INFOGR – Computer Graphics

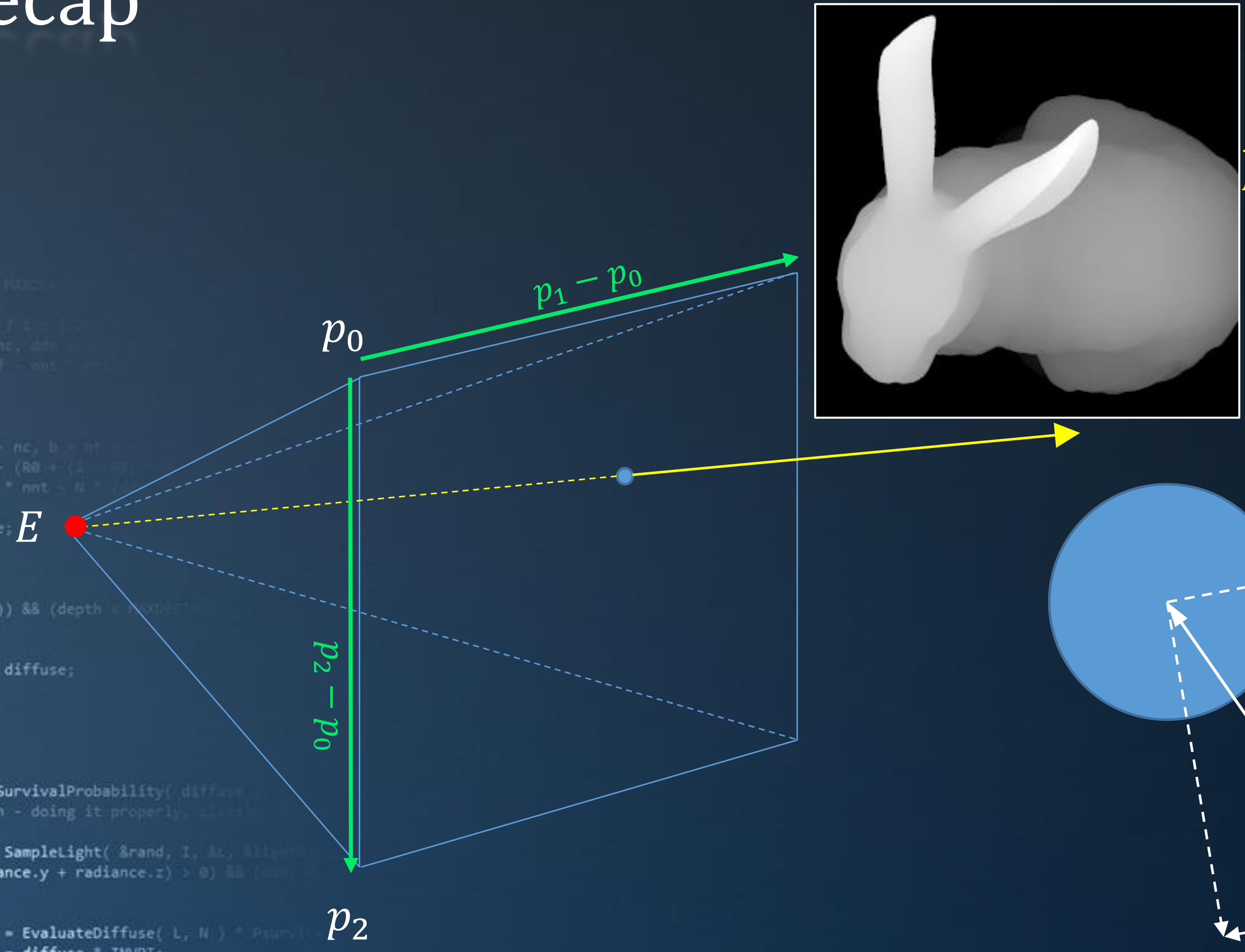
Jacco Bikker - April-July 2016 - Lecture 4: “Ray Tracing (2)”

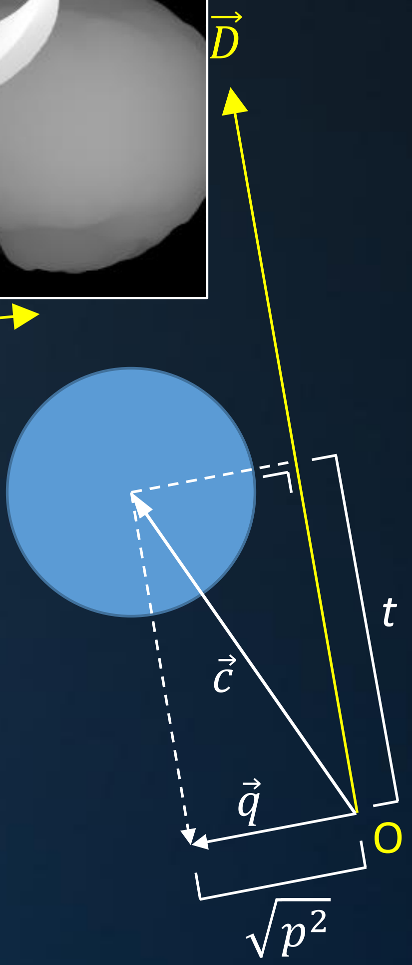
Welcome!

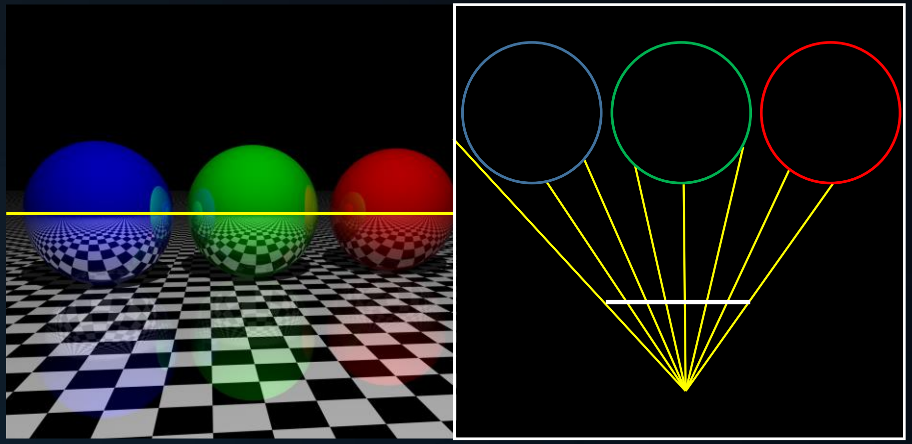
Today's Agenda:

- Recap
- End of the Primary Ray
- Normals
- The Camera
- Assignment P2

Recap



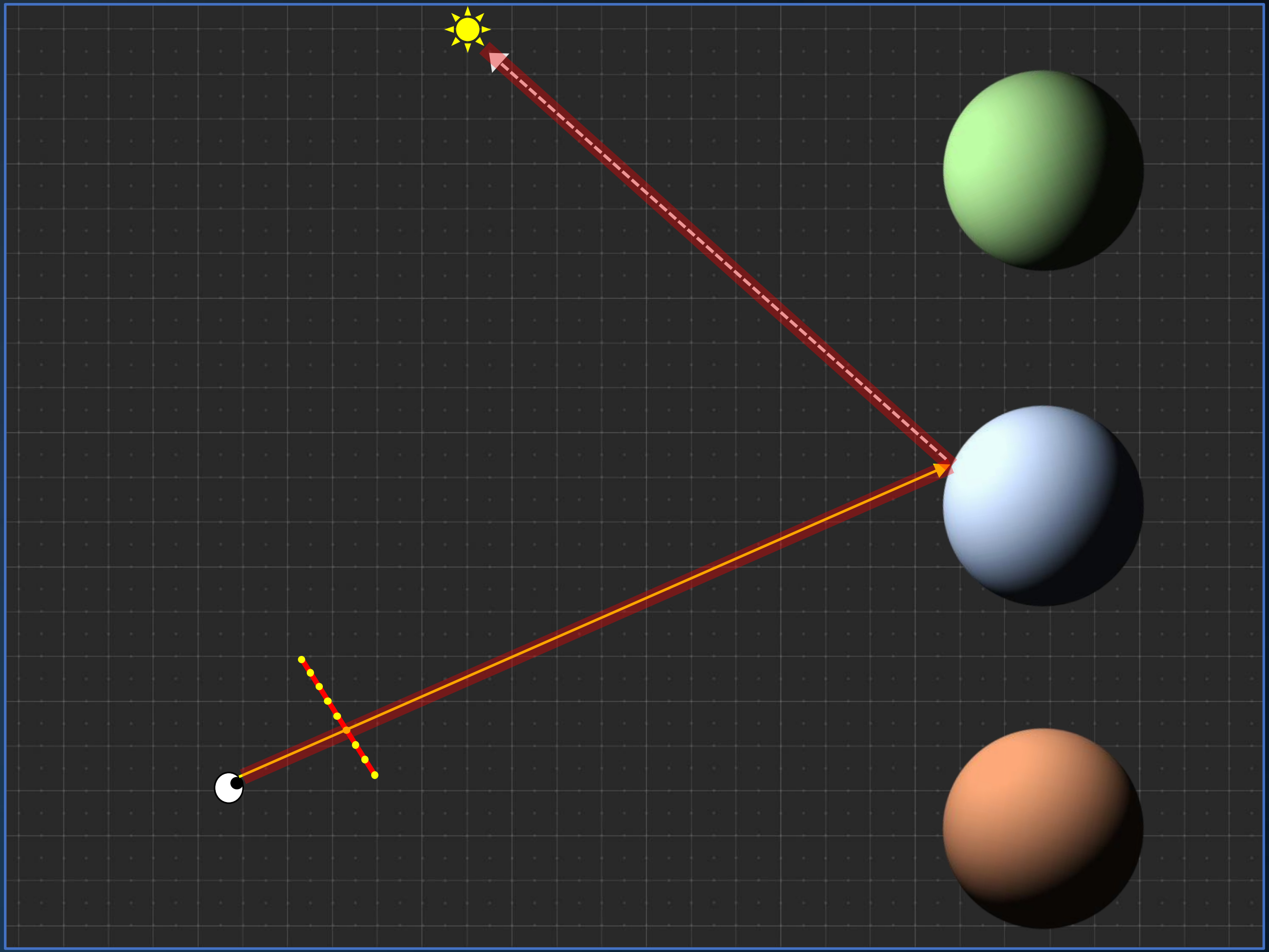




Today's Agenda:

- Recap
- End of the Primary Ray
- Normals
- The Camera
- Assignment P2

Shading



Shading

The End

We used *primary rays* to find the *primary intersection* point.

