MSc Computer Games and Entertainment

Maths & Graphics Unit 2011/12

Lecturer: Gareth Edwards

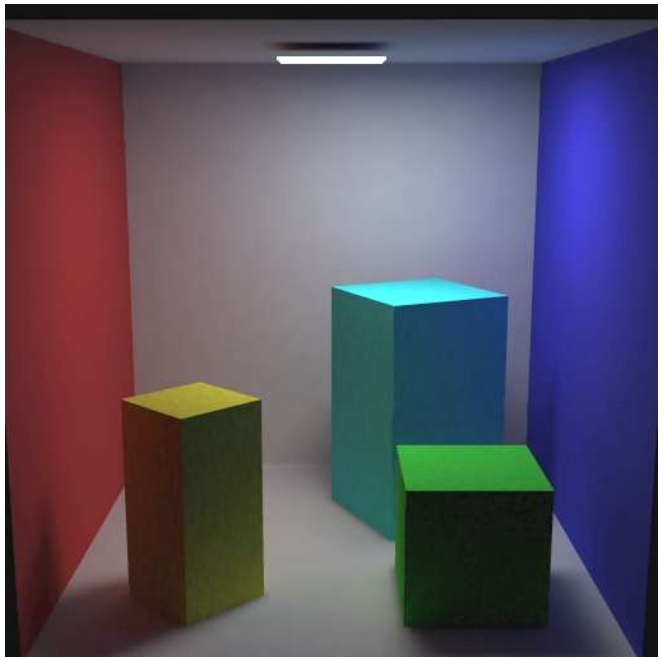
Ray Tracing
Introduction & pseudo-code

What is Ray Tracing?

- Ray tracing is a technique for rendering three-dimensional graphics with very complex light interactions.
- You can create pictures full of mirrors, transparent surfaces, and shadows, with stunning results.
- It is a very simple method to both understand and implement.

What is Radiosity?

- It is based on ray tracing.
- It simulates all reflections from objects in the scene.
- It is a more accurate but also more process-intensive.
- It is an application of the finite element method to solving the rendering equation for scenes with purely diffuse surfaces.

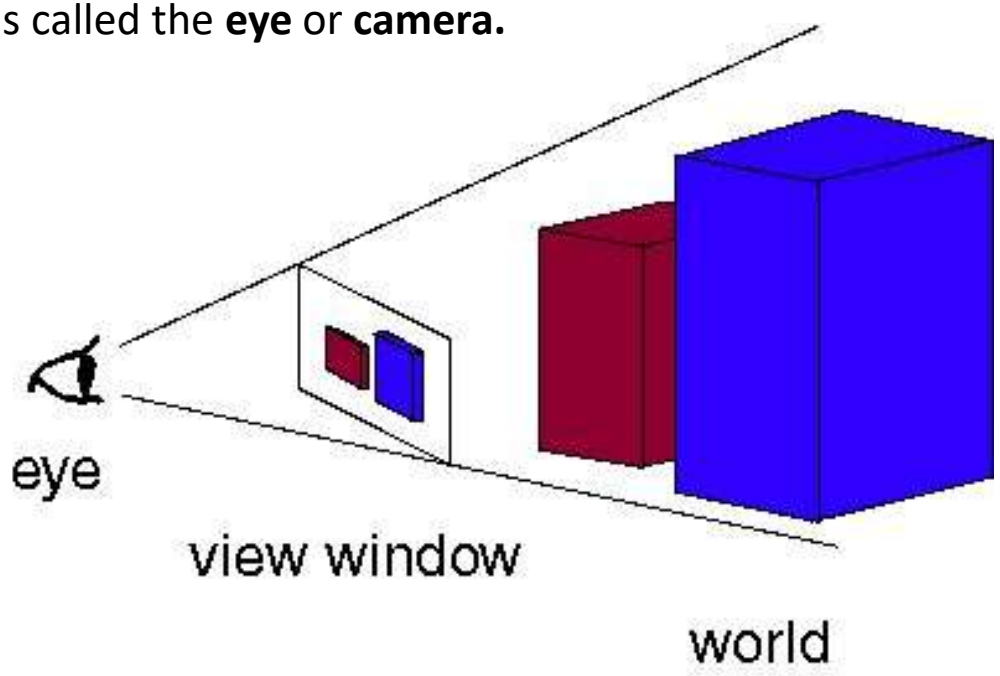


Ray tracing is recursive

- Ray tracing is based on the idea that you can model reflection and refraction by recursively following the path that light takes as it bounces through an environment.

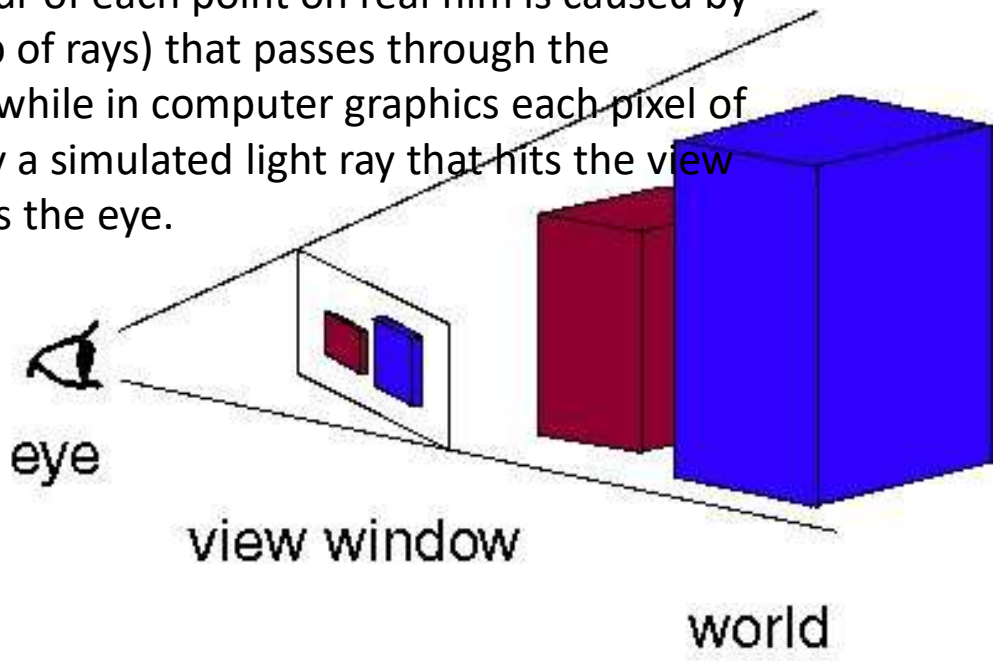
Looking at the world

- When creating any sort of computer graphics program, you must have a list of objects that you want your software to render.
- These objects are part of a **scene** or **world**, so when we talk about "looking at the world", we are referring to the ray tracer drawing the objects from a given viewpoint.
- In graphics, this viewpoint is called the **eye** or **camera.**



