LATE TO THE STAGE

CS 307

Design Document

Team 2

Chen Kai Chuang

Garner Newton

Evan Dunning

Mitchell Augustin

Lenny Meng

Parker Lawrence

Index

Purpose

There are many times when you and your friends cannot physically meet to have fun. It could be that you have exhausted all nearby establishments and have “seen it all,” or it could be a sweeping global pandemic that discourages public gatherings. There is a demand for “lazy companionship”, a way to connect and enjoy each other's companionship without needing to schedule or plan for logistics.

The purpose of this project is developing a party game that can be played individually online while being in a voice call with friends. Existing party games like Kahoot tend to limit controls to cell phones. Existing cooperative games like those in Call of Duty tend to crowd the screen and limit the max number of players to ~ 4 max. Games like Jackbox have themes that are more geared towards board games, and games like Fall Guys tend to have little variety between the games.

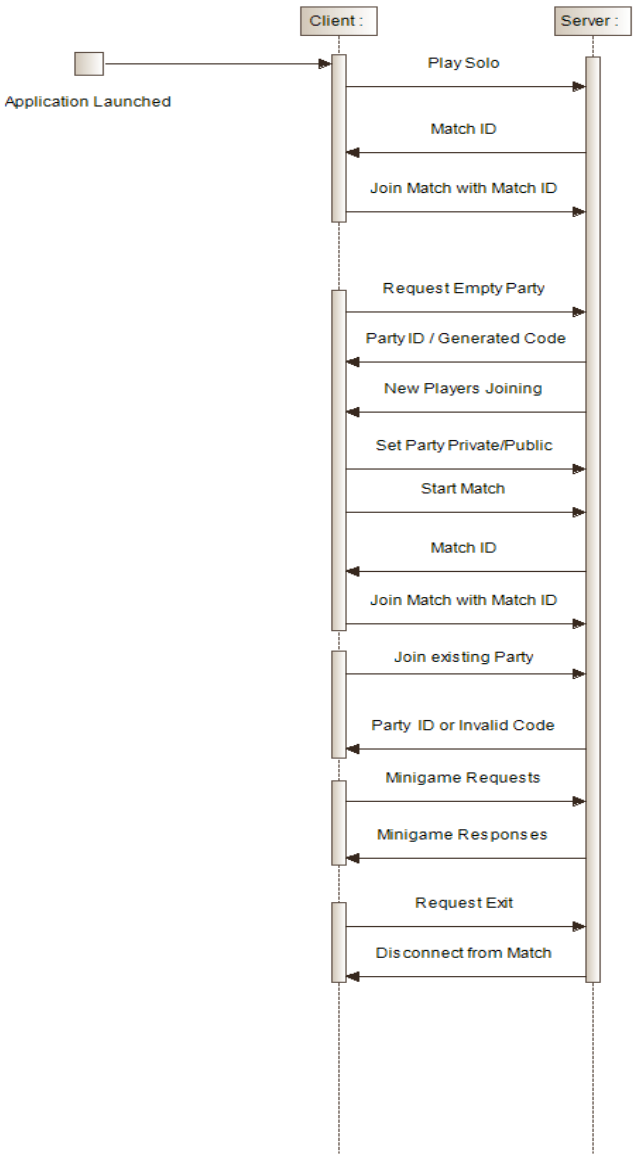
Our project would combine the variety of Jackbox Party Games and the scale of Fall Guys into one package. Our project aims to give everyone with a different game genre preference (RPGs, FPS, Racing) a shot at showing off their skills to their friends.

Design Outline

This project will be a game that utilizes the client-server architecture. By synchronizing all clients with the server, we can avoid inaccuracies that may arise from individual clients' own technical environments. One server will host multiple match sessions that each house 0...20 players, and it will also host the program that matchmakes players and parties together. Our client-server architecture will also function as a model-view-controller, but the view and the controller are packaged together.

1. Client
    a. The client will facilitate the view and the controller portion of the model-view-controller pattern.
    b. The client will receive raw data from the server such as player positions, current health, etc.
    c. The client will send raw data to the server such as player action, movement, and direction.
2. Server
    a. The server will facilitate the model portion of the model-view-controller pattern.
    b. The server will handle all raw data sent in by the clients and update its own internal match state.
    c. This match state will be considered the "gold standard" across all clients. All rules for each minigame will follow the internal match state stored in the server.
    d. The server will then propagate the new, updated data across to all clients.

Communications and requests from the client to the server will occur based on user action. If the user clicks on the main menu buttons, depending on the button, the client will send requests to the server. During the actual match, the client will be sending requests to the server continuously, and the server will be sending data to the client continuously. The server will send raw data to the client, which will then render, parse, and interpret the data. The client will also send raw data to the server, which will then be updated internally, and propagated to the rest of the clients in the match.



Design Issues

Non-Functional Issues

1. What game engine/development environment will we use?

    a. Unreal Engine

    b. Unity

    c. Godot

    d. Adobe Flash

    Choice: Godot

    Justification: Unreal Engine and Unity are very heavyweight tools, and not all members of the group have hardware at Purdue that can run them effectively. They are also more difficult to get started with, and have difficulties adapting them to work for two-dimensional games. In contrast, Godot is lightweight, and can easily run on any of our group members' computers. Additionally, Adobe Flash is discontinued and is being replaced by HTML5. Therefore, we chose Godot.

2. What VPS should we use?

    a. DigitalOcean

    b. AWS (Amazon Web Services)

    c. Google Cloud

    d. Personal Server Rack

    Choice: Digital Ocean

Justification: Choosing a hosting service is particularly important for a multiplayer game. We needed a solution that was cheap, simple to set up, and unlikely to experience downtime. It is also important for our chosen hosting service to be part of our continuous integration toolchain, so that we can automatically deploy to it by merging into a branch on GitHub. Our chosen hosting service should also be accessible by a static IP address and with low latency by anyone who wants to play our game. Although one of our group members owns a personal server rack, running it in the dorms is ill-advised because of the amount of power it draws, as well as the downtime and latency concerns. It would also be difficult to integrate it into a CI workflow. AWS (Amazon Web Services) and Google Cloud meet those requirements, but have pricing models tuned towards enterprise solutions, which convinced us to turn towards DigitalOcean, since we have prior experience with it. It only costs $5/month and has been brought into the continuous integration workflow quite nicely.

