15-466 Computer Game Programming
Project 1

For this assignment, you will be implementing in Unity some simple movement behaviors for virtual agents.

1.) Movement Behaviors

We have provided a Unity project you must use as a framework for your code. It includes a couple of simple environments, a 3d human character model (10 instances of it), and a framework for switching environments and resetting the simulation. A download link will be emailed and will be available on Max's class website.

To complete this part of the assignment, you need to find "BehaviorScript.cs" and fill in the blanks in the switch statement inside the UpdateDesiredVelocity() method. There is a switch case for each behavior you have to implement. That method returns, for every frame, the desired velocity for the currently selected behavior. In class on 9/13 we may update the list of required behaviors.

Read the comments we have left for you in "BehaviorScript.cs"

Feel free to modify parts of the script outside of the UpdateDesiredVelocity() method itself. For example, if you want to remember something between frames, add public or private members to that script's class and use them to save information between frames. If you add anything *outside* of that script, please make note of your changes in your submission. It also might not be a bad idea to talk to John (drake@seas.upenn.edu) if you are thinking of changing something outside of that script. I do not think you should have to.

We have provided for you, at the top of "BehaviorScript.cs", a variable name “goalRef”. Every time the level is changed, the “LevelConfigurationScript” loads each character's goal GameObject into its corresponding “goalRef”. For this assignment, you may assume that goalRef doesn't change after initialization, however if you use this reference instead of caching its position, you could form complex behaviors like “follow” by affixing one character's goal to something that moves.

Read the appendix section at the end of this document for more detailed information on how the provided Unity project is set up and for tips.

2.) Obstacle Avoidance

On top of the behaviors themselves, you must also implement obstacle avoidance. Obstacle avoidance should take some level of priority over the movement behaviors. That is, if a movement behavior would have the character walk through a wall, do not allow the character to walk through the wall. Do not rely on the physics engine for this; in grading, all collider components in obstacles will surely be disabled or absent. It is okay if this produces situations where a character appears confused and stuck, like in a maze. Later assignments will look at more intelligent movement.

Also, do not let characters fall off the edges of the map. For now, you can assume the map ground planes will always be centered at the origin and have dimensions of 200m x 200m.

For this assignment we will assume that all obstacles are cylindrical in shape (oriented vertically). We have prepared for you two lists named “obstacleRefs” and “obstacleCollisionCylinderRefs” at the top of "BehaviorScript.cs" for you to access the locations and sizes of the obstacles. You actually may only need the latter list.

obstacleCollisionCylinderRefs[0].transform.position
yields the position of the collision cylinder for obstacle 0.
obstacleCollisionCylinderRefs[0].transform.lossyScale
yields the approximate scale (in world units) of the collision cylinder for obstacle 0. It is only approximate because combinations of rotations and scale transformations in the hierarchy above that collision cylinder might skew its shape. For our purposes, and since this is a cylinder, we should never encounter this skewing or squashing (as long as you do not scale your obstacles non-uniformly), so you can read either the X or Z component of lossyScale as the diameter of the collision cylinder.

3.) Documentation

Explain in a few short paragraphs what approach you took for each implemented behavior. Pseudocode is not good enough; please use English.

Also, if you changed anything outside of the "BehaviorScript.cs" file, please let us know so that we do not inadvertently break your project.

4.) Submission

Submit your entire Unity project by compressing it into a .zip or .rar archive. .tar.gz/.tgz is fine too.
If you have time, use a screen-recording program like CamStudio or Fraps (my favorite; works the best) to record a short video of characters walking with each of your implemented movement behaviors and include these in your submission archive. This is not required, but it may help resolve problems or confusion while grading and would only act in your favor.

It will probably be too large to email it to John, so we are working on setting up something else. Look for an email in the next few days about submission for Project 1.

The deadline is September 20 at 11:59PM

Grading Criteria

Your submission will be graded on these criteria:

- realism of the motion (e.g., no instantaneous velocity switching, no oscillations, no collisions, no slowdowns due to computationally-intensive operations)
- concise/clear documentation
- support for all the required behaviors

Appendix: The Provided Unity Project

We will try to explain how the project is initially laid out and try to point out some tips and limitations here. If you have any doubts, please email John (drake@seas.upenn.edu).

<u>Scenes</u>

To open the project, open "InitialScene.unity" in the "ProjectFiles\Assets\P1 Scenes" directory. To change scenes while the game is running, click the "Show Load Buttons" button on the GUI.

"InitialScene.unity" has as its roots a blank flat environment ("Ground"), an object called "Level Configuration", and an aptly-named object "Stuff that won't get destroyed as levels are loaded".