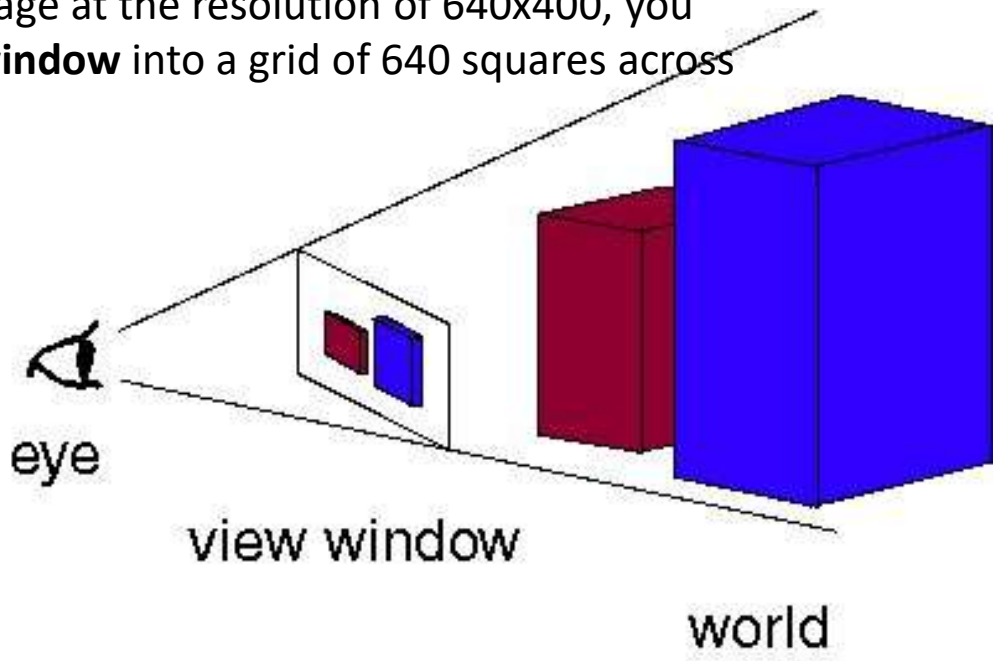
The view window

- Following this camera analogy, just like a camera needs film onto which the scene is projected and recorded, in graphics we have a **view window** on which we draw the scene.
- The difference is that while in cameras the film is placed *behind* the aperture or focal point, in graphics the view window is in *front* of the focal point. So the colour of each point on real film is caused by a light ray (actually, a group of rays) that passes through the aperture and hits the film, while in computer graphics each pixel of the final image is caused by a simulated light ray that hits the view window on its path towards the eye.



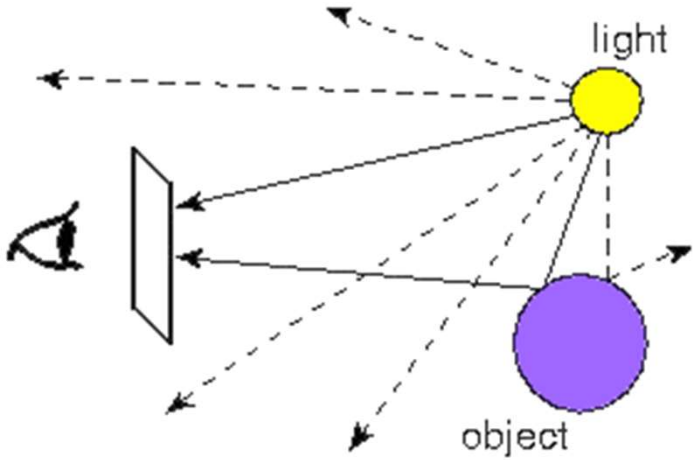
The goal of ray tracing

- The goal of ray tracing is find the colour of each picture element, pixel, on the **view window**.
- To do this, the program subdivides the **view window** into small squares, where each square corresponds to one pixel in the final image.
- If you want to create an image at the resolution of 640x400, you would break up the **view window** into a grid of 640 squares across and 400 square down.



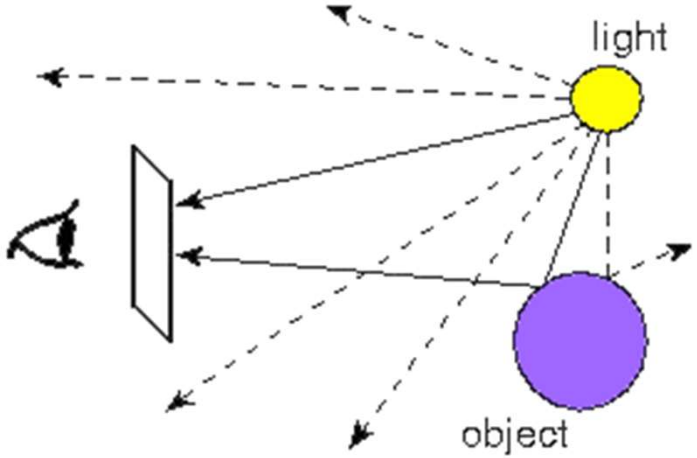
Why is it called ray tracing?

- Ray tracing is so named because it tries to simulate the path that light rays take as they bounce around within the world - they are *traced* through the scene.
- The objective is to determine the colour of each light ray that strikes the view window before reaching the eye.



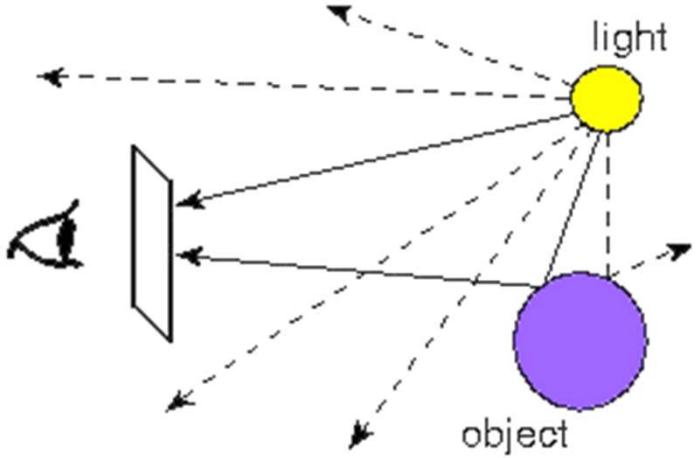
Why is it called ray tracing?

- A light ray can best be thought of as a single photon (although this is not strictly accurate because light also has wave properties).
- The name “ray tracing'' is a bit misleading because the natural assumption would be that rays are traced starting at their point of origin, the light source, and towards their destination, the eye.
- This would be an accurate way to do it, but unfortunately it tends to be very difficult due to the sheer numbers involved.



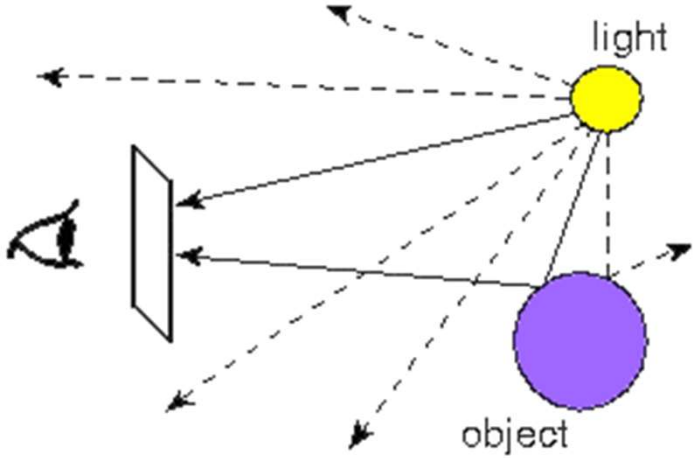
How would ray tracing be done properly?

- Consider tracing one ray in this manner through a scene with one light and one object, such as a table. We begin at the light bulb, but first need to decide how many rays to shoot out from the bulb.
- Then for each ray we have to decide in what direction it is going.
- There is really an infinite of directions in which it can travel - how do we know which to choose?