3. How should we acquire assets for the game?
    a. Public Domain (e.g., Kenney Assets)
    b. Make assets ourselves
    c. Commission new assets from an artist
    d. Purchase assets

    Choice: A mixture of public domain assets and making assets ourselves.

    Justification: Acquiring assets is a difficult step in a game development process. It is often deceptive, because even if one finds a large repository of public domain assets, it is still likely that there will be some smaller asset that is needed that will not be in said

repository. It is therefore a good idea to use public domain assets to supplement, rather than replace, handmade assets created by the development/design team. It is also important that we do not use public domain assets if we feel that we would not be able to create assets of our own in a matching art style, because if a time comes when a specific asset is needed that is not in the repository, we need to be able to create that asset ourselves if necessary. It is also inadvisable to purchase or commission assets for a hobbyist game such as this. Purchasing premade assets comes with the same pitfalls as using public domain assets, and commissioning assets from an artist is prohibitively expensive, even for just a single art piece, not to mention the deceptively large amount of art assets required to create a game.

Functional Issues

1. How will we choose between different perspectives for some of the minigames?
    a. Isometric
    b. 2-Dimensional
    c. 3-Dimensional

Choice: Some games will use an isometric perspective, and some will use a 2-dimensional perspective.

Justification: 3D rendering would likely increase the computational requirements for our game considerably and would also require more development time than we may have for this project due to the difficulty involved in 3D environment creation and physics handling. Therefore, we will not be building any 3D perspective-based minigames. Instead, some of our simpler minigames will use a 2D perspective, and others will use an

isometric perspective, since that can provide a near-3D quality experience without many of the drawbacks of 3D.

2. How should we eliminate players from minigames?
    a. Hard Time Limit
    b. Soft Time Limit
    c. First come first serve

Choice: All games will have a limit on the amount of time they are played. Some games, like the racing game, will have an explicit counter that counts down to zero, ending the game. Other games, like the platformer or the Battle Royale, will have barriers that push all players to a certain location, so that it is impossible that the game takes longer than a certain predetermined amount of time.

Justification: It is important for each minigame to be limited on time. The purpose of this game is to be fast-paced and entertaining for a large group of people, and it would run counter to our goal if there were players that were eliminated at the beginning of the match that had to wait long amounts of time to be put back into the game. We should, therefore, ensure that no one player or group of players could unintentionally or intentionally prolong one of the minigames in perpetuity. The problem, though, is that players do not, in general, enjoy playing with a timer hanging over their heads. Luckily, a solution has already been presented by Battle Royale games, in which a moving barrier forces players towards the center of the map, to force encounters between players on

disparate corners of the map, and to ensure that no player can prolong the game by finding a clever hiding spot or leaving their game idle.

3. Should players log in?
    d. Username/Password
    e. No password, just a username
    f. No username, No password

Choice: Players will not create accounts, but rather enter with a username that they select.

Justification: Keeping persistent data related to players increases the development profile of our project by a significant margin. Our current network architecture involves deploying the backend version of our game to a VPS, which players connect to in order to play our game. If we were to keep persistent data related those players, we would need to allow them to create accounts that they could log into, which means we would need to use a trusted third-party service to handle the login information. We would follow the software development adage, “Don’t roll your own cryptography”. Even in an application like a simple game, a disaster could occur if a malicious actor were able to determine users’ passwords, since they are often reused.

4. How many players should be in each game?
    a. 20
    b. 40
    c. 60

d. 80

e. 100

Choice: 20 Player Maximum

Justification: The number of characters in a game is a particularly important choice- having a lot of players in each game can be extremely rewarding for players, especially for a game following in the same archetype as Fall Guys. However, on the other side, the more players there are, the less of an impact each individual player has on the outcome of the game; players are less engaged when they cannot see their own names on the leaderboard. Also, we wanted to ensure that players could play our game when there are fewer other players online at the same time, without severely impacting the way the game plays. We settled on having a maximum of 20 players, which seemed like a nice tradeoff.

5. What powerups should be available during the racing game?

    a. No powerups.

    b. Just boost powerups.

    c. Boost powerups, missiles, traps, and heat-seeking projectiles.

Choice: Boost powerups, missiles, traps, and heat-seeking projectiles

Justification: Racing games derive value either from realistic, fast paced driving mechanics, or from the chaos induced from the presence of powerups. Since our game does not have a high degree of realism, it is not reasonable to expect a player to have fun when they are just doing laps on a plain, flat racetrack. Mario kart, another party game

from which we draw inspiration, not only has powerups, but also uses ‘rubber banding’ techniques that cause players to be more likely to be neck-and-neck, such as giving speed boosts to players in the back of the pack or giving slower players a higher chance of receiving better powerups. Powerups and rubber banding techniques serve to interrupt the long stretches of time that players would otherwise spend driving alone, which would be unsuitable for a party game. A driving game without these features would have little content from a player’s perspective, since drivers would tend to either all clump together, or spread out irreversibly along the track. Also, it is trivial for us to implement since many of the powerups for the driving game would already be implemented by necessity in the demolition derby game.

6. Should rubber banding be employed in the racing game to encourage competition?
    a. Yes, rubber banding should be used.
    b. No, rubber banding should not be used.

Choice: No, rubber banding should not be used

Justification: Rubber banding is an important part of many party games. It is implemented to solve a paradox relating player skill to entertainment. In most games, the more skilled a player is, the faster they are able to continue through game content, and therefore they are consuming content at a faster rate, which is more entertaining. In fact, the motivating principle that motivates a player to become better at a game is that playing at a higher level of skill is more entertaining than playing poorly. In racing games, however, skilled players can drive away from all other players, which means that they are

not interacted with for the entire race. They are not threatened by other cars passing them, or to be hit by powerups launched from other cars. This results in boring play. However, due to the reduced amount of time that our minigames take, it is unlikely that a skilled player would be able to get far away from most of the other players. Also, projectile weapons launched from other players would be able to hit players even if they have pulled in front of other players, especially if they are given a heat-seeking projectile powerup.

