# Last of Sus : a Zombie Apocalypse simulation

William Santosa*, Arrian Chi†, Allen Liang‡, Brenna Kjorness §
Computer Science Department
University of California, Los Angeles
Email: *wsantosa@g.ucla.edu, †alienchi@g.ucla.edu, ‡aliang20@g.ucla.edu, §brennakj3@g.ucla.edu

***Abstract*—For our project, we created a simulation of a zombie apocalypse. The simulation shows emergent behaviors from three groups of entities: humans (prey), zombies (predator), and fungi (symbiotic relationship with predator). We found that the system starts out chaotic, but gradually descends towards an equilibrium. In this paper, we explain our idea, details of implementation, and observed behaviors of the simulation.**

***Index Terms*—Artificial Life, Simulation, L-Systems, Behavior Trees, Boids**

## I. Introduction

ZOMBIE apocalypses are a common literary trope in popular media. Usually occurring after a result of human negligence, the zombies wreak havoc in the city, while the humans plan and strategize their survival. Other biological lifeforms emerge as the plot progresses, such as evolved zombies, abominations, carnivorous plants, etc..

For our project, we took inspiration from the video game and television franchise *The Last of Us*. In the simulation, there are fungal growths that grow on zombies that have a symbiotic relationship with the zombies, specifically assisting them in hunting/following humans.

In the following sections, we will explain the role of each entity, their implementation, and our observations of the overall simulation.

## II. Zombies

Our simulation involves having zombie entities. Zombies will travel in hordes as much as they can and can detect the presence of humans. Usually they will wander aimlessly within their hordes. However, if they detect humans or fungal growths notify them of where they can find humans, they will chase after humans to attempt to consume them. If they are able to eat the human, the human becomes a zombie.

### A. Hording behavior

Since the zombies are supposed to travel in hordes, integrating boid behavior was the natural choice to give them this characteristic. According to this behavior, each zombie will classify other zombies into three categories. They need to be able to determine which zombies are part of its horde, the ones that are not, and the ones that are part of this horde, but too close to the zombie. For this behavior, each zombie needs to calculate three trajectories:

- **Alignment:** The average velocity of the group.
- **Cohesion:** The average position the group is heading towards.
- **Separation:** The direction to travel in to avoid others that are too close.

Afterwards, a weighted sum of these three values are taken to generate the final acceleration vector to apply to the zombie.

The process for doing so is as follows. For each zombie, they first determine which zombies are close enough to be considered part of the same horde by checking their distance from each other zombie. At the same time, a second distance check is run with a smaller distance value to determine the zombies that are too close. Every horde member's velocities and positions are then summed up separately. A separate summation of distances from zombies that are too close is also kept. Afterward, if the zombie is part of a horde, we can then calculate our three vectors. For each of the three, the process is similar: take the relevant summation, divide that by the count of neighbors or, in the case of the separation vector, count of zombies that are too close to get the average, normalize that result and multiply by a preset maximum force value (excluding the cohesion vector, where instead before normalization we want to subtract the current zombie's position from the horde's average position), and then subtracting the zombie's current speed from it, clamping the magnitude to keep it within the bounds of the preset max force.

Now that we have our three vectors, we take a weighted sum of the three to get the final acceleration vector to apply to the zombie. After some experimenting, setting the alignment and cohesion vectors to weigh the same while letting the separation vector weigh 80% more got the best results: the zombies were able to stick together in hordes if they happened to be in or approach one and they also did not collide with each other.

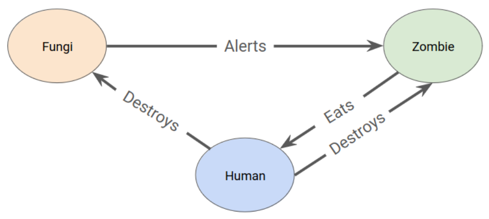


Fig. 1. Interaction Diagram Between Entities

### B. Behavior control

Human and zombie behavior is controlled via a behavior tree, which is a special type of decision tree with nodes that can be categorized into either control or execution nodes. Execution nodes can only be leaves, meaning that they cannot have any children of their own. They either execute an action or check for a condition. Meanwhile, control nodes only occur at non-leaf nodes. They control the logical flow and determine how to interpret the execution nodes' return values. The simplest version of a behavior tree has 4 types of control nodes and 2 types of execution nodes. However, it is common to add more node types with probability and other behaviors.

