Raytracing Implementation

Friday 13th(!!!) Nov 2015

Raytracer Project

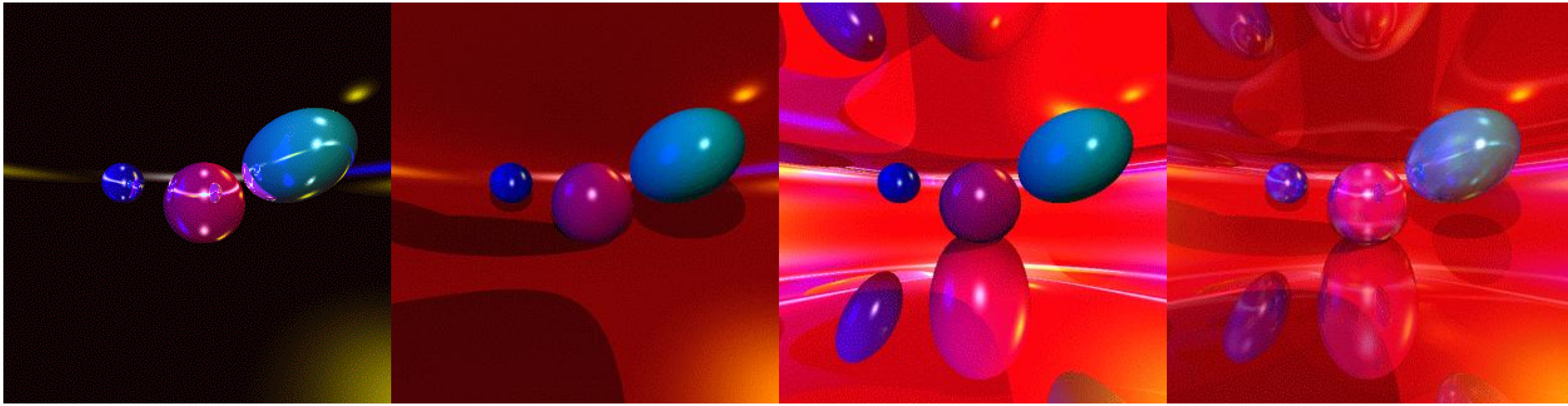
- Instructions
- Sample images
- Real-time OpenGL adaptation

Instructions

- You will be implementing a Ray Tracer.
- Your system need only handle the rendering of spheres
- Certain things ensure your output matches ours
  - Camera situated at the origin
  - Right handed, negative Z
  - Local illumination, reflections, and shadows must be implemented

Raytracer Project

- Pros:
  - Simple C++ Template
  - No code learning curve or OpenGL calls
  - Yet results look this good



Raytracer Project

- Cons:
  - Not real time
  - Project focuses on file input/output instead
  - No OpenGL (including key controls, image display)
  - Spec says you can use OpenGL for debugging, sounds impractical
- But…
  - We give you a big head start; more starting code than the spec says

Results

- We give you several sample test files and the expected output images
- Some of the output images were made with an imperfect implementation
- Exact match is less important than physical correctness of the lighting
- Extra step: Culling objects inside the near plane

Phong-Blinn Shading model

- Still applies
- Ambient, diffuse, specular
- Still get different specular results from using $(R \text{ dot } V)^n$ versus the halfway vector $(H \text{ dot } N)^n$
- We can add in additional terms to Phong-Blinn since it's a raytracer – reflections and refractions

Phong-Blinn Shading model

Exact model you're supposed to use:

PIXEL_COLOR[c] = K_a*I_a[c]*O[c] +
for each point light source (p) { K_d*I_p[c]*(N.L)*O[c]+K_s*I_p[c]*$(R.V)^n$ } +
K_r*(Color returned from reflection ray)

File Parsing

- Load a text file into your data structures
- Formatted like this:

NEAR <n>
LEFT <l>
RIGHT <r>
BOTTOM <b>
TOP <t>
RES <x> <y>
SPHERE <name> <pos x> <pos y> <pos z> <scl x> <scl y> <scl z> <r> <g> <b> <K_a> <K_d> <K_s> <K_r> <n>
LIGHT <name> <pos x> <pos y> <pos z> <I_r> <I_g> <I_b>
BACK <r> <g > <b>
AMBIENT <I_r> <I_g> <I_b>
OUTPUT <name>

File Parsing

```
void parseLine(const vector<string>& vs)
```

- All that's left for you to implement is the switch statement that handles each type of line
- How to make a switch statement branch based on a string? Cleaner than if's

```
const int num_labels = 11; //0      1      2              3       4      5      6             7       8       9
const string labels[] = { "NEAR", "LEFT", "RIGHT", "BOTTOM", "TOP", "RES", "SPHERE", "LIGHT", "BACK", "AMBIENT", "OUT
unsigned label_id = find(  labels, labels + num_labels, vs[0] ) - labels;

switch( label_id )
{
       case 0:       g_near   = toFloat( vs[1] );             break; // NEAR
       case 1:       g_left   = toFloat( vs[1] );             break; // LEFT
       case 2:       g_right  = toFloat( vs[1] );             break; // RIGHT
       case 3:       g_bottom = toFloat( vs[1] );             break; // BOTTOM
       case 4:       g_top    = toFloat( vs[1] );             break; // TOP
```