7. Should racing maps have stage hazards?
    a. Yes
    b. No

Choice: Just boost pads and static hazards, like fences off-road.

Justification: Stage hazards are a staple of Mario Kart games. For example, when racing on an arena, players must take care to not be crushed under the weight of a block that drops down to deliberately crush them. These add visual interest to the maps that players race on and makes it extremely relevant which maps have been chosen for the race. They are often accompanied by boost pads, which speed the player up when they drive over them. These reward the player for maneuvering over to the right spot on the track and test their ability to handle the influx of speed granted by the pads. Boost pads are relevant to our racing game since it adds something for racers to think about when they are driving on the stretches of road between powerups and difficult turns. Static hazards are also relevant for our game. These are the hazards that lie outside the track and incentivize

players from driving off-road. These include fences or bottomless pits, both of which are easy for us to implement. Dynamic hazards such as blocks that deliberately attempt to crush the player take much more care to implement and are therefore outside the scope of this game type.

8. What guns should be available during the Battle Royale game?
    a. Only one projectile weapon that every player shares.
    b. Every weapon has ammo, and players can hold some number of weapons determined by inventory size.
    c. Alternate weapons are thought of as temporary powerups.

Choice: Alternate weapons are thought of as temporary powerups- they take up no inventory space and are used up after a fixed number of shots. Only one powerup can be held at a time, and once that powerup is spent, the player automatically drops back to their default weapon.

Justification: Battle Royale games offer an interesting choice to players; players may choose to drop in more disparate areas of the map where less action lies, and they face less danger, or they may choose to drop where lucrative weapons and loot are. Pursuing those weapons gives the player an immense advantage over players that do not find better weapons, but at the cost of heightened danger when acquiring those weapons. This mechanic is core to the gameplay of a Battle Royale game. However, our minigames are designed to elapse in a short amount of time, and it is not worth it to have the player learn to use any sort of inventory system to store weapons, or to have to prioritize certain ammo types over others. Even if players can quickly get comfortable with those systems,

the time spent dealing with the player's inventory or weapon slots would occupy an inappropriate percentage of the minigame's short playtime. We therefore propose that additional weapons in this game type should be thought of as powerups. Each player will have a gun that will serve its purpose adequately, but they may also collect another weapon that will replace their gun until it is used up after a fixed number of slots. The powerup weapons will have effects like making the player's shots into a laser beam or three shots that separate diagonally.

9. How should we handle isometric animation?
    a. Render 3 dimensional assets
    b. Use pre-rendered rotations of 3 dimensional assets

Choice: Use a blender script to create still images for 64 rotations of a 3-dimensional asset. The engine will switch the images to fit with player camera perspective.

Justification: Since the games are fundamentally 2 dimensional, by using images rather than 3 dimensional renders, it allows us to treat these seemingly 3 dimensional objects as 2 dimensional in the game engine. This simplifies development as 3-dimensional physics are significantly harder to deal with than 2-dimensional physics.

Design Details

Main Menu

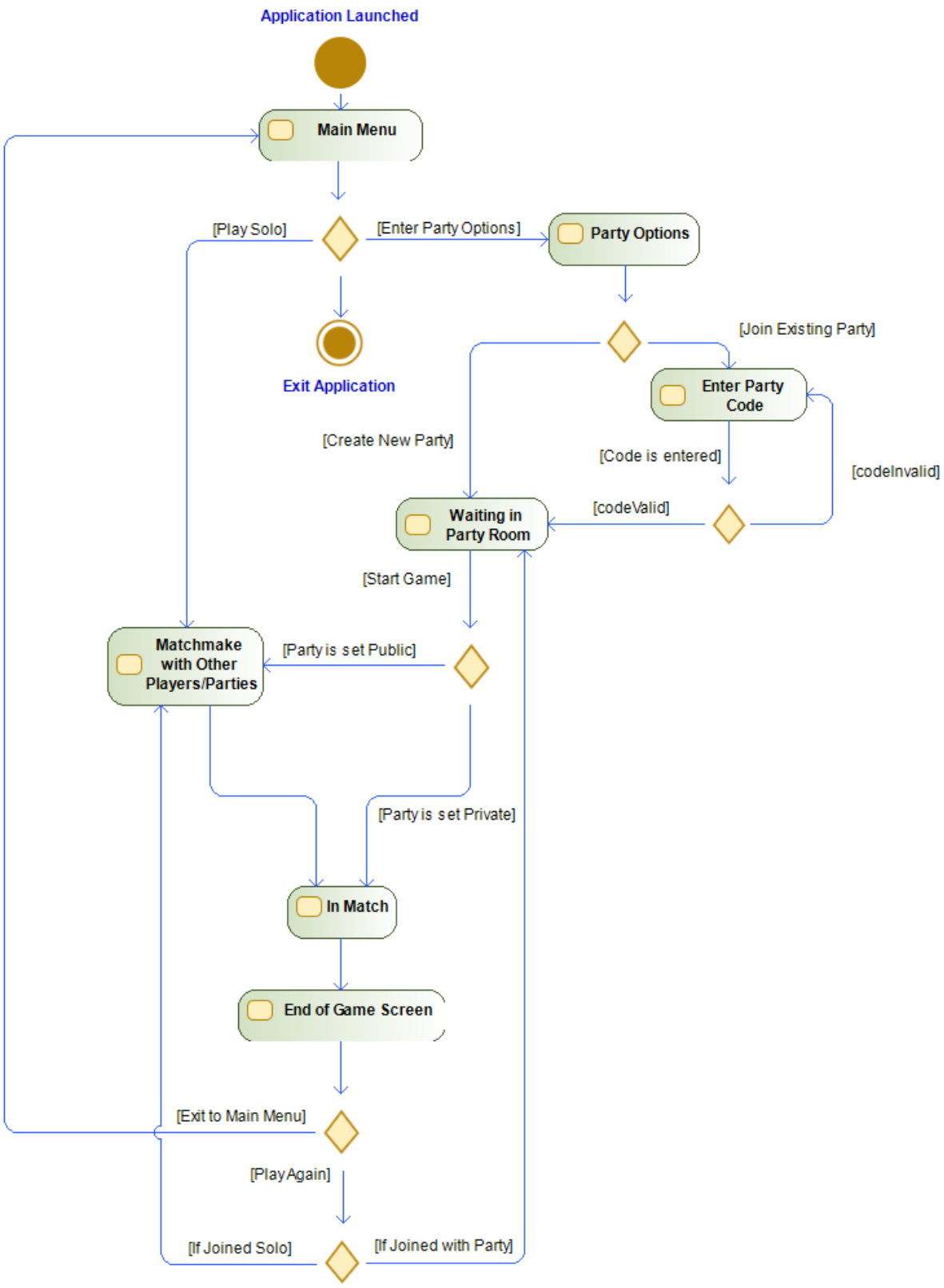
After launching the application, each user will be presented with a login screen.

