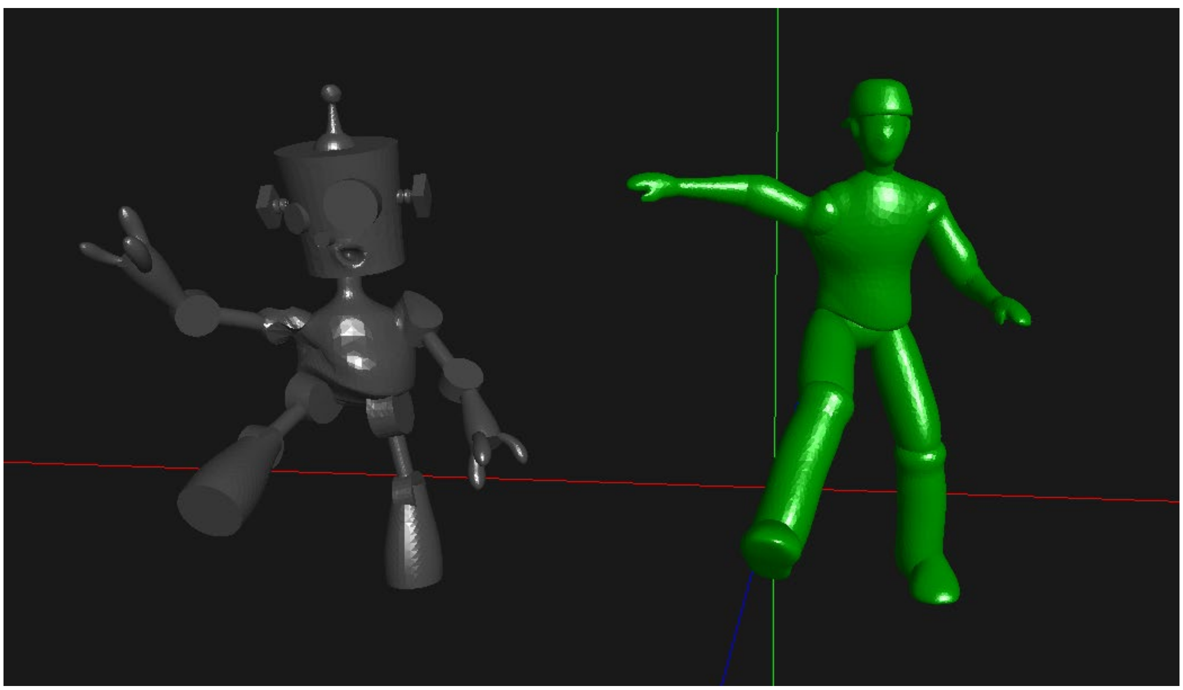
Assignment 4

CPSC 566 – Advanced Computer Graphics
Assignment 4: Hierarchical Modeling and SSD
Spring 2025
Professor: Dave Garcia-Gomez
Version: 2025-03-28

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
2. Summary of Requirements
3. Hierarchical Modeling
4. Skeletal Subspace Deformation
5. Submission Instructions

1 Getting Started

Download the code as provided, build the executable with make, and run the resulting executable on the first test model: **`(./editor`**). The familiar editor will open up, except now you will have the option to Import Character. Selecting the Import Character shape will load a model of your character, and eventually your editor will contain a rendering of your character. The other difference is that your shape properties for an imported character will contains a list of *articulation variables* or simply joints.

By clicking on joint slider and dragging left or right, or by entering a value in the slider will lets you manipulate that variable. You can change the camera view using mouse control just like in previous versions of the editor: the left button moves, the middle scroll wheel zooms, and the right button rotates.

The sample solution **`editor_solution`** shows a completed version of the project, including loading and displaying a skeleton, loading a mesh that is bound to the skeleton, and deforming the skeleton and mesh based on the joint angles. You can toggle between displaying the skeleton and displaying the mesh through the shape properties.

Additionally, some of the previous functionality of the editor is missing. You can use your completed code from your previous assignments and move them into your editor. For example, copy your **Curve.cpp** and **Surface.cpp** files from your previous assignment regarding **Curves** and **Surfaces** into your **src** folder. This will allow the functionality you've built in your previous assignment over to this version of the editor.

