AUEB COMPUTER GRAPHICS GROUP

COMPUTER GRAPHICS COURSE

Geometry Representation



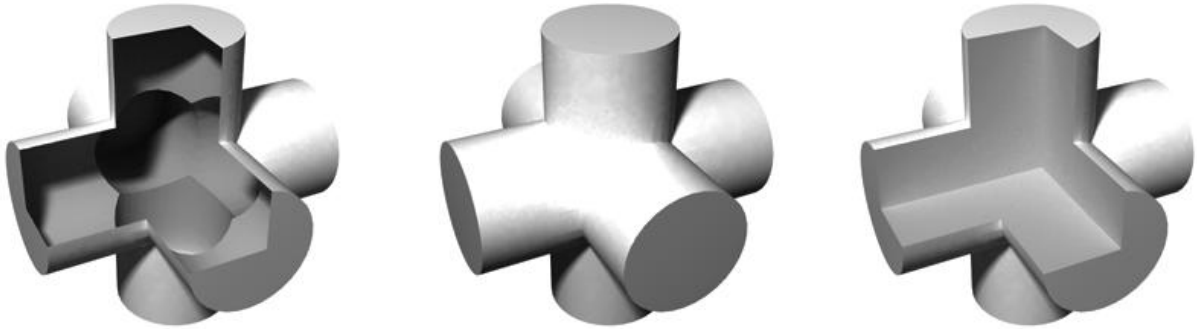
Andreas A. Vasilakis

How do we Represent our World?

- In general, the mathematical world we display in graphics consists of entities describing:
  - The geometry of the surface of objects
  - The volume of the space inside and outside the surfaces
  - The energy (light) that is transmitted
  - The materials (substance qualities) the energy interacts with
  - The spatial relationships of entities
  - The dynamics of all the above (motion)

Representing Geometry - Surfaces

- In most cases, we are interested in displaying surfaces, i.e. the “shell” of 3D objects that is the interface between one medium and another
- They are 2D embeddings in a 3D space



In rendering, we are usually interested in the surface of an object

- This is because most lighting “events” occur on or near this interface

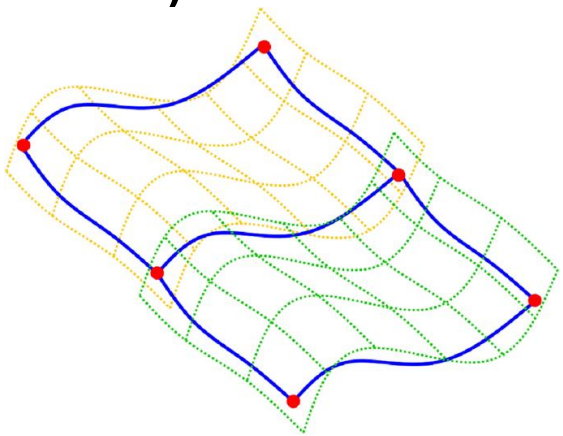
Representing Geometry - Volume

- The volume of a 3D entity becomes important when light interacts with it as it travels though the medium
- Most objects are opaque, absorbing the transmitted light fast
- However, the rendering of important light scattering phaenomena must take into account transmission through dense media such as:
  - Clouds, smoke, wax, milk etc.
  - These are expressed as volumetric data
  - Data defined everywhere inside *boundary surface*



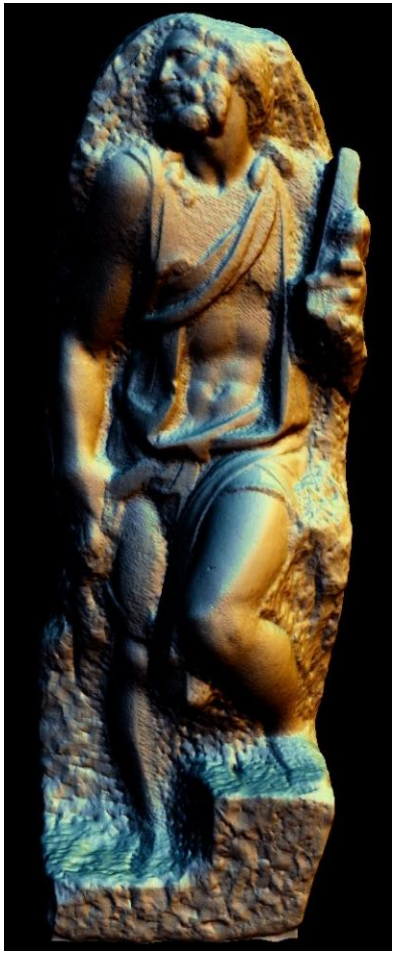
Representing Geometry - Curves

- 2D/3D curves are by definition infinitesimally thin and insubstantial, and therefore non-renderable per se.
- They are 1D embeddings in 2D or 3D space
- They are used in graphics to define boundaries, trajectories and higher-dimension entities (e.g. parametric surfaces)
- We sometimes “plot” the curves, i.e. approximate them by pixels (of non-zero area) on an image plane



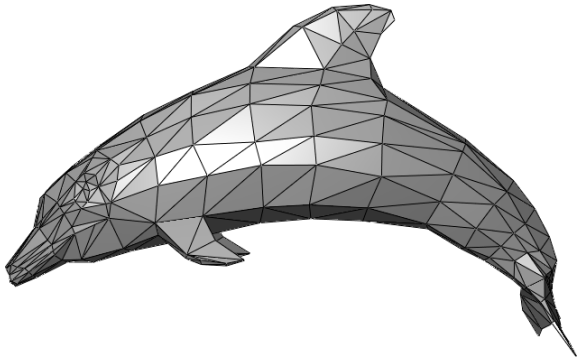
Representing Geometry - Points

- Points (isolated vertices) in 3D and 2D space are sometimes drawn to represent:
  - Scattered data of variable density
  - “very small” (sub-pixel) objects, such as particles
- We typically use these in massive quantities to approximate either surface data or volume data (a processes called *point-based* rendering)
- Points are not necessarily rendered as single pixels



