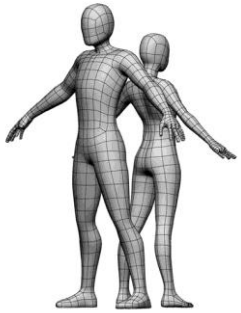
Mesh based Modeling and Animation

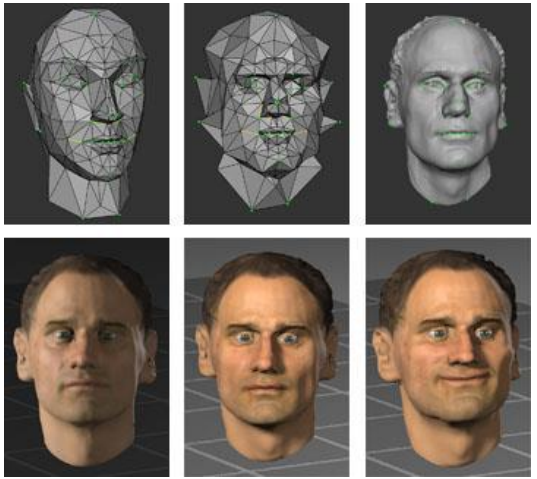
CMPUT 307

3D Modeling



Topics in computer graphics

- Imaging = representing 2D images
- Rendering = constructing 2D images from 3D models
- Modeling = representing 3D objects
- Animation = simulating changes over time



Using 3D skills, what can you do with them?

Making 2D animation. Eg: Futurama, American Dad.
TV and video.
Films and pre-visualization. Eg: Sintel
Stereoscopic 3D. Eg: Avatar movie
Web animation
Games
Fight and driving simulators
Digital signage
3d Printing
Architectural walkthroughs
Virtual Reality
Virtual sets
Interactive instruction

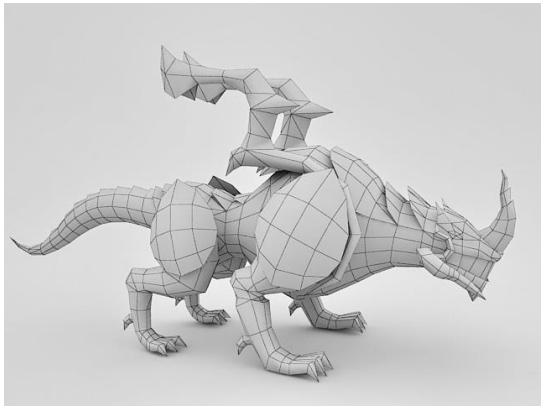


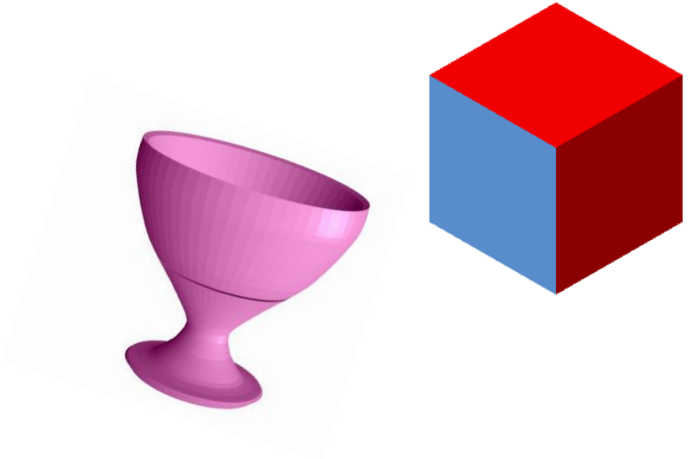





How do we represent 3D objects?

Using a collection of points in 3D space, connected by various geometric entities such as triangles, lines, curved surfaces, etc.





Representation

3d models can be divided in two categories:

- **Solid**: These models define the volume of the object they represent (like a rock) and are usually built with constructive solid geometry.

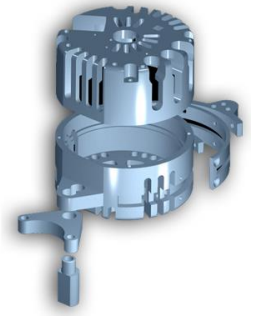
- **Shell/boundary**: these models represent the surface and must be manifold, e.g. the boundary of the object, not its volume (like an infinitesimally thin eggshell)

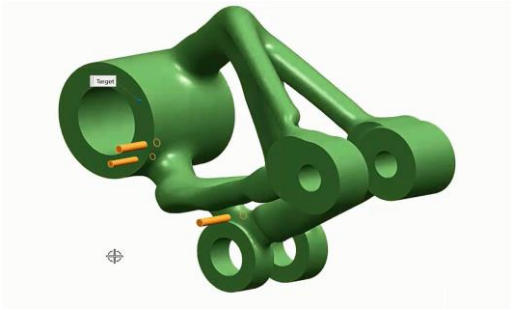
Differences

- *Solid* and *shell modeling* can create functionally identical objects.
- Differences between them are mostly variations in the way they are created and edited.
- Conventions of use in various fields.
- Differences in types of approximations between the model and reality.

Constructive solid geometry (CSG)

It is a solid representation technique that allows a modeler to create a complex surface or objects (called primitives) by using Boolean operators (union, intersection and difference) to combine simpler objects.

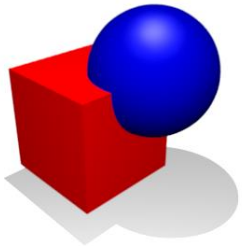




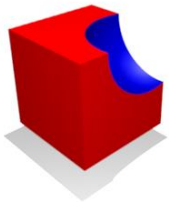
The geometry in solid modeling is fully described in 3-D space; objects can be viewed from any angle.

Example

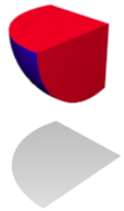
A sphere may be described by the coordinates of its center point, along with a radius value. These primitives can be combined into compound objects using operations like these:



Union

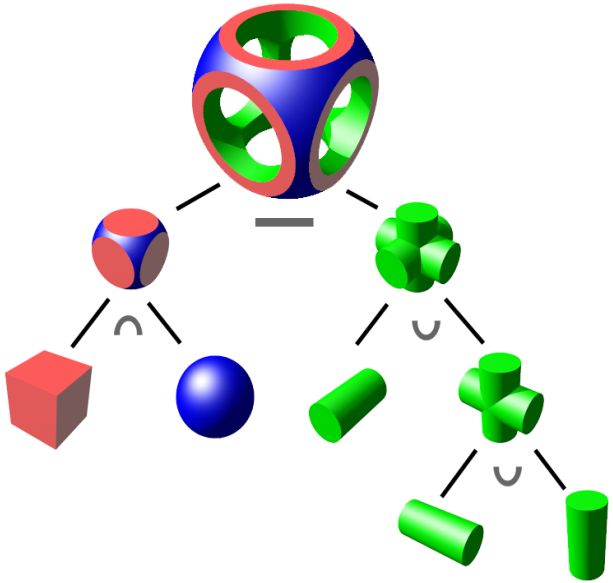


Difference



Intersection

CSG objects can be represented by binary trees, where leaves represent primitives, and nodes represent operations. In this figure, the nodes are labeled ∩ for intersection, ∪ for union, and − for difference.

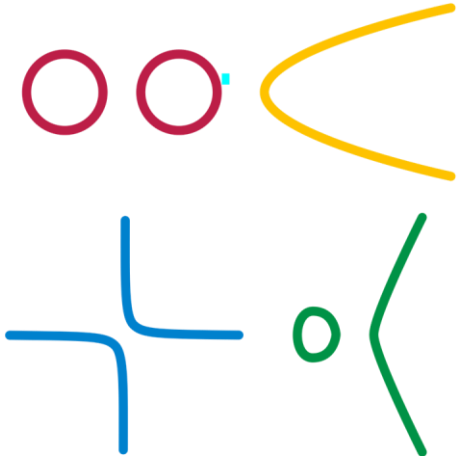


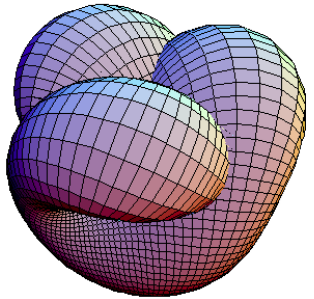
Shell/Boundary models must be manifold

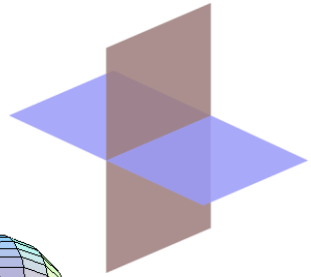
No having holes or cracks in the shell to be meaningful as a real object.

More precisely, each point of an *n*-dimensional manifold has a neighbourhood that is homeomorphic to the Euclidean space of dimension *n*.

- One dimensional
  - Lines and circles
- Two-dimensional (Surfaces)
  - Plane, sphere and torus.
- Three dimensional





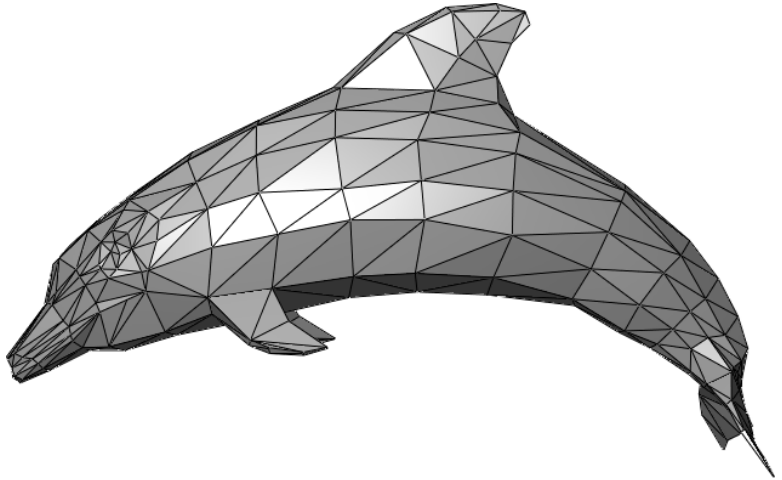


Polygonal meshes

A **polygon mesh** is a collection of **vertices**, **edges** and **faces** that defines the shape of a polyhedral object in 3D computer graphics and solid modeling.

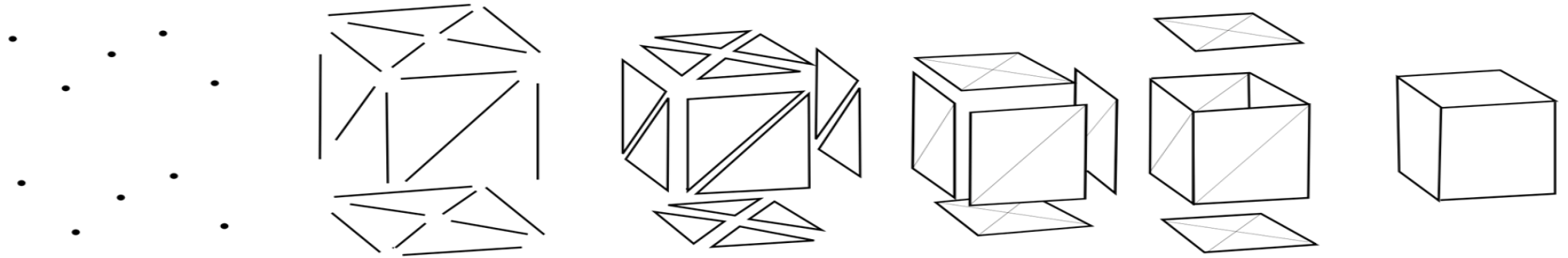
The faces usually consist of:

- Triangles (triangle mesh)
- Quadrilaterals
- Convex polygons
- Concave polygons
- Polygons with holes



Example of a triangle mesh representing a dolphin.