The files that you would be editing are **MatrixStack.cpp**, **SkeletalModel.cpp**, and **ImportCharacter.cpp**. Go through each of these source files, along with the header files of each class, and try and understand the data structures associated to the classes to get a better understanding of how they work. You can also look at **FileImporter.cpp** as well as the **.skel**, **.attach** and the **.obj** files of your characters to see what information is being loaded and how they are utilized.

2 Summary of Requirements

2.1 Hierarchical Model (40% of grade)

For part one of this assignment, you are required to correctly load, display, and manipulate a hierarchical skeleton. The loading portion will be already developed for you. Your implementation must be able to correctly parse any of the provided skeleton files (**`*.skel`**), construct a joint hierarchy, and use a matrix stack in conjunction with OpenGL primitives to render the skeleton. Finally, you will write the code to set joint transforms based on joint angles passed in from the user interface.

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 parse a mesh without normals and generate them at display time. The system already contains a parser to load *attachment weights*, which specify, for each vertex, the importance of each joint. Finally, you will implement the actual SSD algorithm which requires applying a number of operations to your transformation hierarchy, and finally draw the mesh with the adjusted weights.

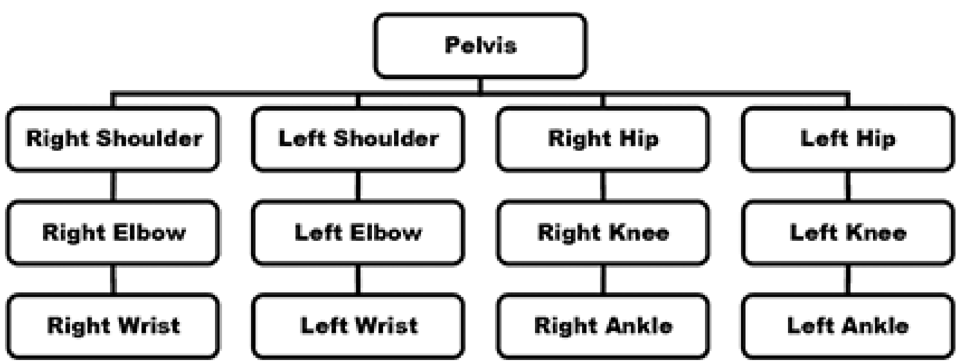
2.3 Artifact (5% of grade)

The artifact for this assignment will be easy to create: simply take a screenshot of two of the character models together with an interesting pose and submit it in PNG, JPEG, or GIF format. You can take a screenshot by taking a screenshot from your host OS or from your virtualized environment. 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. Use some of the other objects in previous assignment to make the scene more interesting.

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.

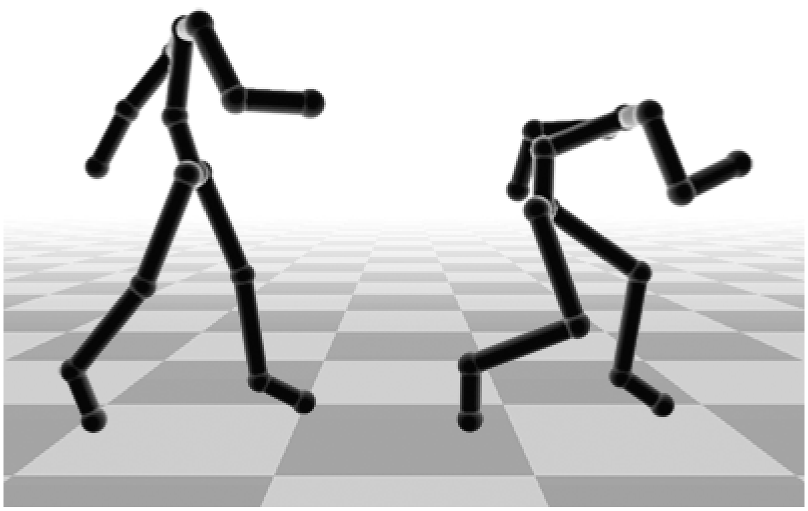
One approach to creating these animations is to individually manipulate vertices and control points. Doing so quickly becomes tedious. A better approach is to define a hierarchy such as a skeleton for a human figure and few control parameters such as joint angles of the skeleton. By manipulating these 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 joint 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 joint. 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*. A matrix stack keeps track of the current transformation (encoded in a matrix) that is applied to geometry when it is rendered. It is stored in a stack to allow you to keep track of a hierarchy of coordinate frames that are defined relative to one another – e.g., the foot's coordinate frame is defined relative to the leg's coordinate frame.

