EECS-395 Assignment 1: Locomotion

To run the code for this assignment, unzip the directory, open the Assets subdirectory of the assignment, and double-click on **defaultScene.unity**. Once you have opened unity, click the Project tab in the pane at the bottom of the screen, and make sure you are in the RPG Engine folder of the project. All the source files we will be editing are in this folder.

# Part 1: Steering

The steering behaviors are in the **CharacterSteeringController.cs** file in the RPG Engine folder. Double-clicking on the icon for it inside Unity will open up an editor for you (either MonoDevelop, or Visual Studio, if you’ve told it to use Visual Studio).

We have implemented the Seek and Stop steering behaviors. We have also implemented behavior combination and inserted a stub for a collision avoidance behavior, in the method **CollisionAvoidanceSteering()**. Fill in the body of this method to compute the unaligned collision avoidance steering behavior discussed in the reading and the lecture.

To make things simpler, we have given you all the tools you need to implement this:

* The method **Registry<CharacterSprite>()** will return to you a list of all CharacterSprite objects. So you can iterate over all characters by doing:  
    
   foreach (var sprite in Registry<CharacterSprite>()) { … }
* These objects have the character’s position available in their **Position** property, so sprite.Position tells you where they are.
* The sprite’s current velocity is in sprite**.Velocity**.
* The CharacterSprite for your own character is in the field **mySprite**.
* We’ve implemented the math functions for you:
  + **TimeOfClosestApproach**(sprite1, sprite2)  
    Returns the number of seconds in the future (or negative if in the past) that the two sprites will reach their point of closest approach.
  + **PredictedPosition**(sprite, time)  
    Computes the predicted position of the sprite the specified number of seconds in the future.

Positions are represented as Vector2 objects. These have the following members:

* v**.x**, v**.y**: the x- and y- coordinates of the vector.
* v**.magnitude**: the magnitude (length of v).
* v**.normalized**: a unit vector in the direction of v, i.e. v divided by its magnitude.
* v**.PerpClockwise**(): the vector rotated 90 degrees.

**Important:** the return type of CollisionAvoidanceSteering() is **not** a Vector2, but rather a **Vector2?** (the ? at the end is part of the type name). This means that it can be either a normal Vector2 or a null. For your purposes, all this means is that if there aren’t any impending collisions, you should return null as your force, rather than zero, because the behavior combination code will then know to completely ignore collision avoidance rather than averaging zero in with the result of the seek behavior.

Note: there was a bug in the original version of the slides, such that the formula for the time of closest approach was missing a – sign. This has been fixed in the version on blackboard.

## Debugging your steering behavior

If you click the **Display Debug Vectors** checkbox in the **Character Sprite** component of a character (use the Inspector in Unity to do this), it will display triangles indicating the outputs of the steering behaviors as follows:

* Red: force generated by the collision avoidance behavior
* Green: force generated by the seek behavior
* Blue: current velocity of the character

Keep in mind that you can pause the game using the pause button in the unity editor, and move characters around if you like (although moving them with the mouse only works under windows). But the characters are arranged so that they will try to follow the same sequence of actions, so they should collide frequently.

Important: Do not expect to make a perfect collision avoidance system. Your characters should be able to get around one another any time the run into one another in a channel that’s at least two tiles wide. But you may find that characters sometimes get wedged if they try to pass one another in a space only one tile wide. You may also find that during collision avoidance, one of the characters gets ”pushed” far enough off its path that the seek behavior will trying to move through a piece of furniture when trying to get back on its path. It will then be stuck permanently. We will try to fix this in part 3.

# Part 2: Path planning

You will find the path planning code in the file **PathPlanner.cs**. It implements the **Path Planner** component, which you will find in the GameObject called **Map** (in the hierarchy panel on the left). All characters use this one component, even though it’s stored in another GameObject.

## Implementing A\*

We have implemented the Dijkstra shortest path algorithm for you.. We have also provided stubs for two other path planners: A\* and a “neurotic” path planner. You should implement these.

First, copy the code from the **Dijkstra**() method and paste it into the body of the **AStar**() method. Since A\* is just a small variation on Dijkstra, this is a good starting point. Now just modify it to implement A\*.

## Debugging your path planner

You will find that the path planner component has two controls available to you in the unity editor. One is the **Display Overlays** checkbox. When this is checked, it will display the actual tiles searched by the planner in yellow, and the path computed in green. Since both characters use the same planner, the overlays simply display whatever was computed for the character that most recently planned a path.

The other control is a popup menu labeled **Algorithm**. You can use this to select between the different path planning algorithms. You should find that you're A\* implementation generally searches fewer tiles (less yellow) than the Dijkstra implementation.

## A neurotic path planner

Now suppose you characters have a morbid fear of house plants and try to steer clear of them when moving about the house. You can implement this by changing the way that your path planner computes the cost of a path.

We’ve added two fields to the path planner, **Phobia** and **Philia**, which you can drag game objects into in the Unity editor. You should make the Neurotic path planner avoid the Phobia object and veer closer to the Philia object, when it plots a path that goes past them.

Copy your A\* code and paste it into the **Neurotic**() method. Now modify it so that it tends to steer clear of the phobia object (you can find out what tile its on with the **PhobiaTile** property) and tends to come closer to the philia object (whose tile position is in the **PhiliaTile** property).

# Part 3: Dynamic replanning (495 students)

As mentioned before, a character will sometimes get into a stuck state during collision avoidance where they end up with a piece of furniture between them and their original path. Modify the **FollowPath** coroutine in **Path.cs** so that it notices when it’s stuck and replans a new path. This is probably easiest to do if you make FollowPath take the path planner as an extra argument so that it has a pointer to it if it needs one.

The hard part for this assignment is figuring out when you’re really stuck and need to replan, versus when you’re just in the middle of trying to get around another character. Fortunately, the CharacterSprite component can give you some information about this. The way Sprites implement physics is to first move to their new location as computed by their Velocities, and then check whether the sprite overlaps some obstacle. If it does, the sprite is moved back to its original position. In the process, it updates two fields in the sprite object:

* **IsBlocked**  
  Is set when the sprite had to abort the sprite’s motion last tick because it would have resulted in a collision.
* **IsBlockedByDynamicObject**  
  Is set when the way was specifically blocked by a moving object (in this case, a character).

So you should be able to detect a stuck state by noticing that IsBlocked is set, but IsBlockedByDynamicObject is not set.

# Turning your code in

When turning in your code, copy the files you modified:

* CharacterSteeringController.cs
* PathPlanner.cs
* Path.cs (495 students only)

Into a new directory. Then make a zip file of the directory and upload it to blackboard. Please do not make a zip file of the whole Unity project.