Polygon meshes elements



- **Vertex:** A position (usually in 3D space) along with other information such as color, normal vector and texture coordinates.
- **Edge**: A connection between two vertices.
- **Face**: A closed set of edges, in which a *triangle face* has three edges, and a *quad face* has four edges.
- **Surfaces:** More often called smoothing groups, are useful, but not required to group smooth regions.
- **Materials:** Generally materials will be defined, allowing different portions of the mesh to use different shaders when rendered.
- **UV coordinates**: are a separate 2d representation of the mesh "unfolded" to show what portion of a 2-dimensional texture map to apply to different polygons of the mesh.

Polygon meshes representations

Face - vertex meshes

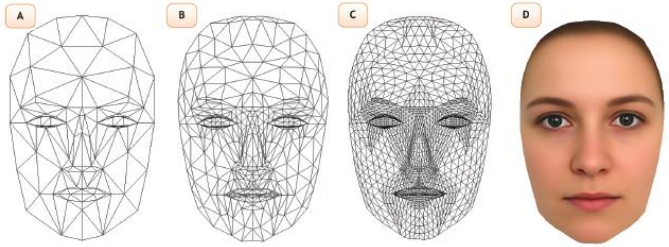
Winged-edge

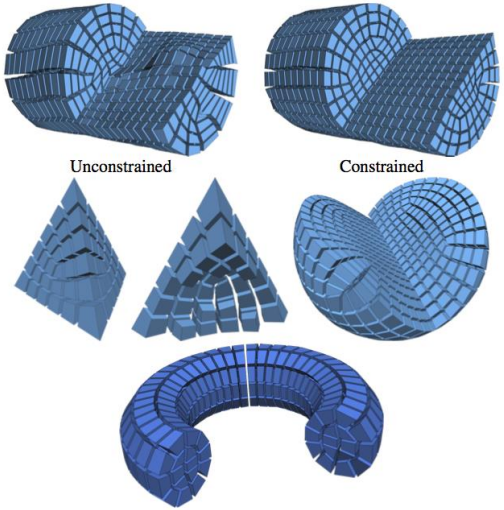
Half-edge meshes

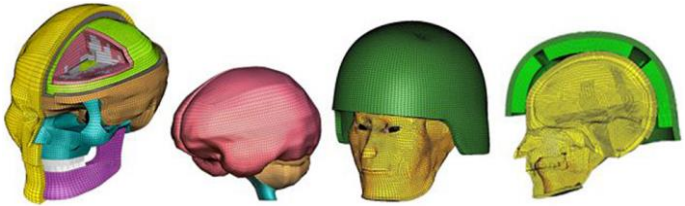
Quad-edge meshes

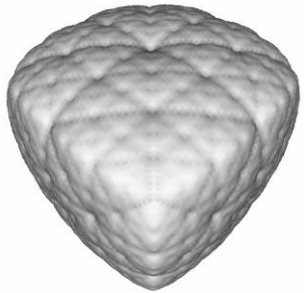
Corner-tables

Vertex-vertex meshes



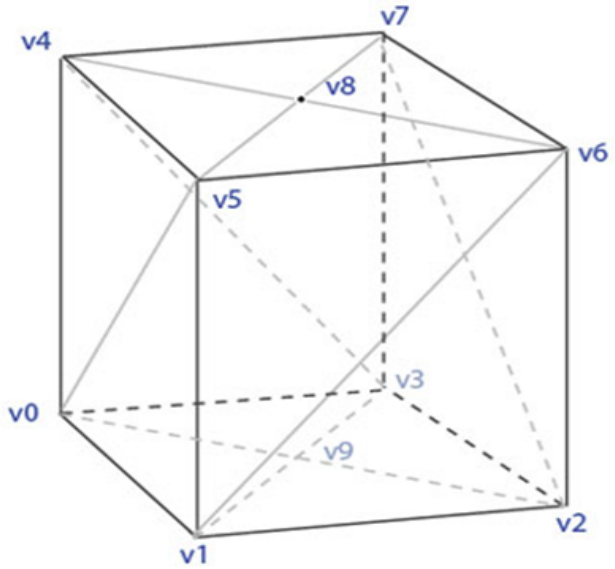






Vertex-Vertex Meshes (VV)

	Vertex List	
v0	0,0,0	v1 v5 v4 v3 v9
v1	1,0,0	v2 v6 v5 v0 v9
v2	1,1,0	v3 v7 v6 v1 v9
v3	0,1,0	v2 v6 v7 v4 v9
v4	0,0,1	v5 v0 v3 v7 v8
v5	1,0,1	v6 v1 v0 v4 v8
v6	1,1,1	v7 v2 v1 v5 v8
v7	0,1,1	v4 v3 v2 v6 v8
v8	.5,.5,1	v4 v5 v6 v7
v9	.5,.5,0	v0 v1 v2 v3



Represent an object as a set of vertices connected to other vertices.

Face-Vertex Meshes

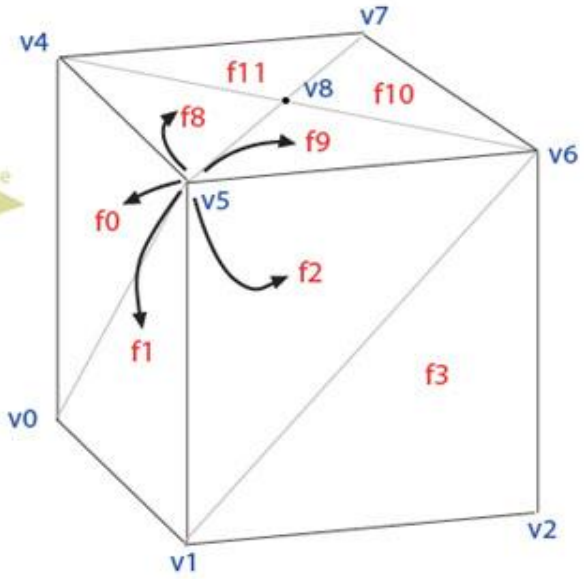
Face List

f0	v0 v4 v5
f1	v0 v5 v1
f2	v1 v5 v6
f3	v1 v6 v2
f4	v2 v6 v7
f5	v2 v7 v3
f6	v3 v7 v4
f7	v3 v4 v0
f8	v8 v5 v4
f9	v8 v6 v5
f10	v8 v7 v6
f11	v8 v4 v7
f12	v9 v5 v4
f13	v9 v6 v5
f14	v9 v7 v6
f15	v9 v4 v7

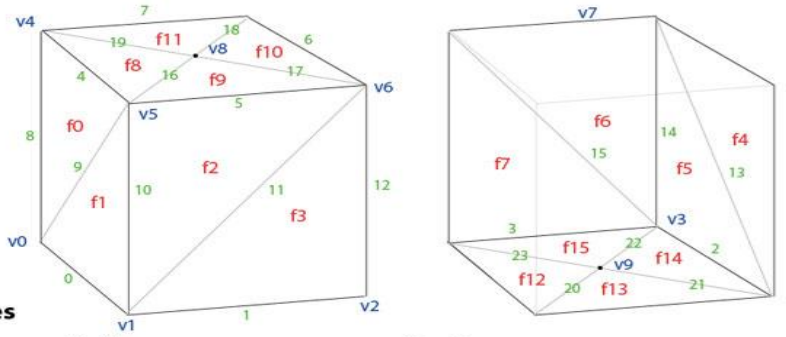
Vertex List

v0	0,0,0	f0 f1 f12 f15 f7
v1	1,0,0	f2 f3 f13 f12 f1
v2	1,1,0	f4 f5 f14 f13 f3
v3	0,1,0	f6 f7 f15 f14 f5
v4	0,0,1	f6 f7 f0 f8 f11
v5	1,0,1	f0 f1 f2 f9 f8
v6	1,1,1	f2 f3 f4 f10 f9
v7	0,1,1	f4 f5 f6 f11 f10
v8	.5,.5,0	f8 f9 f10 f11
v9	.5,.5,1	f12 13 14 15

example



Face-vertex meshes represent an object as a set of faces and a set of vertices. This is the most widely used mesh representation. Vertex v5 is highlighted to show the faces that surround it.



Winged-Edge Meshes

Face List

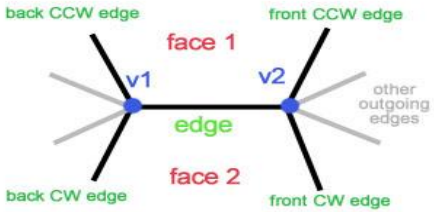
f0	4	8	9
f1	0	10	9
f2	5	10	11
f3	1	12	11
f4	6	12	13
f5	2	14	13
f6	7	14	15
f7	3	8	15
f8	4	16	19
f9	5	17	16
f10	6	18	17
f11	7	19	18
f12	0	23	20
f13	1	20	21
f14	2	21	22
f15	3	22	23

Edge List

e0	v0	v1	f1	f12	9 23 10 20
e1	v1	v2	f3	f13	11 20 12 21
e2	v2	v3	f5	f14	13 21 14 22
e3	v3	v0	f7	f15	15 22 8 23
e4	v4	v5	f0	f8	19 8 16 9
e5	v5	v6	f2	f9	16 10 17 11
e6	v6	v7	f4	f10	17 12 18 13
e7	v7	v4	f6	f11	18 14 19 15
e8	v0	v4	f7	f0	3 9 7 4
e9	v0	v5	f0	f1	8 0 4 10
e10	v1	v5	f1	f2	0 11 9 5
e11	v1	v6	f2	f3	10 1 5 12
e12	v2	v6	f3	f4	1 13 11 6
e13	v2	v7	f4	f5	12 2 6 14
e14	v3	v7	f5	f6	2 15 13 7
e15	v3	v4	f6	f7	14 3 7 15
e16	v5	v8	f8	f9	4 5 19 17
e17	v6	v8	f9	f10	5 6 16 18
e18	v7	v8	f10	f11	6 7 17 19
e19	v4	v8	f11	f8	7 4 18 16
e20	v1	v9	f12	f13	0 1 23 21
e21	v2	v9	f13	f14	1 2 20 22
e22	v3	v9	f14	f15	2 3 21 23
e23	v0	v9	f15	f12	3 0 22 20

Vertex List

v0	0,0,0	8	9	0	23	3
v1	1,0,0	10	11	1	20	0
v2	1,1,0	12	13	2	21	1
v3	0,1,0	14	15	3	22	2
v4	0,0,1	8	15	7	19	4
v5	1,0,1	10	9	4	16	5
v6	1,1,1	12	11	5	17	6
v7	0,1,1	14	13	6	18	7
v8	.5,.5,0	16	17	18	19	
v9	.5,.5,1	20	21	22	23	



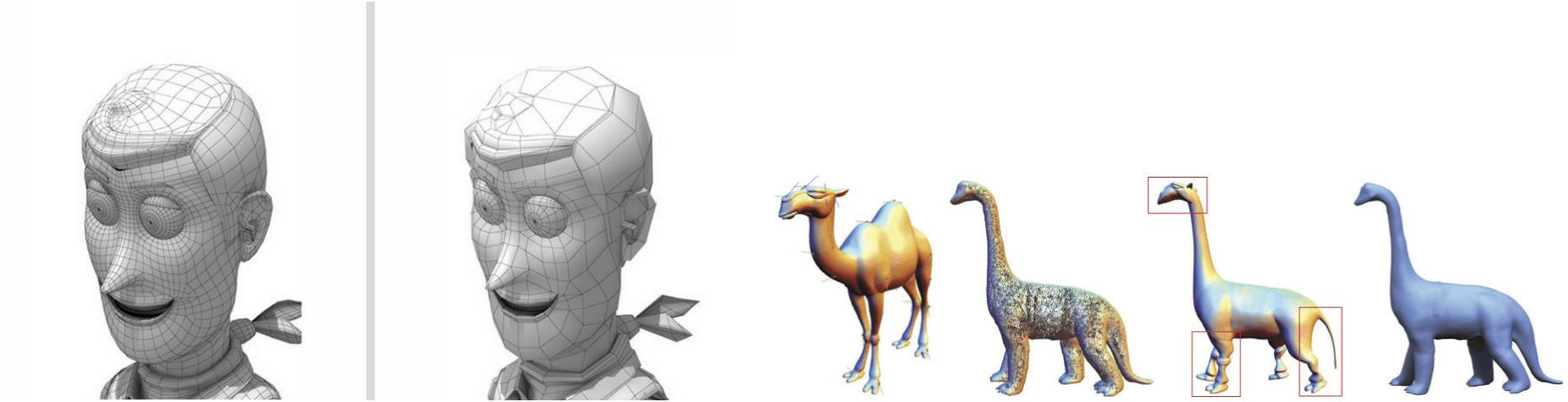
Winged Edge Structure

Introduced by Baumgart 1975, **winged-edge meshes** explicitly represent the vertices, faces, and edges of a mesh. This representation is widely used in modeling programs to provide the greatest flexibility in dynamically changing the mesh geometry, because split and merge operations can be done quickly.

