Lecture 16: Ray Tracing

Rui Wang
Outline

- What is ray tracing?
- Brief history
- Basic algorithm
- Ray-shape intersection

- Constructing camera rays
- Illumination and shading, shadows
- Reflection, refraction.
- Ray tracing acceleration structures
Illumination and Shading

Camera camera;
vector<Primitive> shapes;
vector<Light> lights;
int im_width, im_height;

Image render(int width, int height)
{
    Image image = new Image(width, height);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            Ray ray = camera.getCameraRay(i, j);
            hit = IntersectScene(ray, shapes);
            image[i][j] = getShadingColor(ray, hit);
        }
    }
    return image;
}
Construct Camera Rays

- Camera is typically given by **eye** point, **target** point, **up** vector.
Construct Camera Rays

Camera is typically given by eye point, target point, up vector.
Construct Camera Rays

- Camera is typically given by eye point, target point, up vector.
- Camera rays (aka eye rays, primary rays)
- Definition of Ray?
  - Starting point $P_0$
  - Travel direction $D$
  - $P(t) = P_0 + tD$
Construct Camera Rays

- **Camera rays**
  - Starting point: eye point
  - \( P(i,j) \rightarrow \) pixel pos in 3D
  - Direction = \([P(i,j) - \text{eye}]\)
Let’s think of a perspective camera.
Construct Camera Rays

- Camera rays (eye rays)
  - Think of a perspective camera.
Construct Camera Rays

- Camera rays (eye rays)
  - Think of a perspective camera.

\[-\text{eye} (0,0,-1)\]
\[-(0,\tan(\text{fovy}/2),-1)\]
\[-(0,0,-1)\]
Construct Camera Rays

- Camera rays (eye rays)
  - Think of a perspective camera.

\[
\begin{align*}
\text{as} &= \text{aspect ratio} = \frac{\text{float(im\_width)}}{\text{float(im\_height)}} \\
(0, \tan(\text{fovy}/2), -1) \\
(0, 0, -1) \\
(\text{as} \cdot \tan(\text{fovy}/2), 0, -1)
\end{align*}
\]
Construct Camera Rays

Camera rays (eye rays)
- Establish pixel correspondence

- (left, top, -1)
- (right, top, -1)
- (left, bottom, -1)
- (right, bottom, -1)
Illumination and Shading

Camera camera;
vector<Primitive> shapes;
vector<Light> lights;
int im_width, im_height;

Image render(int width, int height)
{
    Image image = new Image(width, height);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            Ray ray = camera.getCameraRay(i, j);
            hit = IntersectScene(ray, shapes);
            image[i][j] = getShadingColor(ray, hit);
        }
    }
    return image;
}
Illumination and Shading

- Use the shading formula we learned, but additionally send **shadow rays** to account for shadows.

- Loop over all lights to sum up the contributions from all

- Shadow hits can be optimized and made faster than standard hits.
rgb getShadingColor(ray, hit) {
    rgb c(0);
    for (int i = 0; i < lights.size(); i ++) {
        lights[i].getLight(hit.p, light_pos, light_dir, light_intensity);
        Ray sr(hit.p, light_dir);
        float light_dist = (light_pos - hit.p).length();
        SurfaceHitRecord shadow_hit;

        if (IntersectScene(sr, shadow_hit) && shadow_rec.t < light_dist)
            continue; /* shadowed */

        /* Compute shading color
        L -> lights_dir;
        N -> hit.n;
        V -> -ray.direction();
        material -> hit.material (Kd, Ks...); */
    }
    return c;
}
rgb getShadingColor(ray, hit) {
    rgb c(0);
    for (int i = 0; i < lights.size(); i++) {
        lights[i].getLight(hit.p, light_pos, light_dir, light_intensity);
        Ray sr(hit.p, light_dir);
        float light_dist = (light_pos - hit.p).length();
        SurfaceHitRecord shadow_hit;

        if (IntersectScene(sr, shadow_hit) && shadow_rec.t < light_dist)
            continue; /* shadowed */

        /* Compute shading color */
        L -> lights_dir;
        N -> hit.n;
        V -> -ray.direction();
        material -> hit.material (Kd, Ks...); */
    }
    return c;
}
rgb getShadingColor(ray, hit) {
    rgb c(0);
    for (int i = 0; i < lights.size(); i++) {
        lights[i].getLight(hit.p, light_pos, light_dir, light_intensity);
        Ray sr(hit.p, light_dir);
        float light_dist = (light_pos - hit.p).length();
        SurfaceHitRecord shadow_hit;

        if (IntersectScene(sr, shadow_hit) && shadow_rec.t < light_dist)
            continue; /* shadowed */

    /* Compute shading color */
    L -> lights_dir;
    N -> hit.n;
    V -> -ray.direction();
    material -> hit.material (Kd, Ks...); */
    } return c;
}
rgb getShadingColor(ray, hit) {
    rgb c(0);
    for (int i = 0; i < lights.size(); i++) {
        lights[i]->getLight(hit.p, light_pos, light_dir, light_intensity);
        Ray sr(hit.p, light_dir);
        float light_dist = (light_pos - hit.p).length();
        SurfaceHitRecord shadow_hit;

        if (IntersectScene(sr, shadow_hit) && shadow_rec.t < light_dist)
            continue; /* shadowed */

        /* Compute shading color */
        L -> lights_dir;
        N -> hit.n;
        V -> -ray.direction();
        material -> hit.material (Kd, Ks...); */
    }
    return c;
}
Shadow Rays: Numerical Issues

- Numerical inaccuracy may cause shadow ray to fall slightly below surface (effect exaggerated in figure)
- Causing surface to incorrectly shadow itself
Shadow Rays: Numerical Issues
Shadow Rays: Numerical Issues

- Solution: use a small $t_{min}$ value when compute intersection.
  - Avoid intersection with geometry that is too close.
  - Think of $z_{near}$ in OpenGL.

