Ray Tracing – Part 2

CSC418/2504 – Introduction to Computer Graphics

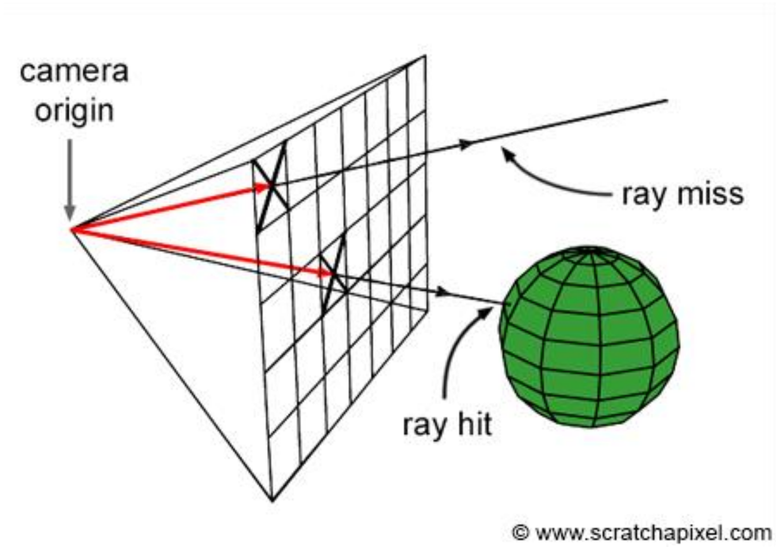
TA: Muhammed Anwar
Email: manwar@cs.toronto.edu

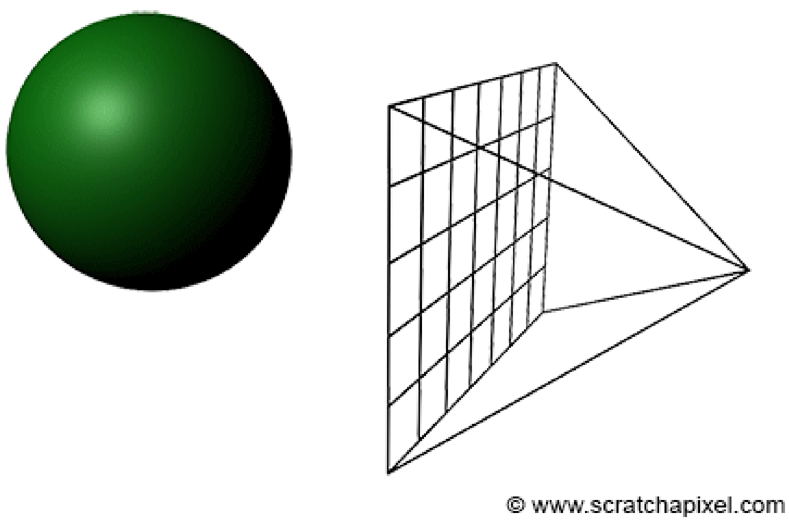
Overview

- Raytracing Recap
- FAQ/Conventions for A3/A4
- Advanced Ray Tracing Features
  - Soft Shadows
  - Anti-Aliasing
  - Glossy Reflections
  - Depth of Field
  - Environment Mapping
- Acceleration Techniques
  - Multi-Threading
  - Acceleration Structures (BVH – Bounding Volume Hierarchies)
- Questions

Ray Tracing - Recap

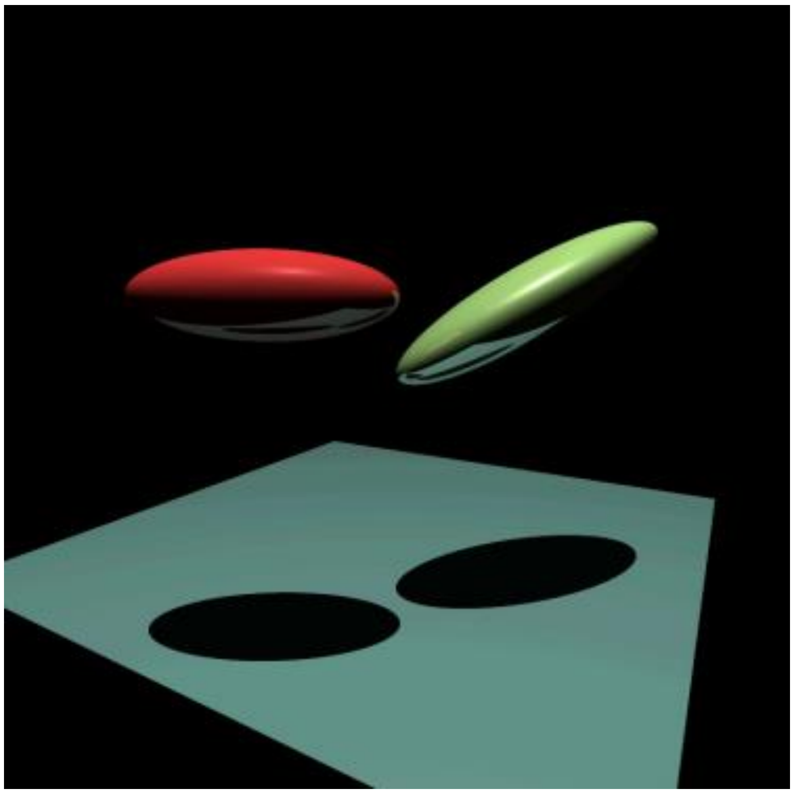
```
for each pixel in image:
- compute and construct viewing ray
-  find the 1st hit object by the ray and compute an intersection
- set pixel colour from the intersection
```





Common Questions for A3

- What color should a point in shadow be?

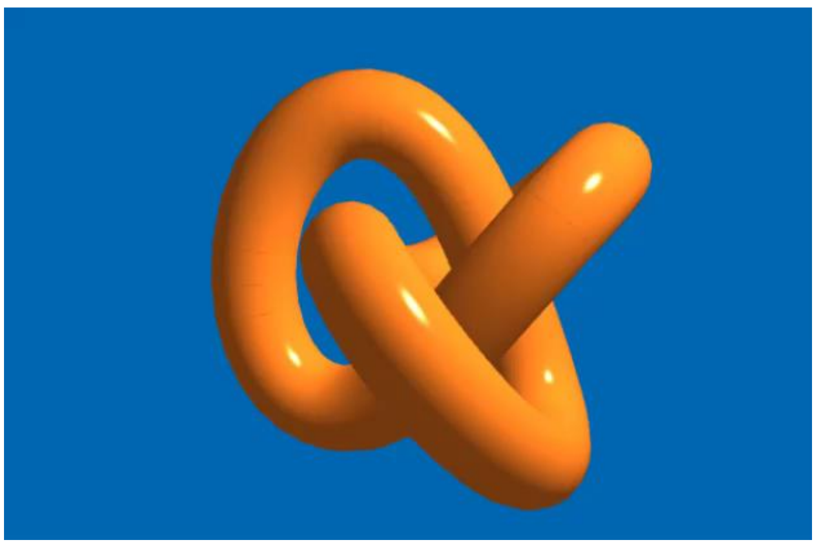


- The renders shown in the assignment handout discard all the Phong model contributions from the light (i.e a point in shadow is black).
- It is OKAY, to use the AMBIENT term as the color.
- In practice the difference is usually very subtle.

