Shapes

Digital Image Synthesis

Yung-Yu Chuang

with slides by Pat Hanrahan

Shapes

- One advantages of ray tracing is it can support various kinds of shapes as long as we can find ray-shape intersection.
- Careful abstraction of geometric shapes is a key component for a ray tracer. Ideal candidate for object-oriented design. Scan conversion may not have such a neat interface.
- All shape classes implement the same interface and the other parts of the ray tracer just use this interface without knowing the details about this shape.

Shapes

- **`Primitive`**`=`**`Shape`**`+`**`Material`**
- **`Shape`**: raw geometry properties of the primitive, implements interface such as surface area and bounding box.
- Source code in core/shape.* and shapes/*

Shapes

- `pbrt` provides the following shape plug-ins:
  - quadrics: sphere, cone, cylinder, disk, hyperboloid (雙曲面), paraboloid(拋物面) (surface described by quadratic polynomials in x, y, z)
  - triangle mesh
  - height field
  - NURBS
  - Loop subdivision surface
- Some possible extensions: other subdivision schemes, fractals, CSG, point-sampled geometry

Shapes

- All shapes are defined in object coordinate space

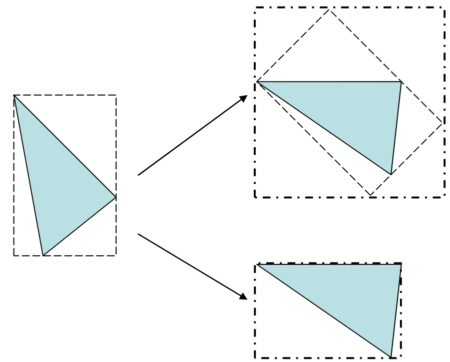
```
class Shape : public ReferenceCounted {
public:
  <Shape Interface>  all are virtual functions
  const Transform ObjectToWorld, WorldToObject;
  const bool reverseOrientation,
             transformSwapsHandedness;
  const int shapeId;  each shape is given an unique id. It
  can be used in adaptive image sampling, pixels with more
  complex geometry often need more samples.
  static int nextShapeId;  initialized as 1 as 0 is
                           reserved as `no shape'
}
```

Shape interface: bounding

- `BBox ObjectBound() const=0;` *pure virtual function*
- `BBox WorldBound() {` *left to individual shape*

  default implementation; can be overridden

```
  return ObjectToWorld(ObjectBound());
}
```



Shape interface: intersecting

- `bool CanIntersect()` *returns whether this shape can do intersection test; if not, the shape must provide*

  `void Refine(vector<Reference<Shape>>&refined)`

  examples include complex surfaces (which need to be tessellated first) or placeholders (which store geometry information in the disk)

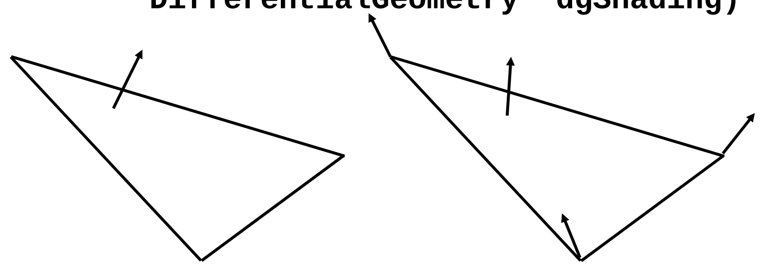
  in world space
- `bool Intersect(const Ray &ray,`
  `float *tHit, DifferentialGeometry *dg)`
- `bool IntersectP(const Ray &ray)`

  not pure virtual functions so that non-intersectable shapes don't need to implement them; instead, a default implementation which prints error is provided.

Shape interface

- `float Area()` *useful when using as an area light*
- `void GetShadingGeometry(` *for object instancing*

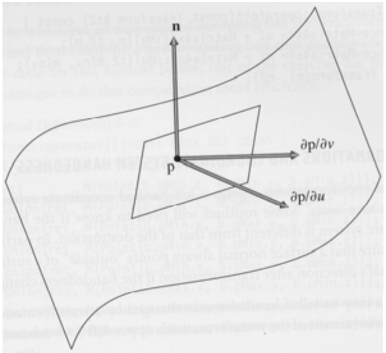
```
      const Transform &obj2world,
      const DifferentialGeometry &dg,
      DifferentialGeometry *dgShading)
```



- No back culling for that it doesn't save much for ray tracing and it is not physically correct

Differential geometry

- **`DifferentialGeometry`**: a self-contained representation for a particular point on a surface so that all the other operations in pbrt can be executed without referring to the original shape. It contains
- Position
- Parameterization (u,v)
- Parametric derivatives (dp/du, dp/dv)
- Surface normal (derived from (dp/du)x(dp/dv))
- Derivatives of normals
- Pointer to shape



