# Project Update
# CSC 584 Building Game AI

**Debalin Das** and **Kunal Kapoor** and **Utkarsh Verma**
Department of Computer Science
North Carolina State University
{ddas4, kkapoor, uverma}@ncsu.edu

**Abstract**

Crimsonland is a 2D (top-down view) survival game where the player is trying to stay alive while various enemies converge on him from all sides of the map. The player is placed in the middle of the map at the beginning of the game and the enemies enter from around the boundary of the map. The player has to shoot the enemies to kill them and stop them from reaching him. Killing enemies also earns the player points, and the objective is to try and maximize the points earned until death.

Our objective is to create a game like Crimsonland [1], where we have tried out, and implemented novel enemy behaviors to make the survival game interesting. The variety of enemies will range from enemies which take cover while on reduced health to beings with interesting flocking techniques that try to use their numbers to corner and kill the player. The goal of the project is to create various interesting types of enemy bots demonstrating different AI techniques to defeat the player and implement at least two different approaches for each enemy type for realizing their goals. Consequently, we test these approaches quantitatively and see which ones of the two provide a more challenging gameplay experience.

## Introduction

The main purpose of our game is to make a testbed for strategic combat movement techniques used in games for enemy AI. The game starts off with the player at the center of the map. With time, enemies emerge from the boundaries of the map and come towards the player. The player needs to fend off the enemies and survive for as long as possible.

For simplicity (and also, as seen in the original game), the enemies don't have any projectile weapon capability. They can only damage the player by coming in physical contact with him. On the other hand, the player has a gun which he can use to shoot and kill the enemies. Different enemy types would inspire different strategies from the player to evade and score higher. As the enemies come in swarms and their numbers increase with time, it will become difficult for the player to keep them off himself and stay alive.

A lot of different AI techniques were developed for the project involving individual and collective movement constructs, finite state machines, decision trees, strategy techniques and predictive learning. We discuss the AI techniques in detail for each implemented enemy type in the following sections. We also discuss one enemy type which needs more work before the final submission. We are open to the idea of creating an assistant bot to help the player pick bonus items, if time permits. Some of our ideas for the assistant are detailed in the future work section.

## Background

When we are building enemy AI for games, movement techniques, decision making, path finding and collective strategies are some of the most important and significant aspects which should be taken care of. We focus mainly on these four concerns.

The original Crimsonland game had very simple movement techniques for the AI, where enemies simply seek the player at a very slow speed. The movement is not continuous, as the enemies keep stopping frequently. This gives them an appearance of slow moving creatures crawling towards the player. For our project, the attempt has been to come up with several of more interesting enemy behaviours. Our movement techniques range from simple kinematic motion to more complicated dynamic motion techniques like pursue, wander and several blended combinations of these. Some enemies sometimes directly seek the player, and some others wander around the map until player comes within a certain view radius, and then dart towards the player to attack him. Other collective intelligent methods like flocking and hive formations have also been implemented to attack the player. Swarm intelligence behaviors along with the ones mentioned before, allows a group of enemies to corner the player by surrounding him so that he has nowhere to move. Some enemies follow their leader and then go berserk once the leader dies. There are tons of fascinating ideas that can be put into a genre of game like this and we have implemented and will evaluate a handful of them.

## Problem

The enemy bots are of varying types, with each type operating on different strategies and decision making schemes. The AI logic for seeking the player, and trying to surround him collectively is tricky to get working in a natural-looking manner. Furthermore, in each of these cases handling obstacle avoidance and collision avoidance has proven to be a tough behavior to successfully implement.

We have used path-finding algorithms extensively in our project. All three of A*, Greedy and Djikstra's algorithm has been implemented and available to all our game objects as utility methods. They can simply switch from using one algorithm to the other and this will help us easily decide which one looks more realistic and suits our needs best. It is often a daunting task to figure out which search algorithm is suitable for a certain situation. Of course, in our case (and in other games as well), it heavily depends on the game graph structure that is used and other game parameters like obstacle placement, types of obstacles, etc. In most cases, we have seen that A* is a better choice and we will stick to using it wherever we need to path-find in our game.

In a game like Crimsonland, there are many enemies in the map at any instant of time. In our case, there are about 6 different enemy types and each of these types may be collective or individual. For example, for such enemy types which try to implement a formation among themselves, getting the right amount of cohesion and separation to avoid collisions among themselves but still look realistic enough has been difficult to perfect. An algorithm for the enemy type which takes cover behind obstacles has to take into account the player's and the environment's information and somehow merge them to have a believable behavior. All of this have been delved into in detail, in the later sections.

## Potential Benefits

Putting all the movement techniques, collective intelligence, pathfinding and decision making into the game, we would see how each aspect plays out with others. More importantly, we would be able to analyze how each of the enemy AI techniques (and a collection of them) would be handled by the human player. Does a culmination of too many enemy types make the game less interesting? Putting it in a different way, would it be more fun if the different types of enemies came one at a time in stages? From the current generation of games, we can draw a conclusion that the answer to the previous questions is somewhere in the middle. But then again, it depends on the player himself and what his primary approach to playing games are. Building these constructs into the game would give us a chance to study the same and comment on them.

More significantly, for each enemy type, we have a respective evaluation criteria. We have a prior-notion as to which technique would be the best for implementing a particular behaviour. We have tried to implement these behaviours using more than one technique and then compare to see which one fares better. The results help us determine best approach for the AI logic, which even other game developers might find useful!

## Tasks

This is a survival game with waves of enemies converging on the player, trying to kill him. This has given us a platform to create many different enemy bots demonstrating a combination of AI techniques. In this section we would touch upon the individual components of the game, giving a brief behavioral overview of each of them. The implementation details are described in a later section.

### Map and Obstacles

For simplicity, our map contains a constant terrain and obstacles can be generated randomly or from a fixed script. The AI bots in the game as well as the player need to avoid these obstacles while moving around the map. These obstacles have been put to some interesting use. An enemy AI will no doubt be fired upon constantly by the player and more often than not, the enemy will die before reaching the player. But we have built a certain enemy type in such a way, that when they have very low health, they will run for cover behind obstacles to avoid incoming fire. This has added a different kind of realism to the game.

