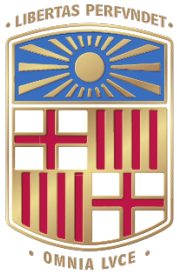
Normative and Dynamic Virtual Worlds



Universitat de Barcelona

Car ChAIse - Final report

Academic year: 2022/2023

Authors: Benet Manzanares Salor, Demetre Dzmanashvili, Cristian Andres Camargo Giraldo, Joaquim Marset Alsina

Contents

1 Introduction

Our project for the NDVW course is the development of Car ChAIse, an endless racing game where the player assumes the role of a driver pursued by computer-controlled police cars along the roads full of civilians of a city. The player's goal is to reach as many destinations as possible, while trying to escape the police and mitigate any damage to the car. Destinations are random points of the city that are endlessly generated. Every time that the player reaches one, his score is incremented and another destination appears. The police tries to chase the player or destroy its car, while avoiding the civilian cars. Police agents communicate between them the last player known location and start to patrol if the player is lost. Civilians, on the other hand, go on about their lives driving around the city, trying not to complicate the task of the police. The game's difficulty increases depending on player's score, adding more police cars. The player is assumed to be caught by the police or get his car destroyed after an amount of time, finishing the playthrough and showing the obtained score. It is expected that the player want to repeat the game and improve that score. The city is procedurally generated on each playthrough, creating new buildings and streets.

The game is developed on the Unity game engine [6]. The Car AI asset package [5] has been used as baseline for the project, significantly extending it to obtain the desired behavior for the police cars.

This document provides an overview of the development, stating a detailed description of the game elements and a definition of the implemented functionalities. All our implementation is accessible at the NDVW_CarChAIse GitHub repository [2].

2 Related work

The police car chase is a classic in many car video games focused on the suburban car culture. The most popular example of this is the Need For Speed saga, which in its open-world games includes the police chasing the player when he breaks the law (e.g., Need For Speed Undercover). Nonetheless, police car chasing is nowadays also included in many non-car-focused open-world games that include driving, as the way of the law forces to catch the player efficiently. This is mainly represented by the well-known saga Grand Theft Auto (GTA). Both kinds of games represent the police chase similarly. The chase starts when any police see the player acting illegally or as a potential suspect. All the police in the area will start to chase the player, trying to stop it and/or destroy its car. The player's objective is to make the police lose sight of him. Usually, as the chasing time increments, the police will start to employ more resources for stopping the player, such as barriers, better cars, or helicopters. The chase ends when the player has been out of the police's sight for a predefined period of time or the police manage to stop the player or destroy its car. Frequently, the police completely forgets about the player once it has been out of sight enough time, not restarting the chase when they see her again (except if it's breaking the law).

Due to the popularity of car games, there are multiple online resources for its creation. For the Unity game engine [6], most of them are behind a paywall. No assets are found for car chasing, probably because chasing systems are very game-specific and there is no point to use an external package. Considering the free assets, we highlight Car AI [5], which provides a very simple automatic car behavior capable of random patrolling and driving to a custom destination. Random patrolling is driving to randomly generated waypoints, creating the new one once the last is reached. The custom destination is a predefined position that the car needs to reach. Unity's NavMesh [7] is leveraged for getting the random waypoints for the patrolling and computing the pathfinding from the car to that waypoints or the custom destination. Car's driving is fully automatic, and physics are based on Unity's WheelCollider component [8], which provides quite realistic wheels' behavior. The package is very limited, requiring several modifications and extensions for our project. For instance, automatic driving is basically always accelerating and pointing the wheels to the next waypoint, not considering breaking for sharp turns or backward movement. Moreover, the code is only prepared for a single custom destination, if a new destination is given or the original is moved, the path will not be updated. Finally, no auxiliary systems are defined for comfortable driving (such as stability or breaking control, see Section 4), so the car is only well-behaved at low speeds. However, the simplicity of the code facilitates its editing, serving as a proper backbone for our project.

3 City

3.1 Specification

At the beginning of every game session, the city will be stochastically generated to offer the player a "new" and interesting experience each time.

The first step will be to create the city roads, followed by the buildings that fill the gaps between them, in a way that the resulting game world is big enough to offer the player a variety of options but not too big to make the gaming session too long or take up too many resources. Afterwards, the initial player position and the target destination will be decided, making sure that they are placed on opposite sides of the city, thus forcing the player to traverse the entire scenario.

The police cars will be distributed in such a way that a car chase will always take place at the start of the game. If necessary, new police cars will be spawned near the player's surroundings (out of sight) to make them feel in danger at all times. Civilians will be randomly generated throughout the city, ensuring that the player keeps paying attention in order to dodge them.

3.2 Road generation

Since the last sprint, we have changed the way we generate the city roads. Before, we first placed the buildings in a grid and then the roads in the spaces in between. In Figure 1, you can see an example of how the road generation worked in the previous sprint. However, we found that this did not offer a fun experience when actually playing the game.

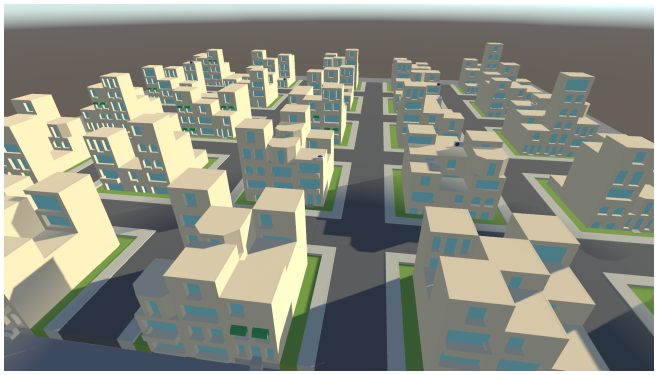


Figure 1: Old road generation

For this reason, we decided to also procedurally generate the roads by using an L-System [1]. An L-System is a parallel rewriting system that builds a tree-like structure by recursively modifying an initial string (called axiom) using a set of rules. To build this system, we need to define a vocabulary and the rules that will rewrite our axiom. The rules are composed of an antecedent and a consequent, and when we encounter the antecedent, we replace it with the consequent. As you may notice, even though we can procedurally generate a new string, we are always bounded by the axiom and the rules, so we cannot consider it totally random.

For example, we may have the vocabulary $\{A, B\}$, the set of rules $\{(A \rightarrow AB), (B \rightarrow A)\}$, and the axiom A. Starting from the axiom, we would check in the set of rules for some antecedent containing A, and we would substitute it by AB. We would recursively perform the same with each symbol in the new string, generating ABA at the second level of depth (first applying the first rule to A and then the second to B). We would call again recursively with ABA and generate a new string. This process will continue endlessly unless we set a depth limit that we consider. In Figure 2, we show the tree that we would build with a maximum depth of 5 by applying the system to the axiom.

[1] https://en.wikipedia.org/wiki/L-system

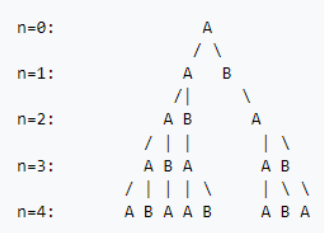


Figure 2: Example of application of the L-System

This algorithm can generate fractal-like structures, like the one in Figure 3. A fractal structure is a structure that is self-similar on all scales, and if you zoom in on a particular place, you will see the same again. Of course, this depends on how we let the recursion grow and the rules to rewrite the axiom. As you can notice, this type of system is used to generate trees. However, we can also use it to generate our roads procedurally.

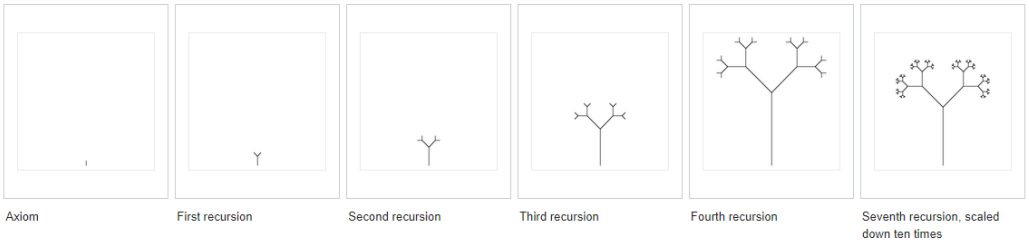


Figure 3: Example of fractal structure

However, this algorithm cannot help us generate the roads unless we give some special meaning to the symbols. And this is where turtle graphics can help us [2]. Turtle graphics are a form of vector graphics (i.e., images created from geometric shapes like points, lines, or polygons) that use particular commands to generate the graphics. For example, we may have a command that says "turn left 90 degrees", and this means rotating the cursor we use to move 90 degrees left. When the cursor moves, it draws, so we can see how we can generate images by allowing different commands and executing them in a particular order.

