Senior Project

Multi-Team Strategic Game Development

Author: Alvin Feng
Advisor: Dr. Michael Haungs
June 5, 2014

Introduction

Video games have been captivating audiences since the Atari released *Pong* for arcades in 1972. Since then, video games have exploded into a variety of different genres, ranging from first person shooters to role playing games to real time strategy games. In recent years, there has been an increased emphasis and importance placed on multiplayer games where players can compete against other players across the internet, with some major video game releases even going so far as to be multiplayer only, such as *Team Fortress* or *League of Legends*. Most multiplayer games are focused on competition and conflict between two teams; there are very few games that even offer an option to have more than two teams. The *Warcraft* series and *Starcraft* series, both developed by Blizzard, are the most famous examples of real time strategy games with a focus on multiplayer action. Although both offer the option to play with multiple teams, this option is heavily disregarded by the multiplayer community, and the number of players is limited to only eight players. A strategic game with a focus on a large number of players separated into three distinct teams can provide interesting insight as to how players cooperate and interact in a multi team environment.

This project is designed to explore the multiplayer experience of video games. Players will compete on teams to control units to compete against one another. Each game match will consist of three teams each having up to five players trying to take control of certain key points on the map to gain more points than the enemy teams.

This paper will first go over the background needed to understand key technologies related to game development. Then, there will be an overview of the gameplay and elements of game design needed for the game. Key technologies in networking and pathfinding AI will then be covered, followed by a look at other multiplayer real time strategy games on the market. Finally, the resulting project will be summarized before a look at possible future improvements on the game.

Background

There are many different game engines available to aid with the development of a video game. This project will be implemented with Unity, a game development engine popular among indie game developers. For non-commercial use, Unity provides a free version with only a few advanced features not available. Among the benefits of using Unity, the first is that basic graphics is integrated deeply into Unity. The developer can easily insert objects into the world based on simple geometric volumes or models, and all the complicated stages of the graphics pipeline (mapping the objects to the camera's perspective and then rendering to the screen) is taken care of. Unity scripts, written in C#, also provides methods for dealing with user input. It is easy to listen for mouse click events and keyboard events. In addition, Unity provides a comprehensive library of methods related to linear algebra that is common to game development. For instance, vector math and quaternion math can be abstracted out to built-in

methods of Unity. The final benefit of Unity is it's cross platform capabilities. With just a configuration change, Unity can export the final project to Linux, Windows, Mac, or even a web browser, allowing almost anyone to play the game, regardless of platform.

One of the most difficult aspect of game development is artificial intelligence. A part of artificial intelligence is path finding, the ability for an agent within a game to find a viable path from one point in the game world to another. One of the best algorithms for path finding is the A* search algorithm. Two important properties of A* search is that it is both complete, and in most circumstances admissible. This means that it will always find a solution if it exists, and if it is admissible, it will find the optimal (shortest path) solution. A* keeps a priority queue of nodes it must visit, and searches them in order based on two functions. The first function is the path-cost function denoted as *g(x)*, which is the known distance from the starting node to the current node. The second function is the heuristic denoted as *h(x)*, which estimates the distance from the current node to the goal. If the heuristic function is admissible, that is if it does not overestimate the distance to the goal, then the A* search algorithm is optimal and will always find the shortest path.

Another important component of multiplayer games is networking. There are two common protocols to choose from for networking, Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). TCP has the benefit of being reliable, meaning that packets are guaranteed to be delivered from one endpoint to another in the order that they were sent. This makes TCP simple to use, but has the down-side of being slower. While UDP does not guarantee that packets will be delivered in order, or even that packets will reach the endpoint, it has the advantage of being very fast. Besides the different packet delivery protocols, there are also several different models used for networking games and transferring data between clients of the game. In a server-client model, clients send information about the game to the server, and the server broadcasts data about the game to all of the clients. There are two types of servers, authoritative and non-authoritative. In an authoritative server, all calculations and computations regarding the current game state are done on the server, and the information if broadcast to the clients. In an authoritative server, the clients simply acts as consoles displaying the information broadcast by the server. In a non-authoritative server, the server will typically rebroadcast the inputs of the clients to the rest of the clients. Then, the clients will perform the calculations and computations to simulate the game state. An important requirement for a non-authoritative server is that the simulations be determinant. Typically, problems arise for non-authoritative servers when performing floating point math on different machines. Comparing the two models, an authoritative server is generally more bandwidth intensive than a non-authoritative server.