As you might expect, the things which are organized within "Stuff that won't get destroyed as levels are loaded" do not get destroyed when one level is loaded to replace the current level. This behavior is courtesy of the "Don't Unload Me Script" script which is attached as a component to the "Stuff that won't get destroyed as levels are loaded" object (you can see it there in the Inspector view). We put the characters, camera, and sun light in there so that they do not have to be specified in each and every level file. However, "Level Configuration" and its children must be present in every Scene (level file). In this way, the Scenes or "Levels" which are not "Initial Scene" represent different simulation configurations.

Loading "Initial Scene" is a special case: everything is destroyed (even the stuff in "Stuff that won't get destroyed as levels are loaded") so that when it is loaded, the things in "Stuff that won't get destroyed as levels are loaded" are not duplicated as that scene is loaded on top of itself.

"Level Configuration" is organized into "Goals," "Start Positions," "Obstacles," and "Behaviors". "Start Positions" are numbered to correspond with the characters in the scene. So, there should be at least as many starting position objects as characters. The start position objects are flattened cylinders with their Collider physics Components removed (so that characters can walk through them). The goal objects are similar. "Behaviors" are also numbered to correspond with the characters in the scene. Each "Behavior" GameObject has a "BehaviorContainer" script Component attached to it. This script merely serves to hold a public variable named "Behavior" by which you can change the movement behavior the corresponding character should perform. Its value is returned when you call getDesiredBehavior() in "BehaviorScript.cs".

If you select "Level Configuration" you will see in the Inspector window that there is a script attached to it. I wrote this to perform some setup actions whenever a level is loaded. This includes moving characters to their start positions from wherever they happened to be before and starting the characters' controller scripts (which starts them moving, which starts calling the code you write in "BehaviorScript.cs"). In the inspector, you will see a "Number of Characters To Search For" parameter. It should be left at 10, to match the number of characters instantiated in the scene.

<u>Virtual Character</u>

The included virtual character object is from the standard assets provided with Unity. It is already textured, rigged, and animated; you will only have to write code. We modified the Unity-provided controller script to take inputs from your code instead of keyboard or joystick input. We also changed some of the tuning parameters to reduce smoothing effects, making the character respond essentially instantaneously to changes in the output of your movement behavior code.

You probably want to leave the character objects alone except to edit their "BehaviorScript.cs" file. If their placement makes it hard to click on something like a start or goal point, feel free to move the character object out of the way. The “Level Configuration” script will reset the character's position when the simulation starts anyway. When you edit the "BehaviorScript.cs" script, it changes it for all characters (you don't have to go through changing the script for each character).

It would be in your best interest not to rename the characters, or a lot of the setup done in “Level Configuration” will fail, since it looks for character objects by their names. The same precaution goes for start and goal points; don't rename them. If you rename something and then stuff breaks, try reverting the name change.

Creating a Custom Level/Environment

Copy one of the existing scenes which are not “Initial Scene” and give it a new name. Move around the starting points and goals as you like. Move around obstacles as you like. Copy obstacles to make new ones. The root of each obstacle must have the “Obstacle” tag selected at the top of the Inspector. Change the Behavior object settings as you like. If you try to run the game, it will not automatically recognize your new level. To get it to show up in the level load list that appears on the game's GUI when the “Show Load Buttons” button is clicked, open your new custom scene and then click on “File->Build Settings”. On the window which pops up, click the “Add Current” button or drag your scene from the Project pane onto the list. You can reorder the levels by dragging them around in the list. Keep “Initial Scene” at the top of the list.

Some notes on Unity

The “Find” and “Get” methods you will find in Unity let you access other GameObjects and Components in the scene, rather than the one the script is attached to. The GameObject the script is attached to is accessible with “gameObject”. The scene graph hierarchy is available via a GameObject's “.transform” member. For example to find a child of the object a script is running in, use “gameObject.transform.FindChild()”. To find another component attached to the same object the script is in, use “gameObject.getComponent()”. If you know the type of the component you want to get (for example, another script named OtherScript), you can use the templated overload for getComponent (in the example, “OtherScript ref_to_other_script = gameObject.getComponent<OtherScript>();”. This avoids having to cast a Component to the type you need it to be.

These methods that search through the scene should not be called every frame. Call them once and cache the result if it needs to be used again later. This is what the provided project does, for the most part, in “LevelConfigurationScript.cs”. It finds and caches a number of important references to things in the scene.

I have not quite figured out the most reliable way to get the debugger working. One time while running my project in Unity with the debugger attached, I closed Unity and it asked if I would like to save my changes so I clicked Yes. When I opened Unity again, the scene I had open before was entirely blank. Another time with the debugger attached, Unity crashed. So, be careful and make backups.

To print information to a console, use “Debug.Log()” and open the Console view, available in Unity's Window menu.