If the user is already logged in, they will be presented with two options:

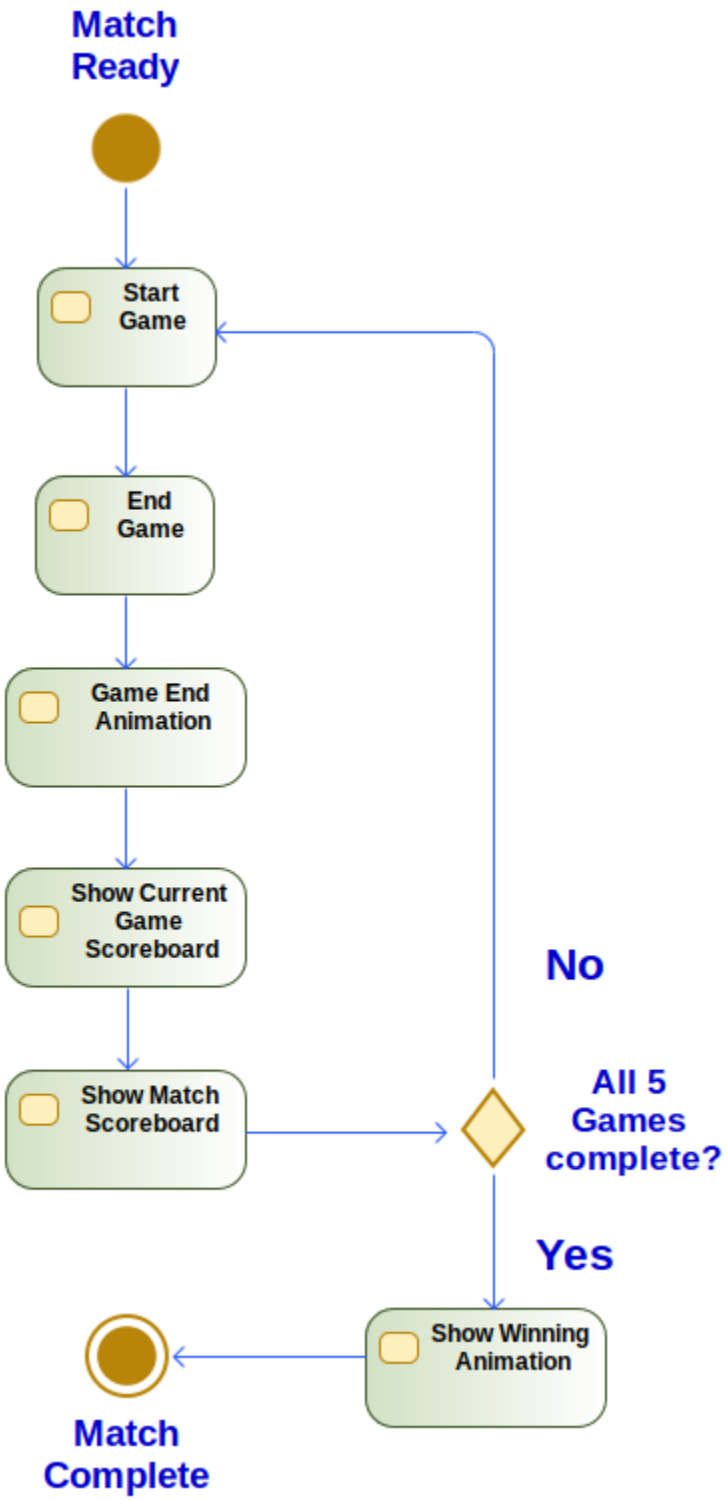
- Party Options
    - Upon selection, the client will present a new screen that has all the options for party creation and joining a party.
- Play Solo
    - Upon selection, the client will treat the player as a party of one and immediately start matchmaking into the game.

Party Options

- Create a Party
    - Upon selection, the server will generate a new party and add the player to it. Then, the host user will be presented with a party code, a visual of how many other players have joined the party, and a “start” button.
- Join a Party
    - Upon selection, the user will be presented with a text box, where they will input their party code (from the host).
    - If the code is valid, they will be presented with a visual of how many other players have joined the party, and a message indicating that they are waiting on the host to start the match.


Application Launched
Main Menu
[Play Solo]
[Enter Party Options]
Party Options
Exit Application
[Join Existing Party]
Enter Party Code
[Create New Party]
[Code is entered]
[codeInvalid]
[codeValid]
Waiting in Party Room
[Start Game]
Matchmake with Other Players/Parties
[Party is set Public]
[Party is set Private]
In Match
End of Game Screen
[Exit to Main Menu]
[Play Again]
[If Joined Solo]
[If Joined with Party]