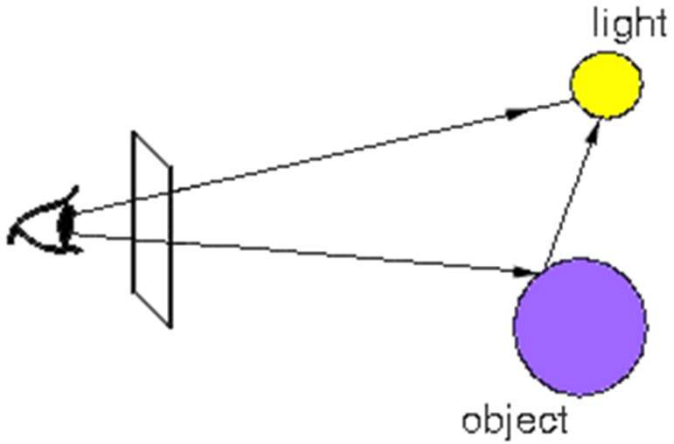
How would ray tracing be done properly?

- Let's say we've answered these questions and are now tracing a number of photons. Some will reach the eye directly, others will bounce around some and then reach the eye and many, and many more will probably never hit the eye at all.
- However for all the rays that don’t reach the eye the effort tracing them would have been wasted.



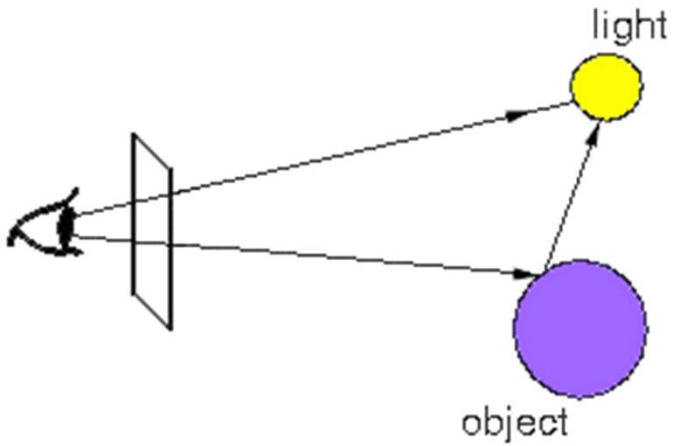
The backwards method

- In order to save ourselves this wasted effort, we trace only those rays that are *guaranteed* to hit the view window and reach the eye. It seems at first that it is impossible to know beforehand which rays reach the eye. After all, any given ray can bounce around the room many times before reaching the eye. However, if we look at the problem *backwards*, we see that it has a very simple solution.
- Instead of tracing the rays starting at the light source, we trace them backwards, starting at the eye.



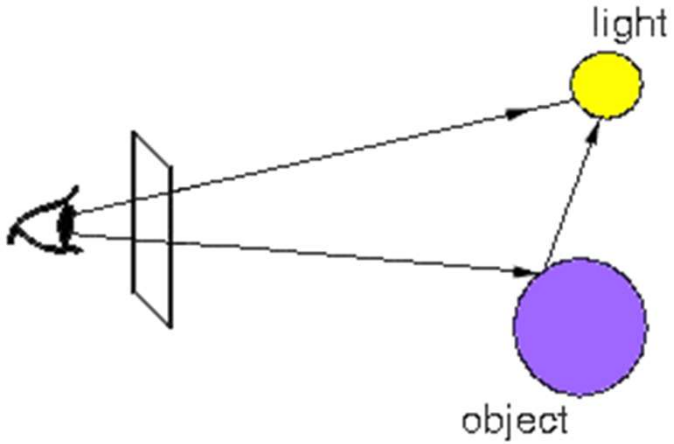
How ray tracing is actually done

- Consider any point on the view window whose colour we're trying to determine. Its colour is given by the colour of the light ray that passes through that point on the view window and reaches the eye.
- We can just as well follow the ray backwards by starting at the eye and passing through the point on its way out into the scene.



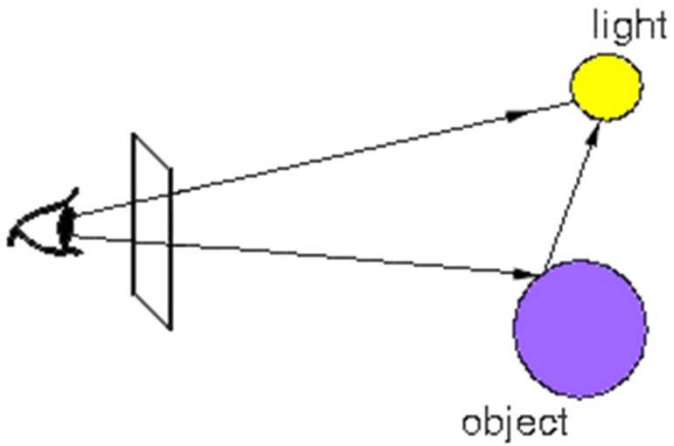
How ray tracing is actually done

- The two rays will be identical, except for their direction: if the original ray came directly *from* the light source, then the backwards ray will go directly *to* the light source; if the original bounced off a table first, the backwards ray will also bounce off the table.
- So the backwards method does the same thing as the original method, except it doesn't waste any effort on rays that never reach the eye.



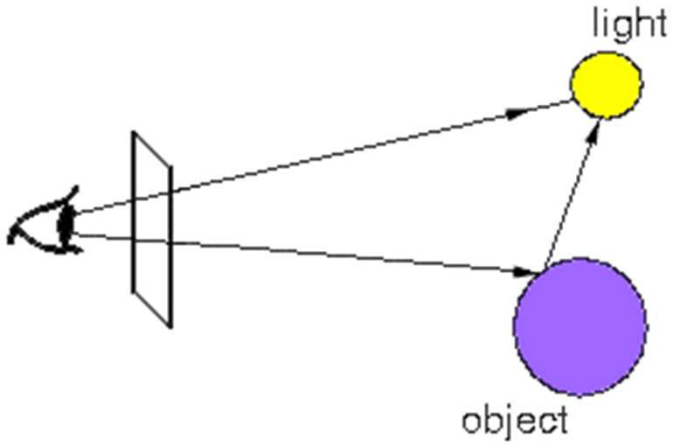
How ray tracing is actually done

- For each pixel on the view window, we define a ray that extends from the eye to that point. We follow this ray out into the scene and as it bounces off of different objects. The final colour of the ray (and therefore of the corresponding pixel) is given by the colours of the objects hit by the ray as it travels through the scene.



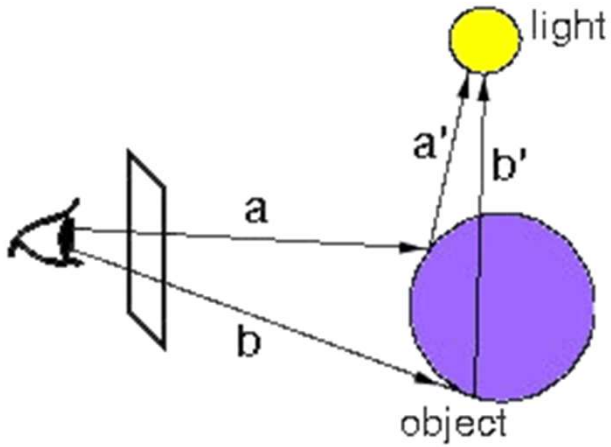
How ray tracing is actually done

- Just as in the light-source-to-eye method it might take a very large number of bounces before the ray ever hits the eye, in backwards method it might take many bounces before the ray every hits the light. Since we need to establish some limit on the number of bounces to follow the ray on, we make the following approximation: every time a ray hits an object, we follow a single new ray from the point of intersection *directly towards* the light source.



