Game Engine Development II

Week 9

Hooman Salamat

Objectives

- To become familiar with the terminology of animated skinned meshes.
- To learn the mathematics of mesh hierarchy transformations and how to traverse tree based mesh hierarchies.
- To understand the idea and mathematics of vertex blending.
- To find out how to load animation data from file.
- To discover how to implement character animation in Direct3D.

FRAME HIERARCHIES

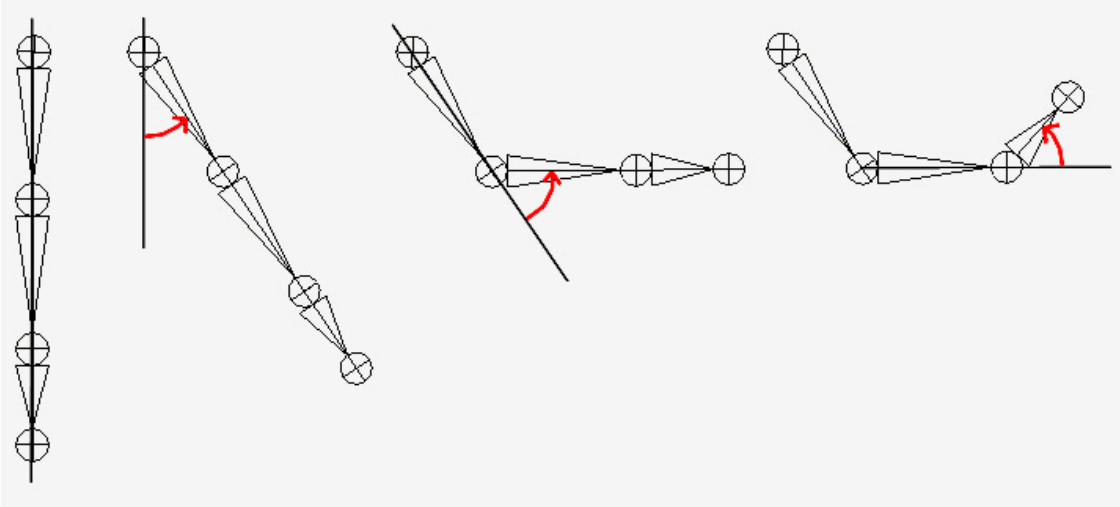
Consider an arm divided into the parts: upper arm, forearm, and hand.

The hand can rotate in isolation about its wrist joint;

However, if the forearm rotates about its elbow joint, then the hand must rotate with it.

Similarly, if the upper arm rotates about the shoulder joint, the forearm rotates with it, and if the forearm rotates, then the hand rotates with it.

we see a definite object hierarchy: The hand is a child of the forearm; the forearm is a child of the upper arm, and if we extended our situation, the upper arm would be a child of the torso.



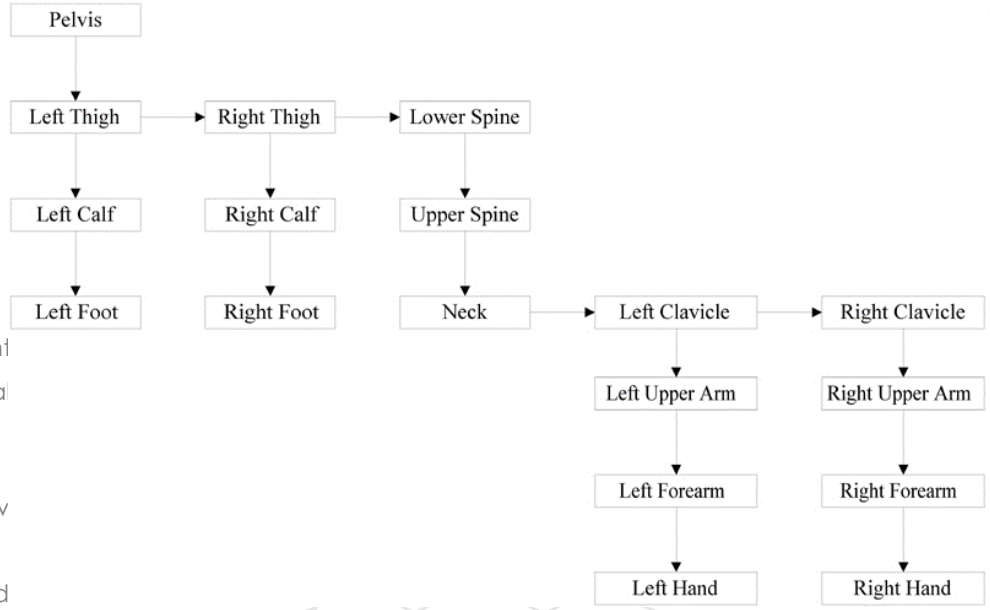
Modelling a bipedal humanoid character

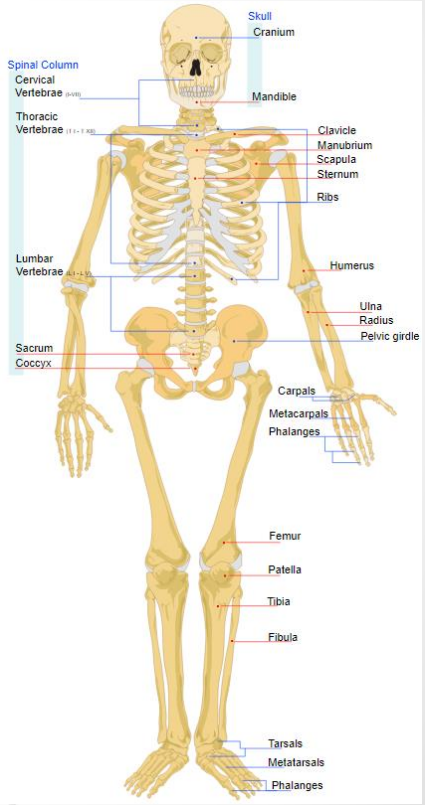
A more complex tree hierarchy to model a bipedal humanoid character.

Down arrows represent "first child" relationships, and right arrows represent "sibling" relationships.

For example, "Left Thigh," "Right Thigh," and "Lower Spine" are a children of the "Pelvis" bone.

The aim of this section is to show how to place an object in the scene based on its position, and also the position of its *ancestors* (i.e., its parent, grandparent, great grand parent, etc.)





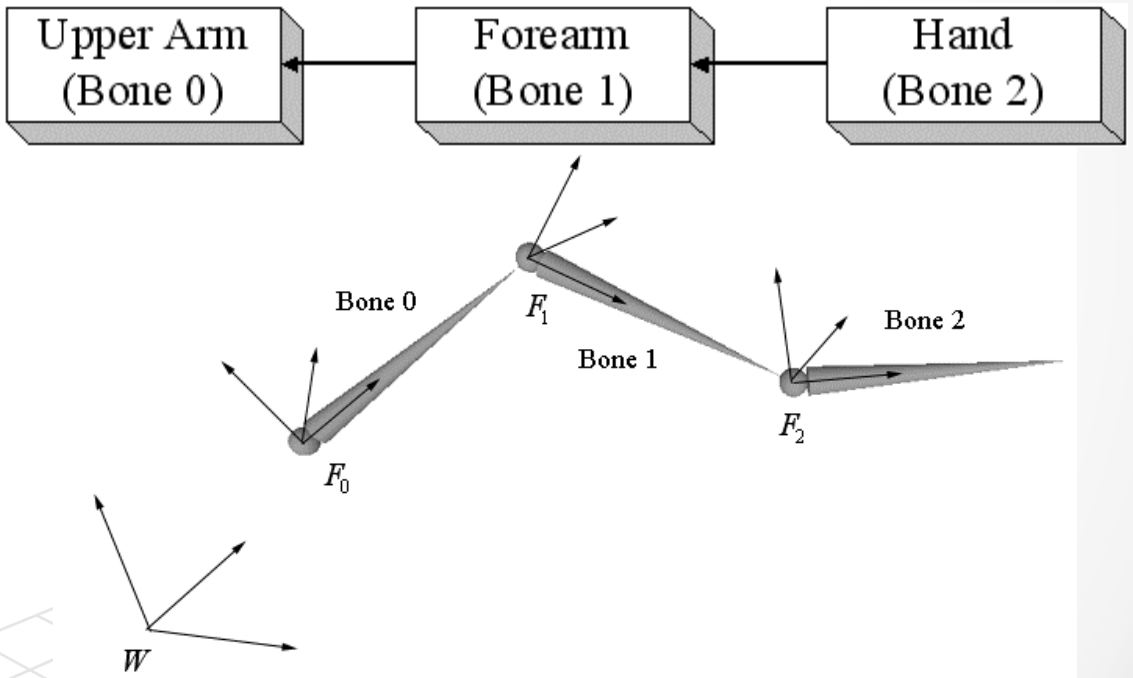
The upper arm (the root), forearm, and hand hierarchy

Given an object in the hierarchy, how do we correctly transform it to world space?

We cannot just transform it directly into the world space because we must also take into consideration the transformations of its ancestors.

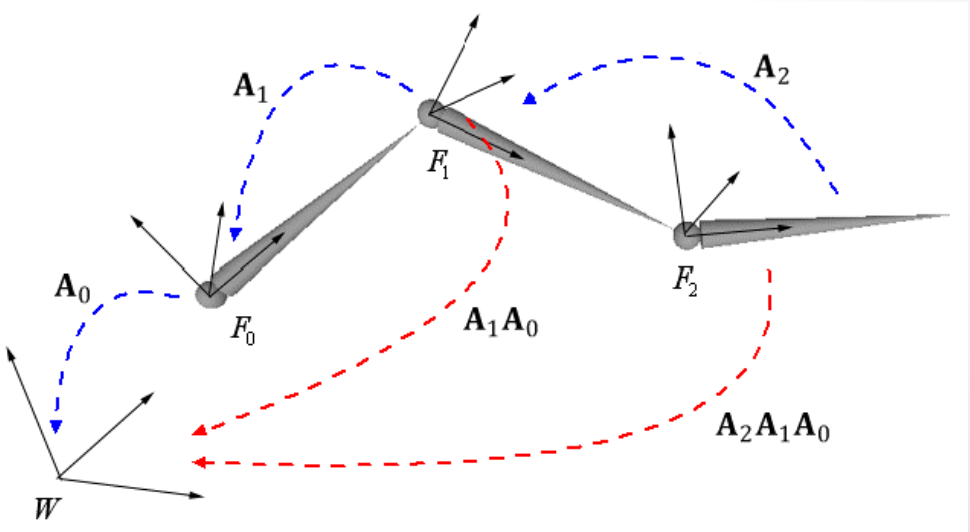
Each object in the hierarchy is modeled about its own local coordinate system with its pivot joint at the origin to facilitate rotation.

The geometry of each bone is described relative to its own local coordinate system. Furthermore, because all the coordinate systems exist in the same universe, we can relate them to one another.



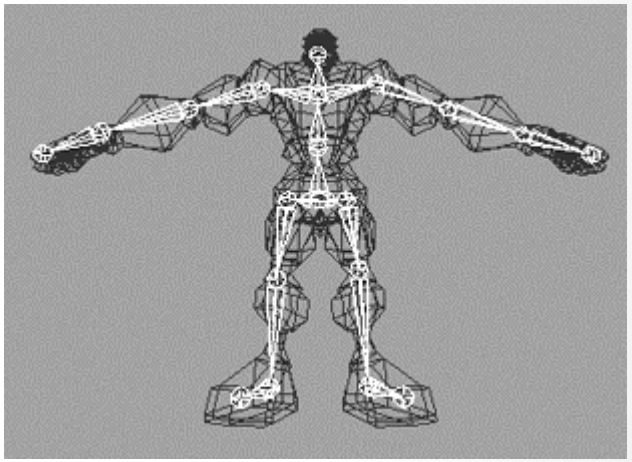
Transforming objects into world space

- we describe each coordinate system relative to its parent coordinate system.
- The parent coordinate system of the root frame *F*0 is the world space coordinate system *W*.
- We transform from the local space to the space of the parent.
- **A**2: a matrix that transforms geometry from frame *F*2 into *F*1,
- **A**1: a matrix that transform geometry from frame *F*1 into *F*0,
- **A**0: a matrix that transform geometry from frame *F*0 into *W*.
- We call **A**i a *to-parent* matrix since it transforms geometry from a child's coordinate system into its parent's coordinate system.
- Then, we can transform the *i*th object in the arm hierarchy into world space by the matrix $\mathbf{M}_i = A_i\ A_{i-1}\ A_{i-2}\ \ldots\ A_2\ A_1$
- Specifically, in our example, $\mathbf{M}_2 = \mathbf{A}_2\mathbf{A}_1\mathbf{A}_0$, $\mathbf{M}_1 = \mathbf{A}_1\mathbf{A}_0$ and $\mathbf{M}_0 = \mathbf{A}_0$ transforms the hand into world space, the forearm into world space, and the upper arm into world space, respectively.
- Essentially, to transform an object in the arm hierarchy, we just apply the to-parent transform of the object and all of its ancestors (in ascending order) to percolate up the coordinate system hierarchy until the object arrives in the world space.



SKINNED MESHES

- Figure shows a character mesh. The highlighted chain of bones in the figure is called a *skeleton*.
- A skeleton provides a natural hierarchal structure for driving a character animation system.
- The dark colored polygons represent the character's skin.
- The skeleton is surrounded by an exterior *skin*, which we model as 3D geometry (vertices and polygons).
- Initially, the skin vertices are relative to the *bind space*.
- The bind space is the local coordinate system that the entire skin is defined relative to (usually the root coordinate system).
- Each bone in the skeleton influences the shape and position of the subset of skin it influences (i.e., the vertices it influences).



Reformulating the Bones To-Root Transform

- Instead of finding the to-world matrix for each bone, we find the *to-root* (i.e., the transformation that transforms from the bone's local coordinate system to the root bone's coordinate system) matrix for each bone.
- We start at the root and move down the tree.
- Labeling the *n* bones with an integer number 0, 1, …, *n* − 1, we have the following formula for expressing the ith bone's to-root transformation:
- $toRoot_i = toParent_i . toRoot_p$
- *p* is the bone label of the parent of bone *i*.
- $\boldsymbol{toRoot_p}$ gives us a direct map that sends geometry from the coordinate system of bone *p* to the coordinate system of the root.
- Why Top-down approach is more efficient?
  - For any bone *i*, we already have the to-root transformation matrix of its parent;
  - We are only *one* step away from the to-root transformation for bone *i*.
  - With a bottom-up technique, we'd traverse the entire ancestry for each bone.
  - Many matrix multiplications would be duplicated when bones share common ancestors.

The Offset Transform

- The vertices influenced by a bone are not relative to the coordinate system of the bone.
- They are relative to the *bind space*, which is the coordinate system the mesh was modeled in.
- We first need to transform the vertices from bind space to the space of the bone that influences the vertices.
- The *offset transformation* does this.
- Once in the space of the bone, we apply the bone's to-root transformation to transform the vertices from the space of the bone to the space of the root bone.
- The final transformation is the combination of the offset transform, followed by the to-root transform.

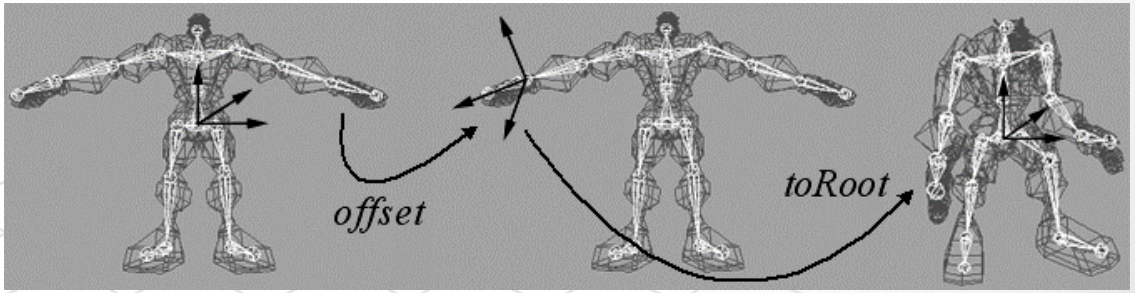
By transforming the vertices by the offset matrix of some arbitrary bone *B*, we move the vertices from the bind space to the bone space of *B*.

Then, once we have the vertices in bone space of *B*, we can use *B*'s to-root transform to position it back in character space in its current animated pose.

We now introduce a new transform, call it the *final transform*, which combines a bone's offset transform with its to-root transform.

Mathematically, the final transformation matrix of the i-th bone $\mathbf{F}_i$ is given by:

$$F_i = offset_i . toRoot_i$$



Animating the Skeleton

- We defined that a *key frame* specifies the position, orientation, and scale of an object at an instance in time
- An *animation* is a list of key frames sorted by time.
- We interpolate between key frames to calculate the placement of the object at times between key frames.
- We now extend our animation system to animating skeletons. These animation classes are defined in *SkinnedData.h/.cpp* in the "Skinned Mesh" demo
- To animate a skeleton, we just animate each bone locally.
- Then after each bone has done its local animation, we take into consideration the movement of its ancestors, and transform it to the root space.
- We define an *animation clip* to be a list of animations (one for each bone in the skeleton) that work together to form a specific animation of the skeleton. For example, "walking," "running," "fighting," "ducking," and "jumping" are examples of animation clips.

```
struct AnimationClip

{

float GetClipStartTime()const;

float GetClipEndTime()const;


    void Interpolate(float t, std::vector<DirectX::XMFLOAT4X4>&
boneTransforms)const;


    std::vector<BoneAnimation> BoneAnimations;

};
```

A data structure for storing our skeleton animation data

- A character will generally have several animation clips for all the animations the character needs to perform in the application.
- All the animation clips work on the same skeleton using the same number of bones.
- We can use an unordered_map data structure to store all the animation clips and to refer to an animation clip:

```
std::unordered_map<std::string,AnimationClip>
mAnimations;
```

```
AnimationClip& clip = mAnimations["attack"];
```

- Each bone needs an offset transform to transform the vertices from bind space to the space of the bone; and additionally, we need a way to represent the skeleton hierarchy:

```
class SkinnedData
{
public:

UINT BoneCount()const;

float GetClipStartTime(const std::string& clipName)const;
float GetClipEndTime(const std::string& clipName)const;

void Set(
std::vector<int>& boneHierarchy,
std::vector<DirectX::XMFLOAT4X4>& boneOffsets,
std::unordered_map<std::string, AnimationClip>& animations);

 // In a real project, you'd want to cache the result if there was a chance
 // that you were calling this several times with the same clipName at
 // the same timePos.
    void GetFinalTransforms(const std::string& clipName, float timePos,
 std::vector<DirectX::XMFLOAT4X4>& finalTransforms)const;

private:
    // Gives parentIndex of ith bone.
std::vector<int> mBoneHierarchy;

std::vector<DirectX::XMFLOAT4X4> mBoneOffsets;

std::unordered_map<std::string, AnimationClip> mAnimations;
};
```

Calculating the Final Transform

Our frame hierarchy for a character will generally be a tree.

