City Goers: An Exploration Into Creating Seemingly Intelligent A.I. Systems

Matthew Brooke

City Goers: An Exploration Into Creating Seemingly Intelligent A.I. Systems

An Undergraduate Honors Thesis

in the

Department of Computer Science and Computer Engineering

College of Engineering

University of Arkansas

Fayetteville, Arkansas

By

Matthew Brooke

Table of Contents

1 Abstract

Artificial Intelligence systems have come a long way over the years. One particular application of A.I. is its incorporation in video games. A key goal of creating an A.I. system in a video game is to convey a level of intellect to the player. During playtests for *Halo: Combat Evolved*, the developers at Bungie noticed that players deemed tougher enemies as more intelligent than weaker ones, despite the fact that there were no differences in behavior in the enemies. The tougher enemies provided a greater illusion of intelligence to the players. Inspired by this, I set out to create a project that captures the essence of this idea. An A.I. system was designed in the Unity game engine using the graph algorithm Dijkstra's Single-Source Shortest Path that would direct a car around a small city-like area. More cars were then added to the level in order to set up the possibility for collisions. The measure of the cars' intelligence is their ability to avoid these collisions. New systems and techniques were developed that would help the cars avoid collisions without making major changes to their fundamental behaviors, thus increasing their measure of intelligence.

2 Introduction

2.1 Background

Artificial Intelligence has quickly become a major concept in Computer Science, and many people today interact with it on some level. One use of A.I. that has always intrigued me was its use in entertainment media, specifically video games. One of the first commercially successful arcade games, *Pong,* tasked two players with competing head-to-head in a tennis-like simulation. The player who achieved the maximum score first would win the match ("Pong," 2020).

Although *Pong's* primary appeal was playing against another player, the option was available for individual players to compete against a computer-controlled opponent in the event that another player is not present. This rudimentary A.I. moved the opposing paddle up and down and followed the ball to hit it back to the human player. Even this early into the development of video games, developers were already attempting to account for ways that players could play games by themselves, despite the game being designed for two players.

In more recent video games, A.I. systems have been developed much further. Computer-controlled enemies appear to be communicating with each other, forming battle strategies against the player, and providing a genuine challenge in the games that incorporate them. An A.I. designed by OpenAI was even able to beat a team of professional players in the game *Dota 2* (OpenAI, 2019).

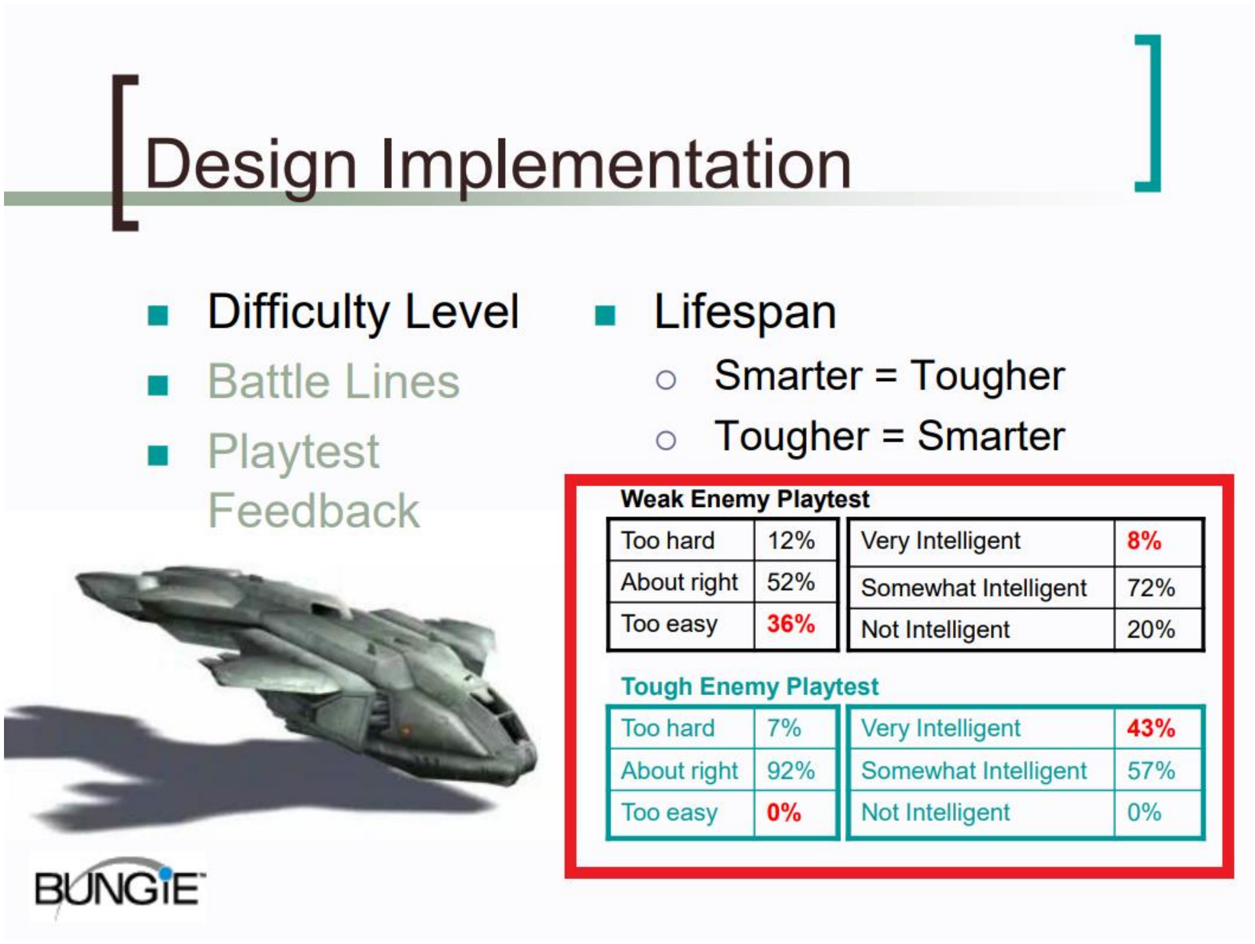
2.2 Motivation

OpenAI's system is designed to participate in a multiplayer match of *Dota 2* as if it was another human player, or set of players. The purpose of the A.I. changes when the goal is not to take control of an agent as if it was another player, but instead function as a unique entity, often with abilities not available to the player. In the 1993 game *Doom*, enemy A.I. can behave very differently from the player while also being distinct among the other enemies, such as the Imp that can throw projectiles at the player, and the Demon, commonly referred to as the Pinky, that charges the player in order to deal up close damage (Sullivan, 2015). In a design scenario such as this, it is often desired to make the A.I. controlled enemies appear intelligent to the player to provide a more engaging and immersive experience. Upon first inspection, it may seem like the best course of action would be to add more intricate decision-making capabilities and behaviors to the A.I. to increase its intelligence, but there may be another way to achieve a similar effect.

(From left to right) the Imp and the Demon (Pinky) from *Doom* (1993). These two A.I. enemies both behave very differently from the player and each other. Imp: (*Reviving lostsoul, Imp, Zombieman AND Shotgunguy*). Pinky: (*Demon (Doom)*).

At the Game Developer Conference (GDC) 2002, developers from the software company Bungie gave a presentation about their experience developing the Xbox launch title *Halo: Combat Evolved*, the first entry in what would become a well-known franchise of First-Person Shooter video games. During the development of *Halo*, the developers ran playtests to help balance the game and fix bugs before release. In one particular testing session, they had two groups of players playing two different versions of the game. The first group was given a version of the game where the enemies had low health values and dealt lower damage to the player while the second group was given a version of the game where the enemies had higher health and dealt more damage to the player (Butcher

& Griesemer, 2002). The players were then surveyed about their experience and the responses were compiled .



Design Implementation

- Difficulty Level
- Battle Lines
- Playtest Feedback

- Lifespan
  - Smarter = Tougher
  - Tougher = Smarter

Weak Enemy Playtest

Too hard	12%	Very Intelligent	8%
About right	52%	Somewhat Intelligent	72%
Too easy	36%	Not Intelligent	20%

Tough Enemy Playtest

Too hard	7%	Very Intelligent	43%
About right	92%	Somewhat Intelligent	57%
Too easy	0%	Not Intelligent	0%

BUNGIE

A key slide from Bungie’s presentation with key information highlighted (Butcher & Griesemer, 2002).

The majority of group A (72%) reported that they felt the enemy A.I. was “somewhat intelligent”, while only 8% of players felt that the A.I. was “very intelligent”. The remaining 20% of the group described the A.I. as “not intelligent”. Group B, on the other hand, had a much more favourable breakdown. 57% of these players felt that the A.I. was “somewhat intelligent”, while the number of players who thought the A.I. was “very intelligent” shot up to 43%. The number of players who thought that the A.I. was “not intelligent” plummeted all the way to 0%. By simply increasing the amount of health the enemies have and the damage they dealt to the player, Bungie were able to improve player perception of the A.I. without having to change anything to do with their actual behaviors and decision-making logic.

The question that I have set out to investigate with this project was inspired by this playtest: what approaches can be used to create an A.I. system that appears to be exhibiting behaviors that it does not actually possess, thus making it look as though it is smarter than it truly is?

2.3 Approach

Rather than creating an environment that would test the perception of intelligence that playtesters felt, I approached this topic with the framework of distributed intelligence, i.e. decision making and logical choices are being handled by many sources, with each source containing only a fraction of the whole intelligence. The same core idea, that being creating an A.I. agent that appears more intelligent than it may actually be, still applies here. The behaviors and apparent decision making of the created agent/agents would more so derive from external systems working in conjunction with the agent’s own behaviors rather than all being handled by the agent itself, thus creating an agent that appears as though it is accomplishing more than it actually is doing. To do this, I decided to choose an area of A.I. relating to video games, design a functional agent (an entity within the world that makes decisions of its own accord based on elements of the current state of the program), and then iterate on it in other ways to build up the “intelligence level,” a factor or factors that demonstrate the A.I.’s intelligence, or the illusion of intelligence in the case of this project. One area that is key to most

A.I. in video games is pathfinding, and this can be done in many different ways, including basic graph algorithms. These would be implemented in a game-like scenario to direct some agents around a level. Then additions would be made to build up the A.I. 's intelligence level without modifying this core pathfinding.

The chosen development environment for this project was the Unity game engine. Unity is a powerful tool that features many robust systems that aid in game development, such as 3D rendering and an advanced physics system. Programming in Unity is doe via C# scripting, and code is run directly on objects in the game environment.

3 Designing The Scenario

3.1 Designing a Pathfinding Environment

The selected pathfinding scenario in which the graph algorithms would be implemented is that of cars driving around city streets. Road networks are essentially graphs, with the streets being links between the intersections which represent the nodes. In this context, the intelligence level of the agents (cars driving around the roads) can be observed based on their ability to avoid collisions.

The cars would need to be able to drive in perpetuity without colliding with one another or slipping off of the road. At a basic level, the cars would follow these steps for each drive they make:

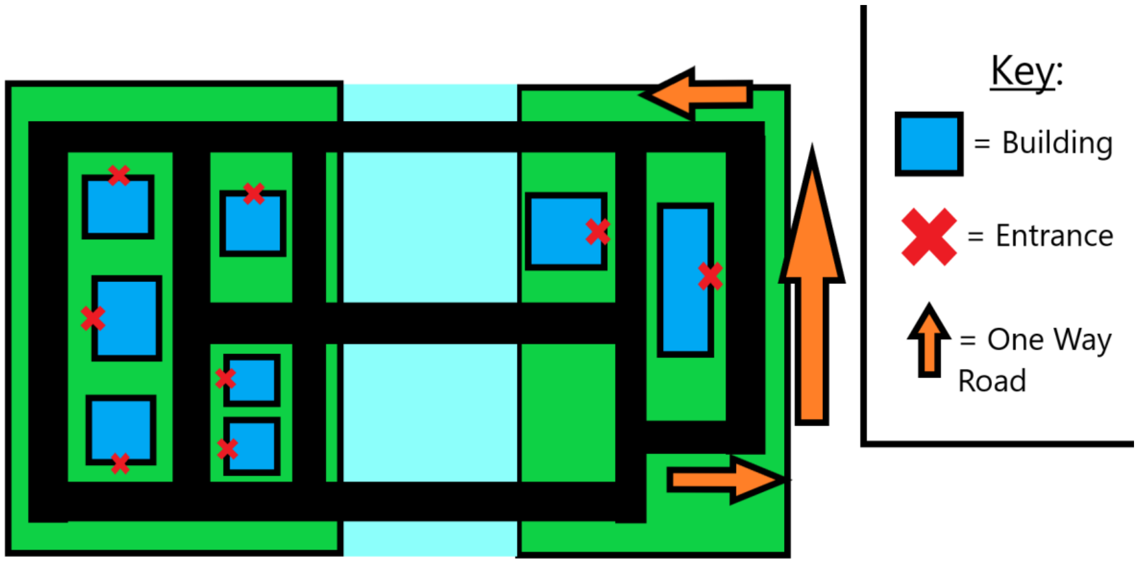
1) Acquire a goal to drive towards
2) Calculate a path between the current position and the goal
3) Traverse the graph until goal is reached

This would allow the cars to drive continuously given that they do not interact with each other in any way. In reality, cars collide on the streets at a rate of nearly 6 million accidents per year in the United States (Driving-Tests.org).

