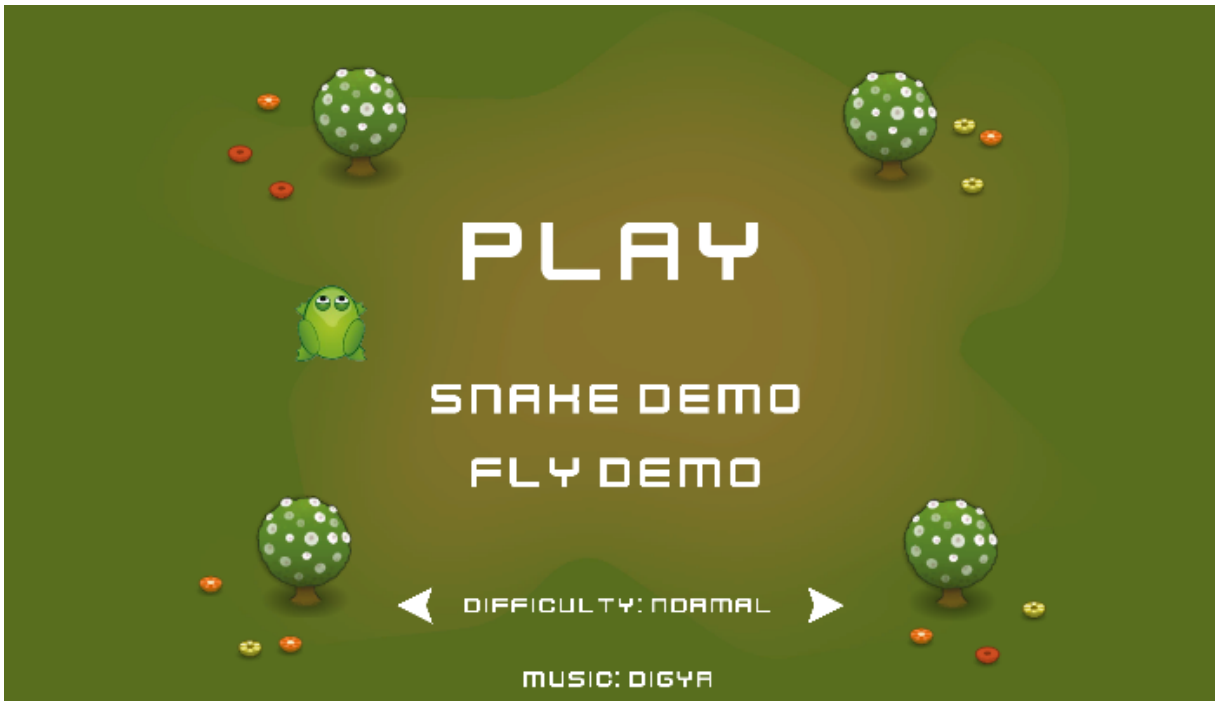
Games and Artifical Intelligence Techniques
Assignment 1

Team 3
Michael Dann and Jayden Ivanovic

September 7, 2014



1 Game Overview

In our game, the player controls a frog that sits in the middle of a food chain. The goal of the player is to eat as many flies (prey) as possible while avoiding the snakes (predators). The flies and the snakes are non-player characters (NPCs) and are controlled automatically through artificial intelligence techniques.

The frog begins with three hitpoints and loses one each time it is hit by a snake. If the frog runs out of health then it dies and the game is lost. To win the game and advance to the next difficulty, the player must eat a certain quota of flies.

The snakes are capable of laying eggs, which eventually hatch and spawn new snakes. It is in the player's best interests to walk over eggs and smash them before they hatch. The player can avoid snakes by entering a lake, where the snakes are not able to follow. The player can also shoot bubbles at the snakes, which temporarily disables them. When a snake is disabled the player can push it into a lake to drown it and remove it from the game.

2 Team Member Contributions

Jayden implemented the steering behaviours, while Michael focussed on pathfinding. Everything else (state machines, game mechanics, music, graphics, etc) was a collaborative effort, since we found that the easiest way to share and develop ideas.

3 Steering Behaviours

The flies' target velocities are calculated as a sum of weighted behaviours. The behaviours we implemented are seek, flee, wander and flocking (via composition of alignment, cohesion and separation components). From these underlying elements we noticed emergent behaviour: generally, each fly heads towards some randomly chosen apple tree along a wavelike path. Once there, it groups with other flies around the tree, sometimes waiting for others to leave (due to the separation component of flocking). An example of this can be seen in Figure 1.

When the flies come within close proximity of the frog, they flee in the opposite direction. However, if the frog is hiding in a pond then the flies have no knowledge of its whereabouts. The predators also ignore the submerged frog as they cannot swim! This can be seen in Figure 2. This mechanic allows the player to ambush flies while remaining safe from predators.

