

Representing the World

Prof. Fabio Pellacini



[WikimediaCommons]

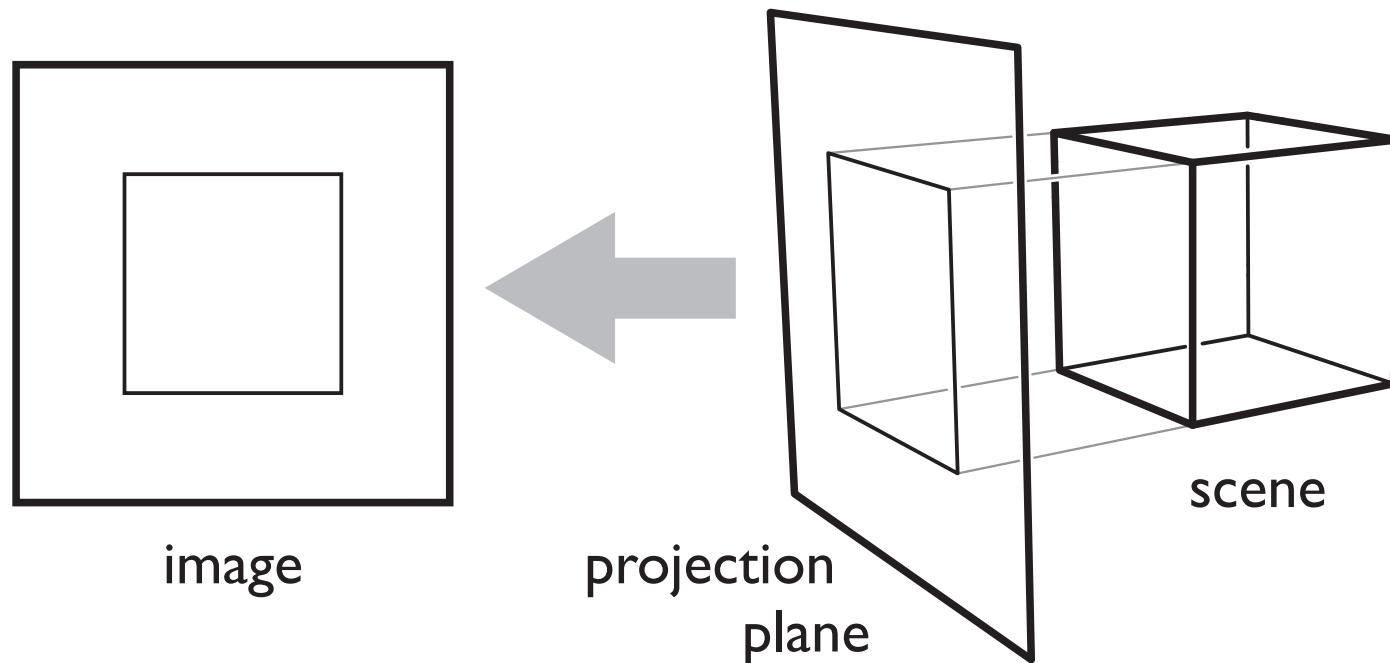
Scene elements

- To make an image, we need to represent
 - cameras
 - shapes
 - materials
- To make a movie, we need to also represent
 - animation (later)

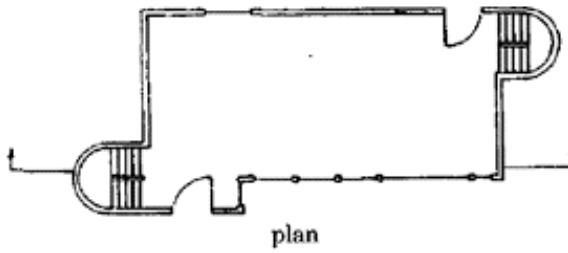
Representing Cameras

Orthographic Projections

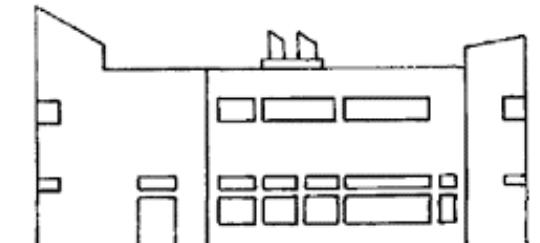
- To render an image of a 3D scene, we project it onto a plane
- Simplest type is parallel projection
- Mostly used in modeling interfaces



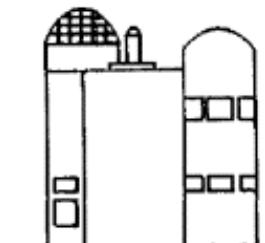
Orthographic Projection



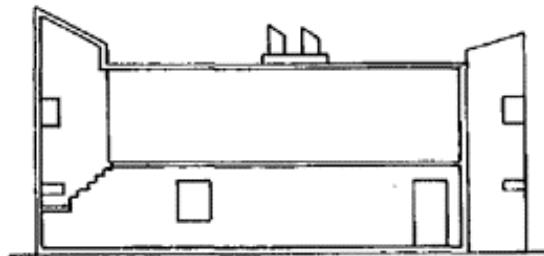
plan



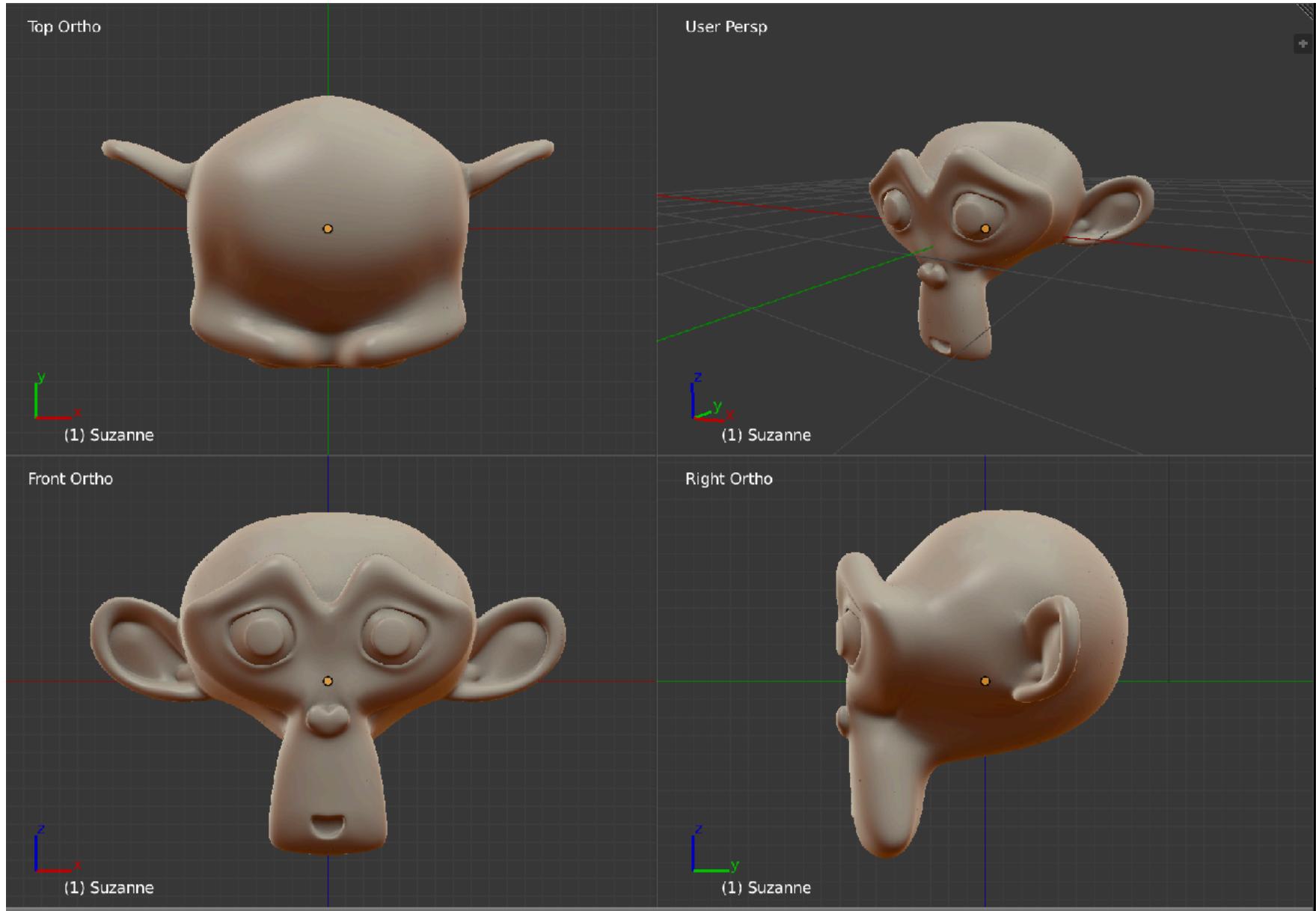
front elevation



right elevation

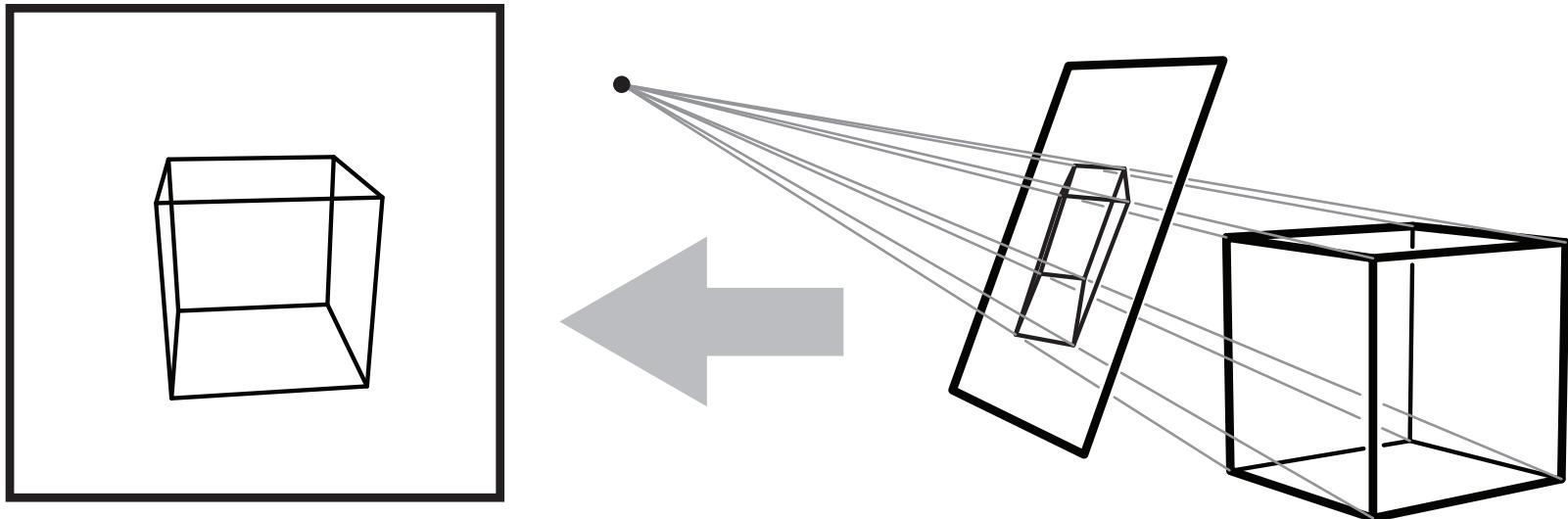


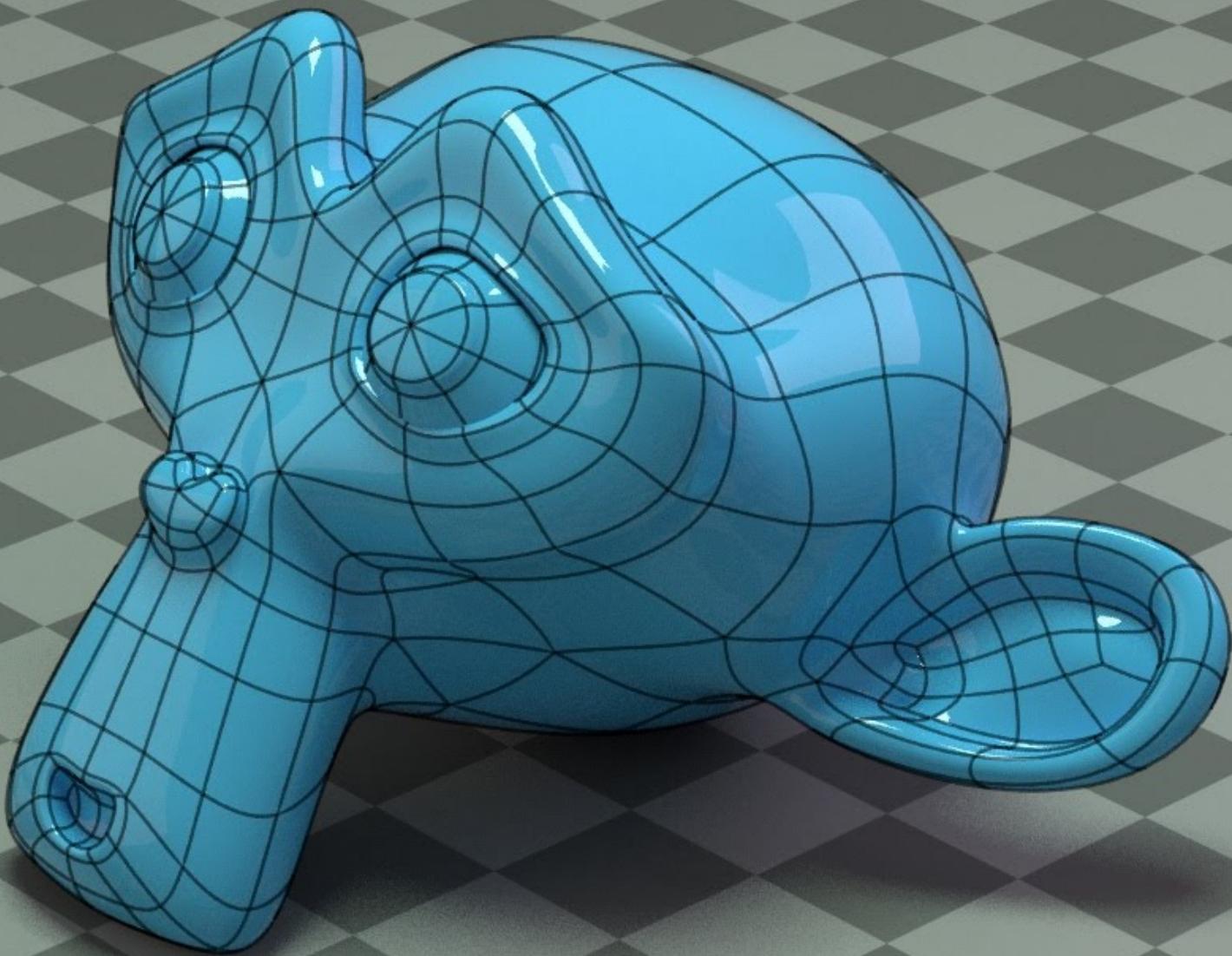
section



Perspective Projections

- Orthographic projection do not emulate a real camera
- Most common projection type is perspective projection
- Simulates a real pinhole camera





The Camera Obscura.



