6.837: Computer Graphics Fall 2010

Programming Assignment 2: Hierarchical Modeling and SSD

Due October 13th at 8:00pm.

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. 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.

This document is organized into the following sections:

1 Getting Started

Download the starter code from Stellar, build the executable with `make`, and run the resulting executable on the Cheburashka test model: (`./a2 data/cheb`). Two windows will pop up. One will be empty, and will eventually contain a rendering of your character. The other contains a list of *articulation variables* or simply *joints*. By clicking on joint names, a slider will appear that lets you manipulate that variable. By shift-clicking and control-clicking multiple names, you can pop up multiple sliders. Once you correctly implement a Matrix Stack, you will be able to change the camera view using mouse control just like in previous assignments: the left button rotates, the middle button moves, and the right button zooms. The sample solution `a2soln` implements SSD control for the right foot—you can drag the slider to rotate Cheburashka's foot. You can press `a`, `j`, `s`, and `m` to toggle drawing of the coordinate axes, joints, skeleton, and mesh, respectively

2 Summary of Requirements

2.1 Hierarchical Model (35% of grade)

For part one of this assignment, you are required to correctly load, display, and manipulate a hierarchical skeleton. Your implementation must be able to correctly parse any of the provided skeleton files (`*.skel`), construct a scene graph data structure, and use a matrix stack in conjunction with OpenGL primitives to render the skeleton. Finally, you will connect the user interface to the skeleton to manipulate its joints.

2.2 Skeletal Subspace Deformation (55% of grade)

For the second part of this assignment, you will implement skeletal subspace deformation to attach a "skin" to your skeleton. SSD will allow you to pose true characters, not just skeletons. This part first requires you to adapt your assignment 0 code to parse a mesh without normals and generate them at display time. You will also need to write yet another parser to load *attachment weights*, which specify, for each vertex, the importance of each bone. Finally, you will implement the actual SSD algorithm which requires applying a number of operations to your transformation hierarchy.

2.3 User Interface (5% of grade)

The starter code provides the familiar camera interface as well as a few example sliders which currently do nothing. To help you better understand the matrix stack, the drawing window will display nothing. You will need to correctly implement the matrix stack to view the initial coordinate axes. Your code should attach sliders to at least 3 joints and provide rotational control. You should also add a slider for translating the root joint to move the character in the world.

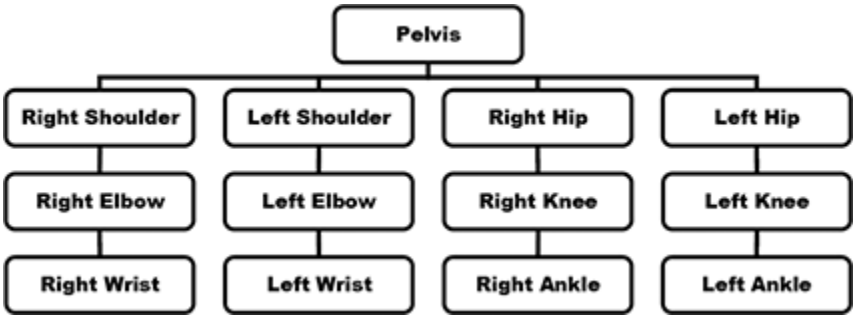
2.4 Artifact (5% of grade)

The artifact for this assignment will be easy to create: simply take a screenshot of one of the character models and submit it in PNG, JPEG, or GIF format. However, please take a few minutes to pose your character interestingly and choose a reasonable camera position. A straightforward extension would be to load multiple characters and pose them interacting together in an interesting way. You may also want to add a floor by drawing a flattened cube.

3 Hierarchical Modeling

In previous assignments, we addressed the task of generating static geometric models. As we've seen, this approach works quite well for generating objects such as spheres, teapots, wineglasses, statues, and so on. However, this approach is limited when applied to generating characters that need to be posed and animated. For instance, in a video game, a model of a human should be able to interact with the environment realistically by moving its limbs to imitate walking or running.

One approach to creating these animations is to individually manipulate vertices and control points. Doing so quickly becomes tedious. A better approach is to define a hierarchy such as a skeleton for a human figure and few control parameters such as joint angles of the skeleton. By manipulating these parameters, sometimes called articulation variables or joints, a user can pose the hierarchical shapes more easily. Furthermore, the surface of the object can also be computed as a function of the same articulation variables. An example of a skeleton hierarchy for a human character is shown below.

