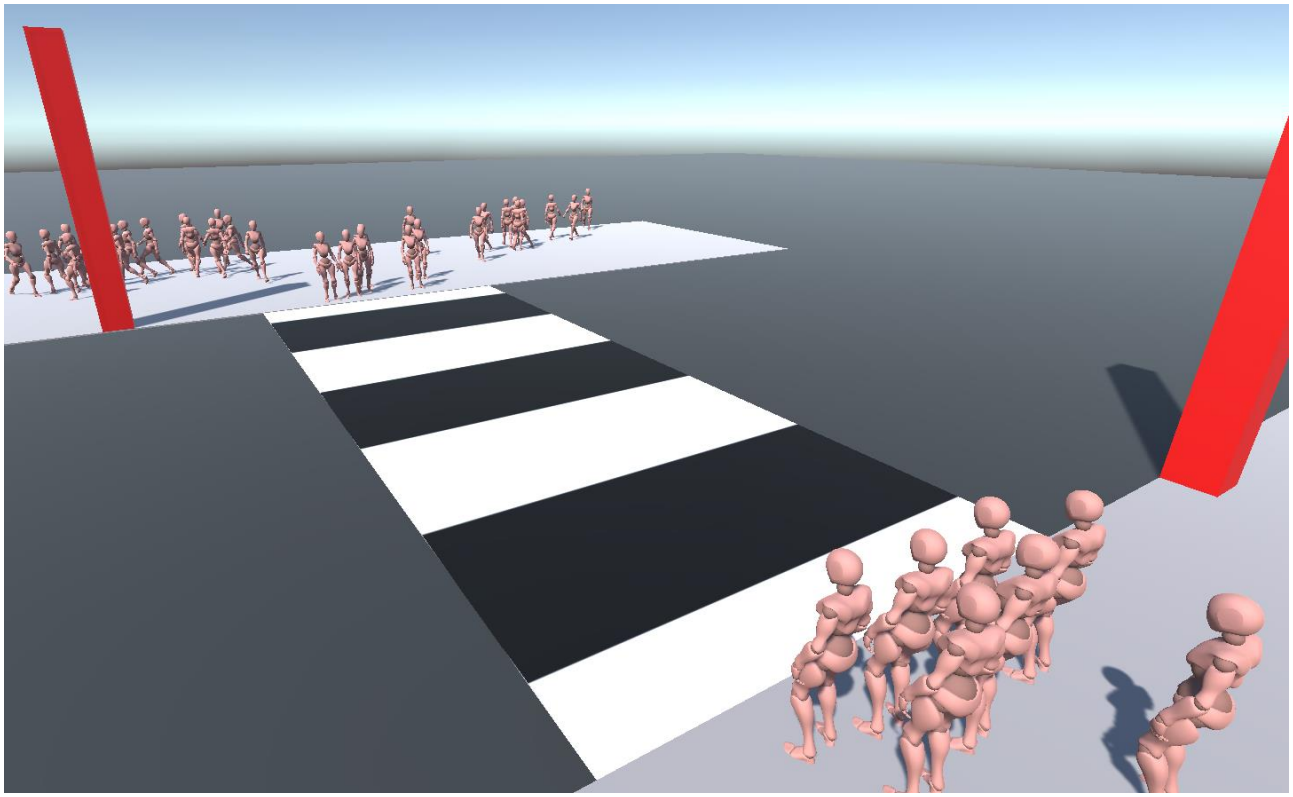
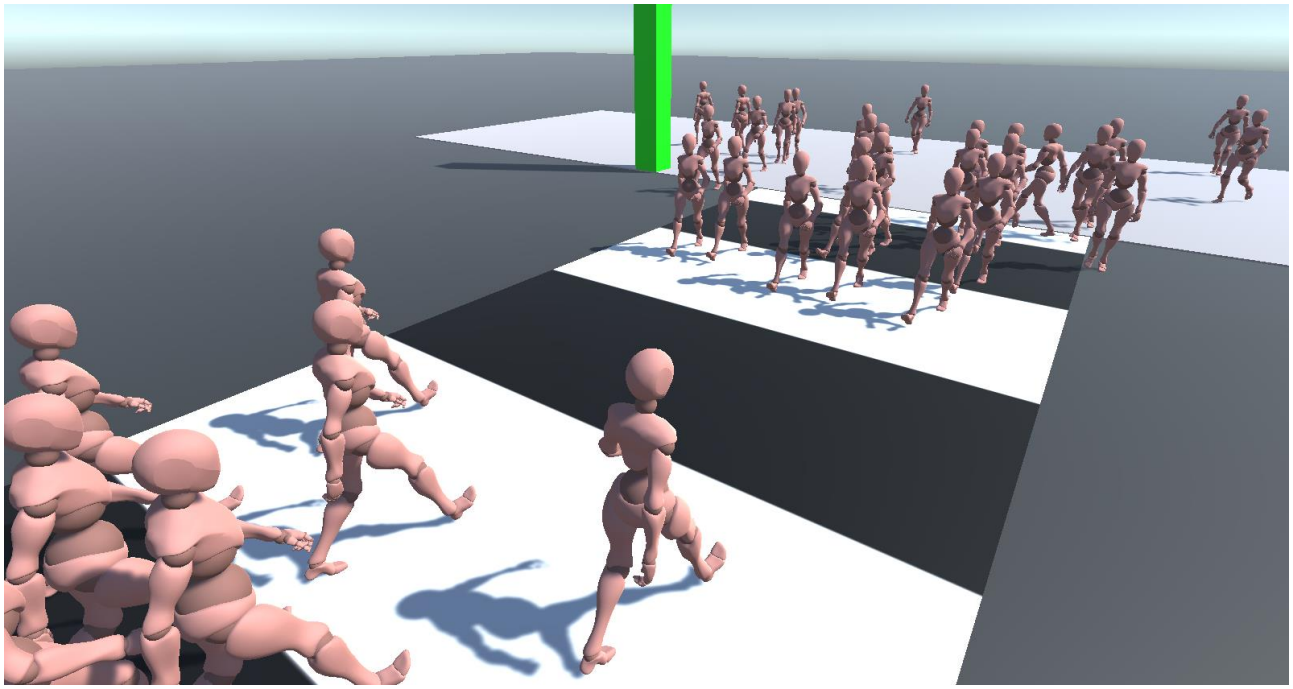
Crossing citizens