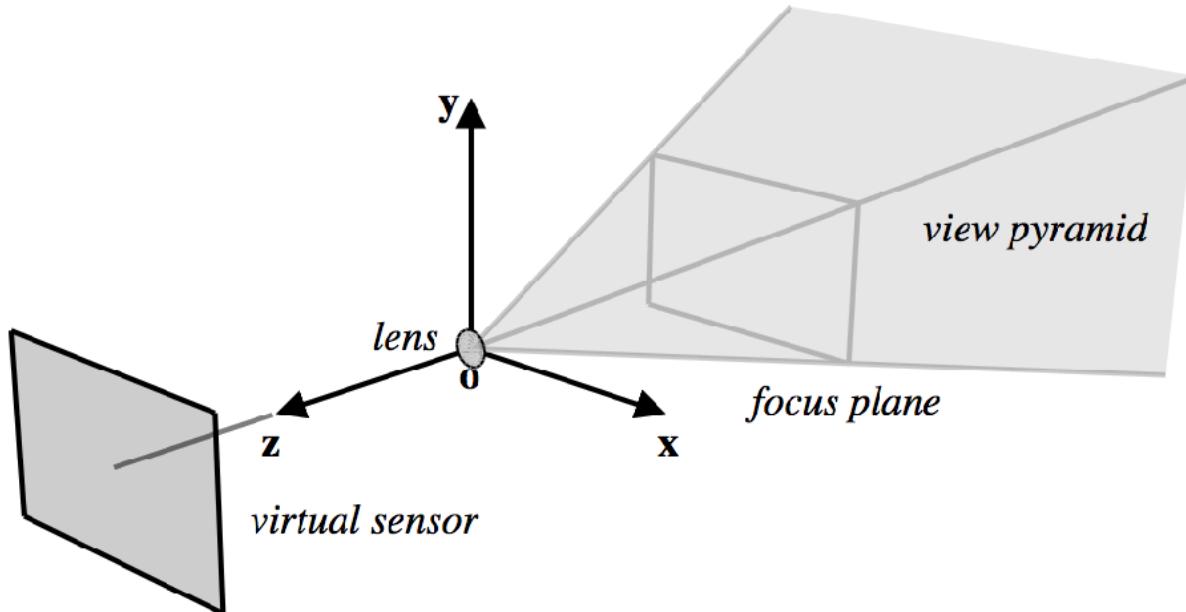
[WikipediaCommons]



[PetaPixel.com]

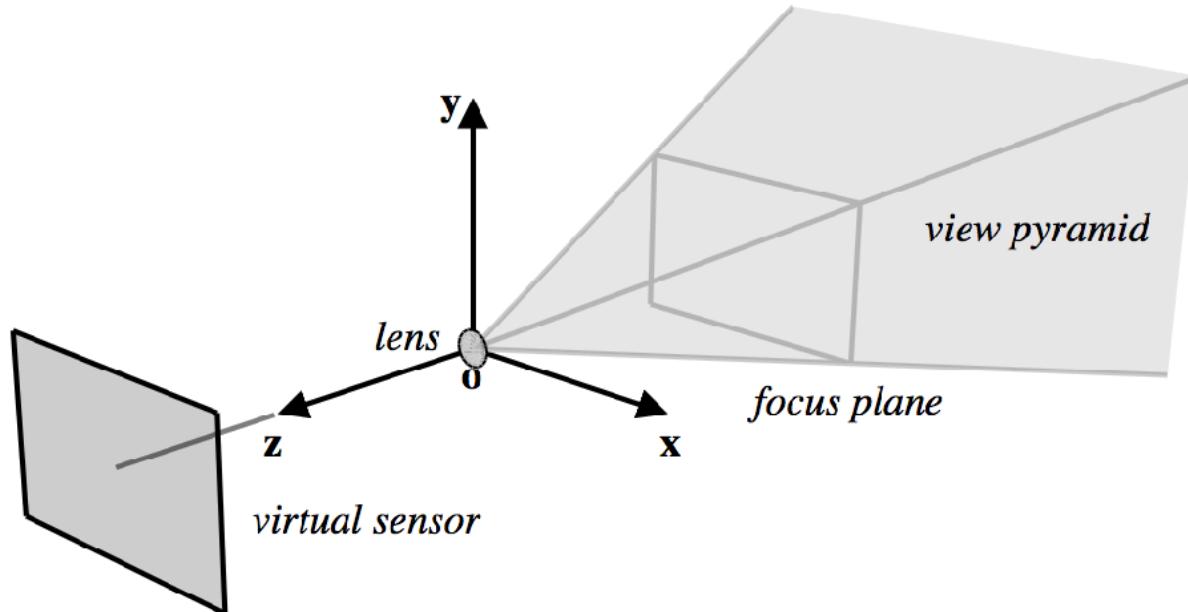
Camera Representation

- The camera position and orientation are defined by a frame
- The camera projection is defined by its lens field of view and the distance to the camera sensor
- The camera focus is defined by the lens aperture and the focus distance



Camera Representation

```
struct camera {  
    frame3f frame;  
    float lens, aspect, film;  
    float aperture, focus;  
};
```



Defining Camera Frame

- Providing the frame directly is unintuitive
- Calculate frame camera position and view direction
 - camera position **o**, where the camera is, set as frame origin
 - view direction **-z**: which way the camera is looking
 - up vector **y**: how the camera is oriented, generally world up
- Since view direction and up vector are not orthogonal, use frame building code shown before

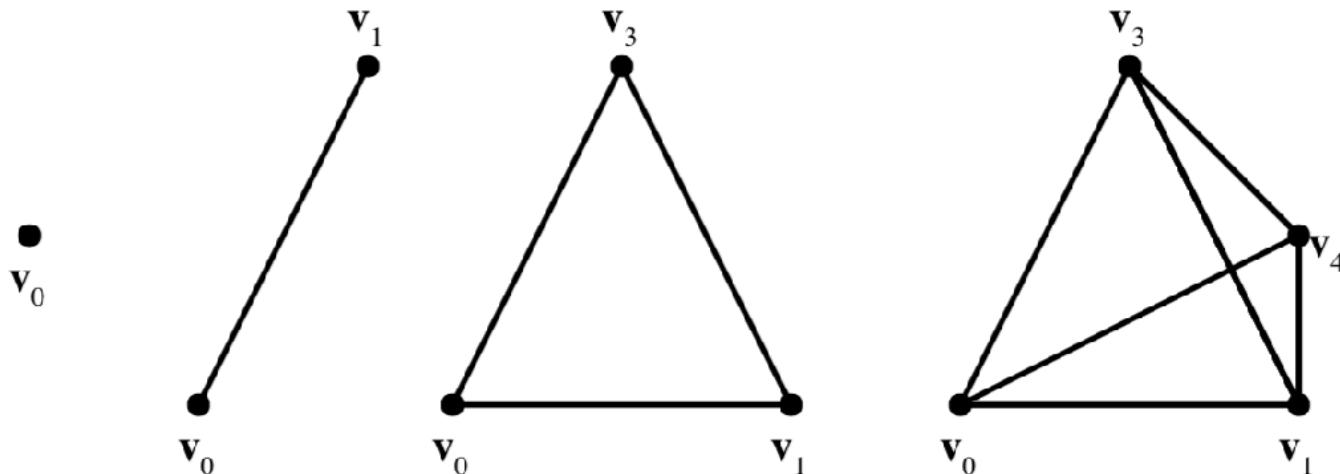
Representing Shapes

Different Representations

- Different representations are sometimes needed
- Optimize for different shape “types”
- Optimize for different algorithms, e.g. rendering vs simulation
- Support intuitive and efficient editing by artists
- But this implies significant software complexity
 - cannot mix-and-match algorithms
 - hard to exchange shapes
- Trend toward using a fewer representations

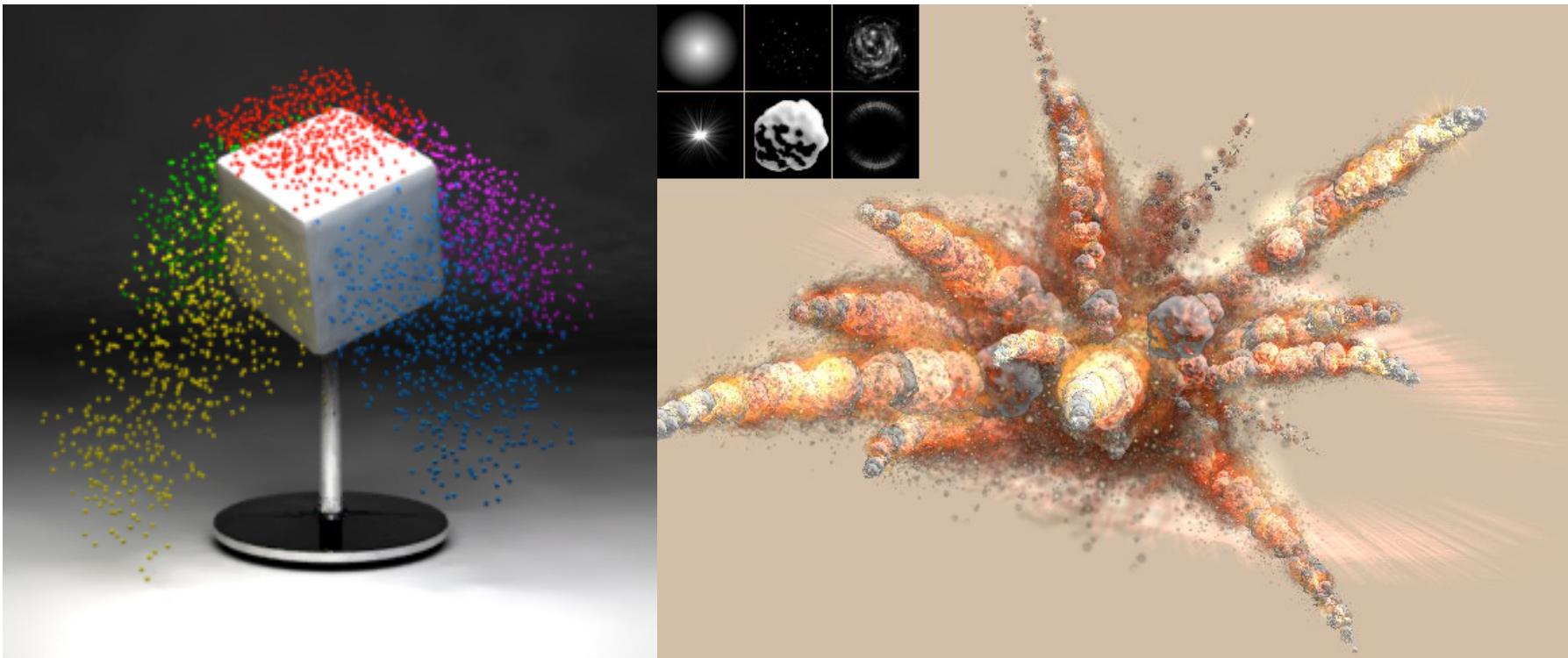
Points, Lines and Triangles

- For many algorithms, we can describe the world as collections of points, lines and triangles
- For simulation, it is sometimes helpful to use tetrahedra to represent the objects' volumes



Points

- Defined by one vertex
- Small enough that their properties do not vary over their “surface”
- Used for small particles (like ember, dust, etc) and simulation



Line segments

- Defined by two vertices
- Properties vary along the segment
- Used to represent hair and 2D outlines



[Blender]

Line segments

- Line tangent and length defined as

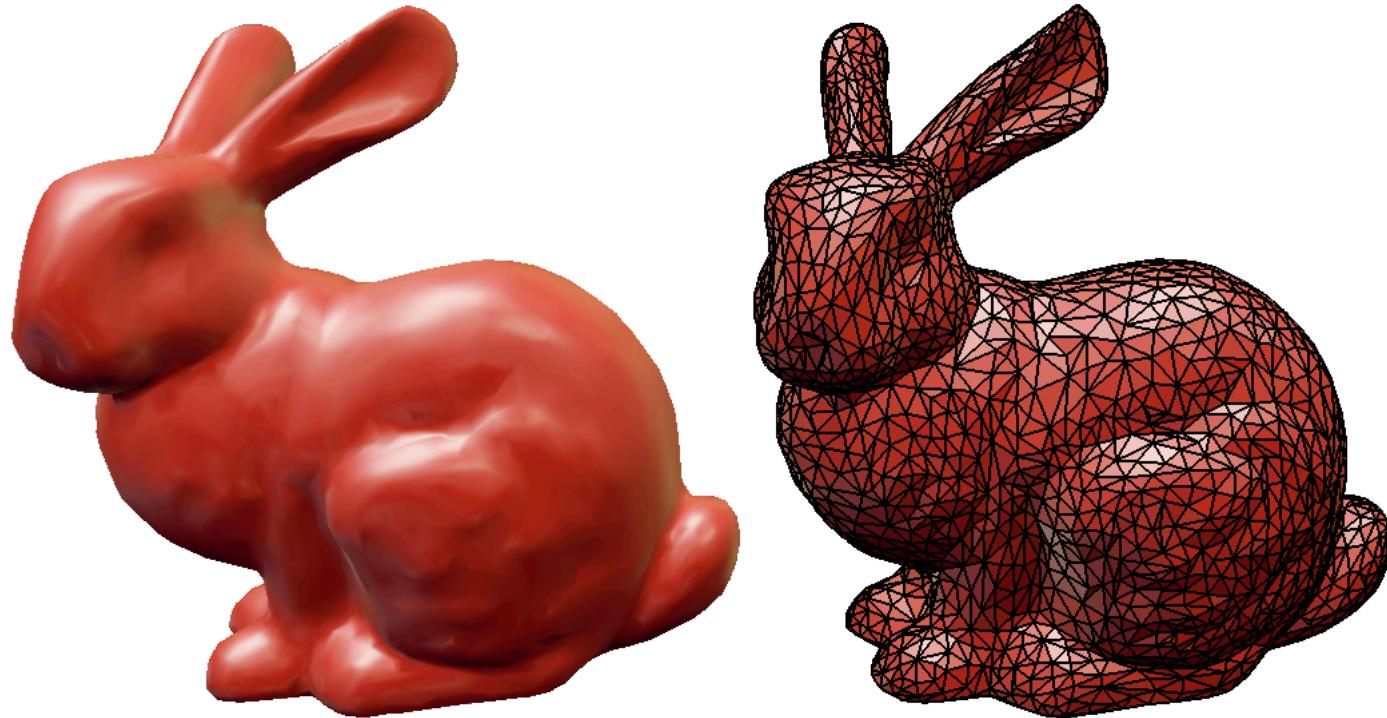
$$\mathbf{t} = \frac{\mathbf{v}_1 - \mathbf{v}_0}{|\mathbf{v}_1 - \mathbf{v}_0|}$$
$$l = |\mathbf{v}_1 - \mathbf{v}_0|$$

- Points on a segment can be written as linear combination between the vertices

$$\mathbf{p}(t) = (1 - t)\mathbf{v}_0 + t\mathbf{v}_1 \quad t \in [0, 1]$$

Triangles

- Defined by three vertices
- Flat shape contained in the plane passing through those vertices
- Used to represent surfaces

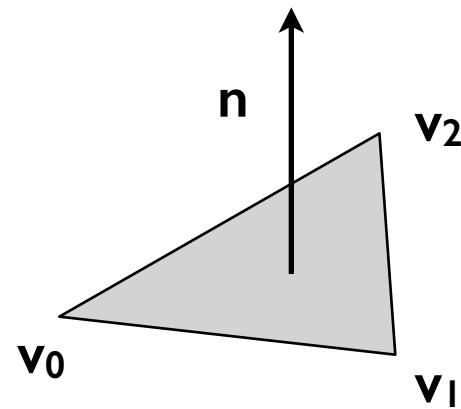


Triangles

- Vector normal to the plane is the triangle's normal, area computed from cross product formulation
- Convention on orientation: vertices are counter-clockwise as seen from “front”, surface normal points “outward”

$$\mathbf{n} = \frac{(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)}{|(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)|}$$

$$A = \frac{|(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)|}{2}$$



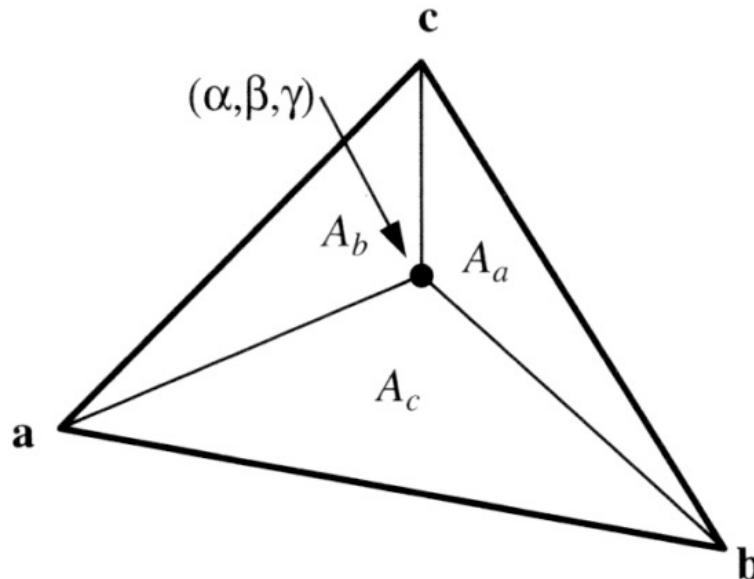
Barycentric coordinates

- Points on a triangles can be written as linear combination of the vertices — the weights are called *barycentric coordinates*

$$\mathbf{p} = w_0 \mathbf{v}_0 + w_1 \mathbf{v}_1 + w_2 \mathbf{v}_2$$

$$w_0 + w_1 + w_2 = 1$$

$$w_0 \geq 0 \quad w_1 \geq 0 \quad w_2 \geq 0$$



Barycentric coordinates

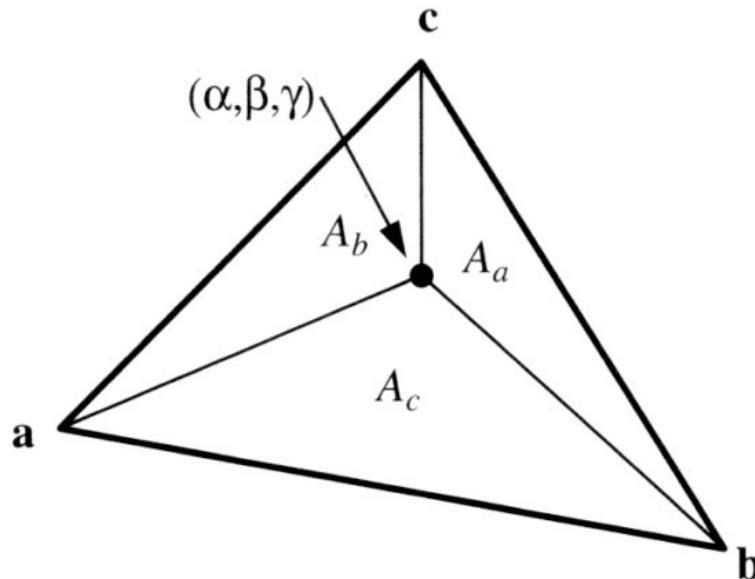
- Algebraic view: linear combination

$$w_0 = 1 - w_1 - w_2 \Rightarrow$$

$$\mathbf{p}(w_1, w_2) = \mathbf{v}_0 + w_1(\mathbf{v}_1 - \mathbf{v}_0) + w_2(\mathbf{v}_2 - \mathbf{v}_0)$$