- When implementing ray-shape intersection, be sure to check with both $t_{min}$ and $t_{max}$.
  - Look at Plane.java as an example.
Ray-Shape Intersection (many shapes)

```c
bool IntersectScene(ray, shapes, SurfaceHitRecord& hit) {
    float min_t = FLT_MAX;
    bool hit_something = false;
    for (i = 0; i < number_of_shapes; i++) {
        SurfaceHitRecord temp_hit;
        if (shapes[i]->hit(ray, 0.001, max_t, temp_hit)) {
            if (temp_hit.t < min_t) {
                min_t = temp_hit.t;
                hit = temp_hit;
                hit_something = true;
            }
        }
    }
    return hit_something;
}
```
Types of Light Sources

- `Light::getLight(const Point& p, Point& pos, Vector& dir, rgb& intensity);`

- **Point** (look at PointLight.java)
  - Quadratic intensity fall off as $1/r^2$

- **Spot**
  - Similar to point, but has a circle of influence.
  - Think of flashlights
Types of Light Sources

- Area (selective features)
  - Soft shadows (penumbra)
  - Easiest way is to discretize the area light to many small point lights
Types of Light Sources

- Ambient Occlusion Shading
Types of Light Sources

- Image-based Lighting
  - Taking into account complex environment lighting
Types of Light Sources

- Image-based Lighting
  - Taking into account complex environment lighting

Image by Paul Debevec 2002
Types of Light Sources

- Image-based Lighting
  - Taking into account complex environment lighting

*Image by Paul Debevec 2002*
Ray Tracing Structure

```java
Camera camera;
vector<Primitive> shapes;
vector<Light> lights;
int im_width, im_height;

Image render(int width, int height)
{
    Image image = new Image(width, height);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            Ray ray = camera.getCameraRay(i, j);
            hit = IntersectScene(ray, shapes);
            image[i][j] = getShadingColor(ray, hit);
        }
    }
    return image;
}
```
Reflection

- When a ray intersects a reflective surface, simply trace recursively along the reflected direction.
Reflection

☐ When a ray intersects a reflective surface, simply trace recursively along the reflected direction

☐ $c = Kr \times \text{color of reflected ray}$
Reflection

- Reflected ray continues to hit other objects (or hit nothing)
- \( c = K_r \times \text{color of reflected ray} \)
Reflection

- Reflected ray continues to hit other objects (or hit nothing)
- \( c = K_r \times \text{color of reflected ray} \)
- Reflection formula?
Reflection

- Reflected ray continues to hit other objects (or hit nothing)
- It is possible that the reflected ray hits another reflective object.
Problem with Recursion

- Reflected rays must be traced repeatedly until they terminate on non-reflective surface.

- What’s the problem?
Problem with Recursion

- Reflected rays must be traced repeatedly until they terminate on non-reflective surface.

- This may cause infinite recursion!

- Solution: set maximum recursion depth
  - Typically 5
  - Depth is initialized to 0, every time recursion happens, depth is incremented.
Refraction

- Works exactly the same way as reflection
Refraction

- Works exactly the same way as reflection
- Refraction Formula?
Putting Everything Together

Raytracing(Ray& ray, int depth)
1. Find intersection of ray with scene
2. If a hit point is found {
   1. If ((Kr!=0 or Kt != 0) && depth < maxDepth)
      Color += Kr * Raytracing(refl_ray, depth+1);
      Color += Kt * Raytracing(refr_ray, depth+1)
Putting Everything Together

Raytracing(Ray& ray, int depth)
1. Find intersection of ray with scene
2. If a hit point is found {
   1. If ((Kr!=0 or Kt != 0) && depth < maxDepth)
      Color += Kr * Raytracing(refl_ray, depth+1);
      Color += Kt * Raytracing(refr_ray, depth+1)
   2. Compute shading color (getShadingColor function)
      - Loop over all light sources
      - Call each light’s getLight function
      - Construct and send shadow rays
      - If no intersection found, compute diffuse and Phong specular color using shading formula.
Sampling

- Aliasing
  - Super Sampling
  - Jittered Samples
  - Adaptive Sampling
Acceleration Structure

- Uniform Grid
Bounding Volume Hierarchy (BVH)

- If a ray doesn’t hit the bounding volume, it doesn’t hit the object.
Other Structures

- BSP (Binary Space Partitioning) tree
- Kd-Tree
- Octree
- ...

Bottom line: hierarchically skip unnecessary intersection computations
Other Structures

- BSP (Binary Space Partitioning) tree
- Kd-Tree
- Octree
- ...

- Bottom line: hierarchically skip unnecessary intersection computations
- What is the theoretical complexity?
- How to handle dynamic scenes?