Augmented Reality Tower Defense

By Roman Ganchin, Samuel Orloff, Tony Trakadas

Video: https://www.youtube.com/watch?v=Jup8oPqyc2s&feature=youtu.be&t=8s

Just in case you want to watch the demo before you read the whole document

Description of the Project Idea

We will implement a basic tower defense game. The setup will have a Kinect sensor and projector positioned over a flat surface. We will place objects on the plane within the field of view of the Kinect. These objects will be observed by the Kinect, and their size, color, and height will determine what they represent in the game. The game will have walls, towers, a start, and an end location. Once we are finished placing the objects, creeps (enemies) will run from the start state to the end state, not being able to move through the walls or towers. The optimal path will be calculated for the creeps before they traverse through the horrors that await them. Using this information, the projector will then present the tower defense game in action. The creeps will be projected onto the flat surface, so the game can be watched. The towers will fire projectiles, also represented by projected images, killing creeps. The game is won if the creeps are all annihilated before they reach the end; the game is lost if a creep reaches the end.

Potential Approaches to Solving the Problem

Game Map: The first step of our project was dubbed Game Map. In this step, we would obtain information about the physical game space from the Kinect, parse the information to locate objects, and then store these objects into data structures that could be read by the tower defense program.

We considered two ways to obtain the Kinect's data: via a colorized Pointcloud2 or through a combination of depth and color imagery. As for interpreting the data, height and color served to be the primary way of grouping the points obtain from whichever Kinect data stream chosen. Several methods could be used in order to then interpret these points into individual objects. Plane filtering could be used to identify the tops of the walls, towers, and start and goal points. Information such as location, height, color could then be derived. The information of start and goal could be represented with a point, while walls and towers would be stored in two separate arrays which held their heights, locations, and other necessary information. Another potential solution to this problem would be to district the information into a two-dimensional grid. The whole space would be divided up into squares. Then based on the tallest point in each square, that square would be assigned a type: ground, wall, tower, start, or goal. One two-dimensional vector array could be used. It would be an array of element structures, which held identifying information (type of object) as well as the other attributes needed.

Path Planning: In tower defense games, the creeps navigate around walls and obstacle in order to move from a start point to a goal. As such, Path Planning is the function which finds the path for the creeps in the game to follow. With the start and goal identified in Game Map, the algorithm would chart a path such that the creeps would not collide into the previously found walls and towers.

To do path planning, the game space would have to be searched. Both search and rapidly-exploring random tree (RRT) algorithms were considered. If we chose to implement Game Map using the two-dimensional game array, then we could use a depth search or A* search to find a path. Each index would represent a node in the search graph. Since our district view of the game space would be relatively small, these search algorithms would be able to quickly move through the space to find a solution. RRT could be used regardless of the Game Map data type. The algorithm could be given a filtered point cloud, removing the ground plane and leaving just the obstacle points. Since creeps cannot move through walls and towers, no distinction would be needed between the two object types.

Creep AI: Creep AI moves the creeps from the start to the goal, following the path is given by Path Planning. It also must handle the health of the creeps, adding the damage dealt by the towers.

Depending on the nature of Find Path's implementation, following the path changes for Creep AI: If the Game Map data type is used, and search is implemented, then the Creep Ai will be given an array of districts to move through. Each district would be mapped to an area of the game space, so the creeps would move to the center coordinate of next square in the array. If RRT is used, then the array will be all the points in a path from the start to the goal. The creeps would just need to move from one point to another in the path. The managing of health would be simple. Each creep has its own health stat stored, and when a tower targets a creep, dealing damage, this damage value is subtracted from the creep's health. Once this reaches zero, the creep is dead and is removed from the game.

We considered implementing Creep Ai in a separate node, written in Python; one member of the group was unfamiliar with C++. So there were two approaches to implementing creep ai. Creep Ai could be called accessed as ROS services written in Python or the functionality could be written in C++ and called as functions. We also considered these options for Path Planning.

Tower AI: The towers must be able to attack and kill creeps. When an enemy moves close to a tower, the tower must begin firing upon it. Given a horde of creeps, the tower must determine which one to attack, as it only can hit one at a time.

