This handout was compiled on 2024-10-11 18:19.

CS-C3100 Computer Graphics, Fall 2024

Lehtinen / Härkönen, Kemppinen, Kynkäänniemi, Kozlukov, Timonen

Assignment 3: Hierarchical Modeling and Animation

Due October 27, 2024 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, animals, mechanical devices, and so on. In the second half of the assignment, you will implement *skeletal subspace deformation* or *skinning*, a simple method for attaching a skin to a hierarchical skeleton so that it deforms when we animate the skeleton's joint angles.

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

1. **Hierarchical traversal and joint-to-world transformations (2 p)**
2. **Rotating the joints (2 p)**
3. **Visualizing joint coordinate systems (1 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! Follow the same process as last time in Assignment 2 to pull and merge changes from our upstream git repository `cs-c3100-2024`. This will leave you with a shiny new `assignment3` subfolder. Alternatively, you can download the `.zip` package from MyCourses.

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 modes. The on-screen text describes the other controls.

For the first part on hierarchical transformations, we recommend `characters/lafan1/dance1_subject1.bvh`, which is the model that is loaded by default. It features a long dance animation, but does not come with a skin.

For the second skinning part, we provide `characters/mocapguy.bvh` that comes with a skin, vertex weights, and also a short dancing animation you can see by pressing SPACE or the `Animate` checkbox in the GUI.

2 Detailed Instructions

We'll be using the terms *joint* and *bone* interchangeably. Both are really just mental helpers: what matters is that we have a set of hierarchically linked transformations — i.e., local coordinate systems — where joints/bones are always defined relative to their parent's coordinate system, which are, again, defined relative to its parent, and so on, until the root node.

Before you delve in, be sure to take a moment to look through `skeleton.h`. 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 `class Skeleton` and `struct Joint`. To the outside caller, the joints are accessed through the accessor functions found in the skeleton class, and are identified using indices.

2.1 Construction of the coordinate systems

We link the coordinate systems together as specified by the BioVision Hierarchy (BVH) motion capture data format (link). Each file contains a number of joints arranged hierarchically. The first joint, at index 0, is the root node. Each joint in the hierarchy has a parent joint and zero of more child joints. We denote the index of the ith joint's parent by $\texttt{parent}(i)$, and define $\texttt{parent}(0) = -1$. The jth child node of the ith node is denoted by $\texttt{child}(i, j)$.

One key entity you'll be constructing is the *joint to parent transformation matrix* $\mathbf{P}_i \in \mathbb{R}^{4x4}$ for every joint. In the code, these matrices are called `to_parent`, and they are computed on the fly What each $\mathbf{P}_i$ does is, simply, to take points in the local coordinate system of the ith joint and map them to the local coordinate system of its parent joint — exactly as shown on the lecture slides *5.1 Introduction to Hierarchical Modeling*. All coordinate transformations specified by the BVH format are *rigid-body transformations*, meaning that they feature only translation and rotation, but no scaling, shearing, or mirroring.

Let's look into how these matrices are constructed.

First, the origin of the local coordinate system of the ith joint is, in the coordinate system of its parent, at position $\mathbf{t}_i \in \mathbb{R}^{3x1}$. If you inspect the contents of a `.bvh` file, these three coordinates are given in the `OFFSET` parts of the joint definitions. What does this tell us about the matrix $\mathbf{P}_i$? An (affine) matrix that maps the local origin $(0, 0, 0)^T$ to point $\mathbf{t}_i$ *must* have the form

$$\mathbf{P}_i = \left(\begin{array}{ccc|c} & & & \vdots \\ & ? & & \mathbf{t}_i \\ & & & \vdots \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{1}$$

This is easy to see by multiplying it with the vector $(0, 0, 0, 1)^T$ that represents the origin $(0, 0, 0)^T$ in homogeneous coordinates.

