

Smooth and Rough Surfaces

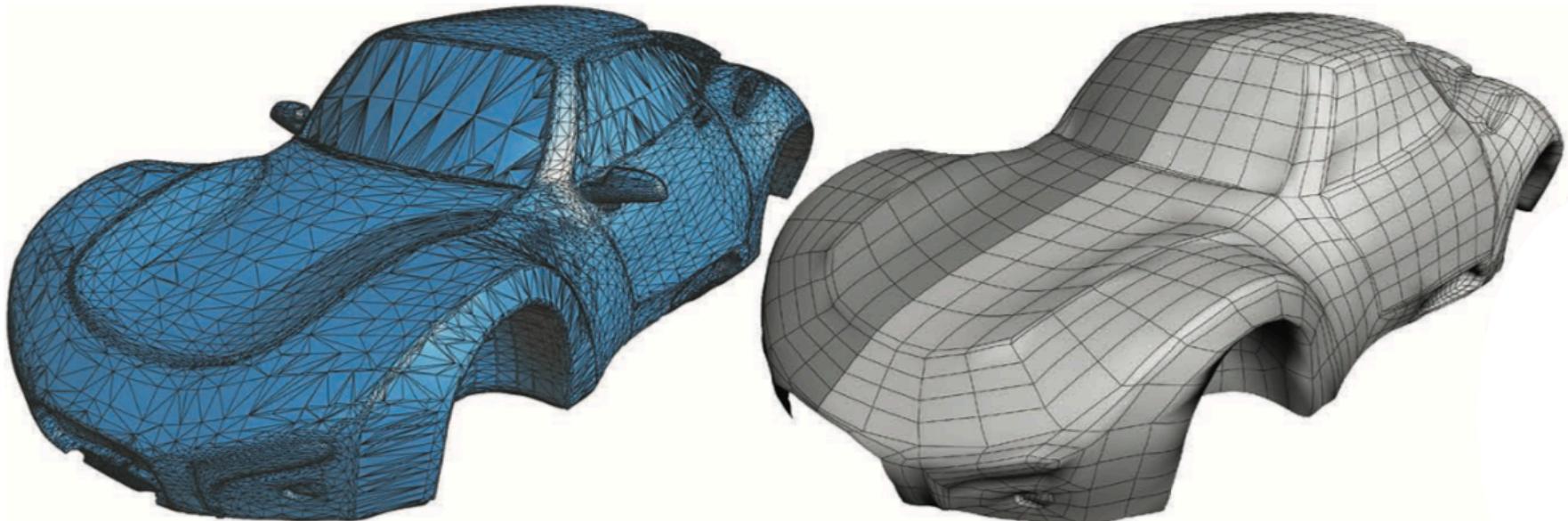
Prof. Fabio Pellacini

Triangle mesh recap

- Triangles have well defined geometric properties
 - they always live in a plane
 - so they have a unique geometric normal
- Triangle meshes are widely used
 - supported in rasterization and raytracing
 - sometimes used in simulation for cloth
- But it is hard to represent “regular” structures with triangles

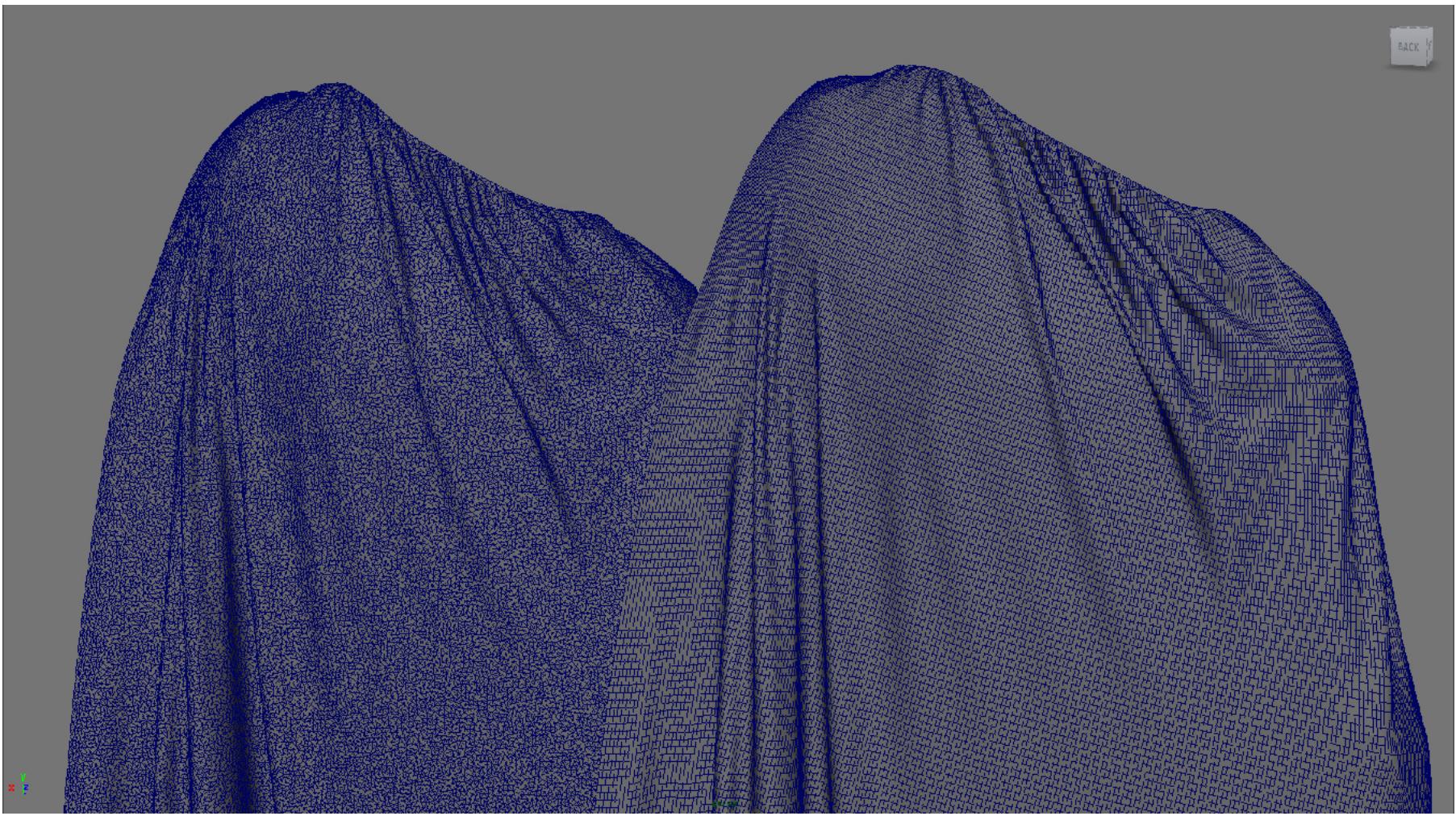
Quad meshes

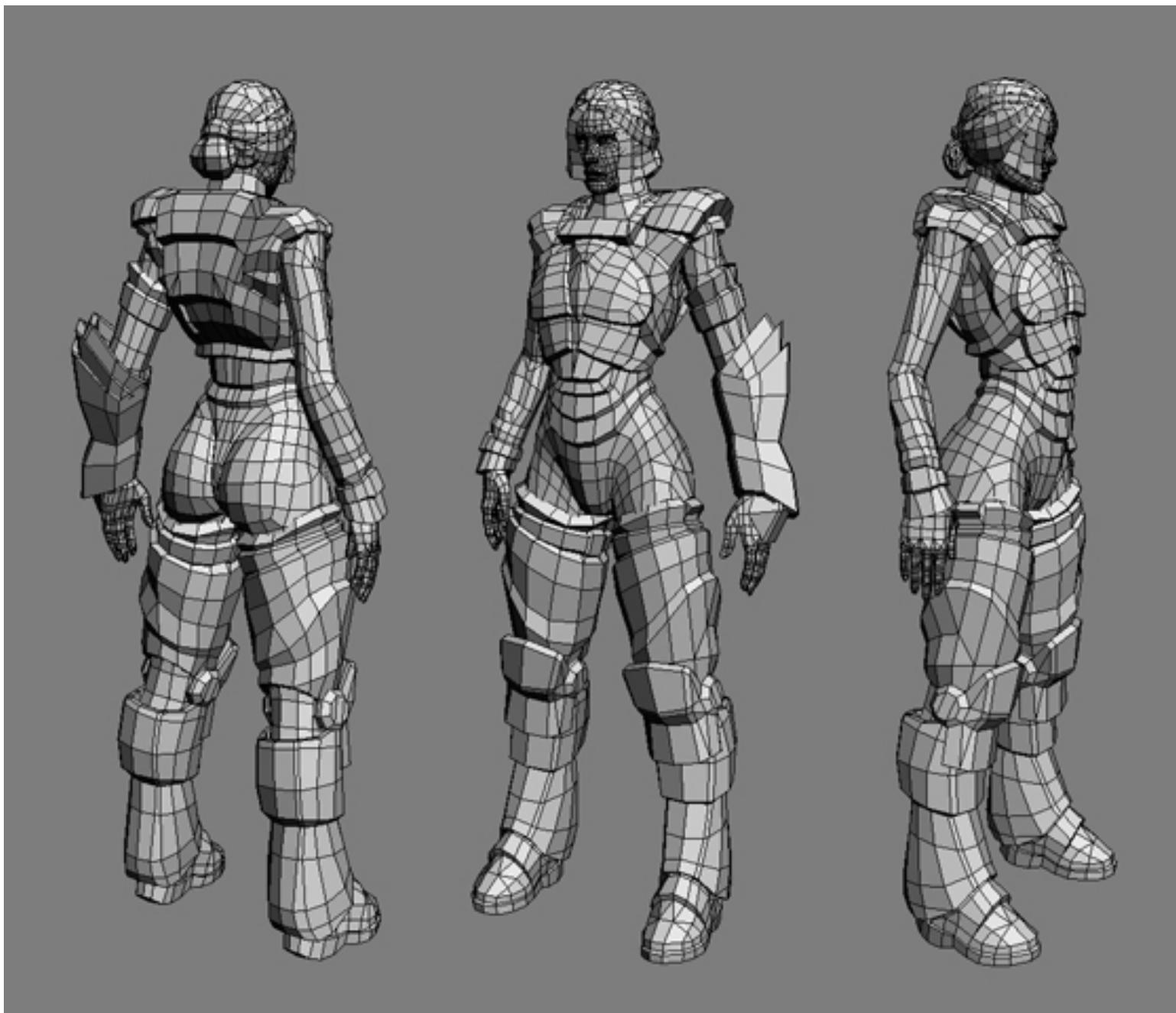
- Quad mesh: a collection of quadrilaterals
 - 4 vertices per face, possibly non-planar
 - use same data structure that triangles use



[source unknown]



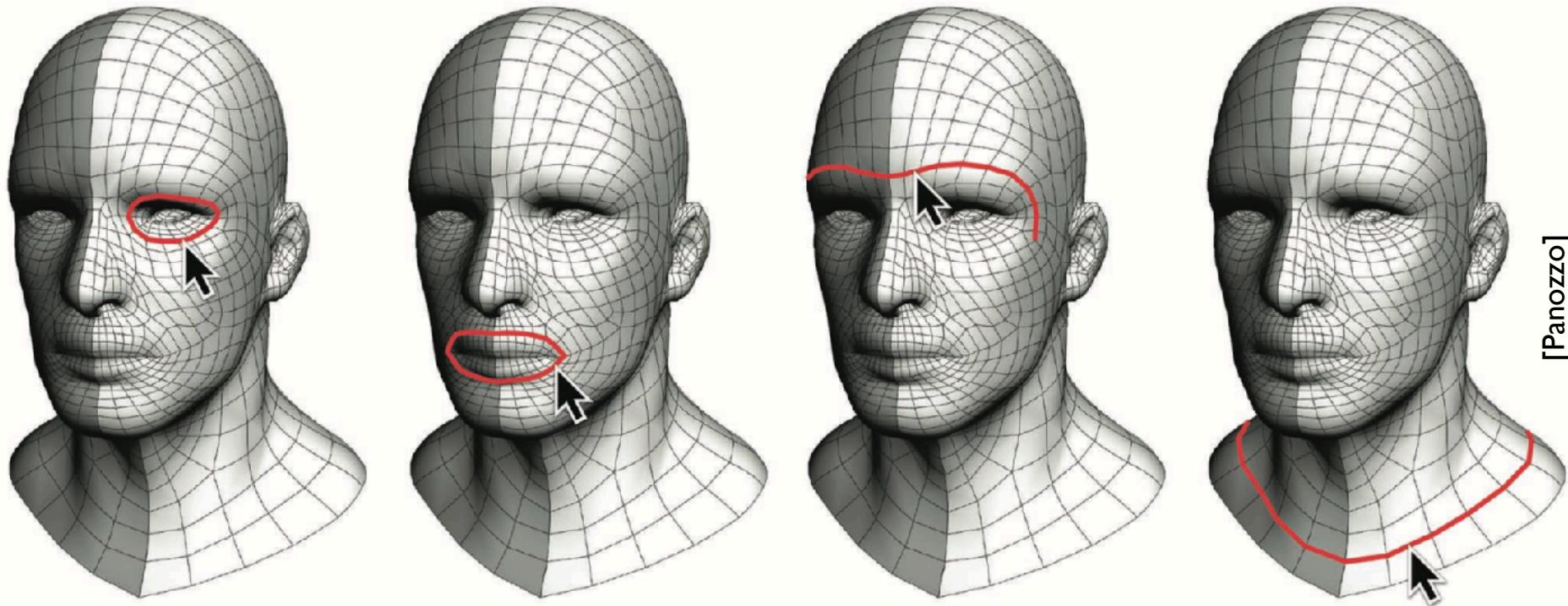


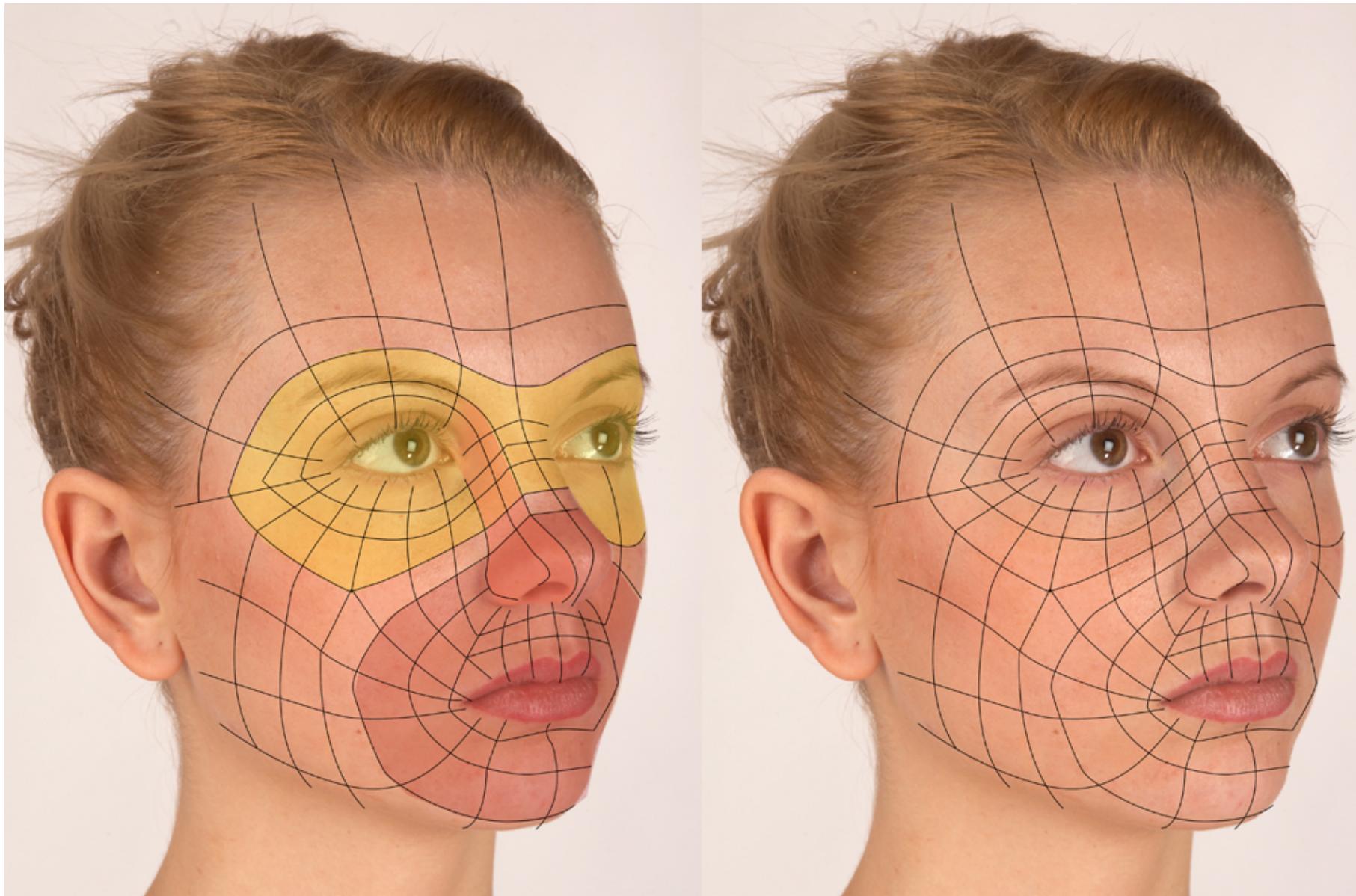


[source unknown]

