Smooth and Rough Surfaces

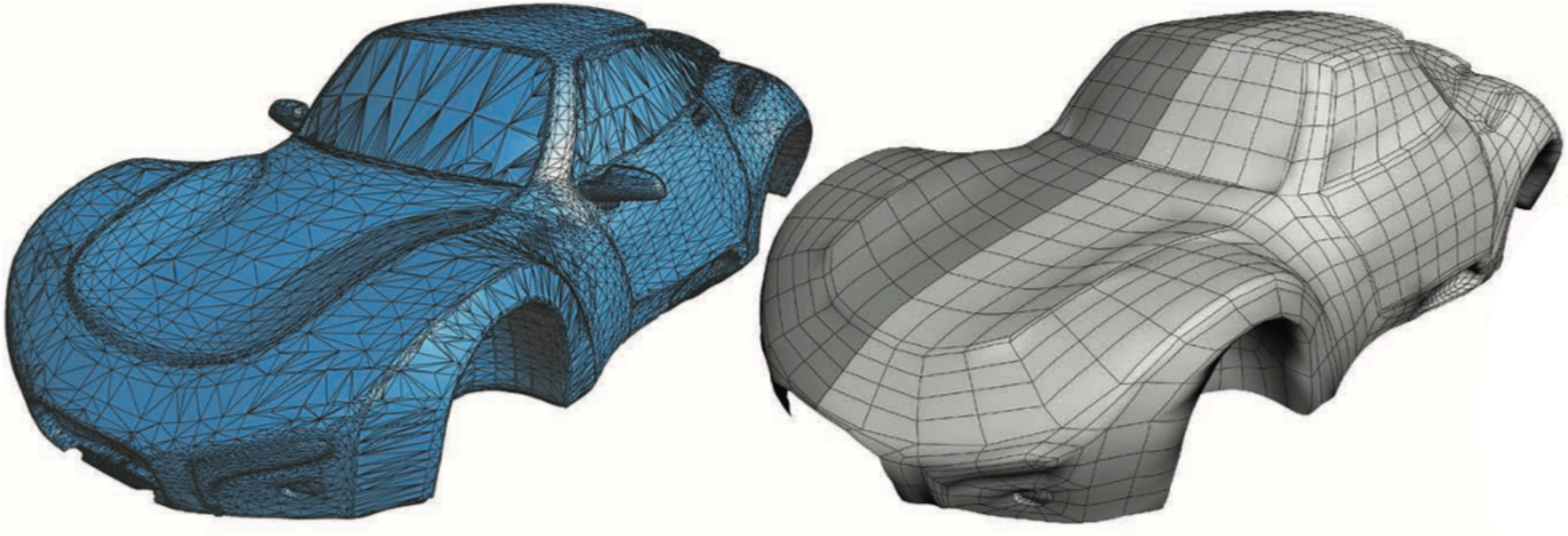
Prof. Fabio Pellacini

Triangle mesh recap

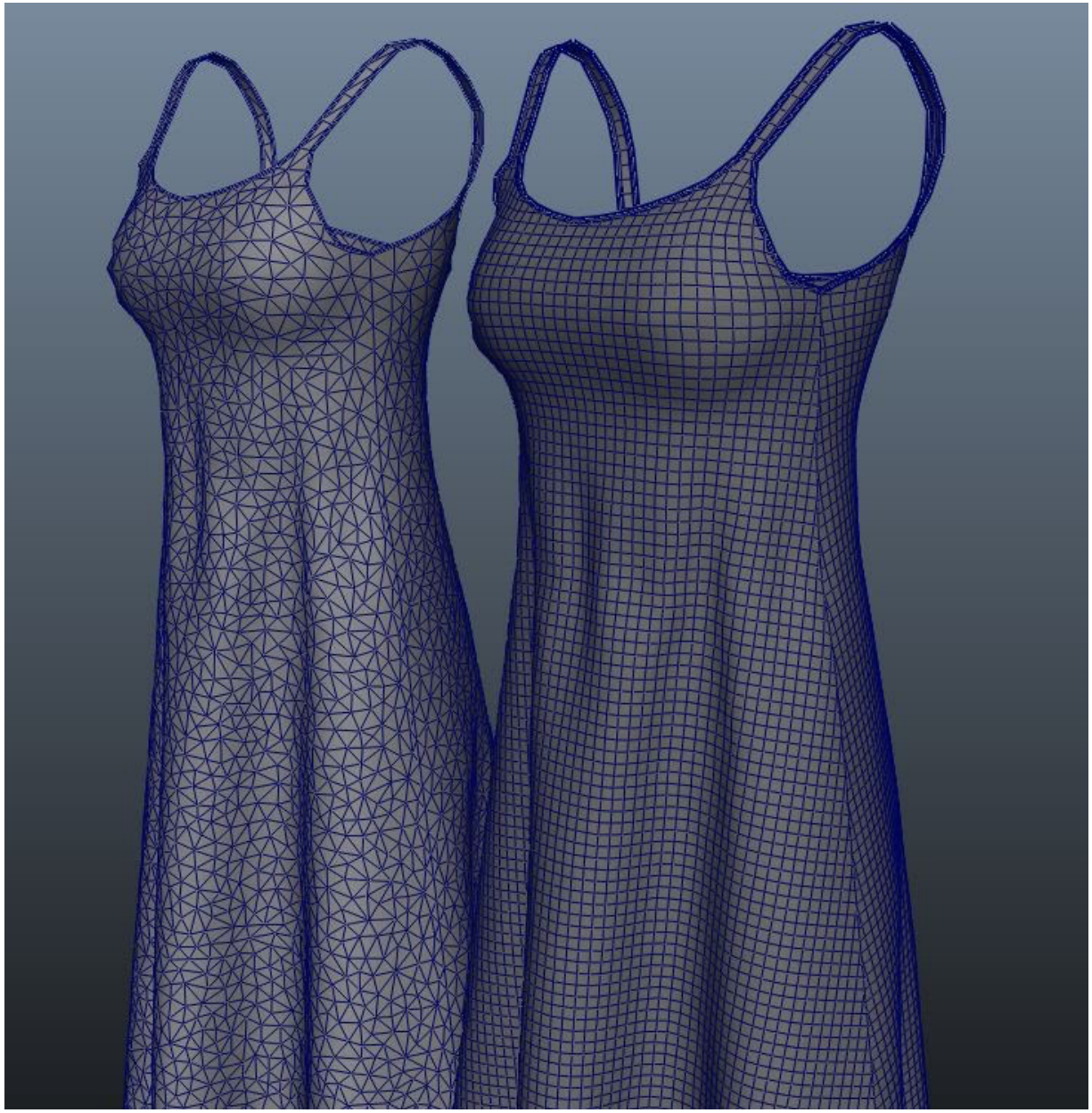
- Triangles have well defined geometric properties
  - they always live in a plane
  - so they have a unique geometric normal
- Triangle meshes are widely used
  - supported in rasterization and raytracing
  - sometimes used in simulation for cloth
- But it is hard to represent "regular" structures with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

Quad meshes

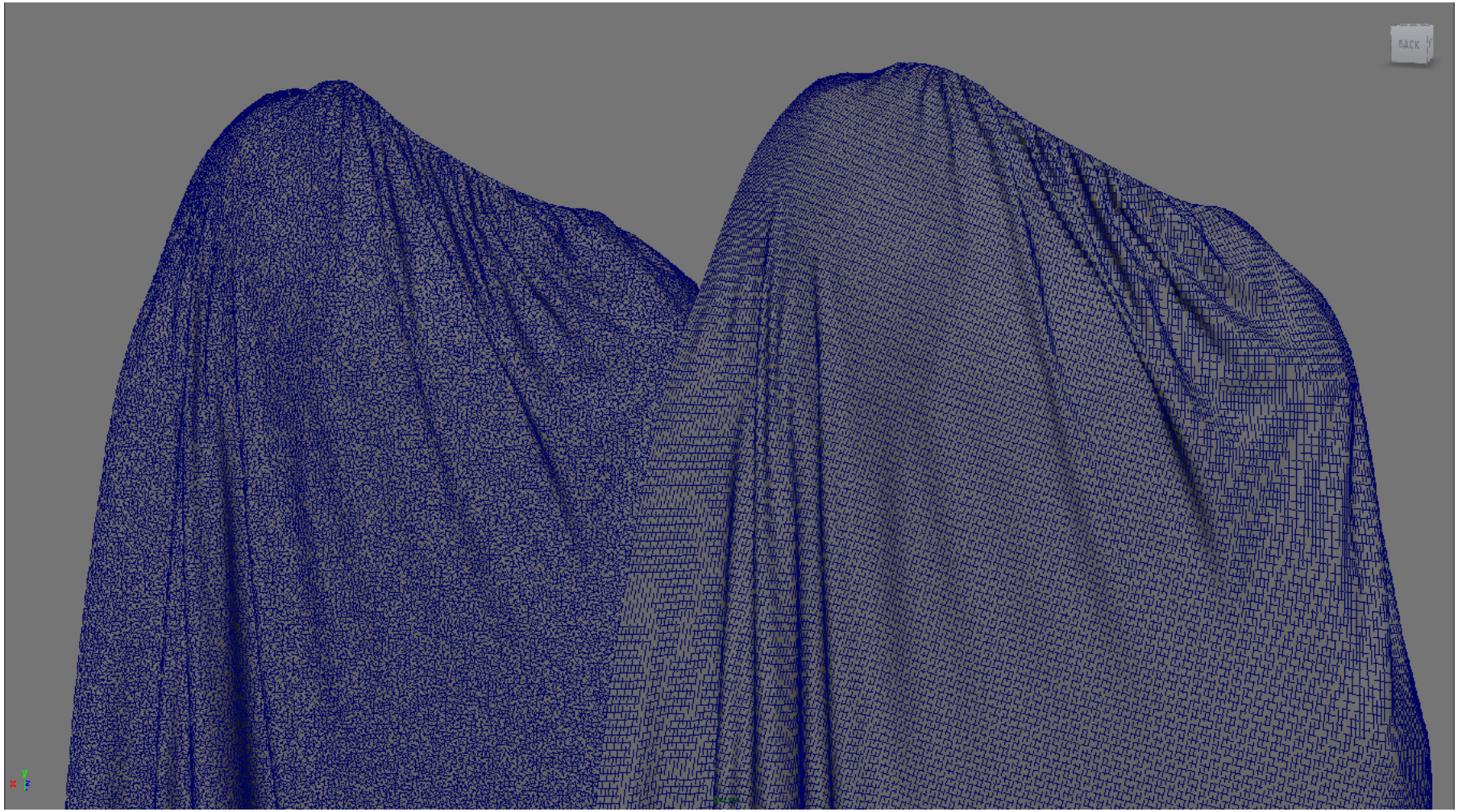
- Quad mesh: a collection of quadrilaterals
  - 4 vertices per face, possibly non-planar
  - use same data structure that triangles use



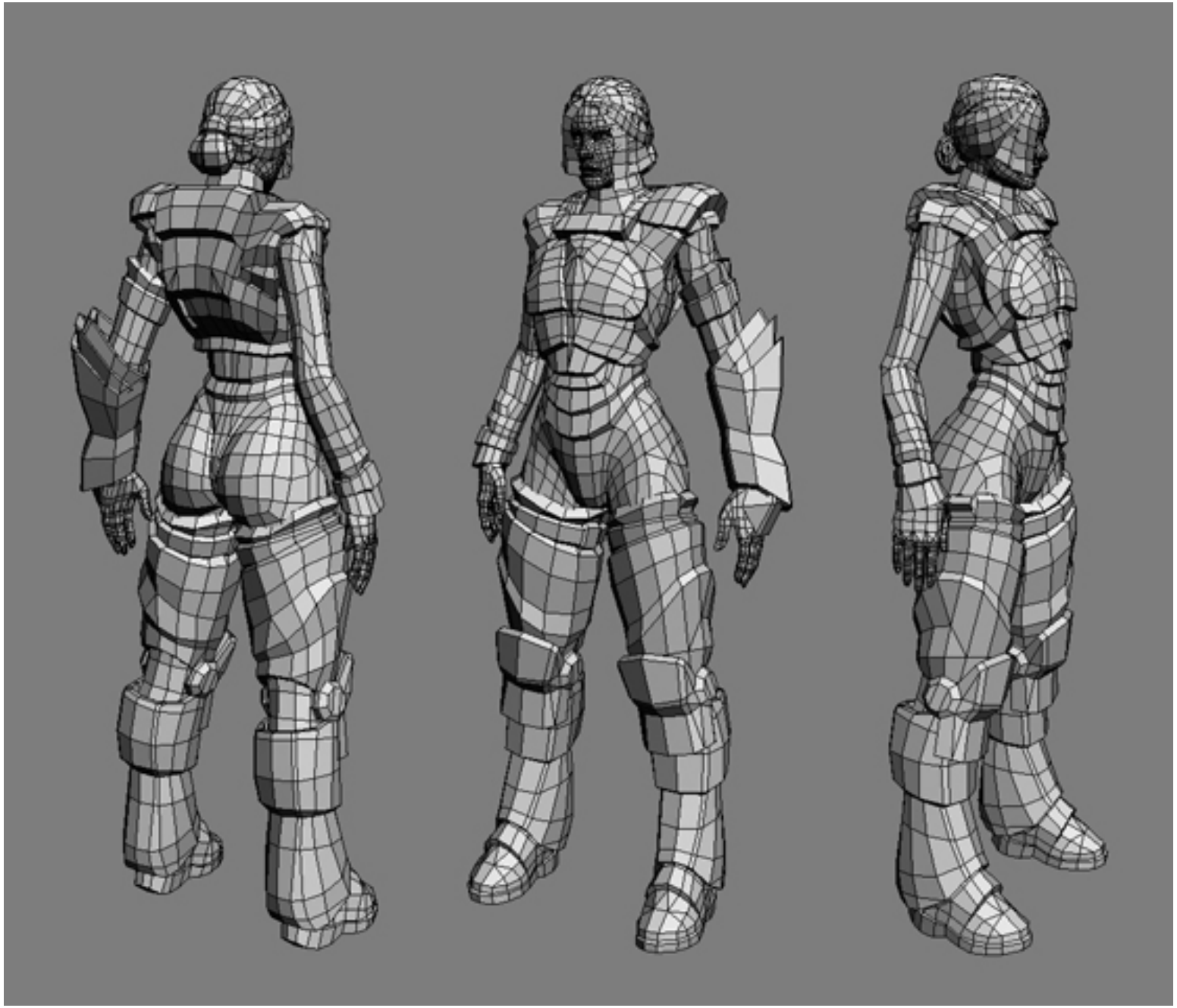
[Panozzo]



[source unknown]



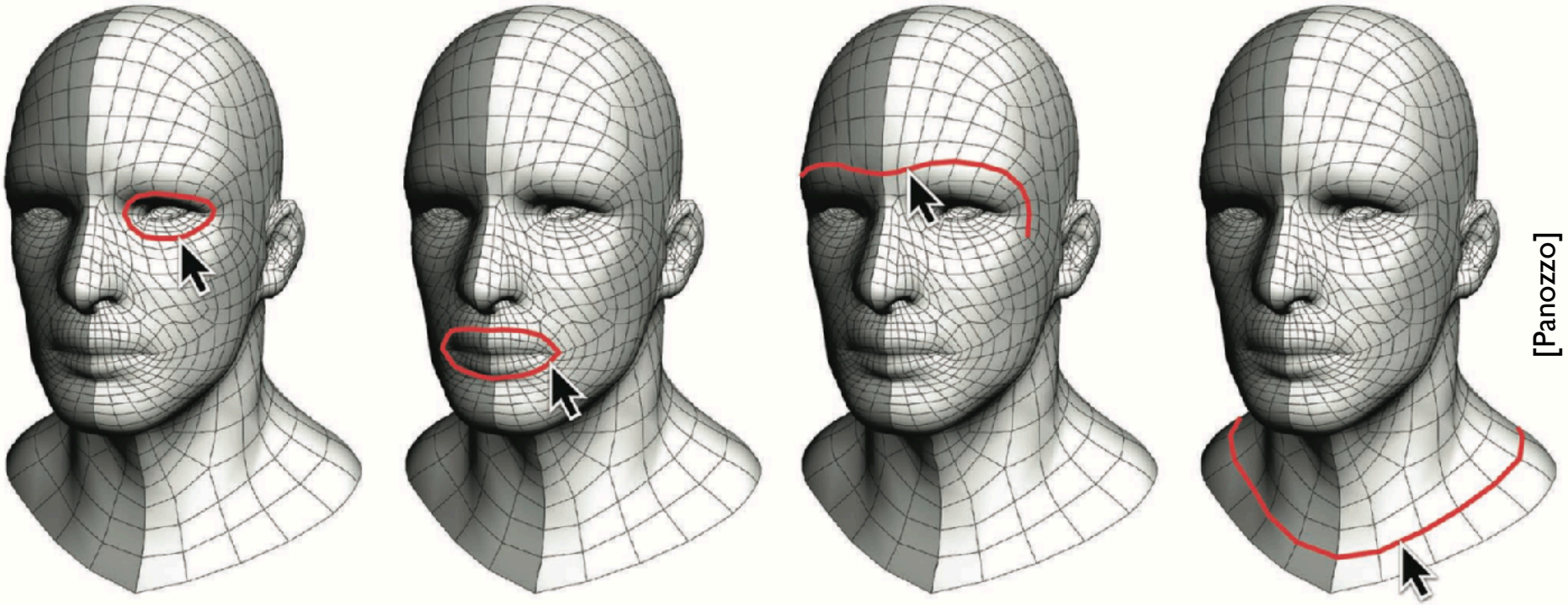
[Ken Solaar]



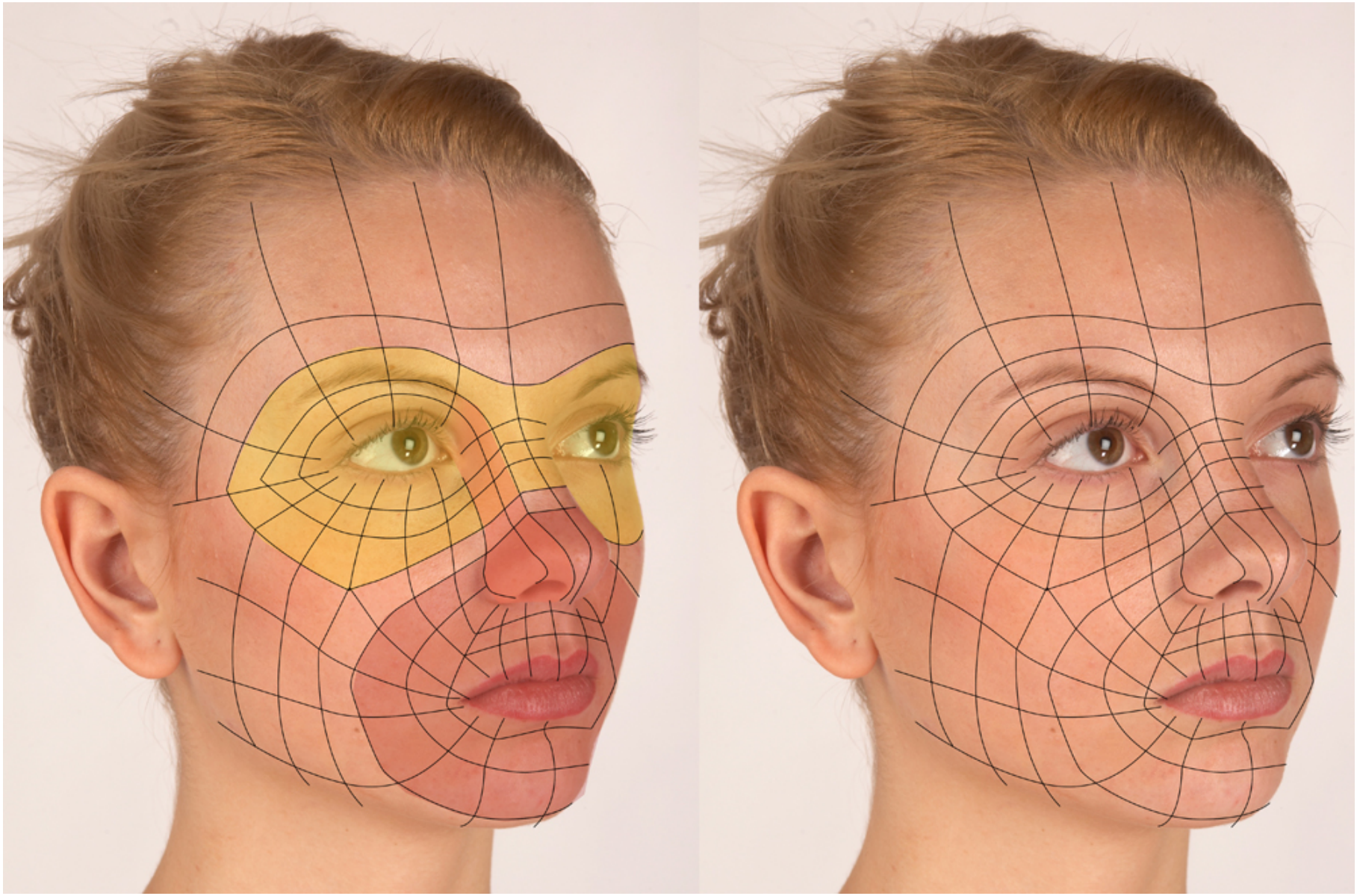
[source unknown]

