Assignment 4: Skeletal Animation and Skinning

CS 4600 Computer Graphics
Fall 2016
Ladislav Kavan

In this assignment we will explore the basics of skeletal animation and linear blend skinning, which are techniques that allow us to display a moving character on the screen. This is an individual assignment, i.e., you have to work independently. All information needed to complete this homework is covered in the lectures and discussed on our Canvas Discussion Boards. You shouldn't have to use any textbooks or online resources, but if you choose to do so, you must reference these resources in in your final submission. It is strictly prohibited to reuse code or fragments of code from textbooks, online resources or other students -- in this course this is considered as academic misconduct (https://www.cs.utah.edu/academic-misconduct/). Do not share your homework solution with anyone -- this is also treated as academic misconduct in this course, even if nobody ends up copying your code.

The framework code is written in C++ with the following dependencies:

- OpenGL 1.0
- GL Utilities (GLU)
- C++ STL
- OpenGL Extension Wrangler Library (GLEW)
- GLFW3

The recommended IDE is Visual Studio 2013 Community Edition, which is available free of charge for educational purposes. The framework code provides precompiled dependencies for Visual Studio 2013. If you choose to use a different platform or IDE version it is your responsibility to build the dependencies and get the project to work.

The assignment should be implemented inside the provided *main.cpp* file using the specified subroutines. No other source code / dependencies / libraries are needed or allowed for this assignment. The provided source code, after being successfully compiled, linked, and executed, should display a rotating capsule (cylinder with spherical caps). All of the skinning data are loaded for you using the function *loadData*. We have prepared two skinned models for you: the Capsule model (which is loaded by default) and the Ogre character which you can load by changing "capsule" to "ogre". The capsule is a smaller model with simple skeleton. It is much faster to load and therefore recommended for first tests and debugging. The Ogre model is larger and therefore takes a while to load in the Debug mode. The Ogre should work reasonably fast in the Release mode.

We have implemented some functionality that should help you with debugging. By default the program is animating the skeleton. By pressing ‘A’, you stop the animation and go to Pose 0 (rest pose). By pressing ‘A’ again, the animation remains stopped but switches to Pose 1 (animated pose). By pressing ‘A’ one more time, the animation starts playing again. The animation interpolates between poses 0 and 1, so when you are finished with this assignment, you should see a moving Capsule and Ogre on your screen. To further facilitate debugging, you can press ‘S’ to display the skeleton. In the framework code, this will not work and only display a point at the origin. This is because the skeleton has not been computed yet -- this will be your first task (see below). Another thing you can do is display the skinning weights by pressing the ‘W’ key. This does work in the framework code immediately (try it!), because the skinning weights are loaded from the input files for you.

1 Skeletal animation (50 points)

Your first task is to complete the function

void *computeJointTransformations(*
const std::vector<Matrix4f>& p_local,
const std::vector<Matrix4f>& p_offset,
const std::vector<int>& p_jointParent,
const unsigned int p_numJoints,
std::vector<Matrix4f>& p_global)

It has a lot of parameters! What do they mean? The integer *p_numJoints* is the number of joints. The array *p_local* is an array of “local joint transformations”, i.e., the transformations that describe the current pose. In the lecture slides this was denoted by “T”. If the character is animating, then *p_local* will be changing every time the function *computeJointTransformations* is called. There are *p_numJoints* transformations in *p_local* and *p_offset*. The matrices stored in *p_offset* are the relative transformations between individual joints. These are not changing even if the character is animating, because they represent the skeleton in the rest pose; in the lecture slides this is denoted by “R” (for “relative”), to indicate that this is the relative transformation between a given joint and its parent. (In case of root joint which has no parent, this is the relative transformation between the root and the origin of the model coordinates, i.e., [0,0,0]). The array *p_jointParent* again has *p_numJoints* elements and *p_jointParent[j]* stores the index of the parent joint of joint *j*. For the root joint (*j = 0*), *p_jointParent[0] = -1*, indicating that the root has no parent. Your task is to compute a vector of *p_numJoints* transformations *p_global*. For every joint *j*, *p_global[j]* needs the concatenated transformations from the root to joint *j*. The formula (also discussed in the lectures) is: *p_global[j] = R(0) T(0) ... R(p_jointParent(j)) T(p_jointParent(j)) R(j) T(j)*. You can assume that the joints are stored in order such that *p_jointParent(j) < j* (check that this is true in the input files!). Note that it is perfectly valid to have bones of length zero, i.e., some joints can be at exactly the same location -- this trick is often used even in professional animation rigs. This should not require any special treatment in the code.