Ray tracing

- In the figure below we see two rays, **a** and **b**, which intersect the purple sphere. To determine the colour of **a**, we follow the new ray **a'** directly towards the light source. The colour of **a** will then depends on several factors, such as shadows, shading, refraction and reflection.
- As you can see, **b** will be shadowed because the ray **b'** towards the light source is blocked by the sphere itself. Ray **a** would have also been shadowed if another object blocked the ray **a'**.



The ray tracing algorithm
A simplistic pseudo code version

Initialise

Select eye point (E), look point (L) and up vector (U)

Set screen plane to be centred on L, perpendicular to vector EL

Set screen size and number of pixels

Main Loop

```
For each pixel {

    Define D to be the unit vector pointing from E
    to the centre of the pixel

    Raytrace(E, D)

}
```

Ray Trace

```
function Raytrace(E, D) returns Colour
{
     nearest_t = infinity
     nearest_object = NULL

     for each object {

          find t, the smallest, non-negative real solution of the
            ray/object intersection equation

          if t exists {
               if t < nearest_t {
                    nearest_t = t
                    nearest_object = current object
               }
          }

     }

     ...
```

Ray Trace

```
...

colour = black

if nearest_object exists {

find normal vector, N, at intersection point

if object is reflective {
     reflected_colour=Raytrace(intersection point,reflection vector)
     colour += reflection_coeff * reflected_colour
}

if object is refractive {
     refracted_colour=Raytrace(intersection point, refracted vector)
     colour += refraction_coeff * refracted_colour
}

...
```

Ray Trace

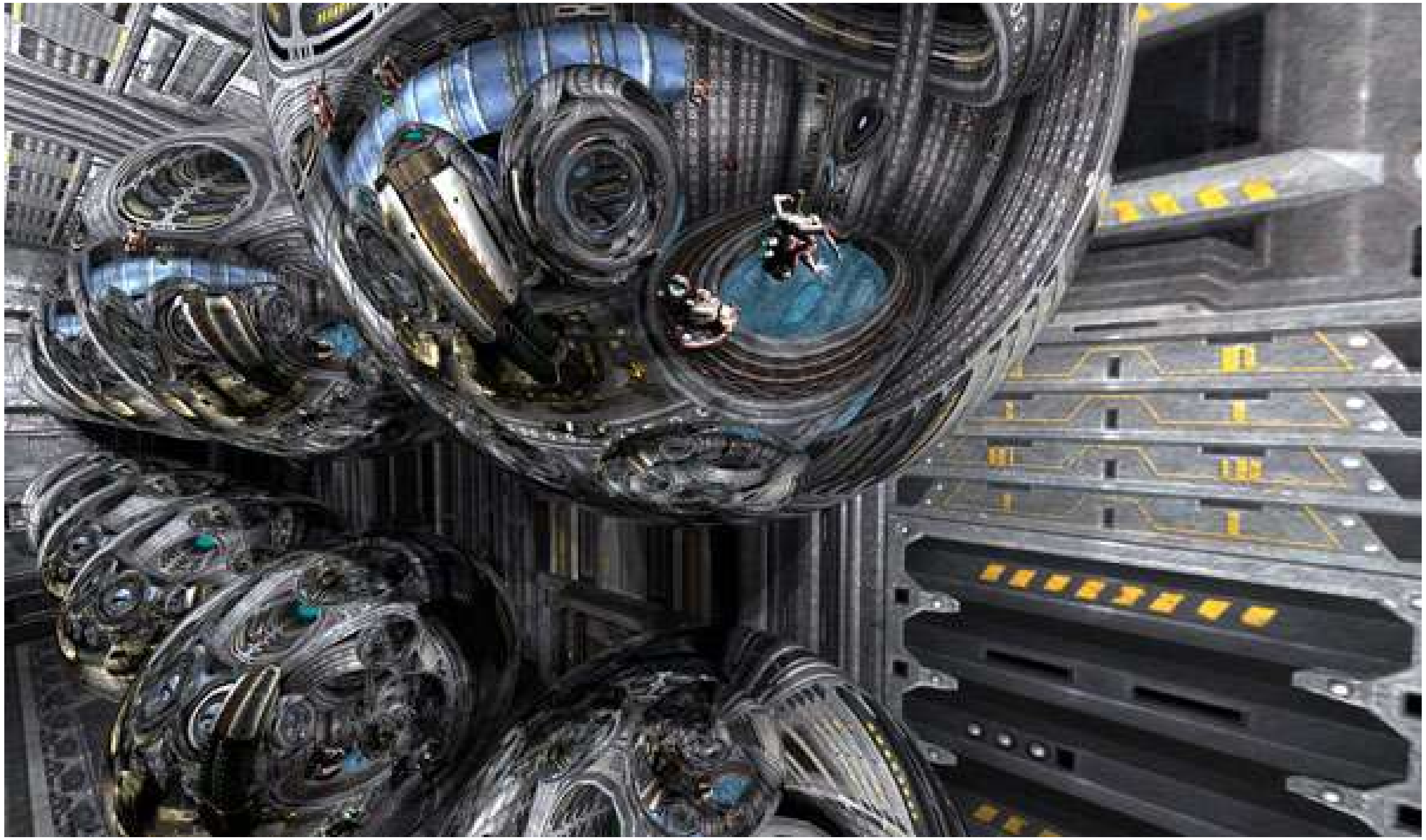
```
    ...

     for each light {
          if shadow_ray(intersection point, light position)
               returns No_Shadow {
                    calculate light's colour contribution by doing the
                       illumination calculations using D, N, the current
                       light, and the object properties
                    colour += light's colour contribution
               }
          }
     }
    return colour
}
```

Shadow Ray

```
function shadow_ray(point1, point2) returns Shadow or No_Shadow {
     ray defined with E=point1, D=point2-point1
     nearest_t = infinity
     nearest_object = NULL
     for each object {
          find t, the smallest, non-negative real solution of the
            ray/object intersection equation
          if t exists {
               if t < nearest_t {
                    nearest_t = t
               }
          }
     }
     if t <1
          return Shadow
     else
          return No_Shadow
}
```