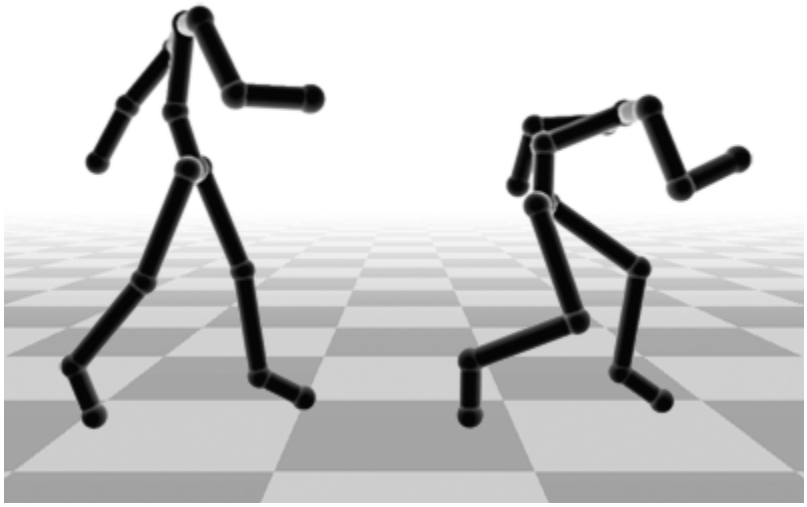


Each node in the hierarchy is associated with a transformation, which defines its local coordinate frame relative to its parent. These transformations will typically have translational and rotational components. Typically, only the rotational components are controlled by articulation variables given 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 node. 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?

In your code, your model will be drawn in this manner. By placing, say, a cylinder in the coordinate frame of the left hip, you can draw a simple thigh for your character. Doing this for all nodes in the hierarchy will result in simple stick figures, such as the ones shown below.



3.1 Matrix Stack (5% of grade)

Your first task is to implement a *matrix stack* similar to the one provided by OpenGL. 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.

The interface for the matrix stack has been defined for you in `MatrixStack.h`. The implementation in `MatrixStack.cpp` is currently empty and must be filled in. We recommend you simply use an STL `vector` for the stack, but you may use any data structure you wish. Once you have a working implementation, your initial display window should display the familiar $x - y - z$ axes in red, green, and blue, respectively. The camera controls should also work.

In order to use our matrix stack for drawing, simply call `glLoadMatrixf( m_matrixStack.top() )`. It will set the top of our matrix stack as the current geometric transformation matrix and subsequent OpenGL primitives (`glVertex3f`, `glutSolidCube`, etc) will be transformed by this matrix. The starter code's `SampleModel` class comes equipped with an instance of `MatrixStack` called `m_matrixStack` (in fact, it's inherited from `ModelerView`. The camera controller pushes the viewing transformation as the first item on the stack).

3.2 Hierarchical Skeletons (30% of grade)

3.2.1 File Input

Your next task is to parse a skeleton that has been built for you. The starter code automatically calls the method `SampleModel::loadSkeleton` with the right filename (found in `sample.cpp`). The skeleton file format (`.skel`) is straightforward. It contains a number of lines of text, each with 5 fields separated by a space. The first field is the integer index of the joint. The next 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, 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.

Your implementation of `loadSkeleton` must populate the following data structures in `SampleModel`: `m_rootJoint`, a pointer to the root joint, `m_jointIndexToNode`, a map that takes a joint index directly to its `SceneGraphNode`, and two utility variables `m_nJoints` and `m_nBones`, which contain the number of joints and bones, respectively. The number of bones is simply one less than the number of joints.

3.2.2 Drawing Stick Figures

To ensure that your skeleton was loaded correctly, we will draw simple stick figures like the ones above.

Joints We will first draw a sphere at each joint to see the general shape of the skeleton. The starter code already calls `SampleModel::drawJoints` with the root node. Your task is to recursively traverse the scene graph starting with the root and use it in conjunction with your matrix stack to draw a sphere at each joint. We recommend using `glutSolidSphere( 0.05f, 36, 36 )` 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. You may find it helpful to verify your rendering with that of the sample solution.

