Ray tracing-II

Computer Graphics
COMP 770 (236)
Spring 2007

Instructor: Brandon Lloyd
From last time...

- Generalizing ray casting
- Intersection tests
  - Plane
  - Polygon
  - Sphere
- Recursive ray tracing
Today’s topics

- Robustness issues
- Code structure
- Optimizations
  - Acceleration structures
- Distribution ray tracing
  - anti-aliasing
  - depth of field
  - soft shadows
  - motion blur
Robustness Issues

- **False self-intersections**
  - One solution is to offset the origin of the ray from the surface when tracing secondary rays
  - May have true self-shadowing that doesn’t match smooth shading

- ... but offsets also cause problems
Design of a Ray Tracer

Building a ray tracer is simple. First we start with a convenient vector algebra library.

class Vector:
    def __init__(self, *args):
        if (type(args[0]) is list):
            self.x, self.y, self.z = args[0]
        elif isinstance(args[0], Vector):
            self.x, self.y, self.z = args[0].x, args[0].y, args[0].z
        else:
            self.x, self.y, self.z = args

    def __str__(self):
        return '(%f, %f, %f)'% (self.x, self.y, self.z)

    def __sub__(self, v):
        return Vector(self.x - v.x, self.y - v.y, self.z - v.z)

    def __add__(self, v):
        return Vector(self.x + v.x, self.y + v.y, self.z + v.z)

    def normalize(self):
        l = math.sqrt(self.x*self.x + self.y*self.y + self.z*self.z)
        if (l != 0):
            l = 1/l
        return Vector(l*self.x, l*self.y, l*self.z)

    def length(self):
        return math.sqrt(self.x*self.x + self.y*self.y + self.z*self.z)

    def cross(self, v):
        return Vector(self.y*v.z - self.z*v.y, self.z*v.x - self.x*v.z, self.x*v.y - self.y*v.x)

    def dot(self, v):
        return (self.x*v.x + self.y*v.y + self.z*v.z)

    def scale(self, s):
        return Vector(self.x*s, self.y*s, self.z*s)
class Ray:
    MAX_T = 1.0e300
    def __init__(self, ovec, dvec):
        self.origin = Vector(ovec)
        self.direction = Vector(dvec).normalize()

    def __str__(self):
        return 'origin = %s, direction = %s' % (self.origin, self.direction)

    def trace(self, objects):
        self.t = Ray.MAX_T
        self.primitive = None;
        for obj in objects:
            obj.intersect(self)
        return (self.primitive != None)

    def shade(self, lights, objects, background):
        return self.primitive.shade(self, lights, objects, background)

This method is not strictly needed, and most likely adds unnecessary overhead, but I preferred the syntax
ray.shade(...) to
ray.primitive.shade(ray, ...)
class Light:
    AMBIENT = 0
    DIRECTIONAL = 1
    POINT = 2
    def __init__(self, type, r, g, b, v = None):
        self.lightType = type
        self.ir, self.ig, self.ib = r, g, b
        self.lvec = v
        if (type == Light.DIRECTIONAL):
            self.lvec = self.lvec.normalize()
Every object in our ray tracer must be able to

1. Intersect itself with a ray
2. Shade itself (determine the color it reflects along the given ray)

```python
def MyObject:
    .
    .
    def intersect(ray):  # returns boolean
        .
        .
    def Shade(ray, lightList, objectList, bgrndColor):  # returns (r,g,b)
        .
        .
```
# An example object
class Sphere:
    def __init__(self, s, c, r):
        self.surface = s
        self.center = c
        self.radius = r
        self.radSqr = r*r

    def __str__(self):
        return 'sphere %s %f' % (center, radius)
def intersect(self, ray):
    c = self.center - ray.origin
    v = ray.direction.dot(c)
    # Check if there is even a chance an
    # intersection might be closer
    if (v - self.radius > ray.t):
        return 0
    # Test if the ray intersects sphere
    t = self.radSqr + v*v - c.dot(c)
    if (t < 0):
        return 0
    # Test if intersection is in the positive
    # ray direction and is the closest so far
    t = v - math.sqrt(t)
    if ((t > ray.t) or (t < 0)):
        return 0

    ray.t = t
    ray.primitive = self
    return 1
def shade(self, ray, lights, objects, bgnd):
    # An object shader doesn't really do too much other than
    # supply a few critical bits of geometric information
    # for a surface shader. It must must compute:
    #
    # 1. the point of intersection (p)
    # 2. a unit-length surface normal (n)
    # 3. a unit-length vector towards the ray's origin (v)
    #
    p = ray.origin + ray.direction.scale(ray.t)
    v = ray.direction.scale(-1.0)
    n = (p - self.center).normalize()