The player will spawn at the center of the map. The enemies will spawn both outside and inside the visible area of the map and will come towards the player.

### Player

The player is represented as a distinct shape/sprite on the map, and is under direct control of the person playing the game. The player can move around the map in 8 directions - `UP`, `DOWN`, `LEFT`, `RIGHT`, `UP-LEFT`, `UP-RIGHT`, `DOWN-LEFT` and `DOWN-RIGHT` (using the W, S, A, D keys on a keyboard). The player will be responsible for moving smartly across the terrain, avoiding obstacles and using them as cover when necessary. The player has to avoid coming in contact with the enemy bots, which can lead to health damage. The fire direction is always towards the current position of mouse pointer.

### Bullets and Bonus Items

The player has infinite ammo, but his firing rate is limited. This firing rate can be increased by picking up bonus items appearing on the map. This will be an incentive to the player to move around the map, possibly coming into the vicinity of wandering enemies. This will also establish the base for our proposed assistant bot, which (as mentioned before) we will implement based on our overall progress and time line for the whole project. The bonus items are of different types and spawn at different rates across the map for the player to pick up. They provide different boosts to the player, and all bonus effects are ephemeral with the effect wearing off after a while. Effect duration varies by the type of bonus item. One such bonus item increases the fire rate of the player's gun. Another such bonus item changes the gun shooting pattern to a dense radial formation like a multi-turret cannon. We will add more interesting bonus items if time permits.

### Enemy Bots

There will be several kinds of enemy bots in the game, which will differ in appearance, and behaviour. However, most enemy bots have one common goal, which is to kill the player by coming in contact with him. By providing a variation of enemies, we plan to test them together in a single session of the game as well as individually. Furthermore, we will test different methods of implementing the same enemy behavior and see which looks best. We will detail this in the later part of this document, when we talk about Implementation Details and Evaluation Methods. The enemy types are listed below:

1. **Soldier**: These are the most commonly spawned enemy units. They directly seek the player, and try to kill him by contact. They move at a constant speed throughout the map. They are unaware of other enemy bots, and thus each such bot moves independently. These bots have normal life reduction rate (LRR). They are randomly spawned in any corner of the map. While moving, it tries to avoid any obstacles that might come in its way.

   The more interesting behaviour of Soldier bot happens when it starts losing health on taking damage from player's gun. When the health goes lower than a certain threshold, it will try and save itself by finding an appropriate wall/obstacle and take cover behind that. Once it has stopped behind that wall, it will start regaining health. After it has reached a certain threshold in the course of regaining health, it will start seeking to attack the player again. While recuperating, if the player comes to that cover point and shoots at it again, it will then find a "different" obstacle than the one it was hiding at right now and take cover behind that. It will continue doing this until it has regained its health till the higher threshold and then it will start seeking the player just like before.

2. **Grunt**: The Grunt aimlessly wanders around the map, serving as another bot for the player to avoid and kill. Grunts always move at a constant speed, and spawn less frequently than Soldiers. Grunt also has a lower LRR, so they are medium-level in difficulty to kill. The idea behind this bot is that in a game with all "intelligent" AI bots, we need a few that are essentially simple and, to an extent, random in order to provide a better gameplay experience. They are like a moving obstacle which the player should avoid coming in contact with. Grunt also steers clear of walls and obstacles in an elegant manner and avoids wandering too close to the walls.

   An interesting alternative behavior of the Grunt which we test is a slightly 'smarter' version of Grunt. In this mode, the Grunt keeps track of where the player is on the map, and finds an 'average' position of the player - essentially the location on map around which player has been spending a lot of time. Grunt tries to wander slowly towards this direction, while still not seeking the player directly, akin to 'artificial stupidity'. This leads to an interesting behavior wherein the Grunt moves towards the area in map where player has been staying for a long time, while still wandering randomly. Our hypothesis is that this will lead to Grunt being a larger menace to the player, and there will be larger probability of it coming in contact with the player. This gives the player incentive to kill Grunts as soon as they spawn on the map before it changes to this 'smart' behaviour.

3. **Hermit**: These bots are move powerful than the soldier bots, as they can move at a high speed and even have lower LRR, making them harder to kill. Hermits do not actively seek the player but they wander around the map in pseudo-random fashion. They can only see as far as a certain radius allows, and are harmless when the player is not within this visible range. However, when the player comes within this range, these bots get immediately alerted. Upon activation they enlarge and turn yellow - we call it 'rage' mode, and then they quickly steer towards the player at a high speed. If the player moves out of the view radius during the pursuit, they return back to their normal appearance and wander behavior.

   While testing the game during development, we observed that it's quite easy to kill Hermits when it is in rage-mode and seeking the player at a high speed. This is contrary to the behavior we actually wanted to achieve. The reason for this is that since the bot is directly moving towards the player, the player can simply keep running backwards, while continuously shooting at the incoming bot. The bot takes in damage from all the bullets and dies rather quickly. To make this more challenging for the player, we proposed a different movement pattern for Hermit bots. When seeking the player in rage-mode, they move in a zig-zag pattern towards the player. This makes it more difficult to shoot at, making the game more fun and challenging. Another proposal is that the bot will spiral down towards the player, moving in a circular motion and slowly converging on it. This might prove even more challenging to kill, as we plan to find out in our evaluation stage.

4. **Flocker**: These are (collectively) intelligent enemy bots which spawn and move around in groups. Each such group has a leader, where all other members follow the group leader. The leader alone can smell the player's position and moves towards him, while the group members keep following the leader. The leader has a pre-defined field of vision which is blocked by obstacles, and once the player is visible, it alerts the group followers, who then abandon their current flock formation to attack the player. If the player moves out of the vision of the leader, the followers reform their flock around the leader and continue to follow him as before. If the leader dies before its flock, then the followers abandon each other and focus on killing the player.

