INFOGR – Computer Graphics

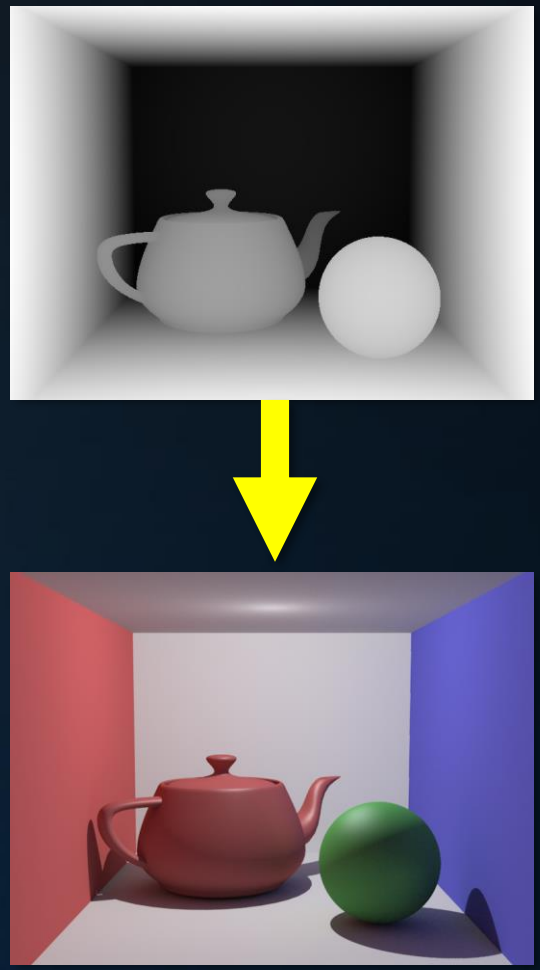
Jacco Bikker & Debabrata Panja - April-July 2019

Lecture 6: “Ray Tracing”

Welcome!

Today’s Agenda:

- Ray Tracing Recap
- Shading
- Textures



Ray Tracing

Ray Tracing:

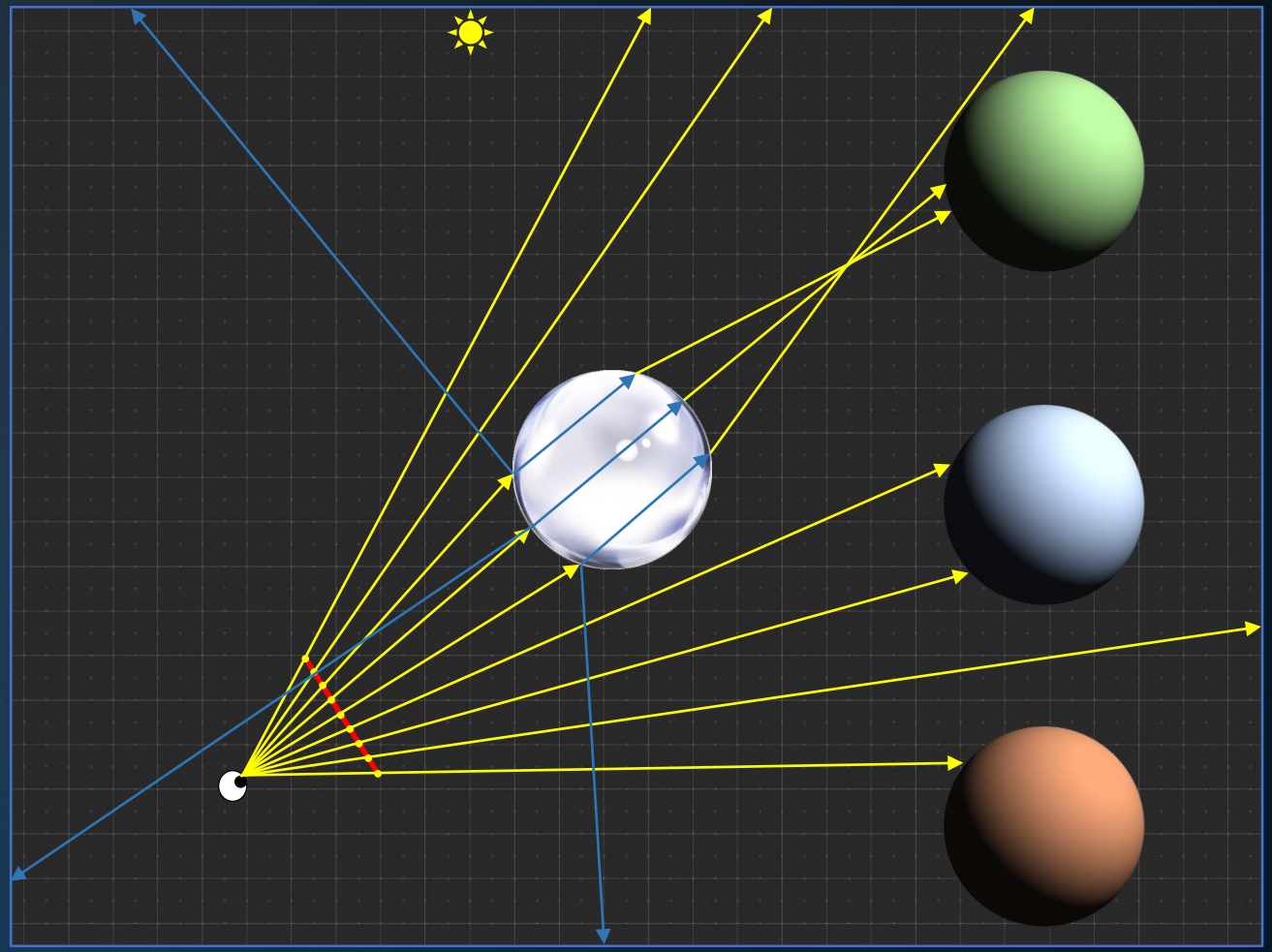
World space

- Geometry
- Eye
- Screen plane
- Screen pixels
- Primary rays
- Intersections
- Point light
- Shadow rays

Light transport

- Extension rays

Light transport



Recap

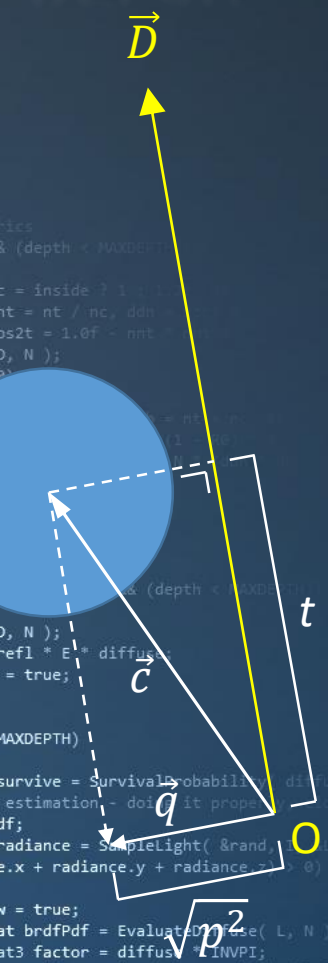
Ray definition

A ray is an infinite line with a start point:

$p(t) = O + t\vec{D}$, where $t > 0$.

```
struct Ray
{
    float3 O;    // ray origin
    float3 D;    // ray direction
    float t;     // distance
};
```

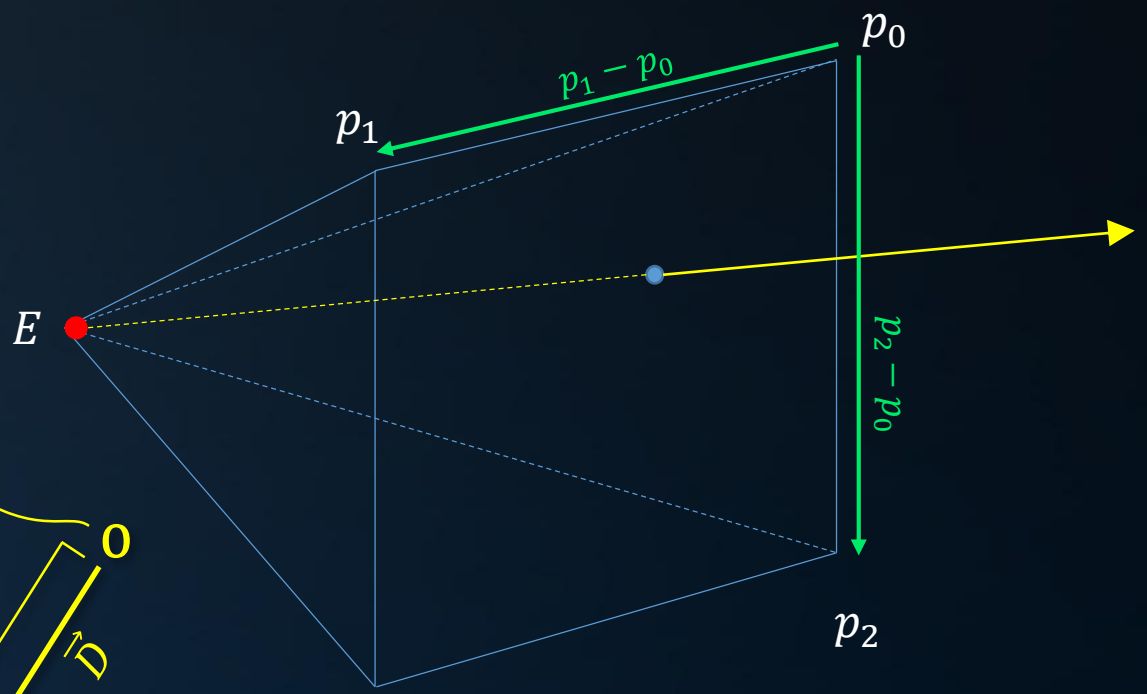
The ray direction is generally *normalized*.

