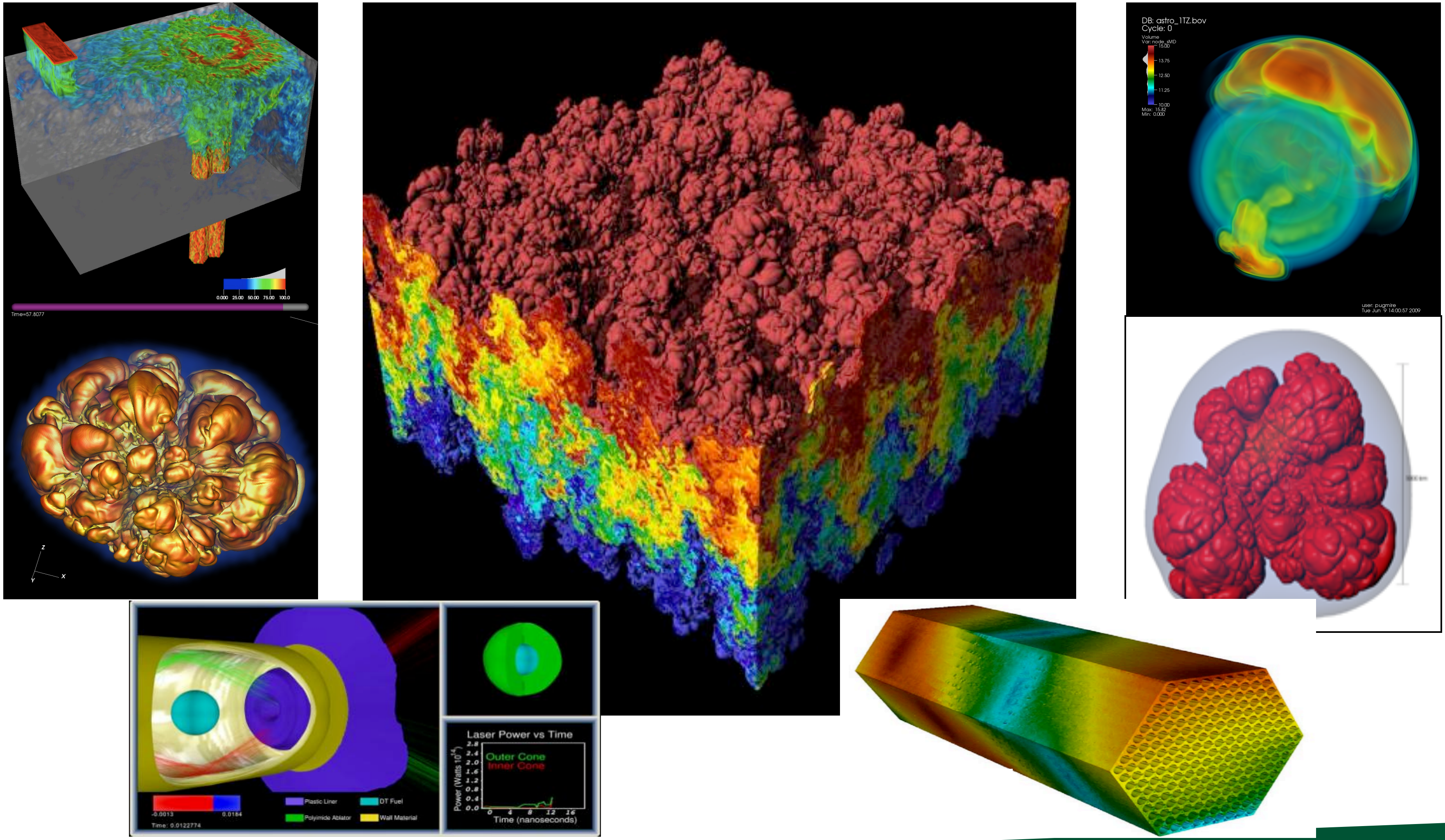
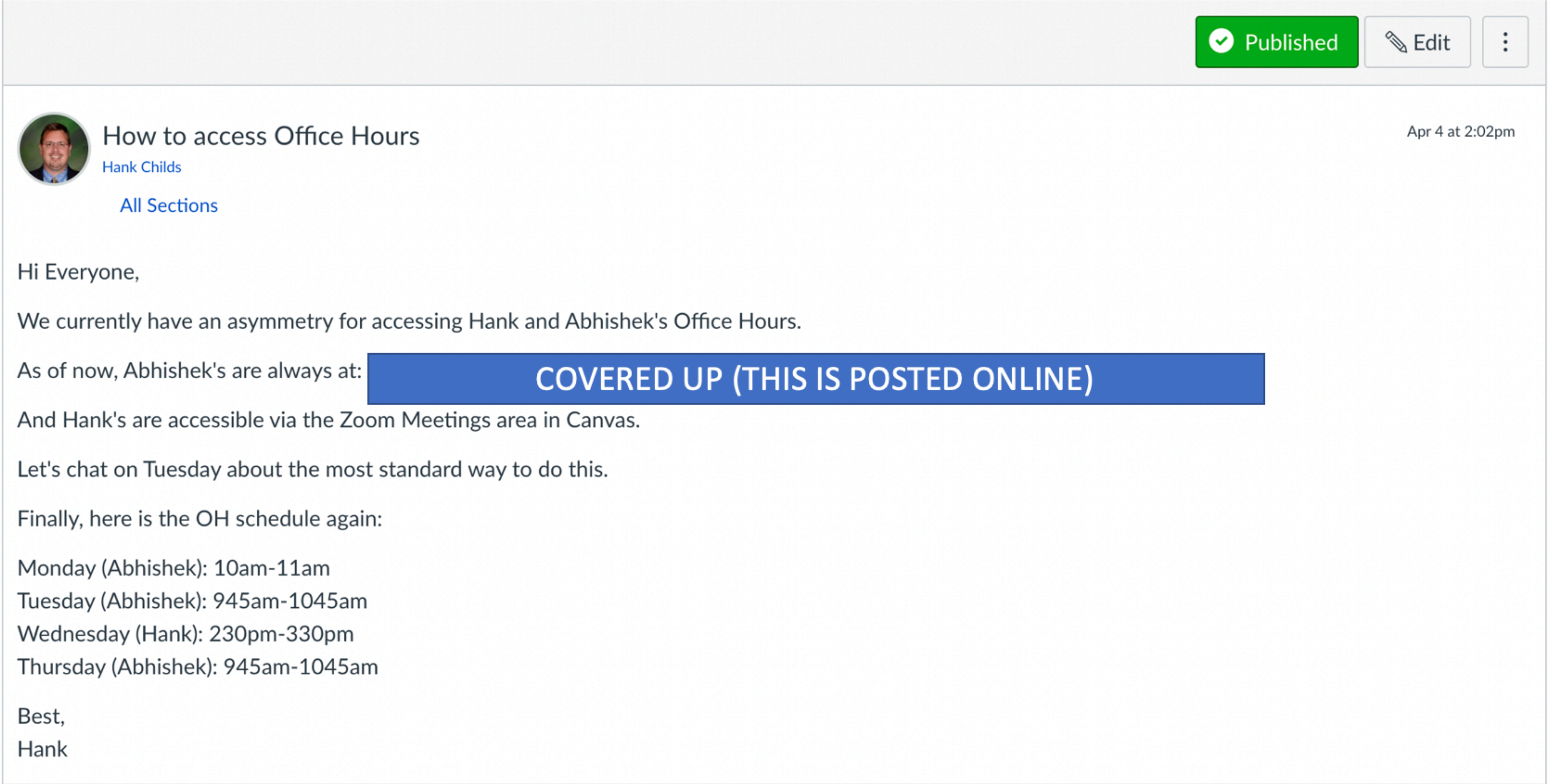
CIS 441/541: Intro to Computer Graphics
Lecture 12: Textures and Ray Tracing



May 18, 2021

Hank Childs, University of Oregon

Office Hours



How to access Office Hours

Hank Childs

All Sections

Apr 4 at 2:02pm

Hi Everyone,

We currently have an asymmetry for accessing Hank and Abhishek's Office Hours.

As of now, Abhishek's are always at: COVERED UP (THIS IS POSTED ONLINE)

And Hank's are accessible via the Zoom Meetings area in Canvas.

Let's chat on Tuesday about the most standard way to do this.

