Space Flight Simulator
Mike Boyer & Shibo Ji
COSC 729 / Spring 2020

Overview:

The goal for our project is to create a multiplayer space simulator using the Unity3D gaming engine. This game is cooperative, having players join a single team to defend the allied forces from enemy forces. Both allied and enemy forces include ships and turrets. To win the game, players must defeat all enemy forces. Should all friendly forces be defeated in battle, the player's team will lose? Virtual reality is proper for this project, as it is a simulation of a space combat scene.

This paper will cover how the environment is defined, and how components such as multiplayer, animation, and avatar interaction was implemented to create this experience. This is a useful example of how a dynamic multi-agent environment can synchronizes across multiple platforms and works in cross-play with Linux and Windows. After addressing these topics, the paper will give a brief overview of gameplay.

Environment:

Since the game is set in space, a hand drawn high resolution background was added as a backdrop to enhance realism as shown in Figure 1. To draw the background, Affinity Photo Editor was used. Avatar models, including ships and turret operators, are freely available at www.kenney.nl. Allied and enemy forces are as shown in Figure 2. At the start of gameplay, both teams have 20 ships and 8 turrets.

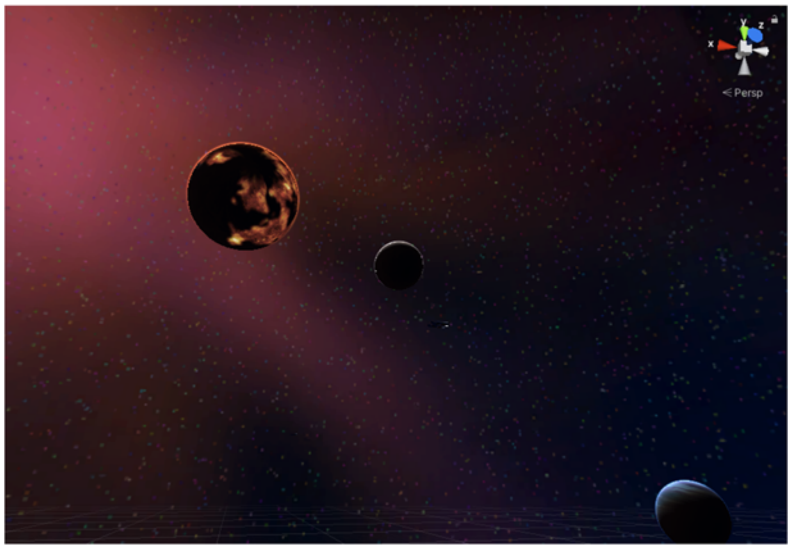


Figure 1 - Space Backdrop with Textured, Orbiting Planets

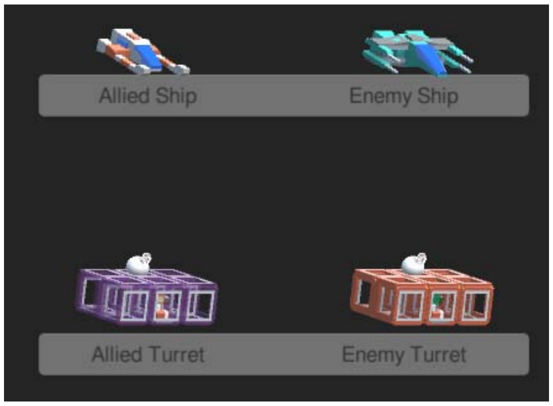


Figure 2 - Allied & Enemy Ships

To accommodate high speed ship travel over larger distances, the field of play must be greater than for a standard Unity3D game. This directly affects visibility at great distances. Using lower polygon count models allowed us to increase the viewing distance by a factor of ten while still running at reasonable framerates (>30 frames/sec) on a slower laptop featuring a first generation AMD A10 APU dating back to 2011 running PopOS (a Linux distribution based heavily on Ubuntu). To further afford players the ability to find enemy and friendly ships, particle trails have been added. This allows the user to not only find a moving object in space, but also to determine the direction of relative motion between another ship and the player. This allows the player to navigate on a course to intercept another ship. Particle trails are simulated in the world space, so that individual particles do not move along with their respective ship. This setting enables particle trails to remain stationary, giving other players an idea of how another ship is maneuvering.