If you have implemented the function *computeJointTransformations* correctly, pressing the 'S' key should display the skeleton with joints at reasonable places. This is a good sanity check, but it does not guarantee you have computed the transformations correctly, because it only displays points and lines.

2 Linear blend skinning (50 points)

Your second task is to use the matrices *p_global* you have computed in *computeJointTransformations* to deform the given model. You will do this be completing the function

void skinning(
const std::vector<Vector3f>& p_vertices,
const unsigned int p_numJoints,
const std::vector<Matrix4f>& p_jointTrans,
const std::vector<Matrix4f>& p_jointTransRestInv,
const std::vector<std::vector<float>>& p_weights,
std::vector<Vector3f>& p_deformedVertices)

Let's go over the input data. The array *p_vertices* is an array of rest pose vertex positions. You may want to convert these to homogeneous coordinates, and for this purpose we have prepared for you functions *toHomog* and *fromHomog* to perform the conversions. The integer *p_numJoints* is the number of joints as before, which matches the sizes of the arrays *p_jointTrans* and *p_jointTransRestInv*. The *p_jointTrans* are exactly the transformations (*p_global*) you have computed in the *computeJointTransformations* function. The array *p_jointTransRestInv* are similar, but these are concatenated joint transformations computed at the rest pose and inverted. We prepare these for you using the function *initRestPose*, which again calls your function *computeJointTransformations*, but with all *p_local* transformations set to identities. The *p_weights* is a 2D array of size numJoints x numVertices. The entry *p_weights[j][v]* contains the skinning weight for vertex *v* and joint *j*. These weights can be visualized by pressing the 'W' key. You may assume that *p_weights[0][v] + ... + p_weights[p_numJoints - 1][v] = 1* for each vertex *v*.

Your job is to compute *p_deformedVertices* (obviously of the same length as *p_vertices*), which will contain the deformed vertices computed by linear blend skinning. These vertices are subsequently used for rendering the model on the screen. If you have done everything correctly, you will see an animated Capsule and Ogre on the screen. Please submit screenshots of the Ogre with skeleton visualization ('S') in Poses 0 and 1.

3 Extra Credit: Dual Quaternion Skinning (up to 20 bonus points at instructor's discretion)

If you are looking for an additional challenge, replace the linear blend skinning algorithm with dual quaternion skinning. You can also try to create more interesting animations -- for this you'll need to understand the pose.dmat file and possibly also create some simple animation interface, that will allow you to preview and edit your animations.

4 Submission

When you're finished with Tasks 1 and 2, you'll need to submit the following:

- Source code (you should only modify the main.cpp file). The main.cpp file is all we need -- please do not submit any other files, especially NOT .exe files and other files created by Visual Studio.
- PDF document describing what you did, screenshots / graphs are recommended. If you used any textbooks or online resources that may have inspired your way of thinking about the assignment, you must reference these resources in this document. This is not recommended because other resources may be using completely different conventions (e.g. transposed matrices) so their formulas may look completely different from ours.
- Screenshots of Ogre with skeleton visualization enabled in Poses 0 and 1.
- [optional] If you have succeeded with Task 3, submit the code "main-extra.cpp" and the images / videos of your DQS implementation

Please pack all of your files to a single ZIP file named Lastname_Firstname_HW4.zip

Please submit this ZIP file via Canvas.