Bones A stick 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 `SampleModel::drawSkeleton`. Unfortunately, OpenGL's box primitive `glutSolidCube` can only draw cubes centered 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 $[-0.5, -0.5, 0]^T$ to $[0.5, 0.5, 1]^T$. Scale the box so that it ranges from $[-0.05, -0.05, 0]^T$ to $[0.05, 0.05, \ell]^T$ where ℓ is the distance to the next joint in your recursion. Rotate the z-axis so that it is aligned with the direction to the next joint: $z \rightarrow z'$. Since the x and y axes are arbitrary, we recommend mapping $y \rightarrow y' = z' \times rnd$, and $x \rightarrow y' \times z'$, with rnd supplied as $[0, 0, 1]^T$. Note that while you should take advantage of your matrix stack for these transformations, they are used only for transforming the drawing primitive and are not part of the actual skeleton hierarchy. As with the joints, you should verify the correctness of your implementation with the sample solution.

3.3 User Interface (5% of grade)

Now that we can draw a skeleton, we will add some primitive widgets to manipulate it. The starter code contains a number of files: the `modelerapp.*` and `modelerui.*` files handle the user interface. In particular, `modelerapp.*` define the `ModelerApplication` class, which you will use to access the values of the sliders in the user interface. Open `sample.h` and jump to the following code:

```
enum SampleModelControls
{
    FX, FY, FZ,
    NUMCONTROLS
};
```

This defines identifiers for the joints that are used to control the model. These are very abbreviated terms, and we'll get to their meanings shortly. Note that the last value listed in the enumeration must be `NUMCONTROLS`.

This enumeration is used to define the sliders in the `main` function:

```
ModelerControl controls[ NUMCONTROLS ];
controls[ FX ] = ModelerControl( "Foot Angle X", -M_PI, M_PI, 0.1f, 0 );
controls[ FY ] = ModelerControl( "Foot Angle Y", -M_PI, M_PI, 0.1f, 0 );
controls[ FZ ] = ModelerControl( "Foot Angle Z", -M_PI, M_PI, 0.1f, 0 );
...
```

The numbers in the ModelerControl constructor define the minimum, maximum, step size, and default value of the sliders, respectively.

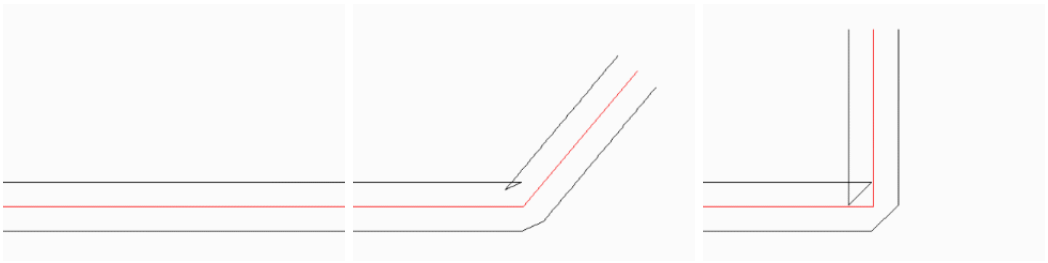
Your task is to add several more articulation variables and map them to transformations of your skeleton. Whenever a slider is dragged, the operating system calls `SampleModel::update`, which calls `SampleModel::updateSkeleton`. You can retrieve the value of a slider using the `VAL` macro: `VAL( FX )` returns the current value of the `FX` joint. You should add Euler Angle rotations for the x, y, and z axes for at least 3 joints, and a translation for the root joint to translate the character in the world. You may find the `m_jointIndexToNode` map helpful in locating the right nodes in the scene graph.

For more details regarding the implementation of the user interface, please read through the reasonably well-commented files `modelerapp.cpp`, `modelerui.cpp`, `modelerview.cpp`, and `sample.cpp` files.