In order to add to the apparent intelligence of the cars, systems would need to be put in place to prevent collisions as much as possible, but the level design needed to ensure that collision scenarios existed to demonstrate these systems in real time.

3.2 Designing the Level

Before building the scene in Unity, a design for the level was drafted to lay out key aspects of the environment.



The design of the environment with a key.

A few key ideas this design captures are:

1) A variety of goals the cars can drive to
2) Roads connected by many intersections
3) Different road types

The goals are spread out between both islands, although they are more prominent on the left island. Eight main intersections connect the roads, leaving plenty of opportunities for cars to collide while driving. The far-right road is specified as a one-way road, so cars can only enter from the bottom and exit from the top when going to the goal on that road.

With a solid foundation for creating the environment and an approach for how to create the cars, work shifted into Unity to begin development. The overall development timeline followed these steps:

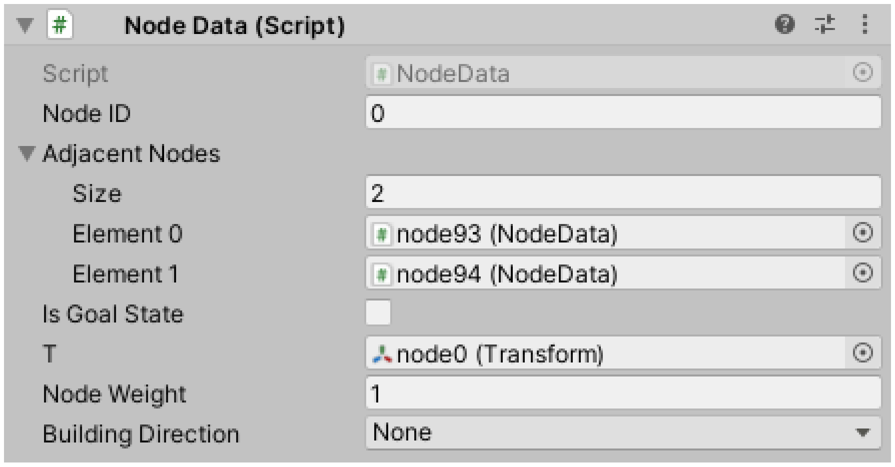
1) Build a prototype to test main ideas
2) Build the level based on the design, make changes as needed/desired
3) Get a single car driving round successfully
4) Create systems to control traffic flow and avoid collisions
5) Implement those systems into the level
6) Run tests, debug, run more tests

4 Early Development

4.1 Creating Functioning Pathfinding

The core of this A.I. is the ability for it to find a path between two points and traverse said path. Unity provides a pathfinding solution in the form of a navigation mesh component. This tool generates an area of space that agents are able to freely move through, however this system was not used. Instead, a manual solution was created. By creating a manual solution to the pathfinding problem, full control over the system and its behaviors was maintained, whereas the use of Unity’s predefined systems could result in undesired behaviors due to its universal implementation.

To create a pathfinding environment utilizing graph algorithms, a structure for the graph itself needs to be developed. Graphs can be represented in data in a few different ways, such as a set of adjacency lists or an adjacency matrix. To implement a graph structure in Unity, a new data type was created. This new data type, called NodeData, would contain all of the important information needed to set up a graph in the game environment without. Unity's navigation mesh works in a similar way, with each point on the mesh being a node in the respective graph, but this manual implementation provides a finer degree of control over node placement and key information that modifies agent behaviors.


Node Data (Script)

Script	NodeData
Node ID	0
Adjacent Nodes	
Size	2
Element 0	node93 (NodeData)
Element 1	node94 (NodeData)
Is Goal State	☐
T	node0 (Transform)
Node Weight	1
Building Direction	None

The NodeData Inspector as displayed in Unity.

The way NodeData works is very effective for setting up a graph to work with a variety of different algorithms. Essentially, it is a container for a few different variables relevant to creating a graph. These key variables are:

- nodeID: The unique ID number associated with this node.
- adjacentNodes: An array of NodeData objects that contains every node that can be reached from this node.
- isGoalState: A boolean that indicates if this node is considered a possible goal on the graph (more on this later).

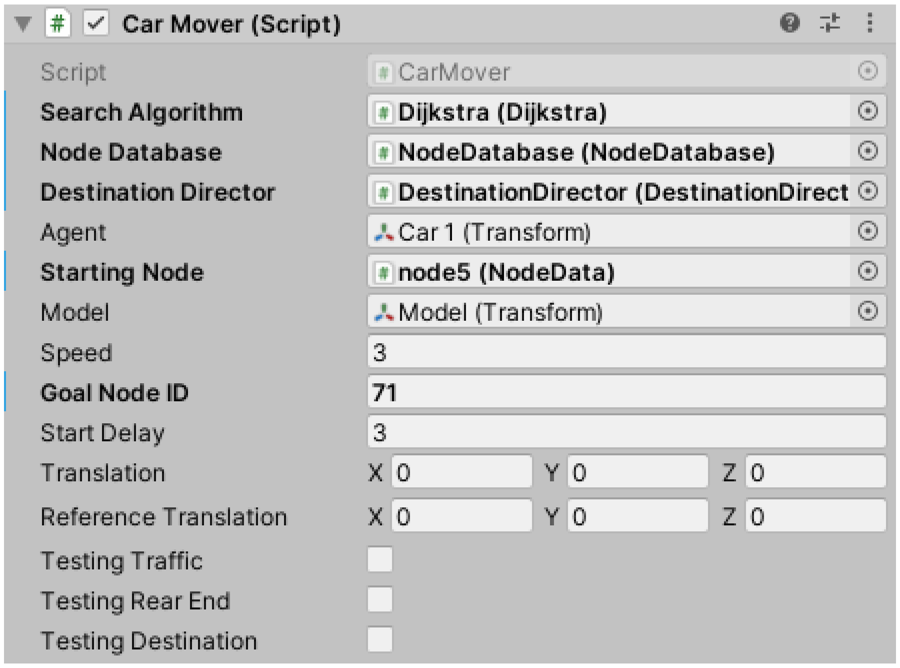
- t: A reference to the node's Transform in Unity. This is used for getting position data that is provided to the cars.
- nodeWeight: An integer that represents the cost of traversing this node. In terms of this implementation, the cost is arbitrarily assigned a value of 1 for each node. These weights can be adjusted to reflect different characteristics in traffic patterns, such as high traffic areas having a higher weight than low traffic areas.

These variables provide enough functionality to create graphs of varying sizes and shapes. NodeData components would then be attached to several GameObjects and positioned in the game scene. Their placement enforced the graph's structure and allowed for movement between the nodes with the agents following paths correctly. The positional data of each node is not simply an arbitrary coordinate; it is the physical position of each node in the world space. This means that the position can be transferred to the cars so they know which way to drive after getting their paths.

The agents themselves would be implemented in a new component called CarMover. This component contains all of the behaviors native to the cars, such as the ability to move between nodes and check if the goal has been reached. Movement was handled by adjusting the agent's position in the game world a fraction of the distance to the next node in its path every frame of execution. This simulates smooth movement. Key variables used by the CarMover are:

- Search Algorithm: A reference to the algorithm the car uses to calculate a path to its goal.
- Node Database: A reference to a database that contains every node on the graph.
- Destination Director: A new component that coordinates car goals. More on this in chapter 6.

- Agent: A reference to the agent's transform for position data.
- Starting Node: The very first node the agent begins on when the application is started.
- Model: A reference to the 3D model that represents the agent in the game world.
- Speed: A modifier that determines how fast the agent moves.
- Goal Node ID: An integer representing the goal node the agent is currently moving towards. This changes as new goals are chosen.
- Start Delay: The amount of time (in seconds) the agent must wait before beginning traversal of the graph.


Car Mover (Script)

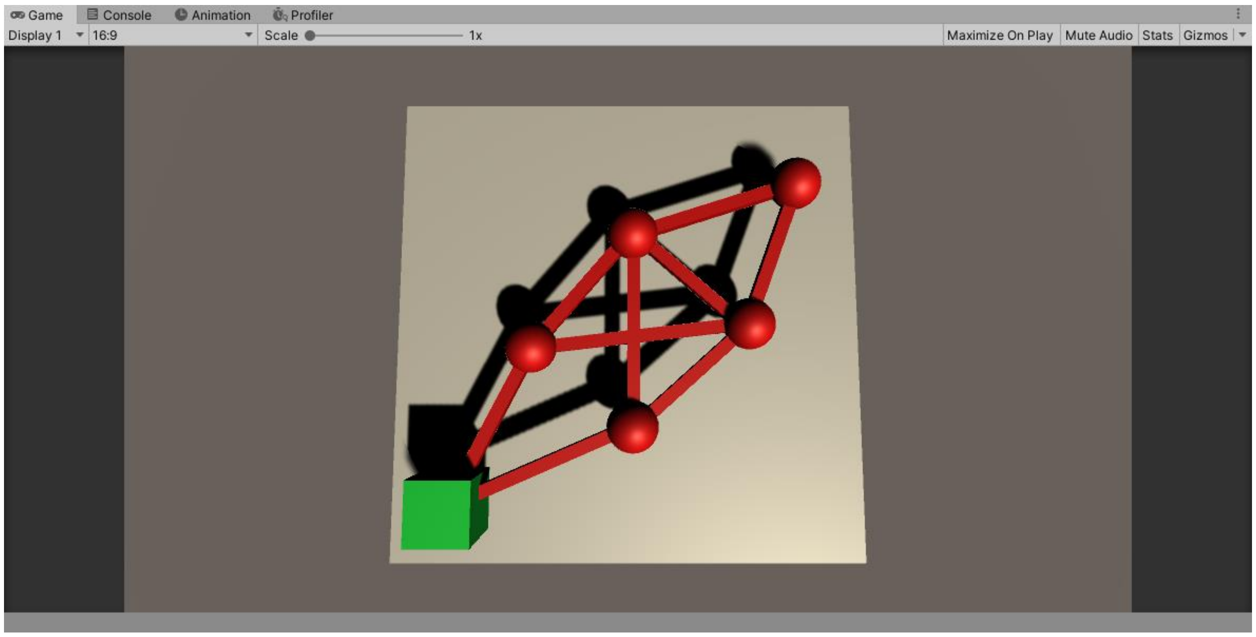
Script	CarMover
Search Algorithm	Dijkstra (Dijkstra)
Node Database	NodeDatabase (NodeDatabase)
Destination Director	DestinationDirector (DestinationDirect
Agent	Car 1 (Transform)
Starting Node	node5 (NodeData)
Model	Model (Transform)
Speed	3
Goal Node ID	71
Start Delay	3
Translation	X 0 Y 0 Z 0
Reference Translation	X 0 Y 0 Z 0
Testing Traffic	☐
Testing Rear End	☐
Testing Destination	☐

The CarMover Inspector.

The first algorithm to be implemented to calculate paths for the agents was a modified Breadth-First Search (BFS). This algorithm works by beginning execution at a provided source node, then traverses the graph one layer of nodes at a time. Under normal operation, the execution would end once all nodes have been discovered by the algorithm. However, the implementation in this project stops executing when the goal node of the agent has been discovered and then the path taken to reach the node is returned to the agent.

4.2 Creating A Prototype

After the creation of the core systems, a small prototype was developed to test/demonstrate these systems in operation. This prototype would consist of a single cube navigating a small graph while receiving random destinations upon reaching each goal. For the purposes of this prototype, every node was made a possible goal. Visual indicators for each of the links on the graph were also implemented to convey the adjacent nodes that can be reached from a given position. Here is what the prototype looked like:



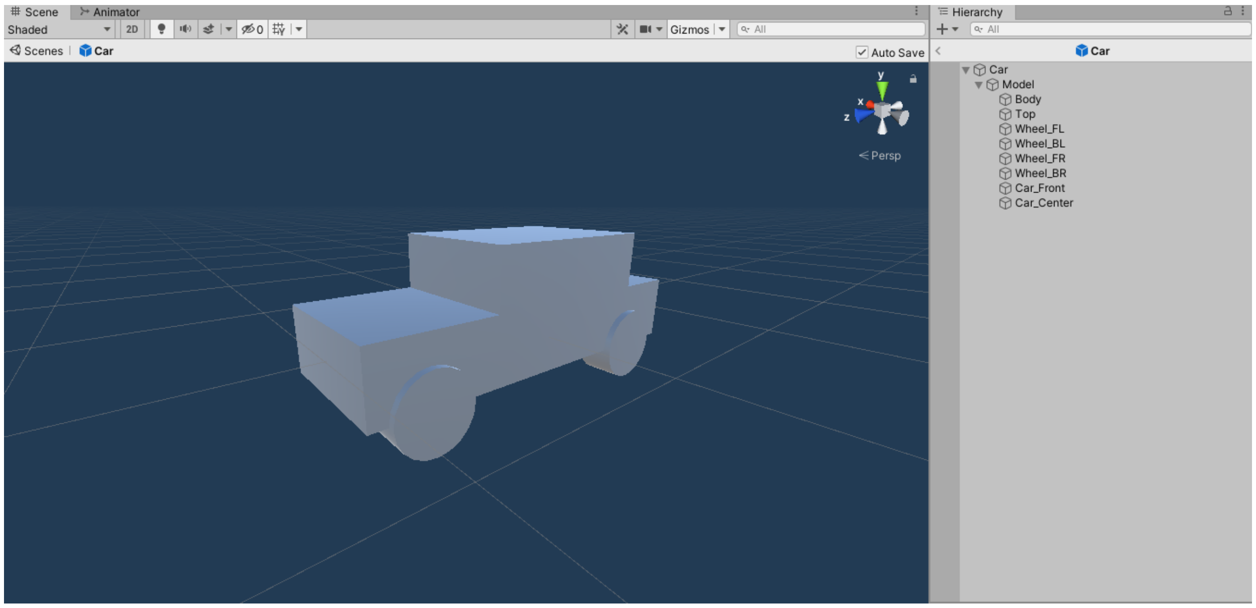
The prototype that was created to test BFS and traversal functionality.

