Rotations, Vertex Blending and Morphing

Dr. Sheng Bin (盛斌)
Shanghai Jiao Tong University
Lecture 6

Polynesian Migration

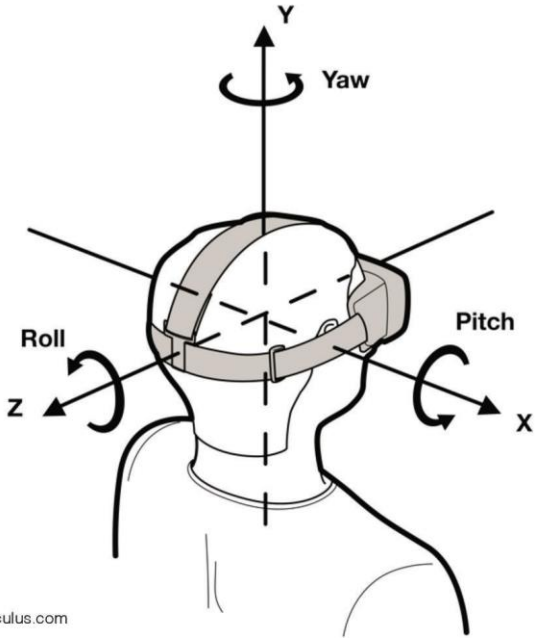
wikipedia

Lecture Overview

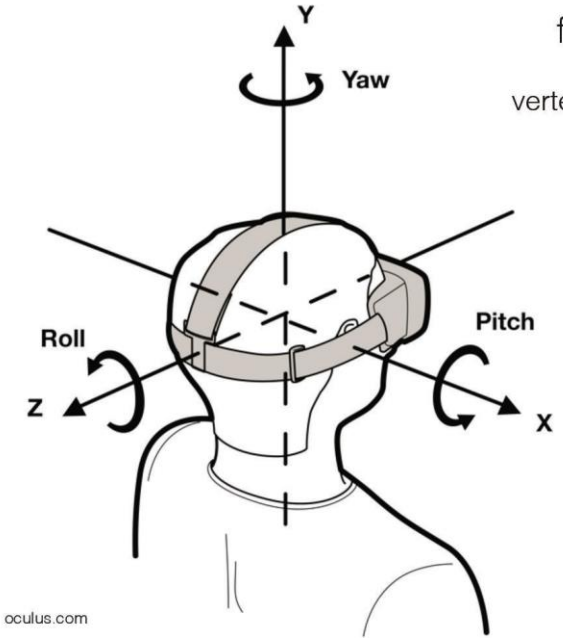
- short review of coordinate systems, tracking in flatland, and accelerometer-only tracking

- rotations: Euler angles, axis & angle, gimbal lock
- rotations with quaternions
- 6-DOF IMU sensor fusion with quaternions

- Vertex Blending
- Morphing

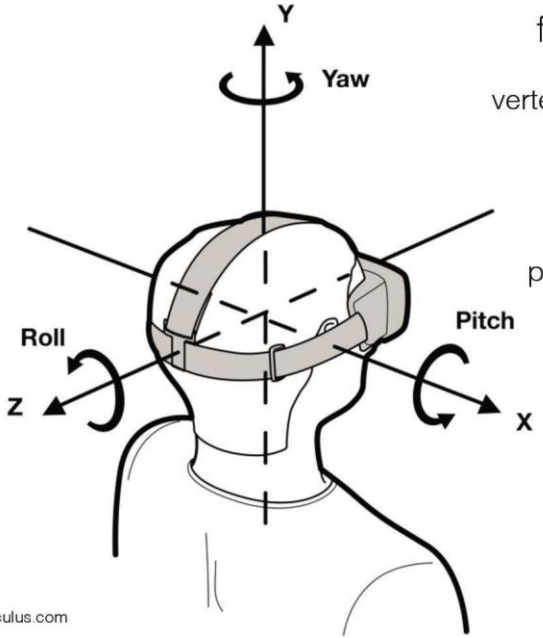


- primary goal: track orientation of head or device
- inertial sensors required to determine pitch, yaw, and roll



from lecture 2:

vertex in clip space ↓ $v_{clip} = M_{proj} \cdot M_{view} \cdot M_{model} \cdot v$ ↓ vertex



from lecture 2:

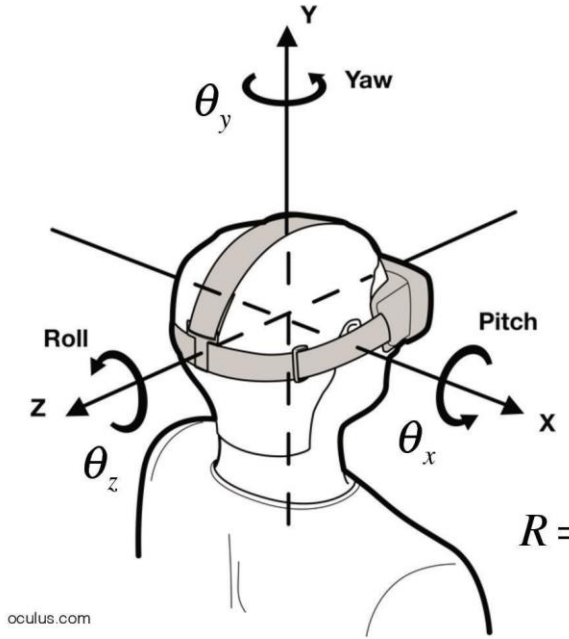
vertex in clip space ← vertex

$$v_{clip} = M_{proj} \cdot M_{view} \cdot M_{model} \cdot v$$

projection matrix · view matrix · model matrix

rotation · translation

$$M_{view} = R \cdot T(-eye)$$



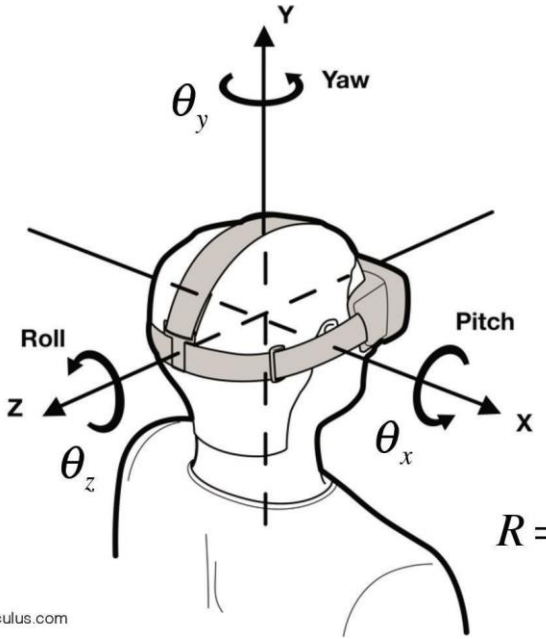
Euler angles

rotation translation

$$M_{view} = R \cdot T(-eye)$$

$$R = R_z(-\theta_z) \cdot R_x(-\theta_x) \cdot R_y(-\theta_y)$$

roll pitch yaw



Euler angles

2 important coordinate systems:

body/sensor frame world/inertial frame

$$M_{view} = R \cdot T(-eye)$$

$$R = \underbrace{R_z(-\theta_z)}_{\text{roll}} \cdot \underbrace{R_x(-\theta_x)}_{\text{pitch}} \cdot \underbrace{R_y(-\theta_y)}_{\text{yaw}}$$

Euler Angles and Gimbal Lock

- so far we have represented head rotations with Euler angles: 3 rotation angles around the axis applied in a specific sequence

- problematic when interpolating between rotations in keyframes (in computer animation) or integration → singularities