Ray Tracing Gallery


© Copyright Phil Wolstenholme 2003
Rendered with Brazil r/s



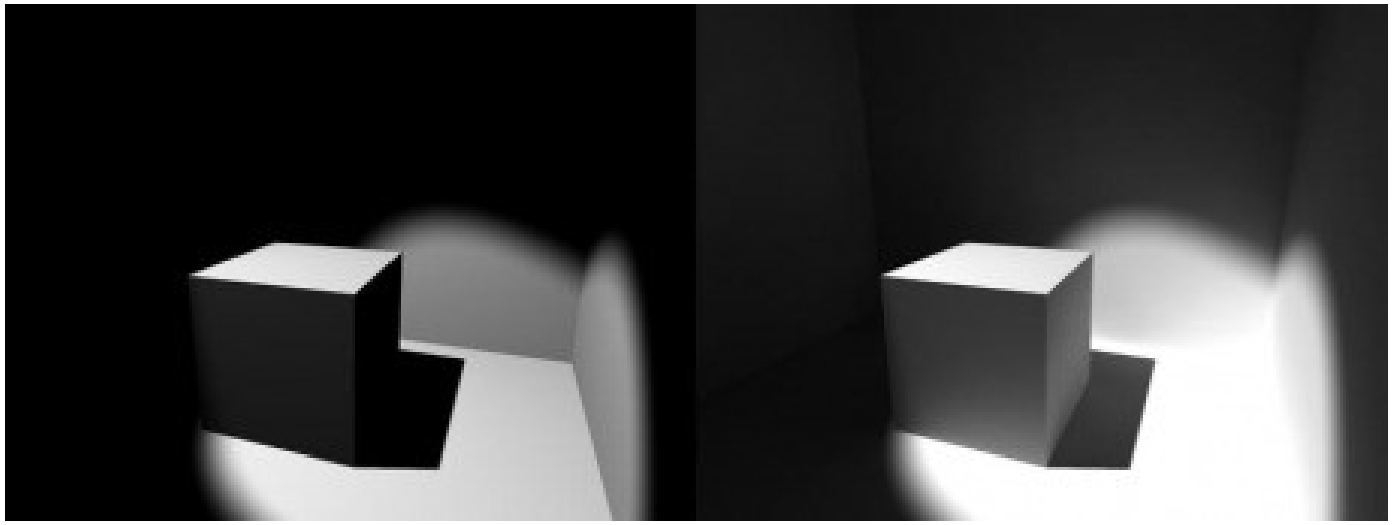

East Bank

Radiosity
A bit more on...

Radiosity

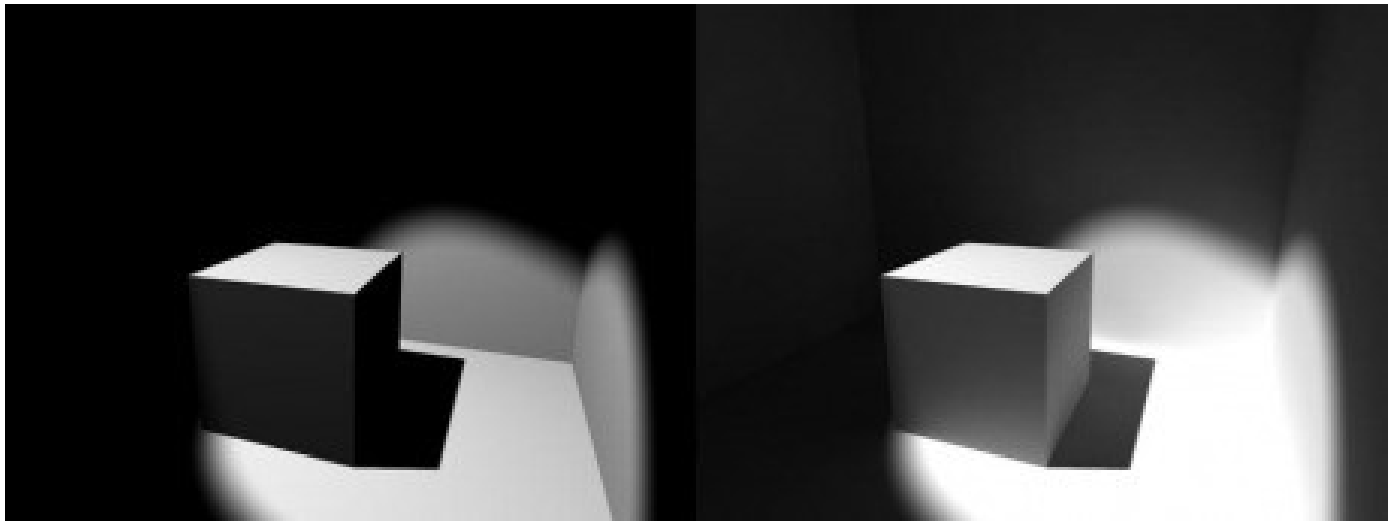
- The render on the left shows a normal spotlight aimed at a corner of the room.

- It hits the top of the box and a few walls, but since radiosity is off, that's where the life of the light ends.



