Review for CS100433 Computer Graphics by Junqiao Zhao

INDEX:

1 Pipeline

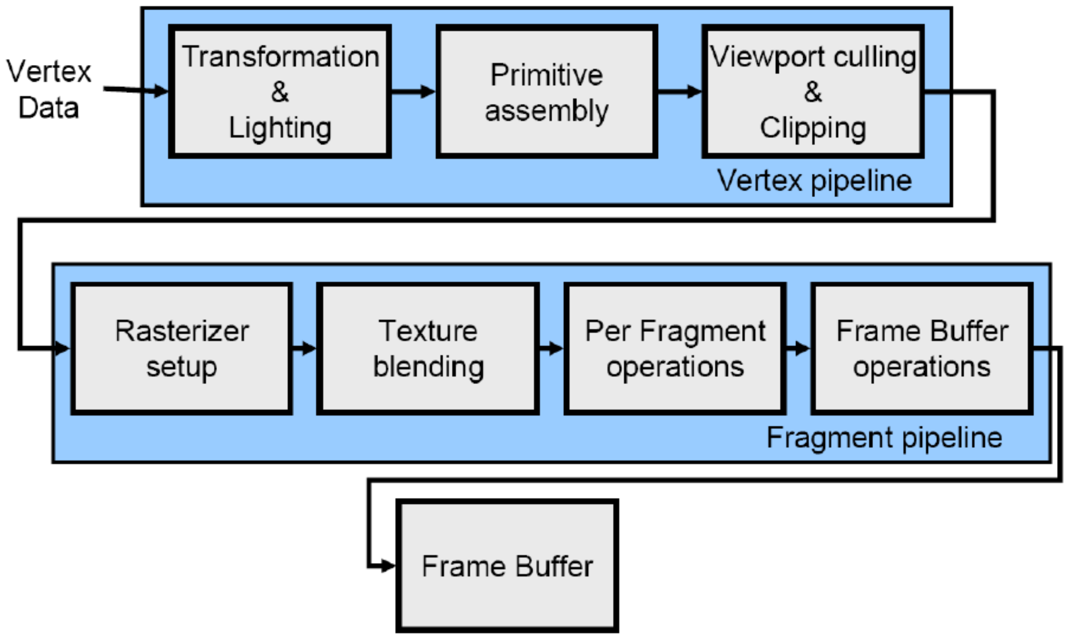
Graphic Pipeline

- Application stage
- Geometry stage
  1. Loaded 3D models
  2. Model Transformation
  3. Transformation: Viewing
  4. Hidden Surface Elimination
  5. Shading: reflection and lighting
- Rasterization stage
  1. Rasterization and Sampling

     Aliasing: distortion artefacts produced when representing a high-resolution signal at a lower resolution

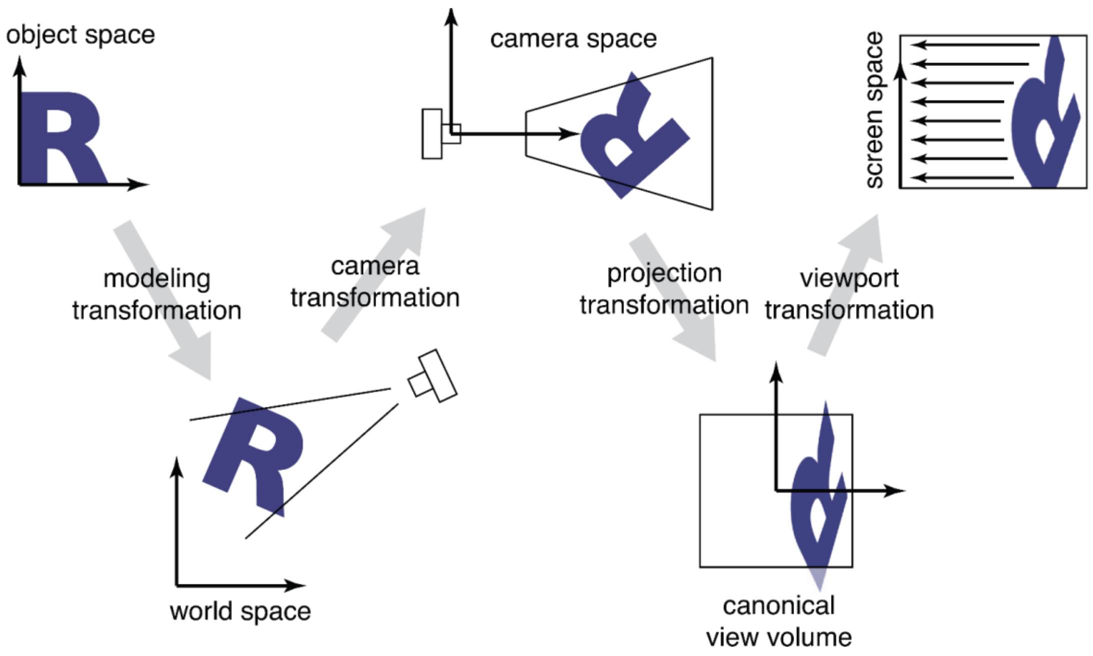
     Anti-aliasing: technique to remove aliasing
  2. Texture Mapping
  3. Image Composition
  4. Intensity and Color Quantization
  5. Framebuffer and Display

OpenGL pipeline



2 2D and 3D Viewing basics

Viewing Transformations



Viewing implementation

- Transform into camera coordinates.
- Perform projection into view volume or screen coordinates.
- Clip geometry outside the view volume.
- Remove hidden surfaces

3 Geometric Modeling Basics

Geometry vs Topology

Generally, it is a good idea to look for data structures that separate the geometry from the topology

- Geometry: locations of the vertices
- Topology: organization of the vertices and edges
- Topology holds even if geometry changes

Modeling

Surface based representation

Boundary representation - B-rep

A type of geometric model in which the size and shape of the solid is defined in terms of the faces, edges and vertices which make up its boundary (ISO19303).

- Representation by **bounding low-dimensional elements**

- **Organized** collection of low dimensional elements
- Simple B-reps (planar faces, straight edges) and complex B-reps (curved surfaces and edges)

Pros: flexible and computers can render them quickly. The vast majority of 3D models today are built as textured polygonal models

• **Cons:** polygons are planar and need approximate curved surfaces using many polygons, representation is not unique

Constructive solid geometry (CSG)

A type of geometric modelling in which a solid is defined as the result of a sequence of regularized Boolean operations operating on solid models (ISO 19303).

Pros: Computer-Aided Manufacturing: a brick with a hole drilled through it is represented as "just that" and CSG can easily assure that objects are "solid" or water-tight

• **Cons:** Relationships between objects might be very complex (search the entire tree) Real world objects may get very complex

Volume-based representation

Voxels

- A voxel is a volume element (3D "pixel")

- A 3D cubical (or spherical array), with each element holding one (or more) data value (boolean, real)

Pros:

– Modelling continues phenomena: Medical, geology, body, etc.

– Regular data

– Easy to compute volume, make slices

Cons:

– Massive data for high resolution

– The surface is always somehow “rough”

Point based representation

Point cloud

- Easily accessed with laser scanning, range camera or stereo image matching
- No connectivity
- **Widely used!**

4 2D transformation

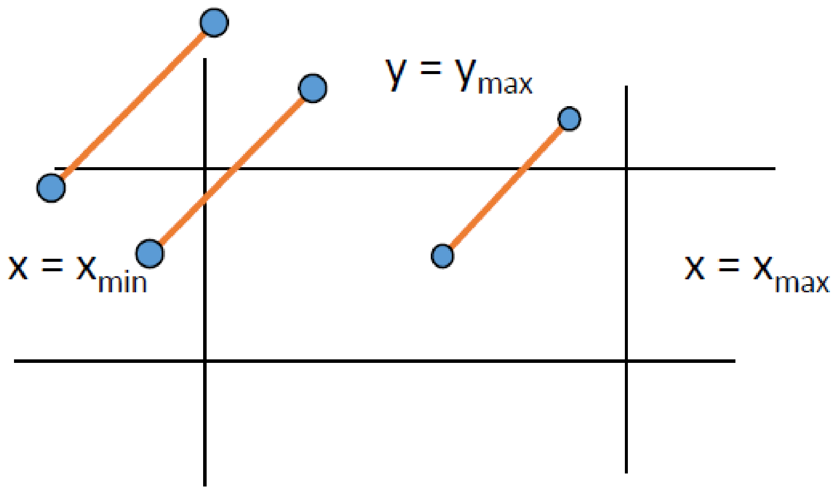
5 3D transformation

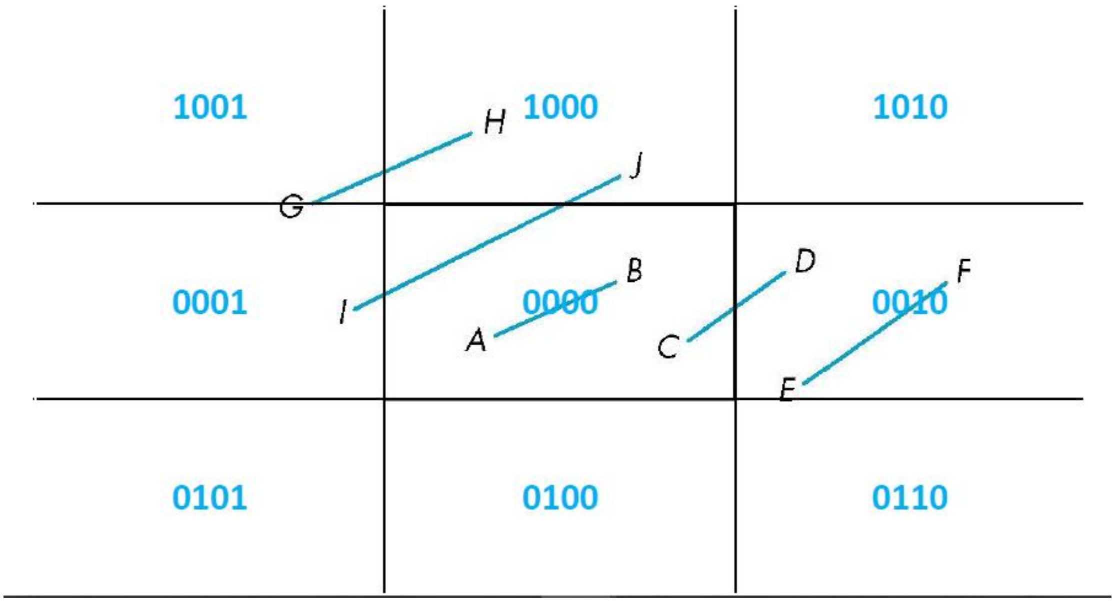
6 Revisiting 2D viewing

Clipping line segment

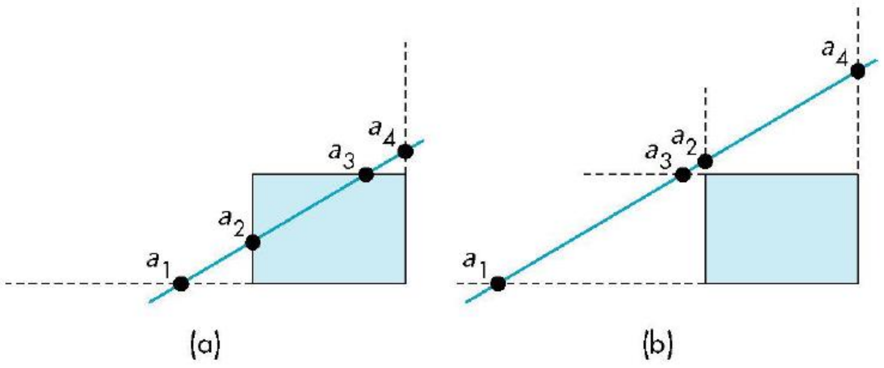
Cohen-Sutherland Algorithm

- Idea: eliminate as many cases as possible without computing intersections
- Start with four lines that determine the sides of the clipping window





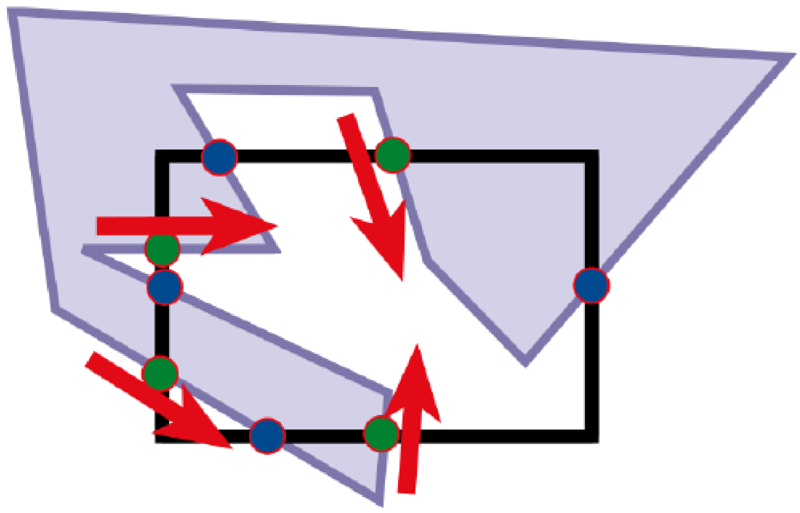
Liang-Barsky Clipping



Clipping polygon

Weiler-Atherton Clipping

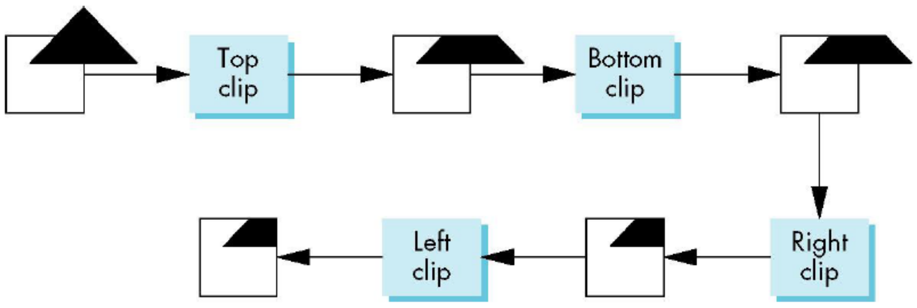
- Strategy: “Walk” polygon/window boundary
- Polygons are oriented (CCW)



Sutherland-Hodgman Polygon Clipping

- Clipping against each side of window is independent of other sides
    - Can use four independent clippers in a pipeline

Pipeline Clipping of Polygons



7 Revisiting 3D viewing

8 Visible Surface Detection

Visibility of primitive

- A scene primitive can be invisible for 3 reasons:
  - Primitive lies outside field of view (Clipping)
  - Primitive is *back-facing*
  - Primitive is occluded by one or more objects nearer the viewer

Visible surface algorithms.