Gimbal Lock

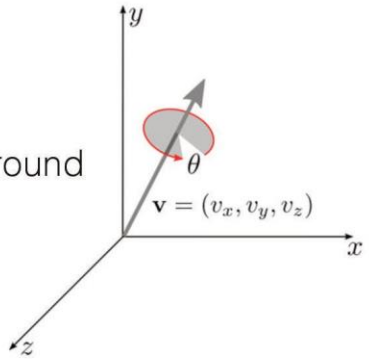
Play the video

The Guerrilla CG Project, The Euler (gimbal lock) Explained – see youtube.com

Rotations with Axis-Angle Representation and Quaternions

Rotations with Axis and Angle Representation

- solution to gimbal lock: use axis and angle representation for rotation!
- simultaneous rotation around a ***normalized*** vector ***v*** by angle θ
- no “order” of rotation, all at once around that vector



Quaternions

- think about quaternions as an extension of complex numbers to having 3 (different) imaginary numbers or fundamental quaternion units *i,j,k*

$$q = q_w + iq_x + jq_y + kq_z$$

$$i \neq j \neq k$$

$$i^2 = j^2 = k^2 = ijk = -1$$

$$ij = -ji = k$$

$$ki = -ik = j$$

$$jk = -kj = i$$

Quaternions

- think about quaternions as an extension of complex numbers to having 3 (different) imaginary numbers or fundamental quaternion units *i,j,k*

$$q = q_w + iq_x + jq_y + kq_z$$

- quaternion algebra is well-defined and will give us a powerful tool to work with rotations in axis-angle representation in practice

Quaternions

- axis-angle to quaternion (need normalized axis ***v***)

$$q(\theta, v) = \underbrace{\cos\left(\frac{\theta}{2}\right)}_{q_w} + \underbrace{i v_x \sin\left(\frac{\theta}{2}\right)}_{q_x} + \underbrace{j v_y \sin\left(\frac{\theta}{2}\right)}_{q_y} + \underbrace{k v_z \sin\left(\frac{\theta}{2}\right)}_{q_z}$$

Quaternions

- axis-angle to quaternion (need normalized axis *v*)

$$q(\theta, v) = \underbrace{\cos\left(\frac{\theta}{2}\right)}_{q_w} + \underbrace{i\, v_x \sin\left(\frac{\theta}{2}\right)}_{q_x} + \underbrace{j\, v_y \sin\left(\frac{\theta}{2}\right)}_{q_y} + \underbrace{k\, v_z \sin\left(\frac{\theta}{2}\right)}_{q_z}$$

- valid rotation quaternions have unit length

$$||q|| = \sqrt{q_w^2 + q_x^2 + q_y^2 + q_z^2} = 1$$

Two Types of Quaternions

- vector quaternions represent 3D points or vectors $\boldsymbol{u}=(u_x,u_y,u_z)$ can have arbitrary length

$$q_u = 0 + iu_x + ju_y + ku_z$$

- valid rotation quaternions have unit length

$$||q|| = \sqrt{q_w^2 + q_x^2 + q_y^2 + q_z^2} = 1$$

Quaternion Algebra

- quaternion addition:

$$q+p=(q_w+p_w)+i(q_x+p_x)+j(q_y+p_y)+k(q_z+p_z)$$

- quaternion multiplication:

$$\begin{aligned} qp &= (q_w+iq_x+jq_y+kq_z)(p_w+ip_x+jp_y+kp_z) \\ &= (q_wp_w-q_xp_x-q_yp_y-q_zp_z)+ \\ &\quad i(q_wp_x+q_xp_w+q_yp_z-q_zp_y)+ \\ &\quad j(q_wp_y-q_xp_z+q_yp_w+q_zp_x)+ \\ &\quad k(q_wp_z+q_xp_y-q_yp_x+q_zp_w)+ \end{aligned}$$