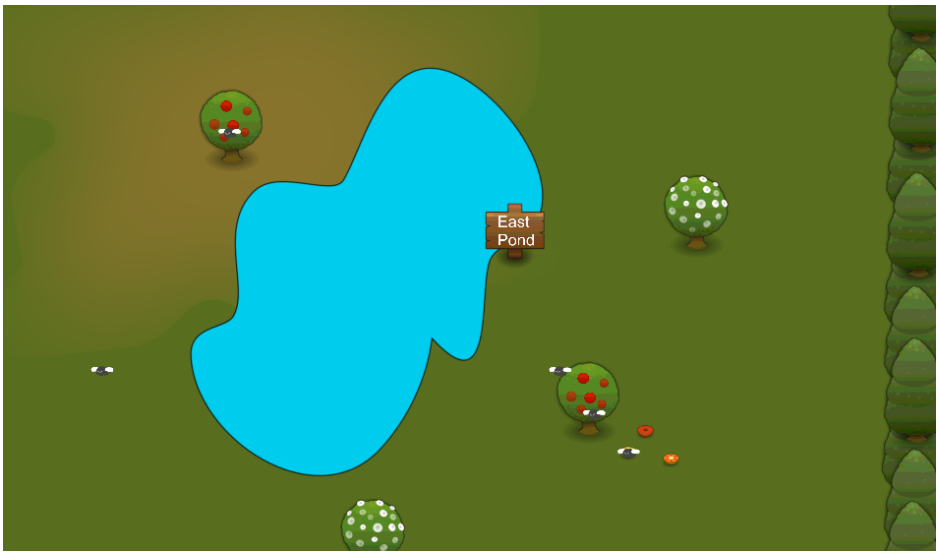


Figure 1: Flies hovering over apple trees to fill their hunger requirements.

Figure 2: The frog is hiding in the pond so the flies and snake ignore its presence.

3.1 Algorithm

We used an algorithm similar to the one introduced in Week 3 to combine our behaviours. It is illustrated in Algorithm 1.

Data: Agent's current position, velocity and neighbours(for flocking).
Result: The agent's new target velocity.
initialization;
targetVelocity ← zero vector;
steeringComponents ← steering behaviours attached to game object;
while *all steeringComponents have not been processed* **do**
    current ← get next component;
    weight ← get next component weight;
    targetVelocity = targetVelocity + current.getTargetVelocity() * weight;
end

Algorithm 1: Algorithm used for combining steering behaviours.

When we wish to alter the overall behaviour of NPCs (for example, to make the flies flee), we change their component weightings via the appropriate finite state machine. This allows us to achieve a range of emergent behaviours from the same underlying components. See Section 4 for the details of the FSMs.

3.2 Challenges

In this section we look at the particular challenges, issues and decisions we tackled in regards to the steering behaviours implemented for this assignment.

3.2.1 Smart Predators

The snakes were originally designed to chase the frog with a naïve "seek" behaviour, but they tended to bunch up behind the player, making them relatively easy to avoid. In real life predator-prey scenarios, predators generally take the velocity of their prey into account. Therefore, we redesigned the chase logic so that the snakes try to intercept the frog. When a snake is so far behind the frog that intercepting is impossible, the snake moves parallel to the frog in a flanking maneuver. Figure 3 shows the resultant behaviour from our new logic. Despite the fact that the snakes are acting indivually, there is a strong illusion that they are co-operating to surround the frog.

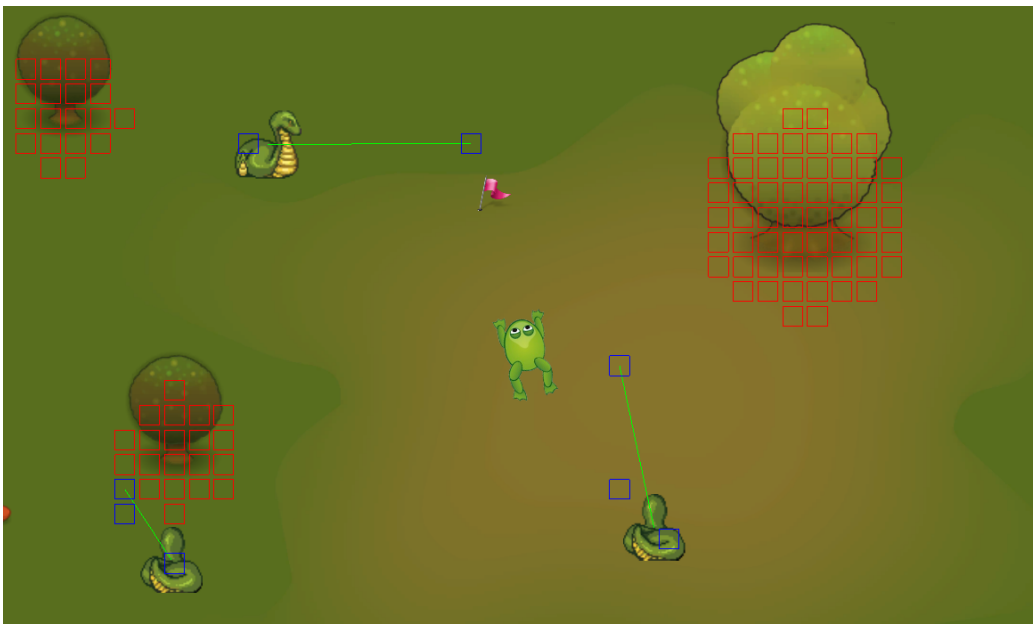


Figure 3: The predators effectively "box in" the frog. The bottom two snakes run parallel to the frog while the top one blocks the destination point, as indicated by the flag.

