CS-C3100 Computer Graphics, Fall 2021

Lehtinen / Aho, Kaskela, Kynkäänniemi

Assignment 3: Hierarchical Modeling

Due Sun Oct 31st at 23:59.

In this assignment, you will construct a hierarchical character model that can be interactively controlled with a user interface. Hierarchical models may include humanoid characters (such as people, robots, or aliens), animals (such as dogs, cats, or spiders), mechanical devices (watches, tricycles), and so on. In the latter half of the assignment, you will implement *skeletal subspace deformation*, a simple method for attaching a “skin” to a hierarchical skeleton which naturally deforms when we manipulate the skeleton’s joint angles.

Requirements (maximum 10 p) ***on top of which you can do extra credit***

1. **Calculating joint positions (1 p)**
2. **Rotating the joints (2 p)**
3. **Visualizing joint coordinate systems (2 p)**
4. **Skeletal subspace deformation (4 p)**
5. **Skinning for normals (1 p)**

1 Getting Started

By now you are familiar with the development environment, so we’ll cut right to the chase!

In this assignment, we’ll be applying what we know about hierarchical transforms and skinning. The end result is an articulated character that you can pose by keyboard controls, and that gets drawn using an actual skinning! For easy extra credit, you can implement the SSD deformation on the GPU’s vertex shaders.

Play around with the solution executable for a bit to see what you’re supposed to achieve. The number keys toggle between different rendering/visualization modes. The on-screen text on the lower left describes the other controls. There are two files you can try, `Model1` and `mocapguy`. `mocapguy` has a short dancing animation you can see by pressing the button in the GUI.

We’ll be using the terms *joint* and *bone* interchangeably. Both are really just mental helpers; what matters for skinning is that we have a set of hierarchically linked transformations (i.e., local coordinate systems) that our vertices are attached to using weights.

2 Detailed instructions

Complete R1 first. You can work with the rest of the requirements in any order. Before you delve in, be sure to take a moment to look through `skeleton.hpp`. It contains the class definitions for the joints and the skeleton, which you'll be working with. You'll see that the important stuff happens in classes `Skeleton` and `Joint`. To the outside caller, the joints are accessed through the accessor functions found in the skeleton class, and are identified using indices.

R1 Calculating joint positions (1 p)

Our first priority is to visualize the skeleton somehow so we can see how our manipulations affect it. Let's kick things off by visualizing the positions of the joints.

The code that loads the skeleton already sets up the `to_parent` transform matrix for every joint. `to_parent` transforms points from the joint's space to the parent joint's space (and in the root joint's case, to world space). Your job is to use these for building transform matrices which take us directly from a given joint space to world space.

Fill in `Skeleton::updateToWorldTransforms()` which recursively walks through the joints and writes the current joint-to-world transformation in each joint. (In this case recursion is simplest, but if you wanted to, you could use a stack data structure instead.) This pretty directly implements the hierarchical traversal we saw on lecture 5.

Code that draws a set of positions as white dots (and the selected joint as red) is already in place in `App::renderSkeleton()`, but you have to add one line there to obtain the world position of the joint once you've computed the joint-to-world transforms. Obtaining the position from the matrix is easy if you've built a good understanding of how transformation matrices work. The joint's position in world space is just the origin of its own coordinate system. (If you're having trouble, think about what happens when you feed the origin (0,0,0,1) through the joint-to-world transformation matrix.)

When this is done, you will see the shape of your skeleton (Figure 1).

R2 Rotating the joints (2 p)

At the beginning, we assume no rotation has been applied to any of the joints. In other words: in the bind pose, our character's bones are translated, but not rotated. `Skeleton` has two methods for modifying its joint positions, `setJointRotation()` and `incrJointRotation()`, which the UI uses to move the skeleton. Your job is to implement `setJointRotation()`. When you are done, you'll be able to move the skeleton using the UI.