Determining light transport:

- Sum illumination from all light sources
- ...If they are *visible*.

We used a primary ray to find the object visible through a pixel:
Now we will use a *shadow ray* to determine visibility of a light source.

Shading

Shadow Ray

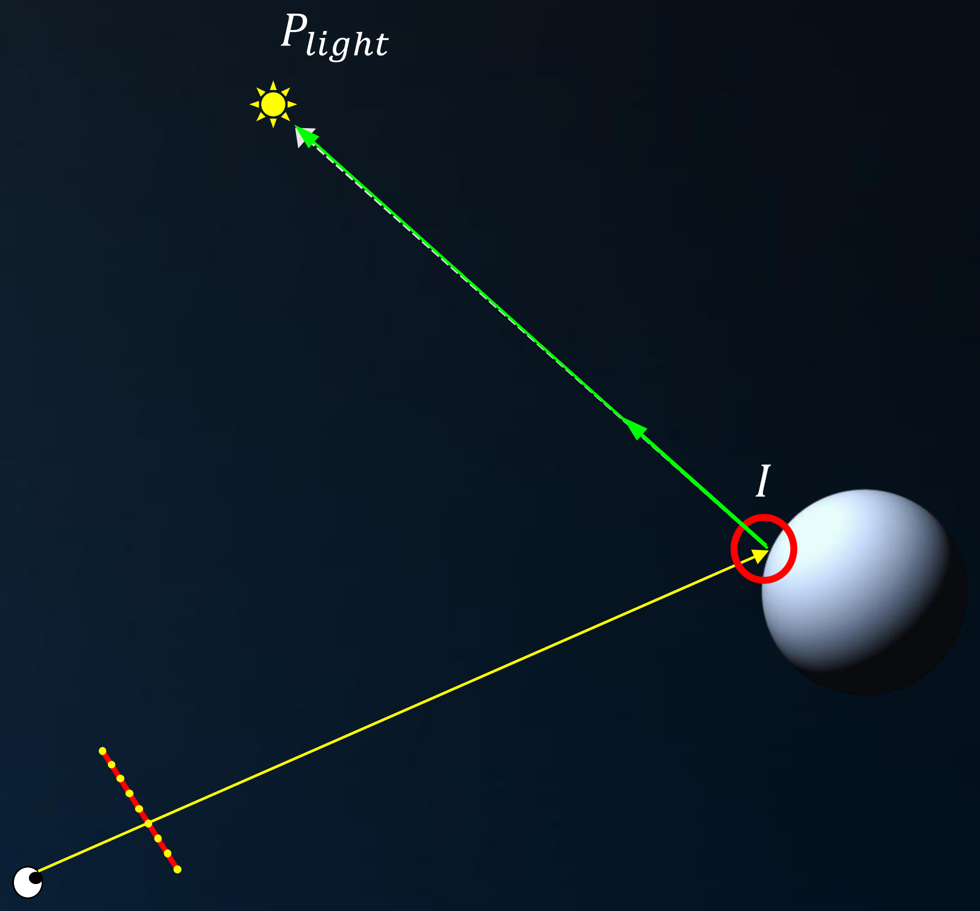
Constructing the shadow ray:

$$p(t) = O + t\vec{D}$$

Ray origin: the primary intersection point I.

Ray direction: $P_{light} - I$ (normalized)

Restrictions on t: $0 < t < ||P_{light} - I||$



Shading

Shadow Ray

Direction of the shadow ray: $\frac{P_{light}-I}{||P_{light}-I||}$

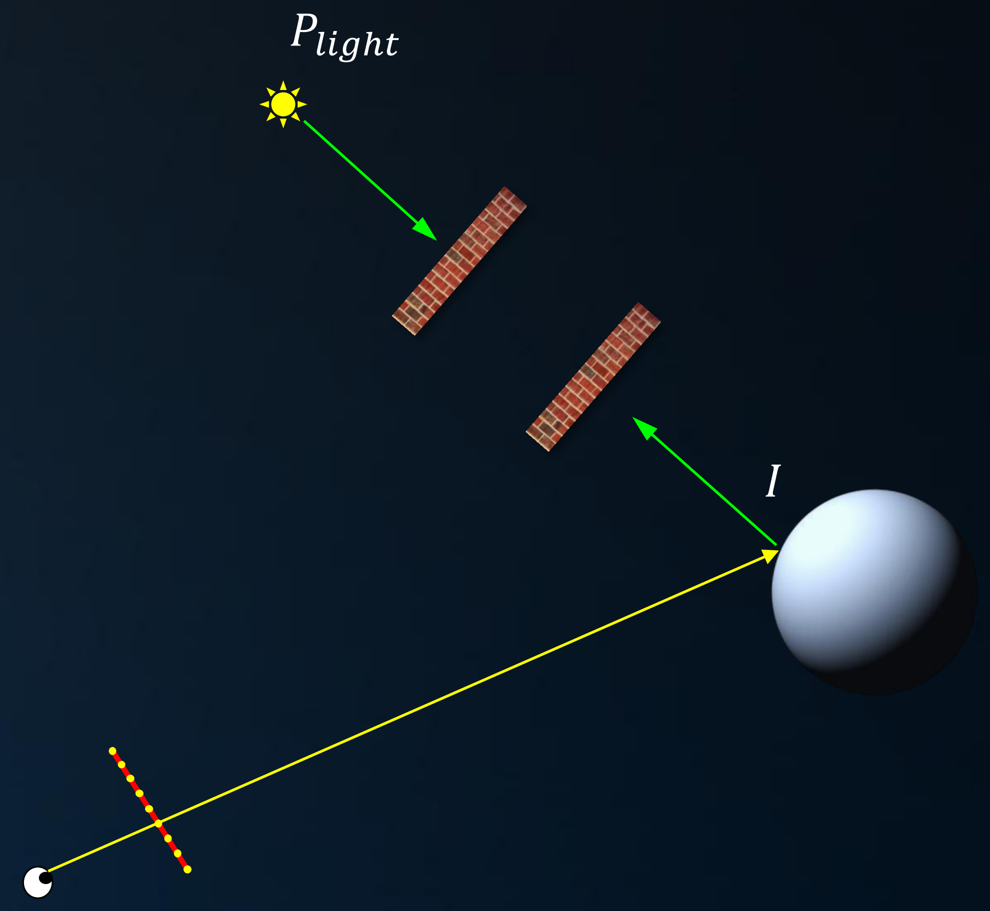
Equally valid: $\frac{I-P_{light}}{||P_{light}-I||}$ or $\frac{I-P_{light}}{||I-P_{light}||}$

Note that we get different intersection points depending on the direction of the shadow ray.

It doesn't matter: the shadow ray is used to determine *if* there is an occluder, not *where*.

This has two consequences:

1. We need a dedicated shadow ray query;
2. Shadow ray queries are (on average) twice as fast. **(why?)**



Shading

Shadow Ray

“In theory, theory and practice are the same. In practice, they are not.”

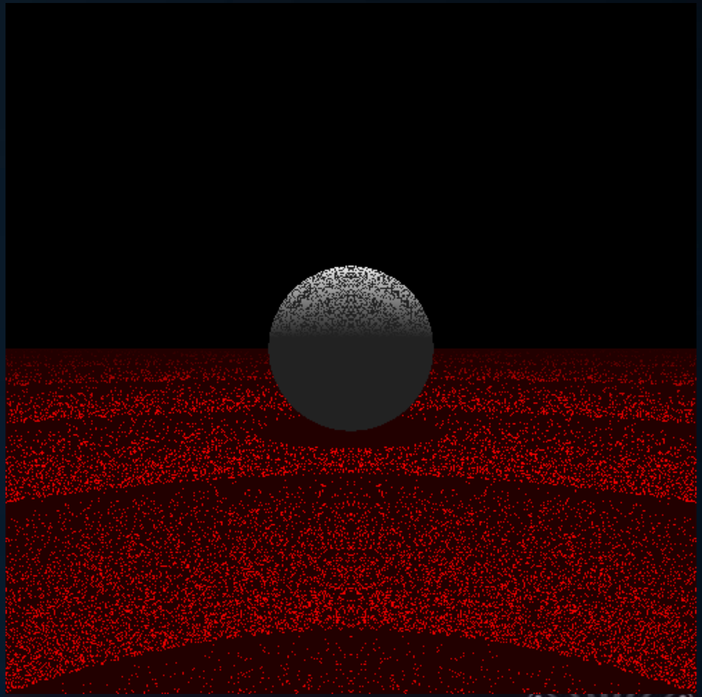
Problem 1:

Our shadow ray queries report intersections at $t = {\sim}0$. Why?