The Tower Ai could iterate through an array of towers, deciding what each tower would do. Each tower has a range and a set damage amount. Whenever a creep enters within the range of the tower, it could be subject to attack. An attack would deduct from the creep health by the damage stat of the tower. The targeting behavior of the tower could be implemented in different ways. A tower could attack whatever is the closest creep on a given frame; it could attack the creep with the lowest health in its range, or it could attack a random creep within its range.

Visualization: A projector would be used to project an image over the physical game space. When the game is run, the projector will illustrate what is occurring in the Tower defense game.

The considered means of visualizing was RVIZ. We could publish markers to represent the game space, based on results of Game Map. The visual frame of RVIZ would be set to give a

top down view of the markers. We would then project the RVIZ window over the game space that the Kinect is observing, lining up the marker with the in-game objects

The Methods Selected

Game Map: We use the openni package to obtain information from the Kinect. A subscriber to the PointCloud2 publisher of the Kinect captured a colorize PointCloud2. To work with this data structure, we also used the Point Cloud Library (PCL). Thus, the PointCloud2 could be converted to the more manageable PointCloud<PointXYZRGB> data structure.

Next, the program needed to locate the distance from the Kinect to the base plane, which would allow us to filter out the ground and find the relative heights of the other objects. Rather than plane filtering, the program found the mode of z values for all the points in the cloud, using the function, maxOccuringValue. The base plane comprises the majority of game space area, so the mode of the z values is a good indication of the depth of the base plane.

With the base plane's depth found, the program filters the point cloud into two separate clouds. A cloud with all the obstacles filtered out is sent to Path Planning, and a cloud with the base plane filtered out is used throughout the rest of game map. To do this filtering the program uses the mode z value; any point within a small distance, to account for error, will be considered part of the base plane. Next, the obstacle points are filtered into three arrays: start, goal, and towers. Walls do not actually need to be identified because RRT is being used to do path planning and walls serve no other purpose than to be an obstacle. The start array contains all the blue points (RGB values are with a range that produces a bluish color) and with a height (relative to the base plane) under the start/goal cutoff. The goal array contains all the red points with a height under the start/goal cutoff. The tower array contains all points with a height greater than the wall cutoff. Note all these cutoff are just the height values (magic numbers) set for our game implementation. With these arrays, the start, goal, and towers will be found. The start point is the average x,y,z from all the points in the start array. The goal point is an average x,y,z as well. To locate the towers, the towers array is passed to another function, towerfind.

Towerfind searches the towers array to find the densest groupings of tower points. A tower is assumed to be tall and narrow. Therefore the tower points are located in clusters. Towerfind will iterate through each point in the towers array, calculating how many points fall within a certain radius from that point. The point with the most points within the radius is considered the center point of the tower of best fit. This center and all the points around it are grouped as one tower and flagged as already part of a tower. The program loops again on the remaining unflagged points, searching for the second best grouping of points. Each grouping it finds that has a significant number of points, a tower must be made up of more than 100 points, is considered a tower. When all the points become flagged, or when there are no more groupings with enough points, the algorithm terminates. Each tower found is put into an array of towers, different from the towers array, which contained all the potential tower points.

Path Planning: The RRT algorithm used differs from the one implemented in class. Instead of checking for collisions with obstacles when the algorithm does the random sampling, it checks if there are points in the area. The program passes Path Planning a filtered point cloud that includes only the points from the base plane, points that a creep can traverse, instead of giving it the

whole map and locations of the obstacles. Then Path Planning randomly picks from within the radius of points in relation to its current location and the goal point, which it is weighted towards. The algorithm builds a graph spreading across the search space until the goal point is reached. To then find the path from the start to the goal in this graph, Path Planning is used again on this search graph, starting at the goal and moving to the start.

Creep AI: Creep AI ended up quite different from the initial approach. moveCreep, was implemented as C++ function. moveCreep uses the path found in Path Planning. The function can both create creeps, as well as just move them. The path is an array of points to travel to, ordered from goal to start. To move a creep, its position changes from its current location at the ith location in the path array to the i - 1th location. This is done for each creep. There is an array of creeps, with creep being a structure consisting of a location and the creeps health. The function takes a boolean, and if it is passed true, then it will move all the creeps and then create a new one at the start point, which is also the last index in the path array. When given false, Creep Ai just moves the creeps.

The functionality of hurtCreep was migrated into TowerAi.