    # The illumination model is applied
    # by the surface's shade() method
    return self.surface.shade(p, n, v, lights, objects, bgnd)
class Surface:
    TINY = 0.001
    def __init__(self, rval, gval, bval, a, d, s, n, r, t, index):
        self.ir, self.ig, self.ib = rval, gval, bval  # surface color
        self.ka = a  # ambient coefficient
        self.kd = d  # diffuse coefficient
        self.ks = s  # specular coefficient
        self.ns = n  # Shininess power
        self.kr = r  # reflection coefficient
        self.kt = t  # transparency coefficient
        self.nt = index  # index of refraction
def shade(self, p, n, v, lights, objects, bgnd):
    global depth
    r, g, b = 0.0, 0.0, 0.0
    for light in lights:
        if (light.lightType == Light.AMBIENT):
            r += self.ka*self.ir*light.ir
            g += self.ka*self.ig*light.ig
            b += self.ka*self.ib*light.ib
        else:
            if (light.lightType == Light.POINT):
                l = (light.lvec - p).normalize()
            else:
                l = light.lvec.scale(-1.0)

            # Further processing with light vectors...
# Check if the surface point is in shadow
poffset = p + l.scale(Surface.TINY)
shadowRay = Ray(poffset, l)
if (shadowRay.trace(objects)):
    continue

lambert = n.dot(l)
if (lambert > 0):
    if (self.kd > 0):
        diffuse = self.kd*lambert
        r += diffuse*self.ir*light.ir
        g += diffuse*self.ig*light.ig
        b += diffuse*self.ib*light.ib
if (lambert > 0):
    if (self.kd > 0):
        .
        .
        if (self.ks > 0):
            spec = v.dot(n.scale(2*lambert) - l)
            if (spec > 0):
                spec = self.ks*math.pow(spec, self.ns)
                r += spec*light.ir
                g += spec*light.ig
                b += spec*light.ib

\[
\hat{R} = 2(\hat{N} \cdot \hat{L})\hat{N} - \hat{L}
\]

\[
I_{\text{specular}} = k_s I_{\text{light}} (\hat{V} \cdot \hat{R})^{n_{\text{skiny}}}
\]
# Compute illumination due to reflection

if (self.kr > 0):
    t = v.dot(n)
    if (t > 0):
        reflect = n.scale(2*t) - v
        poffset = p + reflect.scale(self.TINY)
        reflectedRay = Ray(poffset, reflect)
        if (reflectedRay.trace(objects)):
            depth += 1
            if (depth < 5):
                rcolor = reflectedRay.shade(lights, objects, bgnd)
                r += self.kr*rcolor[0]
                g += self.kr*rcolor[1]
                b += self.kr*rcolor[2]
            depth -= 1
        else:
            r += self.kr*bgnd[0]
            g += self.kr*bgnd[1]
            b += self.kr*bgnd[2]
if (kt > 0):
    # Add code for refraction here

    return (r, g, b);

That’s basically all we need to write a ray tracer. Compared to a graphics pipeline the code is very simple and easy to understand. Next, we’ll write a little driver application.
def renderPixel(i, j):
    global background, depth
    v = Vp + Du.scale(i) + Dv.scale(j)
    ray = Ray(eye, v)
    rayColor = background
    if (ray.trace(objectList)):
        depth = 0
        rayColor = ray.shade(lightList, objectList, background)
    return rayColor
def renderLine(y):
    global start, image
    if (y == 0):
        print 'Starting Rendering Thread...'  
        start = time.time()
    for x in xrange(width):
        pix = renderPixel(x,y)
        pix = (int(256*pix[0]), int(256*pix[1]), int(256*pix[2]))
        image.putpixel((x,height-y-1),pix)
    y += 1
    if (y == height):
        print 'Rendering time = %6.1f secs' % (time.time() - start)
    else:
        glutTimerFunc(1, renderLine, y)
    glutPostRedisplay()
We can use a simple input parser similar to the one used for Wavefront OBJ files. Here is an example input file:

```
eye 0 2 10
lookat 0 0 0
up 0 1 0
fov 30
background 0.2 0.8 0.9
light 1 1 1 ambient
light 1 1 1 directional -1 -2 -1
light 0.5 0.5 0.5 point -1 2 -1

surface 0.7 0.2 0.2 0.5 0.4 0.2 3.0 0.0 0.0 1.0
sphere -1 -3 -2 1.5
sphere 1 -3 -2 1.5
sphere -2 -3 -1 1.5
sphere 0 -3 -1 1.5
sphere 2 -3 -1 1.5
sphere -1 -3 1 1.5
sphere 1 -3 1 1.5
sphere -2 -3 0 1.5
sphere 0 -3 0 1.5
sphere 2 -3 0 1.5
sphere -1 -3 1 1.5
sphere 1 -3 1 1.5
sphere -2 -3 2 1.5
sphere 0 -3 2 1.5
sphere 2 -3 2 1.5

surface 0.4 0.4 0.4 0.1 0.1 0.6 100.0 0.8 0.0 1.0
sphere 0 0 0 1
```
Example