Second, the orientation (rotation) of the ith joint's coordinate system, again relative to its parent, is specified by three *Euler angles* $\{\alpha_1, \alpha_2, \alpha_3\}$ (link)[1]. Euler angles can be used to define an arbitrary rotation $\mathbf{R}$ of 3D space as a sequence of three successive rotations around fixed axes $\{\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3\}$ through

$$\mathbf{R} = \mathbf{R}_{\text{AA}}(\alpha_1, \mathbf{a}_1)\, \mathbf{R}_{\text{AA}}(\alpha_2, \mathbf{a}_2)\, \mathbf{R}_{\text{AA}}(\alpha_3, \mathbf{a}_3), \tag{2}$$

where $\mathbf{R}_{\text{AA}}(\alpha, \mathbf{a})$ is a matrix that rotates α radians around axis $\mathbf{a} \in \mathbb{R}^{3x1}$. (The AA stands for axis-angle; see Eigen's documentation). Euler angles, specifically, use the main coordinate axes $(1, 0, 0)$, $(0, 1, 0)$, and

[1]To be perfectly accurate, the angles stored in BVH files are typically so-called Tait-Bryan angles, but the difference does not matter in practice. We'll be calling the Euler angles regardless for simplicity.

$(0,0,1)$ as the axes $\mathbf{a}_i$. However, the order in which the matrices are multiplied together is important, as matrix multiplication is noncommutative! We'll come to exactly how to determine the order later in R2. All together, the joint-to-parent matrices have the form

$$\mathbf{P}_i = \left(\begin{array}{ccc|c} & & & \vdots \\ & \mathbf{R} & & \mathbf{t}_i \\ & & & \vdots \\ \hline 0 & 0 & 0 & 1 \end{array}\right), \tag{3}$$

where the rotation matrix $\mathbf{R} \in \mathbb{R}^{3x3}$ depends on the Euler angles.

R1 Computing the joint-to-world transformations (2 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.

Your first task is to implement the hierarchical traversal that uses these matrices to set up the *joint-to-world* transformation matrices $\mathbf{T}_i$ that take us directly from a given joint's local coordinates to world space. The starter code already sets up the translation[2] component of the joint-to-parent matrices $\mathbf{P}_i$ in function `Skeleton::computeJointToParent`.

There are two versions of the function `Skeleton::updateToWorldTransforms()`. The first that takes no inputs you needn't touch: it initializes the recursion with an object-to-world space transformation matrix that is constructed elsewhere to make the scale of the scene reasonable.

Fill in the the second version that takes in a joint index and the parent-to-world matrix. It should compute the joint-to-world transformation for the current joint using the joint-to-parent matrix and the incoming parent-to-world matrix, store it to the data member `Joint::joint_to_world` of the current joint, and then recursively descend to the child joints. Their indices can be found in the data member `Joint::children`. 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 a larger red one) 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. See the warmup on constructing the matrices above if you feel stuck.

