Week 12

Rigging and Skinning

Non-Rigid Transformations

Defining complex 3D objects as a hierarchical data structure allows us to define hierarchical transformations.

Hierarchies of transformations allow parts of complex objects to move relative to each other.

All vertices in a basic part are transformed the same way

- Basic parts retain their shape
- We generally have rigid transformations

This works well for objects like tanks with rigid parts

But what about human characters and other objects that have parts that rotate and move, but with surfaces that are flexible rather than rigid?

Rigging is building controls

- Animation is usually specified using some form of low-dimensional **controls** as opposed to remodeling the actual geometry for each frame.
- The joint angles (known as **bone transformations**) in a hierarchical character determine the pose.

Skinning

Skinning provides a means to animate a model while reducing the "folding" and "creasing" of polygons as the model animates

A model is represented in two parts:

1. **Skin** (this is described by the triangular mesh)
2. **Skeleton** (the underlying **bones** of the model)

Define a bone hierarchy in a model

- The mesh (triangle) data is defined separately.
- The bone hierarchy is linked to the model's mesh, so that animating the bone hierarchy animates the mesh's vertices.

Note that multiple bones can affect each vertex in the skin mesh - this produces a **smoothing effect**

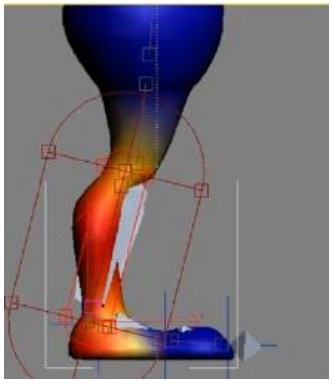
So, we can have a **weighting value** for each bone to determine how much influence it has in proportion to the others

This allows smooth transitions around bone joints

Triangles

	Colored triangles are attached to one bone (different colour means different bone) Black triangles are attached to more than one bone Note that black triangles are near the joints

Smooth Skinning and Bone Weights



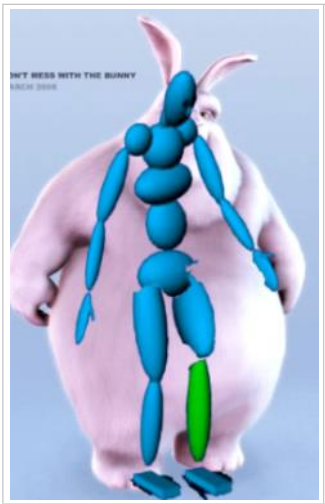
To make deformations appear smooth, we need to average between the effects of different bones close to joints.

Bone weights can be set using modelling software. They are displayed using colours in Blender in **weight paint mode**: red is for close to 1, blue for close to 0.

There are a number of standard ways of setting weights.

- **Automatic geometric weighting**
- **Manual weight-painting** (Often used for tweaking)
- **Bone Envelopes** which specify a distance that each bone affects.

Skinning a Character



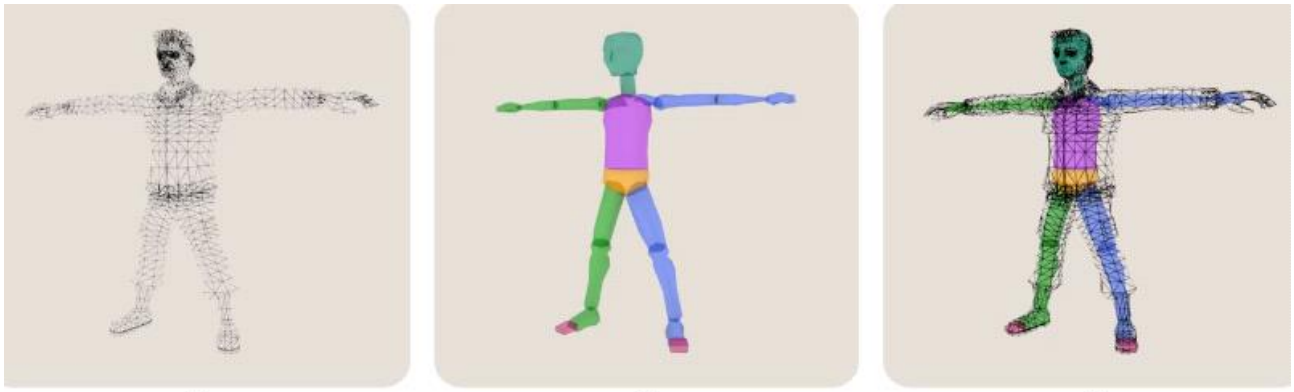
1. Embed a skeleton into a detailed character mesh