To enhance immersion in a space environment, we added planets and orbiting moons as well as an asteroid field. All in-game models can be found at www.kenney.nl and the Unity Asset Store as free assets. The game objects representing these planets and asteroids are tagged such that the player will be destroyed upon collision with these objects.

Ship and base textures are also modified to better show team alignment. Material assignments are applied as shown in Figure 1 so the player can differentiate between and ally and an enemy. Planet surface textures used are on the planet objects as defined by the Unity asset package. This shows a more realistic surface on the planet, enhancing immersive quality by improving realism.

For lighting, our project uses a single directional light for ambience. Additional point lights are added as appropriate to enhance visibility and realism. One example is the turret point light. A small light is added to the turret, to help light up the base, and turret locations as shown in figure 3. Another example is each ship's exhaust has a small point light added in the same location of the previously mentioned particle trail. This helps give a better illusion of a ship's thrusters. Other point lights are instantiated with the laser blasts. Each laser's material color and light color are selected upon instantiation for NPCs. Player-fired lasers are green with a large green point light, while friendly and enemy NPCs use the colors blue and red, respectively. The point lights give off a glowing effect to any exposed surface within a predetermined radius.

Changing the color of lights helps in a team-based environment to enable a player to better understand which ship or turret is shooting by the color emitted at greater distances.

Audio is added to two object types for this simulator. When a laser is fired, a blasting sound is played. Additionally, when a ship or turret explodes, an explosion clip is played. This audio feedback helps to increase immersion without adding so much noise that the scene becomes unpleasant. Audio sources are ships and turrets, enabling both proximity and relative positioning to effect how sound is heard in Unity3D.

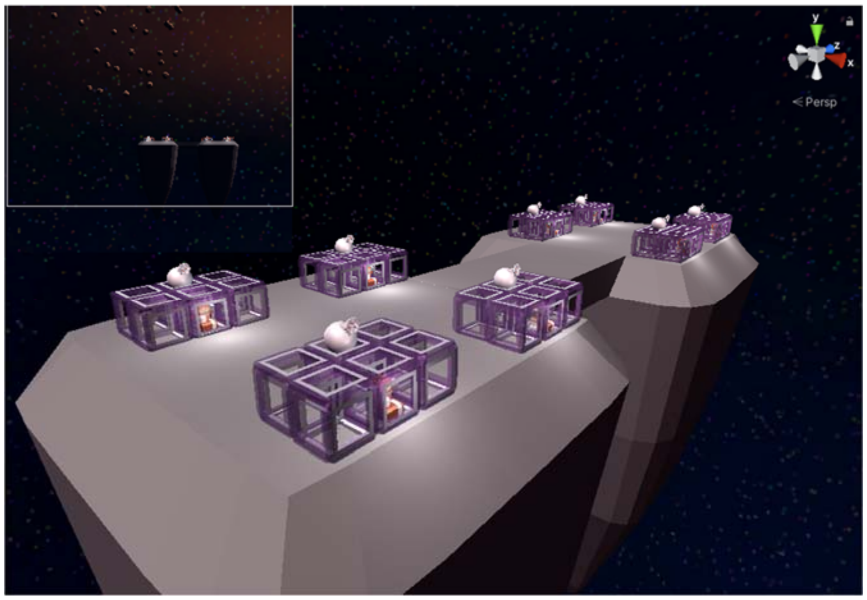


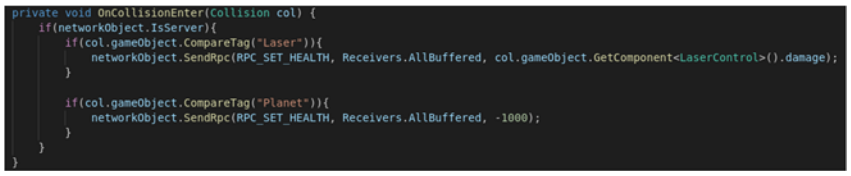
Figure 3 - Point Lights on Near and Far Bases

Multiplayer:

To enable cooperative play over the internet, the Forge Networking Remastered libraries were added. Forge Networking is a free and open source system designed to implement a multiplayer game over the internet or local network. It supports many different protocols, such as UDP and TCP. Additionally, it works cross-platform between Windows, Mac, Linux, iPhone/iPad, and Android. For our scenario, we chose a master server implementation, allowing local network or internet play. This afforded the ability to test remotely during the current pandemic with both Linux and Windows platforms. The master server can be found at http://www.titomoskito.com/forgenetworking/.