Game Description

When first starting up the game, the user is presented with the main title screen shown below in Figure 2. In the upper left corner, the user is show the current state of the network

connection, and can type in an IP address into the Server IP text field. To connect to a game, the user must type in the IP address of the server, and then click the Connect To button.

Figure 1. Main Title Screen

Player Objectives

Upon entering the game, the player will be assigned to one of three teams: red, blue, or green. They will then be assigned a squad of five tanks to control that are spawned in their team's base. The map is set up so that the three teams have bases at the endpoints of an equilateral triangle. The objective of the game is to accumulate the most points for your team. Points are gained by capturing control points on the map. In each base is a control point, as long as one in the center of the map. For each control point controlled, the team will earn ten points per second. Furthermore, each additional base past the first will grant more and more bonus points (3 per second for each additional base), providing incentive for teams to acquire as many bases as possible. Table 1 shows the point earned per second with each base.

Bases controlled	Points earned per base	Total points earned
0	0	0
1	1st base: 10	10
2	1st base: 10 2nd base: 13	23
3	1st base: 10 2nd base: 13 3rd base: 16	39
4	1st base: 10 2nd base: 13 3rd base: 16 4th base: 19	58

Table 1. Points Accumulation for Bases Controlled

User Controls

Users can select individual units they own by left-clicking on them. To select a group of units, the user can left-click and hold, dragging out a selection box. When they release the left mouse button, any units they own within the box will be selected and respond to user input. The player can order the units to move and attack by right clicking when units are selected. Right clicking on the map will cause selected units to move toward the clicked location. Right clicking on an enemy will cause selected units to attack that specific enemy if they are in range. If they are not in range, then the units will move towards the targeted enemy until they are in range.

Control Points

To capture a control point, a team must move at least one unit within a small radius of the control point. This will start the capture process for that team, and as long as there remains at least one alive unit for that team within the radius, the capture process will continue. If at any point there are no alive units for the team that started the capture process in the radius, the process will be cancelled and must be restarted. After the capture process continues for thirty seconds, the ownership of the control point will switch from the previous owner to the new team, and will remain under the control of the new owner until another team successfully captures the point.

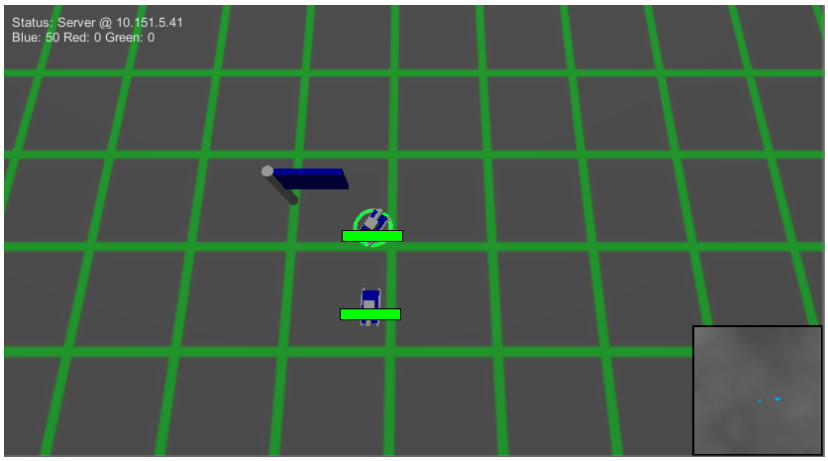


Figure 2. A blue unit has successfully captured a control point

Heads Up Display

There are several elements that help provide the player with information about the game. The HUD shown below in Figure 3 provides four different pieces of information. In the upper left corner, the score of each team is shown. In the bottom right corner is a mini-map, which shows the locations of other units in the game. Each unit is designated by a small block matching the color of the team that the unit belongs to. Finally, across each unit is a bar

indicating the health of that unit, and underneath each selected unit is a green circle indicating that the unit will respond to commands by the user.