- Geometric view: relative areas

$$w_0 = A_0/A \quad w_1 = A_1/A \quad w_2 = A_2/A$$

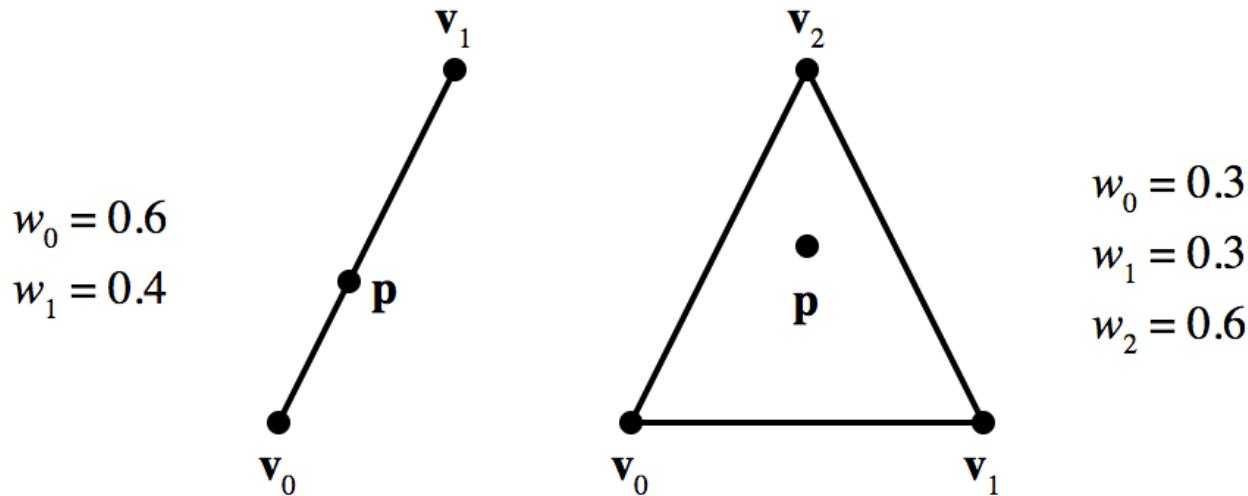


[Shirley 2000]

Barycentric coordinates

- General concept for points, lines, triangles, tetrahedra
- Can write points as linear combination of vertices

$$\mathbf{p} = \sum_{i=0}^{n-1} w_i \mathbf{v}_i \quad \sum_{I=0}^{n-1} w_i = 1 \quad w_i \geq 0$$

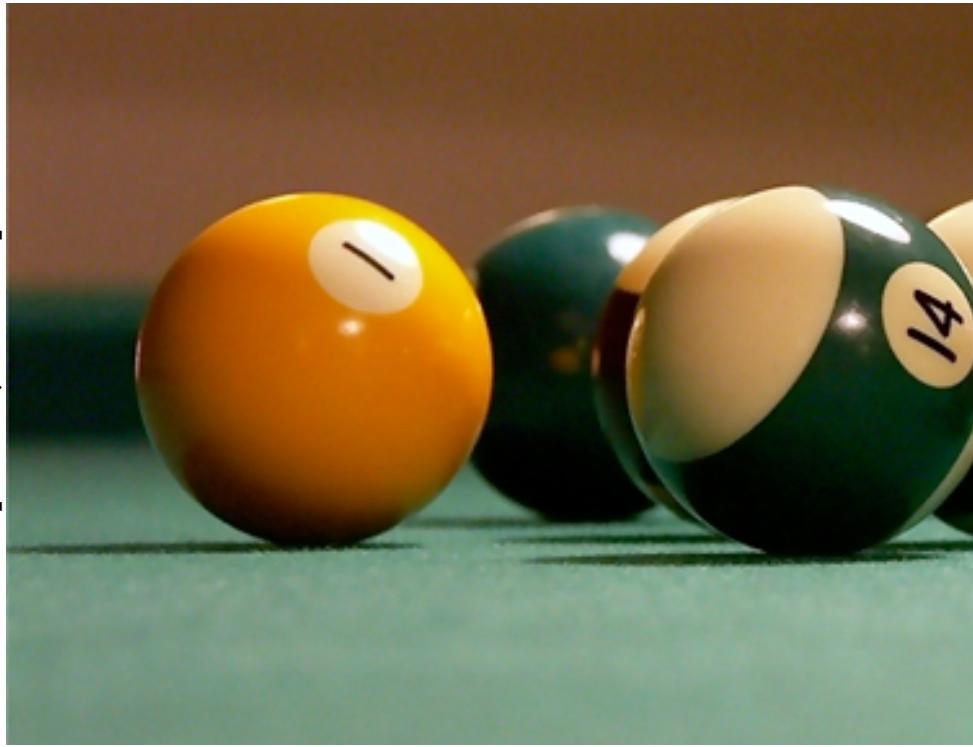


Triangle Meshes

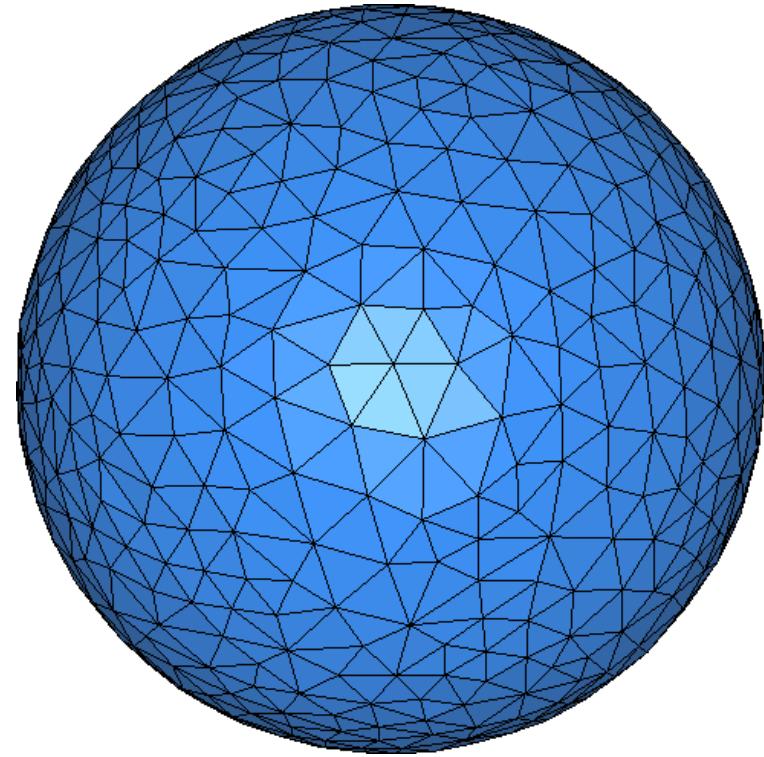
Triangle meshes

- A set triangles in 3D space connected together to form a surface
- Most common shape used in 3D graphics applications
- Geometrically, a mesh is a piecewise planar surface: it is planar except at the edges where triangles join
- Often, it's a piecewise planar approximation of a smooth surface
 - in this case the creases between triangles are artifacts—we don't want to see them

Approximate geometry



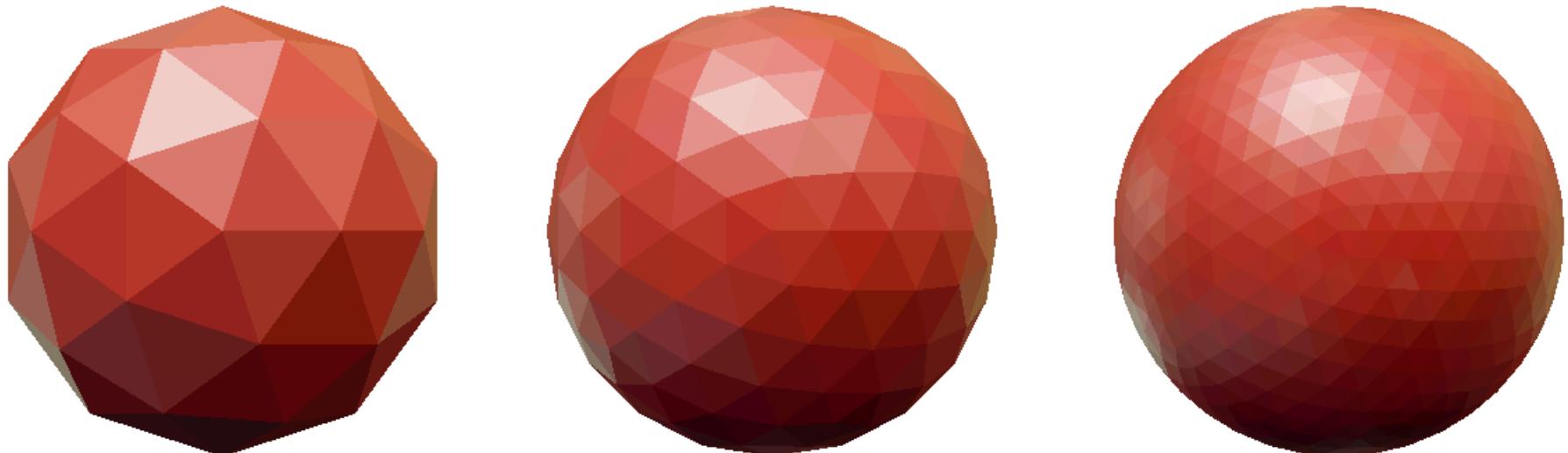
spheres



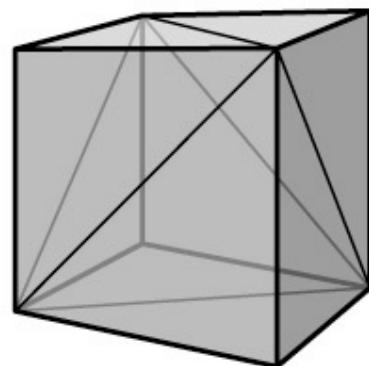
approximate
sphere

Approximate geometry

- More triangles, better approximation



A small triangle mesh



12 triangles, 8 vertices

A large mesh

10 million triangles
from a high-resolution
3D scan

[Traditional Thai sculpture—scan by XYZRGB, inc., image by MeshLab]









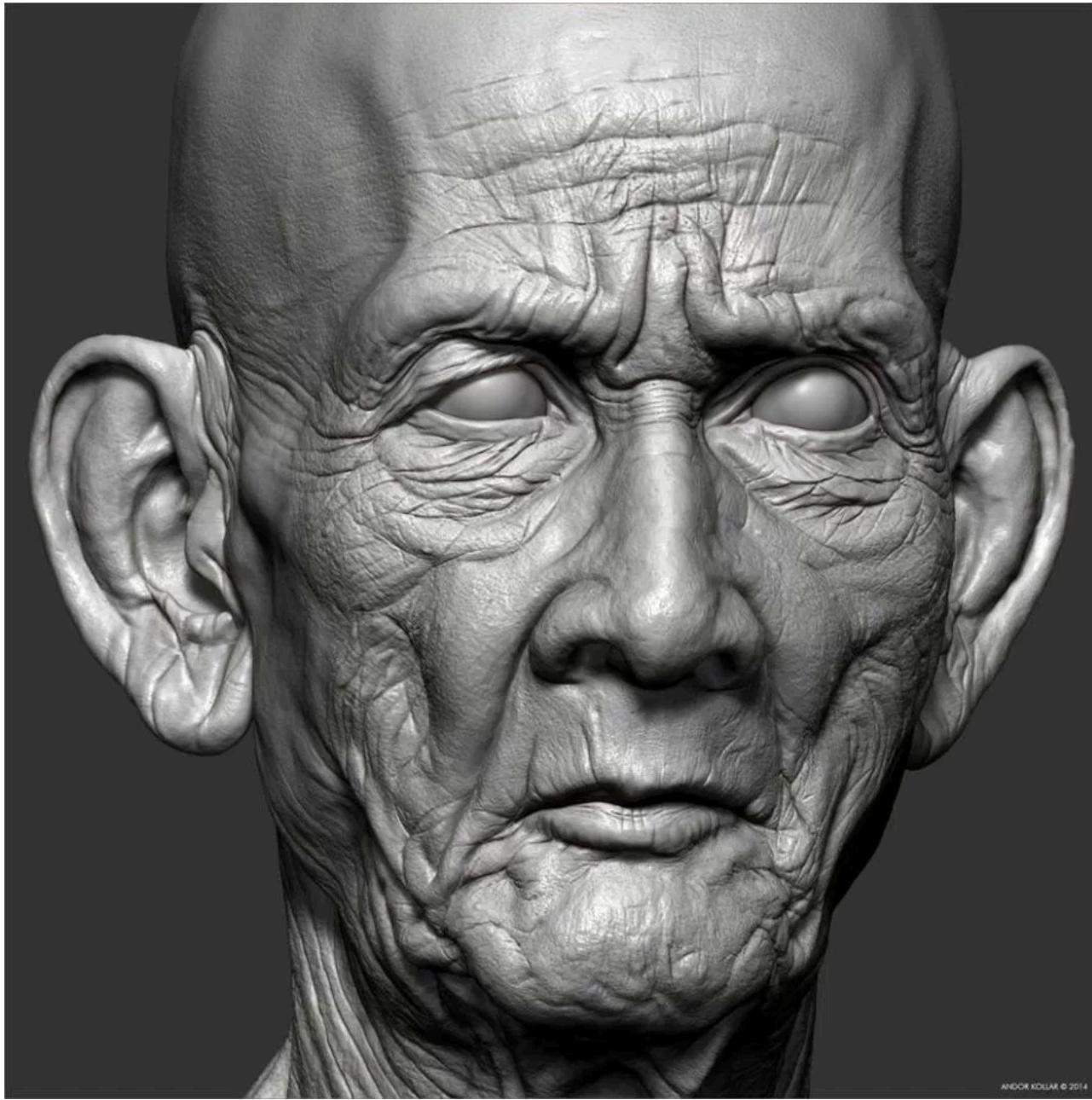
**about a trillion triangles
from automatically processed
satellite and aerial photography**

Google earth



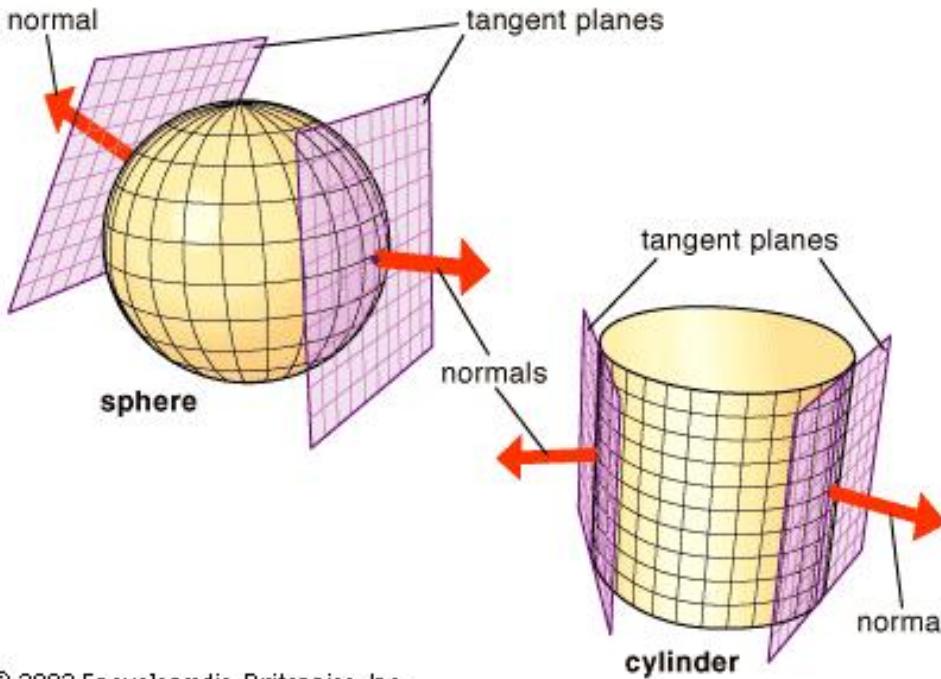
[Andor Kollar]

[Andor Kollar]