CW: Clockwise

CCW: Counter Clockwise

Mesh Animation



Basic surface deformation methods

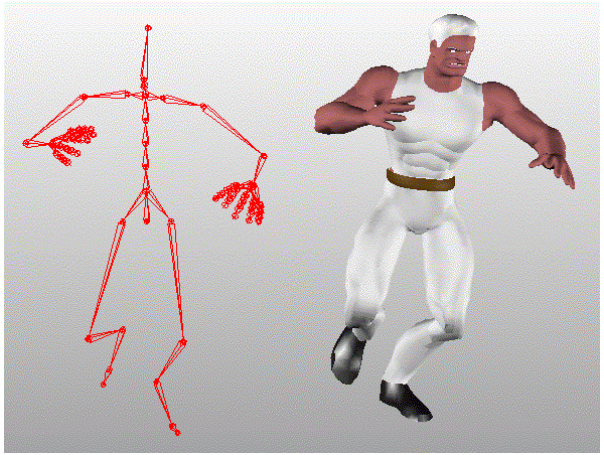
Blend Shapes : make a mesh by combining several meshes

Mesh skinning: deform a mesh based on an underlying skeleton

Both use simple linear algebra

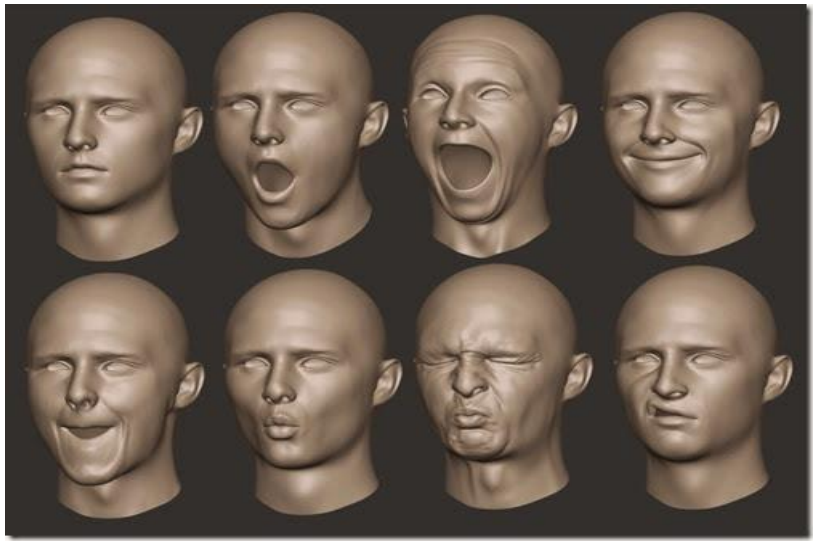
- Easy to implement - first thing to try
- Fast to run - used in games

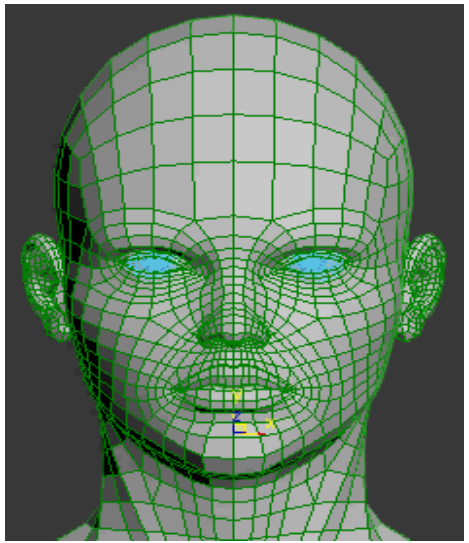
The simplest tools in the offline animation toolbox

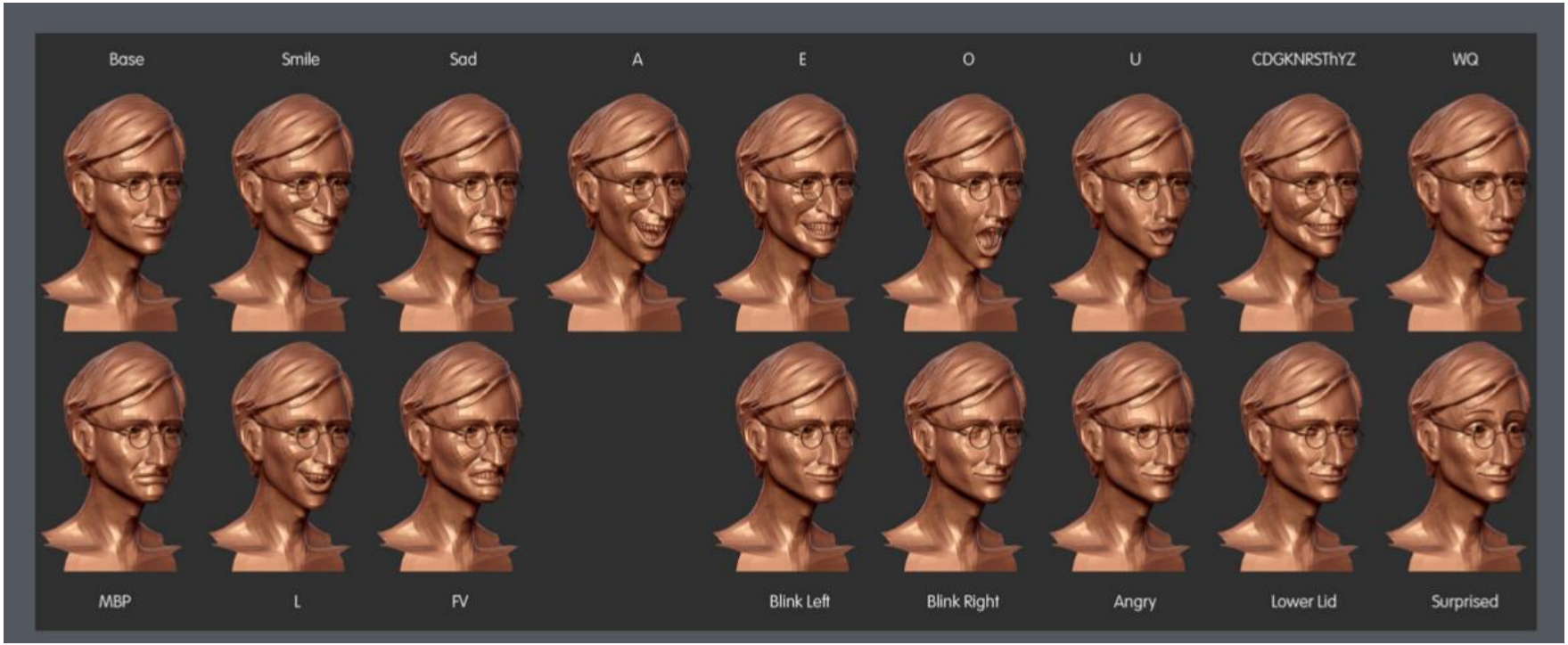


Blend Shapes

Blend shapes also called Morph target animation, per-vertex animation, shape interpolation or shape keys is a method of 3D computer animation used together with techniques such as skeletal animation.






Base
Smile
Sad
A
E
O
U
CDGKNRSThYZ
WQ
MBP
L
FV
Blink Left
Blink Right
Angry
Lower Lid
Surprised

Skeleton-Based Deformations

Skinning/Enveloping

-Need to infer how skin deforms from bone transformations.

-Most popular technique: ***Skeletal Subspace Deformation (SSD)***, or **simply Skinning**. Other aliases:

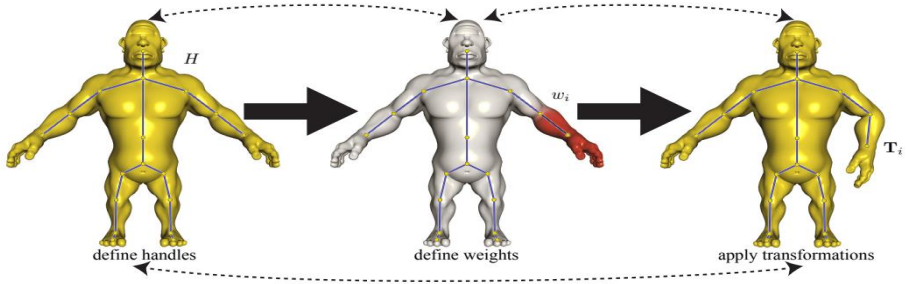
- vertex blending
- matrix palette skinning
- linear blend skinning

Skeleton-Based Deformations

The most common approach for deforming articulated character's skin is to define the surface geometry as a function of an underlying skeletal structure. The challenge is to obtain high-quality skin deformations.

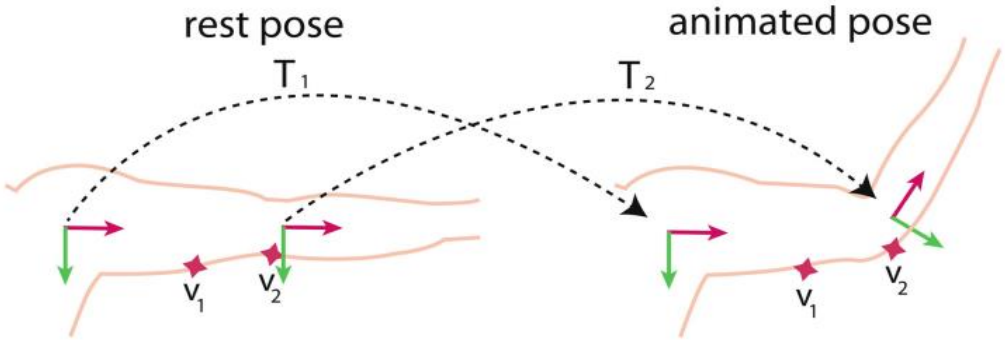
The current skeleton-based deformation techniques can be divided into two sections:

1. Geometry-based methods (also called smooth skinning)
2. Example-based methods



Geometric Skinning Techniques

- **Linear Blend skinning(LBS)** also called Skeleton subspace deformation (SSD) is the most popular technique due to its effectiveness, simplicity, and efficiency.
  - In LBS, the basic operation is to deform the skin according to a given list of bone transformations.



An example illustrates the main concept of LBS. There are two transformations T1 and T2, corresponding to the transformations of shoulder and elbow joints from the rest pose to an animated posture.