With this solution, we can set up synced variables (SyncVars), such as ship position and rotation or remote procedure calls (RPCs) to handle events that must be synchronized over the network. SyncVars and RPCs are registered by scripts referred to as network contracts. A network contract is set up to manage network-instantiated objects and contains each object's respective RPCs and SyncVars in addition to settings such as interpolation. One example of a synced variable is the player's ship. Every FixedUpdate call of the game loop first checks to see if the object is registered as owned by the local player. If not, the position and rotation are updated by the network. If the player is in control, they are free to maneuver their ship as required, and the network is informed of the updated position and rotation. Interpolation is applied to all ship positions in the game to smooth movement over time. This helps overcome network limitations in creating an immersive environment by reducing the jerkiness of non-client-owned ships.

RPCs are handled in a similar manner. One example is the SetHealth function. If the locally owned player ship collides with a projectile, planet, or asteroid, an RPC call is made to decrement the player ship's health value by a specified amount. This call is sent to all clients of a server, in a buffered fashion. Buffering allows for clients that join late to see the current representation of the virtual environment. Refer to Figure 4.



```
private void OnCollisionEnter(Collision col) {
    if(networkObject.IsServer){
        if(col.gameObject.CompareTag("Laser")){
            networkObject.SendRpc(RPC_SET_HEALTH, Receivers.AllBuffered, col.gameObject.GetComponent<LaserControl>().damage);
        }

        if(col.gameObject.CompareTag("Planet")){
            networkObject.SendRpc(RPC_SET_HEALTH, Receivers.AllBuffered, -1000);
        }
    }
}
```

Figure 4 - RPC Call Example

Input:

Player configuration relies on keyboard inputs for control. To turn left or right, the player may use either the A and D keys or the Left Arrow and Right Arrow keys. To change pitch, the player may use either the W and S keys or the Up Arrow and Down Arrow keys. See relevant keys highlighted red in Figure 5 below. The Z and C keys increase or decrease speed as shown in green in Figure 5. The space bar, highlighted blue, fires lasers.

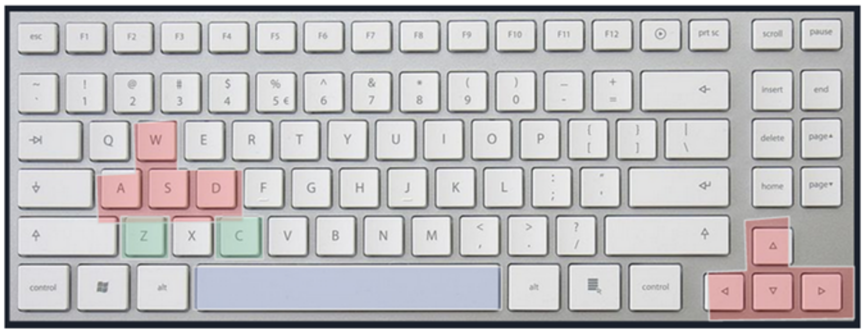


Figure 5 - Game Controls

Sensors:

Several different proximity sensors in the scene implement collision and detection functions. For collisions, player ships and the floating space bases use a mesh collider to match the shape of the actual ship. Planets have large spherical colliders. To implement the collision response, Unity3D requires that a rigidbody to be present. To hold position constant, certain rigidbodies are kinematic and have gravity effects disabled. Failing disable gravity so forces each object to fall (move in the -Y direction). Enabling the kinematic features of a planet would allow smaller objects such as lasers or ships to collide with the planet, pushing it in the opposite direction in space. As a result, marking these large bodies kinematic enhances realism.

For proximity detection, a combination of spherical colliders, box colliders, and ray casting is employed in this project. Proximity spherical colliders on friendly and enemy ships, set with an exceptionally large radius enable detection. To avoid collisions in space, rays cast directly of the ship's port and starboard wings. Each wing has a ray extending perpendicular to the ship's direction of intended motion and one ray extending in the direction of intended motion. Should any of these rays touch an object, the ship steers away, avoiding a collision. Once the requirement to maneuver has passed, the ship will either continue pursuing its target (if assigned) or continue patrolling by selecting a new random waypoint.