Matches

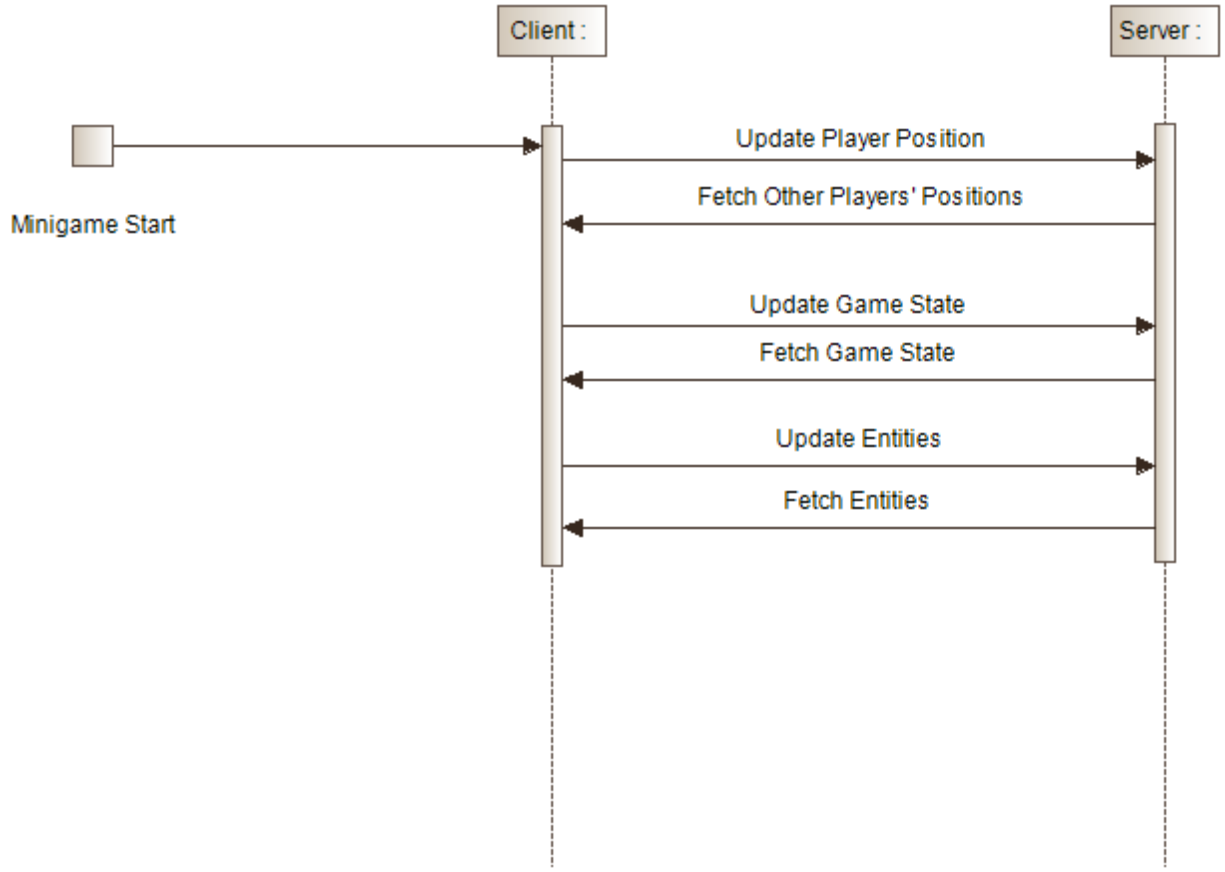

Match Ready
Start Game
End Game
Game End Animation
Show Current Game Scoreboard
Show Match Scoreboard
No
All 5 Games complete?
Yes
Show Winning Animation
Match Complete

Matches consist of a conglomerate of minigames, the order of which will be randomized. No minigame will appear twice in the queue. The matches will adhere to the following set of rules:

- Each match will consist of 20 players.
- Every player will play 5 minigames.
- The match will continue until the last minigame is complete.
    - Players who finish early will be shown a spectator view of the other players.
    - Time cutoffs will have to be determined on a per-game basis to prevent players from waiting in spectator mode for too long.
    - After each minigame, a leaderboard will appear for both the current minigame and the overall match score, showing the top players in both.
        - This will play an animation that shows the movement of your current rankings.
- A transitional scene appears after the leaderboard appears to show the next minigame and possibly the map/theme for that minigame.
- Once a minigame ends, the next will begin for all players after the server places all players in a new minigame.
- Once the final minigame is complete, the final scoreboard will be displayed, and the match will end after a brief animation showing the winner.

Minigames

To minimize the effects of latency and technological advantages/disadvantages, the server will need to host a standard perception of what it believes the game state currently is. This will include every aspect of information such as player statistics, entity information/properties, player positions, and much more. All clients connected to the server and in the game would try and match the server's game state. This ensures an "even" playing field. The following sequence diagram generalizes the communications between the client and the server during a minigame:



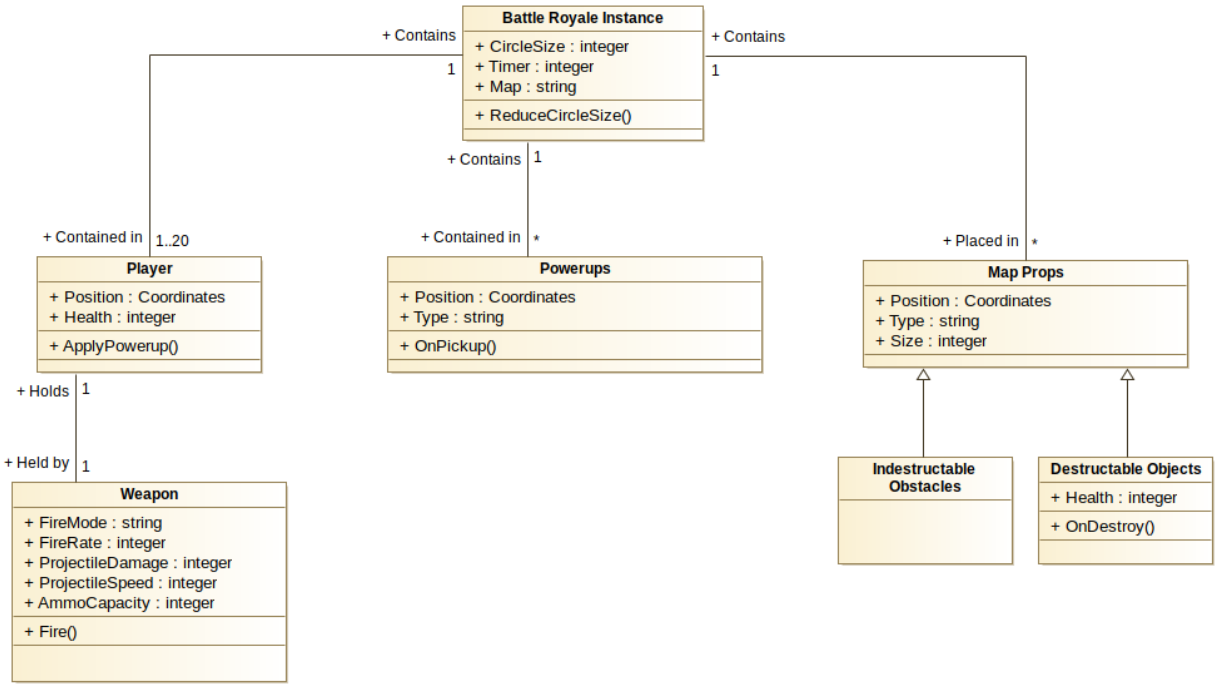
Depending on the specific minigame, the types of data transmitted can change. For example, in Battle Royale, we would need to transmit weapon types and properties, whereas for racing, we would not need to transmit weapon types and properties, but possibly car properties.