LBS Requires the following input:

- Surface mesh: a 3D model represented as a polygon mesh, where only vertex positions will change deformations.
- Bone transformations, representing the current deformation using a list of matrices $C_1, \dots, C_m \in \mathbb{R}^{4\times4}$

Mesh skinning math: setup

Surface has control points p_i

Triangle vertices, spline control points, subdivide base vertices

Each bone has a transformation matrix M_j

Normally a rigid motion

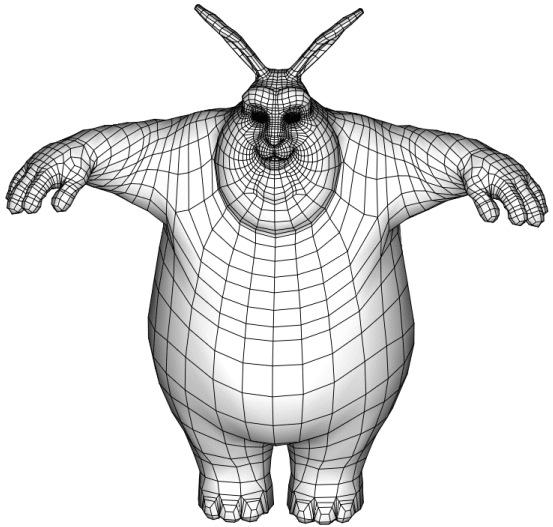
Every point-bone pair has a weight w_{ij}

In practice only nonzero for small # of nearby bones

The weights are provided by the user

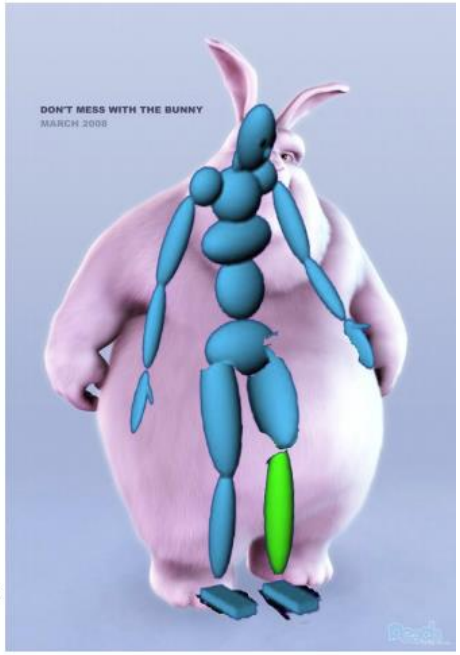
Points are transformed by a blended transformation

Various ways to blend exist



Skinning Characters

- Embed a skeleton into a detailed character mesh.
- Animate ‘bones’
  - Change the joint angles over time.
  - Keyframing, procedural, etc.
- Bind skin vertices to bones
  - Animate skeleton, skin will move with it
  - Bone transformations



LBS/Skinning

- Each bone has a deformation of the space around it (rotation, translation)
  - What if we attach each vertex of the skin to a single bone?
    - Skin will be rigid, except at joints where it will stretch badly
  - Let's attach a vertex to many bones at once!
    - In the middle of a limb, the skin points follow the bone rotation (near-rigidly)
    - At a joint, skin is deformed according to a "weighted combination" of the bones

Linear blend skinning

Assume p joints are associated to local coordinate system in a rest pose.

The transformation from rest-pose of joint $\boldsymbol{j} \in \{1, \dots, p\}$ to its actual position in the animated posture can be expressed by a rigid transformation matrix $\boldsymbol{C_j}$.

Then assume that vertex $\boldsymbol{v}$ is attached to joints j with weights $\boldsymbol{w} = (w_1, \dots, w_n)$.

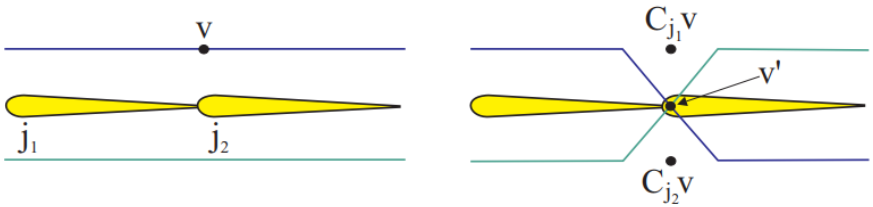
The weight w_i represents the influence of j_i on vertex v

The vertex position in the mesh deformed by linear blend skinning is computed as:

$$v' = \sum_{i=1}^{n} w_i C_{j_i} v$$

Linear blend skinning

- Consider a very simple arm rig with only two joints: $j_1$corresponding to the shoulder and j_2 to the elbow joint(see figure below).



Vertex v in the figure is equally influenced by both joints, i.e., $w_1 = w_2 = 0.5$ in order to achieve smooth skinning.
Let us further assume that the arm is animated by twisting joint j_2 by 180 degrees around the x-axis. Then the matrix M is written as

$$C_{j_1} = \begin{pmatrix} I & 0 \\ 0 & 1 \end{pmatrix}, C_{j_2} = \begin{pmatrix} R_x(180°) & 0 \\ 0 & 1 \end{pmatrix}$$

I denotes the 3x3 identity matrix, R_x denotes de rotation about x-axis.

Linear blend skinning

- See that averaging $C_{j_1}v$ and $C_{j_2}v$ produces vertex v'exactly at the position of joint j_2, i.e., the skin collapses to a single point.

- See that averaging $C_{j_1}v$ and $C_{j_2}v$ produces vertex v'exactly at the position of joint j_2, i.e., the skin collapses to a single point.

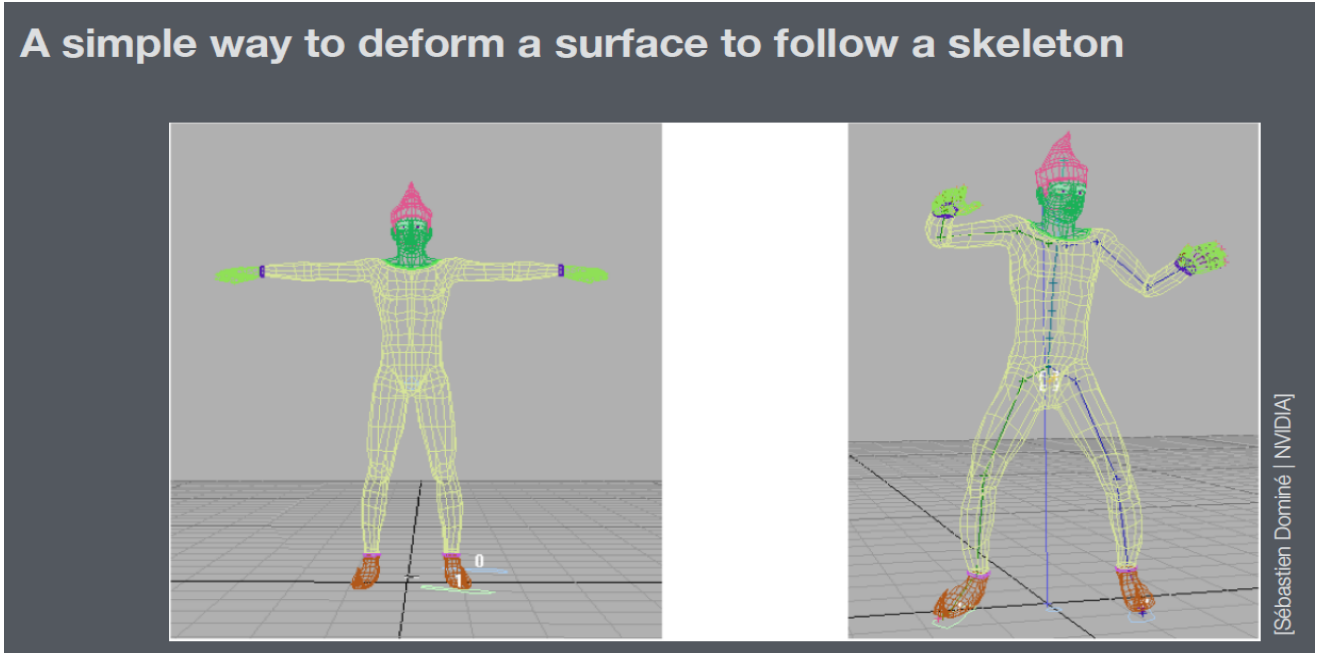
- To gain a better insight into linear blend skinning, the main equation can be re-written using distributivity of matrix-vector multiplication

$$v' = \sum_{i=1}^{n} w_i C_{j_i} v = \left(\sum_{i=1}^{n} w_i C_{j_i}\right) v$$

- To gain a better insight into linear blend skinning, the main equation can be re-written using distributivity of matrix-vector multiplication

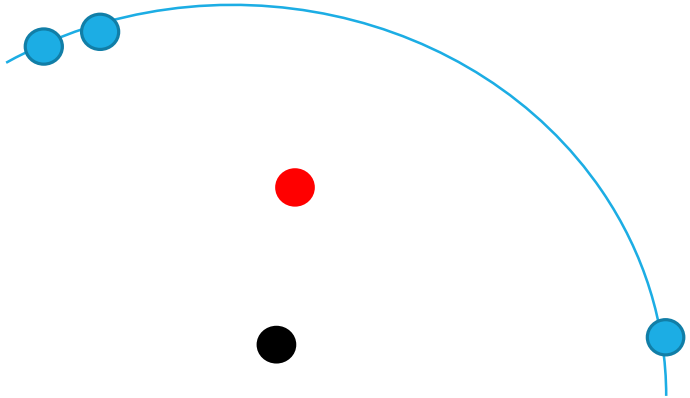
$$v' = \sum_{i=1}^{n} w_i C_{j_i} v = \left(\sum_{i=1}^{n} w_i C_{j_i}\right) v$$

Skinning of skeletally deformable models is extensively used for real-time animation of characters, creatures and similar objects. However the standard solution, linear blend skinning, has some serious drawbacks that require artist intervention.



LBS Artifacts (Candy Wrapper Effect)

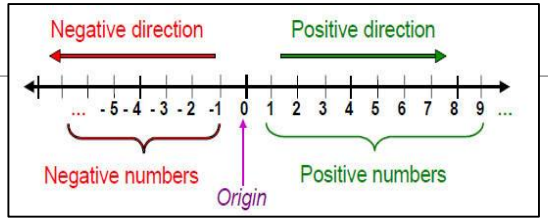
Cause of Candy Wrapper Effect



The weighted average of the 3 Blue points does NOT lie on the arc containing these 3 points. It is the Red point closer to the center of the arc.

A better approach: **Dual Quaternion Skinning**

Real numbers

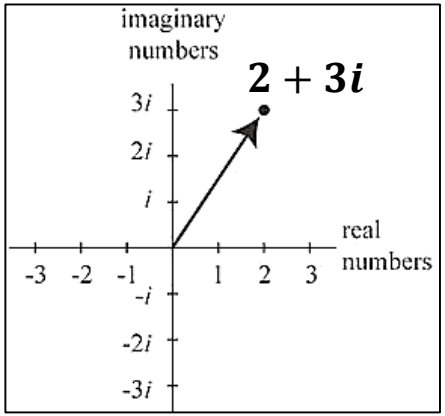


A complex number is a **two-dimensional** extension of the real numbers.

Complex numbers

By definition:

$i^2 = -1$

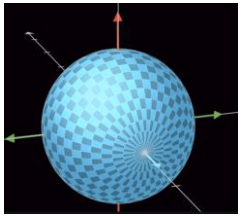


Quaternions

In1843 mathematician William Rowan Hamilton was out walking along the Royal Canal in Dublin with his wife when the solution in the form of the following equation suddenly occurred to him

$$\text{“}i^2 = j^2 = k^2 = ijk = -1\text{”}$$

He added two more imaginary dimensions to the complex plane.



$$4 + 6\boldsymbol{i} + 1\boldsymbol{j} + 9\boldsymbol{k}$$

Real part *Scalar*

Imaginary part *Vector*

A quaternion is a **four-dimensional** extension of the complex numbers. They have pragmatic utility describing rotation in 3D and even quantum mechanics.