Thus, once we rewrite the axiom into a different string, we need to traverse it to apply the real meaning of those symbols. In particular, we have considered a subset of the symbols defined in [3], which follows the same idea of turtle graphics. One of those symbols refers to moving forward by a certain length and drawing a line while moving. Like [3], and to facilitate the process, we only consider drawing lines. In our context of road generation, drawing a line means placing a straight road tile. Therefore, we will have to swap the straight roads with curves, intersections, or dead ends when needed. Below, we define the symbols we have considered:

- "F": Place a particular number of straight roads tiles (the number is a parameter that can be modified)
- "+": Rotate the direction we move 90 degrees to the right
- "-": Rotate the direction we move 90 degrees to the left
- "[": Save the current cursor position and direction in a stack
- "]": Restore the last stored position and direction in the stack

Like in turtle graphics, we move through our Unity scene, placing roads. This is why we need to rotate the moving direction to avoid an endless straight road. Do not get confused. The rotation here does not mean rotating the road tile by some degrees but rotating how we move. Imagine the classic snake game, moving through a map and placing road tiles. We compute the rotation of the road tiles when placing them.

[2] https://en.wikipedia.org/wiki/Turtle_graphics

We use the other two symbols to avoid always following the same path but allowing to start from a different place. For this reason, we use a stack to save and restore positions when we encounter those particular symbols.

Regarding the axiom and the rules, [3] defined them as $F--F$ and $F \rightarrow [+F][-F]$ respectively. As you can see, these two generate completely symmetric structures. It is nice to generate a tree, but not for our purpose. Thus, we need some stochasticity. In our project, we have applied three mechanisms:

- Add more rules: In particular, we modified the one from the book by adding an extra "F" in between ($F \rightarrow [+F]F[-F]$), which allowed to place a third road and open a new path to continue. We also created a third one by changing the plus and minus signs.
- Randomly select the consequent: As you may have noticed, all the rules use the same antecedent. For this reason, we choose the consequent randomly to allow for some variability.
- Ignore a rule with certain probability the user can tune. That is, not applying the consequent.

We have said that we only place straight roads when applying the symbol meanings. But, as we have also said, we need to fix not only the rotation, but also the tile. Fixing the tile takes into account the neighbor roads a road can have. And for that, we store all the placed roads in a set. Then, depending on the number of neighbors, we place one or another (e.g., to place a four-way intersection, we need four neighbors). To fix the tile rotation, we focus on the neighbors' position and the current road position to rotate it accordingly.

As the L-System is based on a tree, the leaf nodes will become road tiles with one end disconnected. For this reason, we have decided to place a special tile representing a dead end, which allow us to re-enter the cars into the map.

In Figure 4, we show an example of an execution of the procedural road generation. As we mentioned, we need to limit the recursion, so we have set it as a parameter the user can tune. By changing it, we achieve a bigger or smaller map.

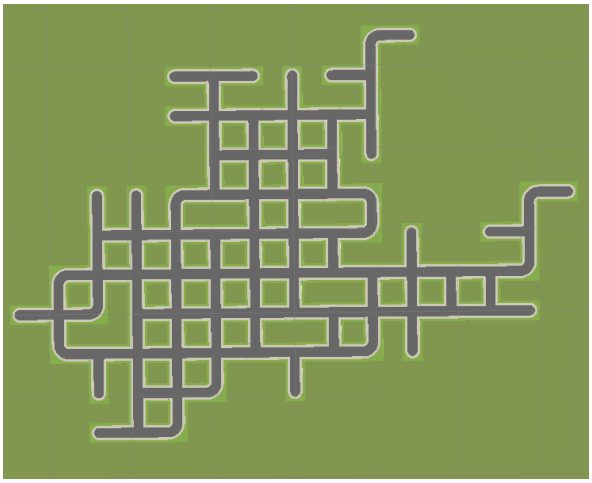


Figure 4: Example of procedurally generated roads

3.3 Building generation

In order to build a city with a dynamically changing appearance, we used Perlin noise which has been a popular approach for procedural content generation and is typically applied to 2D/3D landscape generation tasks in the context of game development. In our case, the Perlin noise will allow us to calculate the appropriate height for the buildings that will be generated throughout the level. We are also taking advantage of the Modular Buildings free asset pack [4] in order to construct visually interesting buildings block by block.

We start by generating an empty 2D texture of size 512x512 that will act as a noise map for calculating how the structures should be placed later on. Afterwards, we iterate over each pixel in the texture map and compute

the corresponding Perlin noise value using the Mathf.PerlinNoise function provided by Unity, and store the result for later reference. With this done for each pixel in the texture, we have effectively generated a noise map that will serve as the basis for our random building generator script.

Once this noise map has been generated, we proceed to iterate over a list of Vector3 positions that will describe the grid cells in the game world that are not currently being occupied by a road tile. This list is populated during the road generation process and before the actual building generation takes place. Our goal in this step is that, at all times, any particular grid cell will contain either a building cluster (i.e. a group of multiple buildings) or a road tile that will fill the entirety of the cell's region.

When it comes to the building clusters, each of them is essentially a grid in itself that will contain 2x2 individual structures. Each of these structures can be thought of as a sort of "tower", made up of different blocks stacked on top of each other so that it reaches a certain height H, which acts essentially as the number of blocks it will have (so H = 2 would mean two blocks). In order to determine this height value, we proceed to sample the Perlin noise value that corresponds to the structure's location in the game world, which basically means we take the value of the pixel from the 2D noise texture at the building's current position. This sampled value is then multiplied by the MaxPieces setting, which is the maximum height allowed for a building, and so we obtain the final height H of the building.

Having calculated the resulting height, we proceed to select random pieces from a pool of available blocks which are divided into three layers: base, middle and top. This is because some blocks would be out of place visually if placed on a particular layer, such as a roof tile on the bottom-most part of a building, so it makes sense to split them according to their structural role. After placing a block, we calculate the offset in the Y axis that is necessary to place another block directly on top and continue with the process. Once we're done picking H blocks in total, the building is complete and is now part of the game world. We continue to do this for the rest of the buildings in the cluster, then for the rest of the clusters in the grid of world points, and so we obtain a whole city in the process.

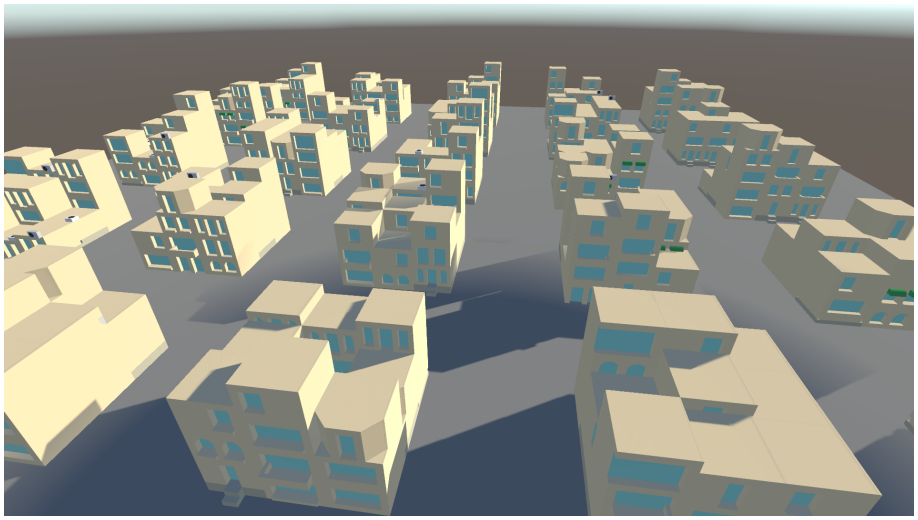


Figure 5: Example of building generation

In Figure 5, we can see an example of the city generation in action. Here we have selected a grid of size 9x9 to be used for the building clusters and each of the individual clusters is composed of 3x3 structures of max height 6. Since we are leaving an empty grid cell between buildings to act as a gap for the roads that will be placed later on, this means that we are looking at a total of 25 building complexes. By using some of the modular building assets, we manage to generate buildings that are visually different from one another during each execution of the algorithm.