Surfaces

- Implicit: F(x,y,z)=0
  you can check
- Explicit: (x(u,v),y(u,v),z(u,v))
  you can enumerate
  also called parametric
- Quadrics

$$
\begin{bmatrix} x & y & z & 1 \end{bmatrix}
\begin{bmatrix} A & B & C & D \\ B & E & F & G \\ C & F & H & I \\ D & G & I & J \end{bmatrix}
\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0
$$

Sphere

- A sphere of radius r at the origin
- Implicit: $x^2+y^2+z^2-r^2=0$
- Parametric: $f(\theta,\phi)$

  $x=r\sin\theta\cos\phi$

  $y=r\sin\theta\sin\phi$

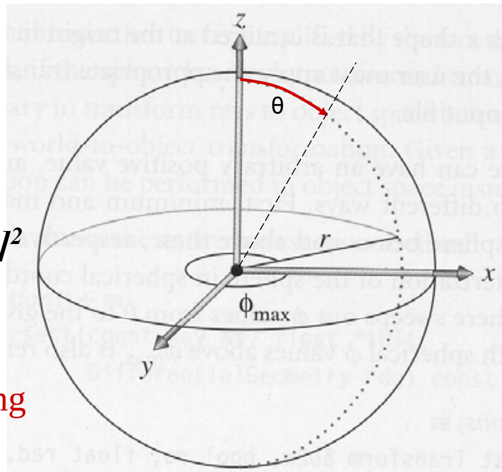
  $z=r\cos\theta$

mapping $f(u,v)$ over $[0,1]^2$

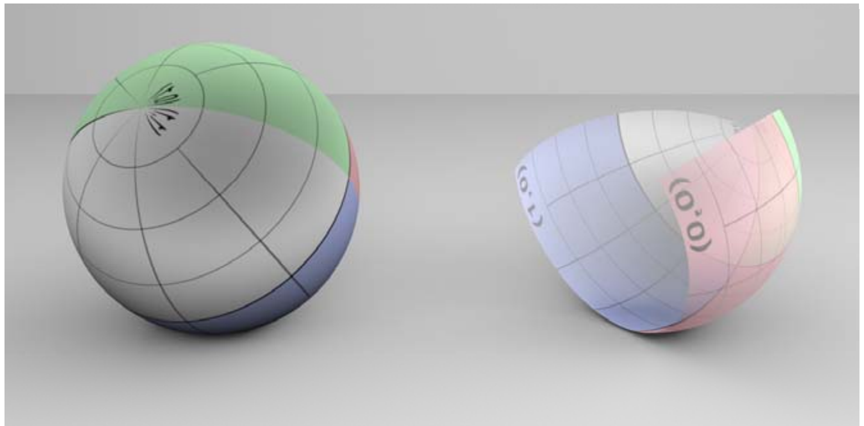
$\phi=u\phi_{max}$

$\theta=\theta_{min}+v(\theta_{max}-\theta_{min})$

useful for texture mapping



Sphere



Sphere (construction)

```
class Sphere: public Shape {
……
private:
  float radius;
  float phiMax;
  float zmin, zmax;      
  float thetaMin, thetaMax;
}
 Sphere(Transform *o2w, Transform *w2o,
        bool ro, float rad,
        float z0, float z1, float pm);
```

thetas are derived from z

- Bounding box for sphere, only z clipping

Intersection (algebraic solution)

- Perform in object space, **WorldToObject(r, &ray)**
- Assume that ray is normalized for a while

$$x^2 + y^2 + z^2 = r^2$$

$$(o_x + td_x)^2 + (o_y + td_y)^2 + (o_z + td_z)^2 = r^2$$

$$At^2 + Bt + C = 0$$

Step 1

$$A = d_x^2 + d_y^2 + d_z^2$$

$$B = 2(d_x o_x + d_y o_y + d_z o_z)$$

$$C = o_x^2 + o_y^2 + o_z^2 - r^2$$

Algebraic solution

$$t_0 = \frac{-B - \sqrt{B^2 - 4AC}}{2A} \qquad t_1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A}$$

Step 2

If (B^2-4AC<0) then the ray misses the sphere

Step 3

Calculate t_0 and test if $t_0<0$ (actually mint, maxt)

Step 4

Calculate t_1 and test if $t_1<0$

check the real source code in sphere.cpp

Quadric (in pbrt.h)

```
inline bool Quadratic(float A, float B, float C,
                      float *t0, float *t1) {
  // Find quadratic discriminant
  float discrim = B * B - 4.f * A * C;
  if (discrim < 0.) return false;
  float rootDiscrim = sqrtf(discrim);
  // Compute quadratic _t_ values
  float q;
  if (B < 0) q = -.5f * (B - rootDiscrim);
  else       q = -.5f * (B + rootDiscrim);
  *t0 = q / A;
  *t1 = C / q;
  if (*t0 > *t1) swap(*t0, *t1);
  return true;
}
```

Why?