Given functions

- `void parseLine(const vector<string>& vs)`
- `void loadFile(const char* filename)`
- `vec4 trace(const Ray& ray`
- `void savePPM(int Width, int Height, char* fname, unsigned char* pixels)`

- A matrix library – simpler and dumbed down functionality – still can do everything you need (inverse, translate, scale, *=, +, dot)
- Casting: You can say vec4(vec3 blah, 1)

Given functions

```
float toFloat(const string& s)
{
    stringstream ss(s);
    float f;
    ss >> f;
    return f;
}
```

Given functions

```
void renderPixel(int ix, int iy)
{
    Ray ray;
    ray.origin = vec4(0.0f, 0.0f, 0.0f, 1.0f);
    ray.dir = getDir(ix, iy);
    vec4 color = trace(ray, vec3(1,1,1) );
    setColor(ix, iy, color);
}

void render()
{
    for (int iy = 0; iy < g_height; iy++)
        for (int ix = 0; ix < g_width; ix++)
            renderPixel(ix, iy);
}
```

A ray

```
struct Ray
{
    vec4 origin;
    vec4 dir;
};
```

Order to do things:

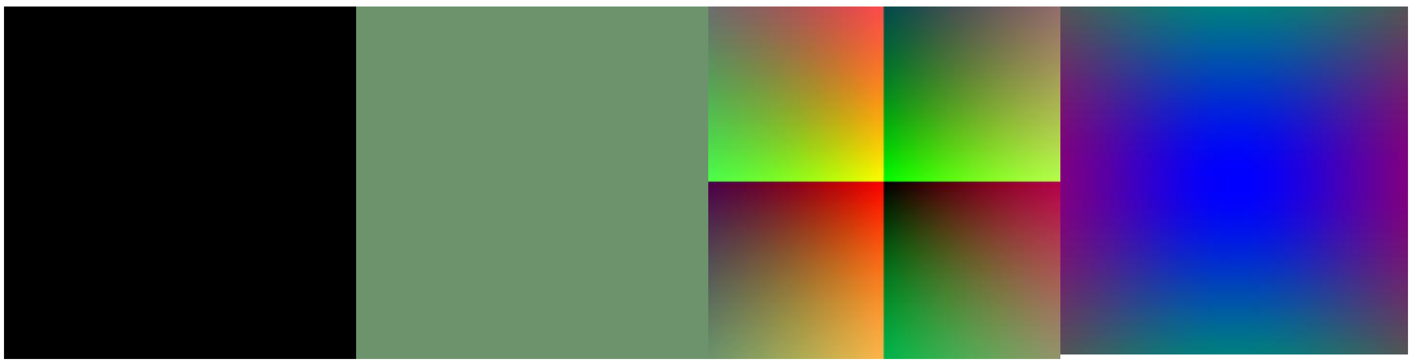
- Build classes Sphere and Light to store all the values you load from the files
- Instantiate a vector<Sphere> and vector<Light>
- Fill in function getDir() to generate your ray directions
- Fill a placeholder into traceRay() so you can see them
- Create a intersect(Sphere&) function

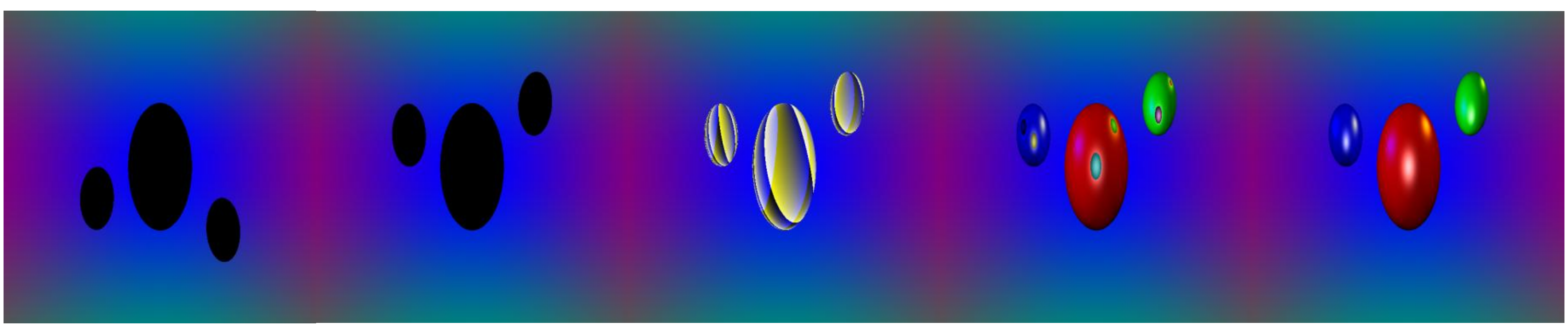
getDir(): Turn screen space pixels into world space rays

```
vec4 getDir(int ix, int iy)
{
    // TODO: modify this. This should return the direction from the origin
    // to pixel (ix, iy), normalized.
    vec4 dir( 0.0f, 0.0f, -1.0f, 0.0f);
    return normalize(dir);
}
```