There are two different types of nodes that fall under the category of execution nodes.

1) **Condition Node**: Returns either success or failure after a single tick of the program. Influences the behavior of the tree and cannot be used to perform an action.
2) **Execution Node**: Executes an action and returns either success, failure, or running as its status. The running status indicates that the action is still being performed and can span over more than 1 tick. The action and condition nodes are usually depicted as a rectangle and as an oval, respectively.

Control nodes influence the flow of the program. They determine how to traverse the tree and how to interpret the return values of the execution nodes. There are 4 different main types of control nodes.

1) **Sequence Node**: Executes children in order until all return success or one returns failure.
2) **Fallback Node**: Executes children in order until all return failure or one returns success.
3) **Parallel Node**: Executes children in "parallel" (ticks in order one at a time).
4) **Decorator Node**: Manipulates the return value of a single child.

Our old behavior tree was extremely large, with over 60 nodes. Our new behavior tree simplified the logic and removed some redundant actions/behaviors, resulting in about half the nodes being utilized. Our behavior tree was also split into two separate behavior trees, one for zombies and humans. We realized that it made more sense to modularize the tree and separate the two, making it easier to work on each part separately.

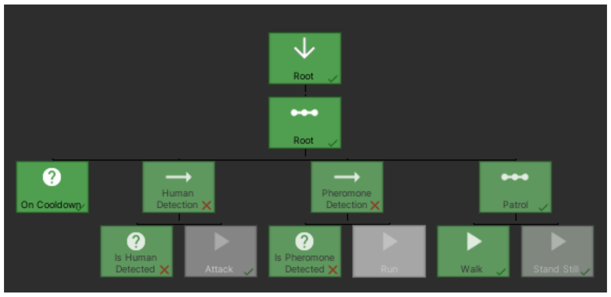


Fig. 2. Zombie Behavior Tree

The zombie behavior tree consists of nodes for checking cooldown (a random number of ticks between 10 and 30) and sequences for detecting humans, detecting pheromones, and patrolling.

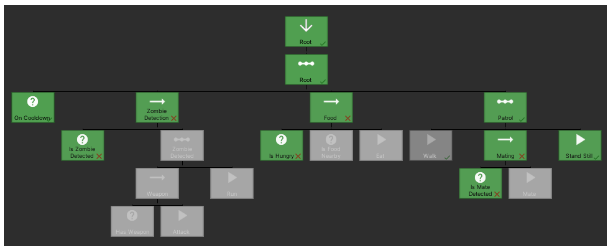


Fig. 3. Human Behavior Tree

The human behavior tree is more complex, consisting of logic for cooldown, detecting zombies, food, and patrolling. When detecting zombies, it checked if the human had weapons. If so, the human would run away. For food, it checked if the human was hungry and if it was then they'd search for food. Else, the human would patrol and mate when they're free.

## III. Humans

Humans, with more developed decision-making skills, require a more complex pathing system than the zombies. While the zombies tend to wander aimlessly with their pack, humans need to be able to independently identify and navigate to targets. This section will be broken down into perception, which is how both humans and zombies identify targets, and navigation, which is how humans path towards targets.

### A. *Perception*

Both humans and zombies use similar perception behavior, but with different parameters. Actors start with both hearing and vision fields around their current position to identify targets. These fields are represented by disks with radius and position. When running, humans trigger sound events that zombies can 'hear'. Zombies will hear a target if their hearing field overlaps with the field of the sound, using circle-circle intersection. In a similar fashion, humans and zombies will 'see' a target if it overlaps with their vision field, with the added constraint that they must be able to cast an uninterrupted ray to this target (seeing it), and it must also be within some field-of-view angle of their forward vector. This results in humans and zombies being able to hear in all directions, but only being able to see objects that are in front of them (shown in Figure 4).