Tower AI: We choose to do damage to the closest creep to each tower if it is within the tower's range. Tower AI accesses the creep array and the Towers array. For each tower in the array of towers, the closest creep to the center point is found, if this creep is in the tower's range the creep is attacked. To damage a creep, the damage dealt by the current tower is subtracted from the health of the targeted creep.

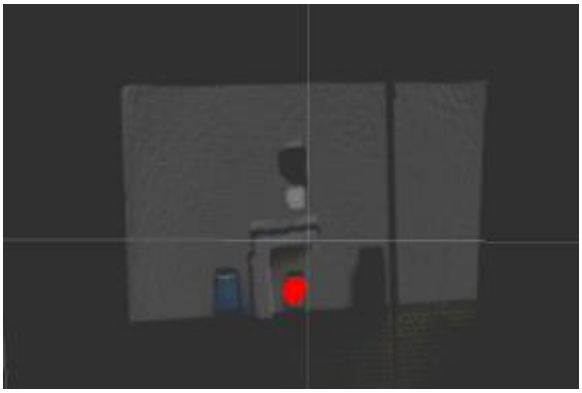
Visualization: Visualization turned out to be quite simple, but requires a good amount of preparation because everything must be stable and the Kinect's viewing angle must be perpendicular. Since the projector overlays an image on top of the physical game space, only a few items needed to be visualized. The creeps were published to RVIZ as a point cloud, with each point representing the location of a creep. The attacks from tower to creep were visualized with a line drawn from the top of the attacking tower, the center point found in towerFind, to the location of the attacked creep.

Evaluation Metrics

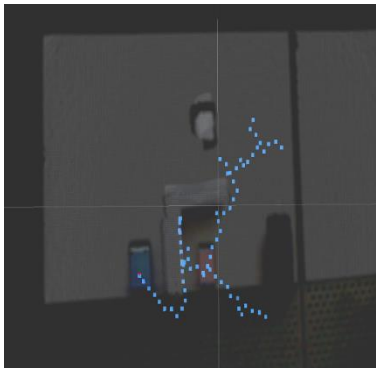
Game Map: Two testing methods were used for Game Map. The first method involved a testing suite. Sample point clouds were manually initialized, depicting various simplified versions of possible situations Game Map would encounter. The gameMap, maxOccuringFunction, and towerFind functions were tested with this data, and the results were compared to the expected (true) values. Through these tests, we could decide if the proper mode z value, start point, goal point, tower numbers, and tower locations were produced by the code.

The second method deployed was the use of actual Kinect data to test the functions in a more comprehensive test. Rosbags of point clouds were taken from sample game set ups. The values for mode z value, start point, goal point, tower numbers, and tower locations were found in with RVIZ. We then ran Game Map on these sample sets and compared the resulting values to the expected values. If resulting and expected values differed only by a small margin, an error of a few millimeters, then the test would be considered successful.

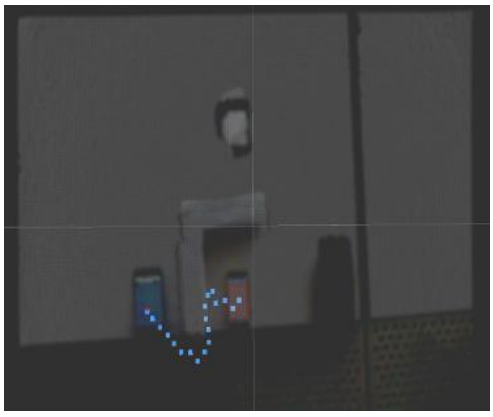
Path Planning: Due to the random element of RRT, no fixed test cases where written. Instead for various pieces of sample data, we ran the RRT algorithm and observed the result, which was visualized in RVIZ, overlaying the filtered point cloud. At no point should the given path collide or intersect with any of the obstacles.



A Visualization of the Search Radius



The Visualization of the Path Tree Produced by Path Planning



The Final Path

Tower AI: Tower Ai will update the creep array with the damage dealt by the Towers. Several test cases will be set up where the set of towers were defined, as well as the location of the creeps. Based on this information, we could determine what the expected damages would be in the updated creep array. By testing the actual values with the expected values, we could determine if Tower AI functions properly.

After the fixed test cases, we will RVIZ visualizations of the Towers running to confirm that the towers attacked the correct creep, and dished out the proper damage.