Currently, we have 6 minigames in mind to comprise of the pool of games the matches will be choosing from. We can add more minigames if we run out during the sprints. The minigames are as follows:

- Battle Royale
- Platform Elimination
- Bomb Elimination
- Platformer Race
- Racing Game
- Demolition Derby

More specific information about these minigames can be found below. The actual names for these minigames will be finalized later in the development process.

Battle Royale



In this minigame, every player is dropped in some location on the map and aims to eliminate all the other players in the group. Each player will have a weapon, which will either be a default weapon or a powerup weapon. Each weapon will have their own stats, which are the fire rate, fire mode, etc. Players can fire their weapon, which releases a projectile. If the projectile collides with an obstacle or player, it will disappear and deal damage if it is not an indestructible obstacle.

The world will contain various powerups that will enhance player attributes, such as movement speed and damage output. These powerups will be placed throughout the map as entities that will be consumed when the player is within range of their physical component on the map. The world will contain multiple distinct types of guns. The type of gun that a player spawns with, if any, will be determined randomly, and other guns will be randomly spawned throughout the world for

players to pick up. A gun will be “picked up” when the player’s position in the map is within range of the gun entity’s position, like powerup entities.

Guns must be shootable. This would involve a muzzle flare asset, and a projectile asset. Both of those assets would need to be synchronized with all other clients. Distinct types of guns could use different bullet types. Depending on whether we want to diversify weapons that much, we could have a rocket asset for rocket launchers, but then we would also need an explosion asset.

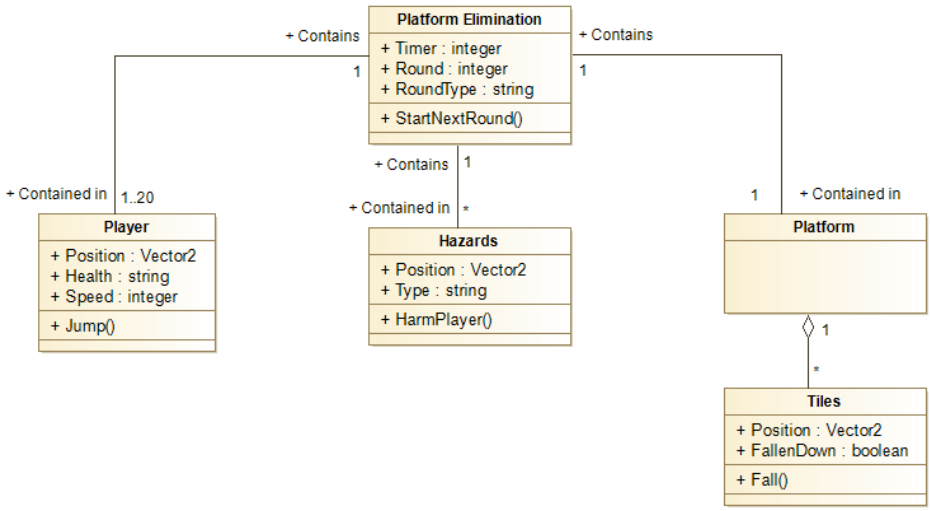
Props such as trees and buildings will be placed throughout the map for player protection.

We will create a mechanism to determine the player’s starting positions. This will likely be implemented with a dropping system like some established Battle Royale games, where players are flown over the map and required to select a landing location. There will need to be protections that stop the player from choosing invalid locations.

There will be a shrinking circle of death on the map. If a player touches it, they will instantly die.

Players will have health and a death animation. The server will keep track of which players have which amounts of health. The server will handle collisions, which can easily be implemented with Godot - this is done by running the server as a headless copy of a client, where all the objects’ positions are synchronized with the clients. The server then runs collision detection so that ‘ground truth’ can be established. This is also how the position and number of all assets is synchronized across games.