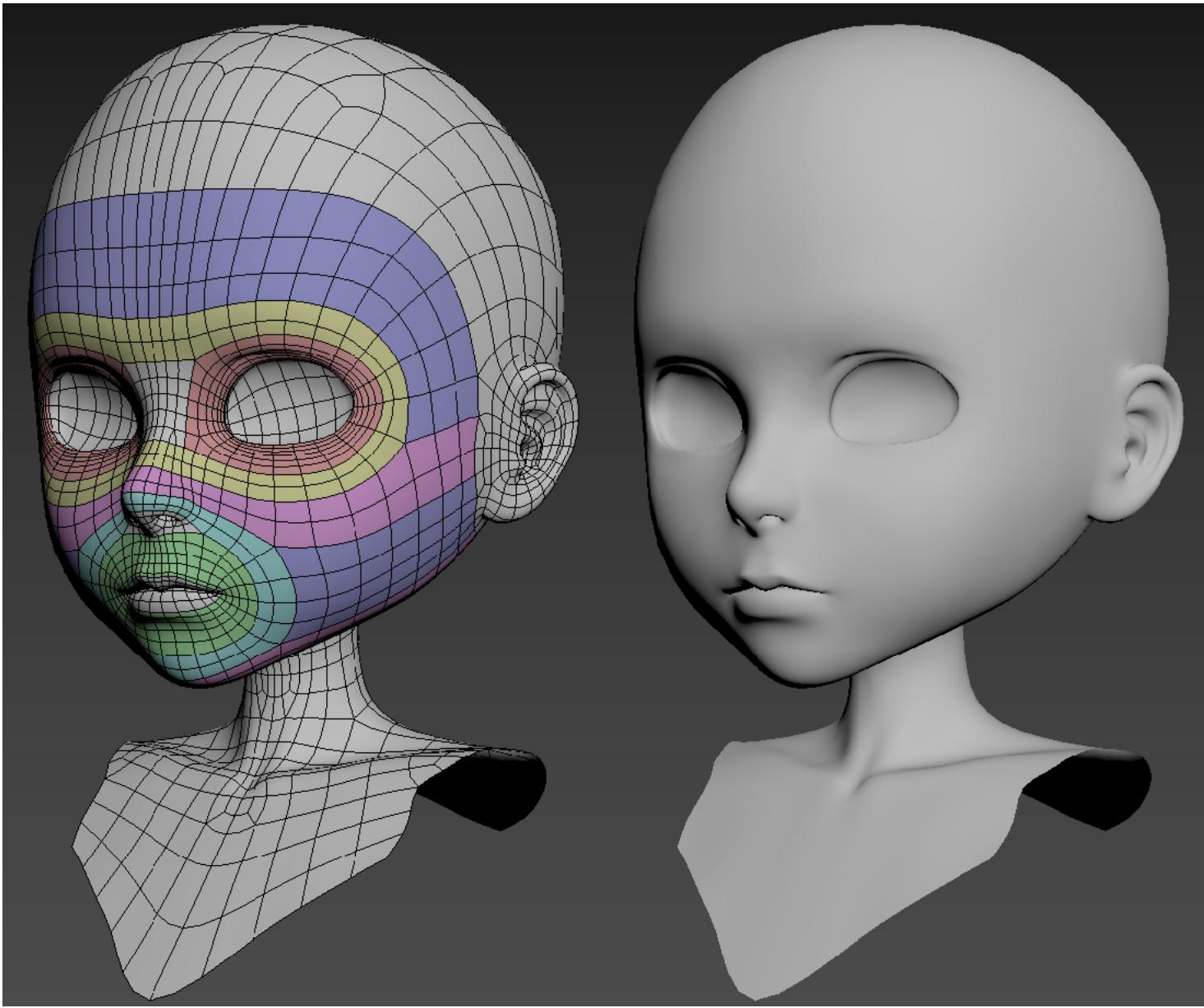
Quad meshes

- Most used modeling primitive
 - better selection and edit of edge loops
 - since it captures better “regularities” in surfaces

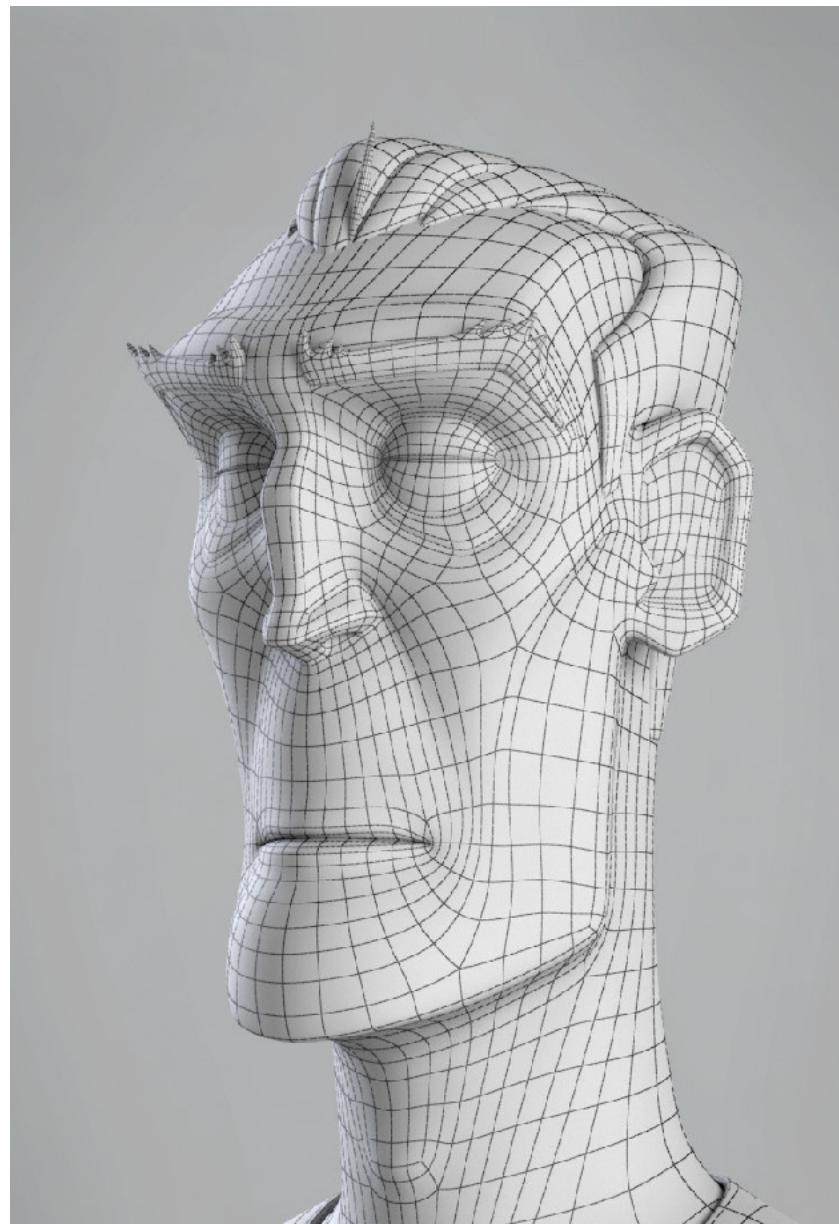
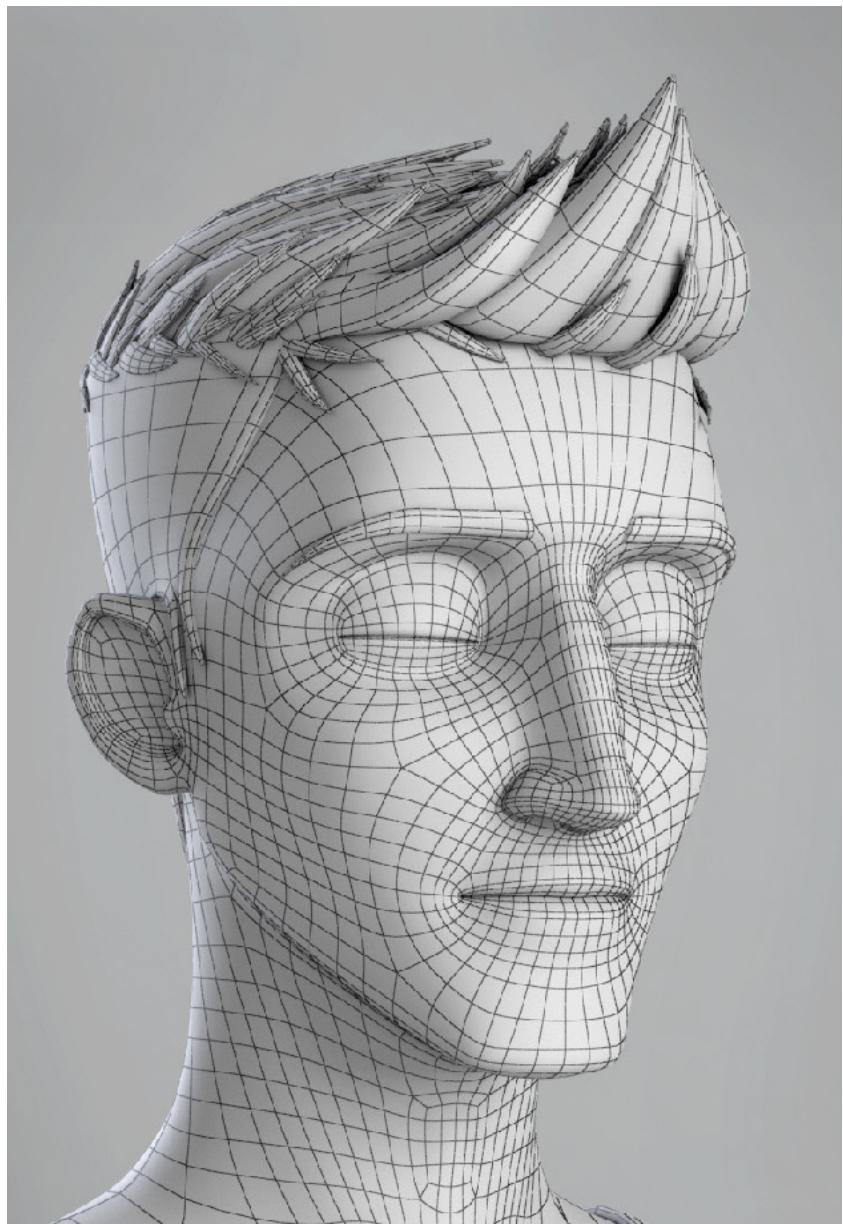




[Athey@deviantart]



[source unknown]



[Ander Liza]

Parametric Surfaces

Parametric surfaces

- Polygonal meshes are piecewise-flat approximations of real surfaces
- For many applications, we want to use alternative representations that are inherently smooth
- Parametric surfaces are one such representation obtained by describing a surface as a functions of 2 parameters

$$\mathbf{p}(u, v) = [x(u, v), y(u, v), z(u, v)]$$

- For example a sphere can be written parametrically as

$$\mathbf{p}(u, v) = [\cos(2\pi u) \sin(\pi v), \sin(2\pi u) \sin(\pi v), \sin(\pi v)]$$

Parametric normals

- Many common surface quantities can be derived from the surface representation directly
- For example the normal at a point (u,v) is the direction orthogonal to the tangent plane, that is the plane that passed through the point with tangents as partial derivatives

$$\mathbf{n}(u, v) = \frac{\mathbf{t}_u(u, v) \times \mathbf{t}_v(u, v)}{|\mathbf{t}_u(u, v) \times \mathbf{t}_v(u, v)|}$$

$$\mathbf{t}_u(u, v) = \frac{\partial \mathbf{p}(u, v)}{\partial u}$$

$$\mathbf{t}_v(u, v) = \frac{\partial \mathbf{p}(u, v)}{\partial v}$$

Surfaces patches

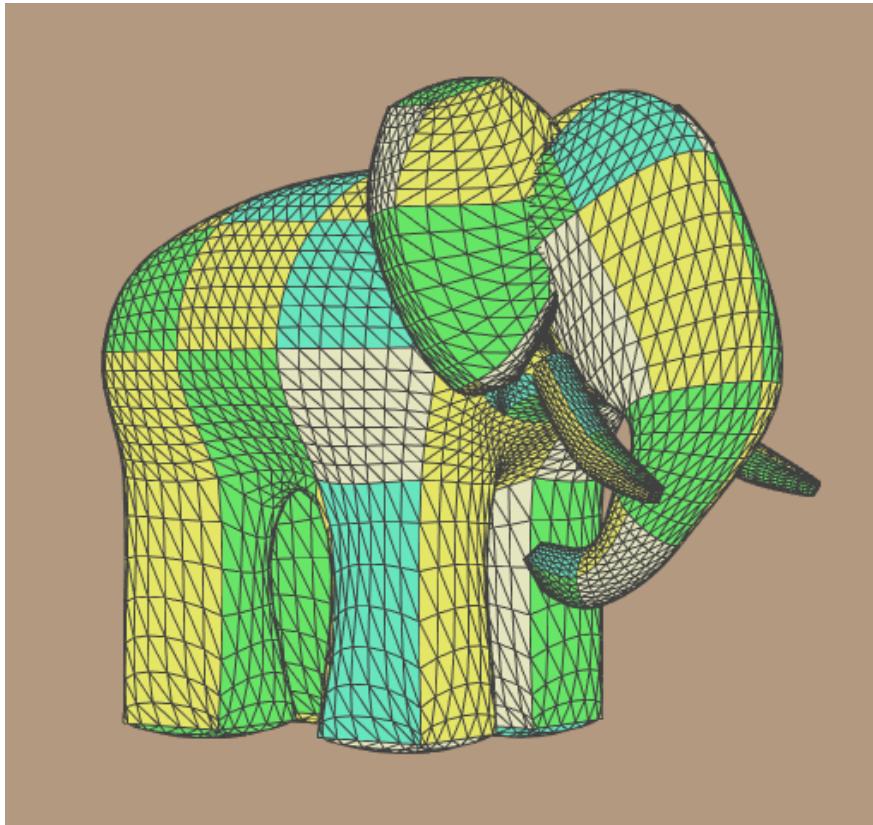
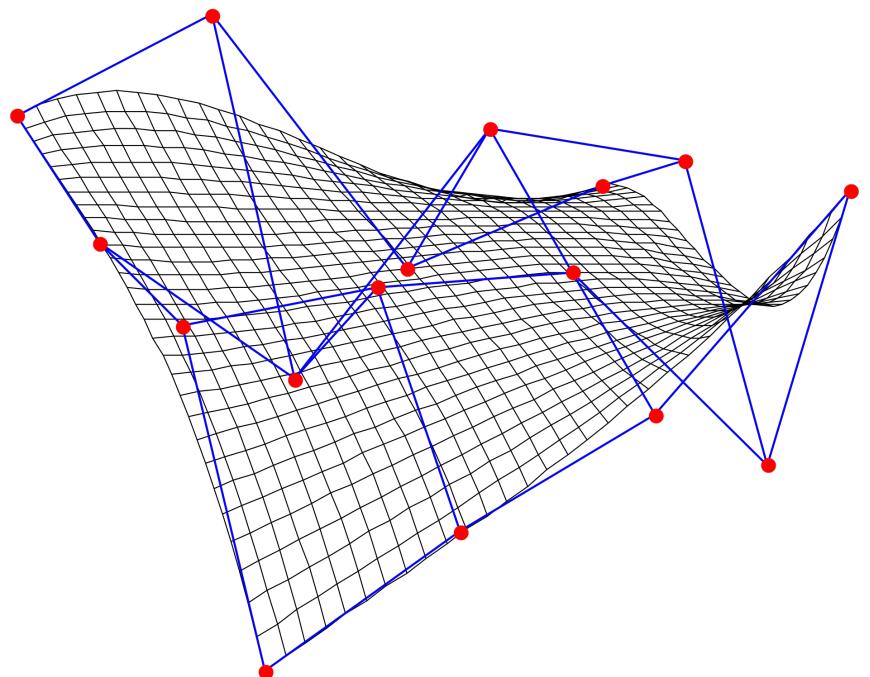
- As for parametric curves, many different parametric surfaces exists
- The most commonly used can be considered extensions of parametric curves called tensor product surfaces
- For example, for a Bezier curve written as

$$\mathbf{p}(u) = \sum_i b_i(u) \mathbf{p}_i$$

- we can derive a Bezier patch written as

$$\mathbf{p}(u, v) = \sum_i \sum_j b_i(u) b_j(v) \mathbf{p}_{i,j}$$

Surface patches



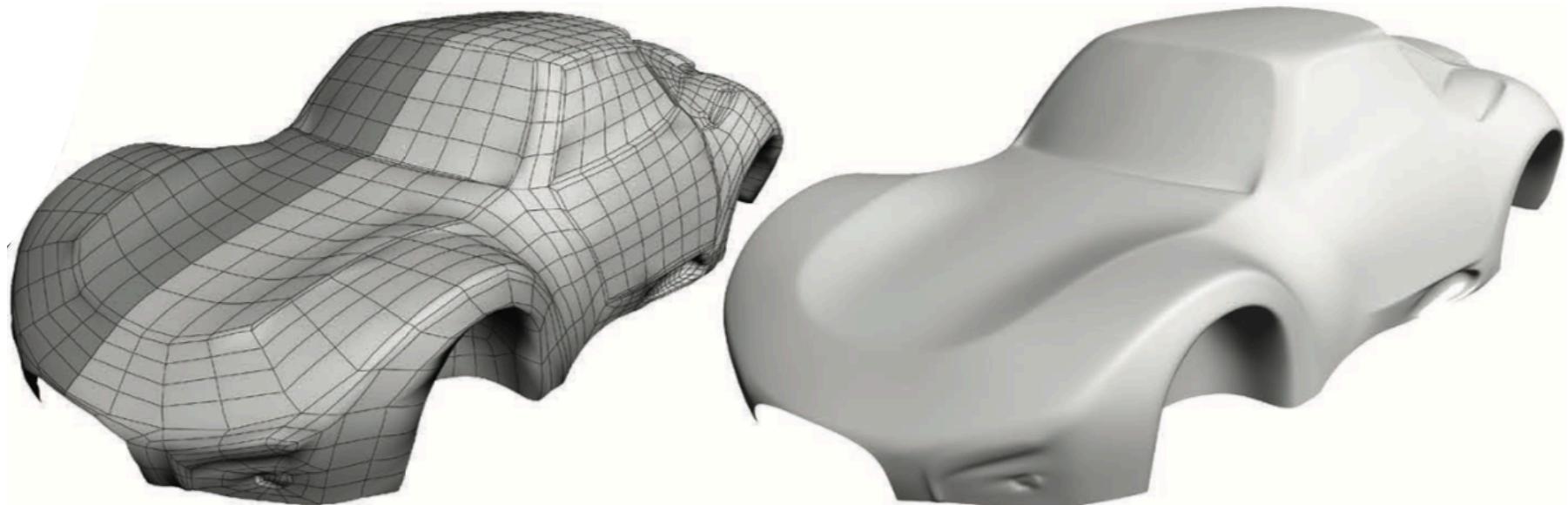
NURBS

- CAD software often uses non-uniform rational B-splines, aka NURBS
- NURBS have the main advantage of being able to represent exactly conic sections, including spheres, cylinders, etc
- Other splines can only approximate these surfaces
- NURBS are rarely used outside of CAD software mostly due to the inherent complexity of their representation