Debugging a ray tracer

- Set colors to intermediate values to help you picture your vectors, and verify them





Helpful utility function

```
inline vec3 toVec3( vec4 in )
  { return vec3( in[0], in[1], in[2] ); }
```

From Lecture slides

Final Intersection

Inverse transformed ray

$$\mathbf{r}'(t) = \mathrm{M}^{-1} \begin{bmatrix} S_x \\ S_y \\ S_z \\ 1 \end{bmatrix} + t\mathrm{M}^{-1} \begin{bmatrix} c_x \\ c_y \\ c_z \\ 0 \end{bmatrix} = S' + t\mathbf{c}'$$

- Drop 1 and 0 to get $\mathbf{r}'(t)$ in 3D space

For each object

- Inverse transform ray, getting $S' + t\mathbf{c}'$
- Find t_h for intersection with the untransformed object
- Use t_h in the **untransformed ray** $S + t\mathbf{c}$ to find the point of intersection with the transformed object

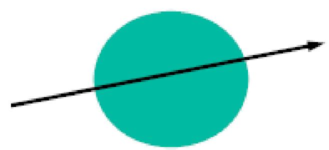
From Lecture slides

Ray-Object Intersections

Intersection of ray with unit sphere at origin:

$$\text{ray}(t) = S + t\mathbf{c}$$

$$\text{Sphere}(P) = |P| - 1 = 0$$



$$\text{Sphere}(\text{ray}(t)) = 0 \Rightarrow$$

$$|S + t\mathbf{c}| - 1 = 0 \Rightarrow$$

$$(S + t\mathbf{c}) \cdot (S + t\mathbf{c}) - 1 = 0 \Rightarrow$$

$$\boxed{|\mathbf{c}|^2 t^2 + 2(S \cdot t\mathbf{c}) + |S|^2 - 1 = 0}$$

This is a quadratic equation

Performance

- Debug vs Release mode
- Most expensive function: 4x4 matrix inverse(). Need it for colliding with a ray.
  – When to compute this for your spheres?

Most useful lecture slides

Solving a Quadratic Equation

$$|\mathbf{c}|^2 t^2 + 2(S \cdot \mathbf{c})t + |S|^2 - 1 = 0$$
$$At^2 + 2Bt + C = 0$$

$$t_h = -\frac{B}{A} \pm \frac{\sqrt{B^2 - AC}}{A}$$
$$= -\frac{S \cdot \mathbf{c}}{|\mathbf{c}|^2} \pm \frac{\sqrt{(S \cdot \mathbf{c})^2 - |\mathbf{c}|^2 (|S|^2 - 1)}}{|\mathbf{c}|^2}$$

If $(B^2 - AC) = 0$ one solution

If $(B^2 - AC) < 0$ no solution

If $(B^2 - AC) > 0$ two solutions

Handling both intersections (determinant positive)

```
// Use the lesser of the two, unless that would be a degenerately near (re-)hit
if( hit_1 < recollision_threshold && hit_1 > minimum_dist  )
      hit_1 = hit_2;

if( hit_1 < recollision_threshold && !(hit_1 > minimum_dist) ) return false;
```

Most useful lecture slides

Summary: Raytracing

Recursive algorithm

function Main

for each pixel (c,r) on screen

determine ray $r_{c,r}$ from eye through pixel

color(c,r) = raytrace($r_{c,r}$)

end for

end

function raytrace(r)

find closest intersection P of ray r with objects

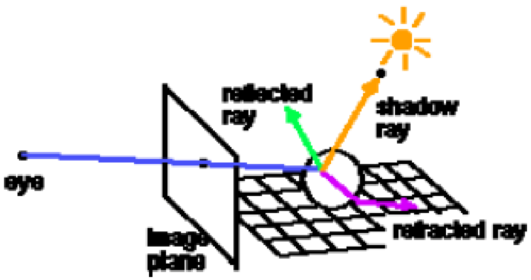
clocal = Sum(shadowRays(P,Light$_i$))

c_{rfl} = raytrace(r_{rfl})

c_{rfa} = raytrace(r_{rfa})

return c = clocal + k_{rfl}*c_{rfl} + k_{rfa}*c_{rfa}

end



Example Real-time OpenGL adaptation

- Example program with live display of pixels (uses cones sticking toward your face)
- A data structure to store all objects to accelerate collision lookup (hash table buckets)
- Foveated – more samples taken in the center
- Refraction rays traced too