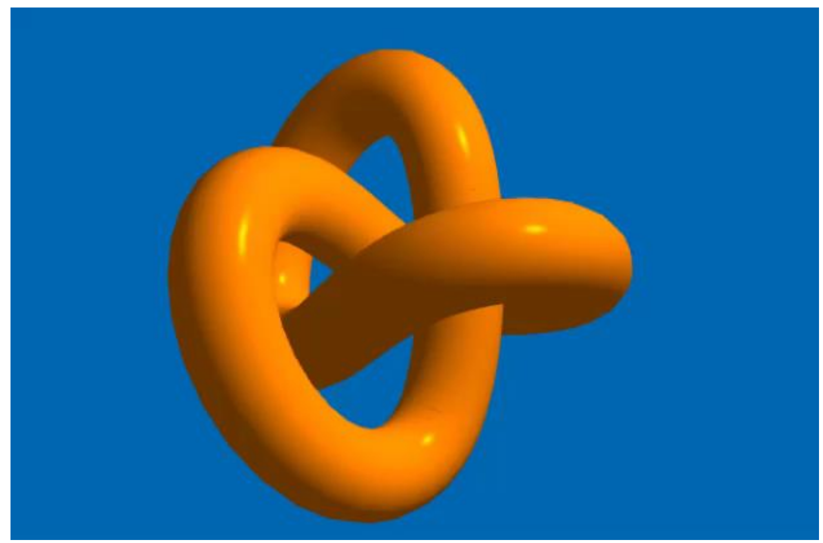
Common Questions for A3

- How do I incorporate object color in the shading model?
  - The most common way is to incorporate the object color in the ambient and diffuse term, but not the specular term. Again subtle difference

    finalColor = objColor * (diffuse + ambient) + specular



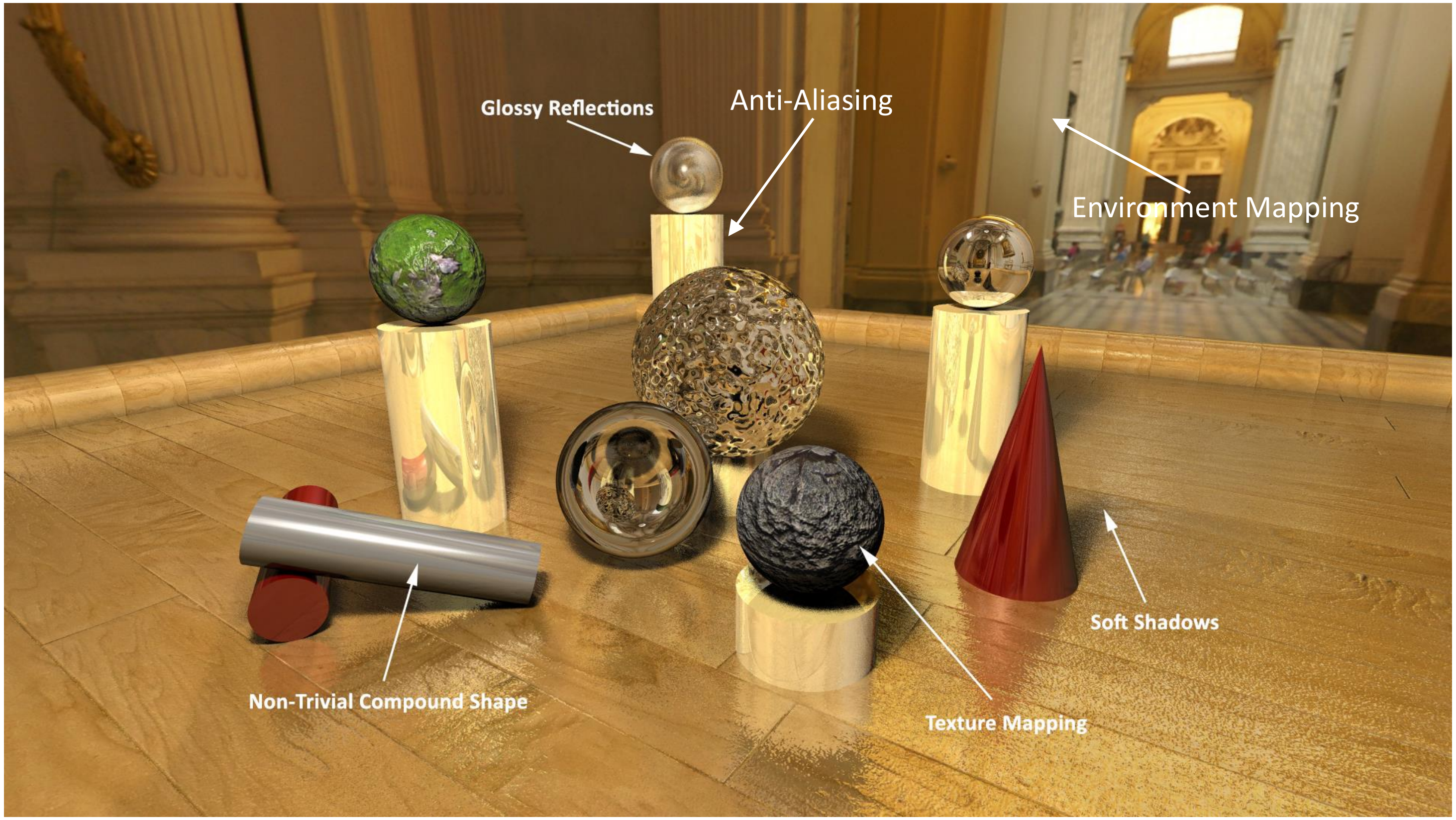
Object Color multiplied by Ambient and Diffuse only



Object Color multiplied by all Phong Components

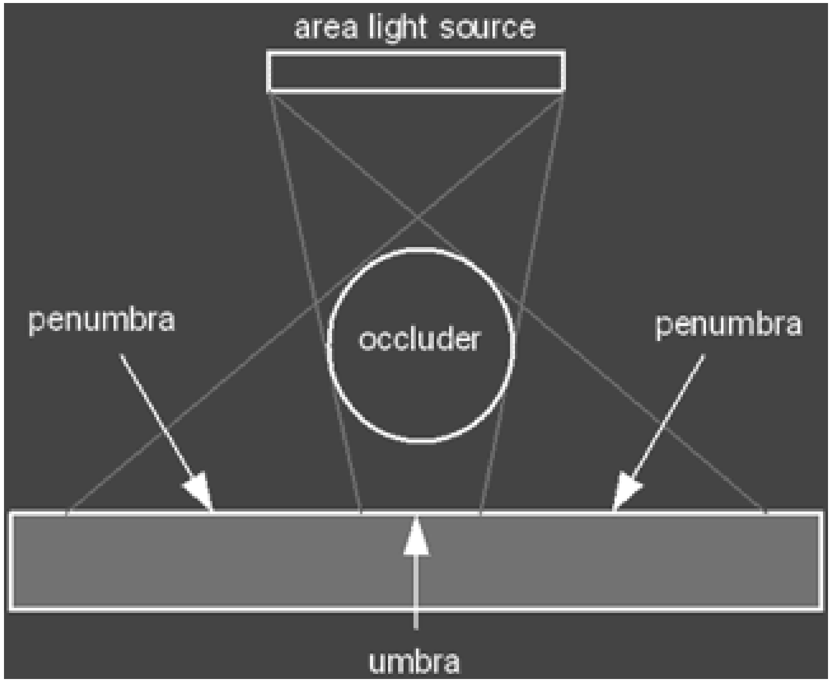
Advanced Ray Tracing

The basic form of raytracing is still limited in the type of effects we can achieve. Need distributed ray-tracing.