Quad meshes

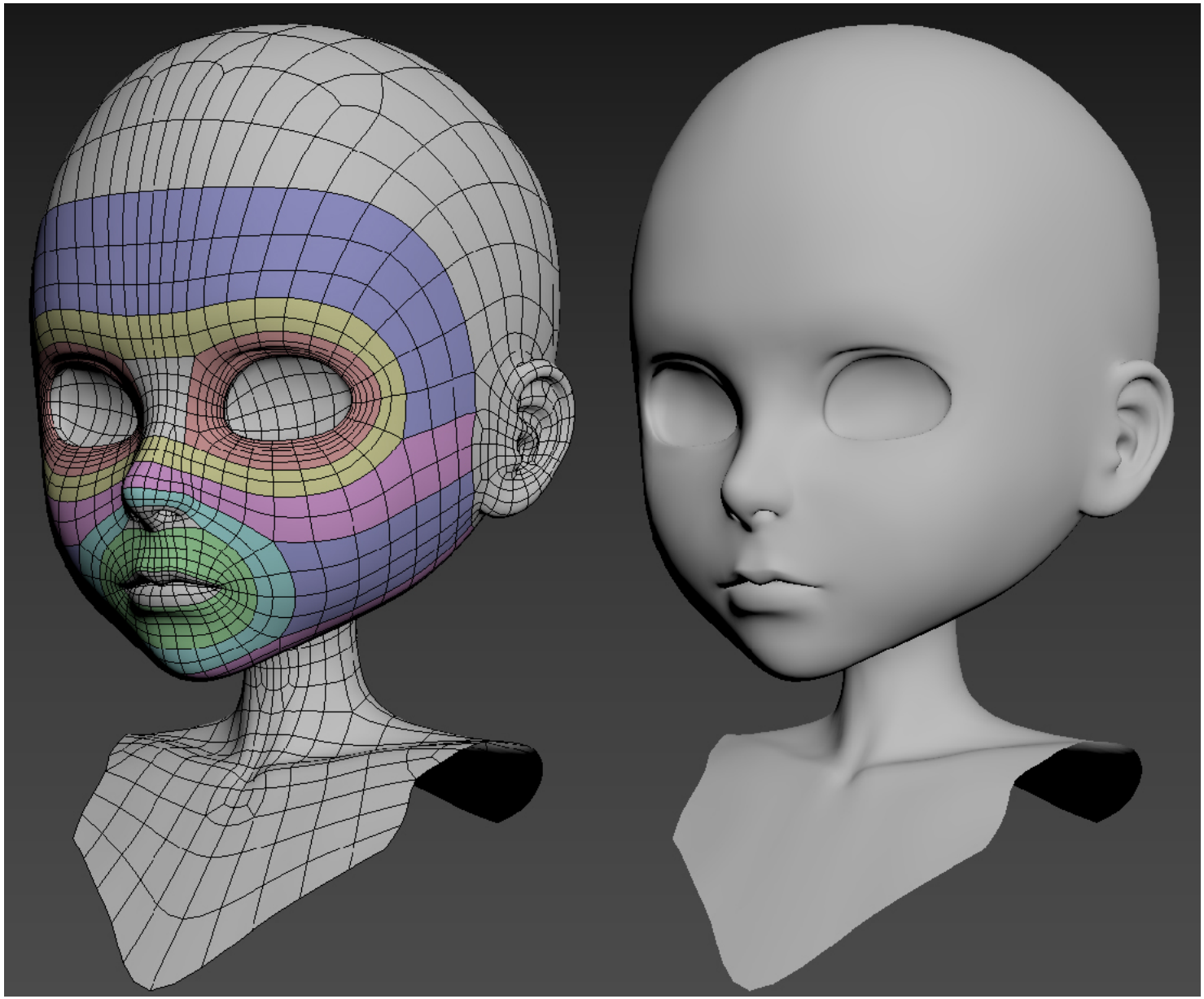
- Most used modeling primitive
  - better selection and edit of edge loops
  - since it captures better "regularities" in surfaces



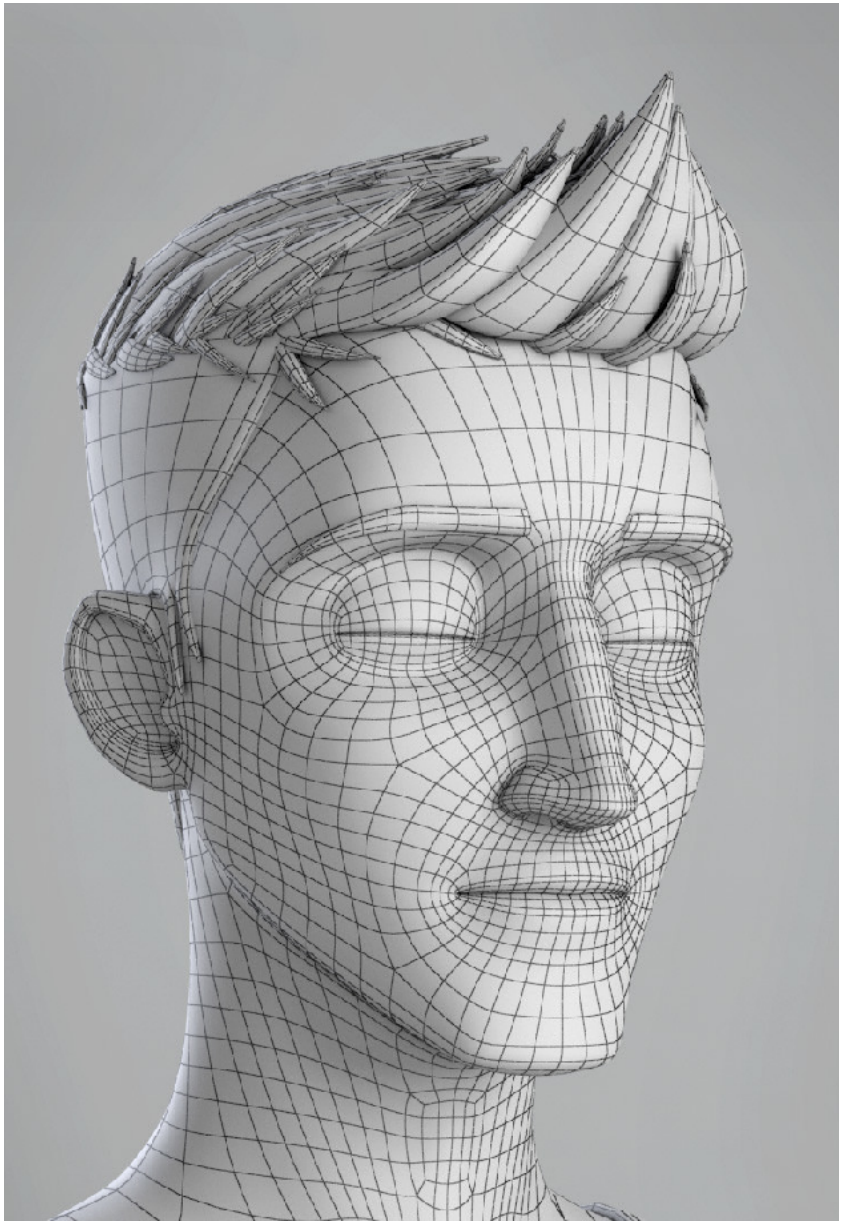
[Panozzo]

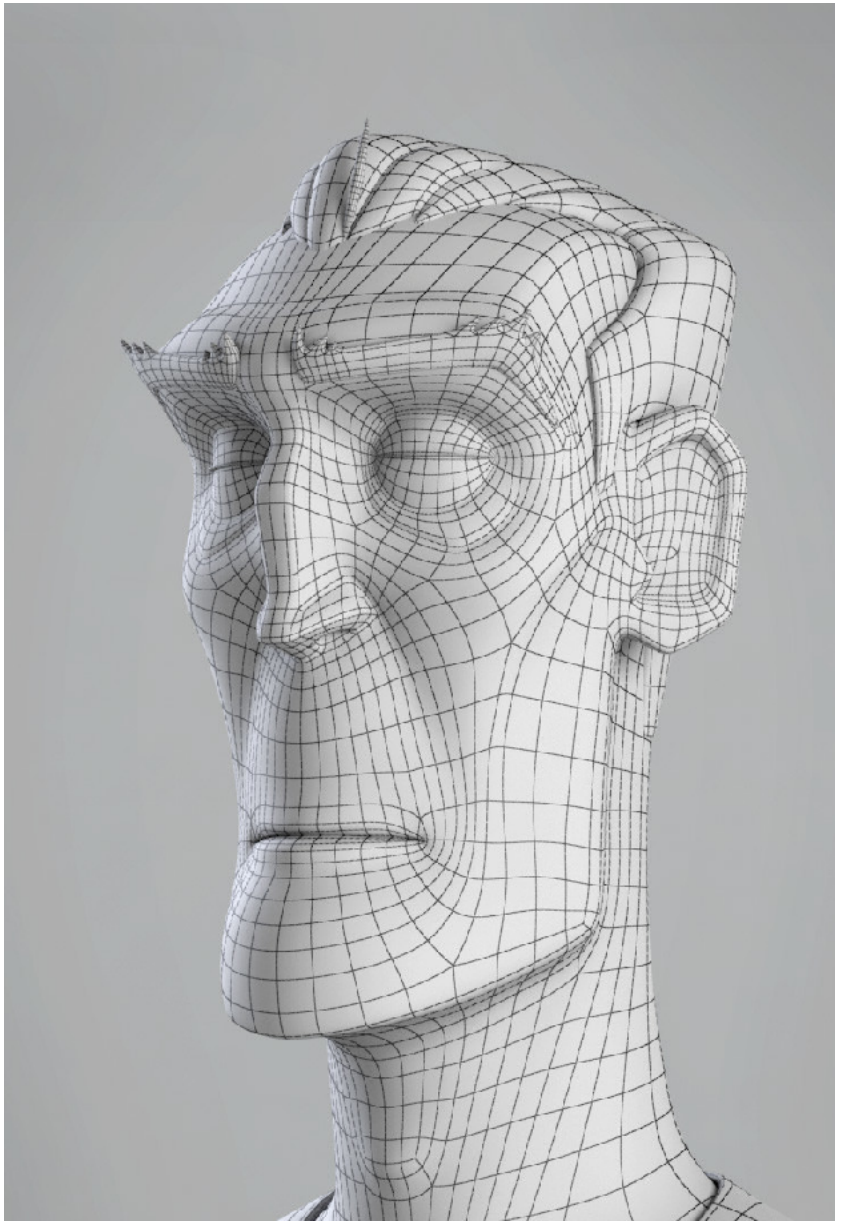


[Athey@deviantart]



[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

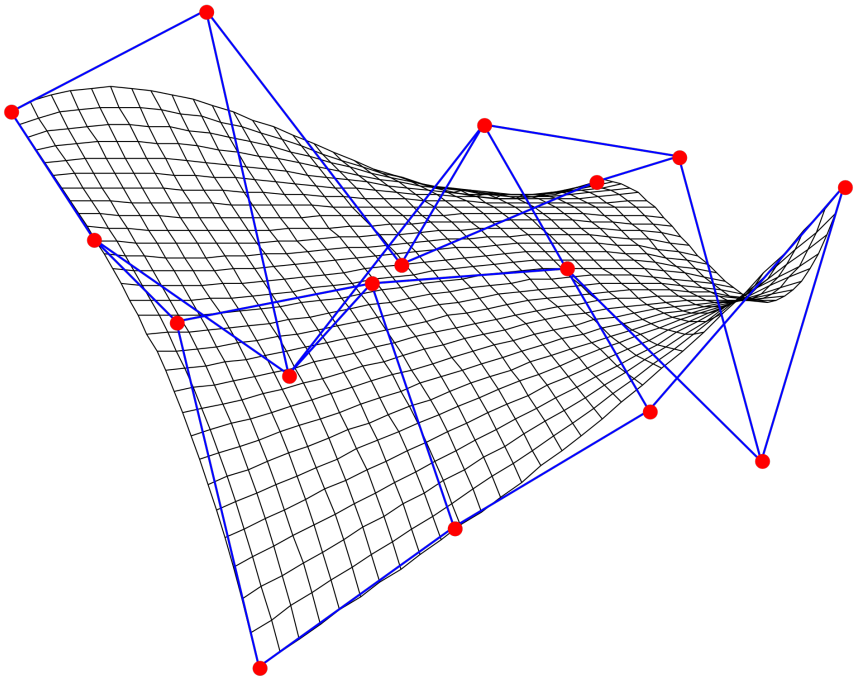
- As for parametric curves, many different parametric surfaces exists
- The most commonly used can be considered extensions of parametric curves called tensor product surfaces
- For example, for a Bezier curve written as

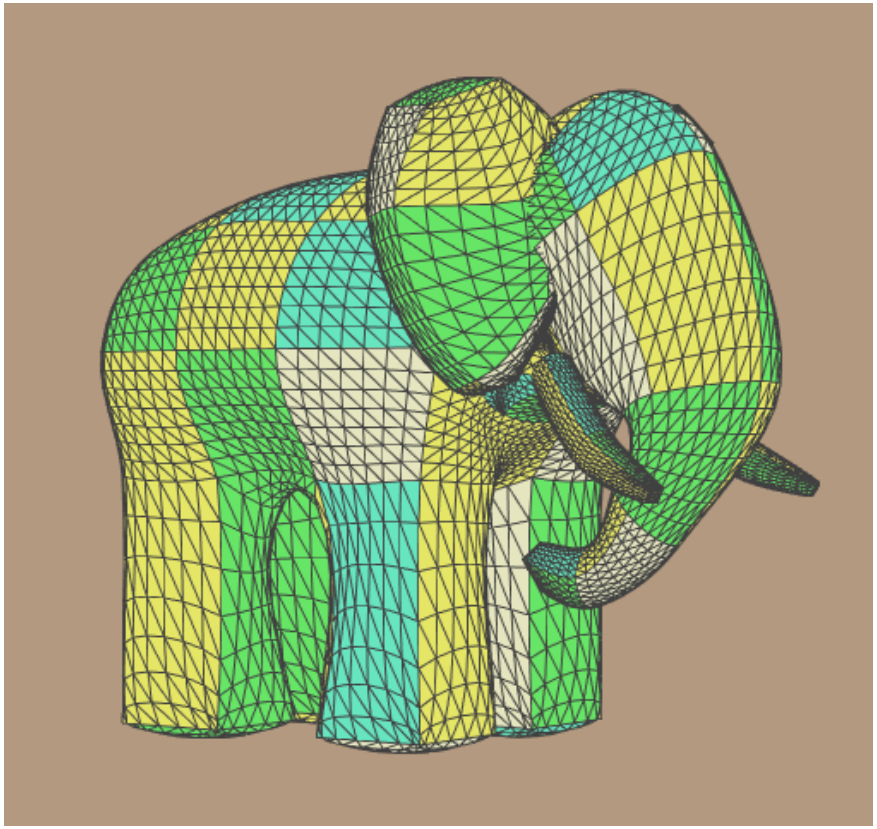
$$\mathbf{p}(u) = \sum_i b_i(u)\mathbf{p}_i$$

- 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





[Wikipedia]

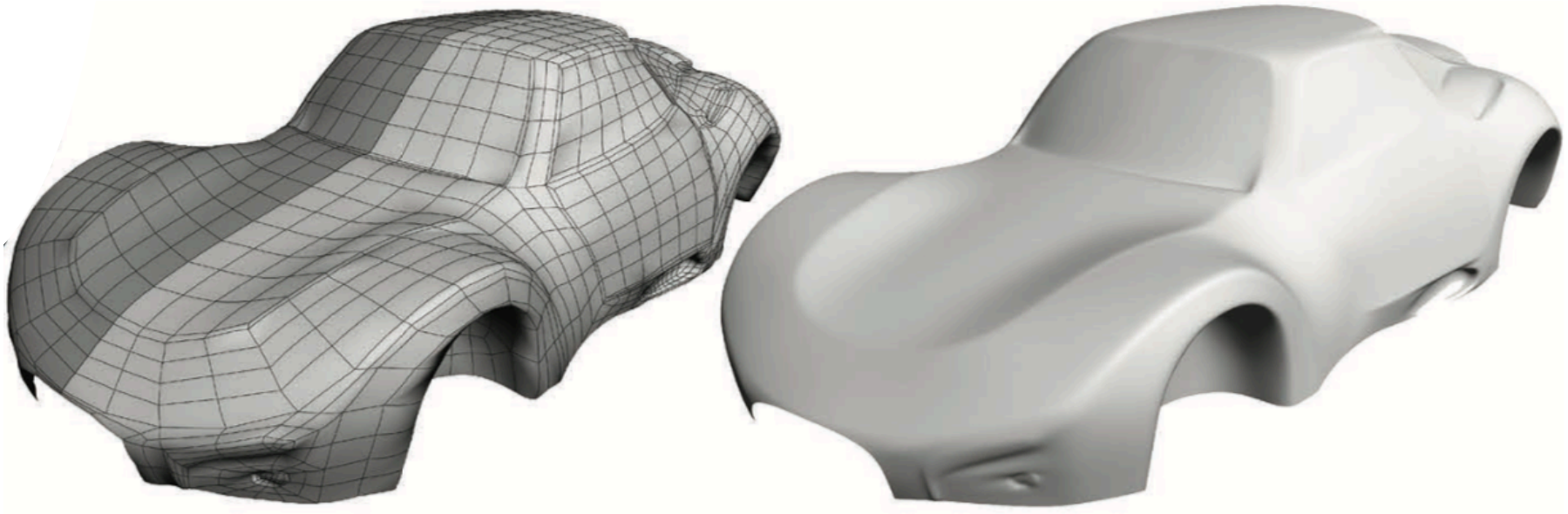
NURBS

- 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

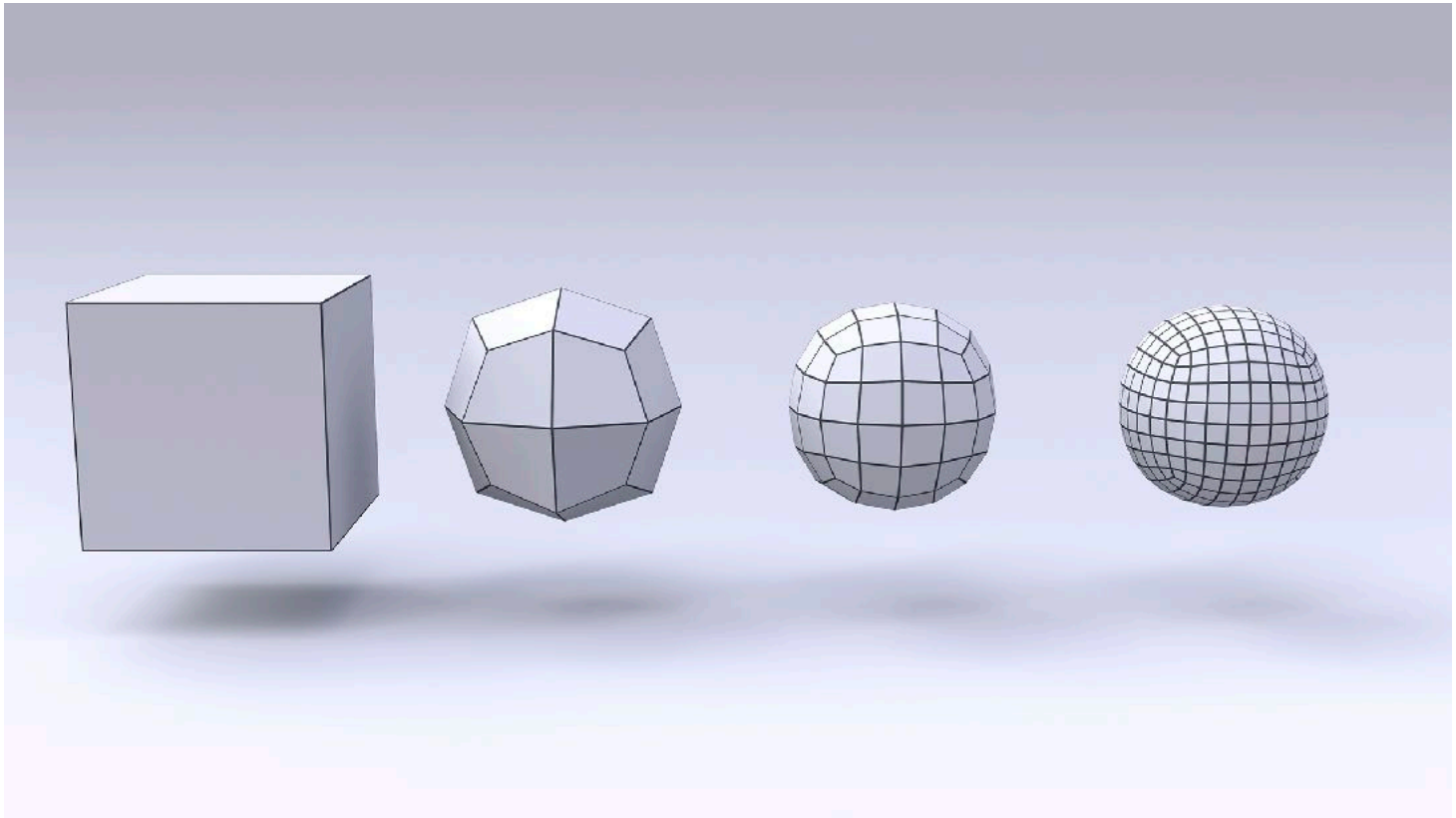
- Represent a smooth surface controlled by a polygon mesh
- Most used method for movies and soon games – works for organic and man-made models



[Panozzo]

Subdivision surfaces

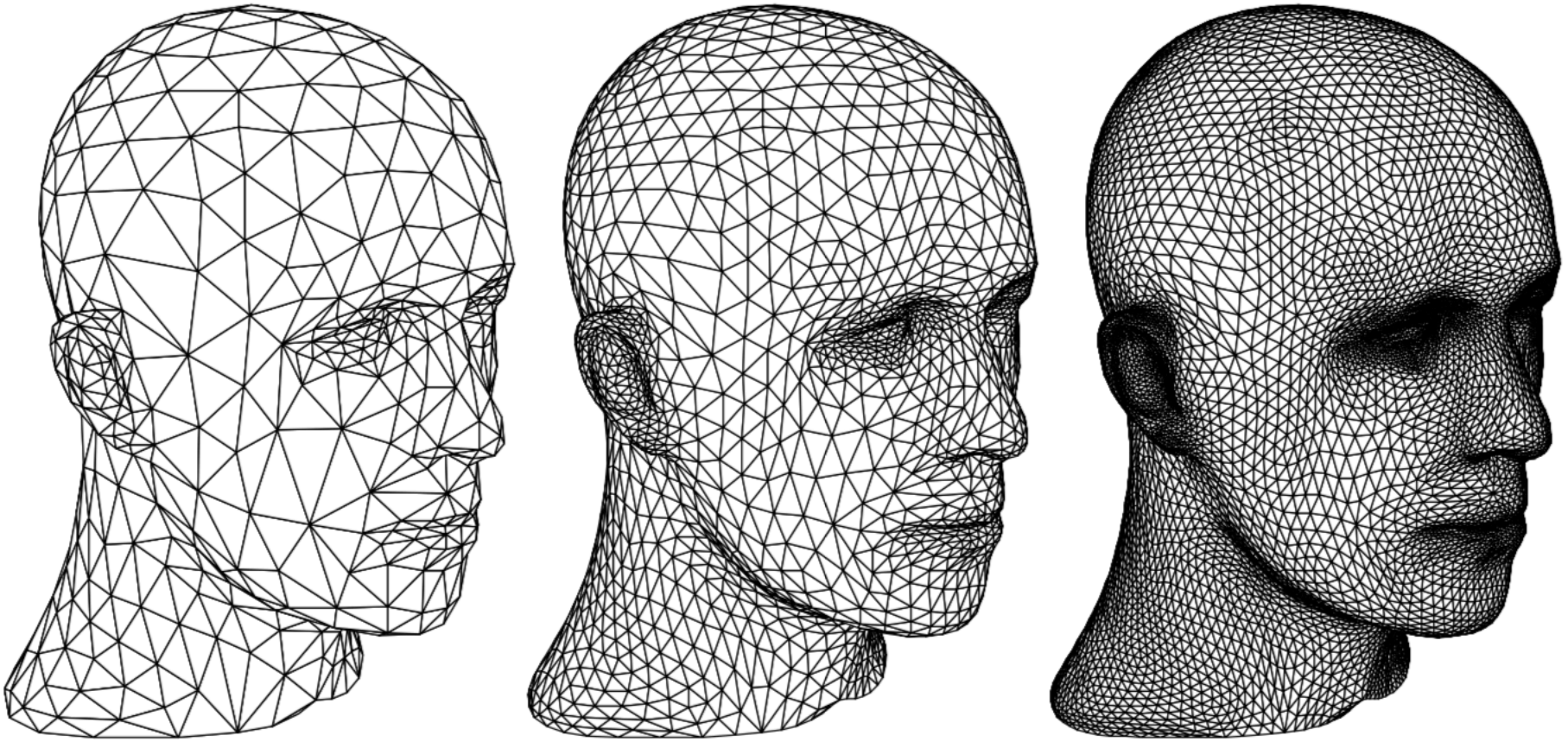
- Defined by recursively apply a refinement rule that add vertices and faces to the mesh and smooths out the vertex positions