Quaternion Algebra

- quaternion conjugate: $q^* = q_w - iq_x - jq_y - kq_z$
- quaternion inverse: $q^{-1} = \frac{q^*}{||q||^2}$

- rotation of vector quaternion q_u by q : $q'_u = qq_uq^{-1}$
- inverse rotation: $q_u = q^{-1}q'_u q$

- successive rotations by q_1 then q_2 : $q'_u = q_2 q_1 q_u q_1^{-1} q_2^{-1}$

Quaternion Algebra

- detailed derivations and reference of general quaternion algebra and rotations with quaternions in course notes

- please read ***course notes*** for more details!

Spherical Linear Interpolation

- Spherical linear interpolation is an operation that, given two unit quaternions, q^ and ^r, and a parameter t ∈ [0,1], computes an interpolated quaternion. This is useful for animating objects, for example. It is not as useful for interpolating camera orientations, as the camera's "up" vector can become tilted during the interpolation, usually a disturbing effect.
- The algebraic form of this operation is expressed by the composite quaternion, ^s, below:

$$\hat{\mathbf{s}}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = (\hat{\mathbf{r}}\hat{\mathbf{q}}^{-1})^t \hat{\mathbf{q}}.$$

Spherical Linear Interpolation

- However, for software implementations, the following form, where **slerp** stands for **spherical linear interpolation**, is much more appropriate:

$$\hat{\mathbf{s}}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = \texttt{slerp}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = \frac{\sin(\phi(1-t))}{\sin\phi}\hat{\mathbf{q}} + \frac{\sin(\phi t)}{\sin\phi}\hat{\mathbf{r}}.$$

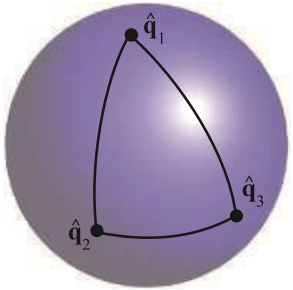
Spherical Linear Interpolation

- To compute φ, which is needed in this equation, the following fact can be used: $\cos\varphi = q_x r_x + q_y r_y + q_z r_z + q_w r_w$. For t ∈ [0,1], the slerp function computes (**unique**[2]) interpolated quaternions that together constitute the shortest arc on a four-dimensional unit sphere from $\hat{\mathbf{q}}$ ($t = 0$) to $\hat{\mathbf{r}}$ ($t = 1$). The arc is located on the circle that is formed from the intersection between the plane given by $\hat{\mathbf{q}}$, $\hat{\mathbf{r}}$, and the origin, and the four-dimensional unit sphere.

$$\hat{\mathbf{s}}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = \mathtt{slerp}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = \frac{\sin(\phi(1-t))}{\sin\phi}\hat{\mathbf{q}} + \frac{\sin(\phi t)}{\sin\phi}\hat{\mathbf{r}}.$$