For a snake to intercept the frog, the time both parties take to reach the intercept point must be equal. Referring to the example in Figure 4, this gives us:

$$
\begin{aligned}
& t_{player} = t_{snake} \\
\Rightarrow & \frac{d}{p} = \frac{\sqrt{a^2 + (d-b)^2}}{s} \\
\Rightarrow & \frac{d^2}{p^2} = \frac{a^2 + (d-b)^2}{s^2} \\
\Rightarrow & \frac{d^2 s^2}{p^2} = a^2 + d^2 - 2bd + b^2 \\
\Rightarrow & (\frac{s^2}{p^2} - 1)d^2 + 2bd - a^2 - b^2 = 0
\end{aligned}
$$

which can be solved for d using the quadratic formula. If either d or the discriminant of the quadratic equation is negative it means that the player cannot be intercepted.[1]

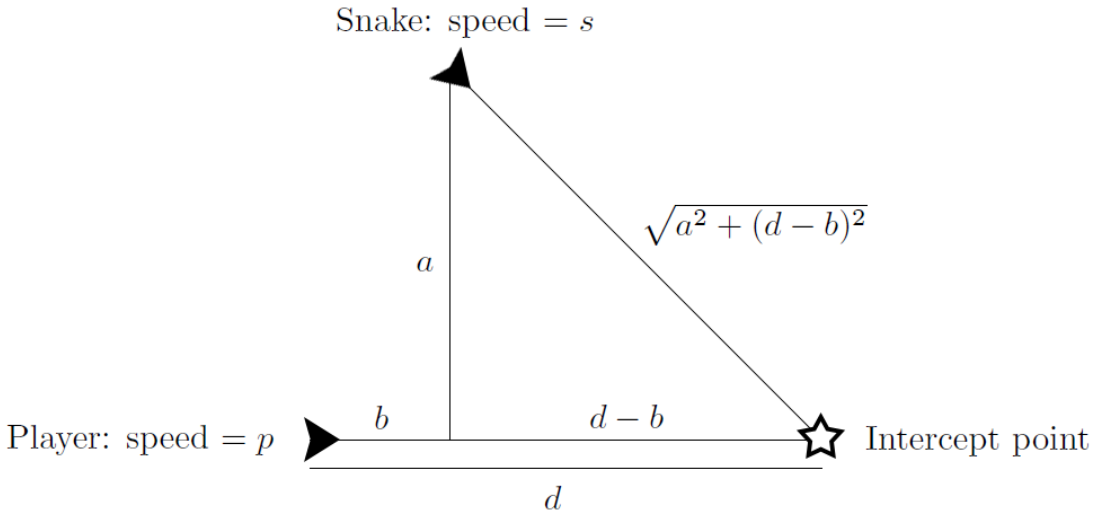


Figure 4: An example that illustrates how the intercept point is calculated. The maths is slighly different if b is negative (i.e. if the snake starts behind the player along the player's direction of travel).

3.2.2 Alternative Fly Behaviours

We attempted to make the flies appear more intelligent by biasing their apple tree selection towards nearby trees. However, this had a negative effect on gameplay since their behaviour became quite predicatable. The flies tended to stay on one side of the map for an extended period of time, oscillating between the two closest trees.

We also tried limiting the number of flies that could eat simultaneously at an apple tree to reduce the problem seen in figure 5. However, it looked unnatural when a fly travelled a long distance to reach a tree, only to give up at the last instant and fly back to where it came from. It also resulted in there being roughly equal numbers of flies at each tree, which was boring from a gameplay perspective since there was no motivation for the player to change locations until all flies around the tree were devoured.

Ultimately, we found that having the flies simply choose a random tree gave us more desirable behaviour. The additional rules that we believed would make the game more interesting actually had the opposite effect. The first modification meant that there were sometimes no flies in one half of the map (after they had all been eaten), while the second modification meant that they were too evenly distributed. With random tree selection, there were generally flies in each quadrant of the map, but the distribution was uneven enough to make the game interesting.

[1] If $d < 0$, it implies that the player needs to travel backwards, which corresponds to a solution in the past.

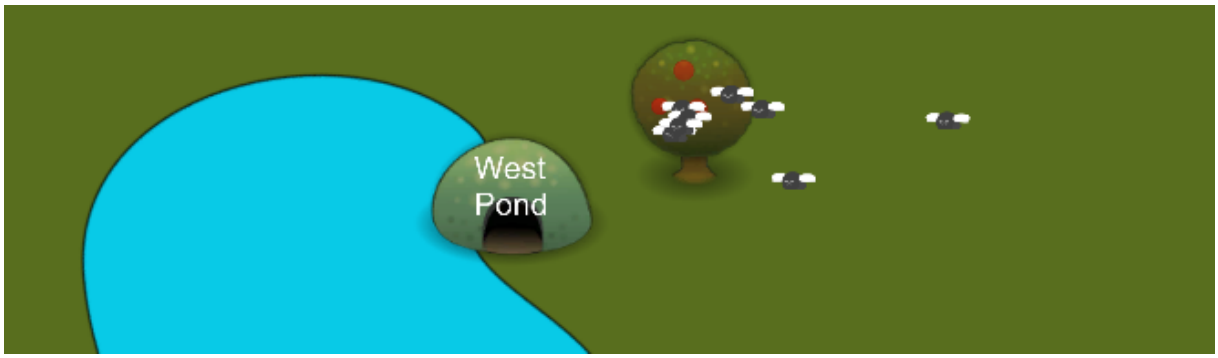


Figure 5: When the flies were configured to favour nearby apple trees, they tended to bunch up so much that it was easy for the player to eat them all.