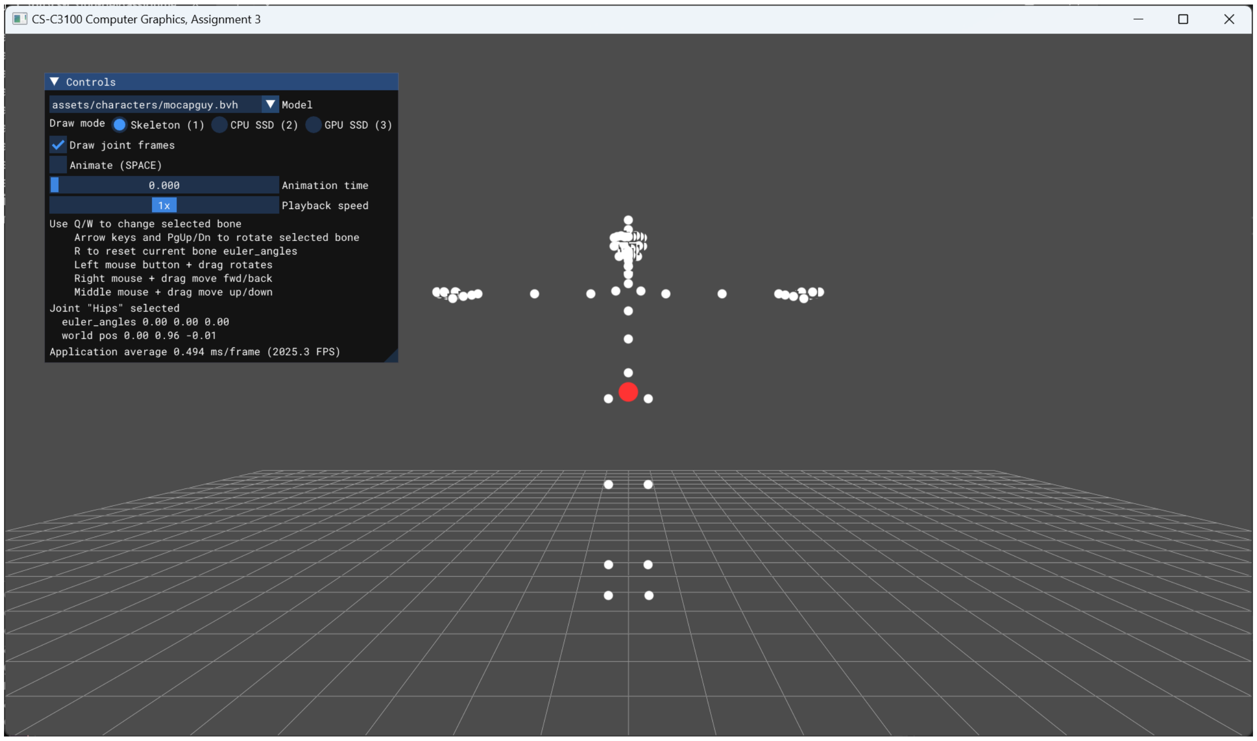
When this is done, you will see the shape of your skeleton (Figure 1). Though not relevant for this task, it will also make the skinning rendering modes display the bind-pose mesh.

R2 Rotating the joints (2 p)

The code you have so far *only* works if the character's default pose has been constructed so that all-zero Euler angles produce a reasonable pose. Why? When all Euler angles are zero, there is no rotation, and the $\mathbf{R}$ block in Equation 3 is simply the identity matrix. The `mocapguy` character that is loaded first upon startup has been constructed this way, but not the second one that features a dance animation. See what happens if you switch to it and compare to the reference.

Your second task is to update the function `Skeleton::computeJointToParent` that constructs the full $\mathbf{P}_i$ matrix for a joint using its Euler angles, including the rotation component. Basically, this means implementing Equation 2.

[2]But not rotation, that comes in R2.


CS-C3100 Computer Graphics, Assignment 3

```
▼ Controls
assets/characters/mocapguy.bvh ▼ Model
Draw mode ● Skeleton (1) ○ CPU SSD (2) ○ GPU SSD (3)
[✓] Draw joint frames
[ ] Animate (SPACE)
0.000                          Animation time
1x                             Playback speed
Use Q/W to change selected bone
    Arrow keys and PgUp/Dn to rotate selected bone
    R to reset current bone euler_angles
    Left mouse button + drag rotates
    Right mouse + drag move fwd/back
    Middle mouse + drag move up/down
Joint "Hips" selected
  euler_angles 0.00 0.00 0.00
  world pos 0.00 0.96 -0.01
Application average 0.494 ms/frame (2025.3 FPS)
```

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

The joint's Euler angles $\alpha_1, \alpha_2, \alpha_3$ are stored in the `Vector3f` data member `Joint::euler_angles`. The axes around which these rotations are to be performed is given in the integer vector `Joint::euler_order`. As detailed in the code comments, the integers 0, 1, and 2 correspond to the x, y, and z axes, respectively. For example, an `euler_order` of, say $(1, 0, 2)$, says that the axes in Equation 2 are $\mathbf{a}_1 = (0, 1, 0)^T$, $\mathbf{a}_2 = (1, 0, 0)^T$, and $\mathbf{a}_3 = (0, 0, 1)^T$. Note that the order can, and sometimes does, vary between joints even within a single BVH file, so you mustn't make any assumptions about it and just use the order specified by each joint. (You can, however, safely assume that the order vector only contains 0s, 1s, and 2s.)

