Programming Assignment 2: Hierarchical Modeling and SSD

MIT 6.837 Computer Graphics

Fall 2021

Due October 13 at 8pm, Boston Time

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 (such as watches, tricycles), and so on. You will implement skeletal subspace deformation (SSD), a simple method for attaching a "skin" to a hierarchical skeleton which naturally deforms when we manipulate the skeleton's joint angles.

1 Getting Started

The sample solution is included in the starter code distribution. Look at the `sample_solution` directory for Linux, Mac, and Windows binaries. You may need to change the file mask on the Linux/Mac binaries, e.g., by executing the following command in your terminal:

```
chmod a+x sample_solution/linux/assignment2
```

To run the sample solution with one of the test models (passing a prefix relative to `assets/assignment2` folder), use, e.g.,

```
sample_solution/linux/assignment2 Model1
```

The sample solution (Figure 1) shows a completed version of the homework, including loading and displaying a skeleton, loading a mesh that is bound to the skeleton, and deforming the skeleton and mesh based on joint angles. You can press `S` to toggle drawing of the skeleton or the deformed mesh. Overlayed are a collection of UI sliders that can be used to set the local rotations (as Euler angles) of each joint. By manipulating one of the sliders, you will be able to change the pose of the characters. You can change the camera view using mouse control just like in the previous assignment: the left button rotates, the middle button moves, and the right button zooms. Like before, you can press `A` to toggle drawing of the coordinate axes.

To build the starter code, follow the same steps as in previous assignments. For instance, on MacOS or Linux, execute the following:

```
mkdir build
cd build
cmake ..
make
```

If you run the starter code now, a window will pop up but it will be mostly empty except for the UI sliders. Eventually it will contain a rendering of your character.

2 Requirements

This section summarizes the core requirements of this assignment. You will find more details regarding the implementation of these requirements later in this document.

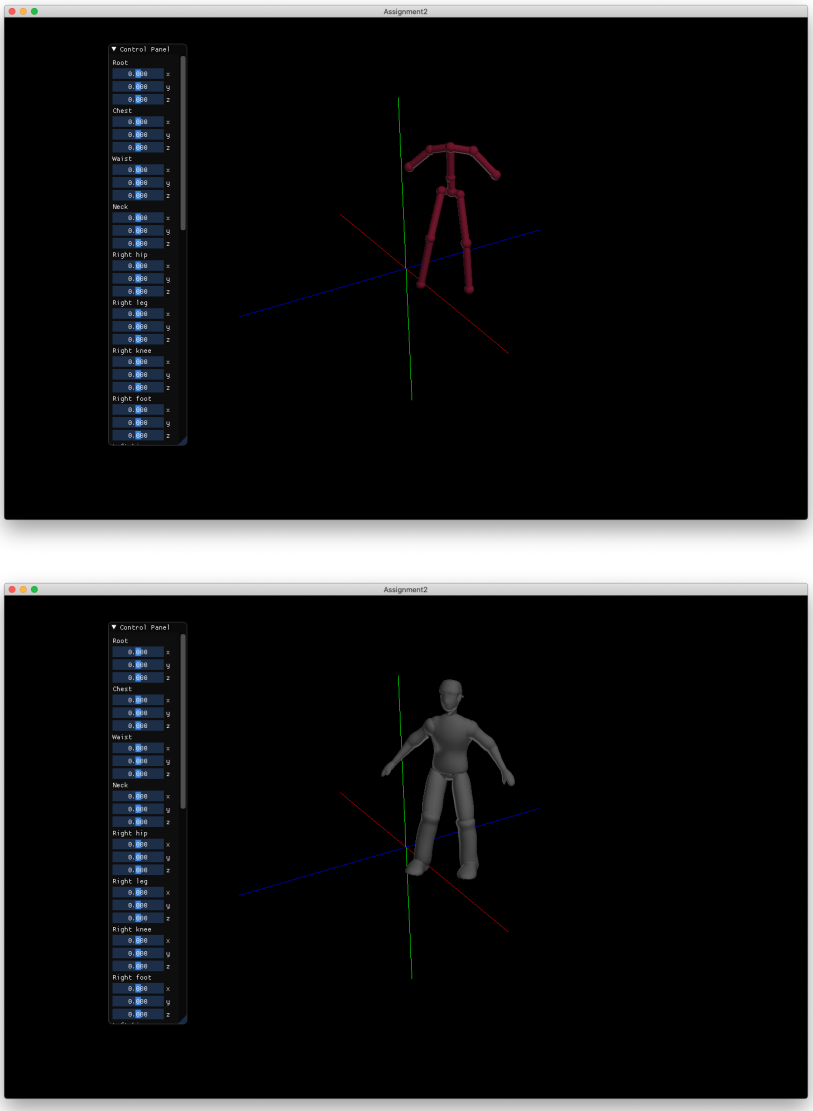

Figure 1: Sample solution. Top: skeleton view. Bottom: SSD view of the same skeleton, after pressing `S`.