Normals and tangents

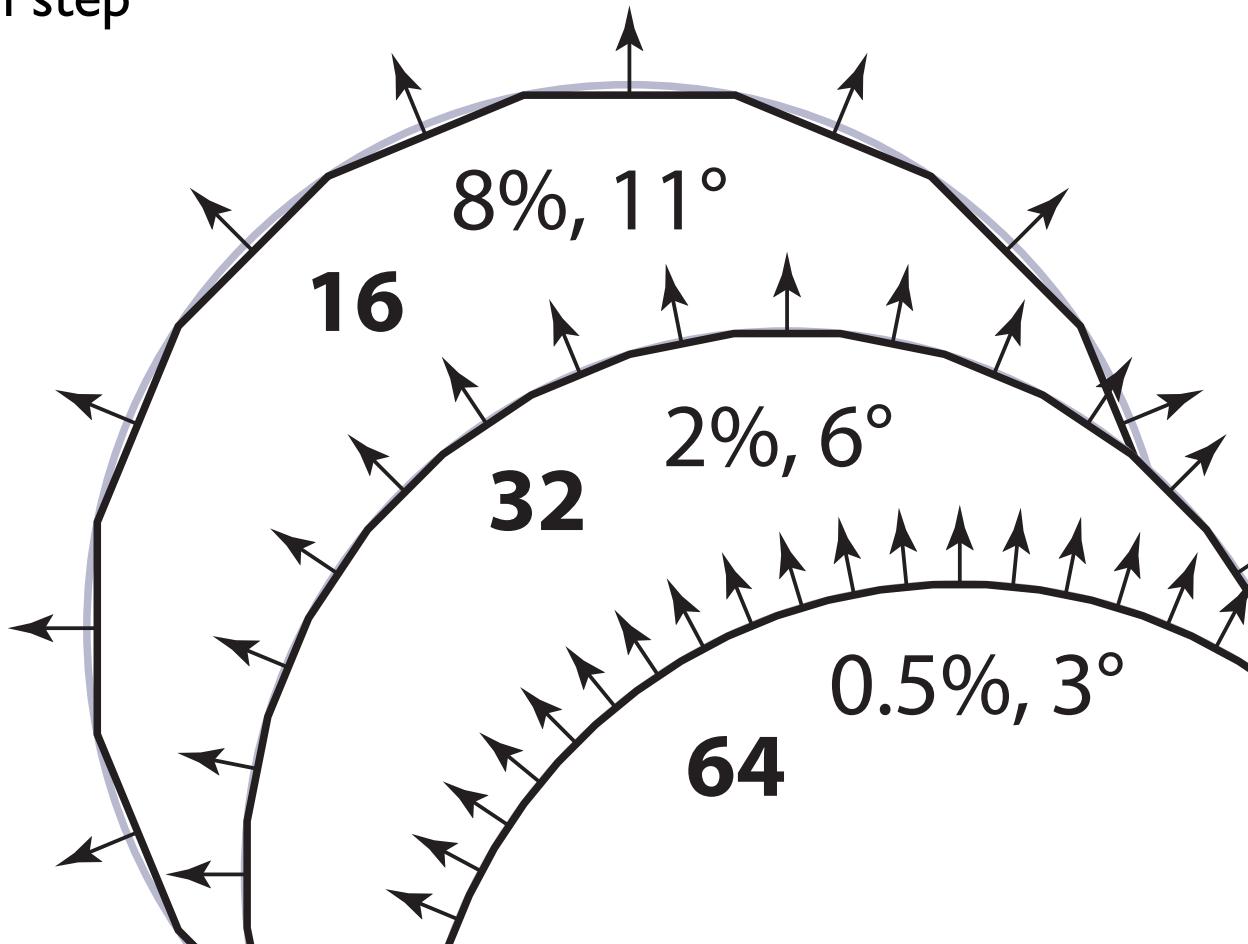
- Tangent plane: at a point on a smooth surface in 3D, there is a unique plane tangent to the surface, called the tangent plane
- Normal vector: vector perpendicular to the tangent plane
 - only unique for smooth surfaces (not at corners, edges)



© 2002 Encyclopædia Britannica, Inc.

Approximation quality

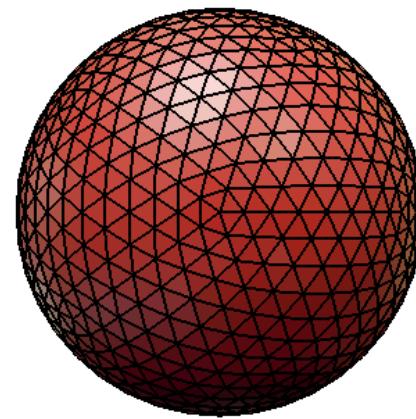
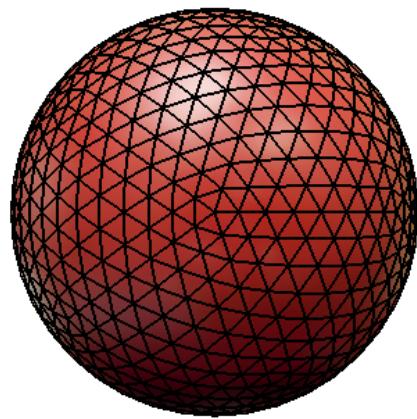
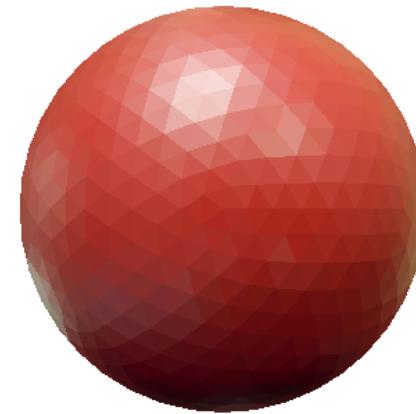
- Approximating circle with increasingly many segments
- Max error in position error drops by factor of 4 at each step
- Max error in normal only drops by factor of 2



Approximation quality

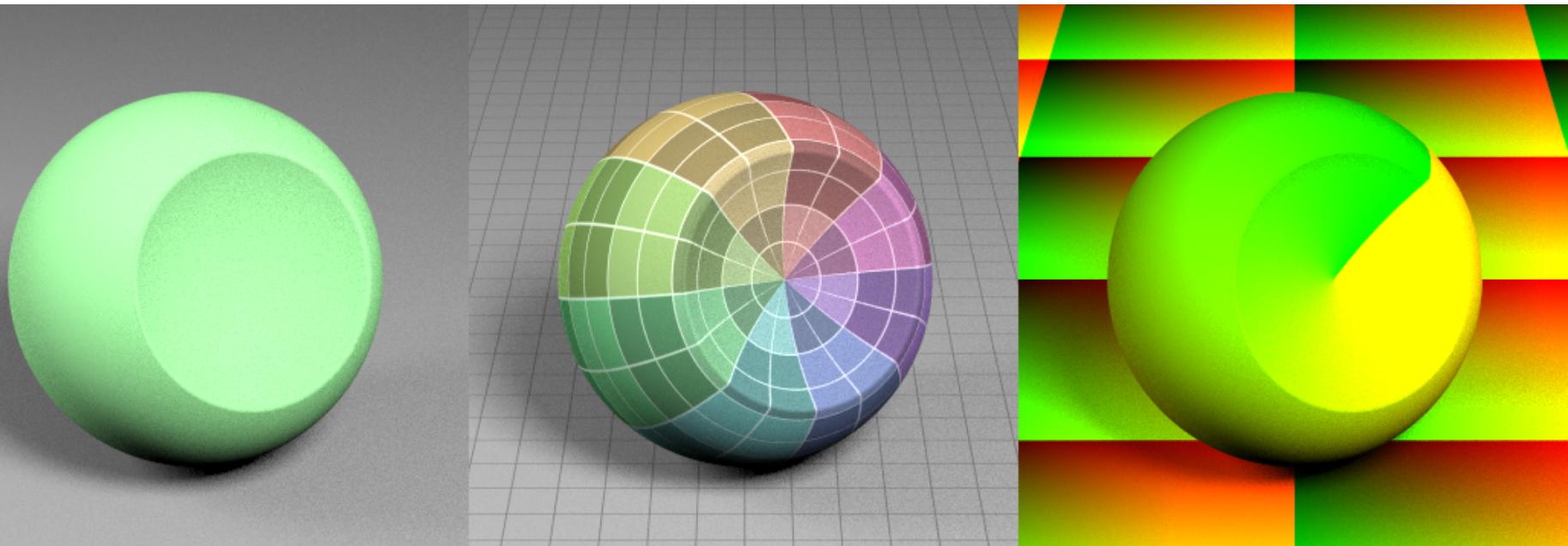
- Piecewise planar approximation converges pretty quickly to the smooth geometry as the number of triangles increases
 - for mathematicians: error is $O(h^2)$
- But the surface normals don't converge so well
 - normal is constant over each triangle, with discontinuous jumps across edges
 - for mathematicians: error is only $O(h)$
- Better: store the “real” normal at each vertex, and interpolate to get normals that vary gradually across triangles

Facet vs. vertex normals



Textures

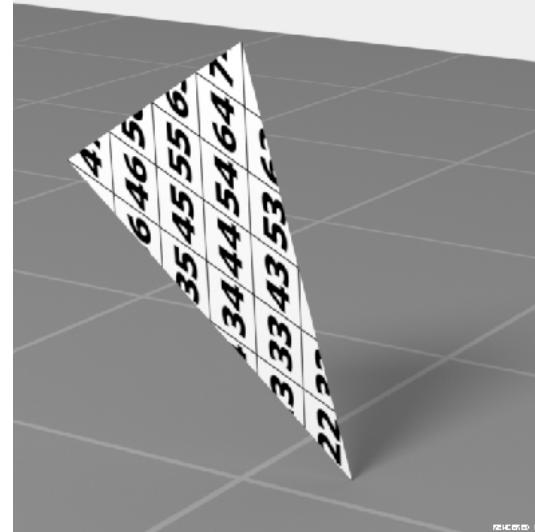
- Surface material parameters can be associated with each triangle
 - so we need ~ 1 triangle/pixel for full color resolution
- Idea: define materials by “pasting” images onto the surface
- We’ll discuss in more details later



Texture coordinates

- A surface in 3D is a two-dimensional entity
- Sometimes we need 2D coordinates for points on the surface
- Defining these coordinates is *parameterizing* the surface
- For triangle meshes, specify 2D coordinates (in $[0,1]^2$) at the vertices and barycentric interpolate in the triangle

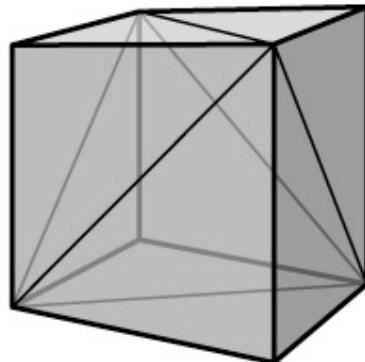
09	19	29	39	49	59	69	79	89	99
08	18	28	38	48	58	68	78	88	98
07	17	27	37	47	57	67	77	87	97
06	16	26	36	46	56	66	76	86	96
05	15	25	35	45	55	65	75	85	95
04	14	24	34	44	54	64	74	84	94
03	13	23	33	43	53	63	73	83	93
02	12	22	32	42	52	62	72	82	92
01	11	21	31	41	51	61	71	81	91
00	10	20	30	40	50	60	70	80	90



Mesh Representations

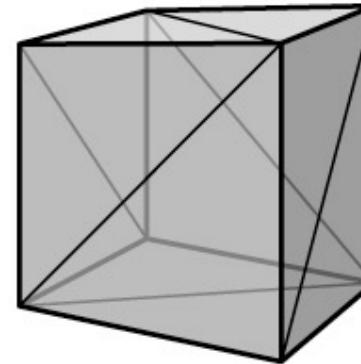
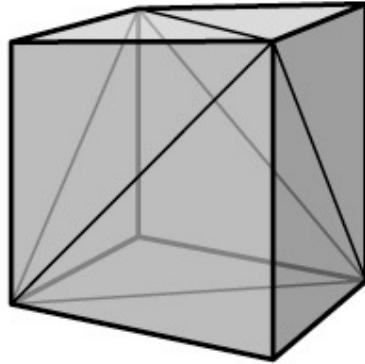
Topology vs. geometry

- Mesh geometry: where the triangles are in 3D space
- Mesh topology: how the triangles are connected
 - ignoring the positions entirely

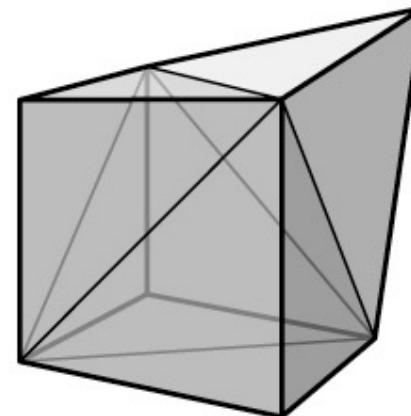
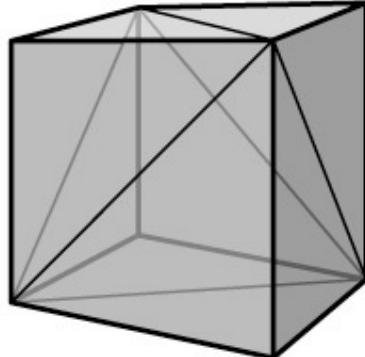


Topology vs. geometry

- same geometry, different mesh topology:



- same mesh topology, different geometry:



Mesh representations

- Compactness
- Efficiency for rendering
 - enumerate all triangles as triples of 3D points
- Efficiency of adjacency queries
 - all vertices of a triangle
 - all triangles around a vertex
 - neighboring triangles of a triangle
 - need depends on application (rendering, subdivs, modeling)

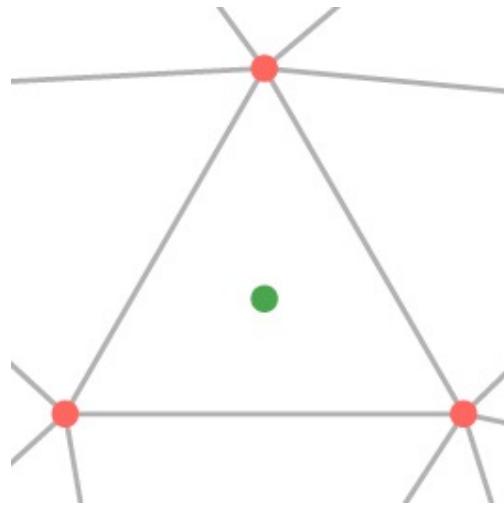
Mesh representations

- Separate triangles
- Indexed triangle set
 - shared vertices
- Triangle strips and triangle fans
 - compression schemes for fast transmission
- Triangle-neighbor data structure
 - supports adjacency queries
- Half-edge data structure
 - supports general polygon meshes