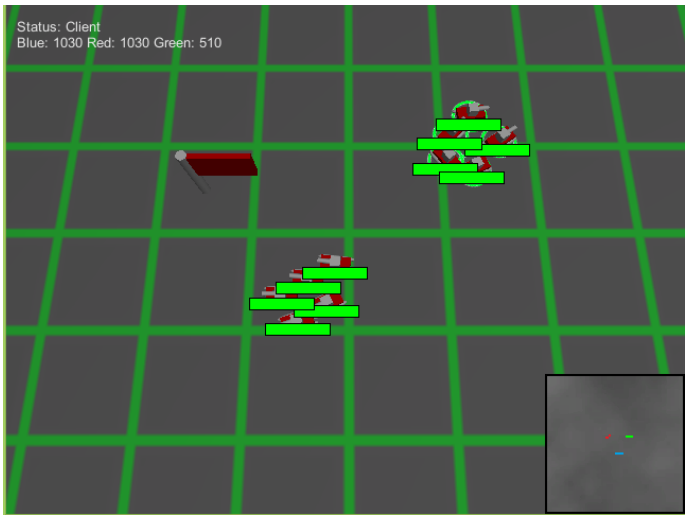


Figure 3. Heads Up Display – Selected units (upper right group) has green circles under them, while not selected units (lower left group) do not have green circles

Design and Implementation

Software Architecture

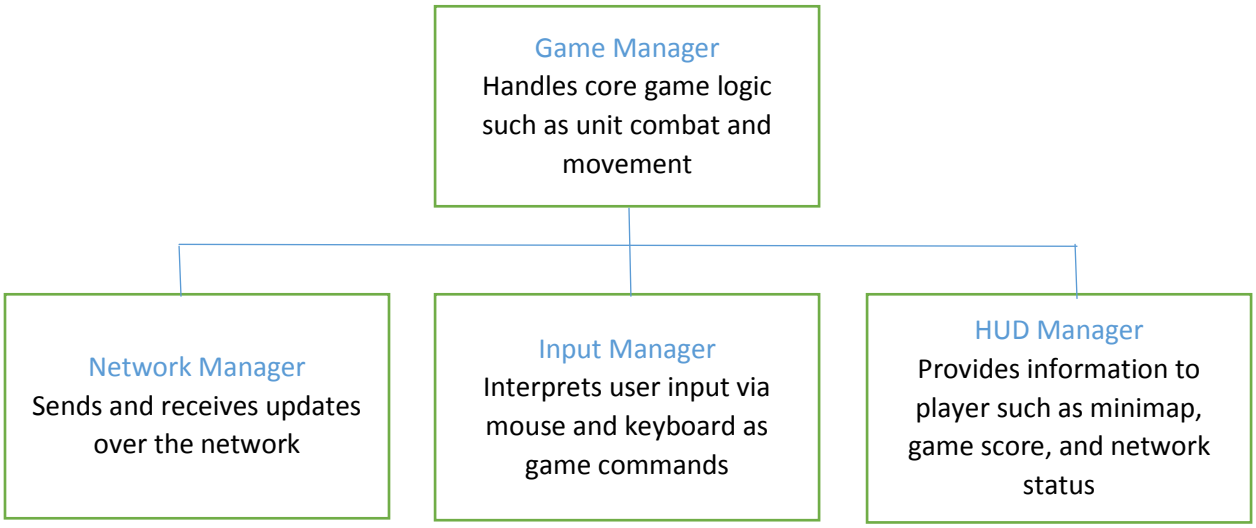


Figure 4. Major component diagram

The game is mainly handled by the primary game manager, which contains a the game scene. This includes a central place to find a list of all of the units and objects, including the map of the game. The game manager also has several other managers, the three most important being the network

manager, input manager, and HUD manager. Because the game manager needs constant updates from the network and input manager, they are very closely linked. All of the input and network updates tend to have an effect on different units, which are contained within the game manager. The HUD manager also requires a little bit of information from both the game manager and network manager. The game manager can pass the network status to the HUD, and information about the map and units for the HUD to use for the minimap.

Pathfinding

A* search is typically the best path-finding algorithm available as it generally finds the optimal path in the shortest amount of time. However, A* search is limited by several requirements. For A* search to work, the algorithm must work on distinct nodes. The placement of objects in the game world of Unity is specified by a vector of floats, which is not suitable for creating a graph for the A* search agent to traverse. To create this graph, the game world is divided into a grid of squares, where each square is ten units wide. Each object in the game world can determine which square it is in by simply dividing its position by ten. At the beginning of the game, a graph is generated based on the map of the world. If two adjacent squares on the map do not have the same height (meaning different levels), then the A* search agent cannot move in between those squares. The A* search agent returns a set of waypoints for each object as a path. However, one weakness of A* search is that the environment must be static, meaning that the graph defining the paths between squares does not change. As different players move units around the map, these units becomes obstacles that must be avoided by other units. This dynamic movement cannot be handled by A* search. Therefore, A* search returns the waypoints for each object, but a different method must be used to navigate an object between waypoints.