3.2.3 Obstacle Avoidance

The flies in our game do not use any form of pathfinding. If they were limited to seeking, wandering and flocking alone then they would tend to get stuck against walls when trying to move between apple trees. To handle this problem, we introduced obstacle avoidance for the flies. Their obstacle awareness is limited to a small nearby radius, but the method we used is sufficient to prevent them from getting stuck in most situations.[2]

When a fly encounters an obstacle, it begins scanning left and right, looking for the narrowest deviation it can take to avoid colliding. Once it finds an unobstructed path, it remembers the direction it chose to deviate (left or right) and continues deviating to that side until the path towards the apple tree becomes free. The deviation angle is limited to a maximum of 100° so that the fly cannot get stuck oscillating backwards and forwards.[3] If the fly cannot find an unobstructed path, it will begin scanning in the opposite direction (see Figure 6c).

To see the flies' obstacle avoidance in action, select "Fly Demo" from the menu screen of our game. Turn on "gizmos" in order to see the red and green lines. Note that the flies must avoid the leaves of the trees, while snakes and frog must avoid the trunks. We discuss obstacle avoidance for the frog and snakes in Section 5: Pathfinding.

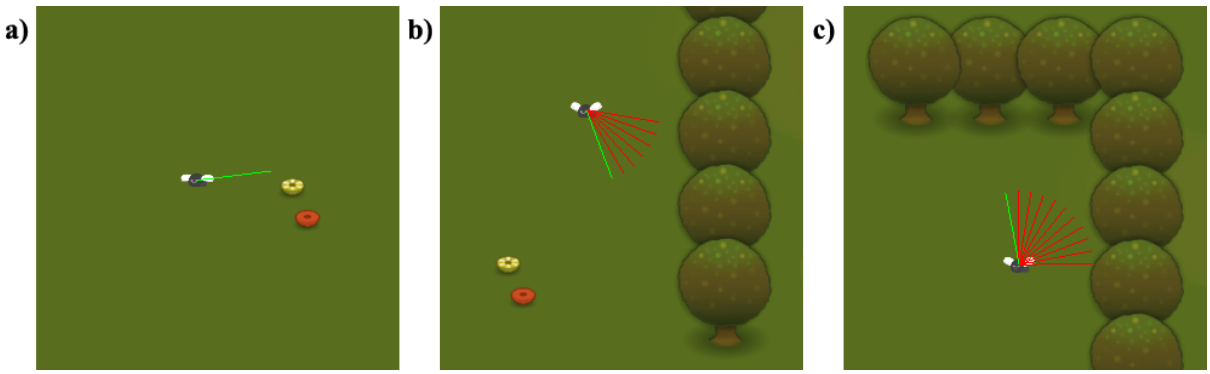


Figure 6: (a) The fly is travelling happily towards an apple tree, unobstructed. (b) The fly is currently deviating right. The red lines indicate directions in which the fly has detected an obstacle, while the green line is the narrowiest deviation it can take to avoid colliding. (c) Since the fly's detection range is limited it has not yet detected the wall above. The maximum deviation allowed is 100°, so once the fly detects the top wall it will flip directions and try scanning right.

[2]If a fly encounters a "bowl" shape then it can still get stuck, but it is hard to see how to avoid this problem without explicitly using pathfinding. In our game, if a fly detects that it is not moving then it will target a different apple tree as a last resort.

[3]We originally limited the deviation to 90° but found that a slight buffer past the right-angle gave better performance.

4 Finite State Machines

In our project we implemented finite state machines (FSMs) for both the flies' and the snakes' behaviours. In this section we discuss both FSMs and provide details regarding their implementation and design.

4.1 Flies

Figure 7 shows our FSM for fly NPCs. It is quite simple since the flies have limited functonality and essentially just roam the map. Likewise, our implemenation is straightforward: each fly's state is stored as an enumerated type and updated by a series of "if" statements. The state is then used to determine the weight of each of the fly's steering behaviours.

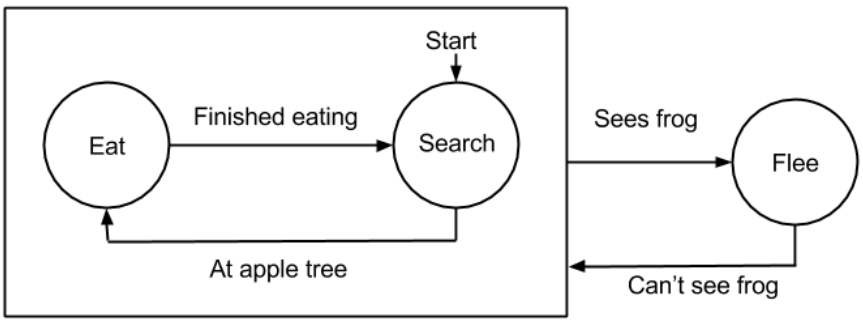


Figure 7: FSM for the fly game objects.

4.2 Snakes

Figure 8 shows our FSM for snake NPCs. The extra complexity here versus the fly FSM is due to the fact that the snakes have many more interactions with the player and the game world. By consulting the diagram we were able to understand and tweak our implementation much more easily than by studying the corresponding code. Additionally, the model allows us to confirm that our state transitions are deterministic.

4.3 Algorithm

Algorithm 2 depicts how we use FSMs in the Unity3D game engine:

Data: Agent specific data (i.e. hungry, position, desires)
initialize agent variables;
while *Game is running* **do**
    UpdateAgentState();
    Transition();
    Act();
end

Algorithm 2: General flow of our FSM implementations.