Separate triangles

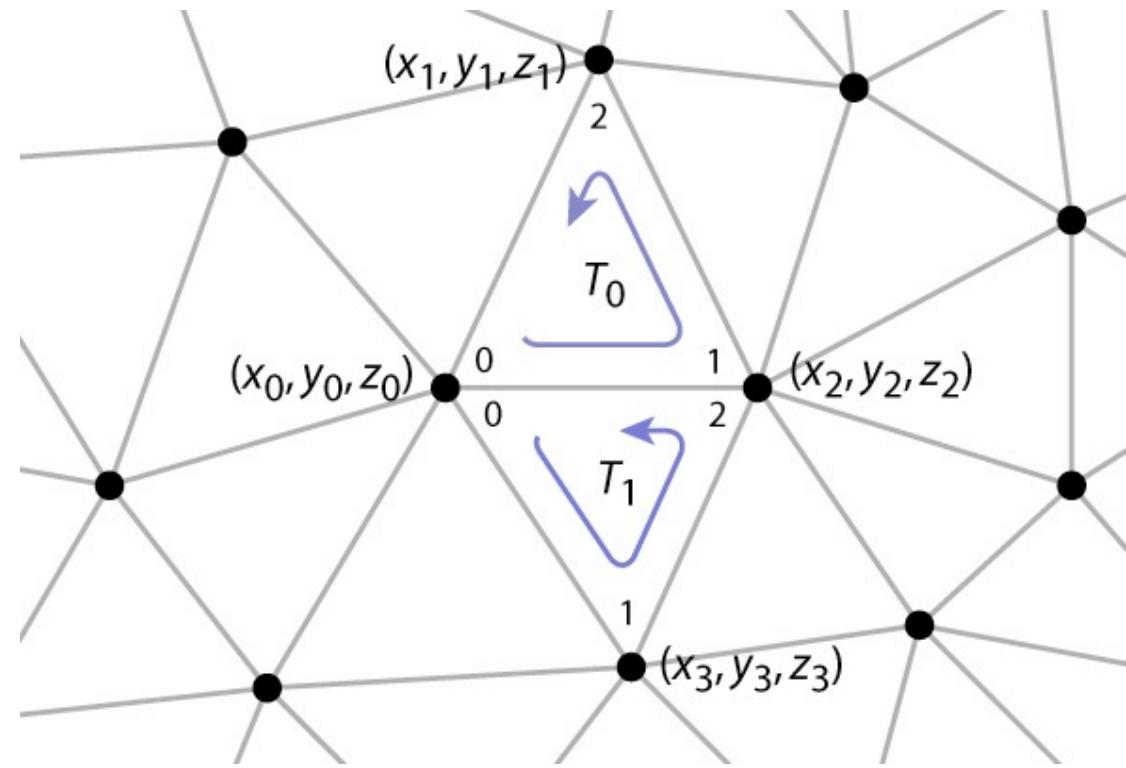
- Store each triangle independently

```
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
	\vdots	\vdots	\vdots



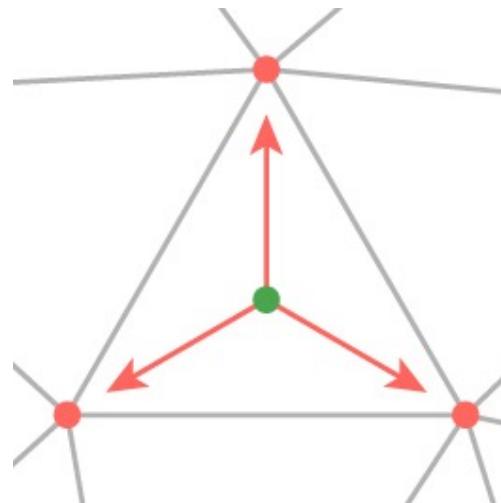
Separate triangles

- 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

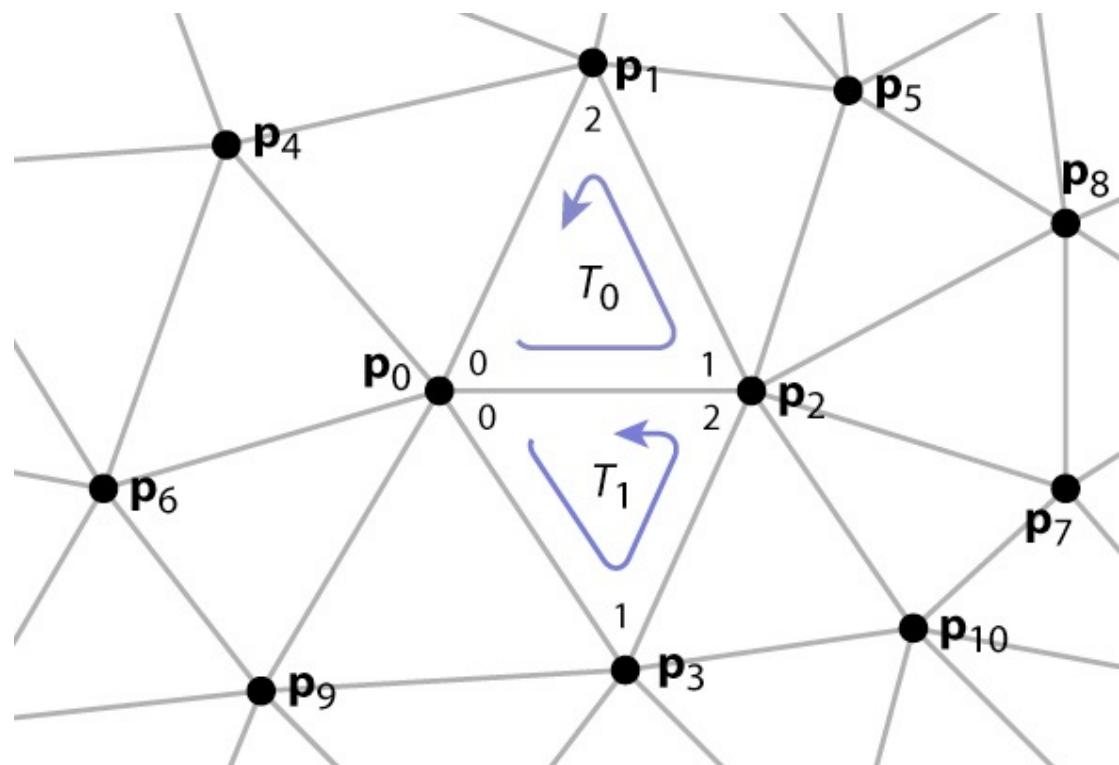
```
struct mesh {  
    vec3i tris[nT];  
    vec3f pos[nV];  
};
```



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
:	



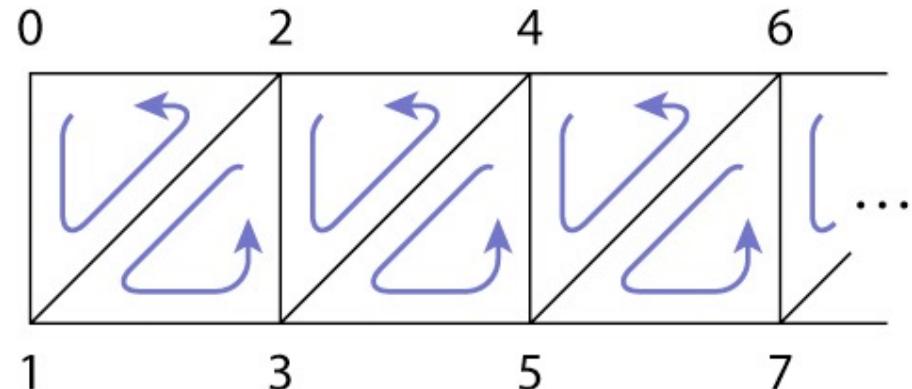
Indexed triangles

- 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

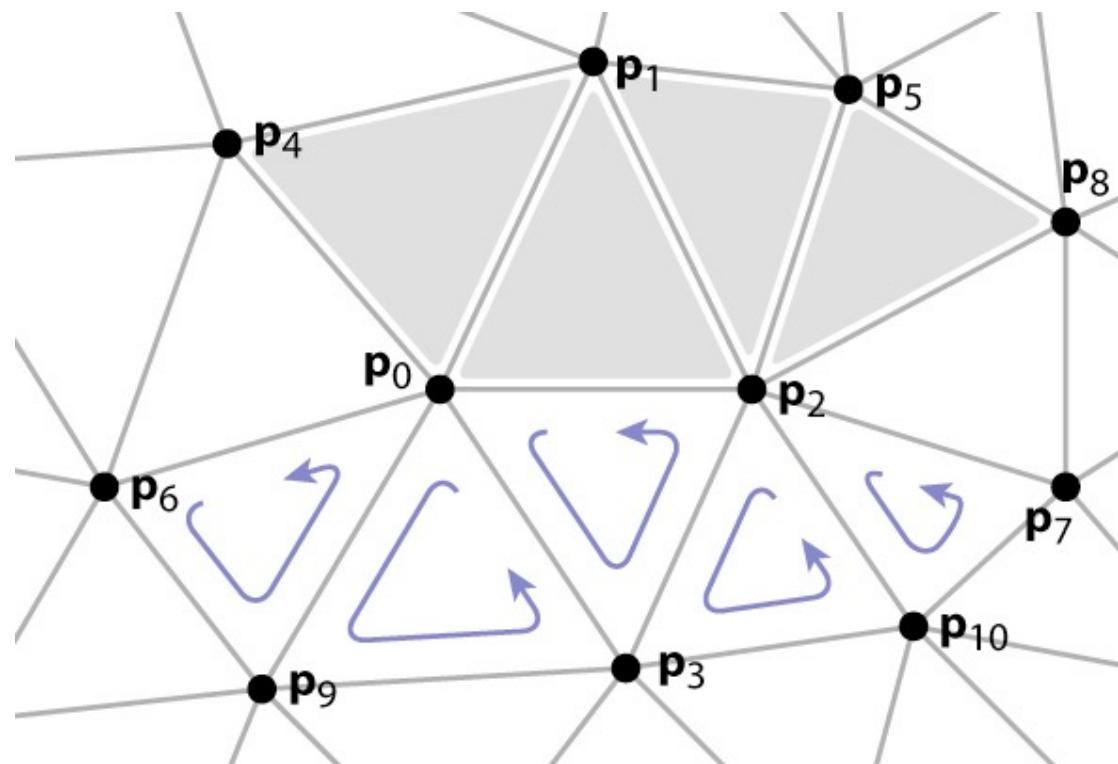
```
struct mesh {  
    int strips[nT][];  
    float pos[nV][3];  
};
```



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
	\vdots

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



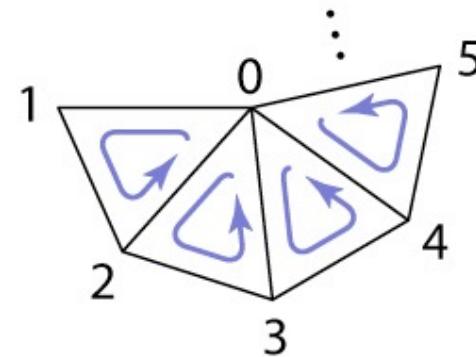
Triangle strips

- Array of vertex positions
 - float[n_v][3]: 12 bytes per vertex
- Array of index lists
 - int[n_s][variable]: (2 + n)×4 bytes per strip
 - on average, (l + 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];  
};
```

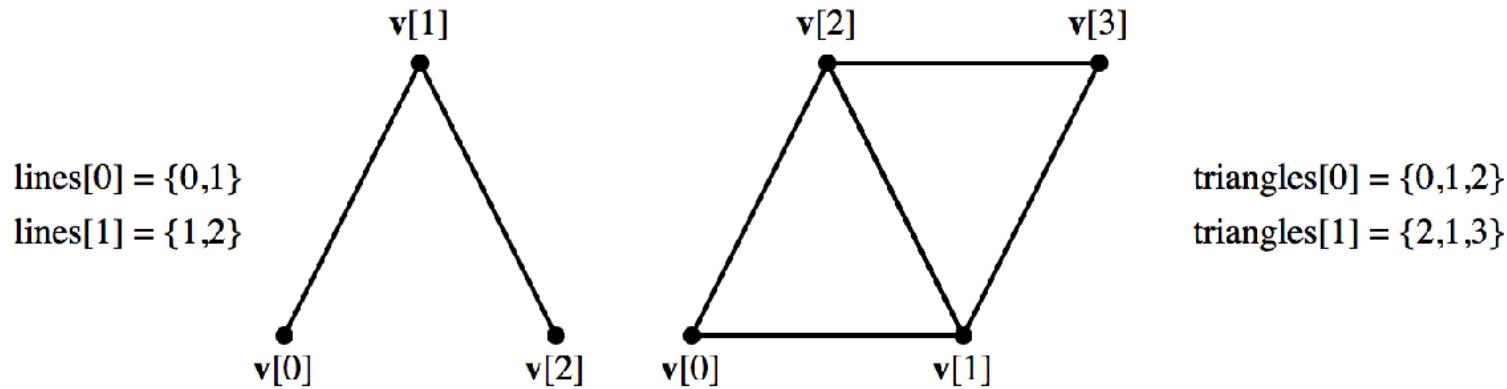


Data on meshes

- 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

Indexed Mesh Data Structure

- Indexed triangles is the most used representation
 - efficient for rendering and many geometry operations
 - can easily add/remove triangles and vertices
- For GPU execution: store data on vertices
- In modeling applications: store data on vertices and faces
- Indexed representation used also for segments and tetrahedra



Key types of vertex data

- Positions
 - at some level this is just another piece of data
 - position varies continuously between vertices
- Surface normals
 - when a mesh is approximating a curved surface, store normals at vertices
- Texture coordinates
 - 2D coordinates that tell you how to paste images on the surface

Indexed Mesh Data Structure

- A typical implementation uses separate arrays for each vertex data
 - this allows to only store data if present
- Element indices are held in either an integer array or a typed array (as here) for easier accessing

```
struct shape {  
    vector<vec3f> pos;          // vertex positions  
    vector<vec3f> norm;         // vertex normals  
    vector<vec2f> texcoord;     // vertex texture coordinates  
  
    vector<int> points;         // point indices  
    vector<vec2i> lines;         // line indices  
    vector<vec3i> triangles;    // triangle indices  
}
```

Computing vertex normals

- Ideally, store normals from the “true” geometry
- Practically, “true” geometry is not available most of the time, e.g. in hand modeling or 3D scanning
- Compute the normal \mathbf{n}_i of vertex i as the average the normals \mathbf{n}_f of the triangles f that are adjacent to i , weighted by the triangles areas A_f

$$\mathbf{n}_i = \frac{\sum_{f \in N_i} A_f \mathbf{n}_f}{\sum_{f \in N_i} A_f}$$

- Implement this by looping over faces and updating vertex normals, which is $O(n)$, instead of a trivial implementation of the equation above which is $O(n^2)$

Computing vertex normals

```
vector<vec3f> compute_normals(const vector<vec3f>& positions,
                               const vector<vec3i>& triangles) {

    // initialize normals to 0
    auto normals = vector<vec3f>(pos.size(), {0,0,0});
    // accumulate triangle normals
    for(auto t : triangles) {
        auto n = triangle_normal(positions[t.x],
                                  positions[t.y], positions[t.z]);
        auto a = triangle_area(positions[t.x],
                               positions[t.y], positions[t.z]);
        normals[t.x] += a * n; normals[t.y] += a * n;
        normals[t.z] += a * n;
    }
    // normalize result
    for(auto& n : norm) n = normalize(n);
    return norm;
}
```

Interpolating vertex data

- Interpolation of vertex data follows barycentric coordinates
 - for normals we normalize at the end

$$\mathbf{p} = w_0 \mathbf{p}_0 + w_1 \mathbf{p}_1 + w_2 \mathbf{p}_2 \quad \mathbf{n} = \frac{w_0 \mathbf{n}_0 + w_1 \mathbf{n}_1 + w_2 \mathbf{n}_2}{|w_0 \mathbf{n}_0 + w_1 \mathbf{n}_1 + w_2 \mathbf{n}_2|}$$

```
vec3f normal(shape* shape, int element, vec2f uv) {
    auto t = shape->triangles[element];
    return normalize((1 - uv.x - uv.y) * shape->normals[t.x] +
                    uv.x * shape->normals[t.y] +
                    uv.y * shape->normals[t.z]);
}

vec3f position(shape* shape, int element, vec3f uv) {
    auto t = shape->triangles[element];
    return (1 - uv.x - uv.y) * shape->positions[t.x] +
           uv.x * shape->positions[t.y] +
           uv.y * shape->positions[t.z]);
}
```

Mesh Validity

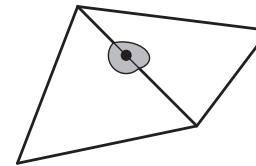
Validity of triangle meshes

- In many cases we care about the mesh being able to bound a region of space nicely
- In other cases we want triangle meshes to fulfill assumptions of algorithms that will operate on them (and may fail on malformed input)
- Two completely separate issues:
 - topology: how the triangles are connected
 - geometry: where the triangles are in 3D space

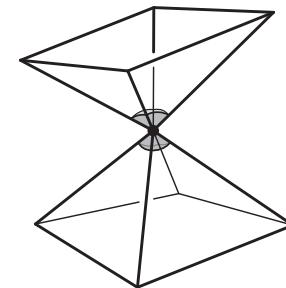
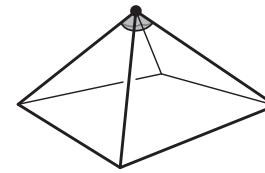
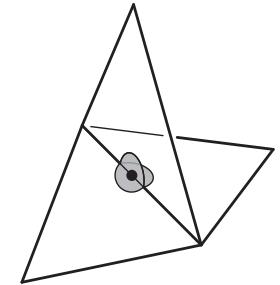
Topological validity

- Strongest property: be a manifold
 - this means that no points should be "special"
 - interior points are fine
 - edge points: each edge must have exactly 2 triangles
 - vertex points: each vertex must have one loop of triangles
- Slightly looser: manifold with boundary
 - weaken rules to allow boundaries