This prototype provided a lot of value. All previously implemented systems were functioning, and the cube was able to navigate the graph without failure. The prototype also demonstrated a few quirks that would need to be addressed in the full implementation. There were some small ones, such as the cube not rotating with respect to its movement, but the main oddity was the pathfinding itself. Although it was entirely functional (as in no paths ever failed to be calculated), it showed that BFS wasn't exactly the most optimal algorithm to use. The main reason being that it simply returned the first path that reached the goal, rather than calculating all of the possible paths and choosing the most optimal one. In the context of the design of this project, the most optimal path is simply the shortest unit distance, as in the path that traverses the least number of nodes. This would be addressed as development moved away from the prototype and towards full implementation.

5 Moving From Prototype To Implementation

5.1 Building the Cars and Environment

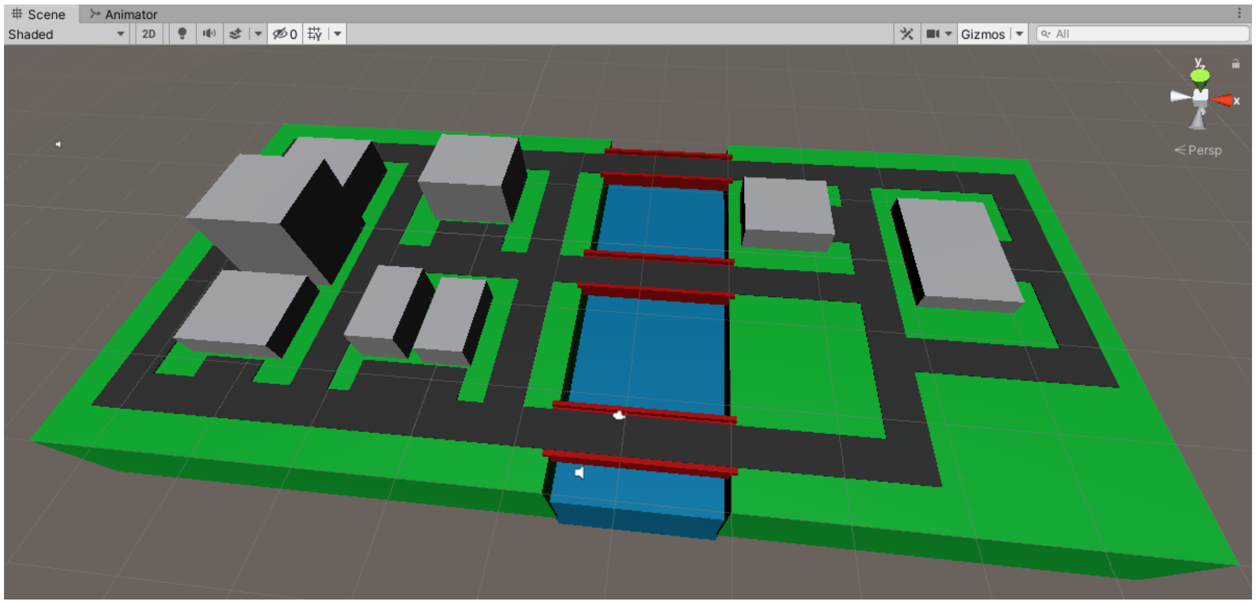
The prototype was very successful in being a proof of concept, but there were some things that needed to be addressed moving into the full-scale implementation. A change that needed to be made was a set of designated goal nodes the car would aim for, rather than arbitrarily stopping at any node. Also, the car would need to face the direction it was going, rather than just slide along the ground. These two changes are easily made using some member functions in Unity and by utilizing the isGoalState variable created for NodeData. Another change from the prototype that would be made is a visual adjustment to the agent and the environment the graph is located in.



The Prefab of the model for the car.

Creating simple models with Unity's built-in shape assets is very intuitive. Rather than creating the models with a traditional sculpting technique, where one would start with a single shape and extrude, manipulate, expand, and morph it into the desired object, modelling in this project was approached the same way one would a set of Lego. This was done by building the models with individual pieces and putting them together. This is able to be done because of the parenting hierarchy system that exists in Unity. If an object is a child of another transform, it will maintain constant relative position, rotation, and scale relative to its parent, so having all of the pieces of the car's model as a child of some parent "container" object, moving the parent will cause the model's pieces to move together as if it was one solid object.

Building the environment followed much of the same workflow as building the car model. Using the designed map layout, the foundation and key landmarks of the map were placed in appropriate locations while adding a bit of design flair to elements such as the bridges that connected the two islands. Here is what the first pass of the environment looked like:



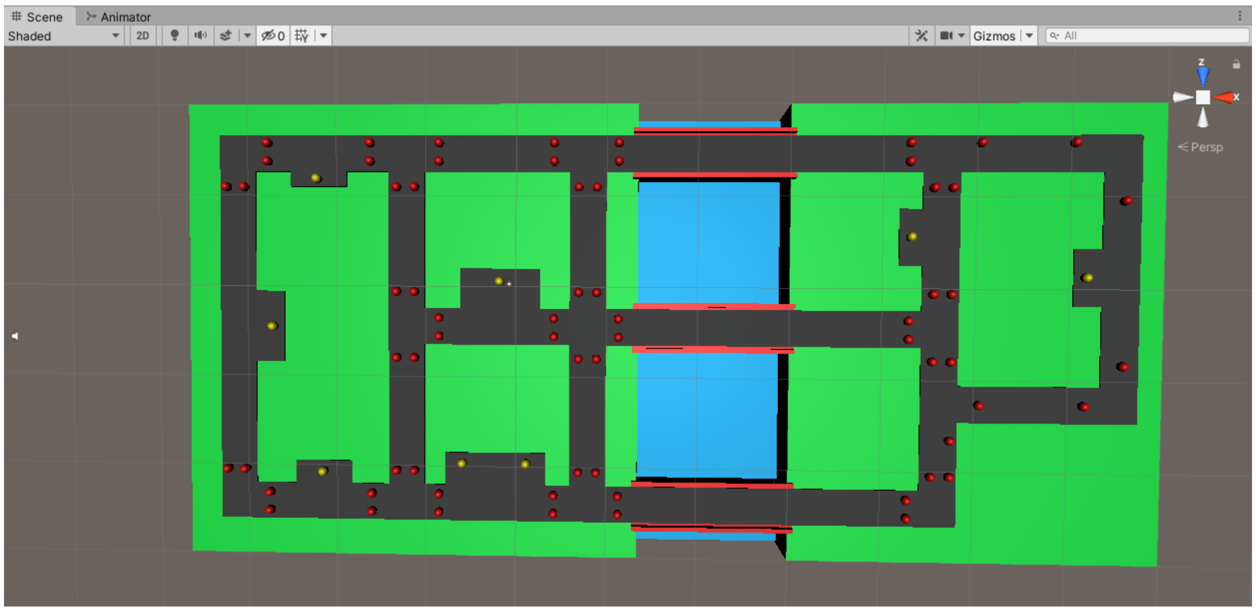
The environment created in-engine.

The only change to the structure of the level from the original design was the placement of two buildings and their entrances, otherwise the core layout remained intact. With the car model created and the environment laid out, it was time to start building up the functionality of this scene.

5.2 Implementing the Full-Scale Graph

Building the graph in the full environment wasn't any more or less difficult than the prototype. However, because of the sheer increase in scale, it took much longer to place all of the nodes and ensure they were lined up properly in order for the car's traversal to function. Nodes were placed only at locations that would require the cars to be able to turn, most notably at intersections. Enforcing proper driving directions along the different lanes of the road wasn't a difficult task to accomplish, as this is a directed graph. There are no paths that link two lanes of traffic, meaning that head-on collisions are natively avoided. The directional nature of this graph ensures cars stay within the correct driving lanes. This also enforces the one-way road, as the nodes at the top do not allow for entry to the road. The path along the one-way road also does not allow the cars to leave from the bottom.

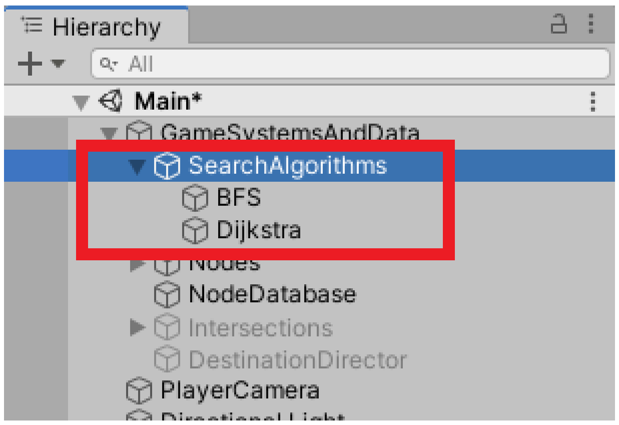
Another change that was compared to the prototype was that the destinations needed to be explicitly defined along the roads, specifically outside of each of the buildings. In order to differentiate standard nodes from building nodes when observing them, they were color-coded: red = normal, yellow = building. This has no effect on the functionality of the nodes; it is purely for visual clarity. Here is what the first pass of the graph looked like with the buildings removed:



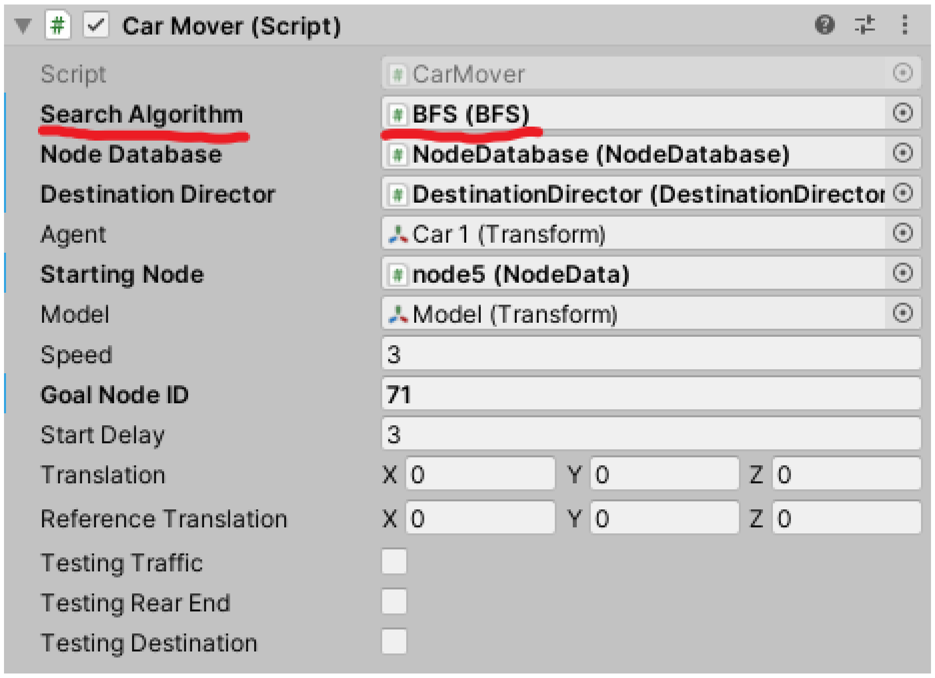
First pass of adding nodes to the environment.

This graph allowed for a car to successfully navigate around the level and find paths between each of the buildings.

To address the issues discovered with BFS pathfinding, an implementation of Dijkstra's single-source shortest path algorithm was created. This would make the car take the most direct path to its destination, rather than simply the first one it finds. This new algorithm is able to be swapped with BFS at any time in the Unity editor by simply dragging the reference to the agent game objects, meaning both BFS and Dijkstra's can be used simultaneously.