On the other hand, Figure 6 shows how the building generation works alongside the road generation. Notice

how the building clusters are neatly distributed along the gaps and sides of the roads, seamlessly fitting in and giving the city a more fleshed out appearance than in the previous example. Also note how we have expanded the building block selection to obtain more visual variety in the generated buildings.

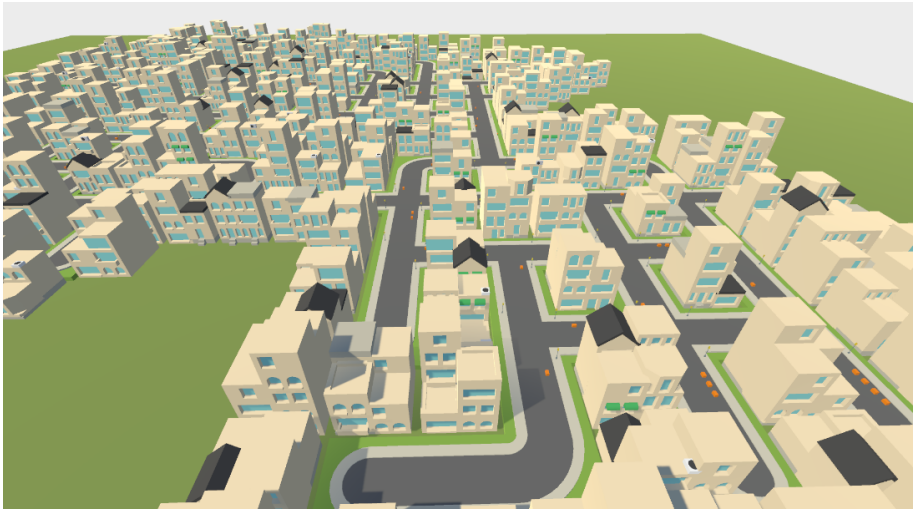


Figure 6: Generating buildings alongside the roads

4 Car physics

In order to offer a more interesting experience and development, our game includes some realistic car physics. These physics affect player, police and civilian cars alike, having to be taken into account for user and AI controls. In the following, background on that physics and our simulation approach are defined. Details on the usage of these physics are provided in Section 5.

Our realistic physics are mainly grounded on the wheels' behavior. Wheels are the car element in charge of transmitting the force of the engine and the brakes to the ground (propelling or stopping the vehicle) and applying steering (changing movement direction). The engine's force arrives to the driving wheels in the form of torque, measured in Newtons per Meter (Nm). For illustration, if a stick of one meter is attached to the engine's axis, the force that an object receives at the end of the stick is the torque (X Newtons * 1 Meter). That torque makes the wheel rotate and, if it is touching the ground, it transforms that torque into force. Concretely, the multiplication of torque by the wheel radius returns the force applied at the wheel's contact point. Not all the force applied at the wheel's contact point contributes to the car's propelling. Wheels have a static and dynamic friction coefficients with the touching surface, the first being greater than the second. These coefficients determine the proportion of force "transmitted to the ground" (never reaching the ideal 1). If the applied force applied by the wheel at the contact point exceeds the maximum possible with static friction (i.e., maximum adherence point), dynamic friction is applied, what is observed as a lost of grip that causes the tire to slip. Subsequently, the proportion of force that contributes to the car's propelling depends on the wheel's material, the touching surface material and the amount of force. For example, applying too much torque to a wheel in a bad condition or in contact with a low-friction surface (e.g., ice) makes it slip and even burning the tire. The propulsion force of all the wheels is summed and, using the classic Newton's equation (F=m*a) the acceleration of the car can be obtained.

This problem of slippage caused by propulsion applies exactly the same to braking. Brakes apply a torque (and subsequent force) in the opposite direction to the wheels advance, aiming to reduce the car's speed. If that braking force is too high, the wheels can lock, so the force "transmitted to the ground" depends on the dynamic friction coefficient. Since the dynamic coefficient is lower than the static one, the braking distance is greater than when the wheels are not locked. Apart from the increased braking distance, wheel locking when braking can be an even bigger problem if it affects the steering wheels. Locked steering wheels only have friction in the

direction of the car's movement, so turning is not possible. In the critical case of trying to avoid a crash by braking and turning the steering wheel, locking causes the car to take longer to brake and not turning at all. This is the reason why modern cars include an Anti-lock Braking System (ABS). This system detects when the wheel is locked (or close to lock) and reduces the braking force for allowing it to continue rolling.

Apart from the aforementioned forward friction, wheels have a sideways friction. This mainly determines the car's steering ability (without considering braking lock). With others static and dynamic friction coefficients, this sideways friction determines the maximum steering possible. If the sideways maximum adherence point is exceeded, the wheels' slip can be observed as drifting.

We simulate the wheel's physics using the Unity's WheelCollider component [8], as it was the case for our base asset Car AI [5]. This component abstracts a single wheel, including the collision detection and a slip-based tire (forward and sideways) friction model. It allows setting the desired acceleration and breaking torques at each frame, automatically computing the force transmitted to the ground (if touching it) and applying it to the RigidBody in a parent GameObject. WheelCollider also simulates the suspension and the yaw, pitch and roll rotations of the car. The friction model is based on the slip of the tire, which is proportional to the speed difference between the car and the tire's rubber. This slip value is transformed to the force exerted on the wheel's contact point by the friction curve function. This friction curve is determined by the user-defined extremum and asymptote points, which roughly represent static and dynamic friction coefficients. Figure 7 depicts the typical form of the friction curve.

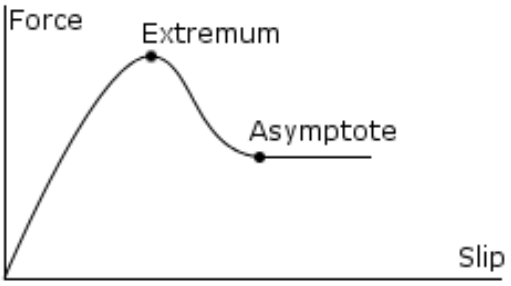


Figure 7: Wheel friction curve of Unity's WheelCollider

Our physics simulation also includes air resistance. We use the real drag force function (see Figure 8), that obtains the force applied in the opposite direction to the car advance based on air density, car speed, car drag coefficient and car frontal area. All these parameters are known, except for the car's drag coefficient and frontal area. We use 0.3 as drag coefficient and 1.89 as frontal area, which corresponds to the best-selling sports car mentioned in [1].

$$F_D = \frac{1}{2}\rho v^2 C_D A$$

Figure 8: Drag force function. $p = AirDensity, v = Velocity, C_D = DragCoefficient, A = FrontalArea$

5 Car controller

For defining a common framework for the player, police and civilian cars, the *CarController* script has been developed. It implements the physics described in Section 4, defining the logic for accelerate forward, accelerate backwards, brake, steer and air resistance. In addition, it provides some driving assistance methods (e.g., ABS) and implements a health system. This controller was inspired by the Car AI asset [5]. The car is defined as a GameObject with a RigidBody, a BoxCollider and four WheelColliders [8] at the bottom. All the wheels provide traction (all-wheel drive) and braking, and the two wheels at the front allow to steer, rotating around the Y axis.

CarController is designed as an abstract class from which the rest of car-based game elements inherit (see Figure 9). Details on the *Police* class inheritance from *CivilianController* will be provided in Section 8.

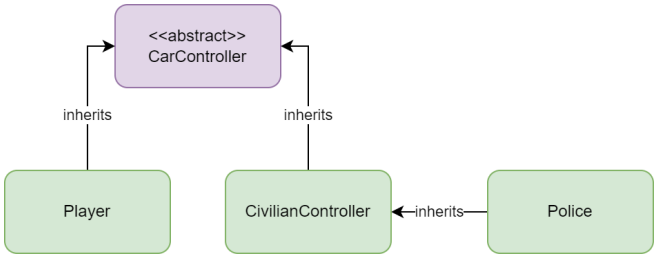


Figure 9: Class diagram for the three car controllers

The following subsections define the three main types of functionalities: movement, steering and health.