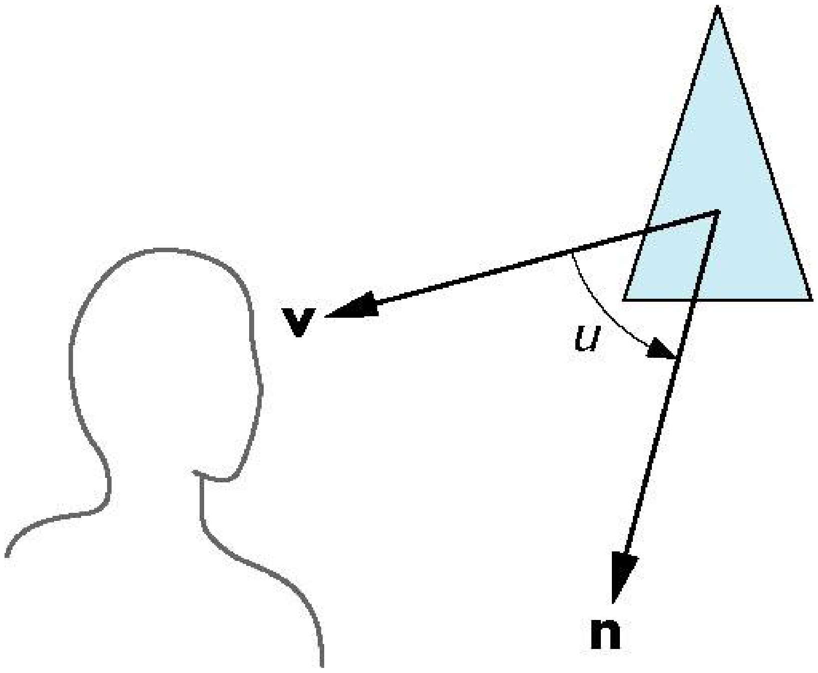
- Object space techniques: applied before vertices are mapped to pixels
  - Back face culling, Painter’s algorithm, BSP trees etc.
- Image space techniques: applied while the vertices are rasterized
- Z-buffering

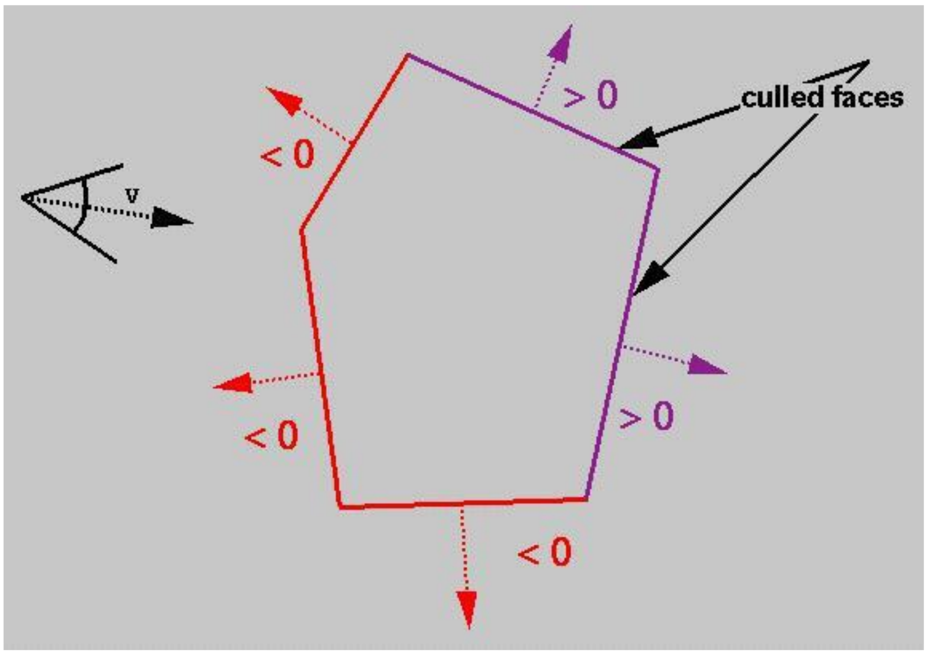
Object space approach

Back-Face Removal (Culling)

- Face is visible if 90 -90
- Equivalently cos or **v• n** 0

- In eye coordinate frame need only test the sign of the Z component



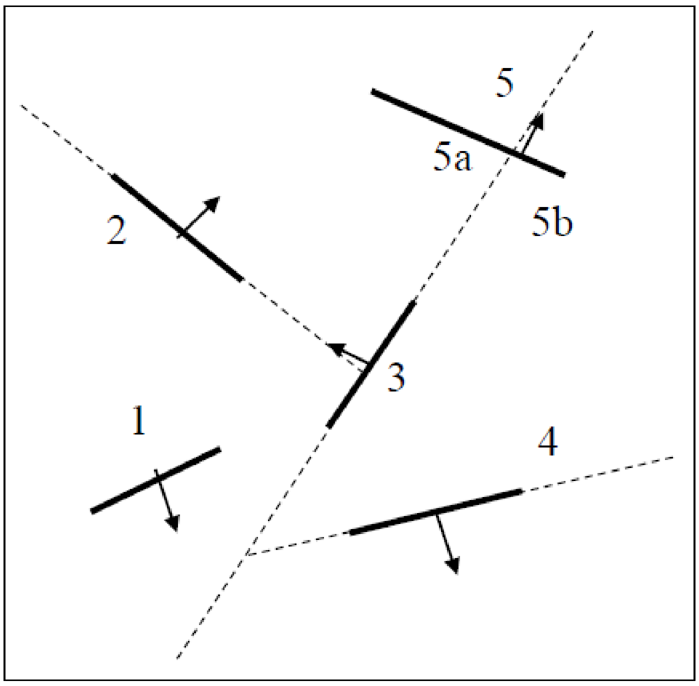


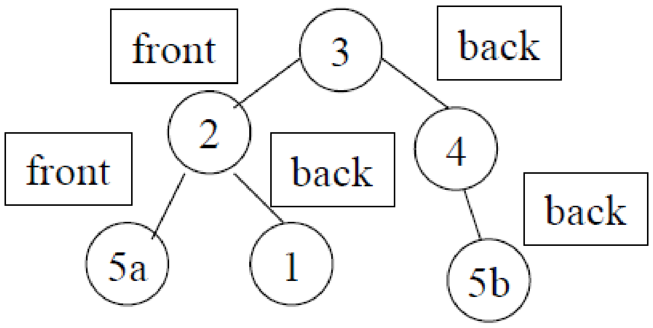
Hidden face removal

Painter's algorithm

•Render polygons a back to front order so that polygons behind others are simply painted over

BSP tree





Display a BSP tree:

•Start at root polygon.

•If viewer is in front half-space, draw polygons behind root first, then the root polygon, then polygons in front.

•If polygon is on edge –either can be used.

•Recursively descend the tree.

Image space approach

Z buffer

Basic Z-buffer idea:

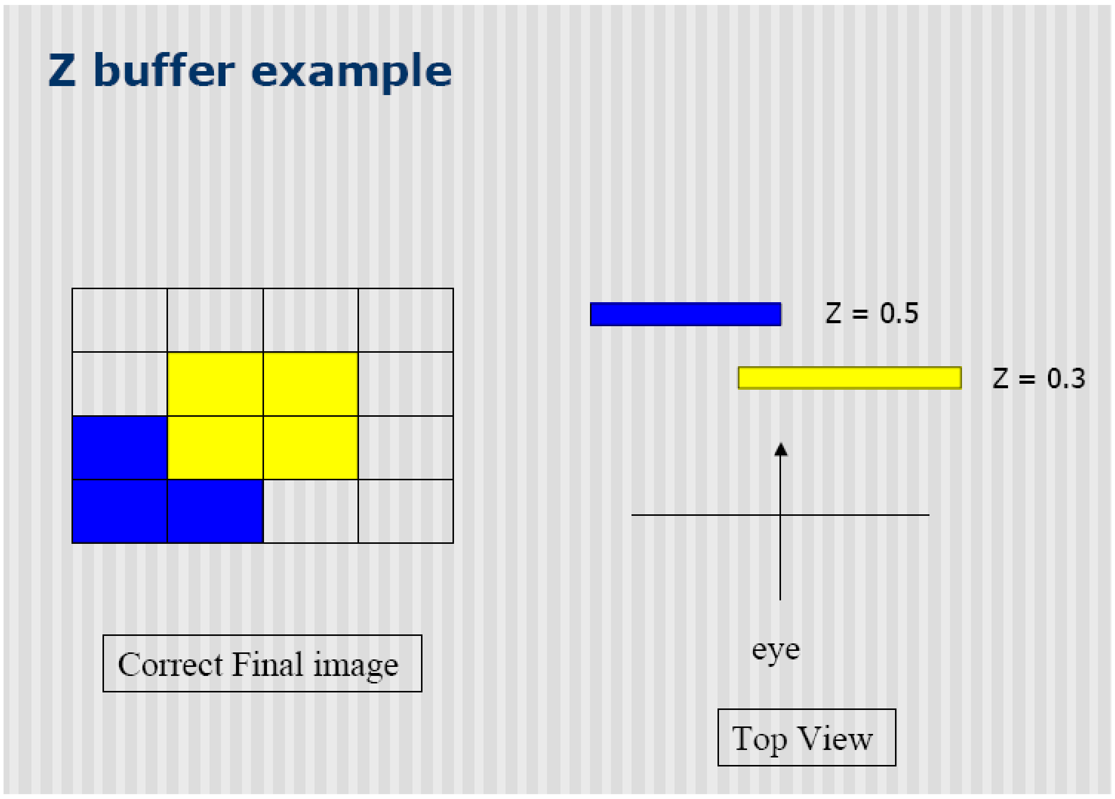
•rasterize every input polygon

•For every pixel in the polygon interior, calculate its corresponding z value (by interpolation)

•Track depth values of closest polygon (smallest z) so far

•Paint the pixel with the color of the polygon whose z value is the closest to the eye.

•Draw in any order, keep track of closest



Z buffer example

Implementation

•Initialise frame buffer to background colour.

•Initialise depth buffer to z = max. value for far clipping plane

•For each triangle

  •Calculate value for z for each pixel inside

  •Update both frame and depth buffer

Filling in triangles

•Scan line algorithm

  •Filling in the triangle by drawing horizontal lines from top to bottom

•Barycentri ccoordinates

•Checking whether a pixel is inside / outside the triangle

Barycentri ccoordinates

Bounding box of the triangle

•First, identify a rectangular region on the canvas that contains all of thepixels in the triangle (excluding those that lie outside the canvas).

•Calculate a tight bounding box for a triangle: simply calculate pixelcoordinates for each vertex, and find the minimum/maximum for each axis

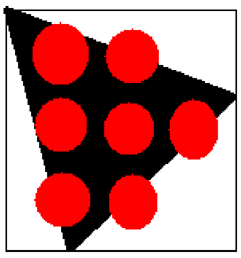
Scanning inside the triangle

•Once we've identified the bounding box, we loop over each pixel in the box.

•For each pixel, we first compute the corresponding (x, y) coordinates in the canonical view volume

•Next we convert these into barycentric coordinates for the triangle being drawn.

•Only if the barycentric coordinates are within the range of [0,1], we plot it (and compute the depth)



Pros and cons

Advantage

•Simple to implement in hardware.

•Memory for z-buffer is now not expensive

•Diversity of primitives –not just polygons.

•Unlimited scene complexity

•Don’t need to calculate object-object intersections.

Disadvantage

•Extra memory and bandwidth

•Waste time drawing hidden objects

Z-precision errors

•May have to use point sampling

9 Rasterization

Vector Graphics

•Algebraic equations describe shapes.

•Can render type and large areas of color with relatively small file sizes

•Can be reduced/enlarged with no loss of quality

•Cannot show continuous tone images (photographs, color blends)

•Examples:

- •Plotters, Oscilloscopes,
- •Illustrator, Flash, PDF

Raster Graphics

•Pixel-based / Bit-mapped graphics

•Grid of pixels

•Size of grid = resolution of image

•Poor scalability : zoom in -loss of quality (jagged look)

•Best for large photographic images

•Modification at pixel level -texture mapping, blending, alpha channels, antialiasing, etc.

•Examples :

- •CRT, LCD, Dot-matrix printers
- •Adobe Photoshop, BMP

Frame Buffer Model

- Raster Display: 2D array of picture elements (pixels)
- Fragments have a location (pixel location) and other attributes such color and texture coordinates that are determined by interpolating values at vertices
- Pixel colors determined later using color, texture, and other vertex properties

Rasterization

•Geometric primitives (point, line, polygon, circle, polyhedron, sphere…)

•Primitives are continuous; screen is discrete

•Scan Conversion: algorithms for *efficient* generation of the samples comprising this approximation

Rasterization

= Scan Conversion

= Converting a continuous object such as a line or a circle into discrete pixels.

Rasterization

•First job: enumerate the pixels covered by a primitive

 •simple, aliased definition: pixels whose centers fall inside

•Second job: interpolate values across the primitive

 •e.g. colors computed at vertices

 •e.g. normal at vertices

Rasterize Lines

DDA

```
dx = x2-x1;dy = y2-y1;
If |dx| > |dy| then step = dx else step = dy;
For(i=0; i<step,i++) {
    x+=dx/steps;
    y+=dy/steps;
```

SetPixel(round(x), round(y));

}

Bresenham's algorithm

•Given x1, y1, x2, y2

•Calculate Δx,Δy

•$d0=2\Delta y-\Delta x$

•SetPixel(x0, y0)

•At each xk, k starts from k=0, do $\Delta x-1$steps

  •If dk<0, SetPixel(xk+1, yk) and

    $dk+1=dk+2\Delta y$

  •If dk>0, SetPixel(xk+1, yk+1) and

    $dk+1=dk+2\Delta y-2\Delta x$

Mid-point algorithm

•Observation

  •$dk+1=Dxk+1+1,yk+1-0.5=(xk+1+1)2+(yk+1+0.5)2-r2$

  •$dk+1=dk+2xk+1+yk+12-yk2-yk+1-yk+1$

  •$yk+1$is either ykor $yk-1$, depending on the sign of dk

•On each iteration:

  •$x+=1$

  •$dk+=2xk+1+1$ if $dk<0$

  •$dk+=2xk+1+1-2yk+1$ if $dk>0$

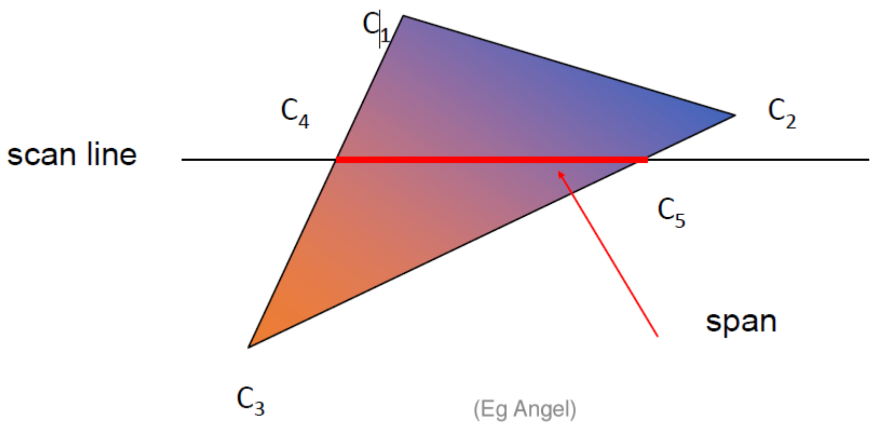
Rasterize polygons

Flood Fill

```
flood_fill(intx, inty) {
   if(read_pixel(x,y)= = WHITE) {
      write_pixel(x,y,BLACK);
      flood_fill(x-1, y);
      flood_fill(x+1, y);
      flood_fill(x, y+1);
      flood_fill(x, y-1);
}}
```

Interpolation for triangles

C_1 C_2 C_3 specified by `glColor` or by vertex shading
C_4 determined by interpolating between C_1 and C_2
C_5 determined by interpolating between C_2 and C_3
interpolate between C_4 and C_5 along span



(Eg Angel)

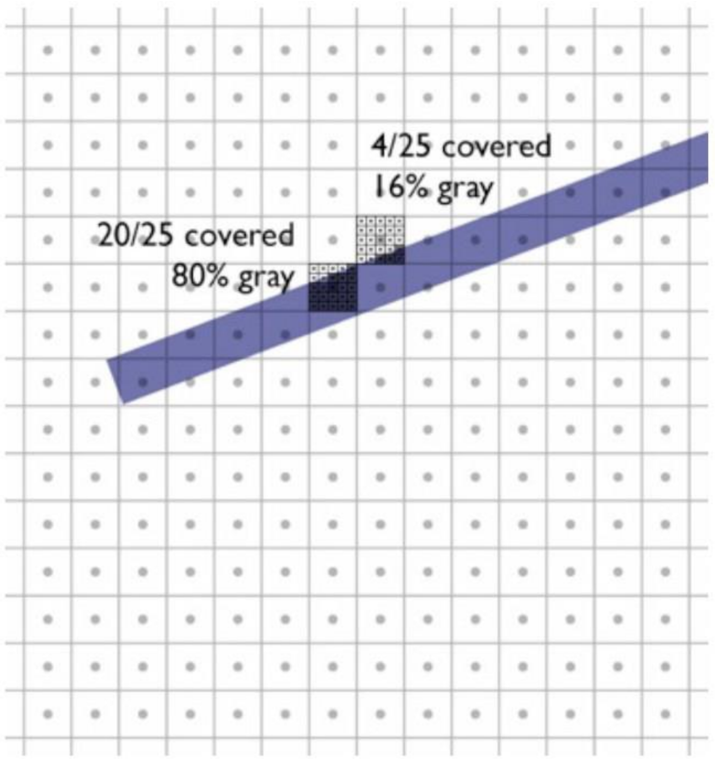
Anti-aliasing

Box filtering

Compute average fraction by counting subpixels

Simple accurate slow

Unweighted filter

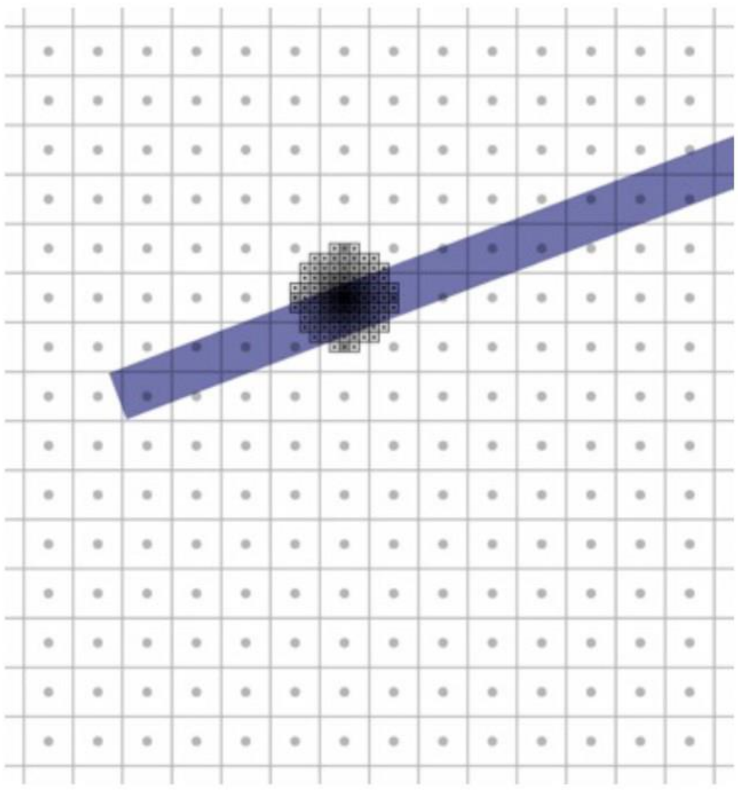


Weighted filtering

Compute filtering integral by summing filter values for covered subpixels

Simple accurate

Really slow

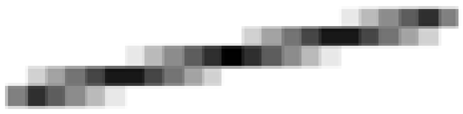


Xiaolin Wu’s line algorithm

Bresenham's algorithm draws lines extremely quickly, but it does not perform anti-aliasing. In addition, it cannot handle any cases where the line endpoints do not lie exactly on integer points of the pixel grid.