We represent the joints' local rotations using Euler angles (yikes!). To rotate a joint with respect to its parent, we just have to alter its `to_parent` matrix. This happens in `setJointRotation()`, which takes in Euler angles and where you have to update the

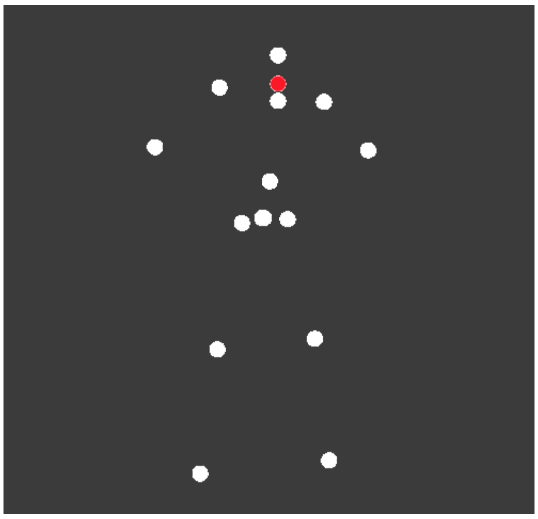
Figure 1: The skeleton with joint positions visualized (R1).

`to_parent` transform matrix to match the desired rotation. `updateToWorldTransforms()` propagates any changes made to `to_world` transforms, and `Skeleton` already calls it before using the transforms.

The dimensions of the rotation angles vector correspond to the rotation angles (so, for example, `euler_angles.x` is the rotation around the x axis), and the order of rotations used in the example is `rotZ*rotY*rotX` – note that this is a completely arbitrary choice.

When you're done, you will be able to see how the joints move around when you change their rotations using the keyboard.

R3 Visualizing joint coordinate systems and parents (2 p)

Seeing the joint locations as points is better than nothing, but it's hard to tell what stance the skeleton is in, and some rotations — those of the bones that have no children in the hierarchy — are invisible.

Fill out the rest of `App::renderSkeleton()`. You have to draw some colored lines to show the local coordinate systems at each joint. This will let you verify that your R2 is really working correctly. Also draw lines for "bones" between joints to make the skeleton look the part. Inspect the header `skeleton.hpp` for the data you need, e.g., parent links.

Contrary to what we said in the last assignment, we will again use old (immediate mode) OpenGL, but only for this specific debug visualization task. Modern OpenGL involves always loading data to GPU buffers and setting up shader programs, and that is too much code to write for simple debug graphics like these. Also note that we are drawing a very small amount of things, so performance is not an issue. You need to use two OpenGL functions: `glVertex3f()`, which specifies a vertex position (naturally, two positions are needed for a line) and `glColor3f()` which sets the vertex color for the following `glVertex` calls. This will be clear when you look at the code.

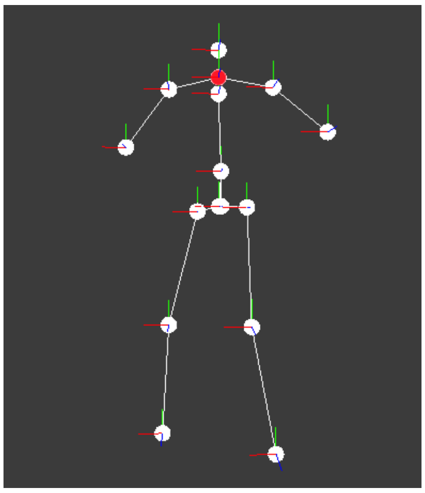

Figure 2: The skeleton, now complete with local coordinate system and parent link visualizations (R3).

See the documentation for `glBegin/glEnd` for details.

R4 Skeletal subspace deformation (4 p)

Now that we have our skeleton, it's time to get it moving with the skin!

Our work starts in the `Skeleton` class. First, fill in `computeToBindTransforms()`. This function is only called once, after the skeleton is loaded and is still in its bind pose, to store the world-to-bind transform B_i^{-1} for each joint. Then, continue to fill in `getSSDTransforms()` which produces transforms between the bind pose and current pose, exactly as we saw in class. You need to use the joint-to-world transforms you've already computed and stored earlier. This function is called each frame by the rendering code.

The last piece of the puzzle is `App::computeSSD()`. It uses the transforms from `Skeleton` and the mesh we previously loaded where each vertex has joint attachment information, and produces a mesh of plain vertices that have been deformed to conform to the current position of the joints. You job is, for each vertex of the skin, to loop over all the weights in every skin vertex (see `WeightedVertex` in `App.hpp`), transform the vertex using the relative SSD transformation, and finally blend the results together using the weights, again, precisely as shown on the lecture slides.