Multiplication of basis elements

A feature of quaternions is that multiplication of two quaternions is noncommutative.

The products of basis elements are defined by $\boldsymbol{i^2 = j^2 = k^2 = -1}$, and:

$$\boldsymbol{ij = k,} \quad \boldsymbol{ji = -k}$$
$$\boldsymbol{jk = i,} \quad \boldsymbol{kj = -i}$$
$$\boldsymbol{ki = j} \quad \boldsymbol{ik = -j}$$

These multiplication formulas are equivalent to $\boldsymbol{i^2 = j^2 = k^2 = ijk = -1}$

In fact, the equality ijk = -1 results from

$$\boldsymbol{(ij)k = k^2 = -1}$$

The converse implication results from manipulations similar to the following. By right-multiplying both sides of $-1 = \boldsymbol{ijk}$ by $-\boldsymbol{k}$, one gets

$$\boldsymbol{k = (ijk)(-k) = (ij)(-k^2) = ij}$$

Quaternion Multiplication

$$(w_1 + x_1 i + y_1 j + z_1 k)(w_2 + x_2 i + y_2 j + z_2 k) =$$
$$(w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2) + (w_1 x_2 + x_1 w_2 + y_1 z_2 - z_1 y_2) i +$$
$$(w_1 y_2 + y_1 w_2 + z_1 x_2 - x_1 z_2) j + (w_1 z_2 + z_1 w_2 + x_1 y_2 - y_1 x_2) k$$

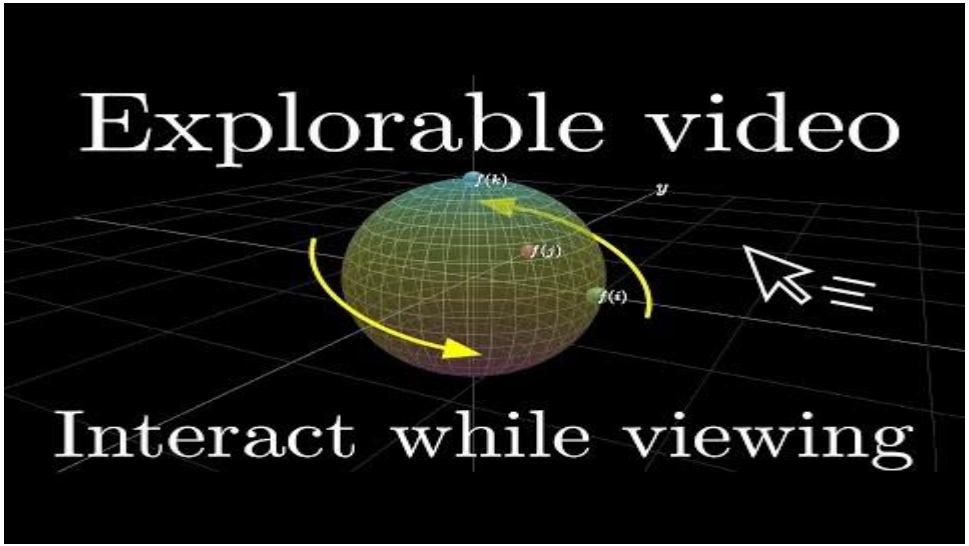
Multiplication rule written in terms of the dot product and the cross product.

$$\overrightarrow{v_1} = \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \qquad \overrightarrow{v_2} = \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix}$$

It is more common to represent the quaternion as two components, the vector component (x, y and z) and the scalar component (w)

$$(w_1 + x_1 i + y_1 j + z_1 k)(w_2 + x_2 i + y_2 j + z_2 k) = (w_1, \overrightarrow{v_1})(w_2, \overrightarrow{v_2}) =$$
$$(w_1 w_2 - \overrightarrow{v_1} \cdot \overrightarrow{v_2}, w_1 \overrightarrow{v_2} + w_2 \overrightarrow{v_1} + \overrightarrow{v_1} \times \overrightarrow{v_2})$$

Interactive explanation of quaternions



https://cs.gmu.edu/~jmlien/teaching/cs451/uploads/Main/dual-quaternion.pdf

Dual Numbers

Similar to complex number that consists of two parts known as the real part and dual or complex part.

$$\mathcal{Z} = a + b\mathcal{E}$$

Except that: $\mathcal{E}^2 = 0$ but $\mathcal{E} \neq 0$

where $\mathcal{E}$ is the dual operator, a is the real part and b is the dual part.

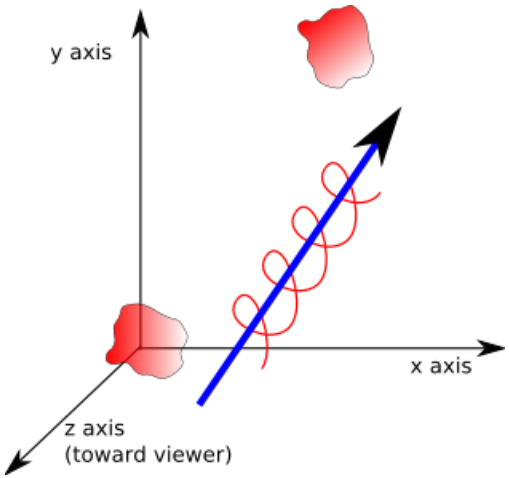
The dual operator $\mathcal{E}$ is added to distinguish the real and dual components.

The real part of a dual calculation is independent of the dual parts of the inputs.

The dual part of a multiplication is a "cross" product of real and dual parts.

Classical quaternions are restricted to the representation of rotations, whereas in graphical applications we typically work with rotation combined with translation, i.e. rigid transformations.

A **Dual Quaternion** can model the movement of a solid object in 3D, which can rotate and translate without changing shape; i.e., distances and angles between points on the object are preserved.



Dual Quaternion

Dual quaternions is composed of two quaternions, one responsible for orientation, and the other responsible for translation. Combining the algebra operations associated with quaternions with the additional dual number $\boldsymbol{\varepsilon}$, we can form the dual quaternion arithmetic.

Can be written as $\hat{q} = w + ix + jy + kz$ where w is the scalar part(dual number), (x, y, z) is the vector part (dual vector) and i, j, k are the usual quaternion units.

A dual quaternion can also be considered as an 8-tuple, or as the sum or two ordinary quaternions, $\hat{q} = q_0 + q_\varepsilon \cdot \varepsilon$

To transform a point using dual quaternion we use: $\mathbf{P = qPq^*}$ (where $\boldsymbol{q^*}$ denotes conjugate)

Dual quaternion skinning blends the dual quaternion of each bone by the blending weights

$$q = \frac{\sum_{i=1}^{n} w_i q_i}{\left\| \sum_{i=1}^{n} w_i q_i \right\|}$$

https://cs.gmu.edu/~jmlien/teaching/cs451/uploads/Main/dual-quaternion.pdf

Dual Quaternion

There are 8 elements, the 4 quaternion elements **(real, i, j and k)** and their duals **(ε, iε, jε and kε).** This gives dual quaternions a 8x8 multiplication table as shown here:

Multiplication table for dual quaternion units

×	**1**	***i***	***j***	***k***	**ε**	**ε*i***	**ε*j***	**ε*k***
1	1	*i*	*j*	*k*	ε	ε*i*	ε*j*	ε*k*
i	*i*	−1	*k*	−*j*	ε*i*	−ε	ε*k*	−ε*j*
j	*j*	−*k*	−1	*i*	ε*j*	−ε*k*	−ε	ε*i*
k	*k*	*j*	−*i*	−1	ε*k*	ε*j*	−ε*i*	−ε
ε	ε	ε*i*	ε*j*	ε*k*	0	0	0	0
ε*i*	ε*i*	−ε	ε*k*	−ε*j*	0	0	0	0
ε*j*	ε*j*	−ε*k*	−ε	ε*i*	0	0	0	0
ε*k*	ε*k*	ε*j*	−ε*i*	−ε	0	0	0	0

Dual Quaternion Calculation

D1 = (1, 2i, j, k) + (1, 2i, j, k) ε

D2 = (1, i, 3j, k) + (1, i, j, k) ε

D1 × D2 DQ part

D1 DQ part: (1, 2i, j, k)
D2 Q part: (1, i, 3j, k)

Use the right-hand rule and distribution property
Cross product:

1 × (1, i, 3j, k) = (1, i, 3j, k)
2i × (1, i, 3j, k) = (2i, -2, 6k, -2j)
j × (1, i, 3j, k) = (j, -k, -3, i)
k × (1, i, 3j, k) = (k, j, -3i, -1)

So:
(1, 2i, j, k) × (1, i, 3j, k) = (-5, i, 3j, 7k)

Dual Quaternion Calculation

D1 = (1, 2i, j, k) + (1, 2i, j, k) ε

D2 = (1, i, 3j, k) + (1, i, j, k) ε

D1 × D2 DQ part

D1 Q part: (1, 2i, j, k)
D2 DQ part: (1, i, j, k) ε
Similarly:
(1, 2i, j, k) × (1, i, j, k) ε = (-3, 3i, j, 3k) ε

D1 Q part: (1, 2i, j, k)
D2 Q part: (1, i, 3j, k)
Similarly:
(1, 2i, j, k) × (1, i, 3j, k) = (-5, i, 3j, 7k)

D1 dual DQ part: (1, 2i, j, k) ε
D2 dual DQ part: (1, i, j, k) ε
(1, 2i, j, k) $\boldsymbol{\varepsilon}$ × (1, i, j, k) $\boldsymbol{\varepsilon}$ = 0

Dual Quaternion Calculation

D1 = (1, 2i, j, k) + (1, 2i, j, k) ε (I made both Q & DQ parts the same here for simplicity, we will work on a more complicated example in class.)

D2 = (1, i, 3j, k) + (1, i, j, k) ε

Result:

D1 $\times$ D2 = (-5, i, 3j, 7k) + (-8, 4i, 4j, 10k) ε

Dual Quaternion for Only Rotation or Only Translation

The dual-quaternion can represent a pure rotation just as a quaternion by setting the dual part to zero.

$$q_r = \cos\left(\frac{\theta}{2}\right) + \sin(\frac{\theta}{2})(x * \boldsymbol{i} + y * \boldsymbol{j} + z * \boldsymbol{k}) + \varepsilon \cdot 0$$

$$= \left(\cos\left(\frac{\theta}{2}\right), n_x \, sin\left(\frac{\theta}{2}\right), n_y \sin\left(\frac{\theta}{2}\right), n_z \, sin\left(\frac{\theta}{2}\right)\right) + \varepsilon \cdot 0$$

$(x * \boldsymbol{i} + y * \boldsymbol{j} + z * \boldsymbol{k})$ specifies the axis for rotation.

To represent a pure translation with no rotation, the real part can be set to identity with the dual part representing translation.

$$q_t = (1,0,0,0) + \frac{\varepsilon}{2}\left(0, t_x, t_y, t_z\right)$$

A point with coordinates (x, y, z) in DQ would be (1, 0, 0, 0) $+(0, \boldsymbol{x} * \boldsymbol{i}, y * \boldsymbol{j}, z * \boldsymbol{k})\varepsilon$

Example 1

Rotate the vector **P** (0, 2, 0) = 2**j** , 90 degrees counter-clockwise about a vertical axis (parallel with k), then apply a translation (2,2,2). This is a simple example and you can easily get the result (0,2,2).

Let's see how it's done using dual quaternion.