To complete this requirement, you need a function that computes a rotation matrix given an angle and an axis, but you're free to use what Eigen offers as linked to above.

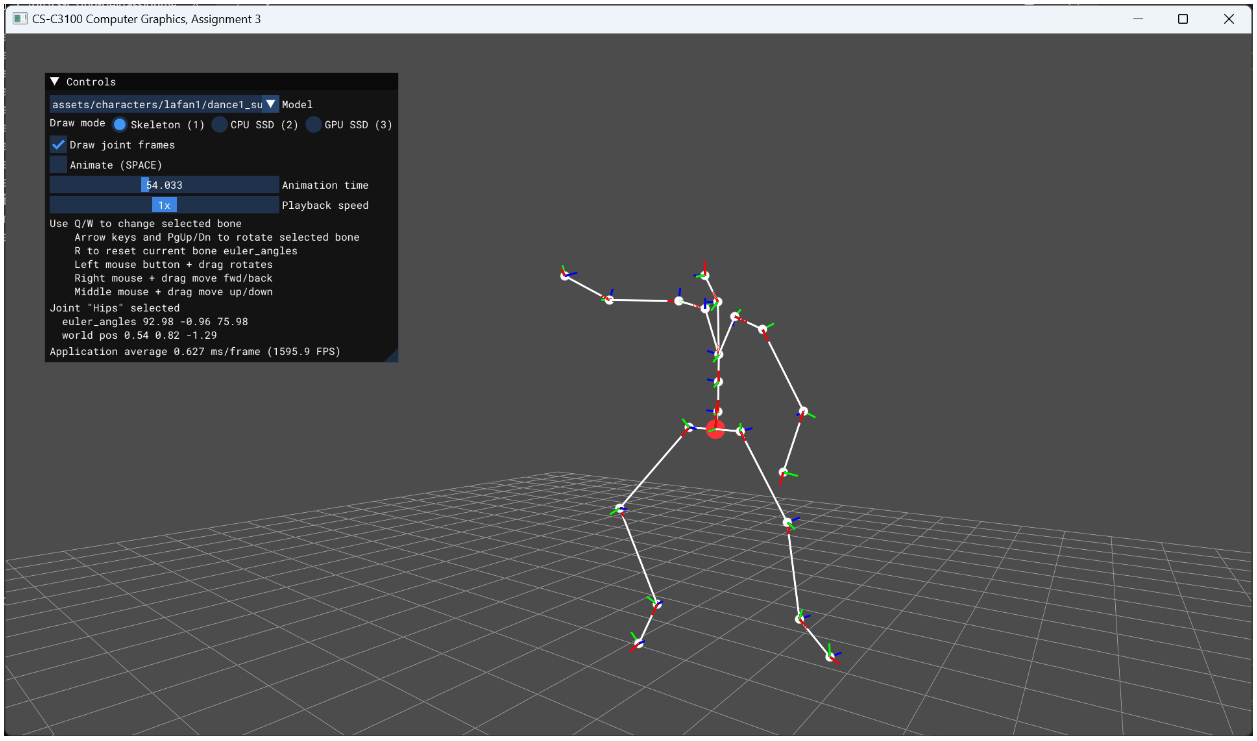
When you're done, and assuming your code from R1 does what it's supposed to, you will be able to see how the joints move around when you change their rotations using the keyboard — and, importantly, starting the animation does something sensible now!