5. **Martyr**: Martyrs are bots that follow their leader with the sole purpose of keeping the leader alive. They try their best to sacrifice themselves in order to keep the leader out of harms way. The interesting part about Martyrs are that they always travel in strategic formations. The leader of the martyr group is always at the center of the formation, surrounding which there are the individual martyrs. The leader pursues the player and the martyrs take their position around the leader. The formation always starts with the leader in the center and 8 martyrs forming a box around the leader.

   There is a reason why we have chosen to call them martyrs though. This is because as the leader seeks the player, the front row of martyrs will get shot and probably get killed as a result. Whenever this happens, a martyr from the back row of the formation will come forward and take its position, thus dynamically changing the formation. Another way that this has been implemented is to dynamically change the complete formation type to adjust the loss in martyrs. These two cases will actually show different behaviors and we have planned to evaluate and see which one fares better in protecting the leader in the middle.

6. **Blender**: These bots start off with very low HP when they spawn, but they have an interesting feature. Every blender will look at a certain radius around itself and tries to spot another blender within that area. If it finds any, then both (or more) of these blenders will come together and unite to form a bigger blender having more HP and speed. This blender will be more difficult to kill than the smaller blenders. So its imperative for the player to kill any blender whenever they spawn, so that they aren't given much time to coalesce and form larger and stronger blenders. This might give an interesting choice to the player to think and decide which enemy to kill first; allowing a blender to roam around for long may be more dangerous to the player than anything else. Furthermore, the goal of a blender could be two-fold. Knowing that to kill the player the blender must be stronger, it could have a dynamic goal matching behavior to initially find other blenders and morph into a stronger being before going after the player.

## Implementation Techniques

We have six different types of enemies showcasing different behaviors and AI techniques - Soldier (seeking, cover taking strategy on low health, decision making, obstacle avoidance, path-finding/path-following, state machines), Grunt (wandering, obstacle avoidance, state machines), Hermit (wandering, obstacle avoidance, alarm strategy on player visibility, state machines), Flocker (seeking, cohesion, separation, obstacle avoidance, collective movement/flocking strategy, dashing strategy for followers on player visibility, state machines), Martyr (seeking, formation strategy, dynamic formation change on unit death, obstacle avoidance, state machines) and Blender (collective coalescing strategy on spawning, seeking, obstacle avoidance).

It is important to note that we are not only trying to showcase the different enemy types (though it is the main goal), but also trying to make a cool and fun experience for any player. It is imperative, because of this and to , that we describe our map generation, tiling methods, obstacles and player capabilities and then move on to a more detailed description of the enemies. This will also help us explain some things while describing the enemies with much more ease.

### Map and Obstacles

Map and obstacles are maintained in our game in an `Environment` and `Obstacle` class and certain generic methods are used from an Utility class when required. For simplicity, our map is a internally represented as a tiled graph with quantization and localization happening dynamically for the player, enemies and bullets. We have played around with the tile density and figured that an 100 x 80 tiling with a total of 8000 nodes (with a resolution of 1000 x 800 pixels) give us enough room to be accurate with individual sprite representations as well as keeping the performance of our search algorithms acceptable. Thus the size of each tile on screen is 10 pixels by 10 pixels, which is about the size of our players and enemies.

The obstacles on the map are represented by a range specified by the starting grid location (top-left-most grid in the obstacle) and an ending grid location (right-bottom-most grid in the obstacle). This is then internally broken up into a list of individual map grids that fall in that range. It is obvious then that each obstacle is a rectangle in our case. This happens to be enough for a start and we plan on later replacing each of these obstacles by some sprite images. Also, it must be noted that complicated obstacles can be formed just by rectangular obstacles. The list of grids for each obstacle are then put into another bigger list at the starting of the game to form the complete list of invalid nodes. This invalid nodes is represented by a HashSet data structure of PVectors in Processing. Thus we can easily look up obstacle data during the time of obstacle avoidance from this set, in an average of constant or O(1) time. This is then passed along to a method in the Environment class which builds the complete graph based on the number of tiles and the list of invalid nodes. Each node connects to its adjacent node in cost 1 and to its diagonal node as well in cost $\sqrt{2}$. The result of building this graph is to get an adjacency list and node information which is later passed on to a GraphSearch class for performing all search algorithms at a later stage.

For now, we have analyzed what obstacles might best suit our needs and scripted the obstacle specification part, which as described before, is read by the Environment class and rendered on the map. For random generation, a fixed number of rectangles of varying sizes will be spawned across the map, and rendered on screen. We haven't implemented this yet, so we will explain this part in our Future Work section in more detail.

### Player and Bullets

Before talking about the player, we should talk about the GameObject class which is a superclass for all dynamic objects in the game. This includes the player, bullets and enemies. This GameObject class again is a child class for the Kinematic class which holds the four kinematic variables - position, velocity, orientation and rotation. The GameObject class holds the display methods for all the subclasses which is nothing but rendering the shape in the specific orientation and position based on the instance variables. Thus the sub classes are only left with implementing the move method which acts as an update point for position, velocity, etc. and for chalking out other behaviors. Also, the Engine class is the one which orchestrates everything in the game and controls spawning of enemies and enemy move method calls.

The Player class does not contain much other than controlling the velocity and updating the position of the sprite (kinematically) based on key presses. It controls another important thing - the firing of bullets which affects the behavior of a lot of enemies. The Bullet class merely contains the current position of the bullet which keeps updating in the move method based on the MAX_VELOCITY of the bullet. The important part is how the list of bullets is maintained. This is present in the player. The data structure used for this is a synchronized (thread-safe) linked list. It is important for it to be thread-safe, because there would be a lot of asynchronous calls to add bullets based on mouse clicks.

Every enemy requests this list from the player and iterates over the list to see if any bullet has hit them or not. If so, the bullet is removed from the list and the life of the enemy is reduced by a certain amount. This is individually tracked by each and every enemy.

## Bonus Items

Bonus Items spawn on the map at different times across the map. The item spawning is taken care of by the Game Engine, and each type of item has it's own spawn rate. At the correct interval, a random valid position is obtained - a position which is well within the borders and does not lie on any obstacle, and the item is spawned at this place. The player can 'consume' these bonus items by walking over them. This consumption is primarily a boolean flag within the item object which turns on when player walks onto the tile occupied by this item. On consuming an item, the player object invokes the appropriate bonus item effect action.