5.1 Movement

Movement includes forward acceleration, backward acceleration and braking. For forward and backward acceleration, a positive or negative (respectively) acceleration torque is applied to all the wheels (i.e., AWD or 4x4). The car is braked by applying a braking torque to all the wheels. No simultaneous braking and accelerating is allowed. Concretely, the car will automatically brake instead of accelerate when the car is going in a direction (forward or backward) and an opposite acceleration is desired. A maximum acceleration and braking torques (the second assumed to be grater) are predefined for the car. The abstract methods to implement by the inheriting class define the proportion of these maximum torques to apply at each FixedUpdate (being 0 a valid value). Every car will have a predefined maximum speed (e.g., 200Km/h), not being possible to accelerate more if it has been reached.

To ensure the steering of the car while braking and incrementing braking effectiveness, an Anti-lock Braking System (ABS) has been developed. Instead of from the classical "stop braking when locking" approach (see Section 4), our method stops braking when a wheel is going slower than the average wheel speed. This improves the steering-while-braking and reduces the drifting problems, since it minimizes the difference between wheels' speeds. Concretely, each time that the braking method is called, the wheels speeds are checked individually, the average speed is obtained and braking is only applied to the wheels rotating at a speed notoriously lower than the average. The forward and sideways friction of the wheels is independent of the *CarController*, so our ABS has to behave independently of the wheels' configuration, as in real life.

For simulating the energy losses due to air friction, the drag force defined in Section 4 is applied each FixedUpdate to the RigidBody, in the opposite direction to the current velocity. Subsequently, the car will lose speed when no acceleration is provided, depending it on the square of the current speed (see Figure 8).

5.2 Steering

Steering rotates the 2 front wheels around the Y axis for changing the car's angle. A maximum angle is defined for these rotations, being possible to rotate the wheels from -Maximum (steering left) to +Maximum (steering right). The inheriting class can control the wheels to steer left or right any angle below or equal to the maximum.

The roll and pitch rotations of the car simulated by the WheelColliders result in several stability problems that would require a very complex stability system to avoid. As a consequence, currently only the yaw rotation (the resulting from steering) is allowed, freezing the rest of rotations around the X and Z axis in the car's RigidBody. This avoids problems with flipping cars, but limits driving to completely flat surfaces, without slopes.

5.3 Health

Health is a metric of the car state, starting with the predefined maximum value (100 by default), decreasing with crashes and, when it gets to zero, the forward and backward acceleration, braking and steering are disabled. A crash is detected when the OnCollisionEnter method is called (i.e., an object touches the car's box collider),

applying a damage proportional to collision speed. This collision speed is the dot product of the velocity difference with the collided object and the collision vector. In the next paragraphs, the calculation of these two vectors would be explained. The damage resulting from the collision is the multiplication of the collision speed by the DamagePerKph setting, which can be modified in the editor. For instance, if DamagePerKph=0.3, for every km/h of the collision speed, 0.3 points will be decremented from the car's health. Using DamagePerKph=0 disables crashing damage.

The velocity difference between the two objects is the subtraction of the two velocity vectors. *CarController*'s velocity is obtained from the velocity property of the RigidBody component (i.e., RigidBody.velocity). This velocity is stored in a class variable named CurrentVelocity in every FixedUpdate. For the damage calculation in the OnCollisionEnter method, we employ this variable instead RigidBody.velocity because the latter is already updated by the collision, and we need the velocity previous to it (stored in CurrentVelocity). In the case that the other object has no component child of CarController, but has a RigidBody component, the RigidBody.velocity (despite being updated by the collision). Otherwise, a velocity of zero is assumed (e.g., if colliding with a wall).

For the collision vector, we consider all the collision points, compute the vectors from the car's center to that points, compute the average collision direction and discretize it. Discretization consist in selecting which of the 4 directions around the car (forward, backward, right or left) is closest to the average collision direction.

Figure 10 depicts an example of a collision between two CarControllers. Yellow arrows represent the cars' CurrentVelocity, the star depicts the collision point, the purple arrow is the velocity difference and the red arrow is the collision direction (discretized to Car 1's forward). It's easy to observe that the dot product between the velocity difference and the collision direction (i.e., collision speed) would be very similar to the magnitude of the velocity difference, since both vectors are almost aligned. That collision speed would be equal for both cars, since the operations are the same. The multiplication of that collision speed by the specific car's DamagePerKph setting will result in the damage received.

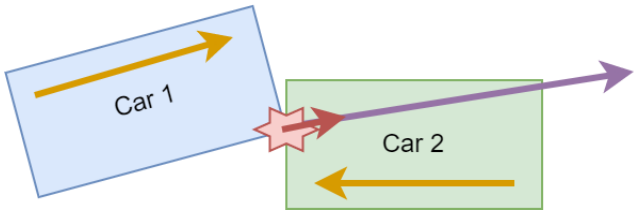


Figure 10: Car collision example

6 Player

6.1 Specification

This is a car agent under the player's control, which will have to escape from the police and reach as many destinations as possible without being destroyed or caught by the police. Destinations are generated in random locations of the city. Each time that a destination is reached, a score counter is incremented proportionally to the initial distance from the player, and a new random destination is generated. The player's car is considered destroyed when its health reaches zero (see Section 5.3), being this caused by collision with walls, police or civilian's cars. The player is considered caught it is going at low speed (i.e., stopped) and there are police cars around. More details are given in Section 8. Both in the event that the car is destroyed and in the event that it is caught by the police, the game finishes, showing to the player the end reason (destroyed or caught) and the obtained score.

Car control should be accessible for the player, while complex enough for offering an interesting experience. Parallelly, the player advantage over the police skills determines the game difficulty. If the police competence is comparable to that of the player, his only option to flee will be to cheat on the trajectory to follow (e.g., pretending to turn right and then going left). But that may be practically impossible if we take into account the numerical superiority of the police, making the game too difficult. Therefore, the player has two technical

advantages over the police: better acceleration and hand brake. The acceleration advantage allows the player to distance himself from the police after the turns. The hand brake, brakes the back wheels (not being possible to activate ABS, so maybe they lock) and reduces their grip. If this is done while steering, the car will drift sideways while reducing speed abruptly. If controlled, this allows to do sharp turns faster than it would be possible for a police car. Otherwise, leaves the car in a vulnerable position (i.e., stopped and perpendicular to the road). This mechanic is considered to give an interesting risk-gain trade-off.

The screen should show all the relevant information to the player, such as the car health, indications for the current destination, current score and a third person view that allows comfortable driving.

6.2 Implementation

We created a *Player* class as a child of *CarController* that allows to manage the car using the keyboard. The movement is controlled with the "W" (or "Up arrow") and the "S" (or "Down arrow") buttons. Being the first for accelerate forward (or brake if going backwards) and the second for accelerating backward (or brake if going forward). Steering is controlled with the "D" (or "Right arrow") and "A" (or "Left arrow") for going right and left, respectively. We defined the maximum speed to 100 km/h and movement torque to 2000 N*m. We defined the appropriate forward and sideways friction curves for the wheels to make driving feasible without constant oversteering or understeering.

The hand brake mechanic is activated with the "Space" button. The brake applied to the back wheels corresponds to the maximum possible for the car, usually locking them. The grip reduction is done by multiplying it by a factor (e.g., 0.2). When the hand brake is released, the original grip is restored. This grip reduction is done for facilitating the drifting, since braking the back wheels is usually not enough for it.

In Figure 11 a frame of the player's Point Of View (POV) is depicted. As can be seen, the car is controlled from a third-person camera behind the vehicle. To ensure that this camera always shows what is relevant to the player, it rotates around the car in line with its speed. For example, if the car is going backwards, the camera is positioned in front of the car pointing backwards. In this way, it shows the car and what is in the direction of movement. Alternatively, the mouse's right click (or left Alt key) can be pressed for rotating the camera *oppositely* to the movement direction, what can be useful in certain pursuit situations. The Heads-Up Display (HUD) shows, at the bottom right, the car's speed (i.e., 52 Km/h) and health (i.e., 93%). A rear camera is displayed at the top center of this HUD to let the player know what is behind him, having the rear spoiler as reference. Below the rear camera, we can observe an indicator of how close the player is to being caught by the police (see Section 8 for details). It is a bar drawn from left to right (in the figure it is at a 25%) and below shows the message "Escaping" if it is decrementing or "Escape!" if incrementing. On the center right of HUD there is an arrow that points to the current destination (represented as a yellow semi-transparent column). The angle for this arrow is that between the camera forward vector and the vector from the car to the destination. The Y axis is ignored for this angle computation. Just below the destination arrow, the current score (i.e., 162) is depicted. It is not shown in this figure, but when the player loses (it is destroyed or gets caught) this HUD is substituted by one that notifies the game over reason and shows the obtained score. The car model is a free-to-use version of a Subaru Impreza 2007 that has a reduced file size (less than 1MB) and a good-looking aspect. This model includes the four wheels, which have a rotation speed that matches the WheelCollider and rotates for steering, providing a clear feedback.