Rotation part: $q_r = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right)(x * \boldsymbol{i} + y * \boldsymbol{j} + z * \boldsymbol{k}) = (\frac{\sqrt{2}}{2}, \boldsymbol{0i}, \boldsymbol{0j}, \frac{\sqrt{2}}{2}\boldsymbol{k})$

Transition part: $q_t = (1, 0\boldsymbol{i}, 0\boldsymbol{j}, 0\boldsymbol{k}) + (0, \boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k})\varepsilon$

Dual quaternion: $q = q_t q_r = \left(\frac{\sqrt{2}}{2}, \boldsymbol{0}, \boldsymbol{0}, \frac{\sqrt{2}}{2}\boldsymbol{k}\right) + (-\frac{\sqrt{2}}{2}, \sqrt{\boldsymbol{2}}\boldsymbol{i}, \boldsymbol{0}, \frac{\sqrt{2}}{2}\boldsymbol{k})\ \varepsilon$

P: $(\boldsymbol{1}, \boldsymbol{0}, \boldsymbol{0}, \boldsymbol{0}) + (\boldsymbol{0}, \boldsymbol{0}, \boldsymbol{2j}, \boldsymbol{0})\varepsilon$

Conjugate: $q^* = \left(\frac{\sqrt{2}}{2}, \boldsymbol{0}, \boldsymbol{0}, \frac{-\sqrt{2}}{2}\boldsymbol{k}\right) + (\frac{-\sqrt{2}}{2}, \sqrt{\boldsymbol{2}}\boldsymbol{i}, \boldsymbol{0}, \frac{\sqrt{2}}{2}\boldsymbol{k})\ \varepsilon$

P'$= qPq^* = (\boldsymbol{1}, \boldsymbol{0}, \boldsymbol{0}, \boldsymbol{0}) + (\boldsymbol{0}, \boldsymbol{0}, \boldsymbol{2j}, \boldsymbol{2k})\varepsilon$

Example 2

Rotate the vector **P1** (1, 2, 0) =(1, 0, 0, 0)+(0, i , 2j, 0) ε , 60 degrees right about a vertical axis (parallel with k).

The axis of the quaternion must be vertical, and must point down in order to represent a rightward rotation by the right hand rule (because when you point your right thumb down, your fingers curl round to the right) thus q=(0,0,-1)=-k

Rotation part: $q_r = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right)(x * \boldsymbol{i} + y * \boldsymbol{j} + z * \boldsymbol{k})$

$q_r = \left(\cos\left(\frac{60°}{2}\right), 0, 0, -sin\left(\frac{60°}{2}\right)\right)$

$q_r = q = (0.87\ ,\ 0,\ \ 0, -0.5) = 0.87 - 0.5\text{k}$

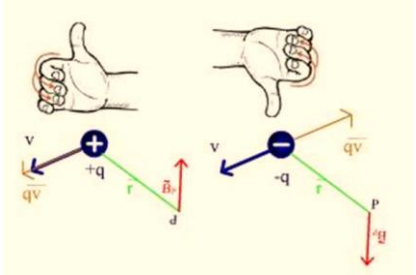
P2 = q P1 q^*

$\text{P2} = (0.87 - 0.5\text{k}) * ((1, 0, 0, 0)+(0, \text{i}, 2\text{j}, 0)\ \varepsilon) * (0.87 + 0.5\text{k})$

$\text{P2} = (1, 0, 0, 0) + (0.87\text{i} + 1.74\text{j} - 0.5\text{ki} - \text{kj})\varepsilon * (0.87 + 0.5\text{k})$

$\text{P2} = (1, 0, 0, 0) + (1.87\text{i}+1.24\text{j}) * (0.87 + 0.5\text{k})$

$\text{P2} = (1, 0, 0, 0) + (2.25\text{i}+0.14\text{j})\ \varepsilon$



Dual Quaternion Skinning

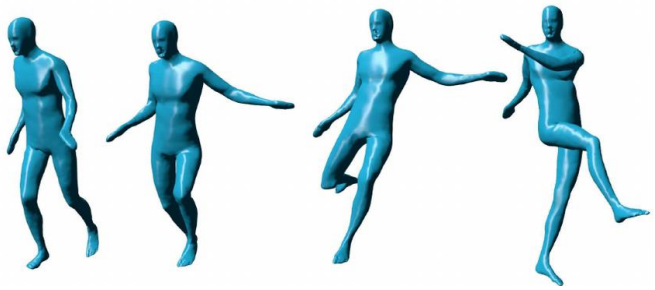
Combining the rotational and translational transforms into a single unit quaternion to represent a rotation followed by a translation we get:

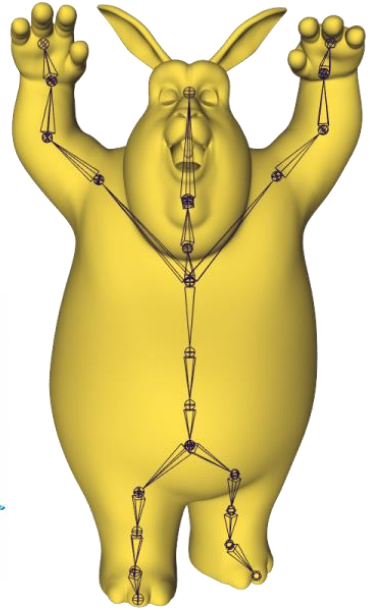
$$q = q_t q_r$$

Applying the dual quaternion q to a vertex v

$$\boldsymbol{v' = qvq^*}$$

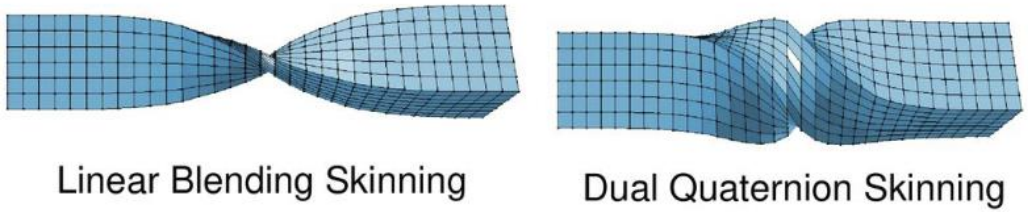
Where $\boldsymbol{q^*}$ is a conjugate of $\boldsymbol{q}$





Dual Quaternion Skinning

Joint collapse and candy wrap can be avoided using dual quaternion skinning



Homogeneous Mat VS DQ

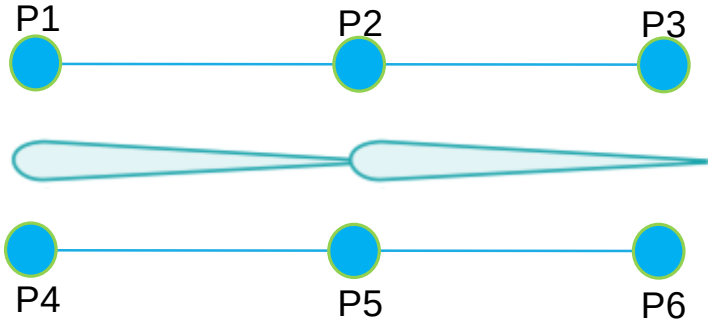
6 vertices:

P1: (0, 2.5, 0) P2: (1, 2.5, 0) P3: (2, 2.5, 0)

P4: (0, -2.5, 0) P5: (1, -2.5, 0) P6: (2, -2.5, 0)

2 joints:

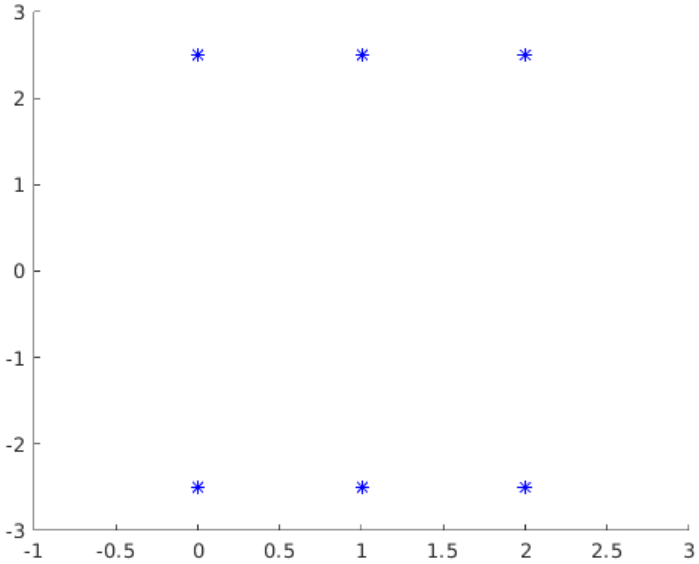
J1: (0, 0, 0) J2: (2, 0, 0)



Weight of every vertices to joints:

$W_{ij}=\begin{bmatrix} 1 & 0 \\ 0.5 & 0.5 \\ 0 & 1 \\ 1 & 0 \\ 0.5 & 0.5 \\ 0 & 1 \end{bmatrix}$ where i refers to the vertices and j refers to the joints.

Homogeneous Mat VS DQ



Homogeneous Mat VS DQ

Now we rotate J1 by 180 degrees, axis of rotation is x axis.

Homogeneous transformation matrix for J1:

$$M_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Homogeneous transformation matrix for J2:

$$M_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Homogeneous Mat VS DQ

Now we do linear blending using homogeneous matrix :

For every point:

$P_i' = \left(\sum_{j=1}^{n} w_{ij} M_j\right) P_i$

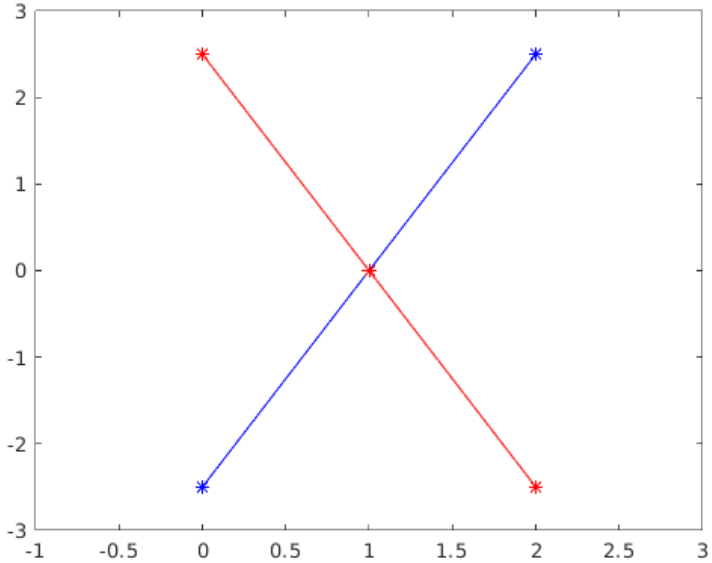
So after the transformation:

P1: (0, -2.5, 0) P2: (1, 0, 0) P3: (2, 2.5, 0)

P4: (0, 2.5, 0) P5: (1, 0, 0) P6: (2, -2.5, 0)

P2 and P5 collapsed to the same point, which will cause the candy wrapper effect.

