To Do

- START EARLY on HW 4
- Milestone is due on Mar 8

Outline

- Camera Ray Casting (choose ray directions)
- Ray-object intersections
- Ray-tracing transformed objects
- Lighting calculations
- Recursive ray tracing

Outline in Code

```
Image Raytrace (Camera cam, Scene scene, int width, int height)
{
    Image image = new Image (width, height) ;
    for (int i = 0 ; i < height ; i++)
        for (int j = 0 ; j < width ; j++) {
            Ray ray = RayThruPixel (cam, i, j) ;
            Intersection hit = Intersect (ray, scene) ;
            image[i][j] = FindColor (hit) ;
        }
    return image ;
}
```
Finding Ray Direction

- Goal is to find ray direction for given pixel i and j
  - Many ways to approach problem
    - Objects in world coord, find dim of each ray (we do this)
    - Camera in canonical frame, transform objects (OpenGL)
  - Basic idea
    - Ray has origin (camera center) and direction
    - Find direction given camera params and i and j
  - Camera params as in gluLookAt
    - Lookfrom[3], LookAt[3], up[3], fov

Similar to gluLookAt derivation

- gluLookAt(eyex, eyey, eyez, centerx, centery, centerz, upx, upy, upz)
  - Camera at eye, looking at center, with up direction being up

Constructing a coordinate frame?

We want to associate w with a, and v with b
  - But a and b are neither orthogonal nor unit norm
  - And we also need to find u

\[
\begin{align*}
  w &= a \\
  u &= b \times w \\
  v &= w \times u
\end{align*}
\]

Camera coordinate frame

\[
\begin{align*}
  w &= a \\
  u &= b \times w \\
  v &= w \times u
\end{align*}
\]

- We want to position camera at origin, looking down –Z dirn
- Hence, vector a is given by eye – center
- The vector b is simply the up vector

Canonical viewing geometry

\[
\begin{align*}
  \alpha &= \tan \left( \frac{\text{fovx}}{2} \times \left( \frac{j - (\text{width} / 2)}{\text{width} / 2} \right) \right) \\
  \beta &= \tan \left( \frac{\text{fovy}}{2} \times \left( \frac{(\text{height} / 2) - i}{\text{height} / 2} \right) \right)
\end{align*}
\]

Outline

- Camera Ray Casting (choosing ray directions)
- Ray-object intersections
- Ray-tracing transformed objects
- Lighting calculations
- Recursive ray tracing
Outline in Code

```c
Image Raytrace (Camera cam, Scene scene, int width, int height)
{
    Image image = new Image (width, height) ;
    for (int i = 0 ; i < height ; i++)
        for (int j = 0 ; j < width ; j++) {
            Ray ray = RayThruPixel (cam, i, j) ;
            Intersection hit = Intersect (ray, scene) ;
            image[i][j] = FindColor (hit) ;
        }
    return image ;
}
```

Ray-Sphere Intersection

$$ray = \vec{P} = \vec{P}_0 + \vec{P}_t$$

$$sphere = (\vec{P} - \vec{C}) \cdot (\vec{P} - \vec{C}) - r^2 = 0$$

Substitute

$$ray = \vec{P} = \vec{P}_0 + \vec{P}_t$$

$$sphere = (\vec{P}_0 + \vec{P}_t - \vec{C}) \cdot (\vec{P}_0 + \vec{P}_t - \vec{C}) - r^2 = 0$$

Simplify

$$t^2(\vec{P}_t \cdot \vec{P}_t) + 2t(\vec{P}_t \cdot (\vec{P}_0 - \vec{C})) + (\vec{P}_0 - \vec{C}) \cdot (\vec{P}_0 - \vec{C}) - r^2 = 0$$

Solve quadratic equations for t

- 2 real positive roots: pick smaller root
- Both roots same: tangent to sphere
- One positive, one negative root: ray origin inside sphere (pick + root)
- Complex roots: no intersection (check discriminant of equation first)

Ray-Sphere Intersection

- Intersection point: $$ray = \vec{P} = \vec{P}_0 + \vec{P}_t$$
- Normal (for sphere, this is same as coordinates in sphere frame of reference, useful other tasks)
  $$normal = \frac{\vec{P} - \vec{C}}{|\vec{P} - \vec{C}|}$$