A simple simulation of pedestrian crossing





Luca Di Gennaro - 961208

Table of Contents

Project goal and overview

The main goal of this project is simulating a believable behaviour for two opposing flocks of pedestrians which are trying to cross a street.

The crosswalk is marked by white stripes and is regulated by a traffic light on each side, which, as in the real world, allows or forbids the citizens to cross the street.

Each agent will behave independently: he will try to approach the crosswalk, move along it, and reach his personal destination on the other side of the street; however, he will need to pay attention to

- other agents, as to avoid collisions as much as possible and cross smoothly
- the environment, such as looking for eventual obstacles, observing the traffic light and so on

Inspiration

The main inspiration that led me to work on this kind of project is the crowd simulation present in the Yakuza series: I was extremely fascinated by how much lively and real the city environment felt with dozens of citizens strolling around, buying their food and, essentially, “doing their stuff”; with relatively simple behaviours, the world felt alive and immersive.

For this reason, I decided to dig up more on the matter and try to model my own self-contained scenario to understand how difficult it would be to accomplish such a task.

Technical overview

The main elements of this project we are going to discuss are:

- The environment
- The citizen spawning system
- The citizen agent (which includes decision making process and steering behaviour)

For each of these, we will present relevant snippets of code and diagrams to better illustrate the underlying systems.

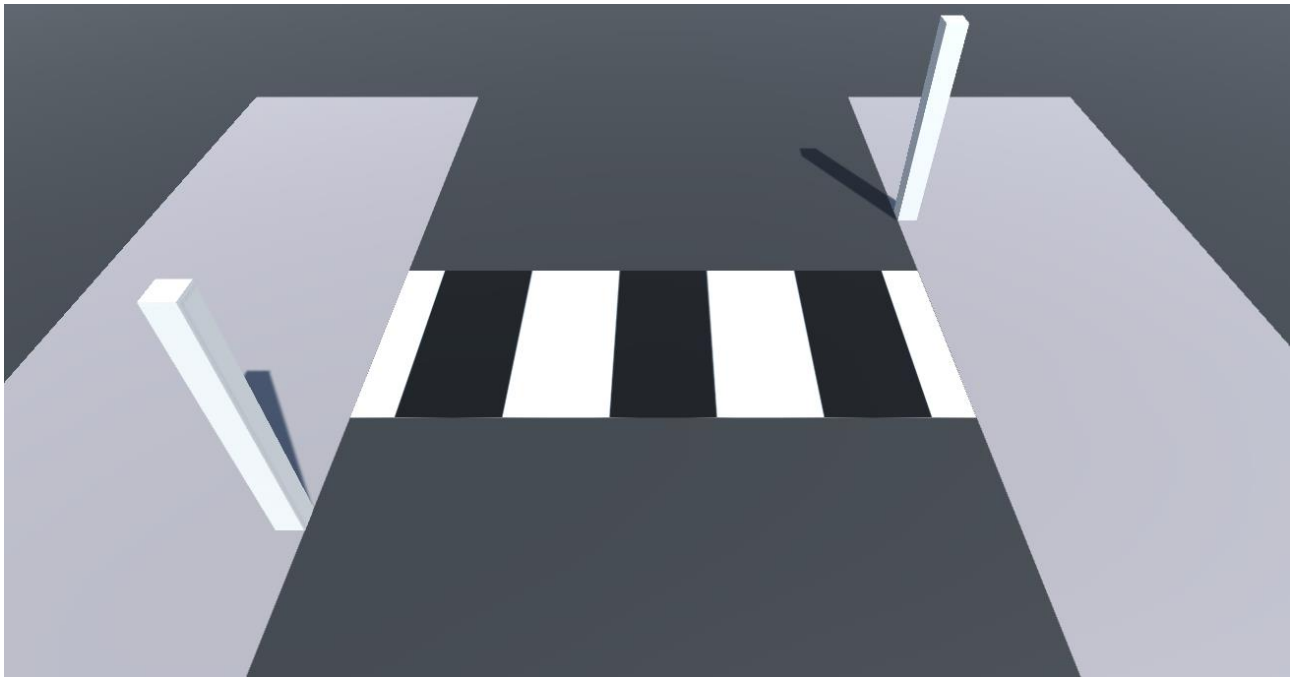
The components offer many parameters to edit the behaviour of the citizens (avoidance, seeking…), the spawning positions, and other variables that may help with customizing the crossing simulation.