All the rendering code that handles the vertices after `computeSSD()` is already in place.

Be aware that SSD is a rather heavy operation for the CPU, and our code is going to be particularly slow with a Debug build. Using a fast desktop computer, we saw about 40 frames per second in Release, and only about one frame per second in Debug.

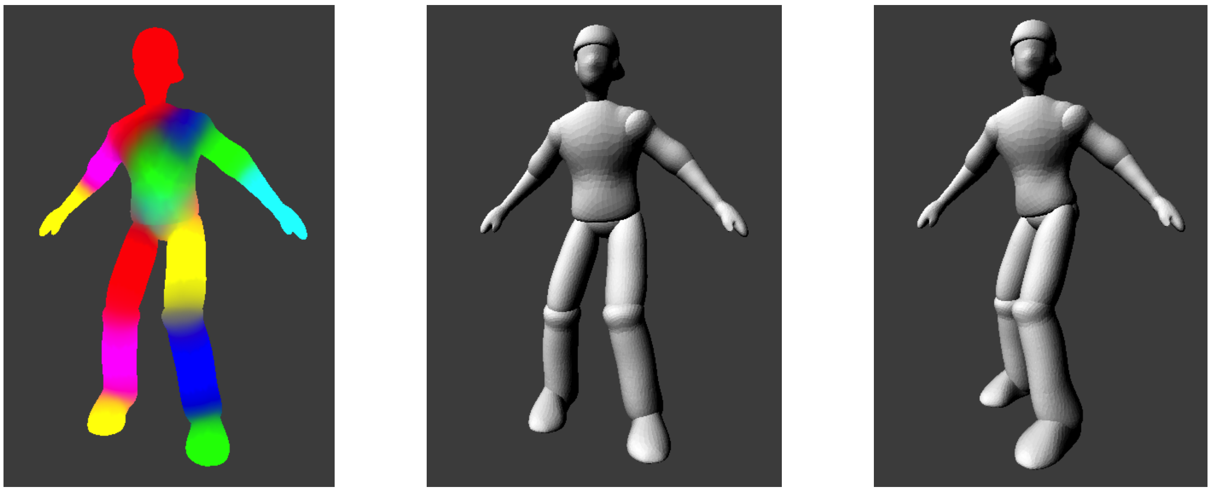

Figure 3: Left: Bone weights visualized on the vertices in bind pose. Middle: bind pose mesh lit using simple directional lighting. Right: Deformed skin, waist rotated to y=0.70.

R5 Normals for skinning (1p)

Extend your SSD code from above to skin the normals as well. You can cut corners and treat the normals as usual vectors and not care about the inverse transpose from the lecture slides. That's what the solution executable does anyway.

The only thing you need to think about here is how transforming vectors ("directions") differs from transforming points. The lecture slides on transforms contain hints. (It's really not hard.) Note that FW has a slight caveat here; multiplying a 4 by 4 matrix with a 3-dimensional vector (which is, in general, undefined) is handled as implicitly converting the vector to 4 dimensions – treating it as a position. You want to do this conversion explicitly instead.

3 Extra credit

As always, you are free to do other extra credit work than what's listed here — just make sure to describe what you did in your README. We'll be fair with points, but if you want to attempt something grand, better talk to us beforehand.

3.1 Recommended

- **Easy: SSD on the GPU (2p).**
  SSD on the CPU is much too slow for the real world; we should do it on the GPU instead. Depending on the computer hardware, this might make your program run 5x faster or more. The starter code contains a stub shader you can extend into performing skinning on the GPU. You only need to write a bit of GLSL code that does exactly what you already did with C++ in R4 and R5. This is a good chance to get familiar with GLSL if you haven't already!

- **Medium: Add a "wrist" joint to both hands (5p)**
  Our skeleton has no way of moving the hands separate from the elbows. Examine the skeleton definition file and add new joints as children of the elbow joints. This is pretty easy; the hard part is changing the weights of the vertices in the skin mesh's hands to tie to the new joint you've just created. We recommend you come up with a proximity-based hack. In any case, report what you did in your README!

- **Medium/hard: IK (8p)**
  Add a method for picking bones in the skeleton visualization by the mouse, and moving the joint directly in the current camera XY plane when some mouse button (right, say) is pressed, and in the current camera Z direction when another button is pressed.