Finally, here is the OH schedule again:

Monday (Abhishek): 10am-11am
Tuesday (Abhishek): 945am-1045am
Wednesday (Hank): 230pm-330pm
Thursday (Abhishek): 945am-1045am

Best,
Hank

Discuss Quiz 3

Two Choices for Final Project

- Custom final project
  - You define the project, should be ~25 hours of work
  - Present project to class/judges on Finals Week
- Pre-defined projects
  - Pick three 8-hour projects from a menu of 4-6 projects

- Whether you do custom or pre-defined, you must attend the final period and watch the presentations
  - -4 points if you skip

Pre-Defined Projects

- Planning on having 4-6 pre-defined projects
- You choose 3
- On Tuesday May 18th, we will release project ~~2C~~ 3A
  - Likely: view manipulation from keyboard events
- On Tuesday May 25th, we will release the rest of the projects
  - ~~(possibly called 2D, 2E, 2F, etc.)~~ (called 3B, 3C, 3D, etc.)
  - These projects are TBD, but likely to include topics such as: texturing, physically based rendering, mirrors

Custom Project Ideas

- Implement a game
- Implement a screen saver
- Build a model of something
- Implement a neat rendering effect
  - Many folks try ray tracing (will discuss this later this lecture)

- ~~… Will show examples in a few slides~~

Custom Project Proposals

- If you want to do a custom project, please send me a proposal
- “Deadline”: ideally Tuesday May 18th (today)
- Why?
  - Get the scope right
  - Make an agreement early on
    - Protects you and me
- Important concept: minimum viable deliverable
- Proposal can be whatever length you see fit
  - One paragraph is fine

Remaining Lectures

- In support of project 3A/3B/3C/…
- In support of custom projects
  - (Ray tracing lecture)

Plan – Parentheticals Are Likely to Change

- This went well before, let’s do it again

Week	Sun	Mon	Tues	Weds	Thurs	Fri	Sat
8		2B due	Lec13 ~~(mouse+camera)~~ (textures) 3A avail Proposals due		Lec14 ~~(ray tracing)~~ Quiz 4 (GL)		
9			~~Lec15 (textures)~~ 3B, 3C, … avail		Lec16 Quiz 5 (rasterization)		
10			Live code		Quiz makeup		
Finals Week			Final Projects due All other work due: 1A-1F, 2A-2B not accepted after this point				

Plan for Thursday

- 830-900: 541 students only (discussion of project G)
- 900am-915am: general class discussion
- 915am: Quiz 4 starts

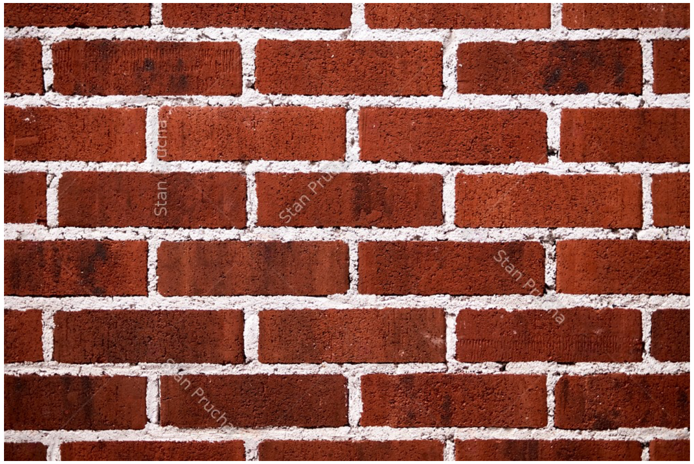
- Note: 441 students join at 9am

Textures

Textures

- “Textures” are a mechanism in OpenGL
- Mechanism is useful for many things
    - One of these is add “texture” to a surface, hence the name
- There are “1D”, “2D”, and “3D” textures
    - Most common is 2D, and placing “texture” on surfaces

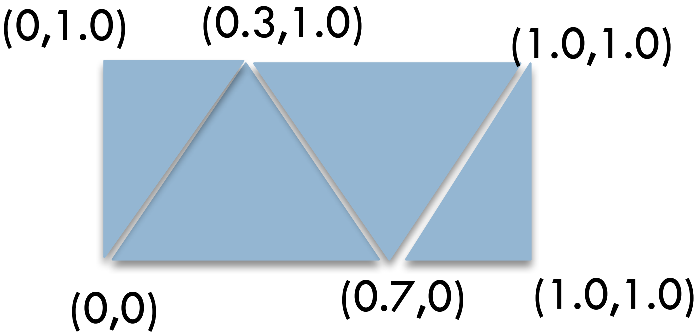
Motivation



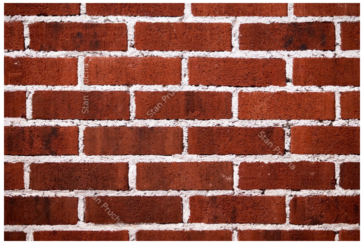
- Making a video game
- Have a brick wall in the background
- Want it to look like a brick wall
- But do not want to make a huge amount of geometry
  - Why not?
- Textures can help
- Value proposition: better look with less geometry

How Do Textures Work?

Step 1: new fields on your geometry
Called “texture coordinates”



Step 2: textures are loaded onto
GPU

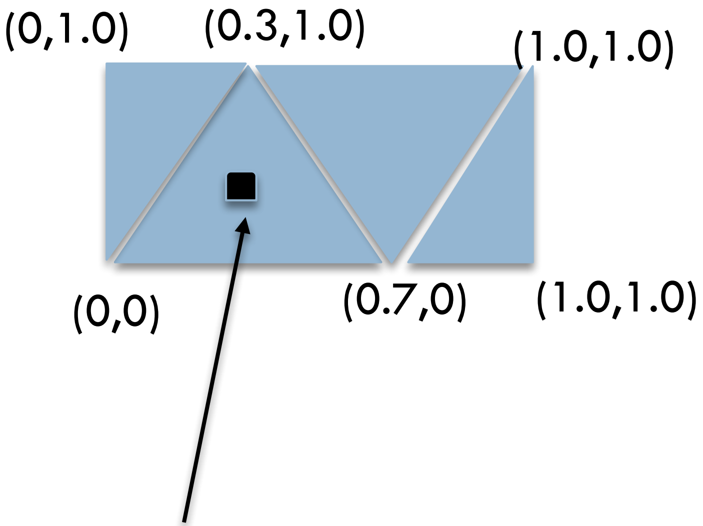


GPU memory

Step 3: texture coordinates and texture data are available during rasterization
Lots of ways to make graphics effects

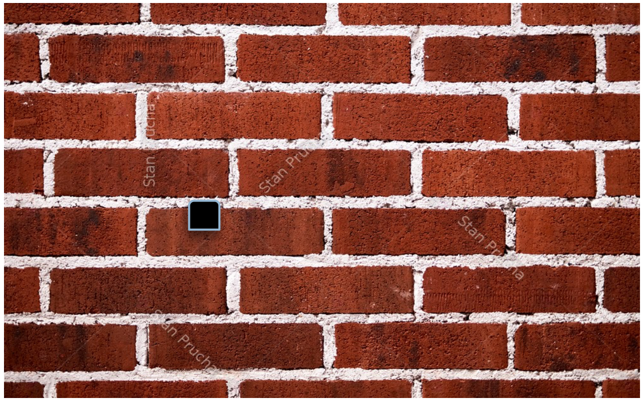
Most common: use texture coordinates to look up pixel color in texture data and then assign fragment that color

Textures for Fragment Color



Assume a fragment lies here

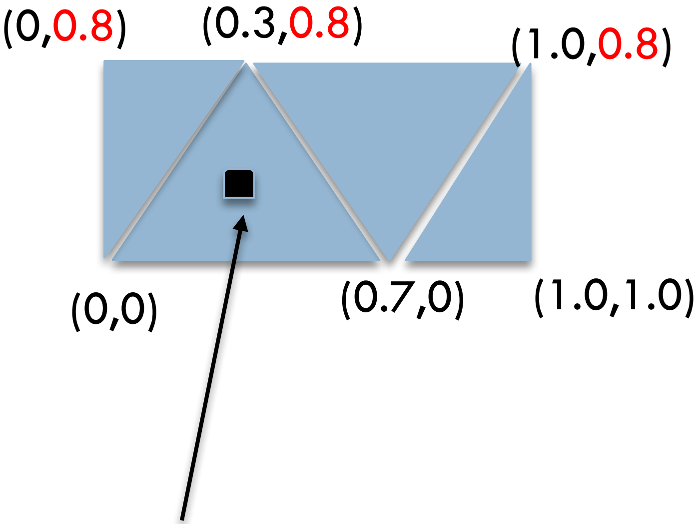
Texture coordinates are a field and will be LERP'ed like any other field
→ (0.3, 0.45)



A fragment shader can then go to the image and find the corresponding color

The color at that pixel becomes the color of the fragment

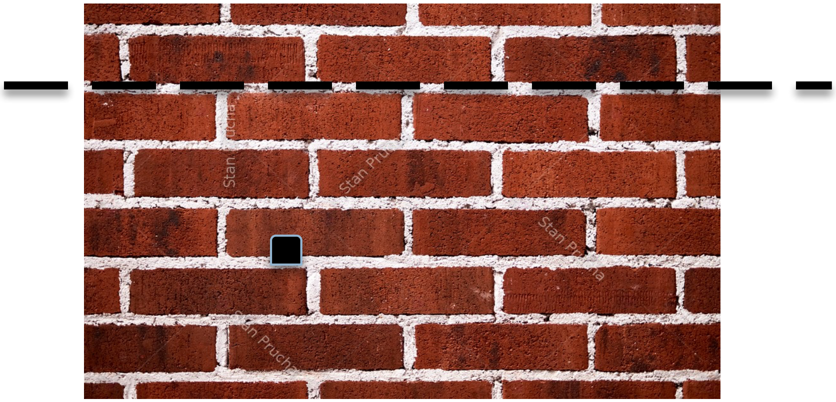
Observation #1



Assume a fragment lies here

Texture coordinates are a field and will be LERP'ed like any other field → (0.3, 0.35)

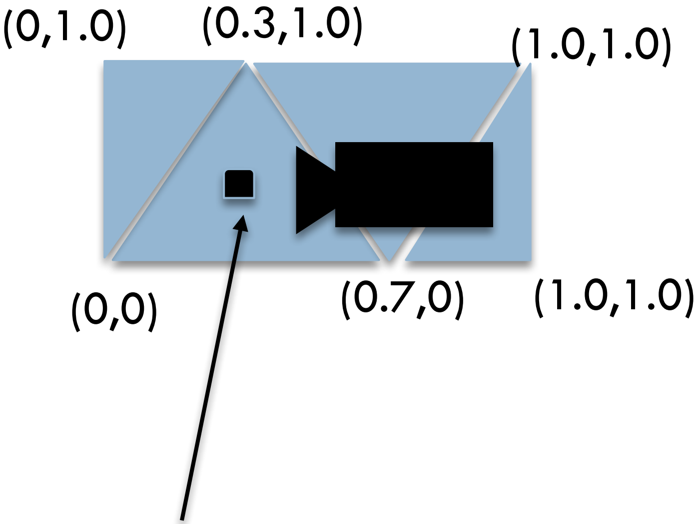
Did not use top 20% of the texture.
No problem. Maybe other triangles will.
Maybe not. Not an issue either way.