[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



[Schröder & Zorin]

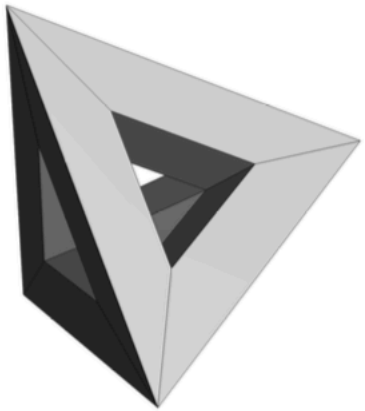
Subdivision surfaces – informal

[CGGorilla/YouTube]

Subdivision surfaces

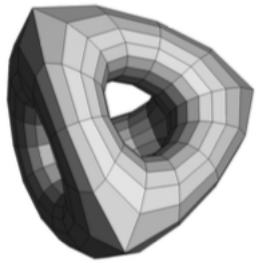
- The subdivision surface is the limit surface *S* obtained by applying the subdivision rule *R* an infinite number of times to a mesh M_0

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



M_0

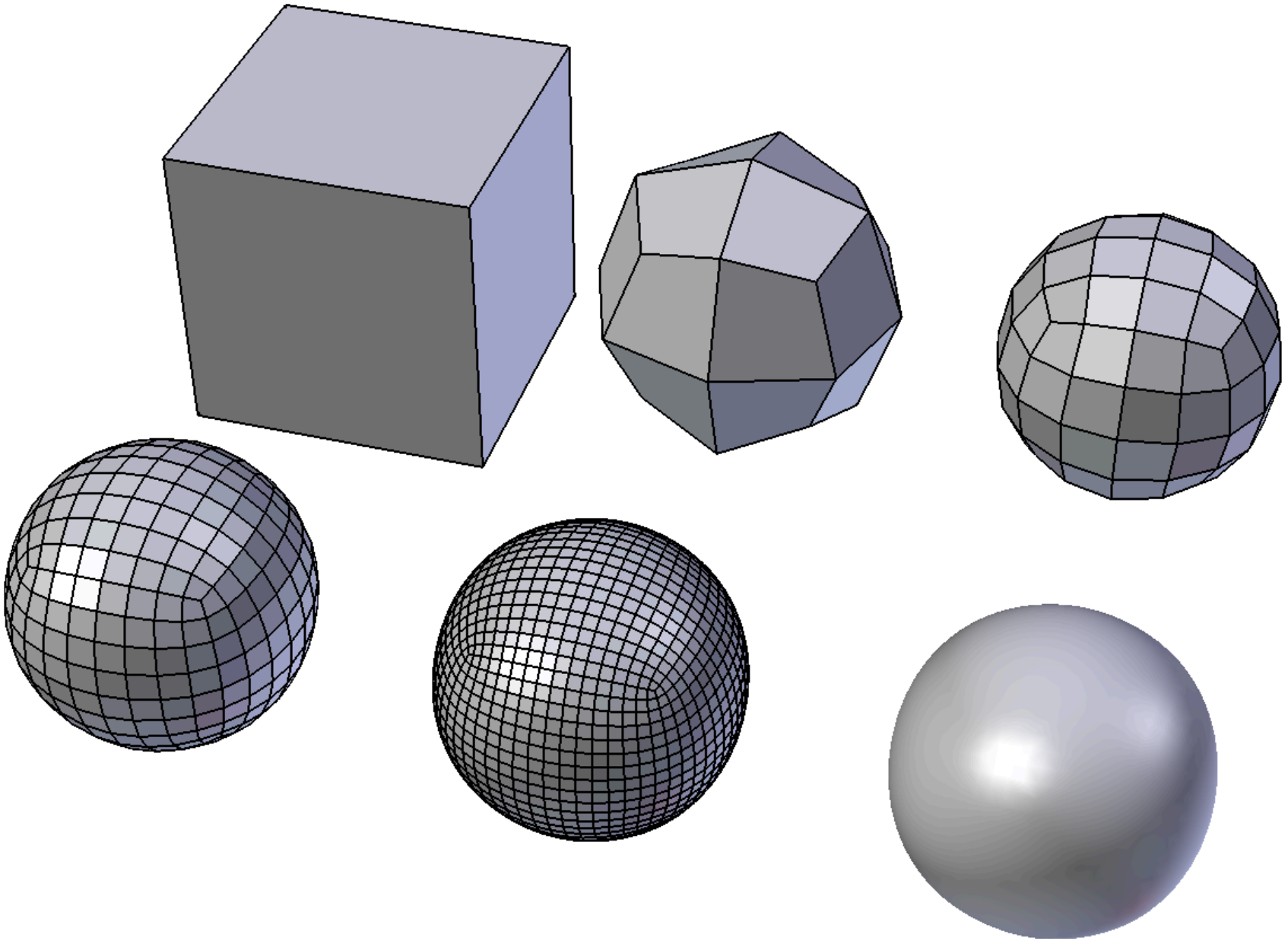
M_1



M_2

$S = M_\infty$

[de Rose et al.]

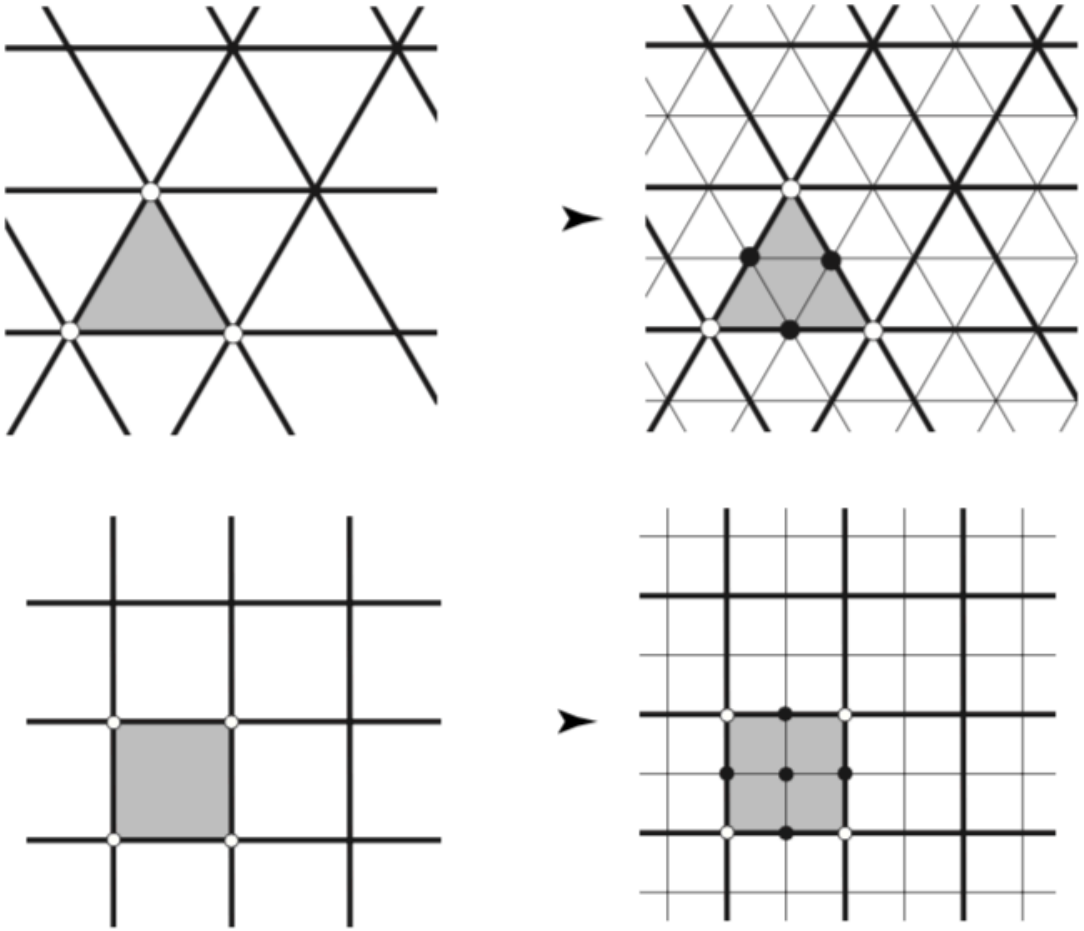


Subdivision rules

- 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

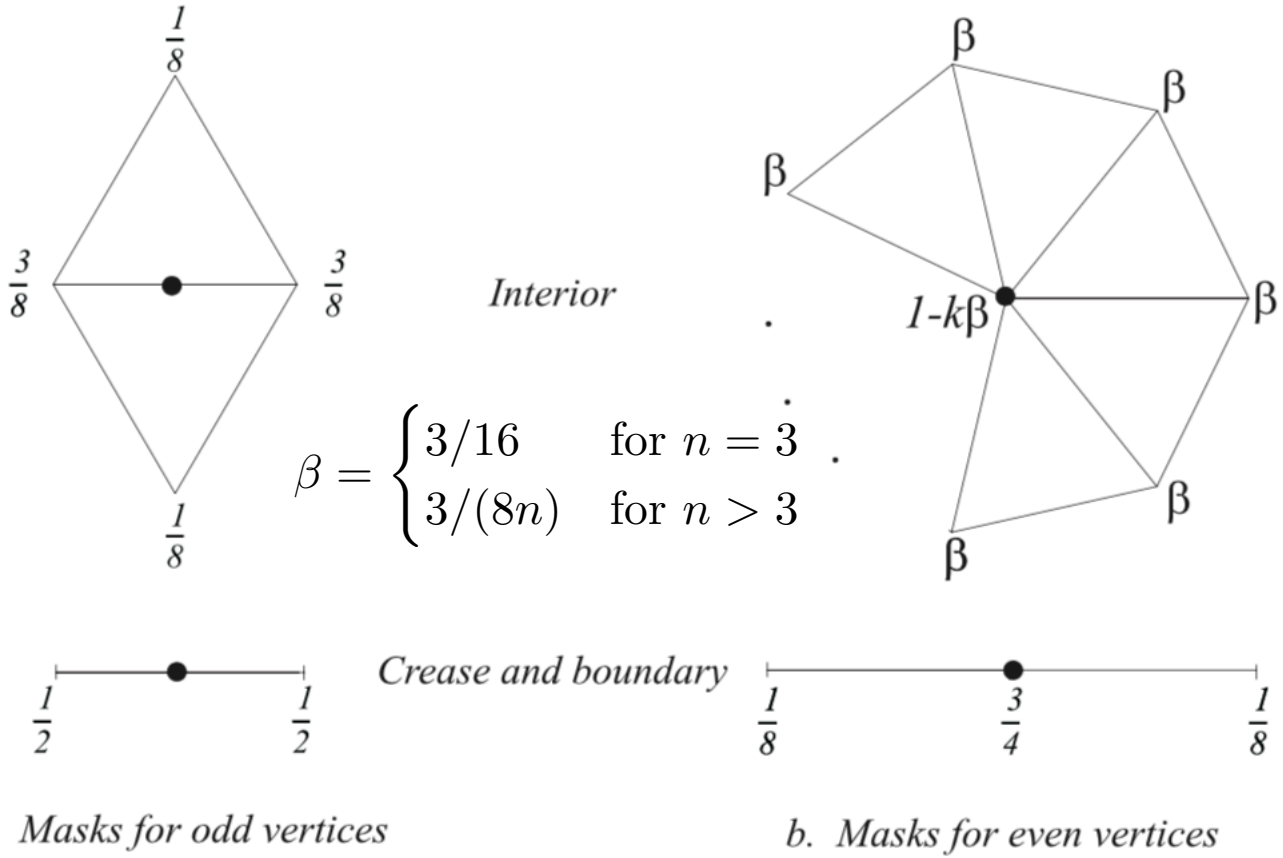
- Edge split methods for triangle (Loop) and quads (Catmull-Clark)



[Schröder & Zorin]

Subdivision smoothing

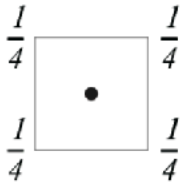
- Loop: rules for existing vertices and vertices on the split edges
  - differentiate between interior and boundaries



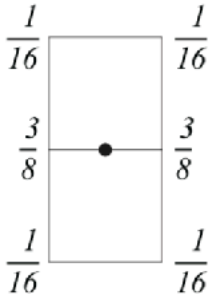
[Schröder & Zorin]

Subdivision smoothing

- Catmull-Clark: rules existing and new vertices

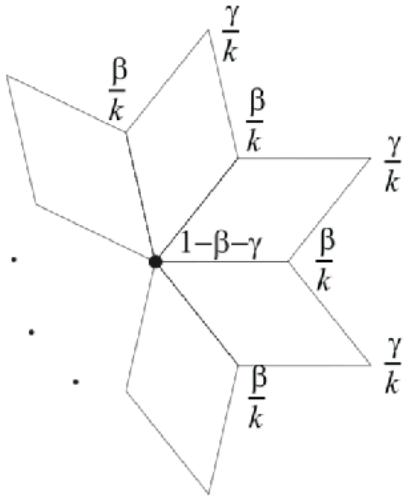


Mask for a face vertex

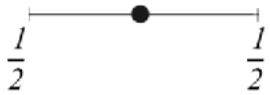


Mask for an edge vertex

Interior

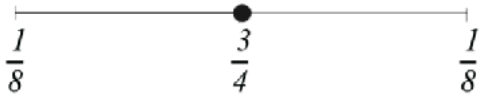


$\beta = 3/(2k) \quad \gamma = 1/(4k)$



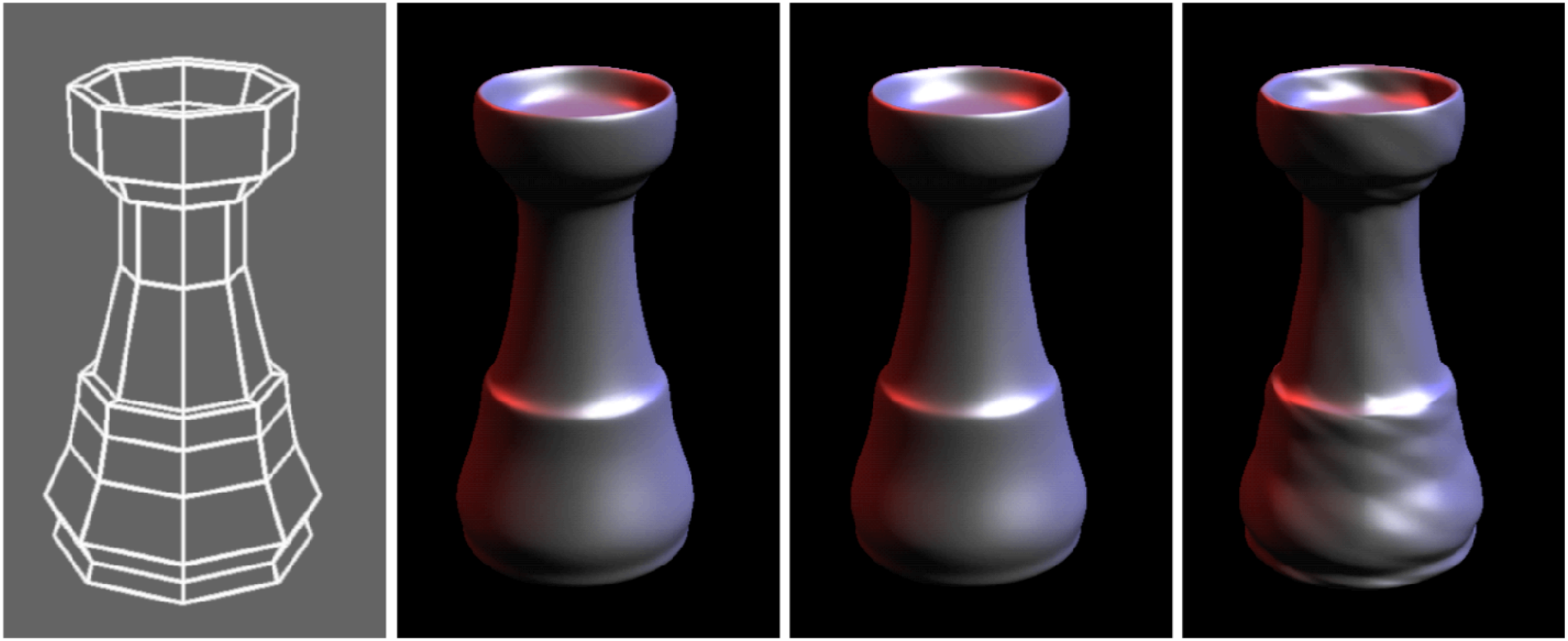
Mask for a boundary odd vertex

Crease and boundary



[Schröder & Zorin]

Model comparison

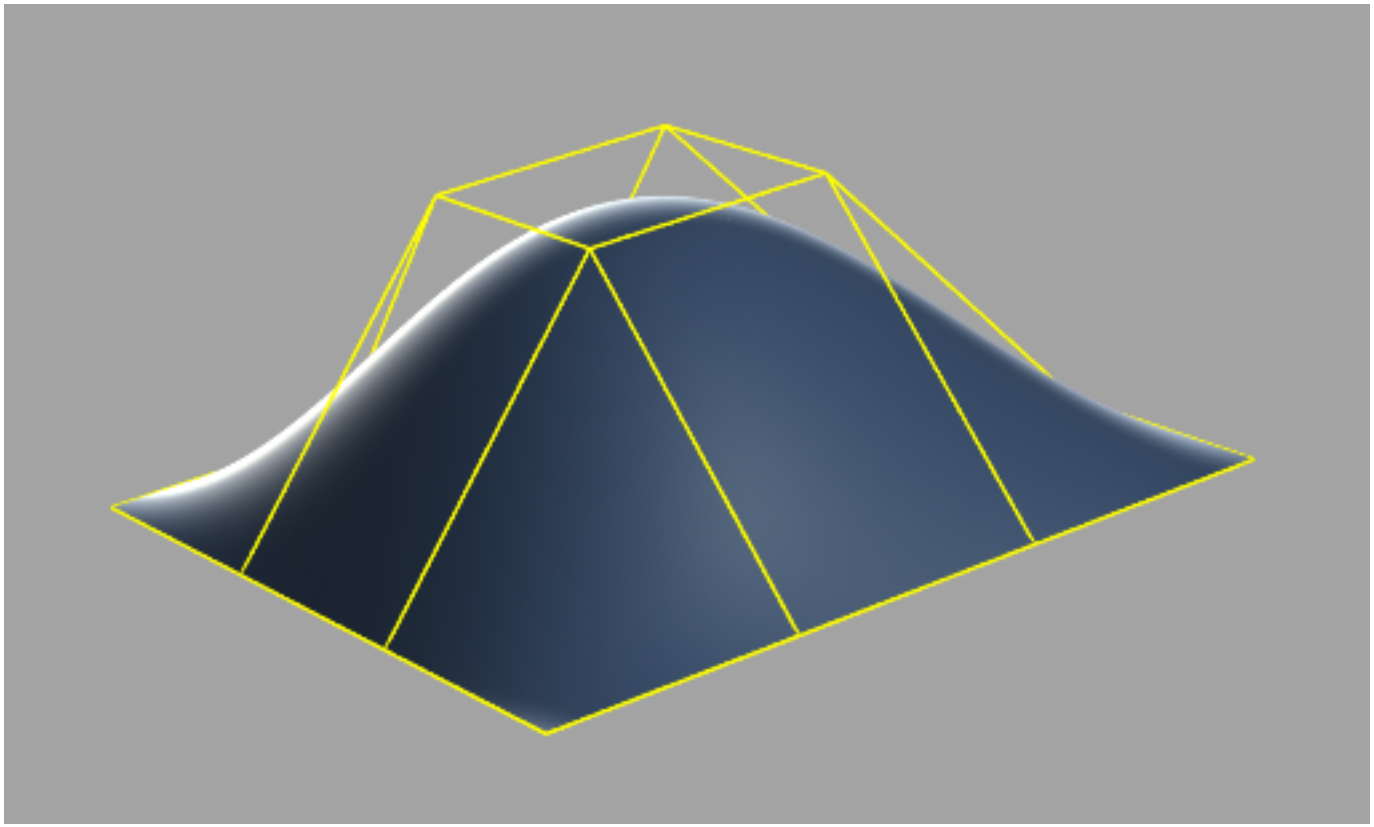


Initial mesh | *Loop* | *Catmull-Clark* | *Catmull-Clark,after triangulation*

[Schröder & Zorin]

Creases

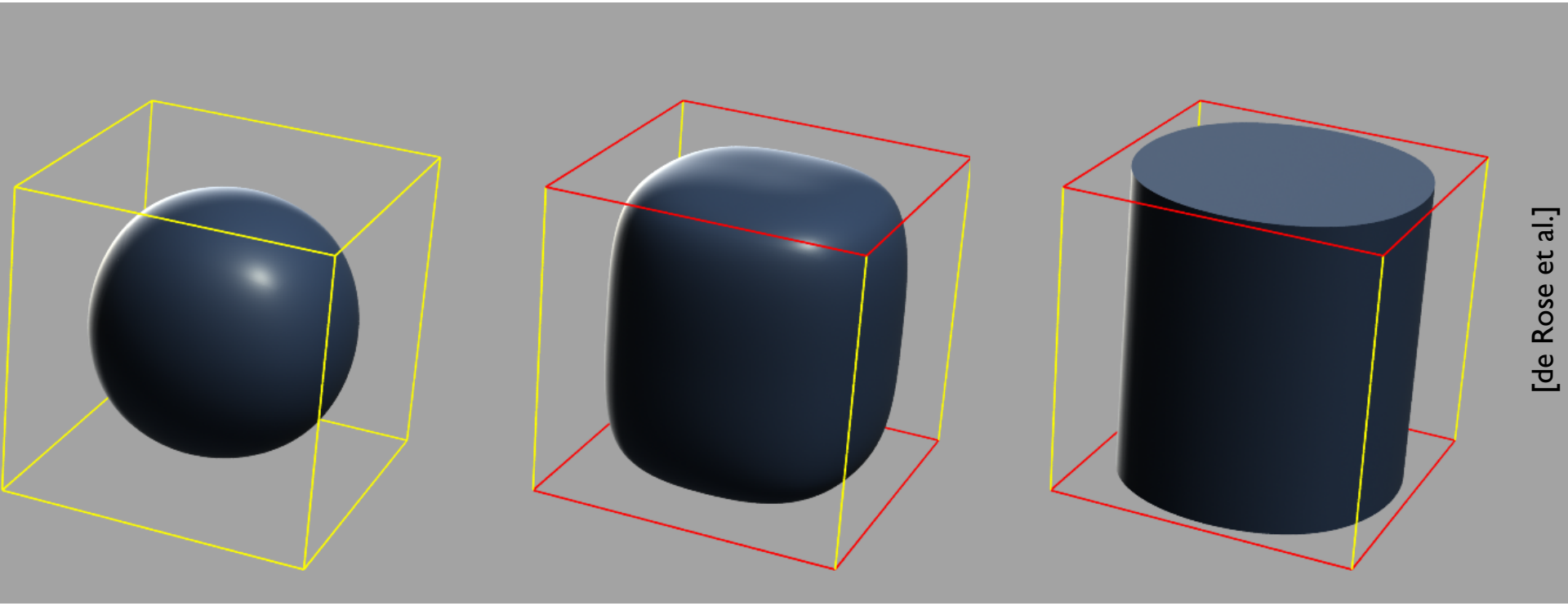
- Control sharpness of edges and vertices



[de Rose et al.]