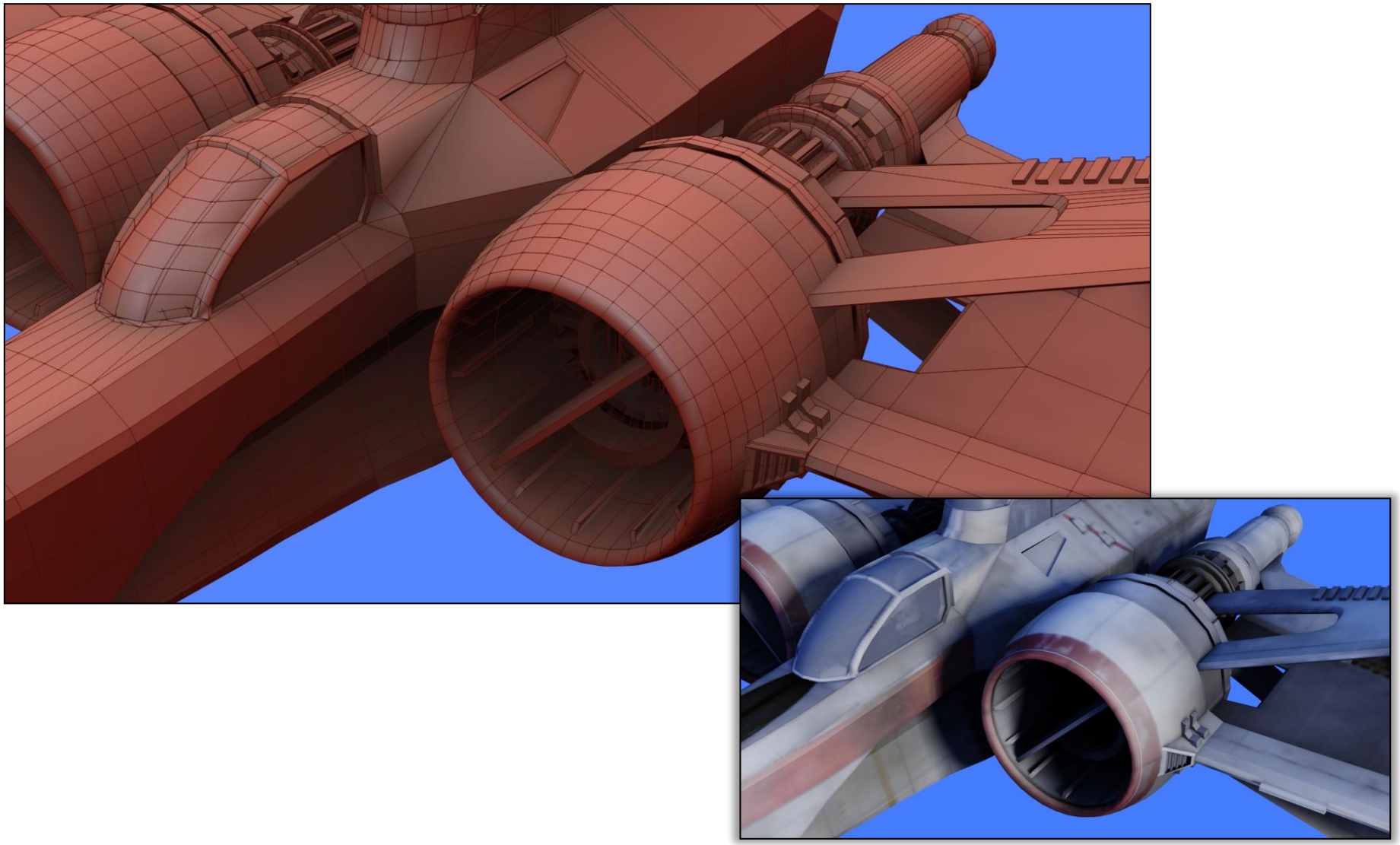
Surfaces

- Surfaces are composed or modelled via an aggregation of surface elements
- These elements can be:
  - Curved parametric patches
  - Flat polygons → usually triangles



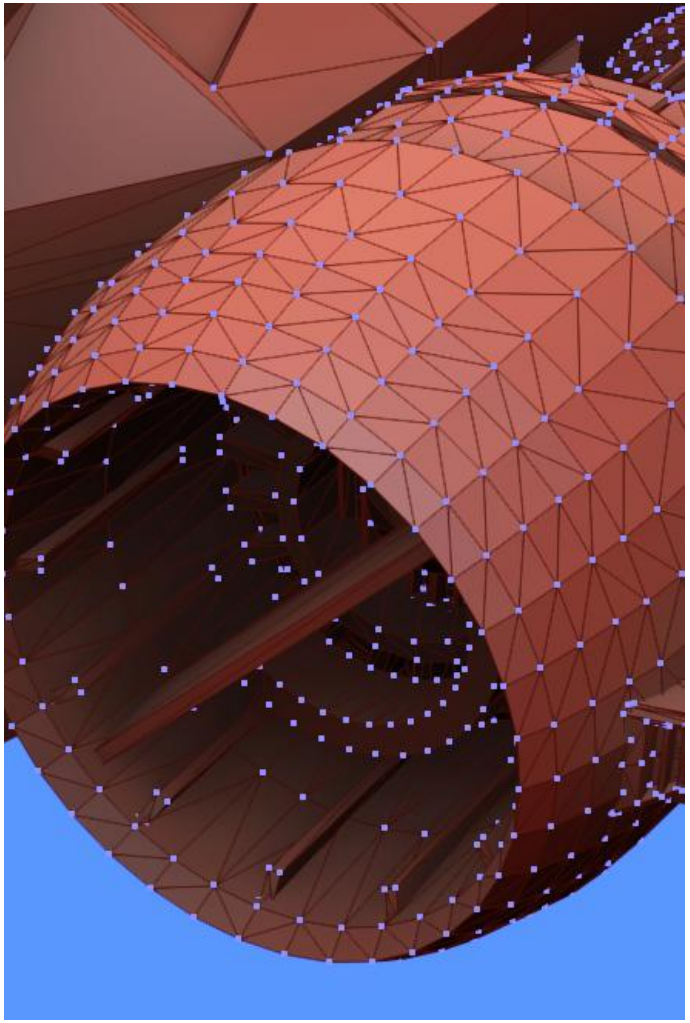
- An organization of a surface into connected polygonal surface elements is called a mesh

Mesh Example



Triangle Meshes

- The most common type of polygonal mesh representation
- Very convenient to use in real-time rendering!