A fragment shader can then go to the image and find the corresponding color

The color at that pixel becomes the color of the fragment

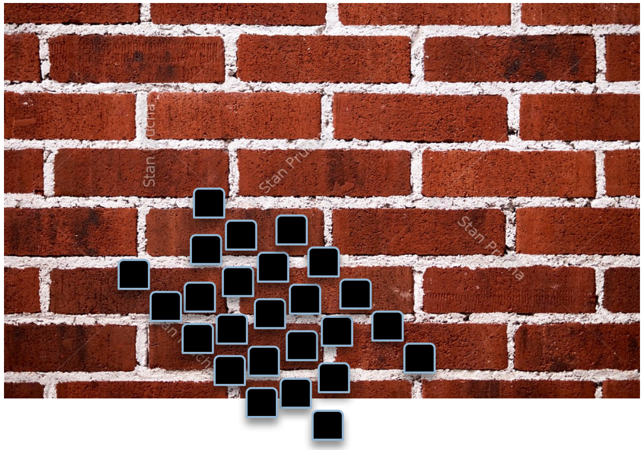
Observation #2



Assume a fragment lies here

Texture coordinates are a field and will be LERP'ed like any other field → (0.3, 0.45)

Camera zoomed in one triangle? No problem. Just get lots of fragments, which means lots of nearby pixels in the texture image

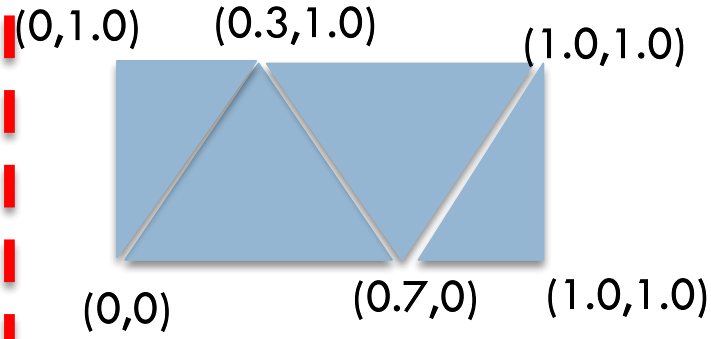


A fragment shader can then go to the image and find the corresponding color

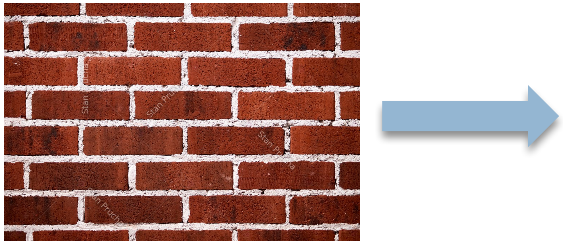
The color at that pixel becomes the color of the fragment

Actual OpenGL Code

Step 1: new fields on your geometry
Called "texture coordinates"



Step 2: textures are loaded onto
GPU



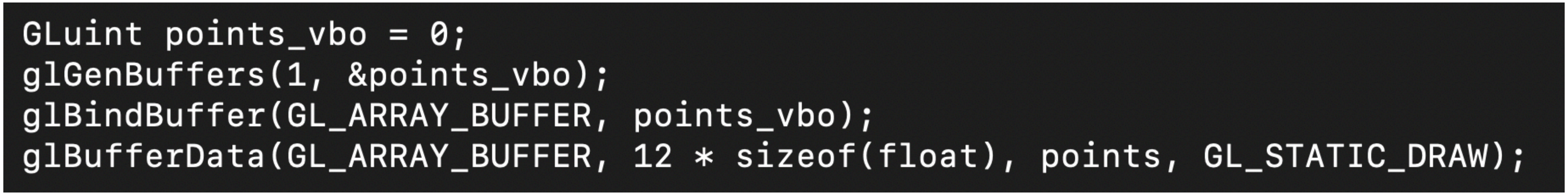
GPU memory

Step 3: texture coordinates and texture data are available during rasterization
Lots of ways to make graphics effects

Most common: use texture coordinates to look up pixel color in texture data and then assign fragment that color

Step 1: New Fields on Your Geometry

- OpenGL is all set up for texture coordinates
- Just another VBO that goes within a VAO

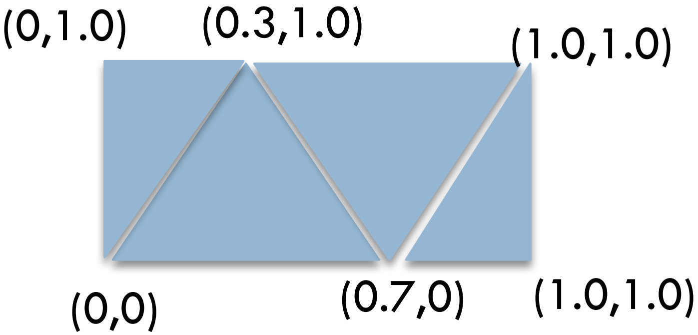


```
GLuint points_vbo = 0;
glGenBuffers(1, &points_vbo);
glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
glBufferData(GL_ARRAY_BUFFER, 12 * sizeof(float), points, GL_STATIC_DRAW);
```

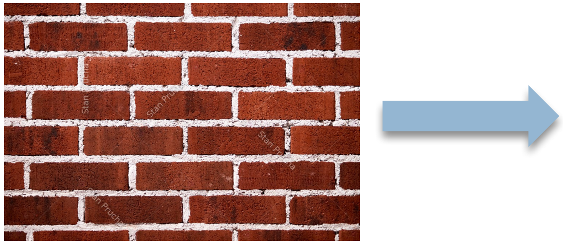
(This is an image from Lecture 9 about making a VBO for points)
(With textures, there are 2 floats per vertex for 2D textures,
1 float per vertex for 1D textures, etc.)

Actual OpenGL Code

<u>Step 1</u>: new fields on your geometry
Called "texture coordinates"



<u>Step 2</u>: textures are loaded onto GPU



GPU memory

<u>Step 3</u>: texture coordinates and texture data are available during rasterization
Lots of ways to make graphics effects

<u>Most common: use texture coordinates to look up pixel color in texture data and then assign fragment that color</u>

Steps for Loading Texture to GPU

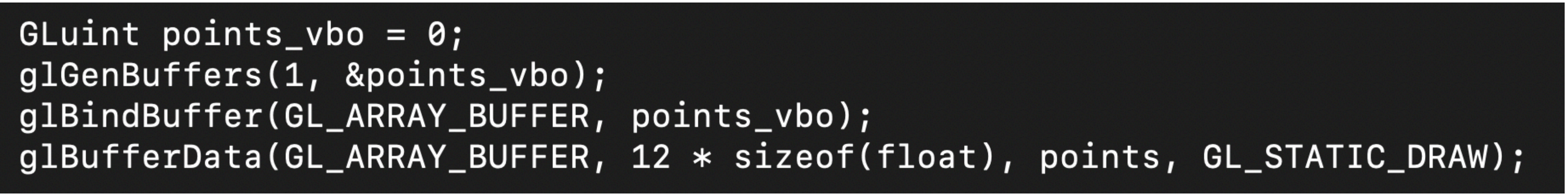
- 1) Generate the texture
- 2) Tell the shader program how to access the texture

Steps for Loading Texture to GPU

- 1) Generate the texture
- 2) Tell the shader program how to access the texture

Generating a Texture

- Very similar to VBOs



```
GLuint points_vbo = 0;
glGenBuffers(1, &points_vbo);
glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
glBufferData(GL_ARRAY_BUFFER, 12 * sizeof(float), points, GL_STATIC_DRAW);
```

- Now:
  - glGenTextures
  - glBindTexture
  - glTexImage2D (like glBufferData)

With 2 extra steps

Generating a Texture

- glGenTextures
- glActiveTexture
- glBindTexture
- glTexImage2D
- glGenerateMipmap

Generating a Texture

- glGenTextures
- glActiveTexture
- glBindTexture
- glTexImage2D
- glGenerateMipmap

glGenTextures

- Generates a handle.
  - You to OpenGL: I want to make some textures
  - OpenGL: let's refer to your textures using the following numbers
- Note: can generate multiple handles
- If you are doing multiple textures, then you call glGenTextures one time total
  - The rest of the calls are once per texture
- Example:
  - `GLuint textures[2];  glGenTextures(2, textures);`

Generating a Texture

- glGenTextures
- glActiveTexture
- glBindTexture
- glTexImage2D
- glGenerateMipmap

glActiveTexture

- The GPU can support many textures
- Each texture is given an ID (texture0, texture1, etc.)
- This call tells the GPU which texture you will be referring to
- Example:
  - glActiveTexture(GL_TEXTURE0);
  - Also:
    - glActiveTexture(GL_TEXTURE1);
    - glActiveTexture(GL_TEXTURE0+1);