A naive approach to anti-aliasing the line would take an extremely long time. Wu's algorithm is comparatively fast, but is still slower than Bresenham's algorithm.

The algorithm consists of drawing pairs of pixels straddling the line, each coloured according to its distance from the line. Pixels at the line ends are handled separately. Lines less than one pixel long are handled as a special case.



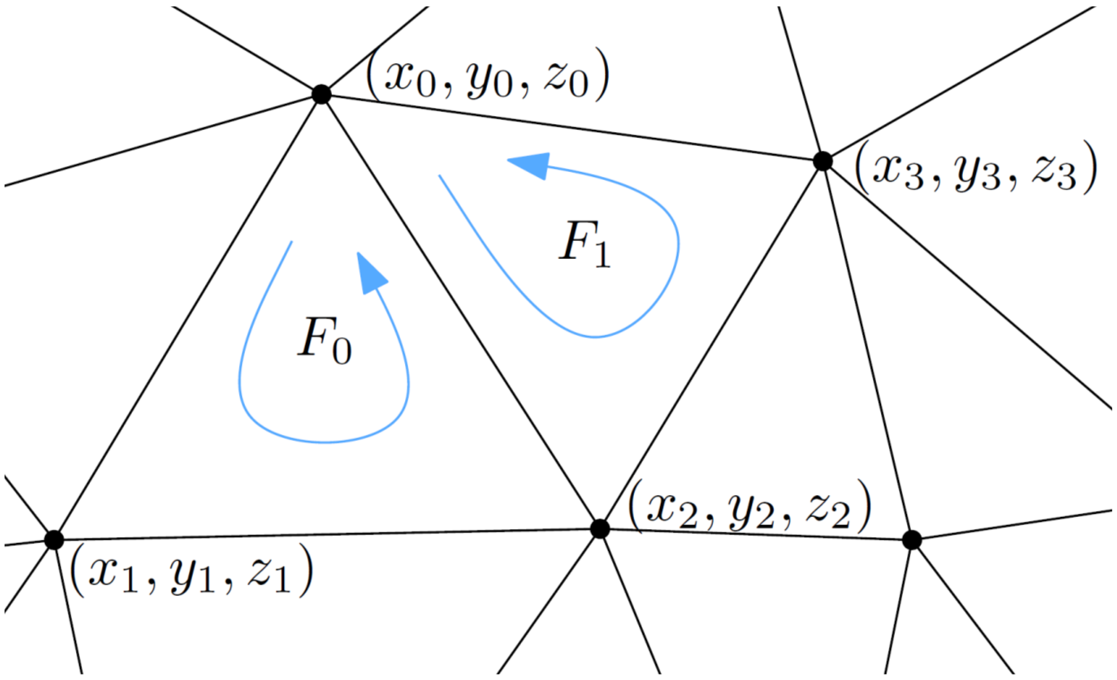
10 Polygon mesh

Mesh data structures

Mesh data structures

- Separate triangles
- Indexed triangle set
- Triangle strips and triangle fans
- Triangle-neighbor data structure
- Winged-edge data structure
- Half-edge data structure

Separate Triangles



Face Table

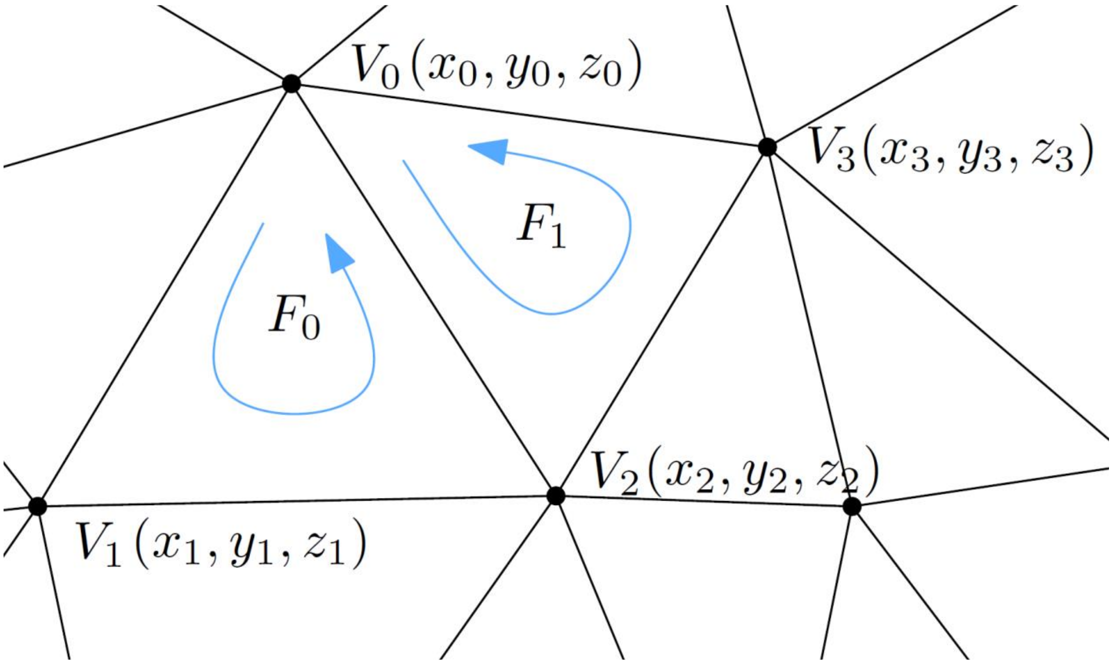
F_0:

$(x_0, y_0, z_0), (x_1, y_1, z_1), (x_2, y$

F_1:

$(x_2, y_2, z_2), (x_3, y_3, z_3), (x_0, y$

Indexed triangle set



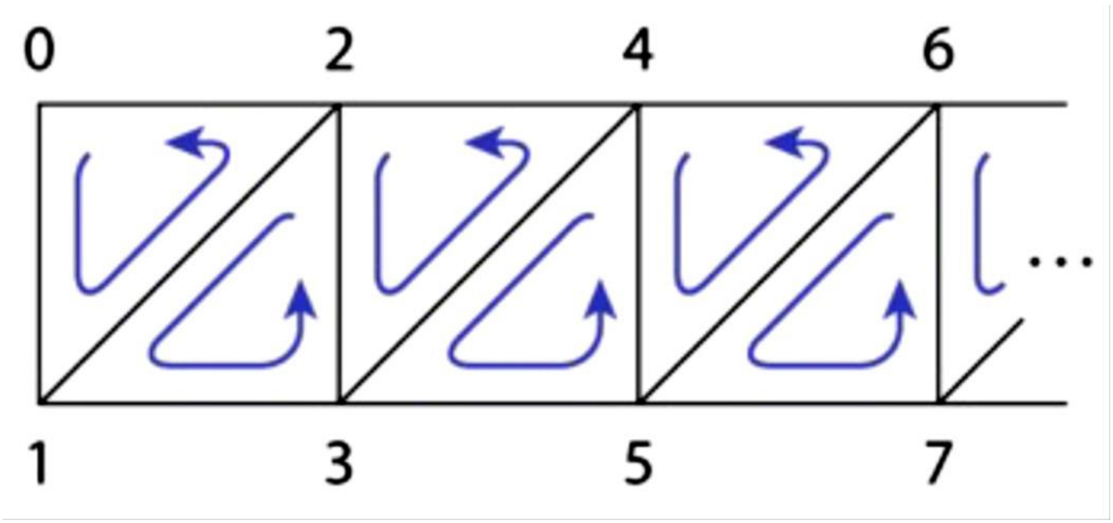
Vertex Table

$V_0: (x_0, y_0, z_0)$
$V_1: (x_1, y_1, z_1)$
$V_2: (x_2, y_2, z_2)$
$V_3: (x_3, y_3, z_3)$

e

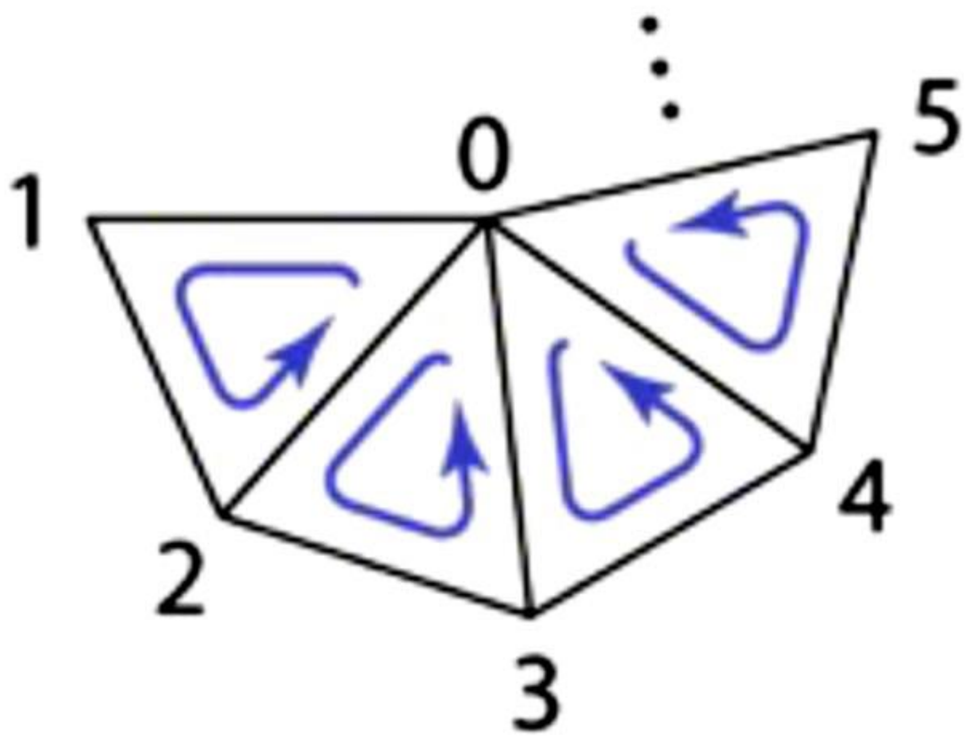
$F_0: V_0, V_1, V_2$
$F_1: V_2, V_3, V_0$

Triangle strips



- Take advantage of the mesh property
  - each triangle is usually adjacent to the previous
  - let every vertex create a triangle by reusing the second and third vertices of the previous triangle
  - e. g., 0, 1, 2, 3, 4, 5, 6, 7, … leads to (0 1 2), (2 1 3), (2 3 4), (4 3 5), (4 5 6), (6 5 7), …
  - for long strips, this requires about one index per triangle

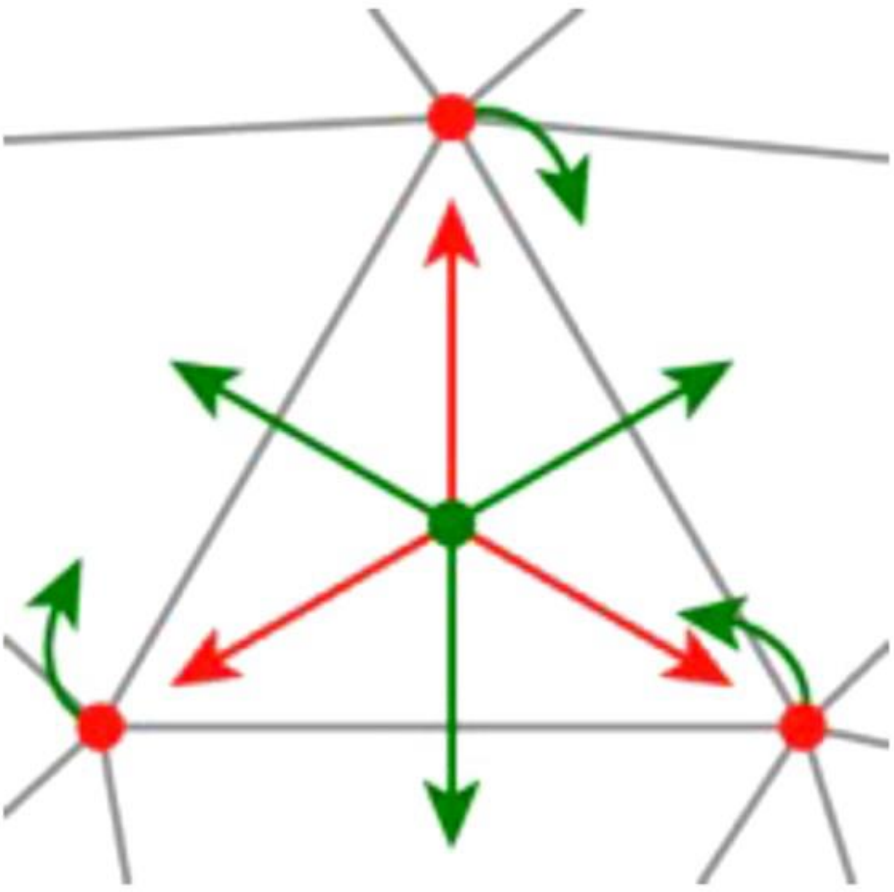
Triangle fans



- Same idea as triangle strips, but keep oldest rather than newest
  - every sequence of three vertices produces a triangle
  - e. g., 0, 1, 2, 3, 4, 5, … leads to (0 1 2), (0 2 3), (0 3 4), (0 3 5), …
  - Memory considerations exactly the same as triangle strip

Triangle-neighbor data structure

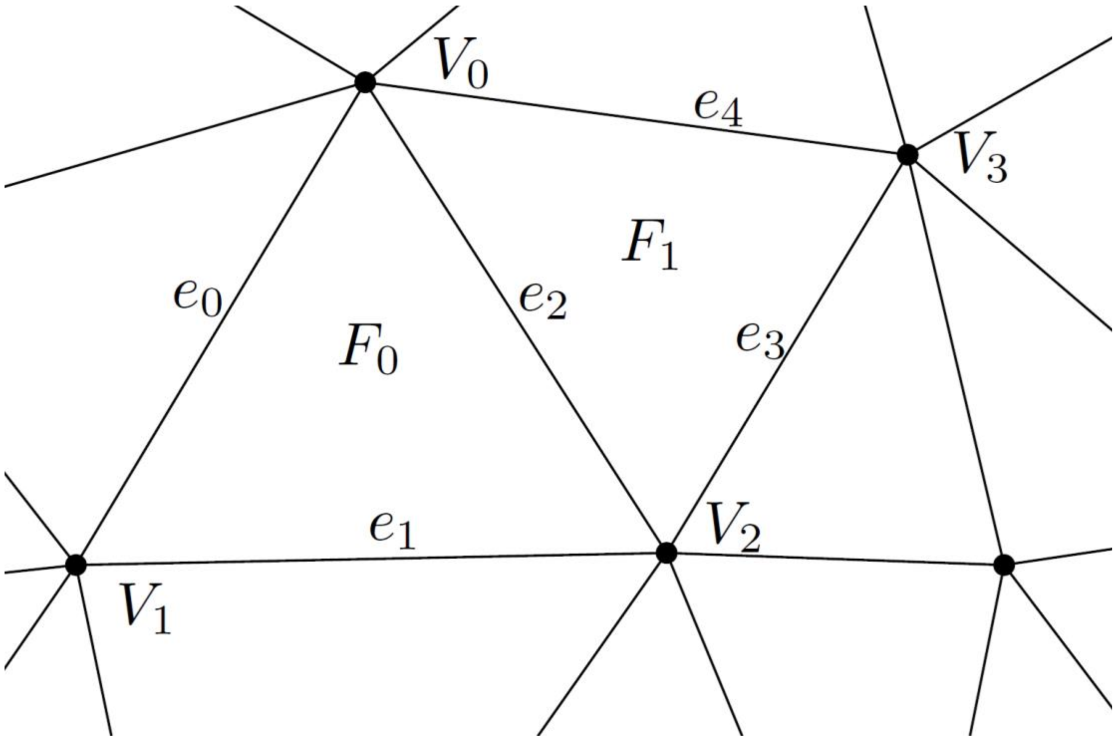
- Extension to indexed triangle set
- Triangle points to its three neighboring triangles
- Vertex points to a single neighboring triangle
- Can now enumerate triangles around a vertex



Winged-edge data structure

· Based on edges

· Store all vertex, face, and edge adjacencies



Edge Adjacency Table

$e_0: V_0, V_1; F_0, \emptyset; \emptyset, e_2, e_1, \emptyset$

$\boxed{e_2: V_2, V_0; F_0, F_1; e_3, e_1, e_0, e_4}$

$e_1: V_1, V_2; F_0, \emptyset; \emptyset, e_0, e_2, \emptyset$

Face Adjacency Table

$\boxed{F_0: V_0, V_1, V_2;\ F_1, \emptyset, \emptyset; e_0, e_1, e_2}$

$F_1: V_2, V_3, V_0;\ F_0, \emptyset, \emptyset; e_2, e_3, e_4$

Vertex Adjacency Table

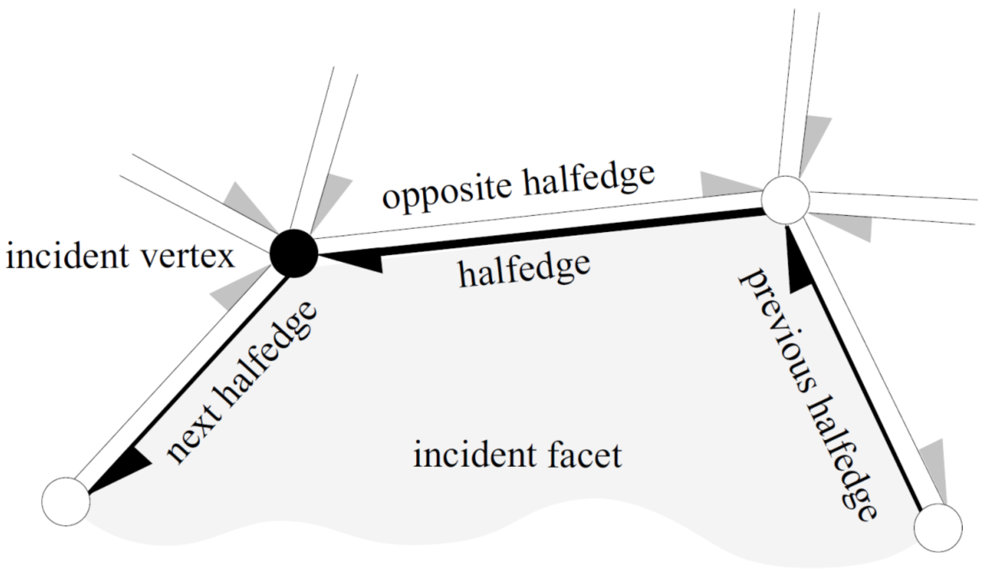
$V_0: V_1, V_2, V_3; F_0, F_1;\ e_0, e_2, e_4$

$V_1: V_2, V_0; F_0;\ e_1, e_0$

Half-edge data structure

· A half-edge data structure is an edge-centered data structure capable of maintaining incidence information Of vertices, edges and faces

· Instead of a single edge, 2 oriented "half edges"



Half Edge Table

h_0: V_2 ; F_1; h_1; h_3

$\boxed{h_2: V_0 ;\ F_1; h_0; h_9}$

h_3: V_0 ; F_0; h_4; h_0

. . .

