SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

50.017 Graphics and Visualization

Assignment 2 – Hierarchical Skeleton

Handout date: 2024.02.15

Submission deadline: 2024.02.26, 11:59 pm

Late submissions are not accepted

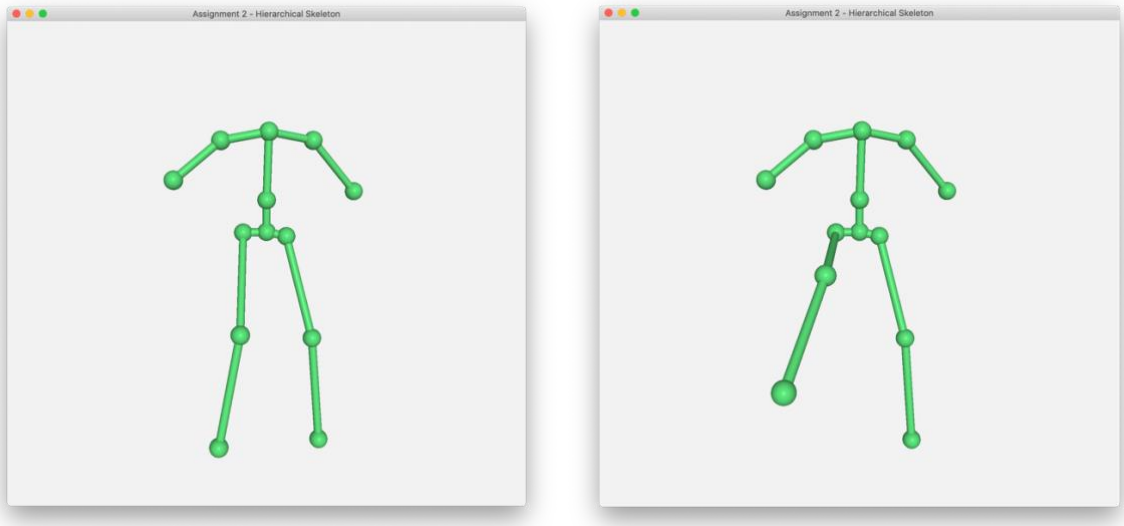

Figure 1. Expected program output by (left) loading Model1.skel and (right) changing the joint angle of its right leg.

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 the following four functions in `SkeletalModel.cpp` to indicate where you need to add your implementations:

```
void SkeletalModel::loadSkeleton( const char* filename )
void SkeletalModel::computeJointTransforms(Joint* joint, MatrixStack matrixStack)
void SkeletalModel::computeBoneTransforms(Joint* joint, MatrixStack matrixStack)
void SkeletalModel::setJointTransform(int jointIndex, float angleX, float angleY, float angleZ)
```

1. Hierarchical Skeleton

An example of a skeleton hierarchy for a human character is shown in Figure 2. Each joint (modeled as a sphere) 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. For instance, in Figure 2, the torso (joint 1) of the character is specified in the local coordinate frame of the head (joint 0), and the thighs (joints 3 and 6) of the character are specified in the local coordinate frame of the hip (joint 2).

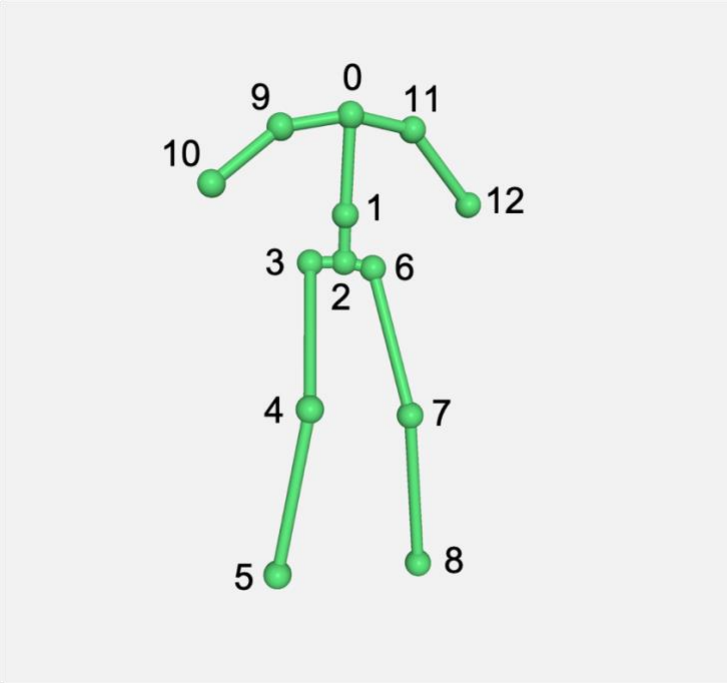

Figure 2. A skeleton hierarchy for a human character with 13 joints.