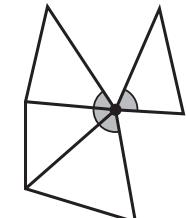
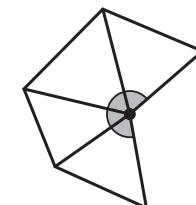
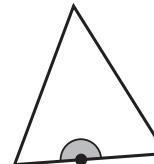
manifold



not manifold

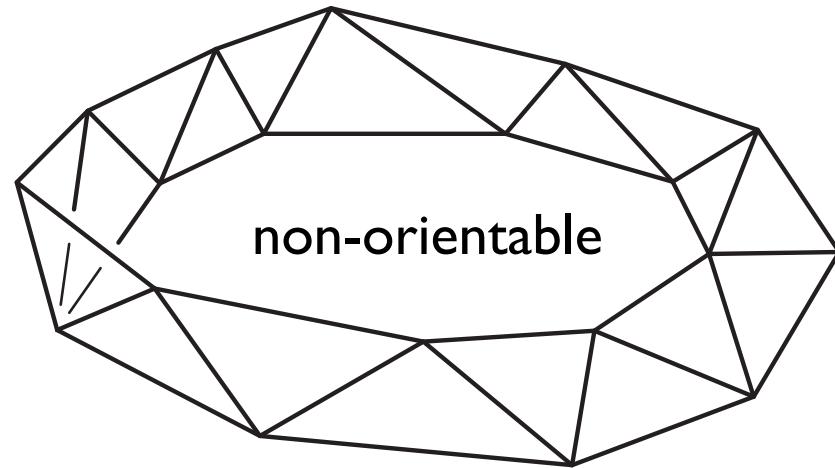
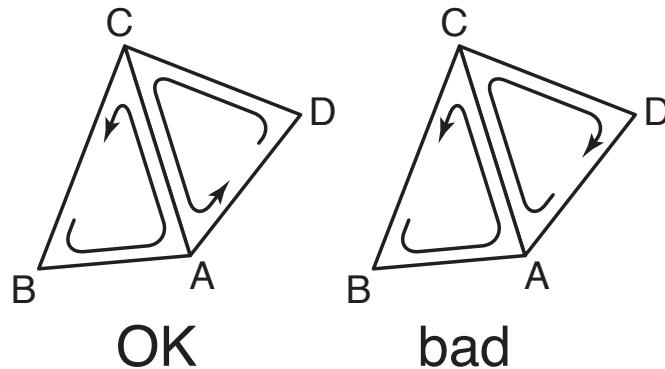


with boundary



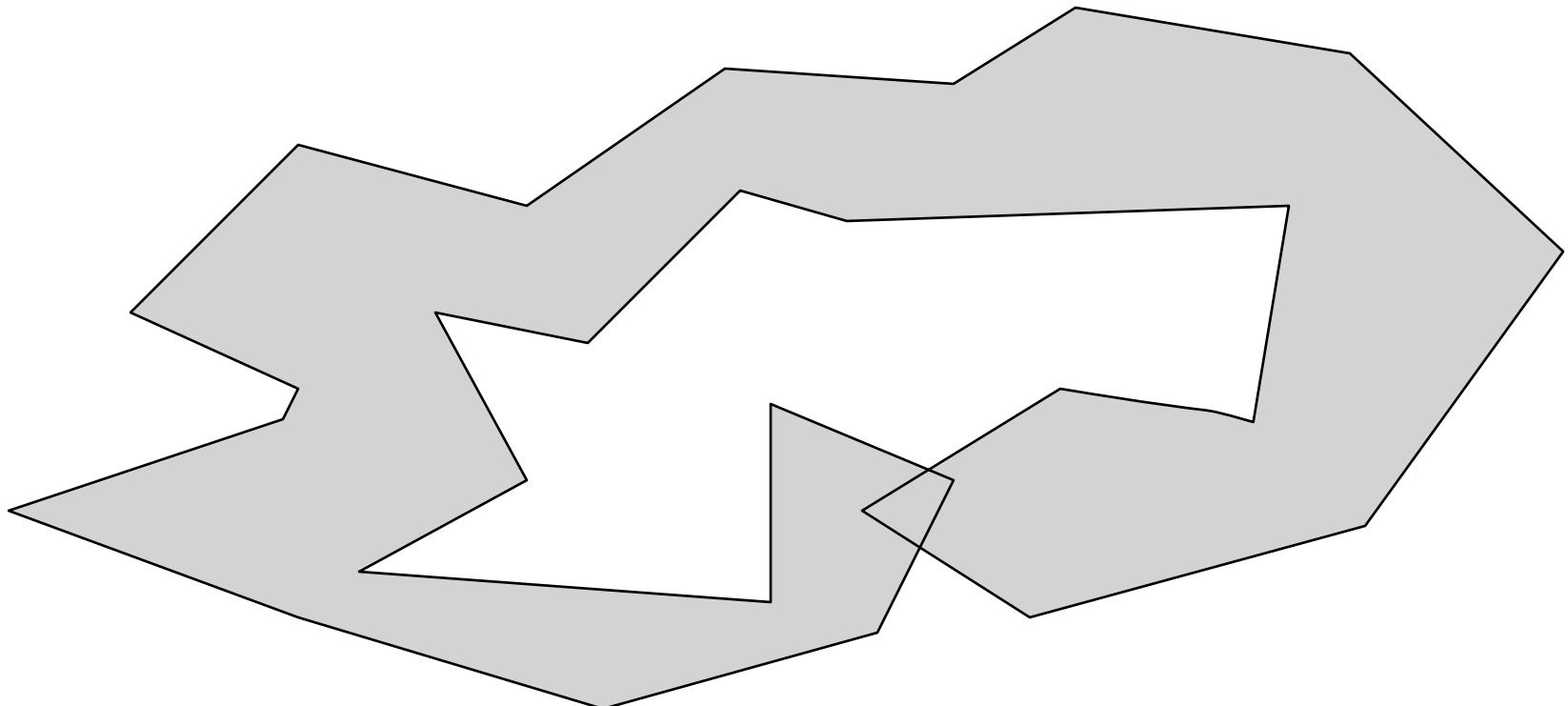
Topological validity

- Consistent orientation
 - which side is the “front” or “outside” of the surface and which is the “back” or “inside?”
 - rule: you are on the outside when you see the vertices in counter-clockwise order
 - in mesh, neighboring triangles should agree about which side is the front!
 - caution: not always possible



Geometric validity

- Generally want non-self-intersecting surface
- Hard to guarantee in general
 - because far-apart parts of mesh might intersect



Representing Materials



Brooklyn NYC Loft by Yarko Kushta | www.behance.net/Yarkokushta

 corona

rendered with Corona Renderer | www.corona-renderer.com

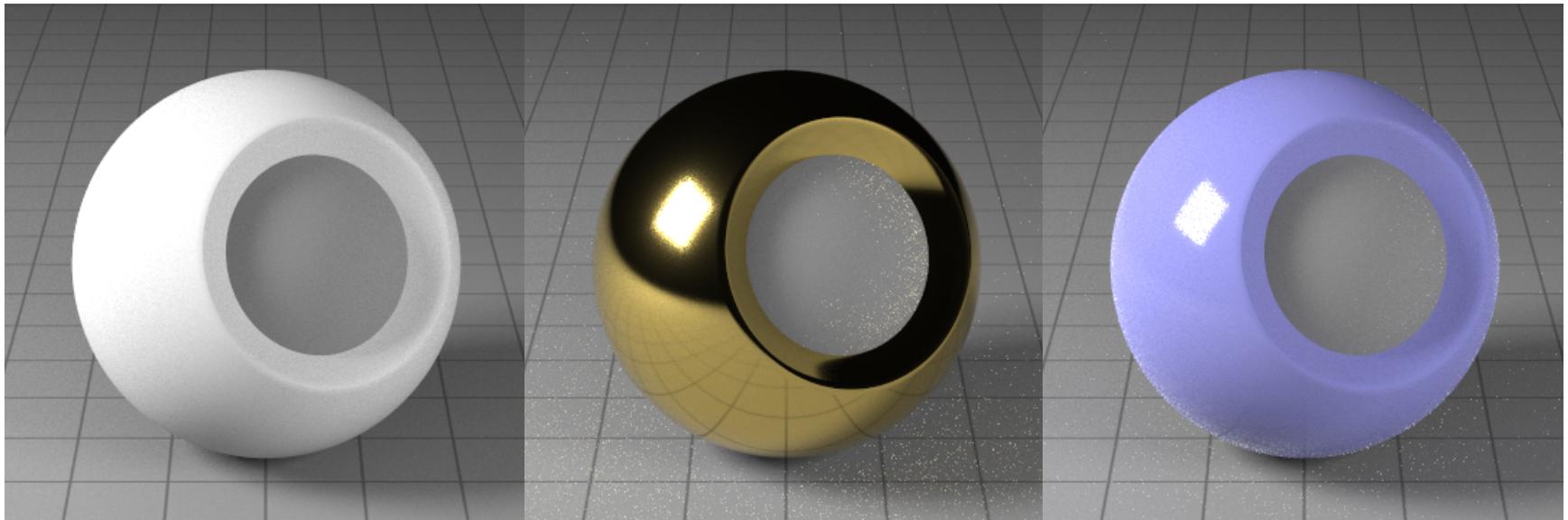


[Activision]

Activision R&D. Property of Activision Publishing. N

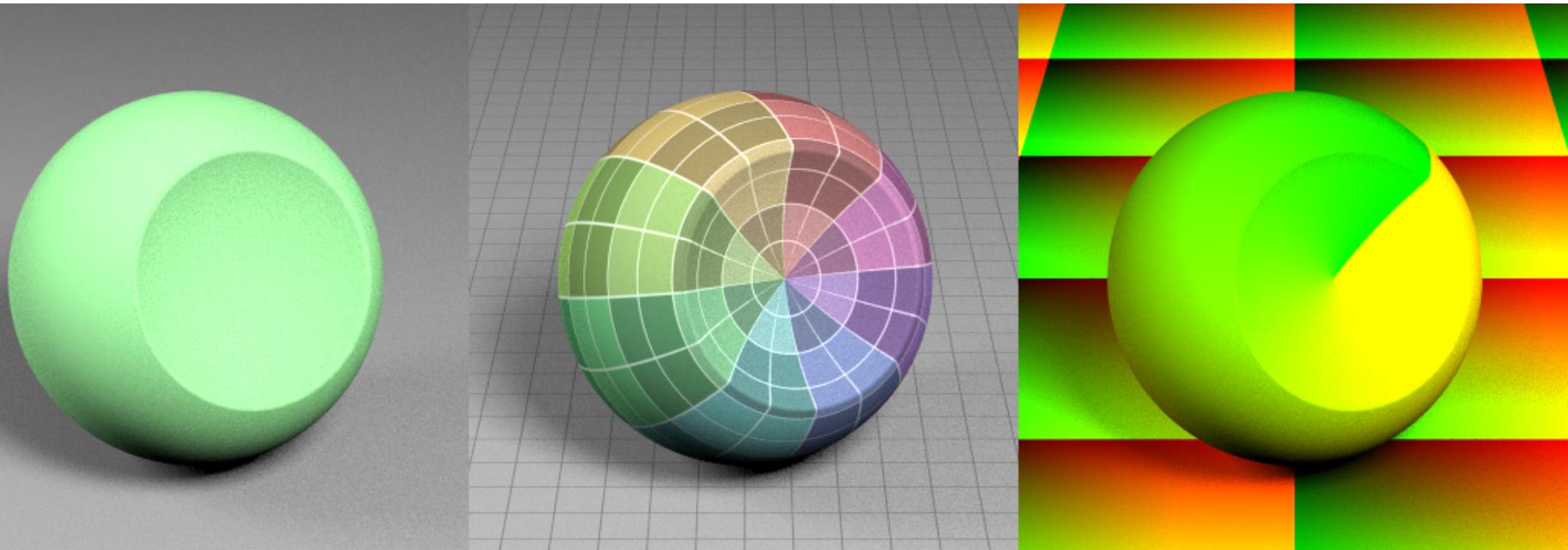
Representing Materials

- Start with a simple model that covers plastics and metals
 - augment the model later for other material types
- Diffuse color: controls color for plastics
- Specular color : controls color for metals
- Specular roughness: controls highlight/reflection size and blur



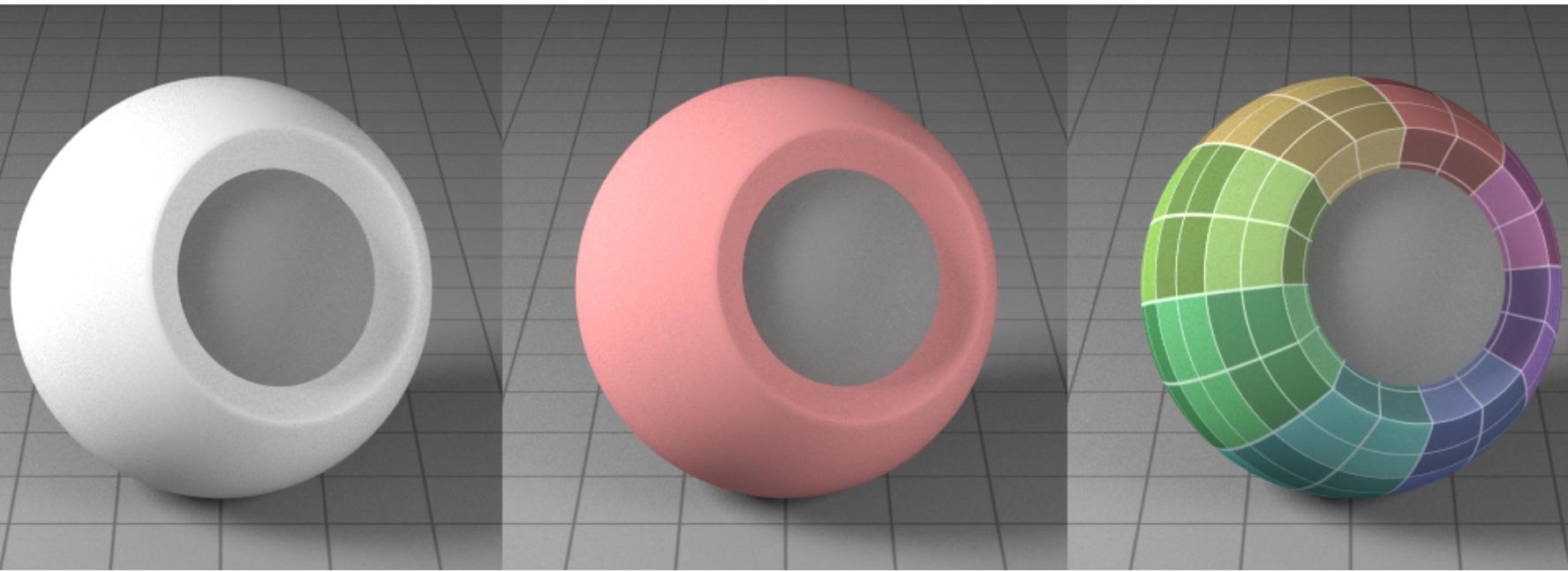
Representing Materials

- Common to specify parameters both as constant and textures
 - take the product of the two



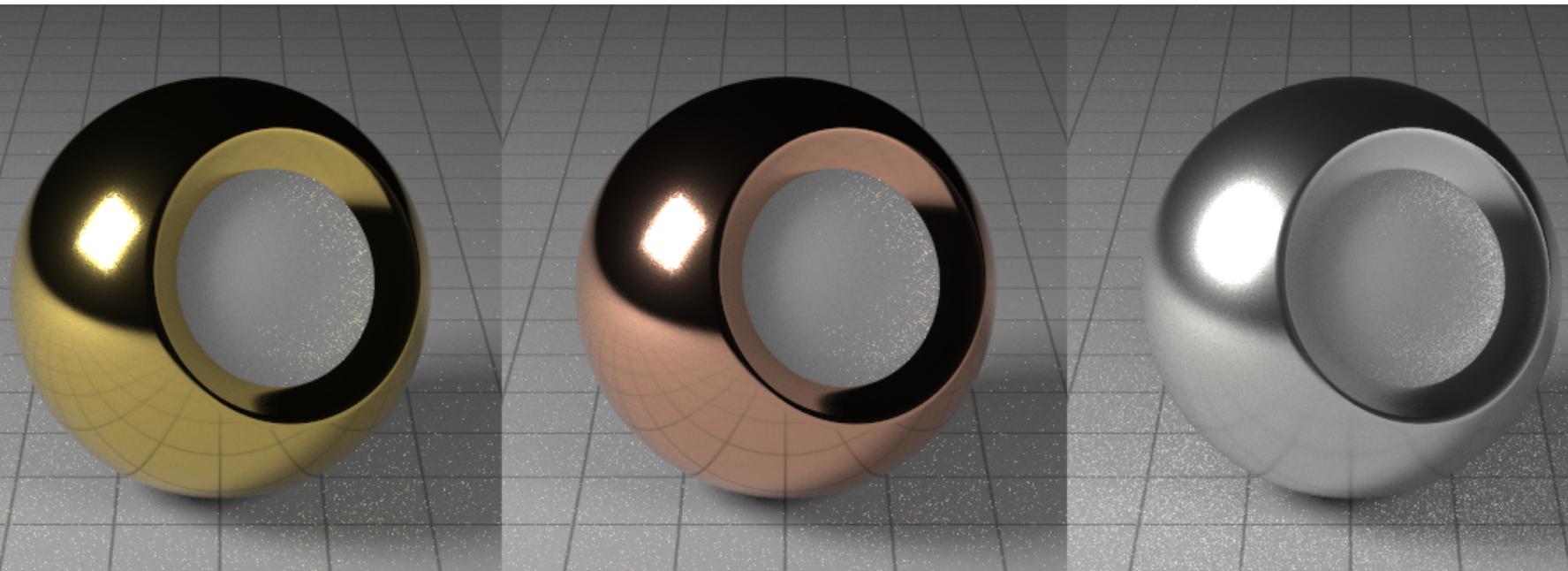
Matte surfaces

- Change color by changing the diffuse coefficient
- Apply textures to specify variations