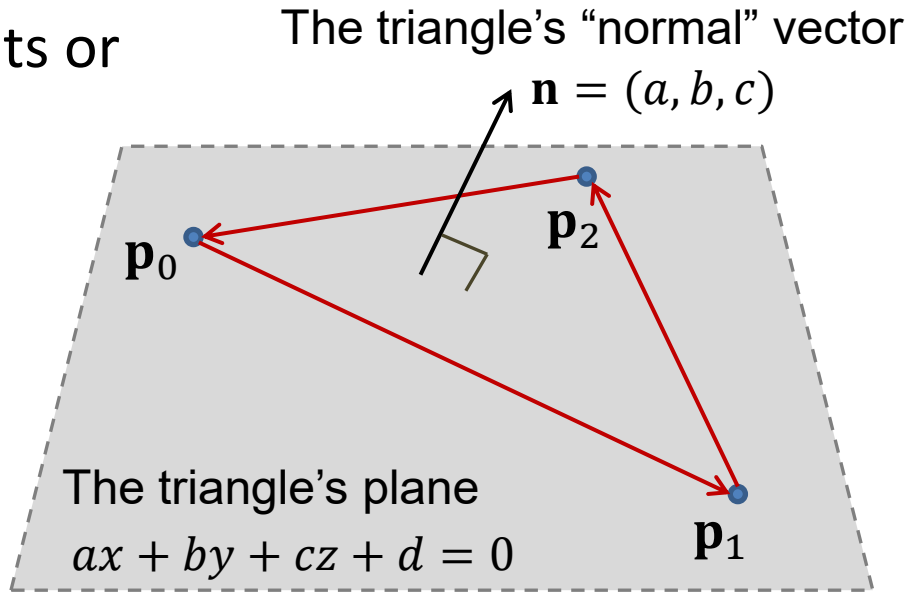


The Ingredients of Meshes

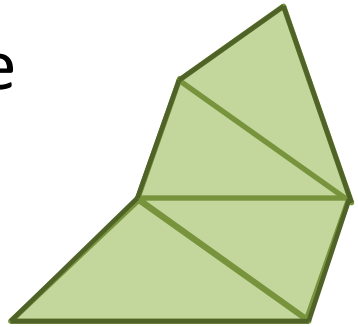
- What are their data?
  - Flat elements approximating both curved and flat surfaces
- Where are they?
  - Vertices of triangles (points on the plane or in space)
- What is their shape?
  - Connectivity among vertices defines the structure of the mesh
- How do they look?
  - Material and shading attributes per vertex – interpolated / predicted inside each triangle

The Triangle

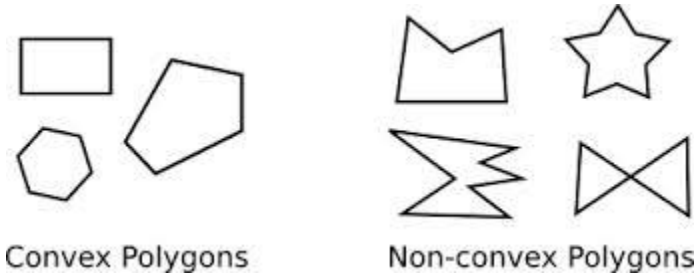
- A set of three (ordered) vertices (points in space)
- Connectivity:
  - Implied by order of points or
  - Given explicitly



Triangles - Useful Properties



- Minimal: The most elementary surface shape (3 points define an oriented plane)
- Always convex!
  - This is not true for generalized *n*-gons
  - A very useful property in calculations: its boundary coincides with its convex hull
- Any other polygon can be decomposed into a set of triangles!



Linear Combinations on the Plane

- Any point on the plane of the triangle can be uniquely written as a linear combination of the three triangle vertices. In fact:
- For any number *k* of vectors and scalar coefficients:

$$\mathbf{q} = \sum_{i=1}^{k} a_i \mathbf{v}_i$$ is their linear combination

Representing the Triangle Interior

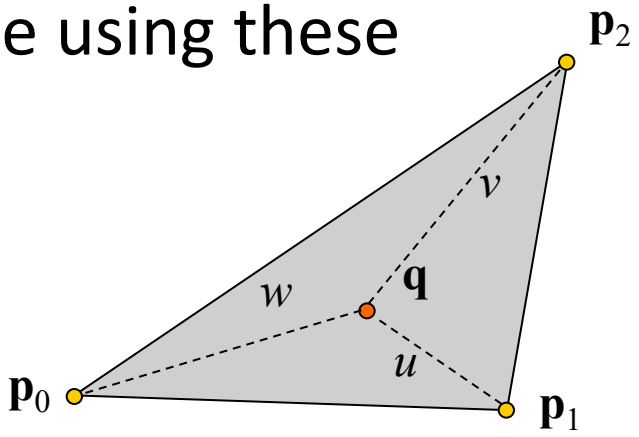
- Given the three triangle vertices (corners), any point inside the triangle (both in 2D and 3D) can be expressed as an affine combination of them
- This is an important property that translates to:
  - We only need the three corners of a triangle to fully describe its interior
- Relates to the topic of “affine transformations”

Barycentric Coordinates

- To interpolate parameters across a triangle we need to find the set of (unique) parameters u, v, w that define $\mathbf{q}$ as the linear combination of all 3 vertices

$$\mathbf{q} = w\mathbf{p}_0 + u\mathbf{p}_1 + v\mathbf{p}_2$$

- All vertex attributes can be linearly interpolated at arbitrary locations on the plane using these barycentric coordinates