OpenGL provides a framework for maintaining a matrix stack. However, in this assignment you will be building your own, and not using OpenGL's. 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.

In your implementation, if no current transformation is applied to the stack, then it should return the identity. Each matrix transformation pushed to the stack should be multiplied by the previous transformation. This puts you in the correct coordinate space with respect to its parent. 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.

When creating your joints, you will be pushing and popping matrices onto and off the stack, and then storing the current state of the matrix with the joint. You will be using these stack functions to build your bind pose, and well as storing the current state of the joints. With every change of your joint position, you will need to rebuilding your stack and store the values with your joints.

3.2 Hierarchical Skeletons (30% of grade)

3.2.1 File Input

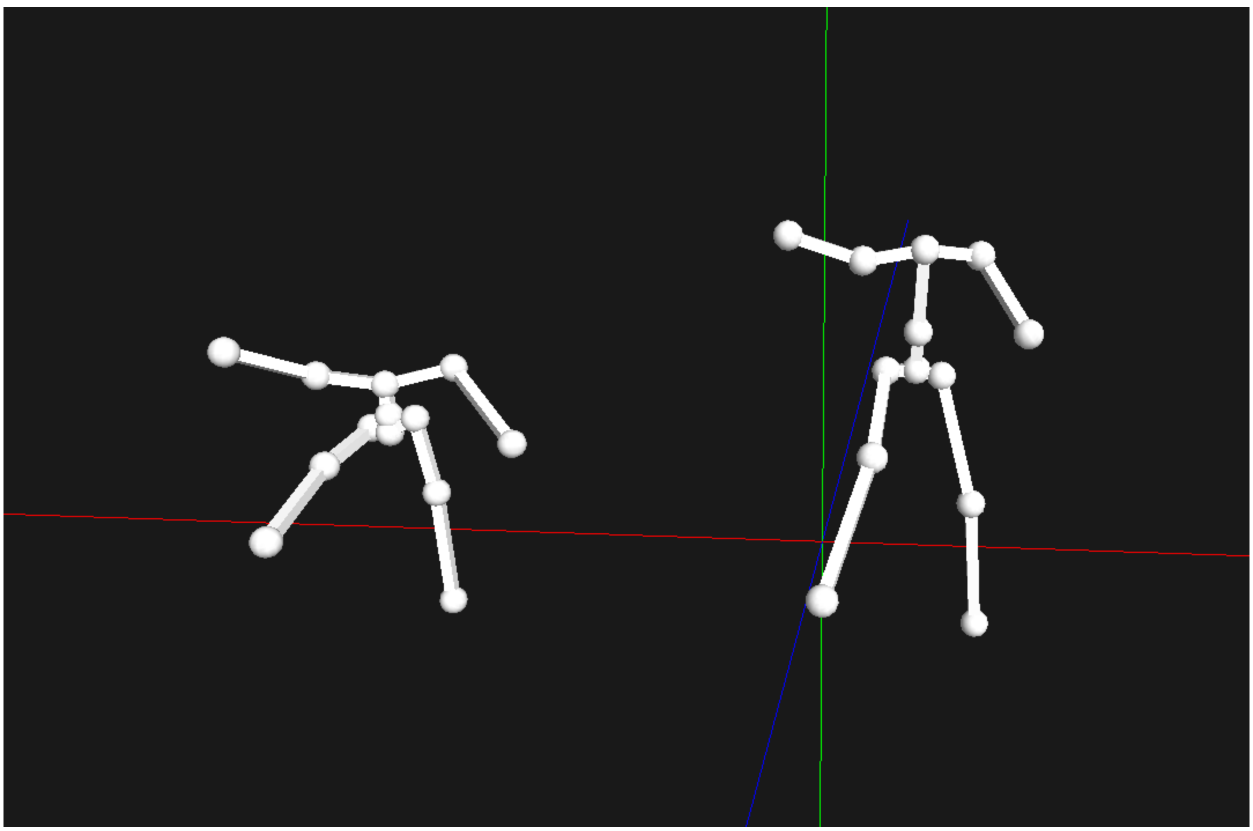
Your next task to parse a skeleton that has been built for you. The starter code automatically calls the method contained in **`FileImporter`** with the right filename (found in **`SkeletalModel.cpp`**). The skeleton file format (**`.skel`**) is straightforward. It contains a number of lines of text, each with 4 fields separated by a space. The first 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 (where a joint's index is the zero-based order it occurs in the **`.skel`** file), 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.