Generating a Texture

- glGenTextures
- glActiveTexture
- glBindTexture
- glTexImage2D
- glGenerateMipmap

glBindTexture

- Two purposes:
  - (1) make this texture be current
    - Meaning subsequent calls refer to this texture
  - (2) specify the texture type (1D, 2D, 3D)
- Example:
  - `glBindTexture(GL_TEXTURE_1D, textures[0]);`

Generating a Texture

- glGenTextures
- glActiveTexture
- glBindTexture
- glTexImage2D
- glGenerateMipmap

glTexImage1D

- Sends texture data to GPU
- Example:
  - glTexImage1D(GL_TEXTURE_1D, 0, GL_RGB, num, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
  - This example: 256 colors
    - num is 256
    - data: array of size 256*3 bytes
    - GL_RGB means colors
      - GL_RED when just a single value
  - Other values are for data layouts we are not using
    - Example: load one giant buffer with vertex position, normals, texture coordinates, etc., and use offsets

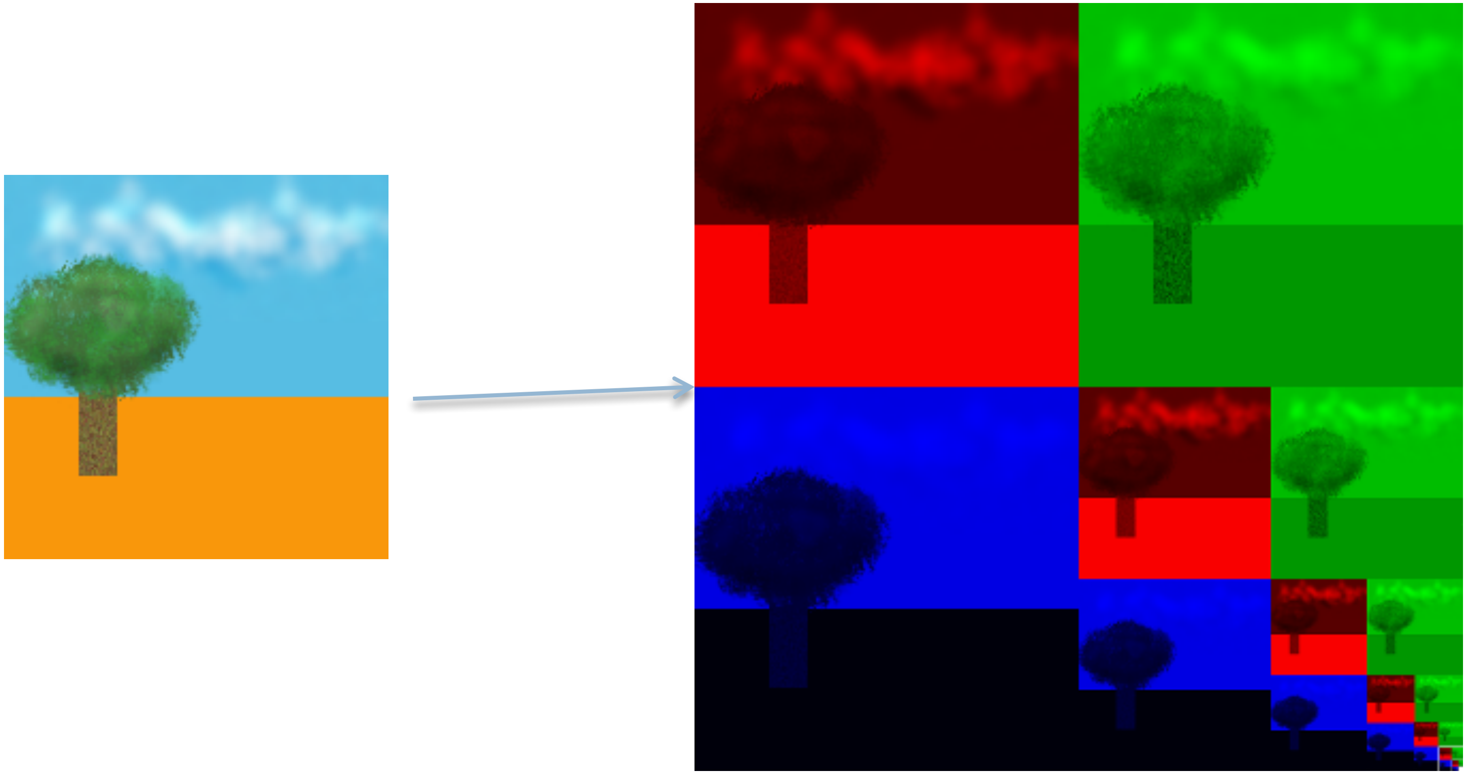
Generating a Texture

- glGenTextures
- glActiveTexture
- glBindTexture
- glTexImage2D
- glGenerateMipmap

Mipmaps

- Mipmaps: pre-calculated, optimized collections of images that accompany a main texture, intended to increase rendering speed and reduce aliasing artifacts
- Widely used in 3D computer games, flight simulators and other 3D imaging systems
- In use, it is called “mipmapping”
- The letters "MIP" in the name are an acronym of the Latin phrase multum in parvo, meaning "much in little”

Mipmaps



glGenerateMipMap

- glGenerateMipmap(GL_TEXTURE_1D);

Steps for Loading Texture to GPU

- 1) Generate the texture <-- We just finished this part
- 2) Tell the shader program how to access the texture

Tell the Shader Program How to Access the Texture

-
  ```
  GLuint texture1Location =
        glGetUniformLocation(shader_program,
                                       "texture1");
  ```
- `glUniform1i(texture1Location, 0);`

- The texture referred to as “texture1” is located in the first texture location (i.e., GL_TEXTURE0)

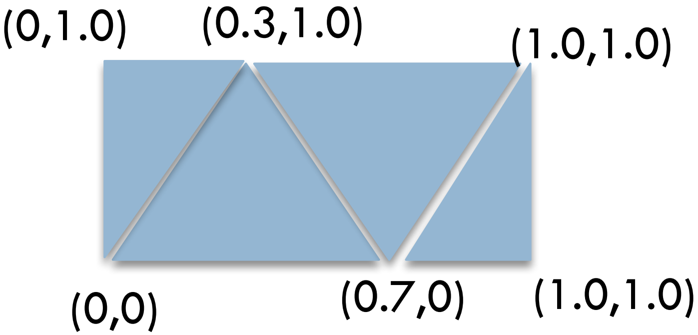
Tell the Shader Program How to Access the Texture

- GLuint texture1Location = glGetUniformLocation(shader_program, ”random_name");
- glUniform1i(texture1Location, 5);

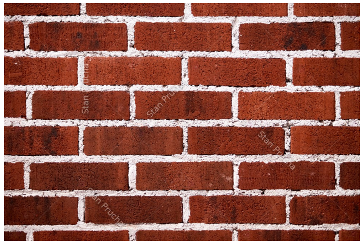
- The texture referred to as “random_name” is located in the sixth texture location (i.e., GL_TEXTURE0+5)

Actual OpenGL Code

Step 1: new fields on your geometry
Called "texture coordinates"



Step 2: textures are loaded onto
GPU



GPU memory

Step 3: texture coordinates and texture data are available during rasterization
Lots of ways to make graphics effects

Most common: use texture coordinates to look up pixel color in texture data and then assign fragment that color

Two New GLSL Constructs for Shaders

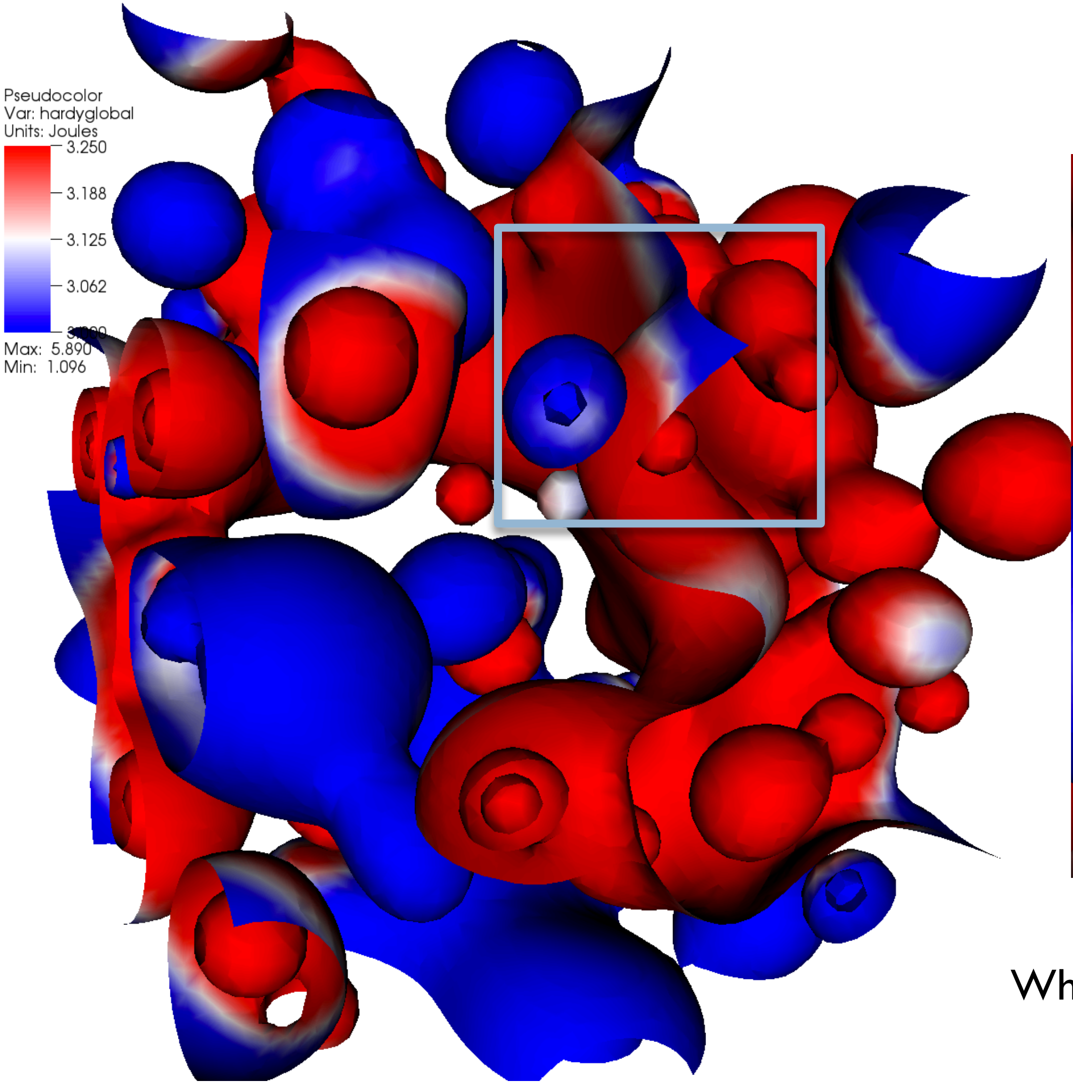
- 1) Textures have their own type: sampler1D, sampler2D, sampler3D
  - Other types we know: float, vec4, etc.
- 2) There is a special function that does texture lookups, called “texture”
- Example fragment shader:

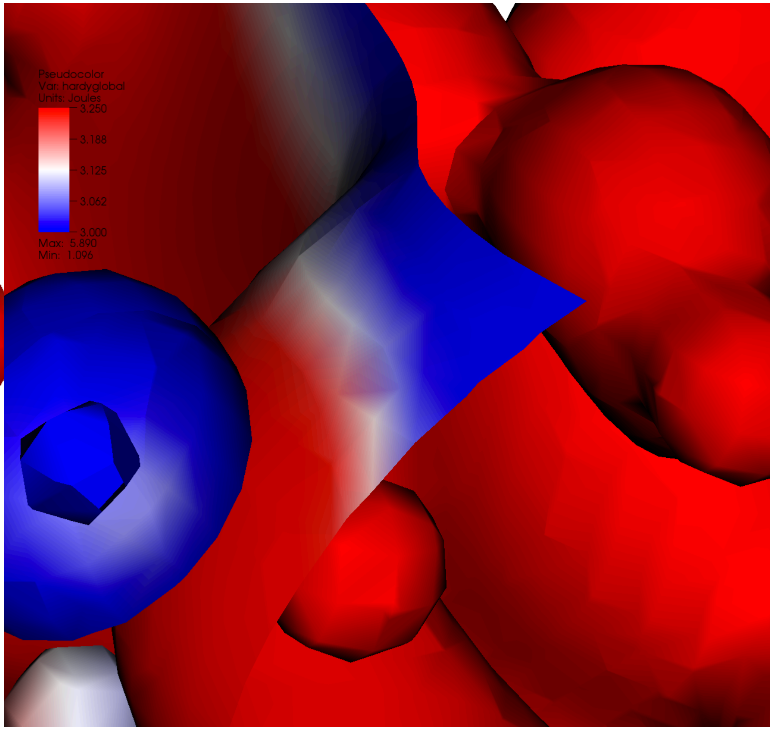
```
in float tex_coord;
uniform sampler1D random_name;
void main() {
    frag_color = texture(random_name, tex_coord);
}
```

Project 3A: 1D Textures

- Will use two 1D textures

Visualization use case

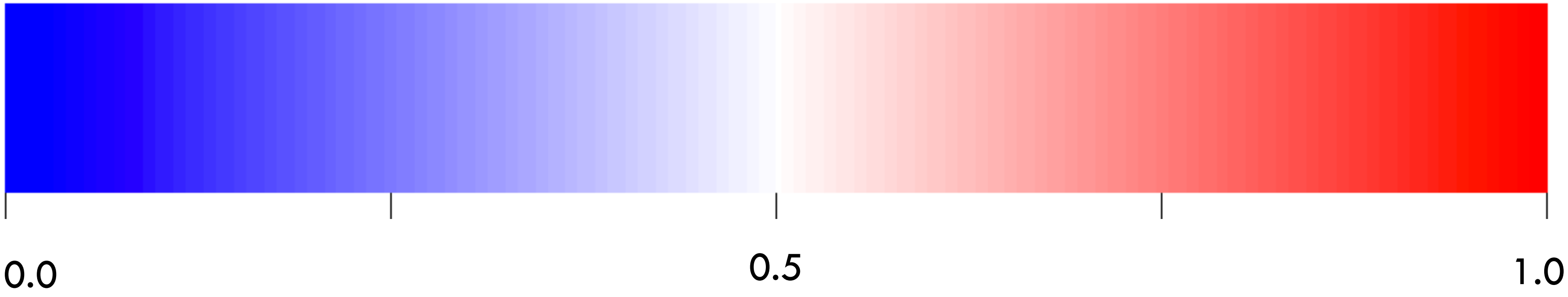




Why is there purple in this picture?

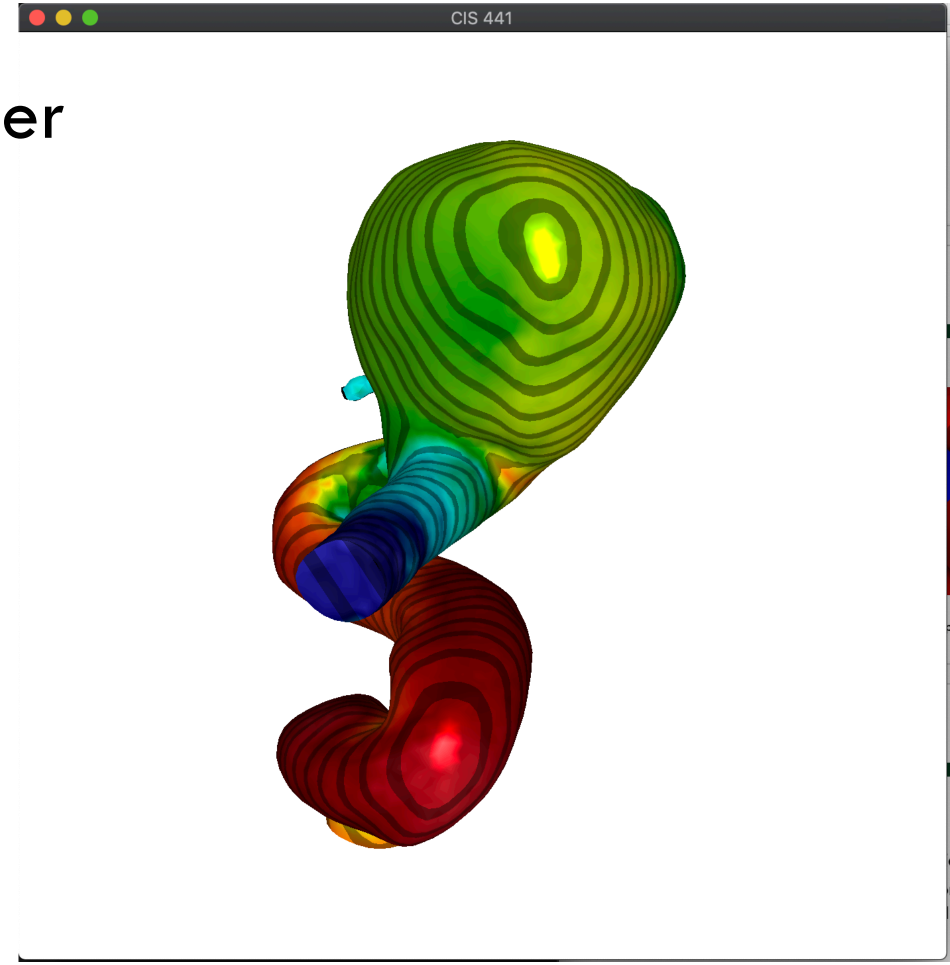
Two Ideas

- Plan #1: send colors through rasterization
  - Problem: get "purple"
- Plan #2: send data values through rasterization
  - Fragment shader map data value to texture coordinate, then uses texture coordinate to look up color in 1D texture



3A: why *two* 1D textures?