Subdivision Surfaces

Subdivision surfaces

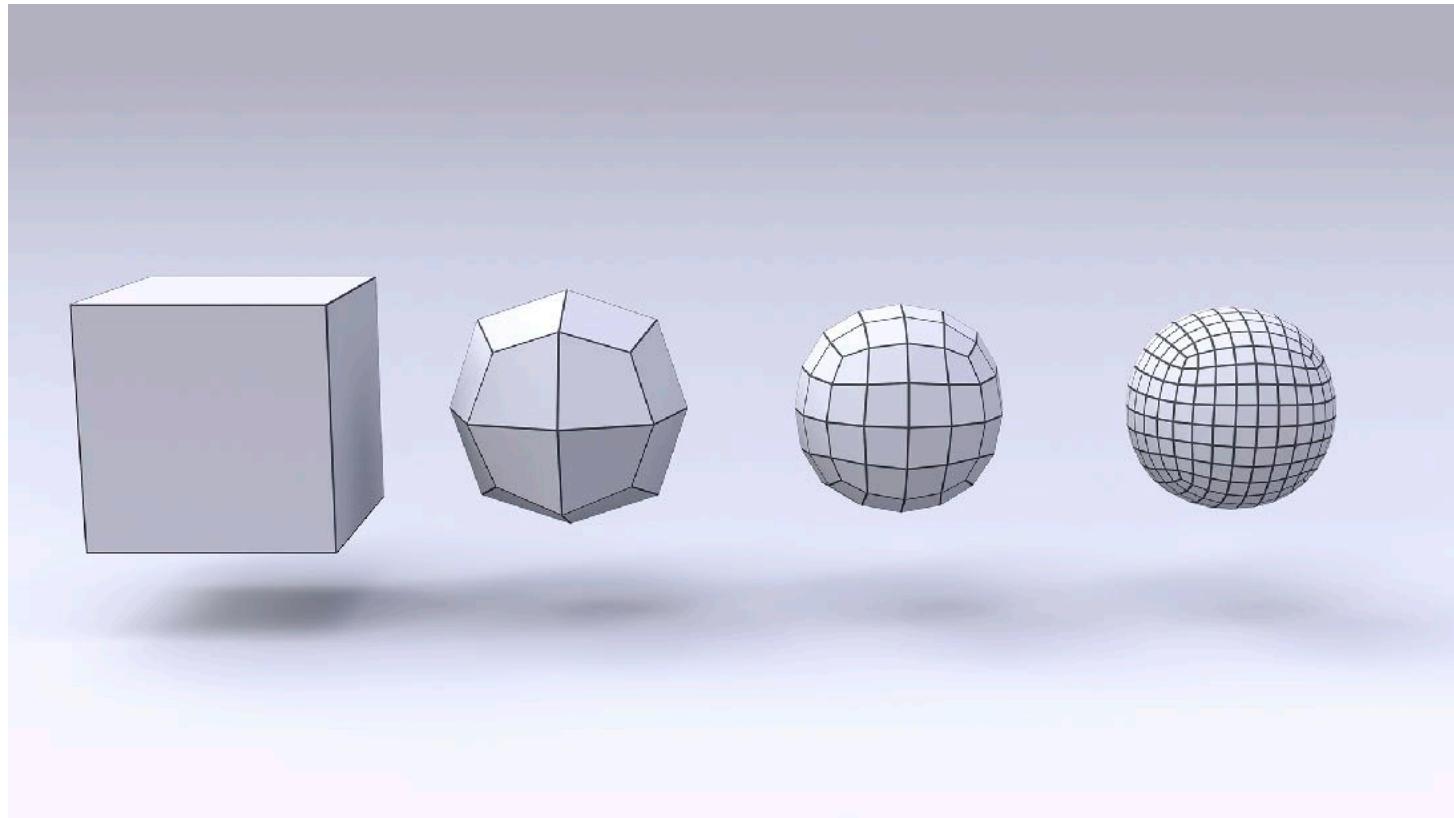
- Represent a smooth surface controlled by a polygon mesh
- Most used method for movies and soon games – works for organic and man-made models



[PanzoZZo]

Subdivision surfaces

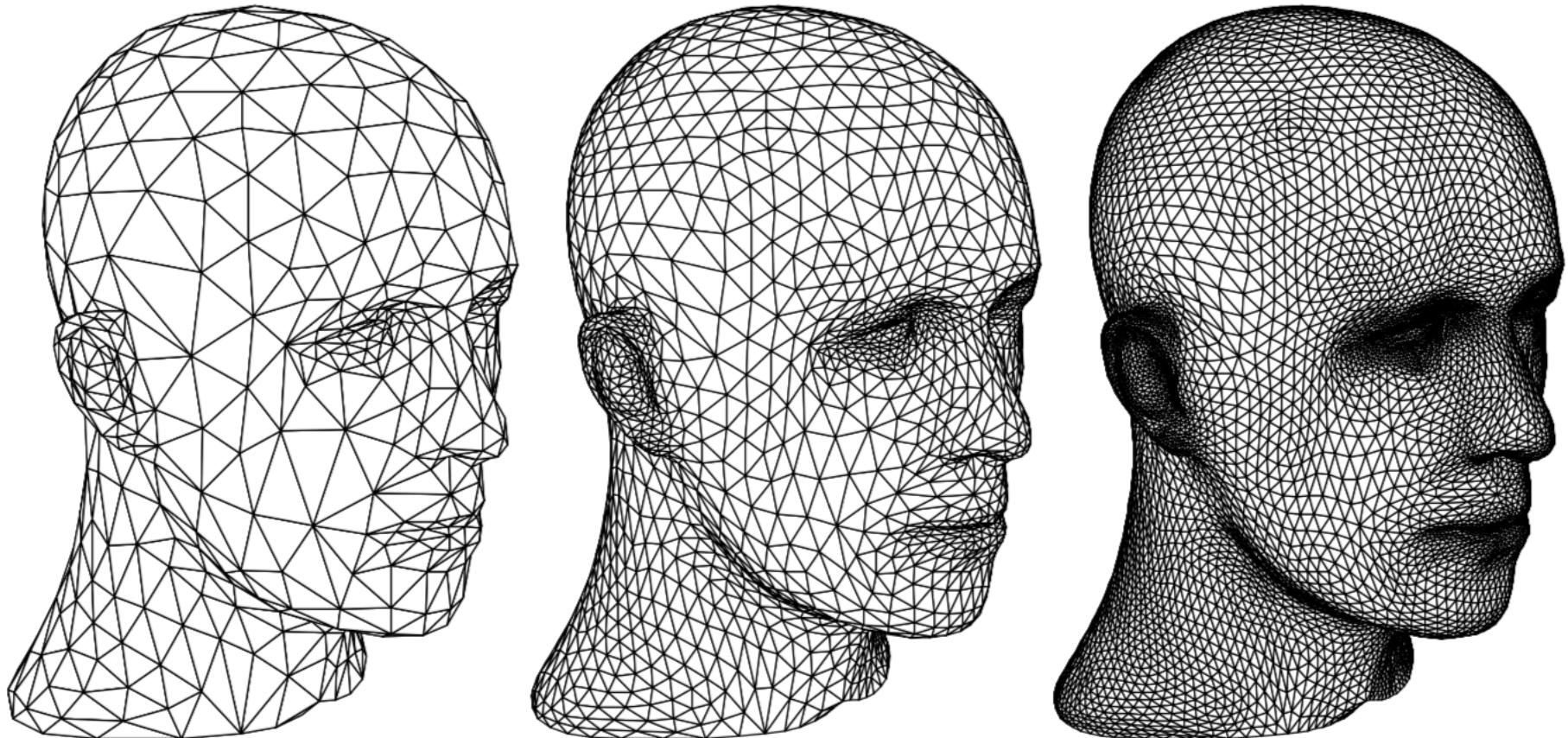
- Defined by recursively apply a refinement rule that add vertices and faces to the mesh and smooths out the vertex positions



[Yining Karl]

Subdivision surfaces

- Defined by recursively apply a refinement rule that add vertices and faces to the mesh and smooths out the vertex positions



Subdivision surfaces – informal

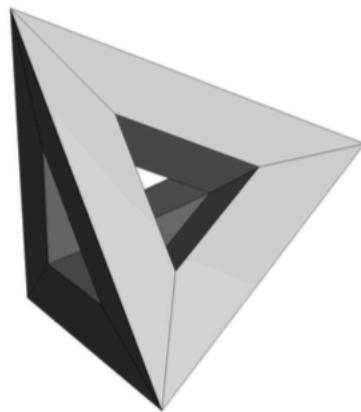


[CGGorilla/YouTube]

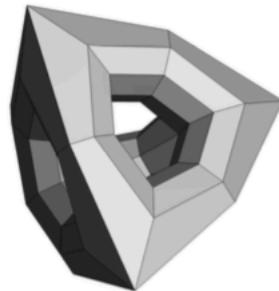
Subdivision surfaces

- The subdivision surface is the limit surface S obtained by applying the subdivision rule R an infinite number of times to a mesh M_0

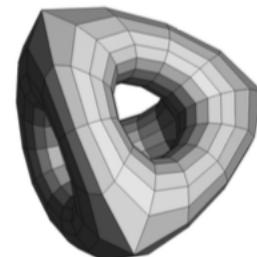
$$M_{i+1} = R(M_i) \quad S = \lim_{n \rightarrow \infty} M_n = \lim_{n \rightarrow \infty} R^n(M_0)$$



M_0



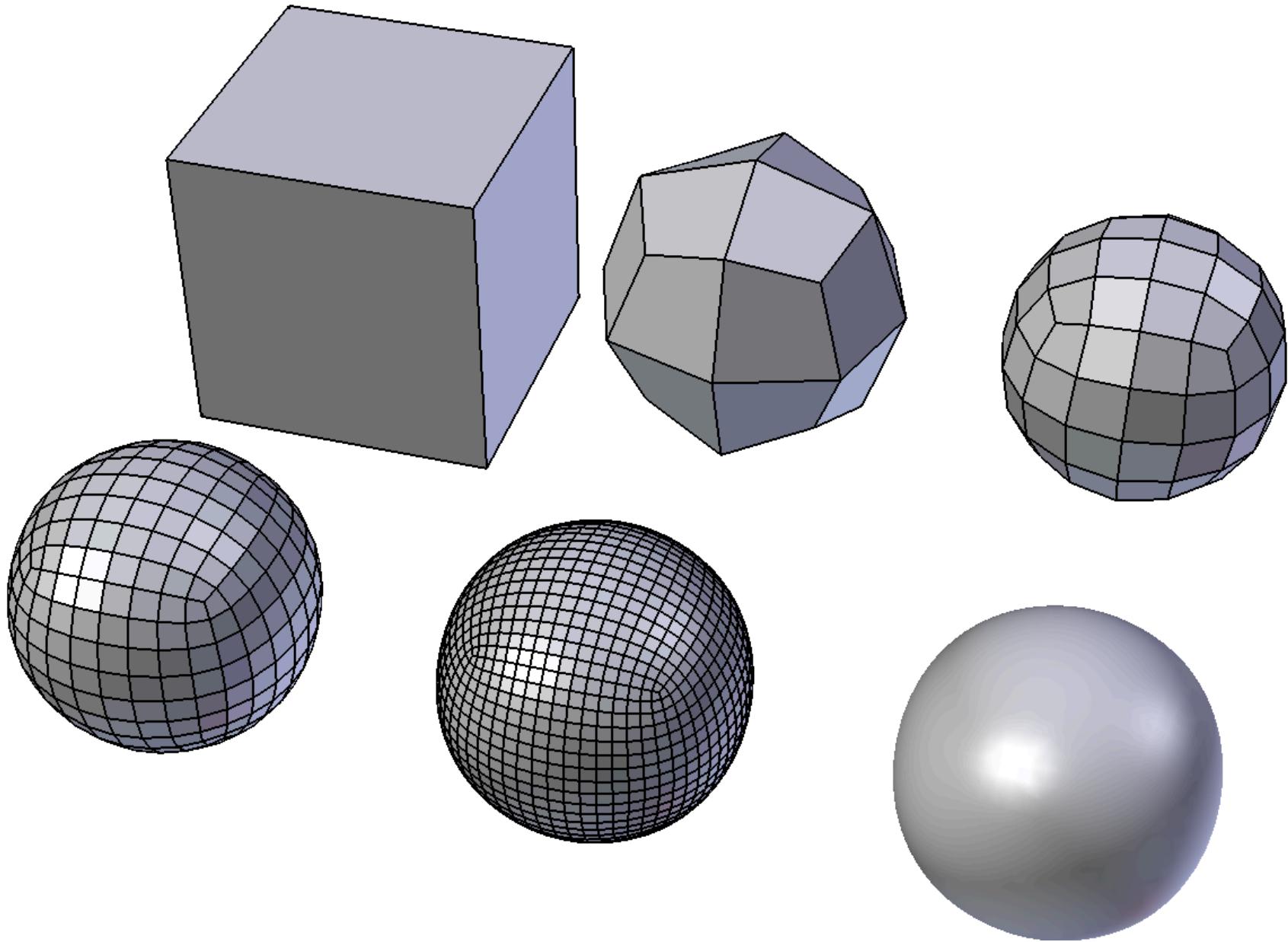
M_1



M_2



$S = M_\infty$

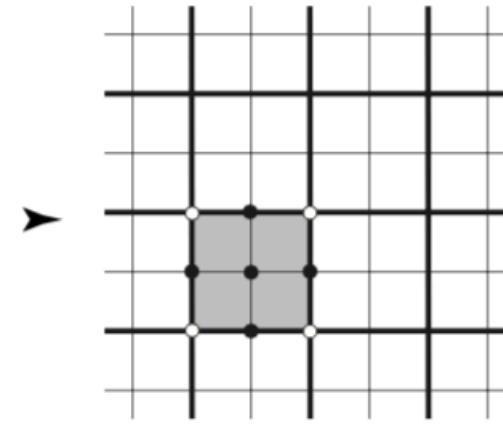
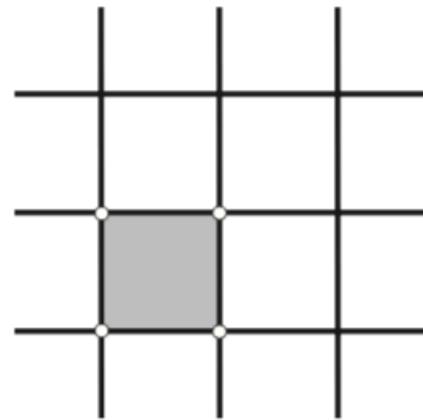
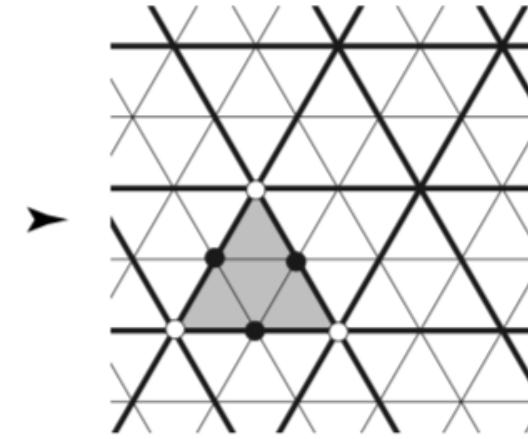
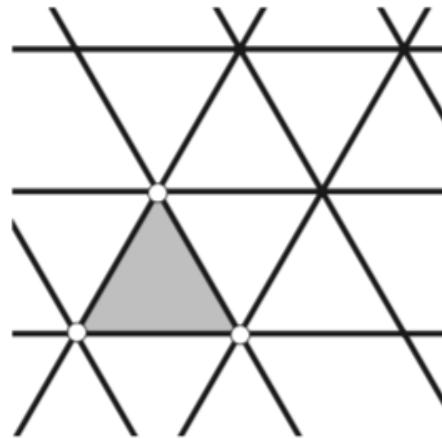


Subdivision rules

- Different rules support different input (triangles or quads) and generate surfaces with different properties
- All rules are composed of two parts
- Step 1: refine the mesh by adding vertices and faces
 - this changes the topology of the mesh
- Step 2: move refined vertices to achieve a smoother surface
 - this changes the geometry of the mesh
- We will only cover the two most common cases
 - Catmull–Clark for quad meshes
 - Loop for triangle meshes

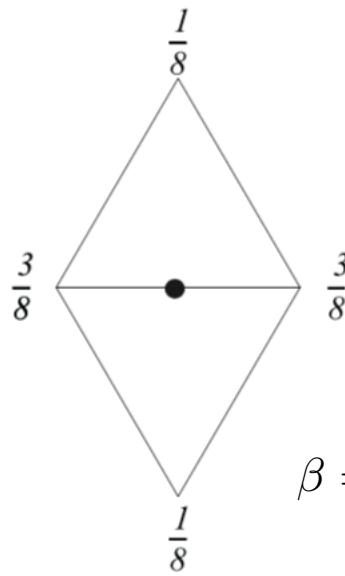
Subdivision refinement

- Edge split methods for triangle (Loop) and quads (Catmull-Clark)



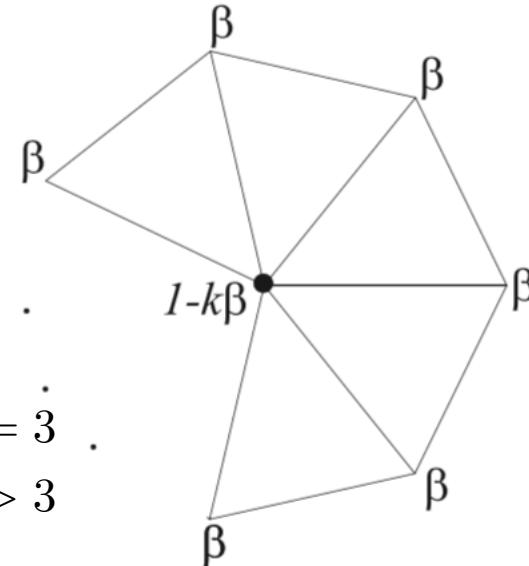
Subdivision smoothing

- Loop: rules for existing vertices and vertices on the split edges
 - differentiate between interior and boundaries

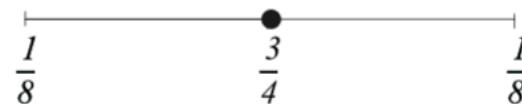


Interior

$$\beta = \begin{cases} 3/16 & \text{for } n = 3 \\ 3/(8n) & \text{for } n > 3 \end{cases}$$