Advantages of Ray Tracing:
- Improved realism over the graphics pipeline
- Shadows
- Reflections
- Transparency
- Higher level rendering primitives
- Very simple design

Disadvantages:
- Very slow per pixel calculations
- Only approximates full global illumination
- Hard to accelerate with special-purpose H/W
Acceleration Methods

- Render time for a ray tracer depends on the number of ray intersection tests per pixel
  - roughly dependent on the number of primitives in the scene times the number of pixels.
- Early efforts focused on accelerating the ray-object intersection tests
- More advanced methods required to make ray tracing practical
  - Bounding Volumes
  - Spatial Subdivision
  - Light Buffers
Bounding Volumes

- Enclose complex objects within a simple-to-intersect objects.
  - If the ray does not intersect the simple object then its contents can be ignored.
  - The likelihood that it will strike the object depends on how tightly the volume surrounds the object.
- Spheres are simple, but not tight.
- Axis-aligned bounding boxes often better
  - can use nested or hierarchical bounding volumes

*Spherical Bounding Volumes*  
*Axis-Aligned Bounding Boxes*
Bounding Volumes

- **Sphere [Whitted80]**
  - Cheap to compute
  - Cheap test
  - Potentially very bad fit

- **Axis-Aligned Bounding Box**
  - Very cheap to compute
  - Cheap test
  - Tighter than sphere
Bounding Volumes

- **Oriented Bounding Box**
  - Fairly cheap to compute
  - Fairly Cheap test
  - Generally fairly tight

- **Slabs / K-dops**
  - More Expensive to compute
  - Fairly Cheap test
  - Can be tighter than OBB
Hierarchical Bounding Volumes

- Organize bounding volumes as a tree
- Each ray starts with the scene BV and traverses down through the hierarchy
Spatial Subdivision

**Idea:** Divide space into subregions

- Place objects within a subregion into a list
- Only traverse the lists of subregions that the ray passes through
- “Mailboxing” used to avoid multiple test with objects in multiple regions

- Many types
  - Regular grid
  - Octree
  - BSP tree
  - kd-tree
Light Buffers

A significant portion of the object-ray intersections are used to compute shadow rays.

Idea:

- Enclose each light source with a cube
- Subdivide each face of the cube and determine the potentially visible objects that could projected into each subregion
- Keep a list of objects at each subdivision cell
Other Optimizations

- **Shadow cache**
  - due to coherence the last object intersected will likely be intersected on the next ray
  - save last hit object and test it first for next ray

- **Adaptive depth control**
  - limit the depth of recursion
  - the color of secondary rays gets modulated in lighting equations. Stop recursing when contribution falls below a threshold

- **Lazy geometry loading/creation**
  - for very complex models or procedural models we can supply a bounding volume and defer the actual loading/creation of the geometry until a ray hits the bounding volume
Cook & Porter, in their classic paper “Distributed Ray Tracing” realized that ray-tracing, when combined with randomized sampling, which they called “jittering”, could be adapted to address a wide range of rendering problems:
Antialiasing

- The need to sample is problematic because sampling leads to aliasing
- Solution 1) super-sampling
  - increases sampling rate, but does not completely eliminate aliasing
  - difficult to completely eliminate aliasing without prefiltering because the world is not band-limited
- Solution 2) distribute the samples randomly
  - converts the aliasing energy to noise which is less objectionable to the eye

Instead of casting one ray per pixel, cast several (sub-sampling).
Instead of uniform sub-sampling, jitter the pixels slightly off the grid.
Depth-of-Field

- Rays don’t have to all originate from a single point.
- Real cameras collects rays over an aperture
  - can be modeled as a disk
  - final image is blurred away from the focal plane.
  - gives rise to depth-of-field effects.
Depth of Field
Depth of Field

- Start with normal eye ray and find intersection with focal plane
- Choose jittered point on lens and trace line from lens point to focal point
Motion Blur

- You can also jitter samples through time to simulate the finite interval that a shutter is open on a real camera. This produces motion blur in the rendering.

- Given a time varying model, compute several rays at different instances of time and average them together.
Soft shadows

- Take many samples from area light source and take their average
  - computes fractional visibility leading to penumbra
Complex Interreflection

- Model true reflection behavior as described by a full BRDF
- Randomly sample rays over the hemisphere, weight them by their BRDF value, and add them together
- Generate ray samples from a distribution that matches the BRDF for the given incident direction and average them samples together
  - This technique is called “Monte Carlo Integration”.

![Diagram of reflection types: directional diffuse, specular, uniform diffuse]
Improved Illumination

Ray Tracing can be adapted to handle many global illumination cases

- Specular-to-specular
- Specular-to-diffuse
- Diffuse-to-diffuse
- Diffuse-to-specular
- Caustics (focused light)
Next time

- Rendering equation
- Global illumination
  - Path tracing
  - Photon mapping
  - Radiosity