We model the hierarchy with an array of integers such that the *i*th array entry gives the parent index of the *i*th bone.

The *i*th entry corresponds to the *i*th BoneAnimation in the working animation clip and the *i*th entry corresponds to the *i*th offset transform.

The root bone is always at element 0 and it has no parent.

For example, the animation and offset transform of the grandparent of bone *i* is obtained by:

```
int parentIndex = mBoneHierarchy[i];

int grandParentIndex = mBoneHierarchy[parentIndex];

XMFLOAT4X4 offset = mBoneOffsets[grandParentIndex];

AnimationClip& clip = mAnimations[“attack”];

BoneAnimation& anim = clip.BoneAnimations[grandParentIndex];
```

Compute the final transform for each bone

```
void SkinnedData::GetFinalTransforms(const std::string& clipName, float timePos,  std::vector<XMFLOAT4X4>& finalTransforms)const
{
UINT numBones = mBoneOffsets.size();

std::vector<XMFLOAT4X4> toParentTransforms(numBones);

// Interpolate all the bones of this clip at the given time instance.
auto clip = mAnimations.find(clipName);
clip->second.Interpolate(timePos, toParentTransforms);

// Traverse the hierarchy and transform all the bones to the root space.

std::vector<XMFLOAT4X4> toRootTransforms(numBones);

// The root bone has index 0.  The root bone has no parent, so its toRootTransform is just its local bone transform.
toRootTransforms[0] = toParentTransforms[0];

// Now find the toRootTransform of the children.
for(UINT i = 1; i < numBones; ++i)
{
XMMATRIX toParent = XMLoadFloat4x4(&toParentTransforms[i]);

int parentIndex = mBoneHierarchy[i];
XMMATRIX parentToRoot = XMLoadFloat4x4(&toRootTransforms[parentIndex]);

XMMATRIX toRoot = XMMatrixMultiply(toParent, parentToRoot);

XMStoreFloat4x4(&toRootTransforms[i], toRoot);
}

// Premultiply by the bone offset transform to get the final transform.
for(UINT i = 0; i < numBones; ++i)
{
XMMATRIX offset = XMLoadFloat4x4(&mBoneOffsets[i]);
XMMATRIX toRoot = XMLoadFloat4x4(&toRootTransforms[i]);
XMMATRIX finalTransform = XMMatrixMultiply(offset, toRoot);
XMStoreFloat4x4(&finalTransforms[i], XMMatrixTranspose(finalTransform));
}
}
```

Bones order

When we traverse the bones in the loop, we look up the to-root transform of the bone's parent:

```
int parentIndex = mBoneHierarchy[i];

XMMATRIX parentToRoot = XMLoadFloat4x4(&toRootTransforms[parentIndex]);
```

This only works if we are guaranteed that the parent bone's to-root transform has already been processed earlier in the loop.

Ensure that the bones are always ordered in the arrays such that a parent bone always comes before a child bone.

Here is some sample data of the first ten bones in the hierarchy array of some character model:

```
ParentIndexOfBone0: -1
ParentIndexOfBone1: 0
ParentIndexOfBone2: 0
ParentIndexOfBone3: 2
ParentIndexOfBone4: 3
ParentIndexOfBone5: 4
ParentIndexOfBone6: 5
ParentIndexOfBone7: 6
ParentIndexOfBone8: 5
ParentIndexOfBone9: 8
```

So take Bone9. Its parent is Bone8, the parent of Bone8 is Bone5, the parent of Bone5 is Bone4, the parent of Bone4 is Bone3, the parent of Bone3 is Bone2, and the parent of Bone2 is the root node Bone0.

Notice that a child bone never comes before its parent bone in the ordered array.

VERTEX BLENDING

The algorithm for animating the skin of vertices that cover the skeleton is called vertex blending.

The skin is one continuous mesh.

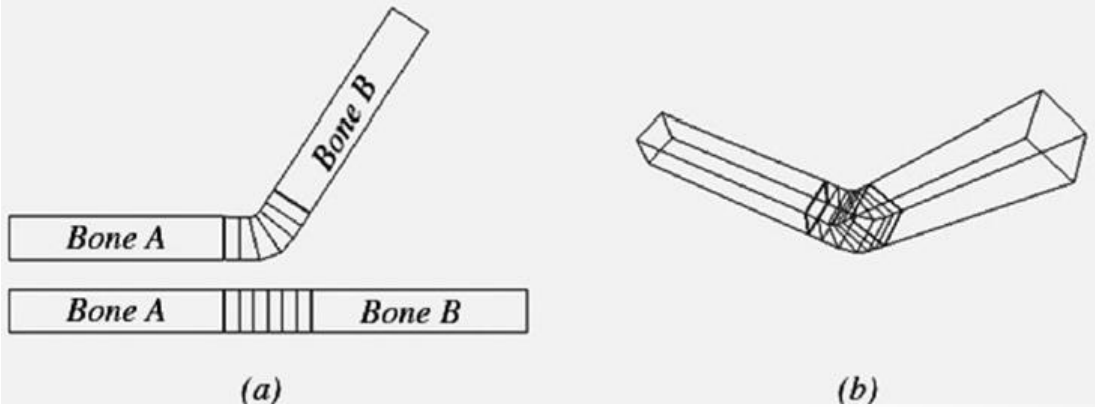
One or more bones can influence a vertex of the skin.

The net result being determined by a weighted average of the influencing bones' final transforms.

The weights are specified by an artist when the model is being made and saved to file.

With this setup, a smooth transitional blend can be achieved at joints (which are typically the troubled areas), thereby making the skin feel elastic.

Note that the vertices near the joint are influenced by both bone *A* and bone *B* to create a smooth transitional blend to simulate a flexible skin.



Vertex Blending

We will consider a maximum of four influential bones per vertex.

Each vertex contains up to four indices that index into a *bone matrix palette*, which is the array of final transformation matrices (one entry for each bone in the skeleton).

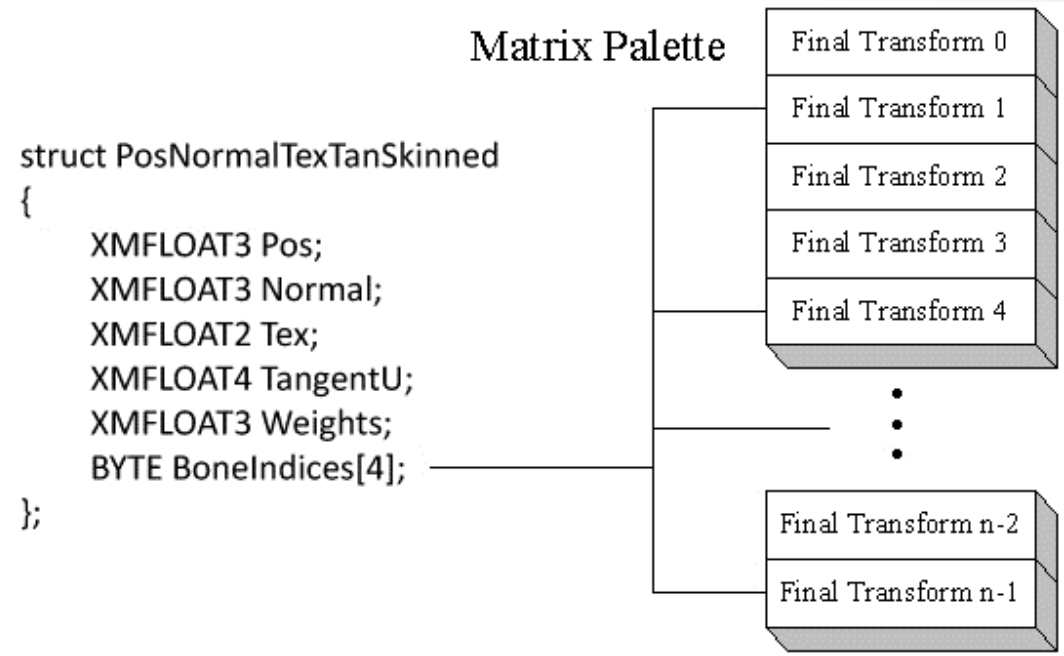
A *skinned mesh:* A continuous mesh whose vertices have this format (i.e. ready for vertex blending)

The matrix palette stores the final transformation for each bone.

The "maximum" 4 bone indices identify the bones of the skeleton that influence the vertex.

We can set a bone weight (index) to zero to effectively remove the bone from influencing the vertex.

```
struct PosNormalTexTanSkinned
{
    XMFLOAT3 Pos;
    XMFLOAT3 Normal;
    XMFLOAT2 Tex;
    XMFLOAT4 TangentU;
    XMFLOAT3 Weights;
    BYTE BoneIndices[4];
};
```



Vertex Shader

The vertex-blended position **v**' of any vertex **v**, relative to the root frame can be calculated with the following weighted average formula:

$$v' = w_0 v F_0 + w_1 v F_1 + w_2 v F_2 + w_3 v F_3$$

The sum of the weights sums to one.

We transform a given vertex v individually by all of the final bone transforms that influence it (i.e., matrices $\mathbf{F}_0$, $\mathbf{F}_1$, $\mathbf{F}_2$, $\mathbf{F}_3$).

Transforming normals and tangents are done similarly:

$$n' = normalize(w_0 n F_0 + w_1 n F_1 + w_2 n F_2 + w_3 n F_3)$$

$$t' = normalize(w_0 t F_0 + w_1 t F_1 + w_2 t F_2 + w_3 t F_3)$$

The following vertex shader fragment shows the key code that does vertex blending with a maximum of four bone influences per vertex:

```
cbuffer cbSkinned : register(b1)
{
    float4x4 gBoneTransforms[96];
};

struct VertexIn
{
float3 PosL    : POSITION;
float3 NormalL : NORMAL;
float2 TexC    : TEXCOORD;
float3 TangentL : TANGENT;
#ifdef SKINNED
    float3 BoneWeights : WEIGHTS;
    uint4 BoneIndices  : BONEINDICES;
#endif
};

struct VertexOut
{
float4 PosH    : SV_POSITION;
float4 ShadowPosH : POSITION0;
float4 SsaoPosH   : POSITION1;
float3 PosW    : POSITION2;
float3 NormalW : NORMAL;
float3 TangentW : TANGENT;
float2 TexC    : TEXCOORD;
};
```

Vertex Shader

```
VertexOut VS(VertexIn vin)
{
VertexOut vout = (VertexOut)0.0f;

// Fetch the material data.
MaterialData matData = gMaterialData[gMaterialIndex];

#ifdef SKINNED
    float weights[4] = { 0.0f, 0.0f, 0.0f, 0.0f };
    weights[0] = vin.BoneWeights.x;
    weights[1] = vin.BoneWeights.y;
    weights[2] = vin.BoneWeights.z;
    weights[3] = 1.0f - weights[0] - weights[1] -
weights[2];

    float3 posL = float3(0.0f, 0.0f, 0.0f);
    float3 normalL = float3(0.0f, 0.0f, 0.0f);
    float3 tangentL = float3(0.0f, 0.0f, 0.0f);
    for(int i = 0; i < 4; ++i)
    {

        posL += weights[i] * mul(float4(vin.PosL, 1.0f),
gBoneTransforms[vin.BoneIndices[i]]).xyz;
        normalL += weights[i] * mul(vin.NormalL,
(float3x3)gBoneTransforms[vin.BoneIndices[i]]);
        tangentL += weights[i] * mul(vin.TangentL.xyz,
(float3x3)gBoneTransforms[vin.BoneIndices[i]]);
    }
```

```
 vin.PosL = posL;
    vin.NormalL = normalL;
    vin.TangentL.xyz = tangentL;
#endif

    // Transform to world space.
    float4 posW = mul(float4(vin.PosL, 1.0f), gWorld);
    vout.PosW = posW.xyz;
     vout.NormalW = mul(vin.NormalL, (float3x3)gWorld);

     vout.TangentW = mul(vin.TangentL, (float3x3)gWorld);

    // Transform to homogeneous clip space.
    vout.PosH = mul(posW, gViewProj);

    // Generate projective tex-coords to project SSAO map onto scene.
    vout.SsaoPosH = mul(posW, gViewProjTex);

     // Output vertex attributes for interpolation across triangle.
     float4 texC = mul(float4(vin.TexC, 0.0f, 1.0f), gTexTransform);
     vout.TexC = mul(texC, matData.MatTransform).xy;

    // Generate projective tex-coords to project shadow map onto scene.
    vout.ShadowPosH = mul(posW, gShadowTransform);

    return vout;
}
```

LOADING ANIMATION DATA FROM FILE

We use a text file to store a 3D skinned mesh with animation data.

We call this an.m3d file for "model 3D."

The format has been designed for simplicity in loading and readability—not for performance.

At the beginning, the .m3d format defines a header which specifies the number of materials, vertices, triangles, bones, and animations that make up the model:

**********m3d-File-Header**********

#Materials 5

#Vertices 13748

#Triangles 22507

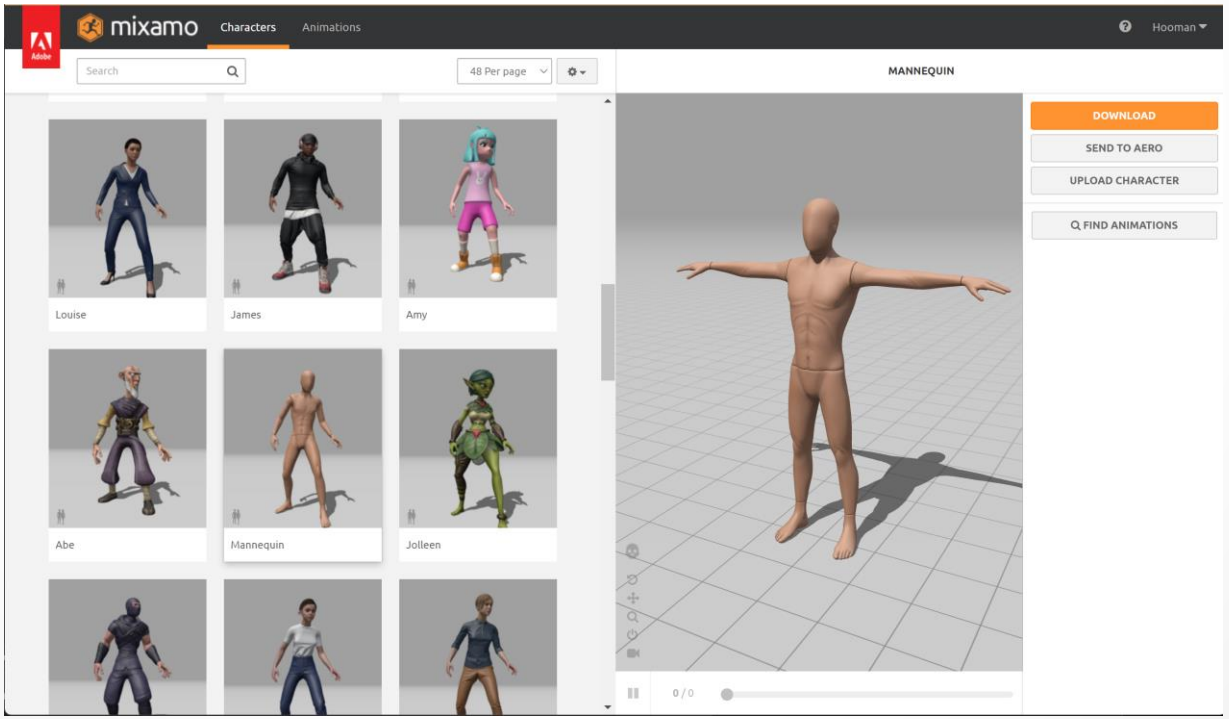
#Bones 58

#AnimationClips 1

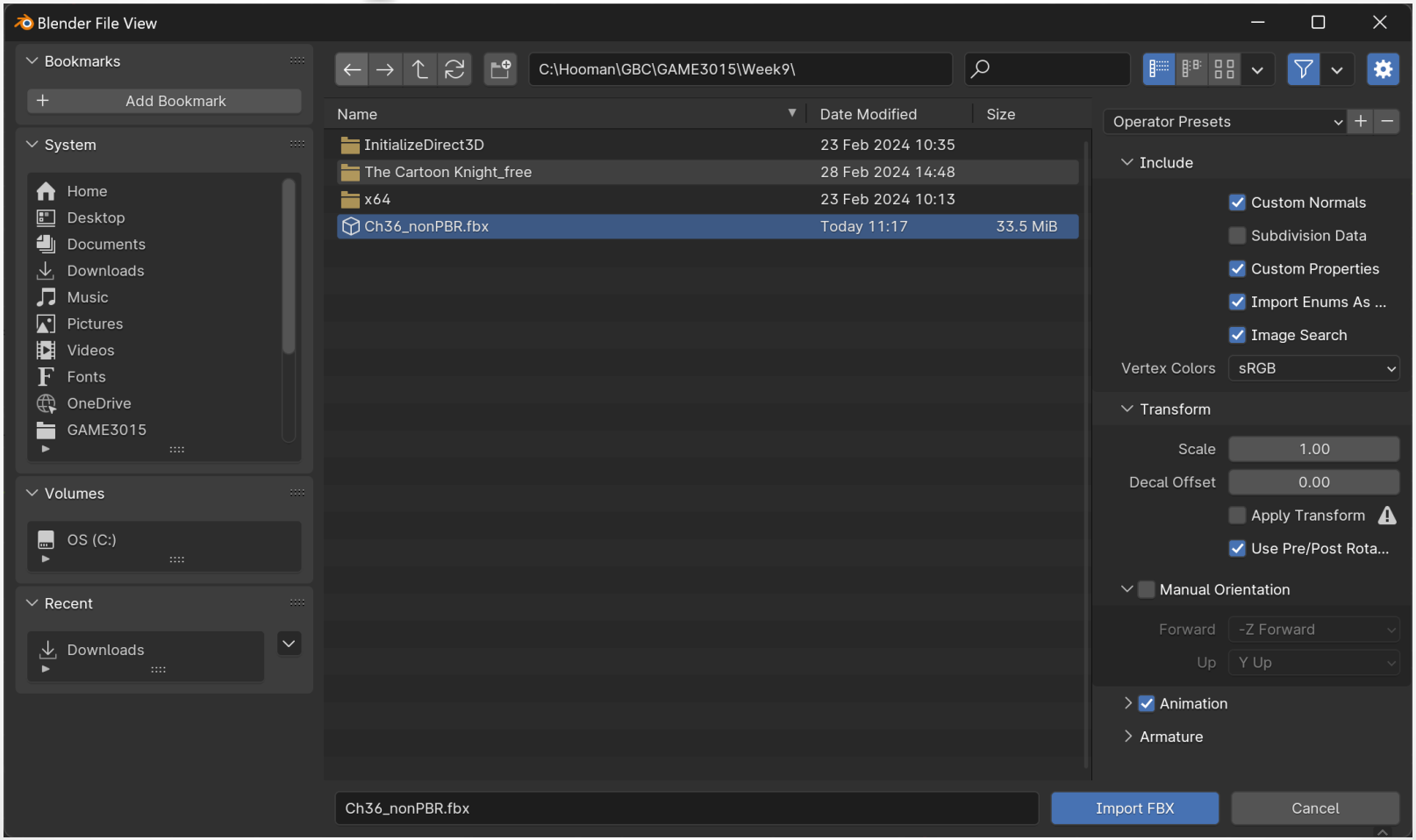
1. #Materials: The number of distinct materials the mesh uses.
2. #Vertices: The number of vertices of the mesh.
3. #Triangles: The number of triangles in the mesh.
4. #Bones: The number of bones in the mesh.
5. #AnimationClips: The number of animation clips in the mesh.