Creases

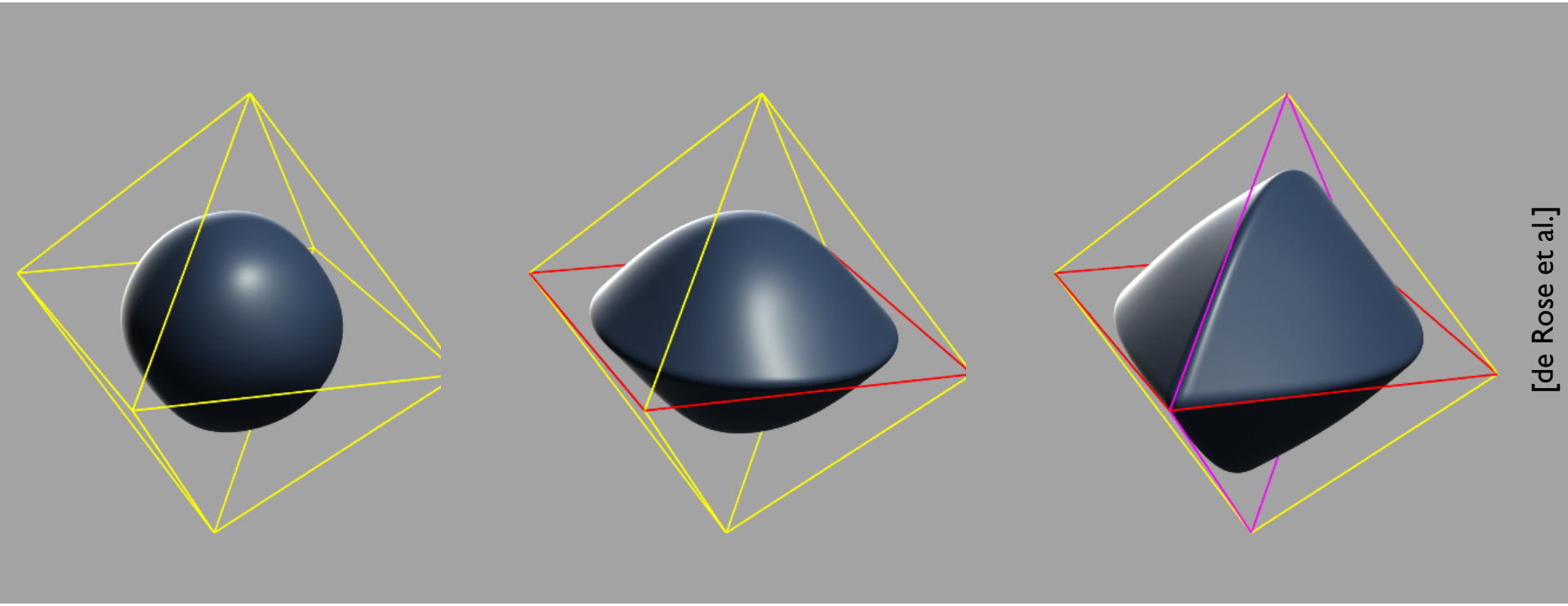
- Control sharpness of edges and vertices



[de Rose et al.]

Creases

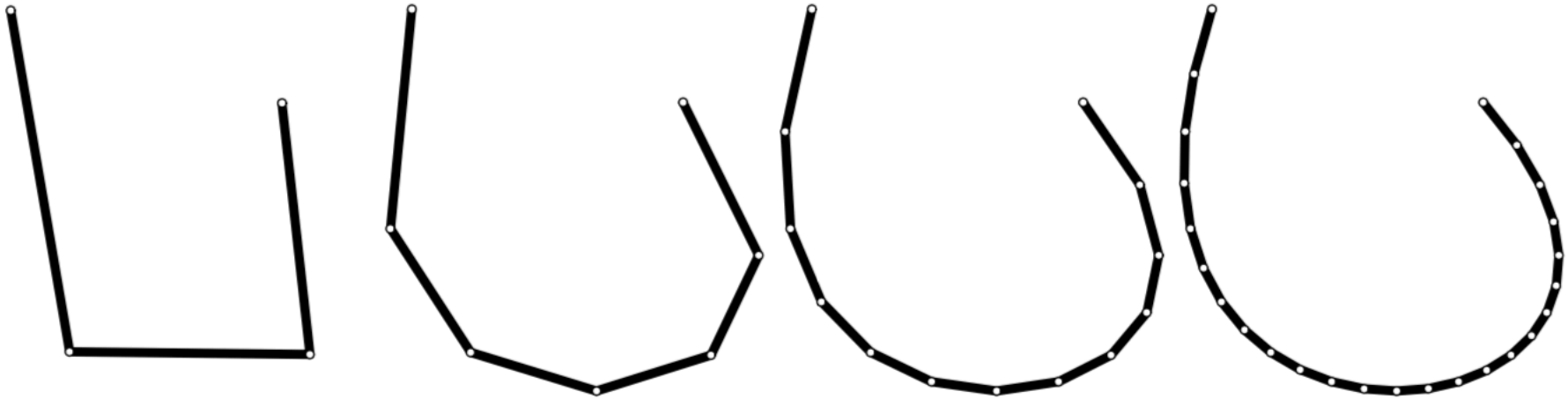
- Control sharpness of edges and vertices



[de Rose et al.]

Subdivision curves

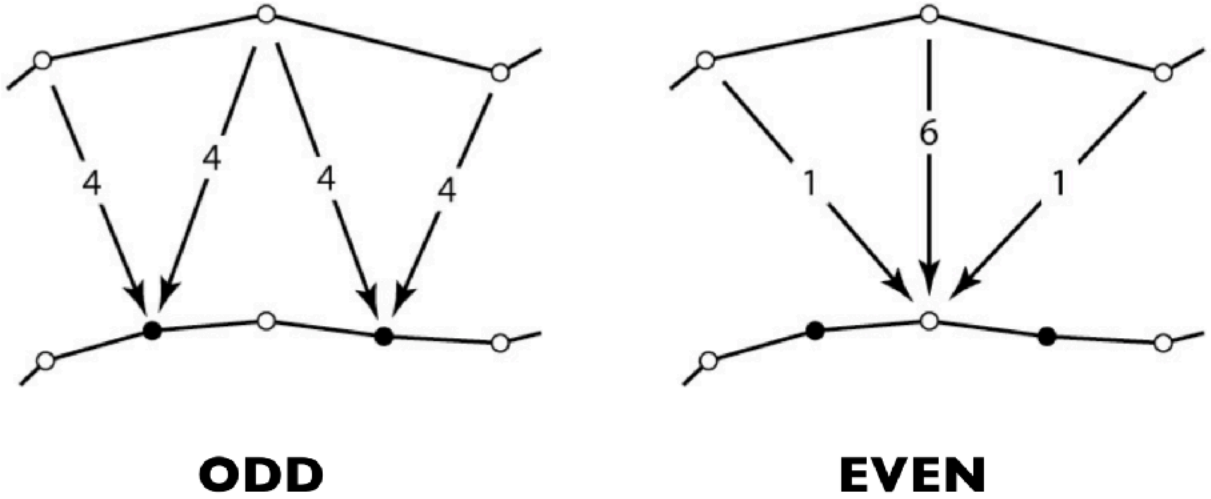
- Main idea also works for curves
  - but less in practice since polynomials are better
- Rule: recursively split each segment in half and move vertex
- Define curves from subdivision rules, instead of polynomials
  - for B-splines we have both definitions and they are equivalent

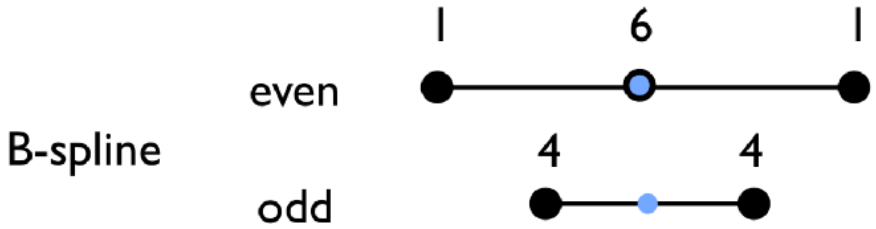


[Schröder & Zorin]

Subdivision rule for B-splines

- New vertices are linear combination of old vertices



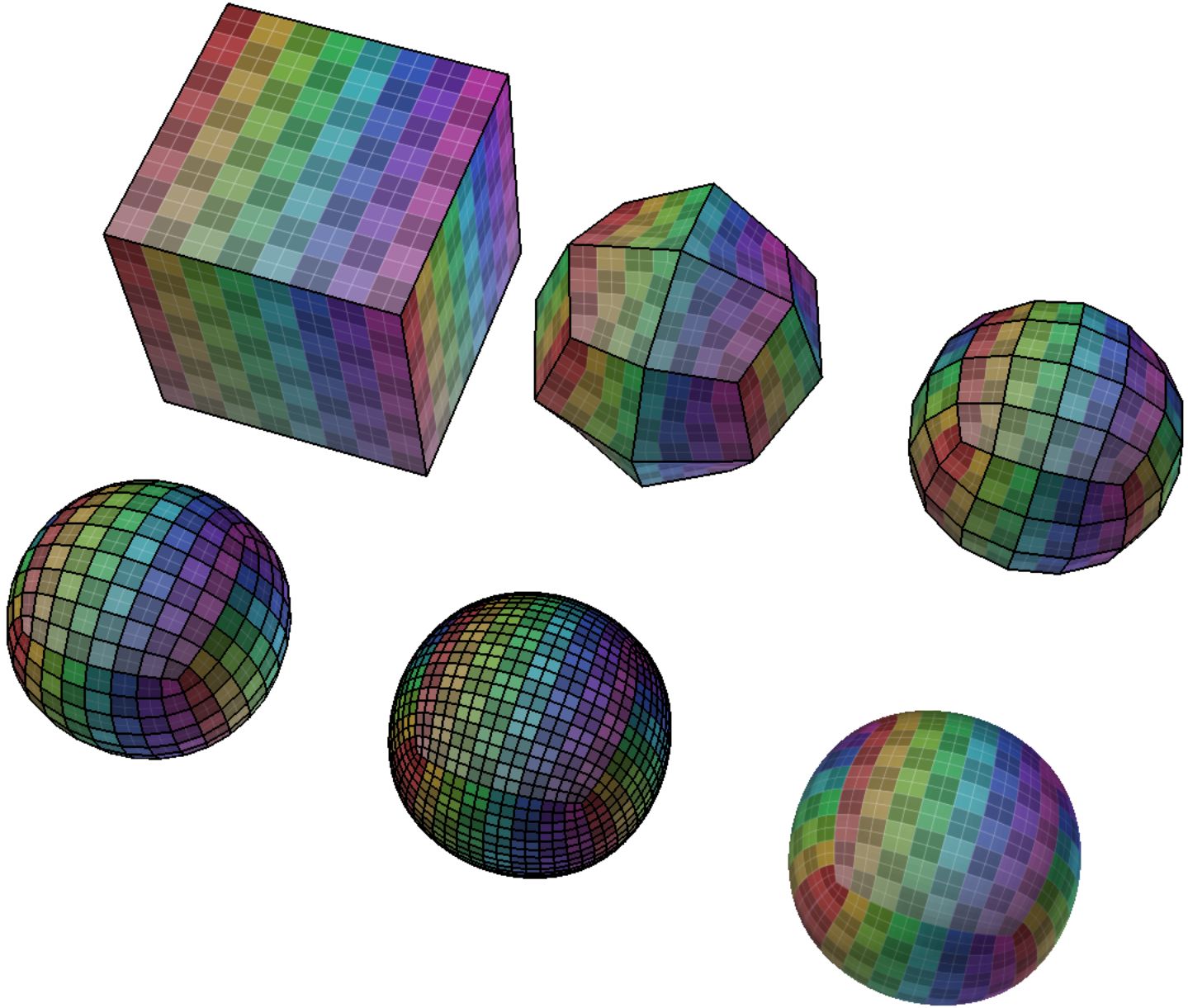


Subdivision scalability

- 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 tesselate 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

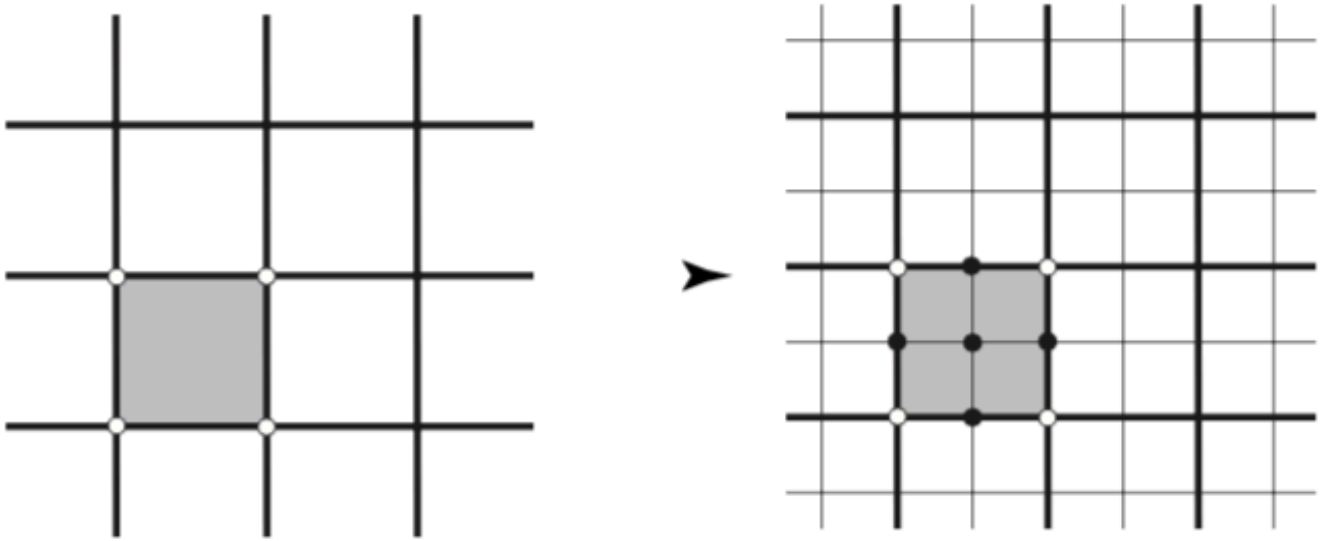
- Classic implementation requires specialized data structures
  - seldomly implemented
  - often results do not match between implementations
  - considerable practical problems for exchanging data
  - special handling for creases
- Classic implementation can be done optimal though
  - view driven level of details
  - non-binary subdivision
  - accelerated raytracing without tesselating 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 advatange: 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 subivision
  - main concern: ensure that edge vertices are shared after split



[Schröder & Zorin]

Implementing subdivs

- Step 1: 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 1: 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 alway insert (min, max) index
  - hash values: index to the new edge

Implementing subdivs

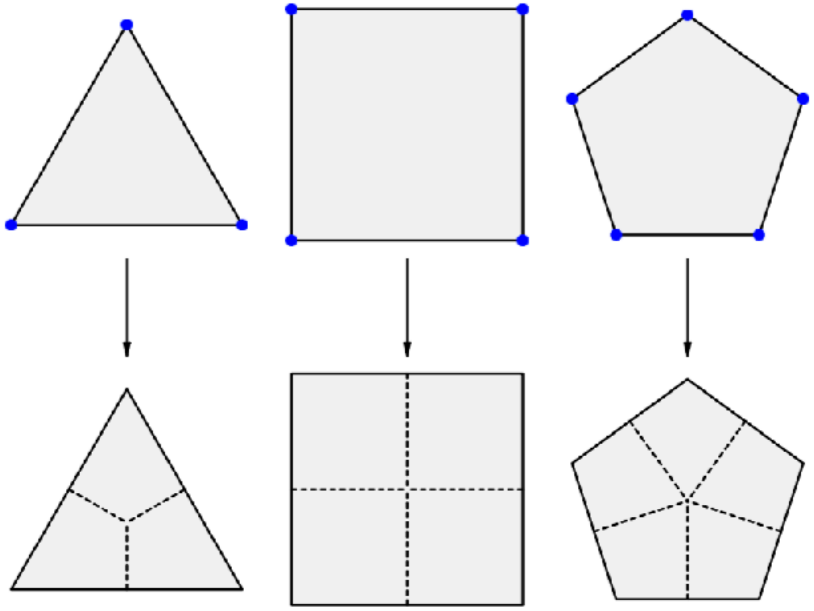
- Step 1: linear subdivision
  - create four faces *F* for each quad *f* in the original
  - use the hash table for the edge indices

$$
\begin{aligned}
F'_{\{f,0\}} &= \{\mathbf{p}'_{f_0}, \mathbf{p}'_{e_{f_0 \to f_1}}, \mathbf{p}'_f, \mathbf{p}'_{e_{f_3 \to f_0}}\} \\
F'_{\{f,1\}} &= \{\mathbf{p}'_{f_1}, \mathbf{p}'_{e_{f_1 \to f_2}}, \mathbf{p}'_f, \mathbf{p}'_{e_{f_0 \to f_1}}\} \\
F'_{\{f,2\}} &= \{\mathbf{p}'_{f_2}, \mathbf{p}'_{e_{f_2 \to f_3}}, \mathbf{p}'_f, \mathbf{p}'_{e_{f_1 \to f_2}}\} \\
F'_{\{f,3\}} &= \{\mathbf{p}'_{f_3}, \mathbf{p}'_{e_{f_3 \to f_0}}, \mathbf{p}'_f, \mathbf{p}'_{e_{f_2 \to f_3}}\}
\end{aligned}
$$

- Works also for non-quad faces

Implementing subdivs

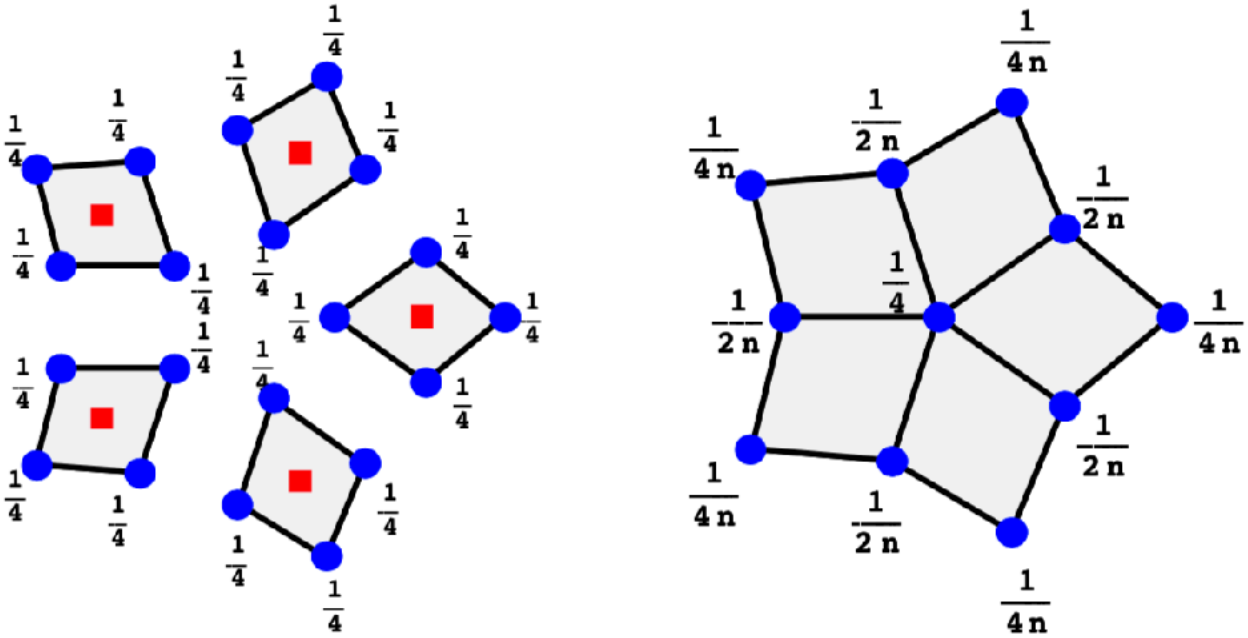
- Step 1: linear subdivision
  - works also for non-quad faces as input
  - after one pass all faces are quads



[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



[Warren & Shaefer]

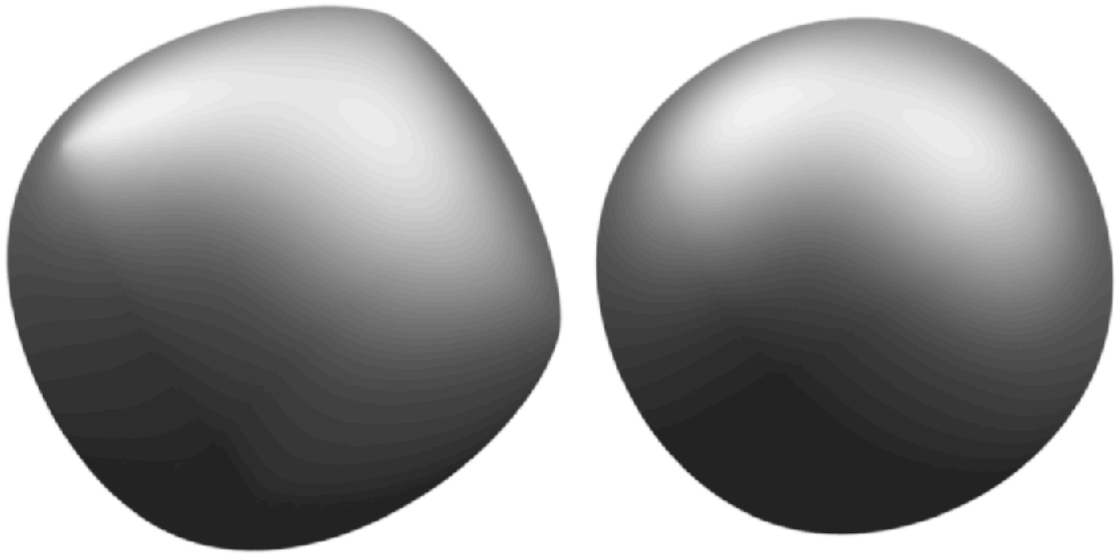
Implementing subdivs

- Step 2: averaging pass
  - smooth vertices based on their neighbors
  - update vertex position as the average of the centroids of the faces touching it
  - 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
  - boundaries 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
  - Takes as input an array of quads, an array of vertex coordinates and a flag to indicate whether the boundary should move

```
pair<vector<vec4i>, vector<vec3f>>
 subdivide_catmullclark(vector<vec4i> quads,
            vector<vec3f> verts, bool lock_boundary) {
  <initialize edges>
  <create vertices> // phase 1
  <create faces>
  <setup boundary>
  <averaging>       // phase 2
  <correction>      // phase 3
  return {tquads, tverts};
}
```


