Animation Foundations. Unique Delivery

Introduction

You are asked to finish a prototype of a game where you shoot a ball against a goal-keeper, which is an octopus. **Before** you start implementing anything, please notice that: Pressing W,A,S,D will move the targets, pressing space or the mouse button will notify a shooting.

Unity version 2018.4.6

Requirements (0 points)

Please make sure you comply with the following requirements:

1. You follow the git flow conventions. In particular, each exercise is in a separate branch, and the final commit of this delivery will be in a merge of the develop branch into the **master branch** with a tag containing **delivery_unique**.
2. File organization. You should provide a github repository where the user davidperezgallego@enti.cat is included. Inside, there should be:
    - A folder called *Octopus_library* with a .gitignore properly configured in the same folder where your .sln file is.
    - A folder called *Unity_project* with a .gitignore properly configured and the relevant files
    - A folder called *Binary* with a binary file called *Delivery.exe*, in addition to the complementary files for the binary to work.
    - A READMEunique_delivery.md file, as well as the READMEunique_deliver.exe

Requirements 1 and 2 are mandatory. This delivery will not be corrected if your repository does not satisfy them.

Exercise 1. Game logic for football (1 point)

Implement the basic mechanics of shooting the ball and the octopus picking it. (1 to 4 count **0.25 points** each)

1. Use keys W,A,S,D to adjust the direction of the shoot
2. Use space to adjust the Strength slider: the longer the press, the stronger the shoot. When it arrives at the end, it decreases. When you unpress, you shoot.
3. Make a shoot consistent with the direction and the strength applied to the ball
4. Make the ball reappear at the initial position so the game can be played indefinitely

Exercise 2. The Magnus effect (2 points)

We will implement the Magnus effect. (1 to 6 count **0.33 points** each) 1. Use keys Z and X to adjust the *Effect strength* slider of the GUI, as well as the horizontal position where the impact will occur, as shown by the small red sphere 2. Calculate the rotation velocity according to the position and direction of the shoot. Show the value in the canvas element called *Rotation velocity text*. Use positive values for clockwise rotations, and negative values for counter-clockwise rotations. Express it in degrees/second 3. Use key I to toggle on and off information on the trajectory. When toggled on, draw the calculated trajectory of the ball as if it had no Magnus effect in grey, and with blue points the calculated trajectory of the ball with the effect. Make both appear before shooting the ball, using the prefabs in the folder *Arrows4Physics*. 4. When information of the trajectory is on, show also the instantaneous velocity (green arrows) and forces (red arrows) and visualize it with the prefabs included in the folder *Arrows4Physics*. In addition, use a grey arrow to show the initial force that will be applied to the ball when shooting. 5. In the README write the formula used to calculate the rotation velocity of the ball, depending on the position and the direction of the shooting. Make a diagram and include it as a .png if it is necessary for your explanation. 6. In the README file Write the formula used to calculate the forces at every instant of time between the shoot and the arrival at the goal. Make a diagram and include it as a .png if it is necessary for your explanation. Consider at least strength of initial impact, air friction, and gravity. Make the elements in the GUI adjust consistently with these values.

Exercise 3. Character animation (3 points)

In this exercise we will add the character animations to the game. You must create a character and make him act as the ball shooter, implementing the following logic of the animations:

- When the player shoots, the character runs to the ball, shoots, and the ball leaves the spot according to the defined strength and effect.
- If the player scores a goal, the character starts dancing, and the robots forming the audience raise their arms enthousiastically, up to when they are stand, and sit back down. The onset of the animation will be delayed in order that they make, collectively, a wave, while cheering
- If he does not score, he approaches the referee and shakes his hand
- After, all participants go back to the initial position

Specifically, you are asked to: (1 to 3 count **1.0 points** each)

1. Create the animation of the robots with keyframes in Motion Builder. One single animation should suffice for all of them. Import the animations in Unity and trigger it in such a way that the animation is triggered delayed to create the wave effect.
2. Transfer the hand shake to the character named Malcolm and to the new character that you will have created (use https://charactergenerator.autodesk.com/). In Unity place the character, and adjust its positions in order for it to fit well with the other elements. The shooting animation, as well as the dancing animation, also needs to be transfered to the character you have created and import it into Unity.
3. Integrate the animations and implement the logic described using boolean variables and Mecanim.

Exercicse 4 (4 points)

You are asked to build a library to control an octopus acting as a goal-keeper in a Unity3D game. The way the octopus arms work is the following: all arms should move randomly keeping the end-effector within a given region. When the user presses the shoot key, the arm taking care of the region where the target is should move to the target, and stay there until the ball arrives. Then the game restarts, the ball goes back to the origin, and the octopus goes back to random movements.

1. In class *MyTentacleController* you implement the method *LoadTentacleJoints* to set up all the joints. Implement also the logic by which each tentacle switches target when the ball is shot in each region. Make the Octopus reach and stop the ball only once every two shots, in alternate shots.
2. In class *MyOctopusController* you implement FABIK and CCD IK methods to move all the joints. Check which has the smallest iterations. You also constrain

the twist of all the joints in order that the suckers of the arms always look towards the camera.

3. In class *MyOctopusController* implement the gradient-based method. Add a novel error function that makes the arm slowly generate undulating waves of movement on the arm, going from the body of the octopus to the end-effector that is in the target position.

Hint: There are at least two strategies to solve this exercise. In one of these strategies, it is useful to limit the movement of each joint to one fixed axis, alternating the axis between successive joints.

4. Complete the following text in the *README_unique_delivery* file, replacing the text in bold :

> Overall, I think that the IK method **FILL_IN** works best for this scenario because **PUT_HERE_A_MOTIVATED_EXPLANATION** .

The motivated explanation should explain how you implemented the two IK methods, including pointers to the relevant lines of code, and discuss the advantages and drawbacks of each method, and eventually how you fixed the drawbacks of one or another method, if you did so.