Character Rigging

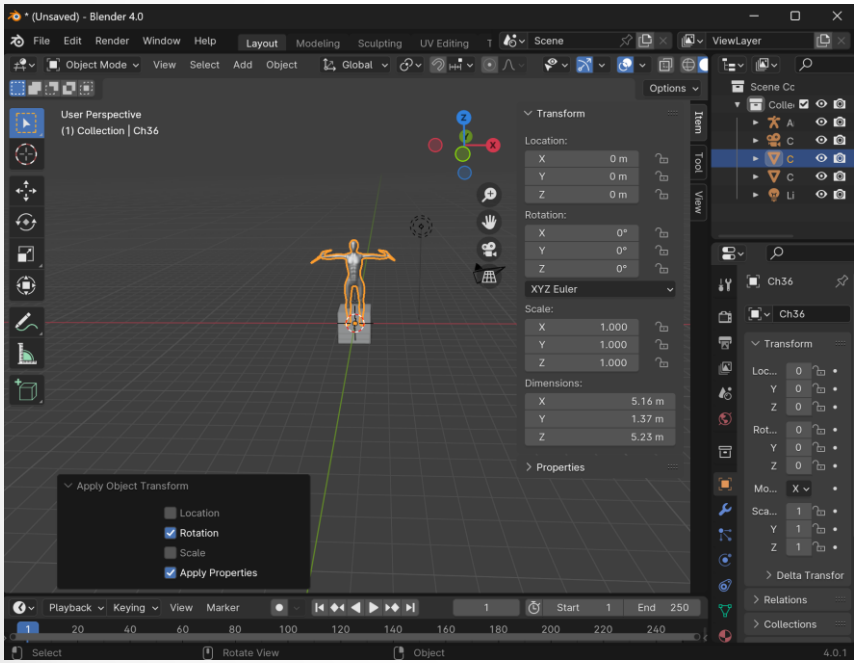
- Mixamo by Adobe
- https://www.mixamo.com/#/?page=1&query=&type=Character



Import to Blender



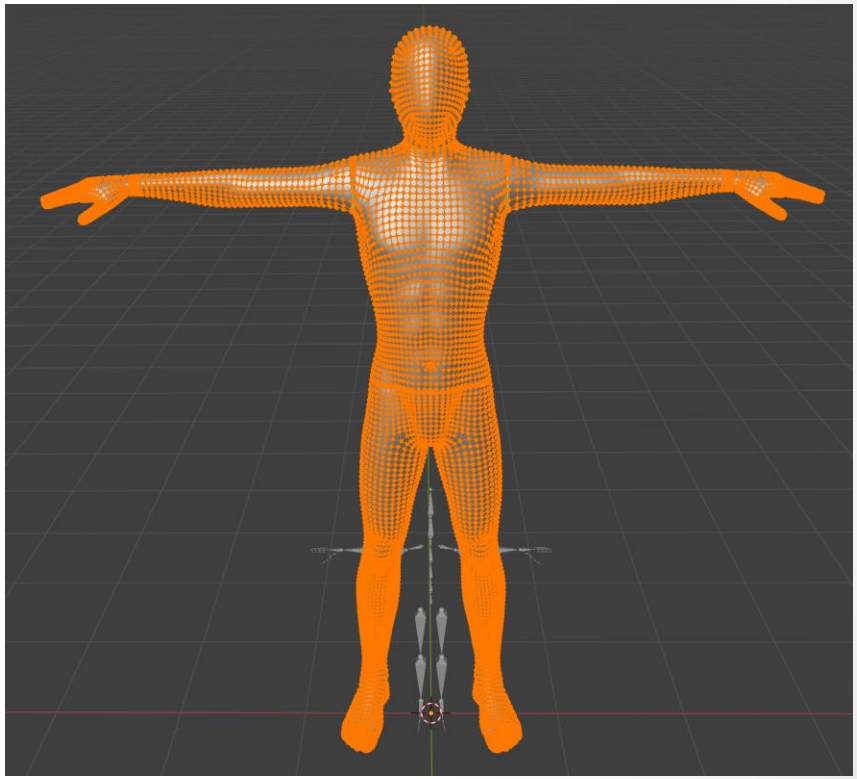
Blender



- We want to create our own poses
  - Alt+p and select "Clear and Keep the transformation"
- Select the character and press S to scale
- Ctrl+A and choose "Scale" to apply the scale
- Ctrl+A and choose "Rotation" to apply rotation
- N (to see the transformation)

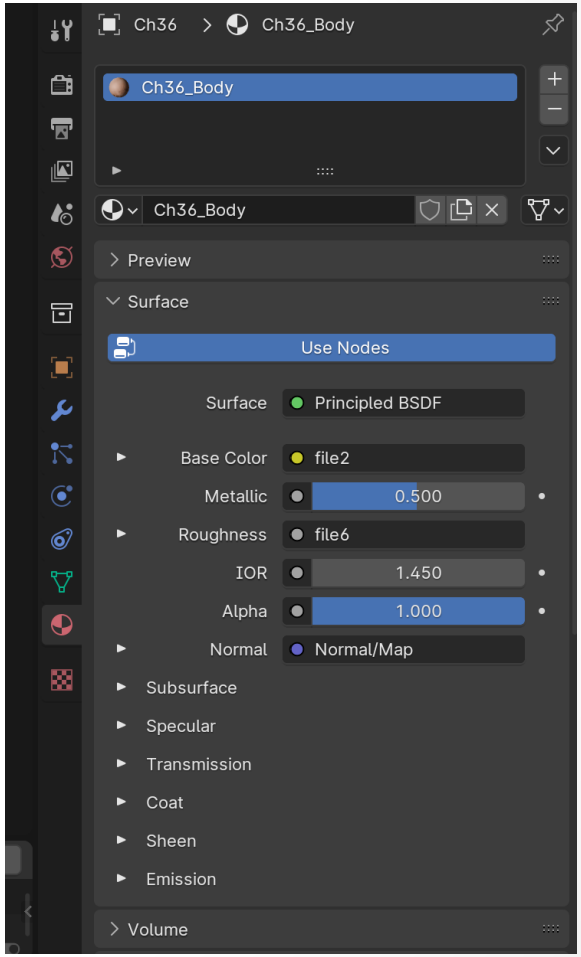
Blender

- Remove the cube and camera from the scene
- Click “Tab” to go to Edit mode
- Alt+A
- F3 and search unsub (for mesh unsubdivide)
- Press “Z” in Edit mode and keep materials



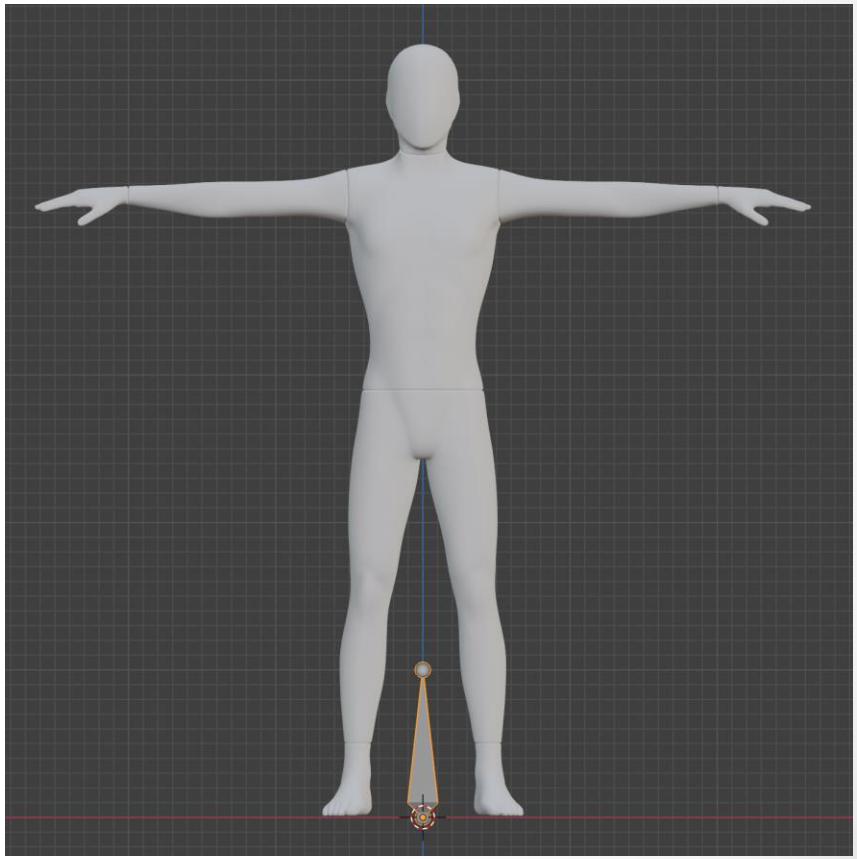
Remove the material

- Select the Material properties
- Click "-" to remove it
- Don't forget to save it!
- Make sure the cursor is in the middle, if not (shift+S)



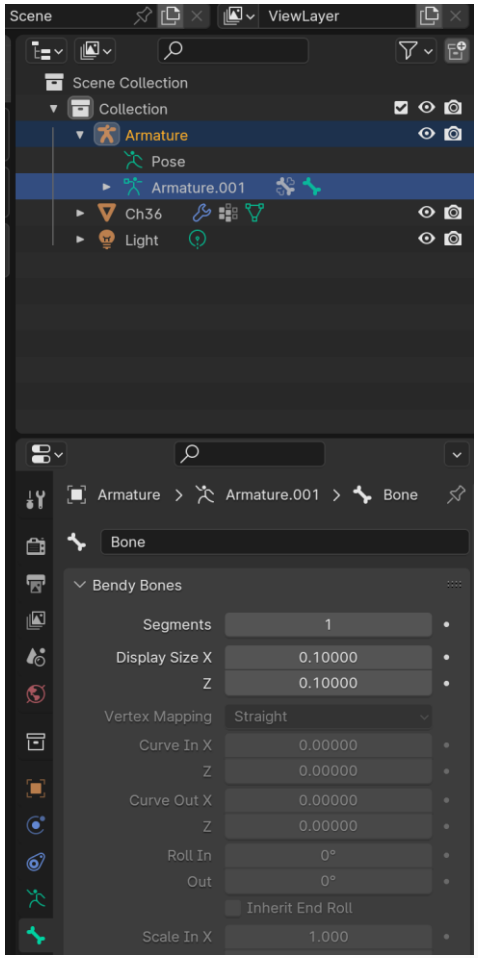
Add Armature

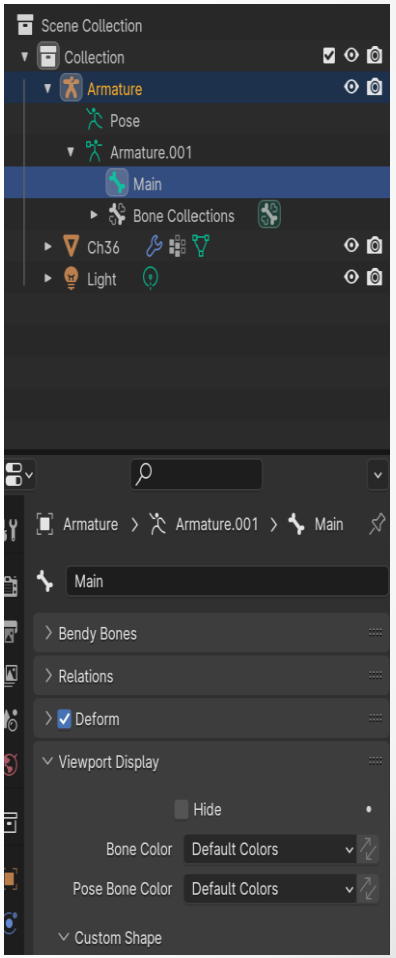
- Shift+A to add an Armature
- Armature is a collection of bones
- By default, it has one bone.



Bone

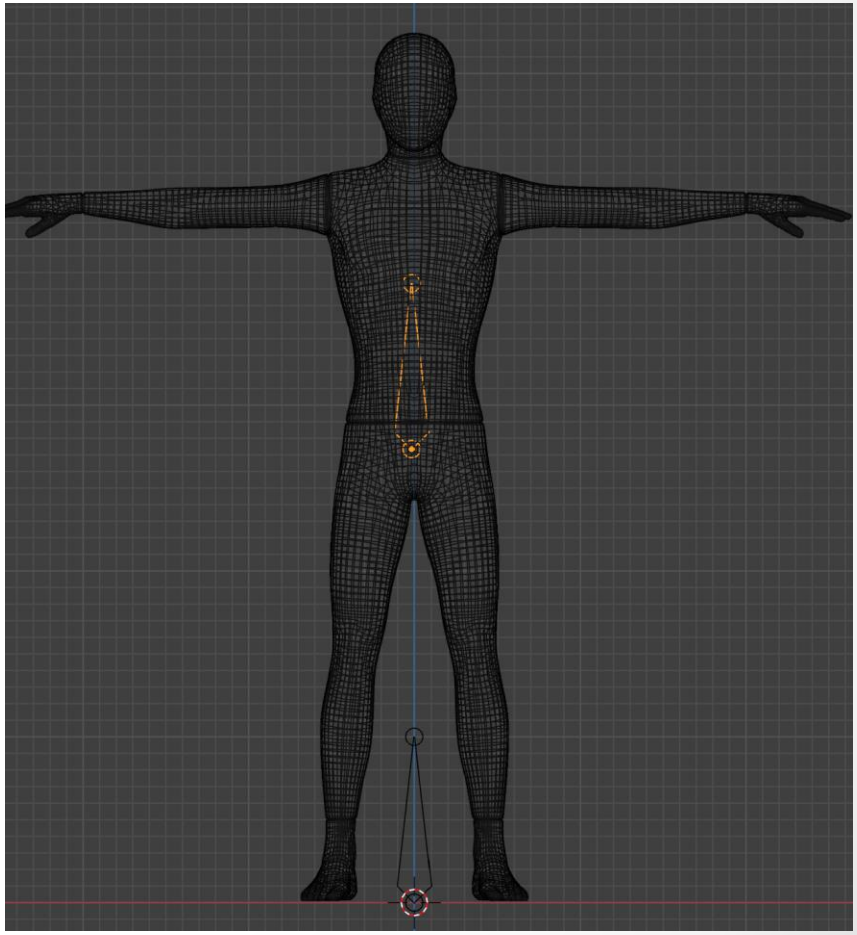
- Go to the bone properties and see the values
- Rename “Bone” to “Main”





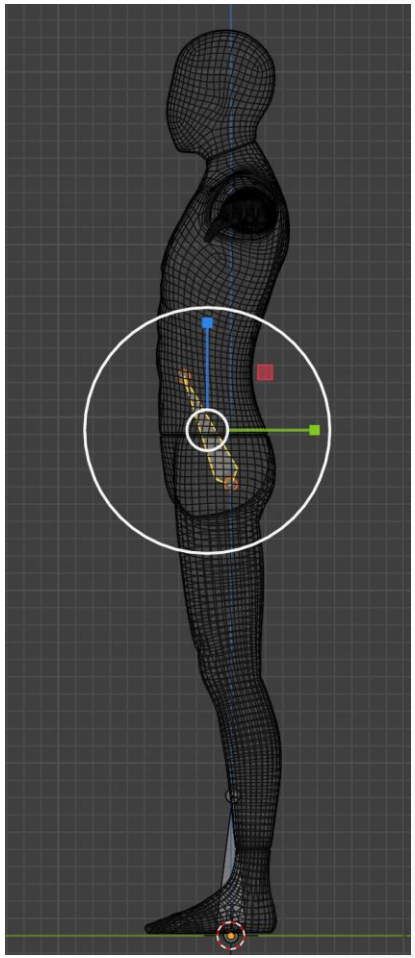
Add another bone

- Click “Tab” to be in “Edit Mode”
- Select the “Main” bone
- Duplicate the bone (Shift+D)
- Move it to the hip part of the character
- Rename it to Hip
- Click “Z” to see it in the wireframe