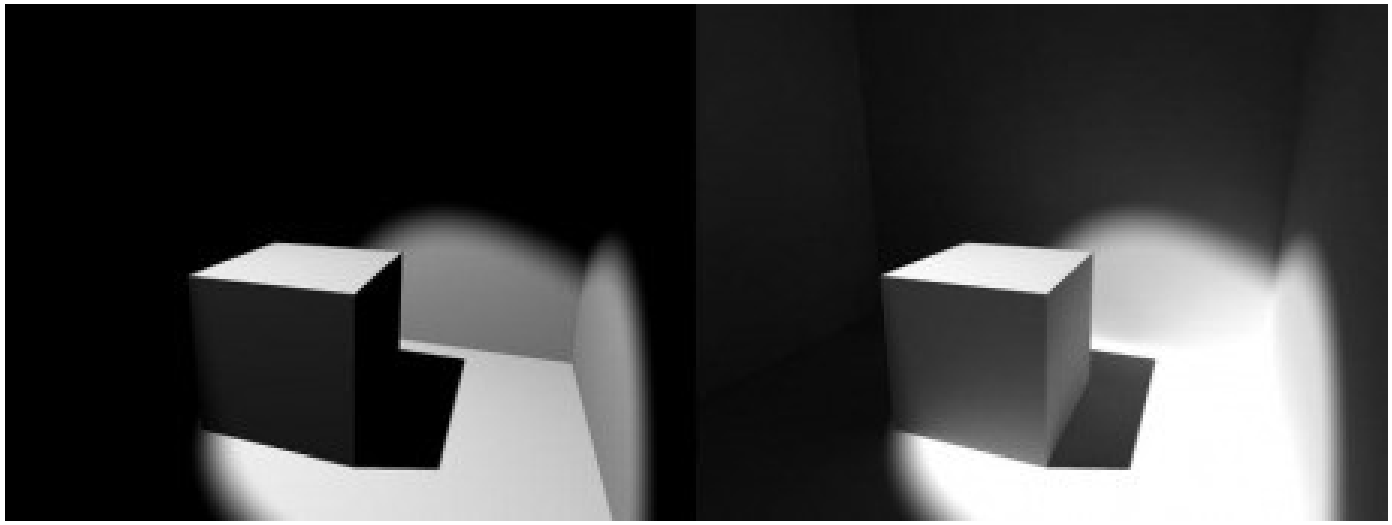
Radiosity

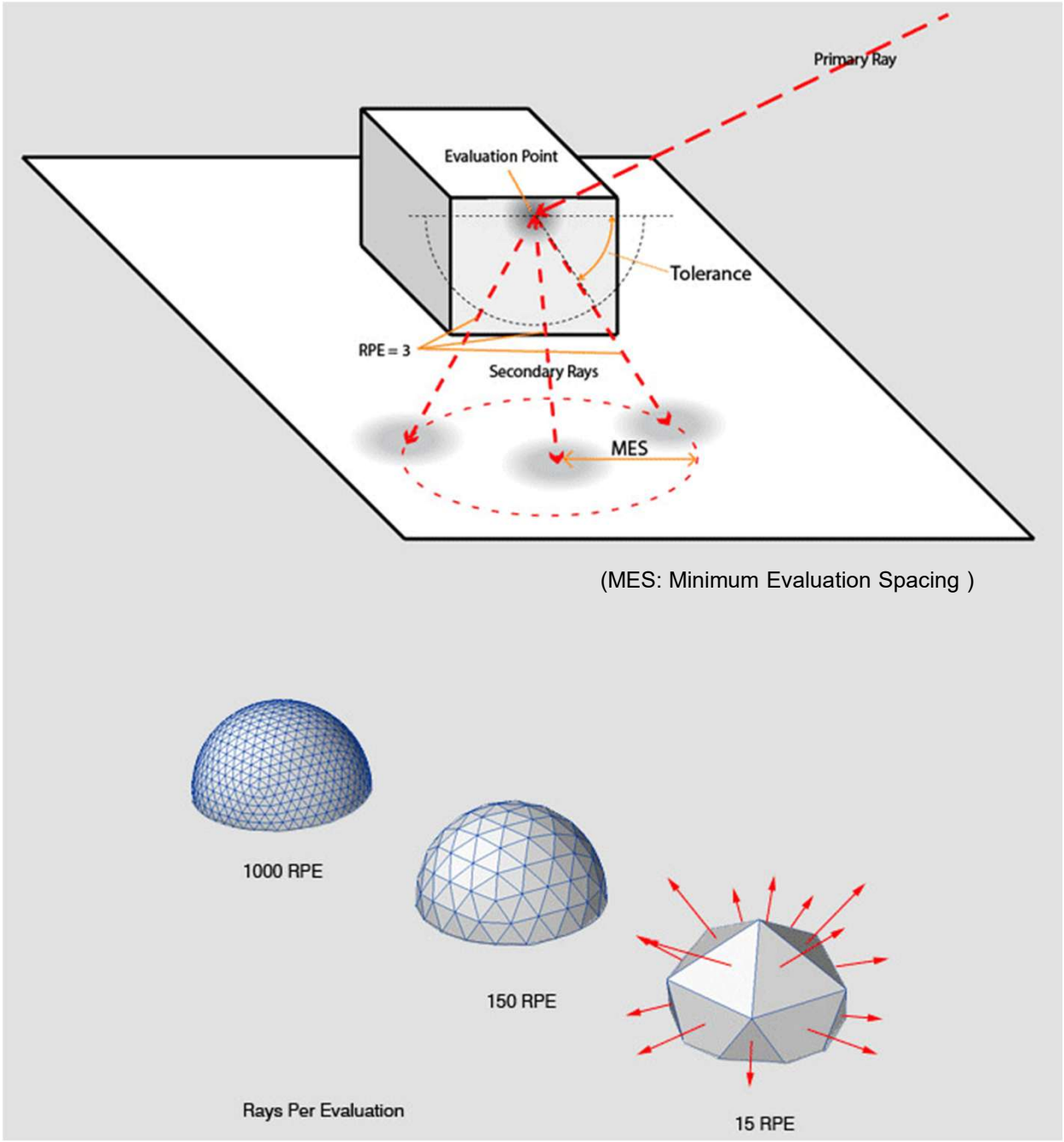
- On the right, we see what happens when radiosity is turned on - light reflects off the walls and the box and now surfaces that were not in the direct path of the spotlight have also been lit up!