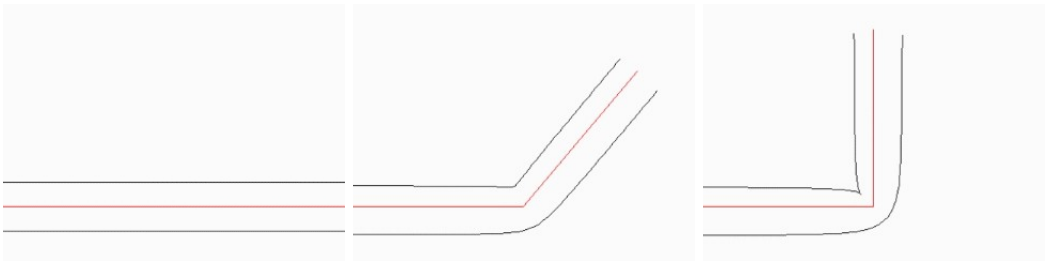
4 Skeletal Subspace Deformation

Hierarchical skeletons allowed you to render and pose vaguely human-looking stick figures in 3D. In this section, we will use Skeletal Subspace Deformation to attach a mesh that naturally deforms with the skeleton.

In the approach used to render the skeleton, body parts (spheres and cubes) were drawn in the coordinate system of exactly one joint. This method, however, can generate some undesirable artifacts. Observe the vertices near a joint.



This is a cross-sectional view of a skeleton with a mesh attached to each node. Notice how the two meshes collide with each other as the skeleton bends. Our stick figures hide this artifact by drawing spheres at each joint. However, it is only a quick fix for the fact that the aforementioned approach rigidly attaches vertices of the character model to individual nodes of the hierarchy. This is an unrealistic assumption for more organic characters (such as humans and animals) because skin is not rigidly attached to bones. It instead deforms smoothly according to configuration of the bones, as shown below.



This result was achieved using skeletal subspace deformation, which positions individual vertices as a weighted average of transformations associated with nearby nodes in the hierarchy. For example, a vertex near the elbow of the model is positioned by averaging the transformations associated with the shoulder and elbow joints. Vertices near the middle of the bone (far from a joint) are affected by only that bone—they move rigidly, as they did in the previous setup.

More generally, we can assign each vertex a set of *attachment weights* which describes how closely it follows the movement of each bone. A vertex with a weight of one for a given bone will follow that bone rigidly, as it did in the previous setup. A vertex with a weight of zero for a bone is completely unaffected by that bone. Vertices in between are blended—we compute their position as if they were rigidly attached to each bone, then average these positions according to the weights we've assigned them.

In the previous section, a vertex was defined in the local coordinates of a given bone—you probably used methods like `glutSolidCube`, then transformed the entire object via a translation to the joint's location. Now, however, vertices don't belong to a single bone, so we can't define vertices in the local coordinate frame of the bone they belong to. Instead, we define the mesh for the entire body, and keep track of the *bind pose*—the pose of each bone in the body such that the bones match up with the locations of the vertices in the mesh. Imagine taking the skin of a character, then fitting a skeleton inside that skin. The skeleton which matches up with the skin's position is in the character's bind pose.

Let's say that $\mathbf{p}$ is the position of a vertex in a character's coordinate frame *in the bind pose*. Say that $\mathbf{p}$ is affected by bone 1 and bone 2. Let's also say that the *bind pose transformation* of bone 1 (the transformation which takes us from the local coordinate frame of bone 1 before the character has been animated to the character's coordinate frame) is $\mathbf{B_1}$. Finally, the transformation from bone 1's local coordinate frame to the character's coordinate frame *after animation* is $\mathbf{T_1}$. Then the position of our vertex after transformation, if that vertex were rigidly attached to bone 1, would be $\mathbf{T_1 B_1^{-1} p}$. Notice that we have to first transform the point into the local coordinate system of the bone ($\mathbf{B_1^{-1} p}$) before transforming it (remember, $\mathbf{p}$ is in the character's bind pose coordinate system). Similarly, the bind transformation of bone 2 is described by $\mathbf{B_2}$, and $\mathbf{T_2}$ describes the transformation from the unanimated local coordinate frame of bone 2 to the animated character coordinate frame. Then the vertex's position, if it were rigidly attached to bone 2, would be $\mathbf{T_2 B_2^{-1} p}$. However, say that the given vertex is near the joint of bone 1 and bone 2, and we want it to be attached to both bones. We assign each bone a weight according to how much influence that bone should have on the vertex. Weights will usually range between 0 and 1 for each bone, and the weights for all bones will usually sum to 1. We want it tied to bone 1 with a weight of w, so it is tied to bone 2 with a weight of $(1 - w)$. Then we compute the final position of the vertex as $w\mathbf{T_1 B_1^{-1} p} + (1 - w)\mathbf{T_2 B_2^{-1} p}$.