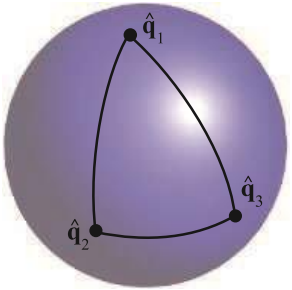
Spherical Linear Interpolation

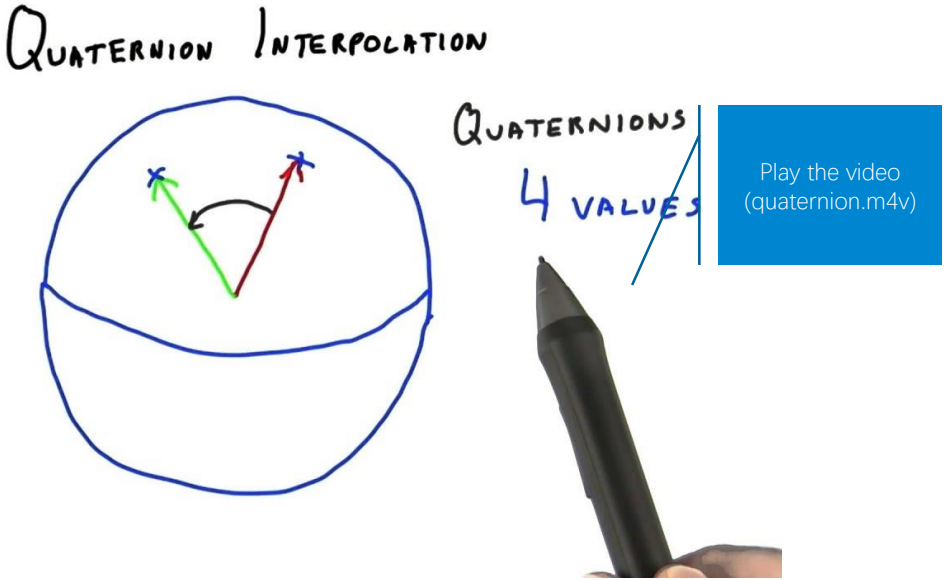
- This is illustrated in the figure below. The computed rotation quaternion rotates around a fixed axis at constant speed. A curve such as this, that has constant speed and thus zero acceleration, is called a *geodesic curve*. A *great circle* on the sphere is generated as the intersection of a plane through the origin and the sphere, and part of such a circle is called a *great arc*.



Spherical Linear Interpolation

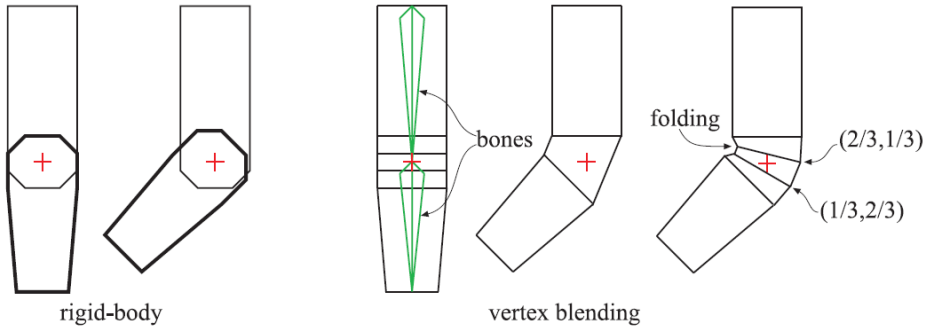
- Unit quaternions are represented as points on the unit sphere. The function slerp is used to interpolate between the quaternions, and the interpolated path is a great arc on the sphere. Note that interpolating from q^1 to q^2 and interpolating from q^1 to q^3 to q^2 are not the same thing, even though they arrive at the same orientation.




QUATERNION INTERPOLATION
QUATERNIONS
4 VALUES
Play the video
(quaternion.m4v)

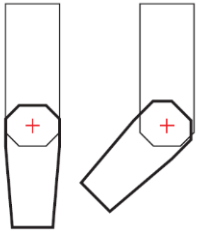
Vertex Blending

- Imagine that an arm of a digital character is animated using two parts, a forearm and an upper arm, as shown below.

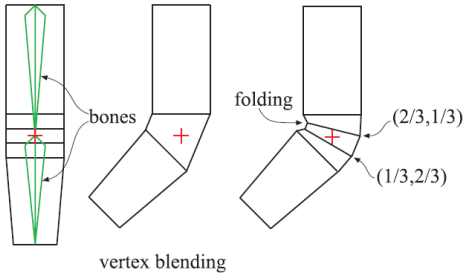


Vertex Blending

- This model could be animated using rigid-body transforms.
- However, then the joint between these two parts will not resemble a real elbow. This is because two separate objects are used, and therefore, the joint consists of **overlapping** parts from these two separate objects.
- Clearly, it would be better to use just one single object.
- However, static model parts do not address the problem of making the joint flexible.