7 Civilians

The civilian is another AI-based agent, aiming to traverse the city while safely driving to no particular destination and without any other purpose. Safe driving includes respecting traffic lights. Moreover, the civilian car has to take measures to avoid any dangers to its existence, such as trying to avoid collisions with other cars. Finally, the civilians move away when they listen to the police on their back trying to cross the street, easing the police's work of chasing the player. Civilians are not harmed in the event of a collision.

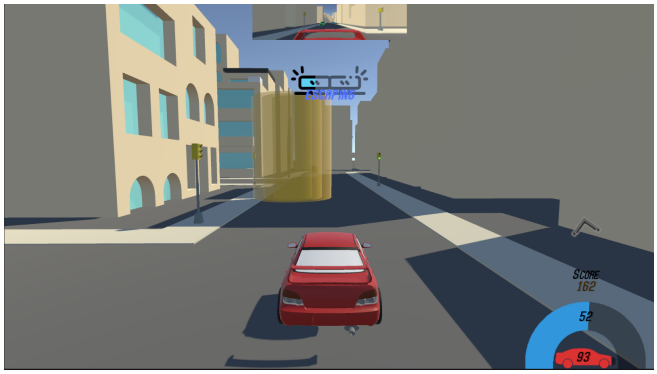


Figure 11: Player's POV

7.1 Civilian Controller

As another type of car in the game, we have leveraged the already implemented *CarController* to handle the driving, steering and braking of the civilian cars. However, we have created a child class, called *CivilianController* because we have to handle slightly different the way it steers and drives forward or backward. For example, the civilian cars drive at a lower speed than the other cars in the game, and to avoid drive away for their pre-defined path (that we will explain below), we need to halve the speed when turning. Also, the conditions upon the car drives backward are different from the other cars.

7.2 Civilian AI

We have created the class *CivilianAI* to handle the AI of the civilians. In particular, we have implemented it as a behavior tree that we show in Figure 12. The logic of the civilians is quite complex, and handling everything in the same script was impossible. Also, the police (but not the player) will reuse some of this logic when not chasing the car. This is why, we have defined multiple scripts that we consider as sub-behaviors, to handle the different conditions and actions of the BT. Thus, this *CivilianAI* script only contains the execution flow of the BT, and calls the needed script when needed.

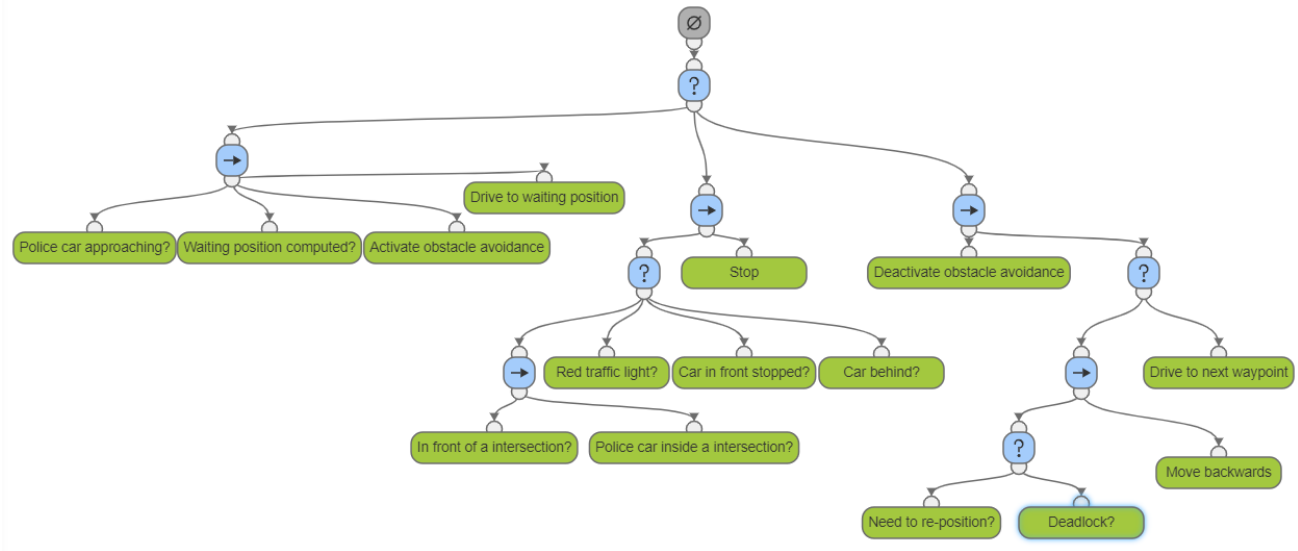


Figure 12: Civilians Behavior Tree

1. If the civilian hears a police car approaching, it tries to compute a position to wait at the road shoulder. If there is a position, it drives towards that position while trying to avoid the other cars that are also waiting. To avoid cars in this situation, it activates a set of sensors.

2. The civilian stops if one of the following conditions happens: a red traffic light; a car stopped in front while driving forward; a car behind while driving backwards (stopped or not); or a police car inside an intersection when the civilian is trying to enter it.

3. If none of the previous happens, the civilian drives towards the next waypoint. However, it deactivates the sensor to detect obstacles and uses a FOV to detect car collisions. In case it is trying to drive towards the next waypoint, and encounters a deadlock, or is perpendicular to the road and needs to re-position itself, it drives backwards until not being needed anymore.

Below, we explain the different scripts that we consider sub-behaviors. As mentioned, each one is in charge of certain conditions and actions from the BT that we have grouped together. For example, all the police avoidance-related functions are grouped in the same script. If something from the BT is not clear, we properly explain it in the corresponding sub-behavior.

7.2.1 Move to Waypoint Sub-Behavior

In the beginning, we decided to use [5] as the base for the civilians' endless loop driving. The problem was that making the car follow the correct road's lane was difficult using the path generated with the NavMesh. We had to create a surface for each side and place modifiers to avoid driving through the road shoulder. For this reason, we have simplified the solution and made the AI follow a set of pre-defined waypoints we place on each road. We connect the waypoints of the same lane of adjacent roads and generate an endless loop the civilians can follow. In Figure 13, you can see an example of a curve road with these waypoints, in which all the waypoints of the same color are connected:

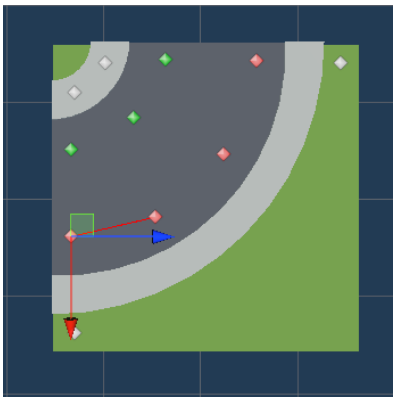


Figure 13: Curve Road Prefab

7.2.2 Stop to Traffic Light Sub-Behavior

We have traffic lights at the three-way and four-way intersections, given that it is the easiest way to handle priorities. We use trigger colliders to make the cars stop in front of a traffic light. In Figure 14, you can see an example of a four-way road with the mentioned triggers.

Therefore, when a car enters the trigger, we will make it stop. In a normal situation where the cars are simply driving, only one will enter the trigger and stop because of the traffic light. Those going behind will stop to avoid a collision with the car in front, rather than detecting the red light. But if the civilians need to let the police go through, they will move to the shoulder, so we can have more than one car entering the trigger. That is why we are not deactivating the trigger collider at the first trigger as we did before, ensuring all the cars stop. It is maybe not very clear what we mean by moving to the shoulder, but we will clarify it when explaining the police avoidance behavior.

To handle the traffic light rotation, we use a co-routine that periodically changes the green traffic light, deactivating its collider while keeping the rest activated. Therefore, every 10 seconds, for example, a different traffic light will be set to green, deactivating its collider to allow the cars to cross the intersection.

