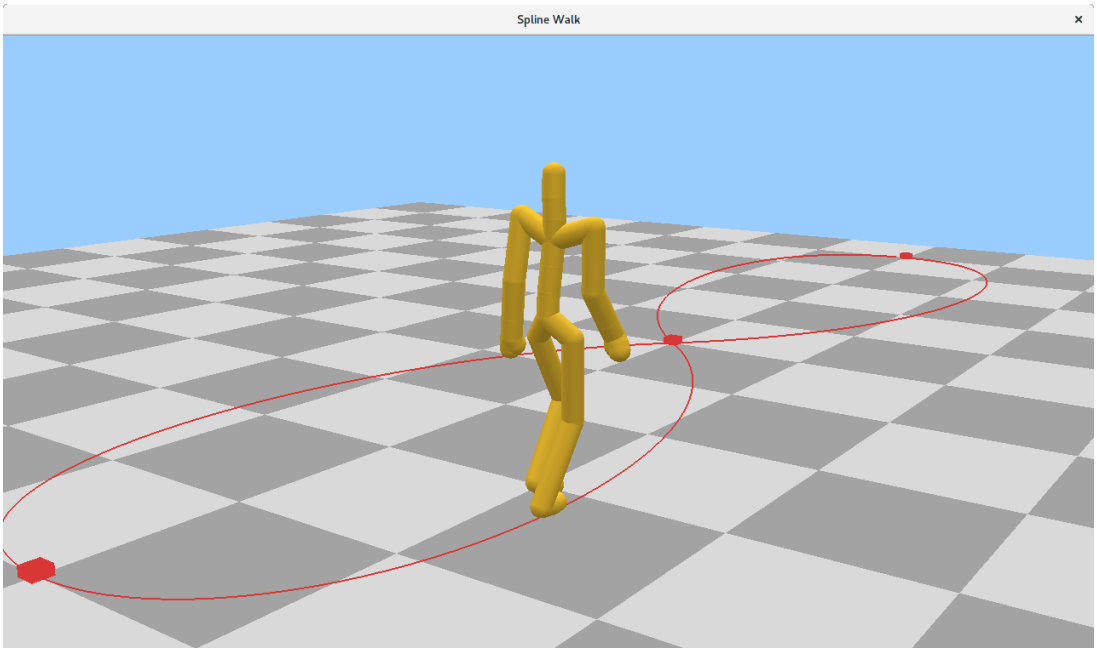
CSCI 4611: Programming Interactive Graphics and Games

Assignment 4: Walk the Spline

Handed out: Wednesday, March 22 (last updated: Monday, March 27)

Due: Wednesday, April 5

Introduction



In this assignment, you will be working with data from the CMU Motion Capture Database. The data we give you will only be in the form of text files, but with your programming and math skills, you will bring this data to life!

The CMU motion capture data is typical of all the skeleton-based motion capture data you'll find in today's games and movies. So, gaining some experience with this type of animation is one of the most important goals of the assignment. The key concept you'll need to work with this data is understanding how to compose transformations together as you render a scene graph. To keep things simple, our rendering will be a pretty simple stick figure.

The other important goal of the assignment is implementing a simple spline curve to define the path of the character. We will see how existing animation data of a character moving uniformly in a straight line can be modified to follow a specified trajectory instead.

In this assignment, you will learn:

- How transformations should be composed to create animated characters
- How to create, navigate, and render a hierarchical model
- How to implement a spline curve to represent smooth functions
- How to transform the speed and orientation of a motion-captured animation

CMU Motion Capture Database

The CMU Motion Capture Database contains 2,605 different motions, most recorded at 120Hz. These motions range from the simple (person walking straight forward), to the complicated (directing traffic), to the silly (someone doing the "I'm a little teapot" dance).

For this assignment, we've found a simple walk motion and chopped up one cycle of it so that it repeats. Playing this back in a loop while translating the character uniformly along *z* results in an infinite walk. First, you will get this to display by drawing the limbs of the character correctly. Next, you will modify your program to make the character walk along an arbitrary spline curve instead of along the *z*-axis. Note that the motion of the character's right arm is a little jittery in the original data, so don't worry about that as long as the rest of the motion is OK.

You are encouraged to search through the CMU database and see what other motions strike your fancy. Once you complete the first half of the assignment, you should be able to simply swap in new motion data files to view them. This can be fun!