- Cancellation error: devastating loss of precision when small numbers are computed from large numbers by addition or subtraction.

```
double x1 = 10.000000000000004;
double x2 = 10.000000000000000;
double y1 = 10.00000000000004;
double y2 = 10.00000000000000;
double z = (y1 - y2) / (x1 - x2); // 11.5
```

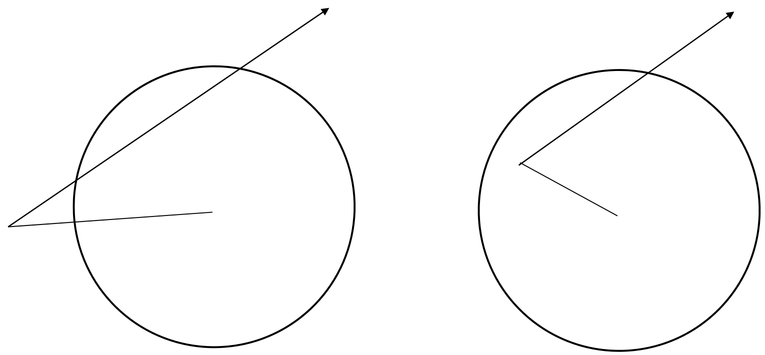
$$t_0 = \frac{q}{A} \qquad t_1 = \frac{C}{q} \qquad q = \begin{cases} -\frac{1}{2}\left(B - \sqrt{B^2 - 4AC}\right) & \text{if } B < 0 \\ -\frac{1}{2}\left(B + \sqrt{B^2 - 4AC}\right) & \text{otherwise} \end{cases}$$

Range checking

```
if (t0 > ray.maxt || t1 < ray.mint) return false;
float thit = t0;
if (t0 < ray.mint) {
    thit = t1;
    if (thit > ray.maxt) return false;
}
...
phit = ray(thit);
phi = atan2f(phit.y, phit.x);
if (phi < 0.) phi += 2.f*M_PI;
// Test sphere intersection against clipping
parameters
if (phit.z < zmin || phit.z > zmax || phi > phiMax)
{
  ... // see if we should check another hit point
}
```

Geometric solution

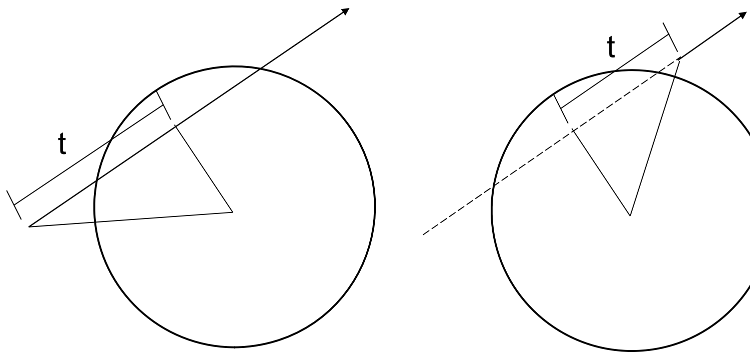
1. Origin inside? $o_x^2 + o_y^2 + o_z^2 > r^2$



Geometric solution

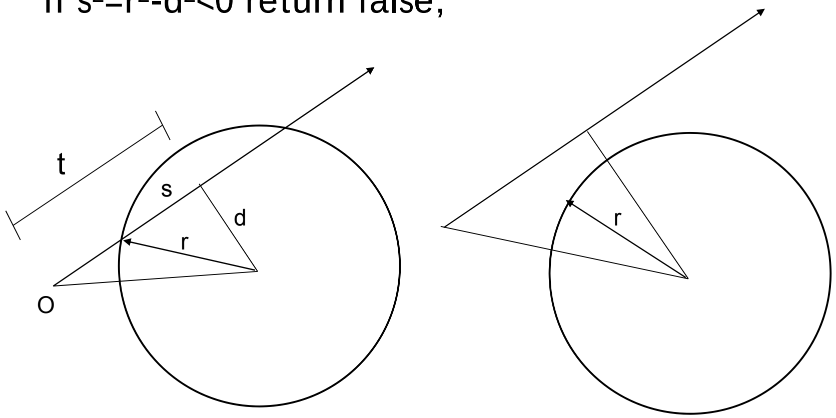
2. find the closest point, t=-O • D D is normalized

