CS 563 Advanced Topics in Computer Graphics

by Emmanuel Agu
- Parsing: uses lex and yacc: core/pbrtlex.l and core/pbrtparse.y
- After parsing, a `scene` object is created (core/scene.*)
- Rendering: `Scene::Render()` is invoked.
PBRT Architecture
Chapter 2: Representation and operations for the basic math:
- points, vectors and rays.
- core/geometry.* and core/transform.*

Chapter 3 (Shapes): Actual scene geometry such as triangles and spheres.

Chapter 4: Acceleration structures (uniform grid, kd-tree, BVH, etc)
Points, vectors and normals:
- 3 floating-point coordinate values: x, y, z defined under a coordinate system.

A coordinate system defined by:
- Origin + frame

Handedness?

PBRT uses right hand coordinate system

OpenGL uses right hand coordinate system
- Diff. b/w 2 points = vector
  \[ \mathbf{v} = Q - P \]
- Sum of point and vector = point
  \[ \mathbf{v} + P = Q \]
• Define vectors

\[ \mathbf{a} = (a_1, a_2, a_3) \]
\[ \mathbf{b} = (b_1, b_2, b_3) \]

• and scalar, \( s \)

Then vector addition:

\[ \mathbf{a} + \mathbf{b} = (a_1 + b_1, a_2 + b_2, a_3 + b_3) \]
- Scaling vector by a scalar

$$\mathbf{a}s = (a_1s, a_2s, a_3s)$$

**Note** vector subtraction:

$$\mathbf{a} - \mathbf{b} = (a_1 + (-b_1), a_2 + (-b_2), a_3 + (-b_3))$$
Magnitude of a Vector

- **Magnitude of** \( \mathbf{a} \)

\[ |\mathbf{a}| = \sqrt{a_1^2 + a_2^2 \ldots + a_n^2} \]

- **Normalizing a vector (unit vector)**

\[ \hat{\mathbf{a}} = \frac{\mathbf{a}}{|\mathbf{a}|} = \frac{\text{vector}}{\text{magnitude}} \]

- **Note magnitude of normalized vector = 1. i.e**

\[ \sqrt{a_1^2 + a_2^2 \ldots + a_n^2} = 1 \]
class Vector {
    public:
    <Vector Public Methods>
    float x, y, z;
}
(no need to use selector and mutator)
Dot product:
\[ \mathbf{v} \cdot \mathbf{u} = \|\mathbf{v}\|\|\mathbf{u}\|\cos \theta \]

Abs Dot product:
\[ \text{AbsDot}(\mathbf{v}, \mathbf{u}) \]

Cross product:
\[ \|\mathbf{v} \times \mathbf{u}\| = \|\mathbf{v}\|\|\mathbf{u}\|\sin \theta \]

(v, u, v x u) form a coordinate system

\[ (\mathbf{v} \times \mathbf{u})_x = v_y u_z - v_z u_y \]
\[ (\mathbf{v} \times \mathbf{u})_y = v_z u_x - v_x u_z \]
\[ (\mathbf{v} \times \mathbf{u})_z = v_x u_y - v_y u_x \]
Normalization

- **PBRT vector methods**
  - `Length(v)` – returns length of vector, v
  - `LengthSquared(v)` – (returns length of v)$^2$
  - `Normalize(v)` returns a vector, does not normalize in place
Coordinate system from a vector

Construct a local coordinate system from a vector.

```cpp
inline void CoordinateSystem(const Vector &v1,
                                Vector *v2, Vector *v3)
```

- V1 normalized already.
- Construct v2: perpendicular vector of v1 by
  - Zero out 1 component of v1
  - Swap other 2 components
- V1 x v2 = v3: 3rd vector
Points are different from vectors

```cpp
explicit Vector(const Point &p);
```

You have to convert a point to a vector explicitly (no accidents, know what you are doing).

- ✗ `Vector v=p;`
- ✓ `Vector v=Vector(p);`
Vector v; Point p, q, r; float a;

q = p + v;
q = p - v;
v = q - p;
r = p + q;
a * p; p / a;

(This is only for the operation $p + b \cdot q$.)