R3 Visualizing joint coordinate systems and parents (1 p)

Seeing the joint locations as points is better than nothing, but it's hard to tell what stance the skeleton is in.

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.h` for the data you need, e.g., parent links.

We use the Im3d immediate mode drawing library for drawing the lines. Its usage is simple: drawing a set of



```
▼ Controls
assets/characters/lafan1/dance1_su ▼ Model
Draw mode  ● Skeleton (1)  ○ CPU SSD (2)  ○ GPU SSD (3)
[✓] Draw joint frames
[ ] Animate (SPACE)
54.033      Animation time
1x          Playback speed
Use Q/W to change selected bone
    Arrow keys and PgUp/Dn to rotate selected bone
    R to reset current bone euler_angles
    Left mouse button + drag rotates
    Right mouse + drag move fwd/back
    Middle mouse + drag move up/down
Joint "Hips" selected
  euler_angles 92.98 -0.96 75.98
  world pos 0.54 0.82 -1.29
Application average 0.627 ms/frame (1595.9 FPS)
```

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

points happens with `Im3d::Vertex(...)` calls sandwiched between `Im3d::BeginPoints()` and `Im3d::End()`. Line segments are drawn similarly between `Im3d::BeginLines()` and `Im3d::End()`, but using two vertex calls per line. The size of points and width of lines can be controlled with `Im3d::SetSize`, and the color with `Im3d::SetColor`.

R4 Skeletal subspace deformation (4 p)

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

Your work starts in the `Skeleton` class.

(1p) First, fill in `Skeleton::computeToBindTransforms()`. This function is only called once, after the skeleton is loaded and in its bind pose, to compute and store the bind-to-joint transforms $\mathbf{B}_i^{-1}$ that transform object-space points on the undeformed skin mesh to the local coordinate frames of the joints. The $\mathbf{B}_i$ are just the joint-to-object transformations, code for which you wrote already in R1[3].

(1p) Then, continue to fill in `Skeleton::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 $\mathbf{T}_i$ you've already computed and stored earlier. This function is called each frame by the rendering code.

[3] You'll note that there is some subtlety going on here: we really mean joint-to-object instead of joint-to-world. The bind pose mappings are applied to undeformed vertex positions that are given in object space. As we apply a further object-to-world transformation in the computation of the joint-to-world mappings, we cannot use the inverses of the joint-to-world transformations for computing the bind pose transformations; this will lead to nonsesical results. To get around this, we call `computeToBindTransforms()` with the object-to-world mapping still set to the identity matrix (cf. the assert on the first line). This ensures that the joint-to-world mappings are, in fact, joint-to-object mappings.

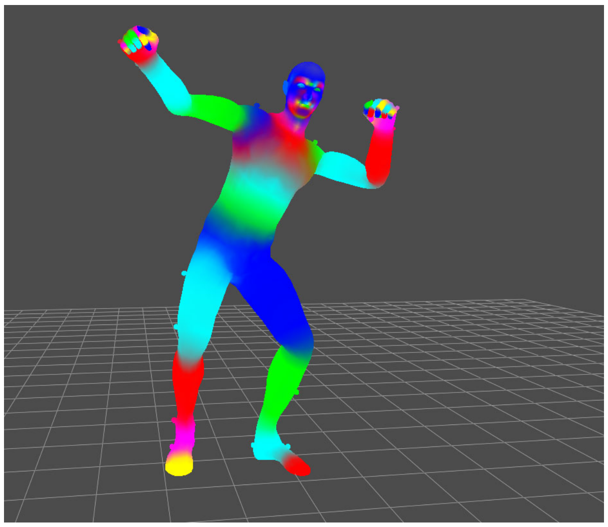
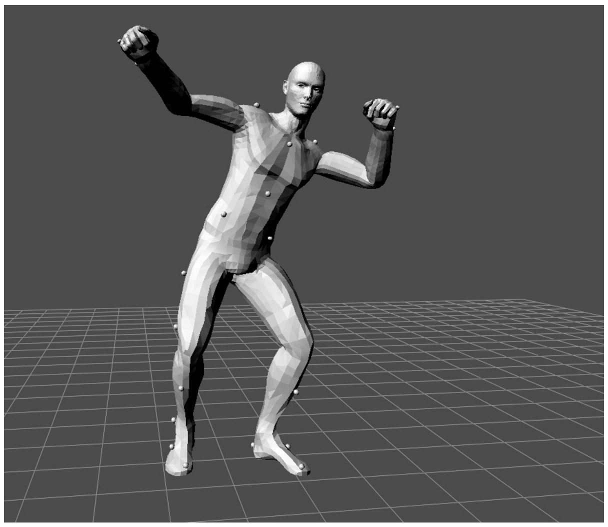



