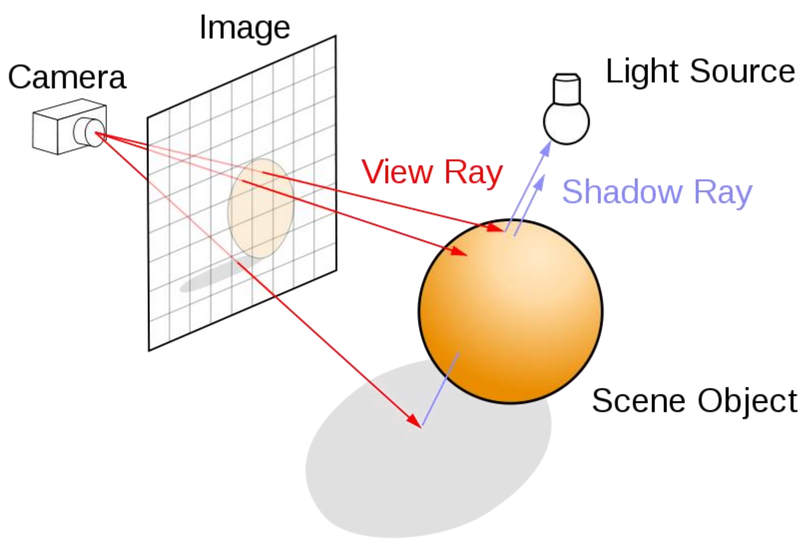
# Raytracing Tutorial 1

# Agenda

- What is Raytracing?
- XML File Spec and Parsing
- Constructing Camera Rays
- Sphere Intersection Test
- Local Illumination
- Shadows

# What is Raytracing?




- A method of computing global illumination
- Simulates how light rays travel through a scene

- **Basic algorithm:**
  - Construct ray from camera to each pixel
  - Find closest object hit by that ray
  - Cast shadow-, reflection- and refraction rays
  - Compute illumination according to some illumination model

# Raytracing Algorithm

```
for(y = 0; y < imageHeight; ++v)
    for(x = 0; x < imageWidth; ++x)
        ray = camera.getRayToPixel(x, y)
        color = trace(ray)
        image[x][y] = color


trace(ray)
    for each object in scene
        intersection = object.intersect(ray)
    determine closest intersection
    if(intersection)
        color = illuminate(ray, intersection)
    else
        color = backgroundColor
    return color
```

# How do we get our scene data?

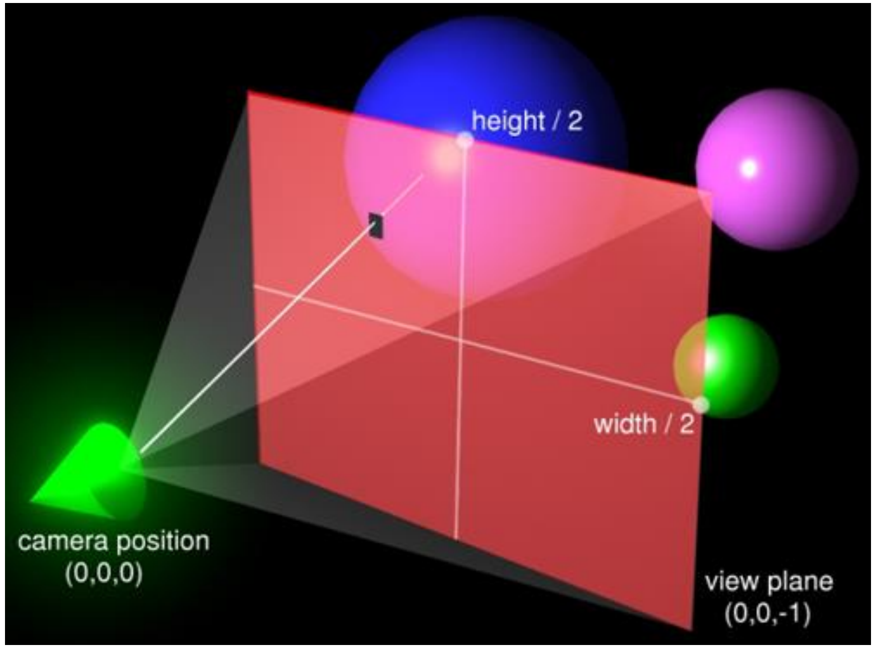
- Read in scene data from xml files provided on the course website
- Specification can be found here
- You can use any XML library you like for this
- Make sure that the xml file can be chosen at runtime

# Hint: Useful data structures

- Make your life easier by structuring your code properly
- Make use of inheritance and other OO concepts
- **Implement a vector class**
    - Make sure to properly differentiate between points and directions
    - Provides all necessary math (dot and cross product, subtraction, addition etc)
- Ray
    - Origin and direction, minimum and maximum distance
- Object/Surface
    - Has material and transformation data
    - Implements an intersection algorithm
- Intersection
    - All data you get from an intersection test: point of intersection, distance, normal, material and texture