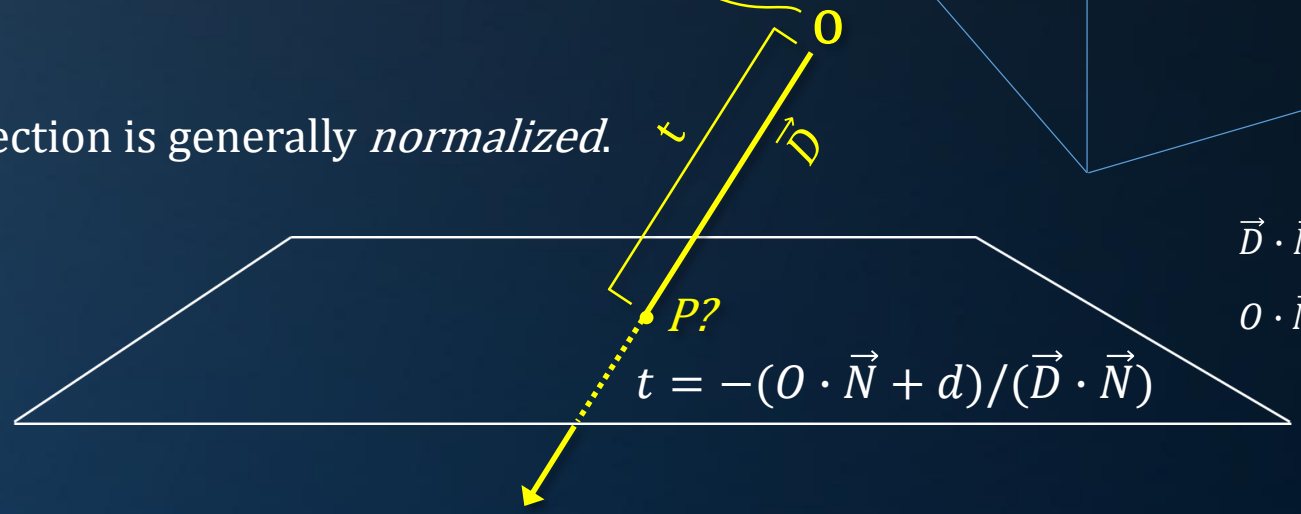


Ray setup

Point on the screen: $p(u,v) = p_0 + u(p_1 - p_0) + v(p_2 - p_0), \quad u,v \in [0,1)$

Ray direction (before normalization): $\vec{D} = p(u,v) - E$



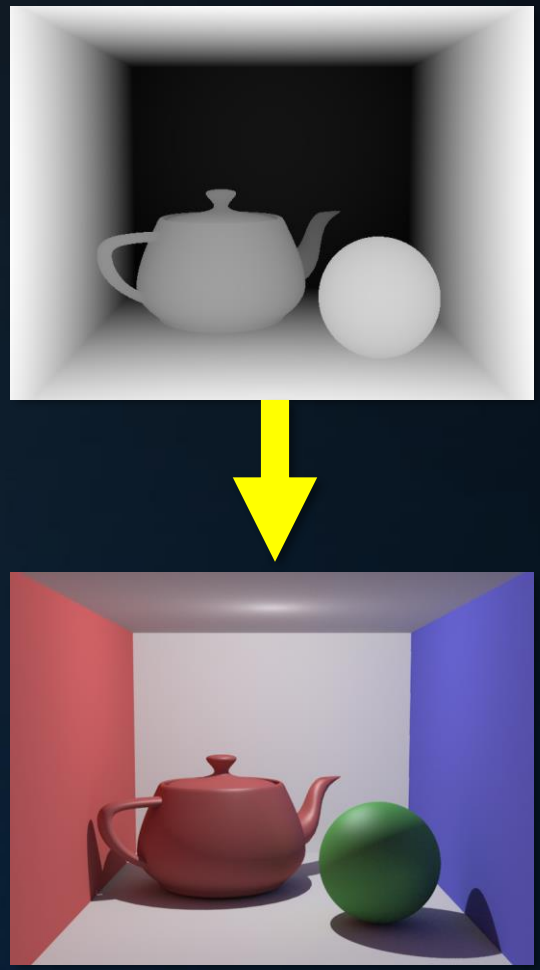


$\vec{D} \cdot \vec{N} = \vec{D}_x\vec{N}_x + \vec{D}_y\vec{N}_y + \vec{D}_z\vec{N}_z$

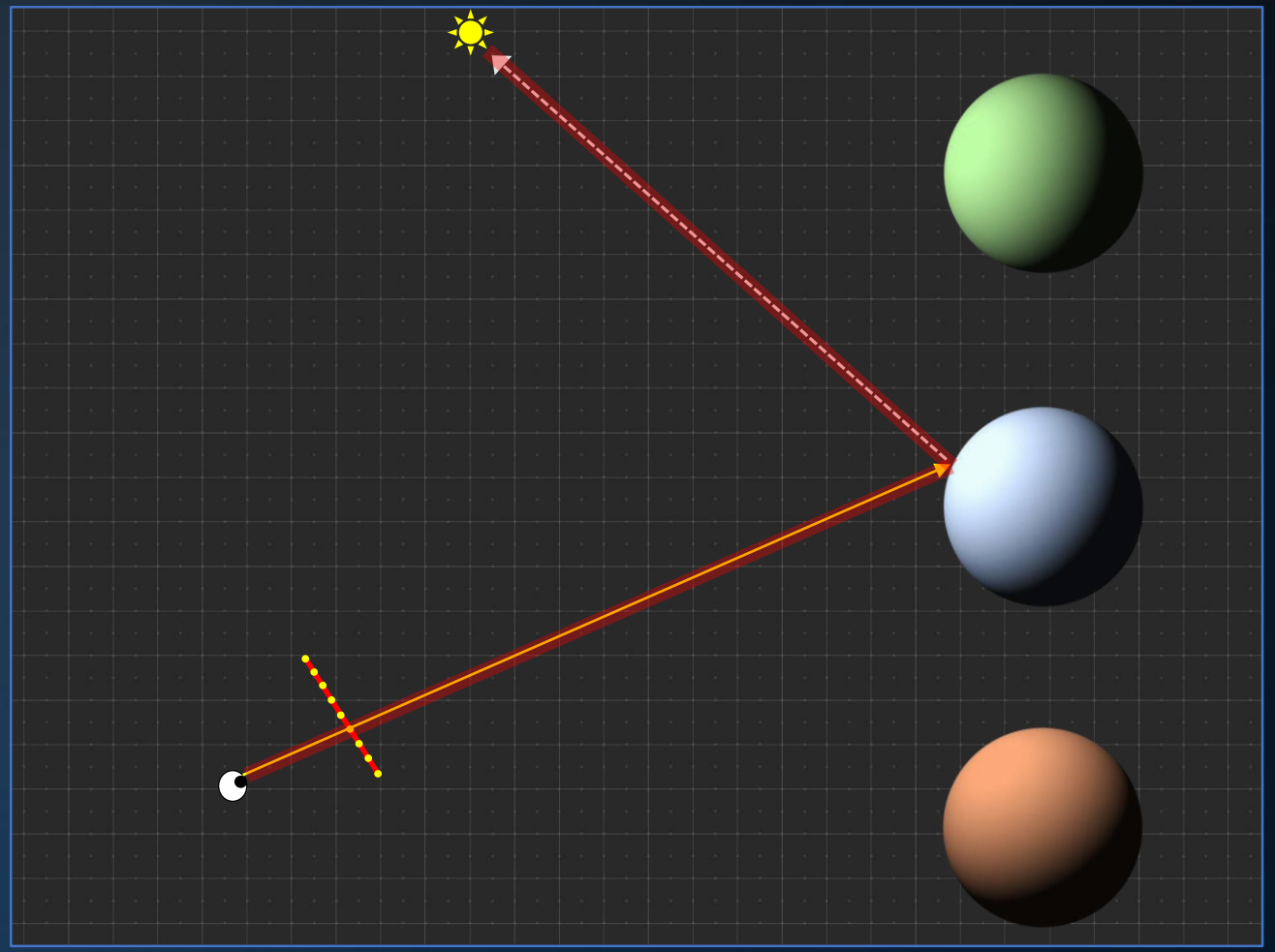
$O \cdot \vec{N} = O_x\vec{N}_x + O_y\vec{N}_y + O_z\vec{N}_z$

Today’s Agenda:

- Ray Tracing Recap
- Shading
- Textures



Shading



Shading

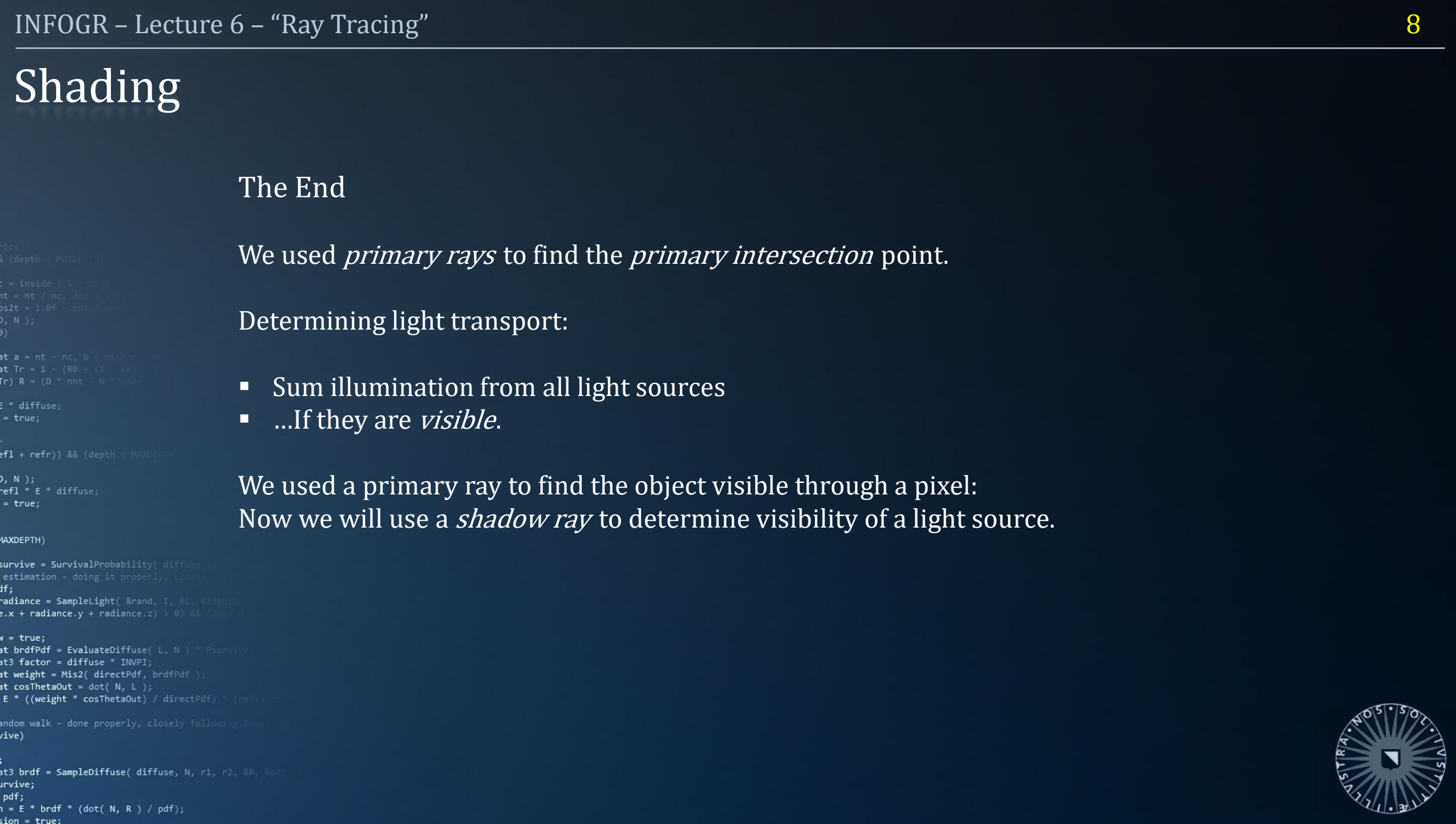
The End

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

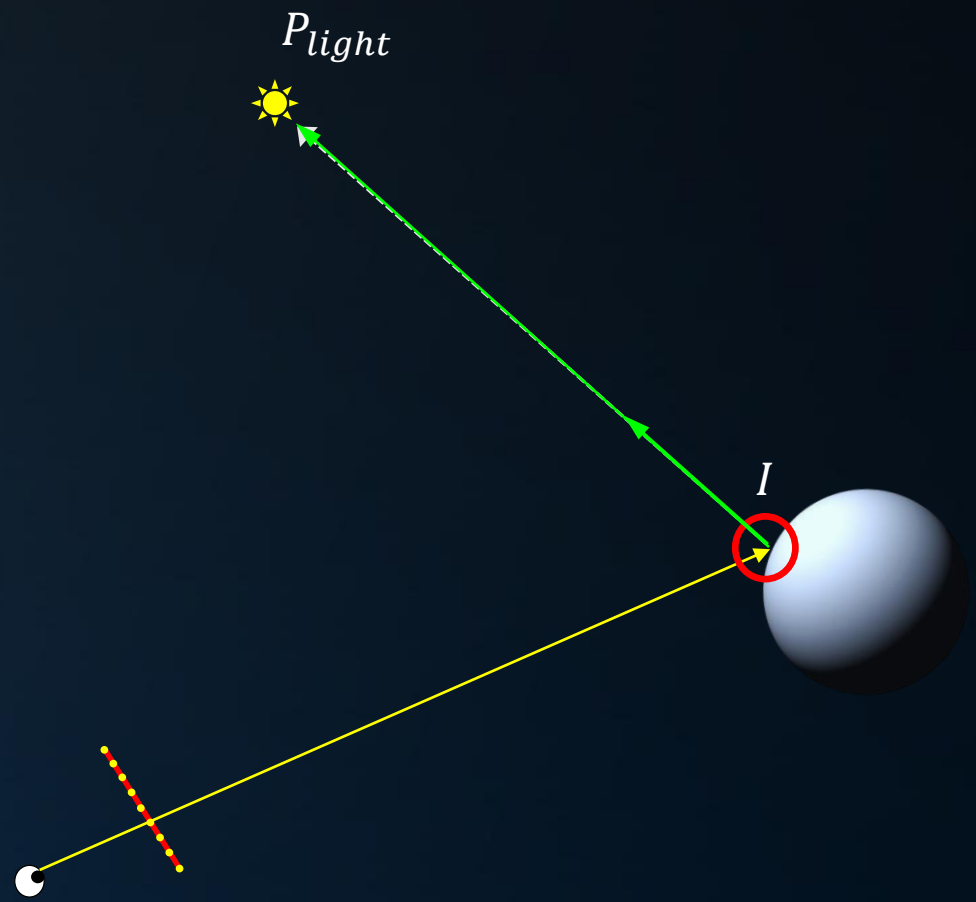
Constructing the shadow ray:

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

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: $\frac{P_{light}-I}{\|P_{light}-I\|}$

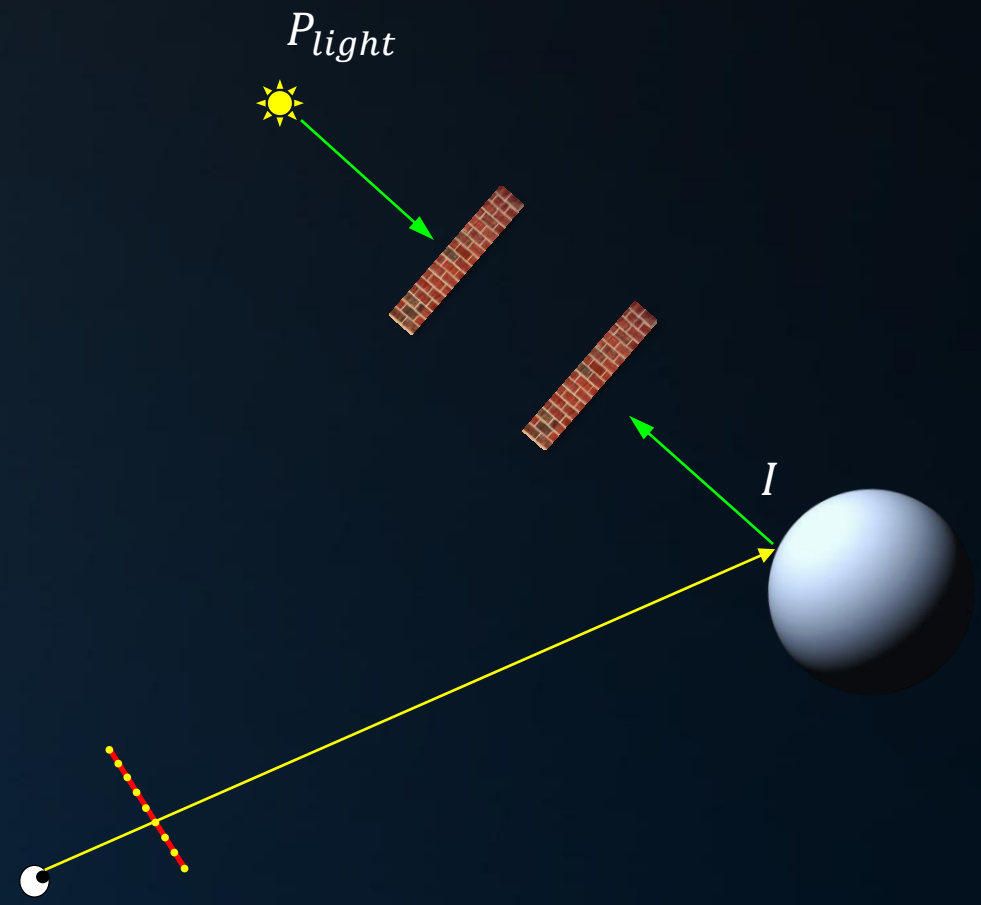
Equally valid: $\frac{I-P_{light}}{\|P_{light}-I\|}$ or $\frac{I-P_{light}}{\|I-P_{light}\|}$

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.”

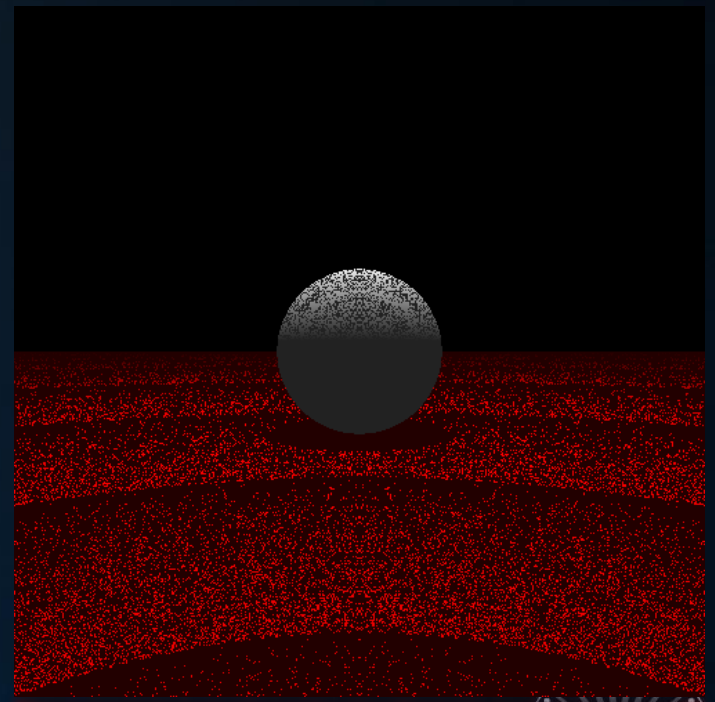
Problem 1:

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 $2 * epsilon$.