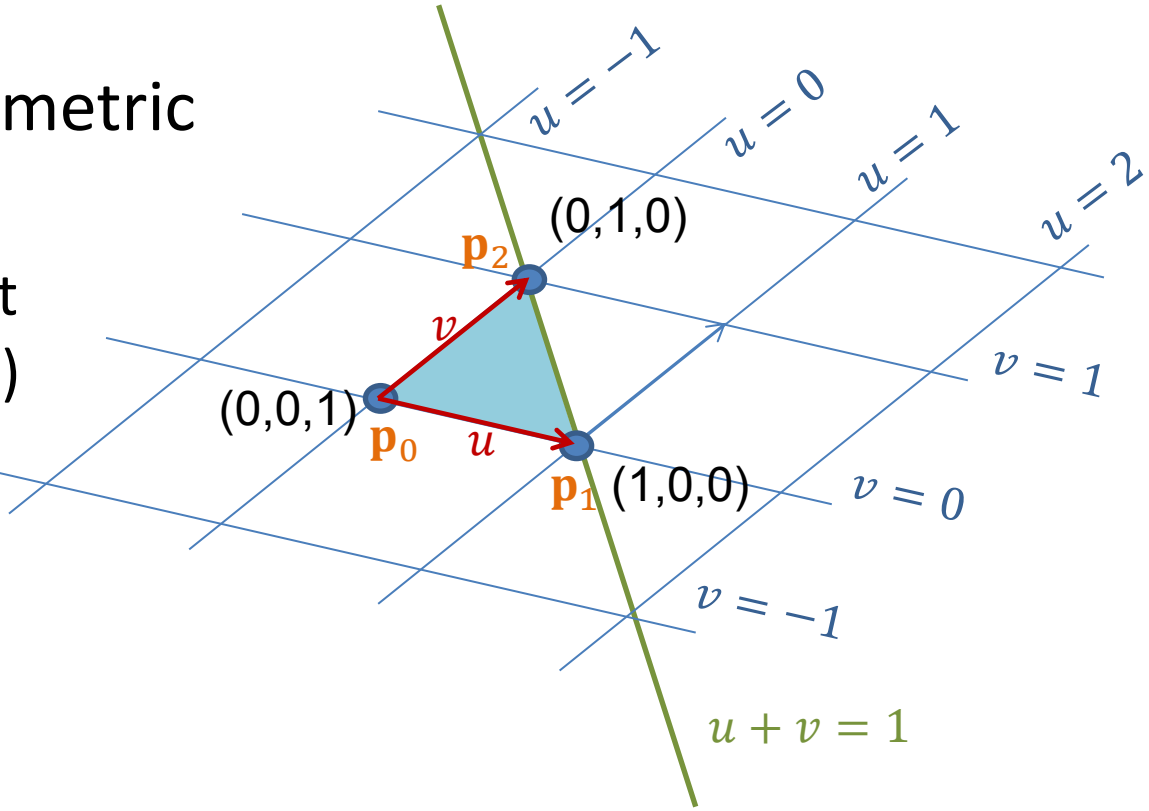
Barycentric Coordinates - Usage

- Barycentric coordinates are extremely useful in triangle rendering, as they are used for:
  - Interpolating triangle properties for arbitrary points inside the triangle
  - Performing point containment tests for various primitive intersection tests

Interpreting Barycentric Coordinates (1)

- They form a parametric space of:
  - two independent parameters (u, v)
  - One dependent parameter

    $(w = 1 - u - v)$



- All points with:

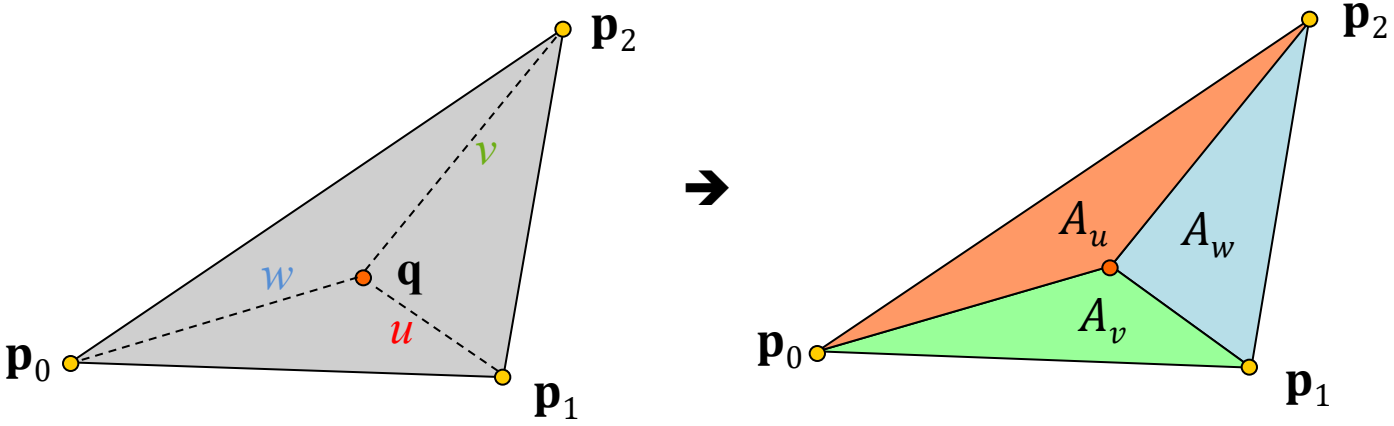
  $u > 0 \; AND \; v > 0 \; AND \; w > 0$

  Lie inside the triangle

Interpreting Barycentric Coordinates (2)

- Geometric properties:
  - Barycentric coordinates equal the ratio of triangle areas formed by the opposite side and the query point against the total triangle area
  - Can be exploited to computed them:

$$w = 1 - u - v, \qquad u = \frac{A_u}{A_{total}}, \qquad v = \frac{A_v}{A_{total}}$$



Finding the Barycentric Coordinates (1)

- Let $\mathbf{p}_i = (x_i, y_i, z_i), i = 0..2$ the triangle vertices and $\mathbf{q} = (x_q, y_q, z_q)$ the query point. Then:

$$\begin{aligned} wx_0 + ux_1 + vx_2 &= x_q \\ wy_0 + uy_1 + vy_2 &= y_q \\ wz_0 + uz_1 + vz_2 &= z_q \end{aligned} \xrightarrow{w=1-u-v} \begin{aligned} x_0 + u(x_1-x_0) + v(x_2-x_0) &= x_q \\ y_0 + u(y_1-y_0) + v(y_2-y_0) &= y_q \\ z_0 + u(z_1-z_0) + v(z_2-z_0) &= z_q \end{aligned}$$

We can drop one coordinate equation and solve the resulting 2x2 system:

$$\begin{aligned} u(x_1-x_0) + v(x_2-x_0) &= x_q - x_0 \\ u(y_1-y_0) + v(y_2-y_0) &= y_q - y_0 \end{aligned} \Leftrightarrow \begin{bmatrix} (x_1-x_0) & (x_2-x_0) \\ (y_1-y_0) & (y_2-y_0) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} x_q - x_0 \\ y_q - y_0 \end{bmatrix}$$

Finding the Barycentric Coordinates (2)

- Using Cramer's rule, we can solve the 2X2 linear system analytically to obtain (u, v) and therefore w