- ☐ One texture creates colors
- ☐ Second texture creates "tiger stripe" effect
- ☐ Practice using multiple textures



Introduction to Ray Tracing

Dr. Xiaoyu Zhang

Cal State U., San Marcos

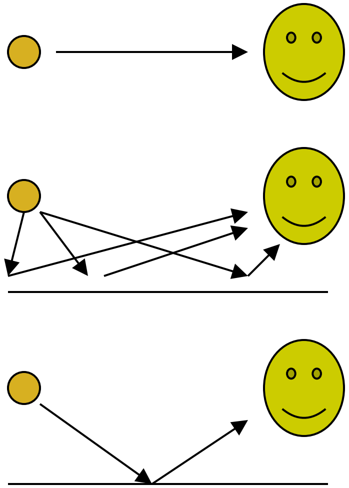
Classifying Rendering Algorithms

- One way to classify rendering algorithms is according to the type of light interactions they capture
- For example: The OpenGL lighting model captures:
  - Direct light to surface to eye light transport
  - Diffuse and rough specular surface reflectance
  - It actually doesn’t do light to surface transport correctly, because it doesn’t do shadows
- We would like a way of classifying interactions: *light paths*

Classifying Light Paths

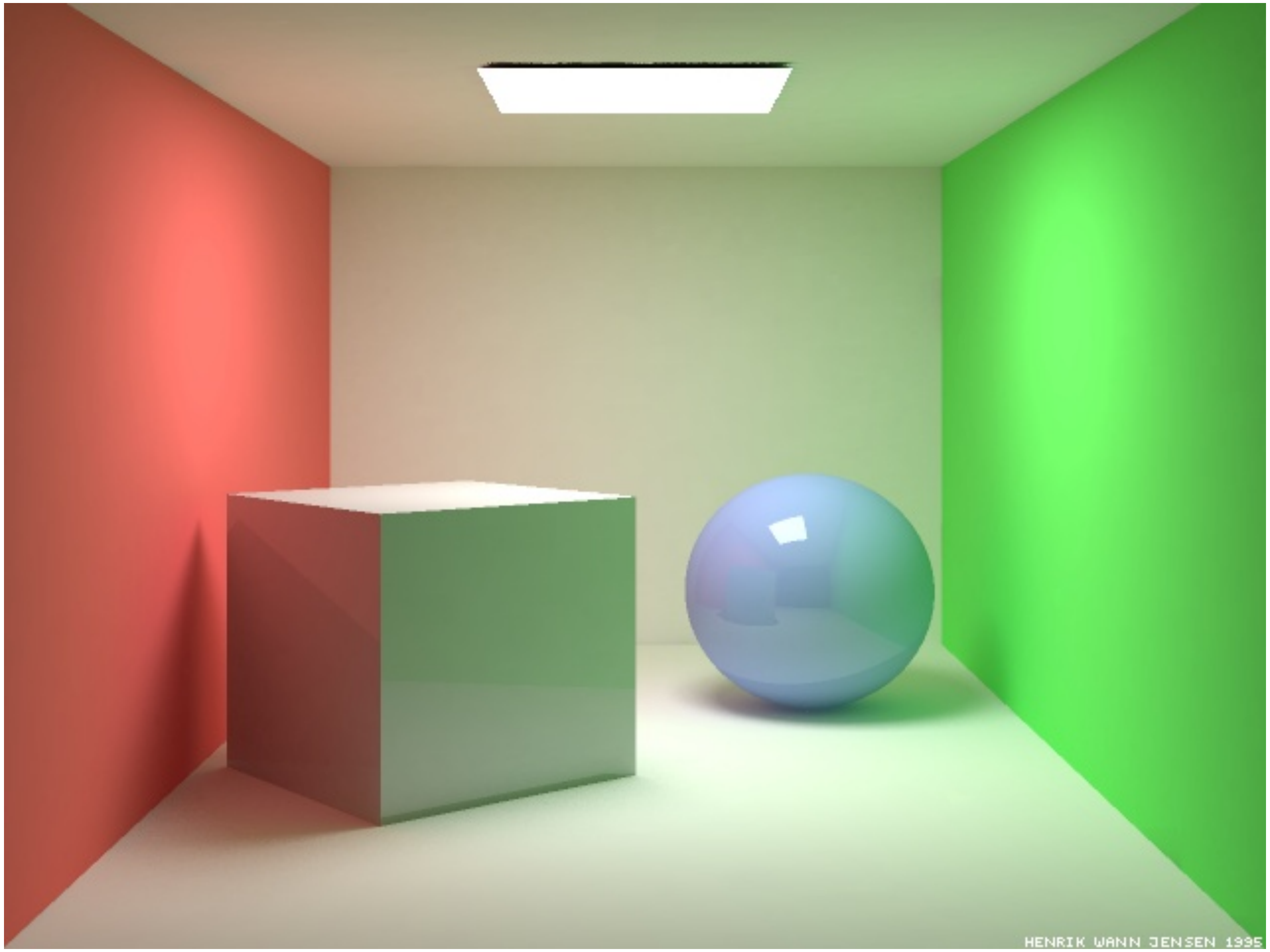
- Classify light paths according to where they come from, where they go to, and what they do along the way
- Assume only two types of surface interactions:
  - Pure diffuse, D
  - Pure specular, S
- Assume all paths of interest:
  - Start at a light source, L
  - End at the eye, E
- Use regular expressions on the letters D, S, L and E to describe light paths
  - Valid paths are L(D|S)*E

Simple Light Path Examples



- LE
  - The light goes straight from the source to the viewer
- LDE
  - The light goes from the light to a diffuse surface that the viewer can see
- LSE
  - The light is reflected off a mirror into the viewer’ s eyes
- L(S|D)E
  - The light is reflected off either a diffuse surface or a specular surface toward the viewer
- Which do OpenGL (approximately) support?

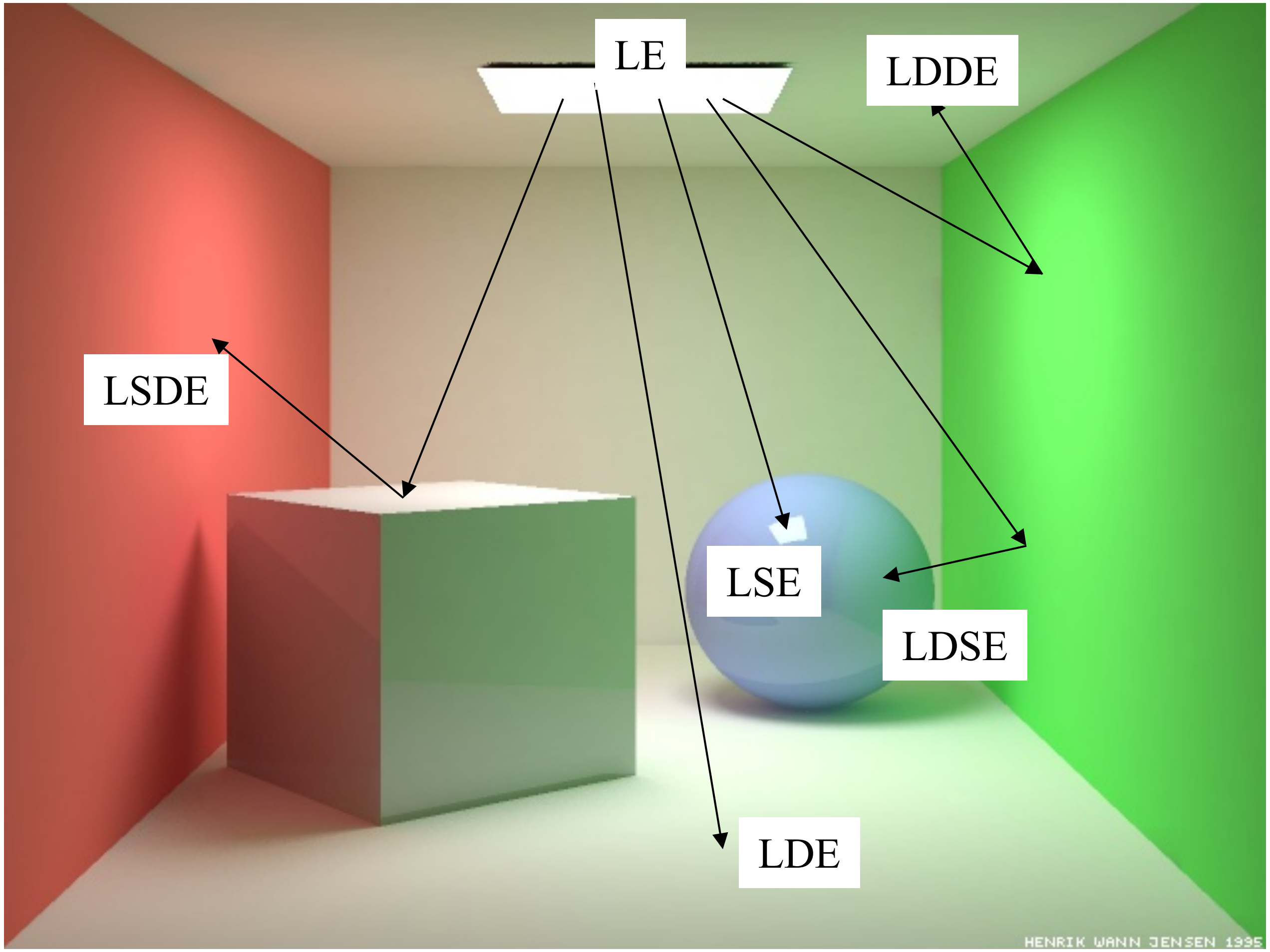
More Complex Light Paths



- Find the following:
  - LE
  - LDE
  - LSE
  - LDDE
  - LDSE
  - LSDE

Radiosity Cornell box, due to Henrik wann Jensen, http://www.gk.dtu.dk/~hwj, rendered with ray tracer