Ray-Triangle Intersection

- One approach: Ray-Plane intersection, then check if inside triangle
  $$plane = \vec{P} \cdot \vec{n} - \vec{A} \cdot \vec{n} = 0$$
Ray-Triangle Intersection

- One approach: Ray-Plane intersection, then check if inside triangle
- Plane equation:
  \[ \text{plane} : P \cdot \vec{n} - A \cdot \vec{n} = 0 \]
- Combine with ray equation:
  \[ \text{ray} : (P_0 + t \cdot P_1) \cdot \vec{n} = A \cdot \vec{n} \]

Ray inside Triangle

- Once intersect with plane, still need to find if in triangle
- Many possibilities for triangles, general polygons (point in polygon tests)
- We find parametrically [barycentric coordinates]. Also useful for other applications (texture mapping)

Other primitives

- Much early work in ray tracing focused on ray-primitive intersection tests
- Cones, cylinders, ellipsoids
- Boxes (especially useful for bounding boxes)
- General planar polygons
- Many more
- Consult chapter in Glassner (handed out) for more details and possible extra credit

Ray Scene Intersection

```
Intersection FindIntersection(Ray ray, Scene scene) {
    min_t = infinity
    min_primitive = NULL
    For each primitive in scene {
        t = Intersect(ray, primitive);
        if (t > 0 & t < min_t) then
            min_t = t
            min_primitive = primitive
    }
    return Intersection(min_t, min_primitive)
}
```
Transformed Objects

- E.g. transform sphere into ellipsoid
- Could develop routine to trace ellipsoid (compute parameters after transformation)
- May be useful for triangles, since triangle after transformation is still a triangle in any case
- But can also use original optimized routines

Ray-Tracing Transformed Objects

We have an optimized ray-sphere test
- But we want to ray trace an ellipsoid...

Solution: Ellipsoid transforms sphere
- Apply inverse transform to ray, use ray-sphere
- Allows for instancing (traffic jam of cars)
- Same idea for other primitives

Transformed Objects

- Consider a general 4x4 transform $M$
  - Will need to implement matrix stacks like in OpenGL
- Apply inverse transform $M^{-1}$ to ray
  - Locations stored and transform in homogeneous coordinates
  - Vectors (ray directions) have homogeneous coordinate set to 0 [so there is no action because of translations]
- Do standard ray-surface intersection as modified
  - Transform intersection back to actual coordinates
    - Intersection point $p$ transforms as $Mp$
    - Distance to intersection if used may need recalculation
    - Normals $n$ transform as $M^{-1}n$. Do all this before lighting

Outline

- Camera Ray Casting (choosing ray directions)
- Ray-object intersections
- Ray-tracing transformed objects
- Lighting calculations
- Recursive ray tracing

Outline in Code

Image Raytrace (Camera cam, Scene scene, int width, int height)
{
  Image image = new Image (width, height) ;
  for (int i = 0 ; i < height ; i++)
    for (int j = 0 ; j < width ; j++) {
      Ray ray = RayThruPixel (cam, i, j) ;
      Intersection hit = Intersect (ray, scene) ;
      image[i][j] = FindColor (hit) ;
    }
  return image ;
}
Shadows: Numerical Issues
- Numerical inaccuracy may cause intersection to be below surface (effect exaggerated in figure)
- Causing surface to incorrectly shadow itself
- Move a little towards light before shooting shadow ray

Lighting Model
- Similar to OpenGL
- Lighting model parameters (global)
  - Ambient r g b
  - Attenuation const linear quadratic
    \[ L = \text{const} + \text{lin} \cdot d + \text{quad} \cdot d^2 \]
- Per light model parameters
  - Directional light (direction, RGB parameters)
  - Point light (location, RGB parameters)
  - Some differences from HW 2 syntax

Material Model
- Diffuse reflectance (r g b)
- Specular reflectance (r g b)
- Shininess s
- Emission (r g b)
- All as in OpenGL

Shading Model
\[ I = K_a + K_e + \sum_{m} K_d \max (I \cdot n, 0) + K_s \max (h \cdot n, 0)^s \]
- Global ambient term, emission from material
- For each light, diffuse specular terms
- Note visibility/shadowing for each light (not in OpenGL)
- Evaluated per pixel per light (not per vertex)