Turret proximity detection is like ship proximity detection for enemy detection purposes. Instead of a spherical collider, a box collider better simulates the turret field of view. If an enemy ship flies below the field of view, they exit the collider. Once this occurs, the turret position resets. With a spherical collider, this field of view would be more difficult to implement. For a pictorial of these colliders, refer to Figure 6.

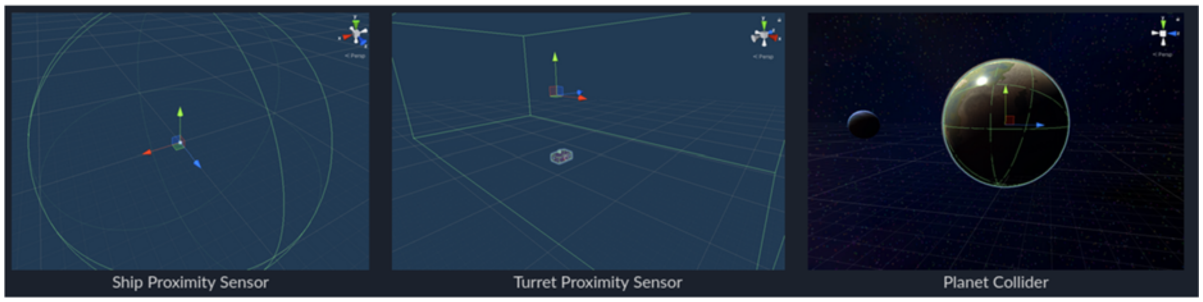


Figure 6 - Colliders

Animation:

Control for ship, turret, and turret operator animations are as shown in figures 7 and 8. For the enemy and allied ships, a decision whether to maneuver takes priority over other actions. This enables enemy ships to steer away from collisions to protect themselves before patrolling to a waypoint or chasing a target, enhancing realism. The ray cast-based collision detection was set to a small enough distance that ships can still collide with asteroids in the middle of the battlefield.

If a maneuver is not needed the computer-controlled ship will prioritize pursuing a target, if available, over patrolling to a randomly selected waypoint on the map. Should no target be available, the ship will patrol to one of nine pre-determined waypoints. To increase the likelihood of a dogfight, both allied and enemy ships share the same set of waypoints.

Turret operator animations occur under two conditions. Upon shooting from a turret, the operator will raise their arms, and lower them. Upon destruction, the turret and its building disappear in an explosion, leaving the turret operator exposed on the base. Each base in the scene has a pre-configured navigation mesh, or NavMesh, for the operator to travel across efficiently. Upon a turret's destruction its operator's Boolean value to enable panic mode is set to true.

During panic mode, the operator's NavMeshAgent functionality is enabled, and the operator picks a random spot on the NavMesh, and begins traveling to that point. To select a random point, a random point in space near the turret operator is chosen. Next, a function called selects the closest spot on the NavMesh to the randomly determined point. As the operator reaches their waypoint, the pattern repeats until the end of the game. Panic mode also triggers the operator's animation state machine, causing it to wave its arms as it moves across the base, as if in a panic.