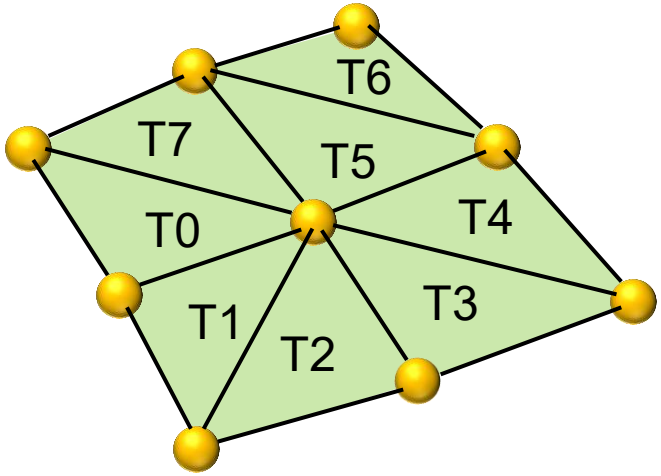
$$u = \frac{\begin{vmatrix}(x_q - x_0) & (x_2 - x_0)\\ (y_q - y_0) & (y_2 - y_0)\end{vmatrix}}{\begin{vmatrix}(x_1 - x_0) & (x_2 - x_0)\\ (y_1 - y_0) & (y_2 - y_0)\end{vmatrix}} = \frac{(x_q - x_0)(y_2 - y_0) - (y_q - y_0)(x_2 - x_0)}{(x_1 - x_0)(y_2 - y_0) - (y_1 - y_0)(x_2 - x_0)}$$

$$v = \frac{\begin{vmatrix}(x_1 - x_0) & (x_q - x_0)\\ (y_1 - y_0) & (y_q - y_0)\end{vmatrix}}{\begin{vmatrix}(x_1 - x_0) & (x_2 - x_0)\\ (y_1 - y_0) & (y_2 - y_0)\end{vmatrix}} = \frac{(x_1 - x_0)(y_q - y_0) - (y_1 - y_0)(x_q - x_0)}{(x_1 - x_0)(y_2 - y_0) - (y_1 - y_0)(x_2 - x_0)}$$

$$w = 1 - u - v$$

Mesh Data Representation and Storage

- (Triangle) meshes are collections of triangles
- Can be:
  - A set of disjoint, independent triangles (a triangle “soup”)
  - A set of vertices “shared” among triangles, given a set of “connections”:



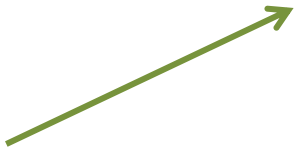
Triangle Representation (1)

- Triangle data are typically stored in:
  - Self-contained triangle arrays
  - Indexed attribute arrays
- Vertex attributes:
  - All data related to a single vertex of a triangle
  - These include at least the position of each vertex
  - May include additional data such as color, per vertex normal vector, texture coordinates, user-defined variables etc
- Per triangle attributes:
  - Data associated to the entire triangle (e.g. material, geometric normal etc)

Triangle Data: An Example (1)

```
class Triangle
{
   float vertex[3][3];          // 3 vertices X 3 coords
   float vertex_normal[3][3]; // 3 vertices X 3 coords
   float vertex_color[3][3];  // 3 vertices X 3 components
   float plane_normal [3];      // xyz
   class Material * p_mat;      // pointer to material data
};
```

Per vertex attributes

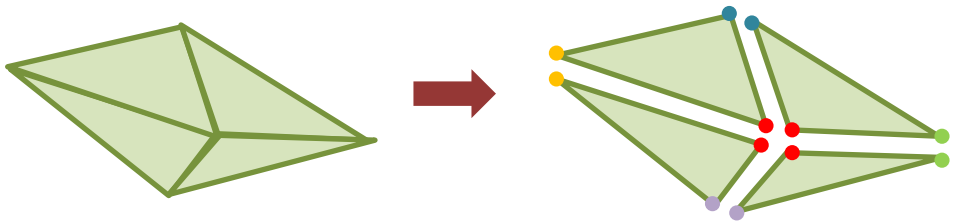


Shared material: many triangles use the same material, so the triangle only points to this single material instance

Can I do better?

Triangle Data: An Example (2)

- Can I do better?
  - Observe that many vertices are repeated in the triangle data:



- We can save significant storage and bandwidth (this will become important later – see real-time graphics) if we:
  - Keep per vertex attribute data separately
  - Index the vertex attributes

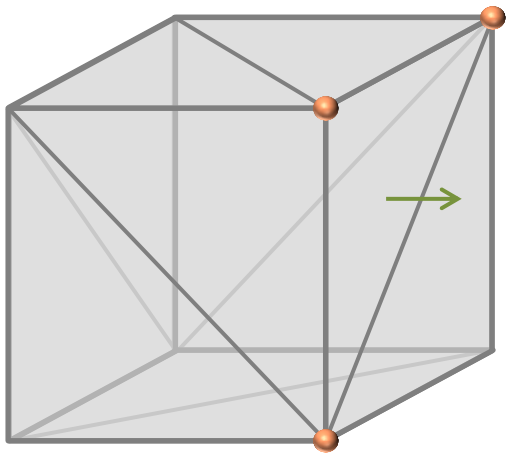
Triangle Data: An Example (3)