UpdateAgentState determines the agent's state variable, then *Transition* updates the relevant steering behaviour weights and other variables. *Act* examines the agent's current state and executes the appropriate behaviour via a switch statement. Note that in the source code, *UpdateAgentState* and *Transition* have been combined into the one method. *Act* corresponds to the Unity3D *Update* method. We present it differently here for readability.

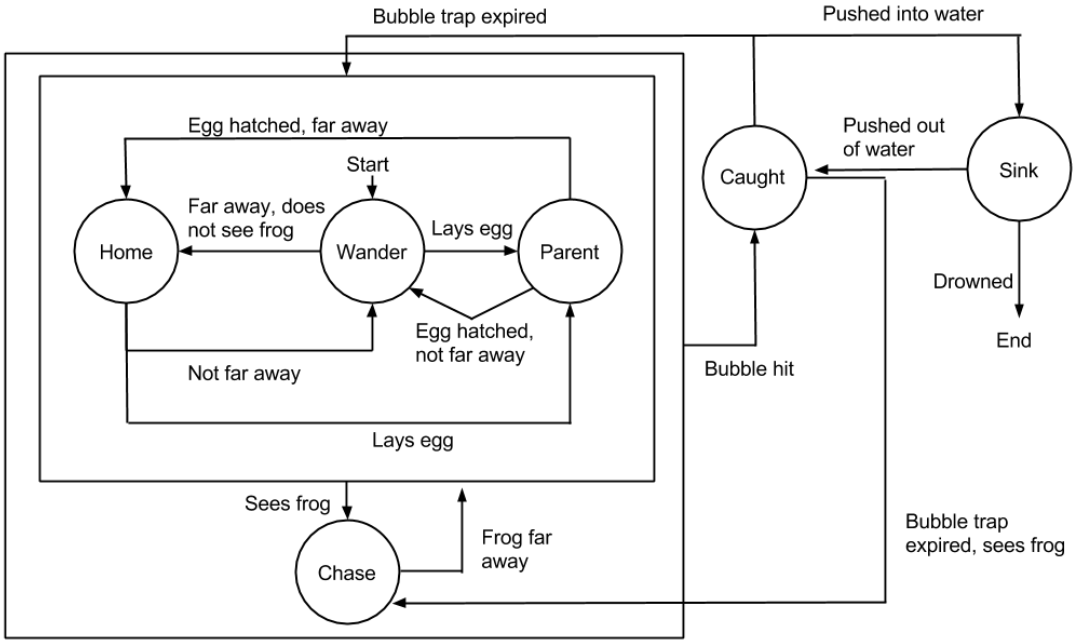


Figure 8: FSM for the snake game objects.

4.4 Challenges

In this section we look at the challenges we encountered during the implementation of our finite state machines.

4.4.1 FSM Complexity

Implementing the fly state machine was trivial, but the snake FSM was far more challenging. The snakes ended up with a *lot* of variables that had to be managed during state transitions, and keeping track of them all when we added new functionality soon became problematic. Our simplistic design allowed us to get up and running quickly, but in the future we would be much more inclined to draw a diagram from the outset and follow an architecture such as one of those recommended in the course text book. For example, creating a class based on transitions which has the abstract methods EntryActions() and ExitActions() would have reduced some of the above issues.

4.4.2 Rapidly Oscillating Transitions

Many of the snakes' transitions are dependent on distance calculations (such as the proximity of the player, or distance from home). We initially had problems when the snakes were right on the cusp of making such a transition. For example, the frog could wander in and out of the snake's range, and the snake would madly run back and forth, chasing the player then immediately giving up. A similar problem can occur with the use of trigger colliders in Unity; we found that they tend to flicker between "trigger enter" and "trigger exit" events when two objects are *just* touching/separated.

Our solution to this problem was to introduce various timers to control the speed at which transitions could occur. For example, once a snake has begun chasing, it will not stop chasing until 5 seconds have passed, regardless of how far it ventures from home (this is omitted from the diagram for readability). An alternative solution would have been to use soft cutoffs between regions, similar to how the Melbourne train lines transition between zones. However, this would have been difficult to implement for the trigger problem since we would have had to add multiple triggers to each snake, and trigger events in Unity do not specify which trigger was activated when there are multiple triggers on a game object.

5 Pathfinding

We implemented two pathfinding algorithms: plain A* and A* with Jump Point Search (JPS). We assume that the reader is already familiar with basic A*, so we focus here on the technical aspects of our implementation rather than the underlying algorithm. We give a brief description of JPS in Section 5.2, but for more details please refer to Daniel Harabor's paper, *Fast Pathfinding via Symmetry Breaking*, which is available on the course Blackboard.

5.1 A*

Our A* implementation behaves as if there is an invisible grid laid over the entire game world. We do not rely on obstacles being aligned exactly with the grid, but rather use raycasts to determine which nodes are blocked. This approach provided two main advantages: it allowed us to use curved obstacles, and it meant that we could adjust the granularity of the grid on the fly to make it as fine as possible without causing slowdown.

The majority of the obstacles in our game are static, which meant that we could use a pre-calculated grid for most pathfinding. However, we found that the snakes had a tendency to get blocked by their own eggs, which can be laid anywhere, at any time in the game. To get around this, we adjusted the algorithm to also calculate the set of squares temporarily blocked by eggs. Since there are rarely multiple eggs active at the same, and since they can be iterated over easily within Unity, this calculation is cheap enough to perform each time a path is calculated.

