here?]