Several types of items have been implemented. One is a gun power pack, which increases the fire rate for a short while. This is done by modifying the GUN_FIRE_INTERVAL in Player object, which affects the frequency of gun fire.

Another item is a spray-gun which temporarily changes the gun to a multi-turret cannon which shoots bullets in a radial pattern around the player. This is done by toggling a boolean variable which is checked every time in the shootBullets() method. When the boolean is true, multiple bullets are shot on each click. Say N bullets are shot in the radial pattern, orientation of each is separated by TWO_PI/N radians, so that the N bullets collectively cover the full 360 degrees. Further, each orientation is shifted by the orientation value of the player. This leads to a rotating radial pattern which is in player's control, and orients itself as per the player's current orientation.

## Enemy

All our enemies have state machines and all decision making is done based on those. Each enemy has a current state which is queried and corresponding actions are performed based on that. A decision tree could have been built for each of them as well, but we chose to stay with state machines for most of our enemies as even if we built decision trees, they won't be having much depth. So the efficiency gained from building a tree and traversing it at each stage does not effectively explain the complexity incurred for the same. Hence, as a design decision, we have chosen to implement state machines instead in most cases.

In the following sections we will dive into the technical details of how we have implemented the behaviors for each enemy type.

1. **Soldier**

   A Soldier has 4 main states:

   (a) SEEK
   (b) PATH_FIND_COVER
   (c) PATH_FOLLOW_COVER
   (d) REGAIN_HEALTH

   The move method (update loop) queries the current state and matches it with each of these states to decide on an action. Initially the soldier is in the SEEK state. In this state it first figures out the current position of the player and gets a seek steering output from our movement classes. We have implemented a horde of movement algorithms like seek, align, cohesion, face, separation and wander. All of these are separately placed in classes having their own data structures which any enemy can use depending on their use case and get a proper steering/kinematic output which is then used to update the kinematic variables for a particular enemy. The soldier uses the seek steering output to update its velocity and position accordingly. Consequently, it has gets its orientation based on a LookWhereYoureGoing class and orients itself using this output. This effectively creates a behavior where it looks like soldier is charging towards the player with its MAX_VELOCITY.

   As the player will be attacking the soldier, its health will keep reducing. When it reaches below a certain threshold, the state will be updated to PATH_FIND_COVER. At this state, it will try to find a cover point. This is akin to the behavior expected in real life battle situation where a soldier tries to take cover when health drops very low. The difficult part in this phase is to find a proper cover point. All our obstacles are generic rectangles, so there is no specific cover points built into the game. Thus, the soldier has to intelligently decide which obstacle and which position near that obstacle is a good place for it to hide. We have decided to test two different approaches in this case and plan to quantitatively evaluate which one is better. The first approach is the Nearest Obstacle Approach. The soldier queries the Environment class and finds out the obstacle nearest to its own position. For this each individual Obstacle instance (while initializing their own information at the beginning of the game) stores the center point within its own grid range, localized to map co-ordinates. Environment uses this information to calculate distances from the soldier to each obstacle and returns the obstacle which is nearest to it. After the nearest obstacle is found, there can be (simply put) four different places around this obstacle that the soldier can go and hide. These places are positions represented by the centers of each edge of the obstacle and offset by a proper distance. The soldier finds the distance of the player from the centers of each edge. The one with the lowest distance will be the one which probably the player facing towards and has the highest chance of shooting to. The soldier will choose the edge opposite to that edge to hide. In this way, it can be sure that it won't be in the direct line of fire. It will immediately path find to this position (after quantizing it to a grid co-ordinate) and path follow to the same. As this path following velocity is much higher that its normal velocity, it gives the feel that the soldier is fleeing the player and running away from it. Our other approach for solving this cover finding problem is to find the obstacle farthest from the player, instead of the one nearest to the soldier. Both of them can work good in certain situations. We will talk about this more in the Evaluation section.

Once the cover position is decided, the state of the soldier will change to `PATH_FOLLOW_COVER` and the player will follow the path till the cover position in this state. Our experiments tell us that kinematic path following works best in our case and hence we will not be using any steering output here. Once it has reached the cover point, the state will change to `REGAIN_HEALTH` and the soldier will begin healing. This healing process is nothing but a gradual increase in its health based on a pre-defined constant rate. In this state, if even one bullet hits it, it will again change the state to `PATH_FIND_COVER`, but with one catch. It will find another nearest obstacle which is NOT this current obstacle that it is hiding behind! This gives the illusion that its top priority is to regain health and that it becomes very weak once its health is low. So the player has to actually run around and keep shooting at a soldier to finally kill it, as it keeps changing its cover position.

After it has regained its health up till a certain threshold, it will change its state back to `SEEK` and continue the above process as long as it is alive. It is also important to note that during all this time, it will keep avoiding obstacles. We have two different techniques for avoiding obstacles which we will document at a later section, specifically for obstacle avoidance.

2. **Grunt**

Grunt immediately starts orientation-matching based wander across the map after spawning. After every specific interval, a new random orientation is calculated, and then Grunt rotates to this new orientation slowly. It also keeps moving simultaneously, and the velocity heading is always in the same direction as the current orientation of Grunt. Grunt also uses obstacle and border avoidance. For this, future position rays are shot in the direction of travel, and validated to see if the future positions are allowed. Several rays of varying lengths are used for better avoidance in cases where the bot is near the corner of an obstacle. Our experiments tell us that four different multiples of its own velocity - 1.5, 3.5, 15 and 50 allow the grunt to form future rays and predict obstacles with almost nil failure rate. This check fails when any of the future positions lies on an obstacle or outside the map boundaries. If the check passes, then Grunt continues wandering as before. Else, it invokes the avoid obstacle behaviour. This provides a new orientation to Grunt, which is guaranteed to move it away from the problematic obstacle/border. This guarantee is in turn made in a similar way, i.e. checking for future ray in the calculated new orientation and confirming if this would be a valid direction of travel. This check would fail for orientations which continue to lead the bot towards the wall. This is essentially implemented as an infinite loop which makes sure the grunt has a valid orientation to travel to in consecutive frames.