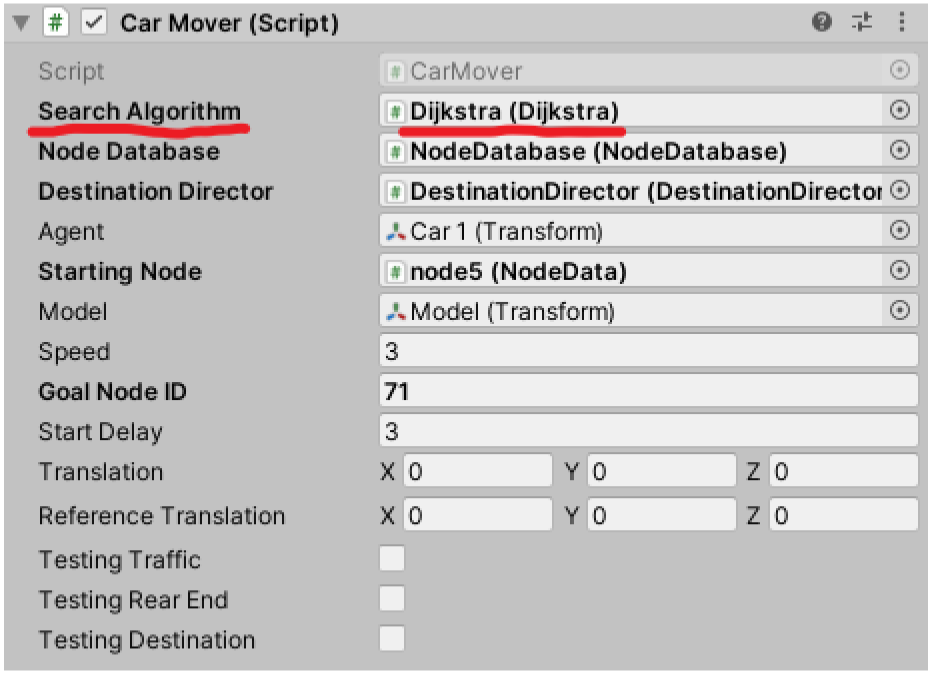
These objects contain the scripts for their respective algorithms.



Car Mover (Script)

Script	CarMover
Search Algorithm	BFS (BFS)
Node Database	NodeDatabase (NodeDatabase)
Destination Director	DestinationDirector (DestinationDirector
Agent	Car 1 (Transform)
Starting Node	node5 (NodeData)
Model	Model (Transform)
Speed	3
Goal Node ID	71
Start Delay	3
Translation	X 0 Y 0 Z 0
Reference Translation	X 0 Y 0 Z 0
Testing Traffic	☐
Testing Rear End	☐
Testing Destination	☐

Those objects are then dragged onto the CarMover script to provide a reference. Here it is with BFS selected.

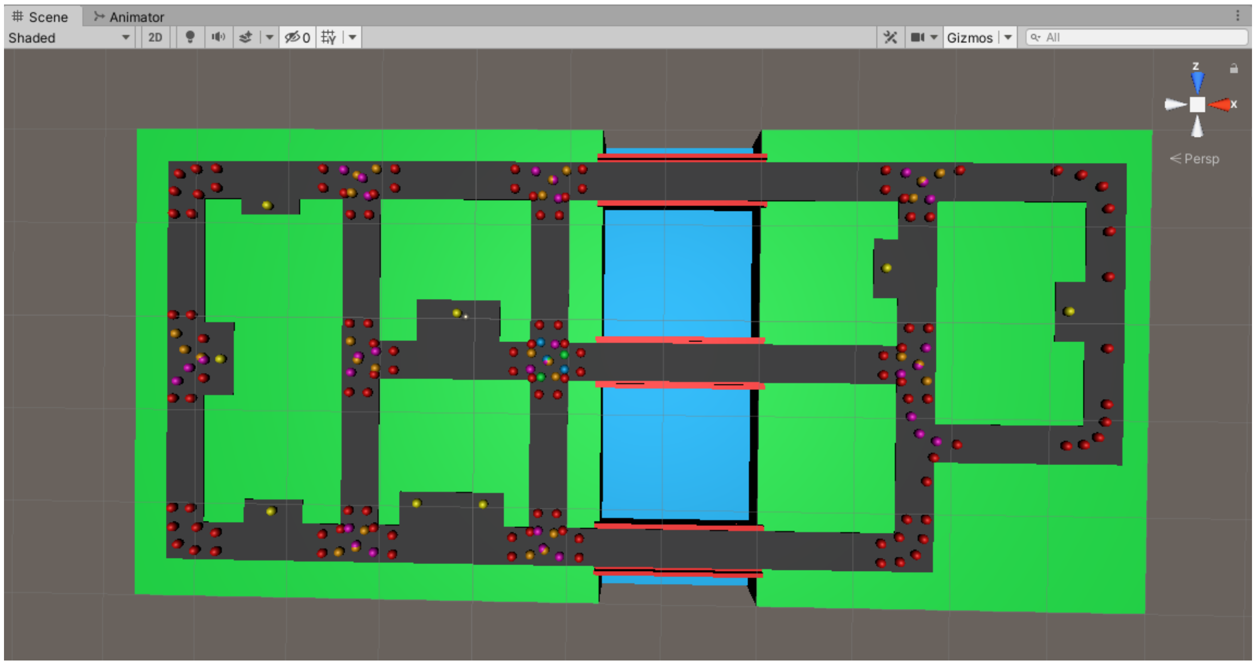


Car Mover (Script)

Script	CarMover
Search Algorithm	Dijkstra (Dijkstra)
Node Database	NodeDatabase (NodeDatabase)
Destination Director	DestinationDirector (DestinationDirector
Agent	Car 1 (Transform)
Starting Node	node5 (NodeData)
Model	Model (Transform)
Speed	3
Goal Node ID	71
Start Delay	3
Translation	X 0 Y 0 Z 0
Reference Translation	X 0 Y 0 Z 0
Testing Traffic	☐
Testing Rear End	☐
Testing Destination	☐

Here is the same car with Dijkstra selected instead of BFS.

Because the car traverses each node by taking the most direct path between them, turns aren’t very smooth. On the original node layout, the turns were very sharp diagonals. In order to address this issue, more nodes were added along each turn to guide the cars on a path that looked like a smoother turn. The logic behind this design choice is the same as that of using high-resolution cameras to take more detailed pictures. When looking at a digital image, what is displayed is a matrix of pixels. On low resolutions, images look blocky and lack smooth edges. By increasing the resolution, and thus adding more pixels, the image begins to look sharper and appears to be a more accurate representation of the subject of the image. After adding in the addition nodes, the graph looked like this:



Second pass of the node layout, now with more nodes on turns.

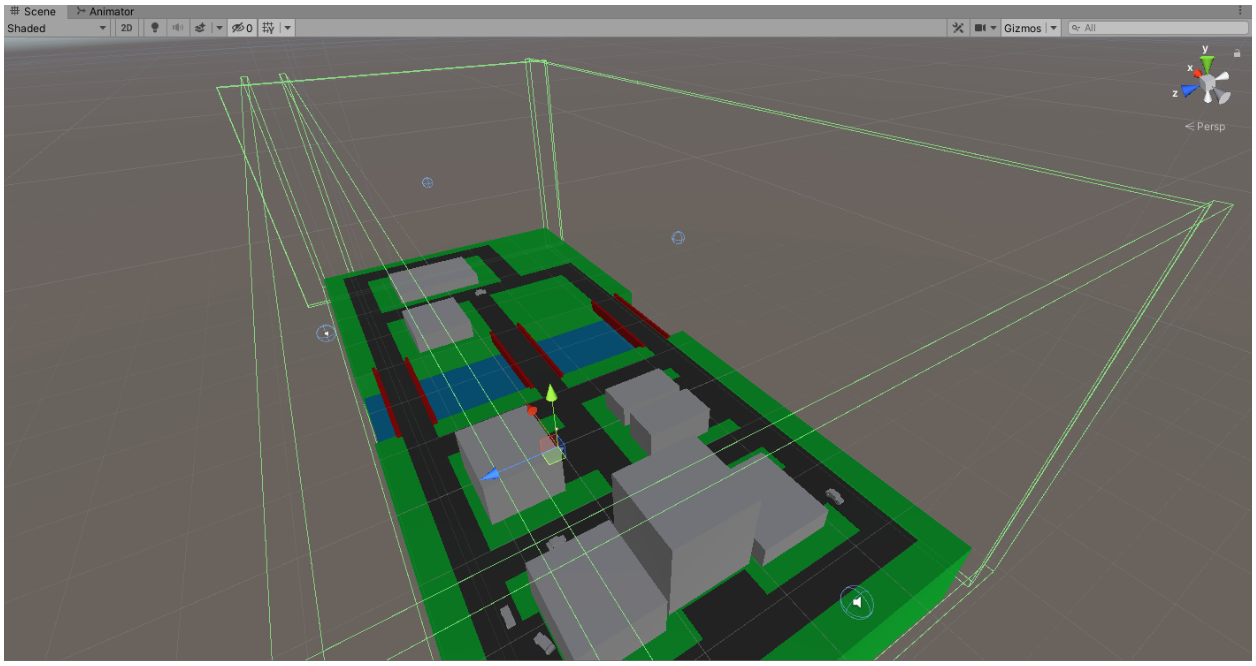
There was a minor side-effect that arose when testing these new turns. Since left turns took up more space, they needed more nodes to look smoother. Because they consisted of more nodes, they had a higher total weight compared to right turns (3 nodes total vs. 5 nodes total), which meant that they were of higher cost to the car's pathfinding. Although this behavior is not inherently flawed, it wasn't desired in this project, so the weight values of the turning nodes were adjusted in order to have all turns be of the same total value.

5.3 Creating an Interactive Camera

Another system that was implemented was a user-controlled camera that had a bird's-eye view of the environment. This camera could be moved around in a plane above the environment and could have its zoom level adjusted. This was created to provide the ability to observe the cars in the event the Unity project is compiled into a separate executable.

Every newly created scene in Unity comes with a Main Camera object. This was modified to allow for user control. It was renamed to PlayerCamera and a CharacterController component was added to it. The CharacterController is a built-in Unity component that handles moving objects around and is especially useful for user-controlled objects. To handle user-input and move the camera around, a new script was called PlayerController. Using Unity's input system, the script reads the user's input every frame and calculates the direction the camera needs to move based on this input. It then sends this direction to the CharacterController to move the camera around the level.

The PlayerController script also contains zoom functionality for the camera by adjusting the camera's field-of-view when the scroll wheel is moved. Finally, in order to prevent the user from being able to fly the camera away from the level, invisible walls in the form of Unity's collider system were placed around the environment that would "cage" the camera and prevent it from leaving. More information on Unity's collider system is provided in Chapter 6.



These green wireframes are the invisible walls. The player camera cannot pass through them. These are not visible in the game window.

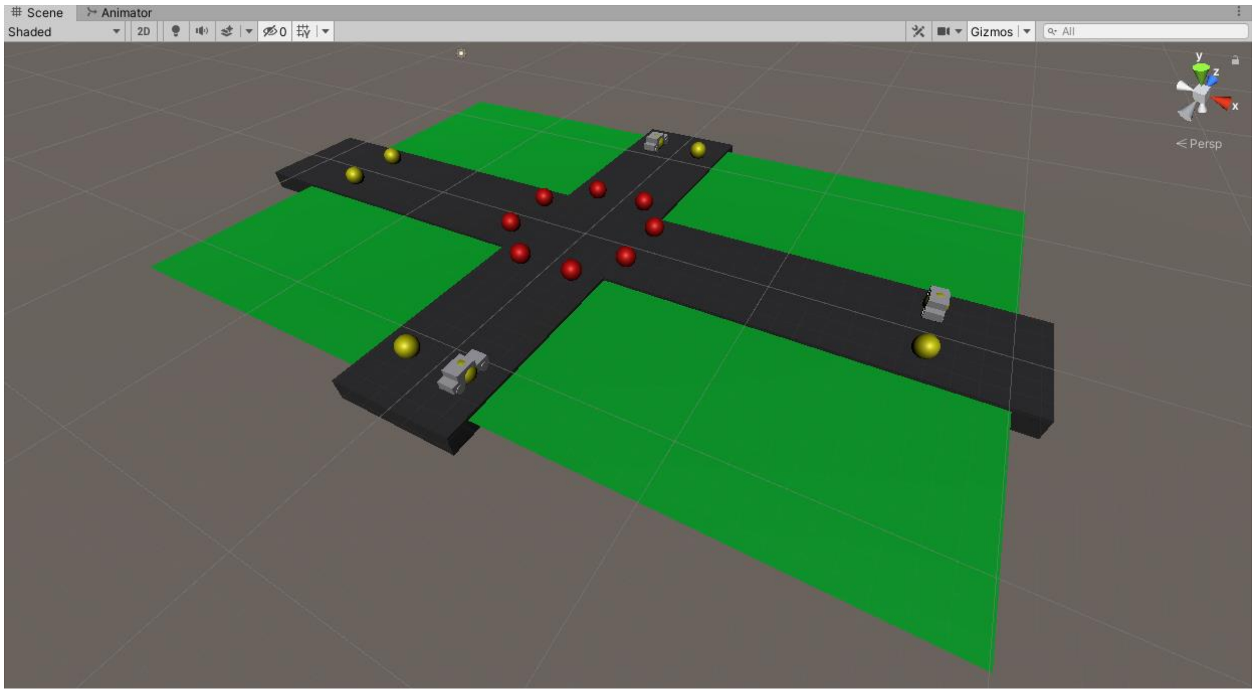
With all of these systems put in place, the main core of the A.I. was complete. The car could drive around building to building with a 100% success rate. The next step was to add more cars to the streets and begin developing new systems to address possible collision scenarios.