Cause: the shadow ray sometimes finds the surface it originated from as an occluder, resulting in *shadow acne*.

Fix: offset the origin by ‘epsilon’ times the shadow ray direction.

Note: don’t forget to reduce t_{max} by epsilon.



Shading

Shadow Ray

“In theory, theory and practice are the same. In practice, they are not.”

Problem 2:

Our shadow ray queries report intersections at $t = t_{max}$. Why?

Cause: when firing shadow rays from the light source, they may find the surface that we are trying to shade.

Fix: reduce $t_{\max}$ by $2 * epsilon$.

Shading

Shadow Ray

“The most expensive shadow rays are those that do not find an intersection.”

Why?

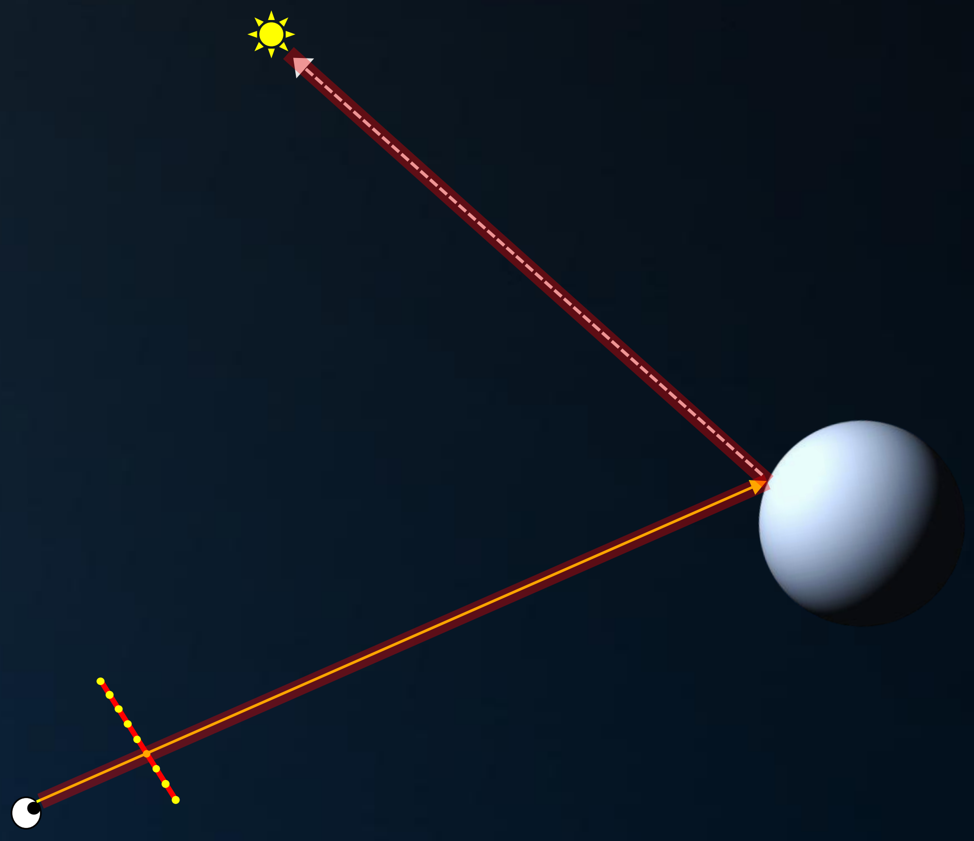
(because those rays tested every primitive before concluding that there was no occlusion)

Shading

Transport

The amount of energy travelling from the light via the surface point to the eye depends on:

- The brightness of the light source
- The distance of the light source to the surface point
- Absorption at the surface point
- The angle of incidence of the light energy



Shading

Transport

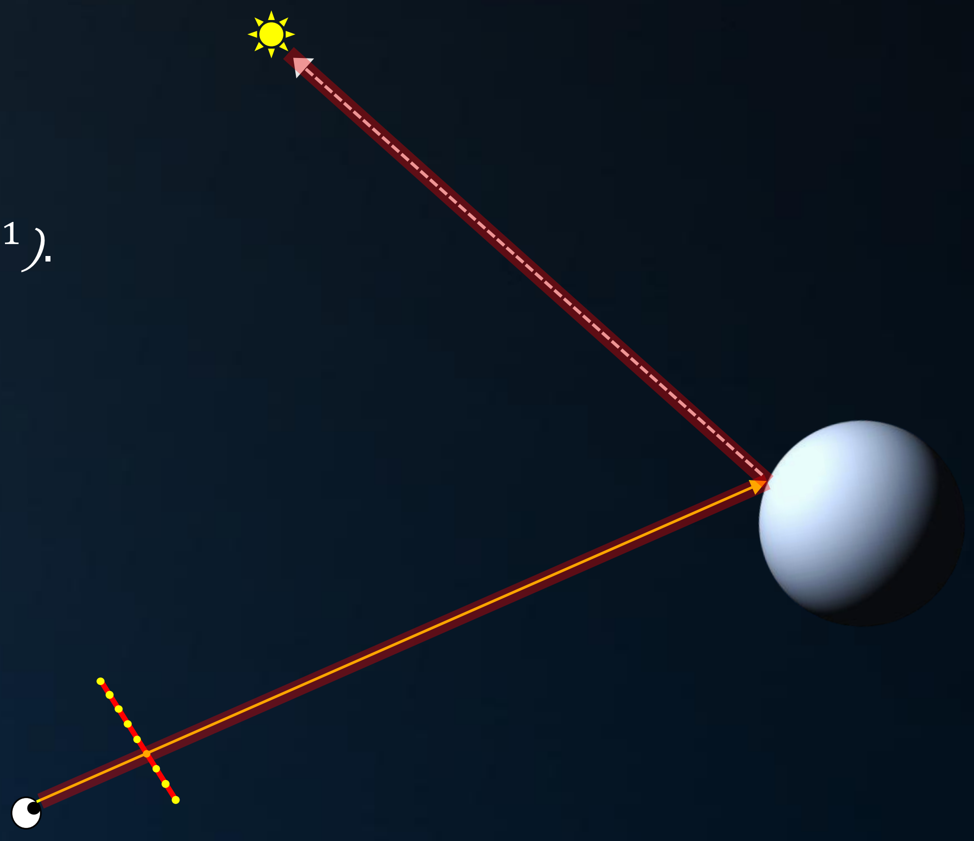
Brightness of the light source:

Expressed in *watt (W)*, or *joule per second* $(J/s \text{ or } Js^{-1})$.

Energy is transported by photons.

Photon energy depends on wavelength; energy for a ‘yellow’ photon is $\sim 3.31 \cdot 10^{-19}$ J.

A 100W light bulb thus emits $\sim 3.0 \cdot 10^{20}$ photons per second.



Shading

Transport

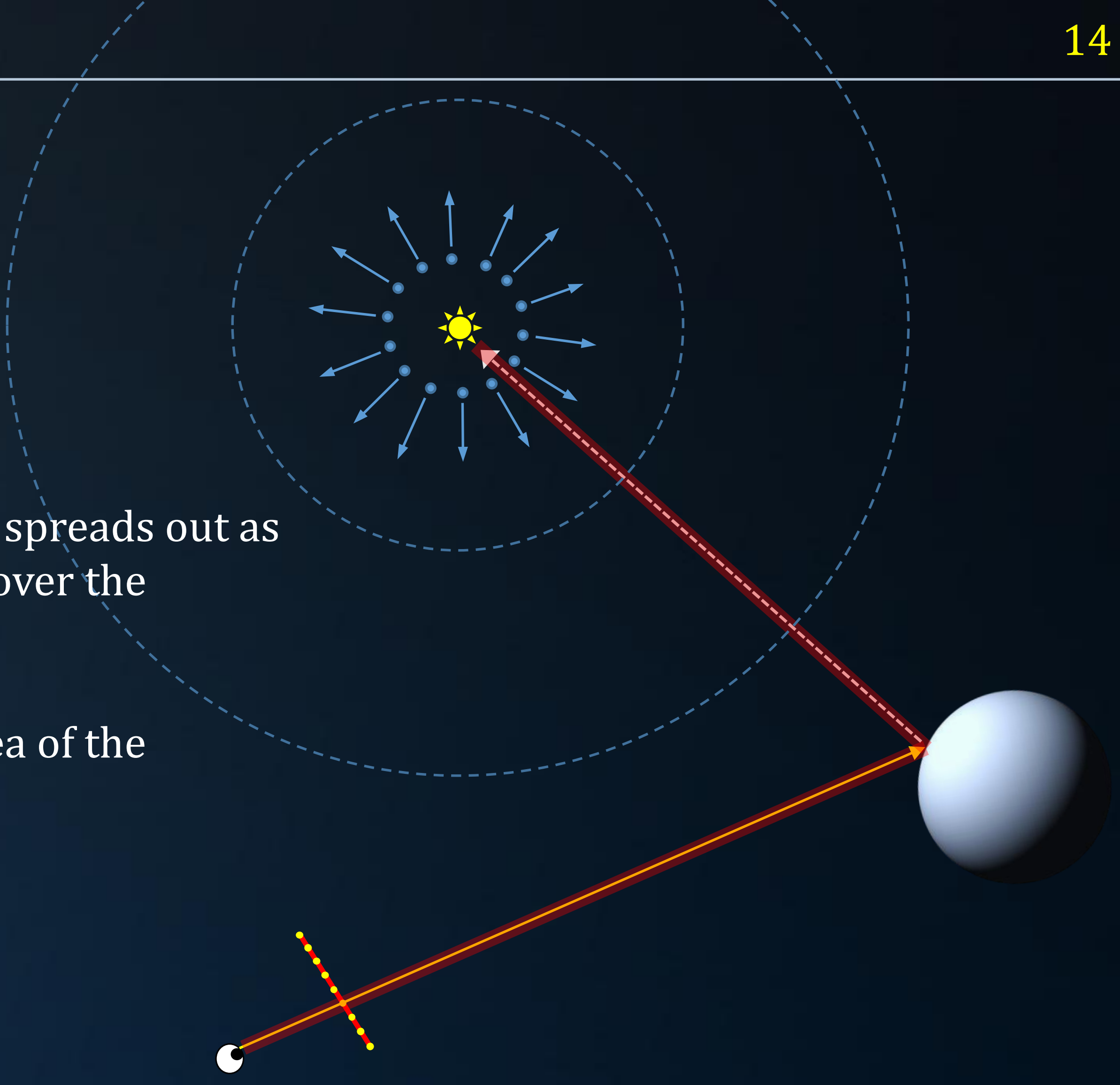
Energy at distance r:

For a point light, a brief pulse of light energy spreads out as a growing sphere. The energy is distributed over the surface of this sphere.