Crease and boundary

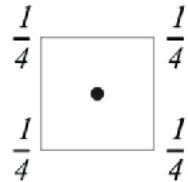


a. Masks for odd vertices

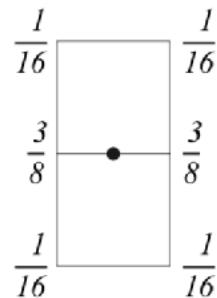
b. Masks for even vertices

Subdivision smoothing

- Catmull-Clark: rules existing and new vertices

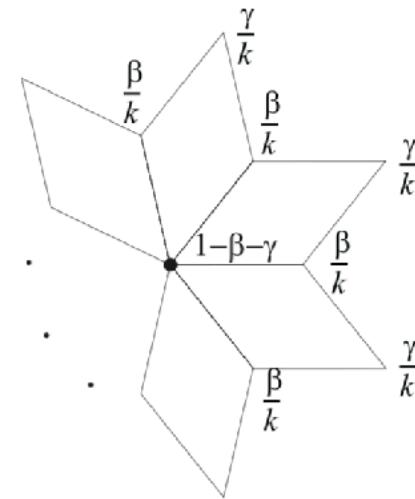


Mask for a face vertex



Mask for an edge vertex

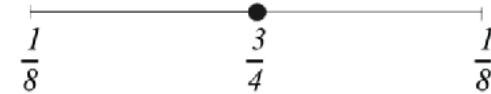
Interior



$$\beta = 3/(2k) \quad \gamma = 1/(4k)$$

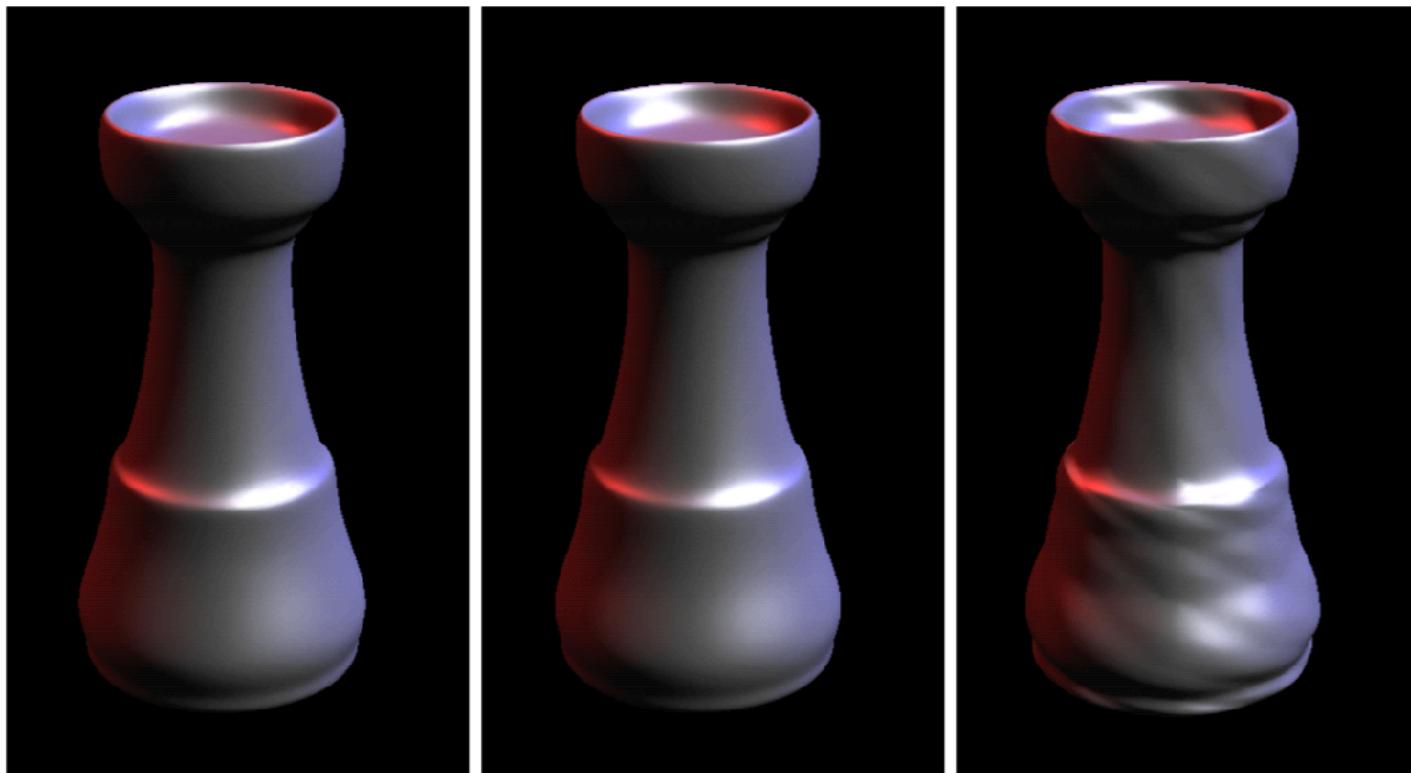


Crease and boundary



Mask for a boundary odd vertex

Model comparison



Initial mesh

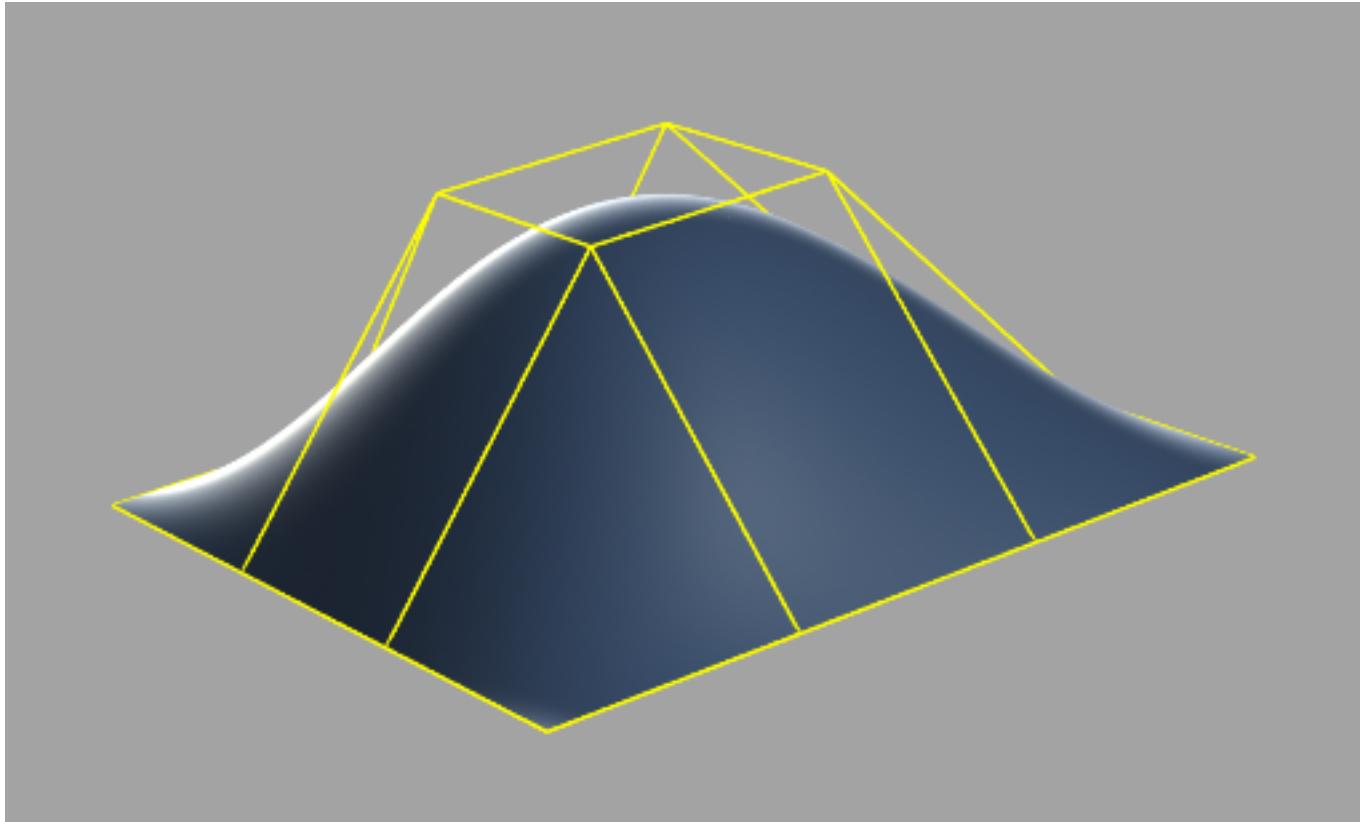
Loop

Catmull-Clark

*Catmull-Clark, after
triangulation*

Creases

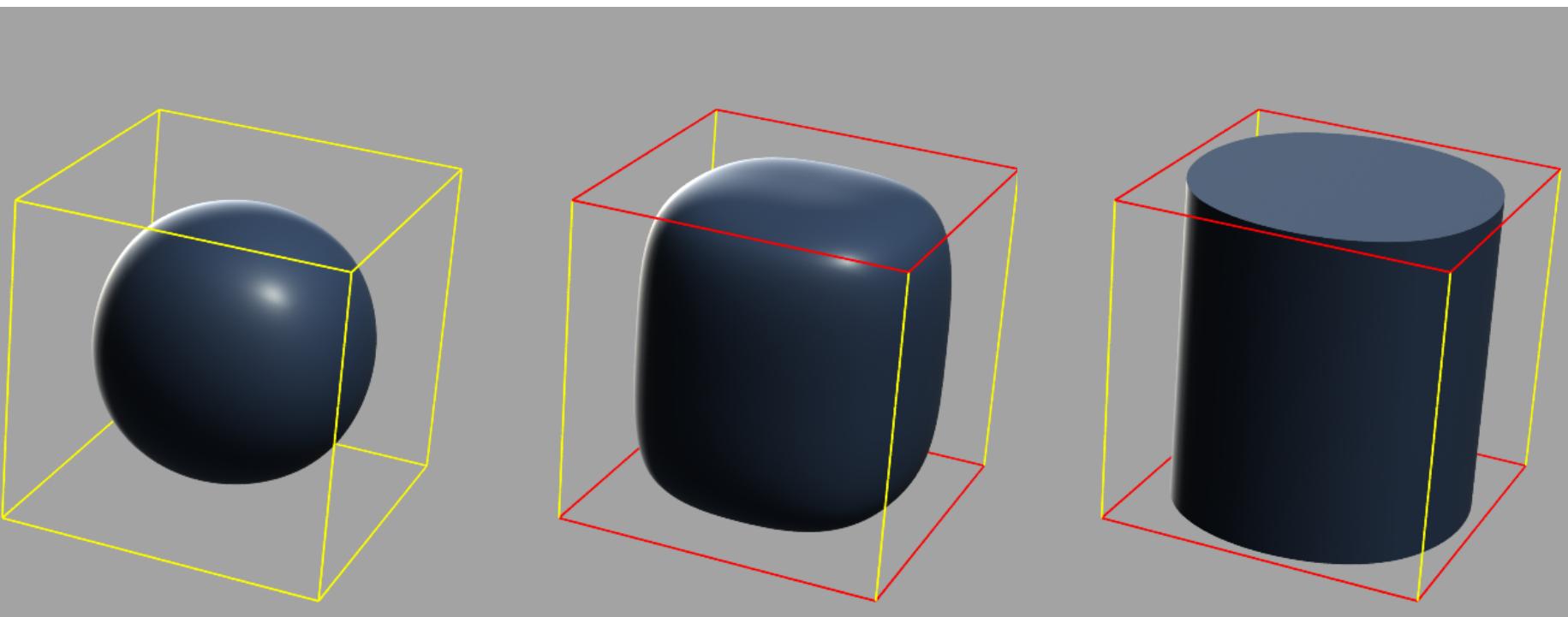
- Control sharpness of edges and vertices



[de Rose et al.]

Creases

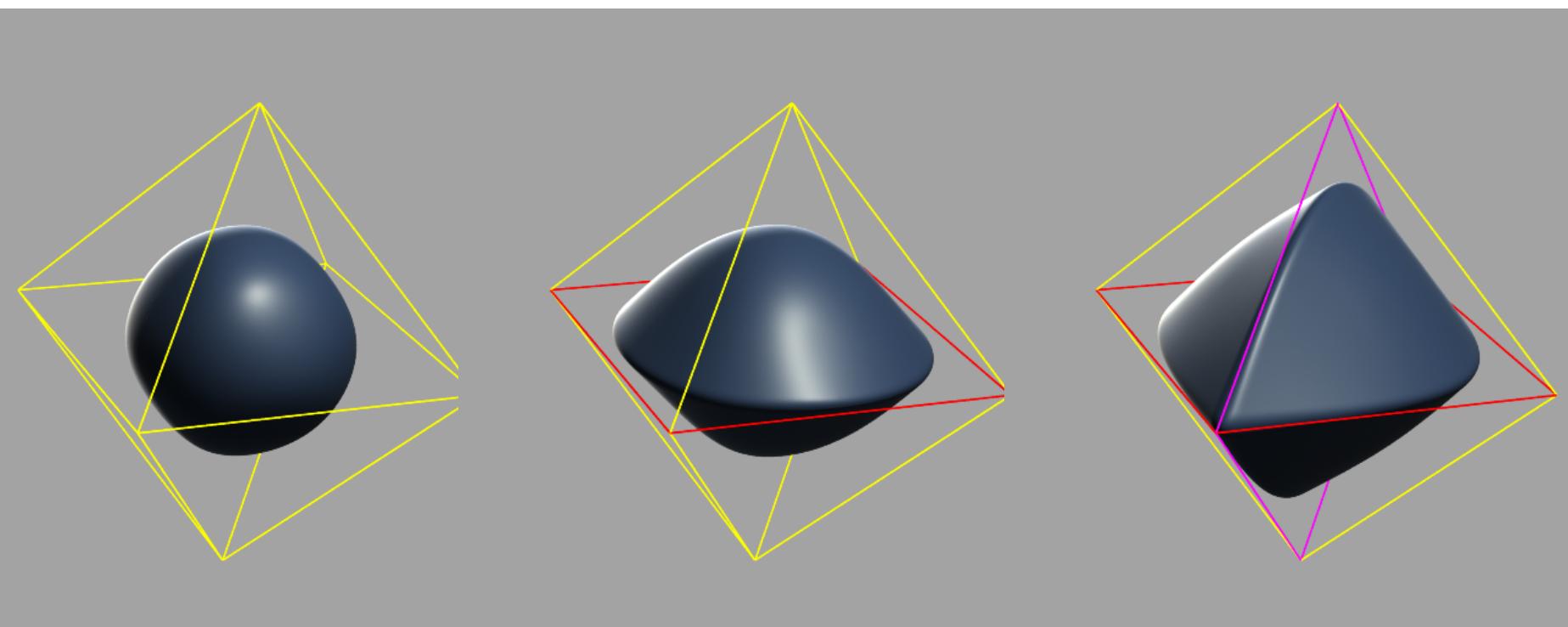
- Control sharpness of edges and vertices



[de Rose et al.]

Creases

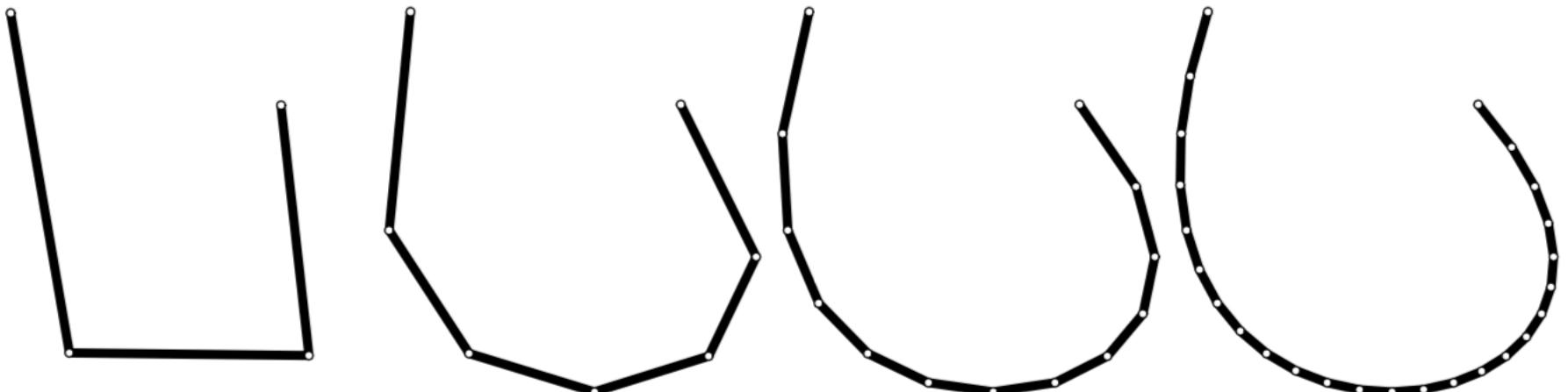
- Control sharpness of edges and vertices



[de Rose et al.]

Subdivision curves

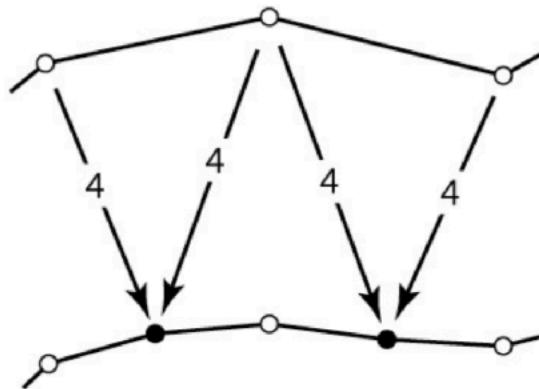
- Main idea also works for curves
 - but less in practice since polynomials are better
- Rule: recursively split each segment in half and move vertex
- Define curves from subdivision rules, instead of polynomials
 - for B-splines we have both definitions and they are equivalent



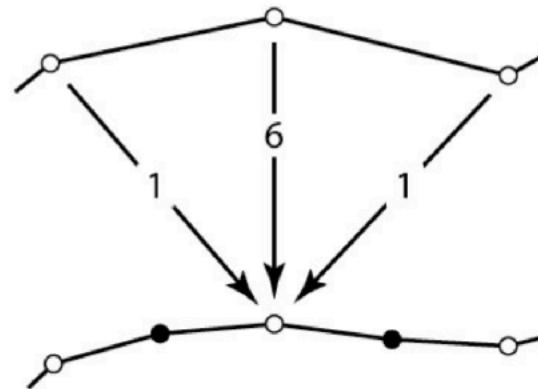
[Schröder & Zorin]