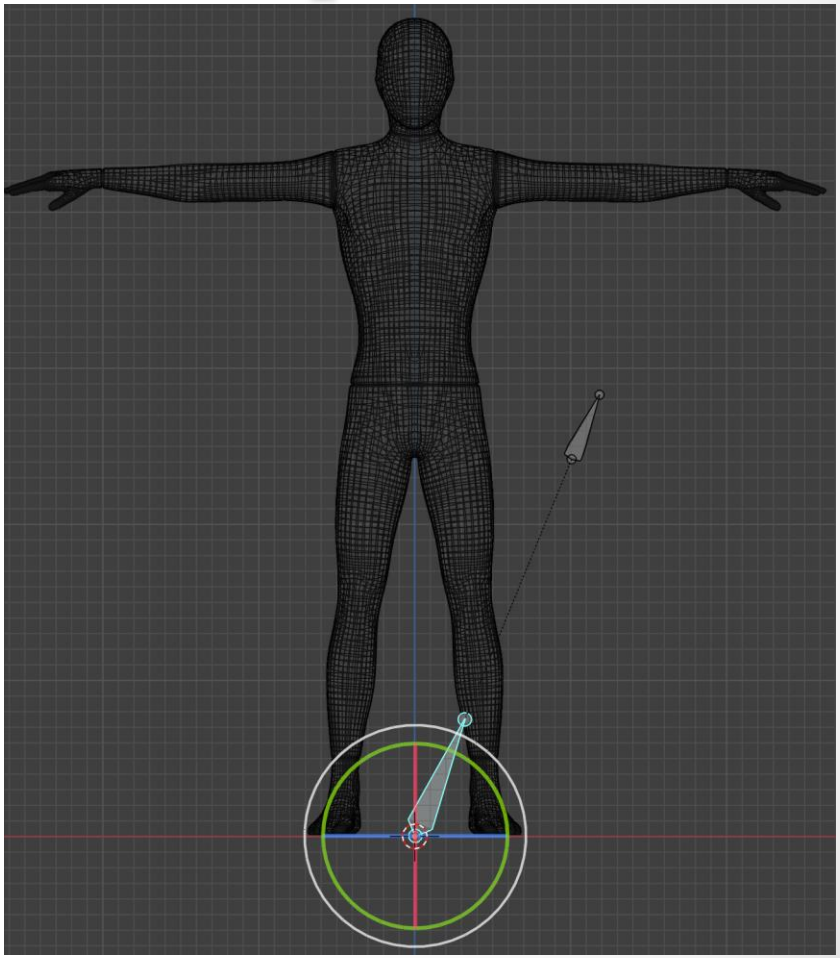
Hip

- Click on 3 key on the num pad to see the side view
- And Rotate the hip bone to move forward (Click on the top nob and press G to move it forward)



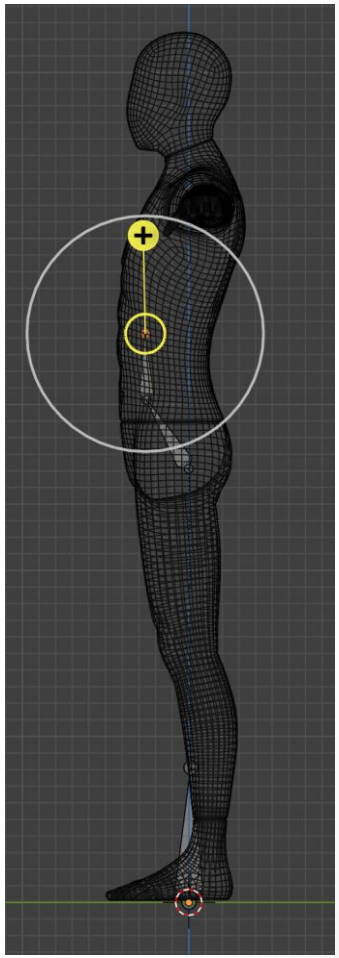
Create a parent child relationship between Main and Hip bones

- Go to "Pose" mode from "Edit" Mode
- Select the "Main" rig(bone)
- If you move the "Main" bone, hip doesn't change
- Go back to Edit mode
- Select the "Hip" bone and then hold down "Shift" and select "Main" bone
- Ctrl+p and click on "keep offset"
- Change back to "Pose" mode and if you move the "Main" bone, "Hip" will change

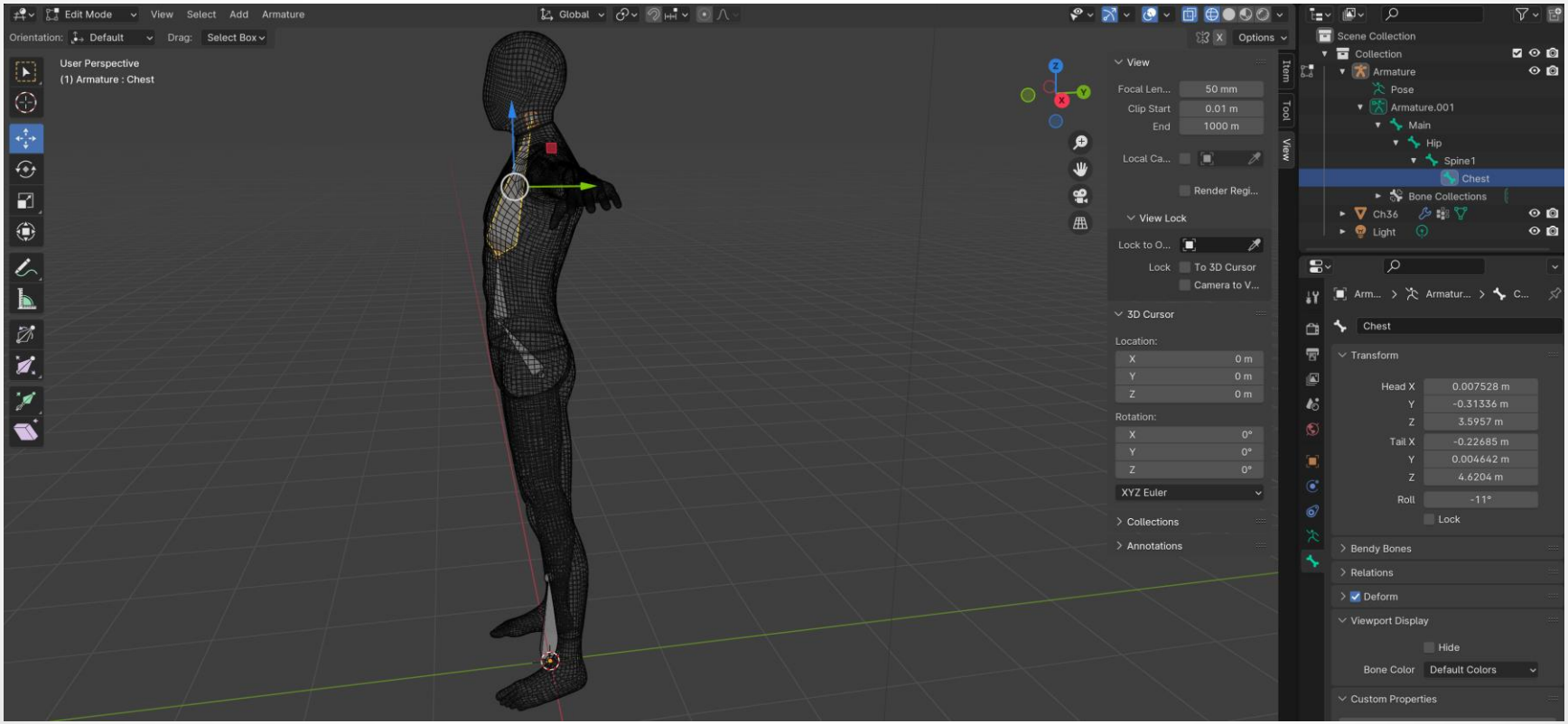


Extrude another bone from Hip Bone

- Select the knob at the top of the Hip node and click on "Extrude" from the left panel to create a new bone
- Rename it to Spine1
- Notice that Spine 1 a child of Hip bone!

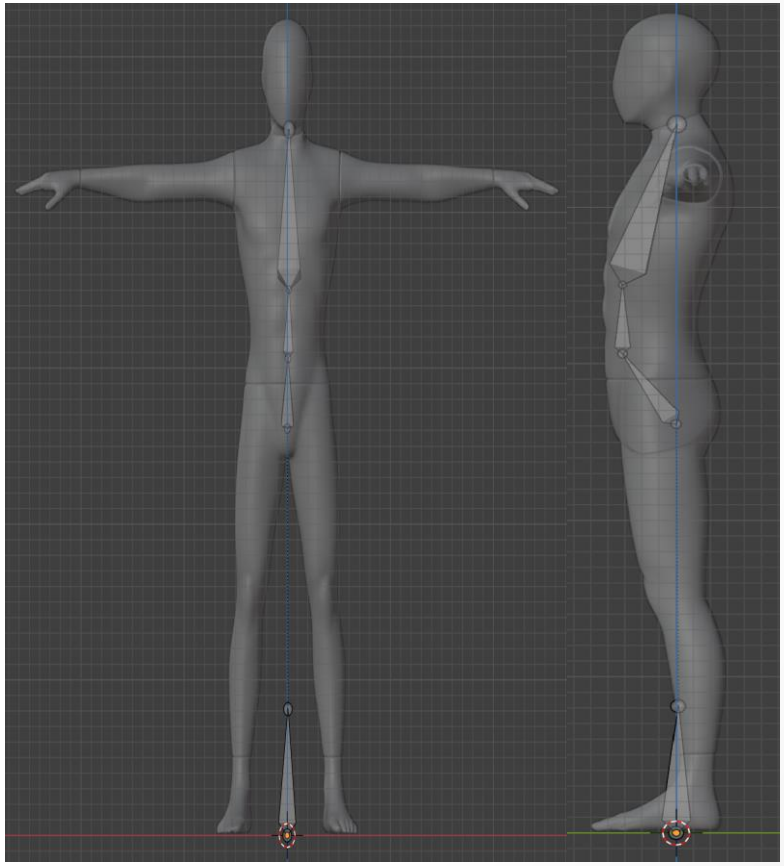


Extrude The Chest Bone

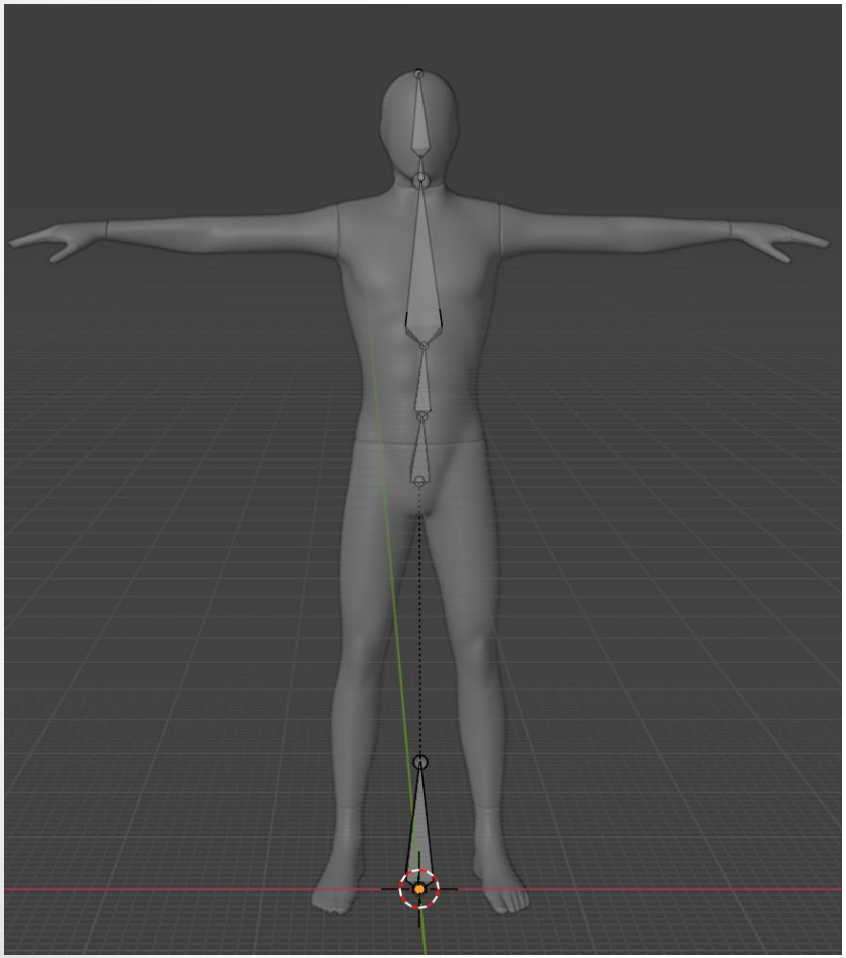


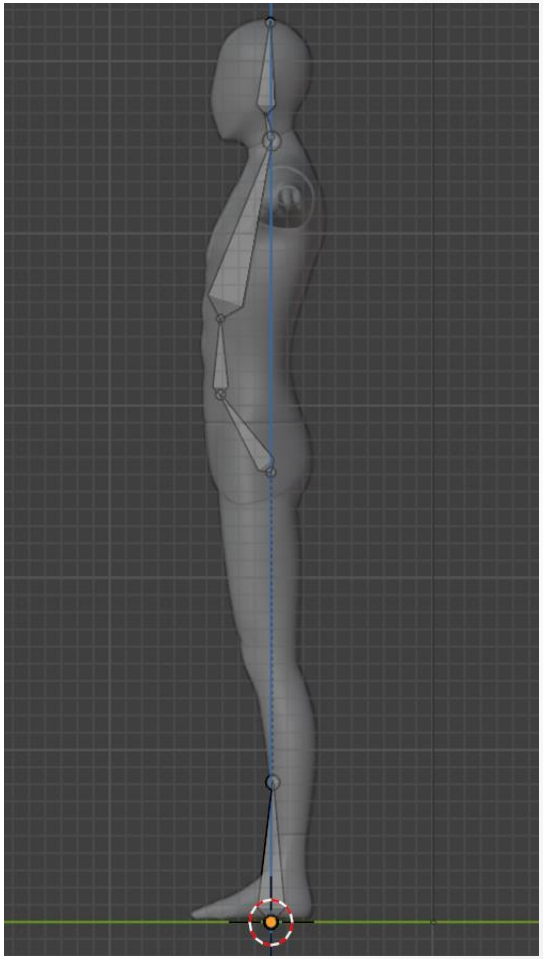
Toggle X-Ray

- Type Z in Edit mode
- Change it to Solid mode
- Click on the Toggle X-Ray icon at the top-right
- Make sure that the bones are aligned properly
- Numpad 1 and 3 keys



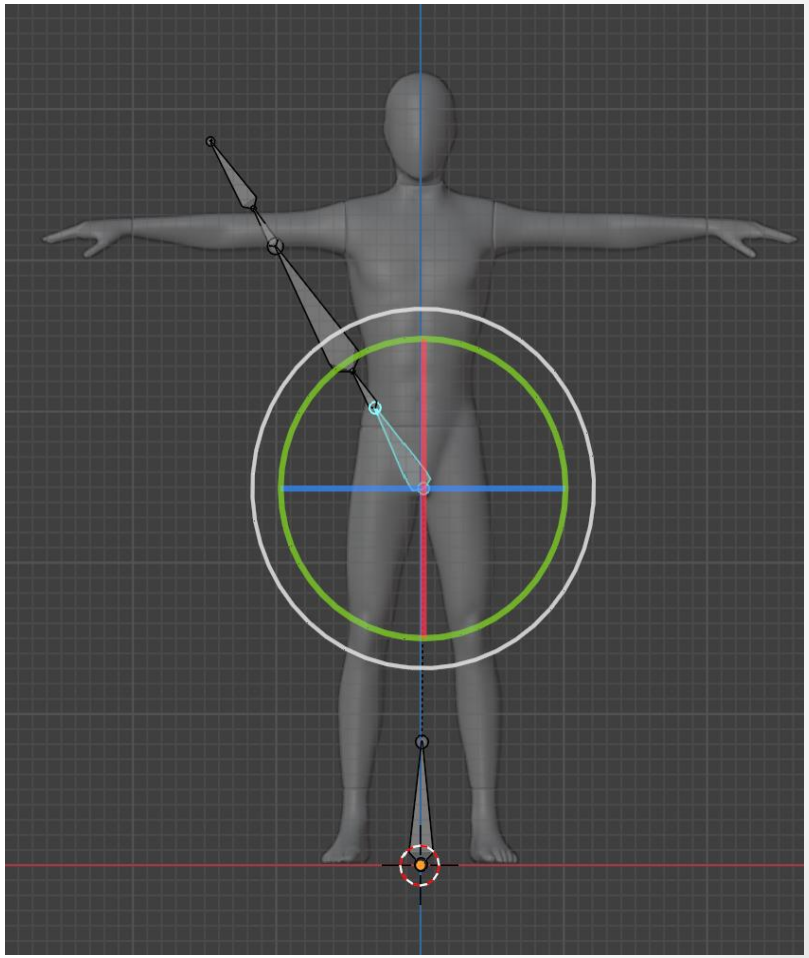
Extrude Neck and Head Bones





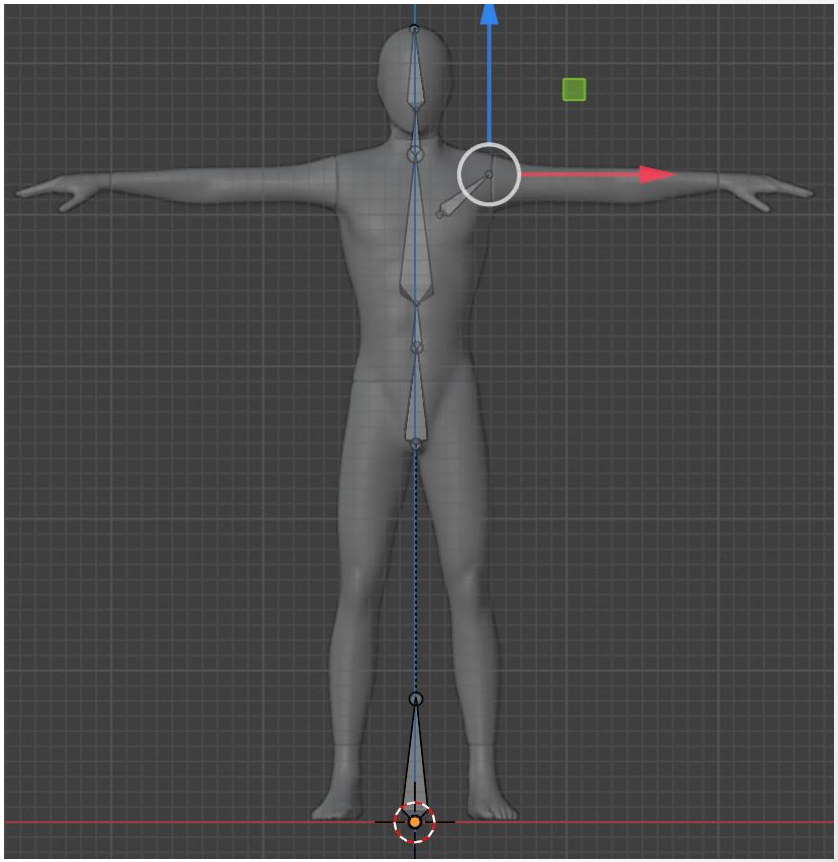
Test in Pose Mode

- Go to the "Pose Mode" and select the hip bone and press G/R and move/rotate the bone to see how parent-child bones react to the movement
- This is called Forward Kinematics because you would be working your way up to the chain of bones.
- Go back to Edit mode!