It is therefore proportional to the inverse area of the sphere at distance r, i.e.:

$$E/m^2 = E_{light}\frac{1}{4\pi r^2}$$

Light energy thus dissipates at a rate of $\frac{1}{r^2}$.
This is referred to as *distance attenuation*.



Shading

Transport

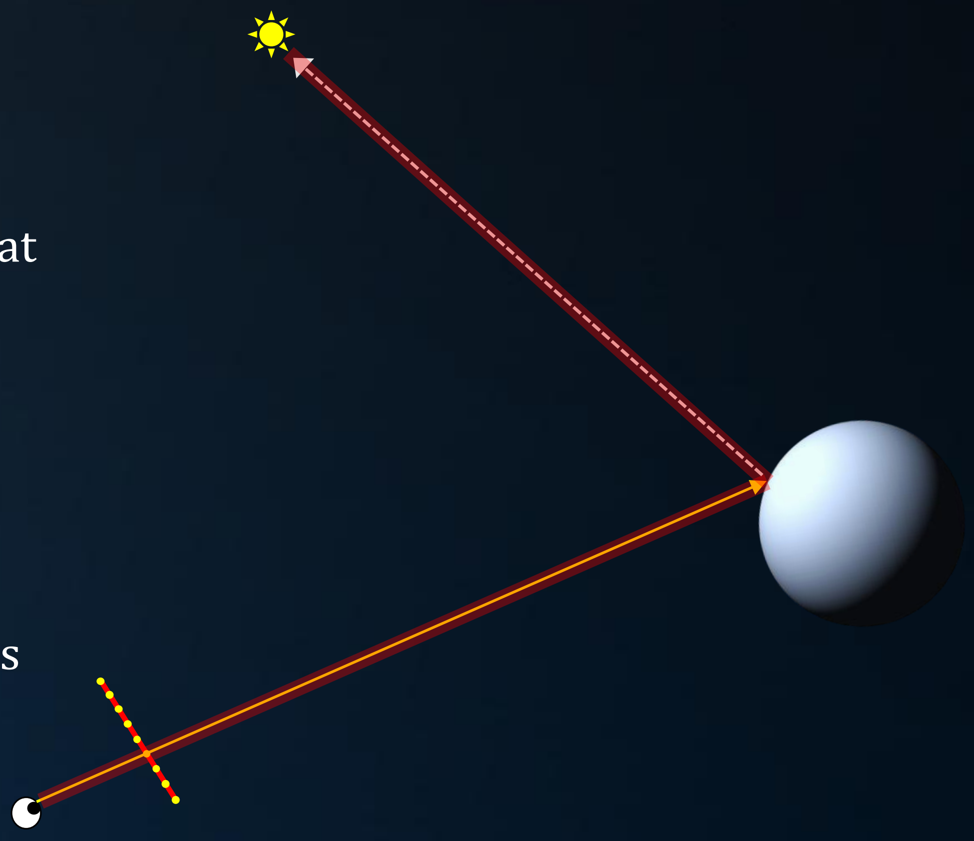
Absorption:

Most materials absorb light energy. The wavelengths that are not fully absorbed define the ‘color’ of a material.

The reflected light is thus:

$$E_{reflected} = E_{incoming} \cdot C_{material}$$

Note that $C_{material}$ cannot exceed 1; the reflected light is never *more* than the incoming light.



Shading

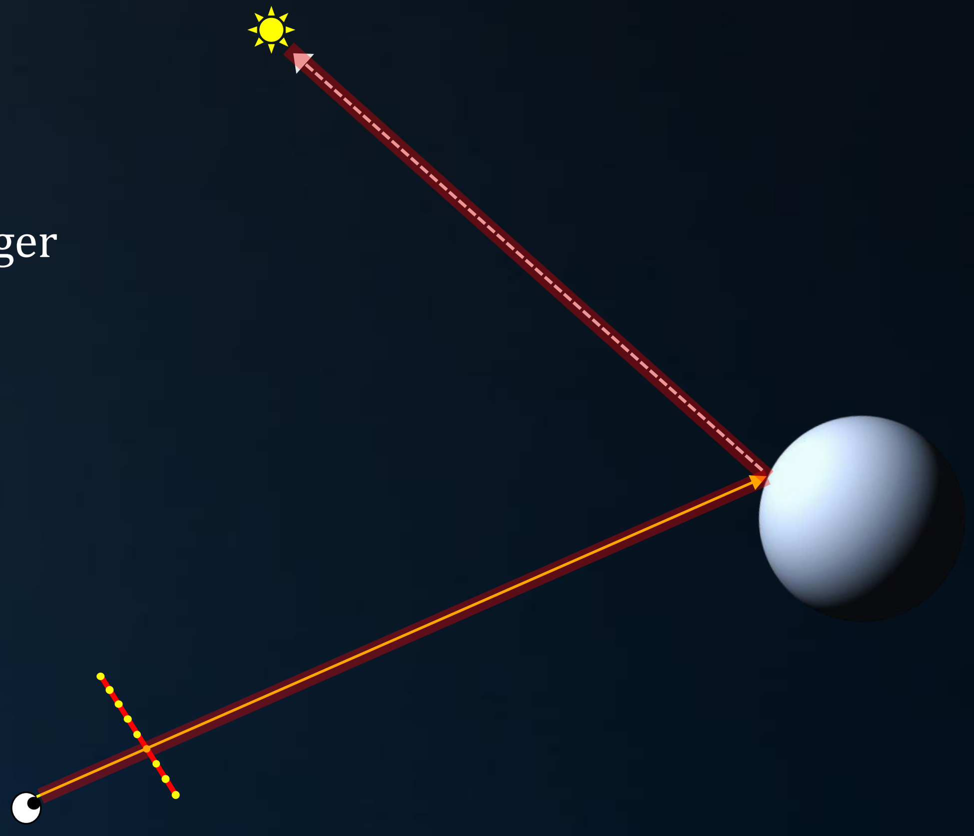
Transport

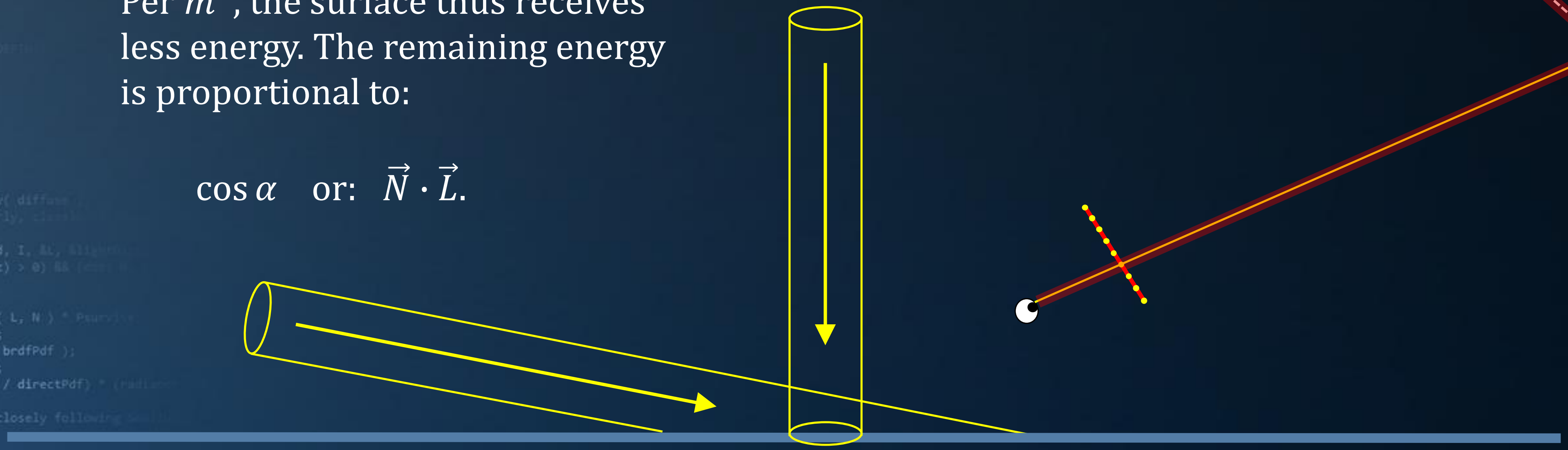
Energy arriving at an angle:

A small bundle of light arriving at a surface affects a larger area than the cross-sectional area of the bundle.