PBRT supports:
Distance(p, q);
DistanceSquared(p, q);
A surface normal (or just normal) is a vector that is perpendicular to a surface at a particular position.
- Different than vectors sometimes
- Particularly when applying transformations.
- Implementation similar to `Vector`, except
  - Normal cannot be added to a point
  - Cannot take the cross product of two normals.
- `Normal` is not necessarily normalized.
- Conversion between `Vector` and `Normal` must be explicit
```cpp
class Ray {
public:
    <Ray Public Methods>
    Point o;
    Vector d;
    mutable float mint, maxt;
    float time;
};

Ray r(o, d);
Point p=r(t);
```

They may be changed even if Ray is const.

Initialized as RAY_EPSILON to avoid self intersection.

\[ r(t) = o + td \quad 0 \leq t \leq \infty \]
Ray differentials

- Used to estimate projected area for a small part of a scene
- Used for **texture** antialiasing.

```cpp
class RayDifferential : public Ray {
public:
    // RayDifferential Methods
    bool hasDifferentials;
    Ray rx, ry;
};
```
Avoid intersection tests inside a volume if ray doesn’t hit bounding volume.

Benefits depends on:
- Expense of testing volume vs objects inside
- Tightness of the bounding volume.

Popular bounding volumes: sphere, axis-aligned bounding box (AABB), oriented bounding box (OBB).
class BBox {
public:
    <BBox Public Methods>
    Point pMin, pMax;
}

Point p, q; BBox b; float delta;
BBox(p, q)   // no order for p, q
Union(b, p) – Given point & Bbox, return new larger bounding box
             containing point (bbox) and Bbox.
Point \(p, q\); BBox \(b\);

\(b.\text{Expand}(\text{delta})\): Expand old bounding box by factor delta

\(p_{\text{Max}} + \text{delta}\)

\(p_{\text{Min}} - \text{delta}\)
Point \( p, q; \) BBox \( b; \)

- \( b.\text{Overlaps}(b2): \) do two bounding boxes overlap each other in \( x, y, z \)
- Returns boolean. True (overlaps) or false (does not overlap)
Point \( p, q; \) \( \text{BBox} \) \( b; \)

- \( b.\text{Inside}(p): \) Is point \( p \) inside bounding box? Returns boolean (true or false)
- \( \text{Volume}(b): \) Returns volume of bounding volume \( (x \times y \times z) \)
Point \( p, q; \) BBox \( b; \)

\( b.\text{MaximumExtent}() \) (which bounding box axis is the longest; useful for building kd-tree)

\( b.\text{BoundingSphere}(c, r) \) (returns center and radius of bounding sphere)

- Example: generate random ray which intersects scene geometry
class Transform {

private:
    Reference<Matrix4x4> m, mInv;

// save space, but can’t be modified after construction

- Transform stores element of 4x4 matrix
- Also computes and stores matrix inverse, mInv (avoid repeatedly computing inverse)
Transformations

- **Translate** \((\text{Vector}(dx, dy, dz))\)
- **Scale** \((sx, sy, sz)\)
- **RotateX** \((a)\)

\[
T(dx, dy, dz) = \begin{pmatrix}
1 & 0 & 0 & dx \\
0 & 1 & 0 & dy \\
0 & 0 & 1 & dz \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

\[
R_x(\theta) = \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & \cos \theta & -\sin \theta & 0 \\
0 & \sin \theta & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

\[
S(sx, sy, sz) = \begin{pmatrix}
sx & 0 & 0 & 0 \\
0 & sy & 0 & 0 \\
0 & 0 & sy & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

\[
R_x(\theta)^{-1} = R_x(\theta)^T
\]

Question: How does x-roll matrix above differ based on axes handedness?
- Rotate(a, Vector(1,1,1))
- Rotate(a, Vector(1,1,1))

\[ p = a(v \cdot a) \]

\[ v_1 = v - p \]

\[ v_2 = v_1 \times a \quad |v_2| = |v_1| \]

\[ v' = p + v_1 \cos \theta + v_2 \sin \theta \]
LookAt Transformation

- Caller specifies:
  - camera (eye position),
  - Look at point
  - Up vector
- Want to compute 4x4 transform matrix that converts from world space to eye space
LookAt(Point &pos, Point look, Vector &up)

Vector dir=Normalize(look-pos);
Vector right=Cross(dir, Normalize(up));
Vector newUp=Cross(right, dir);
Applying transformations

