Skin

CSE169: Computer Animation
Instructor: Steve Rotenberg
UCSD, Winter 2008
Rendering Review
Rendering

- Renderable surfaces are built up from simple primitives such as triangles.
- They can also use smooth surfaces such as NURBS or subdivision surfaces, but these are often just turned into triangles by an automatic tessellation algorithm before rendering.
Lighting

- We can compute the interaction of light with surfaces to achieve realistic shading.
- For lighting computations, we usually require a position on the surface and the normal.
- GL does some relatively simple *local* illumination computations.
- For higher quality images, we can compute *global* illumination, where complete light interaction is computed within an environment to achieve effects like shadows, reflections, caustics, and diffuse bounced light.
Gouraud & Phong Shading

- We can use triangles to give the appearance of a smooth surface by faking the normals a little

- Gouraud shading is a technique where we compute the lighting at each vertex and interpolate the resulting color across the triangle

- Phong shading is more expensive and interpolates the normal across the triangle and recomputes the lighting for every pixel
Materials

- When an incoming beam of light hits a surface, some of the light will be absorbed, and some will scatter in various directions.
Materials

- In high quality rendering, we use a function called a BRDF (bidirectional reflectance distribution function) to represent the scattering of light at the surface:

\[ f_r(\theta_i, \phi_i, \theta_r, \phi_r, \lambda) \]

- The BRDF is a 5 dimensional function of the incoming light direction (2 dimensions), the outgoing direction (2 dimensions), and the wavelength
Translucency

- Skin is a translucent material. If we want to render skin realistically, we need to account for subsurface light scattering.
- We can extend the BRDF to a BSSRDF by adding two more dimensions representing the translation in surface coordinates. This way, we can account for light that enters the surface at one location and leaves at another.
- Learn more about these in CSE168!
Texture

- We may wish to ‘map’ various properties across the polygonal surface
- We can do this through texture mapping, or other more general mapping techniques
- Usually, this will require explicitly storing texture coordinate information at the vertices
- For higher quality rendering, we may combine several different maps in complex ways, each with their own mapping coordinates
- Related features include bump mapping, displacement mapping, illumination mapping…
Smooth Skin Algorithm
Weighted Blending & Averaging

- Weighted sum: \( x' = \sum_{i=0}^{i} w_i x_i \)

- Weighted average: \( \sum_{i=0}^{i} w_i = 1 \)

- Convex average: \( 0 \leq w_i \leq 1 \)
Rigid Parts

- Robots and mechanical creatures can usually be rendered with rigid parts and don’t require a smooth skin.
- To render rigid parts, each part is transformed by its joint matrix independently.
- In this situation, every vertex of the character’s geometry is transformed by exactly one matrix.

\[ v' = W \cdot v \]

where \( v \) is defined in joint’s local space.
Simple Skin

- A simple improvement for low-medium quality characters is to rigidly bind a skin to the skeleton. This means that every vertex of the continuous skin mesh is attached to a joint.
- In this method, as with rigid parts, every vertex is transformed exactly once and should therefore have similar performance to rendering with rigid parts.

\[ v' = W \cdot v \]
Smooth Skin

- With the smooth skin algorithm, a vertex can be attached to more than one joint with adjustable weights that control how much each joint affects it.
- Verts rarely need to be attached to more than three joints.
- Each vertex is transformed a few times and the results are blended.
- The smooth skin algorithm has many other names: blended skin, skeletal subspace deformation (SSD), multi-matrix skin, matrix palette skinning...
Smooth Skin Algorithm

The deformed vertex position is a weighted average:

$$\mathbf{v}' = w_1 (\mathbf{M}_1 \cdot \mathbf{v}) + w_2 (\mathbf{M}_2 \cdot \mathbf{v}) + \ldots + w_N (\mathbf{M}_N \cdot \mathbf{v})$$

or

$$\mathbf{v}' = \sum w_i (\mathbf{M}_i \cdot \mathbf{v})$$

where

$$\sum w_i = 1$$
**Binding Matrices**

- With rigid parts or simple skin, \( \mathbf{v} \) can be defined local to the joint that transforms it.
- With smooth skin, several joints transform a vertex, but it can’t be defined local to all of them.
- Instead, we must first transform it to be local to the joint that will then transform it to the world.
- To do this, we use a binding matrix \( \mathbf{B} \) for each joint that defines where the joint was when the skin was attached and premultiply its inverse with the world matrix:

\[
\mathbf{M}_i = \mathbf{W}_i \cdot \mathbf{B}_i^{-1}
\]
Normals

- To compute shading, we need to transform the normals to world space also.
- Because the normal is a direction vector, we don’t want it to get the translation from the matrix, so we only need to multiply the normal by the upper 3x3 portion of the matrix.
- For a normal bound to only one joint:

  \[ n' = W \cdot n \]
Normals

- For smooth skin, we must blend the normal as with the positions, but the normal must then be renormalized:

\[ n' = \frac{\sum w_i (M_i \cdot n)}{\sum w_i (M_i \cdot n)} \]