For implementing the 'artificial stupidity' behaviour we need to keep track of several things. This involves a change of behavior states for use in State Machine. We have 2 states for Grunt:

(a) `TRACKING_WANDER`
(b) `DIRECTED_WANDER`

When Grunt is in the first state, it wanders around the map randomly, and keeps polling the player position every few frames. This position is pushed to a PVector `AVG_PLR_POS`, which stores the average position calculated till now. Each time a new position is polled, the average vector value is updated accordingly. Once a fixed count of polling has been done, the Grunt knows that the player has been moving around this calculated average position. The average position is quantized to derive a grid position - this is required to ensure a valid position is found even if the average position lies on an obstacle.

The Grunt now shifts to the second state (`DIRECTED_WANDER`), and calculates a random orientation within a particular range directed towards this `AVG_PLR_POS`. The angular range is within the 2 tangents drawn to an imaginary circle of a fixed radius centered at `AVG_PLR_POS` [see figure 1]. Grunt moves in the direction of this new orientation, till it arrives within the said imaginary circle - essentially in the vicinity of the average calculated position.

On arrival, it switches back to state 1, and starts wandering randomly. It also starts the average player position calculation again, and the whole cycle is repeated indefinitely.

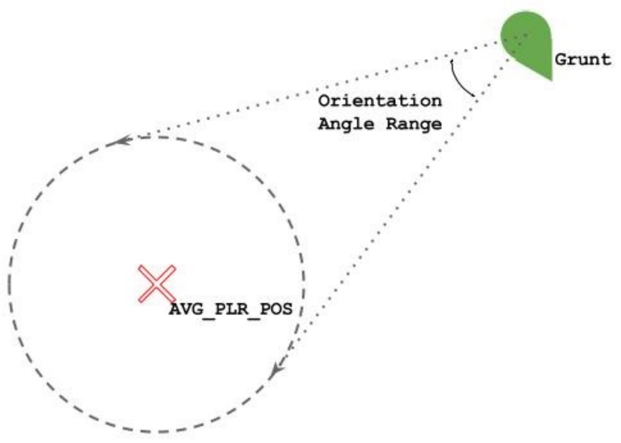

Figure 1: Orientation range for Grunt in 2nd state

3. **Hermit**

As with the tracking mode of Grunt, Hermit wanders on the map initially. Like Grunt, it changes its orientation in intervals, but shoots four rays of different lengths to make sure that it does not run into a wall or an obstacle. We do not use whisker rays, as currently, the scripted obstacles do not need this complication for obstacle avoidance. Although when using random generation of map in future, this might be something that we will need to take care of and we plan on testing and consequently adding them if need be. The Hermit's wander method is not as smart as Grunt's and is completely random.

There are two top level states that the Hermit has:

(a) `WANDER`
(b) `RAGE_MODE`

Coming back to the `RAGE_MODE`, this state will make the hermit run towards the player at a high speed to try and kill it. Having a simple direct seek behavior towards the player makes the bot easy to kill. We have two competing ideas - the first one is where the hermit will spiral towards the player, thus making it difficult to shoot at. This is possible when we enter the radius of deceleration at an angle and the decelerating force acts in such a way that it is not enough to direct the bot towards the destination instantaneously; instead there always remains a tangential velocity component. For our use case, we have determined an initial velocity towards the tangent of the virtual circle whose radius is the same as the `RAGE_MODE` radius. Accordingly we chose an acceleration which will pull the hermit towards player but in a spiral fashion, essentially delivering a weakly decelerated seek behavior.

In another approach, we have implemented a zig-zag movement behavior for the hermit when seeking the player in rage mode. This was done using a weighted blending approach. The bot needs to take care of 2 concerns - one is to keep seeking the player, another is to move sideways. Both are independent steering behaviours and are combined to give us the desired resultant motion. We evaluate both these movement patterns to see which one if more effective as an enemy bot in the game.

4. **Flocker**

The Flocker actually consists of two different types of enemies that flock together namely the leader and it's multiple followers.

The leader has the following states:

(a) `SEEK_PLAYER`
(b) `KILL_PLAYER`
(c) `LEADER_DEAD_KILL_PLAYER`

The followers use the below states:

(a) `STAY_WITH_LEADER`
(b) `KILL_PLAYER`

Before we talk about the implementation of the leader and followers, it is important to stress on how the interaction between them actually occur. According to the game engine, the entire flock (leader and followers) is considered to be a single enemy type. Hence, the engine just interacts with the leader to inform it to move or draw itself on the screen. The leader is responsible for interacting with its followers and ensuring they do what he commands.

As with the implementation of the soldier, the move method (update loop) is responsible for querying the current state and matching it with each of the above states to decide on an action. The leader starts off in the `SEEK_PLAYER` state where it finds the position of the player and uses a `Seek` steering behavior to move towards him. The orientation of the leader and followers is set by the `LookWhereYoureGoing` orientation matching behavior which attempts to orient the character in the direction of its movement. In order to ensure fairness in the gameplay, the leader is relatively slower than most other enemies, which means that it's maximum speed and acceleration are low. It, upon "seeing" the player, goes into an alert mode which is depicted by the `KILL_PLAYER` state. The actions of the leader remain the same and it keeps seeking the player. The change in behaviors occurs in the followers which will be discussed belowre mml. There is one special state called `LEADER_DEAD_KILL_PLAYER` which indicates that the leader has been killed by the player but there are still some followers left alive. This state is necessary because, as we mentioned earlier, the engine only interacts with the leader to move and it is the leader's job to interact with it's followers. The leader class must, therefore, keep track of it's death and ensure that the followers continue to do their jobs until the entire flock is dead.

The vision of the leader is programmed quite realistically unlike the `Hermit`. The Flocker leader is only alerted when the player is in front of it and the view is not blocked by an obstacle. Since, the leader always seeks the player, our first criteria for vision is that the player should be within a certain distance of the leader. If this criteria holds true, we then check if there is an obstacle blocking the vision of the leader. In other words, the leader can always smell the player which explains why it seeks him. Only when it sees the player in direct vision does it alert its followers to attack the player.