For example, some parameter combinations may suit more crowded flocks to reach their goal, while some other combinations may ensure better avoidance quality given a limited number of citizens active at once.

Environment

The empty crossing scene is composed by

- **two sidewalks**, from which citizens are spawned in bulk via spawners (more on that later)
- **two traffic lights**, one for each sidewalk, which follow the usual green-yellow-red cycle and are synchronized
- **a crosswalk**, which defines a bounded area the citizens will walk onto while crossing



Traffic light

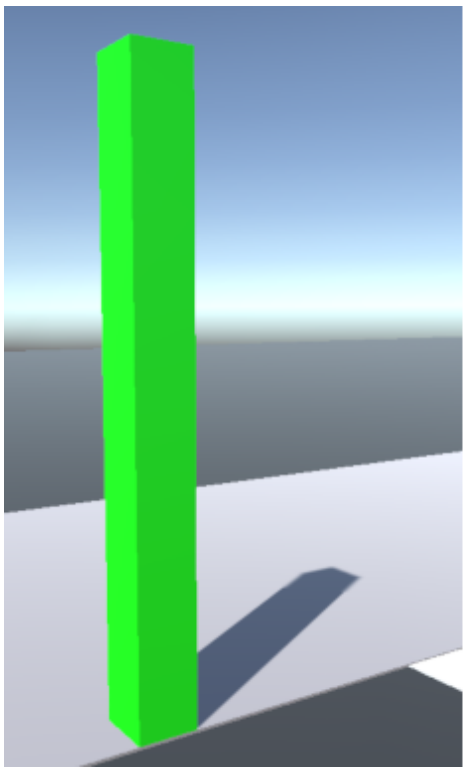
Assets/Scripts/SimpleTrafficLight.cs

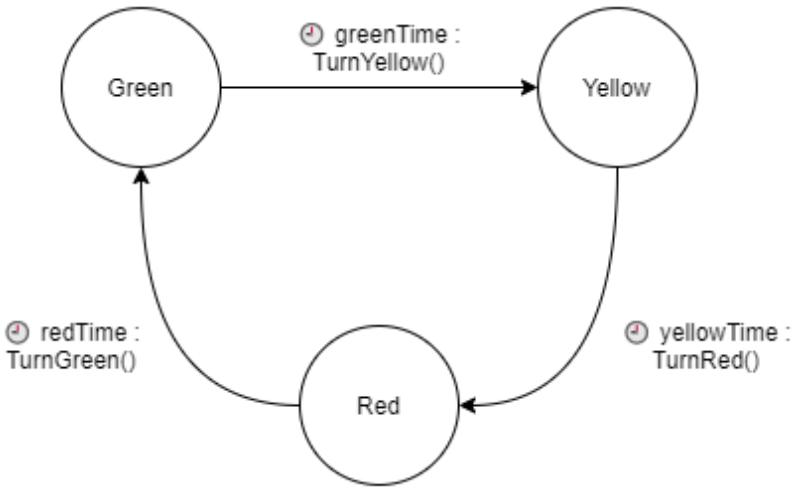
The traffic light is a simple pole which changes colour based on its internal status.