Soft Shadows

- Real life lights are not point lights, they have some area/volume

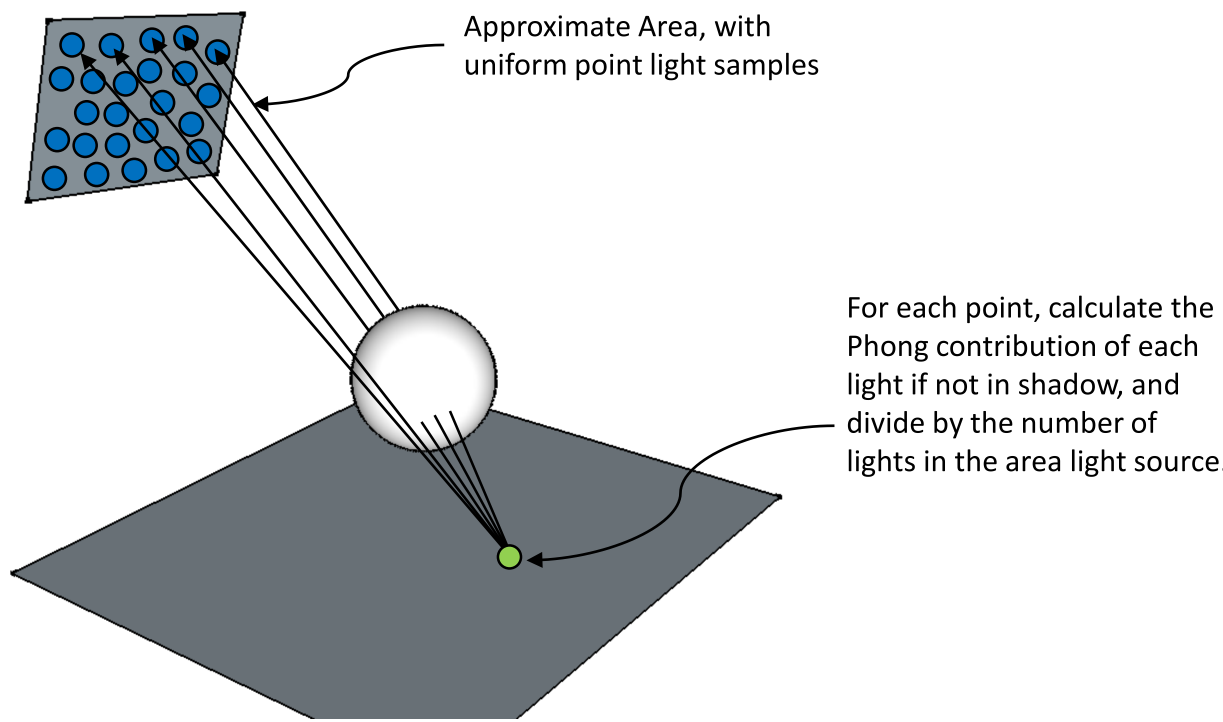


Penumbra: The lighter part of the shadow, slowly gets brighter towards the edges.
Umbra: The darkest part of the shadow, no light reaches.

- **2 ways** to achieve this effect in a ray tracer. One extends the basic ray tracing we have so far (Assignment). The other leverages the “randomness” of distributed ray tracing.

Soft Shadows – Method 1

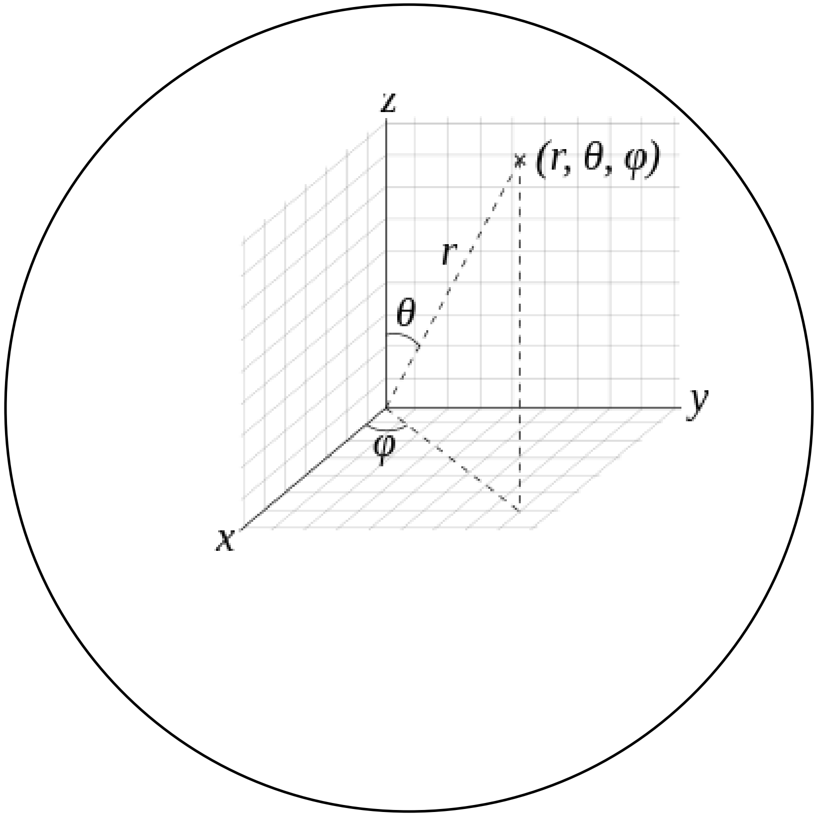
- Approximate an area light source with **multiple** light sources (Assignment)



- Area light can be uniformly randomly sampled or arranged in a grid. Quality improves as more lights are added.

Soft Shadows – Method 2

- Use the concepts of distributed ray-tracing and its randomly distributed oversampling.
- Approximate a point light source as a sphere, and for each intersection point, randomly sample a point on that sphere to do lighting calculations. Only do one check per “area” light.

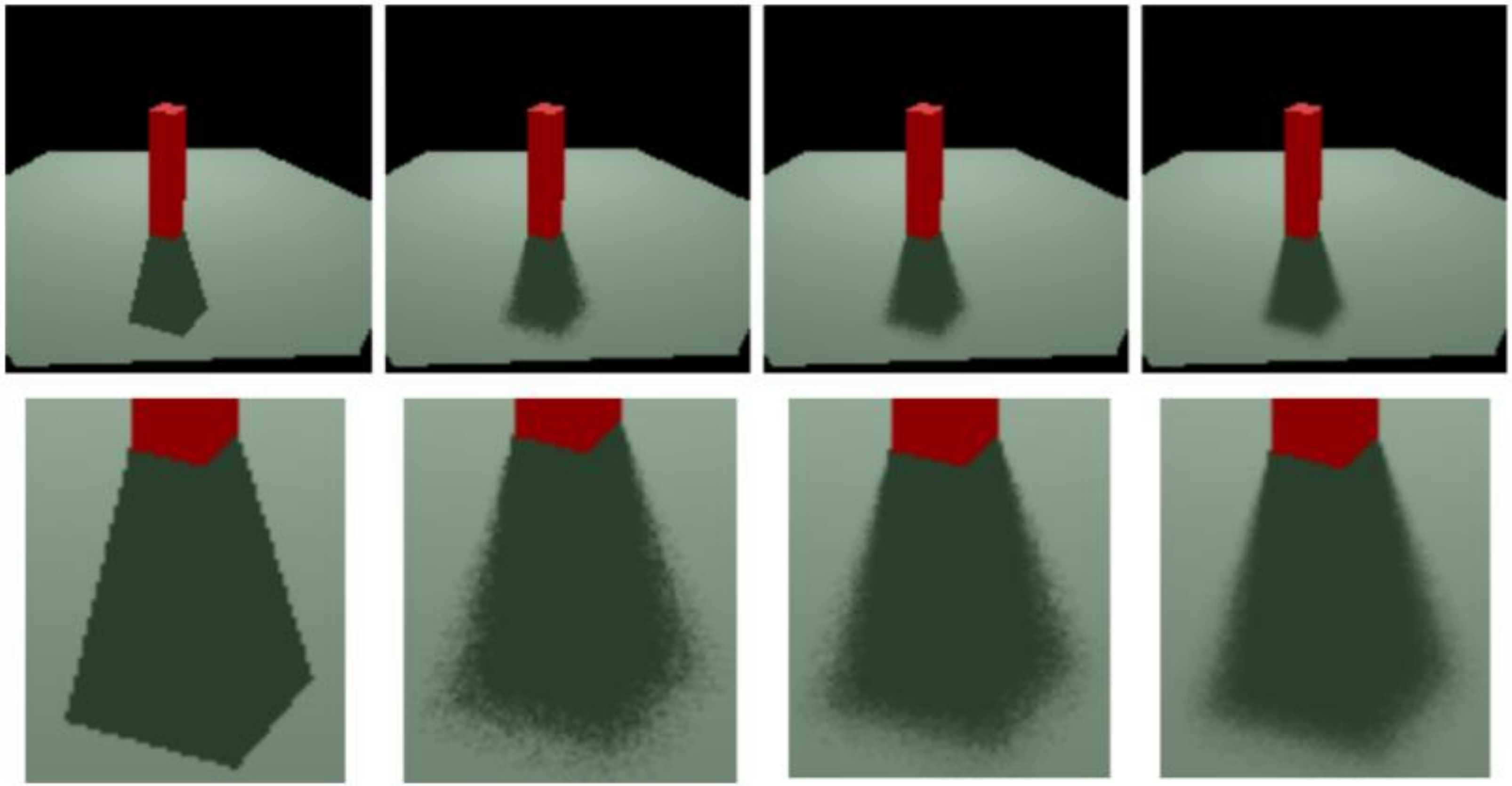



```
float r = radius* rand();
float theta = 2 * PI * rand();
float phi = 2 * PI * rand();

randomLightPos = curlight->get_position()
 + Vector3(r*cos(theta)*sin(phi),
           r*sin(theta)*sin(phi),
           r*cos(phi));
```