if t<0 and O outside return false



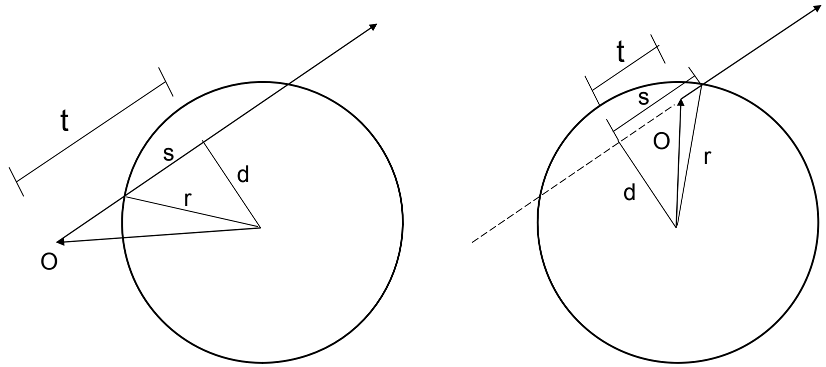
Geometric solution

3. find the distance to the origin, $d^2=O^2-t^2$
   if $s^2=r^2-d^2<0$ return false;



Geometric solution

4. calculate intersection distance,
   if (origin outside) then t-s
   else t+s



Sphere

- Have to test sphere intersection against clipping parameters
- Fill in information for **`DifferentialGeometry`**
  - Position
  - Parameterization (u,v)
  - Parametric derivatives
  - Surface normal
  - Derivatives of normals
  - Pointer to shape

Partial sphere

$u=\phi/\phi_{max}$

$v=(\theta-\theta_{min})/(\theta_{max}-\theta_{min})$

- Partial derivatives (pp121 of textbook)

$$\frac{\partial p}{\partial u}=(-\phi_{\max}y,\phi_{\max}x,0)$$

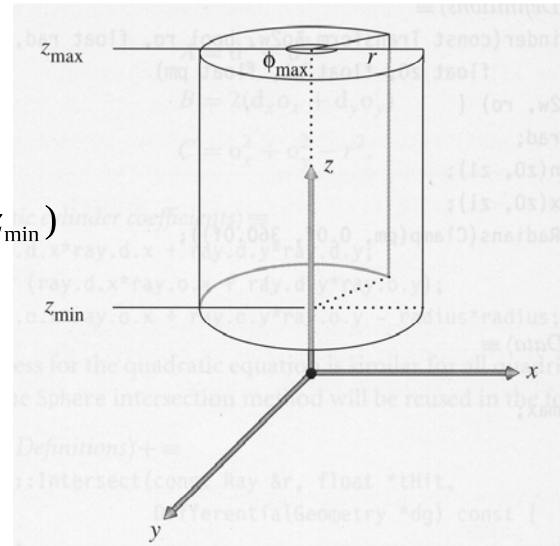
$$\frac{\partial p}{\partial v}=(\theta_{\max}-\theta_{\min})(z\cos\phi, z\sin\phi, -r\sin\theta)$$

- Area (pp123)

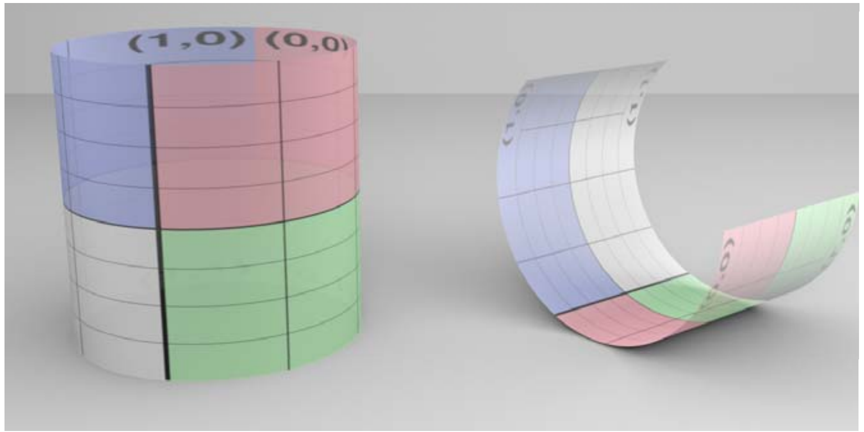
$$A=\phi_{\max}r(z_{\max}-z_{\min})$$

Cylinder

$$\phi = u\phi_{\max}$$
$$x = r\cos\phi$$
$$y = r\sin\phi$$
$$z = z_{\min} + v(z_{\max} - z_{\min})$$



Cylinder



Cylinder (intersection)