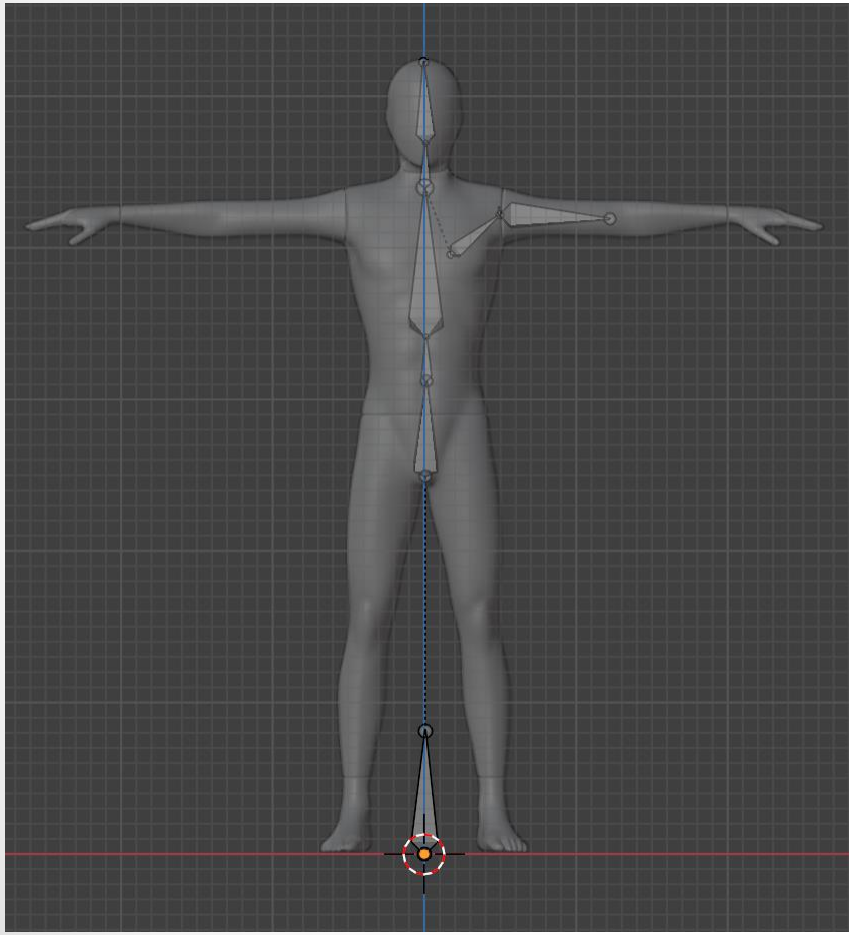


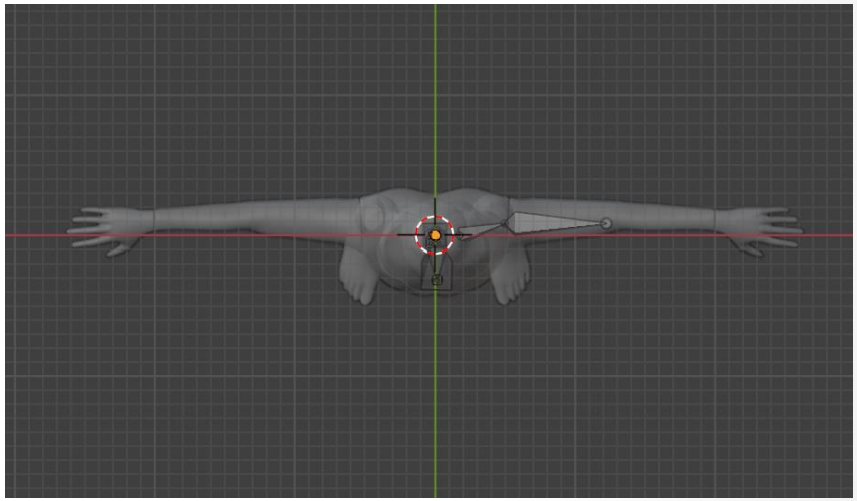
Clavicle Bone

- In Edit mode, select the "Main" bone and Shift+D to create a duplicate and move it to the right clavicle location.
- Select the Clavicle bone, then click Shift, then choose the Chest bone, then Ctrl+P, Choose "Keep offset"

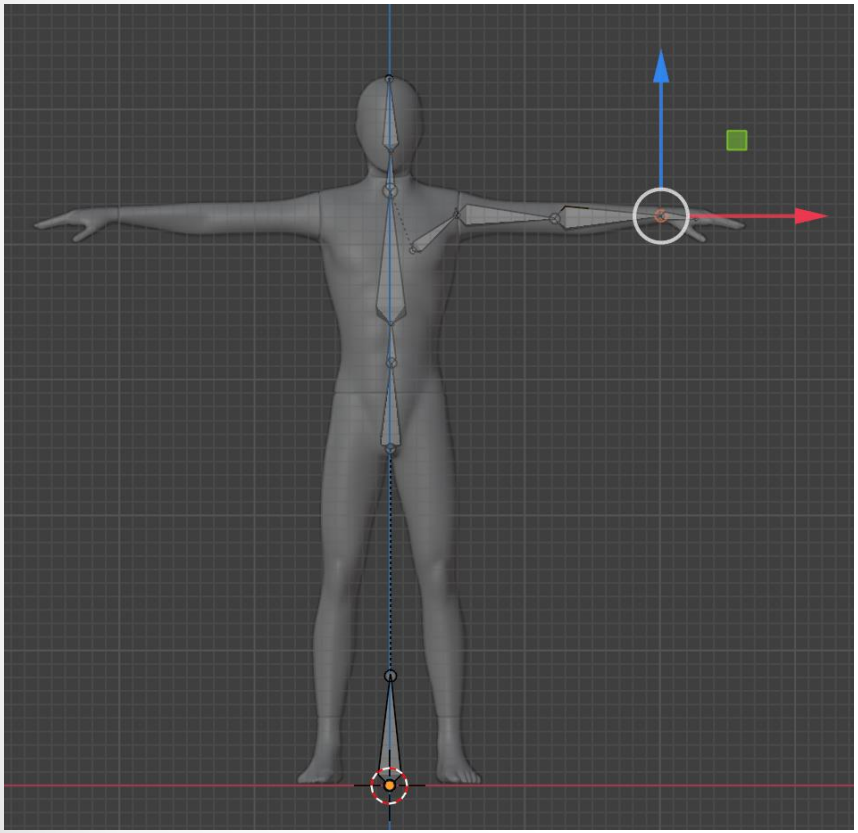


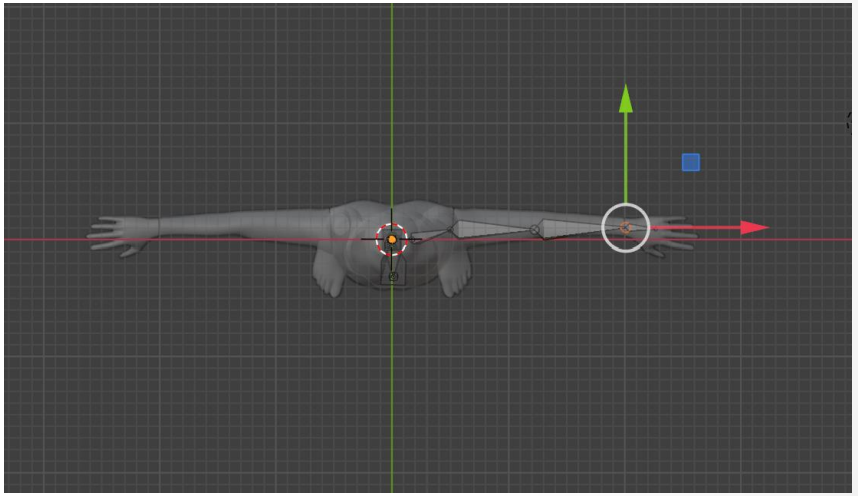
Extrude Upper Arm Bone





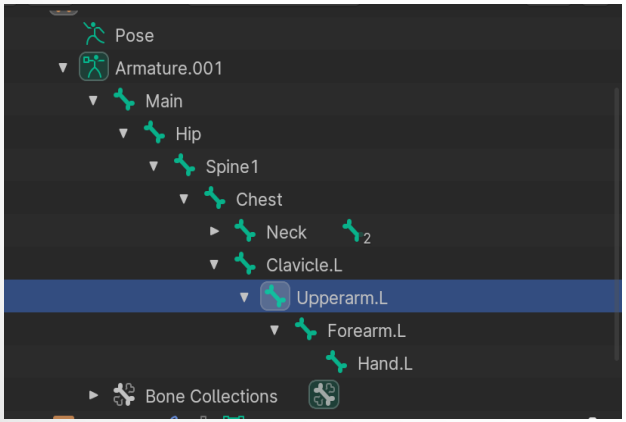
Extrude Forearm and Hand bones

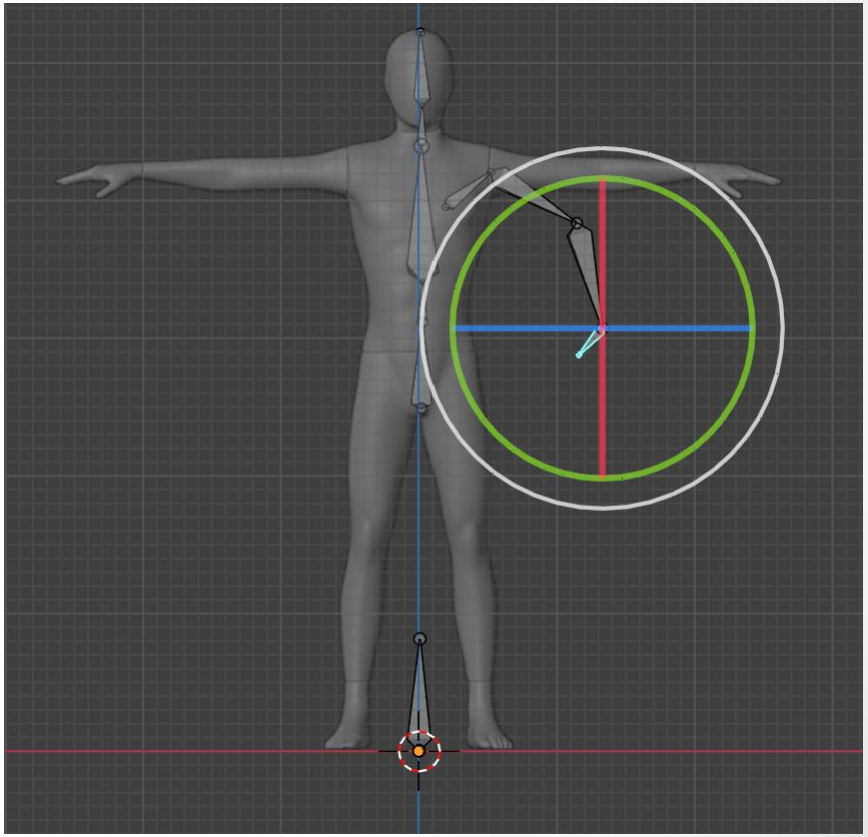




Naming Convention

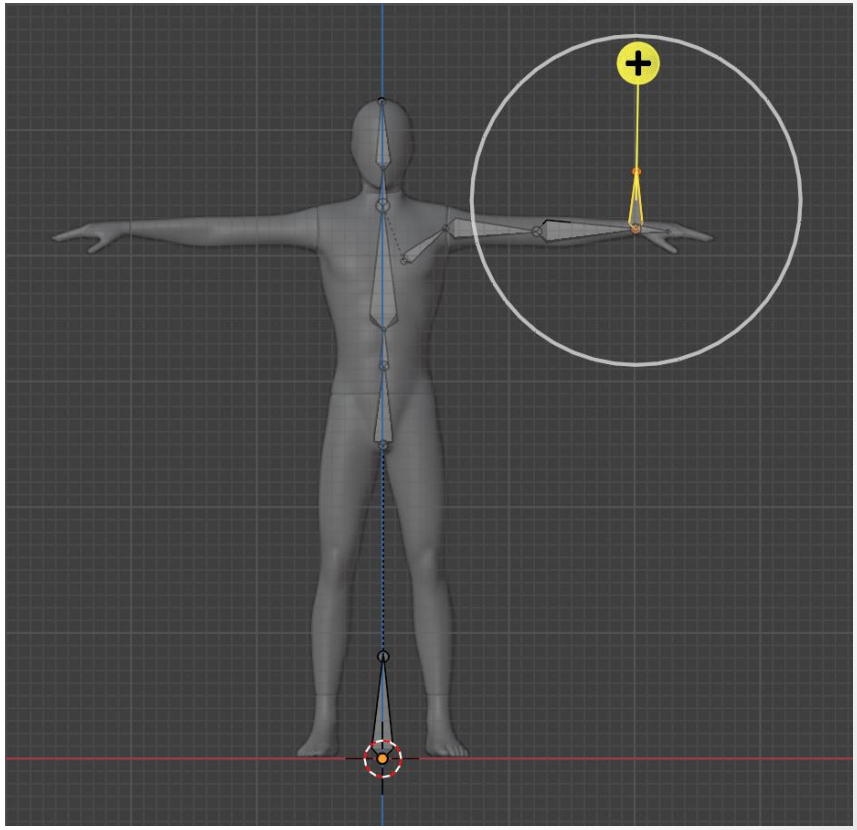
- All the left bones names must be ended with .L (upper case!)
- Go to "Pose Mode" and make sure everything is working fine.





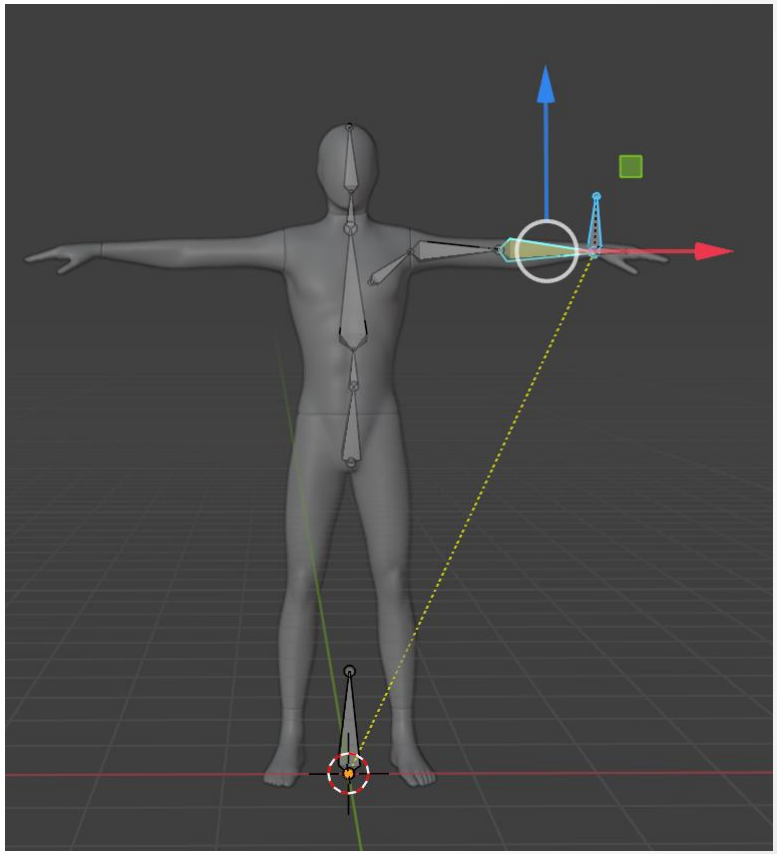
Set up an IK(Inverse Kinematics)

- Go back to Edit mode
- Select the knob behind the hand
- Click E to Extrude a vertical bone
- Alt+P
- We wanted to be a separate bone. Click on "Disconnect Bone"
- Alt+P again and click on "Clear parent" to remove any relationship to the parent node.
- Select the Hand bone and Shift select the IK bone, Ctrl+P and then select "Keep offset" to make Hand.L a child of the IK bone
- Go to "Pose Mode", G to move the IK bone, and R to rotate it. You see the hand moves accordingly!
- Notice that the IK bone is controlling the hand bone and not the arms!
- Rename the IK node to Hand_IK.L