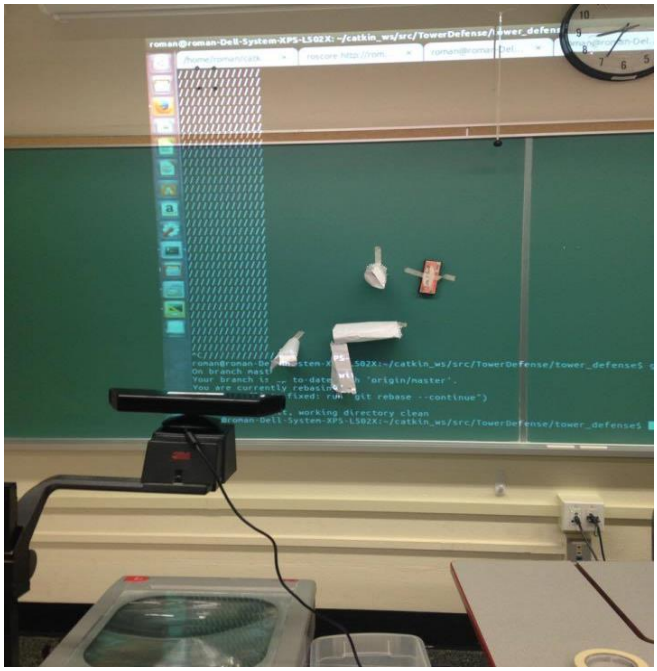
Creep AI: moveCreep moves the creep on the path and checks if a creep has reached the goal position. The function will be called n times on an n length path, alternating between true and false as the given value. Each time moveCreep was called the location of all the creeps were checked against their expected locations. Since true and false is alternating, the test should end with n/2 made creeps, which cover every other point in the path. Furthermore, after n steps moveCreep should also have reported that a creep reached the goal.

Move creep functionality is very visible in RVIZ test as well. We just needed to observe the creeps moving down the found path.

Visualization: Testing the visualization was done in two ways. First, RVIZ was used to visualize the markers for various sample point clouds. The location of the markers was checked to see if they lined up with the expected location. For example, the tower's attack must start from the top a tower and stop at the location of the creep it is attacking. The positions of the creeps were printed out and then checked against the position of the creeps in the visualization in RVIZ.

Next visualizations were tested with the projector. Similar to the first testing procedure, the markers must line up. The difference is that instead of comparing the markers to the sample point clouds, the marker locations are compared to the locations of the real objects. The tower attacks must look as if their origin is the top of a tower. Furthermore, the creeps should appear to be navigating around the obstacles, as well as looking to start at the start and head for the goal.

Experimental Results and the Evaluation of the Chosen Metrics



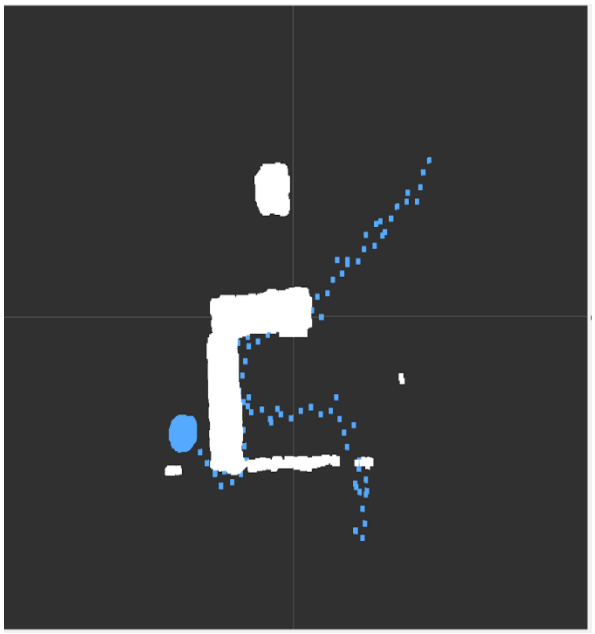
The experimental setup from behind the Kinect's point of view. The red eraser is the start point.

