Virtual Skeleton

- Composed of joints which behave in the same way as a scene graph
- Joints are ordered like a scene graph as well
  - One parent joint
  - Multiple child joints
  - Overall transformation matrix
Use in animation

- Virtual skeletons’ primary use is the posing of geometric meshes
- Do not have to manually transform each vertex of the mesh to the desired position
  - Instead, associate each vertex with several joints of the skeleton
- When a joint is transformed, the vertices associated with it are transformed relative to the joint
Computing new vertex positions

- A mesh can be “bound” to a skeleton so it is influenced by the joints
  - Upon binding, each joint’s current overall transformation is computed, then inverted
    - This is known as a Bind Matrix
    - Overall transformation is the concatenation of its transform and all its parents’ transforms
  - This inverted matrix takes positions in world space and finds their position relative to the given joint (local joint space)
Computing new vertex positions

- Bind matrix example: A single joint has a rotation of \(0, 0, 90\) followed by a translation of \(2,2,0\)
  - A vertex at the point \(1,0,0\) would be at the point \(-2, 1, 0\) relative to the joint
- Quick way to compute the inverse transformation: apply the opposite of each transformation in reverse order
  - e.g. translate \(-2, 0, -2\) followed by rotation \(0, -90, 0\)
Bind Matrix: Clicker

- Vertex position relative to Joint 2 is:
  - 1) <-3, 1, 0>
  - 2) <-1, 3, 0>
  - 3) <1, -3, 0>
Joint Influence

- Now that we know the bind matrix of each joint, we can compute the position of any vertex relative to that joint.
- Given a relative vertex position, we can also compute that vertex’s world position when transformed by a joint’s transformation matrix.
Multiple Joint Influence

- Grouping vertices by single joint is useful for transforming portions of a mesh, but it leads to unsightly mesh deformations.
- Instead, associate multiple joints with each vertex and assign each joint an influence weight such that the weights sum up to 1.
  - Blend the transformations of all joints together based on their influence weight.
  - This produces an overall transformation matrix that affects the vertex.
Assigning joint weights

- Procedurally generating “correct” joint weights is a difficult task.
- Naive solution: for each vertex, find the N closest joints and assign each joint a weight of (distance_from_vert / summed_distances_from_vert).
  - Approximates some useful weights, but does not account for mesh connectivity.
Assigning joint weights

- Better solution: treat each joint as a heat source and diffuse influence as heat along the mesh surface
  ○ The half-edge data structure makes this easy because it gives us adjacency information
1. For each joint, find the closest vertex on the mesh
   a. Use its distance to find the set of vertices within a certain minimum distance threshold
   b. Set the influence of this joint on these vertices to 1.0
Assigning joint weights

2. For all vertices connected to the original set of vertices, set their influence for this joint as a function of their distance from vertices with an influence weight > 0
   a. The function should be relative to the original vertices’ distance from the joint
   b. We want to approximate the heat diffusion function:
      \[
      \frac{\partial u}{\partial t} - \alpha \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) = 0
      \]
   c. A cubic falloff function over distance works well for this
Assigning joint weights

3. Repeat for all non-visited vertices adjacent to the just-visited vertices.
   a. Note: this can also be done for all vertices on the faces connected to our starting vertex/vertices, which will give a slightly different result.
Assigning joint weights

3. Repeat for all non-visited vertices adjacent to the just-visited vertices.
Assigning joint weights

3. Repeat for all non-visited vertices adjacent to the just-visited vertices.
Assigning joint weights

4. Stop influence diffusion once the weight of the current iteration is below a certain threshold
5. Repeat for all children of the current bone.
6. After weighting all vertices, normalize their weights
   a. This lets us avoid having to write the influence transfer function so that it implicitly creates normalized weights, which is tricky
Assigning joint weights

- Heat diffusion simulation gives better skin weight results than a closest-joint function, but it may not be entirely what the user wants.
- Modern 3D modeling software supports user-dictated skin weights in addition to the methods previously discussed:
  - For each joint, visualize its influence on vertices as a color gradient.
  - Adjust the weights for the vertices on a particular joint until the mesh deforms as desired.
Linearly Blending Transforms

- Linearly interpolating translate and scale matrices works as expected
- Linearly interpolating rotation matrices does not work!
  - LERPing a rotation matrix causes scale skewing to occur
    - The numbers no longer represent sine/cosine outputs
  - This causes the mesh to collapse as joints are rotated
Linearity Blending Transforms

- Benefit of linear blend skinning: quick and easy to compute
- Problem with linear blend skinning: does not preserve volume very well
  - Example: Twisting an arm causes it to collapse because the vertex connections become crossed
Dual Quaternion Skinning

- Solution to mesh collapse: blend joint transformations using dual quaternions
- Dual quaternions are *dual numbers*
  - Of the form $a + \varepsilon b$, where $\varepsilon^2 = 0$
  - If $a$ and $b$ are quaternions, we get $q_a + \varepsilon q_b$
  - $q_a$ represents the rotation portion of a transformation, as we discussed previously
  - $\varepsilon q_b$ represents the translation portion of a transformation
  - The entire expression $q_a + \varepsilon q_b$ is a dual quaternion; $q_a$ and $q_b$ cannot be decoupled!
- Dual quaternion skinning was proposed by Dr. Ladislav Kavan et al. in their paper "Geometric Skinning with Approximate Dual Quaternion Blending"
Dual Quaternions