Each line of the **`.skel`** file refers to a joint, which you should load as a pointer to a new instance of the **Joint** class. You can initialize a new joint by calling:

```
Joint *joint = new Joint;
```

Because Joint is a pointer, note that we must initialize it with the 'new' keyword to allocate space in memory for this object that will persist after the function ends. (If you try to create a pointer to a local variable, when the local variable goes out of scope the pointer will become invalid, and attempting to access it will cause a crash.) Also note that when dealing with a pointer to an object, you must access the member variables of the object with the arrow operator -> instead of . (e.g., **`joint->transform`**), which reflects the fact that there is a memory lookup involved.

The implementation of **`ImportCharacter`** must create a hierarchy of Joints, where each Joint maintains a list of pointers to Joints that are its children. You must also populate a list of all Joints **`m_joints`** in the **`SkeletalModel`** and set **`m_rootJoint`** to point to the root **`Joint`**.



3.2.2 Drawing Stick Figures

To ensure that your skeleton was loaded correctly, we will draw simple stick figures like the ones above. To keep it simple, we will be using points for our joints and lines for our bones.

Joints: We will first draw a point at each joint to see the general shape of the skeleton. The starter code calls `ImportCharacter::setupJointBuffer`. Your task is to setup your VAO with the vertices needed to populate the joints of the character. To do this, you will use your **MatrixStack** to build your Joint transform and store them into your Joints data structure in your **ImportCharacter** shape. Once you've created your Joint transforms , use `getJoints()` to traverse through each joint, get the current Joint-to-World transform for the joint, and build your vertex data that will be used to build your **jointVAO**. Once your buffers are built, bind your joint buffer using `glBindVertexArray(jointVAO)`, use `glDrawArrays(GL_POINTS, … )` to draws the joints as points (each joint is a point in space), and then unbind your **jointVAO** using `glBindVertexArray(0)`. Additionally you can adjust the size of your points using `glPointSize(8.0f).` Change the argument to any size you like.

You can also add other commands to help with the look of your points (joints.) Explore the different OpenGL commands available to you such as:

```
glEnable(GL_PROGRAM_POINT_SIZE);
glEnable(GL_POINT_SMOOTH);
```

You must use your matrix stack to perform the transformations. You will receive no credit if you use the OpenGL matrix stack. To do this, you must push the joint's transform onto the stack using `matrixStack.push()`, load the transform by calling `matrixStack.top()`, recursively draw any of its children joints, and then pop it off the stack using `matrixStack.pop()`. You may find it helpful to verify your rendering by comparing it to the sample solution. Similarly, you can use the transforms you will build for SSD described below to traverse your set of joints to set your transforms prior to pulling them from each joint.

Alternately, you can also build your joints using spheres. Use your knowledge of building spheres in your previous assignments and incorporate this into your `setupJointBuffers()` to build spheres instead of points.

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 `SkeletalModel::setupBoneBuffers`. As with joints, it is up to you to define a separate function that will traverse the joint hierarchy. At each joint, you should draw a line between the joint and the joint's parent (unless it is the root node). Alternately, you can draw a box between the joint and the joint's parent that orientates correctly. Of course, this may take a bit more coding but may provide you an opportunity to create your skeleton closely reflecting the solution version of the assignment.

Unfortunately, there is no automated modern OpenGL's functions to build cubes, so you would have to build one yourself similar to your previous assignment. Start with a cube and then scaling it along the axis between the two joints. therefore, we recommend the following strategy. Start with a cube with side length 0.5. 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 $[-0.025, -0.025, 0]^T$ to $[0.025, 0.025, \lambda]^T$ where λ is the distance to the next joint in your recursion. Finally, you need to rotate the *z*-axis so that it is aligned with the direction to the parent joint: *z = parentOffset.normalized()*. Since the *x* and *y* axes are arbitrary, we recommend mapping *y = (z × rnd).normalized()*, and *x = (y × z).normalized()*, with *rnd* supplied as $[0, 0, 1]^T$.

For the translation, scaling, and rotation of the box primitive, you must push the transforms onto the stack before drawing the cube, but you must pop it off before drawing any of its children, as these transformations are not part of the skeleton hierarchy. As with the joints, you should verify the correctness of your implementation with the sample solution.