The following two code snippets contain pseudocode for the path finding code. The first snippet demonstrates the algorithm for traversing between nodes on the map. Starting from a single node, more nodes are added to a priority queue as they become accessible. The priority queue sorts the nodes based on the sum of two scores: the first is the number of nodes traversed to reach this node, and the second is the estimated distance to the goal. Combined together, a lower score means that the node is more likely to part of the shortest path to the goal, and is popped off the queue first to examine.

```
AStarFindPath(Unit unit, Node start, Node end) {
    Node current = start;
    PriorityQueue q;
    current.cost = 0;
    Enqueue(current, 0);

    while q is not empty and current is not equal to end:
        current = q.dequeue();
        For each neighbor node in each cardinal direction:
            if neighbor is passable and not visited:
                Mark neighbor as visited;
                neighbor.parent = current;
                neighbor.cost = current.cost + 1;

                /* priority queue will place node in queue based on cost to reach node
                 * from start and distance from node to the goal node */
                Enqueue(neighbor, neighbor.cost + DistanceToEnd(neighbor, end));

    /* current will be the end node when exiting previous loop, and need to reconstruct
     * list of waypoint nodes for the unit to follow */
    while current is not null:
        Add current to unit list of waypoint nodes;
        current = current.parent;
}
```

Code Snippet 1 – Pseudocode for A* pathfinding for a single unit. Pop and push nodes using a priority queue to quickly retrieve the node with the lowest combined cost and expected distance to the goal.

The second code snippet is used to find the path in between the nodes returned by the A* search. This direct path also uses flocking behavior to avoid colliding with other nearby units. Nearby units will have a small force that affects the movement of the unit that is stronger the closer the units are.

```

Unit.MoveToPoint(Point dest) {
    Point next = this.position + ((dest - this.position) * moveSpeed * Time.deltaTime);
    Vector avoidCollisionForce;

    for each Unit u within collision avoidance radius:
        float ratio = DistanceBetween(this.position, dest) / collision avoidance radius;
        avoidCollision += ratio * (u.position - this.position);

    next += avoidCollision;
}
```

Code Snippet 2 – Pseudocode for collisions avoidance. Compute an avoidance vector for each nearby unit and scale it based on how far the other unit is. Add the total avoidance vector to the unit's next point so that the unit will steer away from other units.

Adjacent squares are close enough that a straight line between two points in each square represents an appropriate path for the game object. During this straight line movement, the object must navigate around dynamic obstacles, such as other units. Flocking behavior will be used for collision avoidance in this scenario. For flocking behavior, each unit will examine nearby units (within some small radius), and add a small components to its current velocity that is directed away from the other units. This will cause units that are near each other to steer slightly away from each other, while still moving in a relatively straight line. One downfall for this solution is that the collision avoidance only works on a local scale. When a unit encounters

a large blockade, where an entire square in the grid is blocked off by other units, the navigating unit can sometimes get stuck.

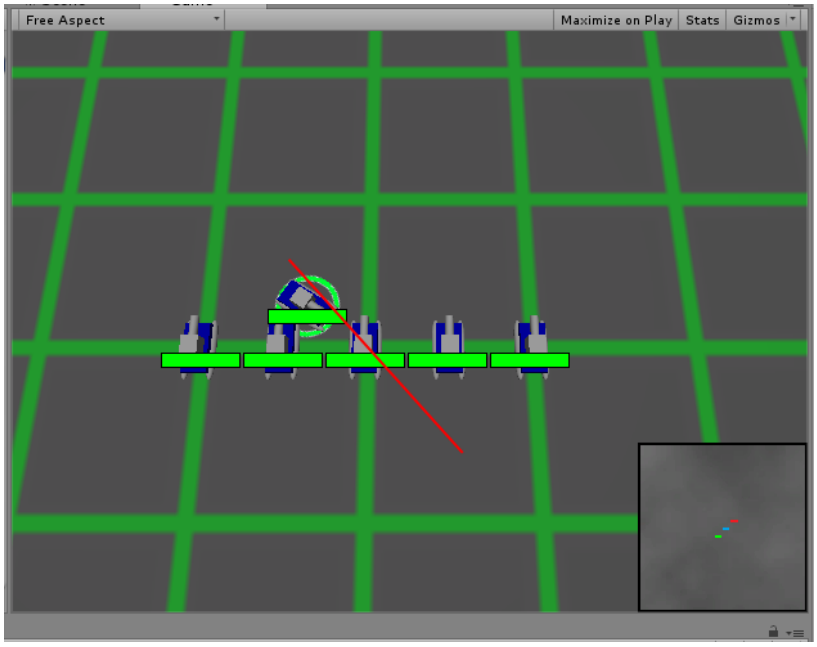
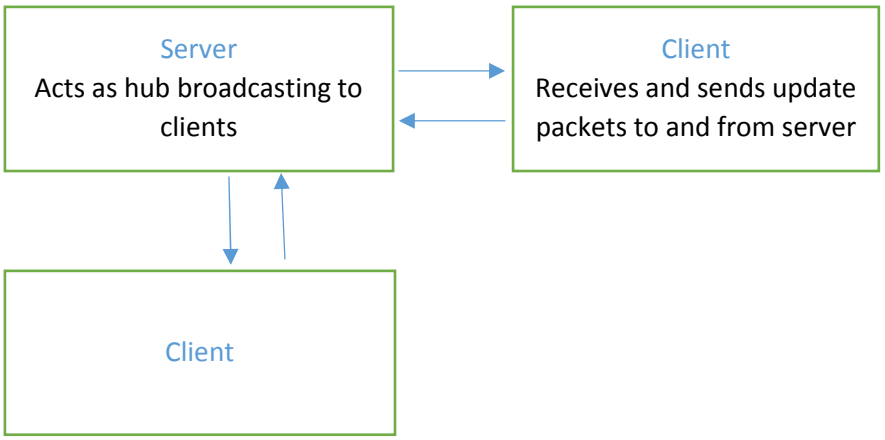