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

- We can build an edge map by storing each edge with indices sorted
- Note: the implementation here is not efficient

```
struct edge_map {
  hash_map<vec2i, int> index  = {}; // edge indices
  vector<vec2i>        edges  = {}; // list of edges
  vector<int>          nfaces = {}; // list of boundaries
};

edge_map make_edge_map(vector<vec4i> quads) {
  auto emap = edge_map{};
  for (auto q : quads) {
    insert_edge(emap, {q.x, q.y});
    insert_edge(emap, {q.y, q.z});
    if (q.z != q.w) insert_edge(emap, {q.z, q.w});
    insert_edge(emap, {q.w, q.x});
  }
  return emap;
}
```


Implementing subdivs

- We can build an edge map by storing each edge with indices sorted
- Note: the implementation here is not efficient

```
void insert_edge(edge_map& emap, vec2i edge) {
  auto key = vec2i{min(edge), max(edge)};
  if (emap.indices.contains(key)) {
    emap.nfaces[emap.index[key]] += 1;
  } else {
    auto idx = (int)emap.edges.size();
    emap.index[key] = idx;
    emap.edges.push_back(key);
    emap.nfaces.push_back(1);
 }
}
```


Implementing subdivs

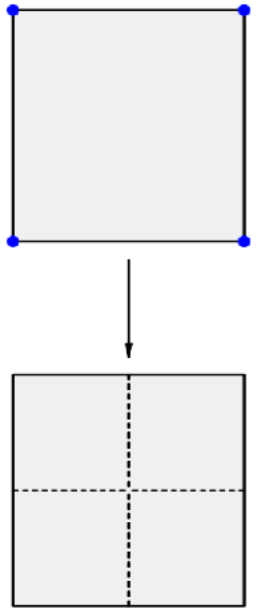
- Then we first initialize the array of vertices and make a vertex for each vertex, edge and face in the original mesh

```
<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.z]) / 3);
}
```


Implementing subdivs

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

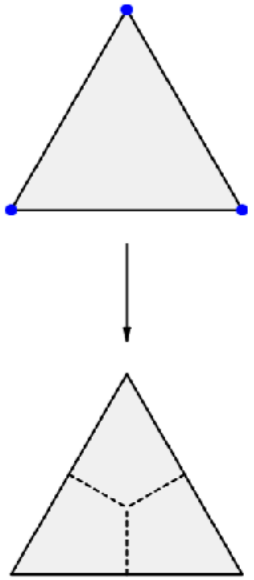
```
<create faces> {
  auto tquads = vector<vec4i>();
  for (auto [i, q] : enumerate(quads)) {
    if(q.z != q.w) {
      tquads.push_back({q.x,nv+emap[q.x,q.y],
            nv+ne+i, nv+emap[q.w, q.x]});
      tquads.push_back({q.y,nv+emap[q.y,q.z],
            nv+ne+i, nv+emap[q.x, q.y]});
      tquads.push_back({q.z, 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 crease 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 acount = vector<int>(tvert.size(), 0);
  for (auto p : tcrease_verts) {
    if (tvert_val[p] != 0) continue;
    avert[p] += tverts[p]; acount[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; acount[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; acount[vid] += 1;
   }
 }
 for (auto i : range(tverts))
   avert[i] /= (float)acount[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)acount[i]);
  }
  tverts = averts;
}
```

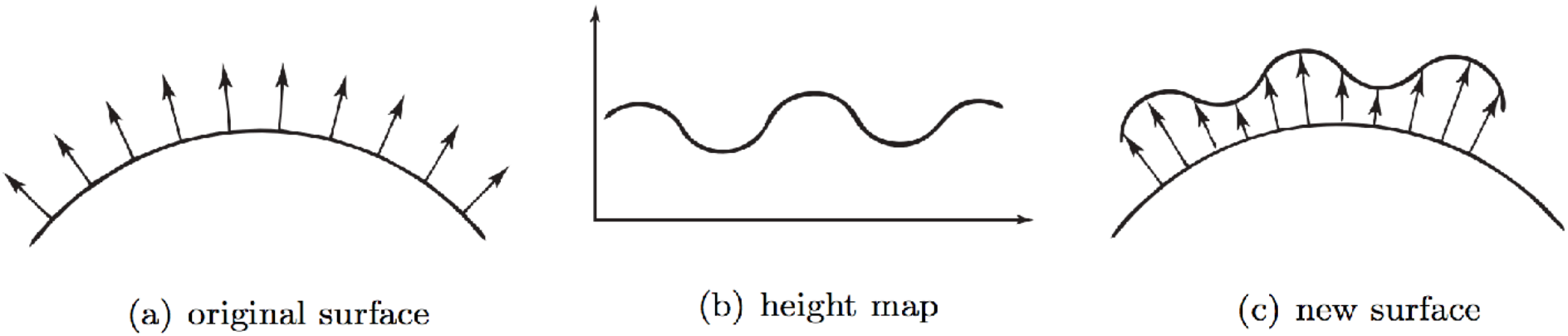

Displacements and Bumps

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

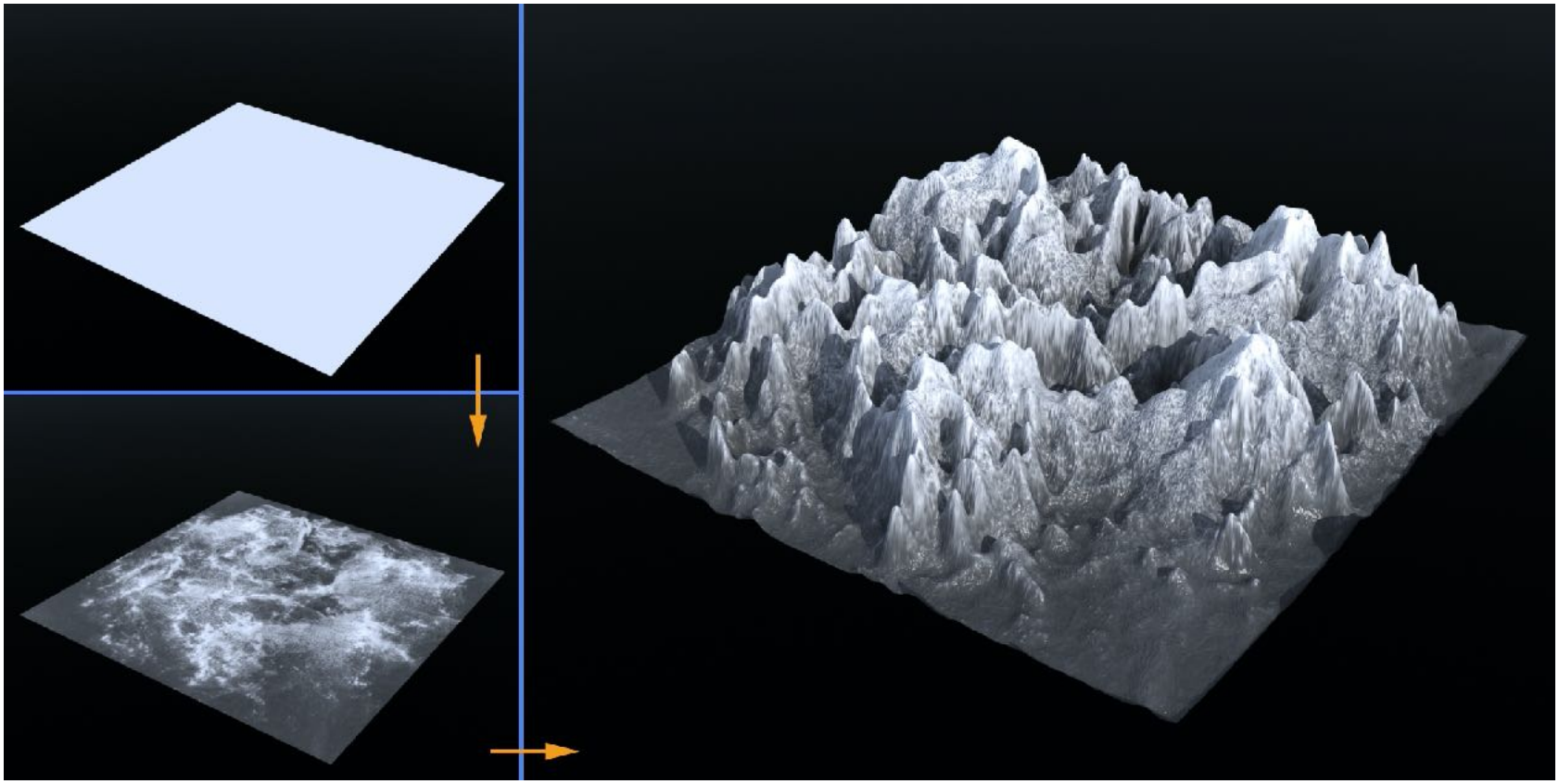
- Displacement mapping: use a texture that defines the displacement along the normal direction at each surface location



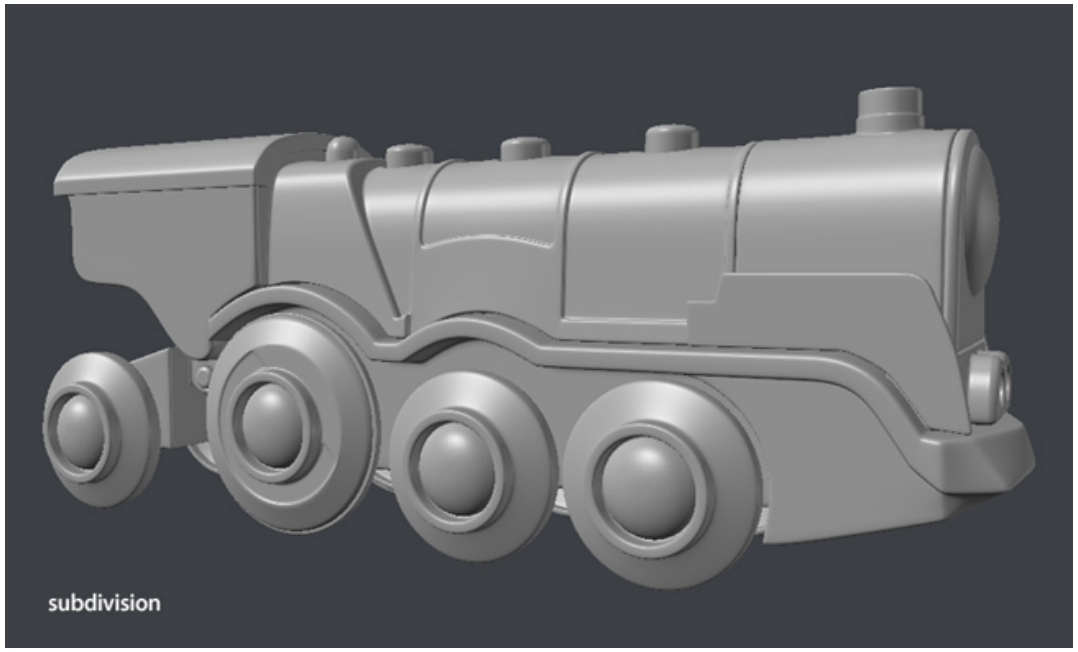
(a) original surface (b) height map (c) new surface

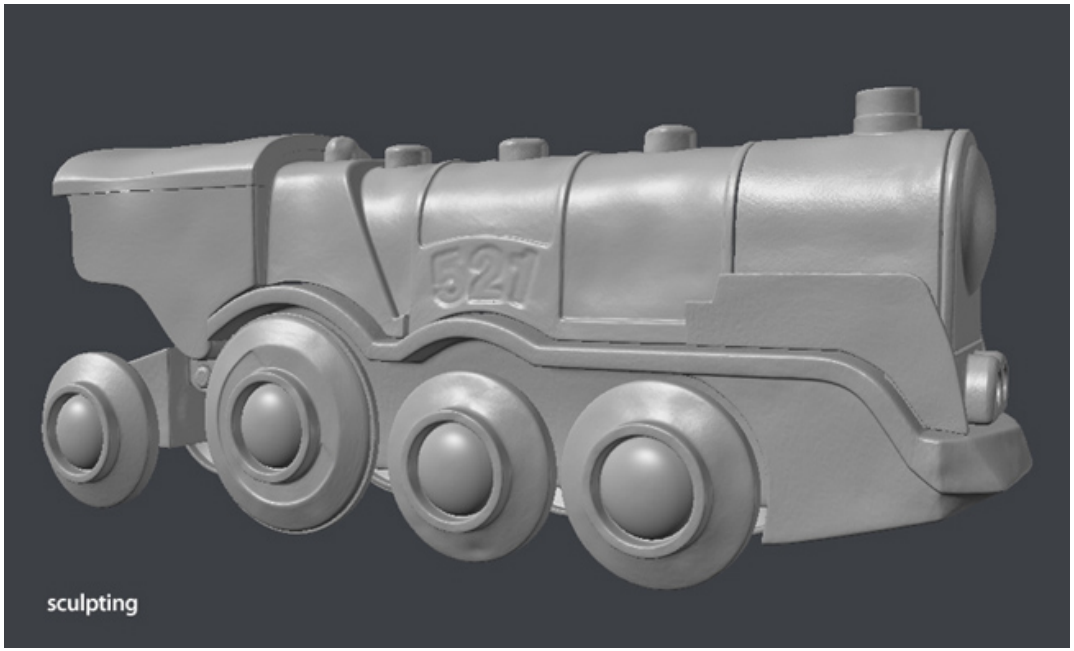
Displacement mapping

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



[WikimediaCommons]

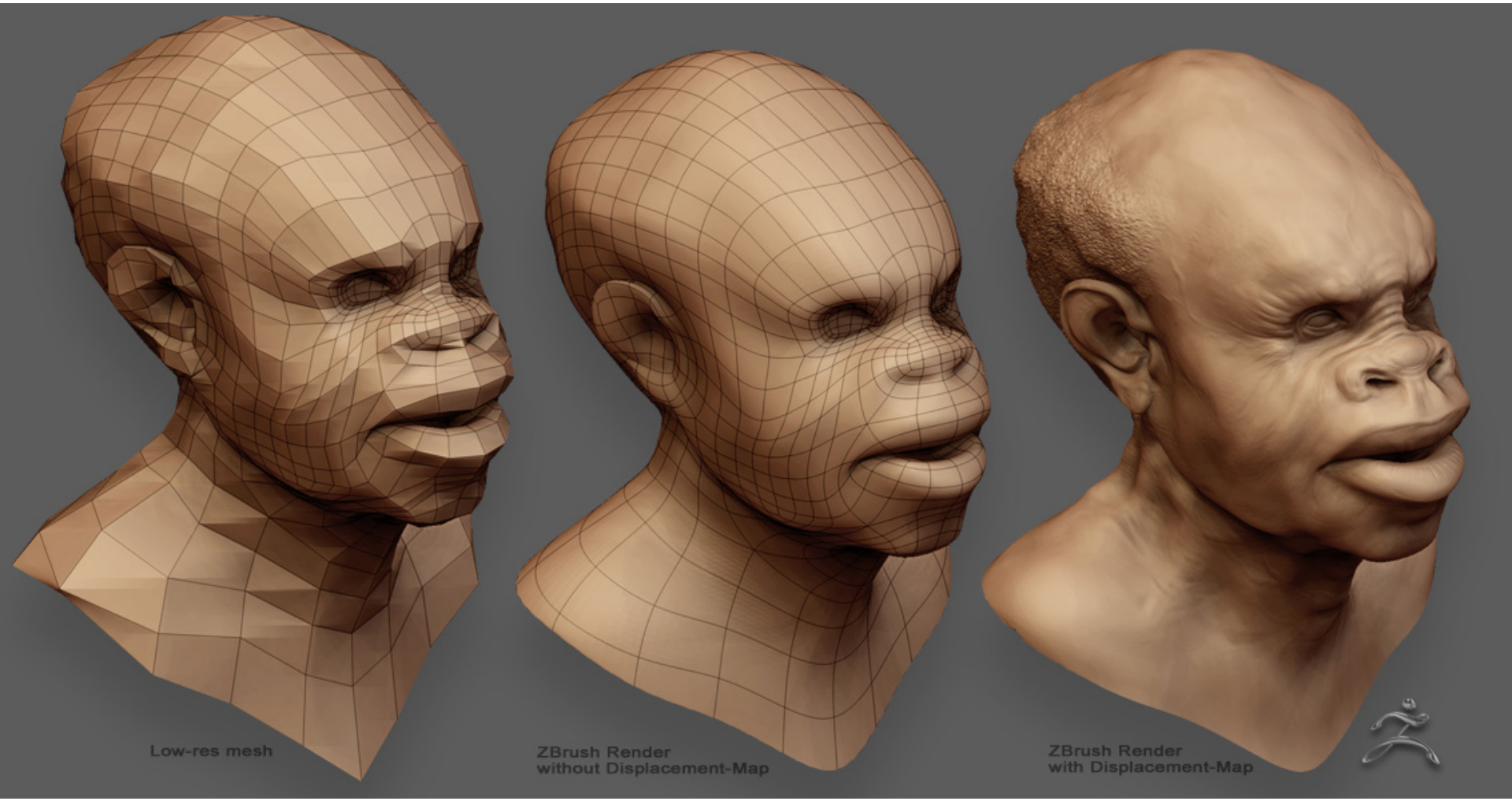




[Pawel Filip]


521
fryrender
physically-based render engine
©2007 Pawel Filip

Displacement mapping



[Pixologic]

Displacement mapping

Displacement mapping

[Yining Karl]

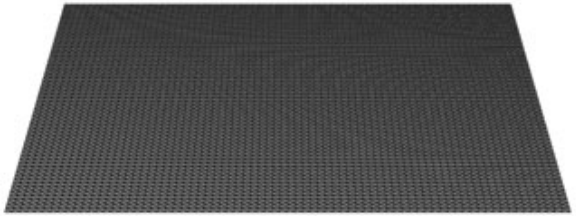
Displacement mapping

- Write vertex position using a parametrization to index the texture

$$\mathbf{p}^d(u,v) = \mathbf{p}(u,v) + h(u,v)\mathbf{n}(u,v)$$

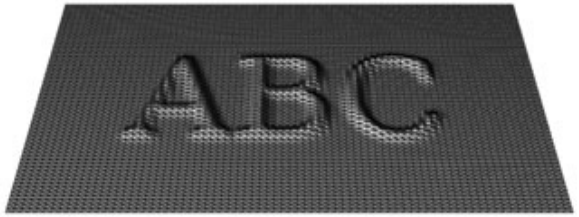
- To render a displaced triangle mesh, we tesselate it at the appropriate resolution and displace each vertex by the amount specified in the texture along the vertex normal

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



ORIGINAL MESH

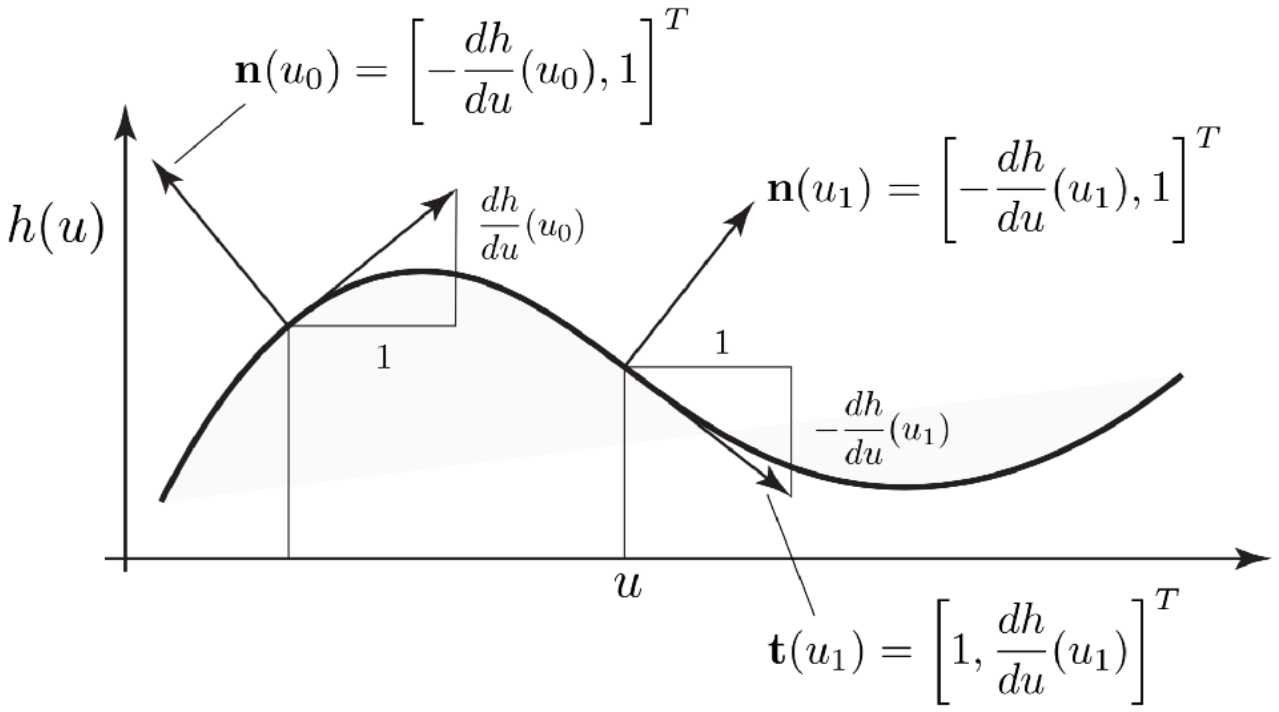
DISPLACEMENT MAP



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

$$\mathbf{n}^d(u,v) = \frac{\partial \mathbf{p}^d(u,v)}{\partial u} \times \frac{\partial \mathbf{p}^d(u,v)}{\partial v} =$$
$$= \frac{\partial \left[\mathbf{p}(u,v) + h(u,v)\mathbf{n}(u,v)\right]}{\partial u} \times \frac{\partial \left[\mathbf{p}(u,v) + h(u,v)\mathbf{n}(u,v)\right]}{\partial v}$$

Computing displaced normals

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

$$\mathbf{n}^d = \frac{\partial[\mathbf{p}+h\mathbf{n}]}{\partial u} \times \frac{\partial[\mathbf{p}+h\mathbf{n}]}{\partial v} \approx \left[\frac{\partial \mathbf{p}}{\partial u} + \frac{\partial h}{\partial u}\mathbf{n}\right] \times \left[\frac{\partial \mathbf{p}}{\partial v} + \frac{\partial h}{\partial v}\mathbf{n}\right] =$$

$$= \left[\frac{\partial \mathbf{p}}{\partial u} + \frac{\partial h}{\partial u}\mathbf{n}\right] \times \left[\frac{\partial \mathbf{p}}{\partial v} + \frac{\partial h}{\partial v}\mathbf{n}\right] = \left[\mathbf{t}_u + \frac{\partial h}{\partial u}\mathbf{n}\right] \times \left[\mathbf{t}_v + \frac{\partial h}{\partial v}\mathbf{n}\right] =$$

$$= \mathbf{t}_u \times \mathbf{t}_v + \frac{\partial h}{\partial v}(\mathbf{t}_u \times \mathbf{n}) + \frac{\partial h}{\partial v}(\mathbf{n} \times \mathbf{t}_v)$$

