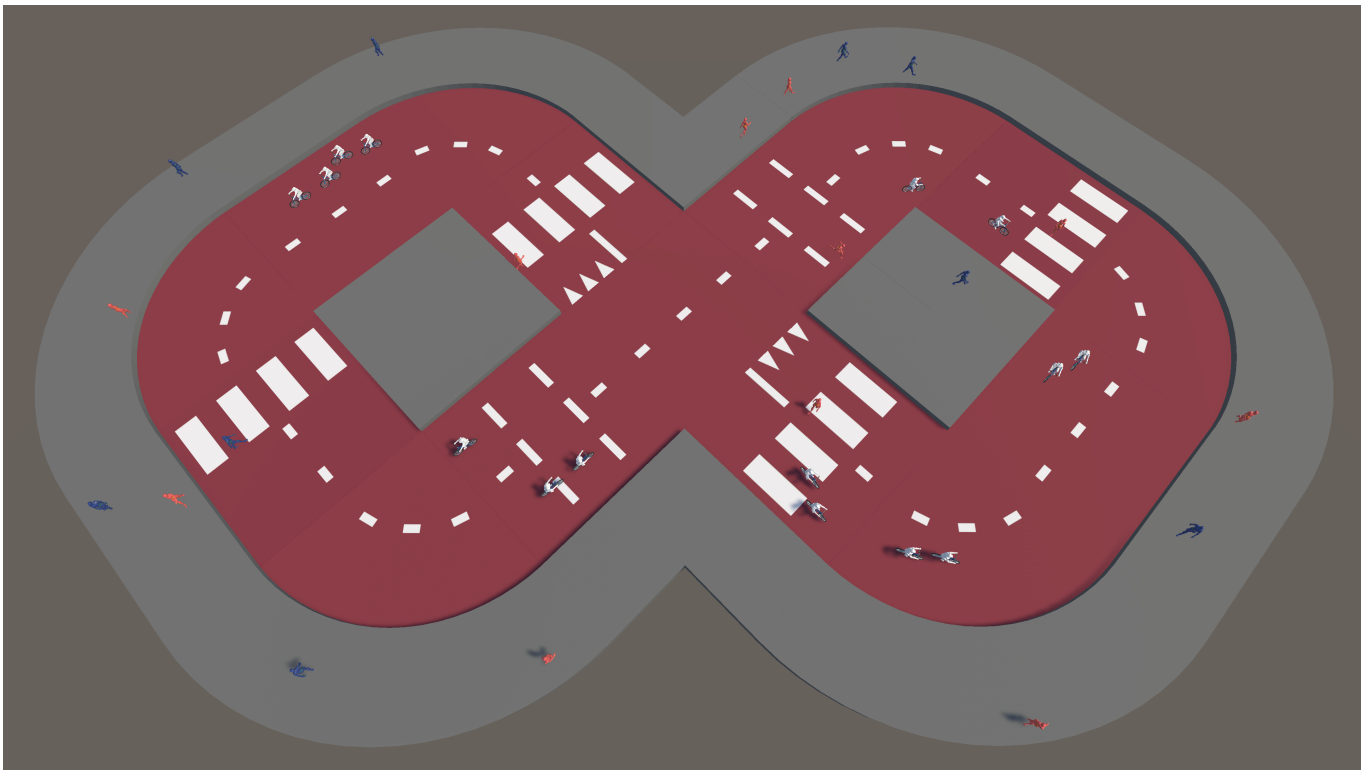
The Cyclist
Crowd Simulation: Assignment 2 Report

Group 7
Matúš Bystrický (2377632), Yang Cheng (5173884),
Alessandro Busacchi (7926340), Arie Klaver (7349033), Matthijs Berkhout (1440918)

January 31, 2025



1 Introduction

With "The Cyclist" assignment, we explore the intricate dynamics of bicycle and pedestrian interactions, focusing on developing an interaction model that prioritizes the agents' behaviors and fluid motion. With urban environments increasingly emphasizing non-motorized transport, understanding and simulating these interactions has become crucial for designing safer, more efficient public spaces.

This project aims to construct a simulation model capable of resolving three core challenges: collision avoidance between two pedestrians, collision avoidance between two bicycles, and collision avoidance between a bicycle and a pedestrian. In addition, priorities at intersections are taken into consideration as we want to simulate real-life behaviors. To achieve this, we assume indicative bicycle routes are randomized by the system in a small map, enabling a controlled environment to analyze behavior and coordination. The primary objective is to develop algorithms and strategies that ensure smooth navigation and minimize collisions.

For the simulation, Unity was selected as the game engine due to its simplicity, its flexibility, and its ability to handle complex interactions in real-time. The project leverages Unity's NavMesh capabilities for pathfinding and decision-making, creating a realistic and responsive model.

By the end of the assignment, the simulation will provide insights into how bicycles and pedestrians can coexist harmoniously, offering practical implications for traffic systems, urban planning, and game design.

2 Start of the project

At the outset of the project, we outlined a set of features to guide the development of our simulation. These features were divided into two categories: must-have features, which were essential for fulfilling the core objectives of the assignment, and optional features, which were intended to enhance the realism and depth of the simulation if time and resources permitted.