Per m^2, the surface thus receives less energy. The remaining energy is proportional to:

$$\cos\alpha \quad \text{or:} \quad \vec{N}\cdot\vec{L}.$$



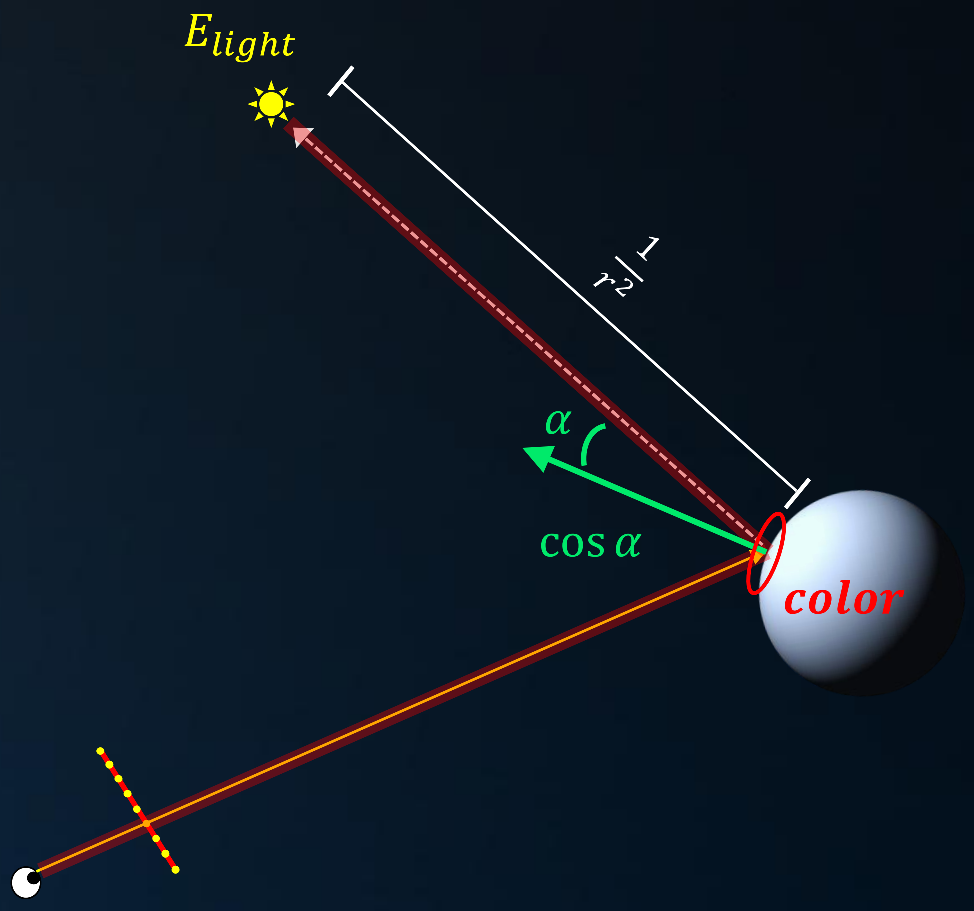


Shading

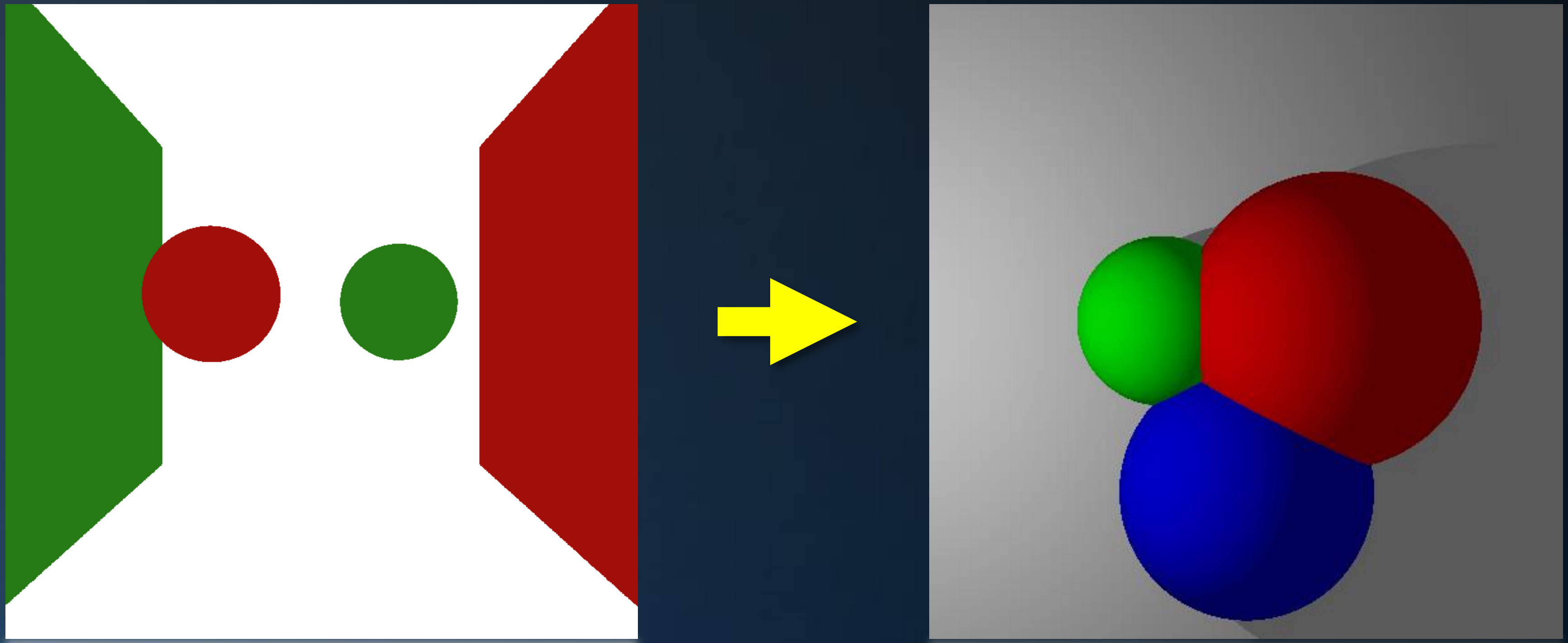
Transport

All factors:

- Emitted light : defined as RGB color, floating point
- Distance attenuation: $\frac{1}{r^2}$
- Absorption, modulate by material color
- N dot L



Shading



Today's Agenda:

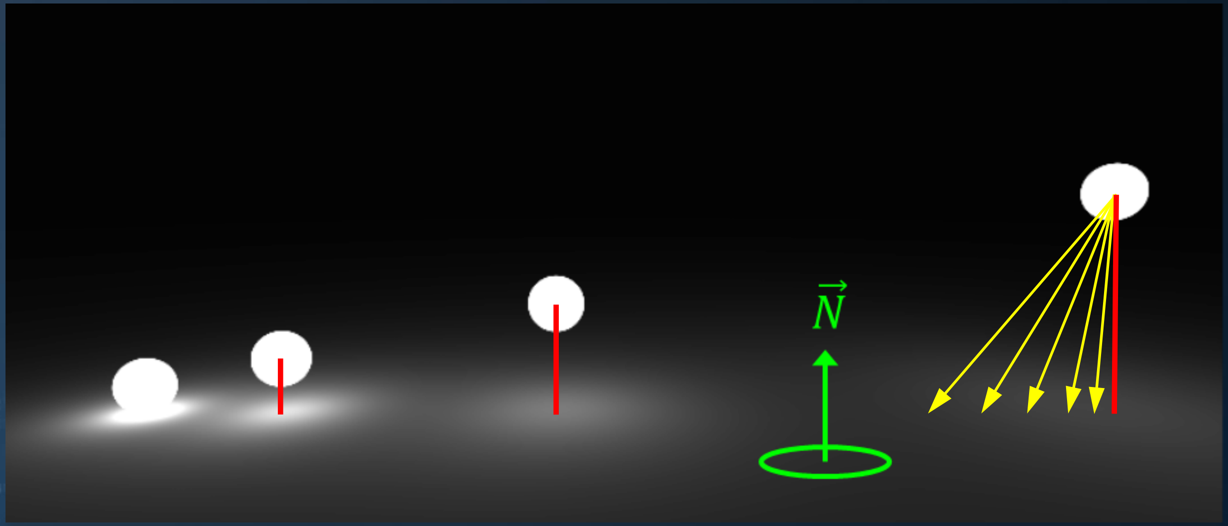
- Recap
- End of the Primary Ray
- Normals
- The Camera
- Assignment P2

Normals

We Need a Normal

For a plane, we already have the normal.

$$Ax + By + Cz + D = 0 \quad \text{or} \quad (P \cdot \vec{N}) + D = 0$$



Distance attenuation: $1/r^2$

Angle of incidence: $N \cdot L$

Normals

We Need a Normal

Question:

How does light intensity relate to scene size?
i.e.: if I scale my scene by a factor 2, what should I do to my lights?

➔ Distance attenuation requires scaling light intensity by 2^2

➔ Scene scale does not affect $N \cdot L$.

Normals

We Need a Normal

Question:

What happens when a light is near the horizon?

➔ Angle approaches 90°; $\cos\alpha$ approaches 0

➔ Light is distributed over an infinitely large surface

Note: below the horizon, $\cos\alpha$ becomes negative.
➔ Clamp $\vec{N}\cdot\vec{L}$ to zero.