- If the matrices have non-rigid transformations, then technically, we should use:

\[ n' = \frac{\sum w_i (M_i^{-1T} \cdot n)}{\sum w_i (M_i^{-1T} \cdot n)} \]
Algorithm Overview

Skin::Update() (view independent processing)
- Compute skinning matrix for each joint: $\mathbf{M} = \mathbf{W} \cdot \mathbf{B}^{-1}$ (you can precompute and store $\mathbf{B}^{-1}$ instead of $\mathbf{B}$)
- Loop through vertices and compute blended position & normal

Skin::Draw() (view dependent processing)
- Set matrix state to Identity (world)
- Loop through triangles and draw using world space positions & normals

Questions:
- Why not deal with $\mathbf{B}$ in Skeleton::Update()?
- Why not just transform vertices within Skin::Draw()?
Rig Data Flow

- Input DOFs

\[ \Phi = \begin{bmatrix} \phi_1 & \phi_2 & \cdots & \phi_N \end{bmatrix} \]

- Rigging system
  (skeleton, skin…)

- Output renderable mesh
  (vertices, normals…)

\[ v', n' \]
Skeleton Forward Kinematics

- Every joint computes a local matrix based on its DOFs and any other constants necessary (joint offsets…)

\[ L = L_{\text{joint}}(\phi_1, \phi_2, \ldots, \phi_N) \]

- To find the joint’s world matrix, we compute the dot product of the local matrix with the parent’s world matrix

\[ W = W_{\text{parent}} \cdot L \]

- Normally, we would do this in a depth-first order starting from the root, so that we can be sure that the parent’s world matrix is available when its needed
Smooth Skin Algorithm

- The deformed vertex position is a weighted average over all of the joints that the vertex is attached to:
  \[ v' = \sum w_i W_i \cdot B_i^{-1} \cdot v \]

- \( W \) is a joint’s world matrix and \( B \) is a joint’s binding matrix that describes where it’s world matrix was when it was attached to the skin model (at skin creation time).

- Each joint transforms the vertex as if it were rigidly attached, and then those results are blended based on user specified weights.

- All of the weights must add up to 1: \( \sum w_i = 1 \)

- Blending normals is essentially the same, except we transform them as direction vectors \((x,y,z,0)\) and then renormalize the results.

\[ n^* = \sum w_i W_i \cdot B_i^{-1} \cdot n, \quad n' = \frac{n^*}{|n^*|} \]
Skinning Equations

- **Skeleton**

\[ L = L_{\text{joint}}(\phi_1, \phi_2, \ldots, \phi_N) \]
\[ W = W_{\text{parent}} \cdot L \]

- **Skinning**

\[ v' = \sum w_i W_i \cdot B_i^{-1} \cdot v \]
\[ n^* = \sum w_i W_i \cdot B_i^{-1} \cdot n \]
\[ n' = \frac{n^*}{|n^*|} \]
Using Skinning
Limitations of Smooth Skin

- Smooth skin is very simple and quite fast, but its quality is limited.
- The main problems are:
  - Joints tend to collapse as they bend more.
  - Very difficult to get specific control.
  - Unintuitive and difficult to edit.
- Still, it is built in to most 3D animation packages and has support in both OpenGL and Direct3D.
- If nothing else, it is a good baseline upon which more complex schemes can be built.
Limitations of Smooth Skin
Bone Links

- To help with the collapsing joint problem, one option is to use bone links.
- Bone links are extra joints inserted in the skeleton to assist with the skinning.
- They can be automatically added based on the joint’s range of motion. For example, they could be added so as to prevent any joint from rotating more than 60 degrees.
- This is a simple approach used in some real time games, but doesn’t go very far in fixing the other problems with smooth skin.
Another extension to the smooth skinning algorithm is to allow the verts to be modeled at key values along the joints motion. For an elbow, for example, one could model it straight, then model it fully bent. These shapes are interpolated local to the bones before the skinning is applied. We will talk more about this technique in the next lecture.
Muscles & Other Effects