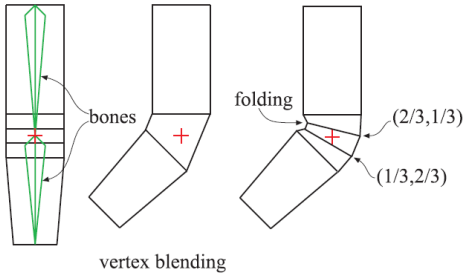


Vertex Blending



- Vertex blending is one popular solution to this problem.
- In its simplest form, the forearm and the upper arm are animated separately as before, but at the joint, the two parts are connected through an elastic “skin”.
- So, this elastic part will have one set of vertices that are transformed by the forearm matrix and another set that are transformed by the matrix of the upper arm.

Vertex Blending



- This results in triangles whose vertices may be transformed by different matrices, in contrast to using a single matrix per triangle.
- By taking this one step further, one can allow a single vertex to be transformed by several different matrices, with the resulting locations weighted and blended together.

Vertex Blending

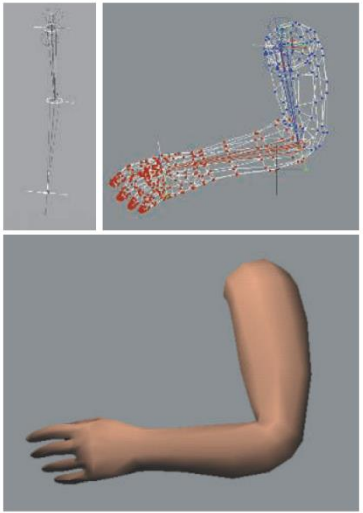


Figure 4.12. A real example of vertex blending. The top left image shows the two bones of an arm, in an extended position. On the top right, the mesh is shown, with color denoting which bone owns each vertex. Bottom: the shaded mesh of the arm in a slightly different position. *(Images courtesy of Jeff Lander [968].)*

- This is done by having a skeleton of bones for the animated object, where each bone's transform may influence each vertex by a user-defined weight (w_i).
- Since the entire arm may be "elastic," i.e., all vertices may be affected by more than one matrix, the entire mesh is often called a skin (over the bones).

$$\mathbf{u}(t) = \sum_{i=0}^{n-1} w_i \mathbf{B}_i(t) \mathbf{M}_i^{-1} \mathbf{p}, \quad \text{where} \quad \sum_{i=0}^{n-1} w_i = 1, \quad w_i \geq 0.$$

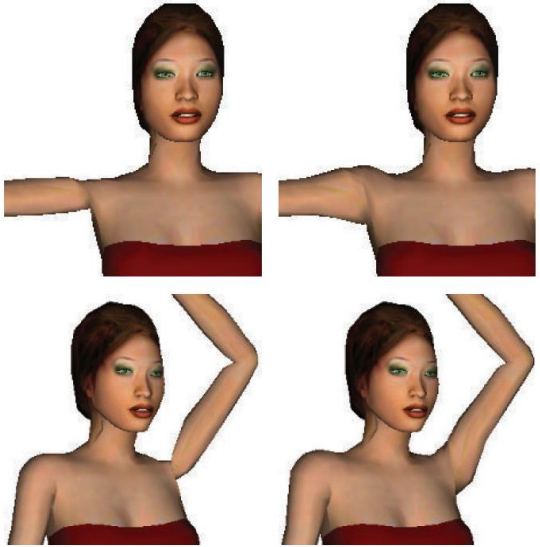
where **p** is the original vertex,
and **u(t)** is the transformed vertex whose position depends on time **t**.

Play the video
(missingblending.mp4)

Morphing

- Morphing from one three-dimensional model to another can be useful when performing animations.
- Imagine that one model is displayed at time t_0 and we wish it to change into another model by time t_1. For all times between t_0 and t_1, a continuous "**mixed**" model is obtained, using some kind of interpolation.