Normals

We Need a Normal

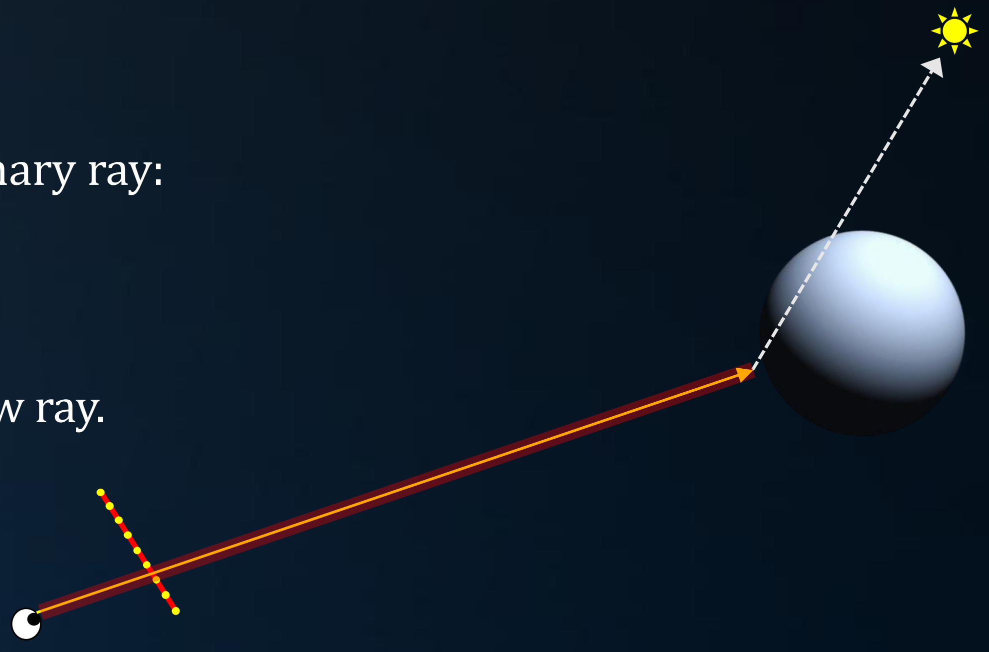
Normals are also used to *prevent* shadow rays.

Situation:

A light source is behind the surface we hit with the primary ray:

$$\vec{N} \cdot \vec{L} < 0$$

In this case, visibility is 0, and we do not cast the shadow ray.



Normals

We Need a Normal

Normals for spheres:

The normal for a sphere at a point P on the sphere is parallel to the vector from the center of the sphere to P.

$$\vec{N}_P = \frac{P - C}{||P - C||}$$

Normals

We Need a Normal

Normals for spheres:

When a sphere is hit from the inside, we need to *reverse* the normal.

$$\vec{N}_P = \frac{C - P}{||P - C||}$$

How to detect this situation when it is not trivial:

1. Calculate the normal in the usual manner $(P - C)$;
2. If $\vec{N}_P \cdot \vec{D}_{ray} < 0$ then $\vec{N}_P = -\vec{N}_P$.

Normals

Normal Interpolation

Simulating smooth surfaces using normal interpolation:

1. Generate *vertex normals*.

A vertex normal is calculated by averaging the normals of the triangles connected to the vertex and normalizing the result.

2. Interpolate the normals over the triangle.

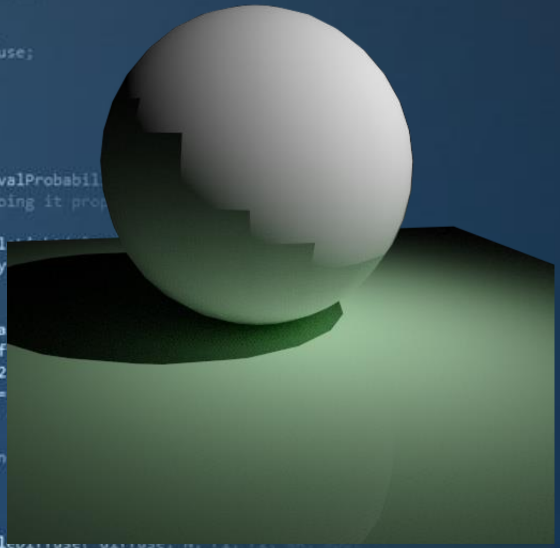
In a ray tracer, use barycentric coordinates to do this. Normalize the interpolated normal.

Normals

Normal Interpolation

Using the interpolated normal:

- Use the interpolated normal in the $\vec{N} \cdot \vec{L}$ calculation.
- Use the original face normal when checking if a light is visible.



Today's Agenda:

- Recap
- End of the Primary Ray
- Normals
- The Camera
- Assignment P2

Camera

Preparing a Free Camera

The view frustum is uniquely defined by:

- A camera position
- A target location
- A field of view (angle)
- A rotation around the view vector

We can limit this further by specifying an ‘up’ vector, e.g. (0, 1, 0):

- Camera and target position
- Field of view

Data we need to produce primary rays:

- Camera position, three screen corners.

Camera

Most Basic Setup

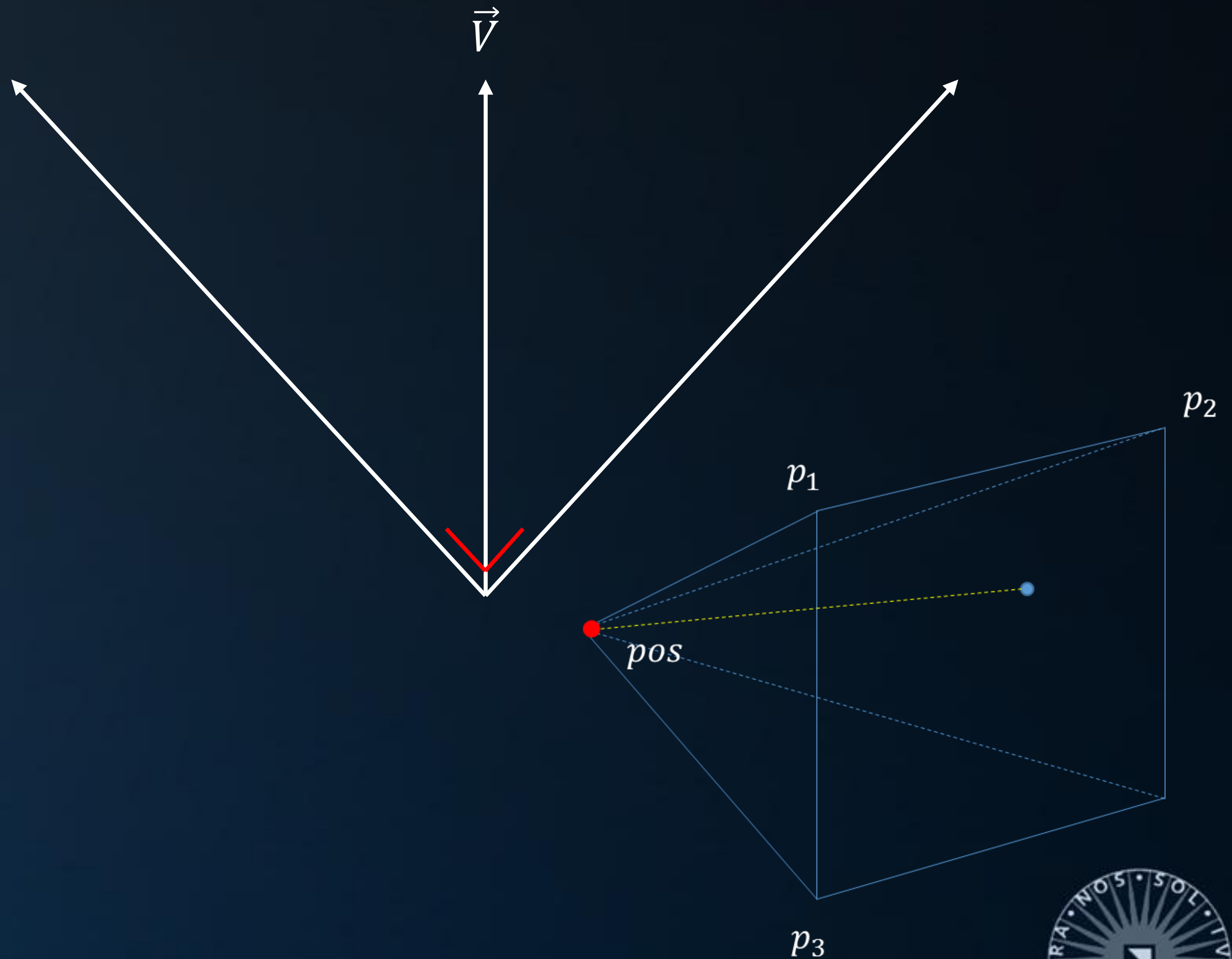
View direction:
`V = normalize( target – pos )`

Center of screen:
`C = pos + V`

Vectors to the left and right:
`(-1, 0, 0)` and `(1, 0, 0)`

Vectors up and down:
`(0, 1, 0)` and `(0, -1, 0)`

Screen corners:
```
p1 = C + left + up
p2 = C + right + up
p3 = C + left + down
```