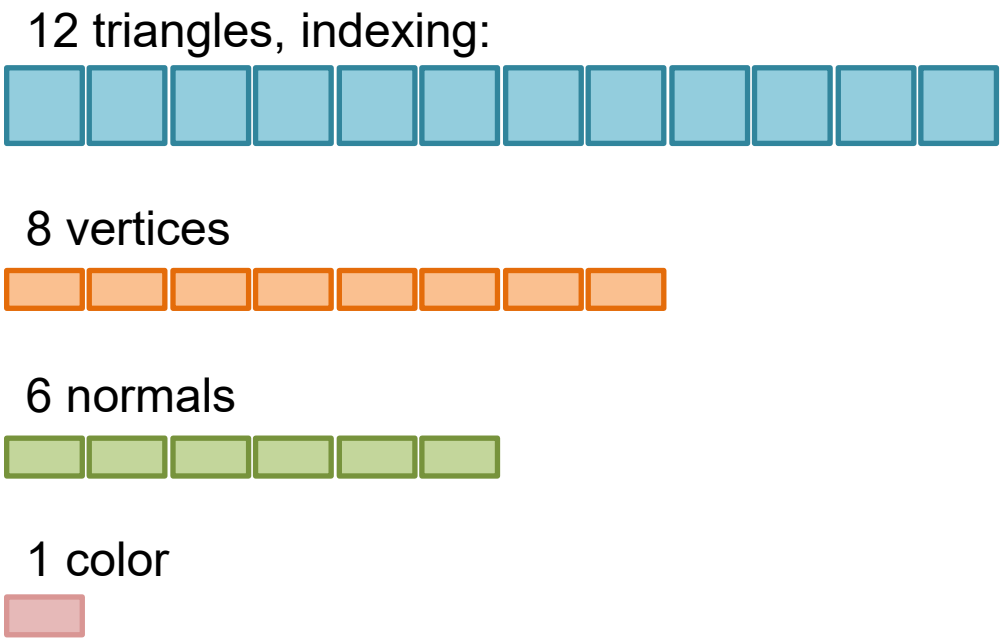
```
class Triangle
{
   unsigned int v_index[3];    // 3 vertices X 1 index
   unsigned int n_index[3];    // 3 vertices X 1 index
   unsigned int c_index[3];    // 3 vertices X 1 index
   float plane_normal [3];     // xyz
   class Material * p_mat;     // pointer to material data
};
```

Per vertex attributes

Triangle Data: Attribute Buffers

- Now we can have separate attribute buffers of different size:





Unindexed vs Indexed Data

- Memory savings from this indexing (Cube example):
  - Cost for per vertex data in unindexed cube triangles:

    27 floats (4 bytes each) X 12 triangles = **1296 bytes**

  - Cost for per vertex data in indexed cube triangles:

    9X12 integer indices (4 bytes each) = 432 bytes

    8 vertices X 3 floats = 96 bytes

    6 normals X 3 floats = 72 bytes

    1 color X 3 floats = 12 bytes

    total: **612 bytes**

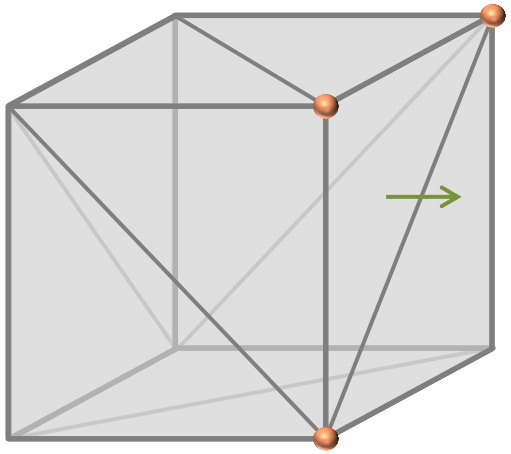
Attribute Structures (1)

- In this example, we waste a lot of memory in indexing
- Alternatively, we could use a per vertex structure and index a single array of *vertex objects*:

```
class Triangle
{
   unsigned int index[3];      // 3 vertices X 1 structure index
   float plane_normal [3];     // xyz
   class Material * p_mat;     // pointer to material data
};
```

Attribute Structures (2)

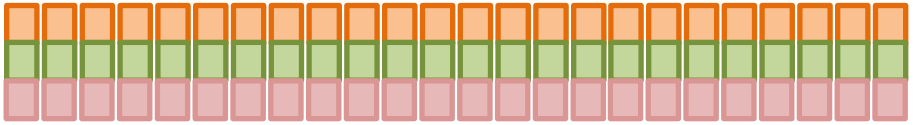
- Now we have a single attribute buffer:



12 triangles, indexing:



24 structures



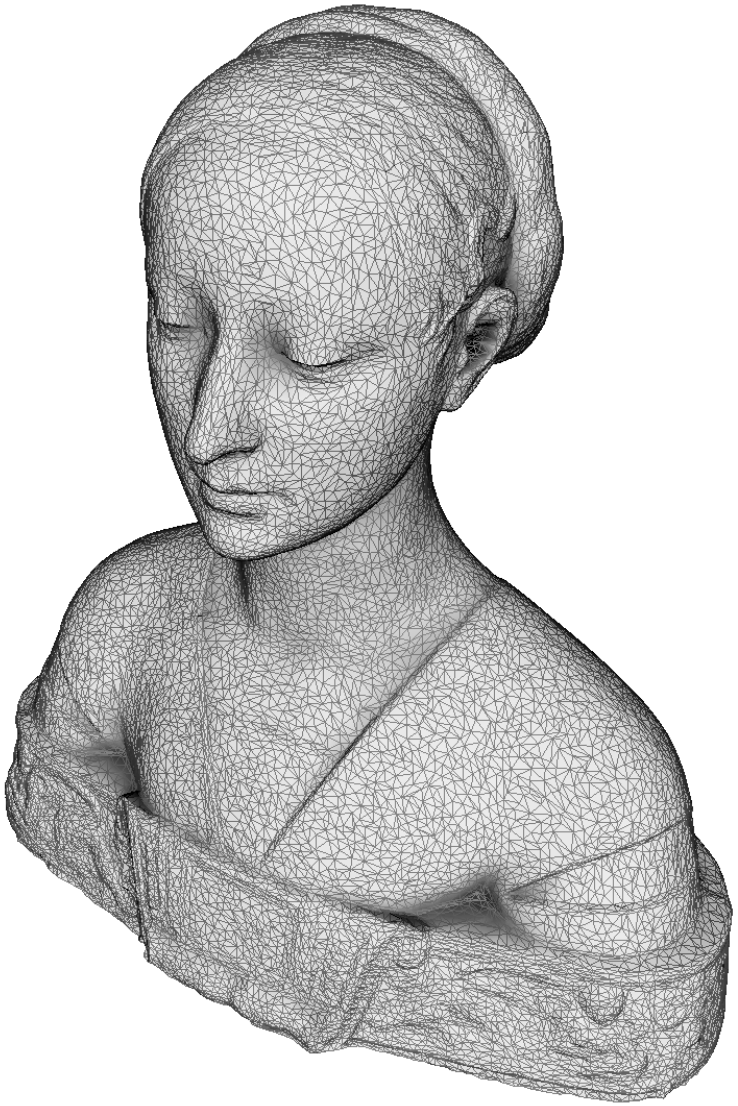
Total bytes: 144 (indices) + 864 (data) = 1008 bytes

This is larger than the separate data buffer size!!! **Then why use it…???**

Attribute Structures (3)