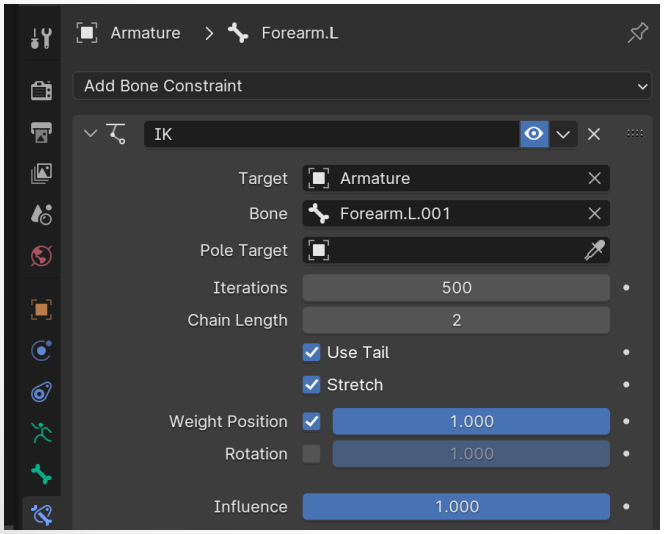
Inverse Kinematics with the Forearm

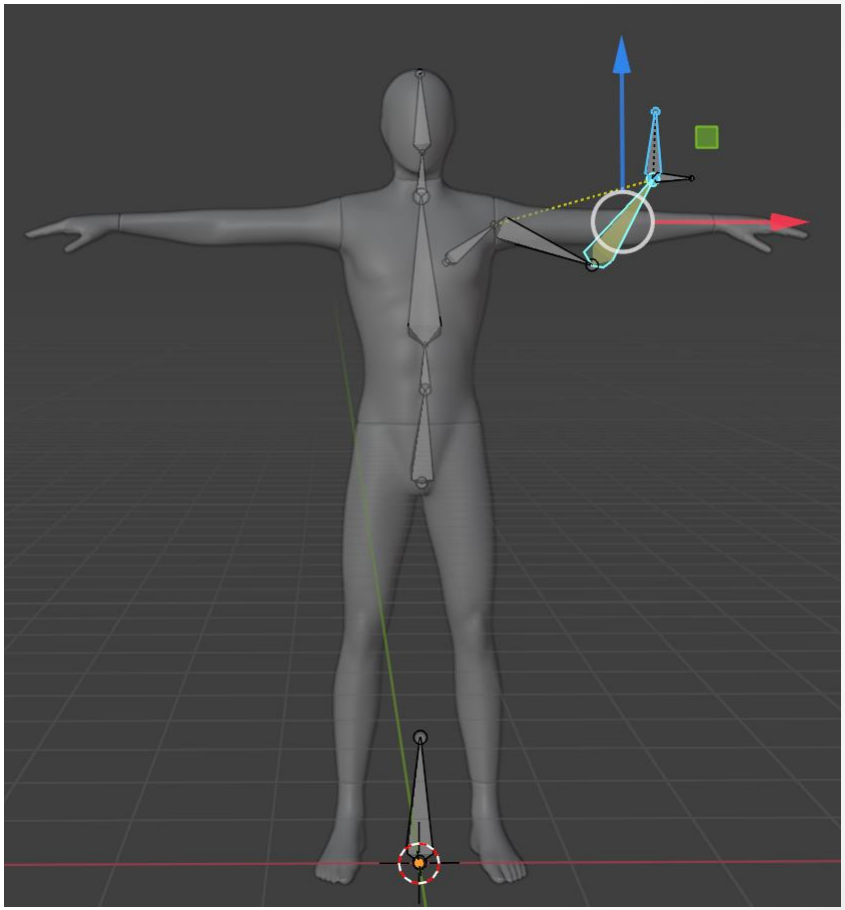
- While In Pose Mode, Select the IK bone and Shift the Forearm bone
- Ctrl+Shift+C and select Inverse Kinematics



Bone Constraints

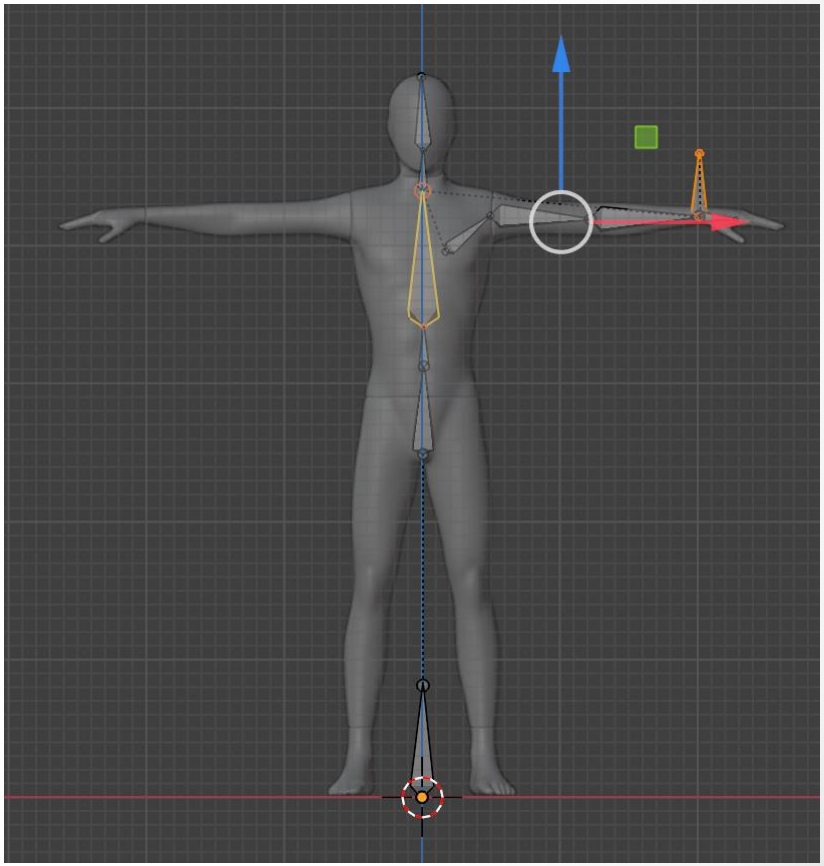
- In pose mode, Select the Forarm, and go to the bone constraints and change the "Chain Length" to 2 to affect 2 bones





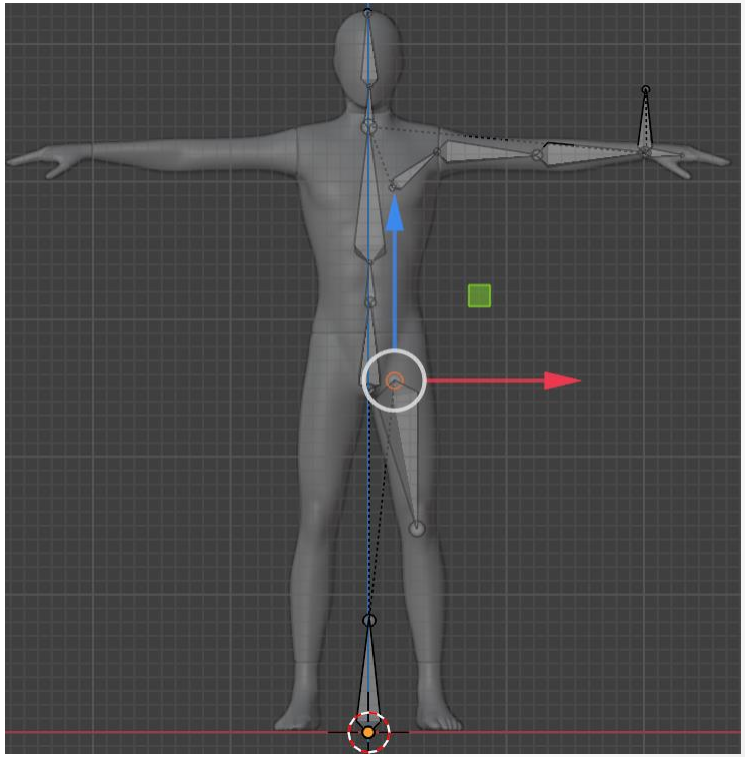
Connect IK bone to Chest

- In Edit mode, select the IK bone, then select the Chest bone, Ctrl+P, and “Keep Offset”
- Go back to “Pose” mode and select the chase and move its bone.



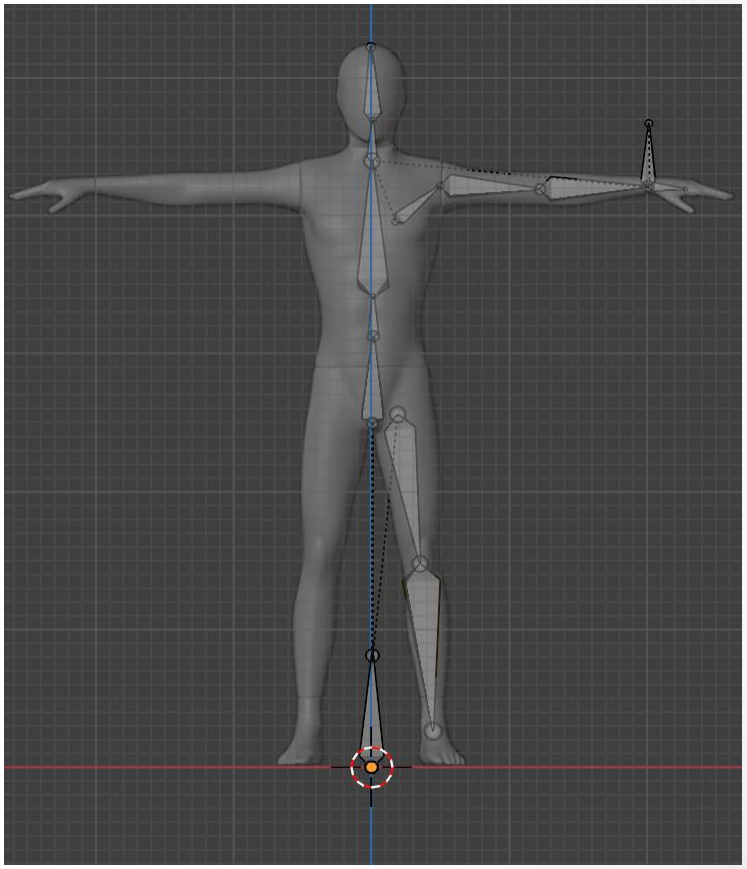
Upper Leg Bone

- Back to Edit Mode
- Select the hip bone, and shift+D to create a new bone (Left Leg)
- Move, Scale and Rotate it
- Rename it Upperleg.L in “Bone” properties



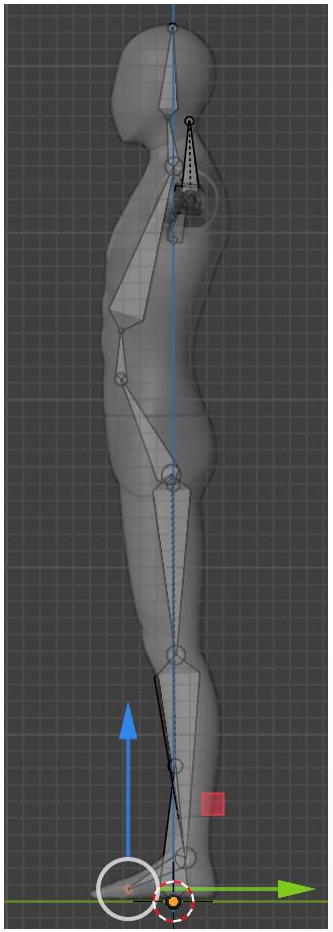
Lower Leg Bone

- Select the knob at the knee level, press E to extrude another bone
- If the bones are twisted, do Ctrl+R to rotate it.
- Rename it to Lowerleg.L in Bone properties



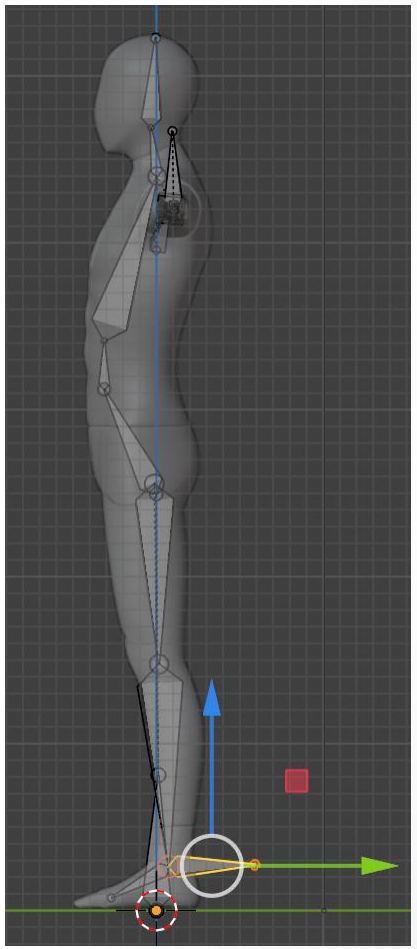
Foot Bone

- In Edit mode, select the knob close o the ankle and press E to extrude the foot node.
- Rename it to Foot.L



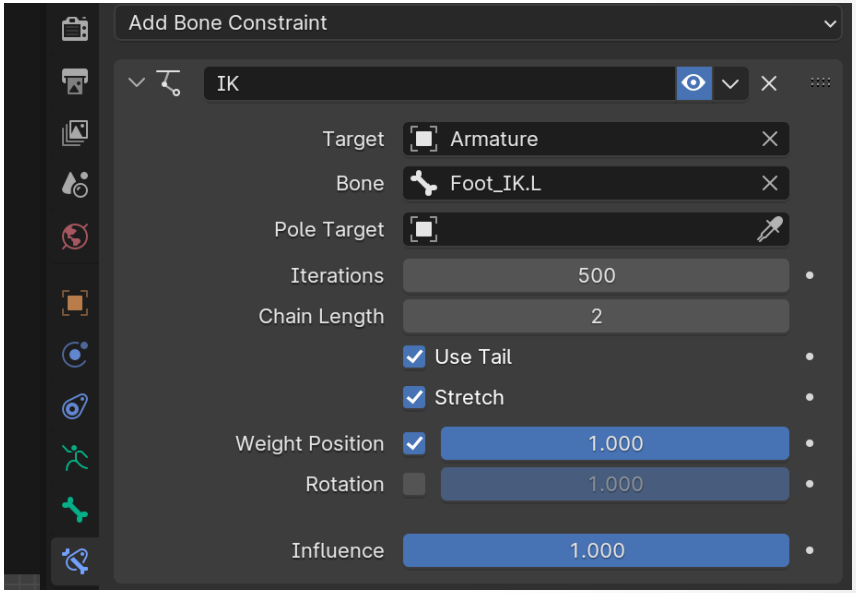
IK for Foot bone

- Create an IK for foot bone. Select the knob by the ankle and press E to extrude an IK bone behind the ankle
- Rename it to Foot_IK.L
- Select the IK bone, Alt+P to both Disconnect and Clear Parent



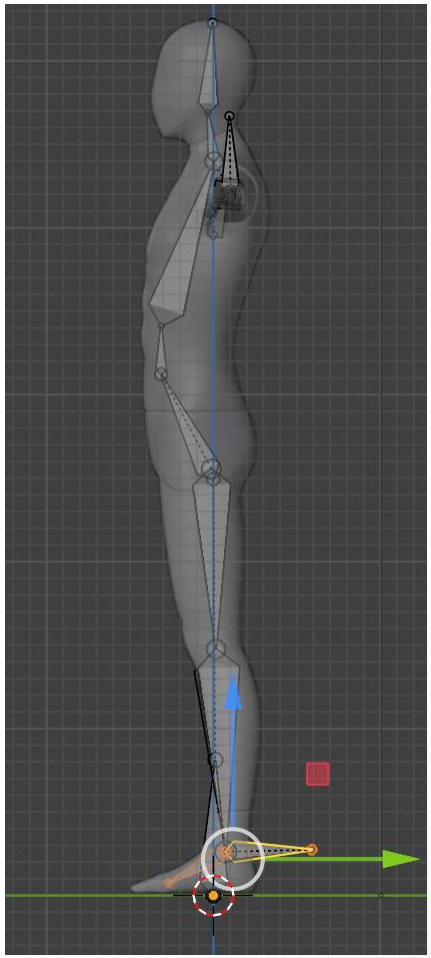
between Upperleg and Hip bone

- In Edit bone, select the upper leg bone, Shift, select the hip bone, Ctrl+P, click on "keep offset"
- Select the Foot IK bone, Shift, select the "Main" bone, Ctrl+P, click on "keep offset"
- Go to "Pose" mode, select the foot IK bone, Shift, Lower Leg bone, Ctrl+Shift+C, click on "Inverse Kinematics"
- Select the Lower Leg bone, Go to "Bone Constraints" and set the Chain Length to 2.