Figure 3: Left: Bone weights visualized on the vertices. Right: mesh lit using simple directional lighting.

(2p) The last piece of the puzzle is `App::computeSSD()`. It uses the transforms from `Skeleton` and a mesh that comes with vertex weights, 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.h`), transform the vertex using the SSD transformations, and finally blend the results together using the weights, again, precisely as shown on the lectures. (Note that the starter code applies the object-to-world mapping to the vertex coordinates explicitly. As the joint-to-world transformations you will compute in R1 and R2 will have this transformation built in, there is no need to apply it during skinning.)

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 your code is going to be particularly slow with a Debug build!

R5 Normals for skinning (1p, +1p extra)

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, but for an extra point, compute it using the inverse transpose instead. That is what `reference.exe` does, anyway. Remember to normalize the normal vectors!

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.

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 dozens or hundreds of times faster. 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! Note that you will have to touch the corresponding bit in `App::render()` as well.

- **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. We recommend you use Eigen for the necessary matrix computations; in particular, it's easy to build a pseudoinverse using the Singular Value Decomposition (SVD).

  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!

3.2 Easy

- **Animation (3 pts)**
  Add methods for taking snapshots of your character's pose so that you can use them as keyframes and create simple animations. You can piggyback on the animation functionality in `skeleton.cpp`, but note that for poses far away from each other, you will likely need to implement a higher-order interpolation. The starter code already implements piecewise linear interpolation for positions and rotations in order to support animation playback speeds less than 1x.

- **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, write code for converting them into the format read by your software, and tell us what to look for when we grade!

3.4 Hard

- **Use pose capturing to control your character (10p)**
  Detectron2 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.

  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[4].

- **Gamepad control through Motion Matching (10+p)** Implement gamepad control for your articulated character through Motion Matching or Learned Motion Matching (link). The second animation clip, `subject1_dance1.bvh`, is extracted from Ubisoft's LAFAN1 motion capture dataset (link), so if you've completed the requirements, you're well on your way of being able to use the database as a basis!

[4]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 Windows with Visual Studio 2022, preferably in the VDI VMs. You may develop on your own platform, but the TAs will grade the solutions on Windows.
- Comment out any functionality that is so buggy it would prevent us seeing the good parts. Use the `reference.exe` to your advantage: does your solution look the same? Remember that crashing code in Debug mode is a sign of problems.
- 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 CMake+vcpkg configuration files, required to build your submission, `README.txt` and any screenshots, logs or other files you want to share into a ZIP archive. *Do not* include your `build` subdirectory or the reference executable. Beware of large hidden `.git`-folders if you use version control - those should not be included.
- Sanity check: look inside your ZIP archive. Are the files there? Test and see that it actually works: unpack the archive into another folder, run the CMake configuration, and see that you can still compile and run the code.
- If you experience a system failure that leaves you unable to submit your work to MyCourses on time, prepare the submission package as usual, then compute its MD5 hash, and email the hash to us at `cs-c3100@aalto.fi` before the deadline. Then make the package that matches the MD5 hash available to us e.g. using `filesender.funet.fi`. *This is only to be done when truly necessary; it's extra work for the TAs.*

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