Must-have features include:

- MHF1: A 2D top-down view
- MHF2: A map with pedestrian areas and bicycle routes
- MHF3: Automatic destinations and autogenerated paths for the agents, to have an infinite simulation

Optional features include:

- OF1: A user-made infrastructure construction
- OF2: On-road obstacles for realism enhancement
- OF3: A 3D simulation
- OF4: Real-time information about agents (paths, destinations, speed)
- OF5: Shared traffic areas
- OF6: Density heat-map
- OF7: Individually deviating behaviors
- OF8: Environmental conditions
- OF9: Group dynamic

We chose to implement our simulation in a 3D environment with a top-down view (MHF1, OF3). The primary reason for this decision is that Unity's NavMesh class is not available for 2D projects. Additionally, using a 3D environment allows us to create a simulation with a greater sense of realism.

3 Technical implementation

3.1 Waypoint system

A Waypoint is used to represent a target point in the path. The parameters of a complete Waypoint are: next Waypoint, previous Waypoint, line segment length, and branch [1].

- **Next Waypoint**: When the agent approaches from the direction of the previous Waypoint, the next Waypoint becomes the target point toward which the agent can continue.
- **Previous Waypoint**: When the agent approaches from the direction of the next Waypoint, the previous Waypoint becomes the target point toward which the agent can continue.
- **Line Segment Length**: A line segment with the Waypoint as its midpoint. Its length limits the range of random destinations (the random point must lie on the line segment).
- **Branch**: When the agent reaches the Waypoint, the branch parameter serves as the basis for deciding the next target point.

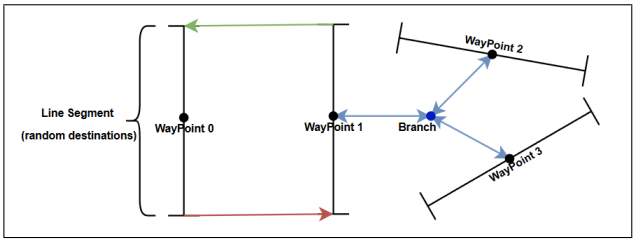


Figure 1: A schema representing the Waypoint system.

The Waypoint system used in our project divides sidewalk into several small sections. As shown in Figure 1, each section has a starting line (WayPoint0) and a destination line (WayPoint1). Pedestrians are free to select a random point on the destination line as their next target (MHF3). We can limit the degrees of freedom by limiting the length of the line. The red and green vectors show the direction, and the Waypoint can be two-way. As a node, a branch can connect multiple next Waypoints. When pedestrians are on the branch, they will randomly select a next Waypoint as their destination.

3.2 Navigation and collision avoidance

In this project, we utilized Unity's NavMesh system to simulate pathfinding and collision avoidance for both bicycle and pedestrian agents [7]. NavMesh offers a solid framework for navigating complex environments and handling dynamic agent interactions, making it well-suited for modeling realistic movement patterns in our simulation.

3.2.1 Key adjustments to NavMesh parameters

To enhance the collision avoidance behavior of agents, we made critical modifications to the default NavMesh settings:

- The *avoidancePredictionTime* parameter was increased to give agents more time to anticipate potential collisions and adjust their trajectories accordingly. This adjustment significantly improved the agents' ability to react smoothly and avoid sudden and abrupt movements.
- The radius of the cyclist and pedestrian agents was increased to better represent their spatial occupancy within the simulation. This modification allowed the agents to maintain more realistic separation distances, reducing instances of near-misses or unrealistic overlaps.

3.2.2 RVO for collision handling

NavMesh uses Reciprocal Velocity Obstacles (RVO) to manage collision avoidance between dynamic agents [5]. RVO is an advanced algorithm that allows agents to cooperatively adjust their velocities to avoid collisions without requiring explicit communication between them. By considering the velocities and positions of neighboring agents, RVO ensures smooth and coordinated movements in crowded scenarios.

In our simulation, RVO proved effective for handling interactions between bicycles, and between pedestrians.

The combination of increased avoidance prediction time and a larger agent radius further enhanced the algorithm's performance, leading to a more natural and visually realistic simulation of traffic flow and collision avoidance dynamics. Regarding the interactions between cyclists and pedestrians, however, RVO is not sufficient as a solution to handle collisions, as the dynamics are more complex due to the involvement of priorities at intersections. Section 3.6 will discuss this in more detail.

3.3 Map

As outlined in Section 2, our objective was to design a map with dedicated areas for pedestrian and bike navigation, as well as shared spaces at intersections where they can interact (MHF2, OF5). We included two types of crossings to reflect real-life scenarios commonly found in the Netherlands. The first is a regular crosswalk where pedestrians have priority. The second, called a marked crossing, gives cyclists priority, requiring pedestrians to wait for a clear path before crossing. This design choice was made to simulate diverse traffic dynamics and realistic interaction scenarios.

In the initial version of our map, showcased in Figure 2 and presented during the prototype demonstration, the map was incomplete due to challenges encountered when exporting textures from Blender to Unity. The final version of the map can be seen in Figure 3.