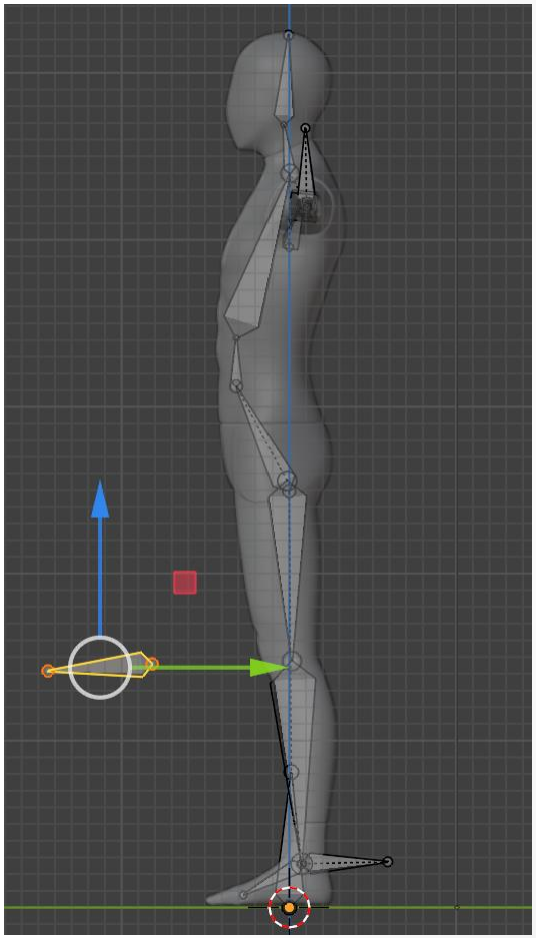
Connect Foot bone to Foot IK

- Back to Edit mode
- Select the Foot Bone, Shift, Foot IK bone, Ctrl+P, Keep Offset
- Go to Pose mode, grap the hip and shake it



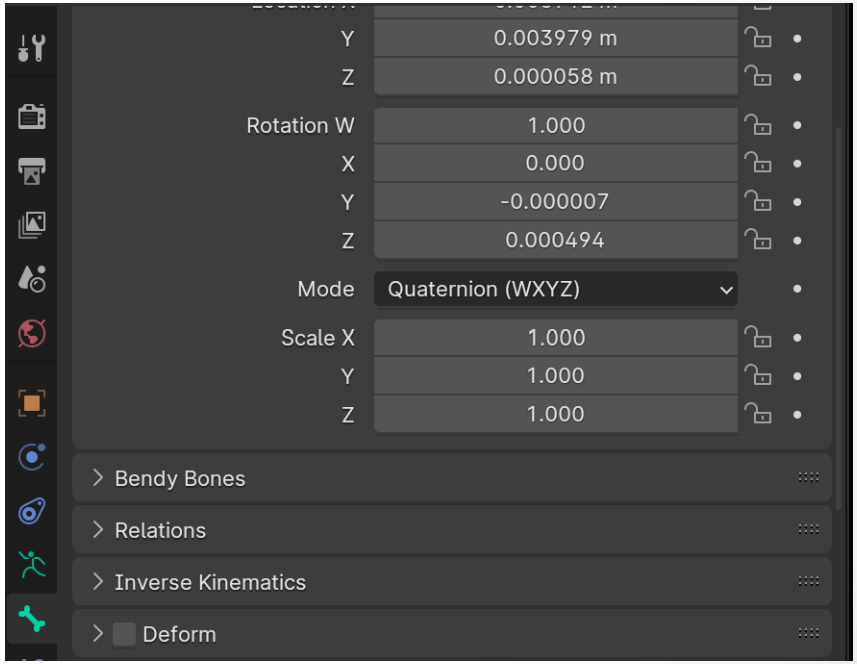
Fixing Knee Rotation

- We are going to make a taget bone to control the rotation of the knee.
- Back to Edit mode
- Select the knee knob, E to extrude forward a bone
- Alt+P disconnect bone, Alt+P clear parent
- G and move it forward
- Rename it Legtarget.L
- Go to pose mode and select the lower leg bone, go to bone constraints, select “Armature” for “Pole Target” constraint and select “LegTarget” for “Bone” constraint, set the “Pole Angle” to 90.
- Homework (Try to do the samething with elbow!)



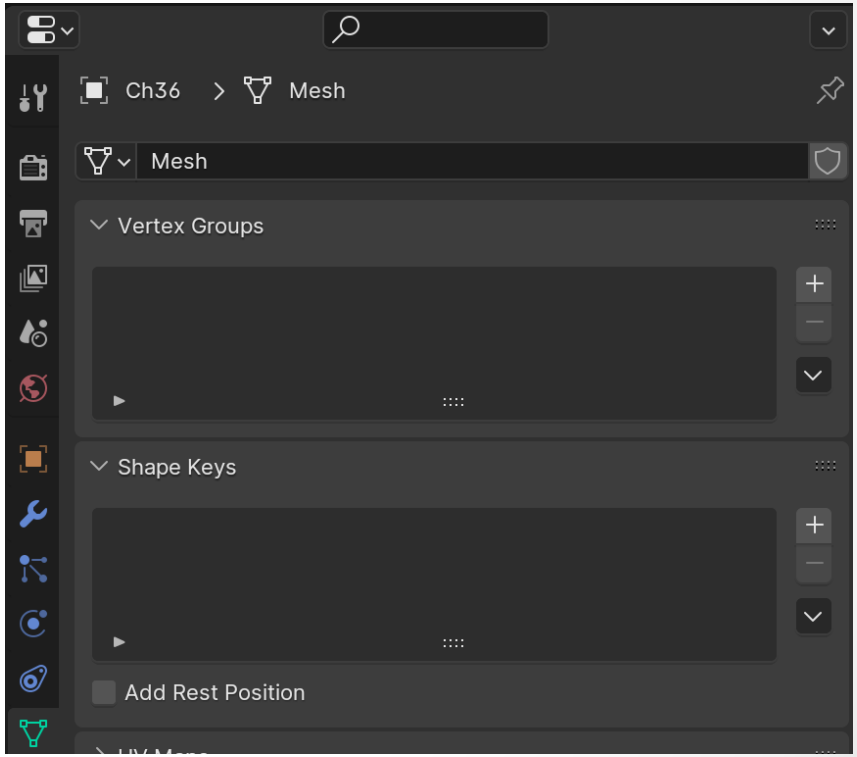
Deformation Vs. Controlling bones

- Deform: Enables bone to deform geometry
- Select the “Main” bone
- Go to the bone property and clear “Deform” checkbox.
- Do the samething for IK bones (Hand&Foot) and the Leg Target



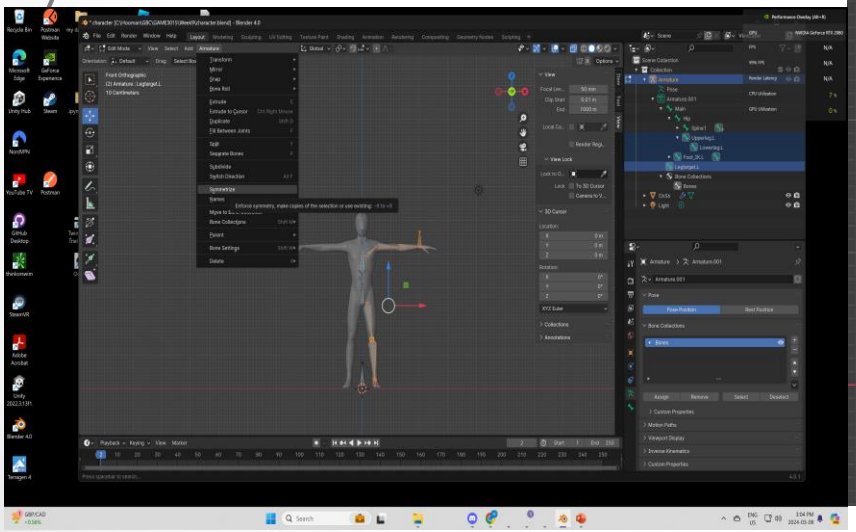
Cleanup Vertex group

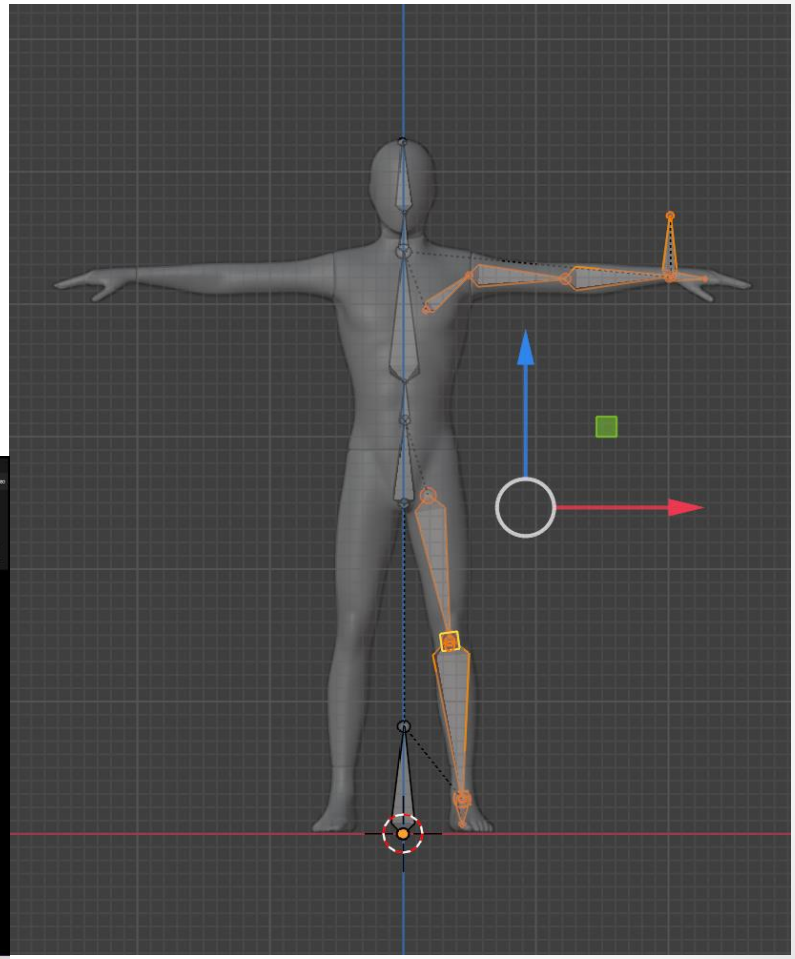
- Go to "Object" mode and select the character
- Go to "Data" properties
- Delete everything in the vertex group that came with the model (we don't need it)
- Any bones that have affects on the mesh, will have a bunch of vertex groups that tells these bones what bits of geometry to use!
- Blender will do it for automatically!



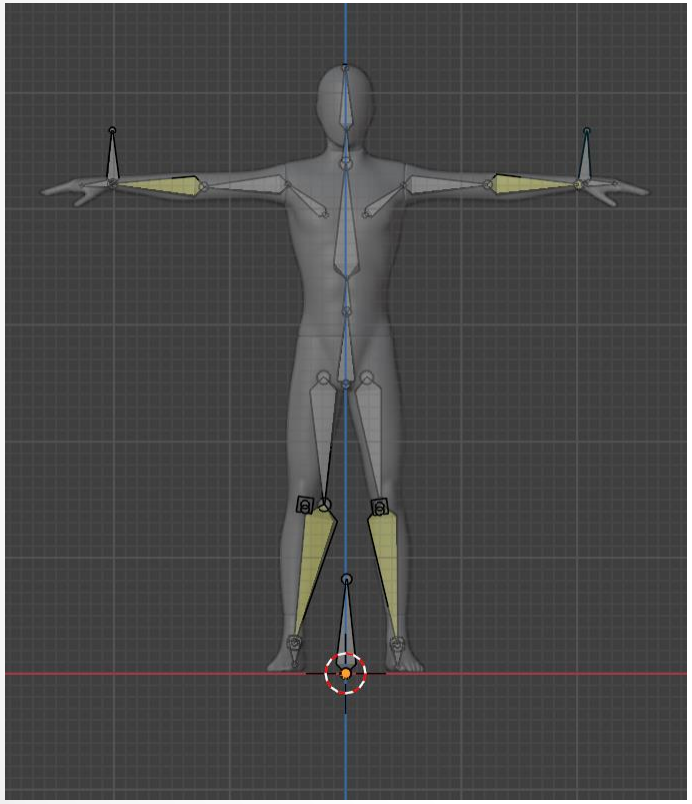
Mirroring the bones

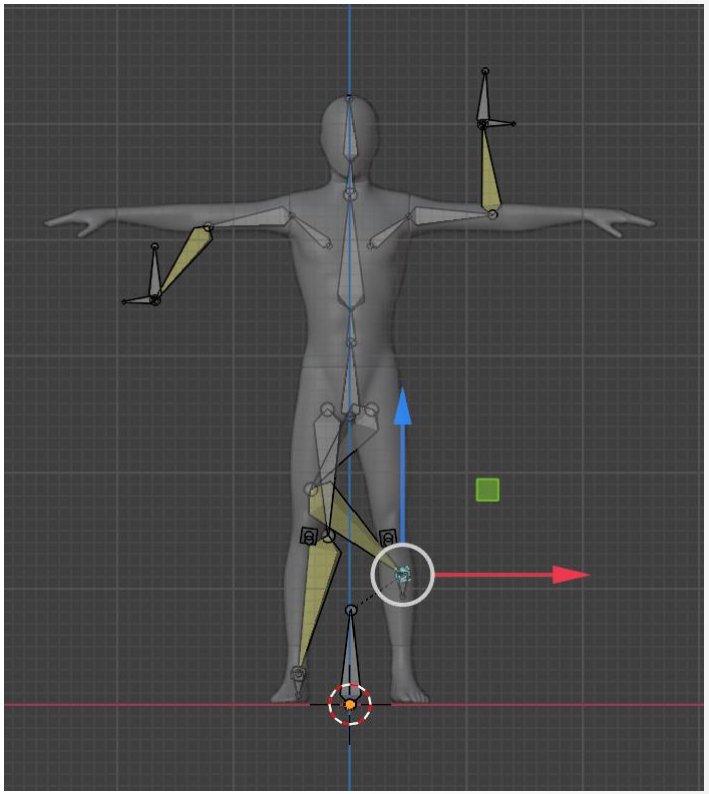
- Go to Edit mode
- Select all the bones on the left side of the character
- Click on Armature, Symmetrize





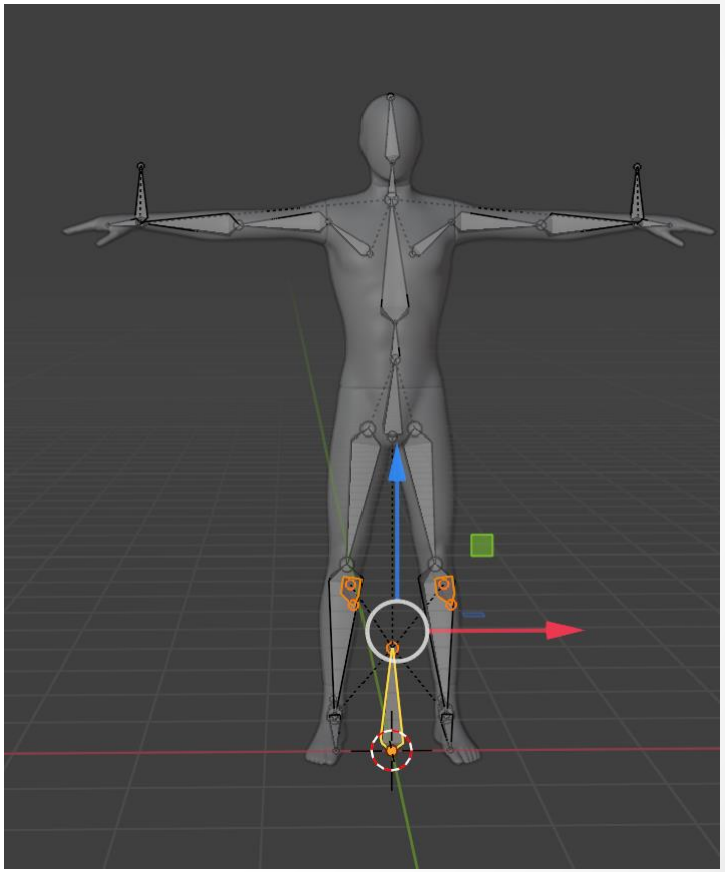
Test in Pose mode





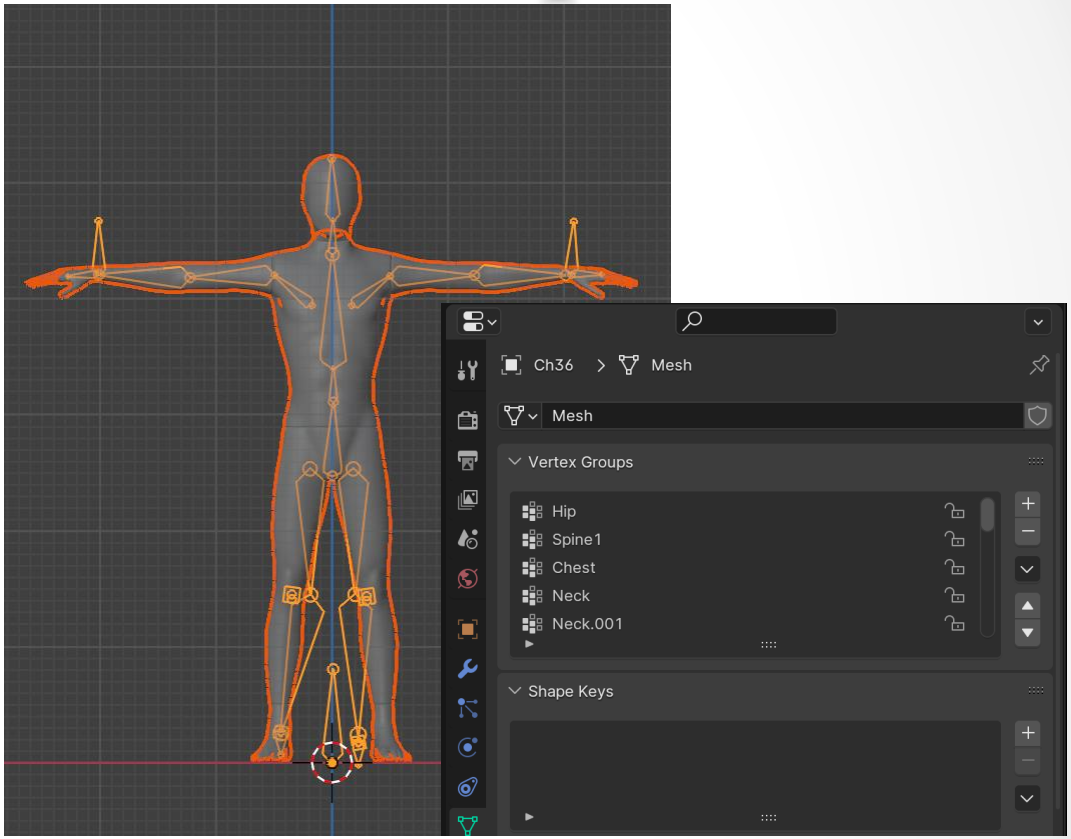
Connect Target Legs with Main bone

- Go back to Edit mode
- Select both tagets and shift with Main, Ctrl+P, Keep offset

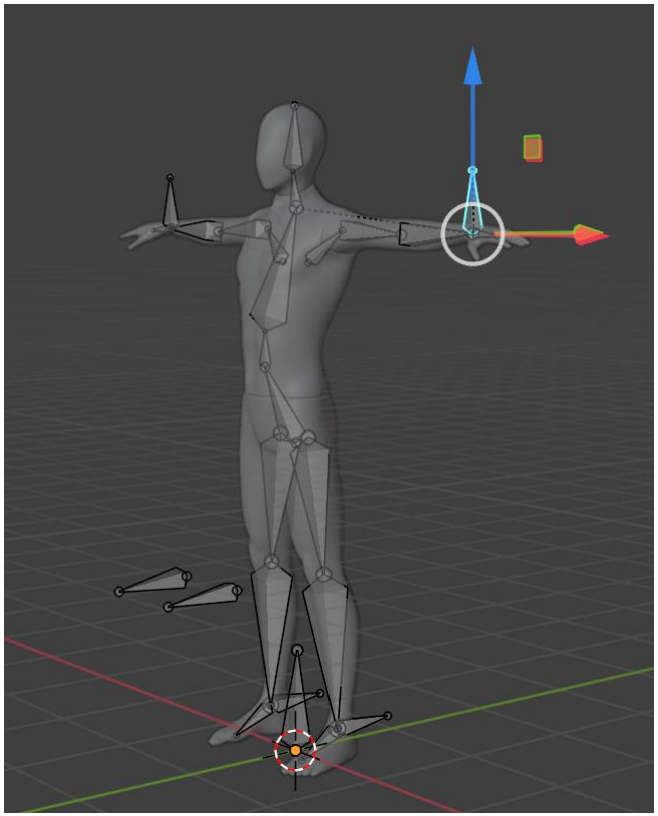


Vertex Group

- Go to Object Mode
- Select the mesh
- Hold and Shift and select the Armature in the object mode
- Ctrl+P with automatic weights
- Select the character, and select "Data" and see all the bones have been added to our vertex group!