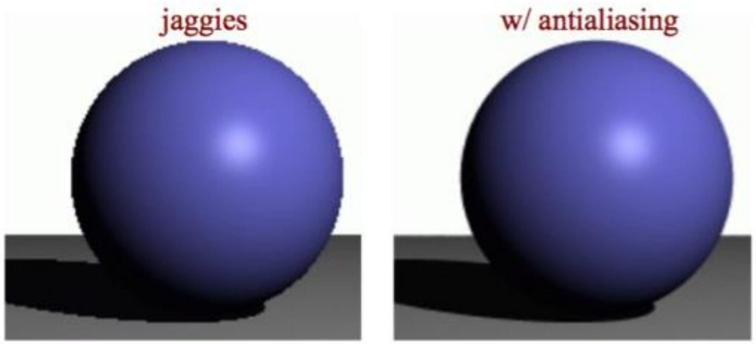
- Use this randomLightPos as the position of the light for that point of intersection. Randomly sample the light position for every intersection. With enough rays will eventually approximate the soft shadow.

Soft Shadows

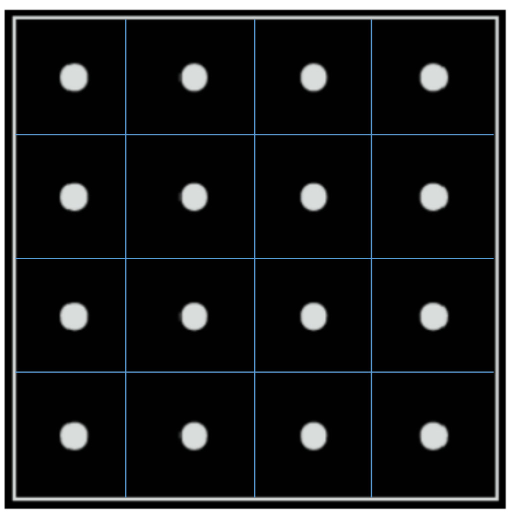


- Using method 1 or method 2, with enough lights or sampled rays, the soft shadow will be approximated.

Anti-Aliasing



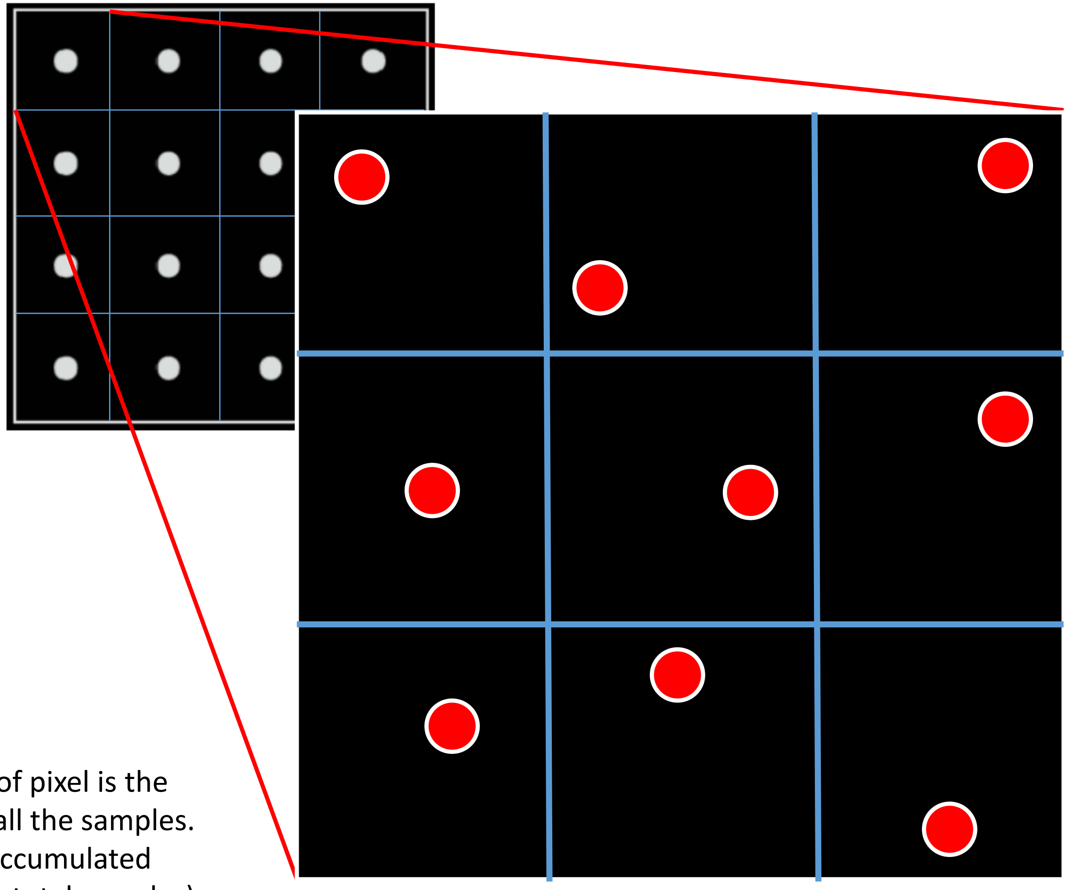
- Spatial aliasing happens when we try to sample a scene at a lower frequency than it truly is.
- We can remedy using super sampling.

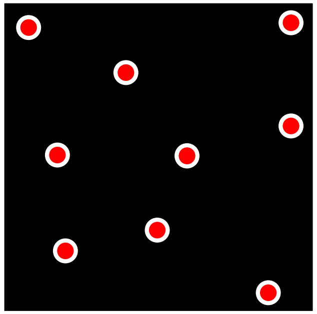


- So far we have been sending a single ray uniformly through the center of our “pixel”.
- Super sampling – Send **multiple** rays **through different sampled** positions in each pixel.

Super Sampling – Jitter Method

- Many ways to super sample a particular pixel. One method is called the Jitter Algorithm. Split pixel into sub pixels and randomly sample from each subpixel.