Camera

Adding FOV

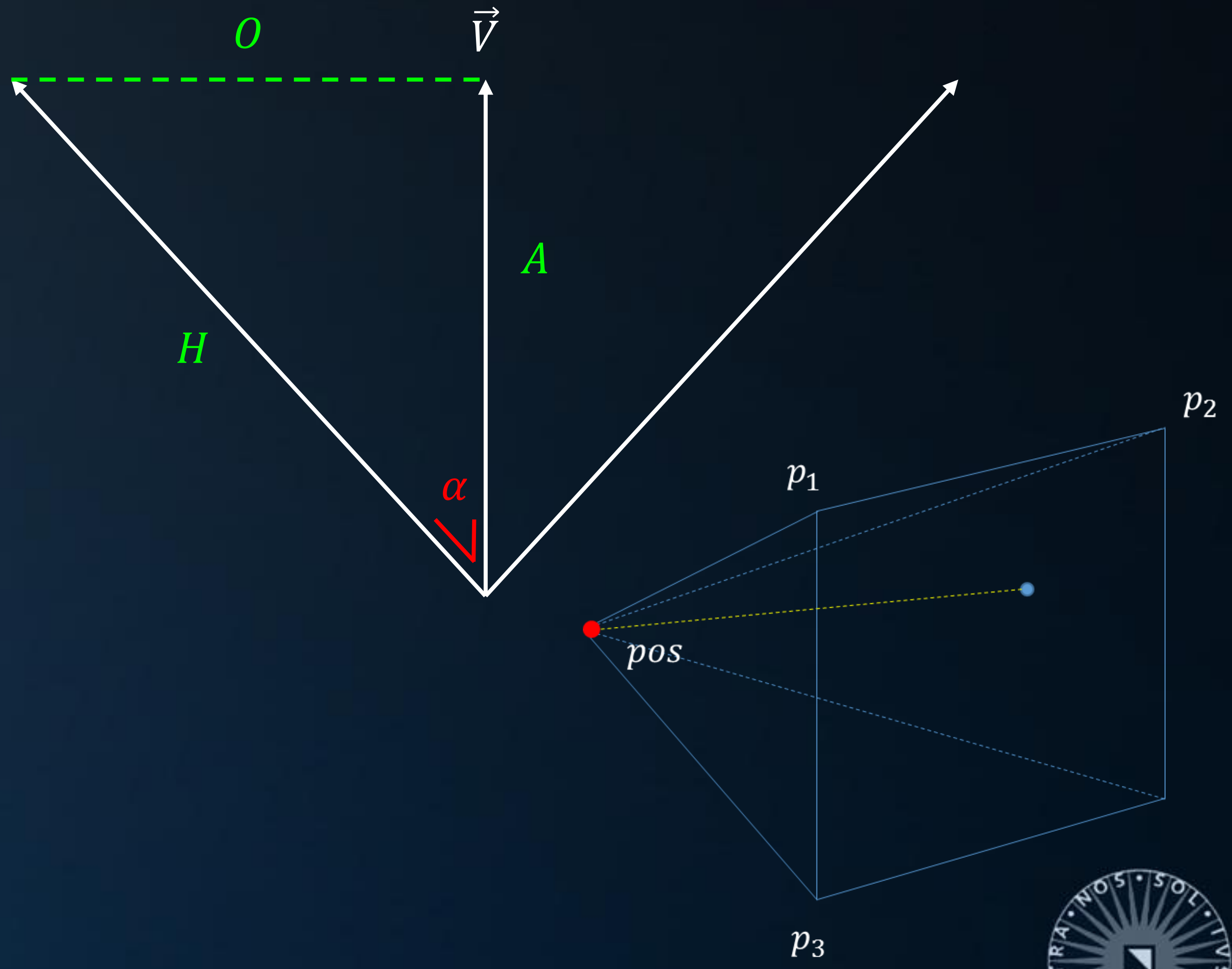
We know that:

$$O = 1$$

$$\alpha = \frac{1}{2} FOV$$

Using SOHCAHTOA:

$$\tan \alpha = \frac{O}{A} \rightarrow A = \frac{O}{\tan \alpha}$$



Camera

Arbitrary Direction

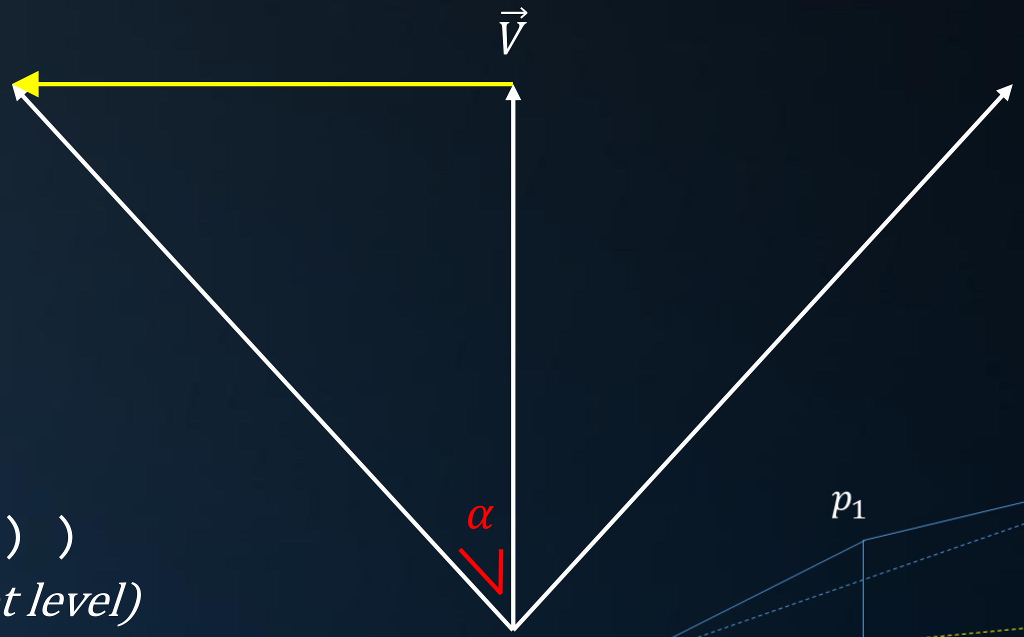
View direction:
`V = normalize( target – pos )`

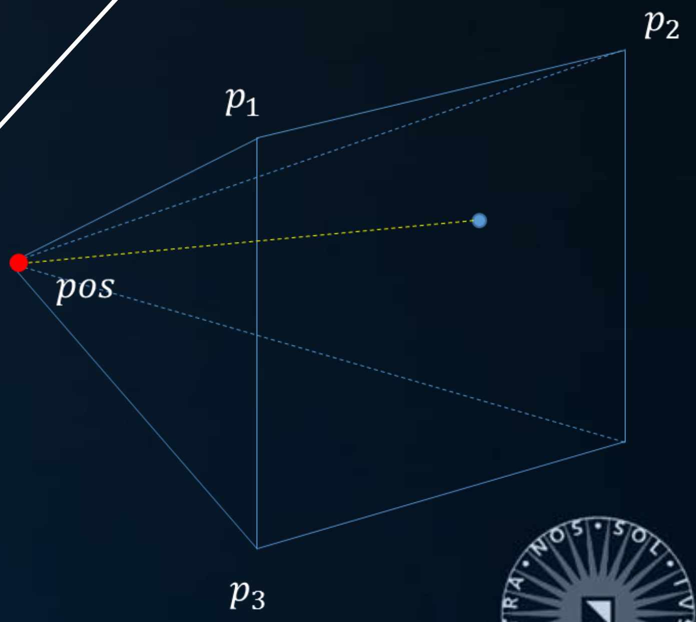
We know that
`up = (0,1,0)`

Therefore
`left = normalize( cross( V, up ) )`
(note that this is true even when $\vec{V}$ is not level)

`up = cross( V, left );`

Screen corners:
```
p1 = C + left + up
p2 = C + right + up
p3 = C + left + down
```





Camera

Aspect Ratio

Basic idea:

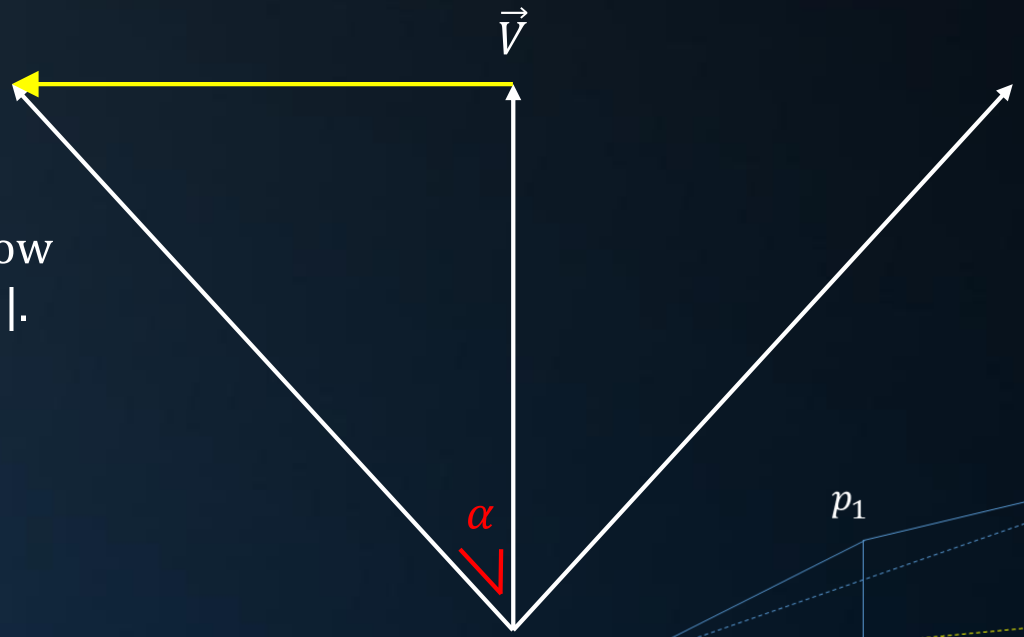
If your window width is 1.3x your window height, ||p2-p1|| should be 1.3x ||p3-p1||.