2. Animate the "bones" by:
   - Changing the joint angles in each keyframe,
   - Interpolating the keyframes.

3. Bind the skin vertices to the bones. So, animating skeleton will move the skin with it.

Bone Weights

Bind the skin vertices to bones

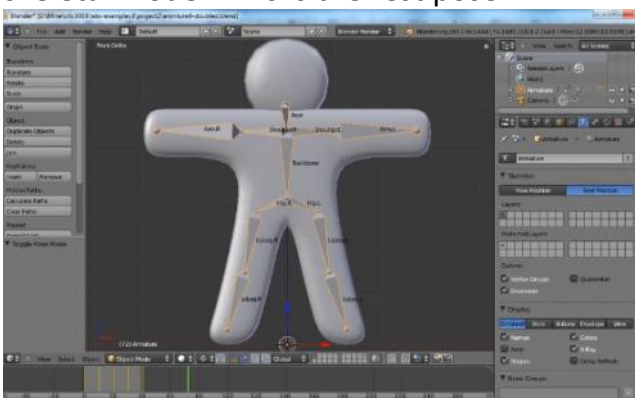


(a) (b) (c)

(a) Vertices of the skin
(b) Bones
(c) Both skin and bones

Blender

A gingerbread man and the underlying skeletal model. This is the rest pose.



Blender is a software application that provides a good interface for 3D modelling, skinning, and rigging.

- The skeletal model and the polygonal mesh can be defined in Blender.
- Key poses for animation can be added and exported to files in several formats.
- Manual weight painting can also be done in Blender.

Bone Movement

Bone Transformations

- Each bone has a 4 X 4 matrix that specifies how it transforms the points and vectors it influences for the current pose
- All bone transformations are defined relative to a rest pose (not a previous pose)

Transforming a bone:

- Transforms all children bones (and their children, etc.)
- Transforms corresponding vertices following the bone weights
- Transforms the normal vectors similarly

So, to calculate the position and normal for a vertex, the bone transformations affecting the vertex are averaged according to the bone weights, then the resultant transformation is applied to both the position and normal.

Bone Weights

For each vertex $\boldsymbol{p}_i$ we'll assign a weight w_{ij} for each bone B_j.

- Q: "How much should vertex $\boldsymbol{p}_i$ move with bone j?
- $w_{ij} = 1$ means $\boldsymbol{p}_i$ is rigidly attached to bone j.
- We can interpret w_{ij} to be the weight being exerted by bone B_j on vertex $\boldsymbol{p}_i$ thus the term "**bone weight**".

Desirable properties for the weight w_{ij}

- Usually want weights to be non-negative, i.e., $w_{ij} \geq 0$
- Also, want the sum over all bones to be 1 for each vertex $\boldsymbol{p}_i$, i.e., we want $\sum_{j=1}^{N} w_{ij} = 1$.

I = Vertex
J = Bone number
W = Bone weight

The sum over all bones should sum up to 1, to simulate a percentage effect

We'll limit the number of bones N that can influence a single vertex

- $N = 4$ bones/vertex is a usual choice
- Why 4? You most often don't need many.
- Also, storage space is an issue.
- In practice, we'll store N pairs of numbers (bone index j, weight w_{ij}) per vertex.

Computing Vertex Positions

Basic step 1: Transform each vertex $\boldsymbol{p}_i$ with each bone j as if it was tied to it rigidly.

$$\boldsymbol{p}'_{ij} = T_j \boldsymbol{p}_i$$

Basic step 2: Then blend the results using the weights.

$$\boldsymbol{p}'_i = \sum_{j=1}^{N} w_{ij} \boldsymbol{p}'_{ij}$$

where $\boldsymbol{p}'_{ij}$ is the vertex i transformed using bone j;

T_j is the current transformation of bone B_j;

$\boldsymbol{p}'_i$ is the new skinned position of vertex $\boldsymbol{p}_i$.

Data Structure Required for Skinning

The necessary data to define relationship between bone hierarchy and the skin are:

- Number of vertices
- Number of bones
- Vertex Weights
- Vertex Indices
- Inverse Matrices

Values are extracted from the modeler when loading an animation

Inverse Bone Matrices

InvBMs are used to transform the skin data from the rest pose **object/skin space** to **bone space**, i.e., to coordinates relative to the position and orientation of a particular bone.

An InvBM is the inverse of the transformation matrix from the **root bone** in the hierarchy to a particular bone in the hierarchy in the rest/default pose.

In the project these are provided by the Open Asset Importer as:
`bone->mOffsetMatrix`
(see the file gnatidread2.h)

High Level Skinning Algorithm

1. Transforms vertex positions from their object space for the **rest pose (skin space)** into **bone space** using the **InvBMs**.

2. Performs the key-frame based animation for that frame, which transform the vertex positions back into skin space in its animated pose.