- **Point**: \( q = T(p), \ T(p, &q) \)
  
  use homogeneous coordinates implicitly

- **Vector**: \( u = T(v), \ T(u, &v) \)

- **Normal**: treated differently than vectors because of anisotropic transformations

\[
0^T T = MtSn
\]

- **Transform** should keep its inverse

- For orthonormal matrix, \( S = M \)

\[
\begin{align*}
    n \cdot t &= n^T t = 0 \\
    (n')^T t' &= 0 \\
    (Sn)^T Mt &= 0 \\
    n^T S^T Mt &= 0 \\
    S^T M &= I \\
    S &= M^{-T}
\end{align*}
\]
Applying transformations

- Transform Bbox?
  - transform its 8 corners and expand to include all 8 points.
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. Contains
  - Position
  - Surface normal
  - Parameterization
  - Parametric derivatives
  - Derivatives of normals
  - Pointer to shape
Ray-Surface Intersection
Ray-Plane Intersection

- **Ray:**
  \[ \vec{P} = \vec{O} + t \vec{D} \]
  \[ 0 \leq t < \infty \]

- **Plane:**
  \[ (\vec{P} - \vec{P'}) \cdot \vec{N} = 0 \]
  \[ ax + by + cz + d = 0 \]

- **Solve for intersection**
- **Substitute ray equation into plane equation**
  \[ (\vec{P} - \vec{P'}) \cdot \vec{N} = (\vec{O} + t\vec{D} - \vec{P'}) \cdot \vec{N} = 0 \]
  \[ t = -\frac{(\vec{O} - \vec{P'}) \cdot \vec{N}}{\vec{D} \cdot \vec{N}} \]
- A sphere of radius $r$ at the origin
- Implicit: $x^2 + y^2 + z^2 - r^2 = 0$
- Parametric: $f(?, ?)$
  
  $x = r \sin\ ? \ \cos\ ?$
  
  $y = r \sin\ ? \ \sin\ ?$
  
  $z = r \cos\ ?$
Algebraic solution

- Perform in object space, \texttt{WorldToObject}(r, &ray)
- Assume that ray is normalized for a while

\[ x^2 + y^2 + z^2 = r^2 \]

\[ (o_x + td_x)^2 + (o_y + td_y)^2 + (o_z + td_z)^2 = r^2 \]

\[ At^2 + Bt + C = 0 \]

\[ A = d_x^2 + d_y^2 + d_z^2 \]

\[ B = 2(d_x o_x + d_y o_y + d_z o_z) \]

\[ C = o_x^2 + o_y^2 + o_z^2 - r^2 \]
Algebraic solution

\[ t_0 = \frac{-B - \sqrt{B^2 - 4AC}}{2A} \quad t_1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A} \]

**Step 2**
If \( B^2 - 4AC < 0 \) then the ray misses the sphere. \( B^2 - 4AC = 0? \)

**Step 3**
Calculate \( t_0 \) and test if \( t_0 < 0 \)

**Step 4**
Calculate \( t_1 \) and test if \( t_1 < 0 \)
\[ \phi = u \phi_{\text{max}} \]

\[ x = r \cos \phi \]

\[ y = r \sin \phi \]

\[ z = z_{\text{min}} + v(z_{\text{max}} - z_{\text{min}}) \]

- First consider sides
- Later consider cap/base
Cylinder

- Implicit equation for cylinder
  \[ x^2 + y^2 - r^2 = 0 \]

- Substituting in ray equation
  \[
  (o_x + td_x)^2 + (o_y + td_y)^2 = r^2
  \]

- Giving
  \[
  At^2 + Bt + C = 0
  \]

  \[
  A = d_x^2 + d_y^2
  \]

  \[
  B = 2(d_x o_x + d_y o_y)
  \]

  \[
  C = o_x^2 + o_y^2 - r^2 \]

  Solve for \( t \)
Cylinder
- Pat Hanrahan, CS 348B, Spring 2005 class slides
- Yung-Yu Chuang, Image Synthesis, class slides, National Taiwan University, Fall 2005
- Kutulakos K, CSC 2530H: Visual Modeling, course slides
- UIUC CS 319, Advanced Computer Graphics Course slides