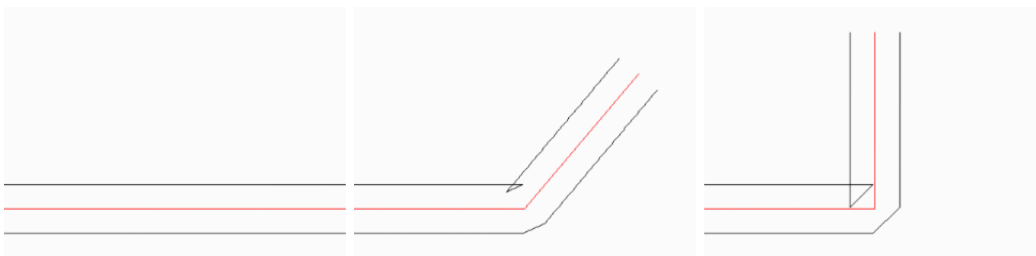
3.3 User Interface (5% of grade)

Whenever a joint rotation slider is dragged, the application calls **`SkeletalModel::setJointTransform`**, passing in the index of the joint to be updated and the Euler rotation angles set by the user. You should implement this function to set rotation component of the joint's transformation matrix appropriately.

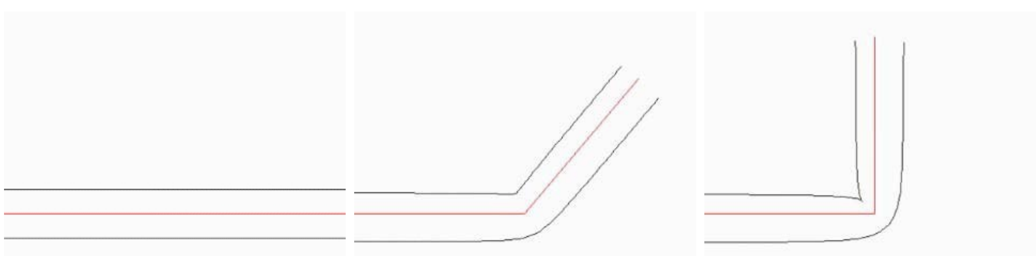
4 Skeletal Subspace Deformation

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 one joint — 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 joint. A vertex with a weight of one for a given joint will follow that joint rigidly, as it did in the previous setup. A vertex with a weight of zero for a joint is completely unaffected by that joint. Vertices in between are blended—we compute their position as if they were rigidly attached to each joint, 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 joint. then transformed the entire object via a translation to the joint's location. Now, however, vertices don't belong to a single joint, so we can't define vertices in the local coordinate frame of the joint they belong to. Instead, we define the mesh for the entire body, and keep track of the *bind pose*—the pose of each joint 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 joint 1 and joint 2. Let's also say that the *bind pose transformation* of joint 1 (the transformation which takes us from the local coordinate frame of joint 1 before the character has been animated to the character's coordinate frame) is $\mathbf{B_1}$. Finally, the transformation from joint 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 joint 1, would be $\mathbf{T_1 B^{-1} p}$. Notice that we have to first transform the point into the local coordinate system of the joint ($\mathbf{B_1^{-1} p}$) before transforming it (remember, $\mathbf{p}$ is in the character's bind pose coordinate system). Similarly, the bind transformation of joint 2 is described by $\mathbf{B_2}$, and $\mathbf{T_2}$ describes the transformation from the unanimated local coordinate frame of joint 2 to the animated character coordinate frame. Then the vertex's position, if it were rigidly attached to joint 2, would be $\mathbf{T_2 B_2^{-1} p}$. However, say that the given vertex is near the connection of two bones corresponding to joint 1 and joint 2, and we want it to be attached to both joints. We assign each joint a weight according to how much influence that joint should have on the vertex. Weights will usually range between 0 and 1 for each joint, and the weights for all joints will usually sum to 1. We want it tied to joint 1 with a weight of w, so it is tied to joint 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 $\mathbf{B_1^{-1}}$ 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 joints, 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

To get started, we will first need to adapt your code from previous assignments to load the bind pose vertices from an OBJ file. The starter code automatically calls **`FileImporter`** with the appropriate filename. The only difference between this part and other assignments 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 the 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

Next, we will verify the correctness your mesh loader by rendering the mesh. The starter code calls **`ImportCharacter::draw`** automatically when the correct shape is loaded. Be sure to render from **`ImportCharacter.currentVertices`** and not **`ImportCharacter.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

The last thing we must load are the attachment weights. The starter code calls **`FileImporter`** automatically with the selected OBJ 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 as many fields as there are joints, minus one, separated by spaces. Each field is a floating point number that indicates how strongly the vertex is attached to the (i + 1)-th joint. The weight for the 0th joint, the root, is assumed to be zero.