Game Map: Game Map had the most extensive test cases. In our final implementation, Game map's final implementation passed all the test. We also did visual testing with RVIZ on rosbags, to also ensure Game Map's functionality. In addition, we also printed out values such as the distance of the ground plane from the Kinect, the start point, the goal point, and the number of towers found. This way when we were testing other parts of the program, we could confirm that Game Map was still working properly. Game Map formed the base of the Tower Defence game and it was imperative that it worked.

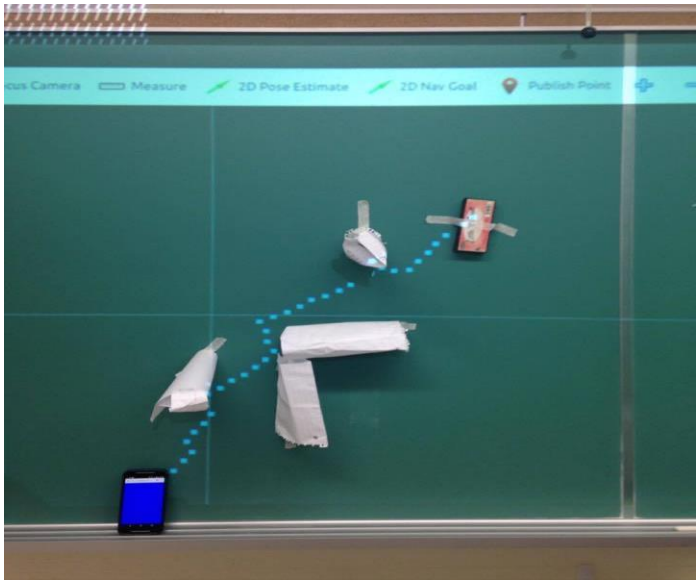
Tower AI and Creep AI: Unfortunately we did not have a chance to test these functions as robustly as we wanted. Compared to the other components of the Tower Defense, Tower Ai, and Creep AI were not complicated to implement. In the time crunch, we debugged these on the fly while working on Path Planning. We caught the noticeable bugs through this testing method, but more testing should be deployed to check for errors, especially for weird, uncommon edge cases.

Path Planning: Path planning caused many problems, and had extensive tests and debugging efforts devoted to it. The non-deterministic behavior of the Path Planning algorithm made it difficult to test. Still, we found many errors and bugs. For example, we tried giving the RRT a radius by comparing the number of points within the current radius to the number of points in the initial radius if they were to match within a certain error then we would know that there is enough space for our creep to move, unfortunately our filtered Kinect point cloud is not evenly distributed across so, therefore, we had denser areas of points near obstacles which led the path planning algorithm to think that it had enough space even near obstacles. This approach was slower the other approach has been used, and for this approach to work, we would improve our base plane finding algorithm such that it is evenly dense in all areas.