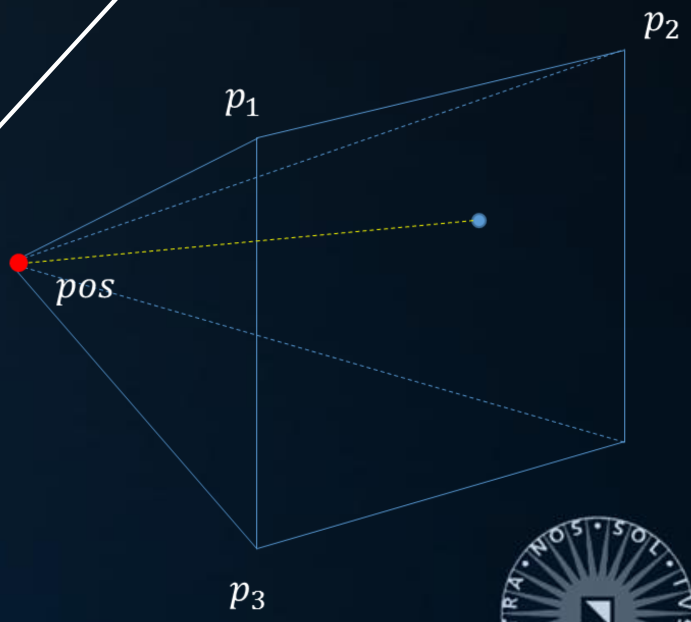
So:

```
aspectRatio = height / width;
```

Screen corners:

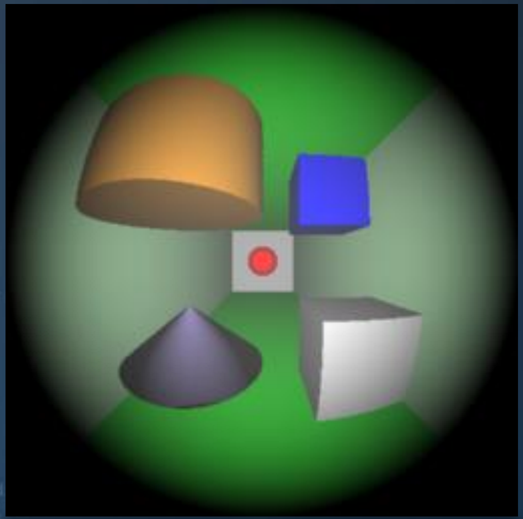
```
p1 = C + left + up * aspectRatio
p2 = C + right + up * aspectRatio
p3 = C + left + down * aspectRatio
```

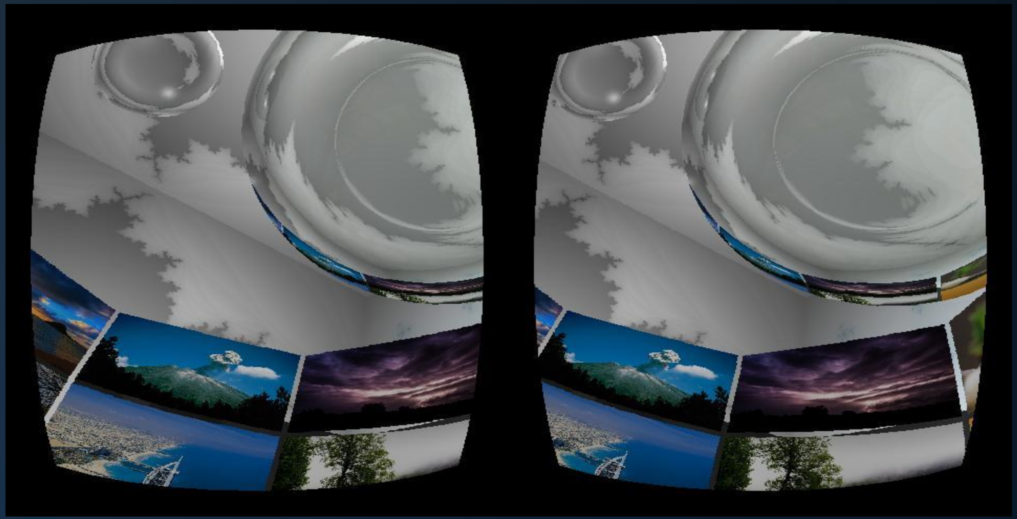




Camera

Fisheye Lens





See:
paulbourke.net/dome/fisheye

See:
rifty-business.blogspot.nl/2013/08/understanding-oculus-rift-distortion.html

Camera

Rotating the Camera

Rotation, "The Hard Way":

- Setup a camera matrix
- Apply this matrix to the screen center.

Camera

Rotating the Camera

Rotate right, “The Easy Way”:

```
target = pos + V;
target += C * right;
V = normalize( target – pos );
```

Rotate up:

```
target = pos + V;
target += C * up;
V = normalize( target – pos );
```

Camera

Framerate

You can make your camera rotate at a constant speed by factoring in frame time.

1. Measure the time (in milliseconds) it takes to render a frame
2. For the next frame, multiply movement by this number

Today's Agenda:

- Recap
- End of the Primary Ray
- Normals
- The Camera
- Assignment P2

Assignment P2

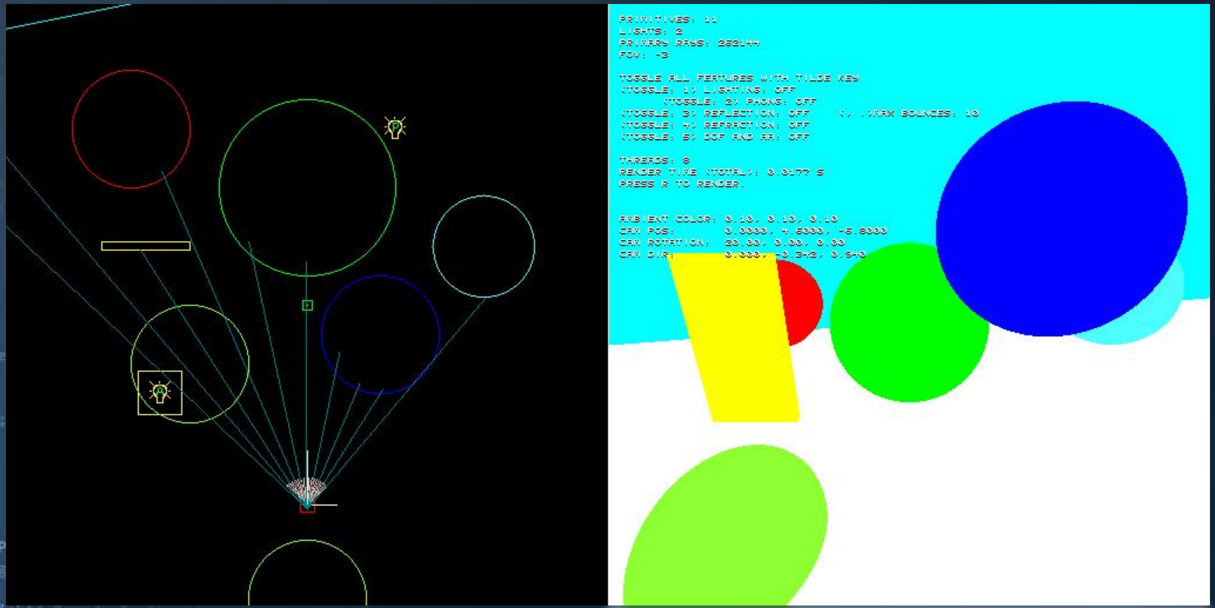
New Challenge

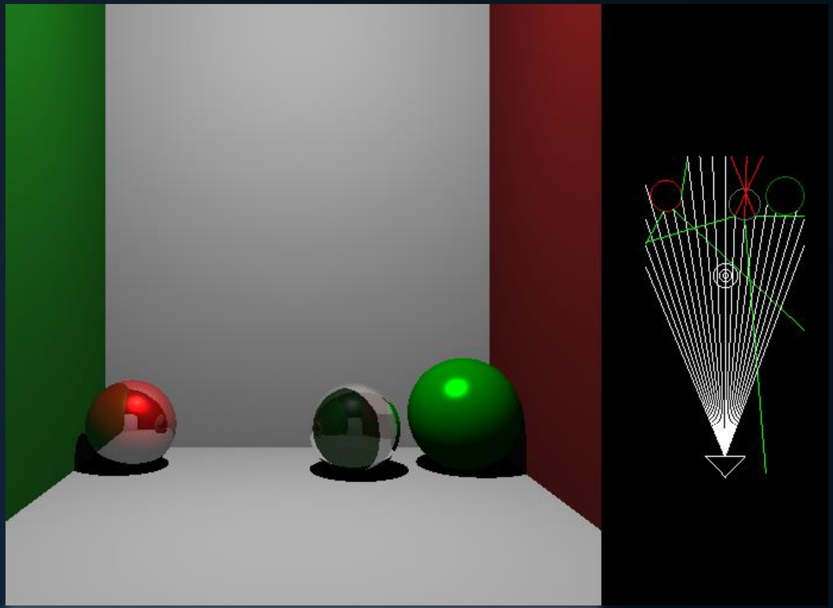
Smallest executable.

1k-sw-raytrace'em all by Tristar & Red Sector Inc. (2004)

Assignment P2

Use That Debug Output!





INFOGR – Computer Graphics

Jacco Bikker - April-July 2016 - Lecture 4: “Ray Tracing (2)”

END of “Ray Tracing (Part 2)”

next lecture: “Ray Tracing (Part 3)”