The duration for each kind of light (green, yellow, red) and with which colour to start from are parameters which can be easily specified and adjusted.

```
public float greenTime;
public float yellowTime;
public float redTime;
public ColorState startWith;
```

The internal colour state is represented with a simple enum (*ColorState*)





In game, the pole will be handled by a perpetual **coroutine**: after the initial colour is assigned to the pole (through its material) the coroutine will be put on wait for the specified amount of time for that colour; after that, depending on the current colour, the coroutine will swap to the next colour and wait again.

```
switch(_currentState)
{
   case ColorState.green:
      TurnYellow();
         yield return new WaitForSeconds(yellowTime);
         break;
   ...
```

For example, suppose we're currently in the green state: if the coroutine is woken up it means it's time to change the pole to the yellow state and sleep again for the amount of time specified for yellow.

Citizen spawning system

Assets/Scripts/CrossingCitizenSpawner.cs

The citizen spawner's job is to make sure a certain amount of citizen agents are spawned in a specified area (which in our case coincides with the sidewalk) and assign them a common "base destination" (which in our case coincides with the opposing sidewalk).

Let's see what parameters it has to offer:

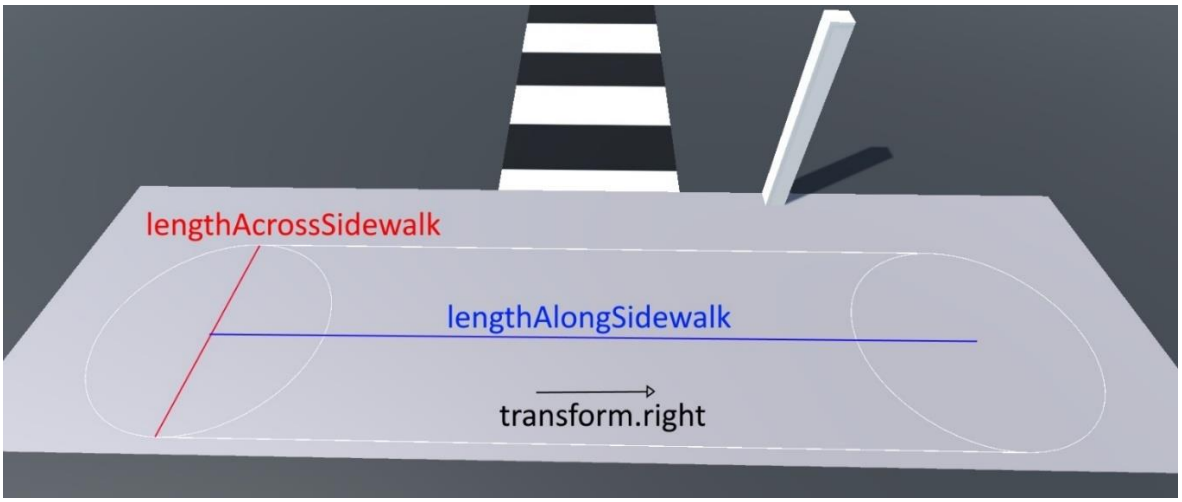
```
public GameObject citizenTemplate;
public int amount;
public float lengthAlongSidewalk;
public float lengthAcrossSidewalk;
public Transform baseDestination;
```

We can see that we can define our own crossing citizen template: the only requirement for the template to be instantiated is to possess the Crosser component (see Citizen agent section) which defines the citizen behaviour. It is required since the spawner must set the base destination for all the instances generated.

We can then choose the spawn amount as well as the spawn area, defined as capsule-like zone, into which spawn our agents with random locations

```
Vector3 spawnPosition =
Utilities.GenerateValidPositionCapsule(transform.position,
                      citizenTemplate.transform.localScale.y,
                      citizenTemplate.transform.localScale,
                      lengthAcrossSidewalk * 0.5f,
                      lengthAlongSidewalk,
                      transform.right);
```

The utility function used is better explained in the Utilities section, but intuitively will lead to a spawn area like the following:

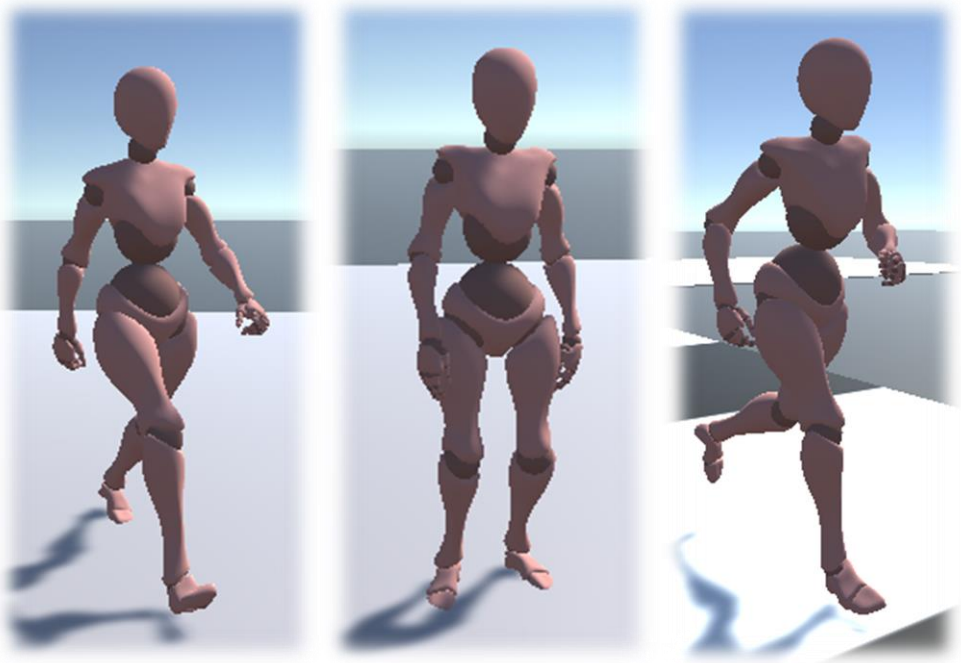


Finally, we must define a transform as base destination: in the scene we used two empty GameObjects as targets, each assigned to one spawner and placed on the sidewalk opposite to it. Essentially, we will see that this target will serve as a base for generating random destinations for citizens, so that the simulation feels more natural.

Citizen agent

Assets/Scripts/Agents/Crosser.cs

The citizen agent is represented in game via a humanoid pawn (obtained from Mixamo), animated accordingly to the current situation.



Walking, idling and jogging animations

Before descending into more detail, let's start by describing briefly how the citizen behaves at a higher level after being spawned:

1. The citizen agent will randomly select a new destination based on the base destination received
2. It will try to approach the zebra crossing in order to reach his destination on the opposite sidewalk
3. It will start crossing by paying attention to the traffic light and neighbours
4. Once crossed, the agent is free to reach his destination

When the destination is reached, the agent will be "despawned", which means it's going to be teleported in a random location near its spawn point and the aforementioned steps will be executed again: this is done to emulate a continuous flow of citizen approaching the crosswalk coming from and going towards a variety of locations.

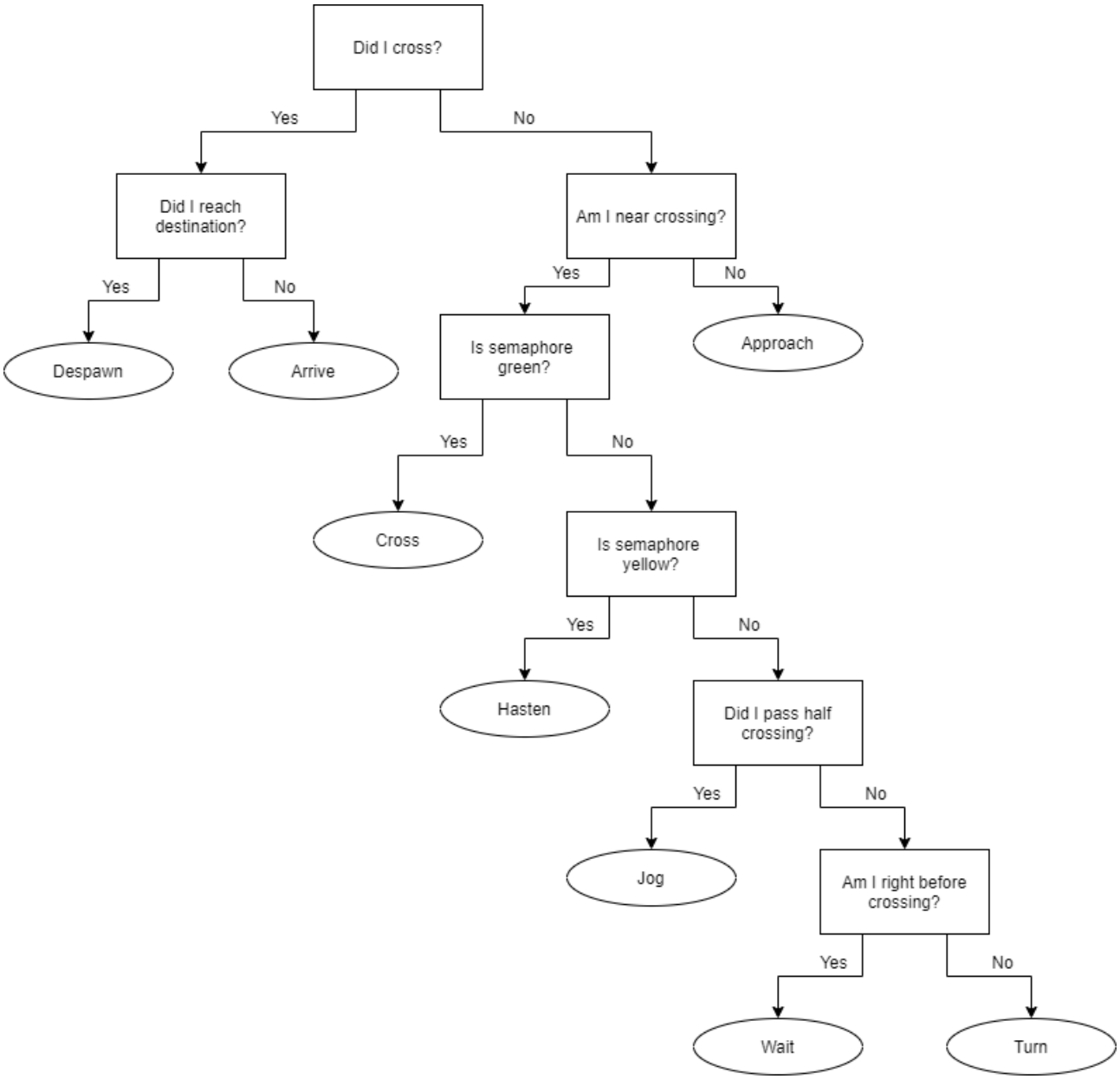
Furthermore, to make the simulation a bit more organic, the citizens