Your code will populate the `attachments` field of the mesh. We structured the starter code's data structure, where **`ImportCharacter.attachments`** is a **`vector< vector< float > >`**. The inner vector contains one weight per joint, and the outer vector has size equal to the number of vertices.

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 joint transformations (once) and the animated pose joint to world transformations (every time the skeleton is changed). The starter code automatically calls **`computeBindWorldToJointTransforms`** and **`updateCurrentJointToWorldTransforms`** at the appropriate points in the code.

`computeBindWorldtoJointTransforms` should set the **`bindWorldToJointTransform`** matrix of each Joint. 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.

`updateCurrentJointToWorldTransforms` is called whenever the skeleton changes. Your implementation should update the **`currentJointToWorldTransform`** matrix of each Joint and will be very similar to your implementation of **`computeBindWorldtoJointTransforms`**. 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 world to joint transform for any Joint should be the inverse of the current joint to world transform.

Once these transforms are complete, the transforms can be stored with their associated joints to easily be retrieved for the rendering of your skeleton in the previous section.

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 straight-forward. The starter code calls **`ImportCharacter::updateMeshVertices`** whenever the sliders change. Your code should update the current position of each vertex in **`ImportCharacter.vertices`** according to the current pose of the skeleton by blending together transformations of your bind pose vertex positions in .

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.

5 Create a Scene with your Characters

5.1 Screenshot Your Two+ Posed Characters (5% of grade)

Once your code is completely working, add some characters into your scene, pose them in different ways, and then take a screenshot. If you want to make your scene more interesting, add some of the other objects you've built in the previous assignments into your code such as your curves and surfaces and your built-in and imported shapes. Develop an interesting scene so it can be showcased to the class. Be creative. For example, you can flatten a cube so it can act like a floor or walls. Looks for other imported shapes to embellish your scene. But make sure at least two of the characters are posed and showcased.

Looking forward to seeing what you build.

6 Extra Credit

6.1 Create Joints with Spheres (additional 5% of max grade)

If you want to earn additional credit, instead of creating your joints with GL_POINTS, use your knowledge of drawing meshes and draw a sphere at the center of the joint. You can retrieve the joint, apply the joint to world translation, and then using the center point of the joint use it to draw your sphere. You can reuse your sphere code from your previous assignment.

6.2 Create Bones with Cubes (additional 10% of max grade)

Additional credit can be earned by drawing a cuboid from one joint to another instead of creating your bones with GL_LINES. Start with a cube and then scaling it along the axis between the two jointsit is recommend to use the following strategy. Start with a cube with side length 0.5. 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 $[-0.025, -0.025, 0]^T$ to $[0.025, 0.025, \lambda]^T$ where λ is the distance to the next joint in your recursion. Finally, you need to rotate the *z*-axis so that it is aligned with the direction to the parent joint: *z = parentOffset.normalized()*. Since the *x* and *y* axes are arbitrary, we recommend mapping *y = (z × rnd).normalized()*, and *x = (y × z).normalized()*, with *rnd* supplied as $[0, 0, 1]^T$.

7 Submission Instructions

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

- Name of student:
- 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.
- Got any comments about this assignment that you'd like to share? Was it too long? Too hard? Were the requirements unclear? Did you have fun, or did you hate it? Did you learn something, or was it a total waste of your time? Feel free to be brutally honest; we promise we won't take it personally.

Submit the following online as a single archive (.tar.gz or .zip). It should contain:

- Your code. Enter your name in the title of the window by updating the value of **windowTitle** in **Globals.cpp**. For example:
    - std::string windowTitle = "CPSC 566 - Assignment 4 - <Your name here>";
- A compiled executable named **`editor`**.
- Any additional files necessary to run and compile your program. Please include the data files to load as well
- The **README** file.
- Your screenshot scene in PNG, JPEG, or GIF format.

- Put all files in a zip or tar.gz file.
- You should only submit one compressed file with all required files. If your submission does not include one or more of these items, you will lose substantial points.