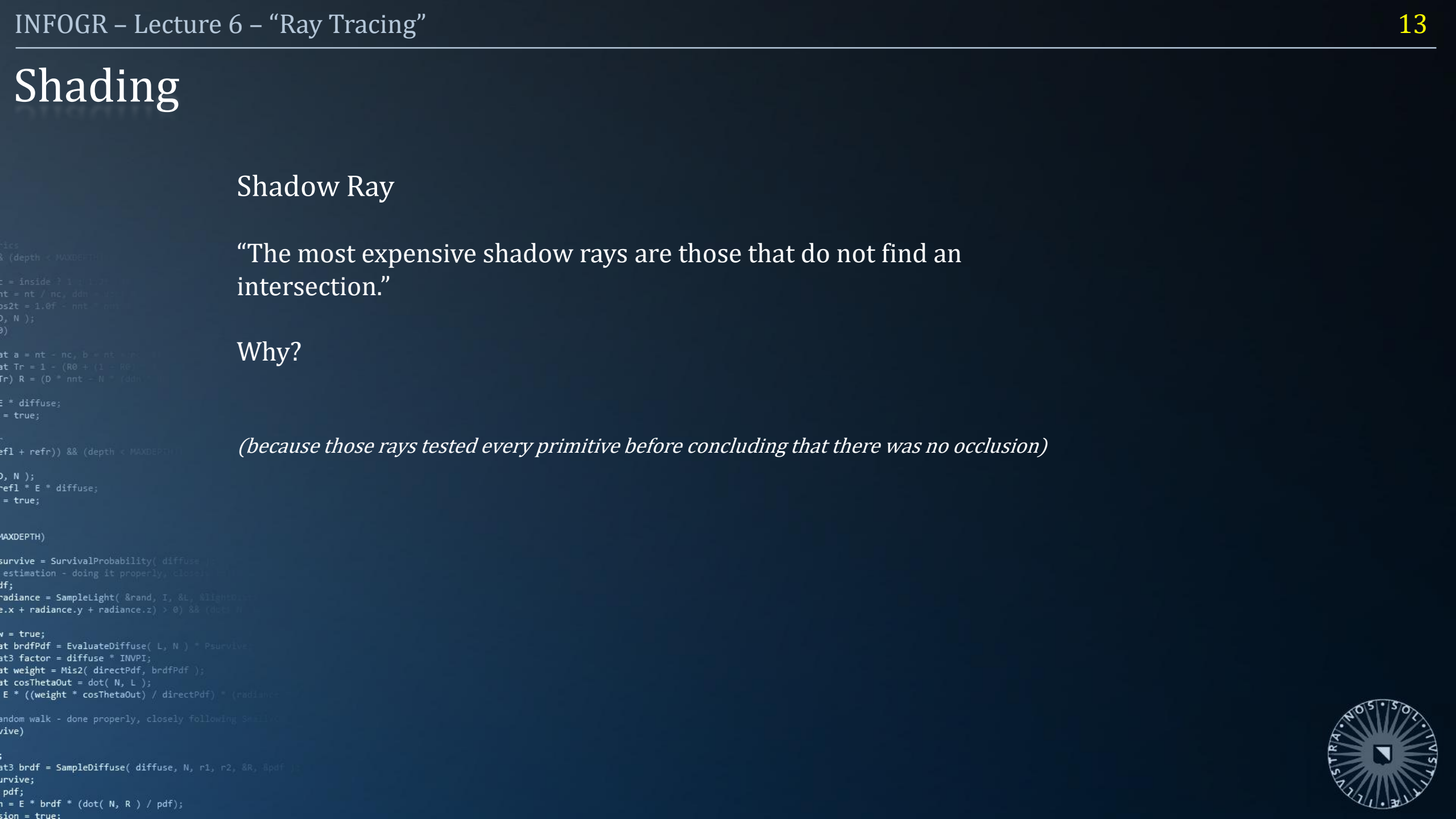
Shading

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)

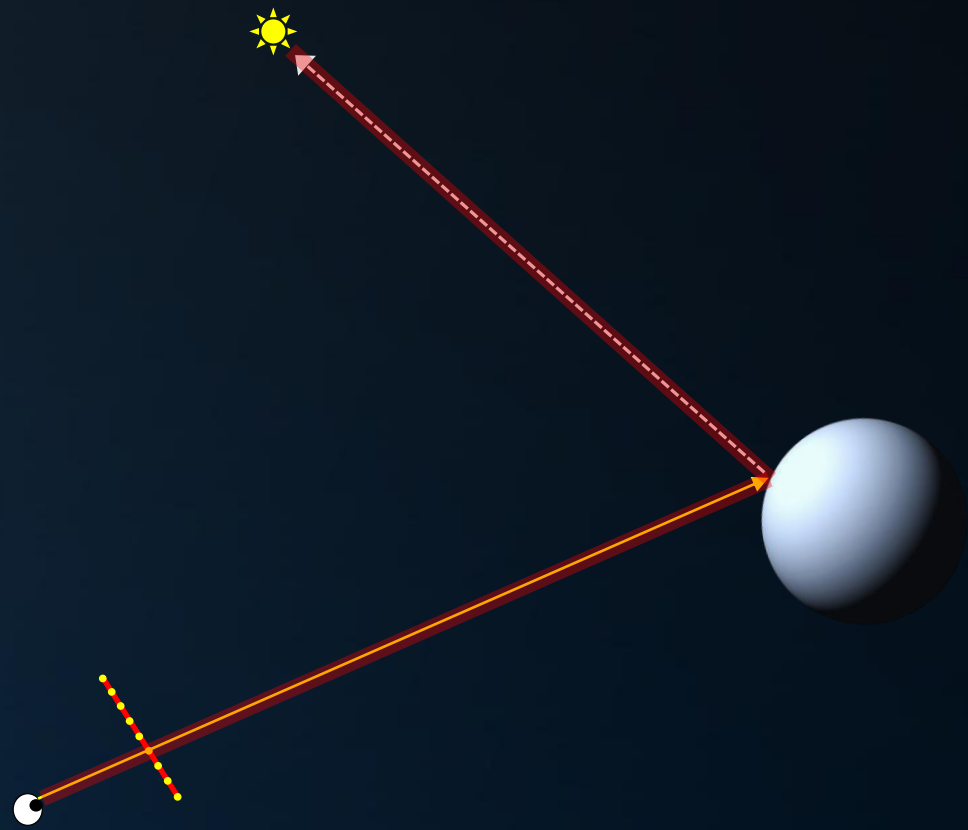


Shading

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

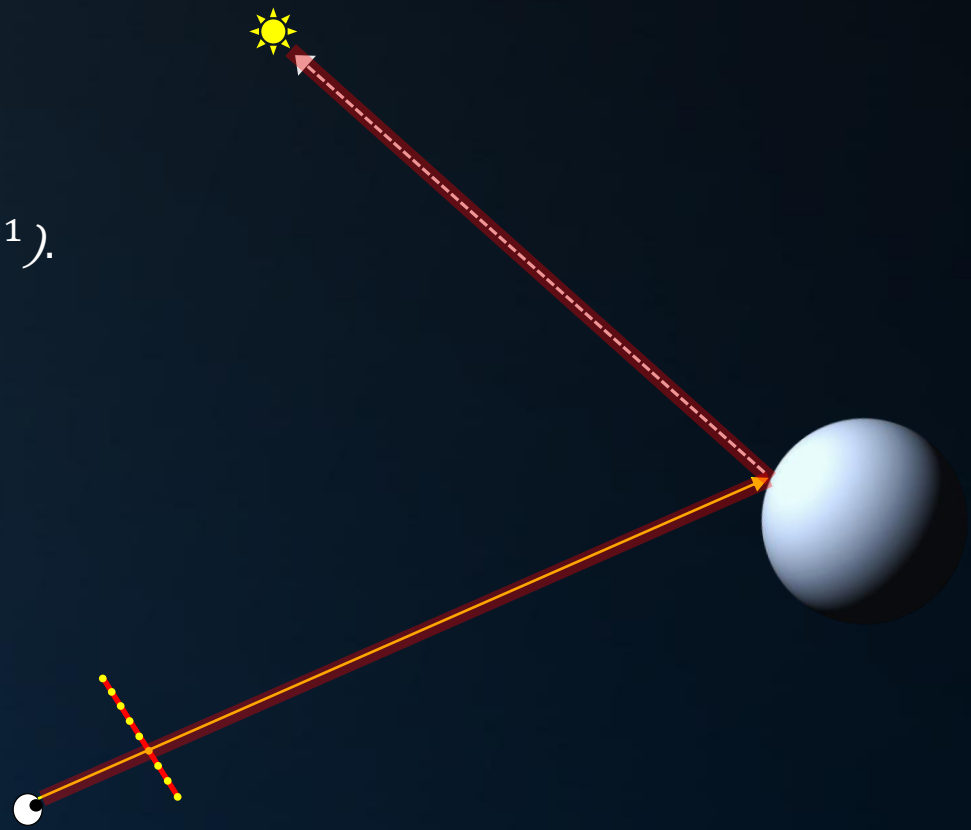
Brightness of the light source:

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 $\sim 3.0 \cdot 10^{21}$ photons per second.



Shading

Transport

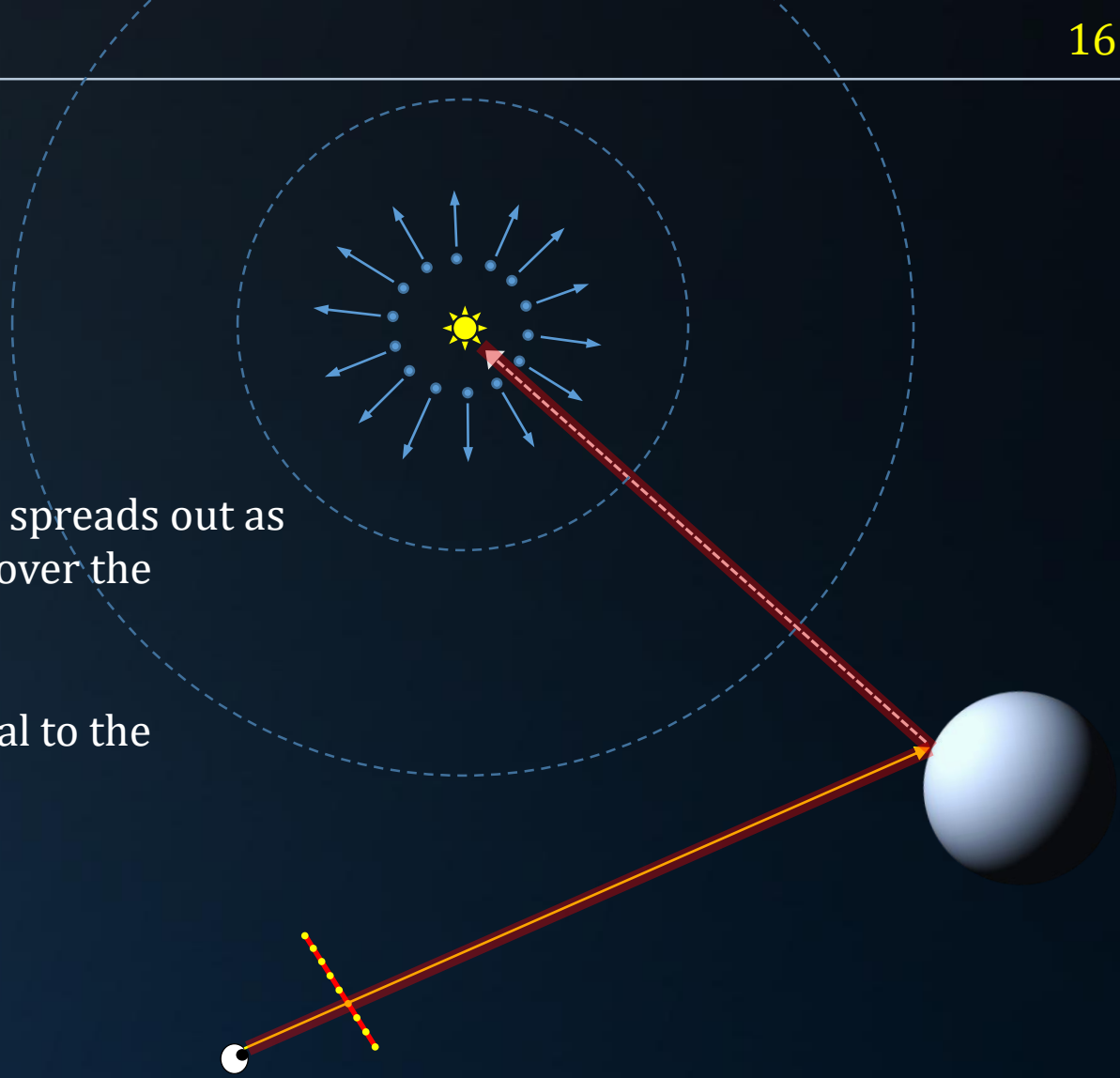
Energy at distance r:

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.