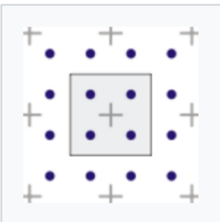




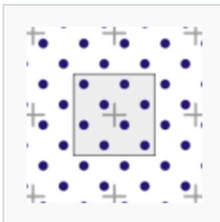
- **Benefits:**
  - Get an even distribution of samples across every pixel.
  - Easy to implement

Final Color of pixel is the average of all the samples. (i.e divide accumulated color by the total samples)

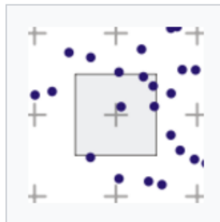
Super Sampling – Other Methods



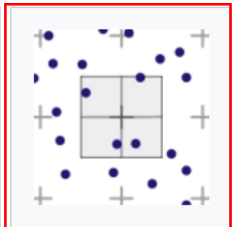
Grid algorithm in uniform distribution



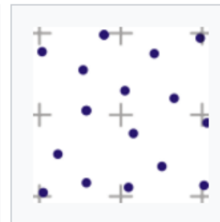
Rotated grid algorithm



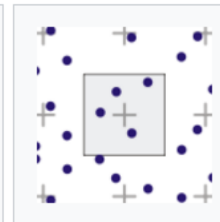
Random algorithm



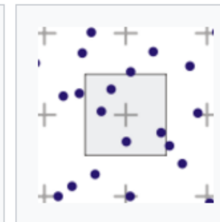
Jitter algorithm



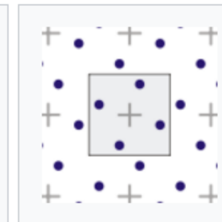
Poisson disc algorithm



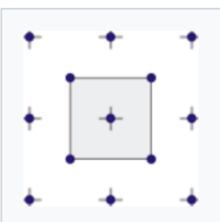
Quasi-Monte Carlo method algorithm



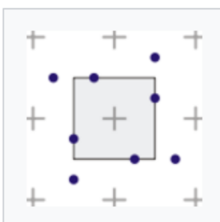
N-Rooks



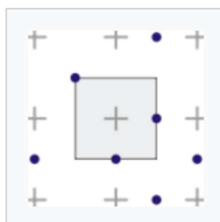
RGSS



High-resolution antialiasing (HRAA), Quincunx



Flipquad



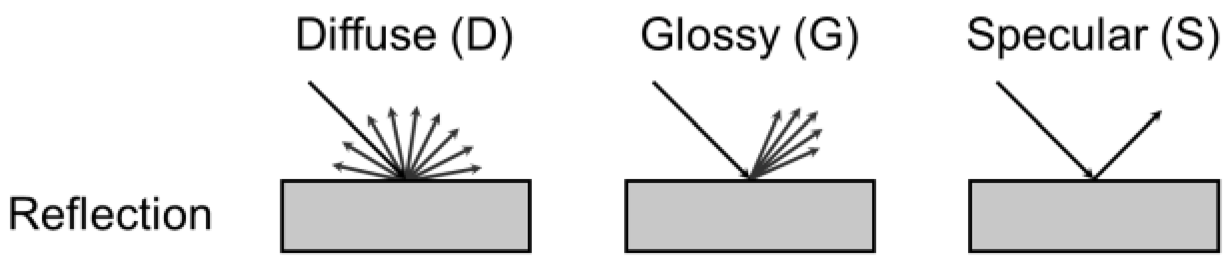
Fliptri

Source:
https://en.wikipedia.org/wiki/Supersampling

- Many different super sampling techniques exists, you can experiment with them.
- Some allow you to get better anti-aliasing effects with fewer samples. However all of them look the same with enough samples.
- Choice is more important in real-time pipeline

Glossy Reflections

- Not all surfaces have perfect mirror like reflections

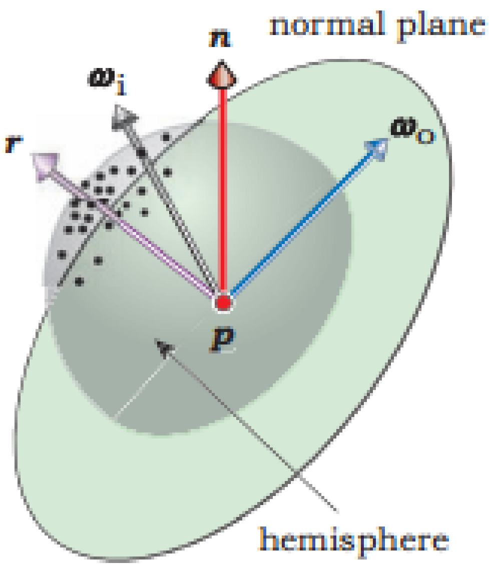


- Glossy surfaces scatter light in the reflected direction but not perfectly and with some jitter. The more glossy it is, the more “jitter” it has.

Glossy Reflections - Jitter

Want to send out a ray jittered in the direction of $\hat{r}$.

1. Construct an orthonormal basis at intersection point.
2. Sample from a hemisphere, constrained depending on “glossiness/roughness”
3. Move sample to the world coordinates using the orthonormal basis.




```
Vector3D R = 2 * N.dot(D)*N - D; //Reflection Vector
R.normalize();

//Orthonormal basis at intersection point
Vector3D u = R.cross(N);
u.normalize();
Vector3D v = R.cross(u);
v.normalize();

// Choose uniformly sampled random direction to send the ray in
double theta = 2 * M_PI * (erand() *roughness);
double phi = 2 * M_PI * (erand() * roughness);
double x = sin(theta) * cos(phi);
double y = sin(theta) * sin(phi);
double z = cos(theta);

// Convert sample to world coordinates using the orthonormal basis
R = x * u + y * v + z * R;
R.normalize();
```

More Details:
http://www.raytracegroundup.com/downloads/Chapter25.pdf

Depth-of-Field

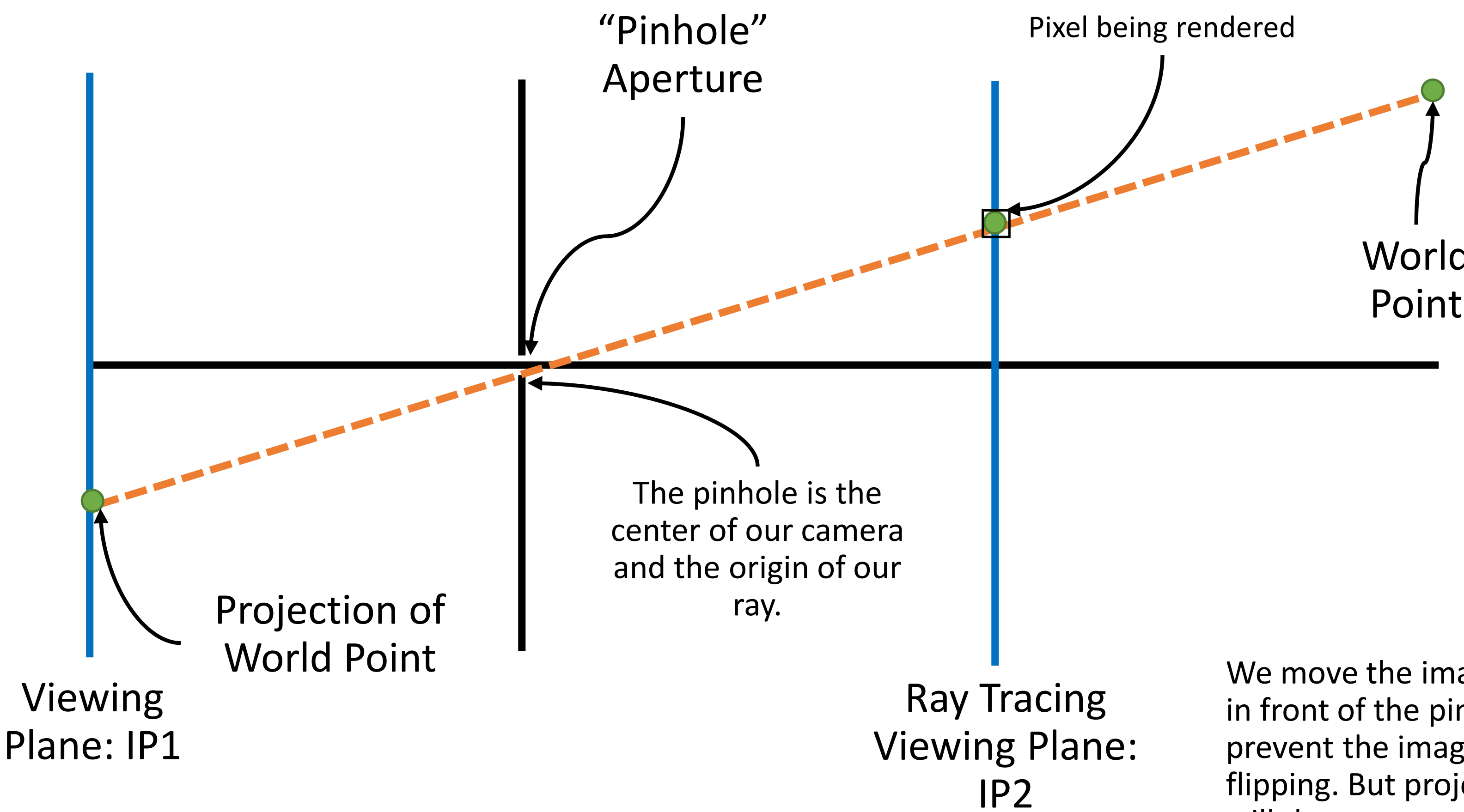
- A camera effect, that essentially determines the range of focus.
- Real world cameras are not pinholes and instead have a size (aperture) and a lens that defines a focal length.