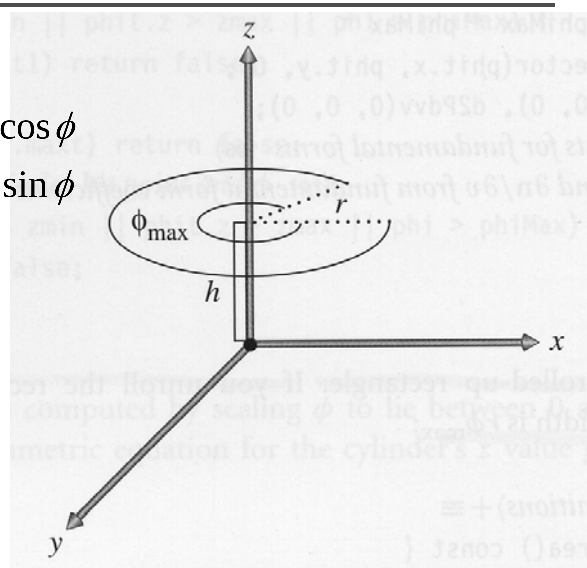
$$x^2 + y^2 = r^2$$
$$(o_x + td_x)^2 + (o_y + td_y)^2 = r^2$$

$$At^2 + Bt + C = 0$$

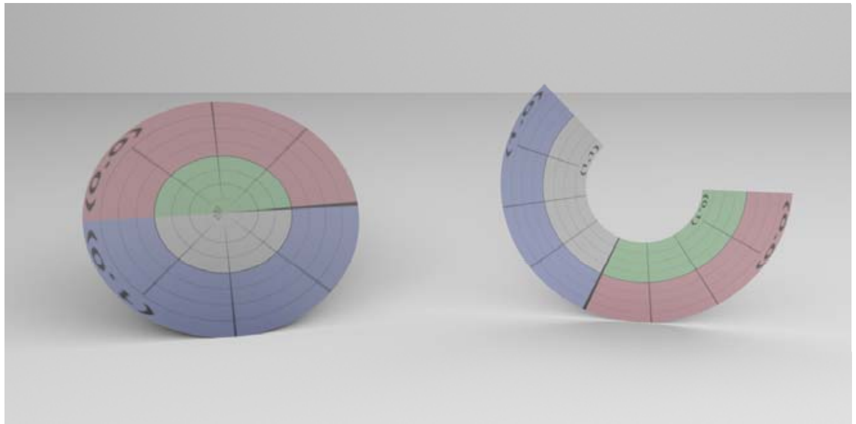
$$A = d_x^2 + d_y^2$$
$$B = 2(d_x o_x + d_y o_y)$$
$$C = o_x^2 + o_y^2 - r^2$$

Disk

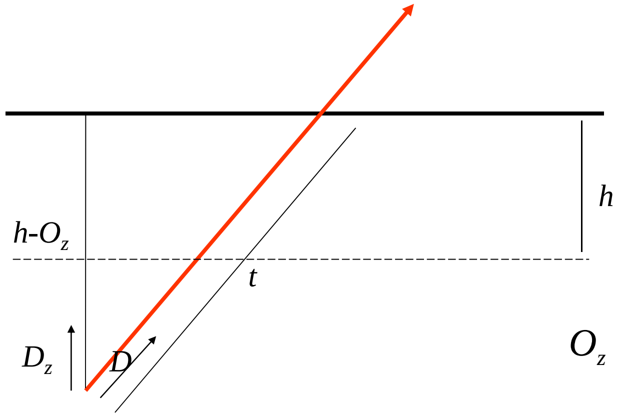
$$\phi = u\phi_{\max}$$
$$x = ((1-v)r_i + vr)\cos\phi$$
$$y = ((1-v)r_i + vr)\sin\phi$$
$$z = h$$



Disk



Disk (intersection)

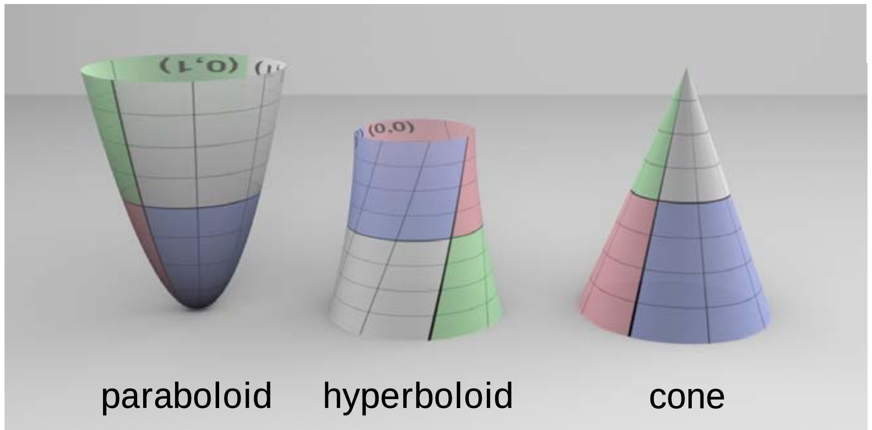


$$O_z + tD_z = h$$

$$t = \frac{h - O_z}{D_z}$$

Other quadrics

$$\frac{hx^2}{r^2} + \frac{hy^2}{r^2} - z = 0 \qquad x^2 + y^2 - z^2 = -1 \qquad \left(\frac{hx}{r}\right)^2 + \left(\frac{hy}{r}\right)^2 - (z-h)^2 = 0$$