6 Adding More Cars to the Environment

6.1 The Intersection Problem

Intersections in everyday roads are handled in a variety of ways. Just looking at a typical 4-way intersection, the rules of traffic can vary. For example, there could be traffic lights present for each lane, with dedicated left-turn lanes, or it could simply be a 4-way stop. Regardless of how these intersections are handled, there needs to be some traffic flow control in order to prevent collisions between vehicles. Intersections in this project were handled in a similar nature to that of an all-way stop in real driving.

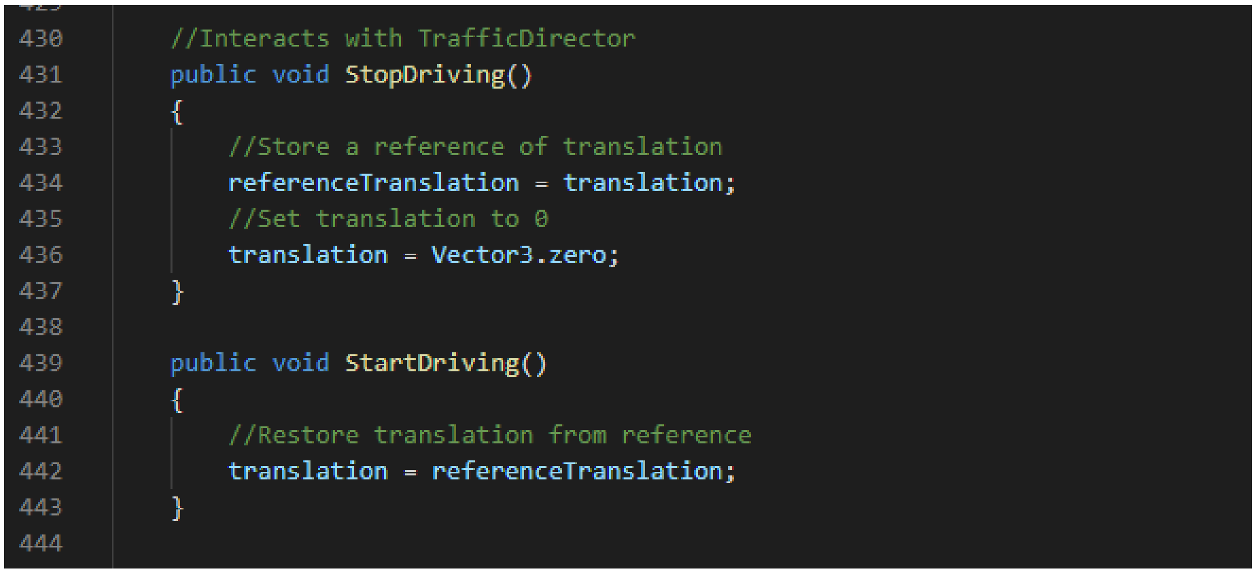
The rules for an all-way stop are quite simple: every car stops when first arriving at the intersection, then the car to arrive first goes first. The next car waits until the intersection is clear before they are allowed to go, and so on. This can be represented in code with a queue: first in, first out. To keep track of the order cars arrive at intersections in the scene, Unity’s collision system was used. Colliders can be attached to game objects which will send an alert to any attached scripts when a collision has been detected via its OnCollisionEnter() function. Similarly, colliders can be set up to act more like a piece of world space that detects objects moving through it instead of being a physical collider. This can be achieved by marking the collider as a Trigger collider, which would turn all of the OnCollision() calls with OnTrigger() calls. A separate testing scene for developing intersection solutions was created.



The intersection testing scene.

Because intersections control the movement of the cars by ensuring they stop upon arrival and only resume driving when the road is clear, functions needed to be created that could tell the cars to stop and resume driving. The functions StopDriving() and StartDriving() were added to the CarMover script. These functions simply manipulate the car's translation value to adjust how it is moving. StopDriving() stores a copy of the current translation, then forces the current translation to zero. This causes the car to stop moving. StartDriving() restores the copy of the translation so the car begins to move again. Any game event that requires a car to stop or start driving will get a reference to the car's CarMover script, then call whichever function is needed.

These new functions may seem like changing the car's behavior to account for new situations instead of using an external system, but this is not the case. Although these functions are a part of the CarMover script, they are never called by the car itself. It is impossible for a car to tell itself to stop driving, even with these functions. Only an outside source can call these functions and force a car to stop or start. This solution worked during the isolated testing, but it came with some nasty side-effects that will be discussed further in the chapter 7.



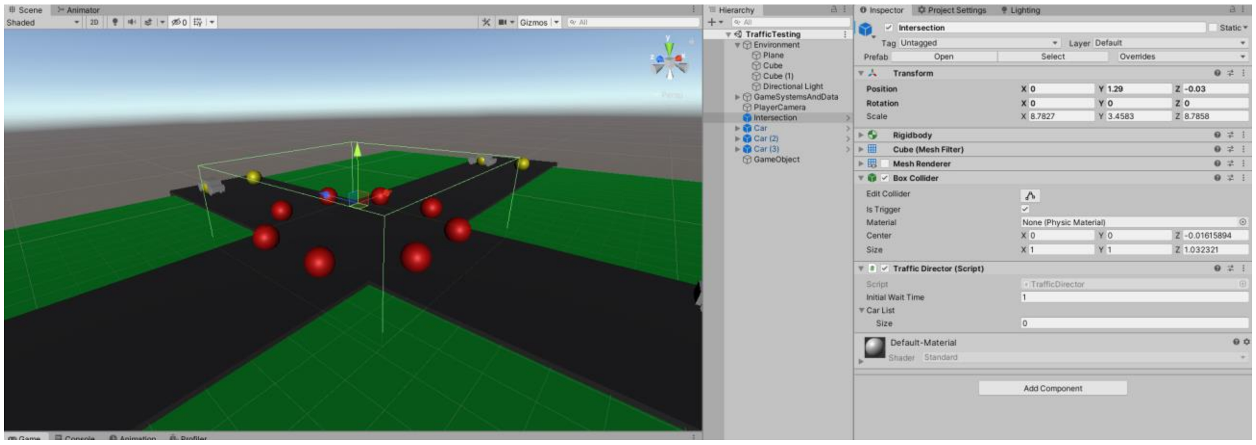
```
    //Interacts with TrafficDirector
    public void StopDriving()
    {
        //Store a reference of translation
        referenceTranslation = translation;
        //Set translation to 0
        translation = Vector3.zero;
    }

    public void StartDriving()
    {
        //Restore translation from reference
        translation = referenceTranslation;
    }

```

The initial implementation of the StopDriving() and StartDriving() functions in the car mover script.

The testing scene served as a controlled environment to force the cars into the intersection scenario by having them all enter the intersection at roughly the same time, which allowed for quick and efficient testing. Implementation of the intersection logic was done in a new script called TrafficDirector. This script contains three functions: Update(), OnTriggerEnter(), and OnTriggerExit(). When the collider detects that a car has entered the intersection, that car is told to stop driving and a few lists on the script are updated. The first one is a list of all of the cars currently waiting at the intersection. This only includes cars that are next to go in their respective lanes, not every single car queued up. The second list is a list of boolean values that keep track of if a car has been told it is clear to drive. Initially, this value is false for all cars and becomes true when the intersection allows the car through. The final list is a list of float values that keep track of how long each car has been waiting at the intersection. Each car must wait a minimum of one second before it is allowed to proceed. However, all of these wait times are adjusted simultaneously. This is because all other cars automatically stop if there is a car in the queue ahead of them, so their wait time can be tracked even if they aren't next to go.



```
Intersection                                   Static
Tag  Untagged              Layer  Default
Prefab   Open        Select        Overrides
Transform
Position     X 0          Y 1.29        Z -0.03
Rotation     X 0          Y 0           Z 0
Scale        X 8.7827     Y 3.4583      Z 8.7858
Rigidbody
Cube (Mesh Filter)
Mesh Renderer
Box Collider
Edit Collider
Is Trigger   ✓
Material     None (Physic Material)
Center       X 0          Y 0           Z -0.01615894
Size         X 1          Y 1           Z 1.032321
Traffic Director (Script)
Script              TrafficDirector
Initial Wait Time   1
Car List
  Size              0
Default-Material
Shader   Standard
Add Component
```

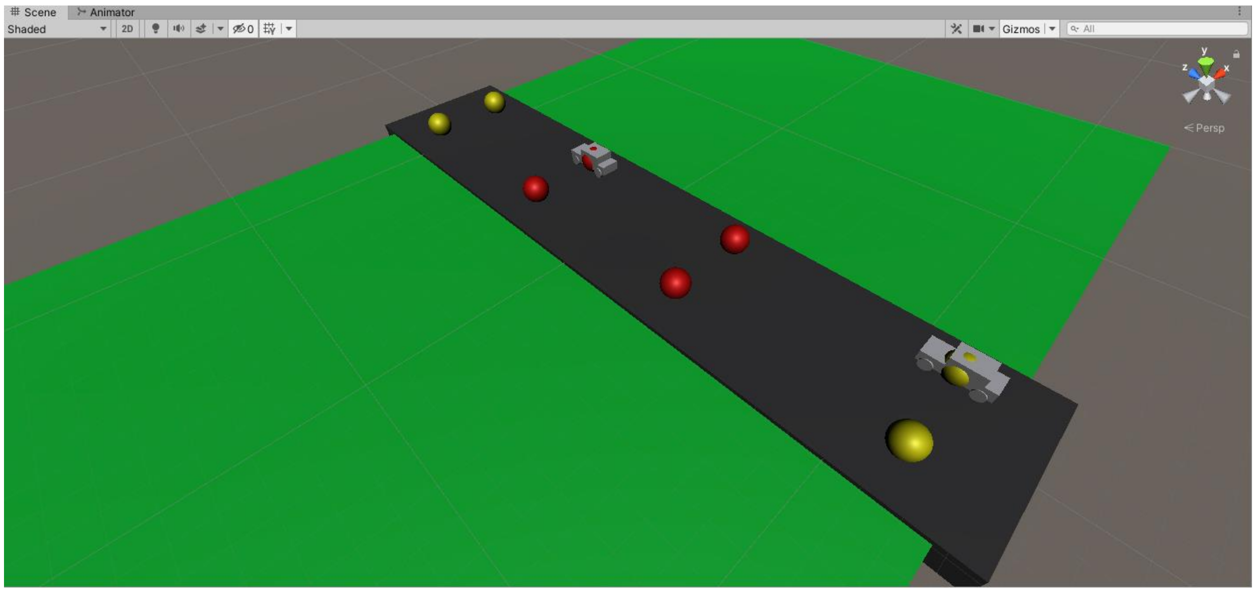
A visual reference for what the intersection area looks like as well as the Inspector for the TrafficDirector script. This wireframe is not visible in the game view.

After the first car has waited a sufficient amount of time, the TrafficDirector tells the car to continue driving by calling its StartDriving() function. It is not popped from the various lists yet, as that only happens once it leaves the intersection collider. This prevents other cars waiting at the intersection to start driving before the roads are clear. With this functionality, collisions at intersections are successfully handled. But, this is not the only type of possible collision. With cars stopping at intersections and taking turns throughout the level, it is possible that a rear-end collision can occur, so another system needs to be created to account for them.

6.2 The Queueing Problem

Rear-end collisions in the real world are usually the result of one driver not paying attention when another is in front of them. In this simulation, the cars do not have any sort of thinking that analyzes current road conditions or to detect their proximity to other drivers, so any time one approaches another stopped car, a collision ensues. To prevent this, a system that could notify a car when it is too close to another one was needed.

The logic behind this scenario is much like that of the intersection, but localized to only two cars. The main difference is that a collision of this nature could theoretically happen anywhere, not just at key points on the road. To begin addressing this issue, another testing scene was established that would force the behavior.