For this and future assignments, you are free to start your implementation from scratch in C++. The starter code, however, implements a number of these requirements already, which **must** be present in your submission; missing features can cause your submission to receive no credit. Also note that you will need to modify source code under the `gloo` folder in this assignment.

2.1 Hierarchical Modeling (50% of grade)

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, wine glasses, statues, and so on. However, this approach is limited when generating characters that need to be posed and animated. For instance, in a video game, the model of a character should be able to interact with the environment realistically by moving its limbs differently for walking and running.

One approach to achieve this is to manually set the positions of control points and their orientations in the world space. However, this quickly becomes intractable for a large number of control points. A better approach is to define a hierarchy (such as a skeleton for a human figure) together with a small

number of control parameters such as joint angles of the skeleton. By manipulating these parameters (often called articulation variables), a user can modify the pose of the hierarchical shapes more easily. An example of a skeleton hierarchy for a human character is shown below (Figure 2).

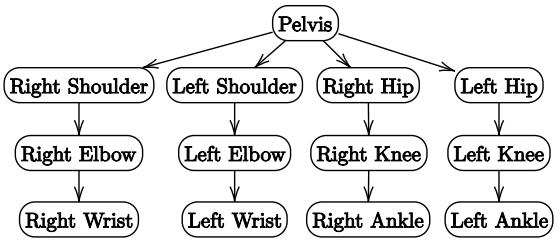

Figure 2: Part of a human skeleton hierarchy.

Each node in the hierarchy is associated with a local transformation, which transforms its coordinate frame to that of its parent. These transformations will have translational, rotational, and scaling components. Typically, the rotational components are controlled by the articulation variables determined by the user. The coordinate frame of the root of the hierarchy tree is considered as the global coordinate frame. Thus, to determine the transformation from the coordinate frame of a node to the global coordinate frame, we just need to multiply the local transformations (realized as matrices) down the tree (make sure you understand the order of multiplication here).

In this part of the assignment, you will implement hierarchical modeling and create stick figures to visualize the hierarchy (such as Figure 3). For example, you will create a sphere for every joint and a

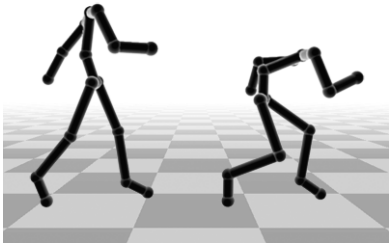

Figure 3: Sticky figures for walk animation.

cylinder for each bone connecting two joints. Think carefully about which coordinate frame to put the cylinders in.

2.1.1 Local-to-World Transformation (5% of grade)

Your first task is to compute local-to-world transformations for each node, in the scene tree. In GLOO, each `SceneNode` is associated with a `Transform` object (See `gloo/Transform.hpp`). In the past assignments and in the starter code of this assignment, the `Transform` object is treated as the transformation that brings the local coordinate frame to the world space and the hierarchical information is not used. Your job here is to treat `Transform` as the transformation that brings the local frame to the coordinate frame of the node's parent. Specifically, in the starter code, you can see that `Transform::GetLocalToWorldMatrix` returns `local_transform_mat_`, which is the transformation matrix corresponding to the translation (`position_`), rotation (`rotation_`), and scaling (`scale_`) in the member variables. You will need to change the implementation of `Transform::GetLocalToWorldMatrix` so that it returns the transformation from the local coordinate frame to the global coordinate frame. This amounts to recursively multiply the transformation matrices from the root down to the node. To get the parent node as a pointer, use `SceneNode::GetParentPtr`. The starter code also declares a slightly

more general method called `Transform::GetLocalToAncestorMatrix` which takes a `SceneNode*` argument named `ancestor`. You can implement this method to compute the transformation from the local coordinate frame to the coordinate frame of `ancestor` (and to the root if `ancestor == nullptr`). With this method, `Transform::GetLocalToWorldMatrix` can be easily implemented by passing in `nullptr`. This `Transform::GetLocalToAncestorMatrix` can become handy for implementing SSD.