We wanted the underlying grid to be compatible with JPS, so we include diagonal neighbours in each node's adjacency list. (Many A* implementations assume that the only direct neighbours are up, down, left and right, but as we shall see in the next section, JPS requires diagonal neighbours.) Diagonal movement increments the g score by $\sqrt{2}$, while sideways movement costs 1 unit as usual. Allowing diagonal movement renders the Manhattan metric inadmissable, since it overestimates the cost if, for example, the goal is a direct diagonal neighbour of the start point. Therefore, we introduced a custom metric that can be thought of as a Manhattan metric that allows for 45° diagonal movement. First it calculates the diagonal distance required to align with the goal, then adds the leftover sideways distance. This is guaranteed to never overestimate the true distance, and it will not underestimate the distance unless there are obstacles blocking the direct path.

Much of the performance of A* relies on using an efficient data structure for the frontier set. We chose to use a binary heap, but since there is no corresponding STL container in C# we had to write our own. Since writing a binary heap did not seem core to the assignment, we did look for existing implemenations but found that they generally had O(n) search time, which is inefficient for updating the f and g scores of nodes in the open set. We supplemented our binary heap with a hash table for this purpose.

Our A* implementation is capable of recalculating the frog's path every time the player right-clicks without any slowdown. We configured the snakes to refresh their paths every 0.2s and found that this was easily fast enough for their decision-making to appear continuous. Paths are generally calculated in 1-2ms, so this refresh rate is well within safe limits for the number of snakes generally active in our game.

5.2 Jump Point Search (JPS)

In the plain A* algorithm, once we have visited a node we must add all of its unvisited neighbours to the frontier set. In JPS, rather than adding the direct neighbours of the current node, the algorithm tries to "jump" to more distant nodes by taking into account the current direction of travel.

Consider the sitution shown in Figure 9a. Since the current node was reached by moving right, it does not make sense to explore any of the nodes that are shaded grey.[4] The situation is more complicated when moving diagonally (see Figure 9b) since the last move may have aligned us with the goal.

If neighbouring nodes are blocked then we can no longer rule out as many possibilities. In Figure 9c, we may have wished to move diagonally up and right on the previous move but this was not possible because the node was blocked. Since we may still need to move up, node B must be considered now. In this type of situation, the additional node is called a "forced neighbour".

The algorithm continues expanding the path recursively until it reaches a dead-end or finds a "jump point successor". Jump point successors are nodes that must be visited on the optimal path to one of

[4]To eliminate multiple equivalent paths, JPS assumes that all diagonal moves are made upfront. Therefore, the fact that we did not move diagonally on the previous move means that we do not have to consider the top-right or bottom-right nodes now.

their neighbours.[5] This concept is probably easiest to understand visually (see Figure 10). When a jump point successor is found, only the successor itself is added to the frontier set. The intermediate nodes are discarded and never have their f, g or h scores calculated.

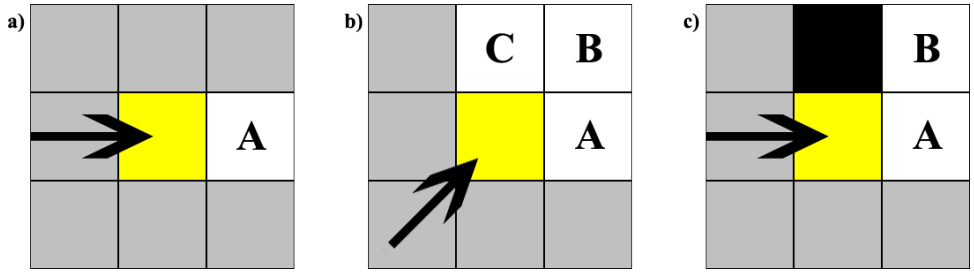


Figure 9: (a) When moving non-diagonally, the only neighbour that needs to be considered is the one in the direction of travel. (b) When moving diagonally, the previous move may have brought us in line with the goal, so two additional nodes (A and C) must be considered. (c) The direction of travel is not necessariliy the direction in which we should proceed, since one of the directions (B) was not available on the previous move.

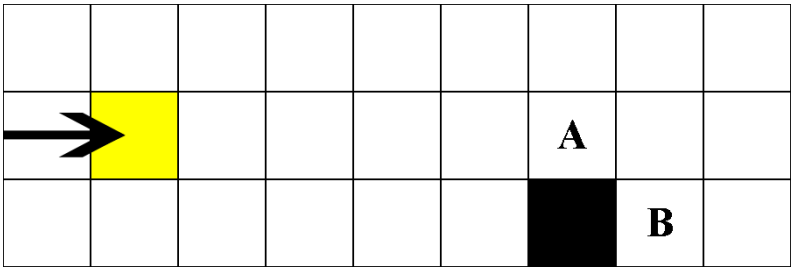


Figure 10: Node A is a jump point successor of the current (yellow) node since the optimal path from the current node to B must go via A. The intermediate nodes between the yellow square and A are skipped over and not added to the frontier.