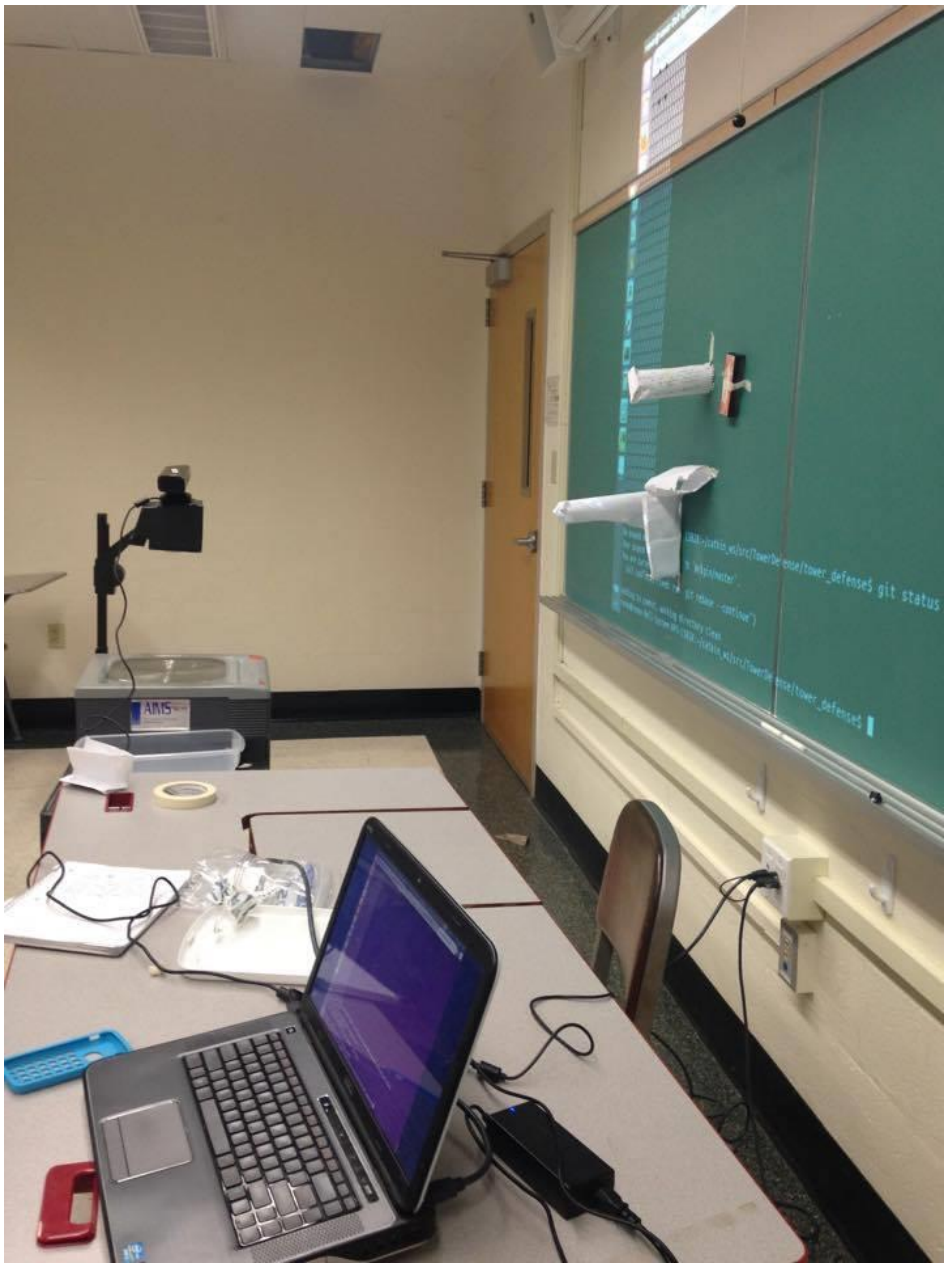
We relied heavily on RVIZ visualizations in our testing of Path Planning, visualizing the paths found when Path Planning. Not only did we use RVIZ but also set up a projector while testing Path Planning. This way we could test our projection method, while also working on the RRT algorithm.



The map rendered in RVIZ. White areas are obstacles, the blue dots are the path, and the large blue dot is the search radius.



The results given by the final RRT solution. The found path from the phone to the eraser is projected onto the board in the form of this blue path.



The general testing setup viewed from the side. Note the Kinect in the mid-left section of the image.

Here is another visualization tooled used to test. The red dots are the base plane, white are the obstacles and the blue are the path from start to finish. Since Path Planning considers points on the base plane, it was useful to visualize them.

We took numerous videos of different stages of testing.

Test of Projector Visualization: Path Planning

- https://youtu.be/LfkiKvjyFMI

Tower AI, Creep AI test:

- Test in RVIZ: One Tower
  https://www.youtube.com/watch?v=upFLYlJA-vw

- Test in RVIZ: Three Towers
  https://www.youtube.com/watch?v=mvp00R7wljM

Final Version Test:

- Short Path
  https://www.youtube.com/watch?v=Jup8oPqyc2s&feature=youtu.be&t=8s
- Long Path
  https://youtu.be/TKH9wbTvMCk

Any Additional Technical Details to Reproduce Work

Things that would have been good to know before we started this project:

- The Kinect works well primarily with a native installation of ubuntu, and the old Kinect works with usb2
- We used ROS Kinetic as well as openni.launch to work with the Kinect
- In order to use the PCL library to convert a pointcloud2 from a Kinect to something meaningful you should change your CMakeLists.txt
- In general, change your CMakeLists.txt whenever you need to add libraries
- The Kinect should be placed roughly a meter away from the wall, to work with ours .rviz. In addition, with the Kinect's minimum range being .8 meters, the Kinect should be situated at least 1 meter away.
- Roman will make it public after finals if he has a chance to clean it up a little.
- https://github.com/romanganchin/TowerDefense