The followers, unlike the leader, are fast, agile units that are commanded by the leader to initially flock around the leader denoted by the `STAY_WITH_LEADER` state and follow it's movement. This flocking technique uses a weighted blend of several different steering algorithms namely, `Seek` towards the position of the leader, `Separation` of each follower from its closest neighbors, `Alignment` of the flock in the direction of the average orientation of its close neighbors, and `Cohesion` towards the center of mass of the flock. The above list is in decreasing order of their weights with the `Seek` behavior having maximum weight. Seeking the leader is the obvious choice for the maximum weight as the highest priority of the followers is to be around the leader. Separation from its close neighbors has more weight than the rest as it works on a very small radius and represents the importance to avoid colliding. Cohesion is the least important as some part of the cohesion happens automatically when we seek the leader. Keeping this behavior with a small weight, however, improves the closeness of the flock and gives a better flocking output. These steering behaviors are all required as they each represent a specific purpose to the flock and they seemed sufficient to generate a realistic flocking functionality. Any additional behaviors in the blend seemed to emit an undesirable or less believable output. When the leader goes into the `KILL_PLAYER` state, it consequently commands its followers also to enter the same alert state. The followers, upon hearing the kill command, turn their attention towards the player. We have thought of two different approaches to the alert behavior of the followers. The first one is that they `Seek` the player's current position and move towards it with their increased speed and numbers. The second idea is to have the followers `Pursue` the player instead. This would make it harder for the player to avoid the followers and increase the difficulty. We will elaborate on this in the Evaluation section of the document. If the player moves out of the view of the leader, then the leader commands its followers to go back to the `STAY_WITH_LEADER` state and regroup around the leader. If the leader dies before its flock, then it gives a final command to the followers to kill the player. Regardless of whether the player is in their vision or not, they seek/pursue the player quickly and try to kill him.

It is important to note that during the above process, both the leader and the followers will avoid collision with obstacles. The leader uses the same technique as the `Soldier` to avoid collisions whereas the followers integrate obstacle avoidance in their blended steering behavior to save computation time.

5. **Martyr**

A martyr group has the same kind of structuring like a Flocker group. There is a martyr leader and the rest are martyr followers. The martyr leader is the one which is started by the `Engine` and it in turn spawns its children. In our martyr construct, every martyr follower has a specific rank among its members. When the leader spawns its children, it provides them with their rank. Everybody gets a rank from 0-7 (the formation always starts with 8 players). Depending upon their ranks, each will take a target position around the leader. Thus a follower with rank 0 will get a position equal to:

$$PVector.fromAngle(leaderOrientation - PI/4)*$$
$$SEPARATION\_OFFSET$$

and so on. At this point, it is best to mention the available states for the leader and the followers. The leader has the following states:

(a) `WAIT_FOR_FORMATION`
(b) `PURSUE_PLAYER`
(c) `ALERT_MARTYRS`

The followers have the following states:

(a) `UPDATE_FORMATION`
(b) `FORMATION_READY`
(c) `FOLLOW_FORMATION`

The leader as mentioned before, gives the ranks to every one of its children and sets their state to `UPDATE_FORMATION` and its own state to `WAIT_FOR_FORMATION`. In this state it does not update its position with any velocity and simply waits for each follower to complete their formations. The followers in their current state updates their target positions according to their ranks as shown above in the formula and seeks that position. On reaching that position it updates its state to `FORMATION_READY`. Meanwhile the leader has been polling the states of each of the followers. As soon as it sees that everybody has their formation ready, it updates their state to `FOLLOW_FORMATION` and its own state to `PURSUE_PLAYER`. It then pursues the player by extrapolating the velocity (within a limit till some frames) of the player and adding it to its position.

The difference in the two approaches that we have taken to implement the Martyr is applicable here. Let's first take the case where the martyrs always keep a fixed formation and any martyr followers killed in the front are replaced by followers in the back. In this scenario, any change will only happen if the players in the front row, i.e. ranks 0-2 have been killed. Then the leader puts its own state to `ALERT_MARTYRS` and alerts each martyr by calling a function and updating their ranks and changing their state to `UPDATE_FORMATION`. It is important to note how it chooses their new ranks. Basically it takes a player from the ranks 5-7 and puts them in front with the rank that just became empty due to the death of one of the martyrs. If there are no 5-7 ranked martyrs any more, it will try 3-4. If there are no enemies with those ranks as well, it does not change the follower's states and keeps on pursuing the player. But if it finds a martyr to bring to front, it changes its own state to `WAIT_FOR_FORMATION` again and waits for martyrs to get ready with their new formation. This continues till no martyrs are left and then the leader pursues the player on its own.

In another approach, after the martyr leader provides the ranks to its children, the children don't take places in the fixed formation but rather completely change the formation type depending upon how many children are left. To achieve this, when the leader alerts the children to update their formations, it also sends them the number of children that are left. Getting the number of martyrs left on the map and its own rank, an individual martyr calculates the position around the leader with the following formula:

$$PVector.fromAngle(leaderOrientation+$$
$$((rank/numMartyrsLeft) * 2 * PI)*$$
$$SEPARATION\_OFFSET$$

This creates a behavior where depending upon the total number of martyrs left, each individual martyr updates their formation to look like a square, or a hexagon, or a triangle and so on.

Both of these techniques create the same behavior of protecting the leader but look different and actually have different capabilities in protecting the player. We will take this up in the Evaluation section to describe how do we plan to test this.