5.3 Performance of JPS versus plain A*

We found that the performance of plain A* versus A* with JPS depends on the type of obstacles nearby, which makes sense intuitively. In open spaces, both algorithms generally find a direct path towards the goal without considering many unnecessary nodes. While JPS adds less nodes to the frontier, the extra time it spends calculating jump points largely cancels out the improvement. However, when the frog or snake is obstructed by a wall, A*'s search area tends to balloon out with many equivalent paths (see Figure 11). This is where JPS comes to the fore, because it is specifically designed to ignore equivalent paths. In these situtions, our JPS implemenation is generally around 1.5 – 2 times faster than plain A*.

"Jumping" in JPS is often implemented via recursive calls, but since this potentially uses a lot of stack space we tried writing an iterative version. Sadly, the modification made no difference at all, probably because the paths in our game are not particularly long.

5.4 Challenges

In this section we look at the challenges we encountered in regards to pathfinding.

5.4.1 Obstacle Avoidance

In a sense, obstacle avoidance is already built into A* since the entire point of the algorithm is to find an optimal path that avoids blocked nodes. However, the algorithm implicitly assumes that movement

[5] We intially found this definition somewhat confusing. If the goal can be reached by travelling in a straight line, it does not mean that every point along the path is a jump point successor. It may help to think of jump point successors as potential points of direction change on the final path.

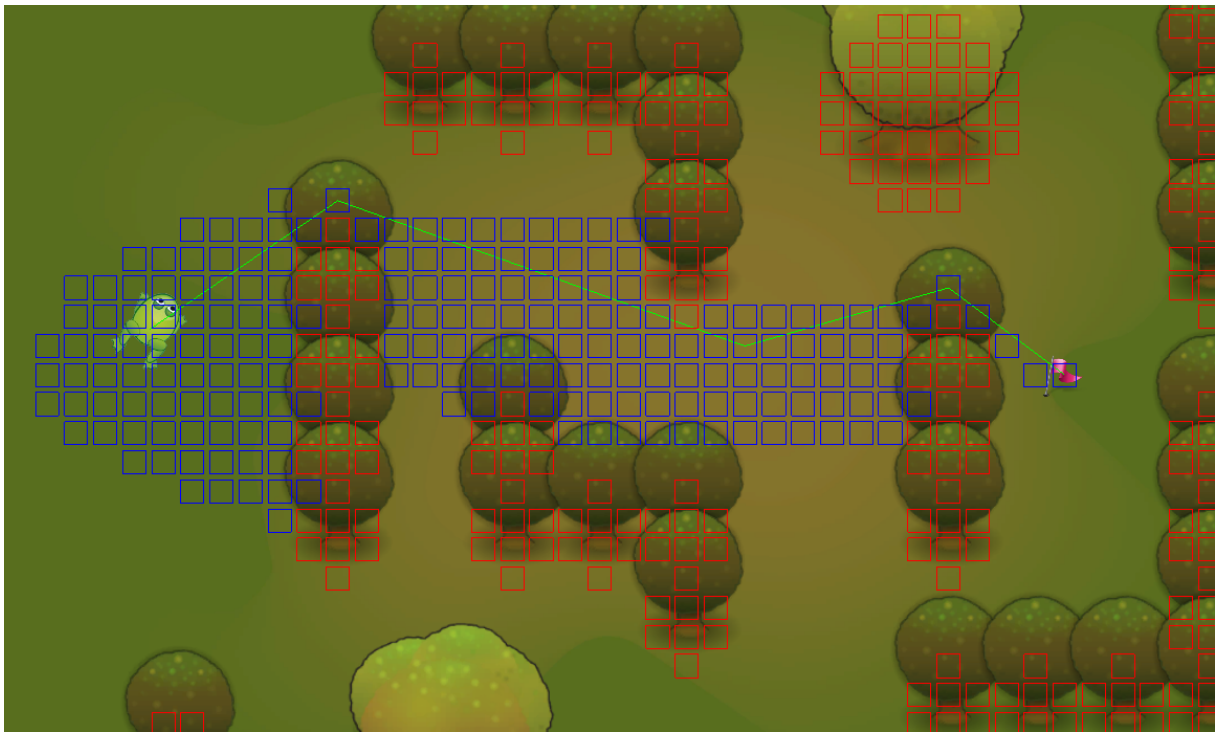


Figure 11: With plain A* pathfinding, the open areas tend to get "filled out" by many equivalent paths. The frontier set can get quite large, so using an efficient data structure to retrieve the most promising unexplored node is quite important.