Figure 5. A unit trying to move along the red path is unable to avoid colliding with a blockade of units and becomes stuck

Networking



The network infrastructure behind a multiplayer game has a great impact on the responsiveness of the game. Networking is an especially difficult problem because it is most limited by real world hardware for transmitting data, and introduces a problem about scaling. Each object in the game needs to be able to transmit important information about the state of the object, such as position, rotation, and health. As the number of players in the game increases, the number of objects greatly increases as well as the network traffic.

For video games, the time it takes for the game to receive updates from the network is directly related to the lag that the player feels. In general, for high action games users begin to feel that performance is degraded when the time of flight for packets start to exceed 100 to 150 milliseconds. Packets that are dropped by the network are not particularly important for this application, because the information in the packets is constantly being updated. For example, if a unit is at one position for one packet, and another position for another packet, the first packet does not really impact the game greatly if it is dropped. TCP introduces an additional delay as it needs to acknowledge every packet, and retransmit packets that were dropped. By using UDP instead, the game can focus on only the most recent, relevant data transmitted over the network, with the benefit of increased speed for updates.

When starting a game, one host will act as the server, and listen for incoming connections on a specific port. This computer will be the central hub that all clients connect to. When a client connects over the network, the server gives that player ownership of a squadron of units. For those units, the player controlling them has authority on the states of that unit. This allows computation for pathfinding to be offloaded to each client, rather than being computed on the central server. This is particularly beneficial because it allows the server to be any ordinary computer, rather than having to be a powerful machine that can process each players input for every unit. On a test laptop with a dual core CPU running at 2.5 GHz, 8 GB of memory, and a dedicated graphics card, performance of the game noticeably decreased when the single computer had to perform the pathfinding calculations for approximately 200 units. In a game of 32 people there might be approximately 200 units in a game. By spreading the computations across each client, the server has more breathing room for other computationally intensive tasks such as graphics. This also means that the game can scale to have more players, more units per player, or just more gameplay in general. It also allows the game to be hosted by older, less powerful machines.

The next code snippet demonstrates the network update for each player. This update is called about sixty times per second in the game. First, the update pushes out any relevant information about units that the player controls. Then, the update looks for any packets sitting in the network buffer, and updates the associated unit.

```
1
2  Player.NetworkUpdate() {
3      for each unit Player controls:
4          Generate packet with unit id, health, position, rotation, and target;
5          Send packet with NetworkManager
6
7      for each packet in buffer:
8          Get unit id, health, position, rotation, and target from packet;
9          Update unit associated with unit id;
10 }
```

Code Snippet 3 – Continuously send UDP packets and receive packets. There is no wait for an ack for any packets.

Whenever a player issues a command to a unit, such as a movement command, that client will compute the waypoints that the unit must take to get to its destination. The network

will continuously send updates about the units current position and orientation to the server, which acts as a hub. The server simply has to pass the packets on to the other clients. This is very similar to a peer to peer implementation, in which there is not an authoritative server.

Related Works

A real time strategy game with many simultaneous players can be categorized as a massively multiplayer online real time strategy game (MMORTS). There are very few games that fall under this category, with only one AAA release.