## Future work

### Blender

We still need to work on our Blender type enemy bot but we have chalked out specific approach which we will be using to implement it. This bot uses dynamic goal matching to decide what action to do at any particular time. This is implemented effectively using a decision tree which helps the bot consider several criteria quickly to arrive upon a particular action [see figure 2]. If the blender 'sees' other Blender bots within a certain radius, it will seek that bot, and upon meeting it merge with it. Both bots merge together to create a slightly larger Blender bot, which will also have lesser LRR (hence, harder to kill) and a slightly higher speed. If the bot is large enough to exceed a certain threshold, it doesn't care about looking for other bots within this view radius, it is 'confident' of taking on the player and seeks to attack him directly. When no other blenders are found nearby, the bot just seeks the player moving at a slow speed. When the health of the bot decreases below a certain threshold, it will stop seeking the player, and seek for other Blenders in the map. The view radius isn't used here, it just seeks other blenders on the map regardless of their distance from it. This is basically the bot trying to become stronger by merging with other blenders. Once a certain threshold health point has been achieved after merging, the bot starts seeking the player again. An alternate behavior for Blenders is where they don't have the confidence check in the decision tree, so they are always open to merging with other nearby blenders. This may or may not produce a more challenging behavior. We shall observe this during our evaluation.

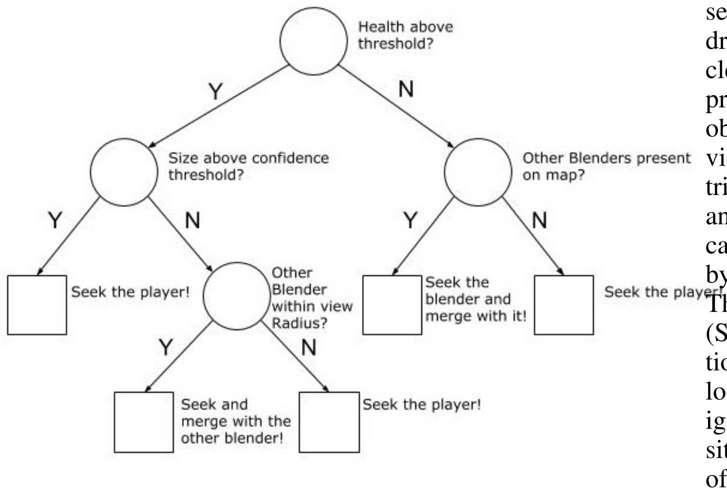
Health above threshold?
Y
N
Size above confidence threshold?
Other Blenders present on map?
Y
N
Y
N
Seek the player!
Other Blender within view Radius?
Seek the blender and merge with it!
Seek the player!
Y
N
Seek and merge with the other blender!
Seek the player!

Figure 2: Decision Tree for Blender

### Random Map Generation

As we had mentioned before, we had scripted a good enough map structure with a reasonable amount of obstacles for now. And for our evaluation purposes, this might be enough, but to really test the extremes of our algorithm, it would be great to see how our different enemy bots tackle random maps. In our case, the obstacles are all rectangular in nature. This actually makes it simpler to randomly generate maps. By specifying a certain range of width and height of any obstacle, the `Environment` class can randomly generate an obstacle and form the invalid nodes (as mentioned before) in a loop. Each time it finds a new obstacle it checks if any of the blocks within the range of the new obstacle corresponds to any of the invalid nodes formed before. If so, it will try and find a new obstacle. This can continue till the number of obstacles that we want are generated on the map.

The importance of random map generation can prove to very important in our evaluation methods as well as testing and debugging our enemy AI behaviors. And as this would be simple enough, we plan on completing this and then performing all experiments and evaluations.

### Assistant Bot

The assistant bot is an AI helper for the player, who can pick and fetch bonus items from the map to the player. The bot will take the shortest path from its starting position to the bonus items spread across the map, collect them and then return to the player. Due to the amount of existing workload in the project, we have sidelined the assistant bot (after discussing with the professor) to being a secondary objective in our project, but if time permits before our deadline, we want to come back to implementing this.

Here we discuss some of the ideas that we had for implementing the bot. The bot is initially unaware of the obstacles on the map. From its perspective, it only has a partially observable environment where it can only see the bonus item drops. Its main goal is to initially seek and pick up the obstacles and then seek and come back to the player. But in this process, the environment provides the bot information about obstacle placement when any such obstacle comes within view radius of the bot. More specifically, the environment triggers an event in the bot notifying him that he has found an obstacle grid in his view radius and provide him the location information about the same. This information is used by the bot to gradually build the map of the environment. This is akin to the simultaneous mapping and localization (SLAM) technique. When the bot has enough map information at his disposal (which basically means enough obstacle location information), he will then start path-finding to navigate through the map and pick up the bonus items. In that situation, he will also prioritize between the different kinds of bonus items according to their power and location (which is nearest to both the bot and the player), thus giving rise to decision trees, and then path-find accordingly.

The successful implementation of an assistant bot will look fantastic and cool in our rendition of Crimsonland and it might be a wonderful and contrasting addition to the existing types of enemies. We hope that we are able to implement the assistant bot as we have envisioned it here.

## Evaluation Methods

Here we talk about the setup of specific test environments to evaluate performance of different enemy types and report them. Each enemy type has at least two different approaches to achieving our expected behavior. Here we describe a quantitative approach to evaluate each type and see which is more efficient and/or looks better. Each enemy type requires a specific test bed setup, which we decide empirically. This is to maintain a consistent experiment environment, unaffected by the seeming randomness the actual game has due to various spawn rates for each enemy. Results will be averaged over 10 experiments for each type. We can be certain that the actual game performance will be a close approximation of the experimental results, since the game just adds random noise in the form of other enemies. Results will be reported in the final submission, as we are still in the process of completing the last enemy type, after which we will to begin evaluation.

1. **Soldier**

   For a soldier, we have implemented two different types of cover taking methods/heuristics - one is seeking the closest obstacle, other is seeking the farthest obstacle and hiding behind it. To test this, we plan to keep 1 soldier, 3 grunts and 3 hermits in the map at a time and evaluate the amount of time the soldier spends on the map multiplied by the absolute amount of health points regained by the soldier. The latter component effectively gives a measure of how long the soldier was able to hide. It is important to note that even if the player comes and shoots the soldier when it is hiding at the farthest obstacle, the soldier will immediately again find the farthest obstacle from the player at that point of time and consequently, make it very difficult for the player to kill it. Our hypothesis, thus, is that the farthest obstacle method will be a clear winner in our experiments.