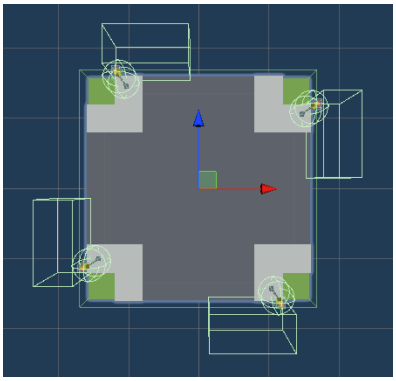


Figure 14: Four-way intersection Road Prefab

7.2.3 Avoid Collision Sub-Behavior

This behavior detects cars in front and behind (whatever type of car in our game, being stopped or not) to stop the civilian car for avoiding collisions. To detect other cars, we search for the colliders overlapping a particular sphere inside the Field Of View (FOV) of the civilian. Therefore, even though we consider the whole sphere, we filter those outside the FOV. In Figure 15, we present the mentioned spheres and FOVs, which are different for each type of collision.

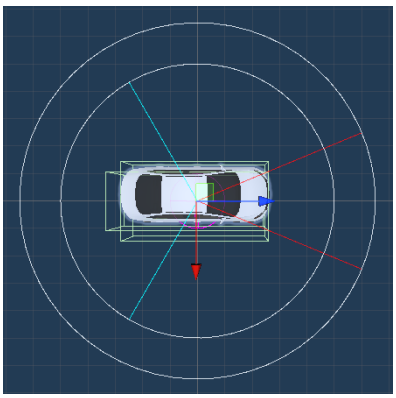


Figure 15: Civilian forward (red) and backward (cyan) FOV

We are considering a small FOV when the civilian drives forward because we want to avoid detecting those in the other lane. It can happen when turning in some intersection. And we are considering a bigger sphere radius given that we want to ensure the cars leave enough distance when stopping and do not bump into another in front (like in real life when we brake in advance considering the braking distance).

Moving backward happens when the civilian has moved to the road shoulder to let the police go through and wants to resume its normal movement. When that happens, it usually ends perpendicular to the road, so it has to move backward while steering to position itself parallel to the road to continue driving. During that time, other cars can pass through the road, so we want to avoid collisions. We are considering a bigger FOV because it has to steer, and we want to ensure we capture all the possible colliders into which the car can bump. And we are considering a smaller radius given that we do not want to detect those in the other lane, and we do not want to block the cars in the shoulder a lot of time (like in real life when we let a bus pass after stopping in a bus stop).

Another situation where we use backward driving is when we face a deadlock. A deadlock can happen in two situations. The first is when a civilian faces another one that, for some reason, is going in the wrong road lane. It can happen if the player bumps into that other civilian, displacing it, and has to return to his lane. In that case, both civilians will detect a car in front, and no one will move. Thus, we decided to check if a we detect another civilian in front that is facing the first one. When this happens, we make them drive back to remove

the deadlock. The second situation is when a civilian tries to leave the road shoulder by driving backwards, and another behind drives normally (i.e., forward). The first will detect the second behind, and the second will detect the first in front. No one will move again. In that case, we make the one driving forward to drive backwards to let the other re-enter the lane.

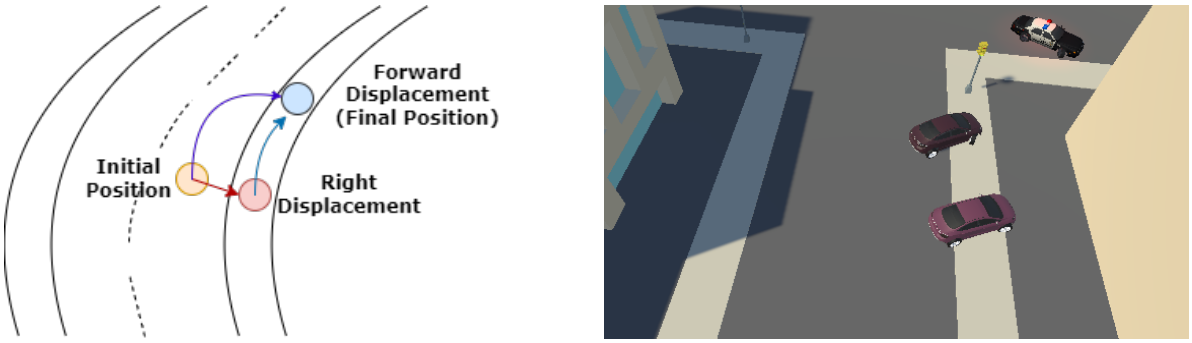
7.2.4 Avoid Police Sub-Behavior

This behavior handles the detection of a police car that chases the player and generates a position on the road shoulder for the civilian to wait and let the police cross the road easily. This behavior works together with the next one.

To detect if the police are approaching, we currently place a big capsule collider in the police cars and imitate what we saw in the lab about the hearing sense. This simulates what happens in real life when civilians hear the police siren and/or see the police lights.

When that happens, each civilian detects the type of road it is currently driving on. We need to know the type, given that we do not allow them to stop in a four-way intersection and only on one lane of a three-way intersection. If the car is on a road or lane where it cannot stop, they continue driving until the next one to find a place to stop if they still detect the police. If the car does not detect it anymore, it returns to driving normally.

Therefore, let's assume the car is on a road where it can stop, and it needs to stop. If you go back to Figure 13, you can see four white markers on the shoulders (two on each side). Those indicate the shoulders, as well as indicating the road's limits. Thus, we compute the waiting position by considering the current car's position, laterally displace it towards the correct road shoulder, and adding a certain forward displacement. In Figure 16a, we depict the process visually.



(a) Waiting position computation

(b) Waiting position visualization

Figure 16: Examples of how to compute the waiting position, and how the civilians wait at the shoulder road

So, let's assume we have the position computed, and the car wants to drive there. It is possible that it is already occupied by another car, as there are no fixed or pre-defined positions. In that case, we try to compute it again in the next frame while continuing driving. It is possible that the car steps on another road, so we repeat the process of detecting the road. And maybe the car stops detecting the police, ending this behavior.

On the other hand, if the initial position is empty, it will drive towards there and remain stopped until it does not detect the police anymore. Then, it will drive backward if necessary to position correctly and return driving towards the next waypoint. In Figure 16b, we can see how the cars move to the shoulder when detecting the police, and remain stopped.

Finally, another component this behavior has is blocking the intersections when a police car is inside, not allowing any civilian to enter until the police car is outside. We use the same triggers used for the traffic lights, activating them all and stopping each detected car. We will not enter in detail how we manage those triggers from two different behaviors, but basically, each trigger has two possible working modes, and we change them

when appropriate. Of course, when we block the intersection, the traffic light rotation is not allowed to modify the triggers until the police car has exited the intersection.

7.2.5 Obstacle Avoidance Sub-Behavior

We use this behavior in combination with the previous one to handle police avoidance. However, we decided to separate it because it is a component of *CivilianController* instead of *CivilianAI* like all the previous sub-behavior scripts. This sub-behavior adds a set of sensors (with a limited range) at the front and right side of the car to detect cars stopped at the shoulder. We only use this behavior once we have computed the waiting position to guide the car there (or as close as possible to that position). We decided to use sensors because we cannot divide the road in pre-defined positions, as we would waste space, so we prefer using sensors to try to fit as many cars as possible on the shoulders. Therefore, if one sensor detects another civilian, it will steer in the opposite direction to avoid it, moving this way toward the destination while avoiding the others. In Figure 17, we depict the sensors we place on the car. We decided not to add them only in front but also at the right to ensure we do not collide sideways. Maybe it is not the best configuration, but we found it useful enough for the current features.

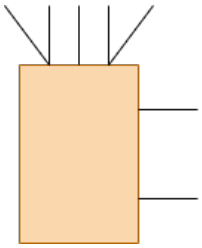


Figure 17: Civilian sensors

7.2.6 Re-Position Sub-Behavior

This sub-behavior is used to check for the condition when the civilian is stopped at the shoulder after the police have crossed the road. In that case, we check if the car is perpendicular to the road and needs to position correctly. If it does, then we make the car drive backward while steering until being parallel to the road. Then, it returns to driving forward to the next waypoint.