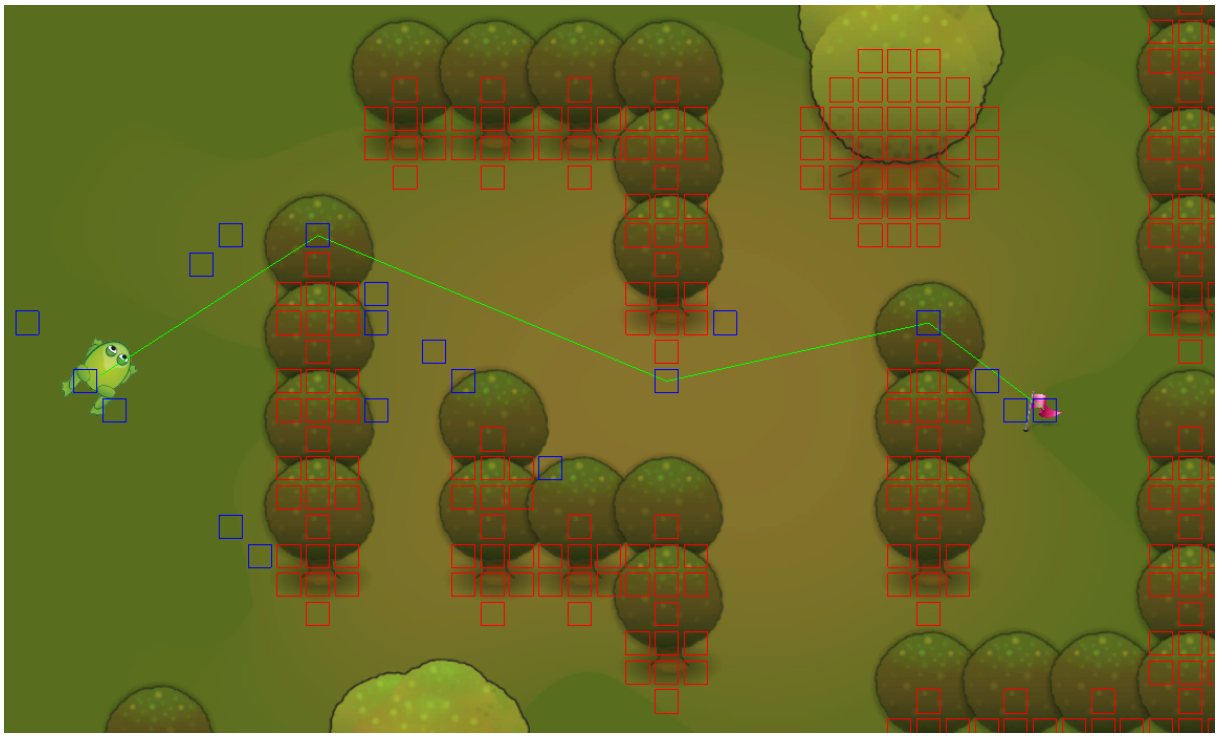


Figure 12: With JPS, only "jump point successors" are added to the frontier. The frontier set generally remains much smaller than in plain A*.

is discrete, i.e. it is not possible to be halfway between two nodes. In our game, movement is continuous and controlled via steering behaviours. When the frog or a snake moves towards a grid square, it may miss its target slightly. A natural way to handle this is to move to the next waypoint on the path once

the creature is within a certain radius of the current waypoint. However, this introduces a secondary problem: if the creature is slightly off course when it shifts to the next waypoint, it may collide with an obstacle that it did not expect to be in the way. Our solution to this problem is to simply recalculate the path whenever a collision occurs. If a creature overshoots a waypoint and gets stuck, the new path will guide it back on course. This happens rarely in our game so the additional calculation time is not an issue.

Another challenge regarding obstacle avoidance is that A* does not take the size of the moving object into account. Since generated paths often come close to obstacle corners, and since both the frog and the snakes are larger than the grid size, this issue would cause our creatures to constantly clip corners had we not accounted for it. Our solution was to pad obstacles with a "buffer zone" of blocked nodes, where the thickness of the buffer is equal to the radius of the creature. Since different creatures have different sizes, this meant that we required two grids: one for the frog and one for the snakes.

5.4.2 Path Smoothing

Since A* only considers movement in eight directions (four if diagonal movement is not allowed), the resultant paths look unnaturally jerky when followed directly. Therefore, we designed our creatures to seek smoother paths by raycasting ahead to distant waypoints. If a clear path is found then the intermediate waypoints are ignored.[6] Examples of this can be seen in Figures 11 and 12, where the green lines representing the path are not restricted to 45° angles.

6 Future Work

In this section we discuss our initial thoughts on what we could do in assignment 2 and what techniques are of interest to us.

- Evolutionary Algorithms
  A lot of work in Evolutionary Algorithms appears to focus on Procedural Content Generation(PCG) so we could try generating the map at run time, using previous statistics about the player from earlier games to make it more challenging/easier. Alternatively, we could attempt to do something more novel and evolve prey and predators to be stronger at avoiding/chasing the player. We could even evolve our own creatures based on a list of characteristics, i.e., a fast predator which has weak attack vs a slow predator with a strong attack. This would probably involve giving the frog more options to counter varying strategies.

- Frog Companion / Enemy
  Introduce another frog into the game that can learn to cooperate with the player or attempt to kill them instead. This could use some form of Reinforcement Learning with Potential-Based Reward Shaping (PBRS) in order to speed up learning in an online setting and still converge to a policy where the second frog wins the game, either with the player or without. Here, the actions the player takes can be used to influence the learnt policy via PBRS. Alternatively, we could try exploring a different learning technique since we are both already familiar with Reinforcement Learning.

- MCTS Predator
  Since we have A* and JPS implemented it could be interesting to see a predator which uses MCTS for pathfinding and compare the differences in regards to speed, memory and general performance. To the best of our knowledge, current literature only compares MCTS to A*. Additionally, we could also attempt to hybridize MCTS with A* i.e. adding in the heuristic to see how performance changes between vanilla MCTS and this hybrid.

- BDI Agent
  It would be interesting to try Agent Oriented Programming in order to learn something new and how to create intelligent agents from a different perspective. However, this is dependent on whether integrating JACK with Unity3D is straightfoward, especially given the timeframe.

[6] Again, we had to be careful to take the creature size into account, so we actually cast multiple rays from around the creature's radius.