Note that since we usually only have one bind pose for a character, the inverse bind transformations $\mathbf{B_i^{-1}}$ need to be computed only once. On the other hand, since we want to animate the character using our user interface, the animation transforms $\mathbf{T_i}$ need to be recomputed every time the joint angles change. This implies that the vertex positions will also need to be updated whenever the skeleton changes. (Although recomputing $\mathbf{T_i}$ is relatively cheap on a character with few bones, updating the entire mesh can be quite expensive. Modern games typically perform SSD on many vertices in parallel using graphics hardware.)

4.1 File Input: Bind Pose Mesh (5% of grade)

To get started, we will first need to adapt your code from assignment 0 to load the bind pose vertices from an OBJ file. The starter code automatically calls `SampleModel::loadMesh` with the appropriate filename. The only difference between this part and assignment 0 is that the meshes we provide for you do not include normals. Instead, we will generate them on-the-fly when we render. Your code should populate the `bindVertices` and `faces` fields of `m_mesh`. Notice that our `Mesh` struct comes with two copies of vertices: the bind pose and the current pose. We will render from the current pose vertices, which are generated by transformations of the bind pose vertices. The starter code makes the initial copy for you.

4.2 Mesh Rendering (5% of grade)

Next, we will verify the correctness your mesh loader by rendering the mesh. The starter code calls `SampleModel::drawMesh` automatically with the right filename. Since the mesh vertices are already in the character's coordinate frame, you do not need to manipulate your matrix stack (except perhaps to load the top, which should contain the camera transformation, depending on how your code is organized). Be sure to render from `m_mesh.currentVertices` and not `m_mesh.bindVertices`.

Unlike meshes from previous assignments, these meshes do not provide any per-vertex normals since they were not computed analytically. Instead, we will generate a single normal for each triangle on-the-fly inside rendering loop by taking the cross product of the edges. Don't forget to normalize your normals. Note how your model appears "faceted": the lighting is discontinuous between neighboring faces because the normals change abruptly.

4.3 File Input: Attachment Weights (5% of grade)

The last thing we must load are the attachment weights. The starter code calls `loadAttachments` automatically with the right filename. The attachment file format (`.attach`) is straightforward. It contains a number of lines of text, one per vertex in your mesh. Each line contains `m_nBones` (number of bones, or one less than the number of joints) fields separated by spaces. Each field is a floating point number that indicates how strongly the vertex is attached to the i-th bone.

Your code should populate the `attachments` field of `m_mesh`. We recommend the starter code's data structure, where `m_mesh.attachments` is a `vector< vector< float > >`. The inner vector contains one weight per bone, and the outer vector has size equal to the number of vertices. You should verify that the attachment weights sum to 1.

4.4 Implementing SSD (40% of grade)

Finally, we will implement SSD as described above. We will first compute all the transformations necessary for blending the weights and then use them to update the vertices of the mesh.

4.4.1 Computing Transforms

As we describe above, we must compute the bind pose world to bone transformations (once) and the animated pose bone to world transformations (every time the skeleton is changed). The starter code automatically calls `computeBindWorldToBoneTransforms` and `updateCurrentBoneToWorldTransforms` at the appropriate points in the code.

`computeBindWorldtoBoneTransforms` should populate `m_bindWorldToBoneTransforms`, which is a `vector` mapping the i-th bone to its transform. You should use a recursive algorithm similar to the one you used for rendering the skeleton. Be careful with the order of matrix multiplications. Think carefully about which

space is the input and which space is the output.

`updateCurrentBoneToWorldTransforms` is called whenever the skeleton changes. Your implementation should update `m_currentBoneToWorldTransforms` and will be very similar to your implementation of `computeBindWorldtoBoneTransforms`. But once again, be careful about which spaces you're mapping between. One convenient method for debugging is that if your skeleton did not change (i.e., you did not touch any sliders), the bind pose world to bone transform for any vertex should be the inverse of the animated pose bone to world tranform.