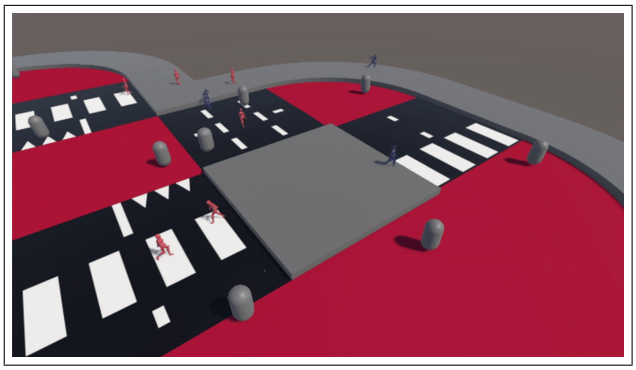


Figure 2: A view of the old version of the map.

The final version of the map is structured as follows. As mentioned in Section 1 and better explained in Section 3.2, we chose NavMesh for pathfinding. Our implementation features a dual NavMesh system with two distinct NavMeshSurfaces [3]: one for pedestrians and one for bikes. Each map section is assigned a specific layer (pedestrian, bike, or crossing) and linked to the corresponding NavMeshSurface, defining which agent type (Pedestrian or Bike) is permitted to navigate that area [6]. This straightforward and effective solution successfully enabled the navigation system.

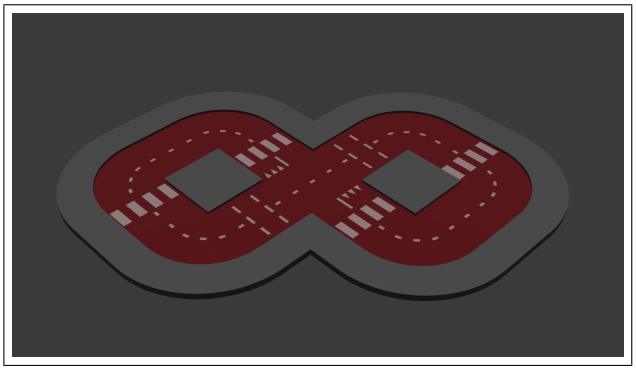


Figure 3: A view of the current version of the map.

3.4 Pedestrian Simulation

For the realistic simulation of pedestrians, our initial approach was to use Unity's NavMesh and add a NavMeshAgent component for pedestrians to achieve simple point-to-point movement [4]. However, this method presented a significant problem: the NavMeshAgent always selects the shortest path, causing characters to cross the square diagonally or navigate awkwardly around corners. This behavior does not align with the natural flow of pedestrian movement in the real world.

To address this issue and ensure that pedestrians move along sidewalks with a realistic flow, we adopted a *Waypoint + NavMeshAgent* approach. This method introduces fixed paths and adds a degree of randomness to simulate deviation from natural movement, making pedestrian navigation more authentic and consistent with real-world scenarios.

3.4.1 Implementation Process

In the Waypoint system described in 3.1, pedestrian agents follow each Waypoint sequentially. The element of randomness lies in the branching system, where multiple Waypoints can be selected as possible future destinations.

In the initial prototype, pedestrian agents lacked a proper collision avoidance mechanism. Additionally, while they navigated in two directions, they did not consistently keep to one side of the street. Implementing this behavior was essential to enhance realism and improve collision avoidance by giving pedestrians well-defined positions along the street.

To address this, we extended upon the existing Waypoint system. For each Waypoint, the destination is calculated based on the street's width and the pedestrian's direction, constraining movement to the right side. This approach enabled proper two-way navigation. Figure 4 shows an example of the simulated pedestrian behavior, where agents tend to keep to the right side of the street, as previously explained.

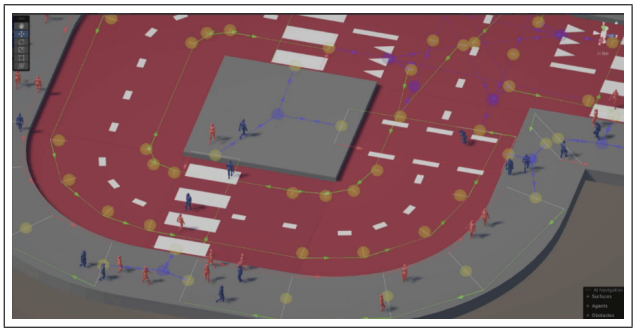


Figure 4: The pedestrian navigation system, with agents that are able to keep the right side of the street.

Furthermore, to simulate realistic behaviors, pedestrians were assigned random speeds, which were linked to corresponding animations.

3.4.2 Limitations and Further Improvements

A limitation of the current system is its inability to account for group dynamics (OF9). Incorporating this factor would significantly enhance the realism of our solution by better simulating the natural interactions between pedestrians, such as walking in groups, avoiding other groups, or dynamically changing formations. However, integrating group behavior with Unity's NavMesh poses technical challenges.

NavMesh is primarily designed for individual agent pathfinding, making it less suited for coordinated group movements. One potential approach would be to implement custom behavior scripts that manage group formations, assign Waypoints collectively, and adjust navigation paths dynamically based on group cohesion rules. Another possible solution is to designate a leader agent within the group, with the other agents following the leader while maintaining formation and avoiding obstacles. Additionally, using local avoidance techniques and boid-like algorithms could help simulate realistic group behavior while maintaining compatibility with NavMesh.