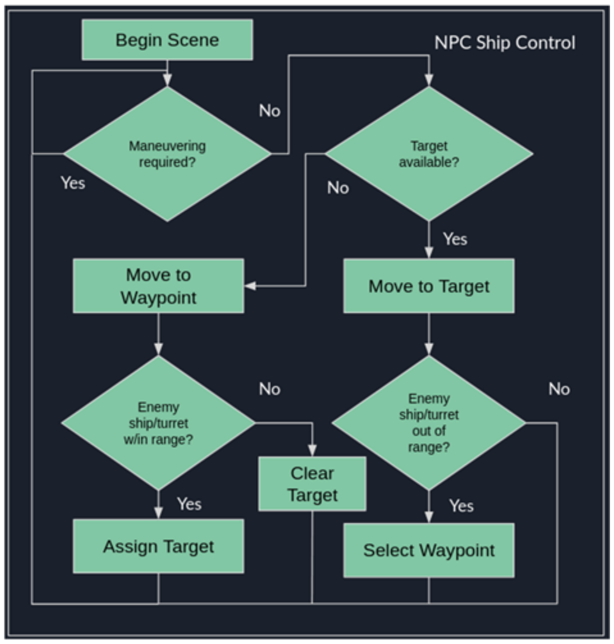
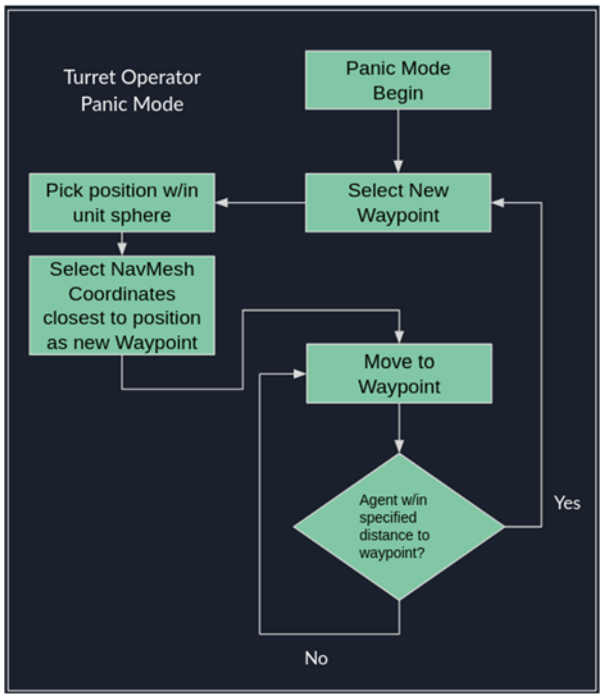
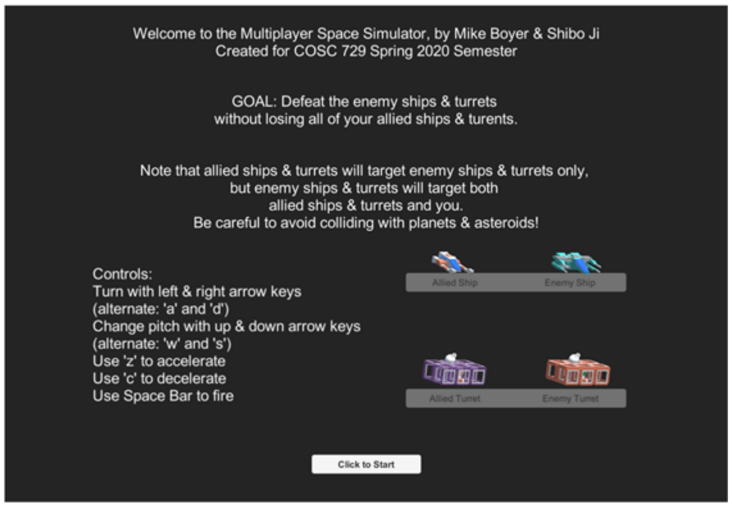


Figure 7 - Non-Player Character Ship Animation Control


Turret Operator Panic Mode
Panic Mode Begin
Select New Waypoint
Pick position w/in unit sphere
Select NavMesh Coordinates closest to position as new Waypoint
Move to Waypoint
Agent w/in specified distance to waypoint?
Yes
No
Figure 8 - Turret Operator Animation Control "Panic Mode"

Gameplay:

At the beginning of the game, players are given an info screen defining objectives and controls for end users. A screenshot of friendly and enemy ships from the indoctrination screen is shown in figure 2. In order to proceed from the beginning screen, the player must click the button to continue to the multiplayer screen. This indoctrination screen is shown in figure 9 below.


Welcome to the Multiplayer Space Simulator, by Mike Boyer & Shibo Ji
Created for COSC 729 Spring 2020 Semester

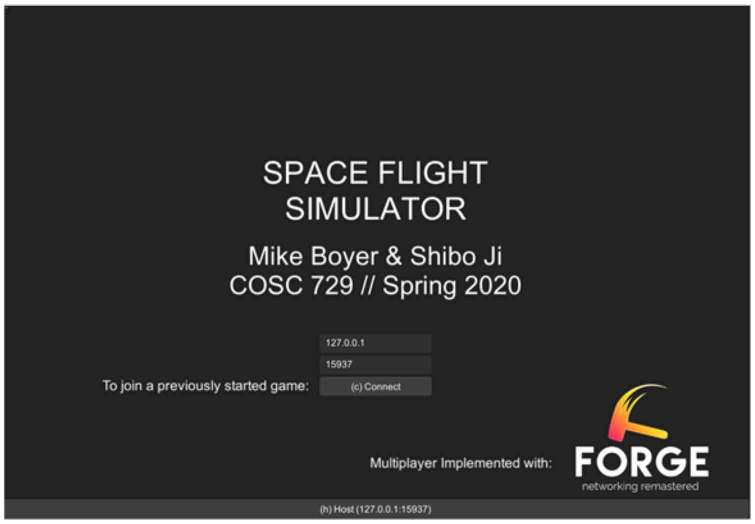
GOAL: Defeat the enemy ships & turrets
without losing all of your allied ships & turents.