# Constructing Camera Rays

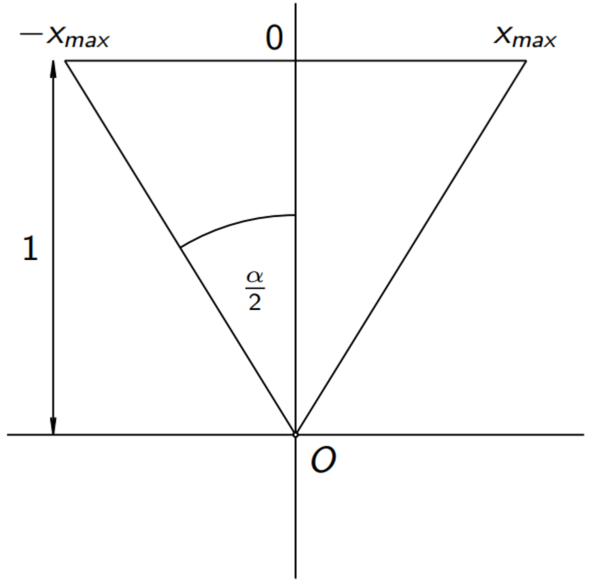
- Given a ray with origin $(0, 0, 0)^T$, we need to calculate the ray direction for a specific pixel
- Need to map original pixel coordinates $u$ and $v$
  to coordinates $x$ and $y$ on image plane at $z = -1$
- Ideally, we want to shoot the ray through the center of pixel




http://www.unknownroad.com/rtfm/graphics/rt_eyerays.html

# Constructing Camera Rays

- First, normalize coordinates (range [0 to 1]):
  - $x_n = \frac{u+0.5}{width}$
  - $y_n = \frac{v+0.5}{height}$
- Then, map to image plane (range [-1 to 1]):
  - $x_i = 2 * x_n - 1$
  - $y_i = 2 * y_n - 1$
- Include FOV and image dimensions
  - $fov_x$: specified in XML file
  - $fov_y = fov_x * \frac{height}{width}$
  - $x_i = (2 * x_n - 1) * \tan(fov_x)$
  - $y_i = (2 * y_n - 1) * \tan(fov_y)$




- The ray is now given by the origin $(0, 0, 0)^T$ and the direction $(x_i, y_i, -1)^T$ (don't forget to normalize)

- Notes
  - This will make your image upside down, keep that in mind when writing to the image file.
  - This assumes that $width \geq height$. If $height > width$, you need to invert aspect ratio and multiply $fov_x$ by it instead

# Surface Intersections

- Ray can be described as $r(t) = o + t * d$, with
  $o$ being the ray's origin
  $d$ being the ray's direction
  $t$ being a real number.
  - We are only interested in $t > 0$

- How to determine if a ray intersects an object?

# Ray-Sphere Intersections

- Given a radius $R$ and a center point $C$, a sphere can be defined as:
  $(P - C)^2 - R^2 = 0$
- Any point $P$ that satisfies this equation lies on the sphere
- To see if a ray intersects the sphere, we just plug in our ray:
  $(o + t * d - C)^2 - R^2 = 0$
- Rearranging the terms gives us:
  $(dt)^2 + 2d * (o - C)t + (o - C)^2 - R^2 = 0$
- Now solve for t

# Ray-Sphere Intersections

- We want to solve $(dt)^2 + 2d * (o - C)t + (o - C)^2 - R^2 = 0$ for t
- This is a quadratic equation with the form:

  $a * t^2 + b * t + c = 0$

  $a = d^2$

  $b = 2d * (o - C)$

  $c = (o - C)^2 - R^2$
- Applying the general solution formula gives us:

  $$t_{1,2} = \frac{-d*(o-C) \pm \sqrt{(d*(o-C))^2 - d^2*((o-C)^2 - R^2)}}{d^2}$$
- Instead of instantly solving this, look at discriminant first:
  - $disc > 0$ means we have two intersections → use closer t
  - $disc = 0$ means we have one intersection
  - $disc < 0$ means we have no intersections

https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection

# Ray-Sphere Intersections

- How to get normal at the intersection point?
- We have $t$ from the intersection test
- Pass $t$ to the ray to get the intersection Point $P$
- Compute vector from center of sphere to $P$
- Normalize vector

# Local Illumination

- Once you found the closest intersection point, compute color using the intersection and material data you got
- Phong shading: like Lab 1b
  - Ambient light: Only one global ambient light
  - Each object can have different coefficients for ambient, diffuse and specular light
- How to handle multiple lightsources?
  - Compute light for each source separately and sum them up

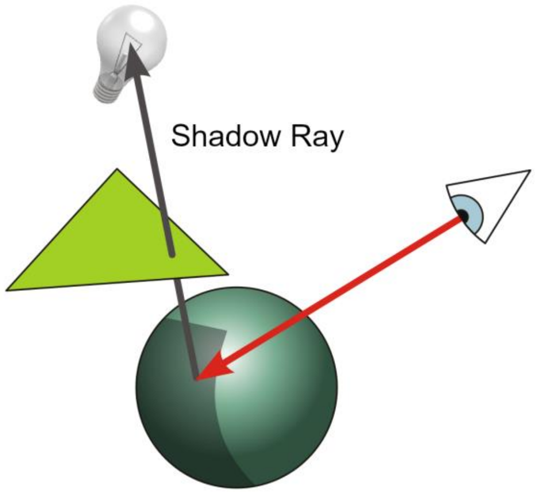
- Hints:
  - Don’t multiply the specular light with the object’s color
  - For specular light, the vector to the viewer is simply the flipped ray direction