Once a target has been identified, humans will add it to their 'memory', or a list of targets. This list of targets can then be scanned to make decisions for the behaviors defined in the behavior tree. After an adjustable interval, humans will remove old targets from their memory, 'forgetting' them. By adjusting the parameters, for hearing and vision radii, memory duration, and field-of-view angle, the humans and zombies can have different perception behaviors. In the show we were originally inspired by, the zombies are often blind, but can echolocate, so we tended to give zombies lower vision radii and FOV angles, but higher hearing radii.

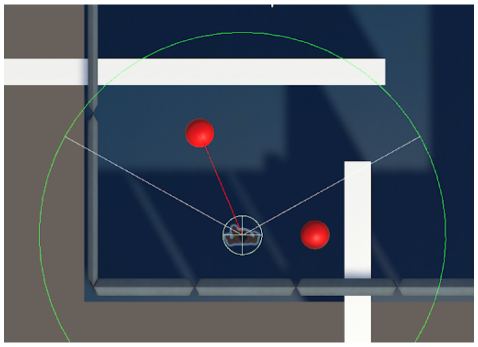


Fig. 4. A human's vision radius and a red ray cast to the target they can see

## B. Navigation

In order to navigate to targets throughout the map, humans need to be able calculate a path, follow this path, and avoid obstacles along the way. Path finding is typically composed of a graph representing the environment and an algorithm that calculates the most optimal path from a start node to an end node on this graph. In our case, we chose to use a grid to represent the environment due to their simplicity, ease of use, and adjustable precision. We implemented a grid 'generator' that takes in a desired size in the x and z dimensions in world coordinates, and desired node radius (the size of each grid cell). The generator then calculates a graph representing the environment, with functions to return the node associated with a world point, the neighbors of a node, and whether a world point is reachable and unblocked. If a section of the environment is blocked by walls or obstacles, these grid nodes are marked as unwalkable when the graph is generated, so they will not be included in path calculations.

**A* Algorithm:**

Our implementation uses the A* algorithm for calculating a path from one node to another on the environment grid. The A* algorithm is similar to the popular Dijikstra's algorithm, in that it uses edge weights to calculate the least-cost path to a particular node. It deviates from Dijikstra's by calculating the shortest path from just one node to the end node, rather than calculating all paths to the end node. Another important distinction is that Dijikstra's finds the actual shortest path to the end node, whereas A* estimates the shortest path to the end node, prioritizing speed and efficiency over correctness. In our implementation, the "cost" is distance, but other factors could also be included.

Starting from the first node, the neighbors of a node are examined. Node examination order is determined based on a priority queue. When examining a node and its neighbor, a new cost is calculated using the sum of the cost from the start until the current node and the cost of the current node to that neighbor. If a neighbor has been determined unwalkable by the grid, it will not be examined. If the new cost is the shortest cost to that neighbor found so far, the neighbor is added to priority queue. The priority is the sum of the new cost and some heuristic of choice that estimates the distance between the neighbor and the final goal. Because we utilize a grid-based graph, we chose to use Manhattan distance as our heuristic to

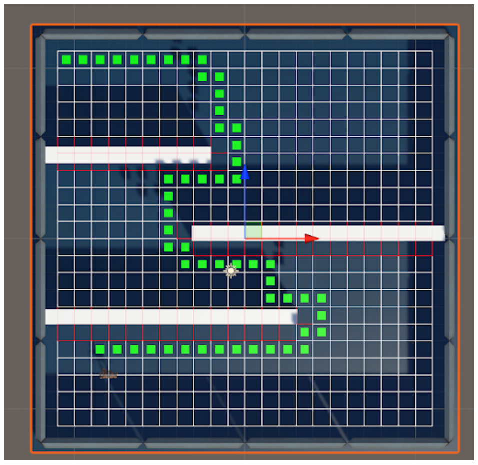


Fig. 5. A sample environment with a human's start and end positions, white grid lines, and green calculated path nodes

estimate distance. This heuristic is just an estimate because the algorithm has not yet calculated if there will be obstacles in the way of this path; the shortest path based on Manhattan distance may not actually be achievable or may become longer due to obstacles in the way. Once all the neighbors of a node have been examined, the algorithm dequeues the next node in the priority queue (with the next lowest cost) and examines its neighbors. This repeats until the goal node has been found. An example of this path-finding in action is shown in Figure 5.