- One can add custom effects such as muscle bulges as additional joints.
- For example, the bicep could be a translational or scaling joint that smoothly controls some of the verts in the upper arm. Its motion could be linked to the motion of the elbow rotation.
- With this approach, one can also use skin for muscles, fat bulges, facial expressions, and even simple clothing.
- We will learn more about advanced skinning techniques in a later lecture.
Rigging Process

- To rig a skinned character, one must have a geometric skin mesh and a skeleton.
- Usually, the skin is built in a relatively neutral pose, often in a comfortable standing pose.
- The skeleton, however, might be built in more of a zero pose where the joints DOFs are assumed to be 0, causing a very stiff, straight pose.
- To attach the skin to the skeleton, the skeleton must first be posed into a binding pose.
- Once this is done, the verts can be assigned to joints with appropriate weights.
Skin Binding

- Attaching a skin to a skeleton is not a trivial problem and usually requires automated tools combined with extensive interactive tuning.
- Binding algorithms typically involve heuristic approaches.
- Some general approaches:
  - Containment
  - Point-to-line mapping
  - Delaunay tetrahedralization
Containment Binding

- With containment binding algorithms, the user manually approximates the body with volume primitives for each bone (cylinders, ellipsoids, spheres…)
- The algorithm then tests each vertex against the volumes and attaches it to the best fitting bone
- Some containment algorithms attach to only one bone and then use smoothing as a second pass. Others attach to multiple bones directly and set skin weights
- For a more automated version, the volumes could be initially set based on the bone lengths and child locations
Point-to-Line Mapping

- A simple way to attach a skin is treat each bone as one or more line segments and attach each vertex to the nearest line segment.
- A bone is made from line segments connecting the joint pivot to the pivots of each child.
Delaunay Tetrahedralization

- This tricky computational geometry technique builds a tetrahedralization of the volume within the skin.
- The tetrahedra connect all of the skin verts and skeletal pivots in a relatively clean ‘Delaunay’ fashion.
- The connectivity of the mesh can then be analyzed to determine the best attachment for each vertex.
Skin Adjustment

- Mesh Smoothing: A joint will first be attached in a fairly rigid fashion (either automatic or manually) and then the weights are smoothed algorithmically.
- Weight Painting: Some 3D tools allow visualization of the weights as colors (0…1 -> black…white). These can then be adjusted and ‘painted’ in an interactive fashion.
- Direct Manipulation: These algorithms allow the vertex to be moved to a ‘correct’ position after the bone is bent, and automatically compute the weights necessary to get it there.
Hardware Skinning

- The smooth skinning algorithm is simple and popular enough to have some direct support in 3D rendering hardware.
- Actually, it just requires standard vector multiply/add operations and so can be implemented in microcode.
Skin Memory Usage

- For each vertex, we need to store:
  - Rendering data (position, normal, color, texture coords, tangents...)
  - Skinning data (number of attachments, joint index, weight...)
- If we limit the character to having at most 256 bones, we can store a bone index as a byte.
- If we limit the weights to 256 distinct values, we can store a weight as a byte (this gives us a precision of 0.004%, which is fine).
- If we assume that a vertex will attach to at most 4 bones, then we can compress the skinning data to \((1+1)*4 = 8\) bytes per vertex (64 bits).
- In fact, we can even squeeze another 8 bits out of that by not storing the final weight, since

\[ w3 = 1 - w0 - w1 - w2 \]
Project 2: Skin
Assignment:

- Load a .skin file and attach it to the skeleton using the world space matrices to transform the positions and normals with the smooth skin algorithm.
- Use GL lighting to display the skin shaded (use at least two different colored lights).
- Add some sort of interactive control for selecting and adjusting DOFs (can be a simple ‘next DOF’ key and ‘increase’ and ‘decrease’ key). The name and value of the DOF must be displayed somewhere.
Skin File

positions [num] {
  [x] [y] [z]
}

normals [num] {
  [x] [y] [z]
}

skinweights [num] {
  [numbinds] [joint0] [weight0] [j1] [w1] … [jN-1] [wN-1]
}

triangles [num] {
  [index0] [index1] [index2]
}

bindings [num] {
  matrix {
    [ax] [ay] [az] [bx] [by] [bz] [cx] [cy] [cz] [dx] [dy] [dz]
  }
}
Suggestions

- You might consider making classes for:
  - Vertex
  - Triangle
  - Skin

- Keep a clean interface between the skin and the skeleton. A Skeleton::GetWorldMatrix(int) function may be all that is necessary. This way, the skeleton doesn’t need to know anything about skin and the skin only needs to be able to grab matrices from the skeleton.

- Make sure that your skeleton creates the tree in the correct order for the joint indexing to work correctly.