8 Police

8.1 Specification

The police car is an intelligent agent whose main purpose is to catch the player or at least disrupt their efforts to reach the goal destinations. When any police car has a visual of the player, it communicates the new known position to the rest, and everyone chases him. If the player has not been visible by any police car for a pre-defined amount of time or the last known position is reached by some police without seeing the player, it is considered lost. When the player is lost, and everyone will start to patrol, behaving as civilians. If the police cars have been patrolling for too much time, they are relocated to positions close to the next player's destination. In this way, the player is likely to be seen again when trying to reach the destination. The player is considered caught if he is going at low speed with police cars close for a long period of time. For example, if the user is stuck in a wall with a police car behind that prevents the escape. As the player improves the score, more police cars will be spawned in order to up the difficulty of the game. The police will always try to avoid civilians. Police cars do not receive damage in the event of a collision.

8.2 PoliceManager

We created the *PoliceManager* script for controlling the police spawning, communication of player position or loss, chasing, relocation, start to patrol and incrementing difficulty. To this end, it abstracts the logic of police cars (defined in the next subsection) to: patrolling, reach a target position (i.e. the player), communicate visual contact with the player and communicate that player is not seen at last known position.

At the start of the game, a predefined amount of police cars is randomly placed along the streets of the city. The markers defined at road generation (see Section 3.2) are used to this end. To avoid collisions, only markers not occupied by other cars are considered. The spawned police cars are ordered to patrol.

Every frame, all the police cars check if they have visual of the player (see next section for details). If someone has visual contact, it notifies the position to the *PoliceManager*. At the following frame, all the police cars are told to catch the player. In particular, a position around the player (i.e., front, behind, left or right) is assigned as target position to each police. This is done in a round-robin manner, so every 4 police cars the target position is repeated. This strategy aims to have the cops surround the player, preventing him from advancing in any direction (see Figure 18).

Figure 18: Four police cars blocking the player

The police cars keep notifying the player of sightings during the chase. If the player has not been seen for a predefined amount of time (i.e., the last player's known position was too long ago) the manager tells the police cars to patrol. If some police car reaches the last known position of the player but has no view of her, it communicates that to the manager, who also starts patrolling. Chasing will restart once a cop sees the player again.

If the patrolling time exceeds a threshold, the police cars are relocated. To this end, all the road markers not occupied by other cars and that are out of the player's sight are gathered. The last condition aims to avoid police cars suddenly appearing on the player's screen (i.e., popping). These selected markers are ordered by the distance to the next player's target. Afterward, for avoiding the nearest possible markers, we remove the first 20% of markers, at first 20% may seem bigger but considering the number of markers it is not that much, because when taking markers we consider the entire map. After removing markers, we take some amount of the markers from the beginning of the list and reshuffle them. The mentioned amount is linked to the number of police cars in the following way: we take 400% of the police cars, for example, If currently, we have 5 police cars we will take the first 20 markers and reshuffle them, and that 5 car will be taking first 5 markers after reshuffling. The reasoning behind reshuffling is that we wanted to avoid putting police cars either not too far away or not too close, so we implement the mentioned algorithm to make it randomized a little bit. Finally, police cars are relocated to these markers, making it probable that the player finds them when moving to her destination, but not all of them around that destination (which would seem unfair).

Another thing, that *PoliceManager* is responsible for is Game Difficulty. We wanted to implement some sort

of difficulties for players to keep them engaged and make the game more fun. For that, we have decided to consider increasing the number of police cars as an increasing the difficulty level. We have a base number for police cars that are set to 3, and it increases gradually depending on the score. Now let's see the relationship between the score and the Difficulty Level. Firstly, we have defined the starting score where we start to increase the difficulty that equals to 250, so after 250 we spawn one more car after we increase the threshold for increasing difficulty by the power of 2 every time it passes the previous difficulty, for example, after passing 250 score we spawn one more police car, and we set next threshold to 500, after passing 500 we spawn one more police car and set next threshold to 1000, 2000, 4000, etc. For every passing threshold we spawn one more police car, our limit, for now, is 20 more police cars. We decided to use the power of 2 because as more police cars are spawned it gets very difficult, because of the ability to block players from every direction. Finally, with that algorithm, we were able to increase the difficulty for players gradually.

8.3 Police

In order to achieve our goals regarding the police agent, we create the *Police* script, that inherits from *Civilian-Controller* (see Figure 9). In this way, when patrolling, the behaviour would be identical to that of a civilian. When chasing the player, civilian behaviors are deactivated and pursuit logic is activated. In Figure 19 the behavior tree of the police is depicted. The patrol node exactly corresponds to the civilian BT from Figure 12. As can be seen, defined nodes are in line with the functionalities considered by *PoliceManager*.

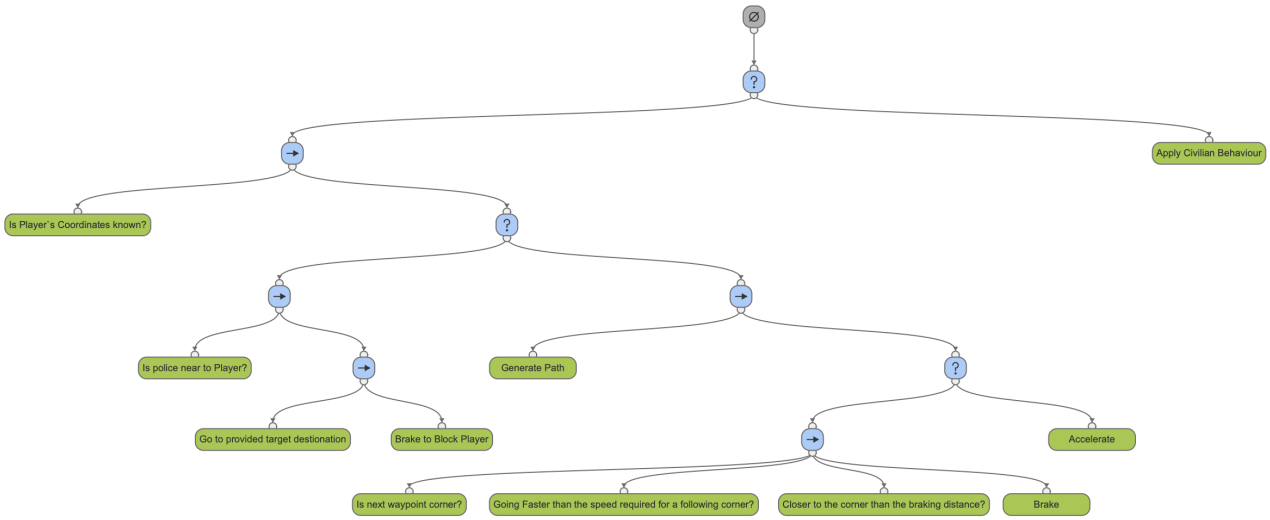


Figure 19: Police behavior tree

The main police-specific functionality is the reaching of a target position. Our pursuit logic is based on path-finding mechanisms, such as those learned in the NDVW course's lab sessions. In particular, we have used the basic navigation methods from the CarAI asset [5] as base. Nonetheless, these methods present a lot of flaws and weaknesses.

The first flaw that we found was the algorithm to track the objects, given that the implementation did not take into consideration the fact that the object that it is trying to reach may be moving. The actual algorithm was to generate the waypoints from source to target objects and follow them, but if the path to target wasn't straightforward. For example, if had to make some corner cuts, additional waypoints are created.

As we can be seen in Figure 20, for every corner two waypoints are generated, one before and another after the vertex. The actual problem is that until it passed all waypoints, Car AI do not generate new ones. This means that even if the target object moves, only when the police has reached the original target's position a new path is generated. This means that the police is always chasing the player's "shadow". Therefore, we needed to make some adjustments to dynamically track the players' movement and behave based on that. So instead of always visiting all waypoints, we decided to create a new variable "ReactionTime" that defines how often police

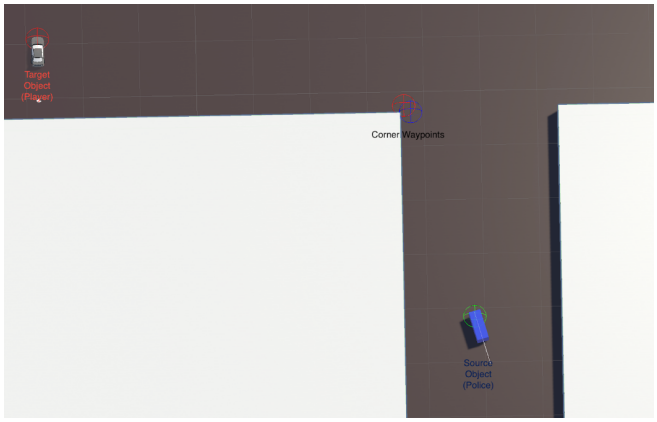


Figure 20: Waypoints Demonstration

should reevaluate and recalculate the path between the source position and the target object.