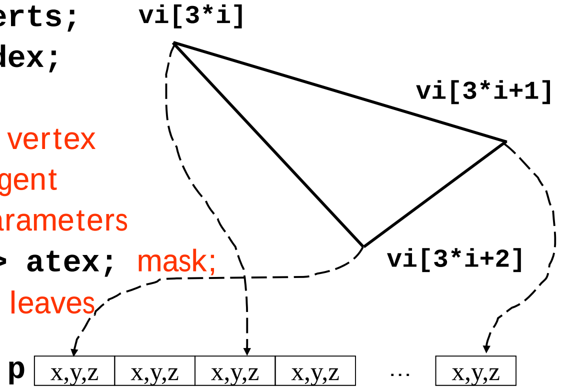
Triangle mesh

The most commonly used shape. In `pbrt`, it can be supplied by users or tessellated from other shapes.

Some ray tracers only support triangle meshes.

Triangle mesh

```
class TriangleMesh : public Shape {
…
  int ntris, nverts;
  int *vertexIndex;
  Point *p;
  Normal *n;  per vertex
  Vector *s;  tangent
  float *uvs;  parameters
  Texture<float> atex;  mask;
 useful for modeling leaves
}
```



Note that **p** is stored in world space to save transformations. **n** and **s** are in object space.

Triangle mesh

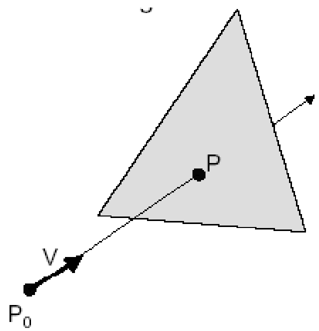
Pbrt calls Refine() when it encounters a shape that is not intersectable. (usually, refine is called in acceleration structure creation)

```
Void TriangleMesh::Refine(vector<Reference<Shape>>
                                    &refined)
{
  for (int i = 0; i < ntris; ++i)
    refined.push_back(new Triangle(ObjectToWorld,
    reverseOrientation, (TriangleMesh *)this, i));
}
```

Refine breaks a triangle mesh into a list of **Triangle**s. **Triangle** only stores a pointer to mesh and a pointer to vertexIndex.

Ray triangle intersection

1. Intersect ray with plane
2. Check if point is inside triangle



Ray plane intersection

Algebraic Method

$$Ray: P = P_0 + tV$$

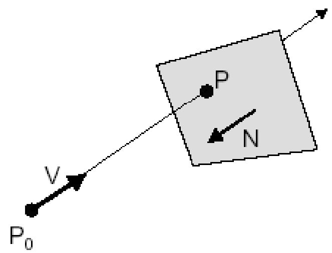
$$Plane: P \cdot N + d = 0$$

Substituting for P, we get:

$$(P_0 + tV) \cdot N + d = 0$$

Solution:

$$t = \frac{-(P_0 \cdot N + d)}{(V \cdot N)}$$



$$P = P_0 + tV$$

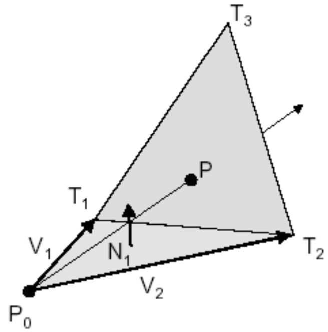
Ray triangle intersection I

Algebraic Method

For each side of triangle:

$$V_1 = T_1 - P_0$$
$$V_2 = T_2 - P_0$$
$$N_1 = V_1 \times V_2$$
$$\text{Normalize } N_1$$
$$d_1 = -P_0 \cdot N_1$$
$$\text{if } ((P \cdot N_1 + d_1) < 0)$$
$$\text{return FALSE}$$

end



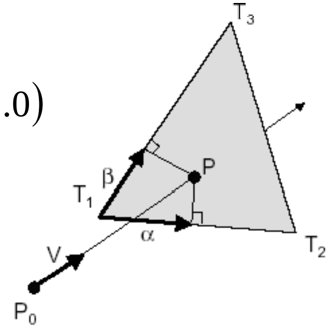
Ray triangle intersection II

Parametric Method

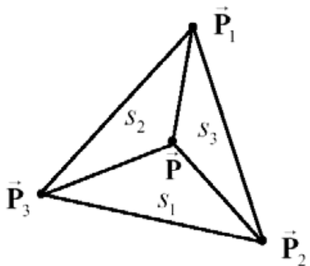
Compute α, β:

$$P = \alpha(T_2 - T_1) + \beta(T_3 - T_1)$$

$$\text{if } (0.0 \le \alpha \le 1.0) \text{ and } (0.0 \le \beta \le 1.0)$$
$$\text{and } (\alpha + \beta \le 1.0)$$
$$\text{then } P \text{ is inside triangle}$$