More Complex Light Paths



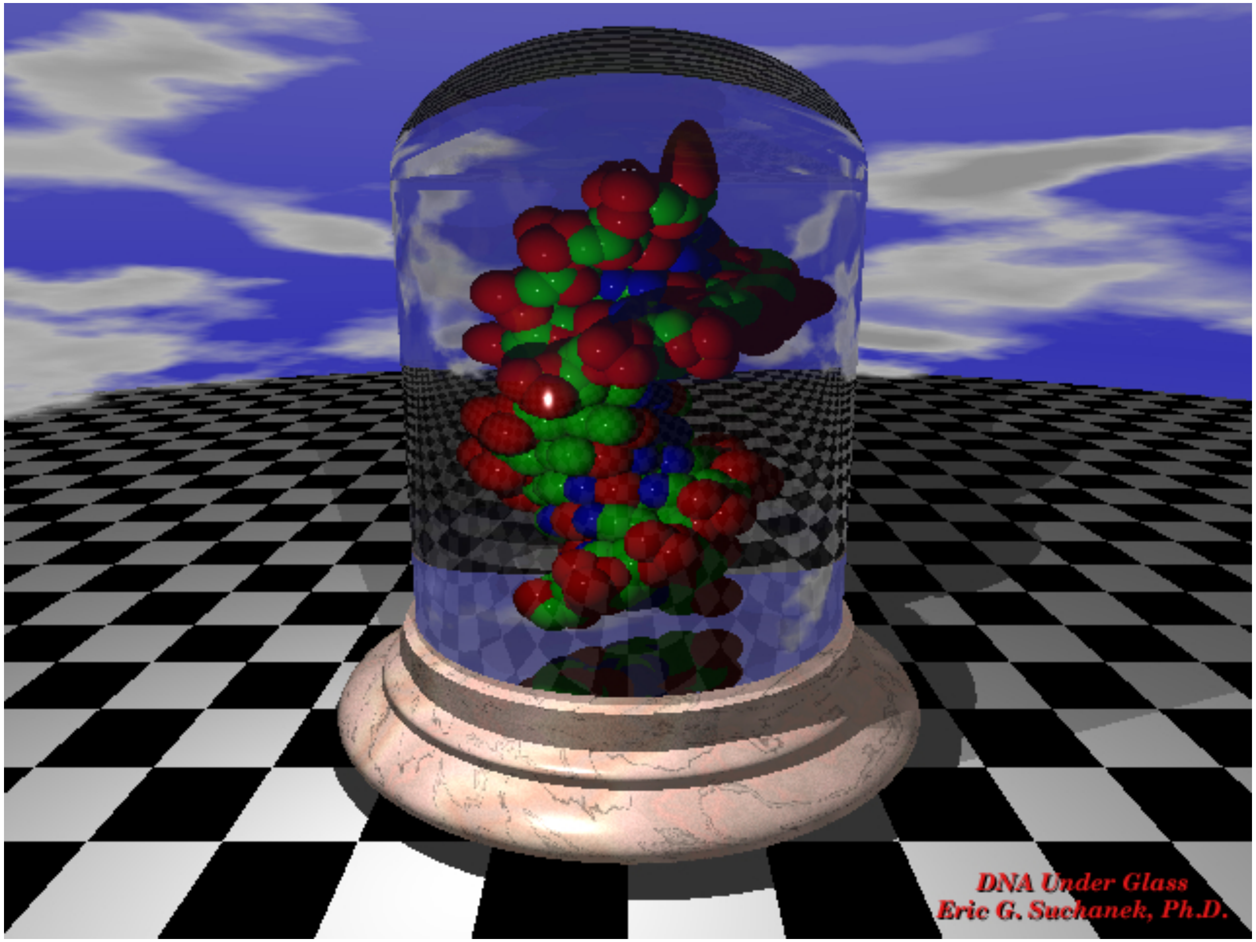
The OpenGL Model

- The “standard” graphics lighting model captures only L(D|S)E
- It is missing:
  - Light taking more than one diffuse bounce: LD*E
    - Should produce an effect called color bleeding, among other things
    - Approximated, grossly, by ambient light
  - Light refracted through curved glass
    - Consider the refraction as a “mirror” bounce: LDSE
  - Light bouncing off a mirror to illuminate a diffuse surface: LS+D+E
  - Many others
  - Not sufficient for photo-realistic rendering

Raytraced Images

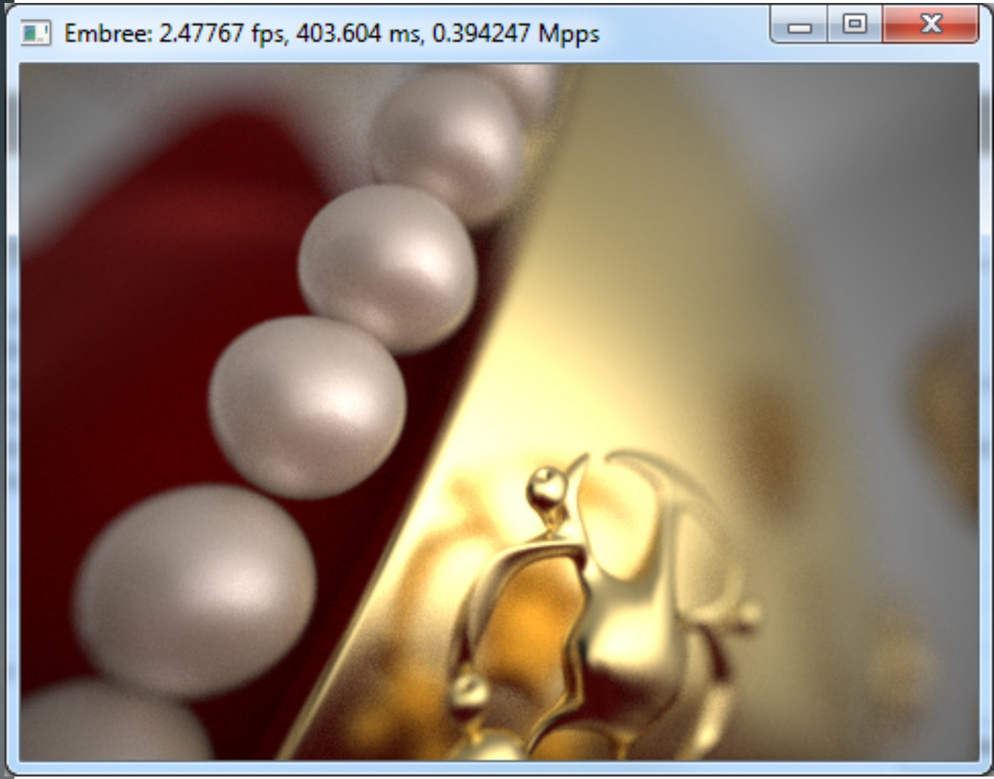
PCKTWTCH by
Kevin Odhner,
POV-Ray

Kettle, Mike Miller, POV-Ray


DNA Under Glass
Eric G. Suchanek, Ph.D.

The previous slides now look like amateur hour…

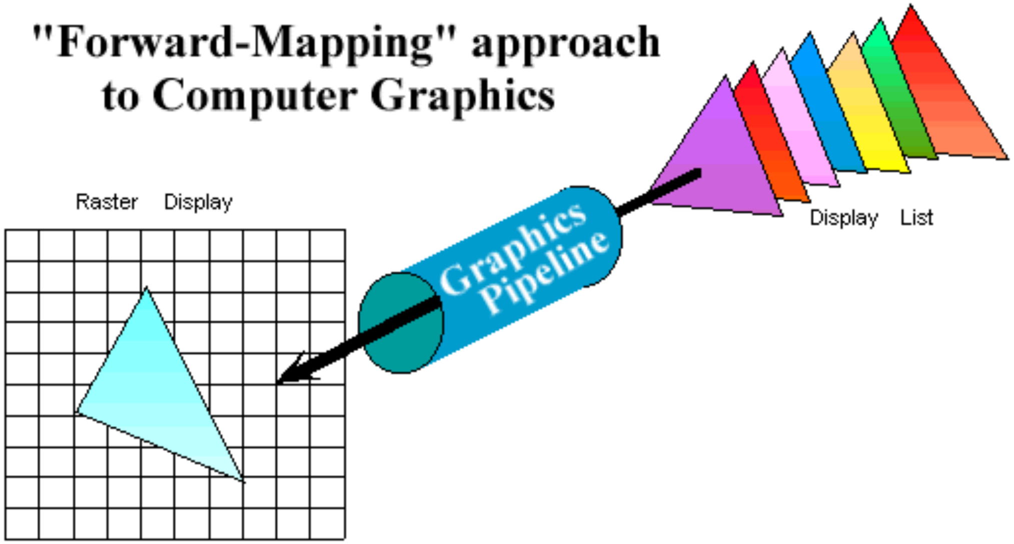
Model courtesy of Martin Lubich, www.loramel.net
HDR light courtesy of Lightmap Ltd, www.lightmap.co.uk



Embree: 2.47767 fps, 403.604 ms, 0.394247 Mpps

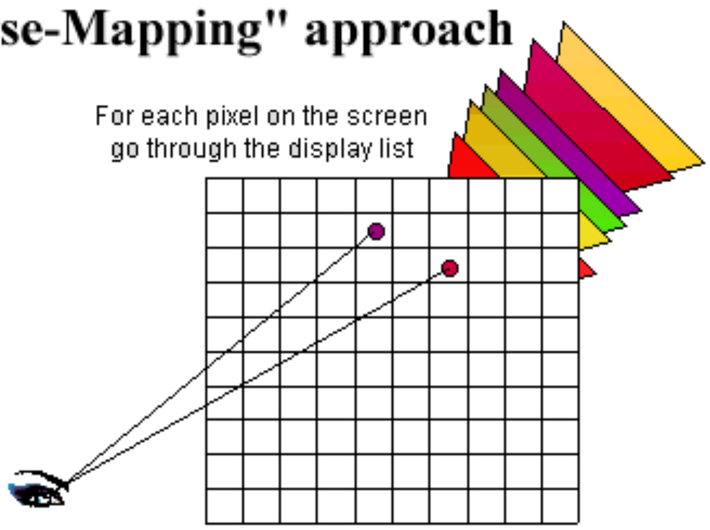
Graphics Pipeline Review

- Properties of the Graphics Pipeline
  - Primitives are transformed and projected (not depending on display resolution)
  - Primitives are processed one at a time
  - Forward-mapping from geometrical space to image space



Alternative Approaches: Ray CASTING (not Ray TRACING)

Ray-casting searches along lines of sight, or rays, to determine the primitive that is visible along it.