Regarding possible improvements, including individual pedestrian behavior could increase the realism and complexity of the simulation. For example, distracted walking, where pedestrians occasionally slow down, deviate from their paths, or exhibit delayed reactions due to mobile phone use, would add diversity to pedestrian interactions. This could be achieved by assigning behavior profiles to agents, where distraction levels or other characteristics are probabilistically determined.

3.5 Bicycle Simulation

The aim for collision avoidance between bikes as outlined by Section 1 is supplemented by including individual bike behaviors, allowing bike simulation in various bike ride scenarios. However, this required addressing issues regarding bike-controller dynamics, overtaking and staying on the right side. To tackle these, the *Waypoint + NavMeshAgent* approach with an RVO agent collision avoidance for the pedestrian simulation was reused to allow for the implementation of necessary functionalities.

Currently, initialized bicycle speed variations allow cycling on the right side until overtaking on the left side must be performed to maintain their speed and avoid colliding with the front cyclist. In this case, the NavMeshAgent and RVO control the speed and direction. Nevertheless, adding custom control to influence speed and velocity simultaneously is yet to be restricted and requires further research, neglecting the possibility of steering bicycles in a realistic manner based on their non-holonomic constraints.

3.5.1 Implementation Process

The bike behavior implementation included various experimental phases examining different approaches building onto the Waypoint system while attempting the potential of overwriting the steering control of the NavMeshAgent. During the implementation, a bicycle path is assumed to consist of two opposite directing lanes as seen in Figure 3.

We first tried the same Waypoint method as pedestrians. The pedestrian's *Waypoint + NavMeshAgent* approach was adapted by incorporating one-way direction Waypoint system for each bike lane as seen in Figure 5, mimicking the two-way opposite lane directions on bike roads. Of these two Waypoint systems, the Waypoint's line segment lengths approached zero, requiring the bicycle NavMeshAgents to choose target locations equal to the next Waypoint location to eliminate randomness and achieve linear movement of the bicycle.

Nevertheless, due to the lack of parameter adjustment in Navmesh Agent at that time, we were stuck with the problem that the bicycle would briefly stop and slow down at each WayPoint.

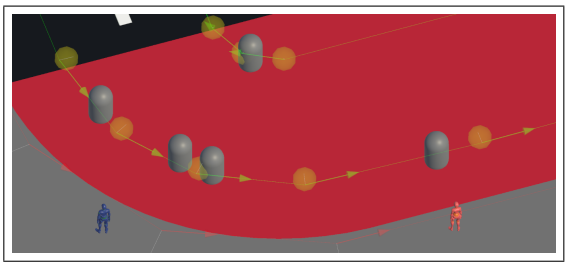


Figure 5: Two Waypoint systems for serially moving bicycle agents in opposite directions.

Meanwhile, the bike NavMeshAgents lacked a representative 3D bike model consisting of two steerable wheel colliders to allow realistic bike movement to follow Dubin's paths due to non-holonomic rotational constraints [2]. Therefore, a physics-based bicycle controller in partial agreement with the bicycle dynamics of [2] had been implemented to follow either a global route over the NavMeshsurface, or steer towards randomly sampled destinations along Waypoint line segments as seen in Figure 6. However, this required the model to contain a Rigidbody component. The Rigidbody component allows the bike to be influenced by forces such as gravity and friction while enabling acceleration and steering through scripted force applications. Nevertheless, enabling the Rigidbody introduced a critical issue as it conflicted with the NavMeshAgent functionality. When Rigidbody is applied, the bike's movement is entirely dictated by the physics engine, which disrupts the NavMesh Agent's ability to control its

position [8]. For example, the NavMeshAgent's collision detection and path recalculations do not properly interact with the physics-driven movement of the Rigidbody, leading to issues where the bike fails to correctly avoid pedestrians or obstacles. In some cases, this resulted in unexpected behaviors such as clipping through objects or an inability to properly adjust the trajectory. This fundamental conflict made it impossible to leverage both the realistic motion provided by Rigidbody and the navigation and collision avoidance capabilities of NavMesh Agent, creating a functional dilemma in our simulation.

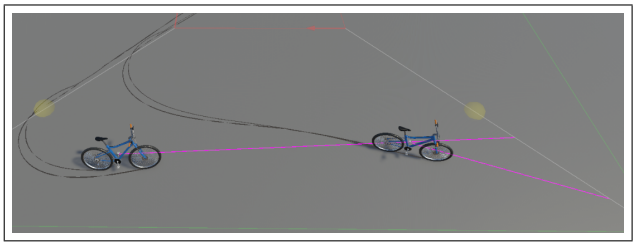


Figure 6: Physics-based bicycle steering based on non-holonomic constraints while moving towards sampled destinations on the Waypoint segments.

In addition to the Rigidbody approach, we also explored Unity Splines as a potential solution, hoping to achieve smoother motion and turns. Splines is a trajectory-based control method that enables objects to move smoothly along predefined curves, making it ideal for applications such as rail transit and predefined motion paths [9]. While Splines offers significant improvements in motion fluidity, it presents a major limitation: objects following a spline path are restricted to a fixed track. This lack of dynamic adaptability made it fundamentally incompatible with the NavMeshAgent system. Specifically, NavMeshAgent dynamically adjusts paths based on environmental changes and obstacles, whereas a Spline-based movement system follows a predetermined route without the ability to recalculate paths or respond to collisions dynamically. For instance, if a pedestrian crosses in front of the bike, the bike on a Spline path cannot make a real-time decision to overtake or re-route—it remains constrained to its predefined trajectory. This rigid movement system proved unsuitable for our simulation, which required a more flexible and interactive approach to navigation.