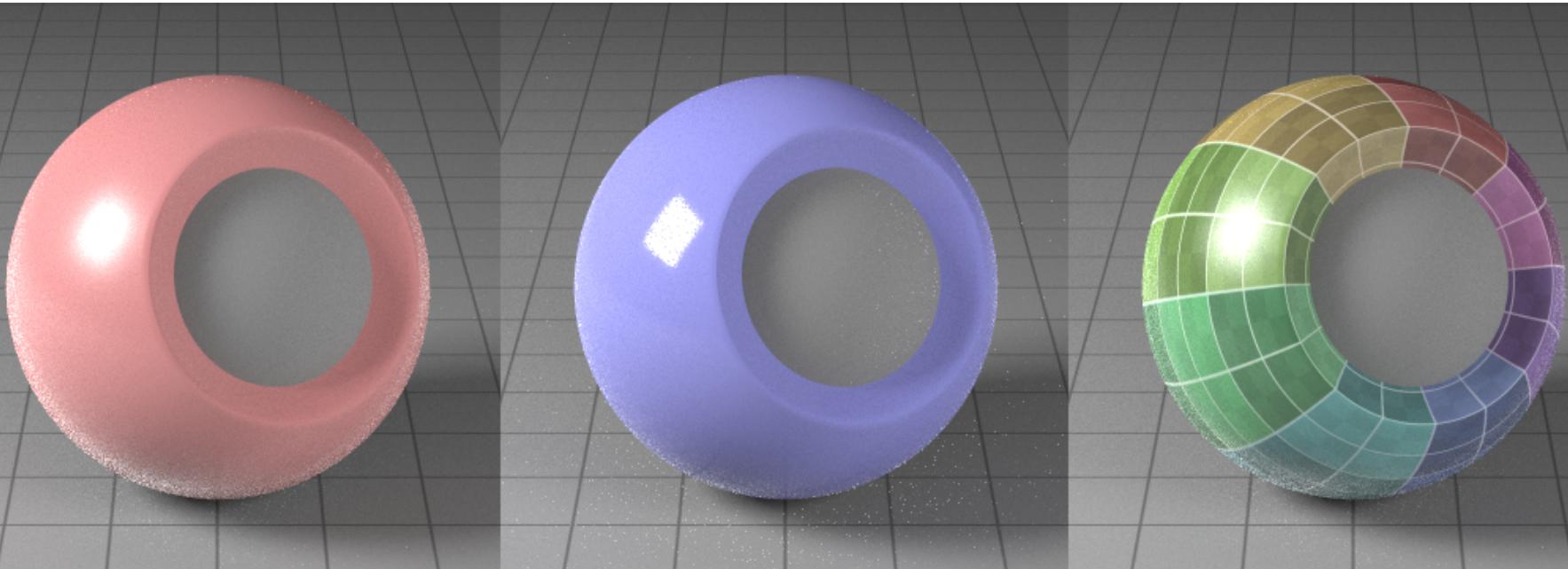
Metal surfaces

- Change color by changing the specular coefficient
- Change reflection blur by changing roughness



Plastic surfaces

- Change color by changing diffuse coefficient
- Change highlights by changing roughness
 - set highlight color to a small value



Representing Materials

- Legacy materials stored coefficients directly
- Store textures as pointers, so they can be shared between materials

```
struct texture {
    image<vec3f> hdr; // either hdr, e.g. for emission
    image<vec3b> ldr; // or ldr, e.g for colors
};

struct legacy_material {
    vec3f emission;           // emission
    vec3f diffuse;            // diffuse
    vec3f specular;           // specular
    float roughness;          // roughness
    texture* emission_tex;    // emission texture
    texture* diffuse_tex;     // diffuse texture
    texture* specular_tex;    // specular texture
    texture* roughness_tex;   // roughness texture
};
```

Representing Materials

- Current trends use one main color and weights to define behaviors

```
struct texture {  
    image<vec3f> colorf; // for color values  
    image<vec3b> colorb;  
    image<float> scalarf; // for scalar values  
    image<byte> scalarb;  
};  
  
struct material {  
    vec3f emission;           // emission  
    vec3f color;              // color  
    float metallic;           // metallic  
    float roughness;          // roughness  
    texture* diffuse_tex;     // diffuse texture  
    texture* metallic_tex;    // metallic texture  
    texture* roughness_tex;   // roughness texture  
};
```

Representing Lights

- No need for special objects: lights are just surfaces that emit light
- Stored as emission in the material

```
struct texture {
    image<vec3f> colorf; // for color values
    image<vec3b> colorb;
    image<float> scalarf; // for scalar values
    image<vec3b> scalarb;
};

struct material {
    vec3f emission;           // emission
    vec3f color;              // color
    float metallic;           // metallic
    float roughness;          // roughness
    texture* emission_tex;    // emission texture
    texture* diffuse_tex;     // diffuse texture
    texture* metallic_tex;    // metallic texture
    texture* roughness_tex;   // roughness texture
};
```

Textures

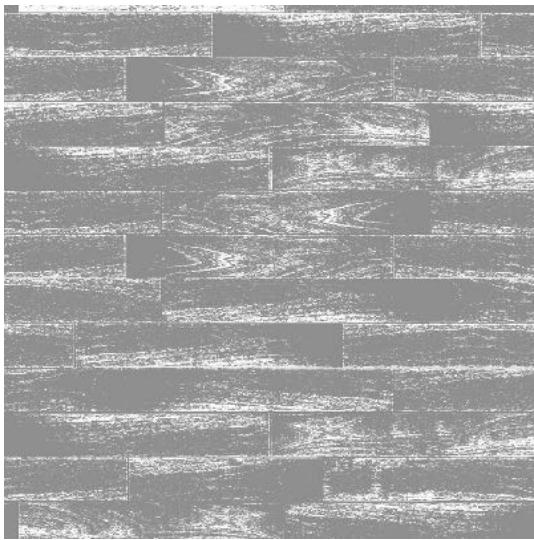
Texture mapping

- Objects have properties that vary across the surface
- Apply images to vary material parameters over the surface

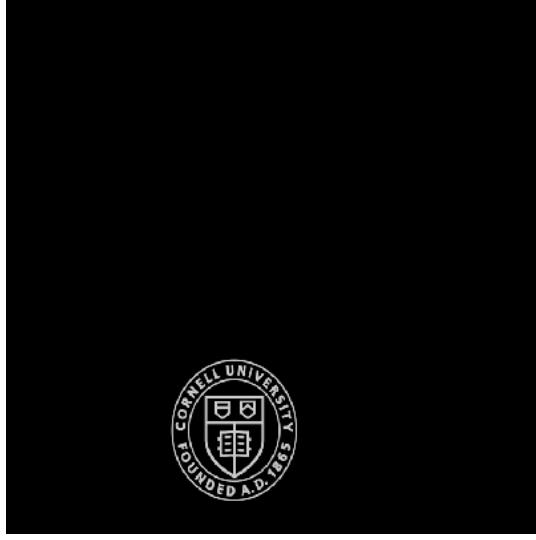




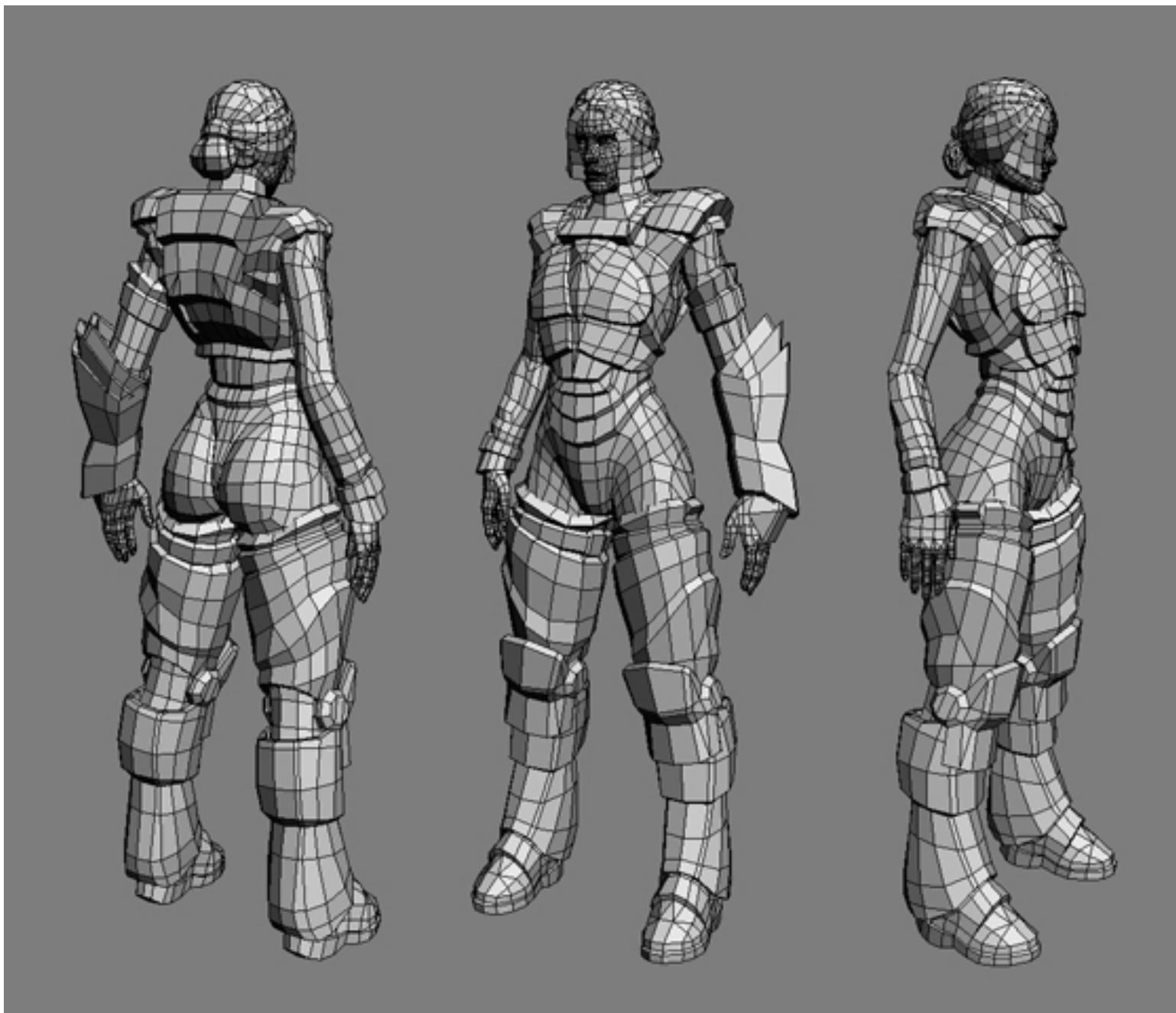








RENDERED USING MITSUBI





[source unknown]



(a) High-res geometry



(b) Real-time hybrid map rendering



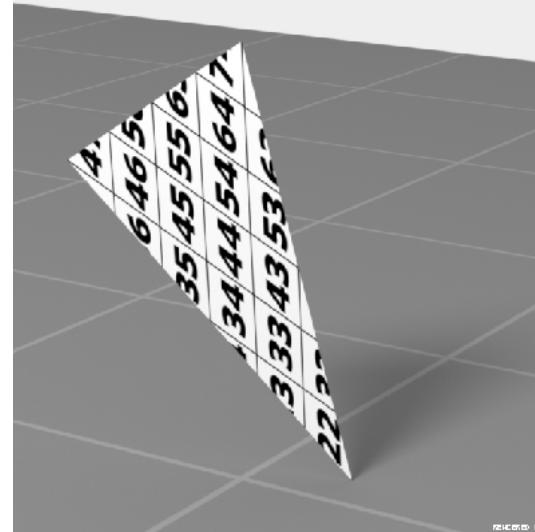
(c) Offline SSS rendering

[Paul Debevec]

Texture coordinates

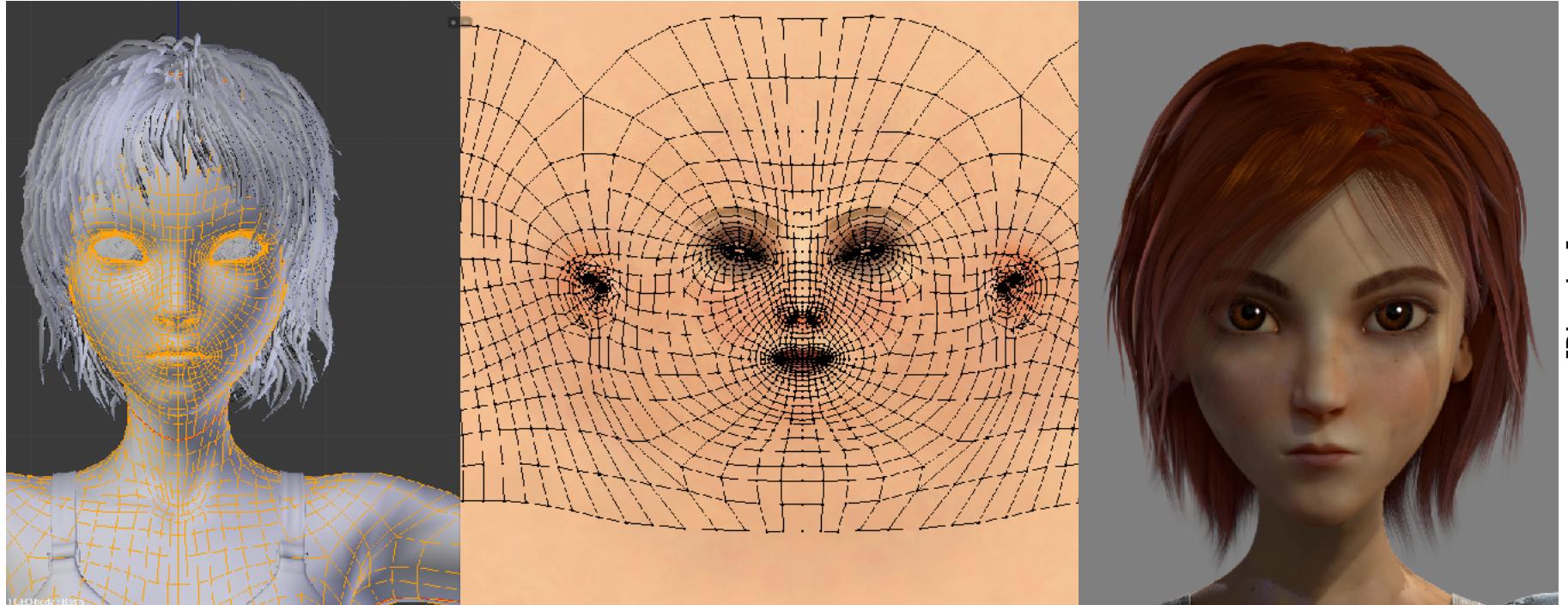
- Texture coordinates are per-vertex data like vertex positions
 - can think of them as a second position: each vertex has a position in 3D space and in 2D texture space
- How to come up with (u,v) s for points inside triangles?