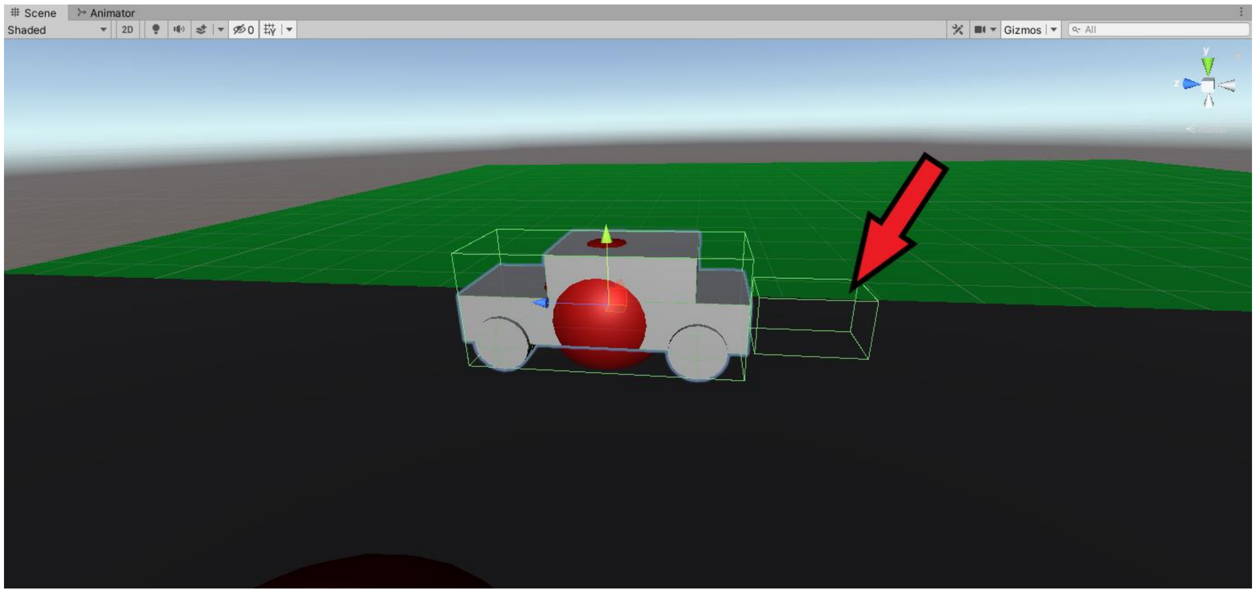


The rear-end collision testing scene.

Because the basic idea is the same as that of the intersection, a similar approach could be used. The car prefab was modified to include an extra collider following behind it. Attached to this collider is a new script called CollisionAvoidance. When a car enters the trigger collider attached to the back of another car, it is told to stop. The car in front does not need to know what’s going on behind it, because that has no bearing on its own motion, so it just keeps driving. After the stopped car exits the trigger collider (by nature of the front car moving away), the CollisionAvoidance script tells the car to wait at least one second before it can resume driving. This ensures that enough space is created between the two vehicles to avoid a constant state of triggering the collision event.

This may seem like it is a modification of the cars’ core behaviors because changes were made to the car objects in the scene, but this is actually not the case. The car itself is not responsible for detecting nearby entities. The collider’s detection of nearby cars is not being monitored or controlled by the car itself. When the collider gets notified that a car has entered it, it tells that car to stop in the same way the intersection does. The car never initiates a stop or start in collision scenarios of its own accord. It is all being handled by external systems. Thus, the cars’ systems and behaviors have remained unchanged.

To test this in the scene, the car in the back starts driving after 3 seconds while the car in the front has to wait 10 seconds. This ensured that the first car would be caught behind the second one in order to observe their behavior.

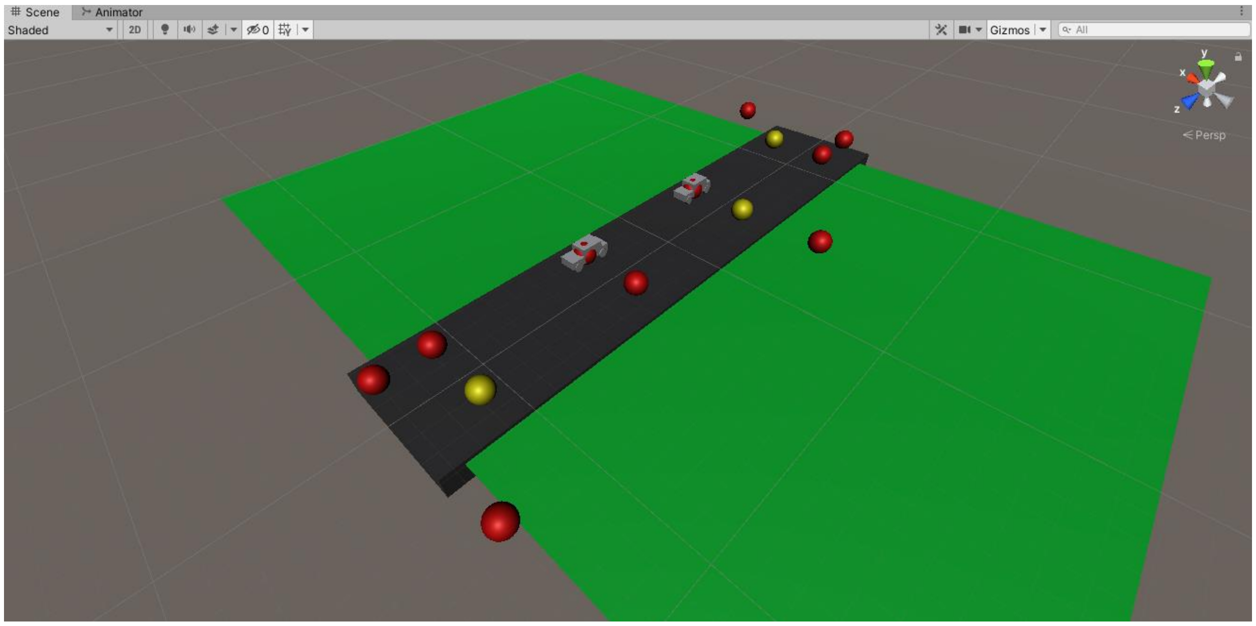


This is a visual of what the new collision box looks like. It is a child object of the Car, so it will follow wherever it goes. Much like the intersection, this wireframe is not visible in the game view.

These two traffic control systems handle everyday driving very well, but a third type of collision can occur that throws off the cars' movement. This is a result of overlapping at a building. When two cars are able to fully overlap, they can end up trapped in each other's CollisionAvoidance boxes, essentially making them stuck. This is what would be called the "Destination Problem."

6.3 The Destination Problem

The main reason why this problem occurred is that cars were allowed to go to the same destination, even if another car is already there/is heading towards it. Because there aren't any parking lots, every time a car arrives at a destination it "parks" on top of the node. A robust parking system would get around this issue, but would require a complete redesign of the node layout and the environment, which would take a non-justifiable amount of time to implement. Instead, a different solution was opted for: do not allow cars to go to currently occupied/soon to be occupied destinations.



The destination testing scene.

The above testing scene was used to create a controlled environment in which one destination is always available to the cars, with ways to get around each destination if they were occupied. The basic idea of keeping track of which destinations are free is similar to the intersection behavior of keeping a running list of all cars waiting to go. A new script called DestinationDirector was created and attached to an empty game object in the scene so all of the cars could have a reference to it.

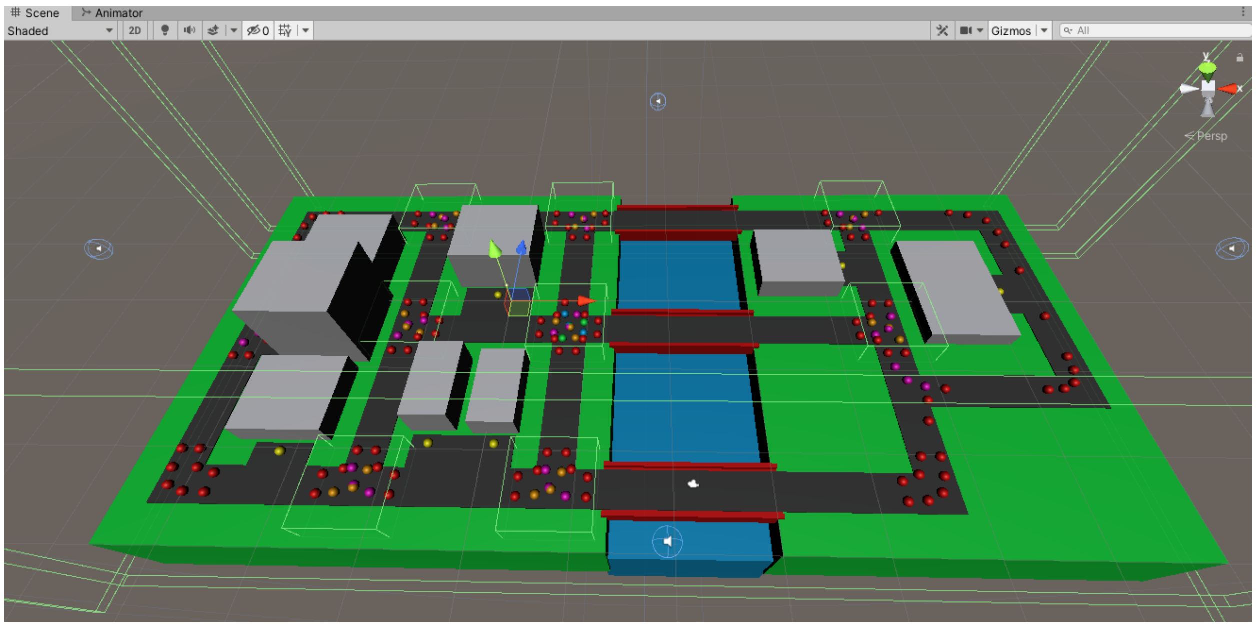
Whenever a car selects a destination to go to, it is added to a list on the DestinationDirector. Initially, this list gets populated with all of the starting destinations each car has assigned before the simulation runs. Upon arriving at its destination, a car will attempt to secure a new one to go to. After randomly selecting from the range of possible destinations, it checks its selection against the list on the DestinationDirector. If the car's selection appears in the list of occupied destinations, it is forced to select a new one. Again, this is done randomly. Once an available destination has been found, the car "claims" it by setting its next goal to that destination and adds it to the list of occupied locations.

It is important to note that a destination is marked as occupied not just when a car is physically there, but whenever a car has it selected as its goal. This is to prevent cars needing to completely recalculate paths mid-drive, especially when we reach the upper limit of how many cars are capable of existing at once. As will be discussed in chapter 8, there is currently no solution for what a car should do when it has no destination, but contingencies can be implemented to address this issue. Certain race conditions can cause a car's goal to become occupied right as it is about to arrive, meaning a new path is impossible to calculate, since it never reaches the node. The DestinationDirector ensures that this can never happen in order to keep cars running smoothly.

One may notice that this system actually did necessitate a change in the behavior of a car. Rather than simply going with the first destination it chooses, it "asks" the DestinationDirector if it is available first. A further discussion of this appears in chapter 7.

6.4 Incorporating the New Systems

Individual testing of each system yielded positive results. The systems worked well in isolation, but the key to this project's success was getting them working in conjunction with one another. Bringing them into the main scene was a simple process thanks to Unity's workflow systems. A new prefab of the intersection object was created that could be duplicated ad nauseum and the car prefab was updated to contain the new collision systems so the existing car in the main scene would automatically receive those changes. The initial intersection boxes were placed at the appropriate locations and the maximum number of seven cars were spread out along the graph. To make things more interesting, BFS was not fully replaced with Dijkstra's. The cars use a mix of the two: four of them use Dijkstra's and three of them use BFS. Here is what the scene looked like after everything was in place:



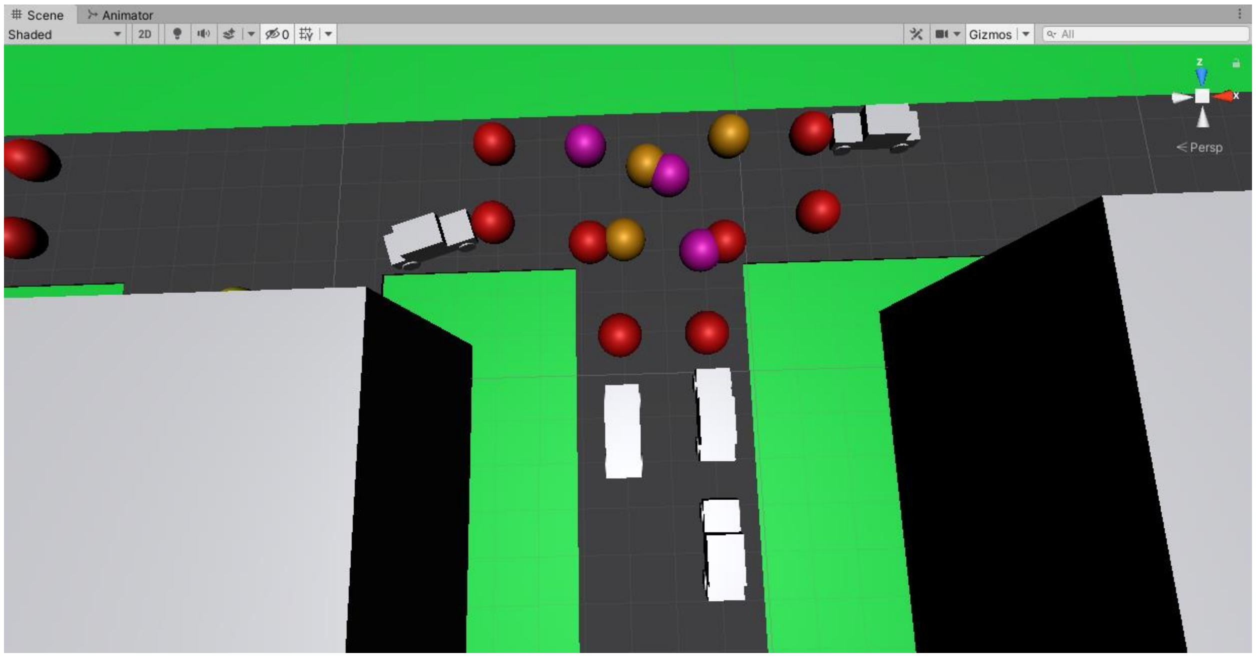
The main scene with added traffic control elements.