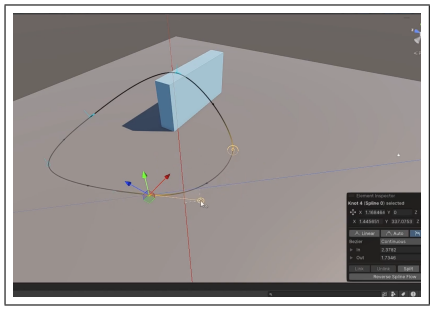


Figure 7: Use spline to draw a fixed track for the bicycle agent. The bicycle cannot deviate from the track to avoid collision

3.5.2 Right-Side

Given these challenges, the initial bicycle Waypoint system was re-implemented with 3D bike models as NavMeshAgents with adjusted parameters, causing realistic-looking steering and slowdown problem-solving. However, the consideration to utilize one Waypoint (spanned over the bike path) or two one-way Waypoint systems (one for each lane) has been experimented with. The former option aligned with the pedestrian Waypoint implementation, where incoming pedestrians randomly sample directions on the right side of the next Waypoint segment to maintain a realistic two-directional flow. This approach for the bicycles yielded too large of a sampling space for bicycles to unnaturally choose target directions with a large variance at each right-side Waypoint segment without considering the moral bicycle rule to stay on the right side. This wide sampling range problem can also be seen in Figure 6.

Therefore, with two opposite one-way Waypoint systems, one for each lane and spanning over the lane widths, agents can sample directions on the right side Waypoint segment length that are already located on the one-directional lane. However, due to the random sampling of locations in the way-points right segment, bicycles would still not align with the right side, allowing overtaking from the left.

Subsequently, the Waypoint sampling on the right side was restricted to sampling only the uttermost right side location of the Waypoint right line segment to always choose the next destination on the extreme right side of the lane. In this setting, the Waypoint segment spanned the one-way lane until the boundary of the NavMeshSurface up to the sidewalk. Nevertheless, against our hypothesis, the RVO of the bike NavMeshAgents did not allow agents to move away with a repulsive force from the NavMeshSurface "wall". Therefore, all Waypoint segments had been scaled down and moved away from the NavMeshSurface border, which mainly resolved to stay on the right. In Figure 8 can be seen how at the top, bicycles can sample target directions on the Waypoint segment right side.

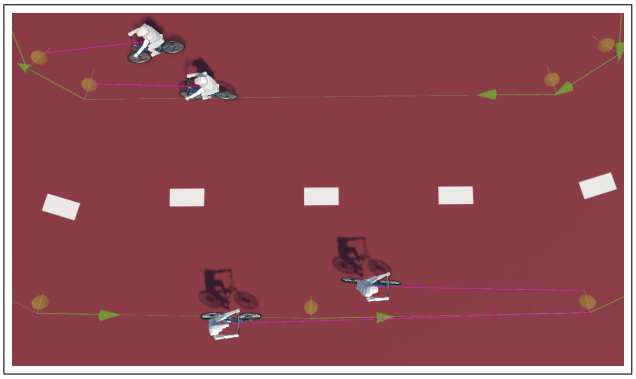


Figure 8: Top: shows random target location sampling within the Waypoint right segment length. Bottom: shows how bike speed dependencies choose on the uttermost left (faster) or right (slower) side of the Waypoint segment.

3.5.3 Overtaking

As the bicycles sampled their directions to the uttermost right side of the approaching Waypoint segment, bicycles with higher speeds could still overtake bicycles from the left and right sides. This likely happens as the Waypoint system span does not have boundaries compared to a NavMeshSurface, allowing agents to move out of the Waypoint areas.

The choice of overtaking left or right is not transparent, mainly by the RVO system. Therefore, it had been tried to implement a vision-based steering method to locate the front bicycle with the neighbouring bicycles query of each NavMeshAgent to be able to steer their direction towards the left if, within a certain frontal distance, a slower bicycle is approaching. However, similarly to challenges implementing the NavMeshAgent with a rigid body, the vision-based mechanisms would overwrite the steering determined by the RVO and NavMeshAgent, which was not yet made possible.

To maintain the steering of the NavMeshAgent mechanism, it was considered to sample on the uttermost left or right side of the Waypoint segment length, dependent on the incoming speed of the agent. Hereby, a trade-off can be made between finding the optimal Waypoint segment scaling threshold for overtaking only from the left and the unrealism of higher-speed bicycles maintaining on the left side of the lane due to choosing the left uttermost Waypoint segment point. For instance, at the bottom of Figure 8, it can be seen that an optimal Waypoint segment scaling is found such that higher speed bikes sample on the uttermost left while returning mostly towards the right-middle side of the lane after overtaking.

3.5.4 Limitations and Further Improvements

The bicycle simulation behavior can be realistically improved based on focusing on the limitations regarding individual steering behaviors, collision anticipation and coherence with other bicycles.