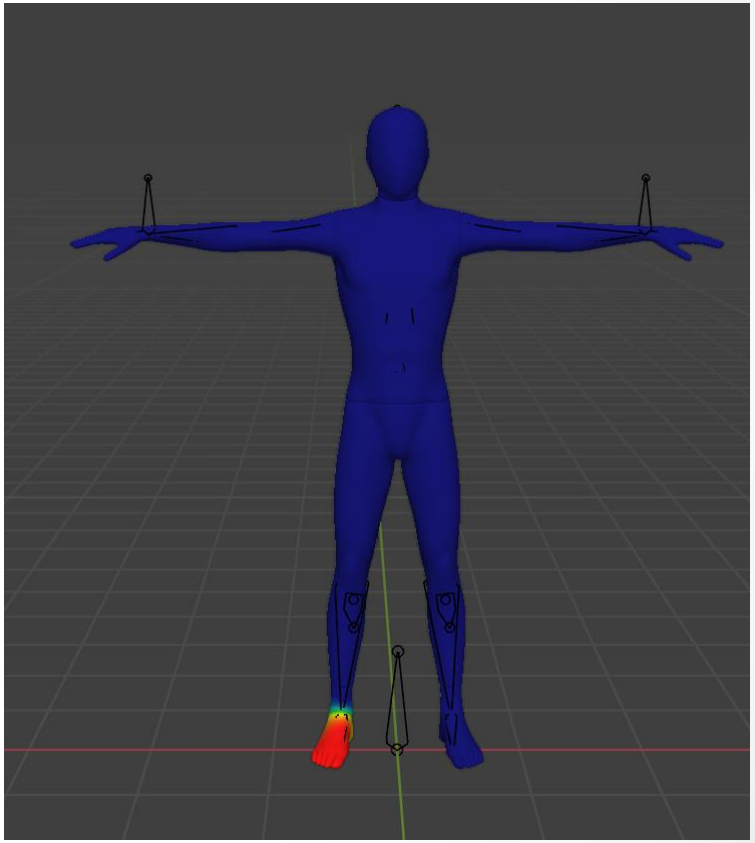


Test in Pose Mode



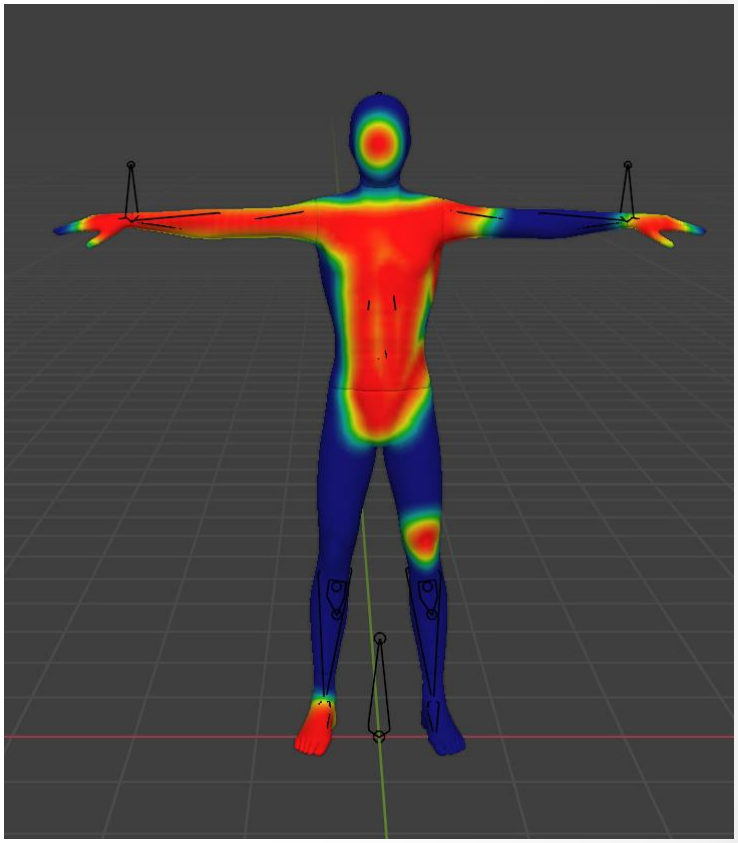
Weight Paint

- Select the mesh and go to “Weight paint”
- Red (Warmer color) means that the bone has more influence on the mesh!



Modify the weight paint

- Go in white paint and if you hold in Ctrl and left click



Materials

The next "chunk" in the .m3d format is a list of materials.

The file contains the material data we are familiar with (diffuse, roughness, etc.), but also contains additional information such as the textures to apply, whether alpha clipping needs to be applied, and the material type name.

The material type name is used to indicate which shader programs are needed for the given material.

In our demo, the "Skinned" type indicates that the material will need to be rendered with shader programs that support skinning.

```
**************Materials**********
Name: soldier_head
Diffuse: 1 1 1
Fresnel0: 0.05 0.05 0.05
Roughness: 0.5
AlphaClip: 0
MaterialTypeName: Skinned
DiffuseMap: head_diff.dds
NormalMap: head_norm.dds

Name: soldier_jacket
Diffuse: 1 1 1
Fresnel0: 0.05 0.05 0.05
Roughness: 0.8
AlphaClip: 0
MaterialTypeName: Skinned
DiffuseMap: jacket_diff.dds
NormalMap: jacket_norm.dds

Name: soldier_pants
Diffuse: 0.8 0.8 0.8
Fresnel0: 0.01 0.01 0.01
Roughness: 0.8
AlphaClip: 0
MaterialTypeName: Skinned
DiffuseMap: pants_diff.dds
NormalMap: pants_norm.dds

Name: soldier_upBody1
Diffuse: 1 1 1
Fresnel0: 0.05 0.05 0.05
Roughness: 0.4
AlphaClip: 0
MaterialTypeName: Skinned
DiffuseMap: upBody_diff.dds
NormalMap: upBody_norm.dds

Name: soldier_upBody2
Diffuse: 1 1 1
Fresnel0: 0.05 0.05 0.05
Roughness: 0.3
AlphaClip: 0
MaterialTypeName: Skinned
DiffuseMap: upBody_diff.dds
NormalMap: upBody_norm.dds
```

Subsets

A mesh consists of one or more subsets. A *subset* is a group of triangles in a mesh that can all be rendered using the same material.

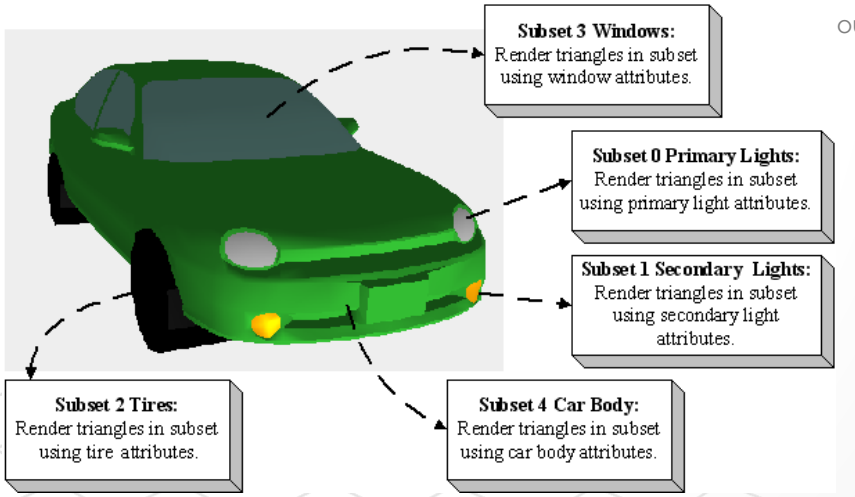
There is a subset corresponding to each material and the *i*th subset corresponds to the *i*th material. The ith subset defines a contiguous block of geometry that should be rendered with the ith material.

The first 7230 triangles of the mesh (which reference vertices [0,3915)) should be rendered with material 0, and the next 4449 triangles of the mesh (which reference vertices [3915, 6899)) should be rendered with material 1.

Figure illustrates how a mesh representing a car may be divided into several subsets.

**********SubsetTable**********

SubsetID: 0 VertexStart: 0 VertexCount: 3915 FaceStart: 0 FaceCount: 7230
SubsetID: 1 VertexStart: 3915 VertexCount: 2984 FaceStart: 7230 FaceCount: 4449
SubsetID: 2 VertexStart: 6899 VertexCount: 4270 FaceStart: 11679 FaceCount: 6579
SubsetID: 3 VertexStart: 11169 VertexCount: 2305 FaceStart: 18258 FaceCount: 3807
ount: 442



Vertex Data and Triangles

The next two chunks of data are just lists of vertices and indices (3 indices per triangle):

**********Vertices**********

Position: -0.9983482 67.88861 4.354969

Tangent: -0.257403 0.5351538 -0.8045831 1

Normal: 0.5368453 0.7715151 0.3414111

Tex-Coords: 0.695366 0.9909569

BlendWeights: 0.7470379 0.2529621 0 0

BlendIndices: 7 9 0 0

**********Triangles**********

0 1 2

3 4 5

6 7 8

9 10 11

12 13 14

15 16 17

Bone Offset Transforms

The bone offset transformation chunk, just stores a list of 4 × 4 matrices, one for each bone.

**********BoneOffsets**********

BoneOffset0 -0.8669753 0.4982096 0.01187624 0

0.04897417 0.1088907 -0.9928461 0

-0.4959392 -0.8601914 -0.118805 0

-10.94755 -14.61919 90.63506 1

BoneOffset1 1 4.884964E-07 3.025227E-07 0

-3.145564E-07 2.163151E-07 -1 0

4.884964E-07 0.9999997 -9.59325E-08 0

3.284225 7.236738 1.556451 1

…

Hierarchy

The hierarchy chunk stores the hierarchy array—an array of integers such that the *i*th array entry gives the parent index of the *i*th bone.

**********BoneHierarchy**********

ParentIndexOfBone0: -1

ParentIndexOfBone1: 0

ParentIndexOfBone2: 1

ParentIndexOfBone3: 2

ParentIndexOfBone4: 3

ParentIndexOfBone5: 4

ParentIndexOfBone6: 5

ParentIndexOfBone7: 6

ParentIndexOfBone8: 7

Animation Data

The last chunk we need to read are the animation clips.

Each animation has a readable name and a list of key frames for each bone in the skeleton.

Each key frame stores the time position, the translation vector specifying the position of the bone, the scaling vector specifying the bone scale, and the quaternion specifying the orientation of the bone.

**********AnimationClips**********

AnimationClip run_loop

{

Bone0 #Keyframes: 18

{

Time: 0 Pos: 2.538344 101.6727 -0.52932

Scale: 1 1 1

Quat: 0.4042651 0.3919331 -0.5853591 0.5833637

Time: 0.0666666

Pos: 0.81979 109.6893 -1.575387

Scale: 0.9999998 0.9999998 0.9999998

Quat: 0.4460441 0.3467651 -0.5356012 0.6276384

…

}

M3DLoader::ReadAnimationClips

The following code shows how we read the animation clips from file:

```
void M3DLoader::ReadAnimationClips(std::ifstream& fin, UINT numBones, UINT
numAnimationClips,
   std::unordered_map<std::string, AnimationClip>& animations)
{
std::string ignore;
    fin >> ignore; // AnimationClips header text
    for(UINT clipIndex = 0; clipIndex < numAnimationClips; ++clipIndex)
    {
        std::string clipName;
        fin >> ignore >> clipName;
        fin >> ignore; // {

AnimationClip clip;
clip.BoneAnimations.resize(numBones);

        for(UINT boneIndex = 0; boneIndex < numBones; ++boneIndex)
        {
            ReadBoneKeyframes(fin, numBones, clip.BoneAnimations[boneIndex]);
        }
        fin >> ignore; // }

        animations[clipName] = clip;
    }
}
```

M3DLoader

The code to load the data from an .m3d file is contained in *LoadM3D.h/.cpp*, in the LoadM3d function:

The helper functions ReadMaterials, and etc., are straightforward text file parsing using std::ifstream.

```
bool M3DLoader::LoadM3d(const std::string& filename,
std::vector<Vertex>& vertices,
std::vector<USHORT>& indices,
std::vector<Subset>& subsets,
std::vector<M3dMaterial>& mats)
{
std::ifstream fin(filename);

UINT numMaterials = 0;
UINT numVertices  = 0;
UINT numTriangles = 0;
UINT numBones     = 0;
UINT numAnimationClips = 0;

std::string ignore;

if( fin )
{
fin >> ignore; // file header text
fin >> ignore >> numMaterials;
fin >> ignore >> numVertices;
fin >> ignore >> numTriangles;
fin >> ignore >> numBones;
fin >> ignore >> numAnimationClips;

ReadMaterials(fin, numMaterials, mats);
ReadSubsetTable(fin, numMaterials, subsets);
ReadVertices(fin, numVertices, vertices);
ReadTriangles(fin, numTriangles, indices);

return true;
 }
   return false;
}
```

CHARACTER ANIMATION DEMO

As we saw in the skinned mesh shader code, the final bone transforms are stored in a constant buffer where they are accessed in the vertex shader to do the animation transformations.

```
cbuffer cbSkinned : register(b1)
{
    float4x4 gBoneTransforms[96];
};
```

We therefore need to add these new constant buffers, one for each skinned mesh object, to our frame resources:

```
struct SkinnedConstants
{
DirectX::XMFLOAT4X4 BoneTransforms[96];
};
std::unique_ptr<UploadBuffer<SkinnedConstants>> SkinnedCB = nullptr;
SkinnedCB = std::make_unique<UploadBuffer<SkinnedConstants>>(
device, skinnedObjectCount, true);
```

We will need one SkinnedConstants for each instance of an animated character.

An animated character instance will generally be composed of multiple render-items (one per material), but all render-items for the same character instance can share the same SkinnedConstants since they all use the same underlying animated skeleton.

To represent an animated character instance at an instance in time, we define the following structure:

```
struct SkinnedModelInstance
{
    SkinnedData* SkinnedInfo = nullptr;
    std::vector<DirectX::XMFLOAT4X4> FinalTransforms;
    std::string ClipName;
    float TimePos = 0.0f;
void UpdateSkinnedAnimation(float dt)
    {
        TimePos += dt;

        // Loop animation
        if(TimePos > SkinnedInfo->GetClipEndTime(ClipName))
            TimePos = 0.0f;

        // Compute the final transforms for this time position.
        SkinnedInfo->GetFinalTransforms(ClipName, TimePos, FinalTransforms);
    }
};
```

RenderItem

Then we add the following data members to our render-item structure:

```
struct RenderItem
{
RenderItem() = default;
    RenderItem(const RenderItem& rhs) = delete;
XMFLOAT4X4 World = MathHelper::Identity4x4();

XMFLOAT4X4 TexTransform = MathHelper::Identity4x4();

int NumFramesDirty = gNumFrameResources;

// Index into GPU constant buffer corresponding to the ObjectCB for this render item.
UINT ObjCBIndex = -1;

Material* Mat = nullptr;
MeshGeometry* Geo = nullptr;

    // Primitive topology.
    D3D12_PRIMITIVE_TOPOLOGY PrimitiveType = D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST;

    // DrawIndexedInstanced parameters.
    UINT IndexCount = 0;
    UINT StartIndexLocation = 0;
    int BaseVertexLocation = 0;

// Only applicable to skinned render-items.
    UINT SkinnedCBIndex = -1;

    // nullptr if this render-item is not animated by skinned mesh.
    SkinnedModelInstance* SkinnedModelInst = nullptr;
};
```

SkinnedMeshApp::UpdateSkinnedCBs

Every frame we update the animated character instances (in our demo we only have one):

```
void SkinnedMeshApp::UpdateSkinnedCBs(const GameTimer& gt)
{
    auto currSkinnedCB = mCurrFrameResource->SkinnedCB.get();

    // We only have one skinned model being animated.
    mSkinnedModelInst->UpdateSkinnedAnimation(gt.DeltaTime());

    SkinnedConstants skinnedConstants;
    std::copy(
        std::begin(mSkinnedModelInst->FinalTransforms),
        std::end(mSkinnedModelInst->FinalTransforms),
        &skinnedConstants.BoneTransforms[0]);

    currSkinnedCB->CopyData(0, skinnedConstants);
}
```

“Skinned Mesh” demo

Figure shows a screenshot or our demo. The original animated model and textures were taken from the DirectX SDK and converted to the .m3d format for demoing purposes.

This sample model only has one animation clip called *Take1*.