Ray triangle intersection III



$$s_1 = \text{area}(\Delta \mathbf{P}\mathbf{P}_2\mathbf{P}_3)$$
$$s_2 = \text{area}(\Delta \mathbf{P}\mathbf{P}_3\mathbf{P}_1)$$
$$s_3 = \text{area}(\Delta \mathbf{P}\mathbf{P}_1\mathbf{P}_2)$$

Barycentric coordinates

$$\vec{\mathbf{P}} = s_1\vec{\mathbf{P}}_1 + s_2\vec{\mathbf{P}}_2 + s_3\vec{\mathbf{P}}_3$$

Inside criteria

$$0 \le s_1 \le 1$$
$$0 \le s_2 \le 1$$
$$0 \le s_3 \le 1$$
$$s_1 + s_2 + s_3 = 1$$

Fast minimum storage intersection

a point on the ray   *a point inside the triangle*

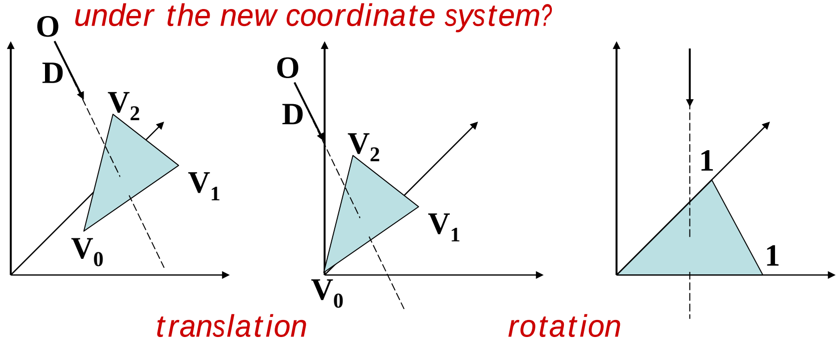
$$O + tD = (1 - u - v)V_0 + uV_1 + vV_2$$
$$u, v \ge 0 \text{ and } u + v \le 1$$

$$\begin{bmatrix} -D & V_1 - V_0 & V_2 - V_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0$$

Fast minimum storage intersection

$$\begin{bmatrix} -D & V_1 - V_0 & V_2 - V_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0$$

Geometric interpretation: what is O's coordinate under the new coordinate system?



translation *rotation*

Fast minimum storage intersection

$$\begin{bmatrix} -D & V_1 - V_0 & V_2 - V_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0$$

$$E_1 = V_1 - V_0 \qquad E_2 = V_2 - V_0 \qquad T = O - V_0$$

$$\begin{bmatrix} -D & E_1 & E_2 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = T$$

Fast minimum storage intersection

- Cramer's rule

$$\begin{bmatrix} -D & E_1 & E_2 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = T$$

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\left| -D, E_1, E_2 \right|} \begin{bmatrix} \left| T, E_1, E_2 \right| \\ \left| -D, T, E_2 \right| \\ \left| -D, E_1, T \right| \end{bmatrix}$$

$$\left| A, B, C \right| = -(A \times C) \cdot B = -(C \times B) \cdot A$$

Fast minimum storage intersection

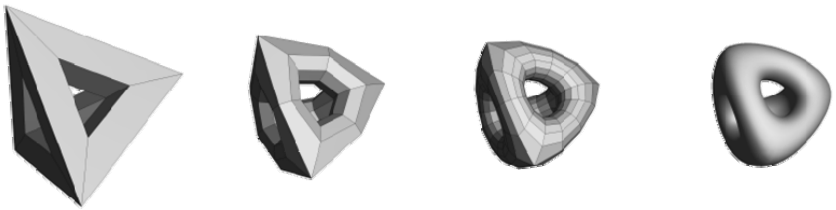
$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\left| -D, E_1, E_2 \right|} \begin{bmatrix} \left| T, E_1, E_2 \right| \\ \left| -D, T, E_2 \right| \\ \left| -D, E_1, T \right| \end{bmatrix}$$