- 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

- T*angent 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

$$\mathbf{p}^d = T \begin{bmatrix} 0 & 0 & h & 1 \end{bmatrix}^T \quad \mathbf{n}^d = T \begin{bmatrix} -\frac{\partial h}{\partial u} & -\frac{\partial h}{\partial v} & 1 & 0 \end{bmatrix}^T$$

- 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 **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

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

Tangent frames for meshes

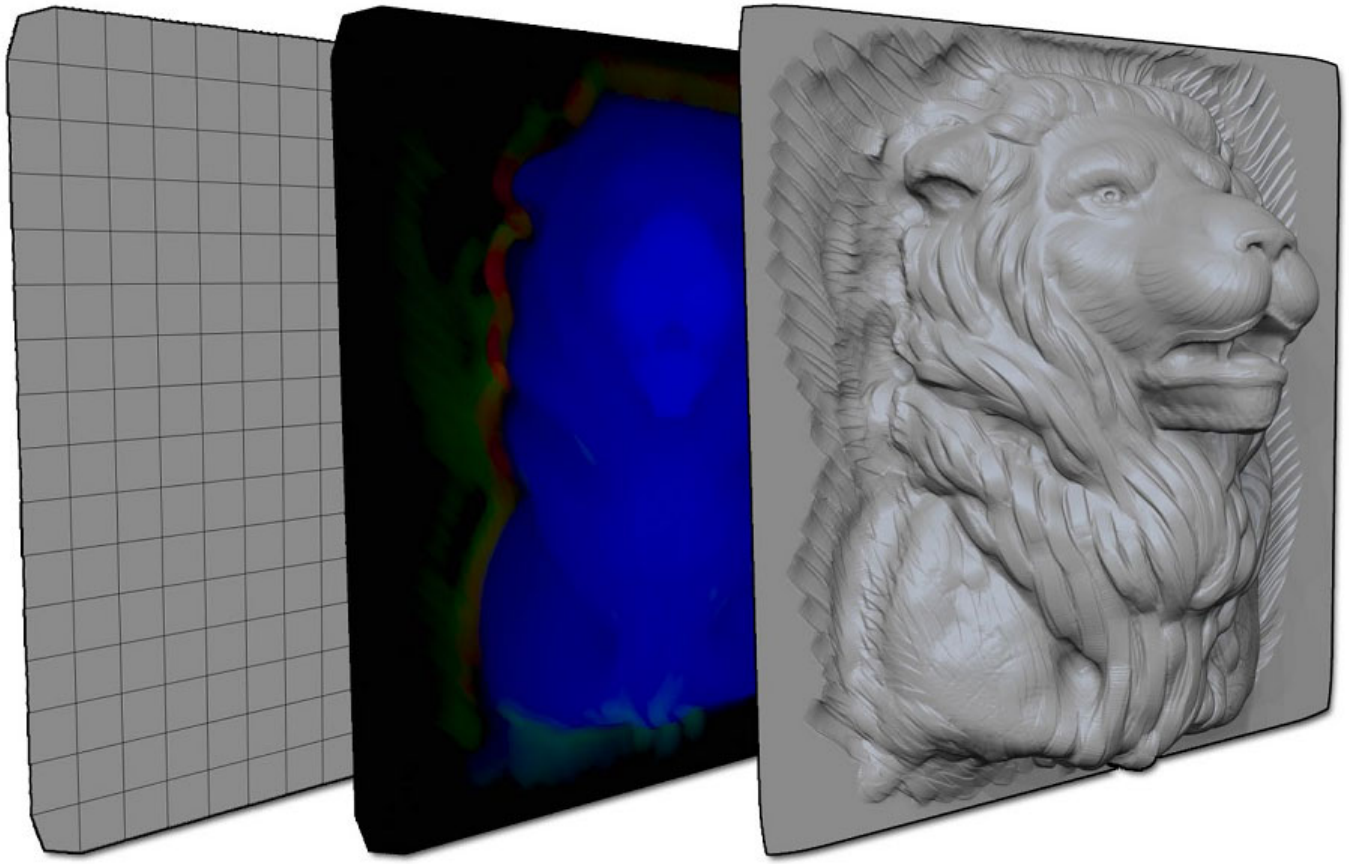
- We now multiply both sides by the inverse of the *(s,t)* matrix

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

- 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



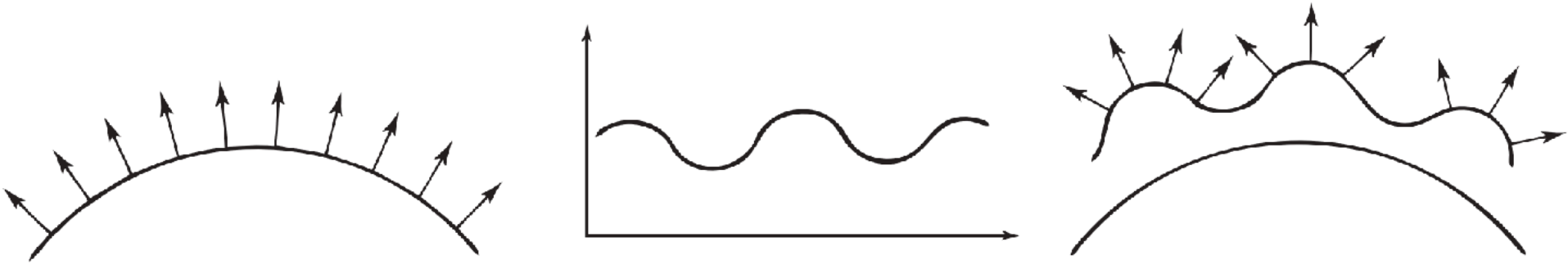
[Pixologic]

Vector displacement

[Yining Karl]

Bump mapping

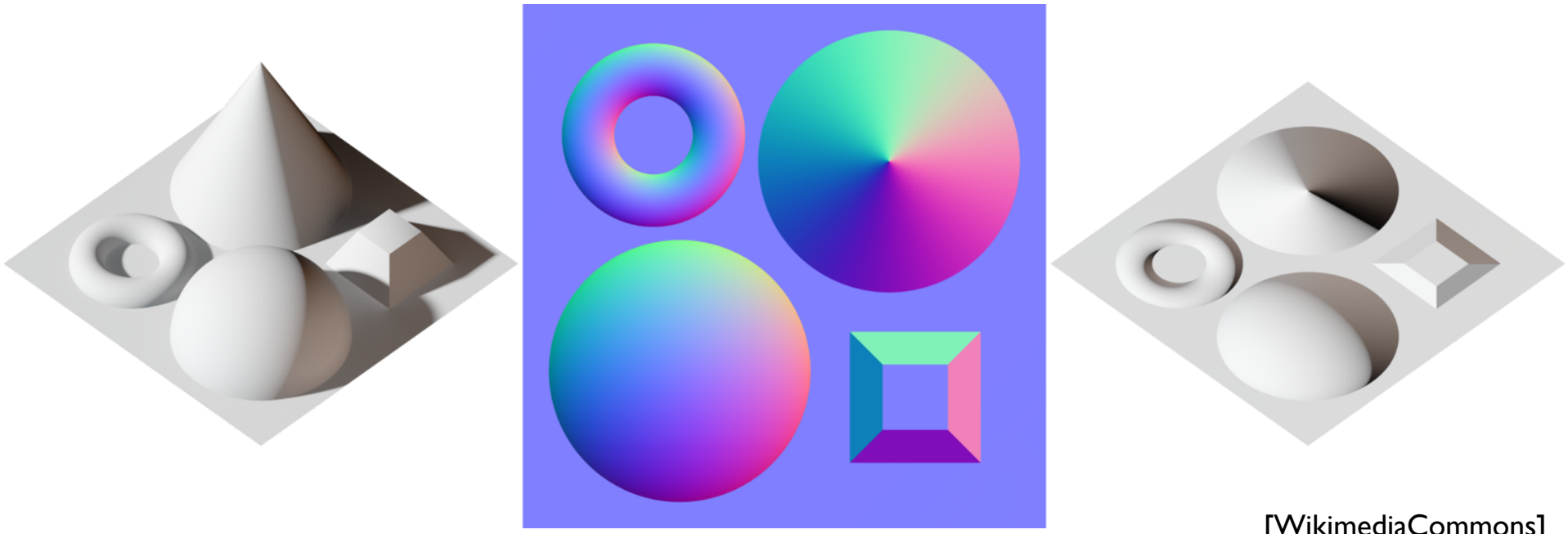
- Displacement mapping is expensive since it requires tessellation
- For small displacement, most effects are in the normal
- *Bump mapping*: us displaced normal for shading, but original geometry for intersection and rasterization



(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

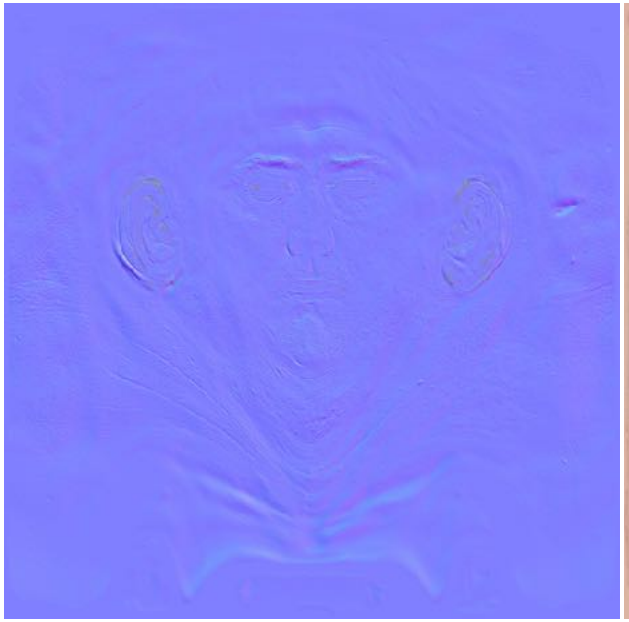


[WikimediaCommons]

Normal mapping

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

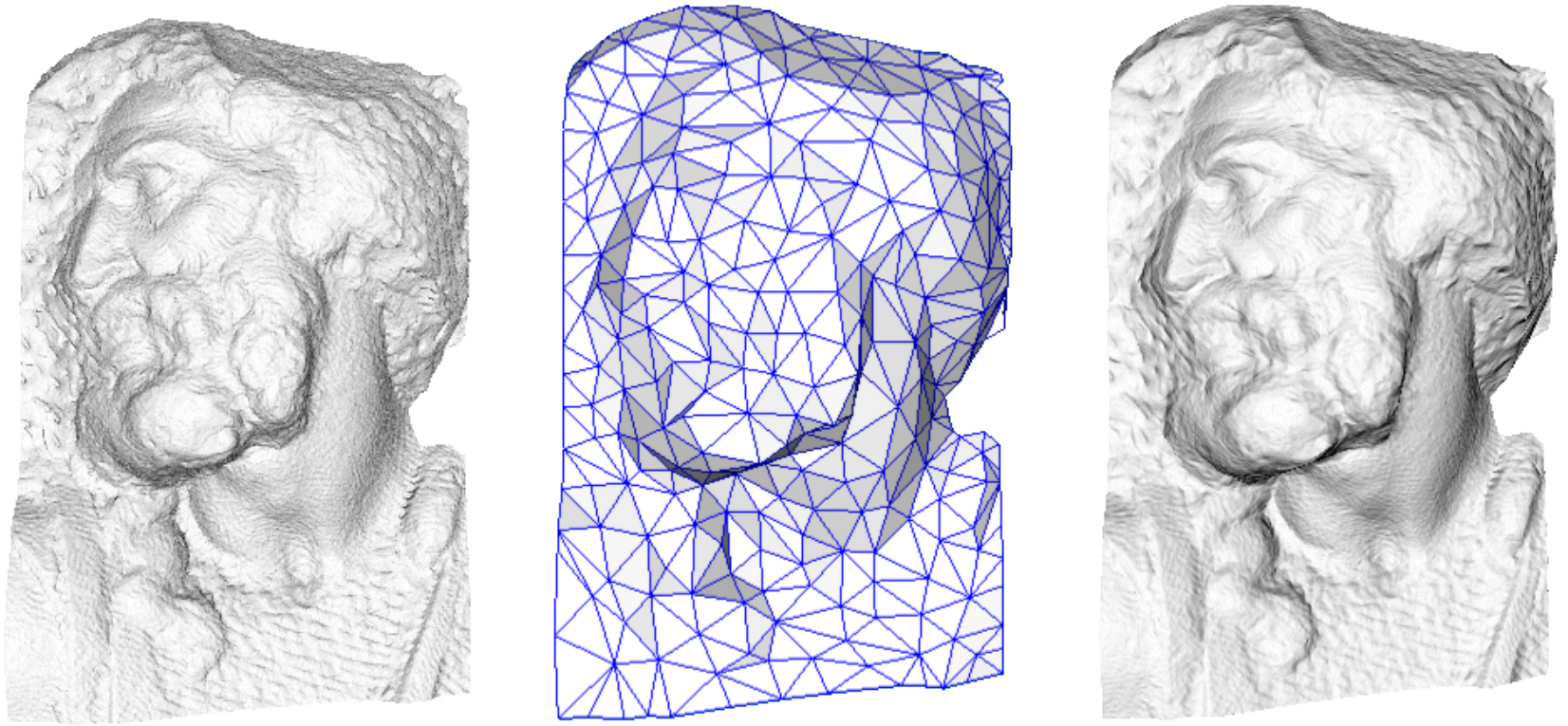
[BlenderNerd]



[Lee Perry-Smith]

Normal mapping

- Mostly used in industry since it supports simplification of meshes



original mesh
4M triangles

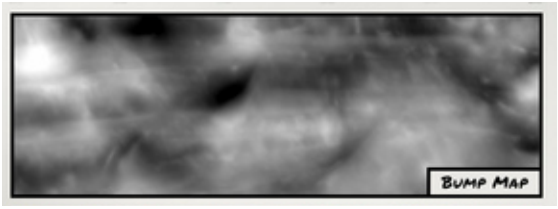
simplified mesh
500 triangles

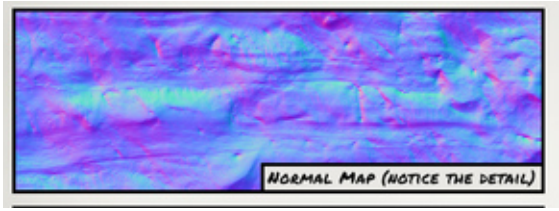
simplified mesh
and normal mapping
500 triangles

[Paolo Cignoni]

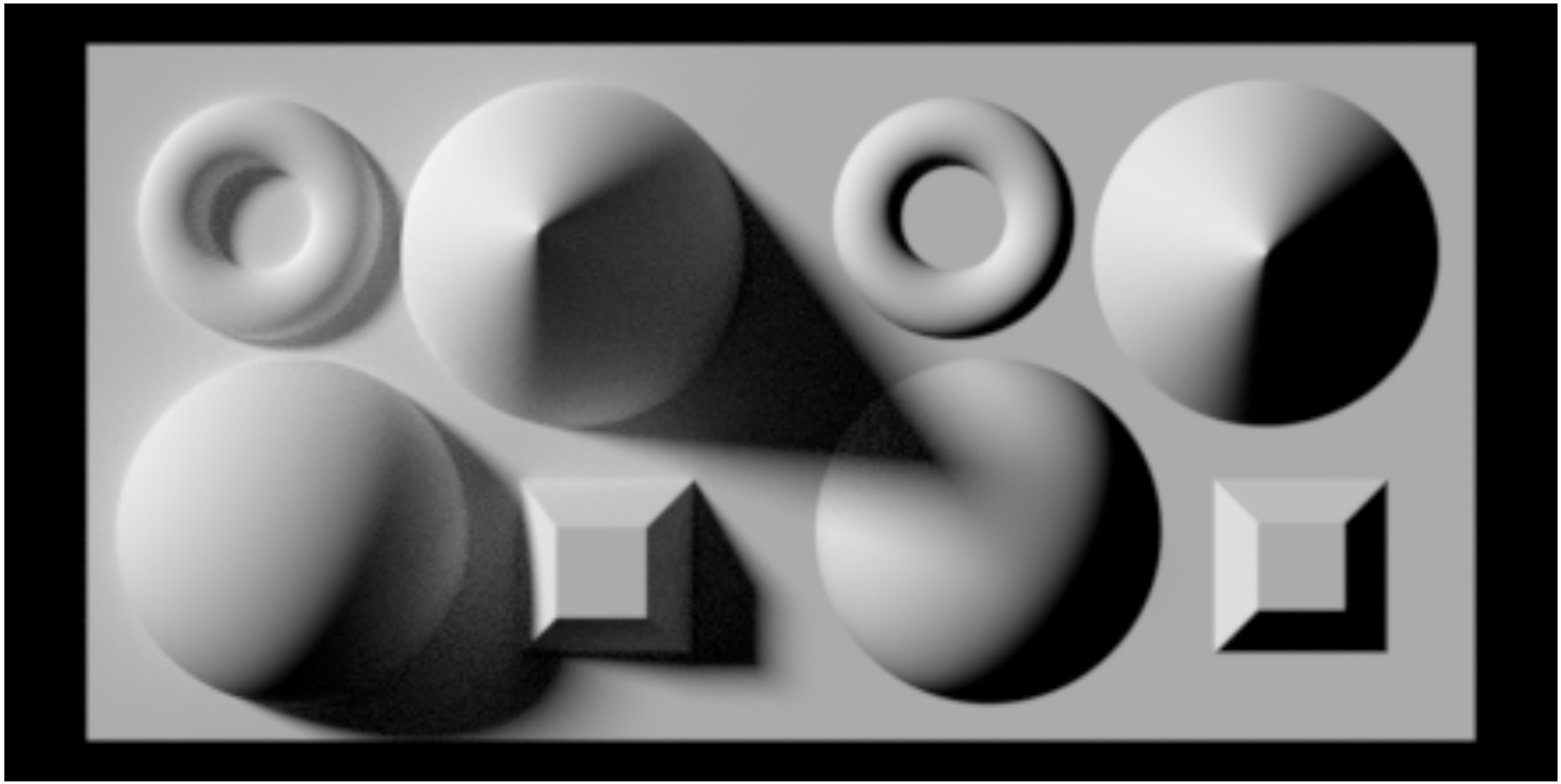
Bump vs normal vs displacement

[Andrew Letourneau]





Bump vs normal vs displacement



[WikimediaCommons]

Normal and displacement

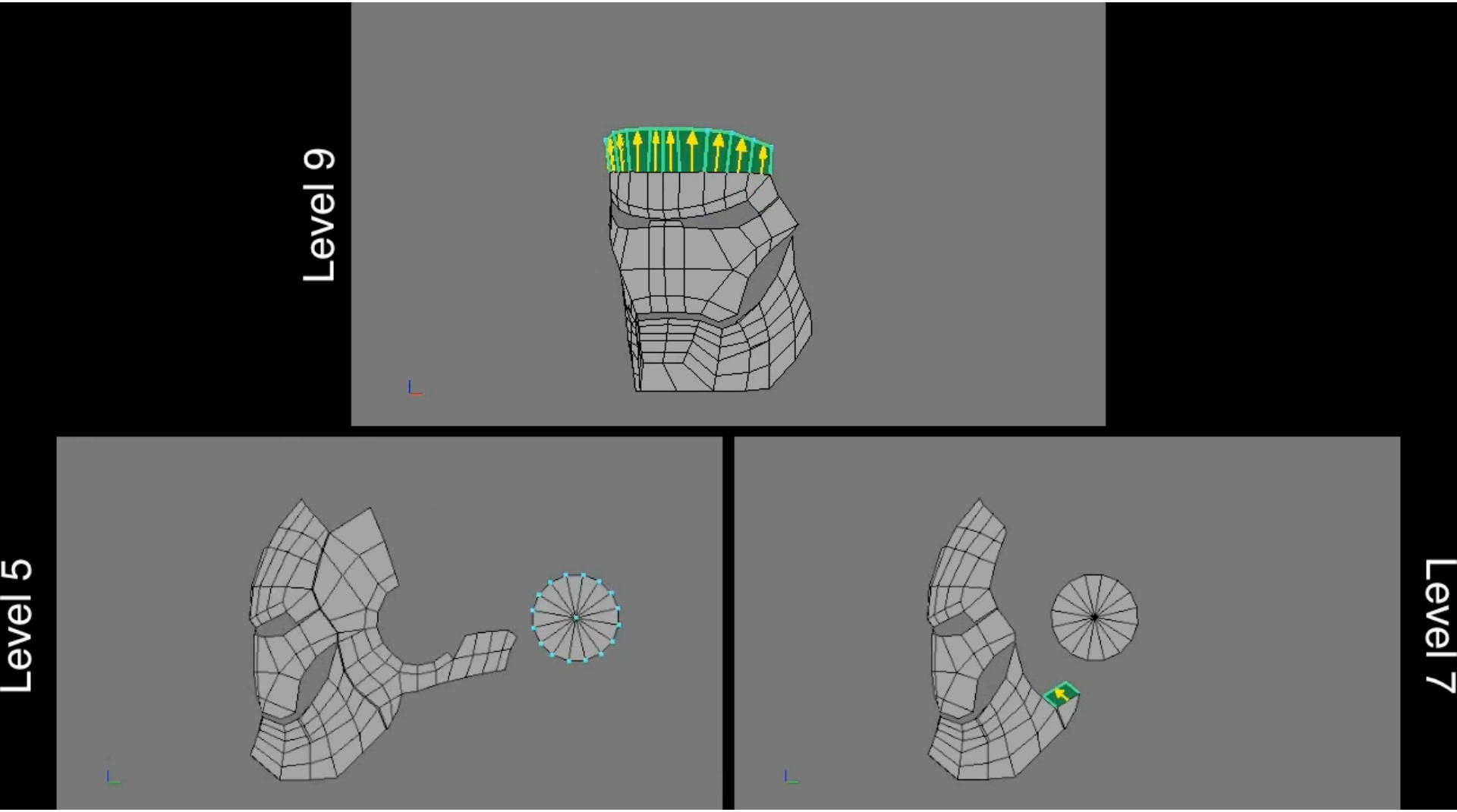
- Entertainment apps use normal or displacement everywhere
- Normal maps: used for real-time applications
  - mostly from compressed meshes
- Displacement maps: used in offline rendering for movies
  - production renderers are optimized for displacement
  - handle high details, while the rest of the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