3. The model is then rendered.

Smooth Skinning is well suited to implementing as a **vertex shader**

- You deform vertices one by one based on the joint transforms
  - Use uniform variables for the transformations of the bones in the current pose and change them each time the model is drawn.
- You need to get the bone weights for each vertex to the GPU
  - Use attributes, i.e., vertex shader "in" or "varying" variables
  - These don't change after the model has been loaded
  - For the best performance, normally only a small number of bones (e.g., N = 4 as mentioned before) are allowed to affect each vertex.

Skinning Code

Vertex Shader

```
// in variables set for each vertex from the static mesh data
in  ivec4 BoneIDs;       // Integer IDs of the 4 bones affecting this vertex
in  vec4 BoneWeights;    // The corresponding 4 weights between 0.0 and 1.0

const int MAX_BONES = 32; // Adjust based on expected model complexity

// the current bone transformations, set each time the model is drawn
uniform mat4 BoneTransforms[MAX_BONES];

void main()  {
    // Calculate a weighted average of the given 4 bones transformations
    mat4 BoneTransform =  BoneTransforms[BoneIDs[0]] * BoneWeights[0]
                        + BoneTransforms[BoneIDs[1]] * BoneWeights[1]
                        + … … …

    vec4 btPosition = BoneTransform * vPosition;     // Transform the position
    vec4 btNormal = BoneTransform * vNormal;         // and the normal
    // The rest of the vertex shader should use btPosition and btNormal
}
```

C++ Initialization

```
for (unsigned int boneID = 0 ; boneID < mesh->mNumBones ; boneID++) {
    for (unsigned int j = 0 ; j < mesh->mBones[boneID]->mNumWeights ; j++) {
        int VertexID = mesh->mBones[boneID]->mWeights[j].mVertexId;
        float Weight = mesh->mBones[boneID]->mWeights[j].mWeight;

        // Insertion sort, keep the 4 largest weights
        for(int slotID=0; slotID < 4; slotID++) {
            if(boneWeights[VertexID][slotID] < Weight) {
               for(int shuff=3; shuff>slotID; shuff--) {
                 boneWeights[VertexID][shuff] = boneWeights[VertexID][shuff-1];
                 boneIDs[VertexID][shuff] = boneIDs[VertexID][shuff-1];
               }
               boneWeights[VertexID][slotID] = Weight;
               boneIDs[VertexID][slotID] = boneID;
               break;
            }}}
```

gnatidread2.h

C++ Bone Transforms

```
// rotIndex is for the rotation key just before or at currentTime
If (rotIndex+1 == channel-> mNumRotationKeys)
    curRotation = channel->mRotationKeys[rotIndex].mValue;
else {
    float t0 = channel->mRotationKeys[rotIndex].mTime;
    float t1 = channel->mRotationKeys[rotIndex+1].mTime;
    float wgt1 = (currentTime-t0)/(t1-t0);
    // Interpolate using quaternions
    aiQuaternion::Interpolate(curRotation,
                              channel->mRotationKeys[rotIndex].mValue,
                              channel->mRotationKeys[rotIndex+1].mValue, wgt1);
    curRotation = curRotation.Normalize();
}
// now build a transformation matrix from it. First rotation, then position.
aiMatrix4x4 trafo = aiMatrix4x4(curRotation.GetMatrix());
trafo.a4 = curPosition.x; trafo.b4 = curPosition.y; trafo.c4 = curPosition.z;
targetNode->mTransformation = trafo; // assign this transformation to the node
```

gnatidread2.h

Quaternion interpolation

C++ Bone Transform Composition

```
mat4 boneTransforms[mesh->mNumBones];
for(unsigned int a=0; a<mesh->mNumBones; a++) {
    const aiBone* bone = mesh->mBones[a];
    // find the node, looking recursively through the hierarchy for the name
    aiNode* node = scene->mRootNode->FindNode(bone->mName);
     aiMatrix4x4 b = bone->mOffsetMatrix;  // start with the mesh-to-bone
   matrix
    // then add the bone's transformation for the current pose by composing all
    // nodes pose transformations up the parent chain
    const aiNode* tempNode = node;
    while( tempNode) {
        b = tempNode->mTransformation * b;
        tempNode = tempNode->mParent;
    }
    boneTransforms[a] = mat4(vec4(b.a1, b.a2, b.a3, b.a4), vec4(…), …  );
}
glUniformMatrix4fv(uBoneTransforms, mesh->mNumBones, GL_TRUE,
                                    (const GLfloat *)boneTransforms );
```

Inverse bone matrix (see slide 16 in PDF slides)

A while loop that applies all the parent note transformations

Tree traversal in while loop