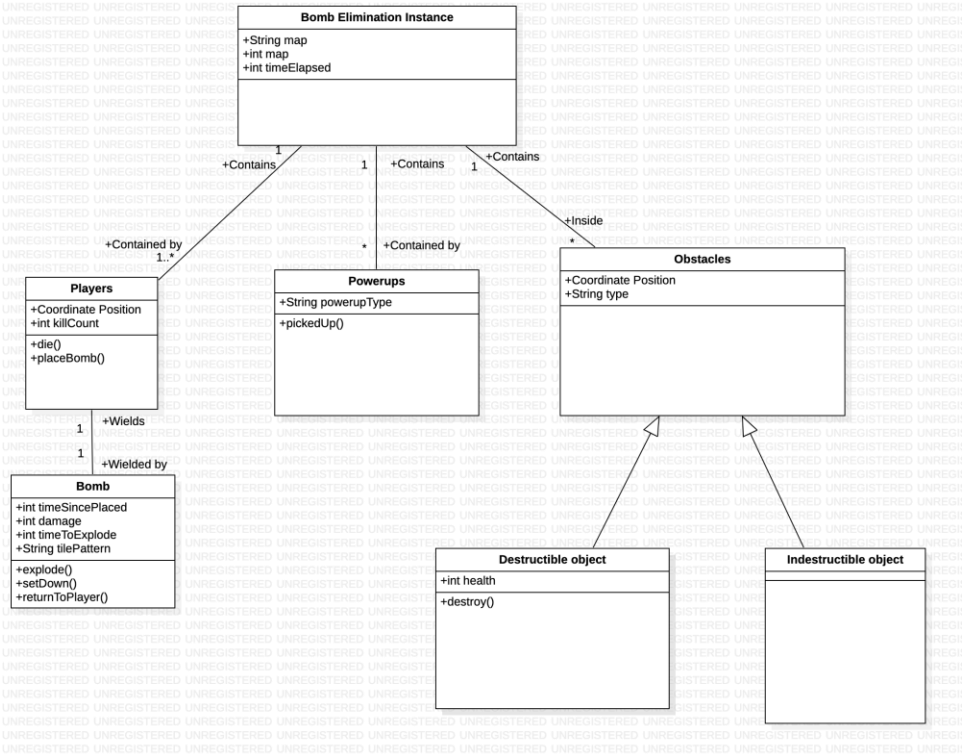
Platform Elimination



This game will require a platform asset and a way to make the players 'fall.' We will also have hazards that knock the players off. These will be lasers, sweeping arms, or another similarly designed asset.

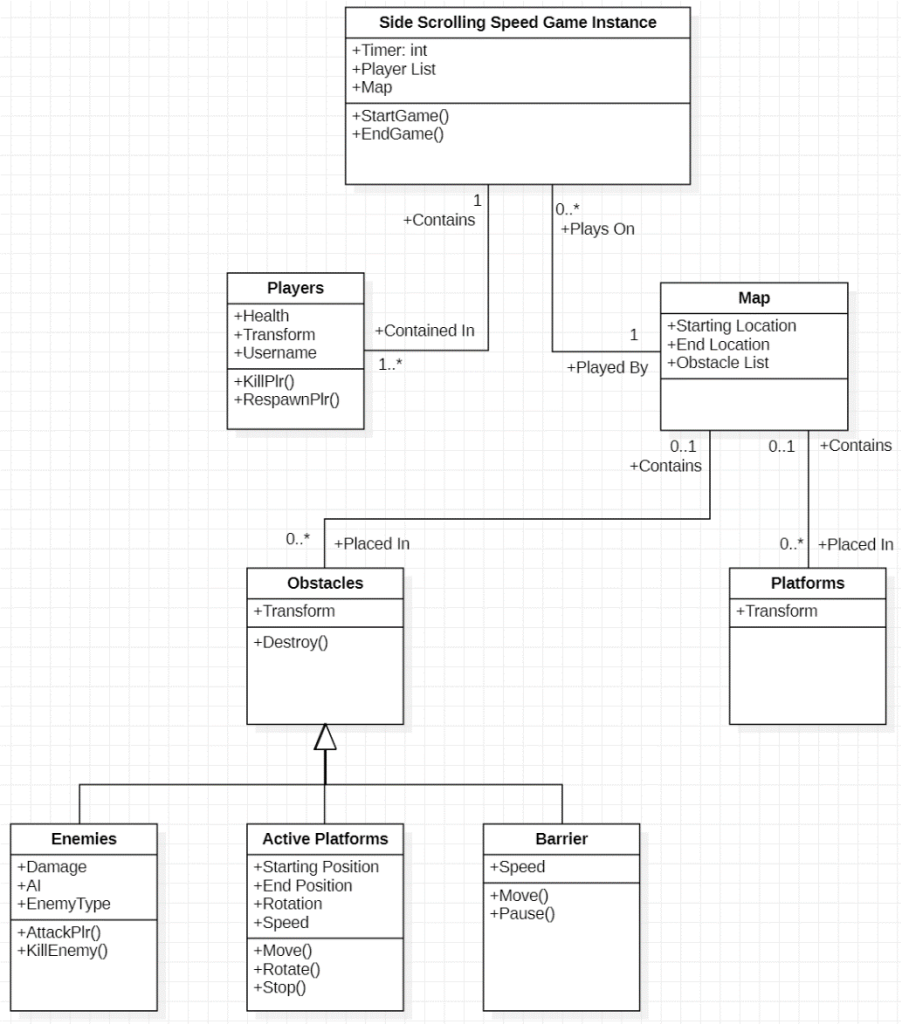
In a slightly different version of this minigame, the hazard would be that the screen is divided into four regions, and a confusing or impossible question is asked, and the players must navigate to the region of the map that contains the correct answer. All other regions will drop out, and those players will fall and be eliminated.

Bomb Elimination



The bomb elimination game will put players onto random positions in the map. These positions will be filled with obstacles that prevent player vision of areas beyond it. Some obstacles will be destructible, with an amount of health and some will be indestructible. Players will each have a bomb that they can place down, which will explode after a set amount of time. When the bomb explodes, it will deal damage to surrounding obstacles and instantly kill players. The goal of this minigame is to kill as many other players as possible and to survive longer. Powerups will be randomly located around the map, which will do things such as increase the player's vision of the area or enhance their bombs in certain ways. The game will end once there is only one player left or a certain amount of time has passed. Players will receive points for surviving longer and killing other players.