Morphing



- The left side shows problems at the joints when using *linear blend skinning*.
- On the right, blending using *dual quaternions* improves the appearance.

Morphing

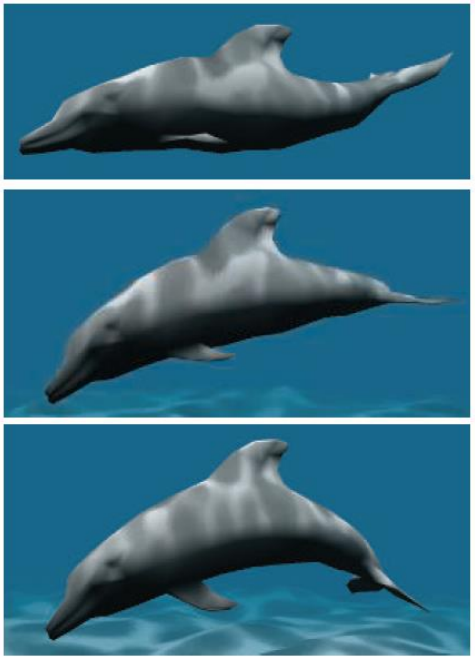
- Linear interpolation can be used directly on the vertices compute a morphed vertex for time $t \in [t0, t1]$, we first compute $s = (t - t_0)/(t_1 - t_0)$, and then the linear vertex blend,

$$\mathbf{m} = (1 - s)\mathbf{p}_0 + s\mathbf{p}_1,$$

where p_0 and p_1 correspond to the same vertex but at different times, t_0 and t_1.

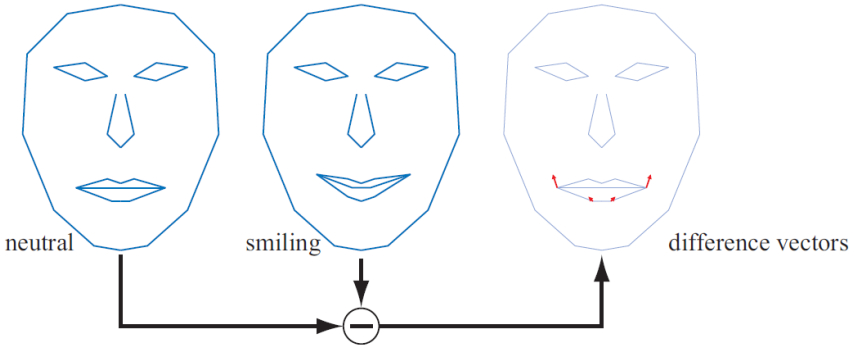
Morphing

- Vertex morphing.
- Two locations and normals are defined for every vertex.
- In each frame, the intermediate location and normal are linearly interpolated by the vertex shader.



Morphing

A variant of morphing where the user has more intuitive control is referred to as **morph targets** or **blend shapes**.



Given two mouth poses, a set of difference vectors is computed to control interpolation, or even extrapolation. In morph targets, the difference vectors are used to “add” movements onto the neutral face. With positive weights for the difference vectors, we get a smiling mouth, while negative weights can give the opposite effect.

Morphing

- We start out with a neutral model, which in this case is a face. Let us denote this model by **N**. In addition, we also have a set of different face poses. In the example illustration, there is only one pose, which is a smiling face. In general, we can allow **k** $\geqslant$ **1** different poses, which are denoted P_i, $i \in [1,\ldots,k]$.