The simulation lacks model assets and steering animations for the bicycle. Currently, we use static models we made ourselves, which results in some of the details of the bicycle's movement in the scene being unrealistic, such as the bicycle's steering, which is more like a direct rotation angle than an arc-like movement in the real world. This is mainly due to the constraints of not being able to implement the non-holonomic rotational constraints of the bicycle.

Furthermore, the social interaction realism between bicycles during cycling and mainly while waiting before a marked crosswalk can be improved. For instance, the lateral distance between NavMeshAgents could be decreased such that bicycles can overtake each other with a smaller separating distance. This can also be applied during group formation while waiting before a crosswalk to align with straight-rotated wheels as a tight group. Particularly, during waiting before an intersection, incoming bicycles tend to push idle bicycles in the front, sometimes causing the group to move accumulatively partly on the sidewalks. An option could be to allow a dynamic stopping distance for the incoming bike related to its current speed or increase the stationary force of the waiting bicycle.

3.6 Priority Control

With the individual bicycle and pedestrian behavior implemented successfully, there was one element of the simulation that was yet to be addressed. This element was the handling of priorities among the cyclists and pedestrians. Our map features three kinds of crossings where such priorities come into play. In the center there is the large bicycle intersection, where cyclists crossing on one lane have priority over the other. Additionally, there are several points where the cyclists and pedestrians will interact with each other. These can be divided into two kinds; regular crosswalks, where pedestrians have right of way, and marked crossings, where cyclists get to cross first. A similar method was used to implement all of these crossings, making use of trigger-zones to detect the number of agents currently occupying the relevant crossing section, as well as which agents have entered a yielding area.

3.6.1 Cyclist Priority Intersection

The first crossing that was implemented was the priority intersection for cyclists, located at the center of our map. This is the only crossing where pedestrians are not involved directly. In order to detect which agents had to wait and which did not, we considered several approaches.

Our initial idea was to make use of the Waypoint system that was already in place. If we could detect the number of agents currently occupying the intersection by tracking those within a specific interval of Waypoints, we could easily determine whether cyclists on the non-priority lane had to wait or not.

However, keeping track of the last Waypoint passed by an agent proved challenging, so we opted for the use of trigger-zones instead. Initially, we made the entire middle section of the intersection one continuous trigger-zone. When an agent entered this zone, the number of active agents on the intersection increased. If an agent entered a separate trigger-zone placed in front of the yield markings while the number of agents currently occupying the intersection was not zero, the agent would have to wait. Once the agent currently crossing left the intersection trigger-zone, the number of crossing agents is decreased, allowing the waiting agent to continue.

This approach worked as intended and seemed promising, but it had limitations. First, using one large trigger-zone for the entire intersection was impractical, as cyclists approaching the intersection on the non-priority lane often waited unnecessarily. Realistically, they only needed to wait for cyclists crossing directly from their left if they were really close, or from the right if they were a further away, as these are the only ones that would actually interfere with their route. Cyclists that are in front of them but coming from the right would likely already have passed by the time the approaching cyclist started crossing. Additionally, cyclists on the non-priority lanes from both directions shared the same intersection trigger-zone, even though the aforementioned dynamics for each direction were precisely the opposite.

In order to solve this, we have essentially split the intersection in two, where each yielding area has its own associated intersection trigger-zones to detect the current occupancy level of the intersection. This way the cyclists

are not waiting for cyclists that are already irrelevant to them. Furthermore, each yielding area now has two rectangular trigger-zones instead of just one, allowing us to more precisely define the relevant region (see figure 9).

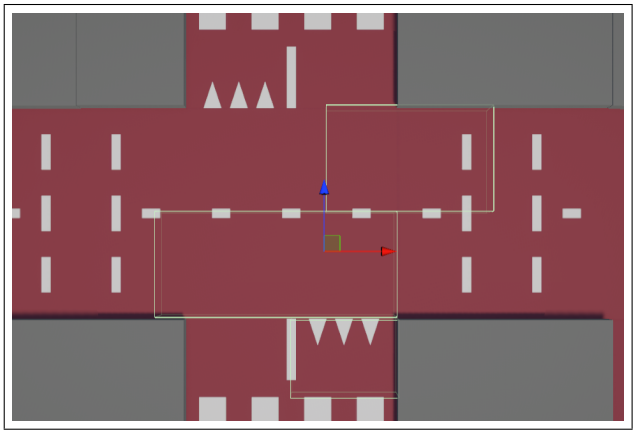


Figure 9: Top-down view of the cyclist priority intersection. Yellow wireframe at the bottom is the yielding area for the bicycles coming from the bottom (waiting area). Two yellow wireframes in the centre of the intersection are the trigger-zones for these bicycles.

With these improvements, the intersection flowed more smoothly, but there was still one minor issue that impacted scalability regarding the number of agents. Let us imagine a scenario where two agents are approaching the intersection from the non-priority lane, and no agents are currently occupying the intersection. One agent is cycling slightly behind the other. Once the front cyclist enters the intersection, the number of agents occupying the intersection increases, causing the cyclist behind them to wait, even though their paths would not interfere. To solve this, we apply a cooldown timer to any agents entering the yielding area. While this cooldown is active, they will not be detected in the trigger-zones, effectively solving the issue at hand. The cooldown is set to a value such that the cyclist is unable to reach that same intersection again before the cooldown has ended, ensuring they are detected appropriately when they should be.