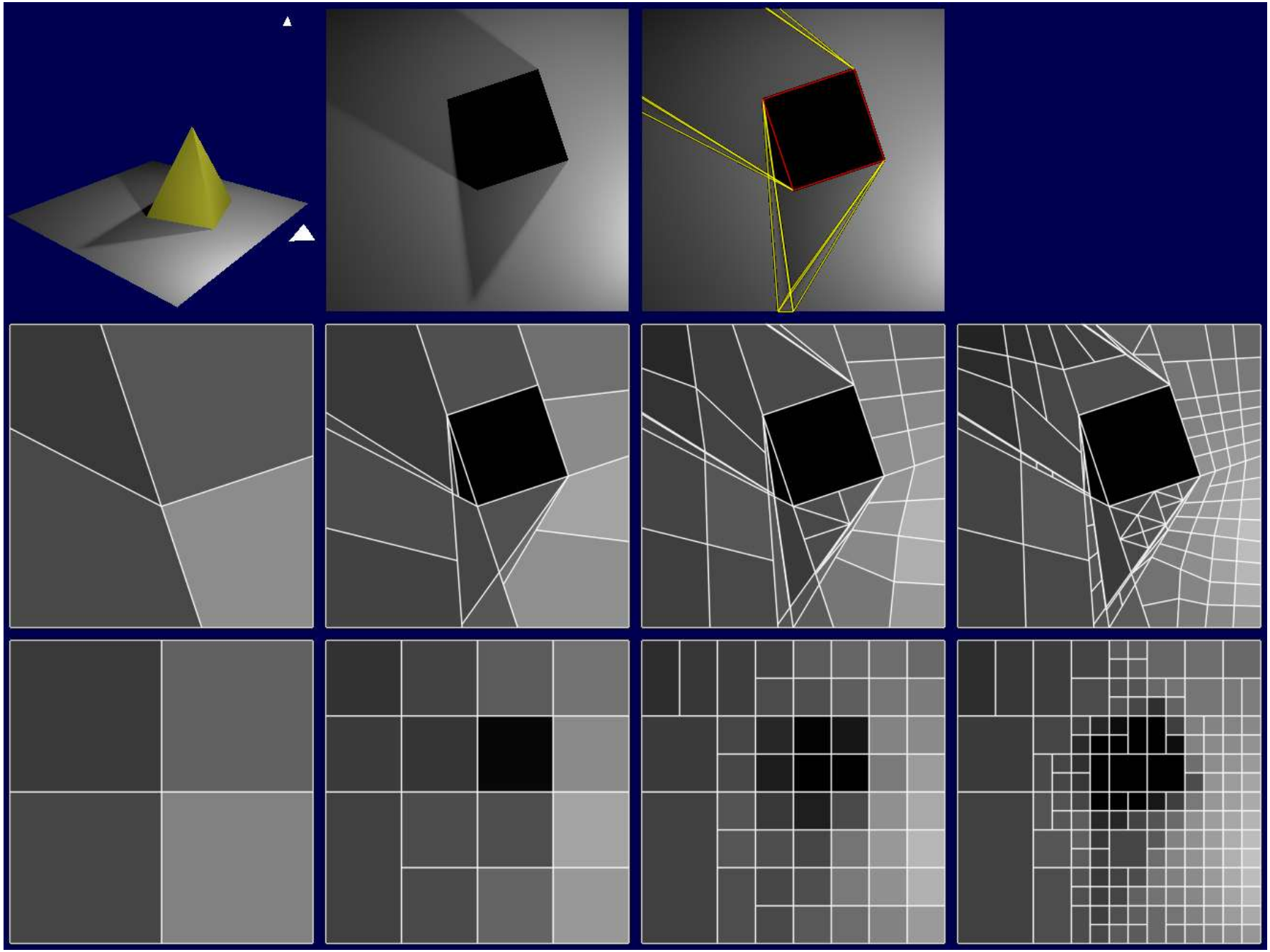


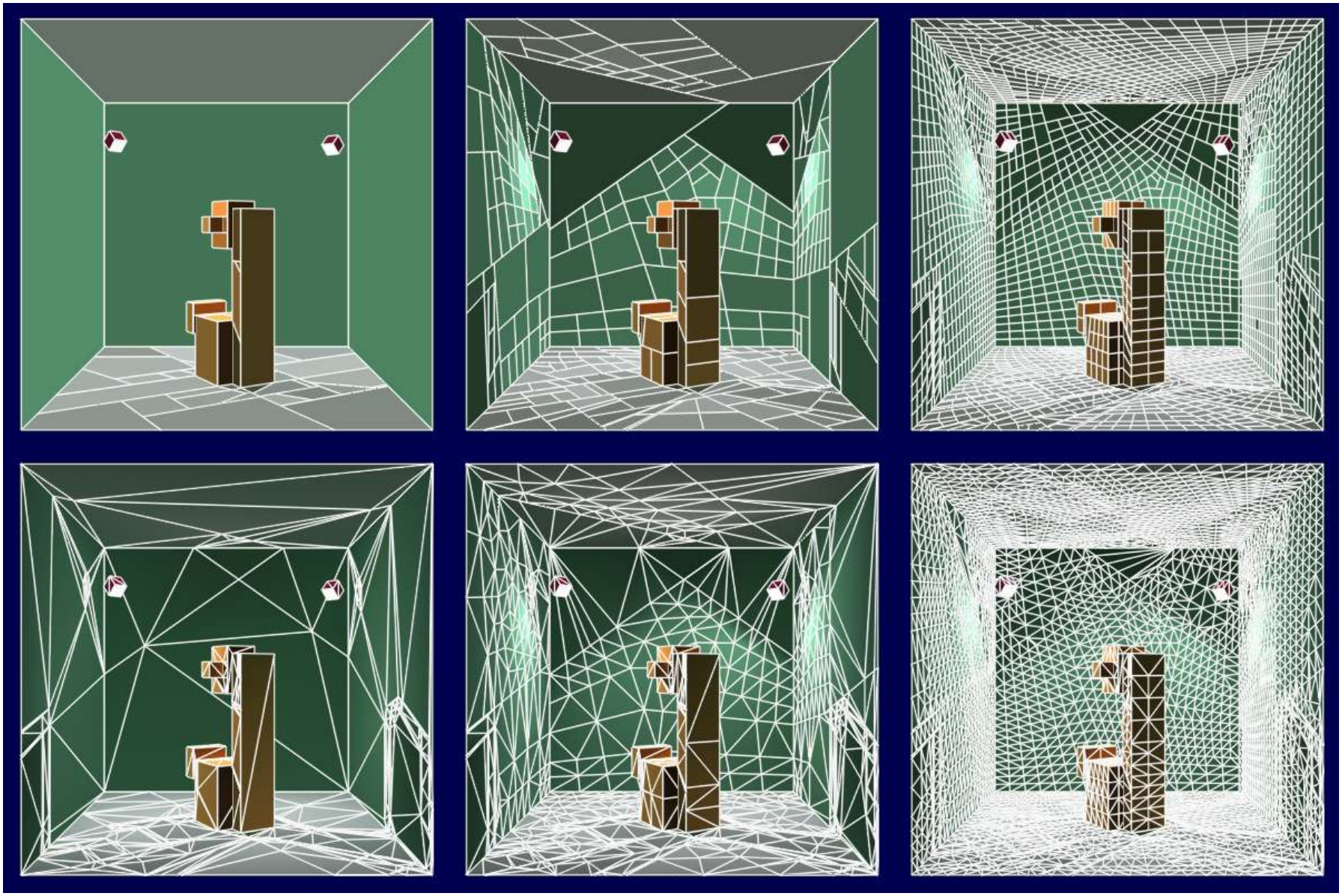
Radiosity

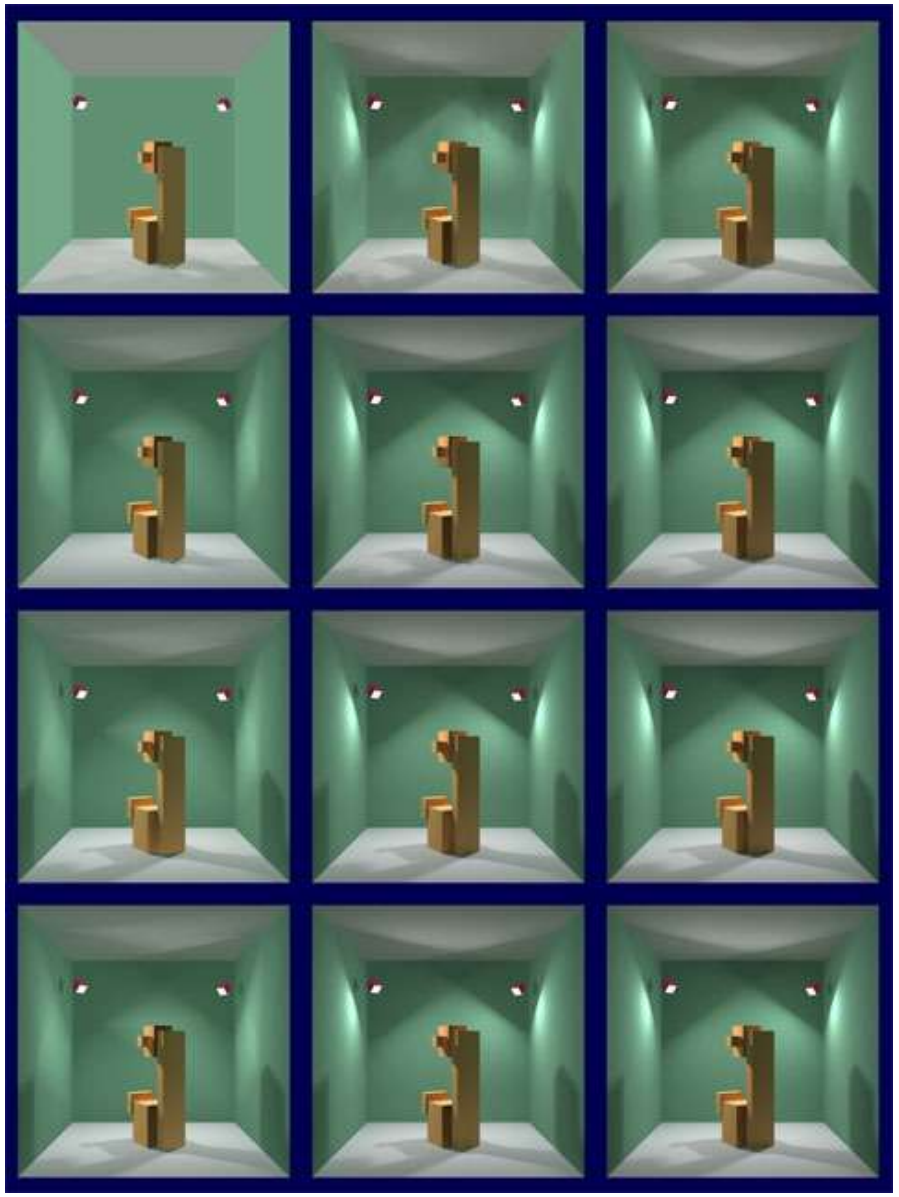
- Note the corner of the room, where the two lit walls bounce light onto each other, causing the crease between the walls to get so much light it's now overexposed – just as it would be in real life.