# Handling multiple light sources

```
trace(ray)
    for each object in scene
        intersection = object.intersect(ray)
    determine closest intersection
    if(intersection)
        for each light source
            color += illuminate(ray, intersection, light source)
    else
        color = backgroundColor
    return color
```

# Shadows

- Before computing local illumination for a light source, cast shadow ray towards it
- If the shadow ray hits something, the light doesn't reach the surface
  - Ignore this light source

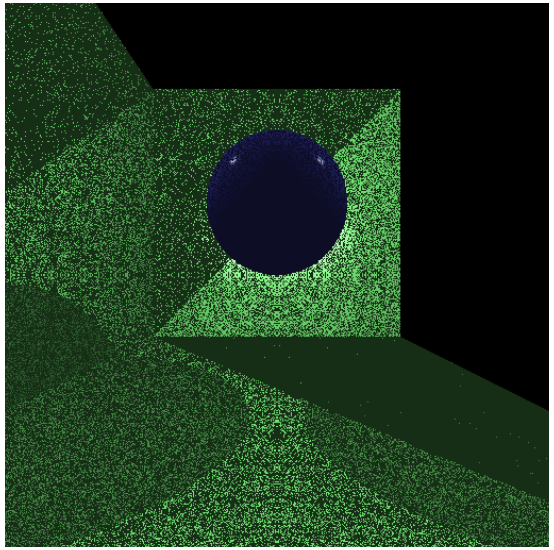


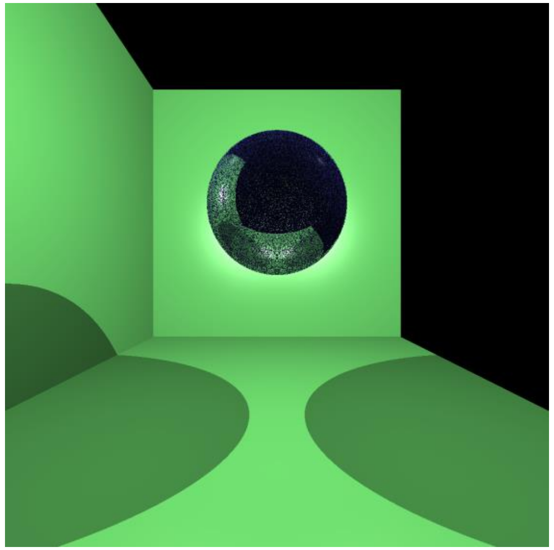
# Casting Shadow Rays

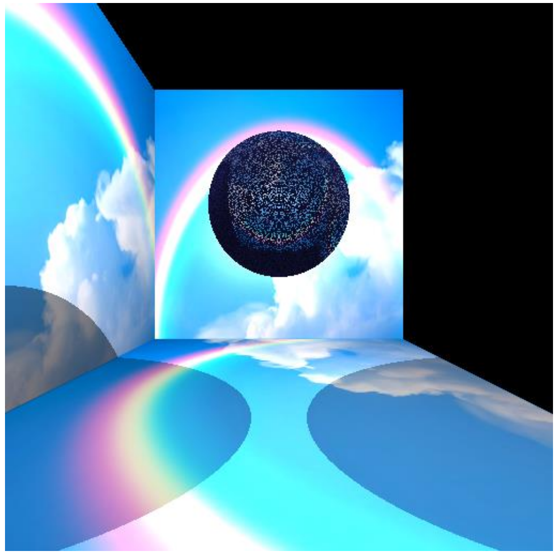
```
trace(ray)
    for each object in scene
        intersection = object.intersect(ray)
    determine closest intersection
    if(intersection)
        for each light_source
            shadow_intersection = cast_shadowray(intersection, light_source)
            if(no shadow_intersection)
                color += illuminate(ray, intersection, light_source)
    else
        color = backgroundColor
    return color
```

# Shadows Acne/Bias

- Due to imprecisions of floating point numbers, a can ray sometimes intersect at a point inside an object.
- Shadow (and also reflection and refraction) rays cast from these points will immediately intersect the object again







# Shadows Acne/Bias

- How to solve this?
- Either:
  - Offset ray origin by small $\varepsilon$ along the ray's/normal's direction
  - Reject intersections that have $t$ too close to 0, i.e. $t < \varepsilon$
  - Might need to adjust $\varepsilon$ depending on your exact circumstances

# Hints & Common Mistakes

- **Normalize your vectors**
- Make sure that your vectors point in the right direction
- For debugging, consider encoding your normal/ray directions to colors
- Whenever you map values from one range to another (e.g. colors from [0; 1] to [0; 255] or vice versa), make sure to watch out for integer truncation
- Make sure to limit the travel distance of shadow rays. You don’t want them to hit things behind the light source.

Questions?