Energy per unit area 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.

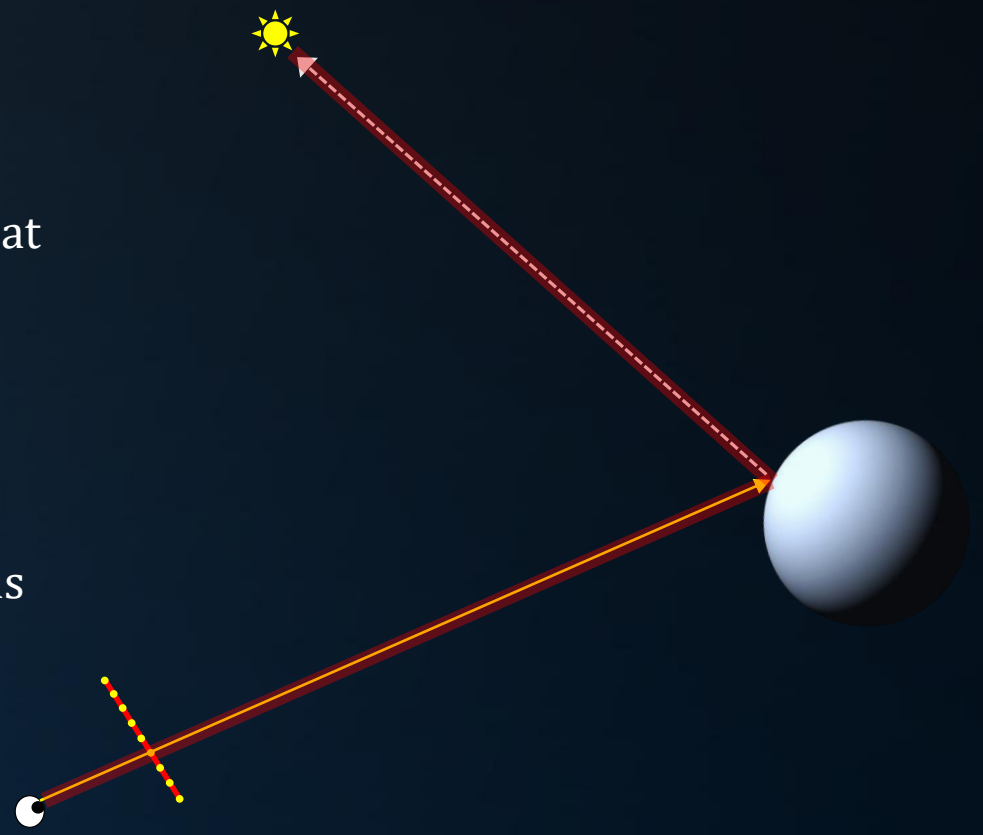
The reflected light is thus:

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

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

$C_{material}$ is typically a vector: we store r, g, b for all light transport.

$$A \circ B = \begin{bmatrix} A_x B_x \\ A_y C_y \\ A_z C_z \end{bmatrix} \qquad \textit{(‘entrywise product’)}$$



Shading

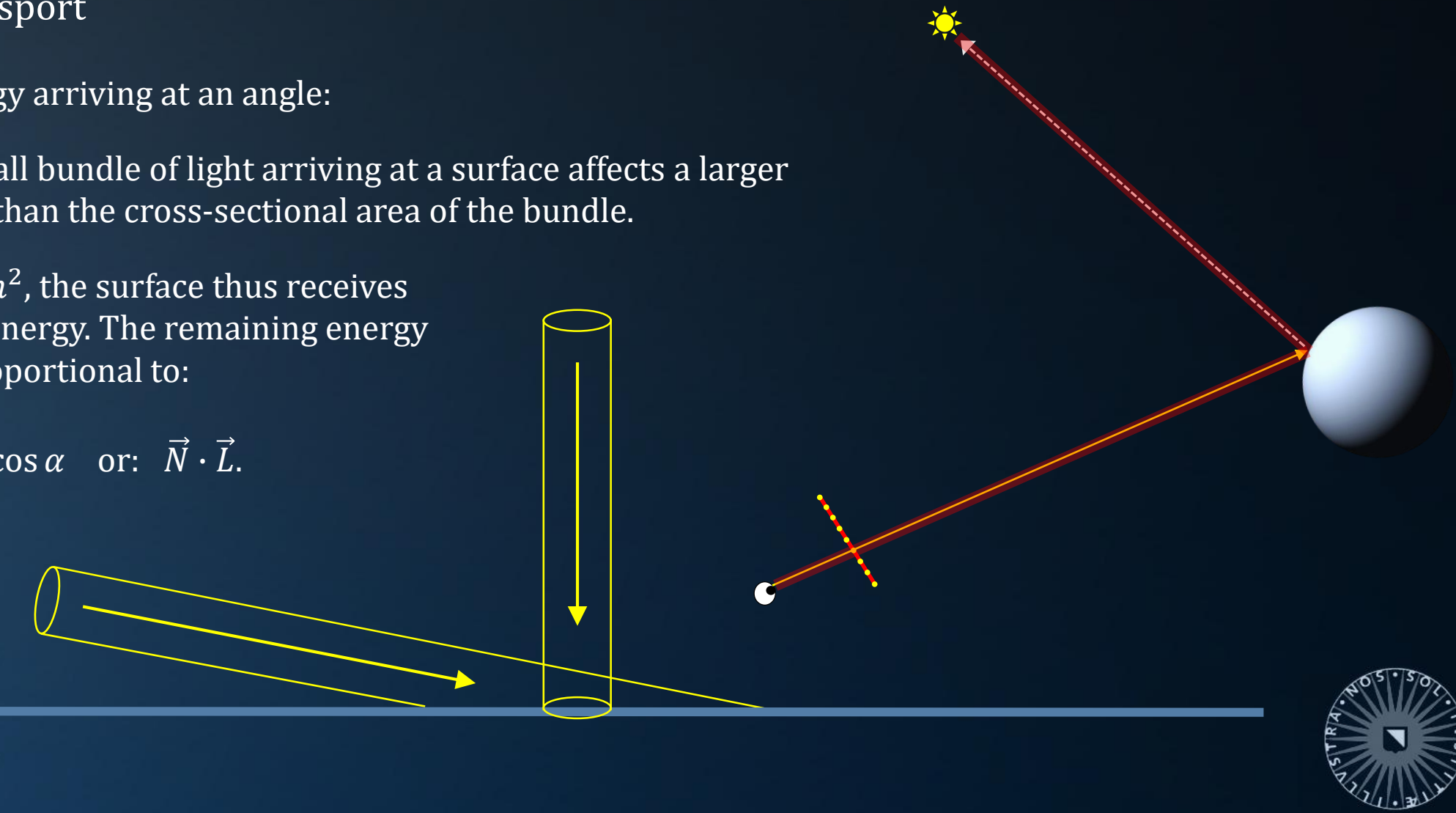
Transport

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:

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

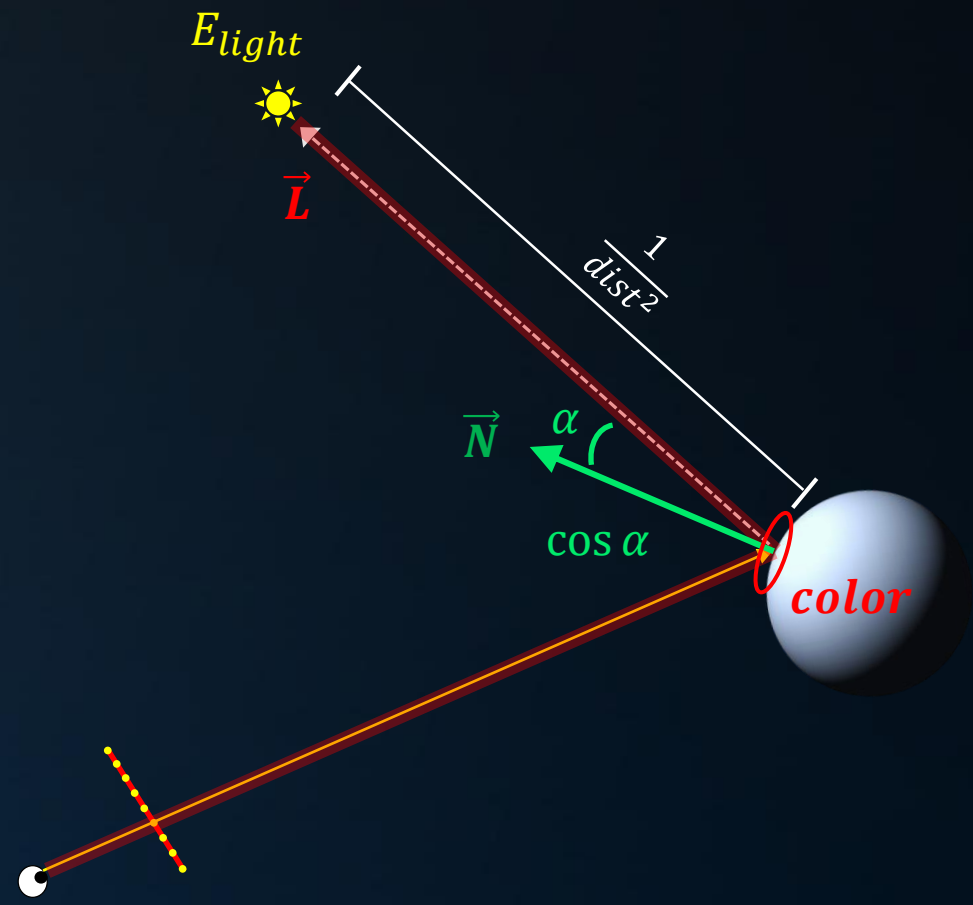


Shading

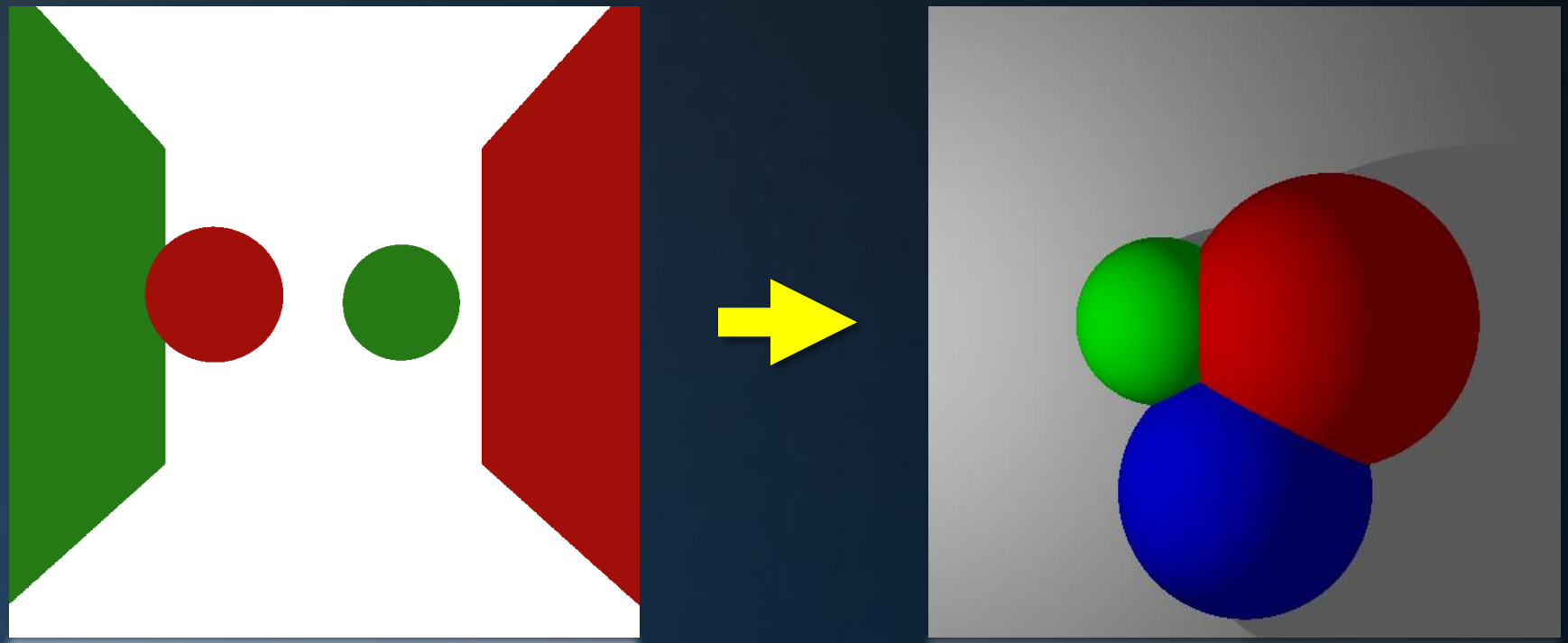
Transport

All factors:

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

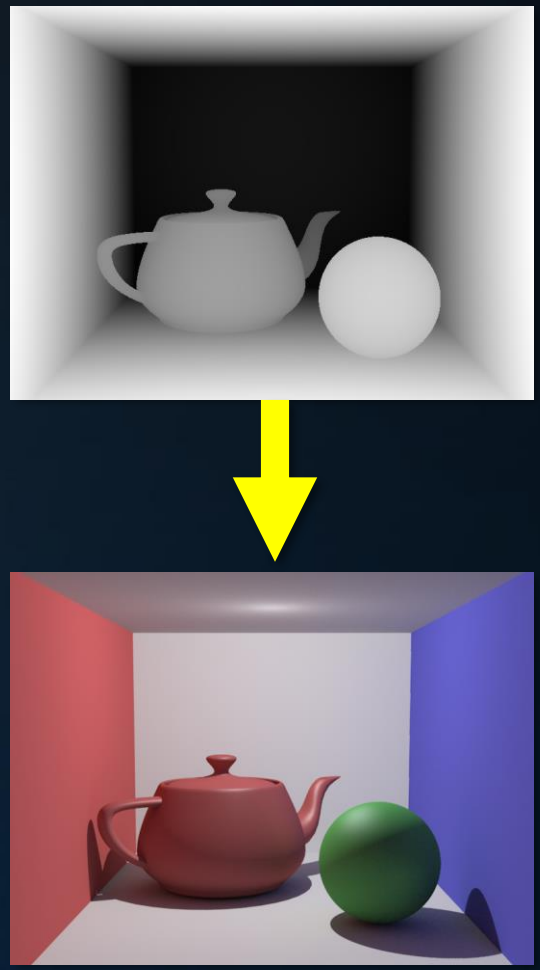


Shading



Today’s Agenda:

- Ray Tracing Recap
- Shading
- Textures



Textures

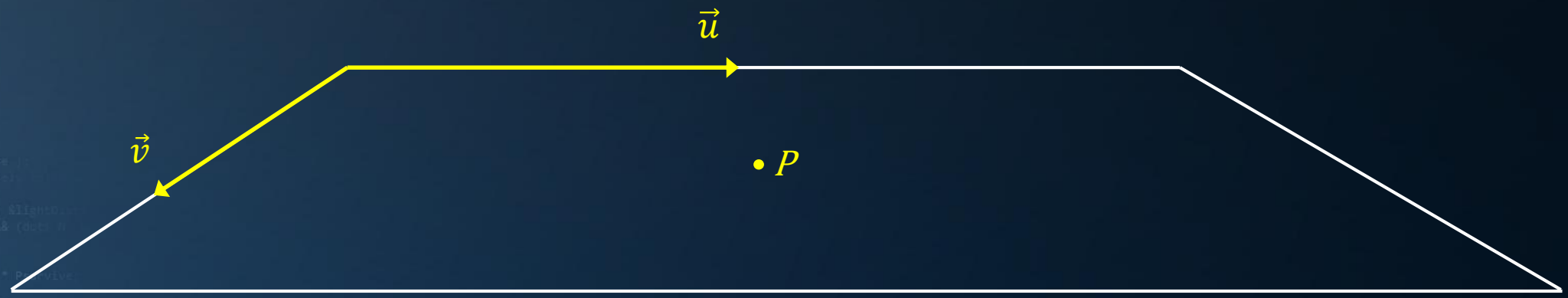
Texturing a Plane

Given a plane: $y = 0$ (i.e., with a normal vector $(0,1,0)$).

Two vectors on the plane define a basis: $\vec{u} = (1,0,0)$ and $\vec{v} = (0,0,1)$.

Using these vectors, any point on the plane can be reached: $P = \lambda_1\vec{u} + \lambda_2\vec{v}$.

We can now use λ_1, λ_2 to define a color at P: $F(\lambda_1, \lambda_2) = \cdots$.



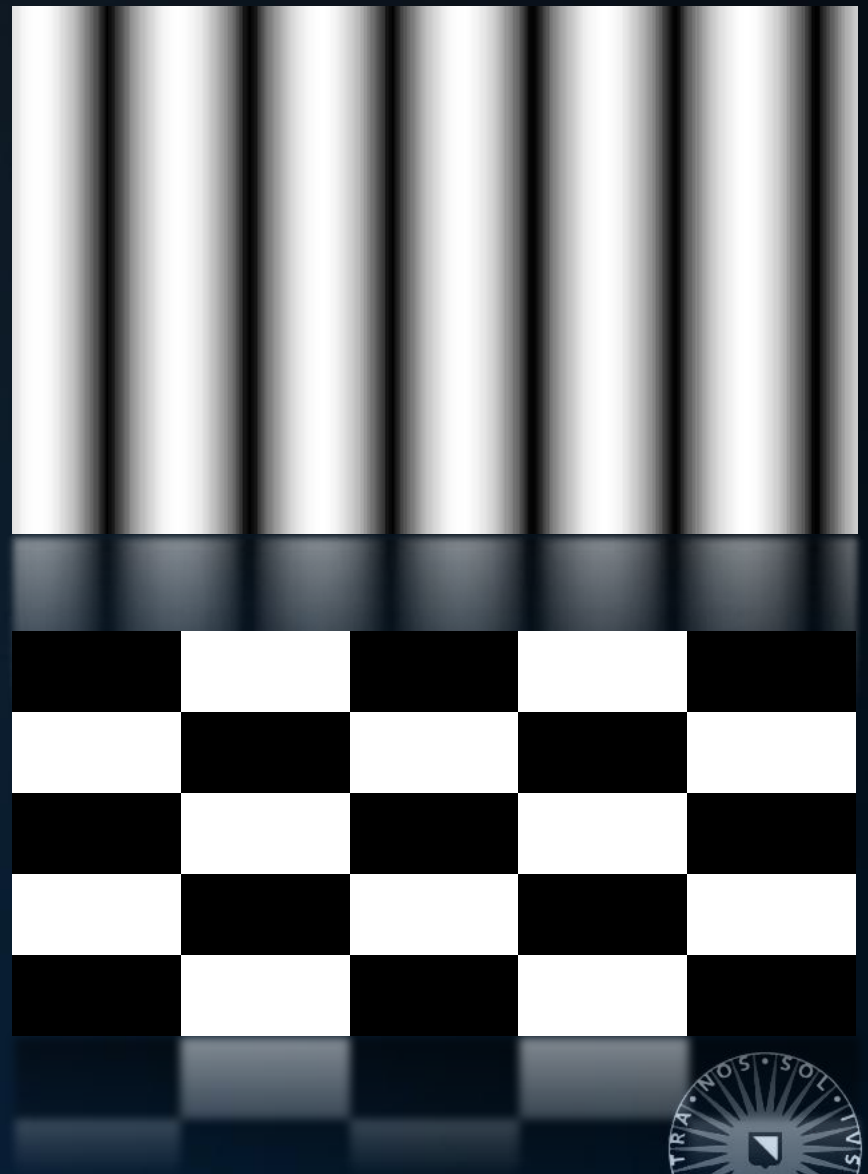
Textures

Example:

$$F(\lambda_1, \lambda_2) = \sin(\lambda_1)$$

Another example:

$$F(\lambda_1, \lambda_2) = ((\text{int})(2 * \lambda_1) + (\text{int})\lambda_2) \,\&\, 1$$

