Project 2: Character Animation

Due Date: Friday, March 10th, 11:59 PM

1 Introduction

The technique of motion capture, or using the recorded movements of a live actor to drive a virtual character, has recently become very popular in video games. Characters that are animated with motion capture data appear very realistic and exhibit subtleties of human motion that are hard to replicate, even by a skilled animator. The amount of time required to produce an animation sequence is also much less with motion capture than with traditional animation techniques.

There are a number of different formats in which motion capture data is stored (Dojo supports the ASF and BVH formats), but the general idea is to represent the character as a skeleton, or a hierarchy of bones. At the top of this hierarchy is the skeleton's root, whose orientation and position in the world defines the entire character's orientation and position (these are the global degrees of freedom). The position and orientation of each bone down in the hierarchy is defined in relation to the bone's parent (the bone directly above it in the hierarchy). Each parent and child pair is connected by a joint that can have from 0 to 3 degrees of freedom (these are the local degrees of freedom; for example a hinge joint = 1 dof, a ball joint = 3 dof). Each frame of motion capture data contains values for all of the skeleton's local degrees of freedom (defining the skeleton's pose), as well as the position and orientation of the skeleton's root in relation to the start frame of the motion sequence (these global degrees of freedom are adjusted as necessary to position the character in a scene).

In this assignment you will use motion capture data to animate a humanoid character in your game. Specifically, you are required to use a *motion graph*[1] and come up with a scheme for controlling your character in-game. This animated character should replace the character entity that you used to navigate the game world in Project 0. For extra credit, you can integrate your control scheme with the path-planning algorithm from Project 1 to create intelligent, animated humanoid NPCs.

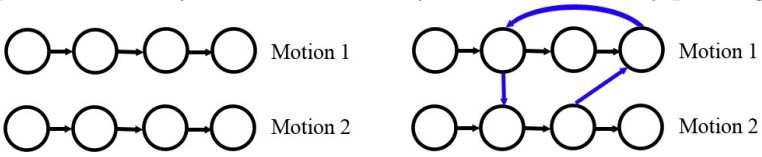
2 Motion Graph

2.1 Structure

A motion graph is a directed graph whose nodes are animation frames. In its simplest (and not very interesting) form, a motion graph just contains a number of node chains, each corresponding to a recorded motion sequence (diagram

[1] Lucas Kovar, Michael Gleicher, and Frédéric Pighin. "Motion graphs." ACM Transactions on Graphics, 21(3):473-482, July 2002.

below, on the left). The motion graph becomes useful when new transitions (edges) are added to connect different motion sequences or different parts of the same motion sequence (diagram below, on the right). The goal is to create a graph such that any traversal of it will yield a valid, visually pleasing motion.



Remember that each frame of motion data stores the position and orientation of the skeleton's root relative to the start of the sequence that contains the frame. When a transition is added to the motion graph, it is necessary to record the transformation (rotation and translation in 3-space) that is needed to align the root across the transition. To understand intuitively why this is necessary, imagine a motion graph that contains a single walking sequence, where the only additional transition is one from the last frame to the first to create a cycle. If the root is not properly aligned across this transition, then the character will simply jump back to its original starting position to repeat the motion over and over, whereas the desired effect is to have him continue walking off to infinity.

2.2 Creating new transitions

In order to generate motions that are as smooth and visually pleasing as possible, transitions must be created only between frames that are "similar" to each other. For example, a transition from a frame where the character is lying on his back to one where he is jogging would be very undesirable. There are a number of ways to measure similarity between frames, but the support code that you are given for this assignment uses a method that compares the orientations of the character's joints between the two frames.[2]

Once the distance, or transition cost, has been computed for each pair of nodes in the graph, it is up to you to decide where to actually create transitions, based on the costs. A typical approach is to select a threshold and to create a transition for each pair of nodes whose cost is below the threshold. However, you will probably find that choosing a single good threshold for all your motion sequences is difficult, if not impossible. Instead, you may want to set different thresholds for different motion categories, for example one threshold to be used for determining transitions between walk sequences and another to be used for transitions from a walk sequence to a run sequence. Bad transitions will result in ugly, jerky motions, so it is important that you leave yourself enough time to experiment with this part of the assignment and to come up with a good solution.