2.1.2 Optimization Using a (Implicit) Matrix Stack (5% of grade)

Now if you take a look at `Renderer::RetrieveRenderingInfo`, you will notice that when drawing the scene we will collect an array of pairs of type `std::pair<RenderingComponent*, glm::mat4>`, where the first component of the pair is a pointer to a `RenderingComponent`, and the second component of the pair is a 4×4 local-to-world matrix that transforms the coordinate frame of the node (where this `RenderingComponent` is attached) to the world space (i.e. the global coordinate frame). In the starter code, the local-to-world matrix is obtained by calling `Transform::GetLocalToWorldMatrix` that you just implemented. However this can become inefficient if the scene tree is deep, as the same transformation matrices will be calculated multiple times during multiple calls to `Transform::GetLocalToWorldMatrix` on the scene tree nodes. In the worst case, the number of matrix multiplications can be $O(n^2)$, where n is the number of nodes. (Think about the case where all nodes form a very long chain.)

A more efficient way is to use a matrix stack as introduced in the lecture, so that the local-to-world matrix of each node will only be computed once and the total number of matrix multiplications is reduced to $O(n)$. You will implement this optimization in `Renderer::RetrieveRenderingInfo`. In your implementation, you don't need to maintain a stack explicitly. Instead, you can use a recursive depth-first search that traverses the scene tree while maintaining the current local-to-world matrix as an argument of a recursive function that gets passed on to the recursive calls on the children nodes. For instance, your recursive function might look like

```
void Renderer::RecursiveRetrieve(const SceneNode& node,
                                 RenderingInfo& info,
                                 const glm::mat4& model_matrix);
```

where `node` is the current node, `info` is a reference to a `std::vector` containing pairs of `RenderingComponent*` and `glm::mat4` (this array type is aliased as `RenderingInfo`), and `model_matrix` is the transform matrix from the local frame of `node` to the global frame. To access the child information of a node, use `SceneNode::GetChildrenCount` and `SceneNode::GetChild`. This approach will be the same as using a matrix stack for rendering as discussed in the lecture, except here the stack is maintained implicitly on the function call stack. Note that you should only collect `RenderingComponent`'s and the local-to-world matrices from active nodes — you can use `SceneNode::IsActive` to check if a node is active. Make sure your implementaion is equivalent in functionality to the starter code of `Renderer::RetrieveRenderingInfo`. In particular, you should not change the signature (e.g. list of arguments, return type) of `Renderer::RetrieveRenderingInfo`.

2.1.3 Hierarchical Skeletons (40% of grade)

After you've finished the previous sections to support hierachical modeling, it is time to write code to visualize a skeleton model and respond to joint rotations from the user control. The starter code has taken care of parsing command line arguments (`main.cpp`) and setting up the scene (`SkeletonViewerApp.*pp`); you don't need to modify these files. It suffices to deal with just `SkeletonNode.*pp` for the remainder of the assignment (you're more than welcome to add more source files if needed). A `SkeletonNode` is a derived class of `SceneNode` whose subtree will represent the hierarchy of the model.

Parsing the skeleton file Your next task is to parse a skeleton that has been pre-built for you. In the starter code, the constructor of `SkeletonNode` calls `SkeletonNode::LoadAllFiles` with the right file prefix. The method `SkeletonNode::LoadAllFiles` calls three load methods that load a skeleton file (`.skel`), a mesh file (`.obj`), and attachment weights (`.attach`). In this section you only need to be concerned with implementing `SkeletonNode::LoadSkeletonFile` to parse skeleton files.