Face Table

F_0: h_3

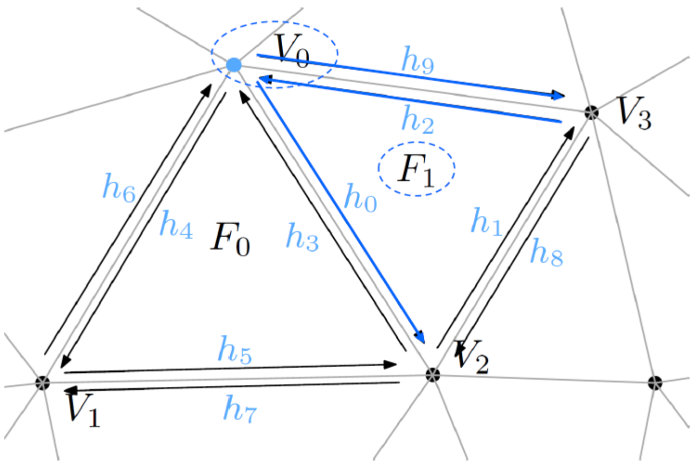
F_1: h_2

. . .

Vertex Table

V_0: h_2

V_1: h_4



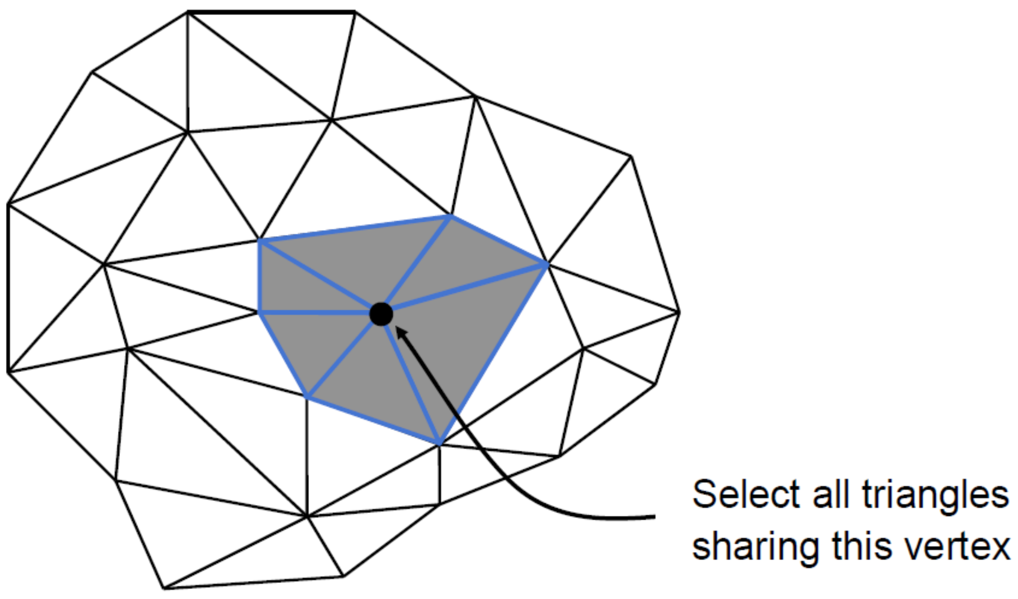
Summary

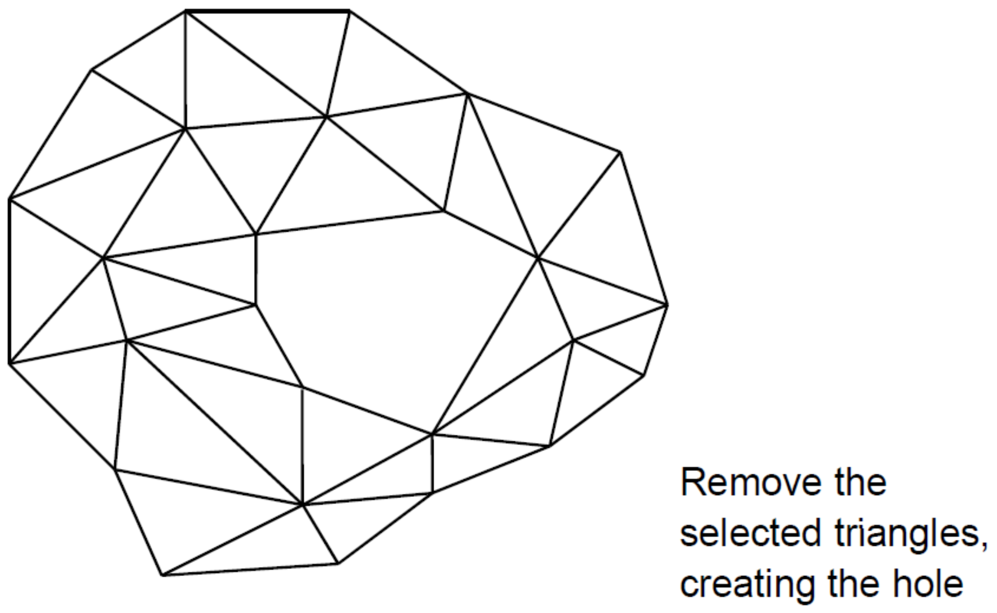
- **For rendering purpose**
  - Triangle strips/fans
  - Independent triangles
  - Indexed triangle sets
- **For query purpose**
  - Winged-edge
  - Half-edge

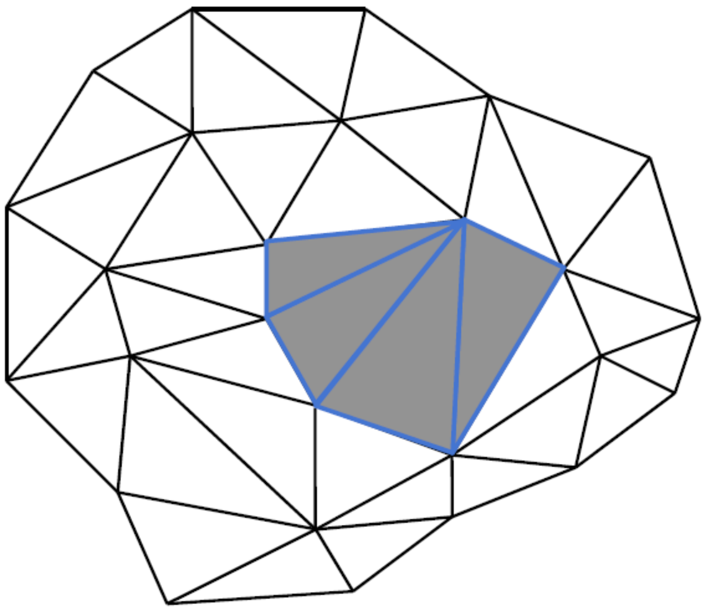
Mesh Decimation Methods

Decimation operators

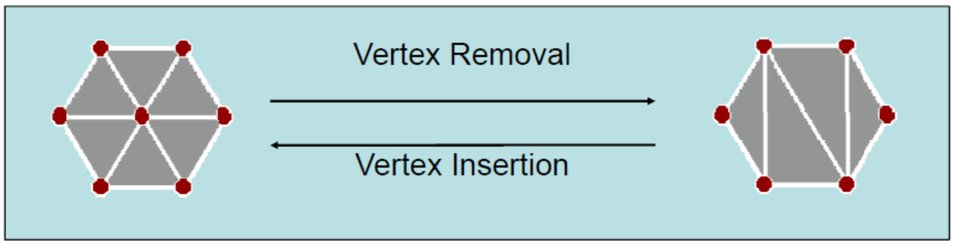
Vertex removal





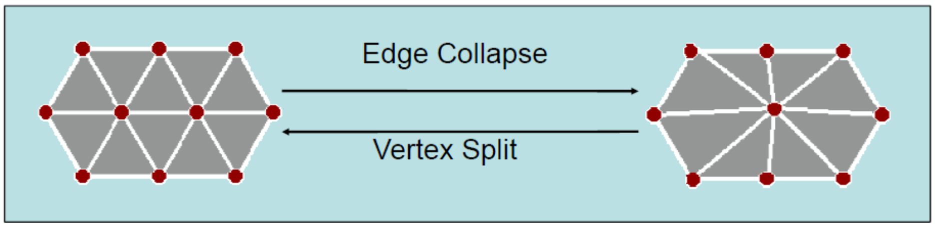


Fill the hole with new triangles



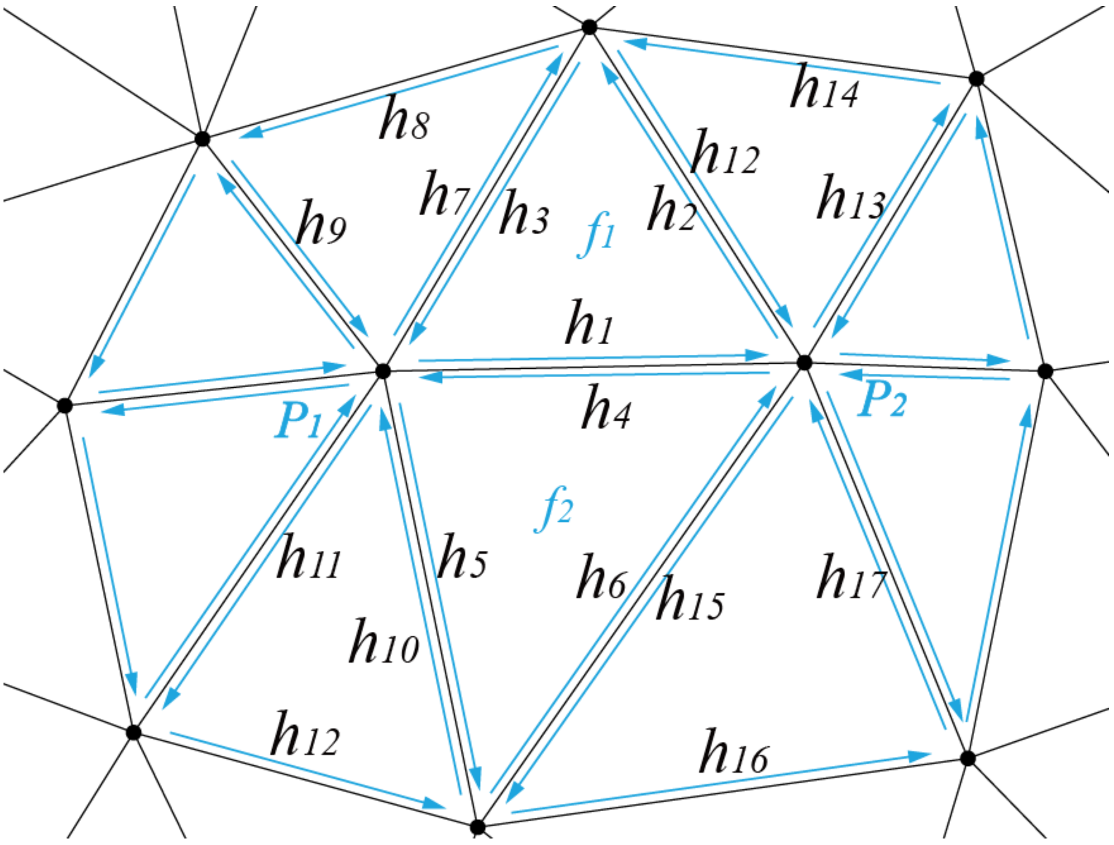
- Remove vertex
- Re-triangulate hole
  - Combinatorial degrees of freedom