Subdivision rule for B-splines

- New vertices are linear combination of old vertices



ODD



EVEN

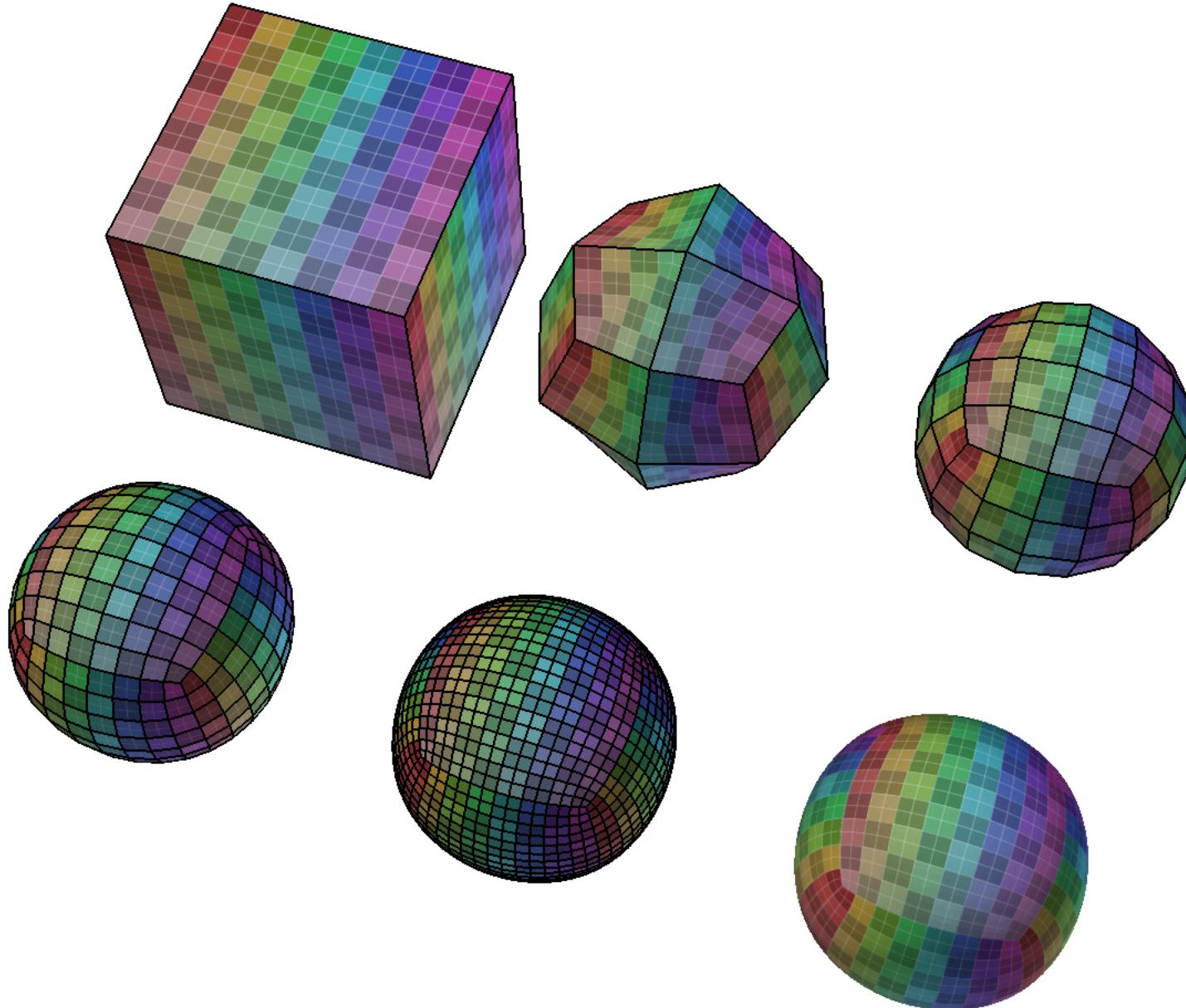


Subdivision scalability

- Subdivision surfaces have many advantages when drawing shapes
- First, they can be subdivided only to the necessary level
 - further away objects are subdivided less than closer ones
 - this gives us “level of details” when rendering
- Second, they can be subdivided partially, only adding details in the regions of high curvature
 - at the expense of complex subdivision algorithms
- Third, they can be supported natively in a ray tracer with specialized intersection code that is optimized compared to simply tessellate to a fixed number of triangles

Face-varying data

- So far, all data we associated with a triangle meshes was defined at the vertices
- There are cases though when associating data to faces is required, for example when representing texture coordinates
- In this case, we use the same weights we would for position, but with a different topology that has duplicated vertices when needed
- This elegant formulation allowed subdues to be used in the entertainment history



Subdivision artifacts – informal



[CGGorilla/YouTube]

Implementing Subdivs

Implementing subdivs

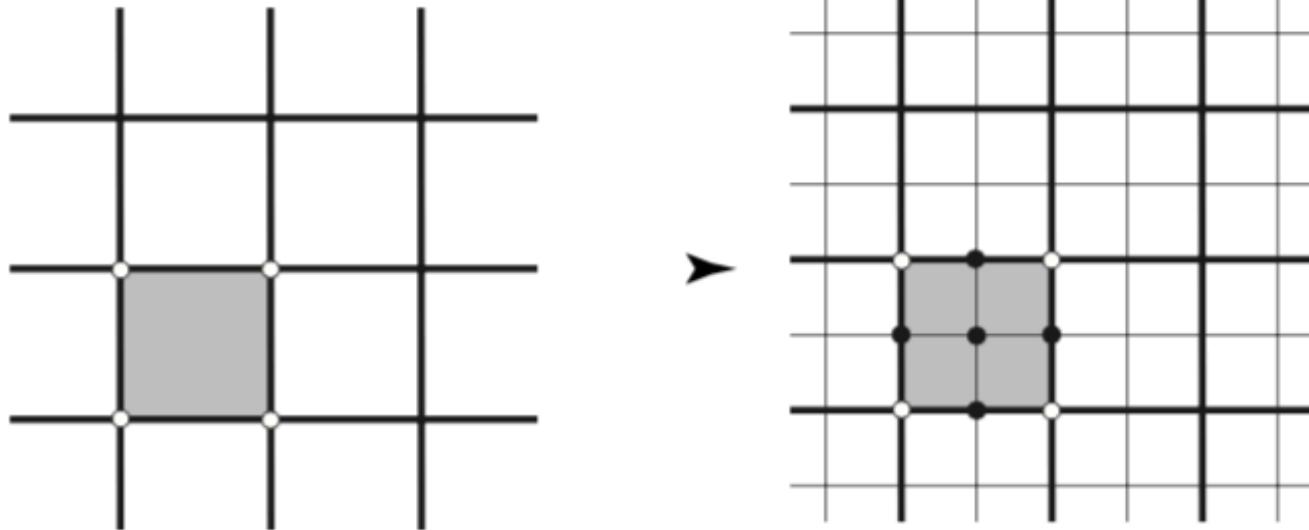
- Classic implementation requires specialized data structures
 - seldomly implemented
 - often results do not match between implementations
 - considerable practical problems for exchanging data
 - special handling for creases
- Classic implementation can be done optimal though
 - view driven level of details
 - non-binary subdivision
 - accelerated raytracing without tessellating to triangles
 - check out Pixar's OpenSubdiv and Intel's Embree

Implementing subdivs

- Simple multipass algorithm for Catmull-Clark, Loop and mixed
 - based on *A factored approach to Subdivision Surfaces* by Warren and Schaefer
 - allow for quick experimentation
- Main advantage: uses only a hash table and no specialized d.s.
- Support triangles, quads and mixed input including creases
 - with no special case handling
- Less efficient but remarkably simpler
- Will present only the Catmull-Clark variant

Implementing subdivs

- Step 1: linear subdivision
 - main concern: ensure that edge vertices are shared after split



[Schröder & Zorin]

Implementing subdivs

- Step I: linear subdivision
 - create one vertex for each vertex, edge and quad in original
 - vertex from vertex: set same position

$$\mathbf{p}'_v = \mathbf{p}_v$$

- vertex from edge: set in the middle between endpoints

$$\mathbf{p}'_e = \frac{\mathbf{p}_{e_1} + \mathbf{p}_{e_2}}{2}$$

- vertex from face: set to face centroid

$$\mathbf{p}'_f = \frac{\mathbf{p}_{f_1} + \mathbf{p}_{f_2} + \mathbf{p}_{f_3} + \mathbf{p}_{f_4}}{4}$$

Implementing subdivs

- Step I: linear subdivision
 - edge vertex: two faces share an edge, but cannot duplicate the edge vertex
 - use a hash table to check whether the edge was already split
 - hash keys: pair of unordered vertex indices (edge)
 - either insert both orders or always insert (min, max) index
 - hash values: index to the new edge

Implementing subdivs

- Step I: linear subdivision
 - create four faces F for each quad f in the original
 - use the hash table for the edge indices

$$F'_{\{f,0\}} = \{\mathbf{p}'_{f_0}, \mathbf{p}'_{e_{f_0 \rightarrow f_1}}, \mathbf{p}'_f, \mathbf{p}'_{e_{f_3 \rightarrow f_0}}\}$$

$$F'_{\{f,1\}} = \{\mathbf{p}'_{f_1}, \mathbf{p}'_{e_{f_1 \rightarrow f_2}}, \mathbf{p}'_f, \mathbf{p}'_{e_{f_0 \rightarrow f_1}}\}$$

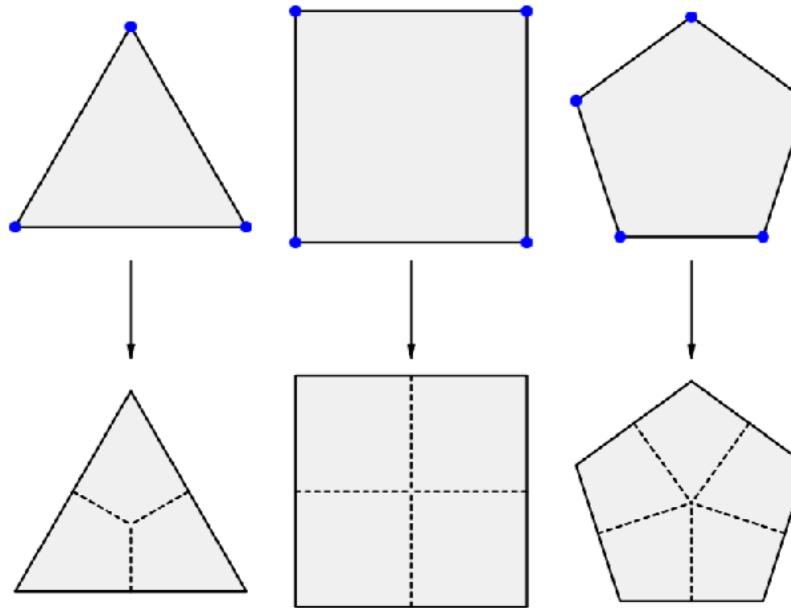
$$F'_{\{f,2\}} = \{\mathbf{p}'_{f_2}, \mathbf{p}'_{e_{f_2 \rightarrow f_3}}, \mathbf{p}'_f, \mathbf{p}'_{e_{f_1 \rightarrow f_2}}\}$$

$$F'_{\{f,3\}} = \{\mathbf{p}'_{f_3}, \mathbf{p}'_{e_{f_3 \rightarrow f_0}}, \mathbf{p}'_f, \mathbf{p}'_{e_{f_2 \rightarrow f_3}}\}$$

- Works also for non-quad faces

Implementing subdivs

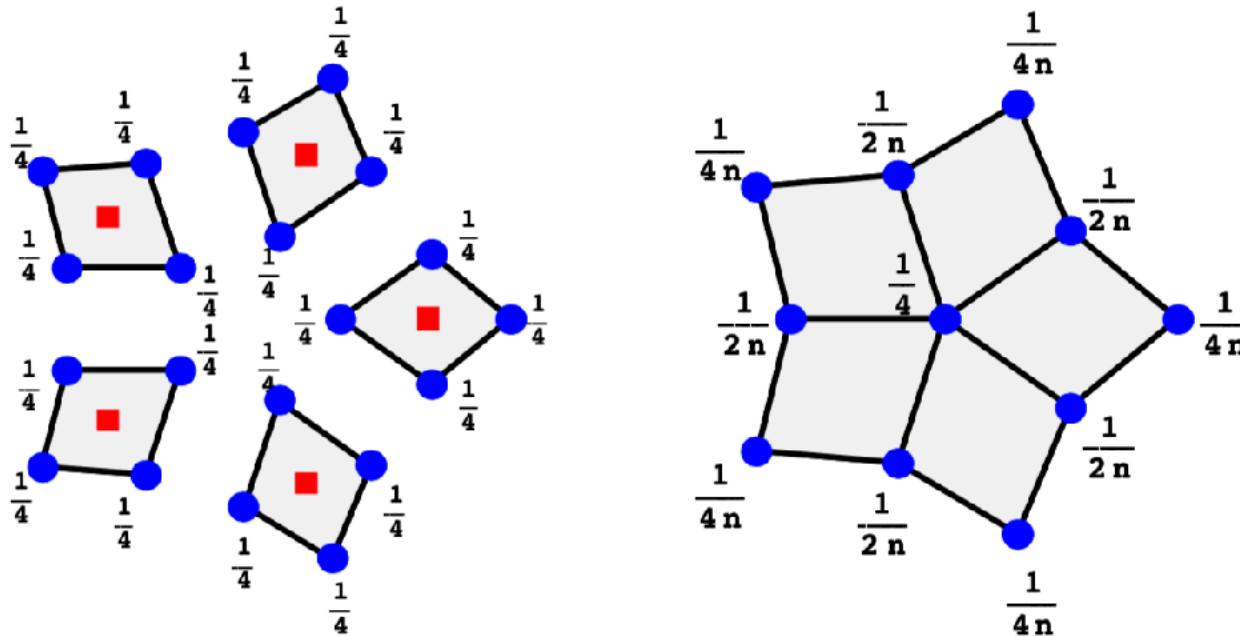
- Step 1: linear subdivision
 - works also for non-quad faces as input
 - after one pass all faces are quads



[Warren & Shaefer]

Implementing subdivs

- Step 2: averaging pass
 - smooth vertices based on their neighbors
 - update vertex position as the average of the centroids of the faces touching it



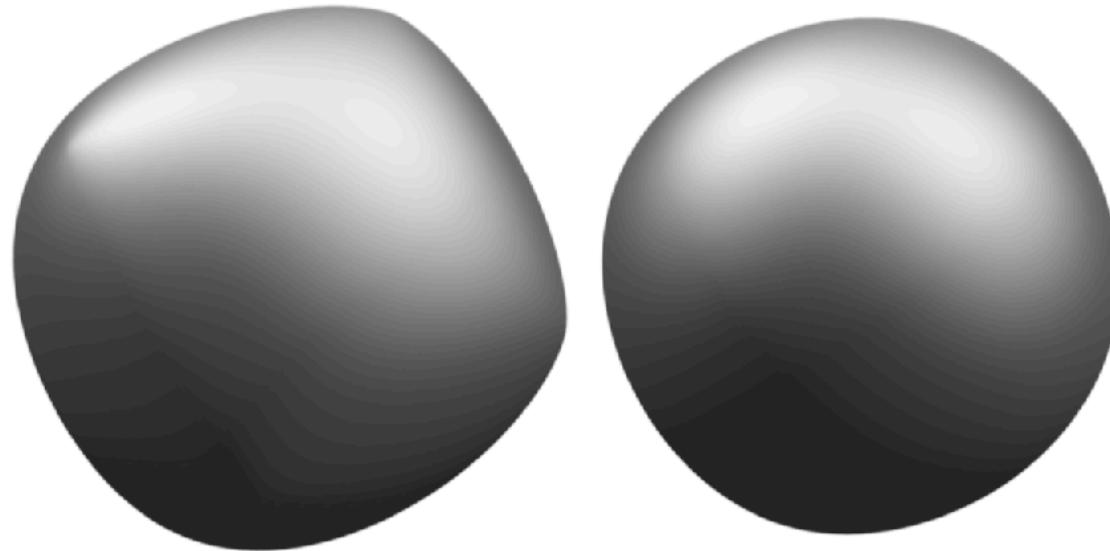
Implementing subdivs

- Step 2: averaging pass
 - smooth vertices based on their neighbors
 - update vertex position as the average of the centroids of the faces touching it
 - implemented just like smoothing normals

$$\mathbf{p}_i'' = \frac{\sum_{F \in adj(\mathbf{p}_i)} \text{centroid}(F)}{\sum_{F \in adj(\mathbf{p}_i)} 1}$$

Implementing subdivs

- Step 3: correction pass
 - need to correct for vertices that have non-standard number of adjacent faces



[Warren & Shaefer]

Implementing subdivs

- Step 3: correction pass
 - need to correct for vertices that have non-standard number of adjacent faces
 - correct vertex position after linear subdivision with vertex position after averaging

$$\mathbf{p}_i''' = \mathbf{p}_i' + (\mathbf{p}_i'' - \mathbf{p}_i') \frac{4}{\sum_{F \in adj(\mathbf{p}_i)} 1}$$

Implementing subdivs

- Handling boundaries
 - boundaries use different rules to fix them during subdivision
 - the previous algorithm can be easily modified to account for that
 - boundary edges can be detected with the edge dictionary
 - we tag each subdivided vertex as either being a boundary vertex, belonging to a binary edge or otherwise
 - for boundary vertices, no averaging is performed
 - for vertices on boundary edges, we average the edge vertices
 - for all other vertices, we apply the corrections before
- Handling texture coords
 - for texture coords we lock boundary vertices to maintain discontinuities

Implementing subdivs

- A possible implementation follows the three phases described before

```
void subdivide_catmullclark(vector<vec4i> quads,
                             vector<vec3f> pos, bool lock_boundary) {
    <initialize edges>
    <create vertices> // phase 1
    <create faces>
    <setup boundary>
    <averaging>        // phase 2
    <correction>       // phase 3
}
```

Implementing subdivs

- During initialization, we create the edge dictionary

```
<initialize edges> {
    // construct edge map and get edges and boundary
    auto emap      = make_edge_map(quads);
    auto edges     = get_edges(emap);
    auto boundary = get_boundary(emap);
    // initialize number of elements
    auto nv = (int)vert.size();
    auto ne = (int)edges.size();
    auto nb = (int)boundary.size();
    auto nf = (int)quads.size();
}
```

Implementing subdivs

- Then we first initialize the array of vertices and make a vertex for each vertex, edge and face in the original mesh

```
<create vertices> {
    auto tverts = vector<vec3f>();
    for (auto v : verts) // vertices [0, nv)
        tverts.push_back(v);
    for (auto e : edges) // edge vertices [nv, nv+ne)
        tverts.push_back((verts[e.x] + verts[e.y]) / 2);
    for (auto q : quads) // face vertices [nv+ne, nv+ne+nf)
        if (q.z != q.w)      // quads
            tverts.push_back((verts[q.x] + verts[q.y] +
                                verts[q.z] + verts[q.w]) / 4);
        else                  // triangles
            tverts.push_back((verts[q.x] + verts[q.y] +
                                verts[q.y]) / 3);
}
```

Implementing subdivs

- Then we create faces by splitting each face into 3 or 4 quads since here we support both triangles and quads

```
<create faces> {
    auto tquads = vector<vec4i>();
    for (auto [i, q] : enumerate(quads)) {
        if(q.z != q.w) {
            tquads.push_back({q.x, nv+emap[q.x, q.y],
                               nv+ne+i, nv+emap[q.w, q.x]}));
            tquads.push_back({q.y, nv+emap[q.y, q.z],
                               nv+ne+i, nv+emap[q.x, q.y]}));
            tquads.push_back({q.z, nv+emap[q.z, q.w],
                               nv+ne+i, nv+emap[q.y, q.z]}));
            tquads.push_back({q.w, nv+emap[q.w, q.x],
                               nv+ne+i, nv+emap[q.z, q.w]}));
        } else { ... }
    }
}
```

Implementing subdivs

- Then we create faces by splitting each face into 3 or 4 quads since here we support both triangles and quads

```
<create faces> {
    auto tquads = vector<vec4i>();
    for (auto [i, q] : enumerate(quads)) {
        if(q.z != q.w) {

            ...
        } else {
            tquads.push_back({q.x, nv+emap[q.x, q.y],
                               nv+ne+i, nv+emap[q.z, q.x]}));
            tquads.push_back({q.y, nv+emap[q.y, q.z],
                               nv+ne+i, nv+emap[q.x, q.y]}));
            tquads.push_back({q.z, nv+emap[q.z, q.x],
                               nv+ne+i, nv+emap[q.y, q.z]}));
        }
    }
}
```