The skeleton file format (`.skel`) is very straightforward. It contains a number of lines, each with 4 numbers separated by a space. Each line corresponds to a joint. The first three floating-point numbers

describe the joint's translation relative to its parent joint. The last integer is the index of its parent where a joint's index is the zero-based index according to the order of occurrence in the `.skel` file (so the joint described in the second line has index 1). For the root node, its parent index is -1 and the translation 3-tuple is its global position in the world. **You can assume that the root node will be on the first line and every line below describes a node whose parent appears before that line.**

You should build a hierarchy of joints in `SkeletonNode::LoadSkeletonFile`. The design choice of how to represent this hierarchy is up to you. In the sample solution, we create a `SceneNode` for each joint and add them to `SkeletonNode` obeying the same hierarchy from the skeleton file (with prescribed parent-child relations and child-to-parent translations). We store pointers to these "joint nodes" as private member variables in `SkeletonNode` (since they as unique pointers will be `std::move`'d to be added to the tree).

C++ tips for reading the files:

1. `std::fstream` (click for doc and examples) is your friend for reading/writing files.
2. You can make use of `std::getline` (click for doc and examples) to read an entire line of the input file.
3. To parse the numbers in a single line in a `std::string`, you can use `std::stringstream` (click for doc and examples).

Drawing the stick figures You will need to draw a sphere at each joint and a cylinder ("bone") between each joint and its parent (except for the root node in the skeleton hierarchy, not to be confused with the root of the scene tree).

To create a sphere mesh, you can use `PrimitiveFactory::CreateSphere`. The first argument sets the radius of the sphere, and the last two arguments set the discretization level of the sphere along longitudinal strip and latitudinal strip respectively. The resulting sphere will be centered at origin in its local coordinate frame. The static method `PrimitiveFactory::CreateSphere` returns a `std::unique_ptr<VertexObject>`, which can be readily attached to `RenderingComponent`. Note that you only need to create one sphere mesh and can reuse it for multiple joint nodes (i.e. multiple `RenderingComponent`). The same goes for the shader class `PhongShader` (you will be using phong shading throughout the assignment). It is recommended to create a new `SceneNode` for displaying the sphere, and attach it as a child to each joint node, instead of directly attaching `RenderingComponent` to joint nodes. This way, every joint node will serve as a bookkeeper of the skeleton hierachy without any mesh information attached.

To create a cylinder mesh as a `VertexObject`, you can use `PrimitiveFactory::CreateCylinder`, which takes a radius, a height, and a discretization level of the number of sides. In its local coordinate frame, the resulting cylinder will have its bottom face in the xz-plane and the center of bottom face at the origin, while the positive y-axis passes through the rotational symmetry axis of the cylinder (remember in graphics convention positive y direction is up). Similar to the sphere, to save memory you can just create a single cylinder mesh and reuse it for different bones. Bones are a bit trickier than joints since each bone links a child node and a parent node. Feel free to design your own solution. You will need to set the correct transformation for this bone node so that it connects the two joints (hint: construct your cylinder in either the local frame of the child or the local frame of the parent, and you need to connect the two origins in both frames). Note that you can scale along y direction to display a cylinder of a desirable length (e.g. calling `SetScale(1.0f, l, 1.0f)` on the `Transform` of the node where the cylinder is attached to via `RenderingComponent`). You will also need to rotate the cylinder so that it aligns with a given direction (e.g. find the correct rotation axis and the angle, and then use `Transform::SetRotation(const glm::vec3& axis, float angle)` to rotate around `axis` about `angle` in radians).

Handling UI When any rotation slider is dragged, the method `SkeletonNode::OnJointChanged` will be called. You should implement `SkeletonNode::OnJointChanged` to do the necessary updates. This includes updating the rotation of each joint based on the Euler angle from the sliders, as well as recomputing the right transformations for the bones. The current Euler angle for each joint can

be retrieved from the private member `linked_angles_` in `SkeletonNode`. To convert an Euler angle to a quaternion that can be passed in `Transform::SetRotation` (one of the overloaded versions), use `glm::quat(glm::vec3(angle.rx, angle.ry, angle.rz))`. Note that `SkeletonNode::OnJointChanged` should also be called once after everything is initialized before any UI input occurs (this is done in the starter code).

After finishing the above steps, you should be able to see your skeleton when you run the code and your skeleton's joints and bones can respond correctly when you drag the sliders.

2.2 Skeletal Subspace Deformation (50% of grade)

Hierarchical skeletons allow you to render and pose vaguely human-looking stick figures in 3D. In this section, we will use skeletal subspace deformation (SSD) to attach a mesh that naturally deforms with the skeleton.