Note that allied ships & turrets will target enemy ships & turrets only,
but enemy ships & turrets will target both
allied ships & turrets and you.
Be careful to avoid colliding with planets & asteroids!

Controls:
Turn with left & right arrow keys
(alternate: 'a' and 'd')
Change pitch with up & down arrow keys
(alternate: 'w' and 's')
Use 'z' to accelerate
Use 'c' to decelerate
Use Space Bar to fire

Figure 9 - Indoctrination Screen

The multiplayer screen allows players to host an internet match or join a match by specifying an address and port of the host game. By default, ports 15937 and 15940 are used for the Master Server and may require a modification to a user's network configuration for hosting a game. Selecting either host or join will take the players into the scenario. The multiplayer screen is shown in Figure 10.


SPACE FLIGHT
SIMULATOR

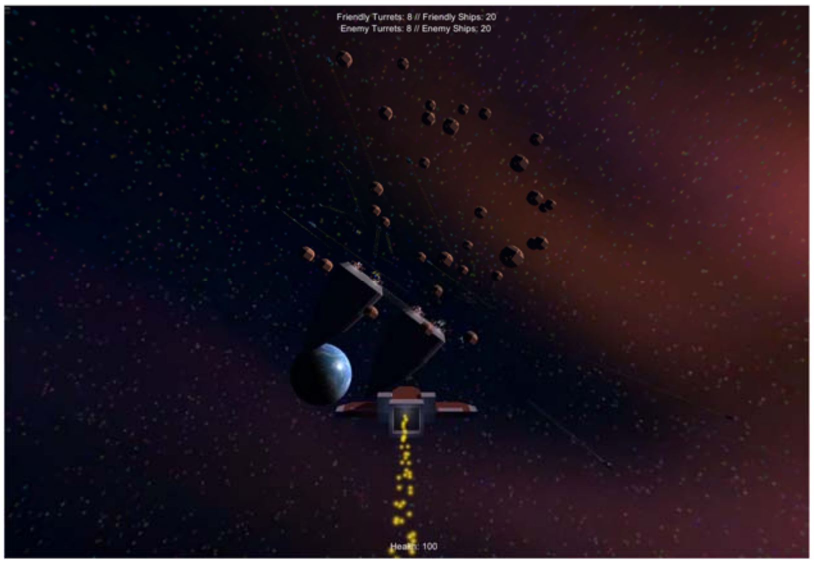
Mike Boyer & Shibo Ji
COSC 729 // Spring 2020