Implementing subdivs

- We then setup the boundary arrays to track the indices of boundary vertices

```
<setup boundary> {
    auto tboundary = vector<vec2i>();
    for (auto e : boundary) {
        tboundary.push_back({e.x, nv+emap[e]} );
        tboundary.push_back({nv+emap[e], e.y} );
    }
    auto tcrease_edges = vector<vec2i>();
    auto tcrease_verts = vector<int>();
    if (lock_boundary) {
        for (auto& b : tboundary) {
            tcrease_verts.push_back(b.x);
            tcrease_verts.push_back(b.y);
        }
    } else {
        for (auto& b : tboundary) tcrease_edges.push_back(b);
    }
}
```

Implementing subdivs

- To apply subdivision rules we need to track whether a vertex comes from a boundary vertex, a boundary edge, or otherwise
- We do so by tracking vertex valence which is 0 for boundary vertices, 1 for boundary edges, or 2 otherwise

```
<setup boundary> {
    auto tverts_val = vector<int>(tverts.size(), 2);
    for (auto& e : tboundary) {
        tvert_val[e.x] = (lock_boundary) ? 0 : 1;
        tvert_val[e.y] = (lock_boundary) ? 0 : 1;
    }
}
```

Implementing subdivs

- We then apply averaging rules

```
<averaging> {
    auto avert = vector<vec3f>(tvert.size(), zero3f);
    auto account = vector<int>(tvert.size(), 0);
    for (auto p : tcrease_verts) {
        if (tvert_val[p] == 0) continue;
        avert[p] += tverts[p]; account[p] += 1;
    }
    for (auto& e : tcrease_edges) {
        auto c = (tverts[e.x] + tverts[e.y]) / 2;
        for (auto vid : e) {
            if (tverts_val[vid] != 1) continue;
            avert[vid] += c; account[vid] += 1;
        }
    }
    ...
}
```

Implementing subdivs

- We then apply averaging rules

```
<averaging> {
    ...
    for (auto& q : tquads) {
        auto c = (tverts[q.x] + tverts[q.y] +
                  tverts[q.z] + tverts[q.w]) / 4;
        for (auto vid : q) {
            if (tverts_val[vid] != 2) continue;
            avert[vid] += c; account[vid] += 1;
        }
    }
    for (auto i : range(tverts))
        avert[i] /= (float)account[i];
}
```

Implementing subdivs

- Finally we perform the correction pass

```
<correction> {
    for (auto i : range(tverts)) {
        if (tvert_val[i] != 2) continue;
        averts[i] = tverts[i] +
            (averts[i] - tverts[i]) * (4 / (float)account[i]);
    }
    tverts = averts;
}
```

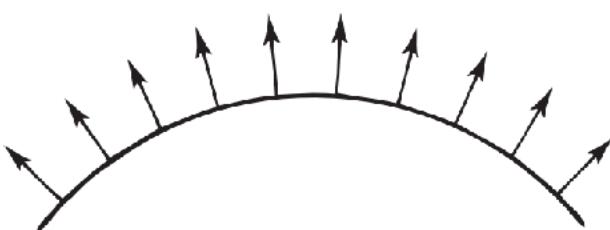
Displacements and Bumps

Adding details back

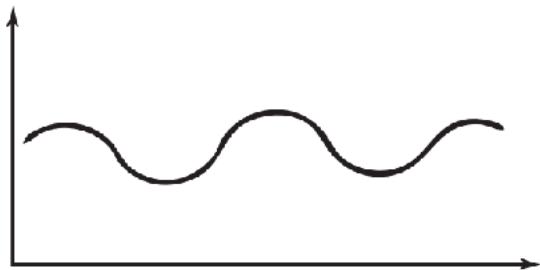
- Subdivision surfaces are a controllable and smooth making them the most common method for modeling smooth shapes
- But subdivs lack fine scale details that detailed meshes have
- To get details, we could increase significantly the resolution for the control mesh
- But this would be essentially make us go back to the same vertex density of triangle/quad meshes, with the added inefficiency of having a subdiv
- Instead we use images to control geometric details, just like textures control material properties
- This is also helpful when rendering triangle meshes on the GPU

Displacement mapping

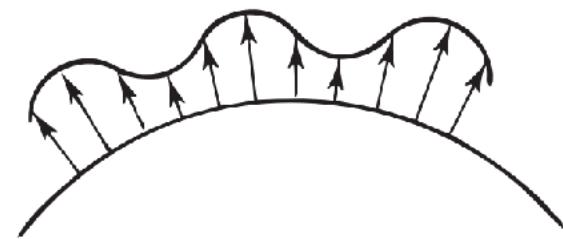
- Displacement mapping: use a texture that defines the displacement along the normal direction at each surface location



(a) original surface



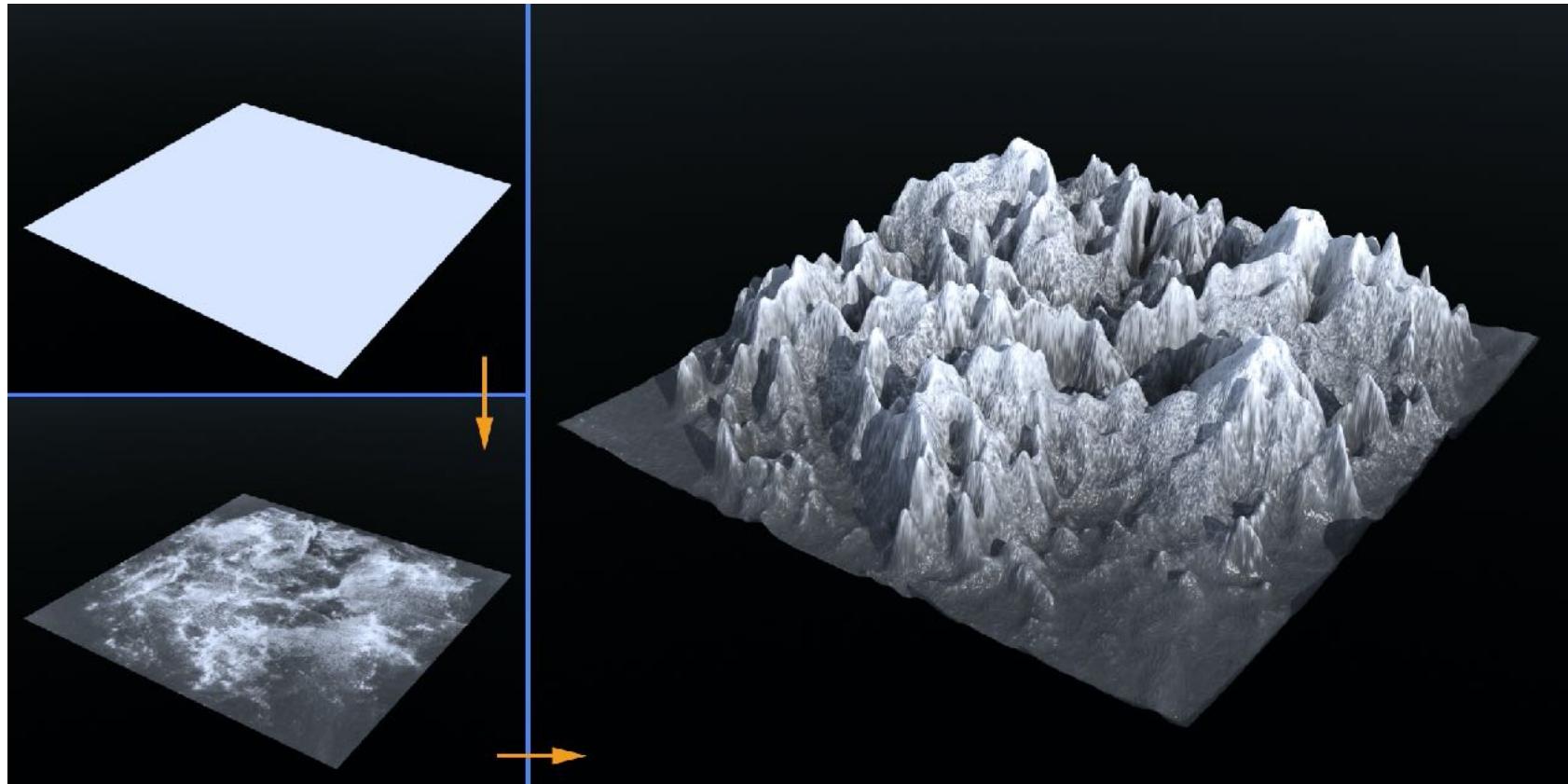
(b) height map



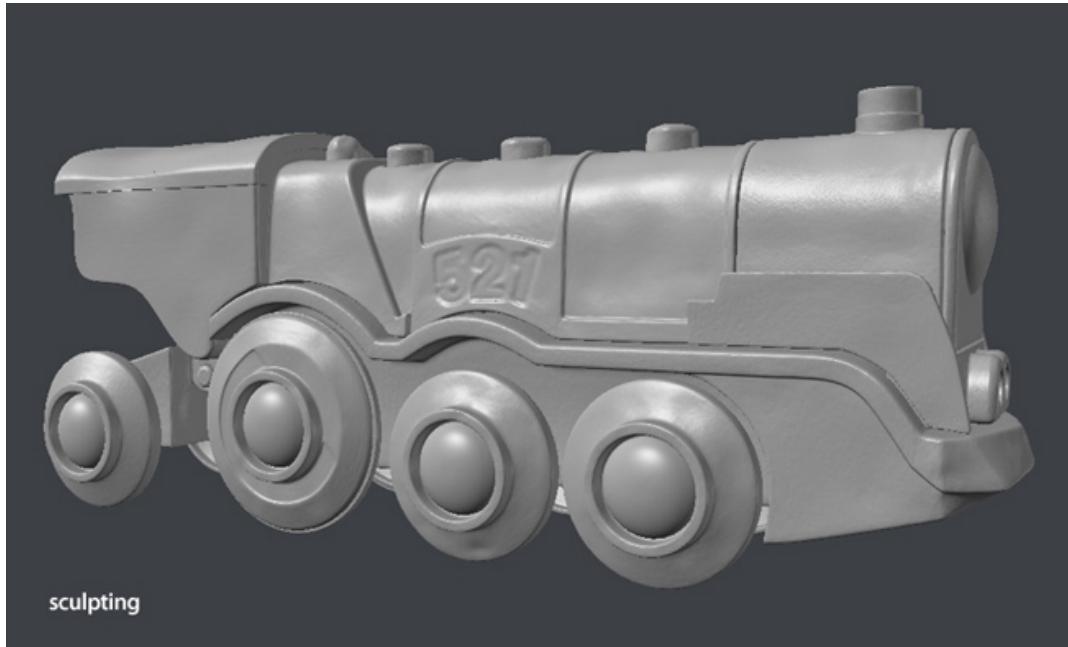
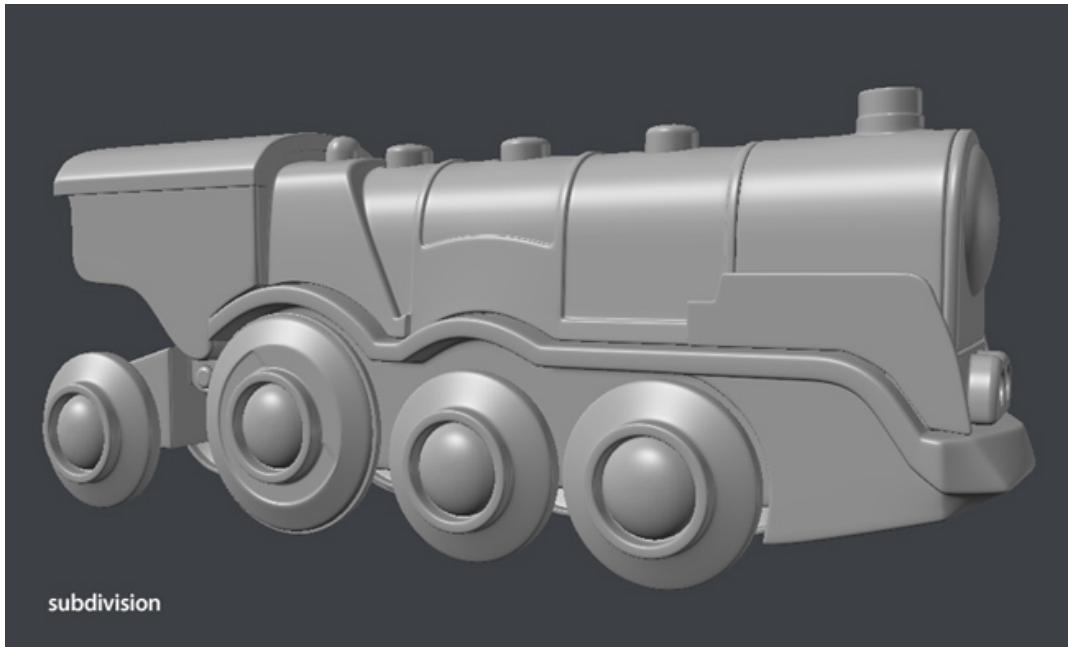
(c) new surface

Displacement mapping

- Displacement mapping: use a texture that defines the displacement along the normal direction at each surface location



[Wikimedia Commons]



[Pawel Filip]



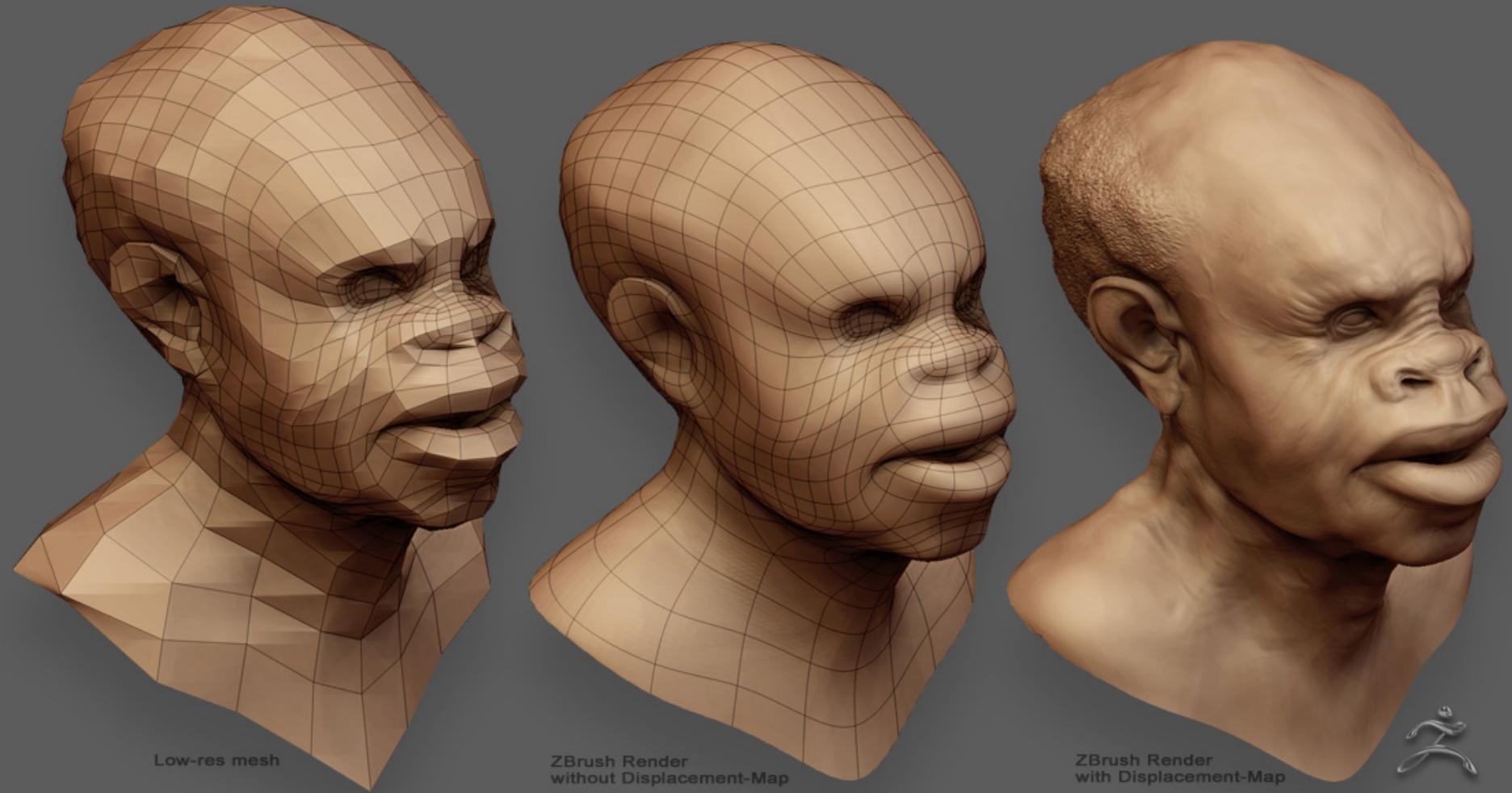
fryrender

physically-based render engine
Computer Graphics

©2007 Paweł Filip

© 2020 Fabio Pellacini • 66

Displacement mapping



Displacement mapping



[Source unknown]

Displacement mapping

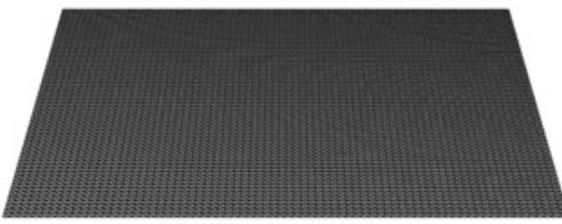


[Yining Karl]

Displacement mapping

- Write vertex position using a parametrization to index the texture
$$\mathbf{p}^d(u, v) = \mathbf{p}(u, v) + h(u, v)\mathbf{n}(u, v)$$
- To render a displaced triangle mesh, we tessellate it at the appropriate resolution and displace each vertex by the amount specified in the texture along the vertex normal

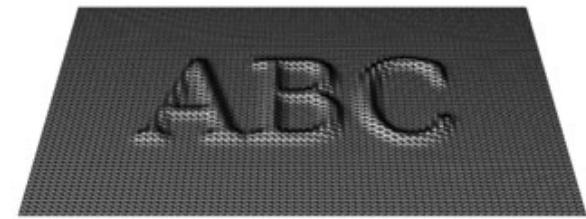
$$\mathbf{p}_i^d = \mathbf{p}_i + h[u_i, v_i]\mathbf{n}_i$$