In the previous approach used to render the skeleton, body parts (spheres and cylinders) were drawn in the coordinate frame of exactly one joint. This can lead to undesirable artifacts during the animation. Consider the scenario pictured below (Figure 4) where two adjacent bones are bent to form 90°. Notice

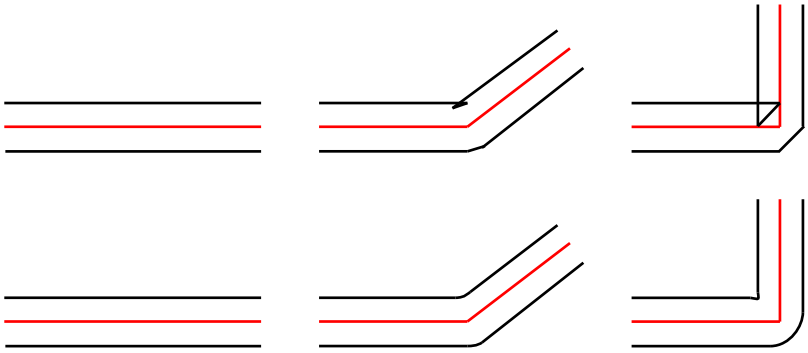

Figure 4: Cross-sectional view of a skeleton with a mesh attached to each node. Top: colliding meshes without SSD. Bottom: nicely deformed meshes using SSD.

how the two meshes collide with each other as the skeleton bends in the top image. Our stick figures hide this artifact by drawing spheres at each joint. However, for more organic characters (such as humans and animals), skin is not rigidly attached to bones. It instead deforms smoothly according to configuration of the bones, as shown in the bottom image. This result can be achieved using skeletal subspace deformation, which positions individual vertices according to a weighted average of transformations associated with nearby joints 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 one joint; they move rigidly, as they did in the previous setup.

More generally, we can assign each vertex a set of attachment weights between 0 and 1 which describes how closely it follows the movement of each joint. The weights for all joints of a given vertex should sum up to 1. A vertex with a weight of 1 on a joint will follow that joint rigidly, as it did in the previous setup. A vertex with a weight of 0 on a joint is completely unaffected by that joint. Vertices with weights between 0 and 1 are blended – we compute their positions as if they were rigidly attached to each joint, and then average these positions according to the weights we've assigned them.

In the previous section, you probably set up spheres and cylinders in the local coordinate frame of a given joint. Now, however, skin vertices don't belong to a single joint, so we can't define vertices in the local coordinate frame of joints. Instead, we start with a given skin mesh in the bind pose, the default pose before animation where the vertices are defined in the character's coordinate frame (a global frame without hierarchy). We require that the initial skeleton (in our case this is specified in the `.skel`

skeleton file) is also in the bind pose, i.e., the joints and bones match the given skin mesh in the bind pose. Imagine taking the skin of a character, then fitting a skeleton inside that skin. The skeleton which matches up with the skin's vertices is in the character's bind pose.

Let's say that p is the position of a vertex in a character's coordinate frame in the bind pose. Suppose p is affected by joint 1 and 2 only. Let B_1 denote the transformation that takes us from the local coordinate frame of joint 1 **before animation** to the character's coordinate frame. Hence $B_1^{-1}p$ is the position of the vertex in the local coordinate frame of joint 1 in the bind pose. Next denote T_1 the transformation from the local coordinate frame of joint 1 **after animation** to the character's coordinate frame. Then the position of our vertex after animation, if the vertex were rigidly attached to joint 1, would be $T_1B_1^{-1}p$. Similarly we define B_2, T_2 to be the transformation from the local coordinate frame of joint 2 before and after animation respectively to the character's coordinate frame. If the weight of joint 1 is w and that of joint 2 is $1-w$, then the final position of the vertex after anmation is computed as $wT_1B_1^{-1}p + (1-w)T_2B_2^{-1}p$.

Note that since we usually only have one bind pose for a character, the inverse transformations 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 transformations 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 T_i is relatively cheap on a character with few joints, updating the entire mesh can be quite expensive. Modern games typically perform SSD on many vertices in parallel using the graphics card's vertex shader. You can implement this for extra credit (see end of handout). For just the assignment, it is okay to update all vertex positions (and all normals) whenever the joint angles change.

Parsing the bind pose mesh file To get started, we will need to load the bind pose skin mesh. This is done in `SkeletonNode::LoadMeshFile` and the parsing of the OBJ file has been implemented for you. You will need to store the bind pose mesh in your preferred way (e.g. as a member variable of `SkeletonNode` class).

