SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

50.017 Graphics and Visualization

Assignment 3 – Hierarchical Modeling

Handout date: 2020.06.10

Submission deadline: 2020.06.17, 11:59 pm

Late submissions are not accepted

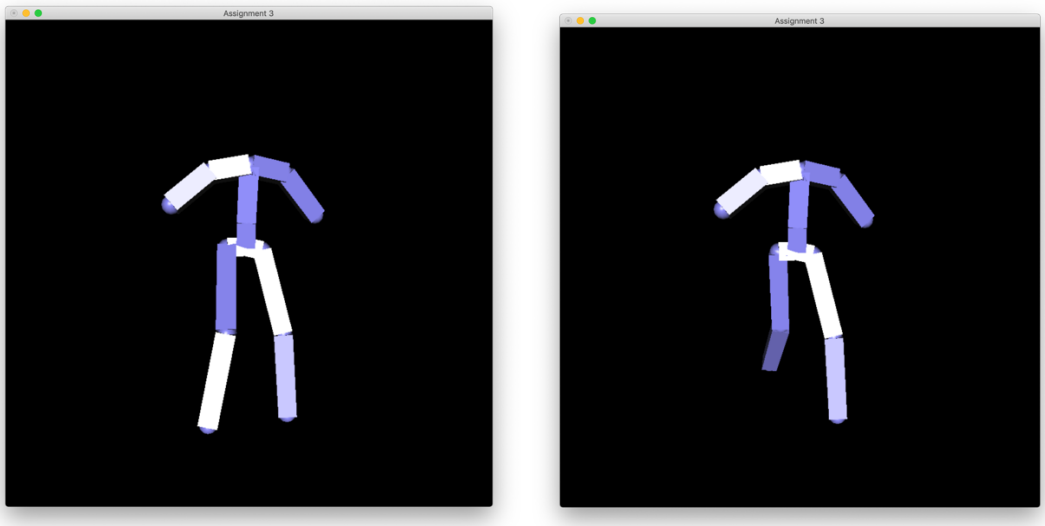

Figure 1. Expected program output by (left) loading Model1.skel and (right) changing the joint angle of its right leg.

In this assignment, you will construct a hierarchical skeleton model whose pose can be controlled by adjusting its joint angles. The method is to define a hierarchy for the skeleton of a human figure and few control parameters (i.e., joint angles of the skeleton). By manipulating these parameters, a user can pose the hierarchical shapes easily. “TODO” comments have been inserted in SkeletalModel.cpp to indicate where you need to add your implementations.

1. Hierarchical Skeleton

An example of a skeleton hierarchy for a human character is shown in Figure 2. Each joint in the hierarchy is associated with a transformation, which defines its local coordinate frame relative to its parent. These transformations will typically have translational and rotational components. Typically, only the rotational components are controlled by articulation variables given by the user (changing the translational component would mean stretching the bone). We can determine the global coordinate frame of a node (that is, a coordinate system relative to the world) by multiplying the local transformations down the tree.

The global coordinate frames of each node can be used to generate a character model by using them to transform geometric models for each joint. For instance, the torso of the character can be drawn in the coordinate frame of the pelvis, and the thighs of the character

can be drawn in the coordinate frame of the hips. Make sure you understand what these global coordinate frames mean; in what space is the input? In what space is the output?

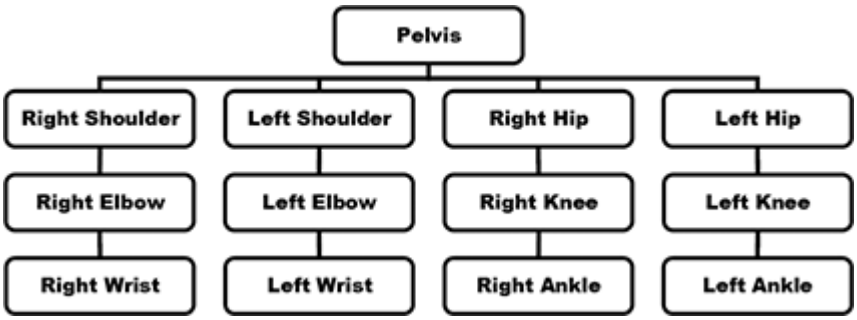

Figure 2. A skeleton hierarchy for a human character.

2. Overview of Starter Code

2.1 User Interface

After running the starter code, it should prompt you to enter filename.skel (e.g., Model1.skel). However, there is no render content in the started code and the window shows all dark.

Once you have filled the rendering code correctly in SkeletalModel.cpp, you should be able to interact with the whole skeleton in the same way as in Assignment 1&2:

- Left mouse drag will rotate the model around a mapped axis based on the mouse motion.
- Left mouse drag + holding shift key will scale the model to make it either smaller or bigger depending on the mouse moving direction.
- You can reset view of the model by pressing the r key.

2.2 Matrix Stack

MatrixStack.h realizes a matrix stack. It keeps track of the current transformation (encoded in a matrix) that is applied to geometry when it is rendered. It is stored in a stack to allow you to keep track of a hierarchy of coordinate frames that are defined relative to one another – e.g., the foot's coordinate frame is defined relative to the leg's coordinate frame.

OpenGL provides a framework for maintaining a matrix stack. However, in this assignment you will use your own, and not using OpenGL's. By building our own matrix stack, we will have a much more flexible data structure independent of the rendering system. For instance, we could maintain multiple hierarchical characters simultaneously and perform collision detection between them.

If no current transformation is applied to the stack, then it should return the identity. Each matrix transformation pushed to the stack should be multiplied by the previous transformation. This puts you in the correct coordinate space with respect to its parent. The implementation for the matrix stack has been provided for you in MatrixStack.cpp.