ORIGINAL MESH



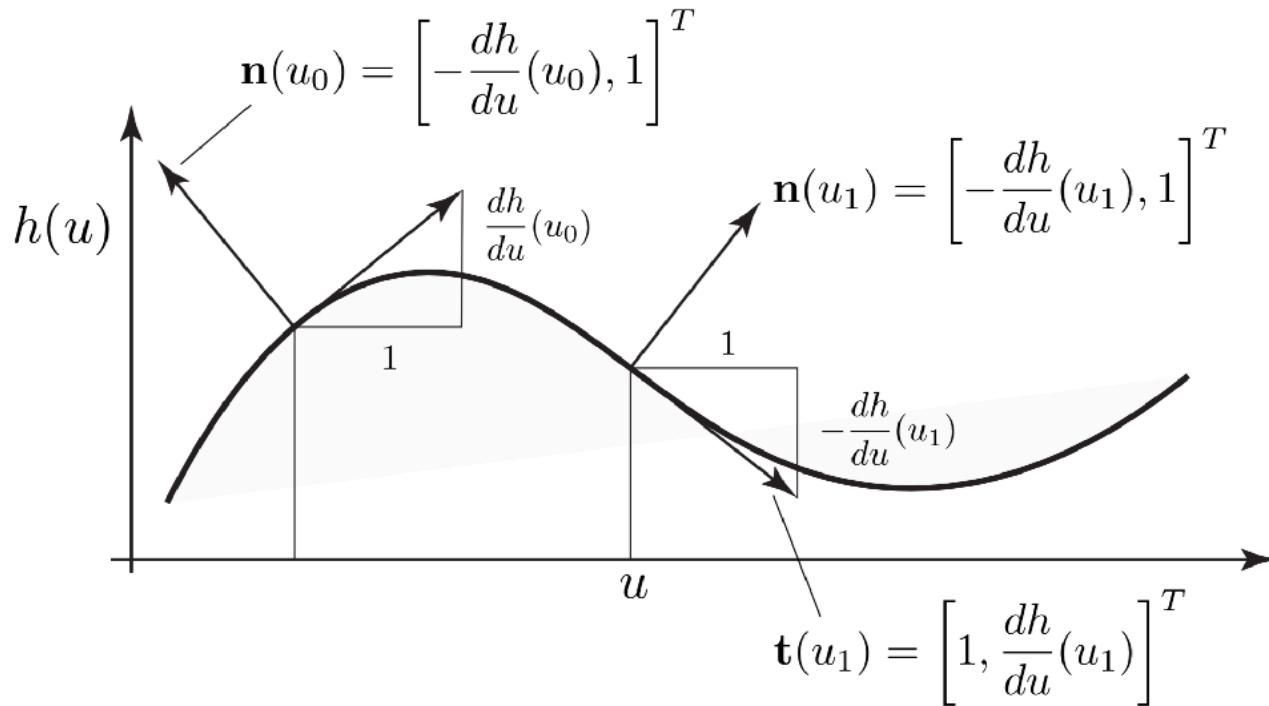
DISPLACEMENT MAP



MESH WITH DISPLACEMENT

Computing displaced normals

- Option 1: smooth normal algorithm ran on the displaced positions
 - sound and practical, but may be inefficient
- Option 2: estimate directly from the positions and texture



Computing displaced normals

- Normals are vectors orthogonal to the tangent plane that can be computed as the cross product of two tangents
- For a continuous surface, we can write displaced normals as the cross product of the tangents of the displaced positions, which are the surface derivatives w.r.t. each parameter

$$\begin{aligned}\mathbf{n}^d(u, v) &= \frac{\partial \mathbf{p}^d(u, v)}{\partial u} \times \frac{\partial \mathbf{p}^d(u, v)}{\partial v} = \\ &= \frac{\partial [\mathbf{p}(u, v) + h(u, v)\mathbf{n}(u, v)]}{\partial u} \times \frac{\partial [\mathbf{p}(u, v) + h(u, v)\mathbf{n}(u, v)]}{\partial v}\end{aligned}$$

Computing displaced normals

- We can simplify the above equation by assuming that the normal changes little around the point

$$\begin{aligned}\mathbf{n}^d &= \frac{\partial[\mathbf{p} + h\mathbf{n}]}{\partial u} \times \frac{\partial[\mathbf{p} + h\mathbf{n}]}{\partial v} \approx \left[\frac{\partial \mathbf{p}}{\partial u} + \frac{\partial h}{\partial u} \mathbf{n} \right] \times \left[\frac{\partial \mathbf{p}}{\partial v} + \frac{\partial h}{\partial v} \mathbf{n} \right] = \\ &= \left[\frac{\partial \mathbf{p}}{\partial u} + \frac{\partial h}{\partial u} \mathbf{n} \right] \times \left[\frac{\partial \mathbf{p}}{\partial v} + \frac{\partial h}{\partial v} \mathbf{n} \right] = \left[\mathbf{t}_u + \frac{\partial h}{\partial u} \mathbf{n} \right] \times \left[\mathbf{t}_v + \frac{\partial h}{\partial v} \mathbf{n} \right] = \\ &= \mathbf{t}_u \times \mathbf{t}_v + \frac{\partial h}{\partial v} (\mathbf{t}_u \times \mathbf{n}) + \frac{\partial h}{\partial v} (\mathbf{n} \times \mathbf{t}_v)\end{aligned}$$

- For tangents that are orthogonal and normalized we have

$$\mathbf{n}^d = -\frac{\partial h}{\partial u} \mathbf{t}_u - \frac{\partial h}{\partial v} \mathbf{t}_v + \mathbf{n}$$

Tangent frame

- *Tangent frame*: coordinate system at each surface location, with surface normal as z and tangents as x and y

$$T = \{\mathbf{t}_u, \mathbf{t}_v, \mathbf{n}, \mathbf{p}\}$$

- Displacement can be written w.r.t. the tangent frame as

$$\mathbf{p}^d = T [0 \quad 0 \quad h \quad 1]^T \quad \mathbf{n}^d = T \left[-\frac{\partial h}{\partial u} \quad -\frac{\partial h}{\partial v} \quad 1 \quad 0 \right]^T$$

- Can think of the tangent frame as defining a plane that locally approximates the surface
- Tangent frame depends on texture orientation since tangents are defined w.r.t. (u, v)

Tangent frame for meshes

- On triangle meshes, derive tangent frames from vertex positions and texture coordinates
 - derivation in www.terathon.com/code/tangent.html
- Would like to define two tangents ($\mathbf{t}_u, \mathbf{t}_v$) such as we can write all points \mathbf{p} on a triangle in terms of the triangle (u,v) as

$$\mathbf{p}(u, v) = \mathbf{v}_0 + (u - u_0)\mathbf{t}_u + (v - v_0)\mathbf{t}_v$$

- We impose two constraints on the other triangle vertices

$$\begin{cases} \mathbf{p}(u_1, v_1) = \mathbf{v}_1 \\ \mathbf{p}(u_2, v_2) = \mathbf{v}_2 \end{cases} \Rightarrow \begin{cases} \mathbf{v}_1 = \mathbf{v}_0 + (u_1 - u_0)\mathbf{t}_u + (v_1 - v_0)\mathbf{t}_v \\ \mathbf{v}_2 = \mathbf{v}_0 + (u_2 - u_0)\mathbf{t}_u + (v_2 - v_0)\mathbf{t}_v \end{cases}$$

Tangent frames for meshes

- We can manipulate a bit the previous equations as

$$\begin{cases} \mathbf{v}_1 - \mathbf{v}_0 = (u_1 - u_0)\mathbf{t}_u + (v_1 - v_0)\mathbf{t}_v \\ \mathbf{v}_2 - \mathbf{v}_0 = (u_2 - u_0)\mathbf{t}_u + (v_2 - v_0)\mathbf{t}_v \end{cases}$$

- For notation we can rewrite it with easier constants as

$$\begin{cases} \mathbf{e}_1 = s_1\mathbf{t}_u + t_1\mathbf{t}_v \\ \mathbf{e}_2 = s_2\mathbf{t}_u + t_2\mathbf{t}_v \end{cases}$$

- This is a linear system of six equations in six unknowns ($\mathbf{t}_u, \mathbf{t}_v$)
- We can rewrite it again in matrix notation as

$$[\mathbf{e}_1 \quad \mathbf{e}_2] = [\mathbf{t}_u \quad \mathbf{t}_v] \begin{bmatrix} s_1 & s_2 \\ t_1 & t_2 \end{bmatrix}$$

Tangent frames for meshes

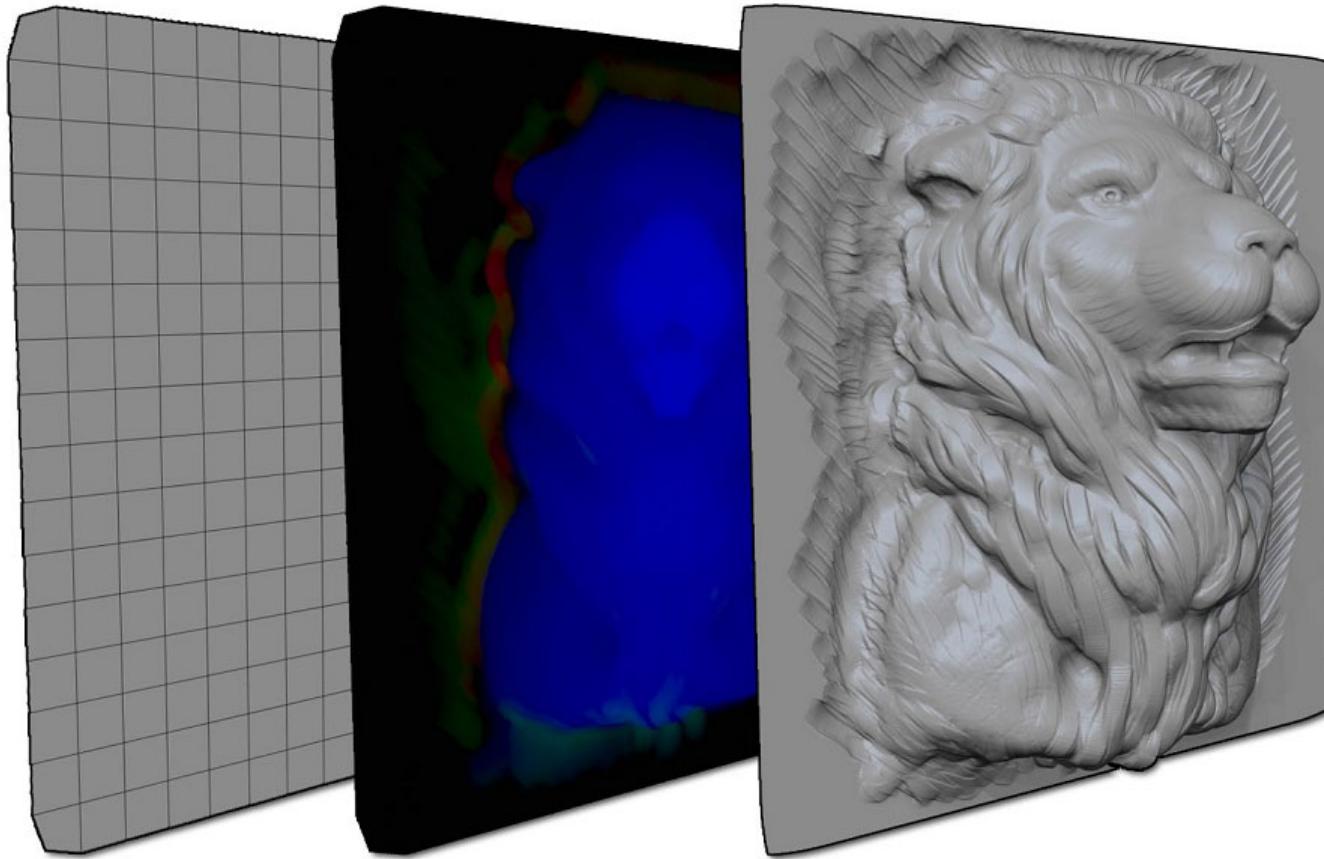
- We now multiply both sides by the inverse of the (s,t) matrix

$$[\mathbf{t}_u \quad \mathbf{t}_v] = [\mathbf{e}_1 \quad \mathbf{e}_2] \left(\frac{1}{s_1 t_2 - s_2 t_1} \begin{bmatrix} t_2 & -s_2 \\ -t_1 & s_1 \end{bmatrix} \right)$$

- This gives us the two required tangents for each triangle
- To get tangents, for the whole mesh, we compute smooth vertex tangents in the same manner we did to compute vertex normals
 - note that the tangents are not orthonormal, but we can address this with orthonormalization as written before
 - the most common way to store the tangents is to only store one of them per-vertex and the sign of their cross product
 - we then recuperate the other tangent with the cross product with the normal

Vector displacement

- Extend displacement mapping by specifying a tangent space vector as the offset for the new position



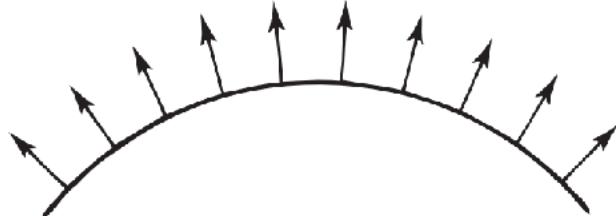
[Pixologic]

Vector displacement

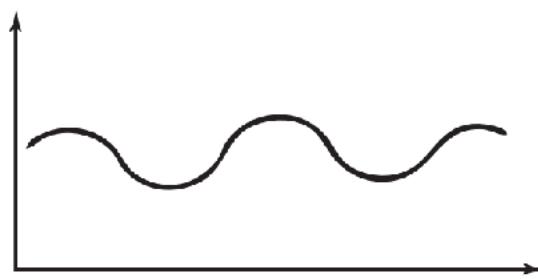


Bump mapping

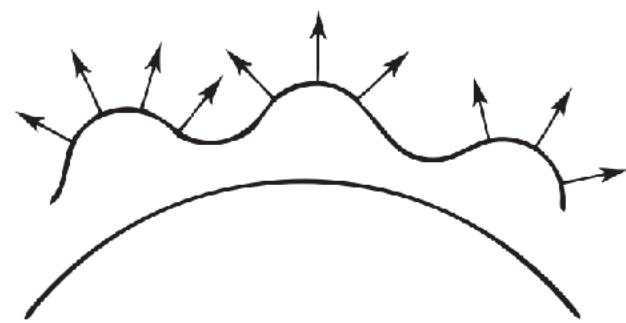
- Displacement mapping is expensive since it requires tessellation
- For small displacement, most effects are in the normal
- *Bump mapping*: uses displaced normal for shading, but original geometry for intersection and rasterization



(a) original surface



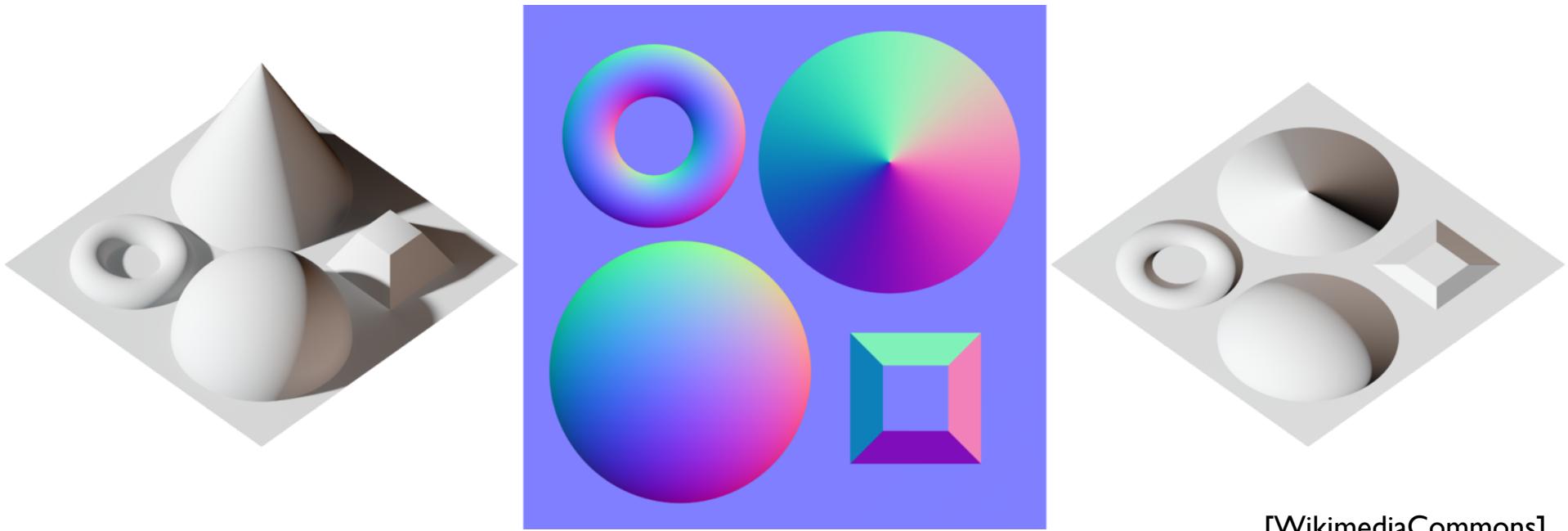
(b) height map



(d) perturbed normals

Normal mapping

- *Normal mapping*: directly store normals as texture colors
 - encode with $(r,g,b) = 0.5(x,y,z) + 0.5$
 - decode with $(x,y,z) = 2(r,g,b) - 1$



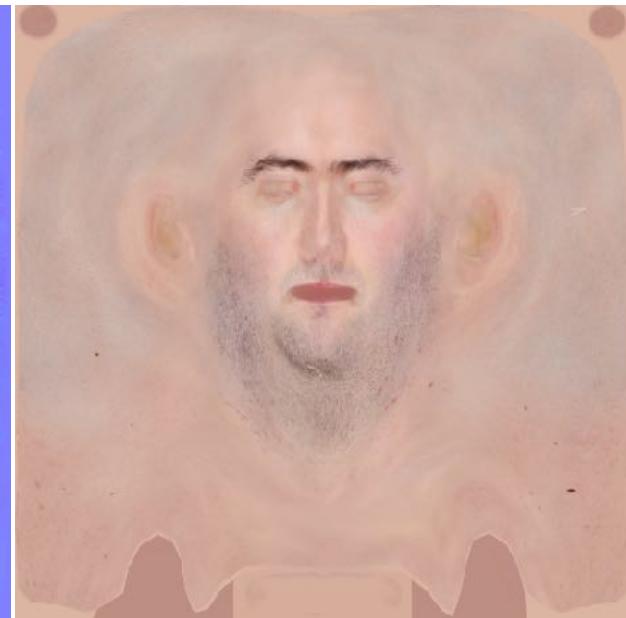
[Wikimedia Commons]

Normal mapping

- More general than bump mapping: can store arbitrary normals
- Mostly used in industry since it supports simplification of meshes



[BlenderNerd]



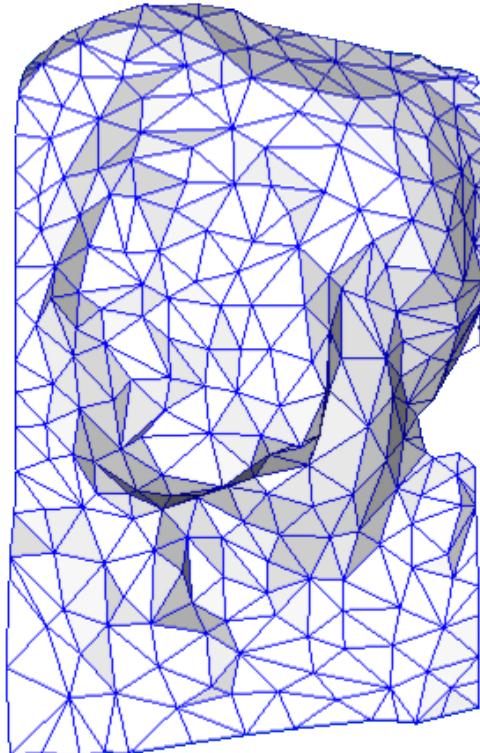
[Lee Perry-Smith]