Homogeneous Mat VS DQ



Homogeneous Mat VS DQ

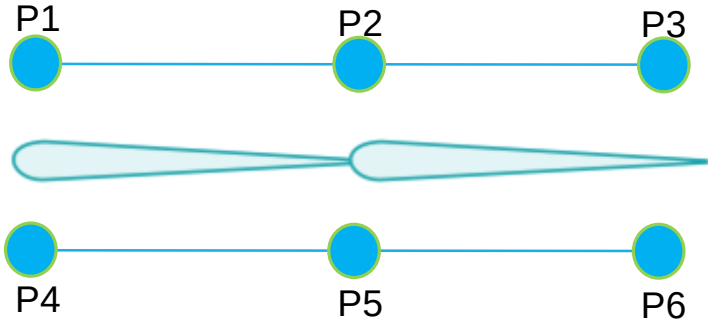
6 vertices:

P1: (0, 2.5, 0) P2: (1, 2.5, 0) P3: (2, 2.5, 0)

P4: (0, -2.5, 0) P5: (1, -2.5, 0) P6: (2, -2.5, 0)

2 joints:

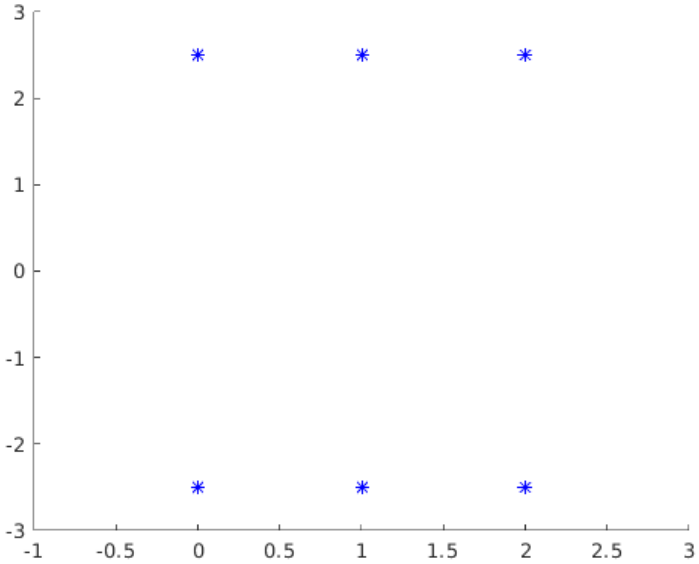
J1: (0, 0, 0) J2: (2, 0, 0)



Weight of every vertices to joints:

$W_{ij}=\begin{bmatrix} 1 & 0 \\ 0.5 & 0.5 \\ 0 & 1 \\ 1 & 0 \\ 0.5 & 0.5 \\ 0 & 1 \end{bmatrix}$ where i refers to the vertices and j refers to the joints.

Homogeneous Mat VS DQ



Homogeneous Mat VS DQ

Now we rotate J1 by 180 degrees, axis of rotation is x axis.

Dual quaternion for J1:

$Q_1 = (0, i, 0, 0) + 0\varepsilon$

Dual quaternion for J2:

$Q_2 = (1, 0, 0, 0) + 0\varepsilon$

Homogeneous Mat VS DQ

Now we do linear blending using homogeneous matrix :

For every point, the transformation DQ:

$$Q_i = \frac{\sum_{j=1}^{n} w_{ij} Q_j}{\left\| \sum_{j=1}^{n} w_{ij} Q_j \right\|}$$

For rotations, $\left\| \sum_{j=1}^{n} w_{ij}\, Q_j \right\|$ is the norm of the real part of $\sum_{j=1}^{n} w_{ij}\, Q_j$.

$$P_i' = Q_i^* P_i Q_i$$

So after the transformation:

P1: (0, -2.5, 0) P2: (1, 0, -2.5) P3: (2, 2.5, 0)

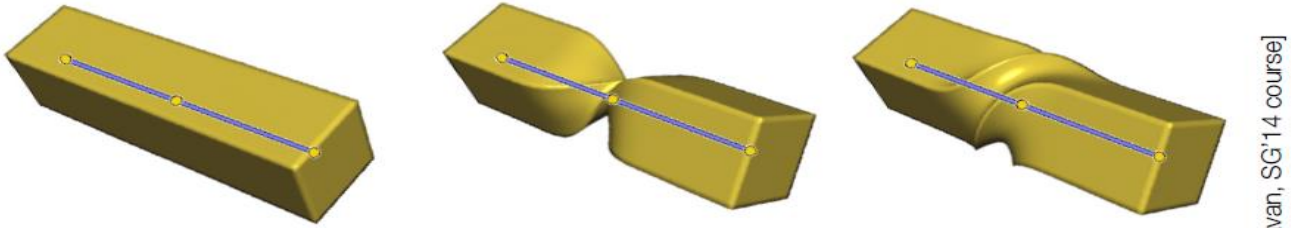
P4: (0, 2.5, 0) P5: (1, 0, 2.5) P6: (2, -2.5, 0)

P2 and P5 maintains the distance to each other, so there's no candy wrapper effect.

Example

Figure 14: Comparison of linear (left) and dual quaternion (right) blending. Dual quaternions preserve rigidity of input transformations and therefore avoid skin collapsing artifacts.

[Kavan et al. SG '08]

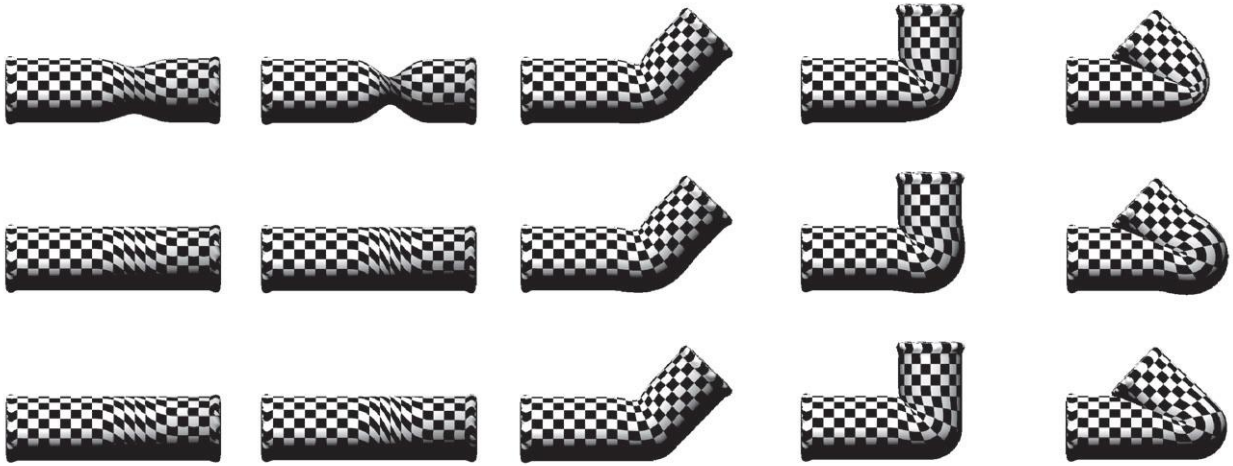


[Kavan, SG'14 course]

Limitations of Dual Quaternion Skinning (Bulging Artifact)

LBS top row, DQS middle row: DQS HAS BULGING ARTIFACT WHEN ELBOW IS BENT

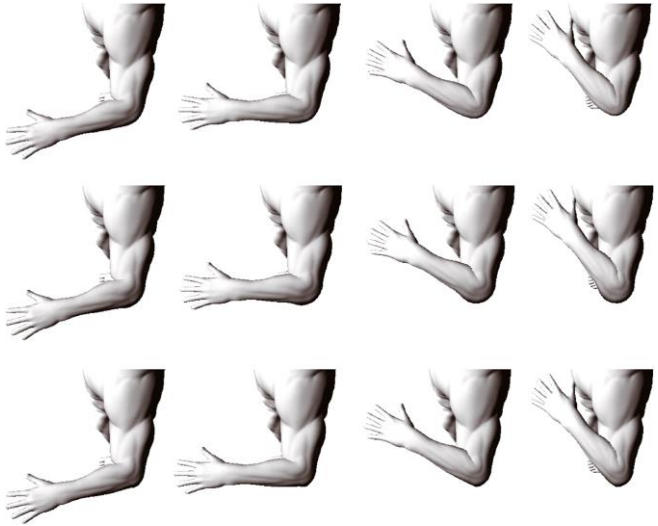
Bottom Row: An improved DQS algorithm tries to fix this problem



Limitations of Dual Quaternion Skinning (Bulging Artifact)

LBS top row, DQS middle row: DQS HAS BULGING ARTIFACT WHEN ELBOW IS BENT

Bottom Row: An improved DQS algorithm tries to fix this problem



Other Approaches to addressing problems with DQS

1. Differential Blending: Can Handle Turning Objects more than 360 degrees.
2. Elasticity Inspired Deformers: Seem to work well for the elbows.
3. Spline Skinning: Seems to work well for Bending a Pipe (Cylinder)

(More details can be found in Paper included in this weeks notes.)

Skinning Limitations

-All skinning methods assume a fixed relationship between skeleton motion and the mesh

-Humans are more complex (e.g., muscles lead to local deformations)

-Skinning only allows animation of predefined body geometry (created by an animator), do not help us create this geometry

To do better, probably we need to look at the Mechanics of Muscles & How they Control Movement; this should produce more realistic Human Animations

Some advanced Animation Tools (like Maya) allow users to choose the Skinning Methods or Combinations (like LBS & DQS) and their relative weights. This may be an alternative to consider as well.

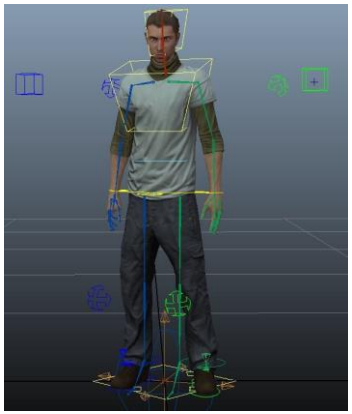
Skin Binding

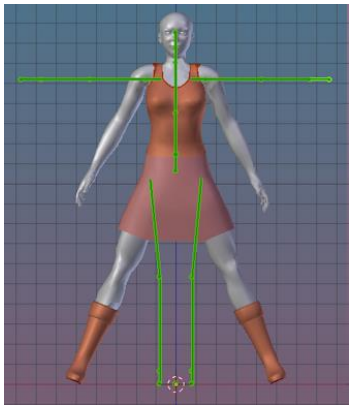
Attaching a skin to a skeleton is not a trivial problem and usually requires automated tools combined with interactive tuning.

Binding algorithms typically involve heuristic approaches

Some general approaches:

- Point-to-line mapping
- Containment
- Delaunay tetrahedralization





Automatic methods

Automatic Methods

In case of skeletal animation, the character mesh is provided with a **hierarchical skeleton**, that specifies how the **mesh** should be deformed in case of movement.

The process of forming the underlying skeleton and fitting it to the model is called **rigging**.

Rigging a character can be subdivided into two consecutive steps. The generation and placement of the skeleton inside the model, and the **skinning**, where it is specified how the mesh is deformed by the skeleton structure.

Rigging by hand, however, can be a rather daunting task for beginning 3D modelers, but also a time consuming job for expert animators, as each character model needs to be rigged separately.

The main idea of automatic rigging is that no user intervention is needed to rig the model.