- are allowed to stray a bit from the zebra crossing
- will turn back or run forward if the red light turns on while crossing depending on their position

Decision tree

Given that we need to simulate a multitude of agents with a relatively not complex behaviour, the usage of decision trees for the decision-making process was preferred thanks to their qualities of being simple and fast to traverse.

The implementation of the following decision tree is based on the framework seen at lesson (*Assets/Scripts/DecisionTree.cs*)



Most of the decisions we'll see are based on an internal subdivision of the crossing to let the agent be able to figure out where he is:

```
public Transform baseDestination;

private Vector3 startPosition;
private Vector3 firstCrossingCheckpoint;
private Vector3 secondCrossingCheckpoint
private Vector3 destination;

public float destinationOffset;

private Vector3 halfCrossingTowards;
private Vector3 secondCheckpointTowards;
```

- `baseDestination`, assigned by the spawner
- `startPosition`, position of the agent right after being spawned
- `firstCrossingCheckpoint`, the closest point from `startPosition` to the crossing bounds, skewed to be closer to the centre of said bounds
- `secondCrossingCheckpoint`, the closest point from `destination` to the crossing bounds, offset to be a bit more internal to said bounds
- `destination`, a random destination chosen by starting from the base destination and moving away from it by `destinationOffset` along its right vector
- `halfCrossingTowards`, the forward vector coming from the center of the crossing towards `firstCrossingCheckpoint`
- `secondCheckpointTowards`, the forward vector coming from `secondCrossingCheckpoint` towards the center of the crossing

Most of the heavy operations (e.g. getting objects and components references in the scene) are done only in the beginning (in the *Start()* function), caching the results in private variables to make the tree walking process lighter on the CPU.

Without further ado, let's see the decisions and actions in detail

Decisions

All decision operations are mainly worked through simple vector dot products, vector distance comparation and simple checks of the traffic light state, thus they are relatively cheap to compute. Furthermore, decisions have been arranged in a way so that the generally most common actions (*approach*, *cross* and *arrive* particularly) are closer to the decision tree root.

- `hasCrossed`, calculates the dot product between `secondCheckpointTowards` and the vector between `secondCrossingCheckpoint` and the current agent position: if less than 0 then the agent is still crossing, otherwise it has crossed successfully
- `isGoalReached`, calculates the distance between `destination` and the current agent position checking whether it is smaller or equal than a specified parameter
- `isNearCrossing`, calculates the distance between the crossing bounds and the current agent position checking whether it is smaller or equal than a specified parameter
- `isGreen`, simply checks whether the current semaphore state is green or not

- `isYellow`, simply checks whether the current semaphore state is yellow or not
- `isHalfPassed`, calculates the dot product between `halfCrossingTowards` and the vector between the centre of the crossing and the current agent position: if less than 0 then the agent is before the half of the crossing, otherwise it is further than it
- `isBeforeCrossing`, calculates the distance between the `firstCrossingCheckpoint` and the current agent position checking whether it is smaller or equal than a specified parameter

Actions

Actions generally involve the manipulation of the steering components to set destinations, speeds or changes in behaviour weights and of the animator to select the correct animation to play for that action.