  You will need to figure out how all the joint rotations need to change in order for the bone to go where the user points. The hard part is figuring out the derivatives of the bone position with respect to all the rotation parameters of all the bones in the hierarchy above, i.e., computing the Jacobian. Using the pseudo-inverse to steer the angles to the right direction is not that difficult afterwards[1].

  Instead of writing the math out in symbolics and using e.g. Mathematica to get you the required derivatives, you can Google for "automatic differentiation"; it's a numerical method for computing derivatives in code. You'd then need to extend the matrix and vector classes to use "dual numbers" instead of just floats. The Wikipedia article is a good source on the general idea; pay special attention to how one computes Jacobians.

  You can combine this with the animation extra below: add a capability for animating the IK target for some more points!

[1] We recommend you use the easy-to-use Eigen C++ template library for the necessary matrix computations; in particular, it's easy to build a pseudoinverse using the Singular Value Decomposition (SVD).

3.2 Easy

- **Animation (3 pts)**
  Add methods for taking snapshots of your character's pose, and a method for interpolating between them. This will allow you to create simple animations! Add functionality to start and stop the playback of the animation by pressing a key.

- **Dual quaternion skinning (4 p)**
  Implement Dual Quaternion Skinning of Kavan et al. in order to alleviate the squashing issues you see with rotating joints.

3.3 Medium

- **Other skeletons and skins (5 pts)**
  Scour the web for skeletons and skinned meshes, and write code for converting them into the format read by your software. Note that you will probably need to extend the file format to support non-identity rotations for the bind pose. Tell us where you got the data and what you had to do to perform the conversion.

3.4 Hard

- **Use pose capturing to control your character (10p)**
  OpenPose is a library that lets you fit skeletons to videos in real time. You can find it here: GitHub. Using some input, extract a skeleton and *retarget* it into your own skeleton to pose it using a human actor. Another alternative is Microsoft's Kinect, which can be found here: Kinect SDK. Use either library to drive your character based on your own movement.

  Getting the code up and running so that you can get to the skeleton is not overly hard; the difficulty is in the next step, where you have to take their skeleton and its pose, and somehow map it to the pose of your own skeleton. You can appreciate the difficulty: nothing says they have the same configuration (or even number) of bones, or that the hierarchy would be the same. You will need to use inverse kinematics -type of methods to perform the retargeting.

  When submitting, you can add a video showing how well your system performs. We will grant you points based on how robustly your system operates, with a maximum of 10p. (If you attempt this, we recommend you also add the capability to record an entire animation and play it back on your own skeleton and mesh — this will earn you an extra 3 points, as per the animation extra above.)

- **Style-based inverse kinematics (10p)** Implement style-based inverse kinematics. This IK technique is based on machine learning. It works by first training on a set of input poses to learn a model of what a valid pose looks like, and then uses this training set to guide an IK solver to better, more natural poses. (If you don't do this in your IK solver, you will notice that the poses produced are not necessarily all that natural. See the video on the linked page for details; we also saw this in class.)

In order to generate the training set, you will most likely need pose capturing as you need many input poses[2].

[2]This is a lot of work for a 2-week deadline; if you end up going this route, you can forgo the extra credit and use this for a B.Sc. thesis subject or similar. If you want to attempt it, talk to Jaakko.

4 Submission

- Make sure your code compiles and runs both in Release and Debug modes on Visual Studio 2019, preferably the VDI environment. Comment out any functionality that is so buggy it would prevent us seeing the good parts. Check that your `README.txt` (which you hopefully have been updating throughout your work) accurately describes the final state of your code.
- Fill in whatever else is missing, including any feedback you want to share. We were not kidding when we said we prefer brutally honest feedback.
- Package all your code, project and solution files required to build your submission, `README.txt` and any screenshots, logs or other files you want to share into a ZIP archive. Note: the solution file itself does not contain any code, make sure the contents of `src/` are present in the final submission.
- Sanity check: look inside your ZIP archive. Are the files there? (Better yet, unpack the archive into another folder, and see if you can still open the solution, compile the code and run.)

Submit your archive in MyCourses folder:
"Assignment 3: Hierarchical Modeling".