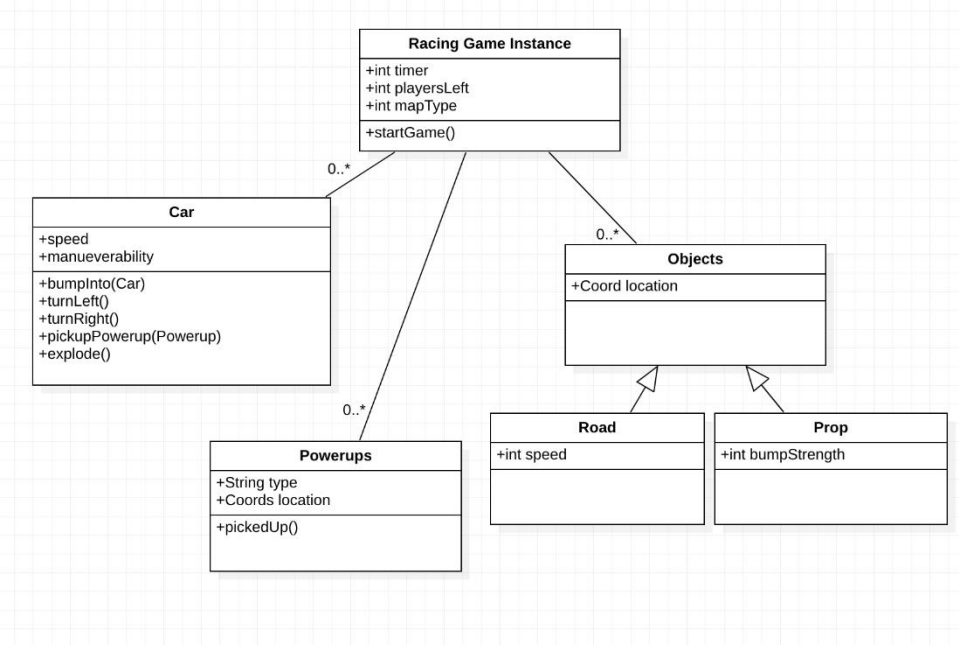
Platformer Race



In this game type, the players will be in a side scrolling map in a race to the finish. This is a platformer, so there will be platforms that the players can jump on, bottomless pits they can fall into, and obstacles that will either kill or impede the player upon impact. There will be moving and/or rotating platforms. There will also be simple enemies throughout each map that walk side to side and deal damage to the player when touched.

To constrain this game type from taking too long, we will have a barrier sweep the map from the left, forcing players to continue in an equivalent manner as the Battle Royale type.

Racing Game



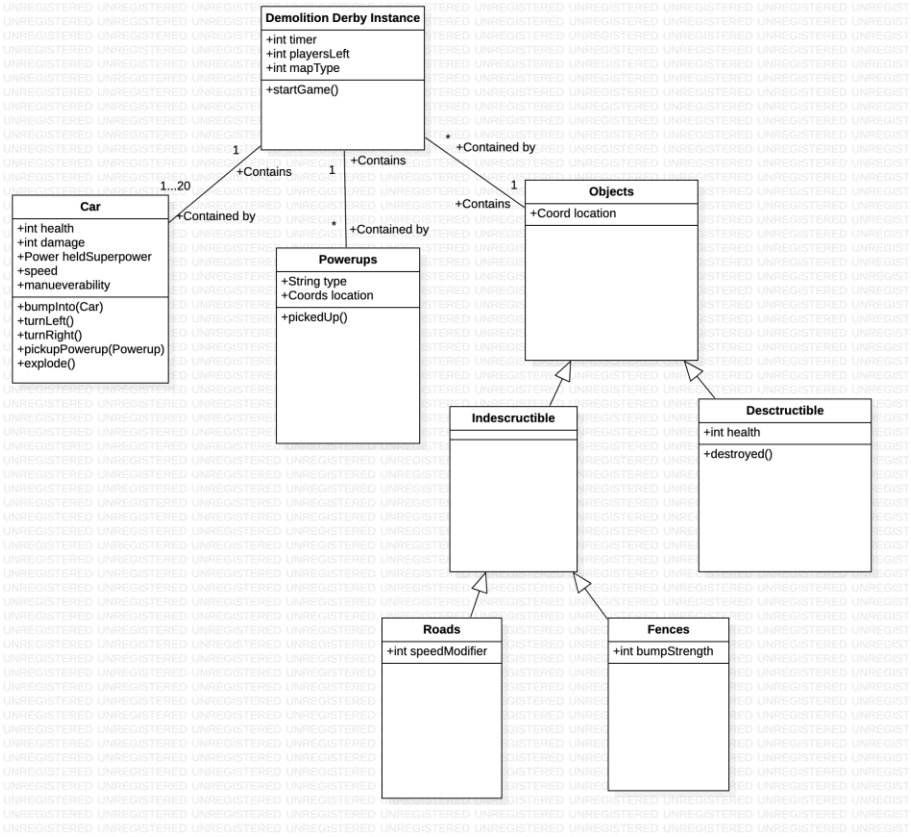
We will need to synchronize player position and velocity with the server. We will also need collision detection between the cars and between cars and obstacles, which will be handled by the server. The server will also need to ensure that player velocity makes sense.

The primary objective of the racing game will be for the players to drive through a series of checkpoints until reaching the finish line. The final player scores for this minigame will be determined by the order in which players pass through the finish line, with penalties applied for missed checkpoints.

Powerups, which will provide speed boosts and other player enhancements, will also be included throughout the track. Powerup locations will be synchronized with the server and handled in the same manner as the other minigames. The server will also replenish them throughout the duration of the game.

Some powerups will have more detailed behavior than stat buffs. Heat-seeking projectiles will need pathfinding behavior, which will be handled server-side so that all clients agree on the projectile's behavior. We will also implement various simple projectiles, such as traps and missiles that are projected toward a specific location and remain stationary until their activation. We will also design various maps for the players to race on. A sort node may be used to allow for features like tunnels. Included in the maps are also objects that focus the player on the road. This would include objects that they can collide with that keep them on the track or an ocean they could fall into.

Demolition Derby



In this game, player cars are placed into an arena of sorts and seek to destroy each other's cars. There will be powerups that are available to them- these will be the same powerups as in the

racing game. These include traps, missile projectiles, and heat-seeking projectiles. The main two differences would be the addition of health bars, and the transformation of the racetrack maps into large arena maps.

The powerups, despite being identical to the ones in the racing game, will perform a different function than in the racing game. Rather than being tools to delay other cars, so that they can be passed, the powerups in the demolition derby will be used to reduce the health bars of the other cars to zero. Once a car's health bar has been reduced to zero, it will play a 'death' animation, and be removed from play. Once there is only one car remaining, it will be declared the winner. Cars that are low in health will play smoking or "on fire" animations.

We will add directional collision detection - when two cars collide with each other, we will need to determine which car was smashed into, and which car did the smashing. If two cars rammed into each other head on, and both were going fast, then both cars will be damaged in the collision. Otherwise, the car that demonstrated less intention will experience a loss in health, while the more aggressive car will bounce away harmlessly.