When rendering, you will be pushing and popping matrices onto and off the stack. After each push or pop, you should call `glLoadMatrixf(m_matrixStack.top())` to tell OpenGL that you want all geometry to be transformed by the current top matrix. Subsequent OpenGL primitives (`glVertex3f`, `glutSolidCube`, etc.) will then be transformed by this matrix. The starter code's `SkeletalModel` class comes equipped with an instance of MatrixStack called

`m_matrixStack`. The starter code also pushes the camera matrix and model matrix as the first few items on the stack.

3. Hierarchical Skeletons

3.1 Load Skeleton File

Your first task is to parse a skeleton that has been built for you. The starter code automatically calls the method `SkeletalModel::loadSkeleton` with the right filename (found in SkeletalModel.cpp). The skeleton file format (. skel) is straightforward. It contains a number of lines of text, each with 4 fields separated by a space. The first three fields are floating point numbers giving the joint's translation relative to its parent joint. The final field is the index of its parent (where a joint's index is the zero-based order that it occurs in the .skel file), hence forming a directed acyclic graph or DAG of joint nodes. The root node contains −1 as its parent and its translation is the global position of the character in the world.

Each line of the .skel file refers to a joint, which you should load as a pointer to a new instance of the Joint class. You can initialize a new joint by calling
```
Joint *joint = new Joint;
```

Because Joint is a pointer, note that we must initialize it with the 'new' keyword to allocate space in memory for this object that will persist after the function ends. (If you try to create a pointer to a local variable, when the local variable goes out of scope the pointer will become invalid, and attempting to access it will cause a crash.) Also note that when dealing with a pointer to an object, you must access the member variables of the object with the arrow operator -> instead of . (e.g., `joint->transform`), which reflects the fact that there is a memory lookup involved.

Your implementation of `loadSkeleton` must create a hierarchy of Joints, where each Joint maintains a list of pointers to Joints that are its children. You must also populate a list of all Joints `m_joints` in the SkeletalModel and set `m_rootJoint` to point to the root `Joint`.

3.2 Draw Skeleton

To ensure that your skeleton was loaded correctly, we will draw simple skeleton figures like the one in Figure 1(left).

Joints. We will first draw a sphere at each joint to see the general shape of the skeleton. The starter code calls `SkeletalModel::drawJoints`. Your task is to create a separate recursive function that you should call from `drawJoints` that traverses the joint hierarchy starting at the root and uses your matrix stack to draw a sphere at each joint. We recommend using `glutSolidSphere( 0.025f, 12, 12 )` to draw a sphere of reasonable size.

You must use your matrix stack to perform the transformations. You will receive no credit if you use the OpenGL matrix stack. To do this, you must push the joint's transform onto the stack, load the transform by calling `glLoadMatrixf`, recursively draw any of its children joints, and then pop it off the stack.

Bones. A skeleton figure without bones is not very interesting. In order to draw bones, we will draw elongated boxes between each pair of joints in the method `SkeletalModel::drawSkeleton`. As with joints, it is up to you to define a separate

recursive function that will traverse the joint hierarchy. At each joint, you should draw a box between the joint and the joint's parent (unless it is the root node).

Unfortunately, OpenGL's box primitive `glutSolidCube` can only draw cubes centred around the origin; therefore, we recommend the following strategy. Start with a cube with side length 1 (simply call `glutSolidCube( 1.0f )`). Translate it in *z* such that the box ranges from [-0.5, -0.5, 0] to [0.5, 0.5, 1]. Scale the box so that it ranges from [-0.025, -0.025, 0] to [0.025, 0.025, *d*], where *d* is the distance to the next joint in your recursion. Finally, you need to rotate the z-axis so that it is aligned with the direction to the parent joint: *z* = *parentOffset.normalized()*. Since the x and y axes are arbitrary, we recommend mapping *y* = *(*$z \times r$*).normalized()*, and *x* = *(*$y \times z$*).normalized()*, with r supplied as [0, 0, 1] .

For the translation, scaling, and rotation of the box primitive, you must push the transforms onto the stack before calling `glutSolidCube`, but you must pop it off before drawing any of its children, as these transformations are not part of the skeleton hierarchy.

3.3 Change Pose of Skeleton

Finally, You should implement `SkeletalModel::setJointTransform` to set rotation component of the joint's transformation matrix appropriately. By implementing this function, you will be able to manipulate the joints based on the passed in Euler angles. The skeleton shown in Figure 1(right) is transformed by adjusting the Euler angles of its right leg only.

You have two ways to specify the joint ID and corresponding Euler angles for changing pose of the skeleton:

1. Specify these parameters in your code (easier way, but you need to recompile your code whenever you want to make a different change);
2. Specify these parameters in the command window. You can write some code to prompt users to input joint ID and corresponding Euler angles (better way, but will involve more work).

4. Grading

Each part of this assignment is weighted as follows:

- Load Skeleton File: 40%
- Draw Skeleton: 40%
- Change Pose of Skeleton: 20%

5. Submission

A .zip compressed file renamed to AssignmentN_Name_I.zip, where N is the number of the current assignment, Name is your first name, and I is the number of your student ID. It should contain only:

- The **source code** project folder (the entire thing).
- A **readme.txt file** containing a description of how you solved each part of the assignment (use the same titles) and whatever problems you encountered. If you know there are bugs in your code, please provide a list of them, and describe what do you

think caused it if possible. This is very important as we can give you partial credit if you help us understand your code better.

- A couple of **screenshots** clearly showing rendered images of skeletons before and after changing the joint angles.

Upload your zipped assignment to e-dimension. Late submissions receive 0 points!