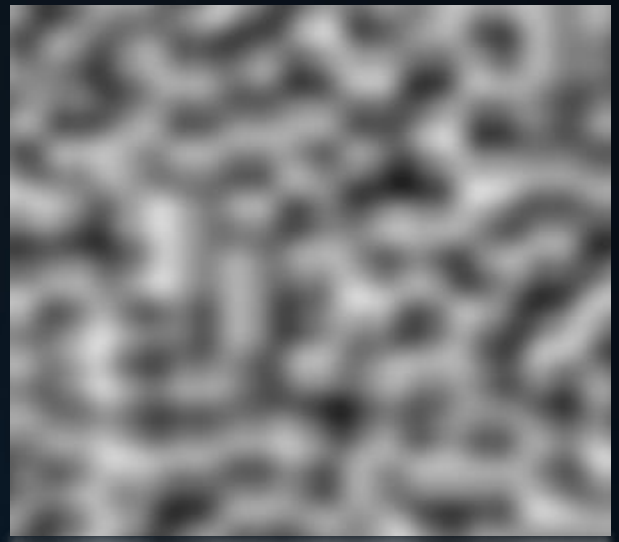


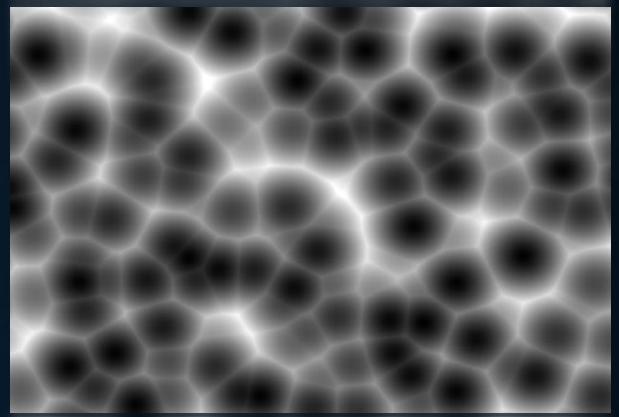
Textures

Other examples (not explained here):

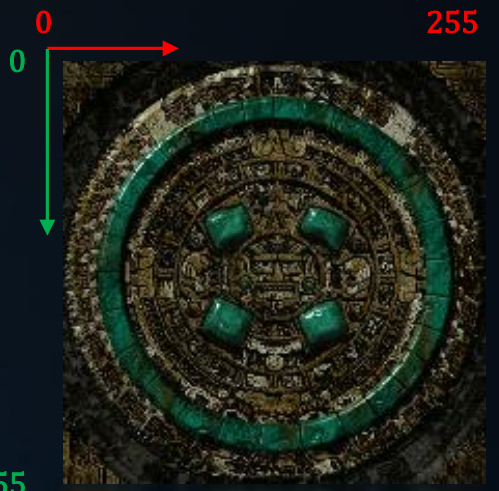
Perlin noise
Details: http://www.noisemachine.com/talk1

Voronoi / Worley noise
Details: “A cellular texture basis function”, S. Worley, 1996.





Textures

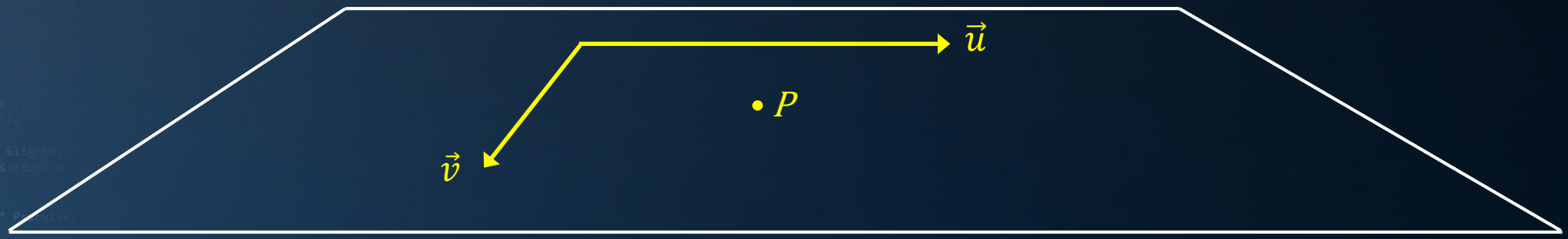


Obviously, not all textures can be generated procedurally.

Finding P using basis vectors $\vec{u}$, $\vec{v}$ and parameters λ_1, λ_2:

$$\begin{pmatrix} P_x \\ P_y \end{pmatrix} = \lambda_1 \vec{u} + \lambda_2 \vec{v} = \begin{pmatrix} \lambda_1 \vec{u}_x + \lambda_2 \vec{v}_x \\ \lambda_1 \vec{u}_y + \lambda_2 \vec{v}_y \end{pmatrix}$$

...But what about the opposite, i.e. can we find λ_1, λ_2 given P?



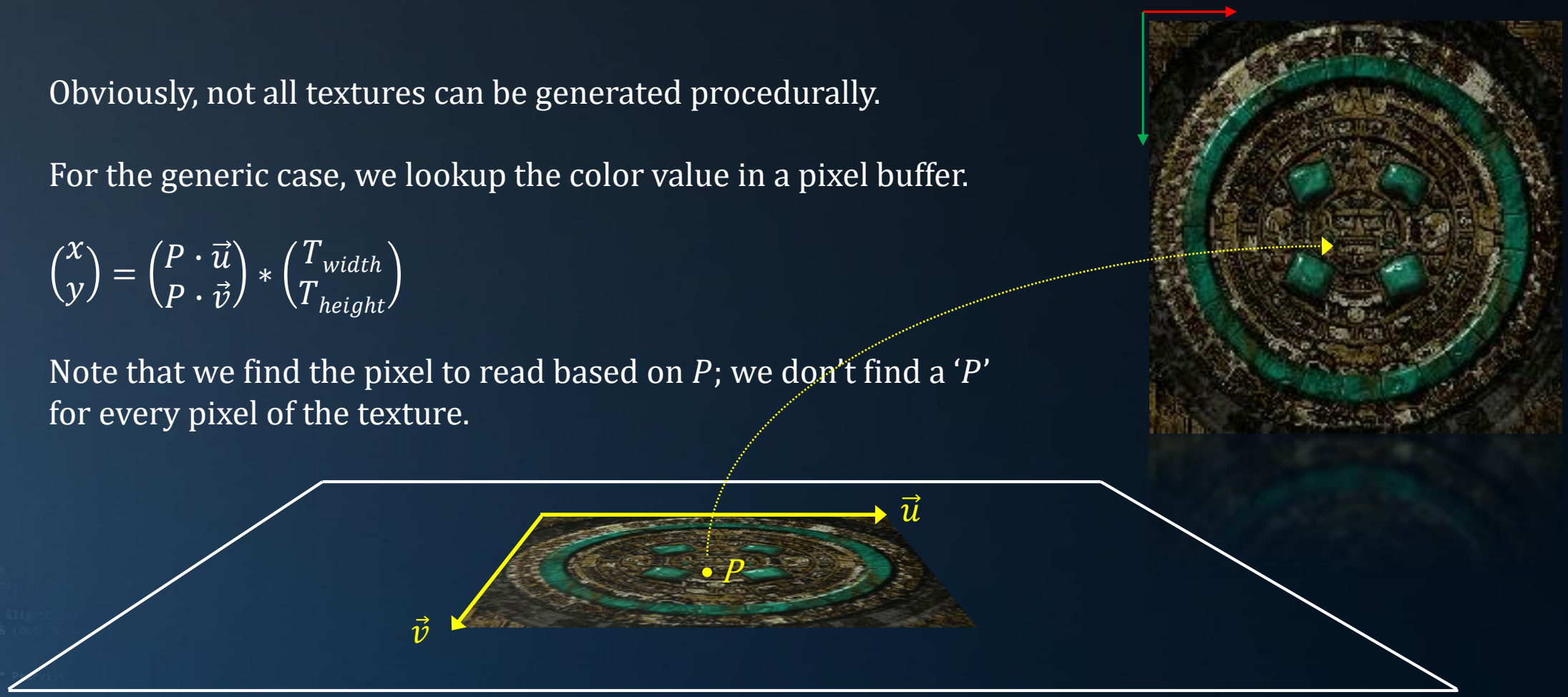
Textures

Obviously, not all textures can be generated procedurally.

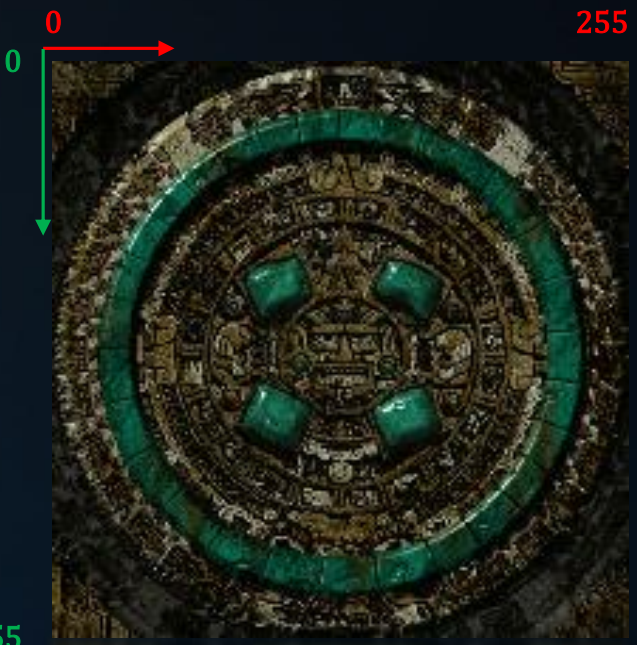
For the generic case, we lookup the color value in a pixel buffer.

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} P \cdot \vec{u} \\ P \cdot \vec{v} \end{pmatrix} * \begin{pmatrix} T_{width} \\ T_{height} \end{pmatrix}$$

Note that we find the pixel to read based on P; we don’t find a ‘P’ for every pixel of the texture.