3.6.2 Crosswalks

The crosswalks are the crossings where pedestrians always have right of way, meaning cyclists must yield. The implementation of these crossings is rather similar to that of the priority intersection, with some slight alterations.

The intersection trigger-zone is now placed on the markings of the crosswalk and is configured to detect only pedestrians. To achieve this, we have tagged each agent as either a pedestrian or cyclist.

The yielding zones are placed right in front of the crosswalks on both cycling lanes. Once a cyclist enters this zone, the number of pedestrians currently crossing is detected and, based on this, the cyclists either yields or crosses.

Initially, also here we had just one trigger-zone on the crosswalk, that was used by cyclists coming from both directions to detect the number of pedestrians occupying the crosswalk. However, since cyclists do not have to wait for pedestrians who have already walked past them or are still relatively far away, we split the zone into two. Cyclists now wait only for pedestrians that are walking on the section of the crosswalk that corresponds with their lane, with a little bit of leeway on both sides to account for pedestrians that are about to cross.

3.6.3 Marked Pedestrian Crossings

The final type of crossing in our map is the marked pedestrian crossing. Here, unlike the regular crosswalks, cyclists have the right of way and pedestrians must wait until the crossing is unoccupied before crossing. To implement this behavior, modifications to the regular crosswalk were required, as simply reversing the priorities was insufficient.

The main difference with this crossing compared to the regular crosswalks is that the yielding agents are no longer walking in lanes, as was the case for the cyclists. Pedestrians yielding for crossing cyclists do not necessarily stay on one side of the walkway. This complicates things, as the pedestrians now also may enter the yielding area on the opposite side of the crossing. In order to prevent the pedestrians who have already crossed from yielding for cyclists behind them, the crossing section requires some overhead data structure. When a pedestrian enters the yielding area on one side, they are added to a list of pedestrians that are currently crossing. If this agent then enters the yielding area on the other side of the road, they are not forced to wait a second time and are removed from the list of crossing pedestrians.

As with the other intersections, each of the two yielding areas has its own trigger-zone to detect the occupancy level of the intersection.

Although this version of the implementation was functioning as intended, there was still an issue: determining a suitable intersection trigger zone for cyclists was challenging due to their higher speed compared to pedestrians. If a pedestrian was crossing, a cyclist that was previously around the corner could suddenly be right in front of the pedestrian.

In order to solve this, we considered three possible approaches.

One option was to simply increase the size of the trigger-zones. Although this would solve the issue of pedestrians crossing when they are not supposed to, it would also have several drawbacks. Firstly, these crossings are located right next to the priority intersection of the cyclists. If we extend the trigger-zone to cover this area, the pedestrians are likely to be waiting for cyclists that are not actually heading in their direction. Additionally, the trigger-zone being this large is not particularly realistic, as pedestrians can only see so far and are unlikely to be checking around the corner. Lastly, increasing the size of trigger-zones this drastically would make it harder for pedestrians to find a suitable time to cross, especially with more cyclists. As a result, pedestrians will have to wait a lot longer before being able to cross, possibly forming large queues in front of the crossing. Hence, this approach did not seem suitable.

The second approach was to add an additional yielding area in the middle of the crossing. Pedestrians would then have an additional point where they can assess whether they can cross safely and wait if necessary. We would not

add an entire refuge island in the middle of the cycling lane, but instead have the pedestrians wait right in between the two lanes. This behavior can also be observed in the real-world. The problem with this approach is that when the number of pedestrians increases, this middle yielding area might get a little too crowded, blocking the way for incoming cyclists. This could be solved by having the pedestrians also keep track of the amount of agents waiting in this middle yielding area, but this seems like a rather complex solution to such a simple problem.

Instead, we opted for a more straightforward, yet still pretty realistic solution. The main issue with the simple implementation we had in place was that the pedestrians were crossing very slowly. To address this, we made the pedestrians run across the crossing instead. This reduced the region that needed to be free of cyclists for pedestrians to cross. Furthermore, although this approach may sound slightly odd at first, it actually leads to pretty realistic behavior. In real life, pedestrians are often in a hurry and want to cross the street as soon as they see the possibility to do so. In such cases, especially at crowded crossings, they will often run across the street to close the gap faster.

3.6.4 Limitations and Further Improvements

The current implementation of the different crossings and the priority control has resulted in a realistic simulation. Agents move smoothly across the map and adhere to the priority rules applied in real-world traffic. However, there are some limitations of our current approach.

First, cyclists have the tendency to bump into each other. The yielding zones are not that large, and when several cyclists are already waiting in the area, an approaching cyclist interacts with the waiting cyclists before reaching the yielding zone. Due to the higher speed of the approaching cyclist, it may bump into the waiting cyclists and move them slightly. This phenomenon becomes apparent especially when the number of agents increases. We attempted to resolve this by applying additional mass to the yielding agents, preventing them from being pushed around, but were unable to find a way to do so within our limited time frame. Another possible solution would be to implement a multilayered yielding zone setup, where several yielding zones are placed in a sequence. Once the front most yielding area has, for example, three agents in it, the yielding zone behind becomes active. This approach would not only prevent cyclists from bumping into each other, but it would also prevent them from halting right on top of the crosswalk, blocking the way of pedestrians.