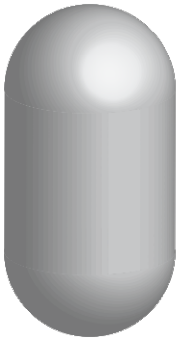
The motions in the CMU database use a skeleton specified in `.asf` files and a separate motion in an `.amc` file. The `.asf` files specify bone names, directions, lengths, and the skeleton hierarchy. In a bone's local coordinate system, the translation from the start of the bone until the end can be found using the bone "direction" and "length" — our support code provides a shortcut for this, look for `getBoneVector()` inside the `Bone` class. The length of each bone and its resting direction does not change from frame-to-frame, but the angle of the bone relative to its joint does change. The rotation angles to apply at each frame are specified in `.amc` files.

Requirements and Grading Rubric

We provide code that parses and loads the CMU data into a hierarchical skeleton data structure. We also provide a utility function that returns the bone's current rotation relative to its parent. We provide you with code that advances through the time in the animation. You will need to implement functionality to composite the transformations in this skeleton to connect the bones together and make your character walk. To accomplish this, you need to add your own code to the `Bone` and `Character` classes in `character.hpp`. Look for the `draw()` routine in each of these classes – this is where most of your code will go. The

comments in the support code should help you get started. Think of `Character` as the root node for your character's scene graph. For a human character, this root node usually corresponds to the pelvis bone. Therefore, the root node usually has 3 "bones" extending from it, one for each of the legs and one for the torso. You need to draw each of these bones within the root node's coordinate frame, which will change frame-to-frame based on the animation data. Inside each bone's `draw()` routine, you'll need to draw the bone itself (see next paragraph) and then draw all of the bone's children. Look back at the slides from class and remember how we use `glPushMatrix()` and `glPopMatrix()` to render scene graphs.

To draw a bone, first start with just drawing a line segment from the origin to the bone vector. Once you get a stick figure made of lines animating, your next task is to render solid bones instead. To help, we've implemented a function `Draw::unitCylinderZ()` for you to use; as the name suggests, it's a cylinder with unit radius and length, with axis from the origin to (0,0,1). You will need to apply a transformation to make its axis match the bone vector. We suggest using a radius of 0.05, and adding spheres of the same radius at both endpoints to fill in the ends. This creates a 3D shape known informally as a "capsule", shown on the right.



Next, you will implement a cubic Hermite spline by adding code to the `Spline3` class. The data the spline contains is simply a collection of control points, each at a specified time and with a specified value and derivative. In this case, the value and derivative are `vec3`s because the data we are interpolating is the translation of the character. You should implement the `getValue()` and `getDerivative()` functions to evaluate the spline functions at arbitrary times. If you get this correct, you should see the spline drawn on the ground as a red curve.

Finally, change the code in `main.cpp` to make the character walk along the spline curve. For this, you will need to do three things: (i) Translate the character to the current position on the spline. (ii) Rotate the character about the *y*-axis to make it face along the current velocity (i.e. the derivative) instead of along the *z*-axis. (iii) Change the amount the animation advances each frame to be proportional to the current speed. This way, the character's walk cycle proceeds more slowly if its speed is low, and you won't see the character's feet "skate" on the ground. The original speed of the character was *baseSpeed* = `glm::length(Config::baseVelocity)`, so the amount you advance the character should be *dt*currentSpeed*/*baseSpeed*.

Here's a quick summary of the programming requirements:

1. *Hierarchical skeleton transformations:* You need to traverse the hierarchical skeleton model to render the skeleton. You need to composite the various rotations and translations involved in the model to cause it to come together as a skeleton.

2. *Bone rendering:* You need to use cylinders capped with spheres to render each bone in the model.

3. *Spline evaluation:* You need to complete the implementation of the cubic Hermite spline, so that it shows up as a smooth curve.

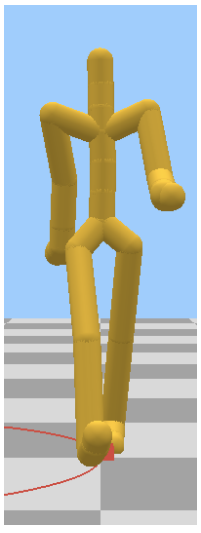
4. *Transformed animation:* You need to modify the animation so that the character walks along the spline facing in the direction of motion, with the animation speed adjusted so that its feet do not skate unrealistically along the floor. (It's OK if the feet partially go through the floor, because that's in the original data.)

Grading for this assignment will be based on the quality of your implementation for each of these four requirements.