Outline
- Camera Ray Casting (choosing ray directions)
- Ray-object intersections
- Ray-tracing transformed objects
- Lighting calculations
- Recursive ray tracing

Mirror Reflections/Refractions
- Generate reflected ray in mirror direction,
  Get reflections and refractions of objects
**Basic idea**

For each pixel
- Trace Primary Eye Ray, find intersection
- Trace Secondary Shadow Ray(s) to all light(s)
  - Color = Visible ? Illumination Model : 0 ;
- Trace Reflected Ray
  - Color += reflectivity * Color of reflected ray

**Recursive Shading Model**

\[ I = K_s + \sum_{i=1}^{n} I_i \left( K_s \max (\mathbf{L}_i \cdot \mathbf{n}, 0) + K_s (\max (h_i \cdot n, 0))^2 \right) + K_f + K_a \]

- Highlighted terms are recursive specularities [mirror reflections] and transmission (latter is extra credit)
- Trace secondary rays for mirror reflections and refractions, include contribution in lighting model
- GetColor calls RayTrace recursively (the I values in equation above of secondary rays are obtained by recursive calls)

**Problems with Recursion**

- Reflection rays may be traced forever
- Generally, set maximum recursion depth
- Same for transmitted rays (take refraction into account)

**Effects needed for Realism**

- (Soft) Shadows
- Reflections (Mirrors and Glossy)
- Transparency (Water, Glass)
- Interreflections (Color Bleeding)
- Complex Illumination (Natural, Area Light)
- Realistic Materials (Velvet, Paints, Glass)

Discussed in this lecture so far
Not discussed but possible with distribution ray tracing
Hard (but not impossible) with ray tracing; radiosity methods

**Some basic add ons**

- Area light sources and soft shadows: break into grid of n x n point lights
  - Use jittering: Randomize direction of shadow ray within small box for given light source direction
  - Jittering also useful for antialiasing shadows when shooting primary rays
- More complex reflectance models
  - Simply update shading model
  - But at present, we can handle only mirror global illumination calculations
**Acceleration**

Testing each object for each ray is slow
- Fewer Rays
  - Adaptive sampling, depth control
- Generalized Rays
  - Beam tracing, cone tracing, pencil tracing etc.
- Faster Intersections
  - Optimized Ray-Object Intersections
  - Fewer Intersections

**Acceleration Structures**

Bounding boxes (possibly hierarchical)
  - If no intersection bounding box, needn’t check objects

Spatial Hierarchies (Oct-trees, kd trees, BSP trees)

---

**Bounding Volume Hierarchies 1**

- Build hierarchy of bounding volumes
  - Bounding volume of interior node contains all children

![Bounding Volume Hierarchies 1 Diagram](image1)

**Bounding Volume Hierarchies 2**

- Use hierarchy to accelerate ray intersections
  - Intersect node contents only if hit bounding volume

![Bounding Volume Hierarchies 2 Diagram](image2)

**Bounding Volume Hierarchies 3**

- Sort hits & detect early termination

```c
FindIntersection(Ray ray, Node node)
{
    // Find intersections with child node bounding volumes
    ...
    // Sort intersections front to back
    ...
    // Process intersections (checking for early termination)
    min_t = infinity;
    for each (intersected child)
    {
        if (min_t > bv_1()) break;
        shape_t = FindIntersection(ray, child);
        if (shape_t < min_t) { min_t = shape_t; }
    }
    return min_t;
}
```

**Acceleration Structures: Grids**

![Acceleration Structures: Grids Diagram](image3)
Uniform Grid: Problems

- Potential problem:
  - How choose suitable grid resolution?

Octree

- Construct adaptive grid over scene
  - Recursively subdivide box-shaped cells into 8 octants
  - Index primitives by overlaps with cells

Other Accelerations

- Screen space coherence
  - Check last hit first
  - Beam tracing
  - Pencil tracing
  - Cone tracing

- Memory coherence
  - Large scenes

- Parallelism
  - Ray casting is “embarrassingly parallelizable”

- etc.