**Following Path:**

Once a path has been calculated to the goal, humans need to be able to realistically follow this path. This is achieved by iterating backwards through the path and raycasting from the human to that path node to determine if the human can see the node, and setting the human's target to the furthest node on the path the human can see (Figure 6). As the human reaches its current target, this repeats to find the next target along the path, until the human has reached the goal node. This results in more natural human movement than following lines on a grid, especially in tight corridors and amongst many obstacles.

**Obstacle Avoidance:**

Alongside this global path following, humans also need to be able to avoid obstacles on a local scale. This is achieved in a similar way to the boiding behavior, using steering vectors. While walking, the humans cast 4 rays out in front of them (Figure 7). If a ray hits another object, the human applies some small velocity in the opposite direction of that ray. This helps humans naturally avoid other humans and walls when turning around corners and navigating through smaller spaces.

## IV. Fungi

The fungi plays the role of a pest against the humans and a mutual symbiote with the zombies. Humans act hostile towards the fungi, destroying them when they find them. However, the fungi releases pheromones which alerts the zombies that there is a human near the fungi. The zombies then navigate to that location, hoping for a scrumptious meal. In the end, the fungi

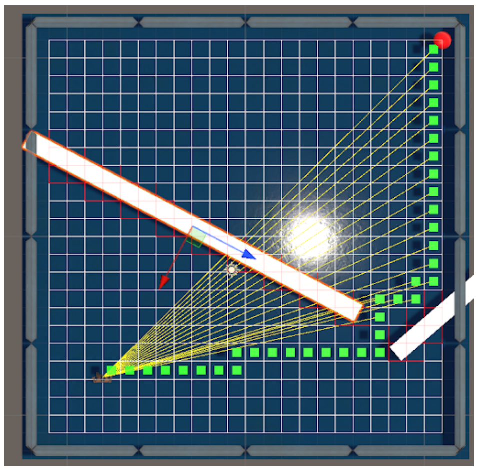


Fig. 6. A human casting yellow rays to each path location until it finds the furthest path node it has direct eyeline to

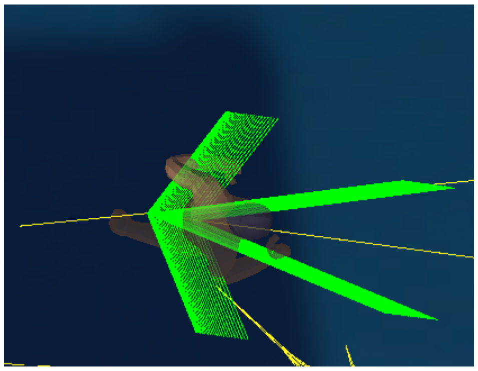


Fig. 7. A human casting green rays out in front of them to detect obstacles, with some rays from previous movement

helps the zombies with hunting, while the zombies drive away the fungi, highlighting their mutualistic symbiosis.

## A. L-system

In the simulation, the fungi is simulated with an Lindenmayer system (L-system, for short). We start with an initial symbolic representation of the system (the current sentence). A grammar contains substitution rules that each represent a growth pattern. At each generation, a symbolic update engine iterates through the sentence, replacing each character with the corresponding pattern in the grammar. Finally, a turtle renderer parses through the sentence and draws the design indicated.

Because the fungi must interact with other entities in the simulation, there are other features that must be implemented. Below, I go over the details of each major feature we added.