Implementation Notes

Drawing a bone in the right direction. Essentially, you have a shape lying along the *z*-axis, and you can obtain a unit vector **b** in the direction of `getBoneVector()`. You must perform a transformation so that the transformed *z*-axis lies along **b**. There are at least two different ways to do this:

a. Directly construct a matrix whose columns are the transformed coordinate axes, as you did in the midterm. Take an arbitrary vector **a** that's not parallel to **b**, and use it to construct two more unit vectors orthogonal to **b** and to each other. Use them as the first three columns of a `mat4 m` (what should the fourth column be?) and apply it as a transformation using `glMultMatrix(&m[0][0])`. *Hint:* you could start by taking **a** = (1,0,0), then if it is too close to **b** or −**b**, change it to (0,1,0).
b. Alternatively, figure out the axis and angle of a rotation that aligns the unit vector **z** = (0,0,1) with **b**. The simplest such rotation is as follows. Imagine the plane containing the two vectors **z** and **b**. If we rotate **z** along this plane by an angle equal to the angle between **z** and **b**, then it will end up coinciding with **b**! The axis of this rotation is simply the normal to the plane, and the angle is something you know how to get.



For reference, the first frame of the walk animation is shown on the right. If you comment out the `character->advance(…)` line in `advanceState()`, your character should look like this.

Splines with $t_0 \neq 0$, $t_1 \neq 1$. In class, we've derived the formula for cubic Hermite interpolation $\mathbf{f}(t)$ with endpoints at 0 and 1. How can we reuse this formula for a cubic Hermite spline, which has a whole sequence of control points at different values t_0, t_1, t_2, …? Here's a simple solution:

1. Find the interval $[t_i, t_{i+1}]$ within which the desired time t lies.

2. Rescale the interval so that t_i maps to 0 and t_{i+1} maps to 1. Find the corresponding value of t, say t_{rescaled}.
3. Now we are working with the standard interval [0, 1]. Apply the cubic Hermite interpolation formula to evaluate $\mathbf{f}(t_{\text{rescaled}})$.

There is one thing to be careful about: When you rescale time from $[t_i, t_{i+1}]$ to [0, 1], the derivatives also get rescaled! (That is, the slope of a function changes if you squash it horizontally.) So you will have to use multiply the derivatives at the endpoints by $(t_{i+1} - t_i)$ when applying the interpolation formula. Conversely, if you are computing the derivative of the spline function itself, you will have to *divide* by $(t_{i+1} - t_i)$ to get the correct value when stretched back to $[t_i, t_{i+1}]$.

You can tell that your spline interpolation is working correctly if the default spline defined in the starter code matches the screenshot on the first page, and the "approximately circular path" defined in the comments in `main.cpp` looks like a circle and not a diamond. To verify that `Spline::getDerivative()` is working correctly as well, which you'll need to get the walking speed right, you could draw a sphere at $\mathbf{f}(t)$ (i.e. `path->getValue(time)`), another sphere at $\mathbf{f}(t+1)$, and an arrow starting from $\mathbf{f}(t)$ along the vector $\mathbf{f}'(t)$ (i.e. `path->getDerivative(time)`). The head of the arrow, $\mathbf{f}(t) + \mathbf{f}'(t)$, should end up somewhat close to $\mathbf{f}(t+1)$.

Above and Beyond

All the assignments in the course will include great opportunities for students to go beyond the requirements of the assignment and do cool extra work. We don't offer any extra credit for this work — if you're going beyond the assignment, then chances are you are already kicking butt in the class. However, we do offer a chance to show off… While grading the assignments the TAs will identify the best 4 or 5 examples of people doing cool stuff with computer graphics. After each assignment, the selected students will get a chance to demonstrate their programs to the class!

There are many ways you can go beyond the requirements in this assignment. You can improve the model rendering, let multiple people walk (or dance) together, or any number of other options. If your ideas for going beyond the requirements would make your code more difficult for the TAs to grade, please help them by submitting a standard version of your assignment first through the normal website link, and then email the TAs the fancier version of your assignment.

Support Code

The webpage where you downloaded this assignment description also has a download link for support code to help you get started. The support code for this assignment is a simple

program using the SDL-based engine, similar to the ones we have used before. You should also download separately the zip file containing the mocap data files.

The support code defines a program structure and everything you need to read and parse the mocap data. **To make locating data files simpler, we have a header file called `config.h` that contains absolute paths to your data files. You should edit this file with the full path (e.g. `"C:\Users\Turing\a4\data"` or `"/home/turing/a4/data"`) where you've placed the data files.** We will modify this file appropriately when grading your assignment.

Handing It In

When you submit your assignment, you should include a `README` file. This file should contain, at a minimum, your name and descriptions of design decisions you made while working on this project. If you attempted any "above and beyond" work, you should note that in this file and explain what you attempted.

When you have all your materials together, zip up the source files and the `README`, and upload the zip file to the assignment hand-in link on our Moodle site. You don't need to include the data files, which are pretty large and we have a copy of them anyway. Any late work should be handed in the same way, and points will be docked as described in our syllabus.

Data Credits

The motion capture data used in this assignment was obtained from the CMU Motion Capture Database. This database, along with additional information, can be found at http://mocap.cs.cmu.edu/.