Small Aperture – Large DOF

Large Aperture – Small DOF

Depth-of-Field : Pinhole Camera

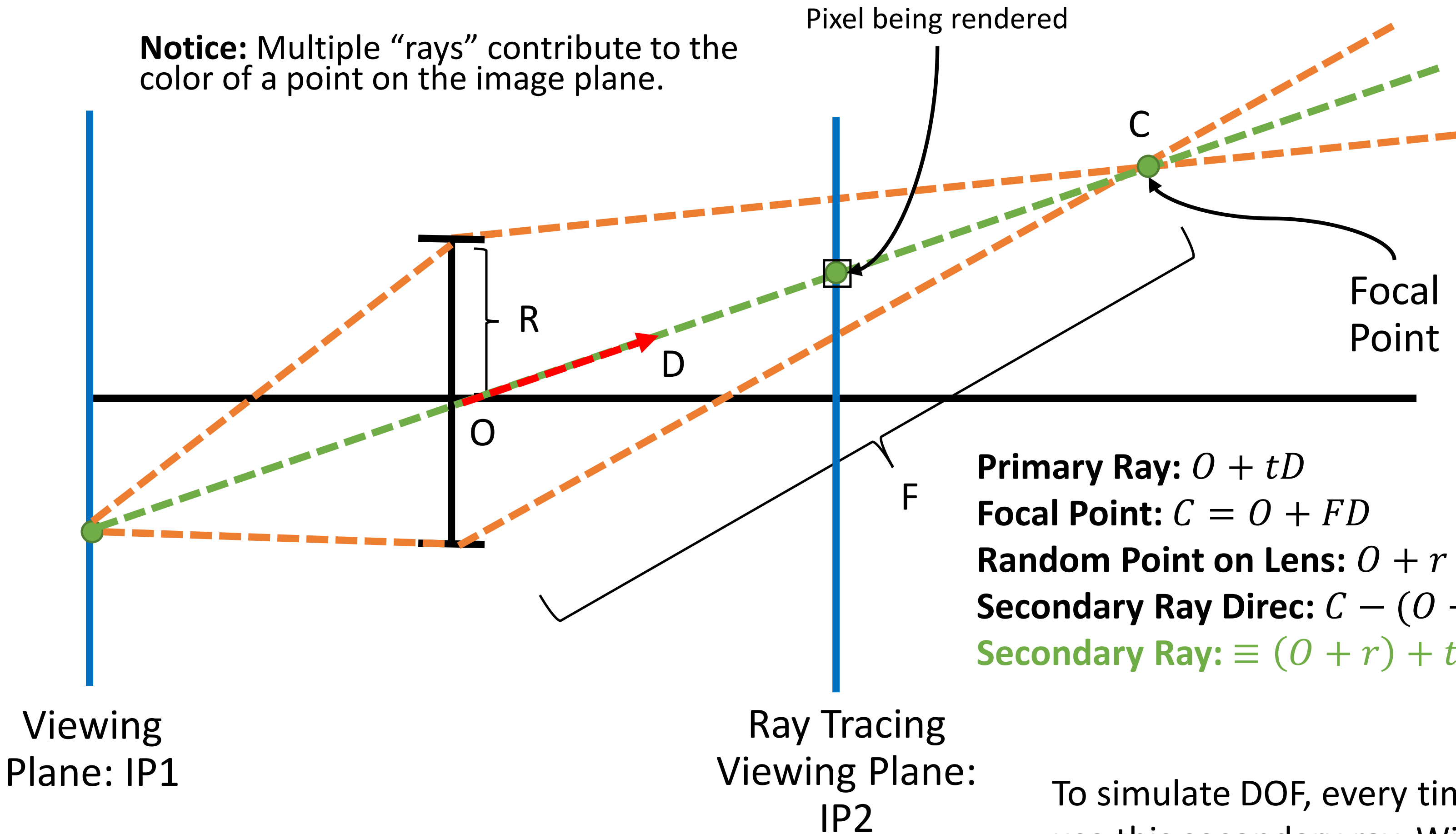
- Lets look at the pinhole model we are using now:



We move the image plane in front of the pinhole to prevent the image from flipping. But projection is still the same.