Editing Surfaces

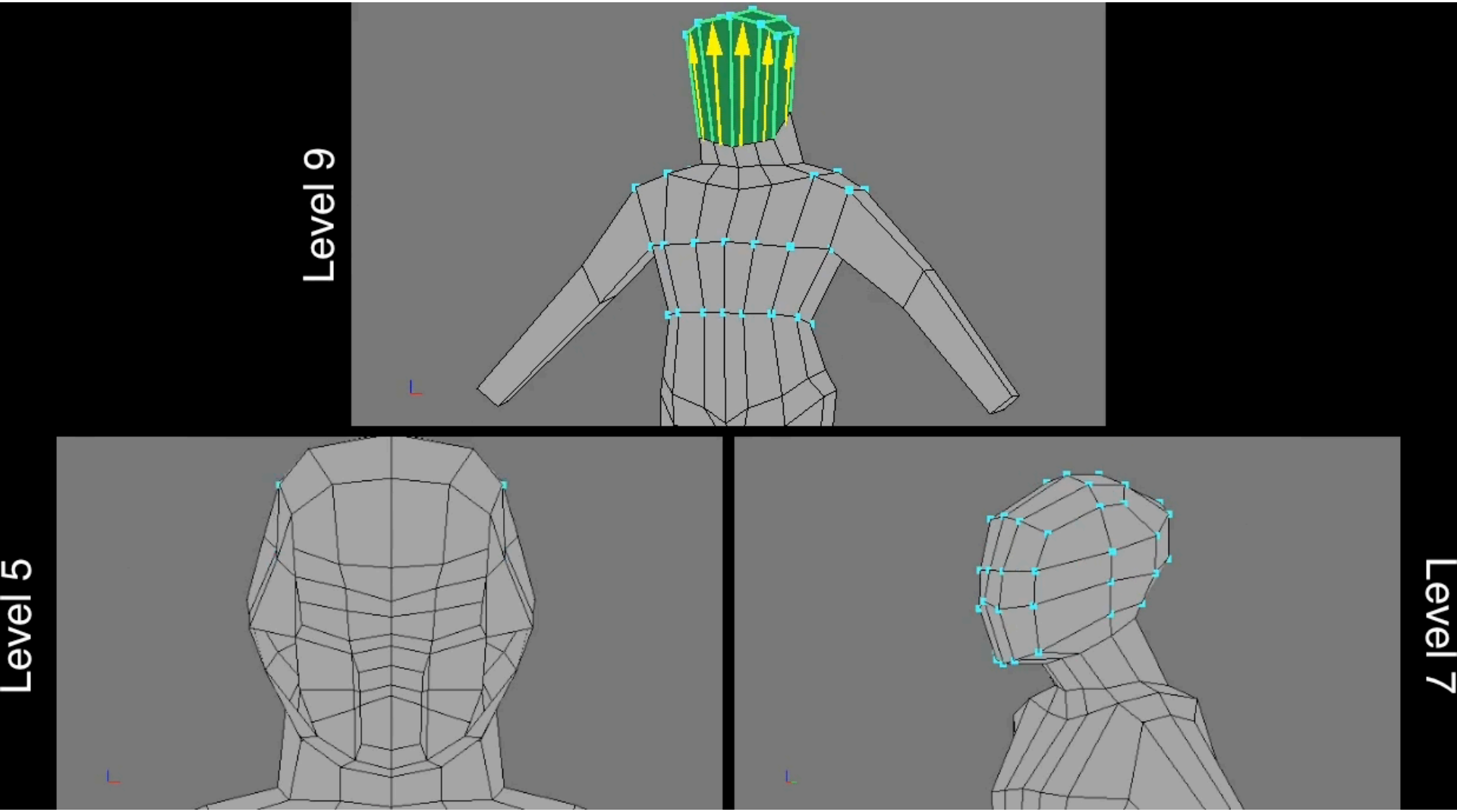
Two different workflows

- 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

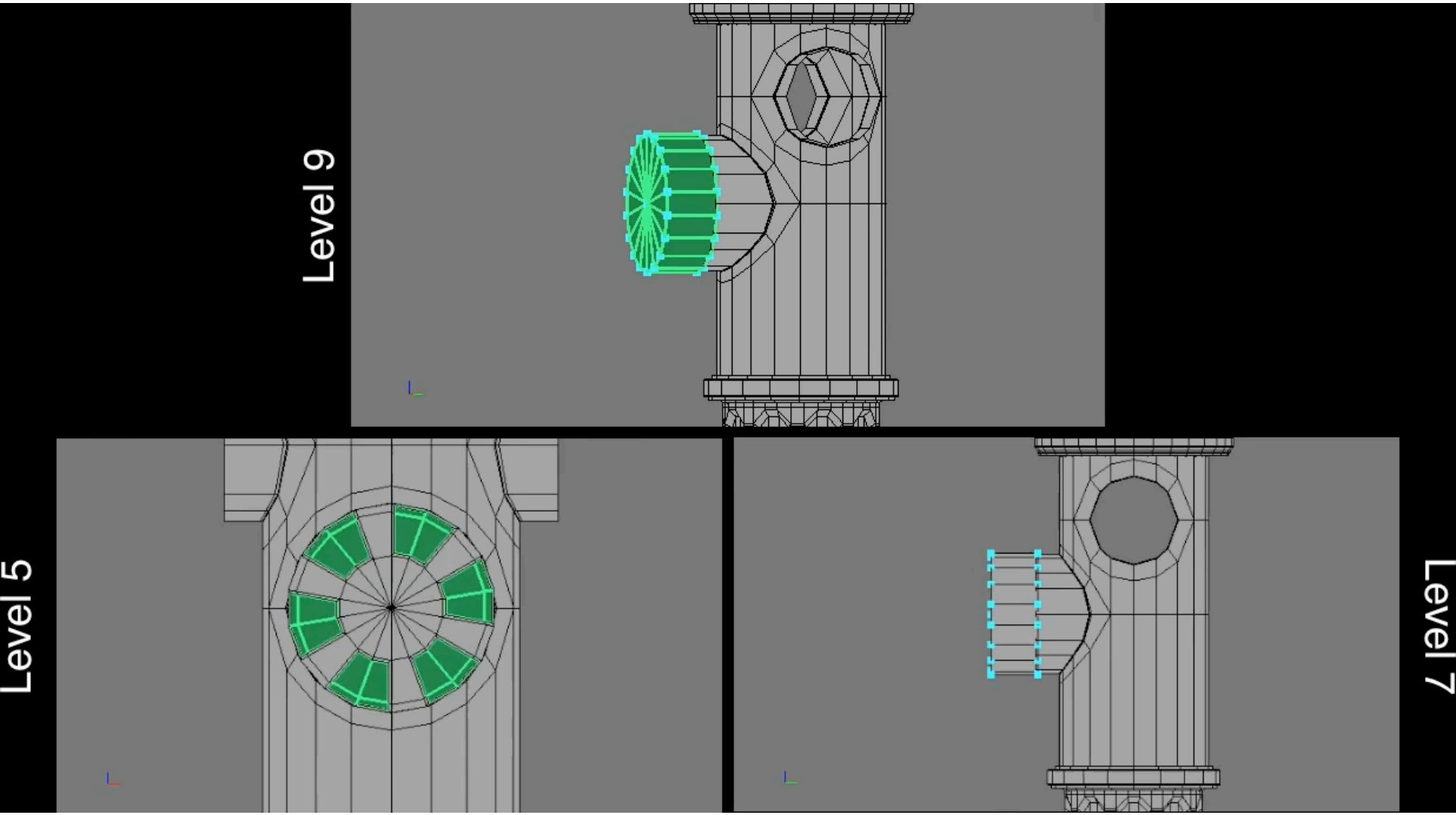
Low poly modeling



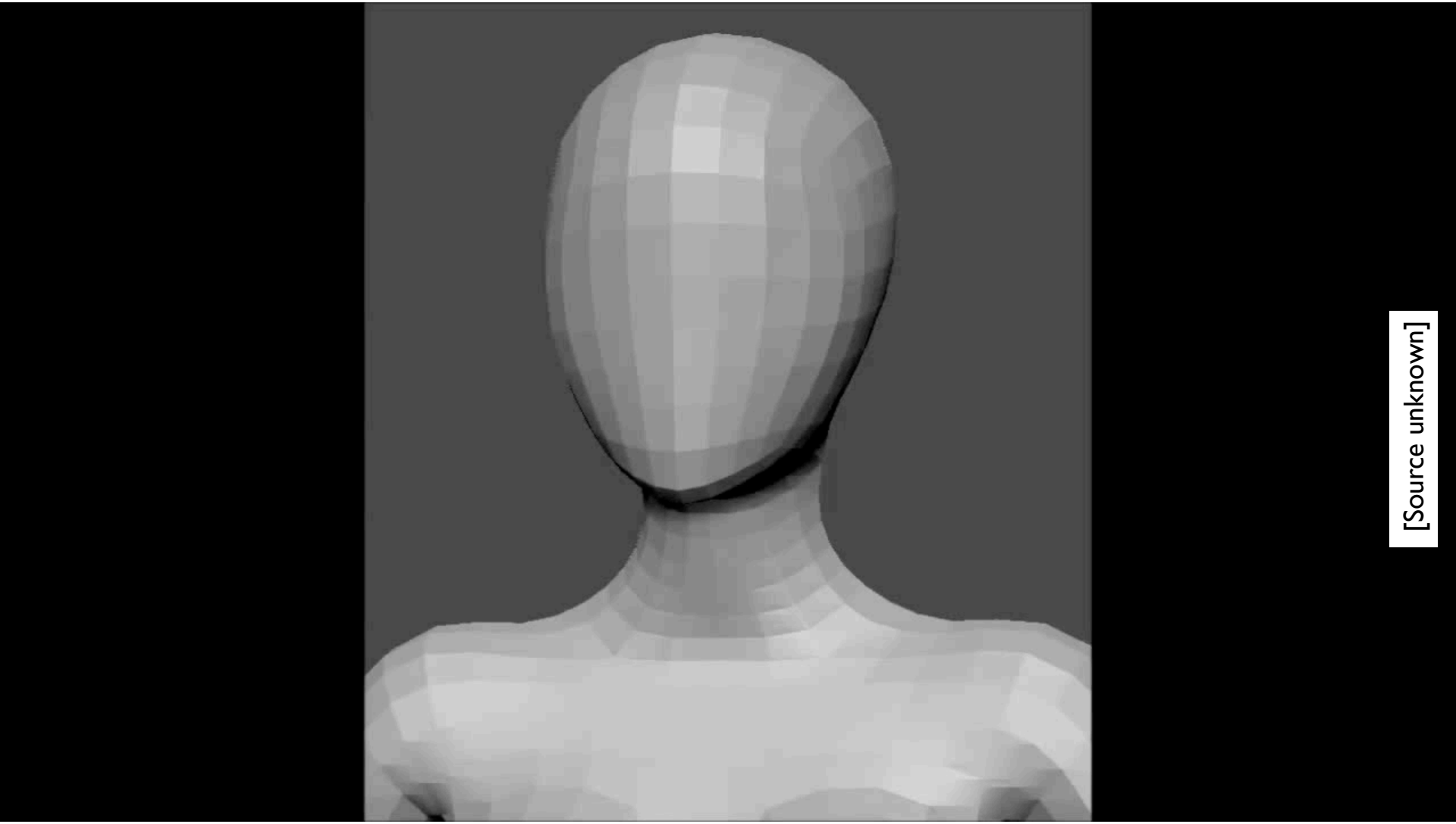
Low poly modeling



Low poly modeling

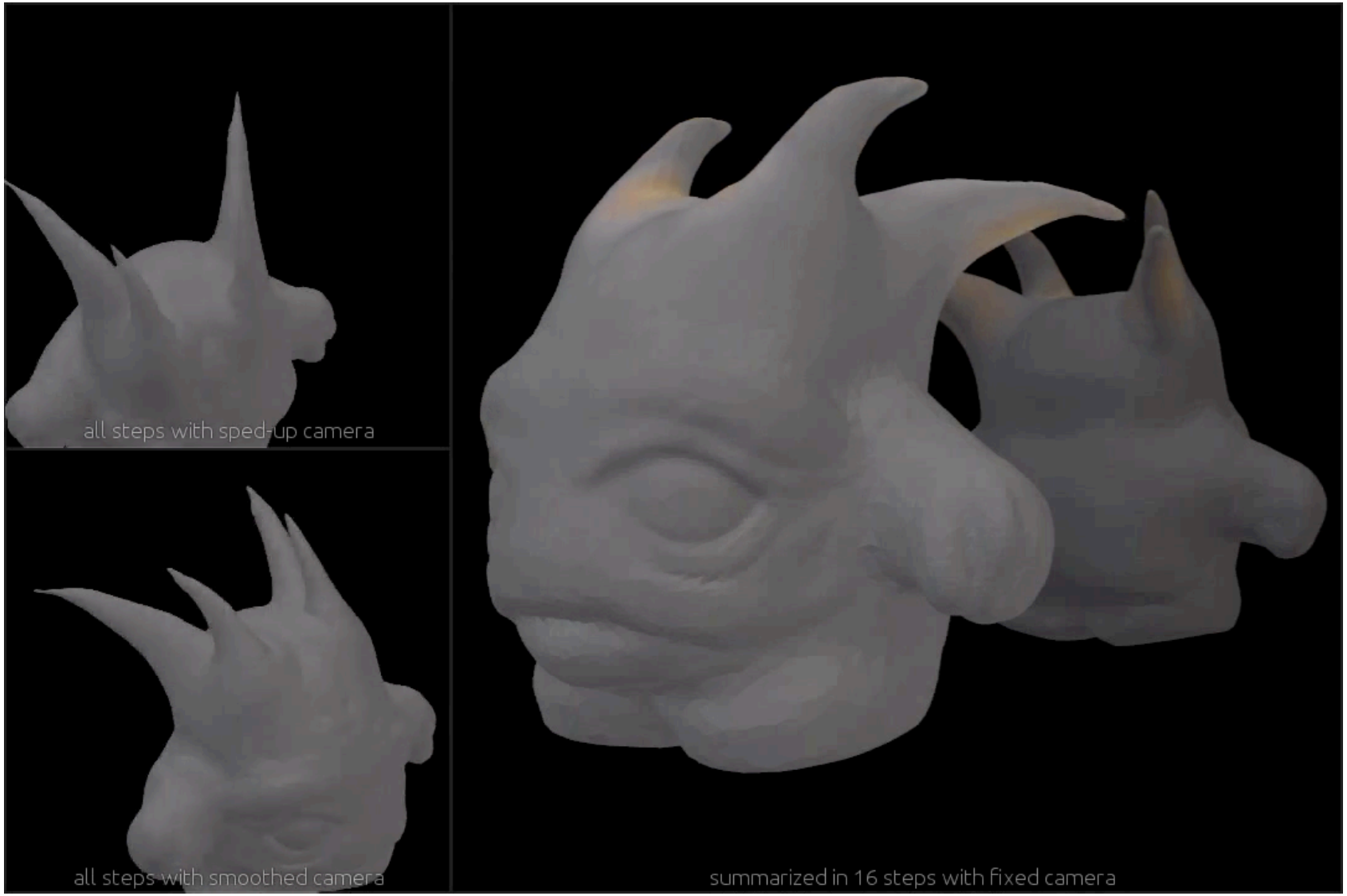


Sculpting



[Source unknown]

Sculpting

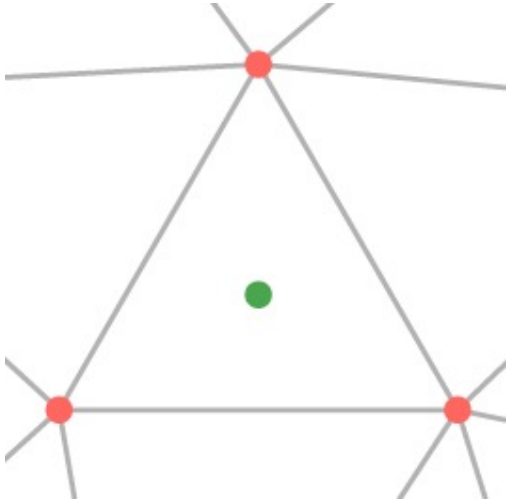


Triangle Mesh Data Structures

Separate triangles

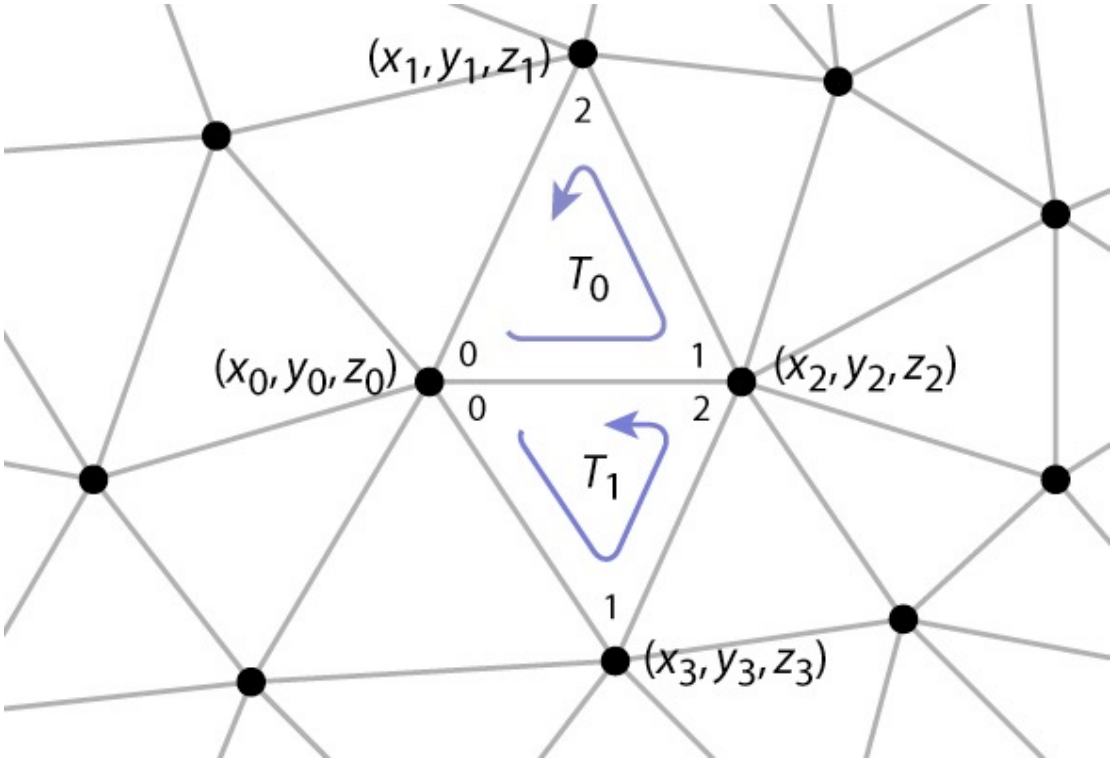
- Store each triangle independently

```
struct Mesh {
  vec3f pos[nT][3];
};
```



Separate triangles

	[0]	[1]	[2]
tris[0]	x_0, y_0, z_0	x_2, y_2, z_2	x_1, y_1, z_1
tris[1]	x_0, y_0, z_0	x_3, y_3, z_3	x_2, y_2, z_2
	⋮	⋮	⋮



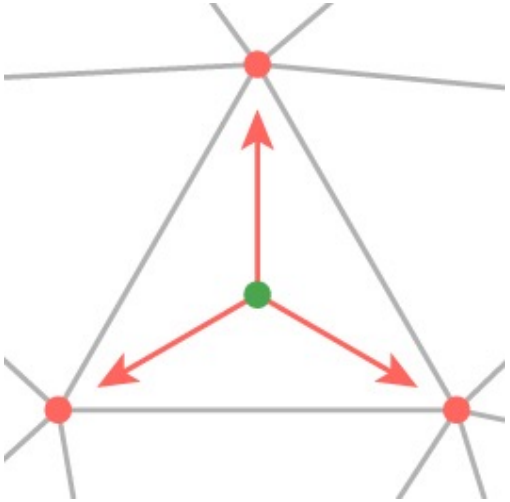
Separate triangles

- Array of triples of points
  - float[n_T][3][3]: about 72 bytes per vertex
  - 2 triangles per vertex (on average)
- Various problems
  - wastes space (each vertex stored 6 times)
  - cracks due to roundoff
  - difficulty of finding neighbors at all

Indexed triangles

- Store each vertex once
- Each triangle points to its three vertices

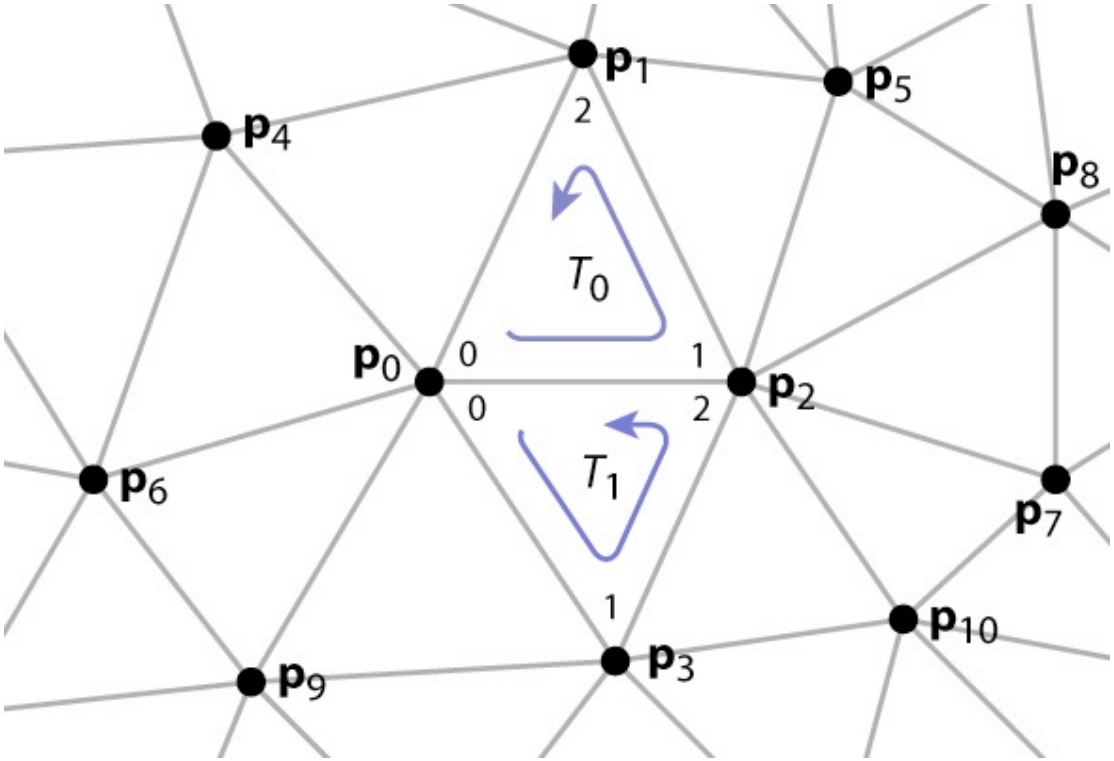
```
struct Mesh {
  vec3i tris[nT];
  vec3f pos[nV];
};
```



Indexed triangles

verts[0]	x_0, y_0, z_0
verts[1]	x_1, y_1, z_1
	x_2, y_2, z_2
	x_3, y_3, z_3
	⋮

tInd[0]	0, 2, 1
tInd[1]	0, 3, 2
	⋮



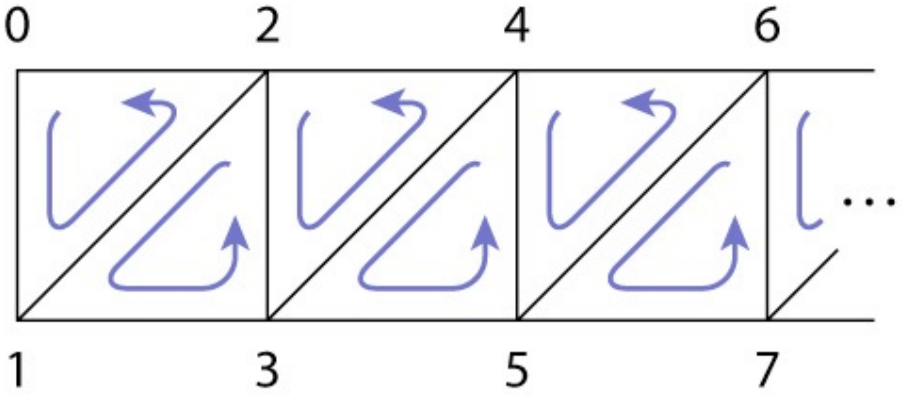
Indexed triangles

- Array of vertex positions
  - float[n_V][3]: 12 bytes per vertex
- Array of triangle indices
  - int[n_T][3]: 12 bytes per triangle
  - 2 triangles per vertex on average: 24 bytes per vertex
- Total storage: 36 bytes per vertex (factor of 2 savings)
- Represents topology and geometry separately
- Finding neighbors is well defined