This multilayered setup would also address another issue. In the current system, the yielding area is the same size for all cyclists, regardless of their speeds. This is not particularly realistic, as the breaking distance increases with higher speed. Besides just activating the additional layers of yielding areas when the front ones are occupied already, it could also be used to have cyclists that approach with higher speeds break earlier.

Another drawback of our current implementation is that cyclists do not consider the waling direction of the pedestrians. We simply check whether the relevant section of the crosswalk is occupied. As a result, cyclists sometimes wait unnecessarily when they could already be crossing. By tracking the pedestrians' walking direction and factoring it into the decision-making process, cyclists could more accurately determine when it is safe to cross.

In some cases, especially when the number of cyclists increases significantly, we noticed that cyclists would accidentally enter the trigger zones of the opposite lane. This occurs when they are pushed by other nearby cyclists. This could possibly be solved by tracking the direction of cyclists entering the yielding zones, or by making sure each cyclists that yields has passed the yielding zones in a specific order.

Finally, the regular marked pedestrian crossing could also benefit from significant improvements.

First, the animation switch between walking and running, as well as their change of pace, is rather abrupt, leading to somewhat unrealistic looking behavior.

Additionally, the pedestrians sometimes miss the yielding zone, especially in a crowded setting. As a result, they occasionally do not start running or keep on running after they have crossed. Increasing the size of the yielding zones could solve this issue but would also increase the likelihood of pedestrians accidentally entering the zone when merely passing by.

Furthermore, despite having the pedestrians run across the crossing, there are still instances where a cyclists interferes with their course. In these cases, in our current system, the pedestrians ignore the cyclists and keep on running, resulting in a collision between the two. In the real world, we would expect the pedestrians to wait for the cyclist to cross, as they are the ones with right of way in this case. This could be solved by adjusting the collision avoidance priorities among the agents.

One final improvement to the crossings could be to adapt the running speed of crossing pedestrians based on the size of the gap. If there is no cyclist in sight, pedestrians would have no reason to run, but if there is a cyclist approaching, it would make more sense for them to speed up accordingly. A more dynamic approach might thus lead to more realistic behavior.

4 Conclusion

In this project, we developed a cyclist and pedestrian interaction simulation using Unity, focusing on realistic movement patterns and collision avoidance. Our approach combined NavMesh-based navigation with a Waypoint system to ensure smooth and realistic agent behaviors. Through iterative improvements, we addressed key challenges such as pathfinding, right-side adherence, overtaking and priority control at intersections.
In the future, this simulation can be expanded to enhance realism and adaptability. For instance, dynamic and static obstacles (such as roadblocks, construction zones, or moving pedestrian groups) could be introduced to create a more complex urban environment. Additionally, time and weather variations could be simulated, such as changing lighting conditions between morning and night or the impact of rainy weather on bicycle braking distances. Another important enhancement would be more diverse agent behaviors, where individuals exhibit different movement patterns and social interactions, such as

walking in groups or stopping for conversations. From the overall perspective of the project, the number of pedestrians and bicycles generated in the project is fixed before running. The project lacks some interactive UI components that allow users to dynamically control the number of agents, which can also be expanded in the future.

5 GitHub Repository

The code is available at the following link: `https://github.com/matus01/The_Cyclist`

References

[1] Game Dev Guide. *Building a Traffic System in Unity*. `https://www.youtube.com/watch?v=MXCZ-n5VyJc`. YouTube video. Aug 5, 2019.

[2] S. Rosman. "Path Planning for Cyclists: Simulating Bicycles in Urban Environments". Master's thesis. Utrecht, Netherlands: Utrecht University, Aug. 2015.

[3] Unity Technologies. *Building a NavMesh*. `https://docs.unity3d.com/2020.1/Documentation/Manual/nav-BuildingNavMesh.html`. Accessed: 2025-01-31. 2021.

[4] Unity Technologies. *Creating a NavMesh Agent*. `https://docs.unity3d.com/2020.1/Documentation/Manual/nav-CreateNavMeshAgent.html`. Accessed: 2025-01-31. 2021.

[5] Unity Technologies. *Inner Workings of the Navigation System*. `https://docs.unity3d.com/2020.1/Documentation/Manual/nav-InnerWorkings.html`. Accessed: 2025-01-31. 2021.

[6] Unity Technologies. *Navigation Areas and Costs*. `https://docs.unity3d.com/2020.1/Documentation/Manual/nav-AreasAndCosts.html`. Accessed: 2025-01-31. 2021.

[7] Unity Technologies. *Navigation System in Unity*. `https://docs.unity3d.com/2020.1/Documentation/Manual/nav-NavigationSystem.html`. Accessed: 2025-01-31. 2021.

[8] Unity Technologies. *Unity Rigidbody Official Documentation*. `https://docs.unity3d.com/ScriptReference/Rigidbody.html`. Accessed: 2025-01-31. 2021.

[9] Unity Technologies. *Unity Splines Official Documentation*. `https://docs.unity3d.com/Packages/com.unity.splines@2.7/manual/index.html`. Accessed: 2025-01-31. 2021.