Parsing the attachment weights file Next, you need to load the attachment weights of the model. The attachment weights file format (`.attach`) is straightforward. Let n denote the number of vertices and m the number of joints. Then the attachment weights file contains a $n \times (m-1)$ matrix of floating point numbers, where each line corresponds to a row in the matrix in the natural fashion. The ith row jth column (0-based index) element of this matrix is the attachment weight of joint $j+1$ for vertex i. The weight for the 0th joint, the root, is assumed to be zero, and that's why we skip its column and only have $m-1$ columns in the matrix. Your code needs to parse this file and store the resulting attachment weights matrix. It is recommended to use a 2D array of type `std::vector<std::vector<float>>` for this matrix.

Deforming the mesh As detailed above, you will need to the compute the transformations B_i, T_i's and then deform the mesh according to the attachment weights. The pre-animation local-to-character transformation B_i's only need to be computed once, whereas the post-animation local-to-character transformation T_i's should be recalculated whenever the joint angles change (this happens precisely when `SkeletonNode::OnJointChanged` is triggered). We will leave the design details for you.

Computing normals One last thing to do is to compute the normals of the deformed mesh. **You will need to compute per-vertex normals.** This can be achieved by first computing the per-face normals by taking the cross product of the edges on each face. Then to get the normal for a vertex, you need to take a weighted average over the normals on all incident faces, where the weight is proportional to the area of each face. You should create a `NormalArray` of size equal to the number of vertices of the mesh and attach it to the same `VertexObject` that contains the vertex positions of the mesh. Similar to the vertex positions, you will need to update all the normals whenever there is a joint angle change. Don't forget to normalize your normals. Note this doesn't guarantee to give consistently oriented normals. Your model may appear "faceted" because the normals change abruptly. You don't need to address this issue in this assignment.

Toggling between views Finally, implement `SkeletonNode::ToggleDrawMode` to change between the skeleton view and the SSD view. The starter code has already handled the pressing of `S` which will call `SkeletonNode::ToggleDrawMode`. This amounts to hide or show a bunch of `SceneNode` depending on the draw mode enum `draw_mode_`. Hiding or showing a `SceneNode` can be done via `SceneNode::SetActive`, since `Renderer` will only pick up active nodes.

If you implemented SSD correctly, your solution should match the sample solution. Feel free to change the appearance of your characters and extend your code to pose multiple characters together to make an interesting scene (you will need to redesign the UI).

3 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.

If you do any of these extra credits, please make sure that your program does not lose any of the required functionality described above.

3.1 Easy (3% for each)

- Generalize the code to handle multiple characters by storing multiple skeletons, meshes, and attachment weights. Optionally, you can reuse data between instances of the same character.
- 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.

3.2 Medium (6% for each)

- Implement SSD using the vertex shader. You should store the bind pose in a vertex buffer, and define a set of uniform variables for the joint transformations. For each frame, upload the latest joint transforms, and compute the transformed vertex positions and normals in the shader.
- Embed the control points of a generalized cylinder's sweep curve in the character hierarchy. 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.
- Build your own model out of generalized cylinders and surfaces of revolution, and pose it using SSD.

- 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.

3.3 Hard (9% for each)

- 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.

4 Submission Instructions

You are to include a `README.txt` file or PDF report that answers the following questions:

- How do you compile and run your code? Specify which OS you have tested on.
- 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. In particular, mention if you borrowed the model(s) used as your artifact from somewhere.
- 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 how you did it.
- Do you have any comments about this assignment that you'd like to share? We know this was a tough one, but did you learn a lot from it? Or was it overwhelming?

You should submit your entire project folder including your **source code** and **executable (at the root directory of your project)**, but **excluding `external/` and `build/`** since they take too much space. **Make sure your code is compilable since we will compile it from scratch.**

To sum up, your submission should be your entire project folder containing your source code and the executable, plus:

- The `README.txt` file or PDF report answering the questions above. Leave this file at the root directory of the project folder.
- Screenshot of one of the character models, one in skeleton view and one in SSD view. Please take a few minutes to pose your character interestingly and choose a reasonable camera position. Feel free to put the screenshots in your PDF report if you are making one. You can leave the screenshots at the root directory of the project folder.

We will follow the late day policy explained in Lecture 1.