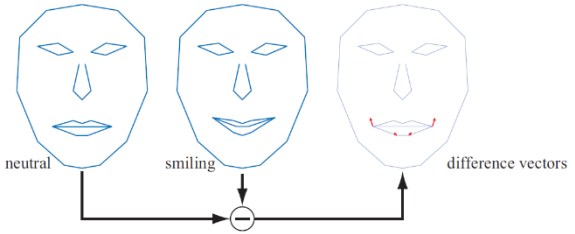
- As a preprocess, the "difference faces" are computed as:

$$D_i = P_i - N,$$

i.e., the neutral model is subtracted from each pose.

At this point, we have a neutral model, N, and a set of difference poses, D_i. A morphed model M can then be obtained using the following formula:

$$\mathcal{M} = \mathcal{N} + \sum_{i=1}^{k} w_i \mathcal{D}_i.$$



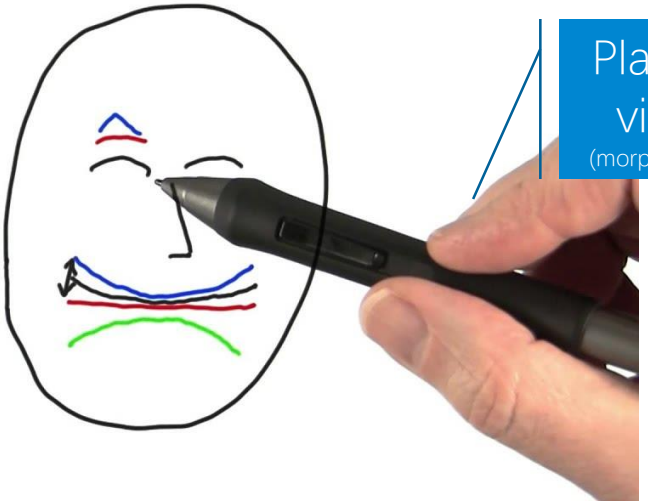
Morphing

The Delsin character's face, in inFAMOUS Second Son, is animated using *blend shapes*.

MORPHING

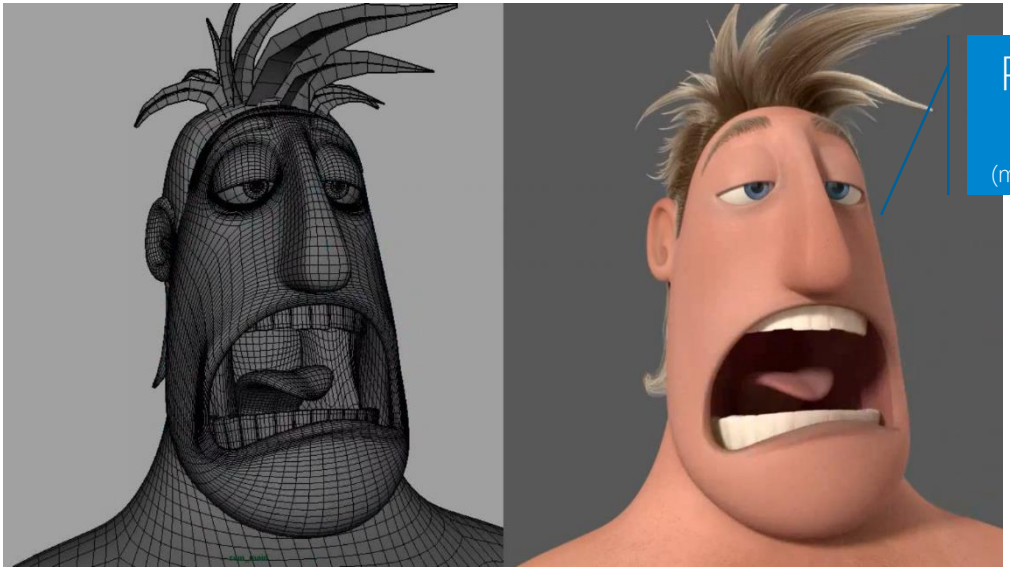
MORPH TARGETS

KEY POSES



Play the video
(morphing.mp4)

Morphing



Play the video
(morphing2.mp4)