4.4.2 Deforming the Mesh

For the final part your assignment, you will deform the mesh according to the skeleton and the attachment weights. Since you've populated all the appropriate data structures, your implementation should be straightforward. The starter code calls `SampleModel::updateMesh` whenever the sliders change. Your code should update the current position of each vertex in `m_mesh.currentVertices` according to the current pose of the skeleton by blending together transformations of your bind pose vertex positions in `m_mesh.bindVertices`.

If you implemented SSD correctly, your solution should match the sample solution. Since you have several more joints in your user interface, you will have much more freedom to pose your characters in interesting ways. Feel free to change the appearance of your characters and pose multiple characters together to make an interesting scene.

5 Extra Credit

As with the previous assignment, the extra credits for this assignment will also be ranked *easy*, *medium*, and *hard*. These categorizations are only meant as a rough guideline for how much they'll be worth. The actual value will depend on the quality of your implementation. E.g., a poorly-implemented *medium* may be worth less than a well-implemented *easy*. We will make sure that you get the credit that you deserve.

5.1 Easy

- Generalize the code to handle multiple characters by storing multiple skeletons, meshes, and attachment weights. For efficiency, much of the data can be shared between multiple instances of the same character.
- Embed the control points of a generalized cylinder's sweep curve in the character hierarchy. If you have implemented SSD and your code from the previous assignment is modular, this should be quite straightforward. This is an alternative way to provide smooth skins for body parts like arms and legs. However, note that this technique is less general than SSD because it can only handles non-branching hierarchies.
- Employ OpenGL texture mapping to render parts of your model more interestingly. The OpenGL Red Book covers this topic in Chapter 9, and it provides plenty of sample code which you are free to use.
- Simulate the appearance of a shadow or reflection of your model on a floor using OpenGL. While performing these operations in general is quite complicated, it is relatively easy when you are just assuming that a plane is receiving the shadows. The OpenGL Red Book describes one way to do this in Chapter 14.
- For your SSD implementation, use pseudo-colors to display your vertex weights. For example, assign a color with a distinct hue to each joint, and color each vertex in the model according to the assigned weights by computing the corresponding weighted average of joint colors.

- There are numerous other tricks that you might try to make your model look more interesting. Feel free to implement extensions that are not listed here, and we'll give you an appropriate amount of extra credit.

5.2 Medium

- Implement intuitive manipulation of articulation variables through the model display. For instance, if the elbow joint is active, the user should be able to click on the arm and drag it to set the angle (rather than using the slider). For an example of such an interface, give Maya a try.
- Implement a method of animating your character by using interpolating splines to control articulation variables. This method of animation is known as keyframing. You may either allow input from a text file, or for additional credit, you may implement some sort of interface that allows users to interactively modify the curves.
- Accelerate SSD by implementing it using *shaders* on graphics hardware using the *OpenGL Shading Language* or *GLSL*. If you choose to do this, please note which hardware platform it works on and include it in your `README` file.

5.3 Hard

- Implement pose space deformation. This method is an alternative to skeletal subspace deformation which often gives higher-quality results.
- Implement inverse kinematics, which solves for articulation variables given certain positional constraints. For instance, you can drag the model's hand and the elbow will naturally extend. Your code should allow interactive manipulation of your model through the drawing window.
- Implement mesh-based inverse kinematics, which allows a model to be posed without any underlying skeleton.

6 Submission Instructions

As with the previous assignment, you are to write a README.txt,pdf that answers the following questions:

- How do you compile and run your code? Provide instructions for Athena Linux.
- Did you collaborate with anyone in the class? If so, let us know who you talked to and what sort of help you gave or received.
- Were there any references (books, papers, websites, etc.) that you found particularly helpful for completing your assignment? Please provide a list.
- Are there any known problems with your code? If so, please provide a list and, if possible, describe what you think the cause is and how you might fix them if you had more time or motivation. This is very important, as we're much more likely to assign partial credit if you help us understand what's going on.
- Did you do any of the extra credit? If so, let us know how to use the additional features. If there was a substantial amount of work involved, describe what how you did it.
- Got any comments about this assignment that you'd like to share?

Submit your assignment as a single archive (`.tar.gz` or `.zip`) on Stellar by **October 13 by 8:00pm**. It should contain:

- Your source code.
- A compiled executable named a2.
- Any additional files (OBJs, textures) that are necessary.
- The `README` file.
- Your artifact(s) in PNG, JPEG, or GIF format.