Building an animated 3D character requires:

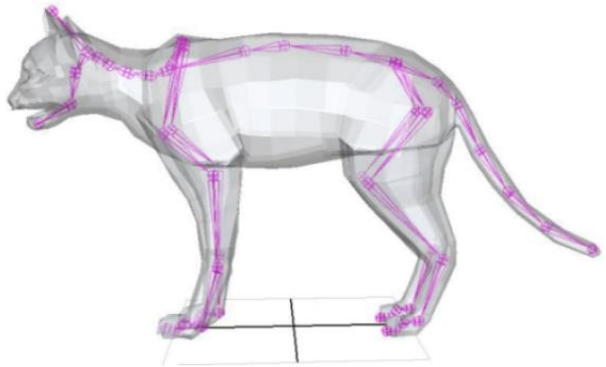
Create a 3D mesh

Set the 3D position and hierarchy of skeleton joints

Set the 3D orientation of skeleton joints

Create high-level controllers for animation (constrains)

Define skinning weights



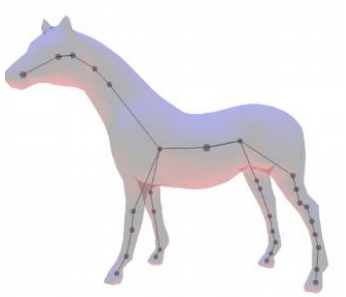
Skeleton computation

Nowadays:

- By hand
- By expert

Automatic methods:

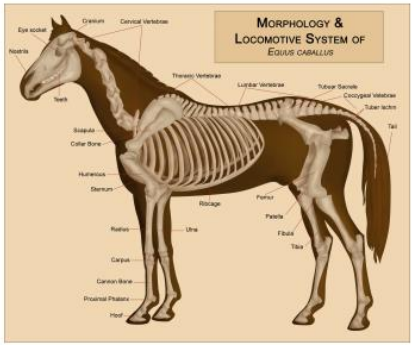
- Little control over the result
- Provide noisy skeletons (unwanted joints)
- Rely only on the geometry of the shape – not the anatomy of the model (bone structure)



[Liu PG'03]

[Lien SPM'06]

[Tierny PG'06]



Skeleton generation and placement

The first part of the rigging process, is done by one of two distinct approaches, namely **skeleton embedding** or **skeleton extraction.**

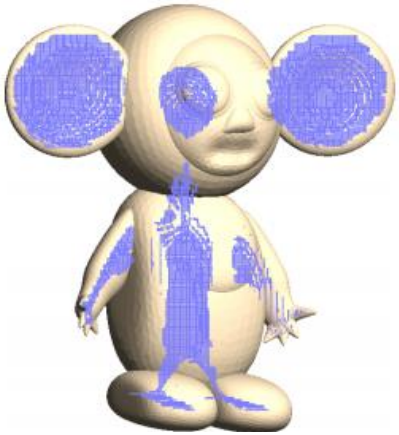
In the case of skeleton embedding, there is a skeleton template to be embedded into a character model, in some optimal way.

In the case of skeleton extraction, the skeleton is extracted by evaluating the interior of a model.

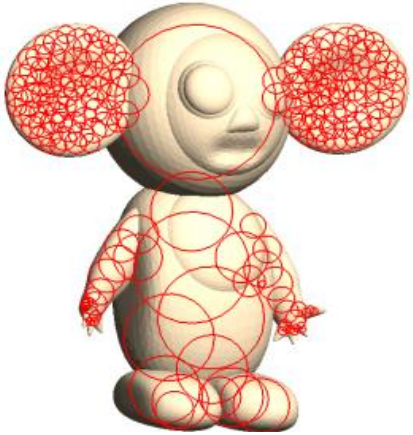
Skeleton embedding

To embed the skeleton in an optimal way,

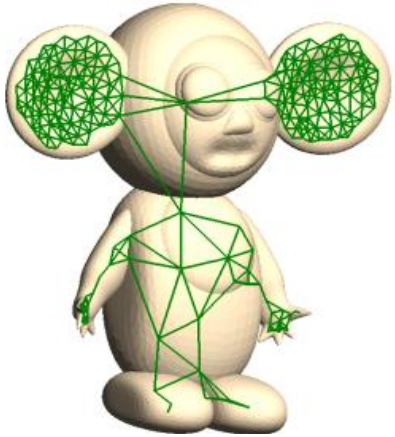
- A **discretization** of the existing volume takes place, so not the complete space is considered for the skeleton placement.
- To this end, the medial surface of the mesh is found. The **medial surface** is like the medial axis, but for a 3D space.
- Create the graph using a method similar to sphere packing.



Approximate Medial Surface



Packed Spheres

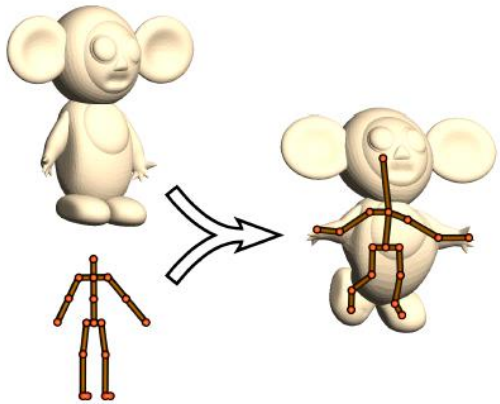


Constructed Graph

Skeleton embedding

The final graph is then constructed, sphere centers become vertices and edges are placed between intersecting spheres. To find a desirable **fitting**, undesired embeddings should be *penalized*. Possible aspects of such undesirable embeddings are:

- Short bones
- Improper orientation between joints
- Length difference in bones marked symmetric
- Bone chains sharing vertices
- Bones marked feet away from the bottom
- Zero-length bone chains
- Improper orientations of bones
- Joints close together in the graph, but far away in skeleton.



Skeleton embedding

To find the optimal weight for each penalty function, a maximum margin linear classifier is learned, that tries to maximize the margin between the best bad and best good embedding (as labeled in the training set):

$$\min_{i=1}^{n} \Gamma^T \mathbf{q}_i - \min_{i=1}^{m} \Gamma^T \mathbf{p}_i$$

Where **p** and **q** are feature vectors (containing the penalties mentioned earlier) of good and bad embeddings respectively, **Γ** is the vector containing weights and **n** and **m** are the number of good and bad embeddings. *Ideally we would like the choice of Γ so that equation 1 is maximized, as this creates a clear division between correct and incorrect embeddings*.

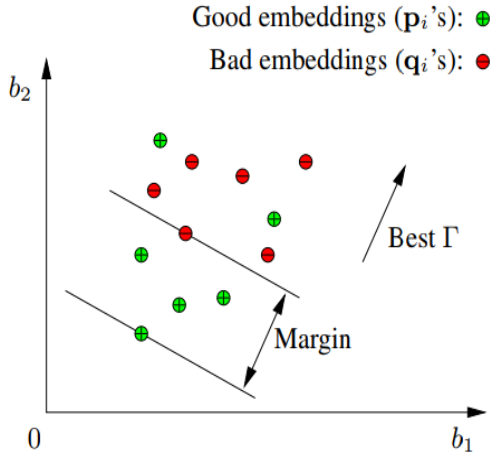
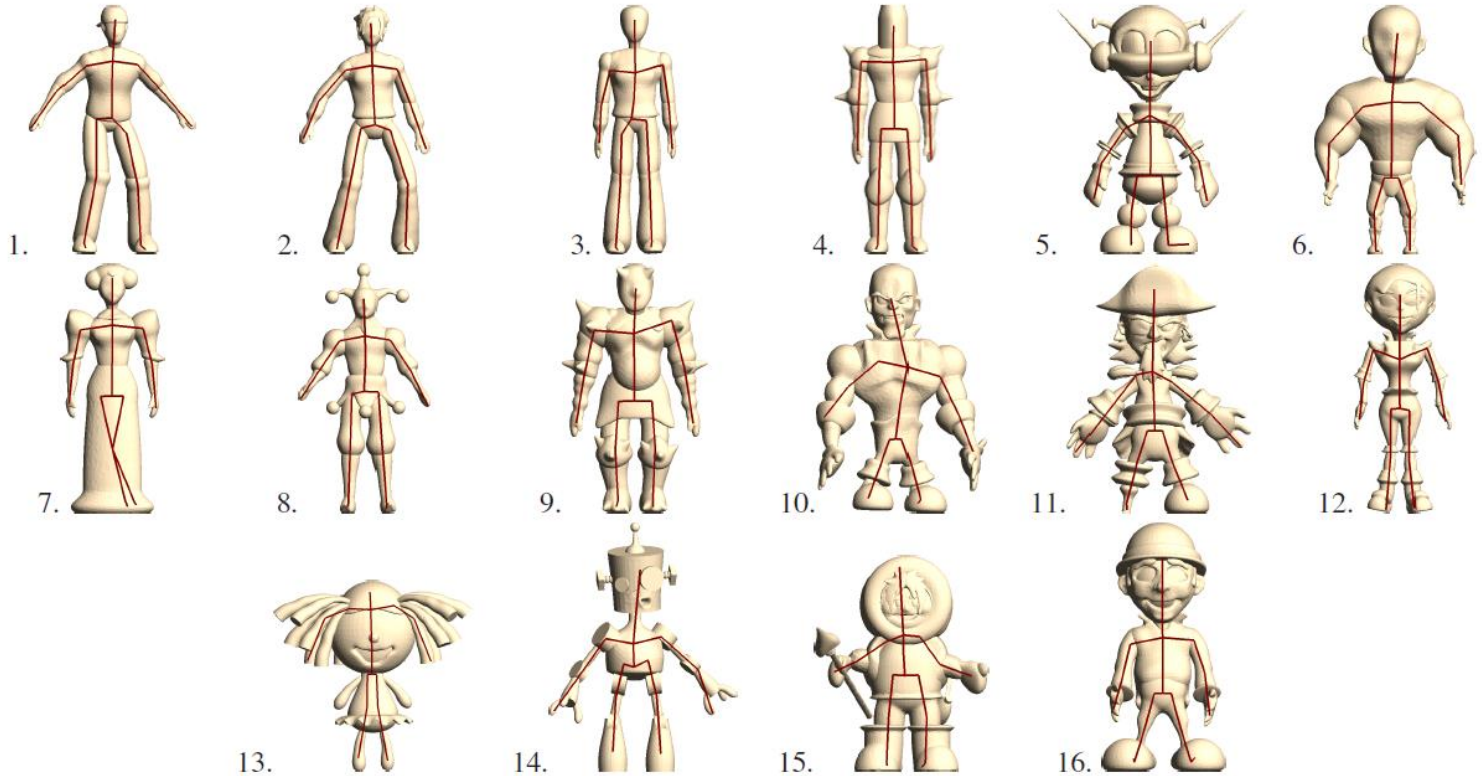
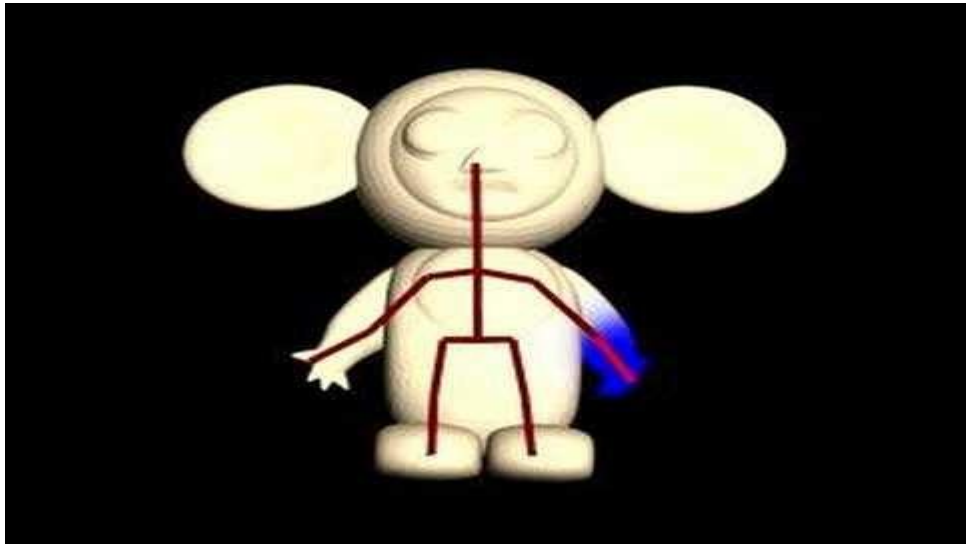


Figure 6: Illustration of optimization margin: marked skeleton embeddings in the space of their penalties (b_i's)



Skeleton embedding examples using Pinocchio
http://people.csail.mit.edu/ibaran/papers/2007-SIGGRAPH-Pinocchio.pdf



The Pinocchio systems automatically places skeletal joints inside 3D characters. Published in SIGGRAPH 2007.