- Recall how to represent a rotation with a quaternion:
  \[ \begin{bmatrix} \cos(\theta/2), \sin(\theta/2)v_x, \sin(\theta/2)v_y, \sin(\theta/2)v_z \end{bmatrix} \]
  - \( \theta \) is the angle of rotation and \( v \) is the axis of rotation

- We can represent a translation with the quaternion \( q_t = [w \ i \ j \ k] \) given a rotation quaternion \( q_r = [q_w \ q_x \ q_y \ q_z] \) and a translation \( t = \langle x \ y \ z \rangle \):
  - \( w = \frac{1}{2} ( xq_x - yq_y - zq_z ) \)
  - \( i = \frac{1}{2} ( xq_w + yq_z - zq_y ) \)
  - \( j = \frac{1}{2} ( -xq_z + yq_w + zq_x ) \)
  - \( k = \frac{1}{2} ( xq_y - yq_x + zq_w ) \)

- The dual part of the dual quaternion can also be written as:
  - \( q_t = 1 + \varepsilon ( t_x i/2 + t_y j/2 + t_z k/2 ) \) where \( i, j, k \) are complex
  - Recall: \( q_r = \cos(\theta/2) + \sin(\theta/2)v_x i + \sin(\theta/2)v_y j + \sin(\theta/2)v_z k \)
Dual Quaternions

- We can combine these two types of quaternion to get a dual quaternion that represents a rotation followed by a translation.
- Recall that a quaternion can be converted to a rotation matrix as follows:

\[
\begin{pmatrix}
    w^2 + x^2 - y^2 - z^2 & 2xy - 2wz & 2xz + 2wy & 0 \\
    2xy + 2wz & w^2 - x^2 + y^2 - z^2 & 2yz - 2wx & 0 \\
    2xz - 2wy & 2yz + 2wx & w^2 - x^2 - y^2 + z^2 & 0 \\
    0 & 0 & 0 & 1
\end{pmatrix}
\]
Dual Quaternions

- We can compute a translation vector from \(q_t = [w, i, j, k]\) and \(q_r = [q_w, q_x, q_y, q_z]\) like so:
  - \(t_x = \frac{2(wq_x + iq_w - jq_z + kq_y)}{\text{length}(q_r)}\)
  - \(t_y = \frac{2(wq_y + iq_z - jq_w + kq_x)}{\text{length}(q_r)}\)
  - \(t_z = \frac{2(wq_z + iq_y - jq_x + kq_w)}{\text{length}(q_r)}\)
- Our final output is the rotation matrix computed in the previous slide with \([t_x, t_y, t_z, 1]^T\) as its fourth column
Blending Dual Quaternions

- Blending dual quaternions is very simple
  - Given an interpolation parameter \( t = [0,1] \), the blending of dual quaternions \( q_a \) and \( q_b \) can be computed as:

\[
DLB(t; \hat{q}_1, \hat{q}_2) = \frac{(1-t)\hat{q}_1 + t\hat{q}_2}{\| (1-t)\hat{q}_1 + t\hat{q}_2 \|}
\]

- It is important that the result be normalized in order to prevent unwanted scaling from occurring
  - Preventing scaling is, after all, one of the main features of dual quaternion skinning
- Also want to find the shortest rotation interpolation, so if \( \text{dot}(q_1, q_2) < 0 \), use \(-q_1\) in this function instead
We can also extend this interpolation of two dual quaternions to a case with $N$ dual quaternions $q_n$ with varying influence weights $w_n$:

$$DLB(w; \hat{q}_1, \ldots, \hat{q}_n) = \frac{w_1 \hat{q}_1 + \ldots + w_n \hat{q}_n}{\|w_1 \hat{q}_1 + \ldots + w_n \hat{q}_n\|}$$
Linear vs Dual Quaternion Blend

Linear

Dual quaternion
Blend Shapes

- Allows one to manually transform each vertex of the mesh to the desired position
- Is most optimally used in conjunction with skeleton-based animation
  - Facial expressions
  - Muscle contraction
  - Small surface detail
Blend Shapes: Implementation

- Store a list of position offsets in each vertex
  - The offsets of all vertices at a given index are one blend shape
- Each blend shape has a modifiable weight in the range \([0,1]\)
  - At weight 1, the vertices are offset by 100% of the offset in a given blend shape
  - Multiple blend shapes can be linearly combined simply by adding their offsets together
Blend Shapes: Implementation

<table>
<thead>
<tr>
<th>Vertex 1</th>
<th>Vertex 2</th>
<th>Vertex 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>&lt;1,1,0&gt;</td>
<td>Position</td>
</tr>
<tr>
<td>Blendshape 1</td>
<td>&lt;1,0,0&gt;</td>
<td>Blendshape 1</td>
</tr>
<tr>
<td>Blendshape 2</td>
<td>&lt;0,0,0&gt;</td>
<td>Blendshape 2</td>
</tr>
</tbody>
</table>

Base mesh
Blendshape 1
Blendshape 2
Blendshape 1+2