127.0.0.1
15937
To join a previously started game: (c) Connect

Multiplayer Implemented with: FORGE networking remastered

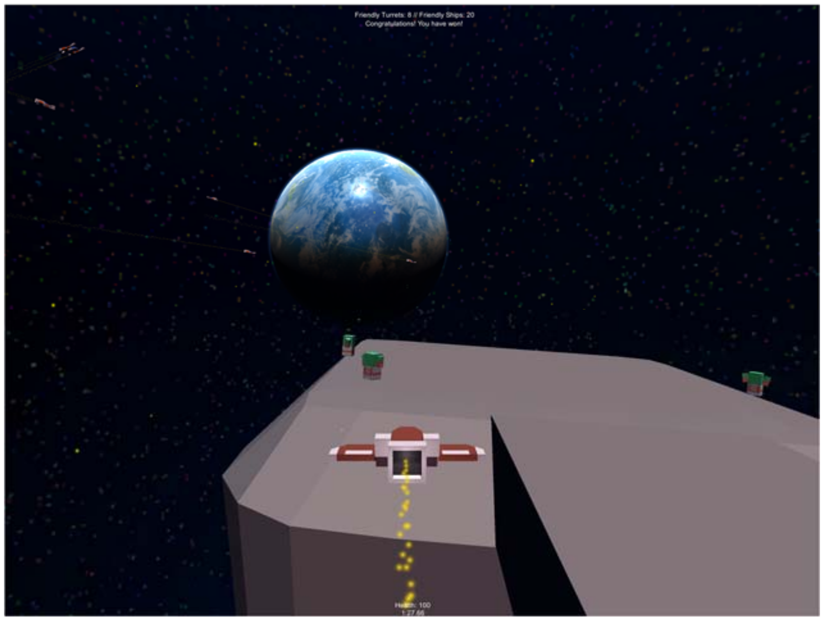
Figure 10 - Multiplayer Screen

As the scenario begins, players are spawned between the asteroid field and the allied base. The heads-up display will indicate it is scanning the current battle scene to determine the density of enemy and allied entities in the battlespace. After short period, the players will see twenty enemy ships, twenty allied ships, eight enemy turrets, and eight allied turrets. During this time, enemy ships will be flying through the asteroid field to begin their battle with allied forces. Upon reaching proximity sensor range of the enemy ships, they will begin attacking either the allied ships or the player, whichever enters their sensor range first. As this occurs, the space battle begins, and players may elect to continue the battle by retreating to the safety of their bases or press the attack onto the enemy base. A screenshot from the simulator at the beginning of training, which shows the heads-up display is in figure 11.


Friendly Turrets: 8 // Friendly Ships: 20
Enemy Turrets: 8 // Enemy Ships: 20
Health: 100
Figure 11 - Gameplay Scene with Starting Heads Up Display

Players are controlled as defined in the input section. Allies and enemies are controlled as shown in figure 7 and prefer collision avoidance to targeting and patrolling as defined in the animation section.

As the battle continues, the counts of enemy and allied forces are updated. If the total number of allied forces reaches zero, the counts will cease to update, and the heads-up display will show failure. If all enemies get defeated first, a message congratulating the players for winning appears. Figure 12 shows an example win condition message, with the game timer frozen, showing the player’s game time.


Friendly Turrets: 8 // Friendly Ships: 20
Congratulations! You have won!
Health: 100
1:27.66
Figure 12 - Gameplay Scene on Winning Conditions

During the battle, players may be affected by laser fire. The “health” indicator at the bottom of the heads-up display updates periodically to reflect player ship’s health. Should health reach zero, the player ship explodes, and their position resets to their starting spawn area. As this space battle can occur anywhere, the player should remain cautious during respawning. If enemy ships are in the nearby area, they will continue to pursue the newly spawned player.

All ships and turrets have the same starting health value. Additionally, all lasers, from turrets or ships, inflict the same amount of damage. Lasers will not affect asteroids, planetary objects, or bases (outside of the turrets). Only players can respawn, not allied or enemy forces.

In addition to lasers, planetary objects such as the asteroids or planets inflict a large amount of damage, instantaneously killing a ship colliding with them. Enemy and allied ship obstacle avoidance will mostly avoid some objects, but they can still be lured into an asteroid field to reduce enemy forces, should the player choose that strategy.

Conclusion/Future Recommendations:

The scenario built can be used as a multi-agent sandbox for space simulators. With minor modifications and testing, scenario parameters, such as number of ships, ship selection, and density of planetary objects could be defined in the multiplayer screen. Additionally, players could be allowed to select from a variety of ships or weapons.

This space simulator could also employ the use of machine learning agents, by adding camera-based inputs in addition to ship and turret proximity sensors. To implement, minor changes to ship control could be made in a similar fashion to examples at https://github.com/Unity-Technologies/ml-agents.

To support a more immersive experience, the game could be modified to support Virtual Reality hardware such as the HTC Vive or Google Cardboard. Should these updates be made, it is recommended to shift the view to a first-person view with a ship's dashboard, as opposed to the view as currently defined. With these minor modifications, an experience similar to what is provided by titles such as Elite: Dangerous, by Frontier Developments, which can be found at https://store.steampowered.com/app/359320/Elite_Dangerous/.

Overall, the multiplayer space simulator is a useful example of a multiplayer game in Unity. With minor modifications in Unity3D, additional features can be added to support reinforcement learning research or even virtual reality.