Depth-of-Field : Thin Lens Model

- Now lets place a thin lens as our aperture instead of a pinhole



Primary Ray: $O + tD$

Focal Point: $C = O + FD$

Random Point on Lens: $O + r$

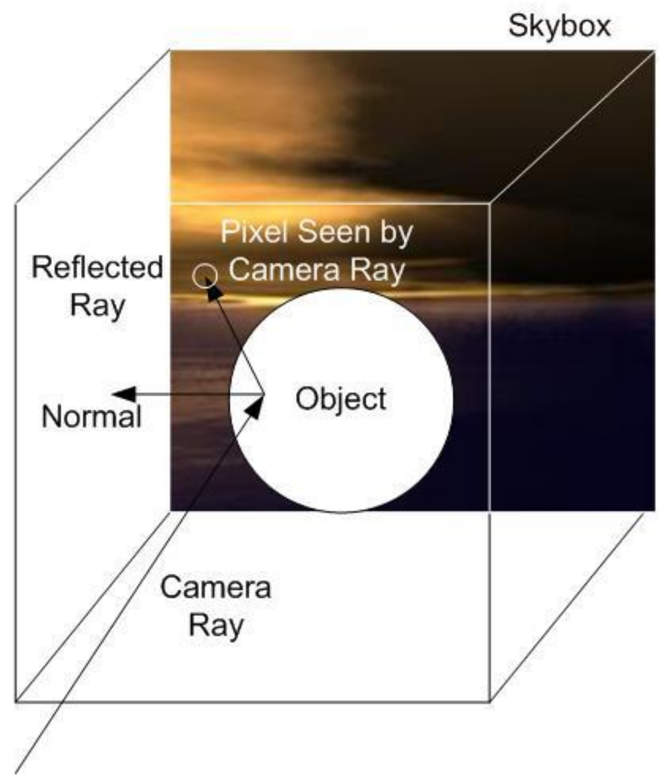
Secondary Ray Direc: $C - (O + r)$

Secondary Ray: $\equiv (O + r) + t(C - (O + R))$

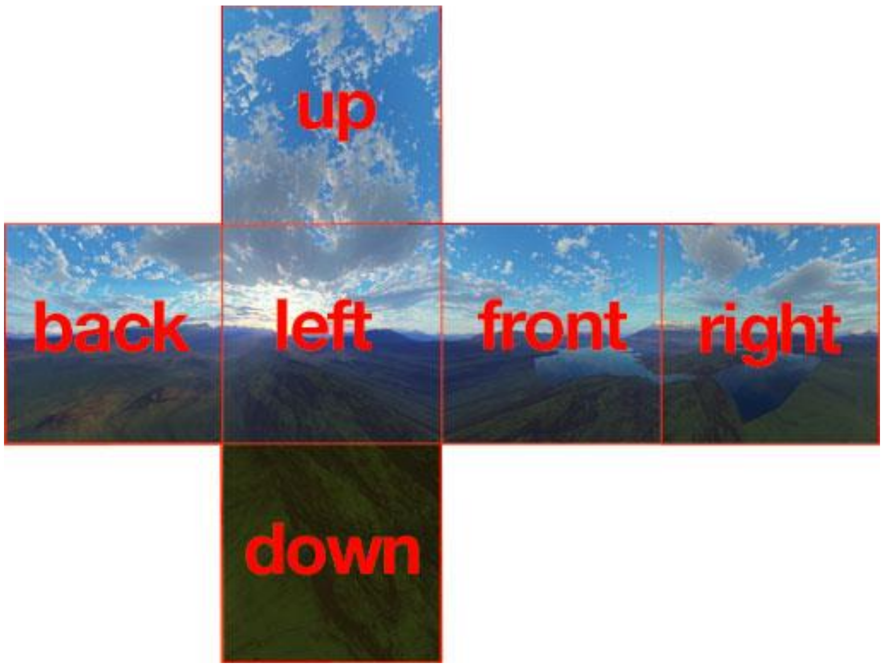
To simulate DOF, every time shade a pixel use this secondary ray. With enough samples (per pixel), we will approximate DOF.

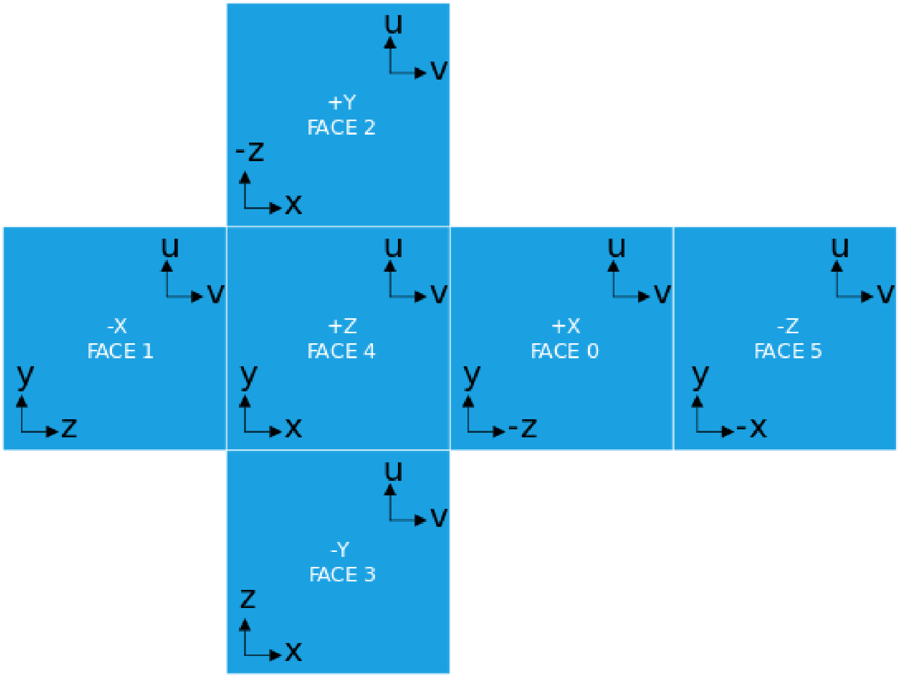
Environment Mapping

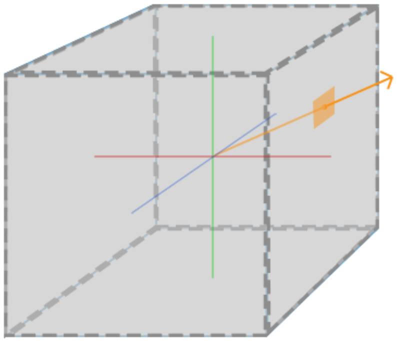
- Creates an environment surrounding your scene, that objects can reflect.
- Whenever your ray doesn’t hit anything, sample from the cube map.



Environment Mapping





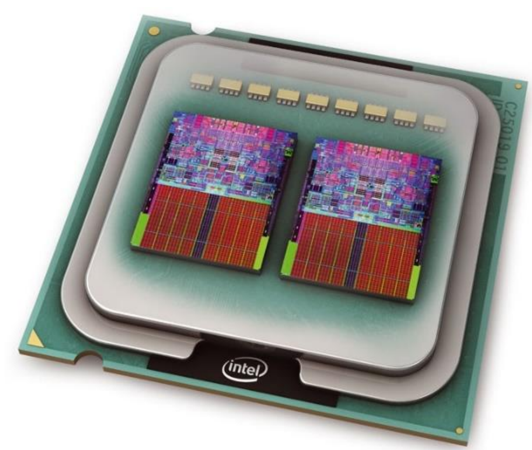


Use the direction of the ray, to sample from the correct face of the cube. The UV coordinates of the sample are carefully chosen values of the (x,y,z) direction vector.

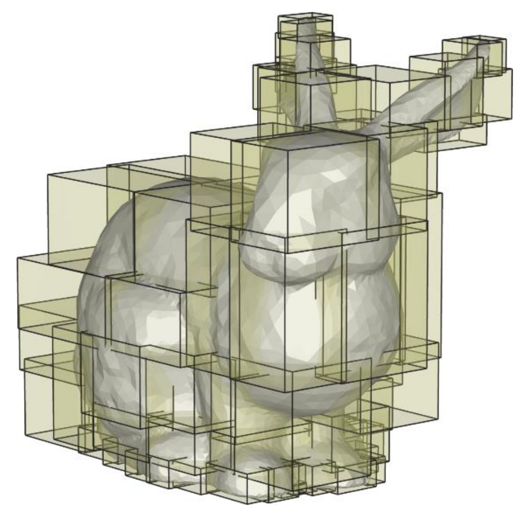
Pseudocode found here: https://en.wikipedia.org/wiki/Cube_mapping

This is cool, but why is it so slow!

- Raytracing is inherently very slow, especially when using distributed ray tracing concepts.
- Fortunately there are ways to speed things up. Some are easy, and some are more involved.
- Lets look at 2 methods:



Multithreading



Acceleration Structures

- **Note:** I am giving this away for free, as it will allow you to iterate faster and produce EVEN BETTER images.

Multithreading

- Ray tracing is truly embarrassingly parallel. Each pixel color can be calculated independently of other pixels.
- Therefore use simple multithreading techniques.
- Use OpenMP's Parallel For-loop

In your source code:

```
#pragma omp parallel for
    for (int i = 0; i < _scrHeight; i++) {
        for (int j = 0; j < _scrWidth; j++) {
                …
```

In your MAKE file

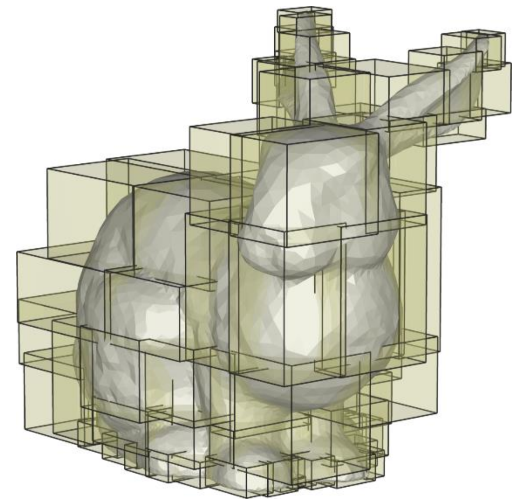
```
CXXFLAGS = -g - O2 - fopenmp - std = gnu++11
```

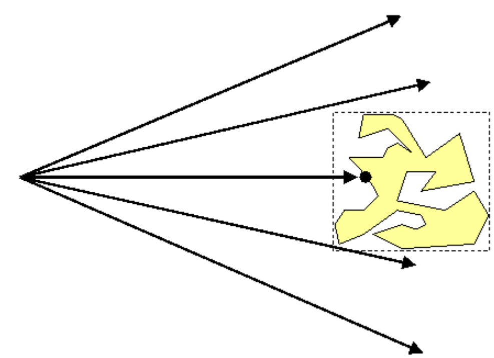
2 Lines of code, and your ray tracer is multi-threaded!