## B. Rendering

Figure 4 shows a list of the commands the turtle renderer (see python turtle module for more details) calls when it encounters each character. Each command corresponds to a modification of the turtle's state that enables it to draw the fungi. The commands most relevant to the fungi are the push and pop commands, which makes the turtle save its current

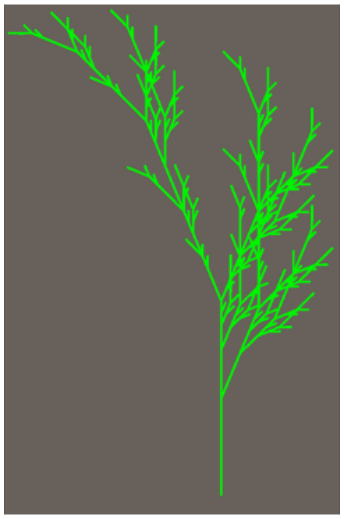


Fig. 8. A tree generated by an L-system

position and angle so that it can get back to it later in the operation. This enables the fungi to have branching structures that may also grow new branches.

One significant problem we faced when implementing this feature was creating a mesh out of the fungi. Unity has a line renderer function, but the documentation says that each line drawn must be associated with a game object. Given that the L-system algorithm is inherently exponential (the effects of substitution rules are compounded at each generation), the amount of game objects created would be concerning if we used the line renderer. Thus it is more efficient to create our own mesh for the fungi.

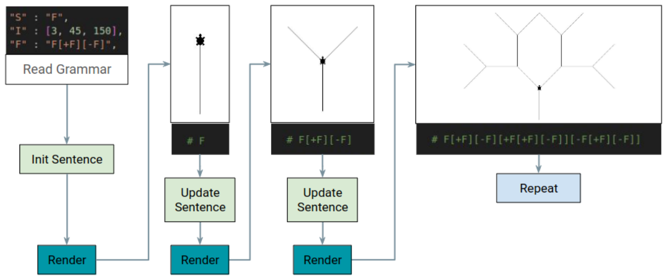
To do this, we kept track of the position of the turtle at each character and added vertices (and the respective triangle indices) every time the turtle moved forward. For branches to appear accurately, we need the turtle state pushed onto the stack needs to include the index of the vertex that the turtle is at. When the state is popped, we cache that index until the turtle moves in that branch (this is to handle the edge case where we immediately push after we pop).

## C. Animation

To animate the fungi (show the intermediate stages between each generation), we will interpolate the lengths and angles of the branches added for the new generation. However, we want to control which parts of the patterns are part of the old generation and the new generation. To accomplish this, we add new commands '(' and ')' that enclose these "new growths" in our grammar. The turtle renderer will turn on interpolation upon encountering a '(' and turn off interpolation for ')'. The interpolation factor is dependent on a timer variable capped at a target time that is updated every frame and reset at the start of every new generation. The result is a continuously growing mesh that looks alive.

## D. Collision

Because we want our humans to destroy the fungi, the fungi needs to have collision geometry. The collision geometry



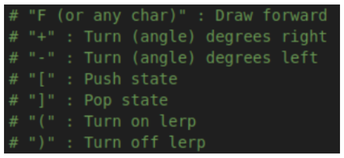
```
"S" : "F",
"I" : [3, 45, 150],
"F" : "F[+F][-F]",
```

```
# F
```

```
# F[+F][-F]
```

```
# F[+F][-F][+F[+F][-F]][-F[+F][-F]]
```

Fig. 9. Overview of the L-system Algorithm



```
# "F (or any char)" : Draw forward
# "+" : Turn (angle) degrees right
# "-" : Turn (angle) degrees left
# "[" : Push state
# "]" : Pop state
# "(" : Turn on lerp
# ")" : Turn off lerp
```

Fig. 10. Commands the turtle renderer reads

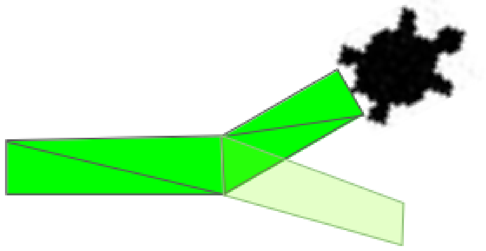


Fig. 11. The turtle must keep track of the back indices in the stack when creating a new branch off of another.