Even if you end up finding a really good transition, you will still most likely see jerky motion if you simply switch from one motion sequence to the next

[2]For more information, see: Jim Wang, and Bobby Bodenheimer. "An Evaluation of a Cost Metric for Selecting Transitions between Motion Segments." Eurographics/SIGGRAPH Symposium on Computer Animation, 2003.

at the transition point. In order to get a smooth transition you will need to interpolate between the two motion sequences in a window around the transition point. You can either perform this interpolation on the fly as you traverse the graph, or you can insert the interpolated frames as new nodes into the motion graph when you create a transition. Note that a longer interpolation window does not necessarily translate into a better looking motion. A window that is too long can produce very unrealistic motion and the ideal window is somewhere between 10 and 30 frames long. Interpolation is not required to for this assignment, and it will be considered extra credit.

2.3 Pruning the graph

When you've created all of your transitions, your motion graph will likely have a number of dead ends, nodes that have no edges leading away from them. The final step in creating the motion graph is to remove all of these dead ends from the graph. However, since this is not particularly interesting, we're leaving it as extra credit. A simple alternative method for getting rid of dead ends is to take the last node of each motion sequence and create a transition from it to the first node of the respective sequence. Whichever method you decide to use, make sure that your character motion avoids unpleasant motion induced by bad transitions.

3 Controlling the character

Once you have a motion graph set up and ready to go, you need to come up with a way to use it to animate your character based on the player's input. As mentioned earlier, any walk along a motion graph should yield a plausible motion, however while a random traversal of the graph might make for a good screen saver, it's not particularly useful for an interactive video game. It is up to you to come up with a control scheme for your character: a strategy for choosing the next path to take in the motion graph based on the current state and the player's input. We are deliberately leaving this part very open-ended because every game will require a different control scheme. Be sure to put a lot of thought into how you're going to approach this problem. A bad control scheme can destroy a game, while a particularly good one can be what makes a game stand out. Feel free to discuss any ideas you might have with the course staff.

Whatever control scheme you come up with, you should make sure that it is sufficiently responsive to the player's input. This means that if the player indicates that the character should turn left, he should not have to wait 10 seconds for that action to take place (unless of course having laggy controls is a design decision, if you're trying to make, say, a drunken boxing simulator... in that situation though you will have to make a *very* good case to the course staff). This issue relates back to the structure of your motion graph. If you find that your controls are very laggy, maybe there are not enough transitions

in your motion graph.

4 Support code

As with most of the materials that we provide you with in this course, you are completely free to ignore this support code if you have another way that you would like to do this assignment. We're giving you an almost complete implementation of a motion graph so that you can focus on working on your control scheme without having to worry about details like what the heck is a quaternion. If something about our implementation is incompatible with an idea that you want to try, or if you think that it's just plain stupid, feel free to change as much or as little as you like, though please list any changes you make in your README for the sake of the graders' sanity.

4.1 MGHumanoid

`MGHumanoid` is a subclass of Dojo's `Humanoid` class that is used for handling motion capture data. The methods that you need to worry about are:

- `static MGHumanoidRef create(const std::string& name, const std::string& filename, bool canMove = false);`

  This is a static method that returns a reference-counted pointer to a new instance of `MGHumanoid`. (Note that you always want to call this method and not the constructor when creating a new `MGHumanoid`).

`name`	A name that you assign to this humanoid.
`filename`	The name of the file containing the skeleton for this humanoid (a BVH or ASF file).
`canMove`	Always set this to `false`.

- `Humanoid::Motion loadMotion(const std::string& filename);`

  This method is inherited from `Humanoid` and is used for loading motion data from a file (either AMC or BVH). The file must contain motion data that is compatible with the skeleton data from the file was used to create the `MGHumanoid`.

- `void setMotionGraph(MotionGraph* graph);`

  Use this method to set the humanoid's motion graph once you create it.