   If the soldier moves towards a closer obstacle (to itself) to take cover, it can be sure that it will go out of the firing range as soon as possible. But the player can be proactive and come to that obstacle and shoot the soldier again. This is assuming that most generally the player will be shooting the enemies near him and hence an obstacle which is closer to those enemies will be closer to the player as well. In contrast to this, if the soldier hides behind the farthest obstacle, the player might get a better chance to shoot at it more while its going towards the farthest obstacle, but once it has reached there, the player might be busy with other enemies and may not immediately go to the farthest obstacle from itself to kill that soldier. Thus, the soldier will get a better chance to regain a lot of its health.

2. **Grunt**

   We test the Grunt's 'smart' wander behavior in a map with 5 Soldier units, and compare it with the 'non-smart' Grunt in the same situation. The soldiers act as random noise to keep the player busy. Evaluation is done based on 2 metrics - the amount of time the Grunt is able to survive on map after being spawned, and the amount of damage it deals to the player by coming in his contact. Our hypothesis is that the 'smart' grunt would have a slight advantage over the other Grunt quantitatively.

   To reason about this hypothesis, we would say that sometimes randomness can give a much easier and better solution on an average (for example, take the random cache replacement policy in Operating Systems - it consistently outperforms some other systematic replacement policies in practical situations). Also, it is important to note that once the grunt starts going towards the vicinity of the player, the player might have moved to some other place by then.

3. **Hermit**

   We test the two different modes of Hermit's rage mode movement - one is zig-zag motion, one is spiral convergence. The experiment is done in a map with just 1 Hermit, player's focus is solely on killing this Hermit as quickly as possible. The evaluation metric is the amount of health damage taken by the bot divided by amount of time spent in rage mode. This gives us a rate of taking health damage while in rage mode. This should be as low as possible, and we compare the average of these values for the 2 different modes over a set of 10 such controlled experiments.

4. **Flocker**

   The Flocker has two variations that can be evaluated. When the player is within the field of vision of the leader, the followers are given the kill command. They can then either `Seek` or `Pursue` the player. We wish to evaluate the difficulty of the game with both these methods. As we know, seek would move towards the current position of the player. This would be predictable and easy to avoid. The pursue behavior, however, would take into account the player's movement and make it harder to dodge. Given the fast speed of the followers, the pursue behavior might make it impossible for the player to survive at close range. We can evaluate this by seeing how long a player can actually survive with a large number of followers for both methods.

5. **Martyr**

   In Martyr's case, we have two approaches of maintaining the formation - one where there is always a fixed formation and any enemy loss is adjusted by pulling in martyr followers from the back row, and another where the formation dynamically changes depending on the number of followers left in the formation. As the sole cause of the martyrs is to protect the leader, we would like to propose the following metric for this enemy type - (amount of hit point damage that the martyr leader has taken) / (amount of time that the martyr group has survived on the map). This metric should be as low as possible. We will run this test along with 3 soldiers and one martyr group.

   Though the latter approach is elegant in design, our hypothesis is that the former will win, when it comes to evaluating their efficiency. This is mainly because as there is only one player, we can be sure that the front row martyrs are the only ones which will be in the firing line. Likewise, if the martyr followers spread out on number reduction, there will be a higher probability of the player finding a gap between them to shoot at the leader.

6. **Blender**

   For Blenders we shall have environment setup with 4 Soldiers, 2 Grunts and 4 Blenders. The two approaches for Blender are - one with a confidence check in the decision tree and one without. The metrics here will be the average amount of time each Blender is able to survive on the map, and the amount of health damage caused to the player. This will give us an idea of how effective the merged or independent Blenders are in their ability to harass the player, and how long they can endure the bullets. Either may turn out to be stronger, as we shall see in the final experiments.

## Conclusion

The multi-enemy combat survival game is a very good platform to showcase several AI techniques, while also being simple and fun to play. That's why we had chosen this project and we have had a very rewarding experience so far, as we implemented the different AI strategies for the enemy bots and still continue to perfect them. Through our evaluation methods, we have not only learnt which methods are more effective to use in different enemy situations but also have hand-tuned lots of parameters to perfect the feel of the game as enjoyable yet challenging, which we feel has given us a valuable experience.

We have a screenshot of our game below (see figure 3). The green object is the player shooting (red) bullets. The cyan colored ones are soldiers; one of them can be seen to have lost a lot of health (can be seen from the health bar above the head) and hence hiding behind the nearest obstacle. The pink object is the leader of a flocker and the small ones following it are its followers. The cream colored object is the hermit, wandering around the map. The small white dots around the map are the bonus item drops and the violet colored objects are obstacles.

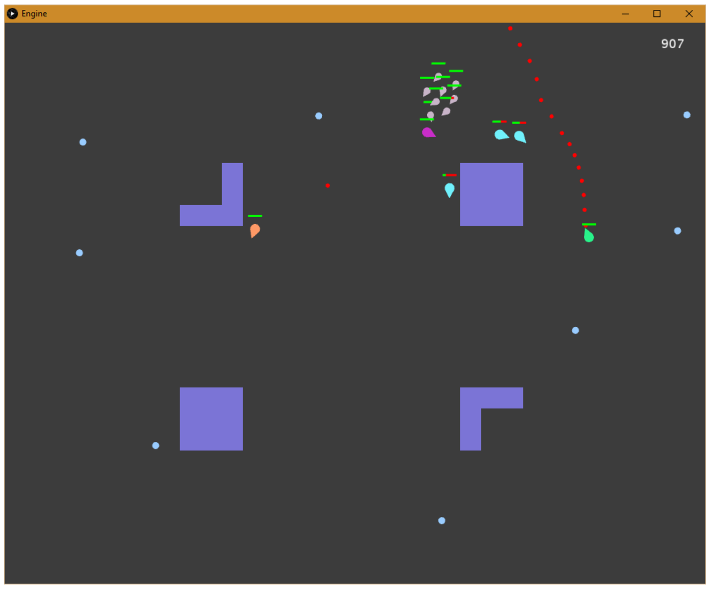

Figure 3: Screenshot from the game

## References

[1] *Crimsonland - the Game*. URL: `http://www.crimsonland.com/`.