Textures



Retrieving a pixel from a texture:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} P \cdot \vec{u} \\ P \cdot \vec{v} \end{pmatrix} * \begin{pmatrix} T_{width} \\ T_{height} \end{pmatrix}$$

We don’t want to read outside the texture. To prevent this, we have two options:

1. Clamping

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} clamp(P \cdot \vec{u}, 0, 1) \\ clamp(P \cdot \vec{v}, 0,1) \end{pmatrix} * \begin{pmatrix} T_{width} \\ T_{height} \end{pmatrix}$$

2. Tiling

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} frac(P \cdot \vec{u}) \\ frac(P \cdot \vec{v}) \end{pmatrix} * \begin{pmatrix} T_{width} \\ T_{height} \end{pmatrix}$$

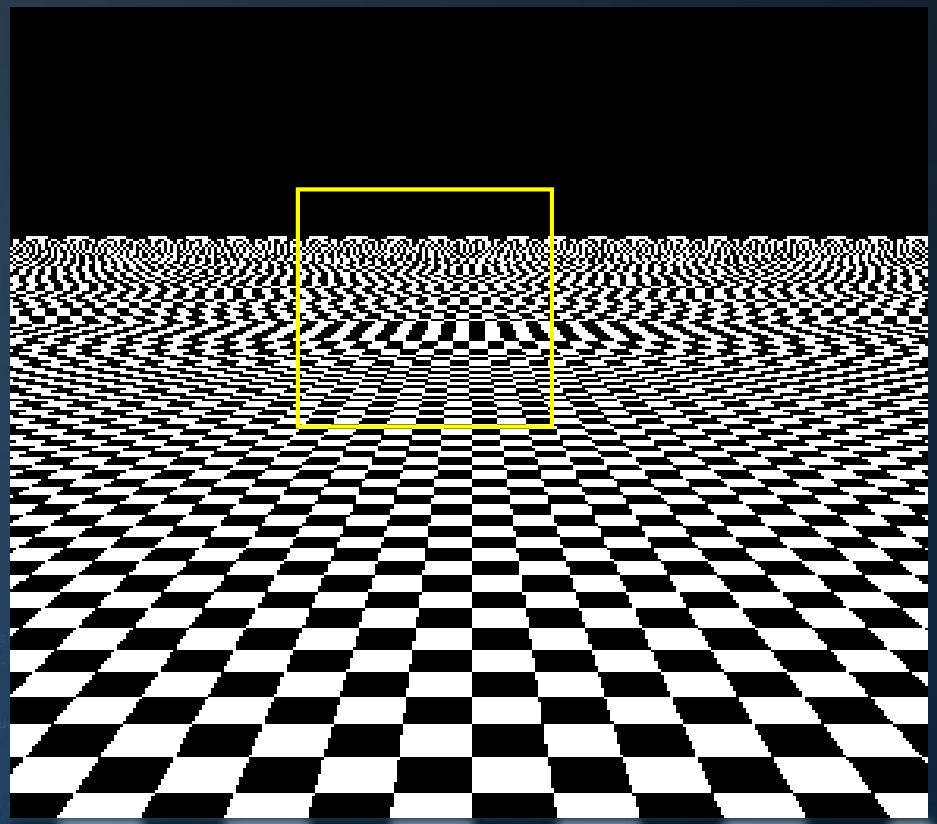
Tiling is efficiently achieved using a bitmask. This requires texture dimensions that are a power of 2.

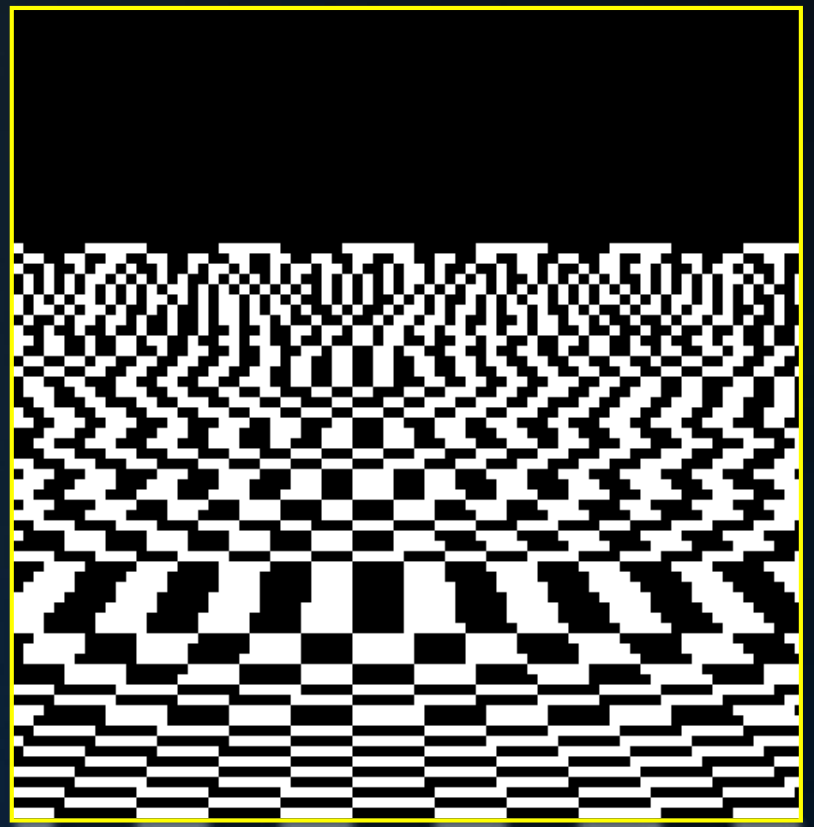
Textures

Texture mapping: oversampling

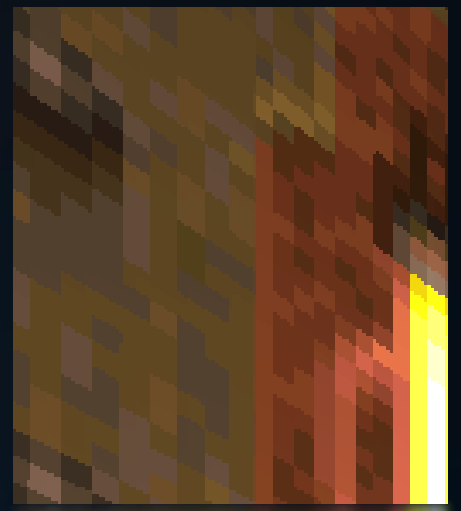
Textures

Texture mapping: undersampling





Textures

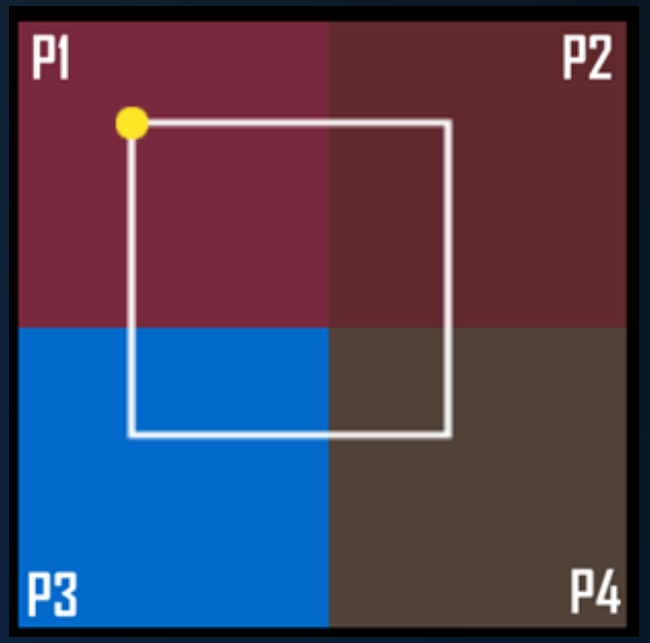


Fixing oversampling

Oversampling: reading the same pixel from a texture multiple times.
Symptoms: blocky textures.

Remedy: bilinear interpolation:
Instead of clamping the pixel location to the nearest pixel, we read from four pixels.

$$w_{p1}: (1 - frac(x)) * (1 - frac(y))$$
$$w_{p2}: frac(x) * (1 - frac(y))$$
$$w_{p3}: (1 - frac(x)) * frac(y)$$
$$w_{p4}: 1 - (wP_1 + wP_2 + wP_3)$$



Textures

Fixing oversampling

Textures

Fixing undersampling

Undersampling: skipping pixels while reading from a texture.
Symptoms: Moiré, flickering, noise.

Remedy: MIP-mapping.

The texture is reduced to 25% by averaging 2x2 pixels. This is repeated until a 1x1 image remains.

When undersampling occurs, we switch to the next MIP level.


NO MIPMAPS

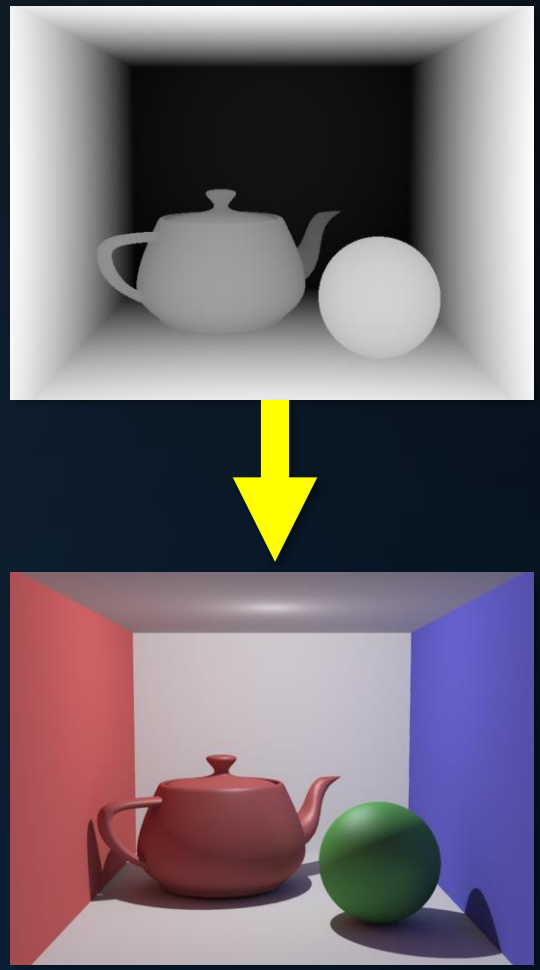

WITH MIPMAPS

Textures

Trilinear interpolation: blending between MIP levels.

Today’s Agenda:

- Ray Tracing Recap
- Shading
- Textures



INFOGR – Computer Graphics

Jacco Bikker & Debabrata Panja - April-July 2019

END OF Lecture 6: “Ray Tracing”

Next lecture: “Ray Tracing (2)”