Company of Heroes Online

Company of Heroes was the most recent release of an MMORTS game, and also one of the biggest. It was published by THQ and developed by Relic Entertainment, and sold thirty thousand units, primarily in Europe[1]. Although having similar gameplay, Company of Heroes still only allowed four players in a match, and did not allow more than two teams.

End of Nations

End of Nations was a game that was announced in 2011, making it the most recent development of a MMORTS. However, after having an open beta to demonstrate the online RTS gameplay, the game was announced to be changing to a different type of gameplay in 2013. Since March 2014, it seems that all development has been halted for End of Nations[2].

Shattered Galaxy

Shattered Galaxy is an old game released in 2001, and was actually the inspiration for this project. Of all the similar MMORTSs, Shattered Galaxy put the most emphasis on MMO, with up to about thirty players per battle, with two teams fighting over control points in each match. Users can play for free, but are severely handicapped. Premium players are charged 9.95 USD per month and receive benefits large enough to consider the game pay-to-win. One of the most noticeable drawbacks of Shattered Galaxy was that it was developed in 2.5D, with all art assets being 2D graphical sprites. It also suffered from terrible netcode that resulted in tremendous amounts of lag for all players. Remarkably, the Shattered Galaxy community still hangs on, and is actually larger than any of the other MMORTSs player bases. At peak hours, there may be up to two hundred players on the server. This is most likely due to the unique gameplay of having many users in each battle, along with the persistent characters for players to develop (similar to an RPG). However, active development of the game was halted in 2006, and no new updates have been released since then despite the significant paying user base.

Conclusion

In terms of implementation, pathfinding and networking are still some of the largest challenges game developers face today. There are many circumstances even in some of the largest AAA games developed today where both fail. For instance, in *Starcraft II*, units will often not take the most optimal path to a location[3]. In the ubiquitous Call of Duty series, peer to peer networking has introduced problems with lag compensation that generates tens of thousands of videos of complaints on Youtube[4]. While pathfinding is generally very efficient and fast, it is not always accurate and units can become stuck in some circumstances. The networking in this project occasionally desyncs, which can affect the players experience negatively.

Strategy games tend to have very dedicated fan bases. By making each player a smaller cog in the larger strategic engine of a team, players are forced to interact and cooperated differently with one another. Strategic decisions made by individual players on the micro level impact the overall state of the game, and even change the team's overall strategy. The splitting of players into three teams further changes the dynamic with team strategy. The addition of another team introduces different options such as picking on the weakest team to gain easy points, or working together to focus on the dominant team to disrupt the state of the game.

Future Work

There are two main areas for future improvements to the game. The first is regarding gameplay. One key feature missing is the inclusion of a fog of war effect. Most strategic games limit player information by not displaying units in the world that are not in sight of the player's units. As each player can currently see everything on the map, each player has perfect information, which makes decision making much easier. Also, there is currently only one unit in the game, the standard tank. By providing different units that have different advantages and disadvantages, gameplay could be greatly expanded. Different combat units would be effective at fighting certain other units, and weaker against others. Some units might specialize in movement to take control of the map, but be generally weak in combat. Support units might heal other units or give more vision of the map. Players would have many more choices on the micro level for which units they bring into battle depending on what the team needs. Another improvement for gameplay would be more interesting and varied maps. The current map is very open, with no special routes or passages from one area to another. A more interesting map would have more areas blocked off, and funnel units through certain zones. Not only would this cause there to be hotspots of action on the map, it would also enhance strategic choice about movement. For instance, players might want to take a longer, less travelled route to go bypass a blockade and go straight for a control point, or take a certain path to flank and surround their enemies.

The other major area for improvement involves what the player sees and hears when playing the game. The inclusion of sound would greatly enhance the player experience, and provide a more immersive experience rather than playing in a completely mute environment.

The current graphics are also aesthetically not pleasant. More detailed models, effects, and terrain would help to make the game more engaging to the user.

Works Cited

1. Company of Heroes Sales
http://www.vgchartz.com/game/7304/company-of-heroes/

2. End of Nations Development Updates
http://www.pcgamer.com/game/end-of-nations/

3. Pathfinding Challenges
https://www.youtube.com/watch?v=IZpgMnu_lAk&index=5&list=FLOFgzMg18UkRR21EFYnm_gQ

4. Lag Complaints Call of Duty On Youtube
https://www.youtube.com/results?search_query=call+of+duty+lag+compensation