Octree traversal

- Trace rays through neighbor cells
  - Fewer cells
  - More complex neighbor finding

Trade-off fewer cells for more expensive traversal

Ray Tracing Acceleration Structures

- Bounding Volume Hierarchies (BVH)
- Uniform Spatial Subdivision (Grids)
- Binary Space Partitioning (BSP Trees)
  - Axis-aligned often for ray tracing: kd-trees
- Conceptually simple, implementation a bit tricky
  - Lecture relatively high level: Start early, go to section
  - Remember that acceleration a small part of grade

CAPE Evaluations

- Fill out now, can be done on phone
- Enthusiasm important to future offerings (one of first time in winter this year, many enrollments 167)
- Comments useful to future years
- Some key innovations: modern OpenGL, GLSL; feedback servers (including code), edX edge, ...

- Separately, please also evaluate the TAs
Math of 2D Bounding Box Test

- Can you find a t in range
  \[ t > 0 \]
  \[ t_{\text{min}} \leq t \leq t_{\text{max}} \]
  \[ t_{\text{ymin}} \leq t \leq t_{\text{ymax}} \]

- If \( t_{\text{xmin}} > t_{\text{ymin}} \) OR \( t_{\text{ymin}} > t_{\text{ymax}} \)
  return false;
else
  return true;

No intersection if x and y ranges don’t overlap

Bounding Box Test

- Ray-Intersection is simple coordinate check
- Intricacies with test, see book
- Hierarchical Bounding Boxes

Hierarchical Bounding Box Test

- If ray hits root box
  - Intersect left subtree
  - Intersect right subtree
  - Merge intersections (find closest one)
- Standard hierarchical traversal
  - But caveat, since bounding boxes may overlap
- At leaf nodes, must intersect objects

Creating Bounding Volume Hierarchy

function bvh-node::create (object array A, int AXIS)
N = A.length();
if (N == 1) {left = A[0]; right = NULL; bbox = bound(A[0]);}
else if (N == 2) {
  left = A[0]; right = A[1];
bbox = combine(bound(A[0]),bound(A[1]));
} else
Find midpoint m of bounding box of A along AXIS
Partition A into lists of size k and N-k around m
left = new bvh-node(A[0…k],(AXIS+1) mod 3);
right = new bvh-node(A[k+1…N-1],(AXIS+1) mod 3);
bbox = combine(left -> bbox, right -> bbox);

From page 305 of book

Uniform Spatial Subdivision

- Different idea: Divide space rather than objects
- In BVH, each object is in one of two sibling nodes
  - A point in space may be inside both nodes
- In spatial subdivision, each space point in one node
  - But object may lie in multiple spatial nodes
- Simplest is uniform grid (have seen this already)
- Challenge is keeping all objects within cell
- And in traversing the grid

Traversal of Grid High Level

- Next Intersect Pt?
- Irreg. samp. pattern
- But regular in planes
- Fast algo. possible
- (more on board)
**BSP Trees**

- Used for visibility and ray tracing
  - Book considers only axis-aligned splits for ray tracing
  - Sometimes called kd-tree for axis aligned
- Split space (binary space partition) along planes
- Fast queries and back-to-front (painter’s) traversal
- Construction is conceptually simple
  - Select a plane as root of the sub-tree
  - Split into two children along this root
  - Random polygon for splitting plane (may need to split polygons that intersect it)

BSP slides courtesy Prof. O’Brien
Final BSP Tree

BSP Trees Cont’d
- Continue splitting until leaf nodes
- Visibility traversal in order
  - Child one
  - Root
  - Child two
- Child one chosen based on viewpoint
  - Same side of sub-tree as viewpoint
- BSP tree built once, used for all viewpoints
  - More details in book

Interactive Raytracing
- Ray tracing historically slow
- Now viable alternative for complex scenes
  - Key is sublinear complexity with acceleration;
    need not process all triangles in scene
- Allows many effects hard in hardware
- Today graphics hardware and software (NVIDIA Optix 5, RTX chips claim 10G rays per second). [Video]

Raytracing on Graphics Hardware
- Modern Programmable Hardware general streaming architecture
- Can map various elements of ray tracing
  - Kernels like eye rays, intersect etc.
- In vertex or fragment programs
- Convergence between hardware, ray tracing
  [Purcell et al. 2002, 2003]
  http://graphics.stanford.edu/papers/photongfx