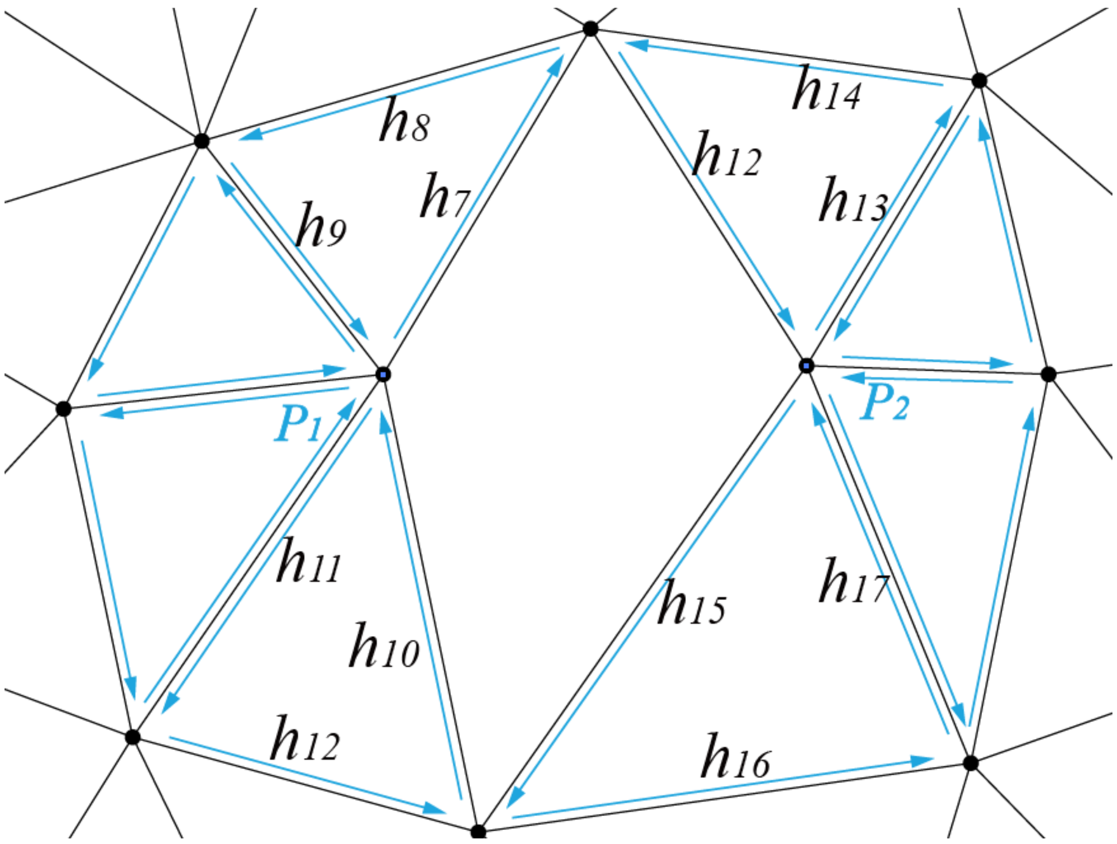
Edge collapse

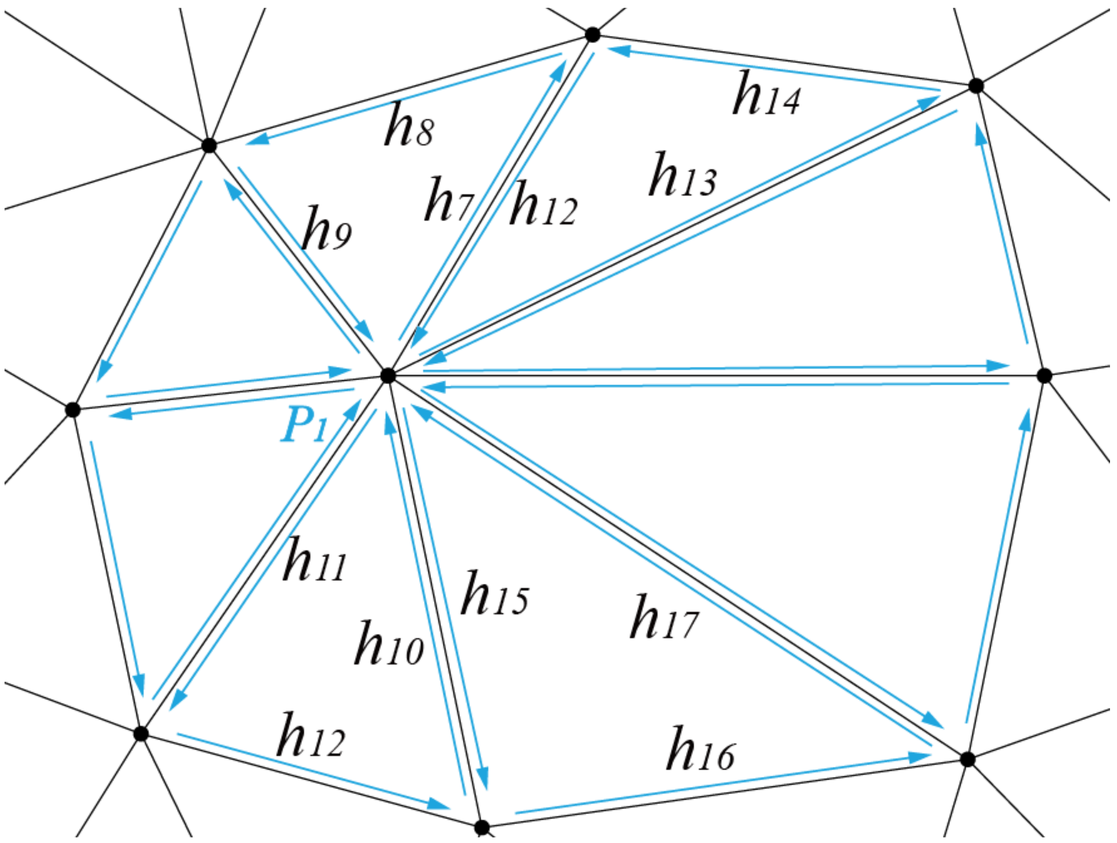


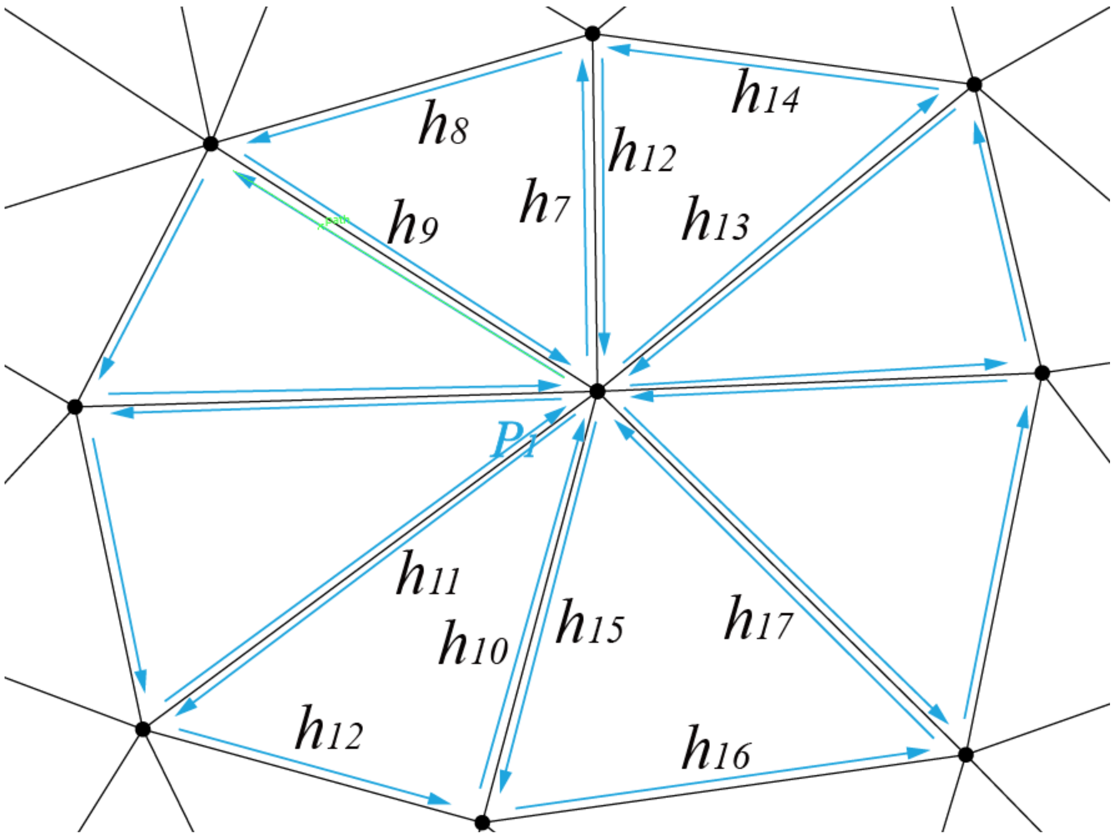
- Merge two adjacent vertices
- Define new vertex position
  - Continuous degrees of freedom
  - Filter along the way

Implementation:

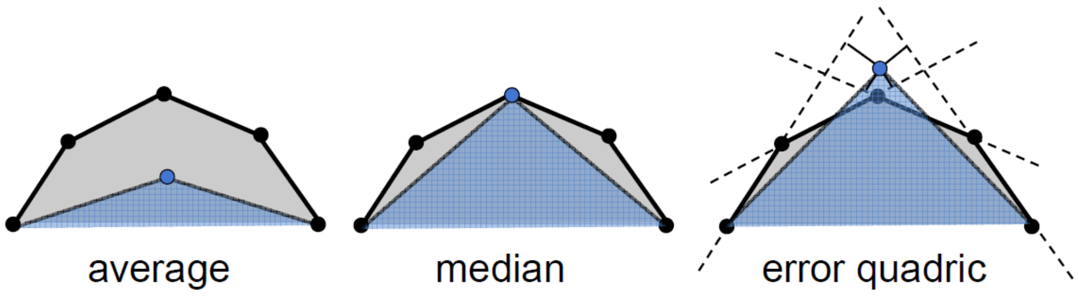



h_8
h_7
h_9
h_{14}
h_{12}
h_{13}
P_1
P_2
h_{11}
h_{10}
h_{15}
h_{17}
h_{12}
h_{16}


h_8
h_{14}
h_7
h_{12}
h_{13}
h_9
P_1
h_{11}
h_{15}
h_{17}
h_{10}
h_{12}
h_{16}



Quadric error metric



11 Spline Curves

Interpolation splines

Hermite spline curve

$$x(t) = at^3 + bt^2 + ct + d$$
$$x'(t) = 3at^2 + 2bt + c \qquad d = x_0$$
$$x(0) = x_0 = d \qquad c = x'_0$$
$$x(1) = x_1 = a + b + c + d \qquad a = 2x_0 - 2x_1 + x'_0 + x'_1$$
$$x'(0) = x'_0 = c \qquad b = -3x_0 + 3x_1 - 2x'_0 - x'_1$$
$$x'(1) = x'_1 = 3a + 2b + c$$

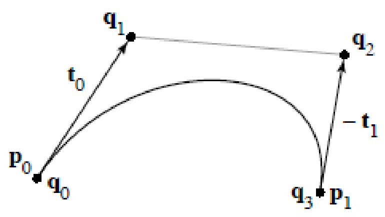
$$\mathrm{f}(t) = \mathrm{a}t^3 + \mathrm{b}t^2 + \mathrm{c}t + \mathrm{d}$$

$$[p_0\; p_1\; t_0\; t_1]\begin{bmatrix} 2 & -3 & 0 & 1 \\ -2 & 3 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix}\begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}$$

$$f(t) = b_0(t)p_0 + b_1(t)p_1 + b_2(t)p_2 + b_3(t)p_3$$

Approximation spline

Bezier spline



$$\mathrm{p}_0 = \mathrm{q}_0$$
$$\mathrm{p}_1 = \mathrm{q}_3$$
$$\mathrm{t}_0 = 3(\mathrm{q}_1 - \mathrm{q}_0)$$
$$\mathrm{t}_1 = 3(\mathrm{q}_3 - \mathrm{q}_2)$$

- $[p_0\ p_1\ t_0\ t_1] = [q_0\ q_1 q_2\ q_3]\begin{bmatrix}1 & 0 & -3 & 0\\ 0 & 0 & 3 & 0\\ 0 & 0 & 0 & -3\\ 0 & 1 & 0 & 3\end{bmatrix}$

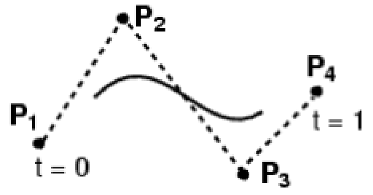
- $Q(t) = [q_0\ q_1\ q_2\ q_3]\begin{bmatrix}-1 & 3 & -3 & 1\\ 3 & -6 & 3 & 0\\ -3 & 3 & 0 & 0\\ 1 & 0 & 0 & 0\end{bmatrix}\begin{bmatrix}t^3\\ t^2\\ t\\ 1\end{bmatrix}$

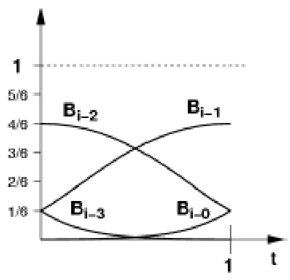
- These are defined as Bernstein polynomials

$$B_i^n(t) = \frac{n!}{i!\,(n-i)!}t^i(1-t)^{n-i}, \qquad 0 \le i \le n$$

$$Q(t) = GBT(t) = Geometry\ G \cdot Spline\ Basis\ B \cdot Power\ Basis\ T(t)$$

BSpline



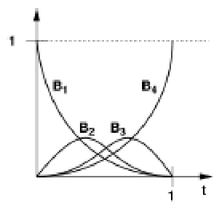


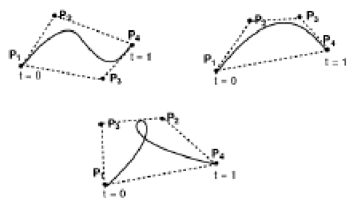
$$Q(t) = \frac{(1-t)^3}{6}P_{i-3} + \frac{3t^3-6t^2+4}{6}P_{i-2} + \frac{-3t^3+3t^2+3t+1}{6}P_{i-1} + \frac{t^3}{6}P_i$$

$$Q(t) = \mathbf{GBT(t)} \qquad B_{B-Spline} = \frac{1}{6}\begin{pmatrix}-1 & 3 & -3 & 1\\ 3 & -6 & 3 & 0\\ -3 & 0 & 3 & 0\\ 1 & 4 & 1 & 0\end{pmatrix}$$

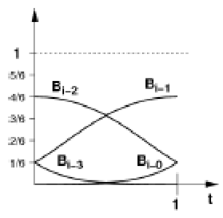
Bezier vs BSpline

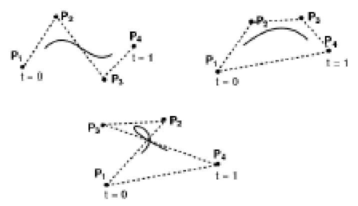
Bézier





BSpline





$$B_{Bezier} = \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

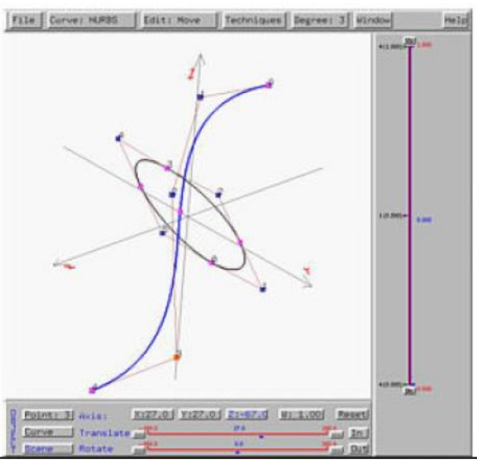
$$B_{B-Spline} = \frac{1}{6}\begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{pmatrix}$$

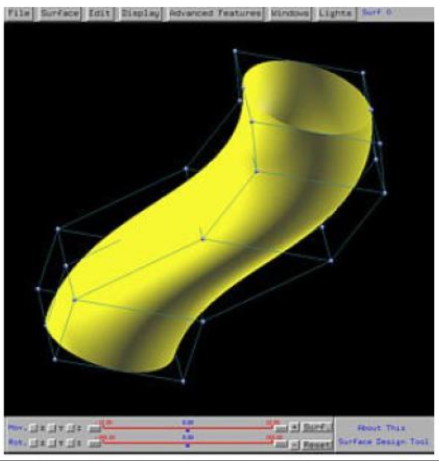
$$Q(t) = \mathbf{GBT(t)} = \text{Geometry } \mathbf{G} \cdot \text{Spline Basis } \mathbf{B} \cdot \text{Power Basis } \mathbf{T(t)}$$

12 Spline surface

Sweeping

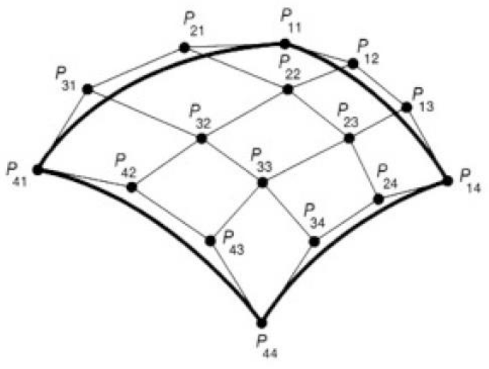
- Surface defined by a *cross section* moving along a *spine*
- Simple version: a single 3D curve for spine and a single 2D curve for the cross section



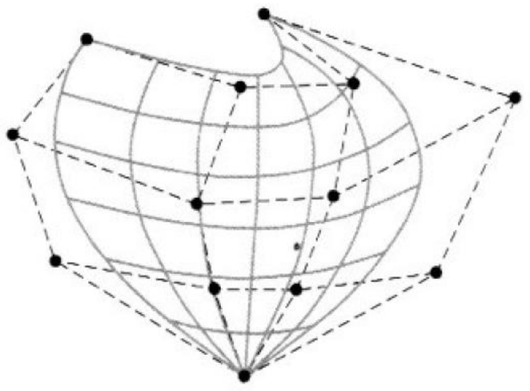


Bezier patch

- Cross product of two cubic Bézier segments



[Foley et al.]