The second problem that we saw was the need for speed adjustments for the police. We humans naturally know that before we are going to make a turn at the corner, we have to slow down the car. However, that is not the case for the Car AI asset, which always accelerates ignoring the turns. We have addressed this problem in four subsequent steps:

1. Calculate the angle of the following curves: The angle of the curves is that between the car's current velocity vector and the vector connecting a pair of following (i.e., not reached yet) waypoints. We evaluate *all* the following curves rather than considering only the immediately next curve. That is, considering the vectors connecting each pair of consecutive (not reached yet) waypoints (e.g., N-1 vectors if N waypoints). This is because the path can contain one or more smooth curves (not requiring braking) closely followed by a sharp curve (requiring braking much before reaching the first smooth turn).

2. Estimate the speed required to turn correctly: This has been modeled by the function $RequiredSpeed = 1000/CurveAngle$. This function results from several empiric evaluations, searching the function type (i.e., $1/x$) and coefficient (i.e., 1000) that better turning provides. In other words, turning at the maximum speed that the curve allows without going out of the path. This is done for all the curve angles.

3. Estimate the braking distance required to reach that speed just to take the curve: We performed several experiments braking from different starting speeds to zero. We observed that the braking deceleration is mainly defined by the brake torque, only being slightly affected by the starting speed (probably due to air resistance). Subsequently, we evaluated the brake deceleration from 100 Km/h to zero using different brake torques. The resulting data points followed a logarithmic tendency fitted by the formula $BrakeDeceleration = -3.4516 * log(0.0465 * BrakeTorque)$. In combination with basic physics, this allows to estimate the brake distance for any initial and final speed. Results slightly overestimate the brake distance, but that is safer than an underestimation, so we ensure that the car always reaches the required speed for the curve. This formula allows proper braking for turns even if we change the maximum brake torque in further steps of development.

4. When the distance to a turn is close or lower than the braking distance required to reach the correct speed, the police car starts braking, aiming to reach that velocity.

The third main problem was that Car AI has not any backward algorithm, the only thing the agent knows is to accelerate forward, even if it is blocked by a wall. In order to solve that problem, we decided to make a simple algorithm that checks if the police agent's speed is close to zero for more than 0.5 seconds. In that case, except if it collides with the player, the car goes backward with full acceleration for 1 second and continues its behavior normally. The steering is inverted during this backward movement for better positioning. With what we have tested, every time that the police is stuck against a wall, it goes backward and continues the chase successfully.

When the police is at the target position, it stops, aiming to block the player's movement in that direction.

For checking the visual contact with the player, ray casting is employed. Concretely, three rays are shot from the front, center, and back of the police car to the player's car. If at least one of them reaches the car without previously colliding with a building (other police or civilian cars are ignored), it is considered that the player is in view. Then, the player's current position is communicated to the *PoliceManager*. During chasing, if the police reach the last player's known position and there is no visual contact (the three rays collide with a building), the loss of the player is communicated to the *PoliceManager*.

To provide the player with visual and audio feedback of the police chase, a siren system has been created. The siren system contains two things that are Lights and Sounds, for lights it was easy, we put 4 lights on the top of the police car, two red and two blue, and we are changing them every 0.05 seconds. For sound, we have the siren sound downloaded and attached to the police car. Unity's 3D sound functionalities are leveraged. However, there was the problem at the beginning that it sounded like one siren because all of them started playing siren together, in order to fix that we have decided to choose any random second for every police that is going to enable that sound. This decision has decreased the possibility to overlap sounds a lot. Additionally, when sirens are activated, a trigger collider around the police car is enabled. This causes the civilian cars within the trigger to start the "avoid police" behavior (see Section 7.2.4), moving to the shoulders of the street and not entering intersections. When the player is lost and police starts to patrol, sirens are deactivated. This causes the lights to turn off, the trigger to be deactivated, and the final part of the siren audio is reproduced simultaneously by all of them previously to stop the audio source. This gives a clear clue to the player that the police have lost it.

9 Conclusions and Future work

During this project, we have tested and improved our skills with the Unity game engine and the interaction of between dynamic systems. Most of the project's complication arises from the coexistence of civilians with other civilians, chasing police cars and the unpredictable player. Creating an interesting police pursuit was challenging, requiring both skillful police cars and the consideration of player's possibilities for a proper balance. The technical complication also arose from the fact that all car-related techniques had to be computationally cheap, as there are many instances working at the same time. Regarding the city generation, while we are satisfied with the results obtained with L-systems and Perlin noise, we still feel that we could improve it further by tuning the algorithms to generate more varied road/building distributions, which in turn would help the player feel more immersed in the game.

Overall, we believe that a decent level of functionality has been achieved for all facets of the game. It is not perfect, since there are too many possible problematic situations that require specific programming to avoid, but it is in fact playable and enjoyable.

In the following, our ideas for future development are described:

- The main lack of physics realism in the game is the impossibility of the cars to rotate in the X and Z axes. A possible improvement would be to enable this rotation, providing a more complex driving experience and the possibility of having non-flat streets (i.e., a sloped street). Nevertheless, this would make it possible that the cars could tip over, what would seriously increase the complexity of the project. First, some real life stability systems such as anti-roll bars would need to be implemented in *CarController*. Otherwise, any car taking a turn at a medium velocity (i.e., 30 Km/h) would tip over. Second, collisions can cause cars to tip over, and these are not avoidable since the player is free to crash. Subsequently, some logic is required to manage the tip over situations of the player, the cops and the civilians.

- For the police, it would be interesting to implement a "Road Block". That is, a group of police cars blocking a street trying to impede the player's progress. These roadblocks can be strategically placed to force the player to take alternative routes to destinations. However, care must be taken not to block the only path to the player's objective. Moreover, it would be interesting to improve the policy to determine which police go to which side of the player. In some cases, there is no police car blocking the front, so the player can easily escape. A possible improvement would be to set an assignment priority based on the side (with the front having the highest priority) and the distance from the police car to that side. It would require special attention not to constantly change the side to which a police car should go, which would make it act crazy.

- Regarding the civilians, we would like to consider more backward driving situations, given that the civilians can get stuck. Also, when the car is displaced because another has bumped into it, driving backward should be easier than making a U-turn. Finally, adding more car models would aesthetically improve the game. We tried multiple models, but we had serious problems with the collisions, and we had to remove them.
- As for the city and road generation, we would work on improving the algorithms so that more complex road systems can be generated (such as elevated highways or wide curved roads) as well as more varied structures and places to go with them (which could include buildings like hospitals, police stations, etc). We would also explore the possibility of adding locations that aren't strictly buildings (such as parks, to name an example) which would greatly improve player immersion.

A Task Distribution

- Buildings generation (Cristian)
- Roads generation (Joaquim)
- CarController (Benet)
- Player (Benet)
- Civilian (Joaquim)
- Police (Demetre & Benet)

References

[1] Justin DK Bishop, Niall PD Martin, and Adam M Boies. "Cost-effectiveness of alternative powertrains for reduced energy use and CO2 emissions in passenger vehicles". In: *Applied Energy* 124 (2014), pp. 44–61.

[2] Benet Manzanares Salor & Joaquim Marset Alsina & Demetre Dzmanashvili & Cristian Andres Camargo. *Car ChAIse Github Repository*. URL: https://github.com/BenetManzanaresSalor/NDVW_CarChAIse.

[3] Gary William Flake. *The Computational Beauty of Nature*. Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262062003.

[4] Kenney. *Modular Buildings*. Accessed: 2022-11-19. URL: https://www.kenney.nl/assets/modular-buildings.

[5] Ahmed Saed. *Car AI (Pathfinding & Vehicle Behaviour)*. Accessed: 2022-09-30. URL: https://assetstore.unity.com/packages/tools/ai/car-ai-pathfinding-vehicle-behaviour-178338.

[6] *Unity Game Engine*. URL: https://unity.com/.

[7] *Unity Navigation System*. URL: https://docs.unity3d.com/2020.1/Documentation/Manual/nav-NavigationSystem.html.

[8] *Unity Wheel Collider*. URL: https://docs.unity3d.com/2020.1/Documentation/Manual/class-WheelCollider.html.