To Do

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

Outline

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

Outline in Code

```java
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;
}
```

Heckbert’s Business Card Ray Tracer

```c
typedef struct (double x, y, z, vec3color, vec3ambient, vec3light) {
    double x, y, z;
    vec3color color;
    vec3ambient ambient;
    vec3light light;
} bp;
```

```
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

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

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

- 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

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

\[ \text{ray} = \vec{P} = \vec{P}_0 + \vec{P}_t \]
\[ \text{sphere} = (\vec{P} - \vec{C}) \cdot (\vec{P} - \vec{C}) - r^2 = 0 \]

Substitute
\[ \text{ray} = \vec{P} = \vec{P}_0 + \vec{P}_t \]
\[ \text{sphere} = (\vec{P}_0 + \vec{P}_t - \vec{C}) \cdot (\vec{P}_0 + \vec{P}_t - \vec{C}) - r^2 = 0 \]

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

Ray-Triangle Intersection

- One approach: Ray-Plane intersection, then check if inside triangle
- Plane equation:
  \[ n = \frac{(C - A) \times (B - A)}{|(C - A) \times (B - A)|} \]
  \[ plane = \vec{P} \cdot \hat{n} - \hat{A} \cdot \hat{n} = 0 \]
Ray-Triangle Intersection

- One approach: Ray-Plane intersection, then check if inside triangle
- Plane equation:
  \[ \text{plane} = \vec{P} \cdot \vec{n} - \vec{A} \cdot \vec{n} = 0 \]
- Combine with ray equation:
  \[ \text{ray} = \vec{P} = \vec{P}_0 + \vec{P}_1 t \]
  \[ (\vec{P}_0 + \vec{P}_1 t) \cdot \vec{n} = \vec{A} \cdot \vec{n} \]
  \[ t = \frac{\vec{A} \cdot \vec{n} - \vec{P}_0 \cdot \vec{n}}{\vec{P}_1 \cdot \vec{n}} \]

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)

\[ \vec{P} = \alpha \vec{A} + \beta \vec{B} + \gamma \vec{C} \]
\[ \alpha \geq 0, \beta \geq 0, \gamma \geq 0 \]
\[ \alpha + \beta + \gamma = 1 \]

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_primitive = primitive
            min_t = t
    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

```java
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

Shadow ray to light is unblocked: object visible
Shadow ray to light is blocked: object in shadow
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
    \[ L_i = \text{const + lin} \cdot d + \text{quad} \cdot d^2 \]
- 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

\[ I = K_a + K_e + \sum L_i (K_d \max (l_i \cdot n,0) + K_s (\max (h_i \cdot n,0))^s) \]

- 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

\[ I = K_s + K_r + \sum_{i=1}^{n} L_i (K_a \max (l_i \cdot n, 0) + K_s (\max(h_i \cdot n, 0))^t) + K_{IR} + K_{IT} \]

- 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 2

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

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 i |
        if(min_t > bc[i].t) break;
        shape_t = FindIntersection(ray, child);
        if(shape_t < min_t | min_t = shape_t;)
    return min_t;
}
```
Uniform Grid: Problems

- Potential problem:
  - How choose suitable grid resolution?

  ![Grid Diagram](image)
  - Too little benefit if grid is too coarse
  - Too much cost if grid is too fine

Octree

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

  ![Octree Diagram](image)
  - Generally fewer cells

Octree traversal

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

  ![Traversal Diagram](image)
  - Trade-off fewer cells for more expensive traversal

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.

CAPE Evaluations

- Fill out now, can be done on phone
- Enthusiasm important to future offerings (one of first time in winter this year, many enrollments167)
- Comments useful to future years
- Some key innovations: modern OpenGL, GLSL; feedback servers (including code), UCSD online, ...

- Separately, please also evaluate the TAs

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
Math of 2D Bounding Box Test

- Can you find a $t$ in range $t > 0$
  \[ t_{x_{\min}} \leq t \leq t_{x_{\max}} \]
  \[ t_{y_{\min}} \leq t \leq t_{y_{\max}} \]

- if $t_{x_{\max}} > t_{y_{\min}}$ OR $t_{y_{\max}} > t_{x_{\min}}$
  \[ \text{return false;} \]
- else \[ \text{return true;} \]

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 \( (N == 1) \) \{ \( \text{left} = A[0]; \text{right} = \text{NULL}; \text{bbox} = \text{bound}(A[0]); \}\)
else if \( (N == 2) \) \{
  \( \text{left} = A[0]; \text{right} = A[1]; \)
  \( \text{bbox} = \text{combine} (\text{bound}(A[0]), \text{bound}(A[1])); \)
}\else

Find midpoint $m$ of bounding box of $A$ along $AXIS$
Partition $A$ into lists of size $k$ and $N-k$ around $m$
\( \text{left} = \text{new bvh-node (A[0...k],(AXIS+1) mod 3)); \}
\( \text{right} = \text{new bvh-node(A[k+1...N-1],(AXIS+1) mod 3));} \)
\( \text{bbox} = \text{combine (left -> bbox, right -> bbox);} \);

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
### 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).

### 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

---

### Final BSP Tree

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
- 168 lectures (UCSD online) more detail re accel