- `Despawn`, selects a new random position around the starting position, teleports the agent to it and calculates a new random destination
- `Arrive`, sets the target for the seeking behaviour to the destination and lets the agent walk towards it
- `Approach`, sets the target for the seeking behaviour to the closest point on the crossing bound from the agent's current position and lets the agent walk towards it
- `Cross`, sets the target for the seeking behaviour to the `secondCrossingCheckpoint` and lets the agent walk towards it, while reactivating collision avoidance checks if they were deactivated by the `Wait` action
- `Hasten`, sets the target for the seeking behaviour to the `secondCrossingCheckpoint` and lets the agent hasten towards it
- `Jog`, sets the target for the seeking behaviour to the `secondCrossingCheckpoint` and lets the agent jog towards it
- `Turn`, sets the target for the seeking behaviour to the `firstCrossingCheckpoint` and lets the agent jog towards it
- `Wait`, sets the target for the seeking behaviour to the `secondCrossingCheckpoint`, deactivates collision avoidance checks and lets the agent idle in place

The helper methods found in the source code `StartWalking()`, `StartHastening()`, `StartJogging()` and `StartIdling()` define exactly what we mean by walk, hasten, jog and idle.

Priority steering

Assets/Scripts/Agents/PrioritySteering.cs
Assets/Scripts/Agents/SteerUtilities.cs

To implement the steering behaviour needed for our agents, we started from the framework seen at lesson and started expanding it with various features.

Firstly, we started by using a simple weighted blending system to regulate our agent movement, but the obtained results weren't encouraging because the collision avoidance algorithm either prevailed too much or impacted too little in the final blend, even after re-tuning the parameters.

For this reason, as suggested by Millington, we decided to implement a steering based on priority groups, in which different blending groups exists and are applied following their

priority order. When a group of higher priority returns a velocity bigger than a certain epsilon, that velocity is used to steer the character and the other groups are ignored; otherwise, the next group is considered in the same manner and so on until all groups have been checked or a velocity is found.

To implement this we introduced a few changes:

```
public class WeightedBehaviours :
SerializableDictionary<SteeringBehaviour, float> { }
```

The new class `WeightedBehaviours` represents a single priority group of behaviours through a dictionary which contains references to multiple `SteeringBehaviour`, each coupled with a float representing its weight.

```
public List<WeightedBehaviours> behaviourGroups;
```

A list of `WeightedBehaviours` is thus what is used to store all the priority groups for an agent; the groups are scanned at runtime to determine which priority group will give a velocity to our agent:

```
foreach (var bhvrGroup in behaviourGroups)
{
      // Blend the group while gathering each acceleration component
      foreach (var bhvrEntry in bhvrGroup)
      {
            components.Add(bhvrEntry.Key.GetAcceleration(status) *
                        bhvrEntry.Value);
      }
      blendedAcceleration = Blender.Blend(components);
      // If the blendedacceleration is more than epsilon than return,
      // else check next group
      if (blendedAcceleration.magnitude > 0.001f){ break; }
}
```

Each `SteeringBehaviour` has the same interface, receiving a `MovementStatus` object filled with useful information which the behaviour can use to compute an acceleration.

Since, as we'll see, we are using a boid-like behaviour (`SeparationBehaviour`), we included an array of colliders, and the number of colliders present in it.

```
public class MovementStatus
{
      public Vector3 position;
      public Vector3 direction;
      public float linearSpeed;
      public float angularSpeed;
      public Collider[] neighbours;
      public int neighboursCount;
}
```

Editor script

To ease and enhance the usage of priority steering in the Unity Editor, an editor script has been developed (*Assets/Scripts/Agents/Editor/PrioritySteeringEditor.cs*): essentially, it loads all the `SteeringBehaviour` components and displays them in their separate groups, each with their own slider to control their blending weight.

As we can see from the code, instead of a standard dictionary, a `SerializableDictionary` was used to help saving the changes made on the groups via the editor interface in Unity.

Behaviours used

Here we list all satellite steering behaviours used for our citizen to move around. Each one has an added `OnDrawGizmos` function for easier visualization of the actual velocity returned.

Seek

Assets/Scripts/Agents/SteeringBehaviours/SeekBehaviour.cs

The `SeekBehaviour` is used to reach a destination: it calculates the vector towards the destination and returns a velocity scaled by various parameters (`gas, steer, brake`)

Separation

Assets/Scripts/Agents/SteeringBehaviours/SeparationBehaviour.cs

The `SeparationBehaviour` is used to keep away from the other objects in the scene in a certain range from the agent (his field of view): the more the agent is closer to an object, the stronger the repulsive force emitted

Drag

Assets/Scripts/Agents/SteeringBehaviours/DragBehaviour.cs

The `DragBehaviour` is used to smooth out both linear and angular acceleration and avoid unnatural or sudden movements.