09	19	29	39	49	59	69	79	89	99
08	18	28	38	48	58	68	78	88	98
07	17	27	37	47	57	67	77	87	97
06	16	26	36	46	56	66	76	86	96
05	15	25	35	45	55	65	75	85	95
04	14	24	34	44	54	64	74	84	94
03	13	23	33	43	53	63	73	83	93
02	12	22	32	42	52	62	72	82	92
01	11	21	31	41	51	61	71	81	91
00	10	20	30	40	50	60	70	80	90



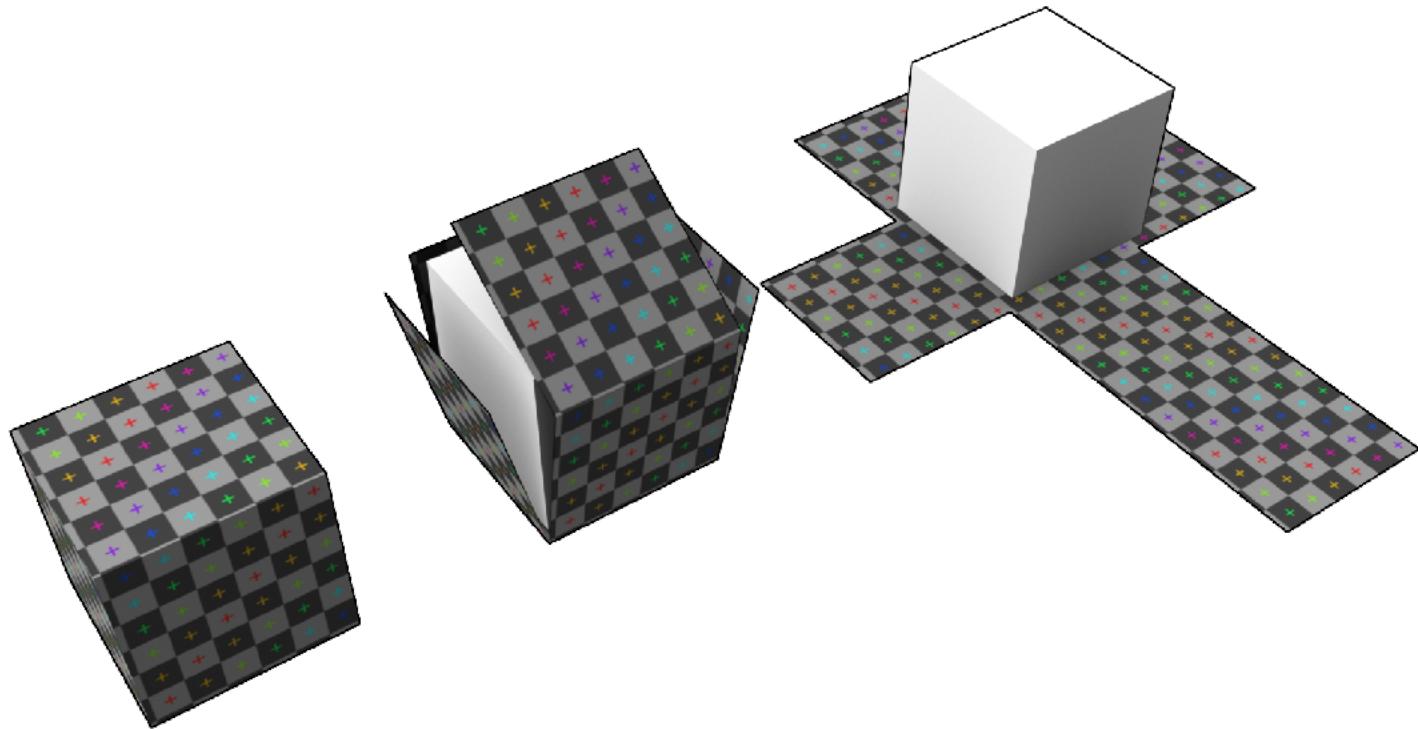
Texture coordinates

- For each mesh vertex, assign a position in the image plane $[0,1]^2$
- Induces a mapping functions from 3D vertices to 2D pixels



Texture coordinates

- Assign this mapping is like “unrolling” the surface onto a plane
- Often done by hand, but some tools exist to help
- Will have discontinuities in most cases

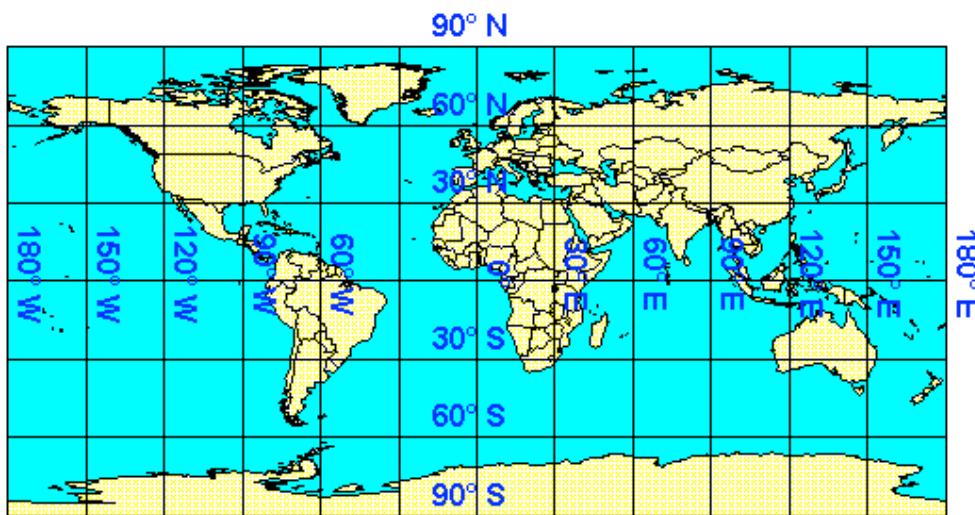


[Zepheris@WikimediaCommons]

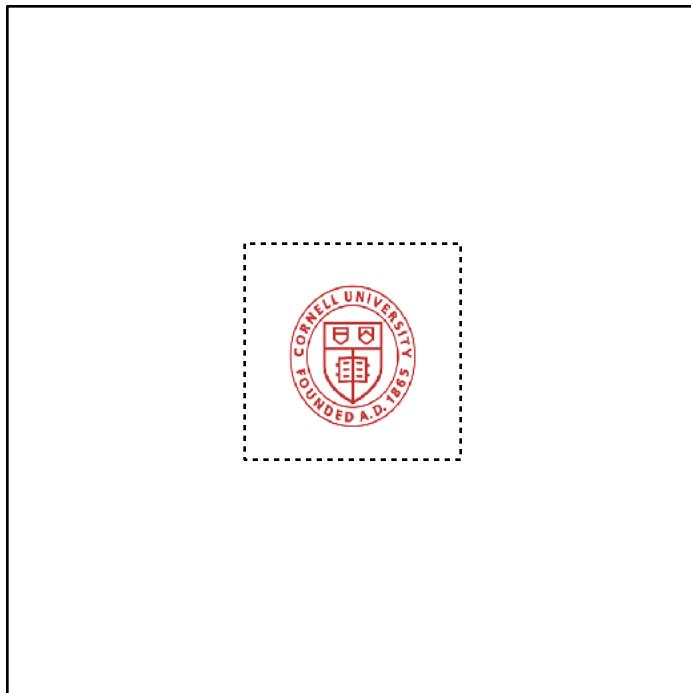
Texture coordinates

- Will also have distortions in most cases
 - Greenland looks really big in the map

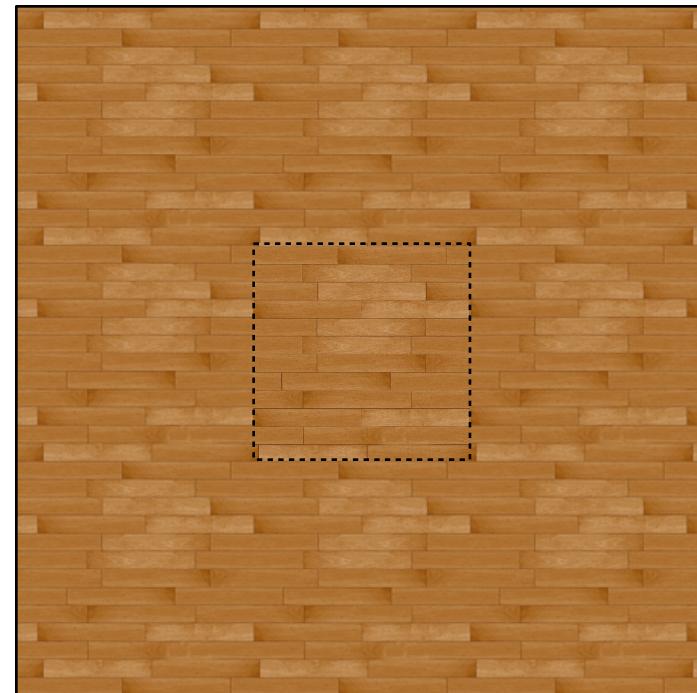
[map: Peter H. Dana]



Wrapping modes



clamp

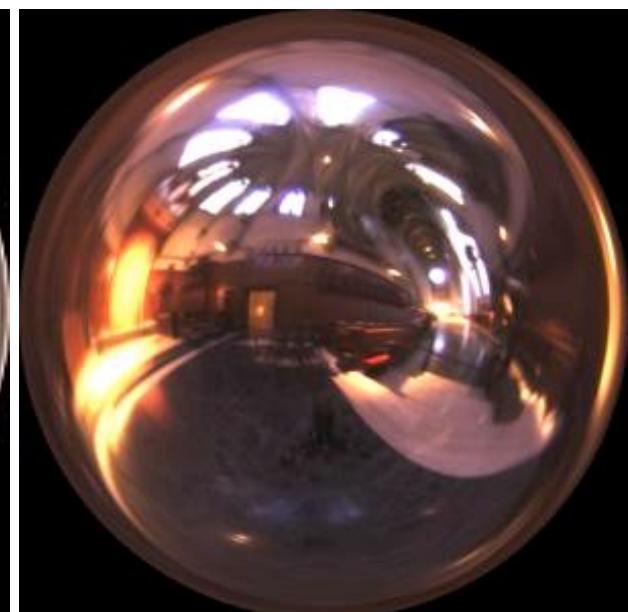
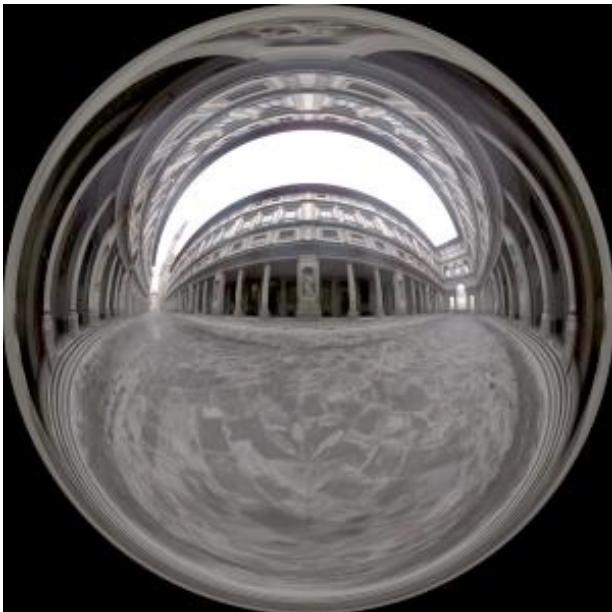


tiled

Environment Maps

Environment Maps

- Store far-away scene in an 360 panoramic image
- Main reasons: optimization and produces realistic rendering when used as a light source



[Paul Debevec]

Environment Maps

- Store far-away scene in an 360 panoramic image
- Main reasons: optimization and produces realistic rendering when used as a light source

```
struct environment {  
    frame3f frame;  
    vec3f emission;           // emission  
    texture* emission_tex;   // emission map  
};
```

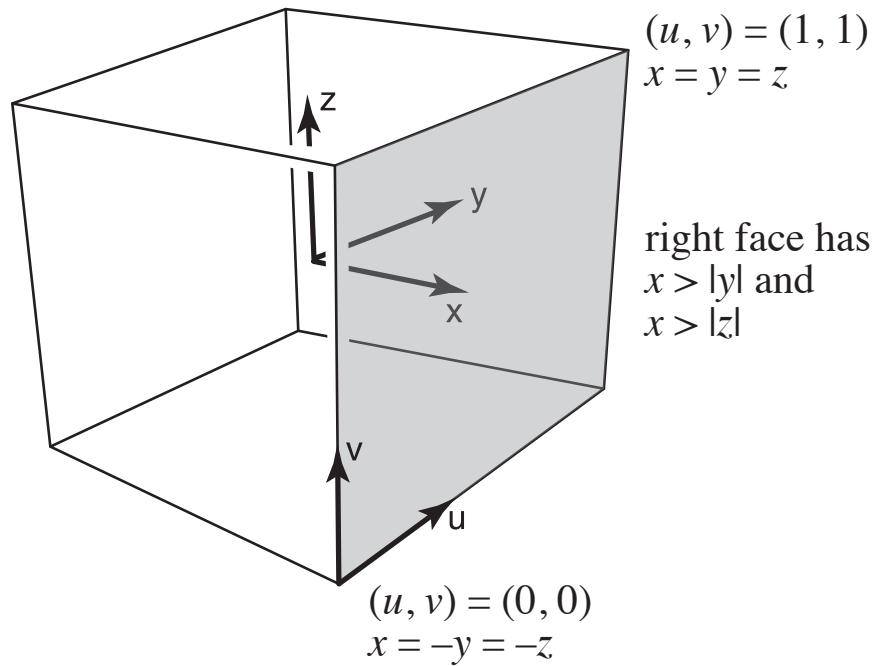
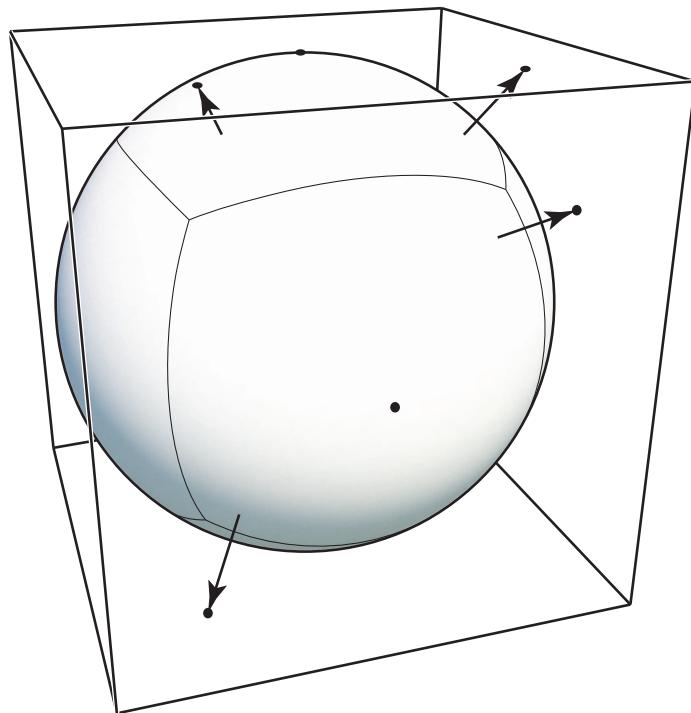
Lat-long Parametrization

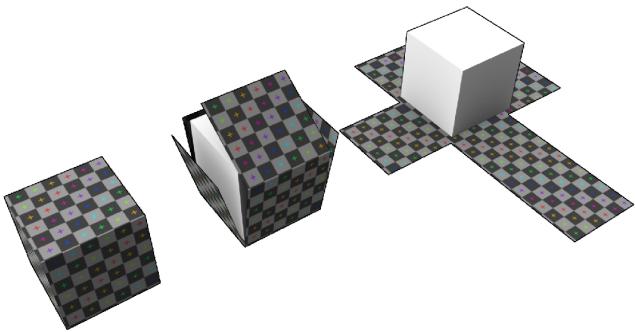


[GiantCowFilms]

Cube Map

- Cube texture with six texture maps



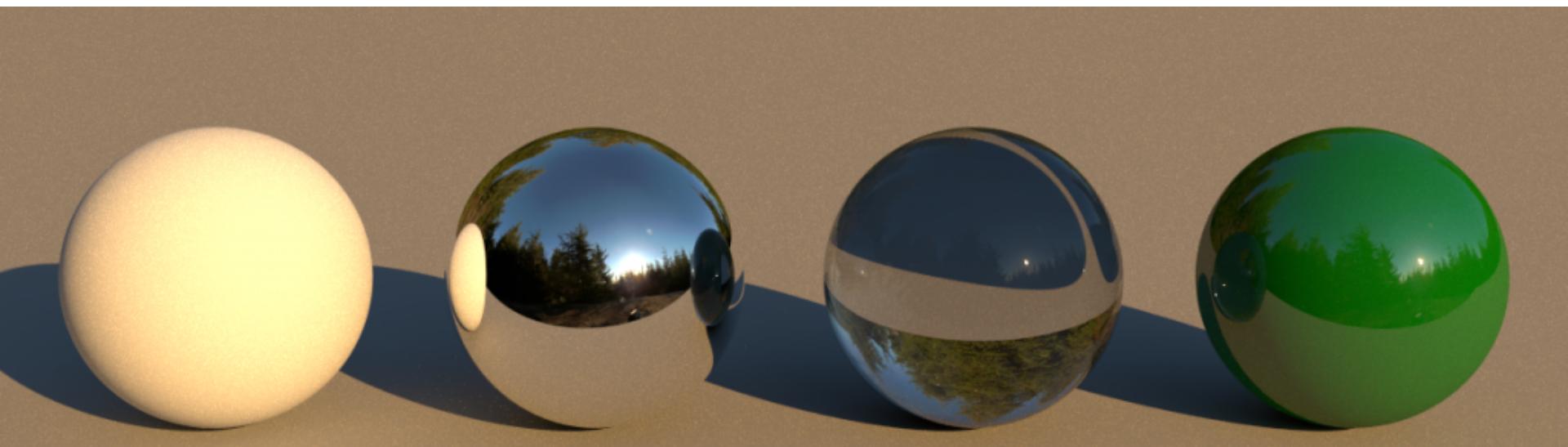


[Zepheris@WikimediaCommons]



[Emil Persson]

HDRI illumination



[GiantCowFilms]

HDRI illumination



[GiantCowFilms]

Representing Scenes

Representing Objects

- Need a way to associate geometry and materials
- Option 1: Add material inside a mesh
 - mixing geometry and shading
 - cannot re-used shape data
- Option 2: Bundle a mesh, a material and a frame into an object
 - with the coordinate frame, we can reposition the mesh
 - *instancing*: can represent large scenes by placing same objects

```
struct object {  
    frame3f frame;          // coordinate frame  
    shape* shape;           // shape data  
    material* material;    // material  
};
```

Instancing



[pbrt.]

Representing Scenes

- For now, we consider scenes as lists of objects, textures and materials and include a camera and environment map
- Later, we will introduce more complex scene modeling

```
struct scene {  
    vector<camera*> cameras;  
    vector<object*> objects;  
    vector<shape*> shapes;  
    vector<material*> materials;  
    vector<texture*> textures;  
    vector<environment*> environments;  
};
```