- Why use a more “wasteful” indexing?
  - The particular cube example is an extreme case that reuses many data
  - Most typical objects have attribute buffers of comparable length
  - But most importantly:
- Using a single per vertex index, abstracts the data that a vertex carries!
  - The triangle (i.e. connectivity) structure does not have to care about how many attributes each vertex has

A More Typical Indexed Mesh



- Using **unindexed triangles** (positions+normals):

  3,596,688 bytes

- Using **separate attribute** buffers:

  1,867,560 bytes

- Using **vertex object buffer**:

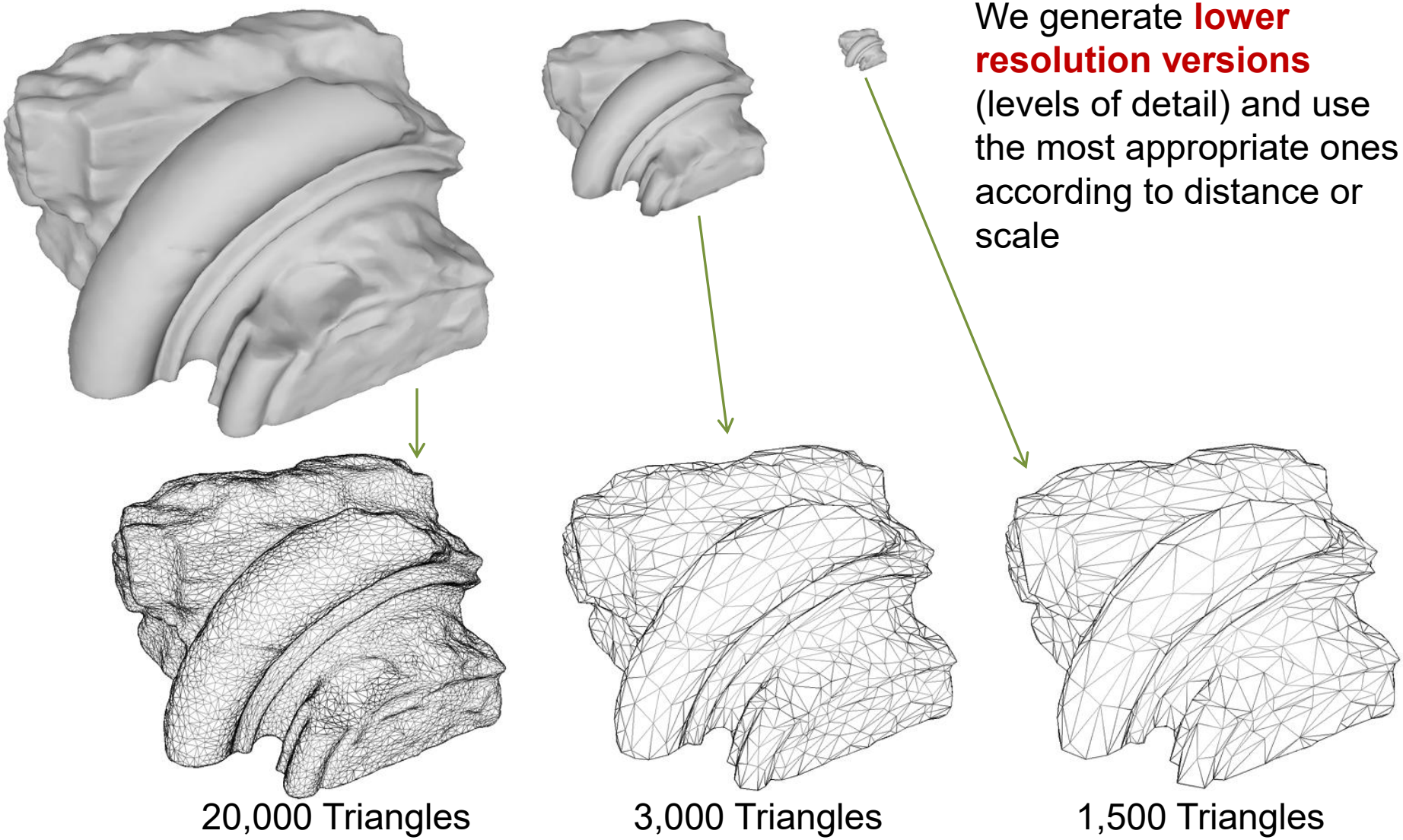
  1,268,112 bytes

Level of Detail (1)

- For distant/small versions of objects, **detail is lost**, i.e. cannot be sufficiently sampled and displayed
- There is no point in attempting to display rich detail if we are not going to see it! → waste of bandwidth and processing power



Level of Detail (2)



We generate **lower resolution versions** (levels of detail) and use the most appropriate ones according to distance or scale

LOD – Selection (1)

- The selection criteria of a LOD must answer the question:
  - What is the simplest LOD for which no visual artifacts appear?
  - Sometimes, in order to be more aggressively efficient, we allow artifacts to become noticeable, or
  - Completely disable the rendering of an object
- Criteria usually map the expected on-screen scale of an object (in pixels) to a LOD

LOD – Selection (2)