7 Testing and Debugging

Locating and addressing anomalous behaviors of the cars is very important for the purpose of this project. Cars that appear to phase through each other, get stuck for seemingly no reason, and have wild driving tendencies would completely shatter all of the effort put towards making the cars appear intelligent. Once all of the systems were in place, rigorous testing was done to iron out potential issues.

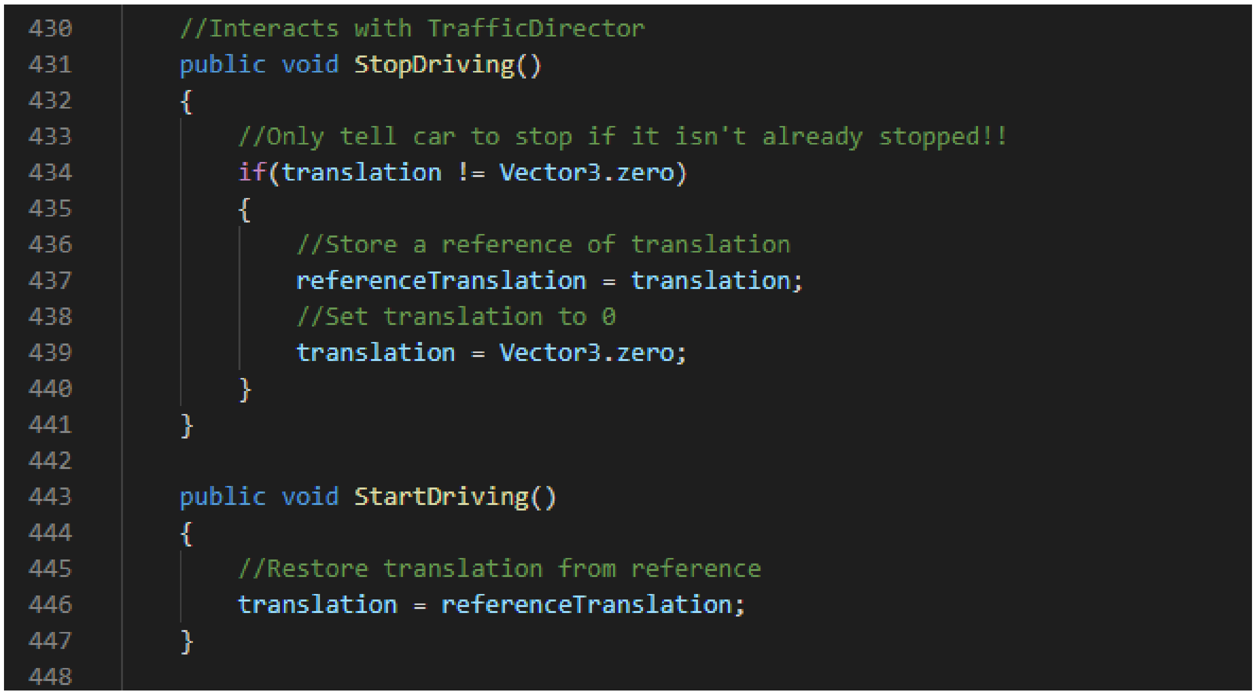
7.1 Addressing the Small Bugs

Through some initial dry runs, cars were observed to become gridlocked. In certain areas of the level, they could get stuck in a “collision loop.” This appeared to be the result of some really poor timing of a car entering an intersection and hitting a rear-end trigger close to the same time, locking them in place. That would then cause any other cars behind them to be stuck in perpetuity as well, eventually leading to all cars stopping completely.



An example of what a gridlock condition might look like. The only car able to move is the left car on the vertical road, as it has just left the intersection. All others are stuck.

When two collision events happen at the same time, the car’s StopDriving() function gets called twice. Under normal operation, the StopDriving() function stores a copy of the current translation for the car and forces the current translation to zero, which stops it from moving. When only one collision event occurs, this functions properly. But, if more than one collision event occurs before the copy of the translation is restored, the copy becomes overwritten by the second StopDriving() call, also becoming zero. Whenever StartDriving() is called again, it reassigns the copy to the actual translation used for movement. However, because the copy is now zero, a value of zero is sent back to the translation, meaning the car is completely stuck. To remedy this issue, a failsafe was added that would prevent the copy of the original translation from being overwritten if the current translation is already zero, i.e., the car is already stopped and hasn't been told to go yet. This solved the gridlock problem. But much like the original implementation of StartDriving() and StopDriving(), this solution wasn't entirely foolproof.



```
//Interacts with TrafficDirector
public void StopDriving()
{
    //Only tell car to stop if it isn't already stopped!!
    if(translation != Vector3.zero)
    {
        //Store a reference of translation
        referenceTranslation = translation;
        //Set translation to 0
        translation = Vector3.zero;
    }
}

public void StartDriving()
{
    //Restore translation from reference
    translation = referenceTranslation;
}
```

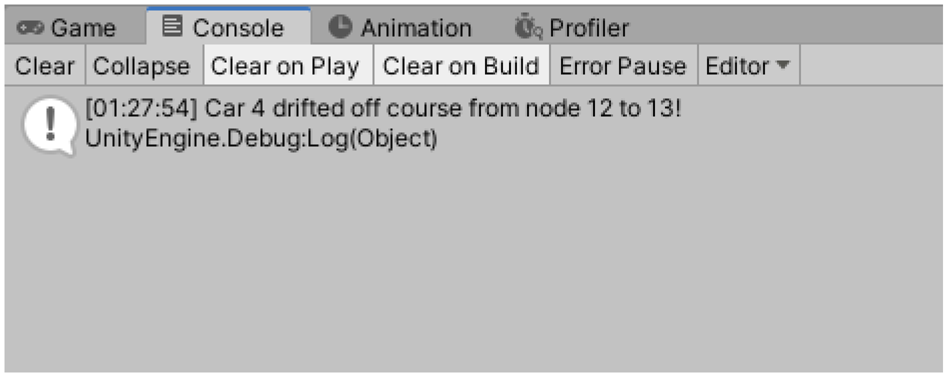
The StopDriving() and StartDriving() functions after the overwrite prevention was introduced.

7.2 Further Testing Methodology

After addressing the gridlock issue, the program was run for an extended period of time to see any anomalies in behavior. There were a few close calls on some collisions, but they were well within the tolerance of functionality as they didn’t break the execution. The big issue was the tendency for some cars to drift off of the road and leave the play area.

Attempts at tracking down the cause of this issue proved rather difficult, as the nature of the cars can vary depending on the state of the simulation. Also, because there are seven cars driving around at once, points of failure can be difficult to spot, and by the time they’ve been noticed it’s often too late to diagnose the cause of the issue or where it went wrong. In order to run meaningful tests, new debugging tools were created that would keep records of errors if/where/when they occur.

The walls that were used to keep the user camera in the play area turned out to be very useful in detecting the drifting issue. Any car that drifts away from the level would eventually have to collide with these walls, meaning the walls could make a log of drifting behavior whenever a car is detected passing through them. A new script (simply named Wall) was created that would make a log in the console and play a sound whenever a car was detected passing through the wall. The log in the console consists of three important pieces of information: the car that drifted away, what node it was driving from, and what node it was driving towards. This provided an exact point of failure on the map to investigate.



```
[01:27:54] Car 4 drifted off course from node 12 to 13!
UnityEngine.Debug:Log(Object)
```

An example log entry that would appear after a car has drifted off course.

Ten trials were run and their results were recorded into a spreadsheet. Each trial began with simply running the program in the Unity editor while a stopwatch was running at the same time. Once a wall detected a car, the stopwatch was stopped and the log was investigated. Below are the results of these ten trials.

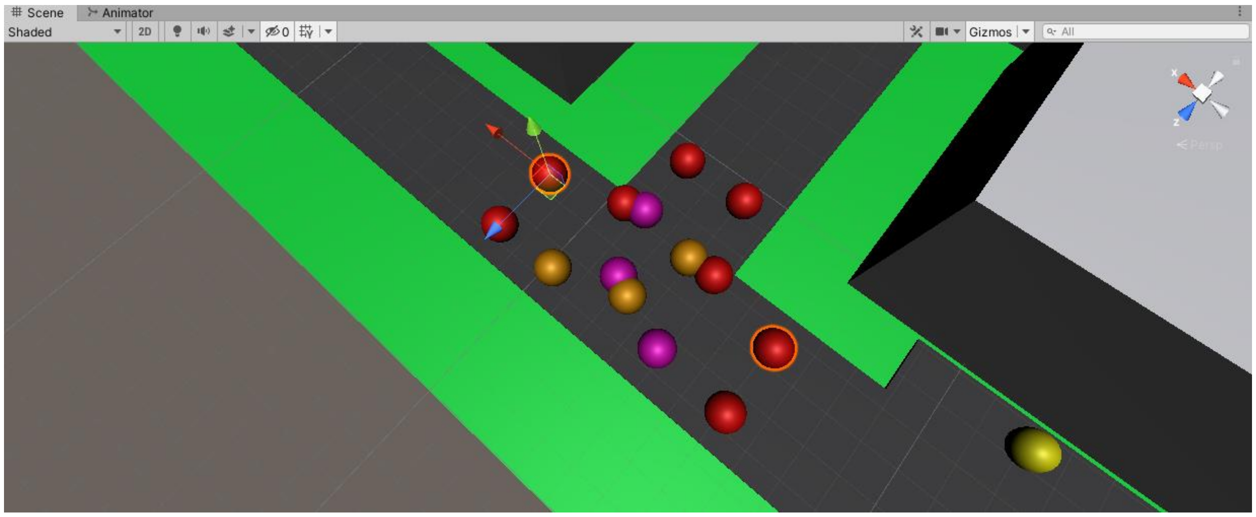
Table 1: Measuring time until failure

Trial #	**Time Until Failure**	**Failed Car #**	**Going From Node**	**Going To Node**
1	4m 17s	2	85	43
2	2m 09s	1	57	12
3	6m 10s	1	57	12
4	6m 07s	5	57	12
5	3m 05s	7	57	12
6	2m 01s	1	57	12
7	3m 40s	3	57	12
8a*	1m 42s	5	57	85
8b*	1m 42s	4	0	93

9	0m 54s	2	16	58
10	1m 48s	2	125	3

*Two cars failed at the same time during this trial, so both logs are included

There aren't any patterns regarding time until failure and which car caused the failure. The times range from 54 seconds to over 6 minutes, and nearly every car failed at some point. Something must be going on between nodes 57 and 12 in order to cause 6/11 of the failures.

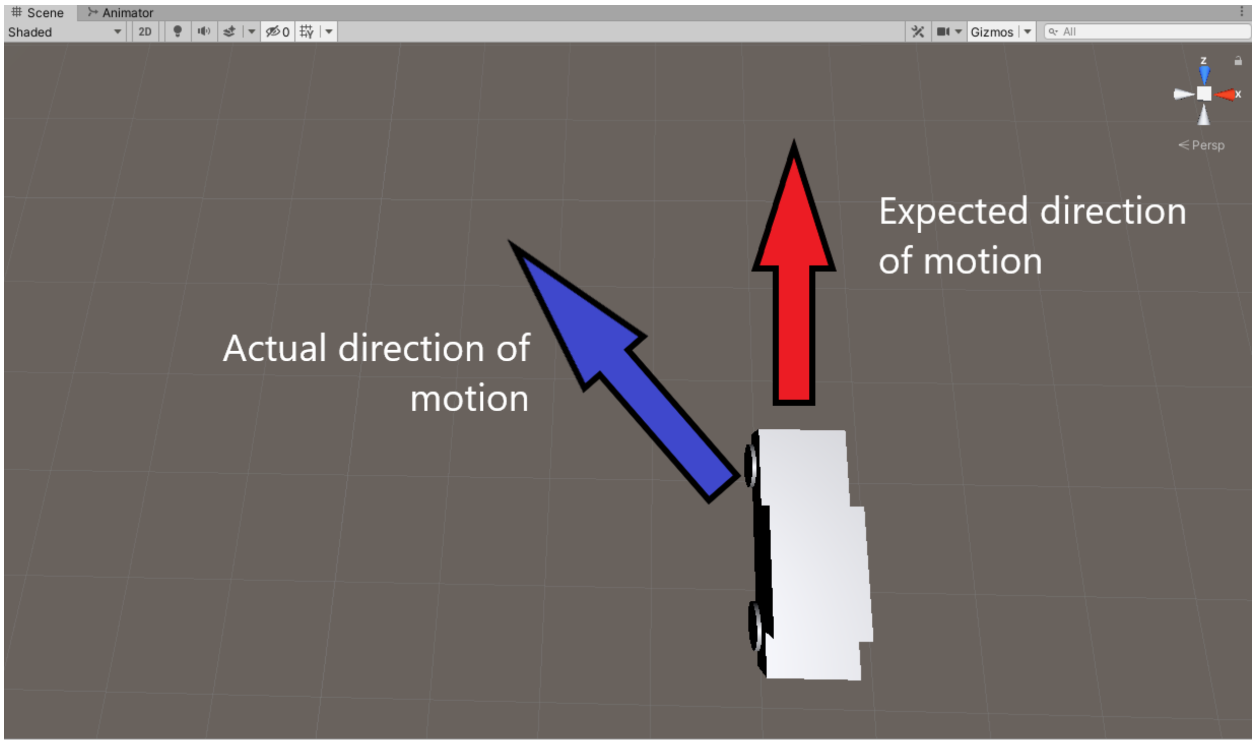


The road between nodes 57 (highlighted right) and 12 (highlighted left).