Collision avoidance

Assets/Scripts/Agents/SteeringBehaviours/AvoidBehaviourVolumeAdaptive.cs

The `AvoidBehaviourVolumeAdaptive` is a variation of the simple volume collision avoidance, where a volume the size of the agent is projected on its forward vector at a certain distance.

Initially, we tried several collision avoidance methods, such as using three ray whiskers, a single forward ray and three-way forward boxcasts, but none of them gave desirable results since the crossing is an extremely crowded area and overturns were the norm.

However, we still needed a collision avoidance system as separation alone is not enough to handle everything.

In the end, we opted for an adaptive variation of the single forward boxcast: instead of having the projection always at the same distance (`sightRange`) in front of the agent, the projection has a max or minimum value and a growth/decay value.

```
public class AvoidBehaviourVolumeAdaptive : SteeringBehaviour
{
	public float minSightRange;
	public float maxSightRange;
	public float sightDecay;
	public float sightRegrowth;
	private float sightRange;

	public float steer;
	 ...
}
```

As long as the agent is colliding with something, the `sightRange` will decrease in decrements of `sightDecay`; otherwise, it will increase in increments of `sightRegrowth`.

This kind of behaviour helps the agent to adapt to the situation experienced: in an extremely crowded area we keep the range to the minimum, only evading those in the immediate vicinity and letting the separation behaviour do the rest; if instead the area is clear of neighbours, we can extend the sight range and evade ahead of time in a more smoothly manner.

Juggling and tuning the aforementioned parameters can help increasing the quality of the collision avoidance in a set scenario.

Utilities

Assets/Scripts/Utilities.cs

In this section we will report some of the utilities functions used in the project:

```
public static Vector3 GenerateRandomPoint(Vector3 position, float
range) {...}
```

`GenerateRandomPoint` returns a random position in the volume of a sphere of radius `range`.

```
public static Vector3 GenerateValidPosition(Vector3 position, float
range, float height){...}
```

`GenerateValidPosition` returns a random position in the area of a circle of radius `range`, at the specified `height`. It uses `GenerateRandomPoint` return value and puts it at the specified height.

```
public static Vector3 GenerateValidPositionCapsule(Vector3 position,
float height, float capsRadius, float capsHeight, Vector3 direction)
{...}
```

`GenerateValidPositionCapsule` returns a random position at the specified `height` in the area of a two-dimensional capsule of height `capsHeight` and radius `capsRadius` that spans along a specified `direction`.

It uses `GenerateValidPosition` to generate two random points in the circles at the ends of the capsule and then interpolates through them with a random alpha to get a point between them.

Furthermore, for the simulation's visualization sake, a simple camera plugin (RTS camera) was pulled from the Asset Store and used to move around the camera while the simulation is ongoing.

Examples

Demos of parameter-tuned scenarios can be found in the *Assets/Scenes* folder.

For example, we can see that in the scene "LightCrowd" the parameter `minSightRange` (0.5) for collision avoidance is higher than in the other two scenes ("Average crowd":0.3, "Crowded":0).

This is because with less agents we don't have to fear about overturning, while in the two latter scenarios agents are going to be toe-to-toe more often, thus we will need to rely more on separation than on "long-range" collision avoidance; by doing this, we can thus minimize the amount of agents stuck in the middle of the street overturning.

Conclusions and future development

All things considered, with the right parameter settings, we can adapt the citizen behaviour according to the situation we want to simulate and achieve believable results, even if we have to admit that the situation is still too "bumpy" sometimes: the collision avoidance algorithm proposed helps dealing with the problems we mentioned before but it is not by any means perfect.

However, by reading a few papers and by observing the collision avoidance provided by Unity's NavMeshAgent, we came to know about the RVO algorithm (Reciprocal Velocity Obstacles) and its extensions which, if implemented correctly, can lead to almost collision-free movement even in crowded areas.

Essentially, instead of using rays or volumes to check for collisions, it is based on the concept of **velocity obstacles**, which is the set of all the possible velocities that would result in a collision with a neighbouring agent; by choosing a velocity outside his velocity obstacle we are guaranteed to avoid collisions with that agent.

Additionally, by choosing the closest collision-free velocity to the preferred one (the one that points towards the objective), we can achieve smooth and near optimal navigation towards the goal.

Thus, to conclude, a possible improvement to this project would be to implement this kind of collision avoidance algorithm to make the simulation look even more natural and immersive.