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 gray.

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

- Left mouse drag will rotate the model around a mapped axis based on the mouse motion.
- Shift + left mouse drag will translate the model in the screen space based on the mouse motion.
- Scrolling the mouse will scale the model to make it either smaller or bigger depending on the mouse scrolling direction.

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.

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. The starter code's `SkeletalModel` class comes equipped with an instance of `MatrixStack` called `m_matrixStack`.

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 as shown in Figure 1(left) and 3. The rendering code framework has been implemented for you in the `main.cpp`, where each joint is drawn as a ball and each bone is drawn as a cylinder. Your task is to compute a transformation matrix for each joint/bone and to store them in the following two vectors respectively:

```
SkeletalModel:: vector<glm::mat4> jointMatList;
```

```
SkeletalModel:: vector<glm::mat4> boneMatList;
```

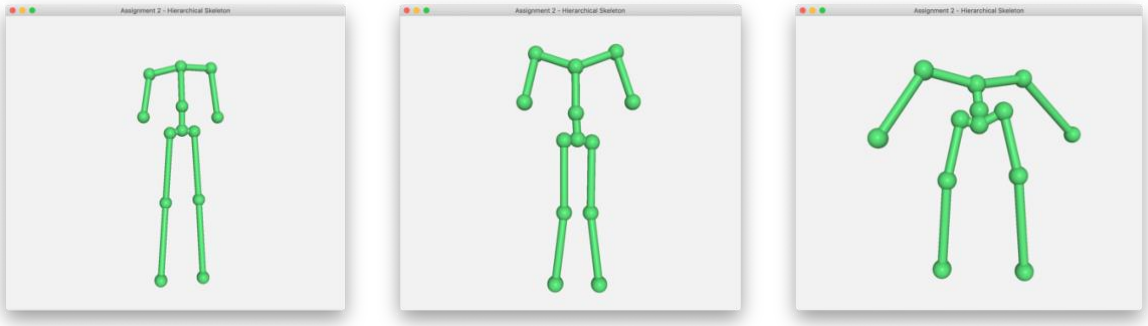

Figure 3. Expected program output by loading (left) Model2.skel, (middle) Model3.skel, and (right) Model4.skel.

Joints. We will first draw a sphere at each joint to see the general shape of the skeleton. The starter code calls `SkeletalModel:: computeJointTransforms()` to compute a transformation matrix for each joint. To achieve this, your task is to implement a separate recursive function

```
void SkeletalModel::computeJointTransforms(Joint* joint, MatrixStack matrixStack)
```

that traverses the joint hierarchy starting at the root and uses your matrix stack to compute the transformation for each joint.

When computing the transformation for each joint, you will be pushing and popping matrices onto and off the matrix stack. After obtaining the transformation matrix, you should call

```
jointMatList.push_back( matrixStack.top() );
```

to save the transformation matrix for the corresponding joint.

Bones. A skeleton figure without bones is not very interesting. We will draw a cylinder between each pair of joints as the bone. More precisely, we draw a cylinder between each joint and the joint's parent (unless it is the root node).

The starter code calls `SkeletalModel:: computeBoneTransforms()` to compute a transformation matrix for each bone. To achieve this, your task is to implement a separate recursive function

```
void SkeletalModel:: computeBoneTransforms(Joint* joint, MatrixStack matrixStack)
```

that traverses the joint hierarchy starting at the root and uses your matrix stack to compute the transformation for each bone.

When computing the transformation for each bone, you will be pushing and popping matrices onto and off the matrix stack. After obtaining the transformation matrix, you should call

```
boneMatList.push_back( matrixStack.top() );
```

to save the transformation matrix for the corresponding bone.

The default cylinder is with radius 1.0 and height 1.0 centred around the origin. Therefore, we recommend the following strategy. Start with the default cylinder. Translate it in *z* such that the cylinder ranges from [-1, -1, 0] to [1, 1, 1]. Scale the cylinder so that it ranges from [-0.01, -0.01, 0] to [0.01, 0.01, *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 cylinder primitive, you must push the transforms onto the stack before calling `boneMatList.push_back( matrixStack.top() )`, but you must pop it off before computing the transformation for any of its children, as these transformations are not part of the skeleton hierarchy.

3.3 Change Pose of Skeleton

To change pose of the skeleton, you need to set rotation component of the joint's transformation matrix appropriately. By implementing the following function

```
void SkeletalModel::setJointTransform(int jointIndex, float angleX, float angleY, float angleZ)
```

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.

We suggest to specify the joint ID and corresponding Euler angles for changing pose of the skeleton in the command window. This starter code has implemented it for you in `main.cpp`:

```
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
```

Pressing key 'c' will prompt users to input joint ID and corresponding Euler angles. Pressing 'enter' key will execute the function to change the skeleton's pose.

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!