[More Information: http://bisqwit.iki.fi/story/howto/openmp/#ExampleInitializingATableInParallelMultipleThreads](http://bisqwit.iki.fi/story/howto/openmp/#ExampleInitializingATableInParallelMultipleThreads)

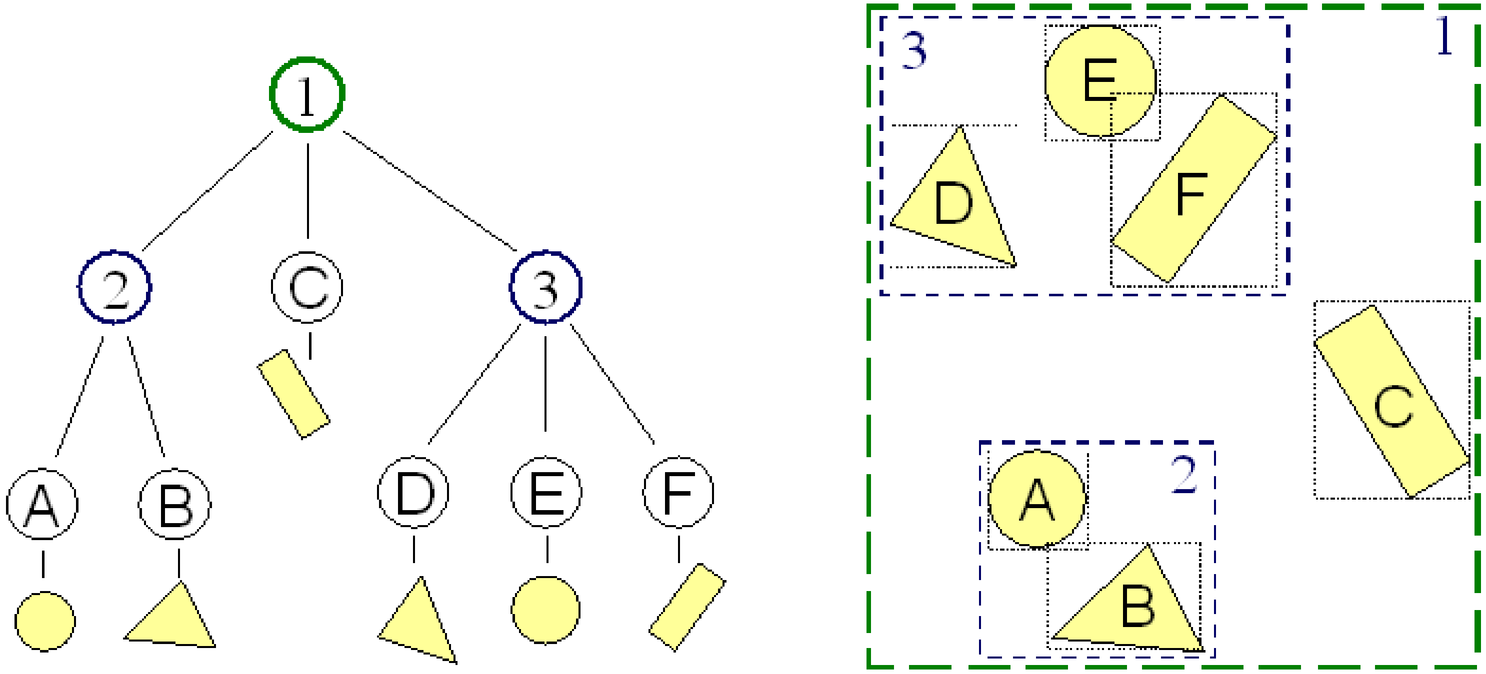
Acceleration Structures

- **Idea:** We want to limit the number of intersection checks that we need to perform.
- Ideally we want to know quickly if a ray will even intersect with a region of geometry, before continuing further.
- One such Acceleration structure is a Bounding Volume

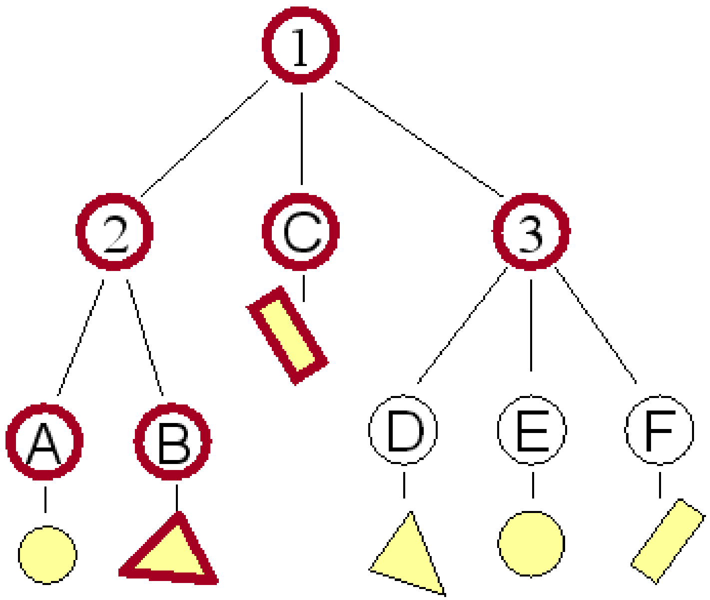


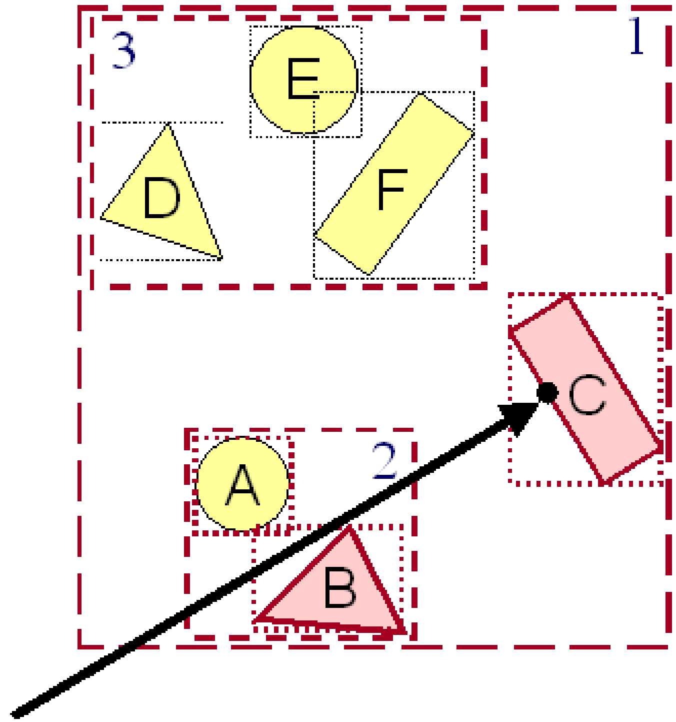


Bounding Volume Hierarchy



Bounding Volume Hierarchy





Implementation Details:

http://fileadmin.cs.lth.se/cs/Education/EDAN30/lectures/S2-bvh.pdf

https://github.com/brandonpelfrey/Fast-BVH - Hard to use, but gives ideas on how to implement such a data structure

https://www.scratchapixel.com/lessons/advanced-rendering/introduction-acceleration-structure/introduction