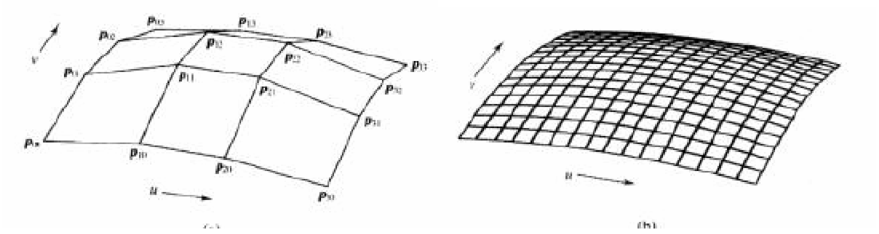
(b)

[Hearn & Baker]

Notation: $\mathbf{CB}(P_1, P_2, P_3, P_4, \alpha)$ is Bézier curve with control points P_i evaluated at α

Define "Tensor-product" Bézier surface

$$
\begin{aligned}
Q(s,t) = \mathbf{CB}(\quad & \mathbf{CB}(P_{00}, P_{01}, P_{02}, P_{03}, t), \\
& \mathbf{CB}(P_{10}, P_{11}, P_{12}, P_{13}, t), \\
& \mathbf{CB}(P_{20}, P_{21}, P_{22}, P_{23}, t), \\
& \mathbf{CB}(P_{30}, P_{31}, P_{32}, P_{33}, t), \\
& s)
\end{aligned}
$$



Subdivision surface

· **Two parts to subdivision process**

· **Subdividing the mesh (computing new topology)**

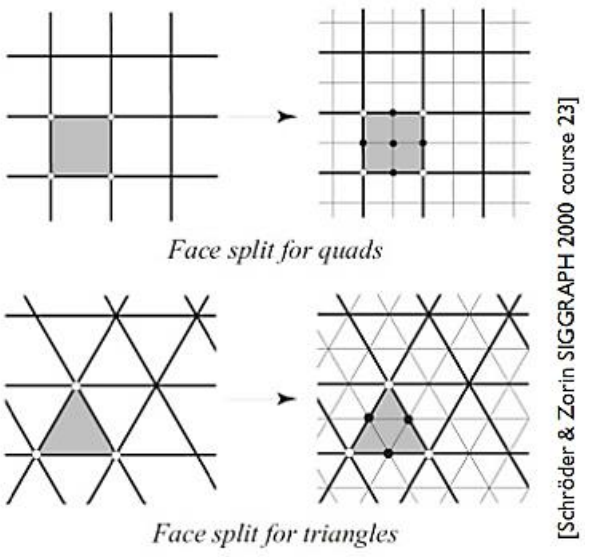
· For curves. replace every segment with two segments

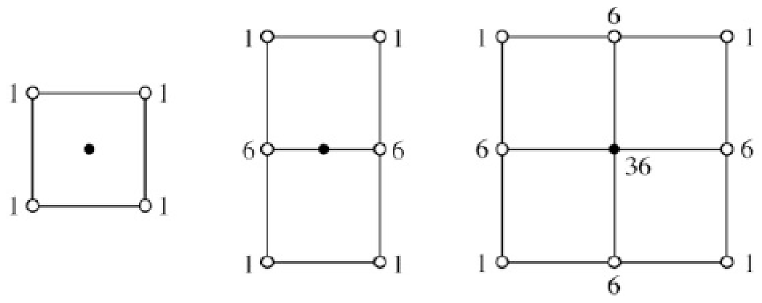
· For surfaces. replace every face with some new faces

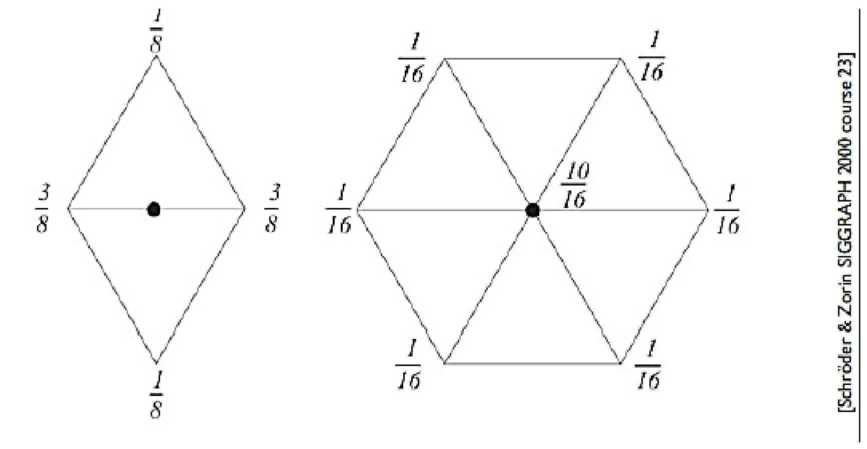
· **Positioning the vertices (computing new geometry)**

· For curves: two rules (one for odd vertices, one for every new vertex's position is a weighted average of positions of old vertices that are nearby along the sequence

· For surfaces. two kinds of rules (still called odd and even) new vertex's position is a weighted average of positions of old vertices that are nearby in the mesh







13 Illumination

What is shading

- **Variation in observed color across an object**
    - strongly affected by lighting
    - present even for homogeneous material

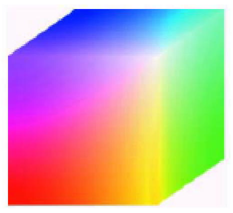
- **caused by how a material reflects light**
  - depends on
    - geometry
    - lighting
    - material

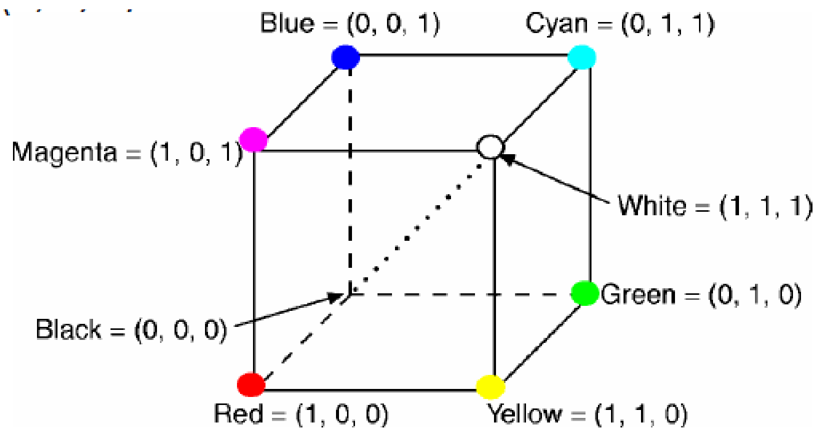
What is material

In the real world, each object reacts differently to light. Steel objects are often shinier than a clay vase for example and a wooden container does not react the same to light as a steel container. Each object also responds differently to specular highlights. Some objects reflect the light without too much scattering resulting in a small highlights and others scatter a lot giving the highlight a larger radius. If we want to simulate several types of objects in OpenGL we have to define material properties specific to each object.

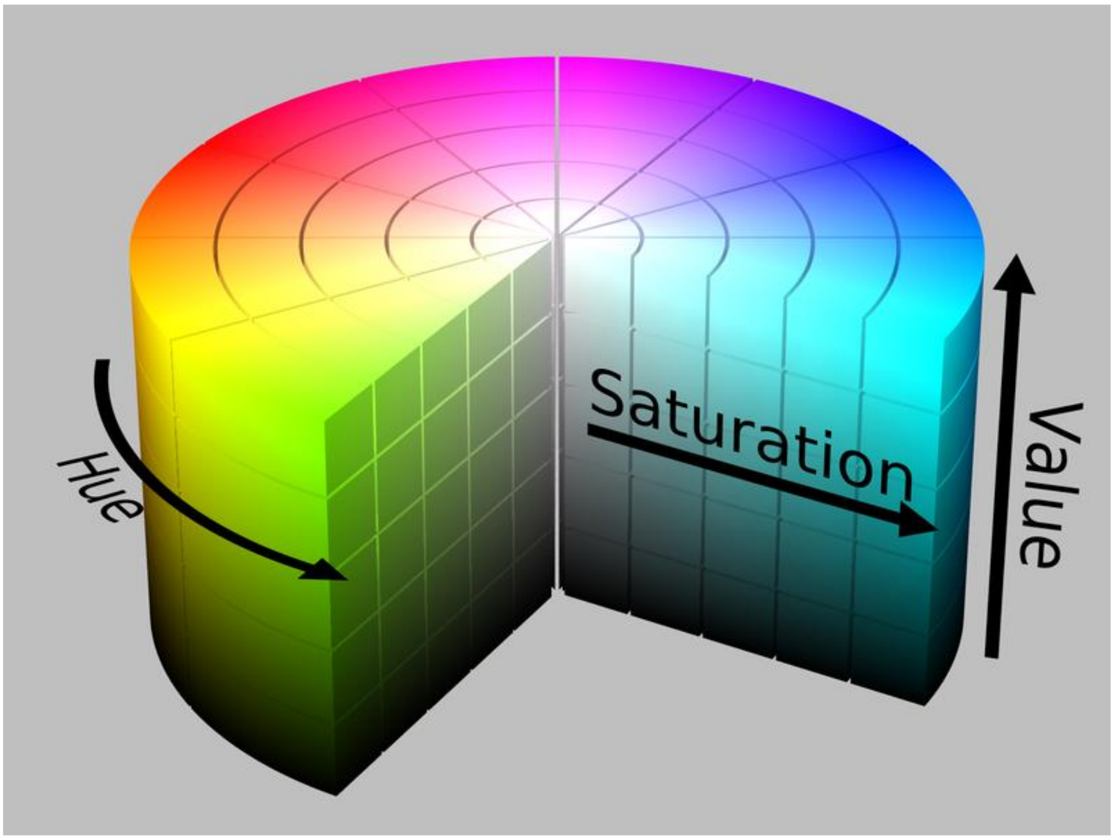
Color Model

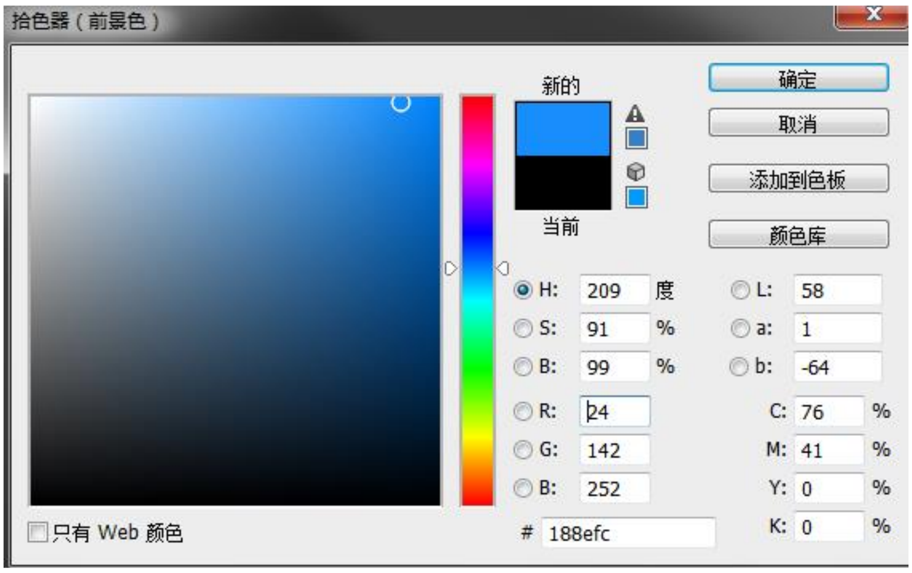
RGB





HSV





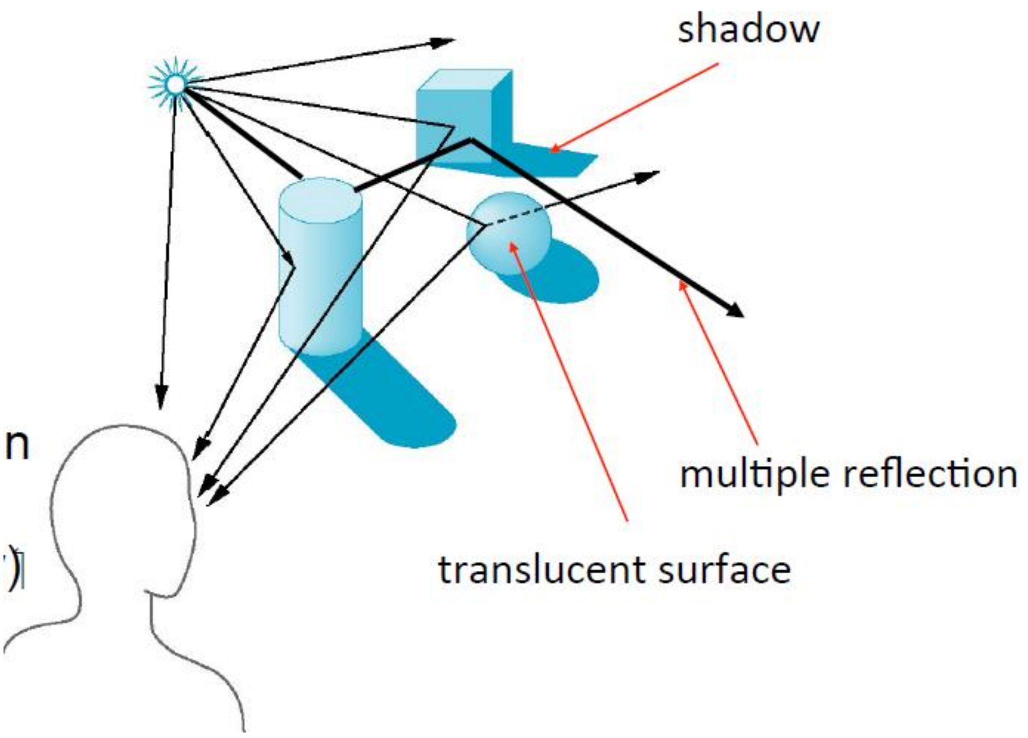
[✓ H:]	209	度	L:	58		
S:	91	%	a:	1		
B:	99	%	b:	-64		
R:	24		C:	76	%	
G:	142		M:	41	%	
B:	252		Y:	0	%	
#	188efc		K:	0	%	

☐ 只有 Web 颜色

Illumination Model

Global illumination

- Ray-tracing
- Radiosity
- Photon Mapping
- Can handle
    - Reflection (one object in another)
    - Refraction (Snell's Law)
    - Shadows
    - Color bleeding
- More computation and slow



Local illumination

- Gouraud shading
- Phong shading
- Shadow techniques
- Can approximate GI!

- Ambient occlusion

- Image based lighting

- Fast and real-time

- Not as accurate as GI

Light sources

Point Light

- The most simple one
- It is omni-directional
- Attributes to specify a point light source
    - Position (*px, py, pz*)
    - Intensity *I* (if it is a chromatic light, three values representing R, G, and B are needed (*Ir*, *Ig*, *Ib*))
    - Coefficients (*a*0, *a*1, *a*2) to specify its attenuation property with distance *d*

$$f_{radatten} = \frac{1}{a_0 + a_1 d + a_2 d^2}$$

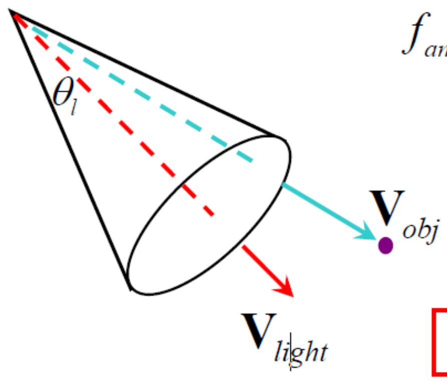
$$I_d = f_{radatten} I$$

Directional light

- This type can be imagined as a point light source lying in infinity, e.g. Sun light
- Light rays from such a source are radiated in parallel.
- Attributes to specify a directional light source
    - Direction *V* (vx, vy, vz)
    - Intensity *I*
    - No attenuation Why?

Spotlight

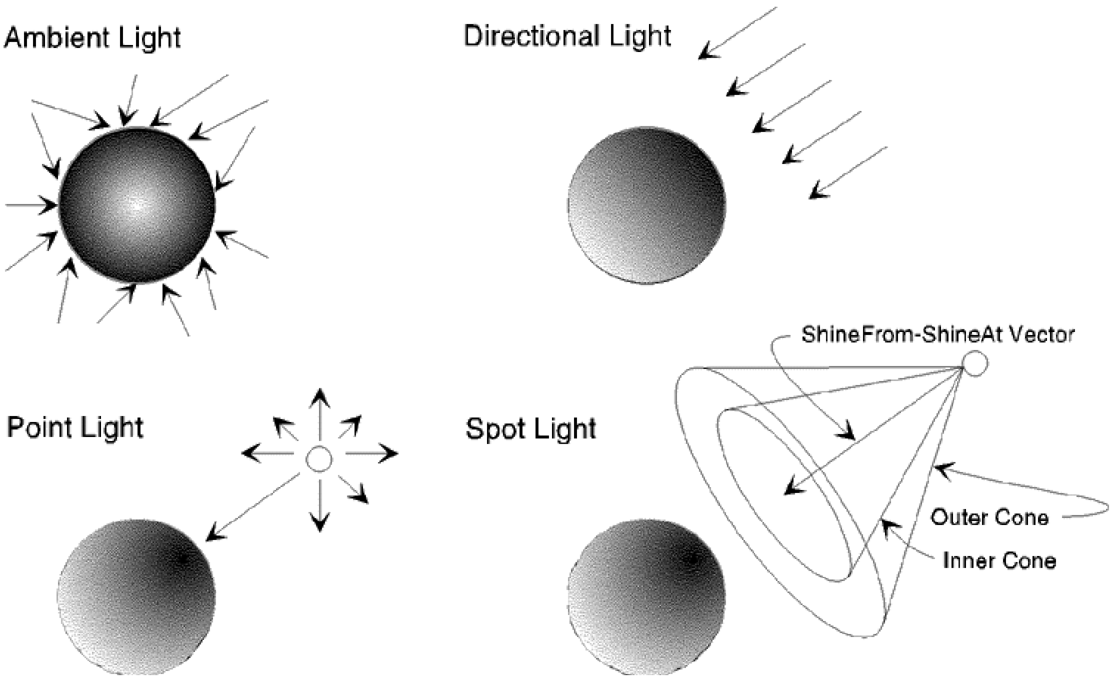
- Besides position and color
- the apex angle of the lighting cone needs to be specified
- (*px, py, pz*), *I*, (*tx*, *ty*, *tz*) and Theta



$$f_{angatten} = \begin{cases} 0.0, \text{when } \mathbf{V}_{obj} \cdot \mathbf{V}_{light} < \cos\theta_l \\ \left(\mathbf{V}_{obj} \cdot \mathbf{V}_{light}\right)^{\alpha}, \text{otherwise} \end{cases}$$

where α is a constant

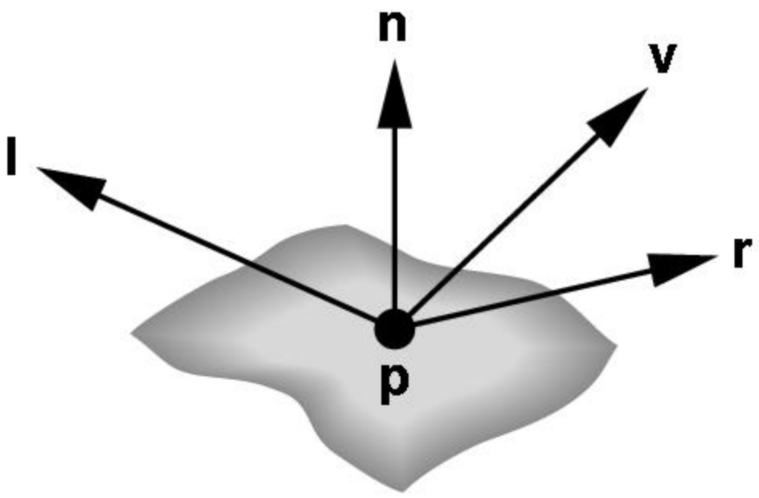
$\boxed{f_{radatten}, f_{angatten} \text{ can be combined}}$



Phong reflection model

- Empirical Model
- Calculate color for arbitrary point on surface
- Basic inputs are material properties and l, n, v:
    - l = vector to light source

- n = surface normal
- v = vector to viewer
- r = reflection of l at p (determined by l and n)

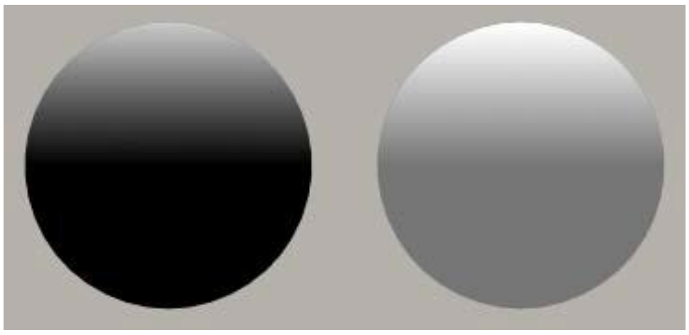


Ambient reflection

- Ambient reflection is a constant for a scene *La*
- Different surface can have different ambient reflection coefficient *ka* (0 ≤ *ka* ≤ 1)
- So, if only consider ambient lighting, the illumination at a point simply is *Ia* = *La* ka*

Diffuse reflection

- I = La*ka +Ld*kd*max(l*n ,0)
- Lack of highlight!