- `bool getNextFrame(const double & dt, Frame & frame);`

  This is where the graph traversal is performed. At each simulation step this method is called to see how the humanoid should be posed at this

step. You have to fill this method with the control scheme that you devise. The method returns `true` if a next frame is available (i.e. no error occurred).

`dt`	The elapsed time since this method was last called.
`frame`	Passed by reference, you use this variable to return the `Frame` of the `MotionGraphNode` that you determine should come next in the graph traversal.

4.2 MotionGraph

`MotionGraph` is an almost complete implementation of a motion graph. Its most important methods are:

- `void addMotion(Humanoid::Motion* motion);`

  Adds a motion to the graph.

- `void computeDistance(MotionGraphNode* a, MotionGraphNode* b);`

  Performs a similarity measure between two nodes in the motion graph (the obtained cost is stored in `MotionGraphNode::distanceMap`). This is already written for you. Tweak the `m_velocityWeight` member variable to specify how much weight is given to the bones' rotational velocities as opposed to just their orientations when calculating this cost.

- `void computeTransitions(MotionGraphNode* node);`

  Creates viable transitions out of the node (call `computeDistance()` first). You need to fill this method in.

Note that you will likely need to add some data to the `MotionGraphNode` class based on how you decide to set up your control scheme. For example, you might want to assign a type to each transition to help in choosing the correct one when you traverse the graph.

This is how you should initialize your `MGHumanoid` and `MotionGraph`:

```
MGHumanoidRef mgHumanoid = MGHumanoid::create("mgHumanoid", "myMocapDir/f_wlk1.bvh");
World::world()->insert((EntityRef)mgHumanoid, CoordinateFrame()));

MotionGraph* motionGraph = new MotionGraph("mgraph", mgHumanoid->skeletonName);
motionGraph->addMotion(&mgHumanoid->loadMotion("myMocapDir/f_wlk1.bvh"));
motionGraph->addMotion(&mgHumanoid->loadMotion("myMocapDir/f_wlk2.bvh"));
mgHumanoid->setMotionGraph(motionGraph);
```

5 Motion capture data

A collection of motion capture data is available to you in `/course/cs196-2/pub/mocap/`. Additionally, you can find more data at these sites:

- mocap.cs.cmu.edu
- www.bvhfiles.com

6 Requirements

Your grade for this assignment will be based on the following:

Quality of transitions	Do you perform transitions at all? How smooth are they? Is there an intelligent strategy for making transitions?	40%
Control scheme	Can switch between at least 3 separate actions. How responsive are the controls?	40%
Creativity	Is the control scheme interesting or well-designed? Is it fun/intuitive/useful/cool? Or is it boring/overly complicated/frustrating?	20%

Please explain your strategy for creating transitions in a README (for example, if you decided to set a bunch of thresholds, explain how you arrived at their final values). Also talk about the reason why you chose your control scheme, why you think it's good and any issues that you think still need to be resolved (remember to actually provide instructions for controlling your character as well). The README must be a text file (.doc file submitters will be prosecuted!).

7 Extra credit

There are a number of things that you can do for extra credit, though remember to get the core assignment working first before attempting any of these.

- Integrate the motion graph with your path-planning algorithm from Project 1 to create animated humanoid NPCs.
- Interpolate between frames around transition points to get smoother transitions.
- Implement a proper graph pruning algorithm to get rid of dead ends.
- Write methods to save/load the motion graph to/from a file. In a real video game you don't want to be creating the motion graph from scratch every time you start the application. Instead the graph should be built once, saved to a file, then loaded by the game.

- Do something that really impresses us. Be creative.

8 Handing In

8.1 What to Hand In

Make sure that you hand in all of your code as well any maps and textures. If you used any mocap files that were not provided by us, please make sure that those are included as well. Along with this, you must handin a README in which you explain your design choices.

8.2 Handin Script

IMPORTANT: Make sure to clean your solution (Build>Clean Solution in Visual Studio) before handing in.

To handin your project, run /u/course/cs196-2/bin/cs196-2_handin asgn2 in your project's base directory.