- Sometimes, we allow LODs to coexist → blending between different levels

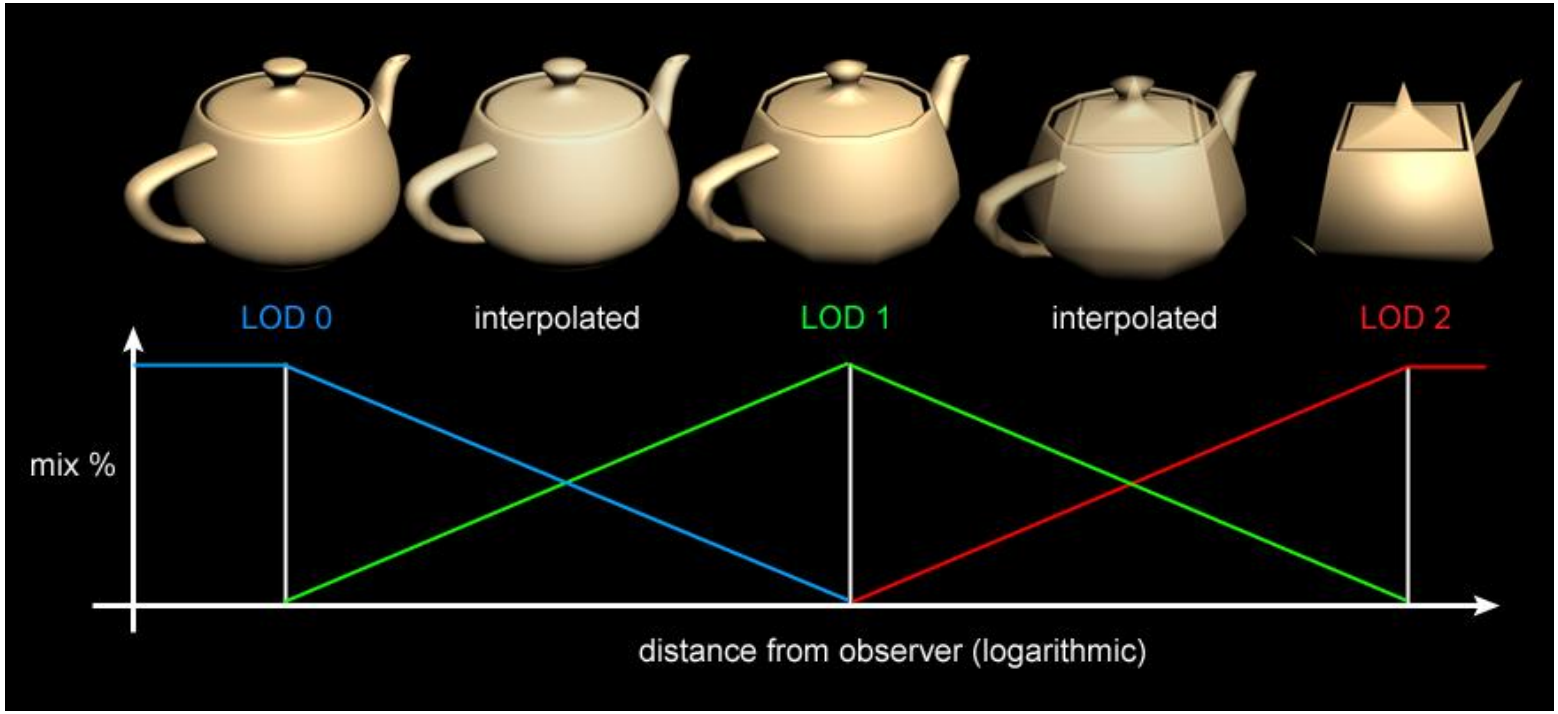


Image-based Rendering: Proxies (1)

- For distant or simply too many repeated objects, it is not efficient to approximate certain LODs with polygonal models
- We can use image-based proxies (billboards, fins etc.) to “host” an image representation of the object, often with transparency

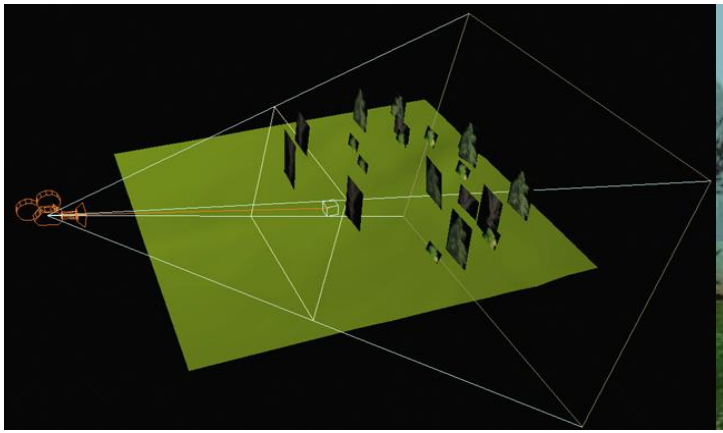
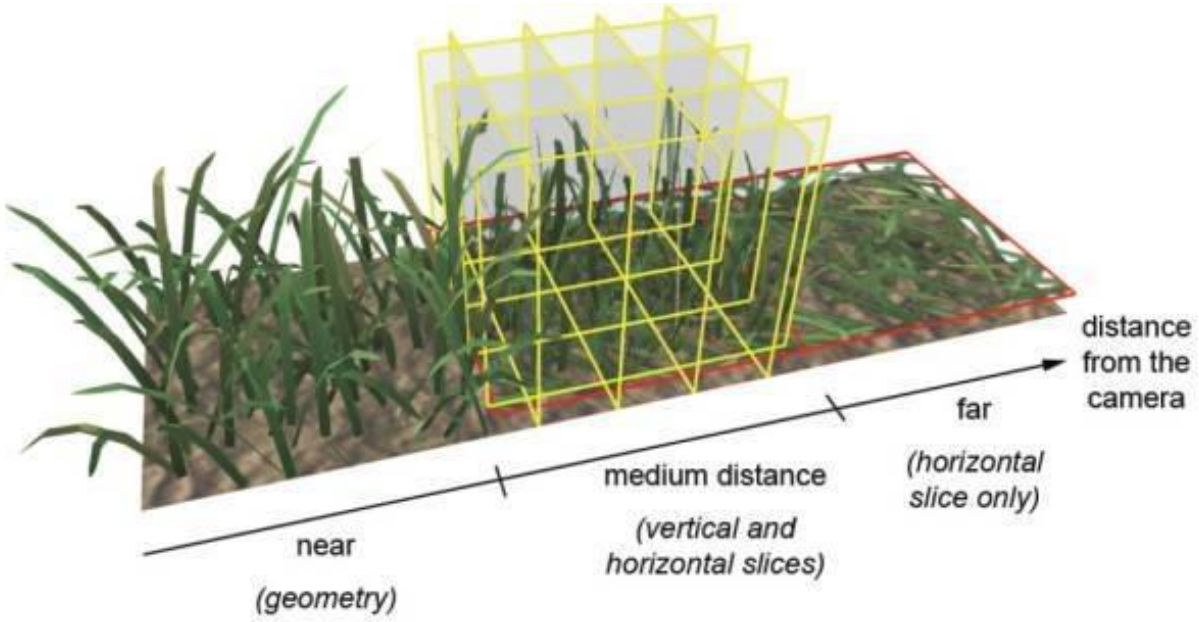




Image-based Rendering: Proxies (2)





Contributors

- Georgios Papaioannou

- Sources:
  - T. Theoharis, G. Papaioannou, N. Platis, N. M. Patrikalakis, Graphics & Visualization: Principles and Algorithms, CRC Press