$$Q = T \times E_1 \qquad P = D \times E_2$$

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix}$$

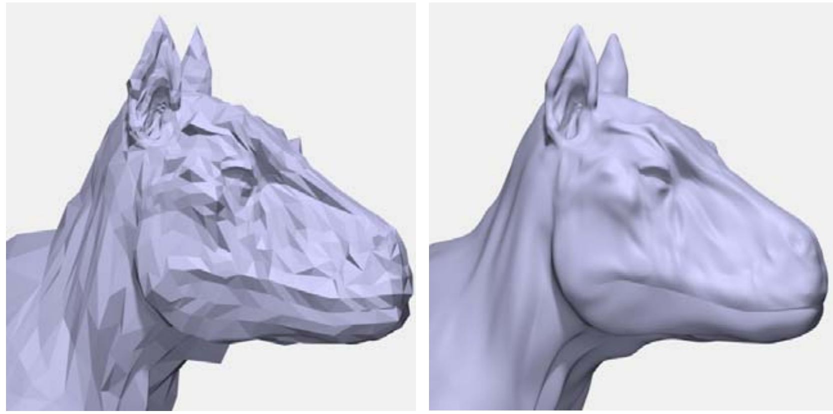
1 division
27 multiplies
17 adds

Subdivision surfaces

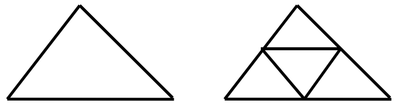


http://www.subdivision.org/demos/demos.html

Subdivision surfaces

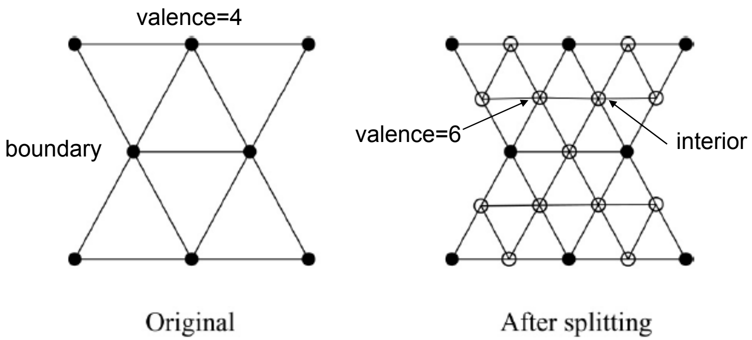


- Catmull-Clark (1978)



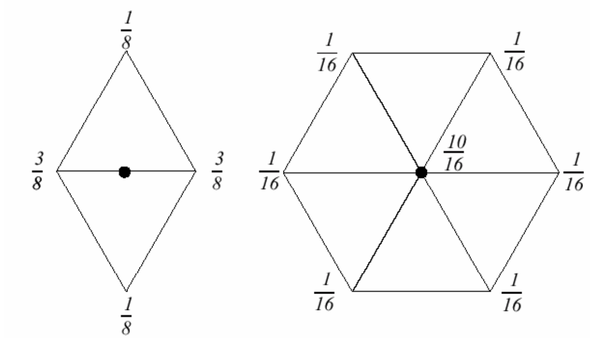
Loop Subdivision Scheme

- Refine each triangle into 4 triangles by splitting each edge and connecting new vertices



Loop Subdivision Scheme

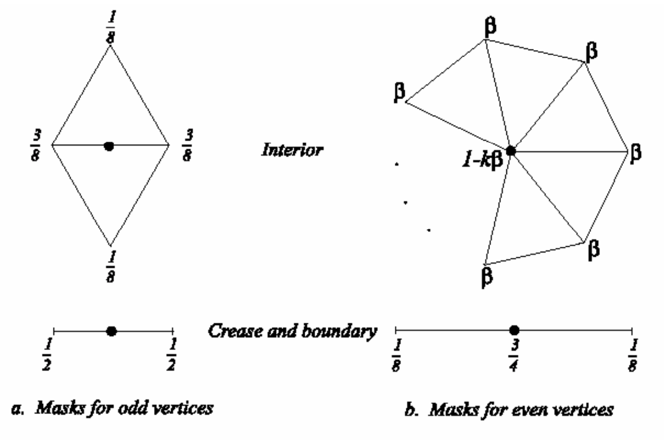
- Where to place new vertices?
  - Choose locations for new vertices as weighted average of original vertices in local neighborhood



odd vertices (new) even vertices (old)

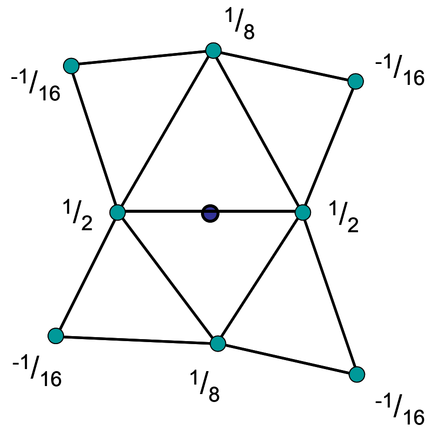
Loop Subdivision Scheme

- Where to place new vertices? $\beta = \frac{1}{n}(5/8 - (\frac{3}{8} + \frac{1}{4}\cos\frac{2\pi}{n})^2)$
  - Rules for *extraordinary vertices* and *boundaries*:



Butterfly subdivision

- Interpolating subdivision: larger neighborhood



Advantages of subdivision surfaces

- Smooth surface
- Existing polygon modeling can be retargeted
- Well-suited to describing objects with complex topology
- Easy to control localized shape
- Level of details

- Demo

Geri's game