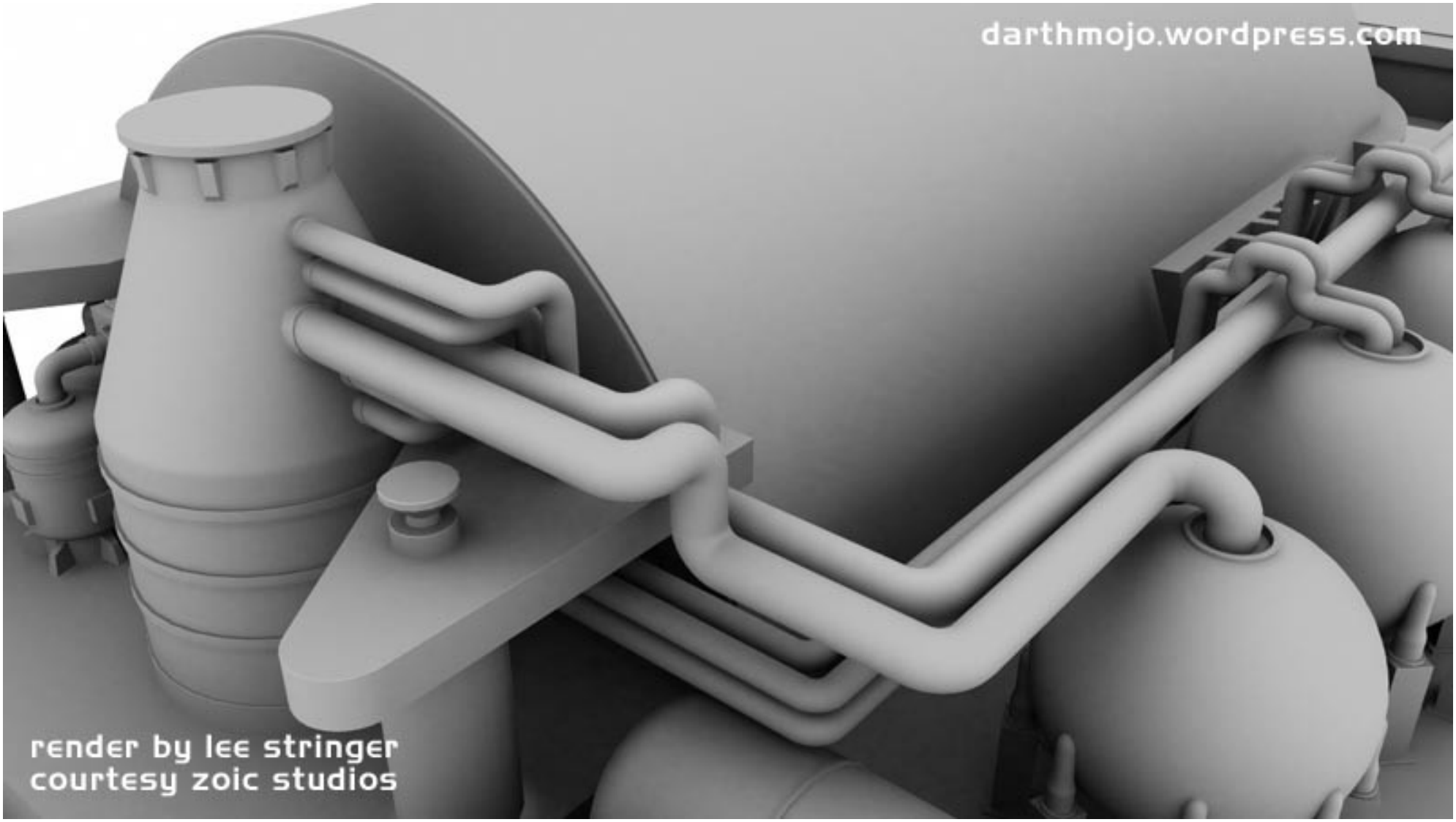

Primary Ray
Evaluation Point
Tolerance
RPE = 3
Secondary Rays
MES
(MES: Minimum Evaluation Spacing)
1000 RPE
150 RPE
15 RPE
Rays Per Evaluation





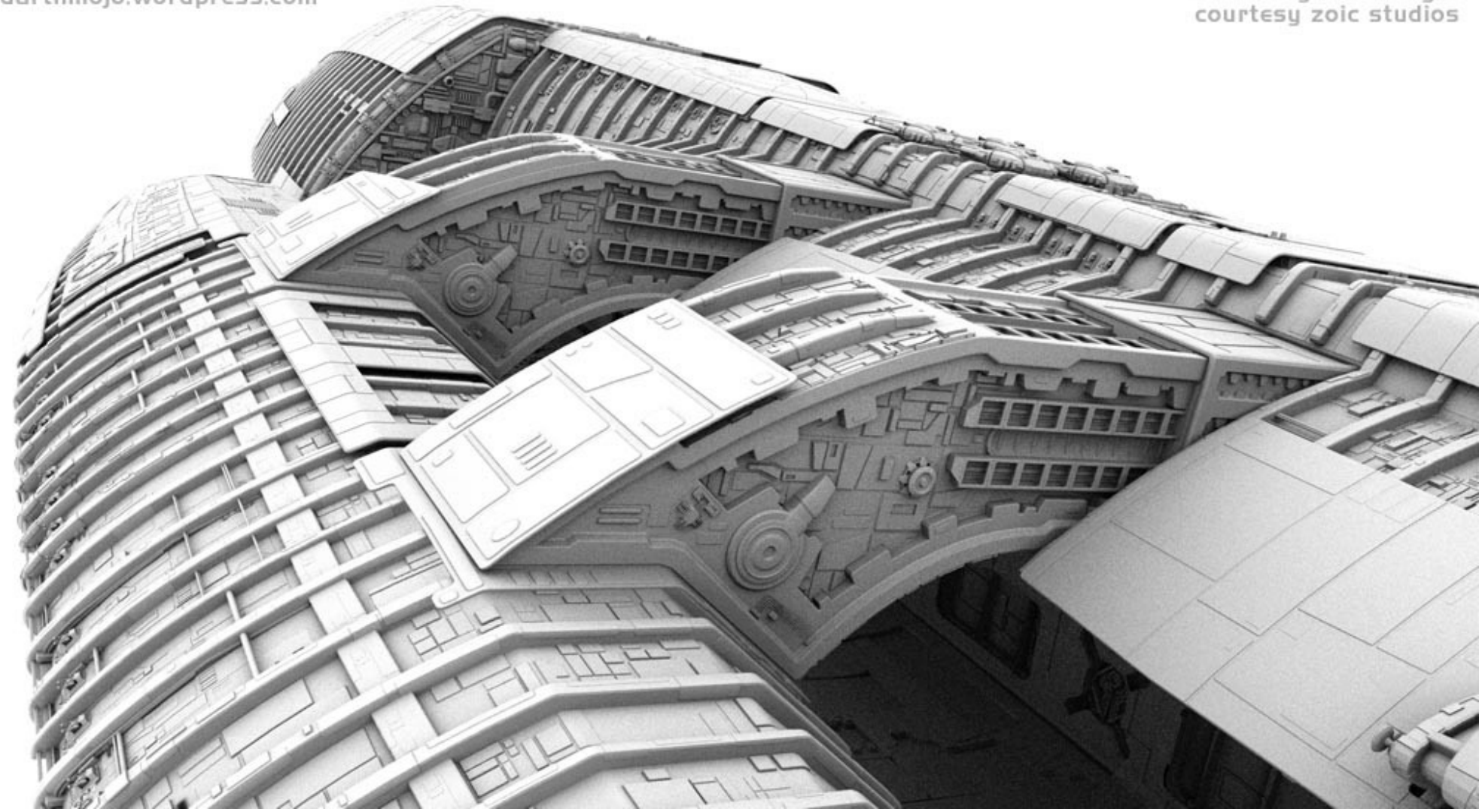


Radiosity Gallery


darthmojo.wordpress.com
render by lee stringer
courtesy zoic studios

darthmojo.wordpress.com

render by lee stringer
courtesy zoic studios



EXAMPLE VIDEO

Real-Time Ray Tracing & Radiosity

- 01 – GPU Retracing with Nvidia Optix
- 02 – Intel Xeon 7500 Processor Performance
- 03 – Intel Labs: Wolfenstein Gets Ray Traced
- 04 – IDF 2010 : Cloud based Retracing for Games
- 05 – Frostbite 2 Engine Real-Time Radiosity

END