Triangle strips

- Compress triangle lists by skipping repeated indices
  - each triangle is usually adjacent to the previous
  - let every vertex create a triangle by reusing the second and third vertex of the previous triangle
  - for long strips, this requires about one index per triangle

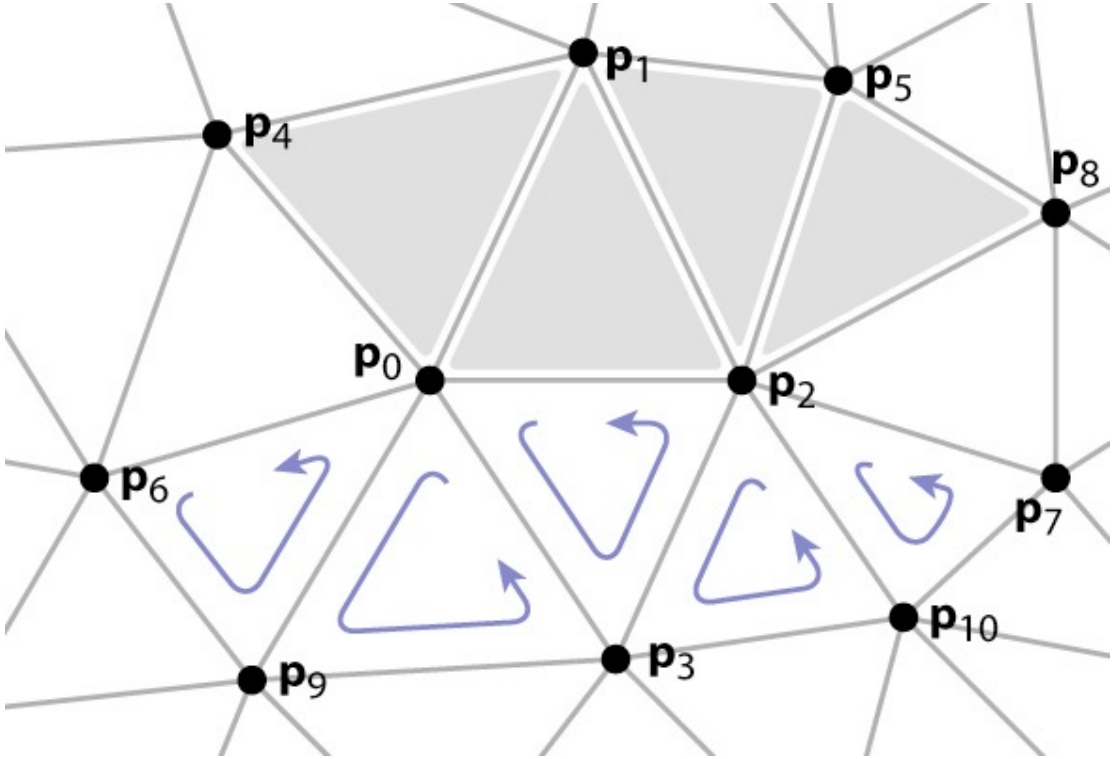
```
struct Mesh {
  int strips[nT][];
  float pos[nV][3];
};
```



Triangle strips

verts[0]	x_0, y_0, z_0
verts[1]	x_1, y_1, z_1
	x_2, y_2, z_2
	x_3, y_3, z_3
	⋮

tStrip[0]	6, 0, 4, 1, 2, 5, 8
tStrip[1]	6, 9, 0, 3, 2, 10, 7
	⋮



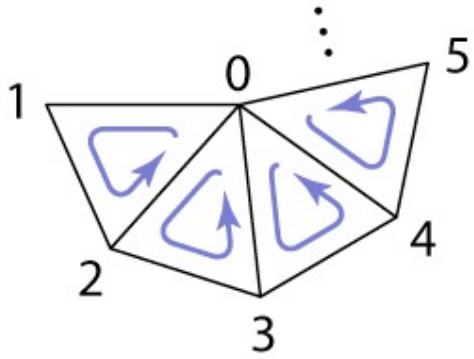
Triangle strips

- Array of vertex positions
  - float[n_V][3]: 12 bytes per vertex
- Array of index lists
  - int[n_S][variable]: (2 + n)x4 bytes per strip
  - on average, (1 + ε) indices per triangle (assuming long strips)
- Total is 20 bytes per vertex (limiting best case)
  - factor of 3.6 over separate triangles; 1.8 over indexed mesh

Triangle fans

- Same idea as triangle strips, but keep oldest rather than newest
  - every sequence of three vertices produces a triangle
  - for long fans, this requires about one index per triangle
- Memory considerations exactly the same as triangle strip

```
struct Mesh {
  int fans[nT][];
  float pos[nV][3];
};
```



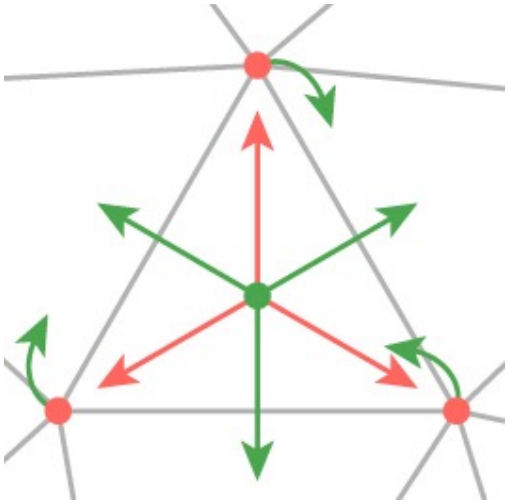
Data on meshes

- Often need to store additional information besides just the geometry
- Can store additional data at faces, vertices, or edges
- Store on faces for discontinuous data, e.g. facet colors
- Information about creases stored at edges
- Quantities that vary continuously (without sudden changes, or discontinuities) gets stored at vertices

Triangle neighbor structure

- Extension to indexed triangle set
- Triangle points to its three neighboring triangles
- Vertex points to a single neighboring triangle
- Can now enumerate triangles around a vertex

```
struct Mesh {
  float pos[nV][3];
  int tris[nT][3];
  int tAdj[nT][3];
  int vAdj[nV];
};
```

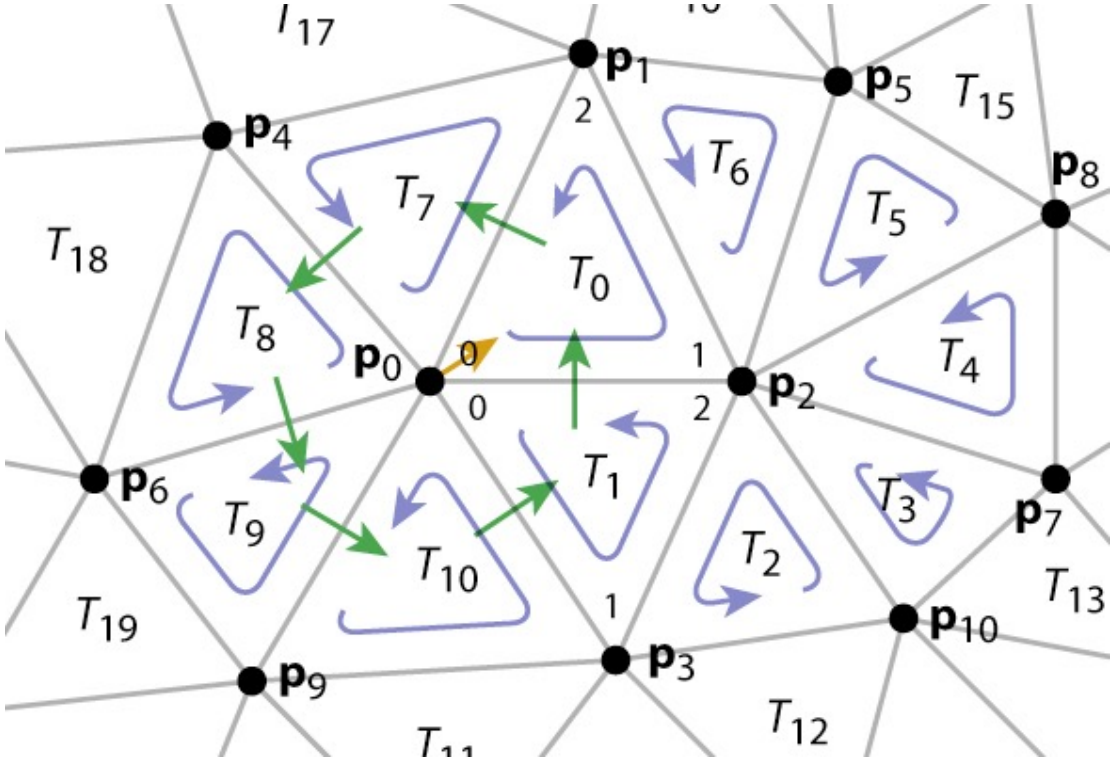


Triangle neighbor structure

tInd[0]	0, 2, 1
tInd[1]	0, 3, 2
tInd[2]	10, 2, 3
tInd[3]	2,10,7
	⋮

tNbr[0]	1, 6, 7
tNbr[1]	10, 2, 0
tNbr[2]	3, 1, 12
tNbr[3]	2, 13, 4
	⋮

vTri[0]	0
vTri[1]	6
vTri[2]	1
vTri[3]	1
	⋮

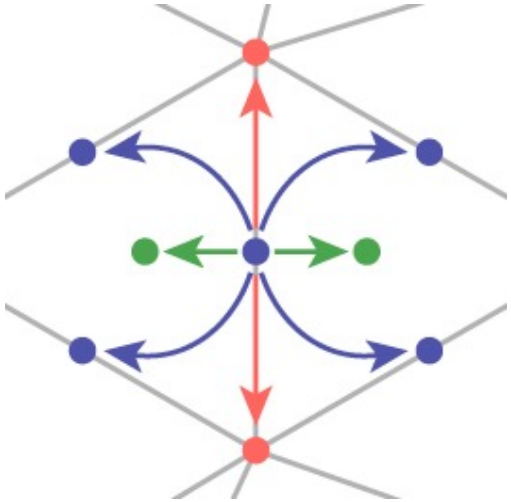


Triangle neighbor structure

- Indexed mesh was 36 bytes per vertex
- Add an array of triples of indices (per triangle)
  - int[n_T][3]: about 24 bytes per vertex
- Add an array of representative triangle per vertex
  - int[n_V]: 4 bytes per vertex
- Total storage: 64 bytes per vertex
  - still not as much as separate triangles

Winged-edge mesh

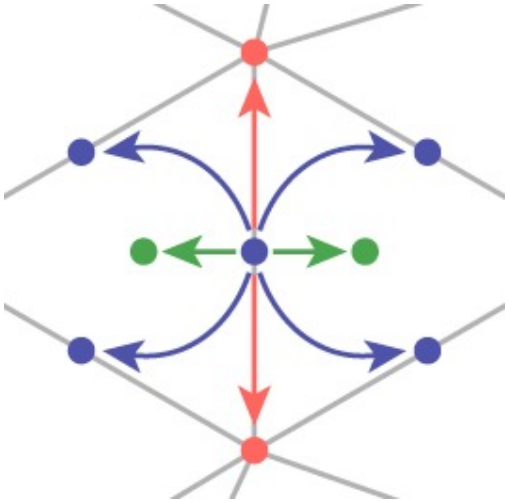
- Edge-centric rather than face-centric (works for general polygons)
- Each (oriented) edge points to:
  - left and right forward edges, left and right backward edges
  - front and back vertices, left and right faces
- Each face or vertex points to one edge



Winged-edge mesh

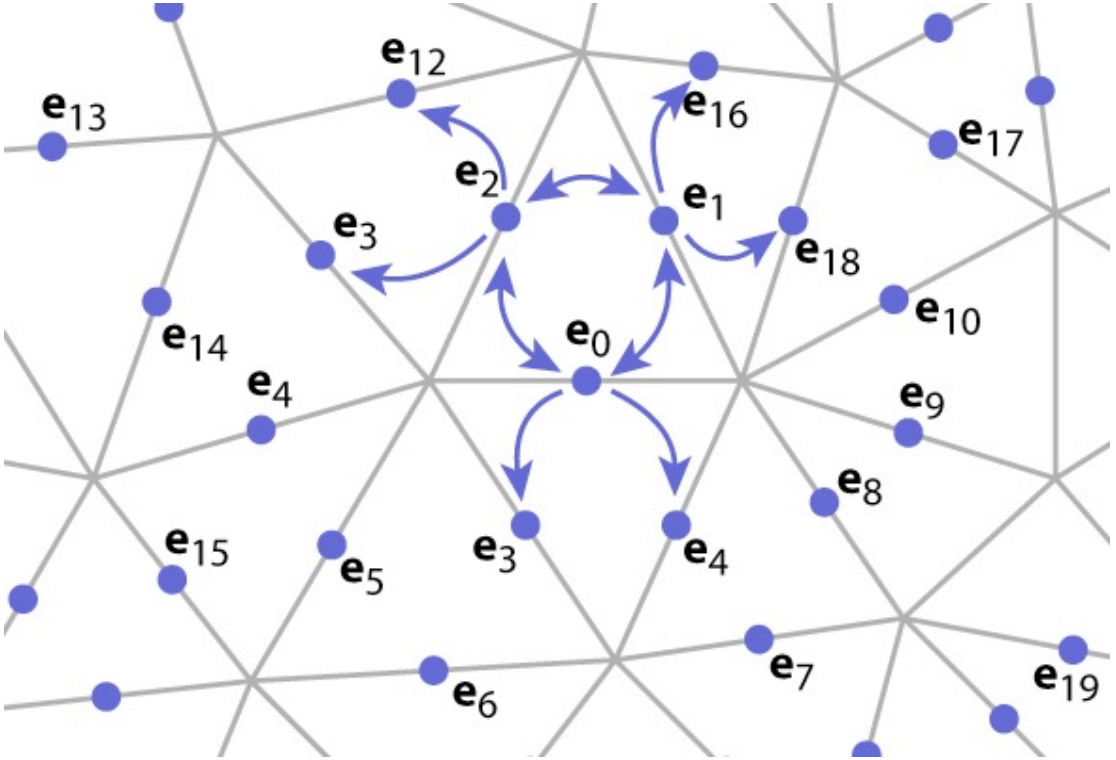
```
struct Edge {
  // edges
  int hl, hr, tl, tr;
  int h, t; // vertices
  int l, r; // face
};

struct Mesh {
  Edge edges[nE];
  int fEdge[nT];
  // vertex props…
};
```



Winged-edge structure

	hl	hr	tl	tr
edge[0]	1	4	2	3
edge[1]	18	0	16	2
edge[2]	12	1	3	0
	⋮			

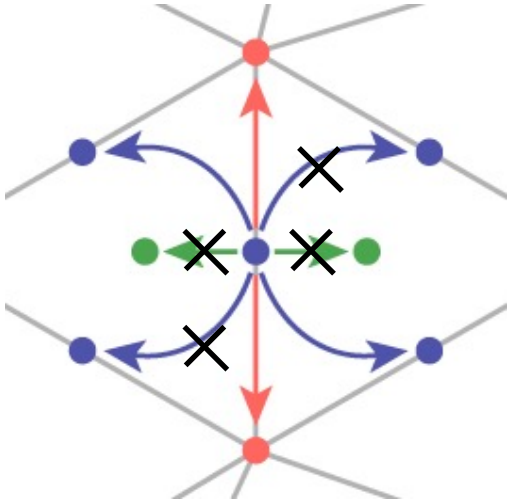


Winged-edge structure

- Array of vertex positions: 12 bytes/vert
- Array of 8-tuples of indices (per edge)
  - head/tail left/right edges + head/tail verts + left/right tris
  - int[n_E][8]: about 96 bytes per vertex
- Add a representative edge per vertex
  - int[n_V]: 4 bytes per vertex
- Total storage: 112 bytes per vertex
  - but it is cleaner and generalizes to polygon meshes

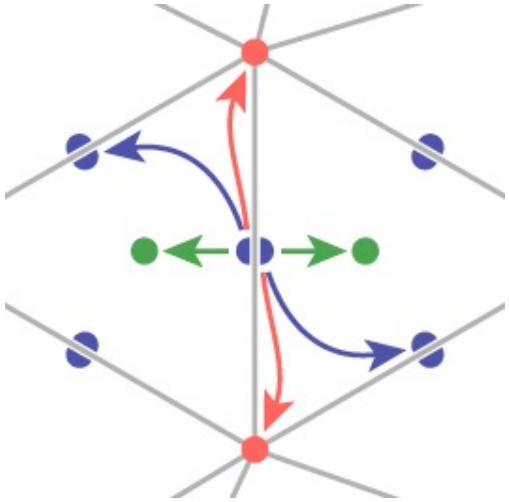
Winged-edge optimizations

- Omit faces if not needed
- Omit one edge pointer on each side
  - results in one-way traversal



Half-edge structure

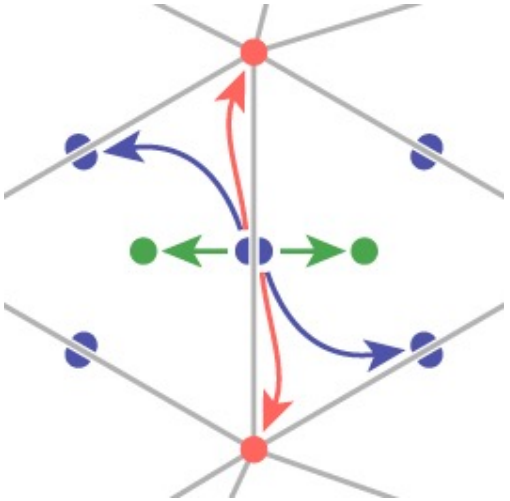
- Simplifies winged edge but still works for polygon meshes
- Each half-edge points to next edge (left forward), next vertex (front), the face (left) and the opposite half-edge
- Each face or vertex points to one half-edge



Half-edge structure

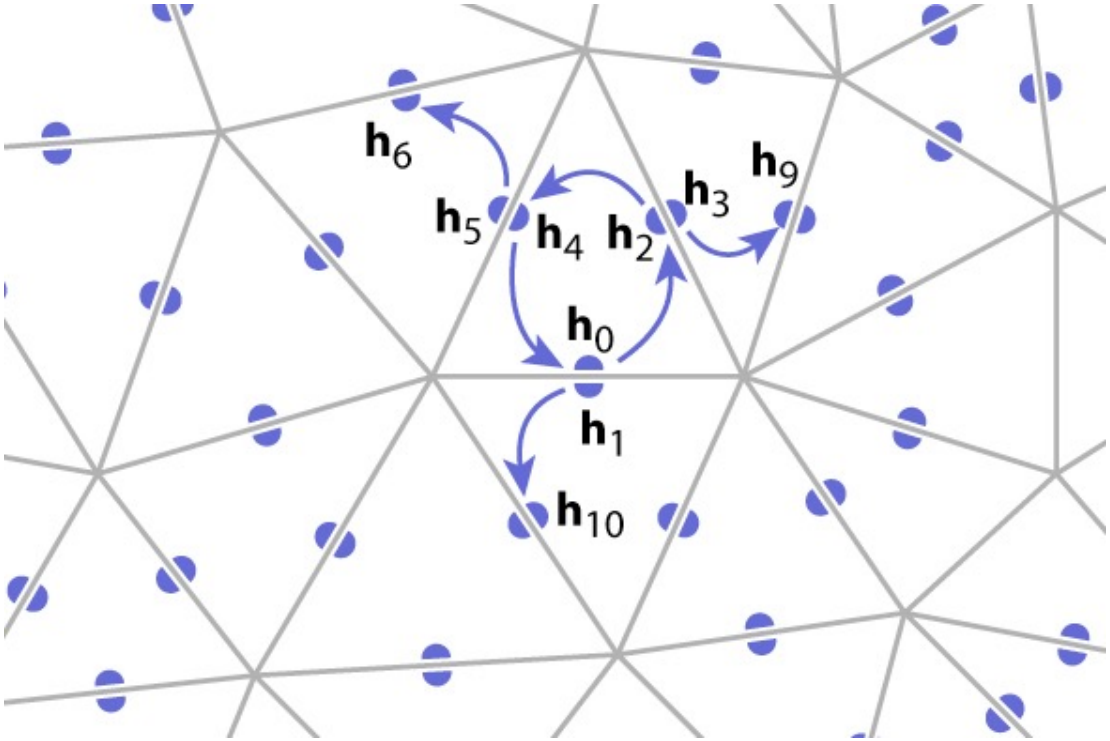
```
struct HalfEdge {
  int next;
  int opposing;
  int face;
  int vertex;
};

struct Mesh {
  HalfEdge edges[nE];
  int fEdge[nT];
  // vertex props…
};
```



Half-edge structure

	pair	next
hedge[0]	1	2
hedge[1]	0	10
hedge[2]	3	4
hedge[3]	2	9
hedge[4]	5	0
hedge[5]	4	6
	⋮	

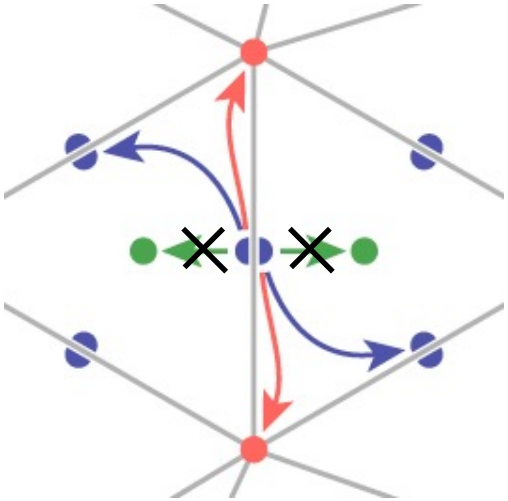


Half-edge structure

- Array of vertex positions: 12 bytes/vert
- Array of 4-tuples of indices (per h-edge)
  - next, pair h-edges + head vert + left tri
  - int[$2n_E$][4]: about 96 bytes per vertex
    - 6 h-edges per vertex (on average)
    - (4 indices x 4 bytes) per h-edge
- Add a representative h-edge per vertex
  - int[n_V]: 4 bytes per vertex
- Total storage: 112 bytes per vertex

Half-edge optimizations

- Omit faces if not needed
- Use implicit pair pointers
  - they are allocated in pairs
  - they are even and odd in an array



Data Structure in Practices

- Most used is the half-edge, but really rare compared to indexed
- Only useful for very specific problems and very hard to use
  - when not needed, performance is often worst then indexed
- In our work, we rebuild adjacency tables if needed