Specular reflection

- Is = ks Ls cosαφ
- cosφ = r∗v
- What is α?

Summary

- Light components
    - Ambient Ia, Diffuse Id and Specular Is
- Material coefficients for each light component
    - ka, kd and ks
- Therefore:
    - I = Laka + Ldkd max(l∗n ,0) +Lsks(r∗v)α

- Attenuation
- $I_{atten} = I_a + \frac{1}{a_0 + a_1 d + a_2 d^2} (I_d + I_s)$
- How about If there are multiple lights?
- $I = L_a k_a + \sum_{i=1}^{m} (L_{d,i}\, k_d \max(l \cdot n, 0) + L_{s,i}\, k_s (r_i \cdot v)^\alpha)$
- How about shadow?
- $I = L_a k_a + s_i \sum_{i=1}^{m} (L_{d,i}\, k_d \max(l \cdot n, 0) + L_{s,i}\, k_s (r_i \cdot v)^\alpha)$

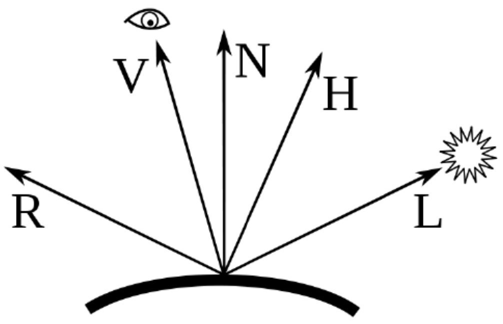
Blinn-Phong reflection model

• A modification to the Phong reflection model developed by Jim Blinn

• In Phong model, one must continually recalculate the dot product or r and v. Instead, one calculates a halfway vector between the viewer and light-source vectors

$$\mathbf{H} = \frac{\mathbf{L} + \mathbf{V}}{\|\mathbf{L} + \mathbf{V}\|}$$

• $L_s k_s (r \cdot v)^\alpha = L_s k_s (n \cdot h)^\alpha$



14 Surface Rendering

• Common used shading algorithms
  • Flat shading
  • Smooth shading

- Gouraud shading
- Phong shading

Flat shading

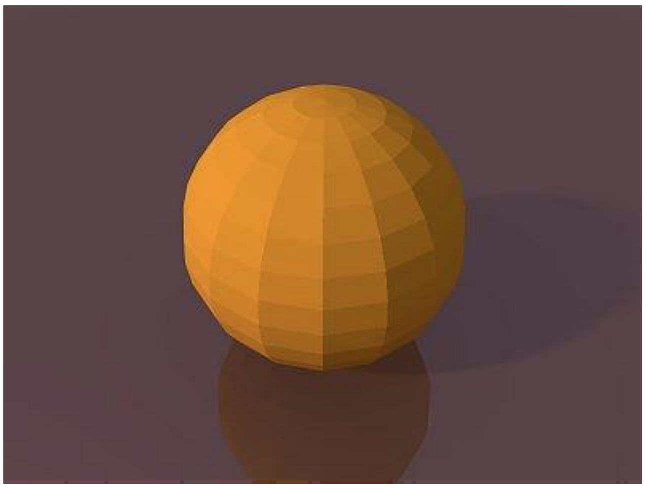
• The simplest one, also called as “constant intensity surface rendering”

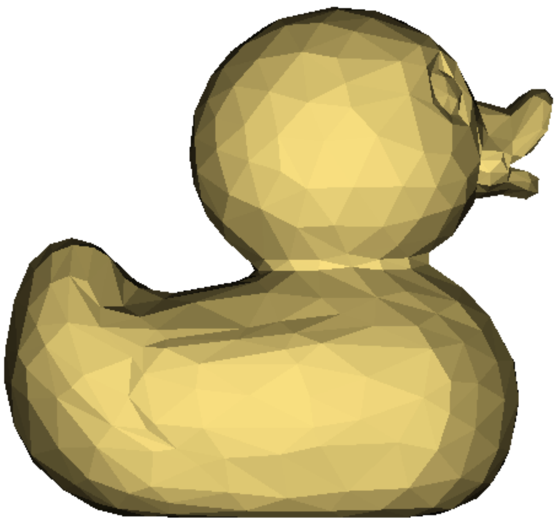
• One illumination calculation per polygon

• Assign all pixels inside each polygon the same color, therefore reveal polygons

and fast

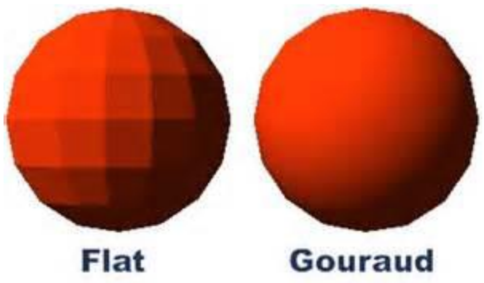
• OK for polyhedral objects, Not good for smooth surfaces





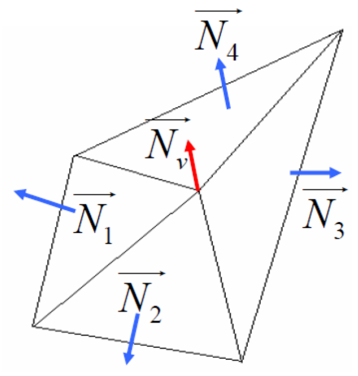
Gourand shading

- Named after Henri Gouraud (1971)
- Produce continuous shading of surfaces represented by polygon meshes
- An interpolation method

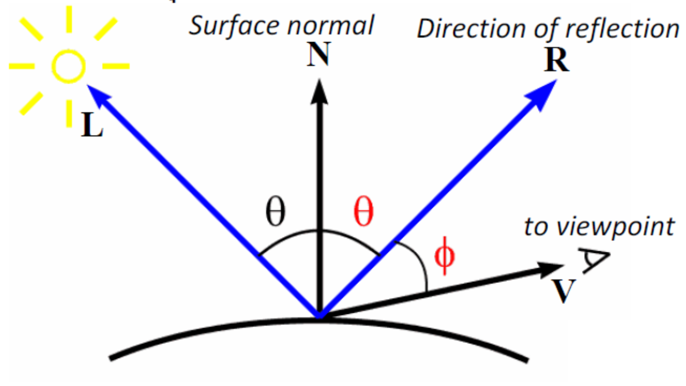


- Step 1: Normal averaging
  - estimate the normal of each vertex by averaging the surface normals of the polygons that meet at each vertex

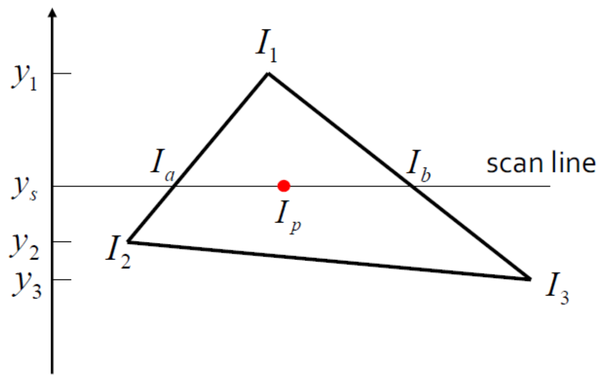
$$\overrightarrow{N_v} = \frac{\sum_{k=1}^{n} \overrightarrow{N_k}}{\left|\sum_{k=1}^{n} \overrightarrow{N_k}\right|}$$



- Step 2: Vertex Lighting
  - compute the color for each vertex based on a shading model (e.g., Phong reflection model)
  - $I = L_a k_a + \sum_{i=1}^{m} (L_{d,i} k_d \max(l \bullet n, 0) + L_{s,i} k_s (r_i \bullet v)^{\alpha})$



- Step3: Interpolation
  - for each **screen pixel** that represents the polygonal mesh, color is interpolated from the color values calculated at the vertices
  - Bilinear interpolation or barycentric coordinates



$$I_a = I_1 \frac{y_s - y_2}{y_1 - y_2} + I_2 \frac{y_1 - y_s}{y_1 - y_2}$$

$$I_b = I_1 \frac{y_s - y_3}{y_1 - y_3} + I_3 \frac{y_1 - y_s}{y_1 - y_3}$$

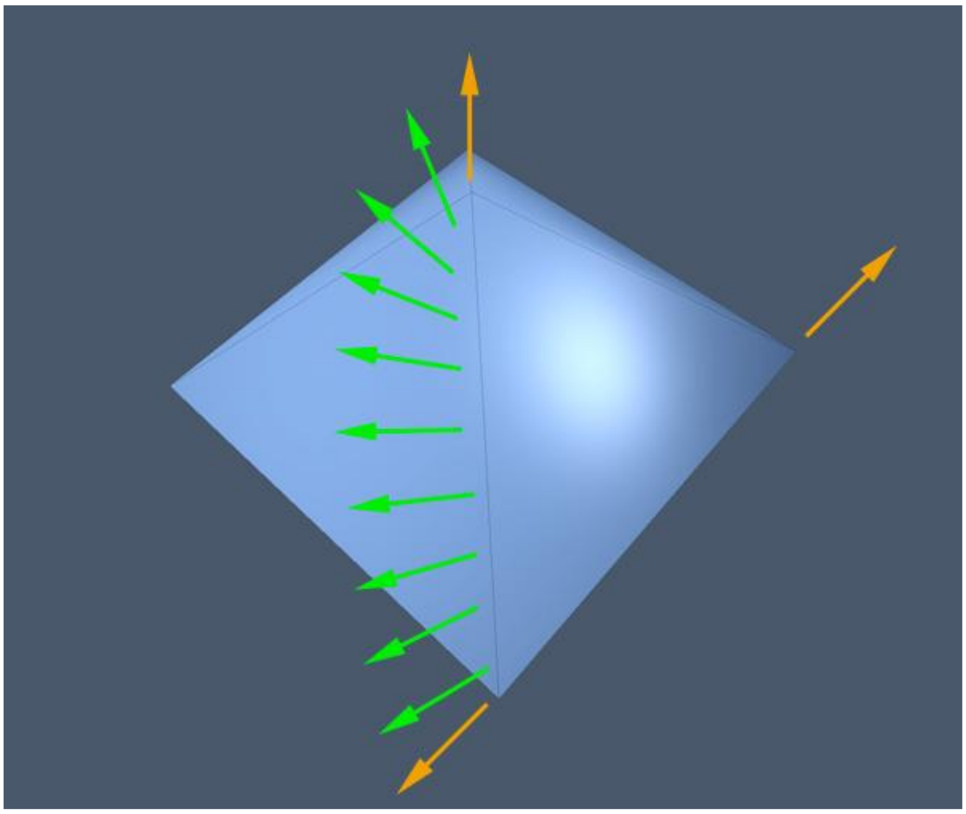
$$I_p = I_a \frac{x_b - x_p}{x_b - x_a} + I_b \frac{x_p - x_a}{x_b - x_a}$$

Phong shading

- Named after Bui Tuong Phong (1973)
- Also produce continuous shading of surfaces represented by polygon meshes
- Also an interpolation method
  - Different from Gouraud shading, Phong shading interpolate normals rather

than colors

• It interpolates surface normals across **rasterized polygons** and computes pixel colors based on the interpolated normals and a reflection model



• Always captures specular highlights, but computationally expensive

• At each pixel, *n* is recomputed and normalized Then *I* is computed at each pixel (lighting model is more expensive than interpolation algorithms)

• Not available in fixed pipeline, but now can be implemented in hardware (per fragment shading)

15 Texture Mapping

What is texture mapping

•Mapping techniques are implemented at the end of the rendering pipeline

•Very efficient because polygons already passed the clipper

How to map a texture on to a geometry?

Three steps to applying a texture

1. specify the texture
   - read or generate image
   - assign to texture
   - enable texturing
2. assign texture coordinates to vertices
   - Proper mapping function is left to application
3. specify texture parameters
   - wrapping, filtering

What is texture coordinate?

In order to map a texture to the triangle we need to tell each vertex of the triangle which part of the texture it corresponds to.

Each vertex should thus have a texture coordinate associated with them that specifies what part of the texture image to sample from.

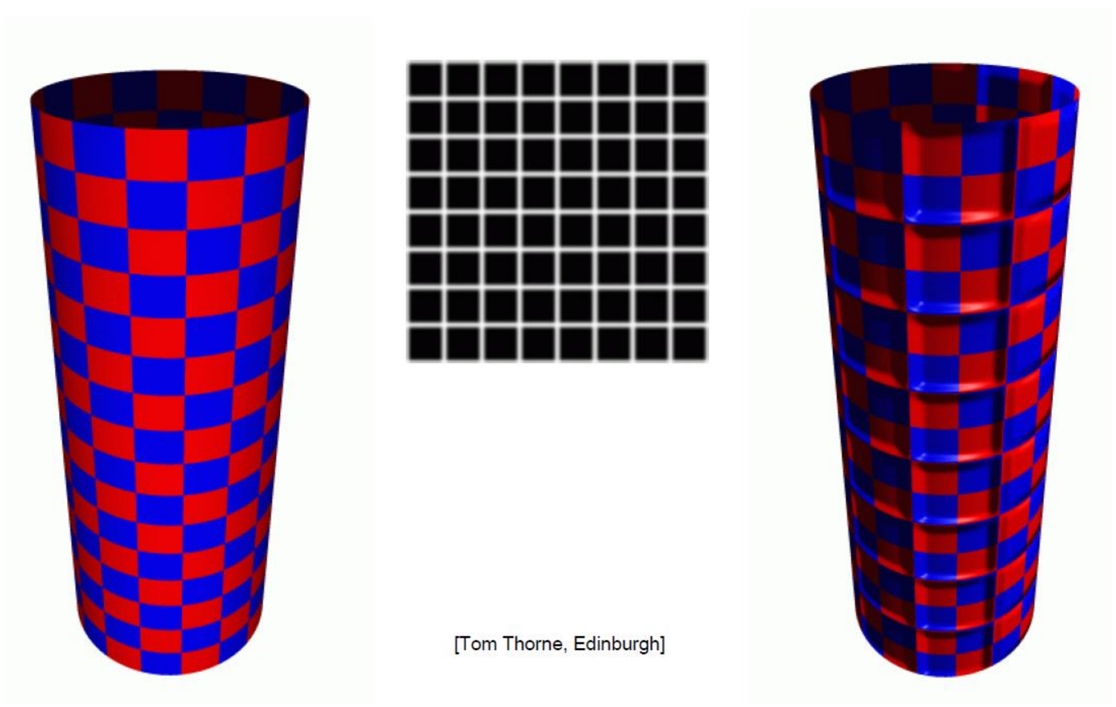
Bump Mapping

•Recall that the normal defines the shading and a plane only have one normal for the wall

•We can use texture to disturb the normal!