7.3 Data Analysis and Bug Searching

Upon first inspection, there does not seem to be anything particularly strange about this stretch of the graph. It just looks like the straight path across an intersection. Also, the other points of failure aren't all straight paths, as some of them are during turns at intersections.

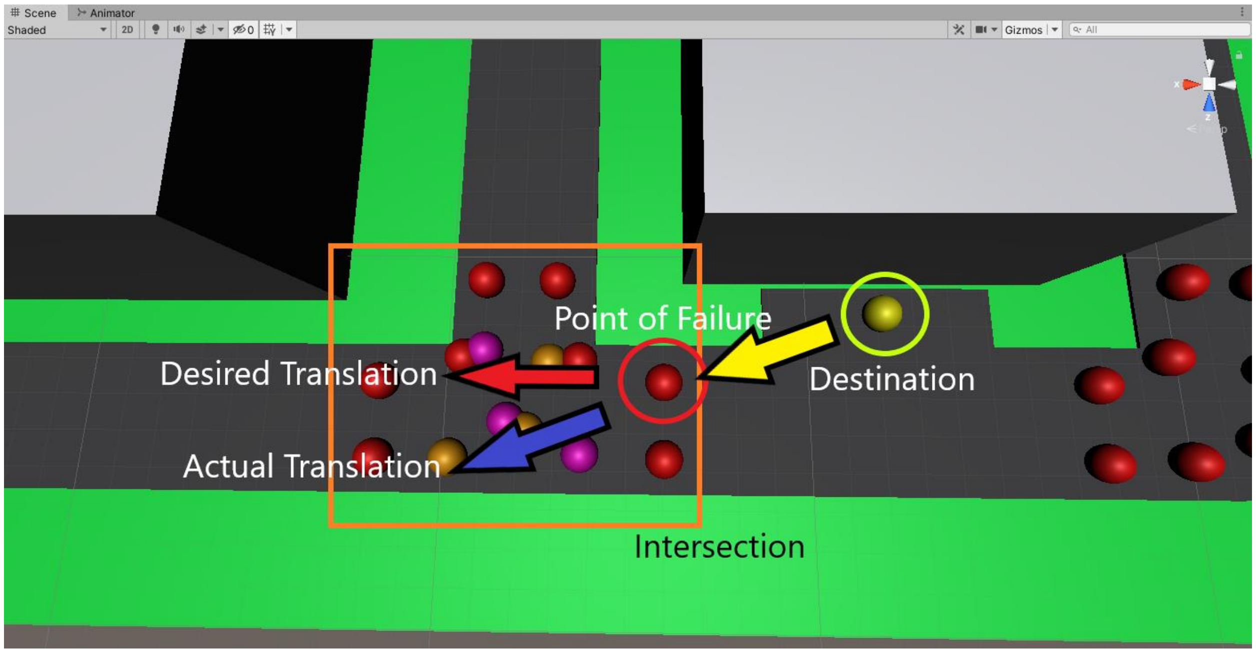
One odd thing that was observed about each failure was the way the cars drifted off of the road. After the log would get reported, the car that drifted away was located and followed. The cars weren't facing in the same direction they were moving in. Below is an example of what I mean.



A visual describing the drifting behavior of the cars.

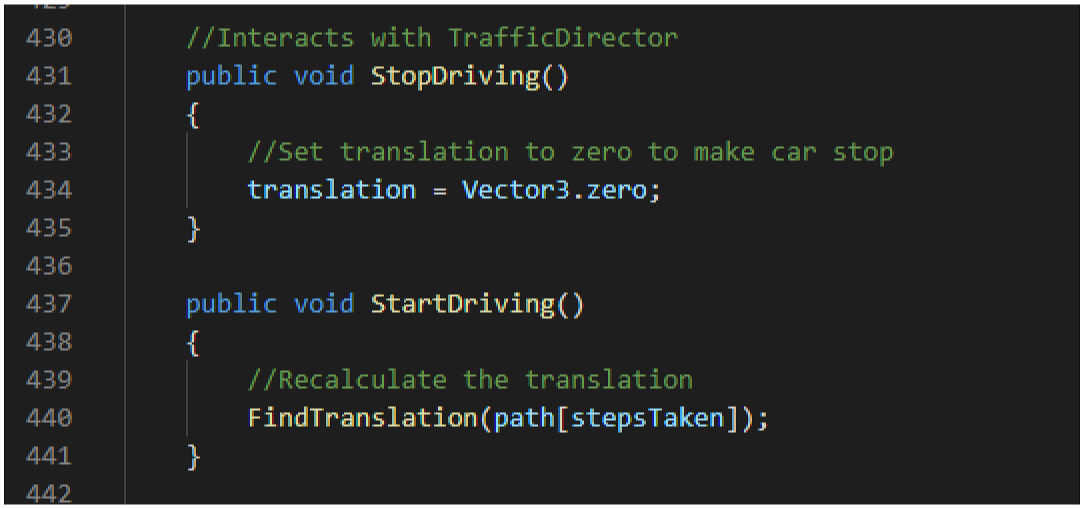
This behavior should be impossible because the cars are told to point directly at the node they are moving towards. The only way this misaligned direction could occur is if a car is using an incorrect translation vector. Cars only get a new translation upon reaching a node, which calculates the translation to the next node, or when they are told to stop or resume driving.

All of the failure points exist right after destination nodes and during intersections. The cars are lined up to travel towards the correct node but contain translations that would move them from the previous destination to the node before the intersection. A visual example of this is provided below.



A visual explaining the intricacies of problem areas on the graph.

Again, this issue was being caused by the CarMover’s StopDriving() and StartDriving() functions. This time, rather than the copy being lost before the car is told to move again, an old copy that is no longer needed is being reapplied, causing the car to continue its motion from the previous node. To fix this issue, the StopDriving() and StartDriving() functions were modified to move away from the idea of storing a copy of the translation and instead recalculate the translation every time StartDriving() is called. This would not only remove the need for checking if the car has already stopped before running the stop logic, but also prevent any old translations from interfering with current motion. After making this change, more tests were run to verify if the issue still remained.



```
//Interacts with TrafficDirector
public void StopDriving()
{
    //Set translation to zero to make car stop
    translation = Vector3.zero;
}

public void StartDriving()
{
    //Recalculate the translation
    FindTranslation(path[stepsTaken]);
}
```

The final implementation of the StopDriving() and StartDriving() functions.

This test went for over two and a half hours without any failures due to drifting. The cars finally followed the traffic rules set in place by the graph and the intersections while also avoiding hitting each other in the same lane. The test ultimately ended because another gridlock situation happened, but this one was caused because a car attempted to leave a destination directly into the left lane while another car in the right lane was in the way. Rather than add more intersections around the destinations, I limited the entrance access to certain destinations to avoid the possibility of these collisions. Further tests after this change went on for more than three hours with no issues.

8 Conclusion

8.1 Results

The goal of this project was to create an agent that relied on a simple core system and then build it up with other systems to increase its “intelligence level.” Through the various traffic control systems, this idea was accomplished via cars driving around a road network. Without the extra systems, the cars would still be able to acquire paths and drive between buildings, but they had a high chance of colliding with each other, which would display a lower level of intelligence.

It’s important to make a distinction between a car’s own intelligence and the other systems that can lead to an increase in its intelligence. The only part of the car’s intelligence is calculating the paths it takes around the level. Other than that, it simply simply follows the orders it gets from the elements of the road. The car didn’t gain the ability to detect other cars approaching an intersection and decide its best course of action. The intersection tells it to stop, so it stops. Until a collision avoidance event happens, the cars have no idea where the other cars are or even if there are other cars in the first place during normal driving. Even during a collision avoidance measure, the car only knows it was told to stop, not where it stopped, why it stopped, when it can go again, or any other information relative to traffic flow. But, when viewed holistically, the cars seem to exhibit these behaviors. This demonstrates that the goal of the project was successfully achieved.

However, this is certainly not a catch-all solution. The DestinationDirector is a good example of this. Although it is an external system, a change in the cars’ behavior was required to interface with it, unlike the TrafficDirector and CollisionAvoidance systems. This particular system needs some way to modify/control the cars’ ability to select a destination, thus modifying an element of the cars’ behaviors. That does not change the fact that intelligence levels can be increased in a meaningful way through the use of external systems.

8.2 Possible Future Developments

People driving cars around city streets in real life could result in many possible scenarios. Some drivers may exceed speed limits in order to arrive at their destination sooner. Other drivers may ignore stopping conditions if they deem them unnecessary, such as not coming to a complete stop at a 4-way intersection if there are no other vehicles present. Expanding the systems in this project would allow for demonstrating these concepts and more. Here are a few examples of areas this project could see future development.

A random chance that real collisions can occur could be introduced. As evident by the previously cited statistic, car accidents happen very frequently in real life even though there are many systems on roads to try and prevent this. Traffic lights, stop signs, turning lanes, right-of-way rules, and others all exist to try and prevent people from crashing into each other. None of these systems are foolproof, and this idea could be captured by introducing a random chance for a stop command to fail to send. For example, the TrafficDirector at an intersection could have a small chance to not send the stop command to an incoming car. Then, new behaviors and systems can be implemented to display the effects of two cars actually colliding.

The methods used to move cars around the level could be reworked. Although this change would be to the cars themselves, it would not modify the decision making and pathfinding processes, so it still fits the core idea of building up the intelligence from other factors. Car movement being handled by Unity’s translate() function works fine, but it has its limitations, such as potentially inconsistent speeds and unsmooth turns. Re-doing the movement systems with a CharacterController instead could make for more consistent movement across the board, because the current turn speeds are still dependent on the distance between the nodes along the turn. Driving could also be made in a more physics-based approach by adding forces to the car GameObjects through Unity’s physics system to direct the cars around the level.

A more robust parking solution could be created. This would increase the maximum number of cars allowed to drive simultaneously, as there would be more places for cars to park outside of each building. Also, adding contingencies for what cars should do when there are no available destinations would theoretically infinitely increase the maximum number of simultaneous cars. This could be handled with large exterior parking lots or an indoor parking lot that simply makes the cars disappear from view. From the perspective of the user, the cars are parking indoors. From the perspective of the program, the cars are being removed from the scene and reappearing when they exit the parking lot.

The user could be given more interactivity with the simulation. If the user could have some control over the state and layout of the roads, they can force the cars into different situations to test their responses, thus providing more opportunities for them to display their level of intelligence. The current systems for pathfinding and traversal are not dependent on the size and shape of the graph representing the streets. New systems could be created to dynamically open and close existing roads, or even create new ones. The graph would need to be adjusted to accommodate these changes, and the cars would be able to seamlessly interact with the new node layout.

Those are just a few ways that this project could be developed further. Even without these other features and ideas, the project as it stands demonstrates the idea that an A.I.'s intelligence can actually derive more from external factors interacting with it rather than just the A.I.'s own behaviors and decision-making skills. With this concept, future A.I. development in fields where perceived intelligence is very important can be looked at from another perspective. When presented with an A.I. problem, one can approach it by figuring out simpler external solutions instead of more complex internal ones.

9 References

1 Britannica, T. Editors of Encyclopaedia (2020, May 7). Pong. Encyclopedia Britannica. https://www.britannica.com/topic/Pong

2 OpenAI. (2019, April 15). OpenAI Five Defeats Dota 2 World Champions [Web log]. https://openai.com/blog/openai-five-defeats-dota-2-world-champions/

3 Sullivan, L. (2015, June 30). *The ingenious designs behind Doom's most iconic demons*. Official PlayStation Magazine. https://www.gamesradar.com/doom-demons-evolution/.

4 ModDB.com. (n.d.). Reviving lostsoul, Imp, Zombieman AND Shotgunguy. https://www.moddb.com/mods/reviving-lostsoul-imp-zombieman-and-shotgunguy/images/indesx-21.

5 Fandom.com. (n.d.). Demon (Doom). https://aliens.fandom.com/wiki/Demon_(Doom).

6 Butcher, C. & Griesemer, J. (2002). The Illusion of Intelligence: The Integration of AI and Level Design in Halo. Presentation, San Jose Convention Center, San Jose, California.

7 Butcher, C. & Griesemer, J. (2002). *The Illusion of Intelligence: The Integration of AI and Level Design in Halo* [PowerPoint Slides]. https://justinmeiners.github.io/shamans/papers/ai/the_illusion_of_intelligence.pdf

8 Driving-Tests.org. (n.d.). 2021 driving statistics: The ultimate list of driving stats. https://driving-tests.org/driving-statistics/