Assignment 4: Character Animation and Simulation

CS 248 Winter 2017-2018

Due Date: Monday, 26 February by 6:30pm

Introduction

In the previous two assignments we explored animation and simulation for simple game objects. In this assignment, we will extend our use of animation and simulation more complex characters and articulated rigid bodies. The use of characters in interactive graphics involves a wide array of interesting topics such as rigging, skinning, and motion capture. In this assignment we will focus on three specific cases split into independent parts: creating a state machine for animations in Unity, creating an articulated rigid body in Unity, and completing an Inverse Kinematics system in `C++`.

1 Animation in Unity

Animations are managed in Unity by using an animator component and an animator controller in conjunction with a rigged character model. An **Animator Component** is used to control the animation of your character in Unity. It requires an animator controller to manage the various animations a character may have. An **Animator Controller** is the primary component you will work with. It is essentially a state machine that allows you to transition smoothly between different animations. You can add animation states to your controller and specify transitions between those states. An animation transition occurs when one of its conditions is satisfied. These conditions can be based on float values, integers, booleans, or triggers.

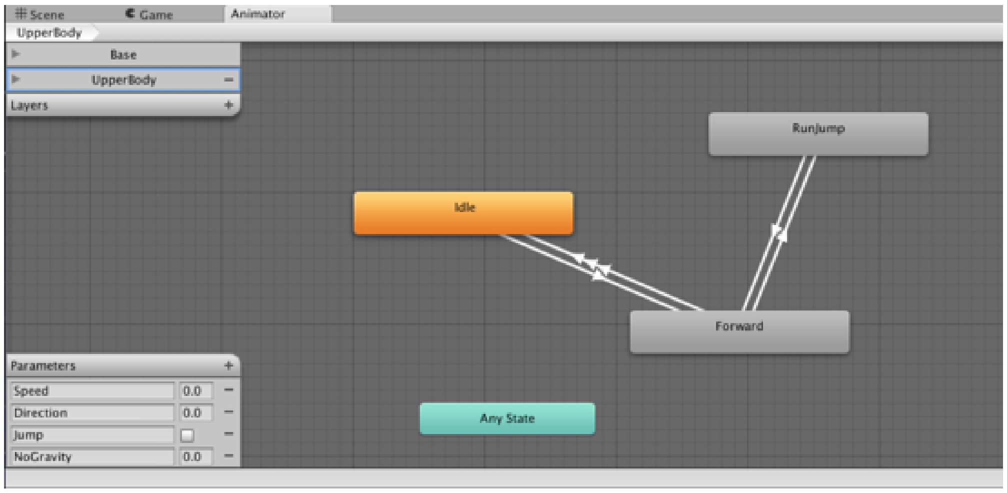

Figure 1: The Animator window in the Unity editor

Your task is to use an Animator Component and an Animator Controller to create a character with a set of nice animations and smooth transitions between those animations. We have provided an adapted version of the rigged model from Unity's Character Controller package. This model has four animations: idle, walking, running, and jumping. If you choose to use this model, you must create a scene where the following criteria are met:

- The player can move the character around the scene.
- The character plays the idle animation when stationary.
- The character transitions from idle to walking when it begins to move slowly, and from walking to idle when it stops moving.
- The character transitions from walking to running when it begins to move quickly, and from running to walking when it slows down.
- From any of the other states (idle, walking, or running), the player can press a key to make the character jump. The jumping animation should play at this time.

Alternatively, you are free (and indeed, encouraged) to create or download your own rigged models and animations and build a state machine for the animation transitions using those assets instead. However, anything you create should be at least as complex as the above specifications.

2 Creating an Articulated Rigid Body in Unity

Unity's physics engine supports three types of joints:

- A spring joint, as the name implies, acts as a spring between two objects. These joints expose controls for their stiffness and damping, in addition to other parameters.
- Hinge joints allow an object to rotate at its anchor point about a specified axis.
- Fixed joints "lock" two objects together. However, you may specify a force threshold for these joints such that the joint breaks when a force exerted on it exceeds its threshold.