•Treat the texture as a height function

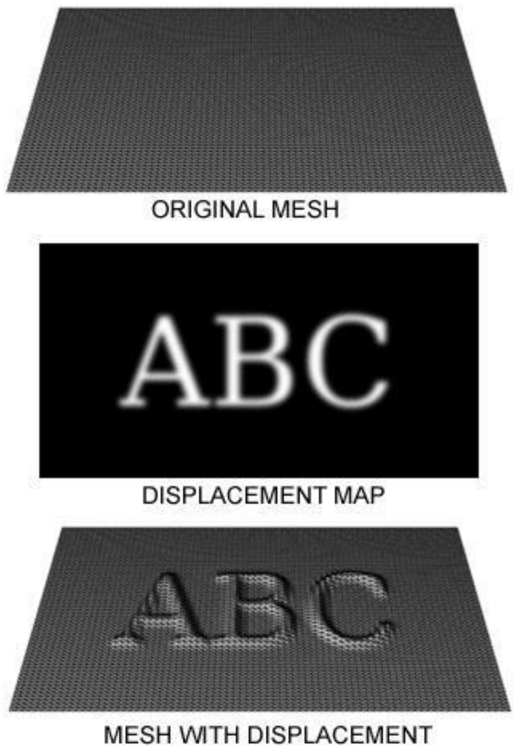
•Compute the normal from the partial derivatives in the texture



Displacement mapping

•Use the texture map to actually move the surface point

•The geometry must be displaced before visibility is determined



Environment mapping

•It’s difficult to generate reflections without GI method

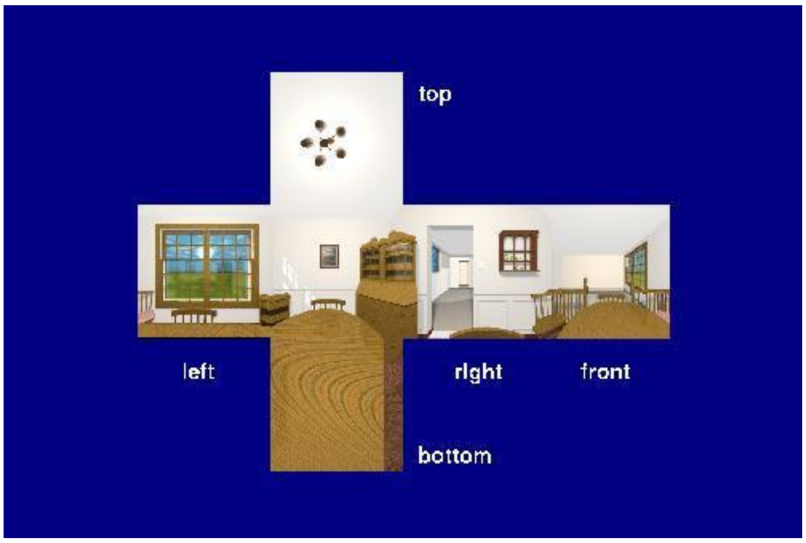
•We can simulate the reflection using texture mapping!

Cubic Mapping

•Simple and efficient

•Place 6 texture maps around the object

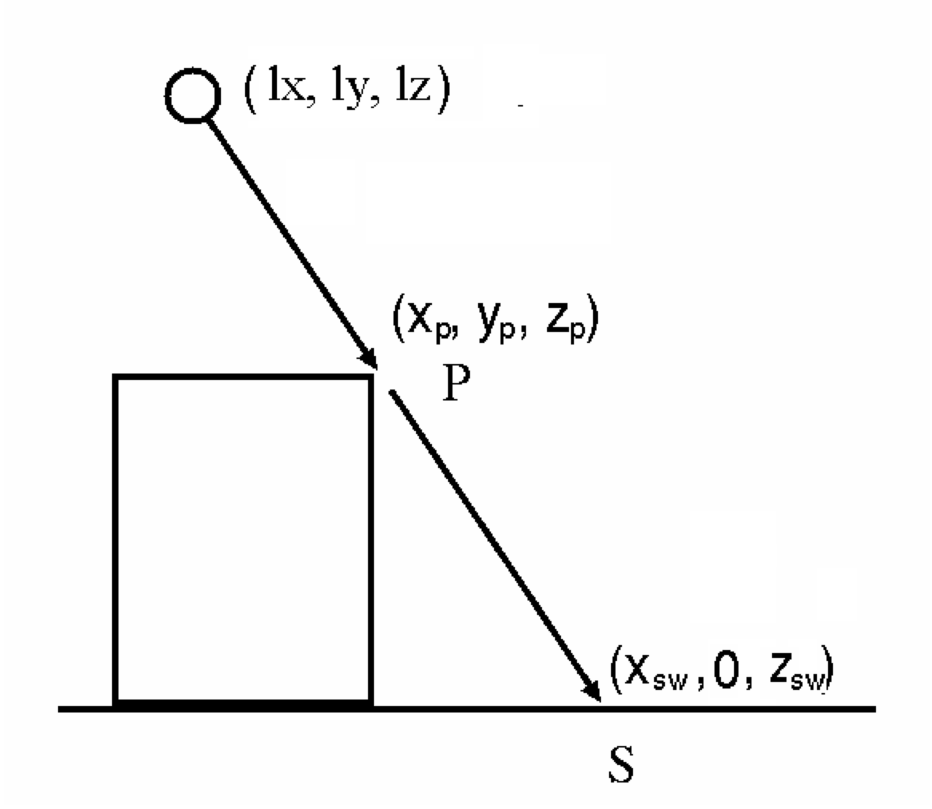
•Then, take 6 photos of the environment with a camera at the object's position



16 Shadow

Ground Shadow

•Shadow cast by objects onto the ground (y=0)



•The matrix transform the object onto the ground

•Thus

- •Draw the object
- •Multiply the shadow matrix
- •Redraw the object in grey

• The shadow only cast onto (ground) planes

• The shadow is hard shadow

• The performance is not optimal in static scene

- • Why?

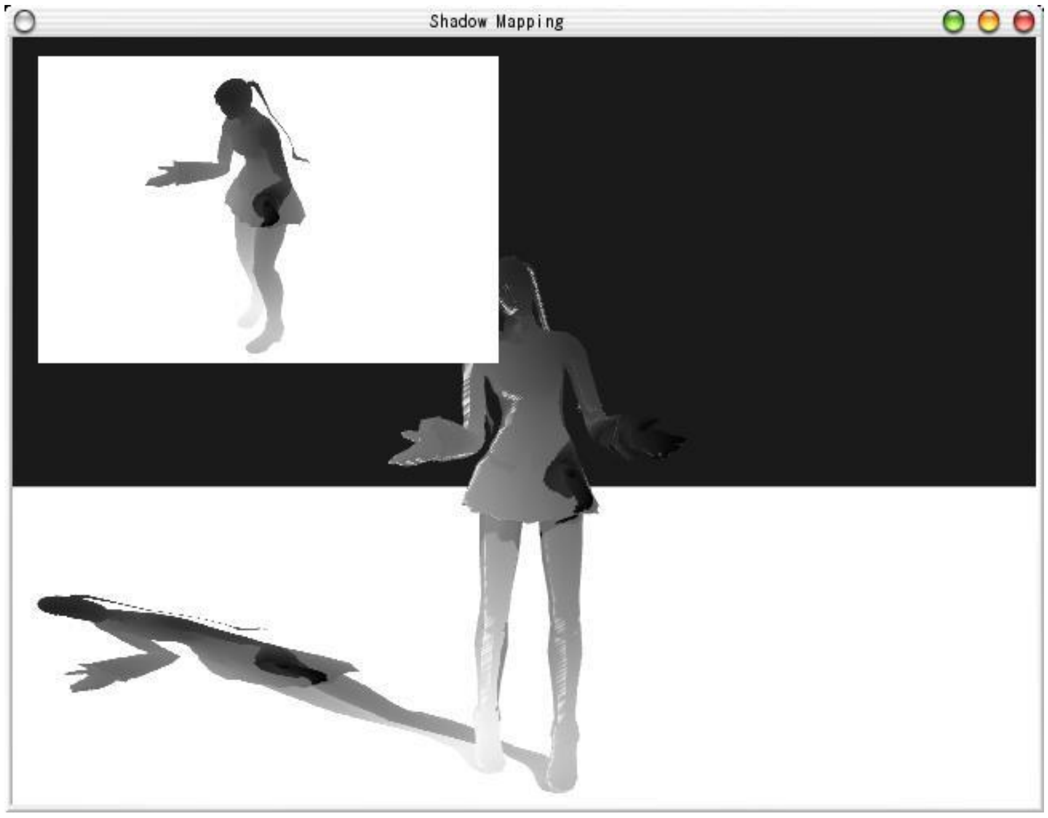
Shadow Texture

•Using a shadow image as a texture

•Occluder from light' s view

•Project the image onto the object

•Can be curved surface or other objects



•Drawbacks

  •It cannot generate self-shadows(shadows on the body of self)

  •The occlude and shadow receiver must be specified

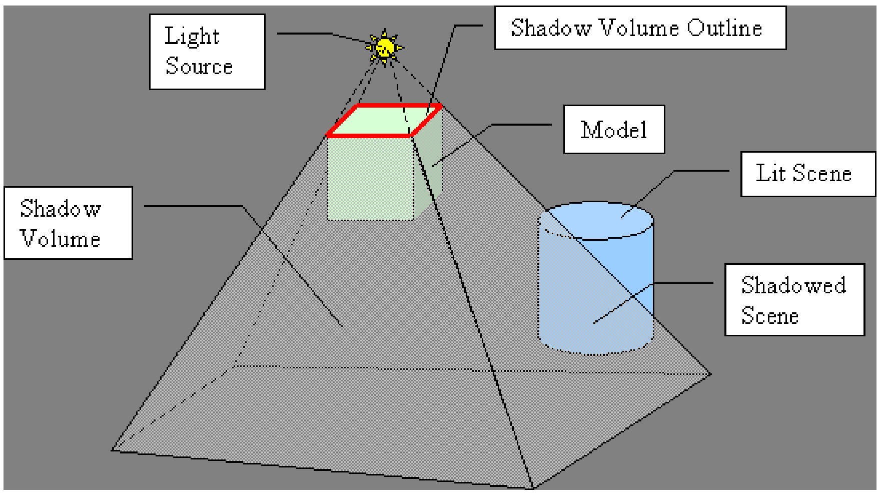
  •Aliasing shadows

Shadow volume

•In the reality, the shadow cast by an object blocking light is a volume which is 3D!

•Any objects intersecting with the volume will get shadow on them.

- •Self-cast shadow
- •General-purpose



Shadow Map

•Using Z-buffer

- •Render the scene from the light source using the Z-buffer algorithm
- •Render the scene from the view point
  - •Compute the coordinates of the sampled points in each light space
  - •If the point is farther from the value, it is in the shadow

•Preparation

- •Prepare a depth buffer for each light

•Render the scene from the light position

•Save the depth information in the depth buffer

•Rendering the scene

•Render the objects; whenever rendering an object, check if it is shadowed or not by transforming its coordinate into the light space

•After the transformation, if the depth value is larger than that in the light's depth buffer it should be shadowed

17 Ray tracing

Basic process

```
For each pixel {
    Shoot a ray from camera to pixel;
    //Perspective

    for all objects in scene
        Compute intersection with ray

    Find object with closest intersection

    Recursively shoot rays from the intersection point

    Display color using object + light property

}
```

The Modelling of rays

Shoot a ray from camera to pixel

•Sometimes the ray misses all of the objects
•and sometimes the ray will hit an object

•If the ray hits an object, we want to know if that point on the object is in a shadow.
•So, when the ray hits an object, a secondary ray, called a **"shadow" ray**, is shot towards the light sources.

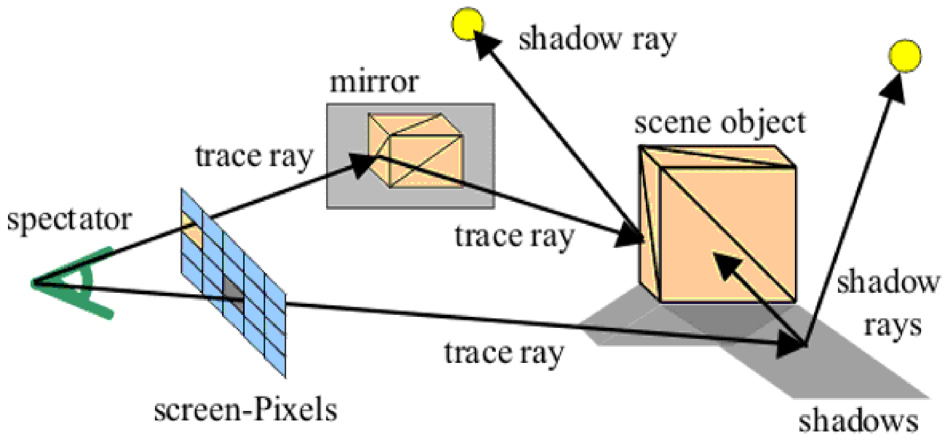
•If this shadow ray hits another object before it hits a light source, then the first intersection point is in the shadow of the second object.
•For a simple illumination model this means that we only apply the ambient term for that light source.

•Also, when a ray hits an object, a **reflected ray** is generated which is tested against all of the objects in the scene.

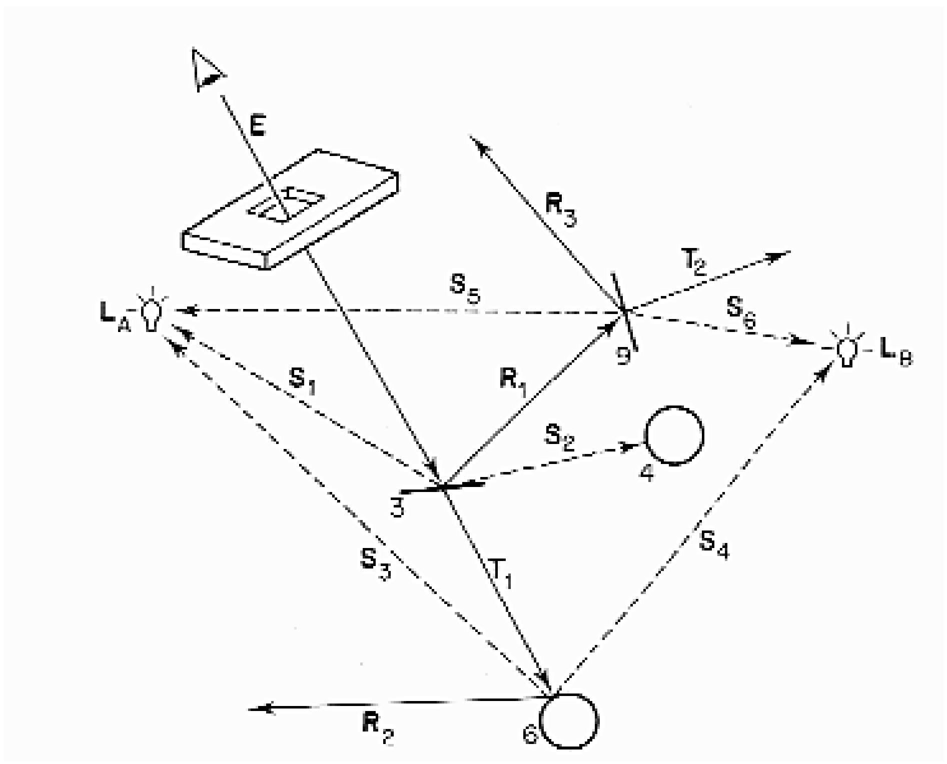
•If the reflected ray hits an object then a local illumination model is applied at the point of intersection and the result is carried back to the first intersection point.

•If the intersected object is transparent, then a **transmitted ray** is generated and tested against all the objects in the scene.

•The reflection ray can be implemented recursively
•There can be no reflection or multiple reflection



Ray Tree



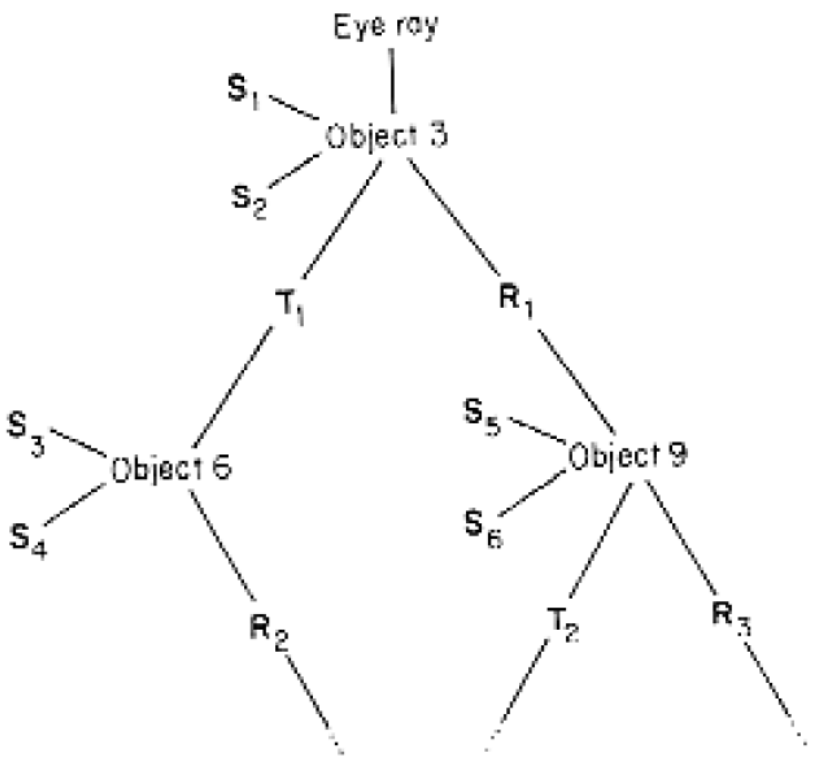


Fig. 12. The ray tree in schematic form.