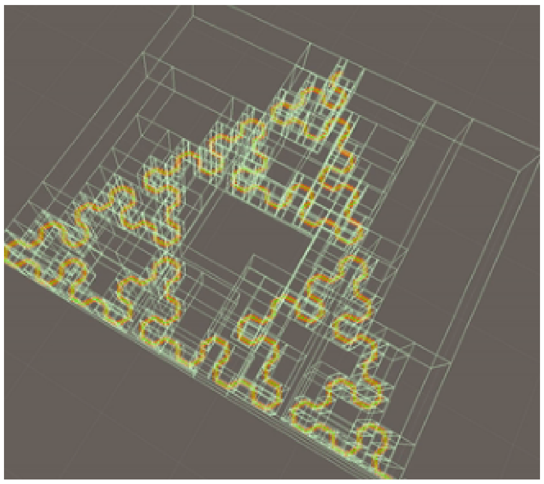


Fig. 12. Bounding volume hierarchy of the Sierpinski triangle

needs to provide us with the exact segment the object collided with (we want to destroy the branch of the fungi from that segment). There are multiple ways to implement this in our simulation. First, we may create a large axis-aligned bounding box (AABB) to enclose the entire fungal structure. While it is efficient, it comes at the cost of accuracy, because in addition to the fungi, the box encloses the space surrounding the fungi (see figure ?). On the other hand, we may use an AABB for every segment of the fungi. However, this is computationally inefficient because we create many boxes (and game objects) and we need to iterate through every segment per collision check (O(n)).

Clearly, we have a dilemma to balance between accuracy and efficiency. To have the best of both worlds, we organized the bounding volumes into a bounding volume hierarchy (BVH). The idea is to enclose the AABBs (children) into larger AABB (parent) and then enclose those AABBs into another even larger AABB until we create an AABB that encloses every single smaller AABB. To sum it up, the hierarchy creates a tree of bounding volumes, where each parent is guaranteed to enclose its children and its leaves are AABBs of each segment. In our implementation, we built the BVH incrementally as the turtle parsed through the current sentence (this will be important in the next section).

This makes every collision check a O(log n) tree traversal. If an object collides with a parent volume, then it is worth traversing its children to find the segment that the object collides with. If there are no colliding segments, then we may check the parent's siblings. If no colliding segments are found throughout the entire tree, then there is no collision.

This does raise the question of how we decide which boxes to enclose. This is left up to the programmer's discretion. For instance, one may enclose the boxes which are the closest together, use the median of the overall bounding box, or group boxes such that the area of each AABB is minimized. We use the final heuristic for the simulation as it seems to yield more accurate results.

*1) Problems with sorted input:* After our first iteration of the bounding volume hierarchy, we found that the BVH was unbalanced (structurally, a linked-list). This is a problem with the incremental BVH construction. Because we are creating enclosing AABBs for segments that are most likely close to each other, the AABBs are more likely to enclose the new AABB created for the previous segment. The algorithm would create boxes that nest each other over and over (refer to figure ?). This is bad for efficiency because our search becomes O(n) (which is no different than iterating and checking every segment). This is essentially the problem of an unbalanced tree. To alleviate this problem, in other problems, we use AVL trees, which have a self-balancing property (guaranteeing that the height of the tree is at most log N). It employs tree rotations to make sure that the tree is balanced and valid. We may use this same idea here: we find alternate valid representations of the hierarchy that yield the next best results (with respect to the heuristic). This ensures that the tree traversal is still O(log n) and valid.

*2) Breaking branches:* Once we obtain the segment where the collision occurs, we want to delete the rest of the branch