Your task is to create a scene in Unity with one or more articulated rigid bodies. You will build these structures by attaching game objects to one another using these joints. You will also implement a way for the player to exert forces on these articulated bodies in order to demonstrate the behaviors of their joints (e.g., exert a force on a body by clicking it, or by pressing a key, or by creating a simple game where the player can shoot other rigid bodies at the joints).

3 Inverse Kinematics

In the previous homework, you compiled `C++` source code to libraries that you then used in conjunction with Unity. The process of iterating over writing code, compiling, and testing in Unity can be cumbersome and time consuming. Another approach is to develop and test your `C++` code independent of Unity, then integrate your library once you are satisfied with its functionality.

You are given starter code for an inverse kinematics system that is incomplete. Currently, the starter code simply displays a stationary articulated rigid body and prints the mouse coordinates to the console whenever the mouse is clicked. Your task is to complete the code by implementing the `updateJacobian` and `updateEndEffector` functions in the `ArticulatedBodies` class. When implemented correctly, the articulated rigid body will adjust its end effector to reach towards any point clicked by the mouse. Note that you are not required to handle collisions between the blocks of the articulated body. You may write and debug your code in the development environment of your choice, and you do *not* need to integrate it with Unity.

Each iteration of the IK solver has five steps:

- compute the Jacobian matrix $\mathbf{J}$,
- compute the change of end effector positions $\delta\mathbf{g}$ based on the current and the target effector positions,
- compute the change of joint angles $\delta\theta$ based on $\mathbf{J}$ and $\delta\mathbf{g}$,
- update joint angles θ, and

- update the positions of end effectors $\mathbf{g}$.

The starter code already implements the second, third, and fourth steps. Your task is to implement the first and fifth steps to calculate the Jacobian matrix and the end effector positions.

In `updateJacobian`, $\mathbf{J}$ is an $n \times m$ matrix where n is the number of DoFs of the end effector and m is the number of DoFs of the joints. In our code $n = 3$ because we specify only the position and not the orientation of the end effector. Furthermore, all of our joints are revolute (rotational about a single axis). Thus, each joint j has one column in the Jacobian matrix, which can be calculated as $\mathbf{J}_j = \mathbf{v}_j \times (\mathbf{s} - \mathbf{p}_j)$. Here, $\mathbf{v}_j$ is the unit vector pointing along the axis of rotation for joint j (the rigid bodies in the starter code move only in the xy-plane, so $\mathbf{v}_j$ is always a unit vector along the z-axis). $\mathbf{s}$ is the position of the end effector, and $\mathbf{p}_j$ is the position of joint j.

In `updateEndEffector` you will need to calculate the position of the end effector $\mathbf{e}_1$ in the world coordinate system using the calculated joint angles θ and the known lengths $\mathbf{L}$ of the rigid bodies. You must properly rotate and translate based on the angle of each joint and the length of each rigid body of the skeleton in order to get $\mathbf{e}_1$. Once you have finished these two functions, you should be able to control the robot arm with mouse input: the end effector should move towards the target position you set by clicking.

Grading

The assignment will be graded out of 10 points according to the following criteria:

- 2 points: Create an Animator Controller with a state machine that includes all the transitions enumerated in the Animation section above (or a controller for your own animations that has equal or greater complexity).
- 1 points: Create a scene where the player controls your character. You must be able to demonstrate each of the prescribed animations and transitions in this scene.
- 2 points: Create one or more articulated rigid bodies in Unity that contain at least one of each type of joint: spring, hinge, and fixed.
- 1 points: Create a scene in which the player can apply a force to each different type of joint in order to demonstrate their behaviors.
- 4 points: Complete the two empty functions in the IK `C++` code. Demonstrate using the provided OpenGL user interface that the end effector correctly moves to positions that the mouse clicks.

Resources

- Unity Tutorial on Animation: `http://unity3d.com/learn/tutorials/modules/beginner/animation/animator-component`
- Tutorial on using joints in Unity: `http://unity3d.com/learn/tutorials/modules/beginner/physics/joints`
- Survey of Inverse Kinematics: `http://www.math.ucsd.edu/~sbuss/ResearchWeb/ikmethods/iksurvey.pdf`