Normal mapping

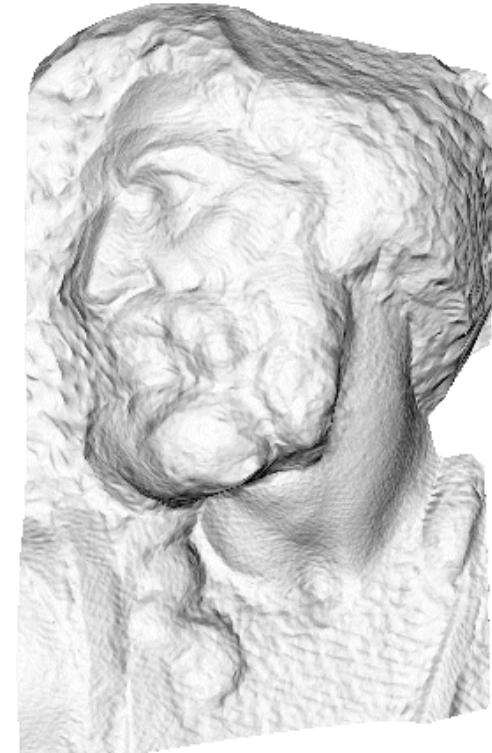
- Mostly used in industry since it supports simplification of meshes



original mesh
4M triangles



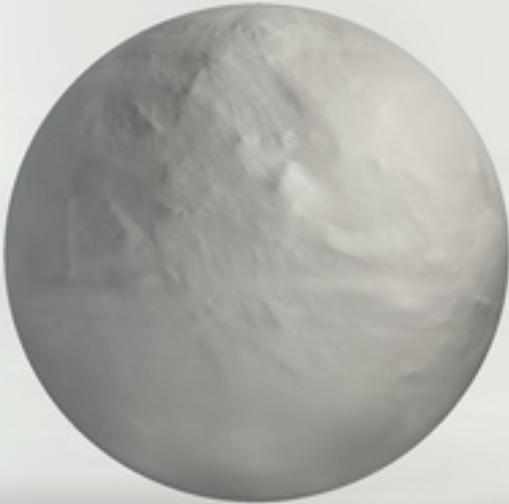
simplified mesh
500 triangles



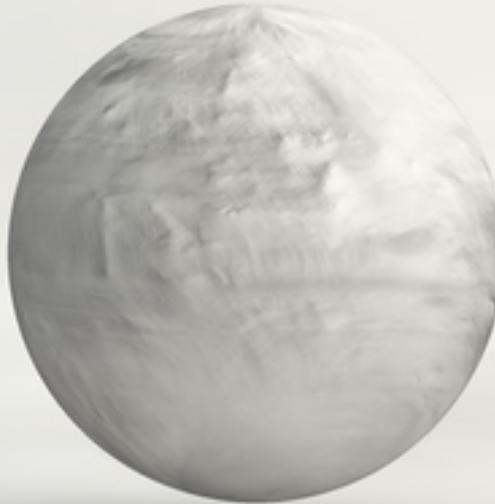
simplified mesh
and normal mapping
500 triangles

Bump vs normal vs displacement

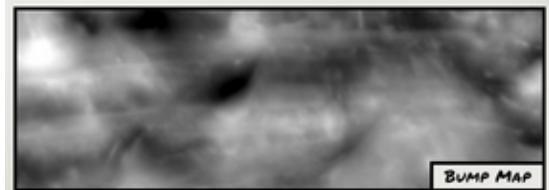
BUMP ONLY



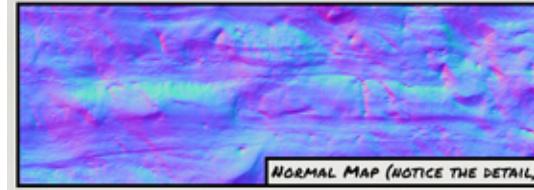
NORMAL ONLY



NORMAL + DISPLACEMENT



BUMP MAP

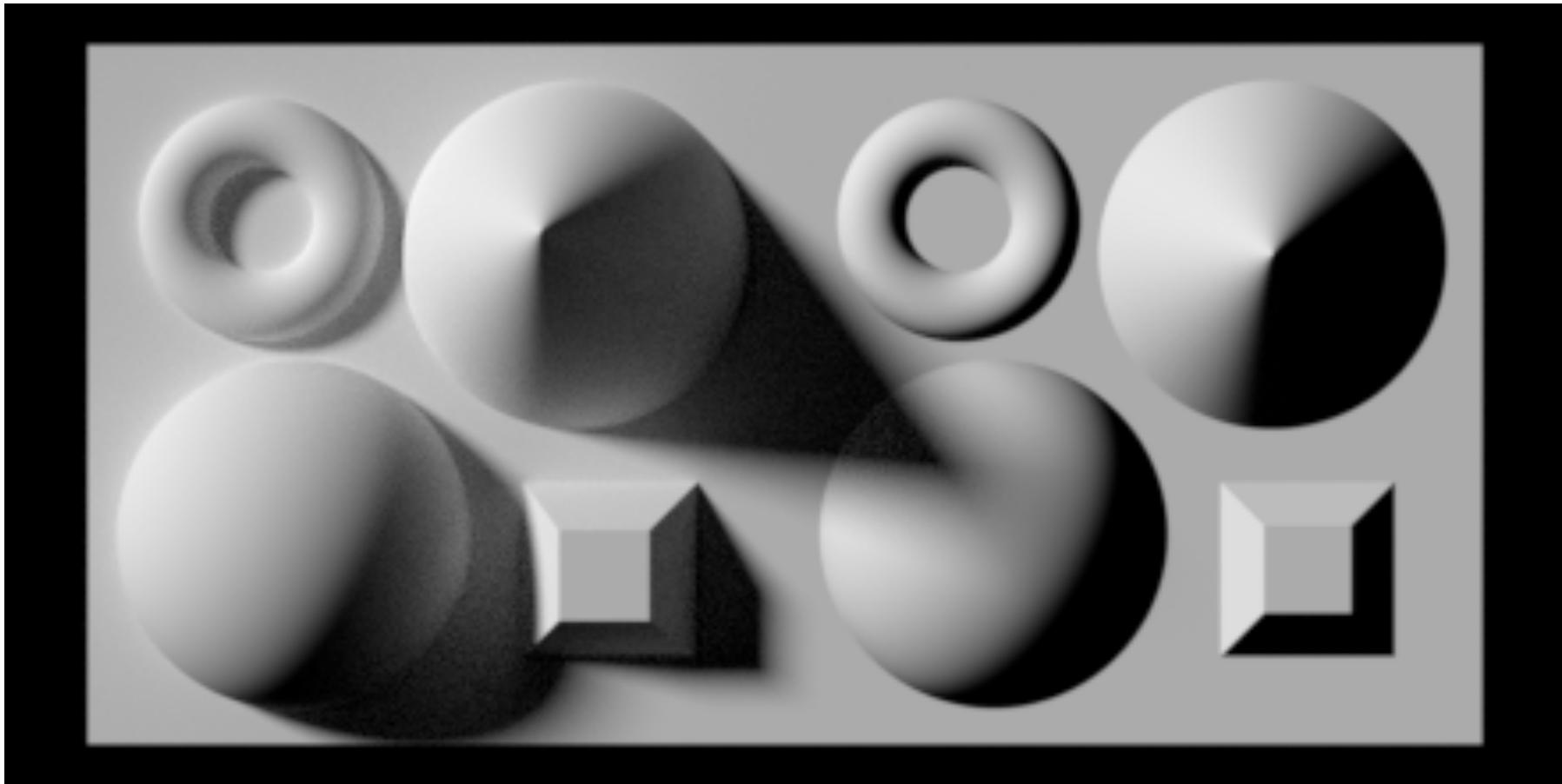


NORMAL MAP (NOTICE THE DETAIL)



DISPLACEMENT MAP (SAME AS BUMP)

Bump vs normal vs displacement



[Wikimedia Commons]

Normal and displacement

- Entertainment apps use normal or displacement everywhere
- Normal maps: used for real-time applications
 - mostly from compressed meshes
- Displacement maps: used in offline rendering for movies
 - production renderers are optimized for displacement
 - handle high details, while the rest of the pipeline uses “low-poly” control vertices

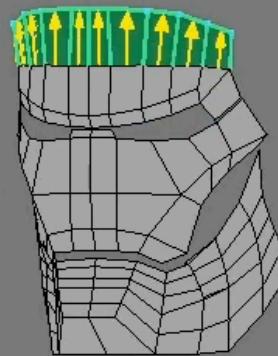
Editing Surfaces

Two different workflows

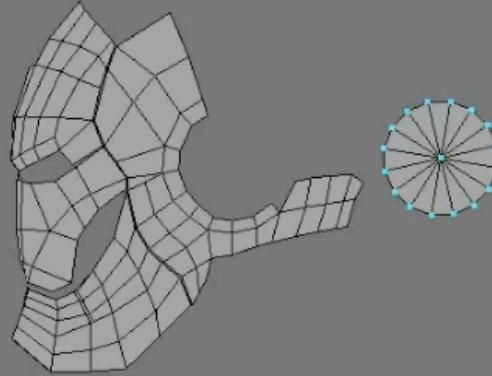
- Low-poly/subdiv modeling
 - topology and geometric operations
 - carefully modify each vertex
 - care a lot about mesh topology
 - hard to use but precise and efficient surfaces
- High-poly sculpting
 - sculpt like clay using “brushes”
 - mostly care about geometry, very few topology changes
 - easy to use but imprecise and highly tessellated surfaces
 - “retopology” tools to convert sculpt to subdiv

Low poly modeling

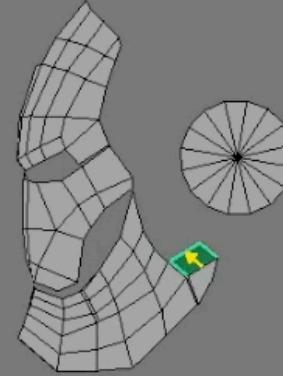
Level 9



Level 5

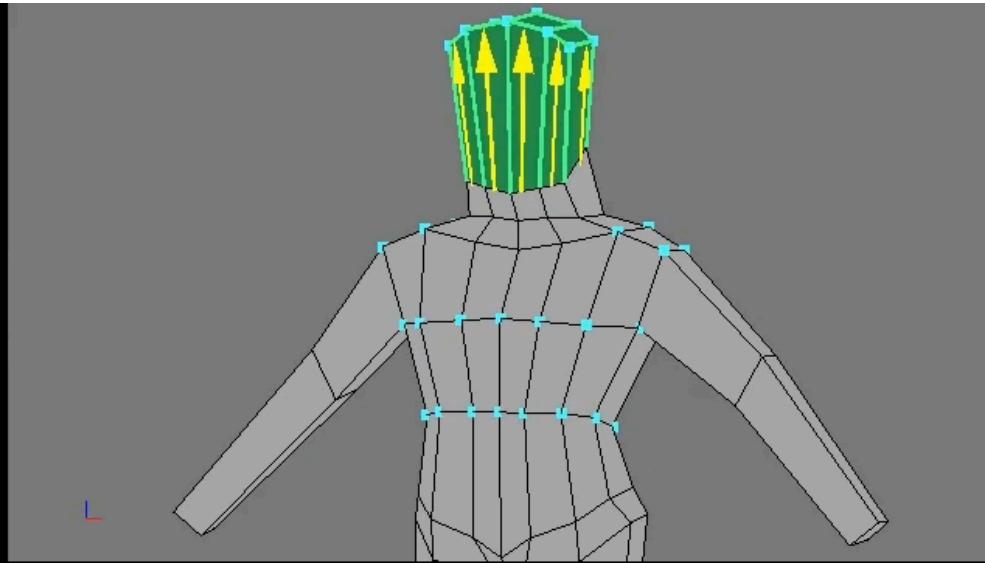


Level 7

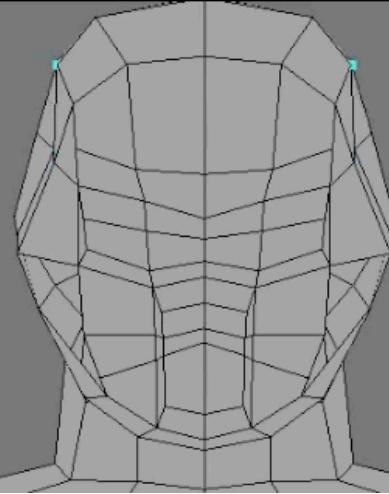


Low poly modeling

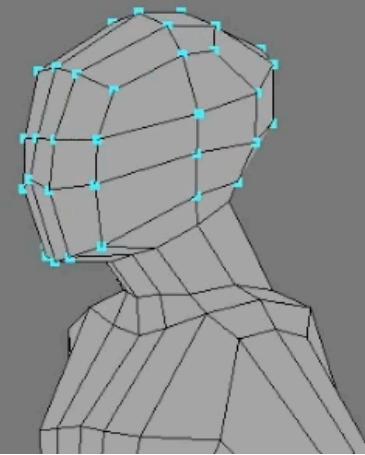
Level 9



Level 5

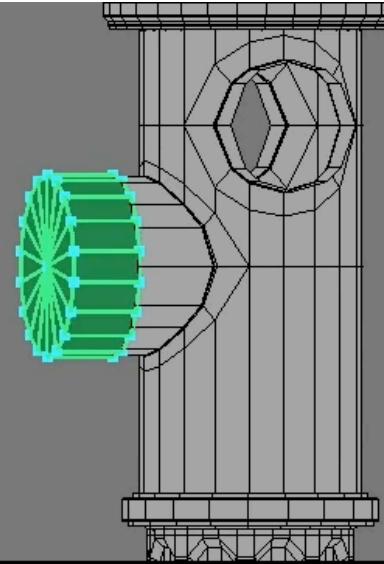


Level 7

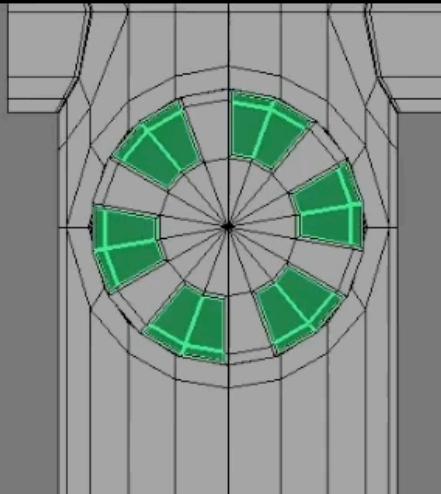


Low poly modeling

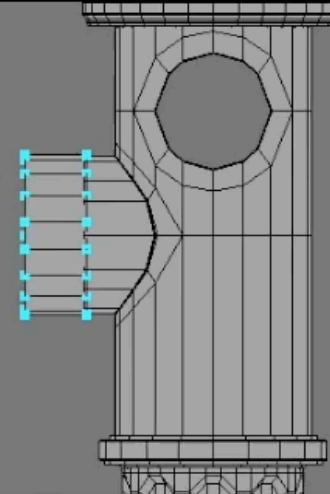
Level 9



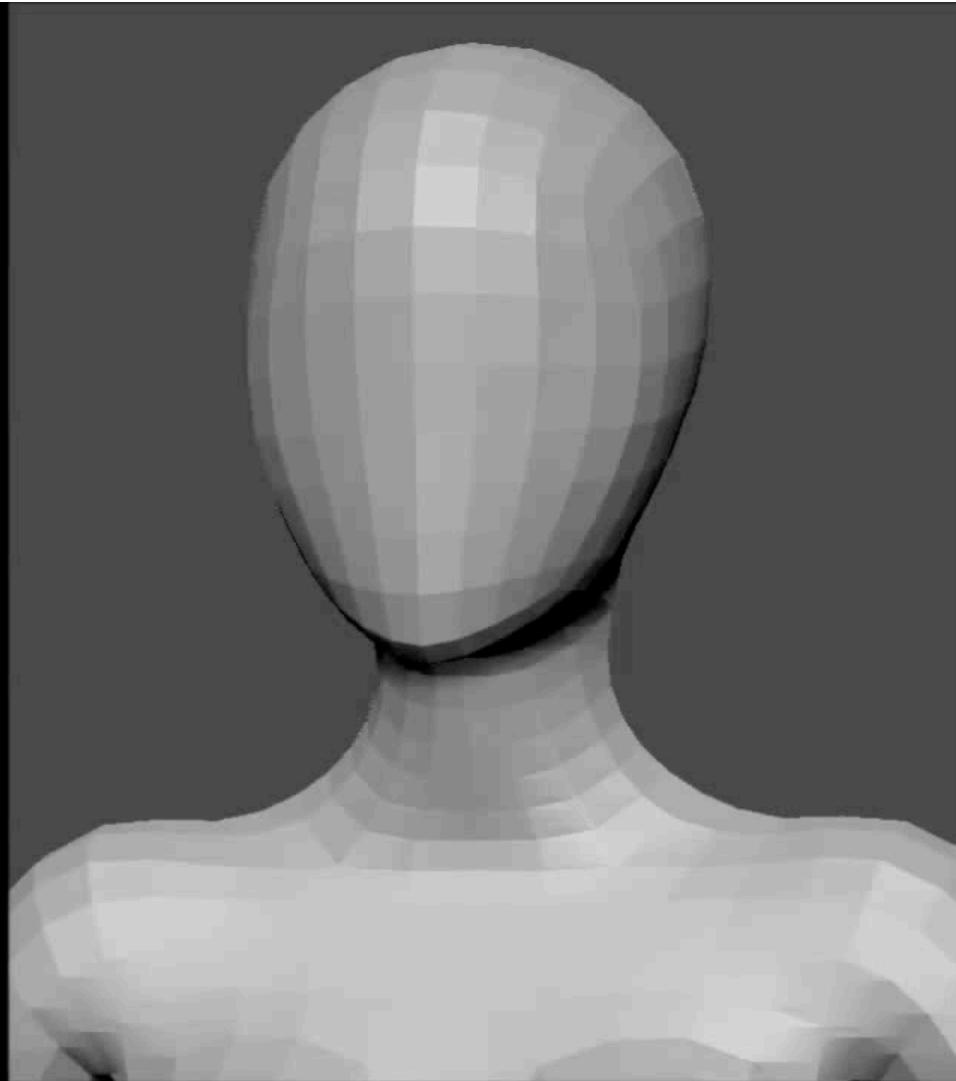
Level 5



Level 7



Sculpting



[Source unknown]

Sculpting