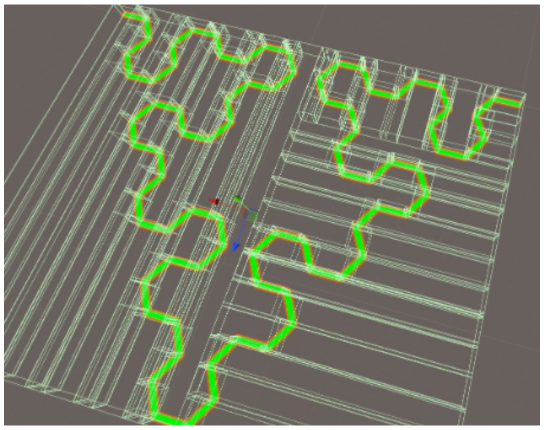


Fig. 13. Nested boxes as a result of an unbalanced hierarchy

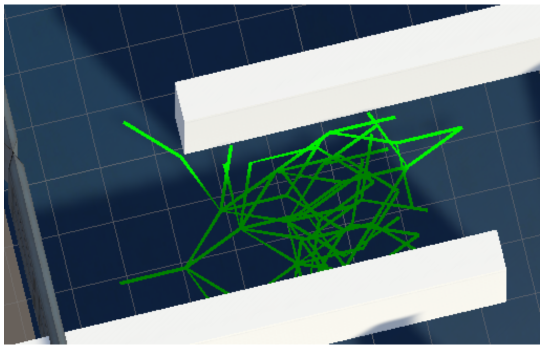


Fig. 14. Fungi cannot grow past walls, filling the corridor as much as possible

starting from that segment. We may accomplish this by modifying our symbolic update engine so that it can delete the corresponding substring (of the deleted branch) from the current sentence. So in addition to the segment collided, we also need to obtain the index of the character the segment represents in the sentence. The substring from that index to the first occurrence of ']' (of the same branch of the index, not a child branch) is deleted from the sentence.

Once the sentence is modified, the fungi must be rendered again with the correct visual and physical representations. This operation must be done with care. If multiple collisions occur at the same frame, it would be naive to apply them in the order they are processed. If this is done, it is possible for indices of some collided segments to become invalid because previous processed segments have deleted them. Collided segments must processed in descending index order (as they appear in the symbolic representation) to prevent out-of-range exceptions. In addition, all sentence updates must be done before the next render because rendering causes the BVH to change, invalidating any collided indices found in the previous frame.

## *E. Stochastic L-system*

Finally, we want our L-system to have variety and randomness. This can be easily implemented by making our grammar stochastic. For each pattern of a production rule, there is a probability associated that determines how likely that pattern is used. The symbolic update engine would roll to decide which pattern to use, resulting in random, spontaneous growths, even when the system works with the same grammar. Not only does this create a variety of patterns, it allows our fungi to branch out into arrows that it was not intended to reach.

## *F. Optimizations*

The L-system algorithm is inherently a slow algorithm. The growth of the fungal structure is exponential because each character is replaced with a string that may contain multiple instances of that same character (F -¿ FF -¿ FFFF -¿ ...). The turtle renderer is implicitly single-threaded (each segment's location is dependent on the segments before it), so there is no way to clear-cut way to parallelize rendering. These reasons make L-systems potentially laggy when segment counts are too high.

There are some optimizations we implemented to mitigate these troubles. First, we do not render new geometry at every frame. The clearest candidate for saving computation time is for the lerped stages between generations. We may consider these transitions from one generation A to another generation A', although upon closer inspection, we are computing the geometry of A' in the beginning of that period (t = 0). A lerp only involves changes in the vertex coordinates not the overall triangle structures (indices) of the mesh, so we only need to recalculate the vertex coordinates and AABBs for each segment at each frame (as opposed to generating new coordinates/AABBs).

When branch deletions happen, the sentence is reconstructed. But the sentence may now have redundancies i.e. empty parentheses/brackets, nested parentheses, etc. Removing these helps the turtle renderer not waste time popping and pushing off the stack. Also pertaining to branches, the number of segments may be limited/capped to a certain amount (instead of letting it grow infinitely). This is analogous to a a natural growth capacity many organisms have. This should be tweaked with care however, since L-systems are exponential algorithms (implying segments may increase exponentially as well, potentially overshooting the segment cap by a lot).

Some other optimizations we could've done included parallelizing symbolic update, caching rotations, and memoizing organ structures. First, the sentence update is an embarrassingly parallel problem, since each substring's successor is independent of another's. So we could've divided the string into multiple workloads, processed each on a different core in parallel, and joined the threads to form our successor sentence. Next, since we know that the angle is constant, there exists a set amount of turns such that we will reach the same angle that we started. These values may be cached, so that there is no need to recalculate the quaternions during runtime. Finally, we could do the precomputation per pattern and memoize the vertices/indices of the mesh so that at generation time, the turtle skips the computation and just dumps the precomputed vertices (scaled and rotated, of course) at its current location. Although these ideas are all sound, we sadly didn't have enough time to implement them. We look forward to looking at them at a later date.

## V. Results

When we put each component together, we observe the system develop some emergent behaviors. First off, the most basic behavior is that if a human gets too close to a zombie, the zombie horde will go after it, killing the human. This utilizes the zombies perception, as well as its boid movement (for hording) and defined behavior tree. Even though zombies have a limited range of perception, the zombies aggressive and cooperative behavior helps it hunt down humans one by one.

Another observation is that when humans collide with the fungi, the zombies get alerted to them and navigate to that location. Although this happens sometimes (when there is no wall between the zombie and fungi), the fungi does symbiotically helps the zombies when they are hunting. We would also like to note here that the fungi, because we didn't disable collision with walls and allowed the fungi to regrow, is able to fill up an entire space and not grow through environment geometry. This creates the a floodfill effect, making the fungi fill up spaces as much as possible and creating a barrier in the maze that humans must pass through (if they dare).

Finally, we found that the zombies are a bit too overpowered. Like the movies in popular media, it seems the zombies almost always finds the humans. There are times when the humans can survive however. Because the zombies have limited perception, the humans could wander and hide into a corner or behind a wall, and if they are fast enough, then the zombies will be unlikely to find them, falling back onto their default random targeting behavior.

## VI. Conclusion

In conclusion, our simulation shows the interactions between three types of entities playing 3 different roles: prey, predator, and symbiote. In this simulation, the predator (zombies) wipe out the prey (humans) with help from the the symbiote (fungi). Each contribute to the emergent interactions of the simulation and the consequences that come out of them.

If we had more time, we would spend it on tuning the behavior tree and ensuring the simulation is more cohesive. The humans and zombies have suboptimal pathing and sometimes get stuck and clump to each other in the simulation. The fungi are also still a bit too slow, so we would also investigate that. Overall, we've accomplished the goal we started out with: to create a simulation of a zombie apocalypse. In fact, if we examine the simulation literally, it corroborates with popular media, stating that humans would probably die out in a zombie apocalypse if the zombies have a symbiote.

## References

[1] Robohub, "Introduction to Behavior Trees," Robohub, [Online]. Available: https://robohub.org/introduction-to-behavior-trees/. [Accessed: 19-Jun-2024].

[2] J. A. Tunbridge and M. A. Jones, "An L-systems approach to the modelling of fungal mycelium," Semantic Scholar, [Online]. Available: https://www.semanticscholar.org/paper/An-L-systems-approach-to-the-modelling-of-fungal-Tunbridge-Jones/26cf8145a41a725df4241f5d0a33ce56b0e91dab. [Accessed: 19-Jun-2024].

[3] E. Catto, "Dynamic BVH," Box2D, [Online]. Available: https://box2d.org/files/ErinCatto_DynamicBVH_Full.pdf. [Accessed: 19-Jun-2024].

[4] D. Terzopoulos, "Deformable Models," UCLA Computer Science, [Online]. Available: https://web.cs.ucla.edu/ dt/papers/gmod07/gmod07.pdf. [Accessed: 19-Jun-2024].

[5] W. Shao and D. Terzopoulos, "Autonomous pedestrians," UCLA Computer Science, [Online]. Available: https://web.cs.ucla.edu/ dt/papers/gmod07/gmod07.pdf. [Accessed: 19-Jun-2024].

[6] R. RedBlobGames, "Introduction to A*," Red Blob Games, [Online]. Available: https://www.redblobgames.com/pathfinding/a-star/introduction.html. [Accessed: 19-Jun-2024].

[7] M. Zucker, "Optimizing L-systems," [Online]. Available: https://mzucker.github.io/2020/03/28/optimizing-lsystems.html. [Accessed: 19-Jun-2024].