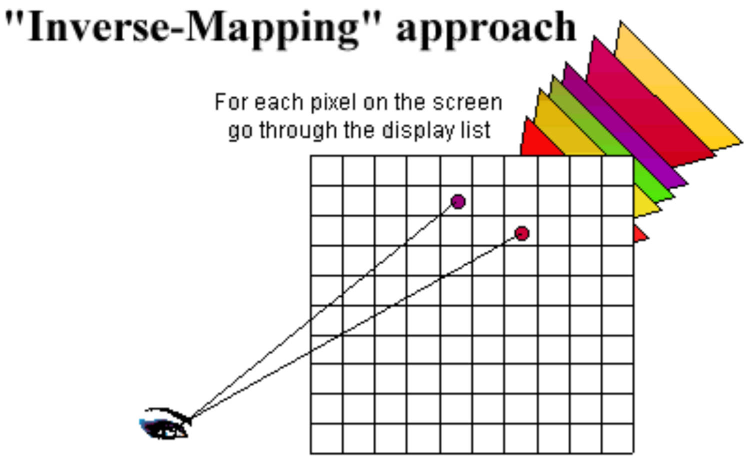


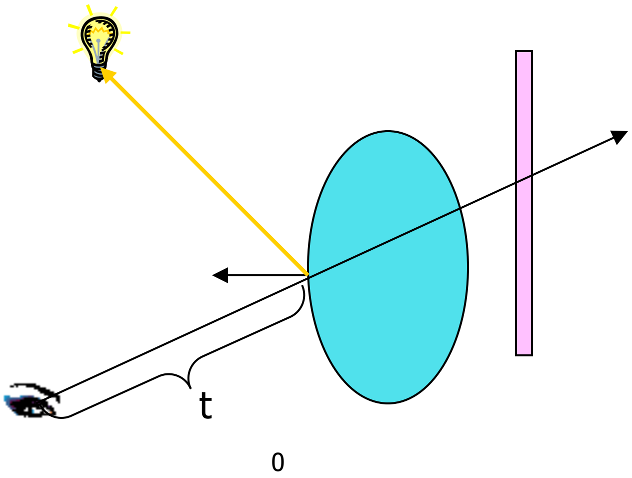
Properties of ray-casting:

- Go through all primitives at each pixel
- Image space sample first
- Analytic processing afterwards

Ray Casting Overview

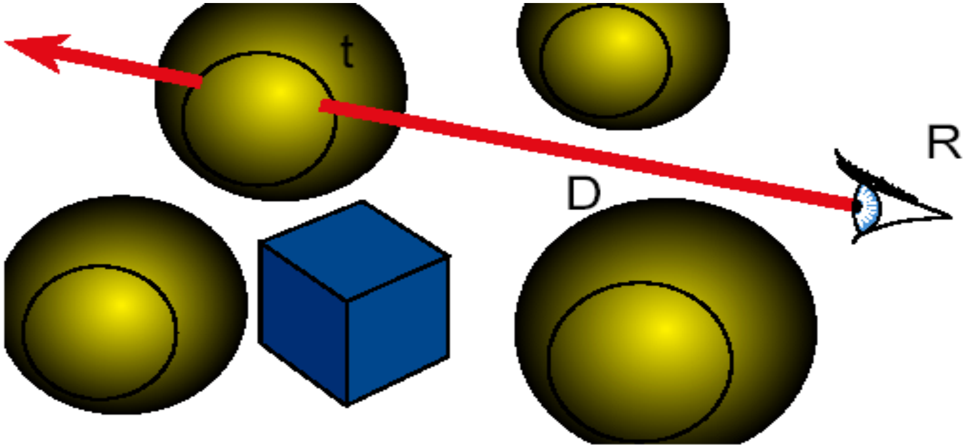
- For every pixel shoot a ray from the eye through the pixel.
- For every object in the scene
  - Find the point of intersection with the ray closest to (and in front of) the eye
  - Compute normal at point of intersection
- Compute color for pixel based on point and normal at intersection closest to the eye (e.g. by Phong illumination model).





Ray Casting

-
```
Ray Cast ( Point R, Ray D ) {
    foreach object in the scene
      find minimum t>0 such that R + t D hits object
    if ( object hit )
    return object
    else return background object
 }
```

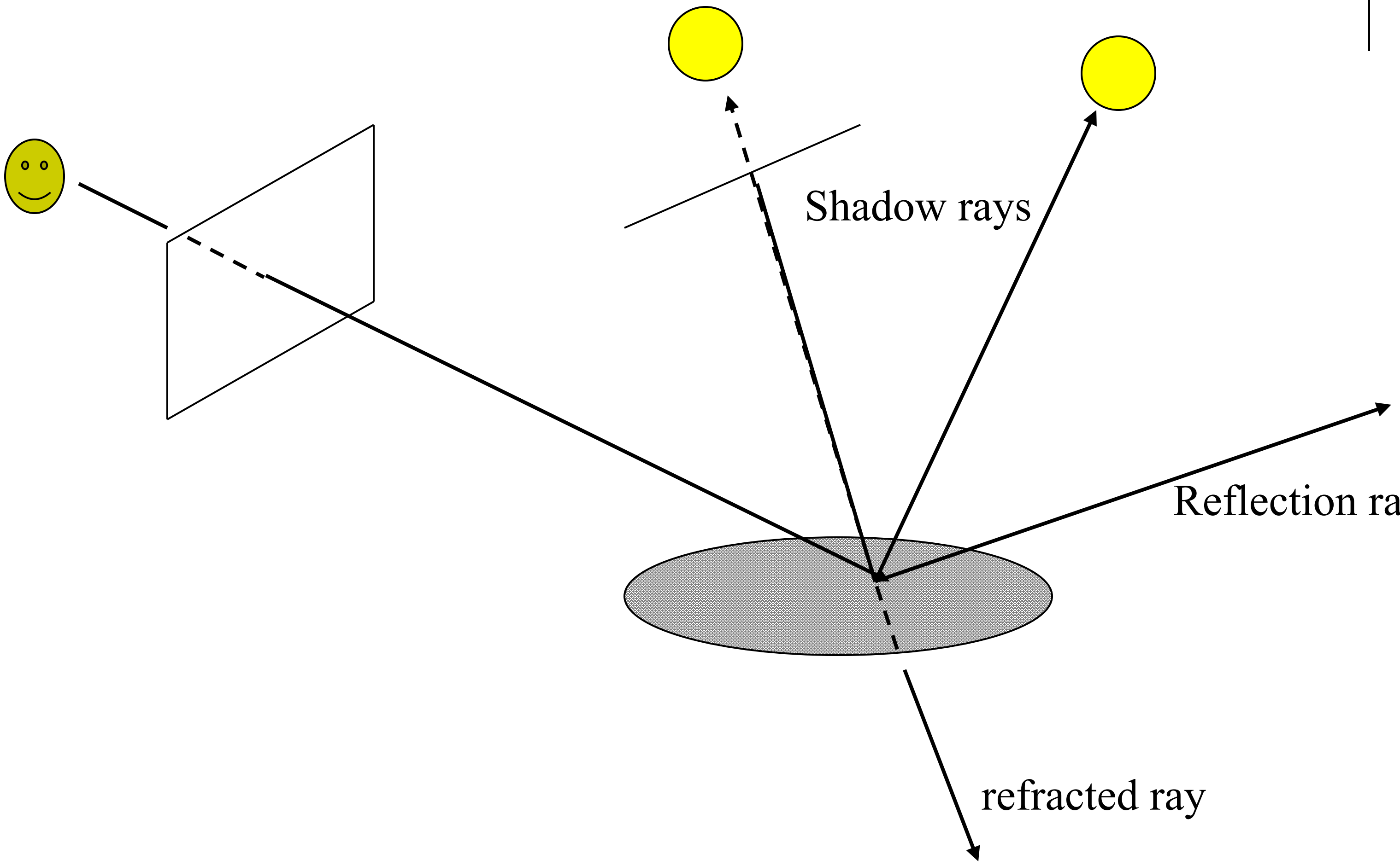


Raytracing

- Cast rays from the eye point the same way as ray casting
  - Builds the image pixel by pixel, one at a time

- Cast additional rays from the hit point to determine the pixel color
  - Shoot rays toward each light. If they hit something, then the object is shadowed from that light, otherwise use “standard” model for the light
  - Reflection rays for mirror surfaces, to see what should be reflected in the mirror
  - Refraction rays to see what can be seen through transparent objects
  - Sum all the contributions to get the pixel color

Raytracing



Recursive Ray Tracing

- When a reflected or refracted ray hits a surface, repeat the whole process from that point
  - Send out more shadow rays
  - Send out new reflected ray (if required)
  - Send out a new refracted ray (if required)
  - Generally, reduce the weight of each additional ray when computing the contributions to surface color
  - Stop when the contribution from a ray is too small to notice or maximum recursion level has been reached

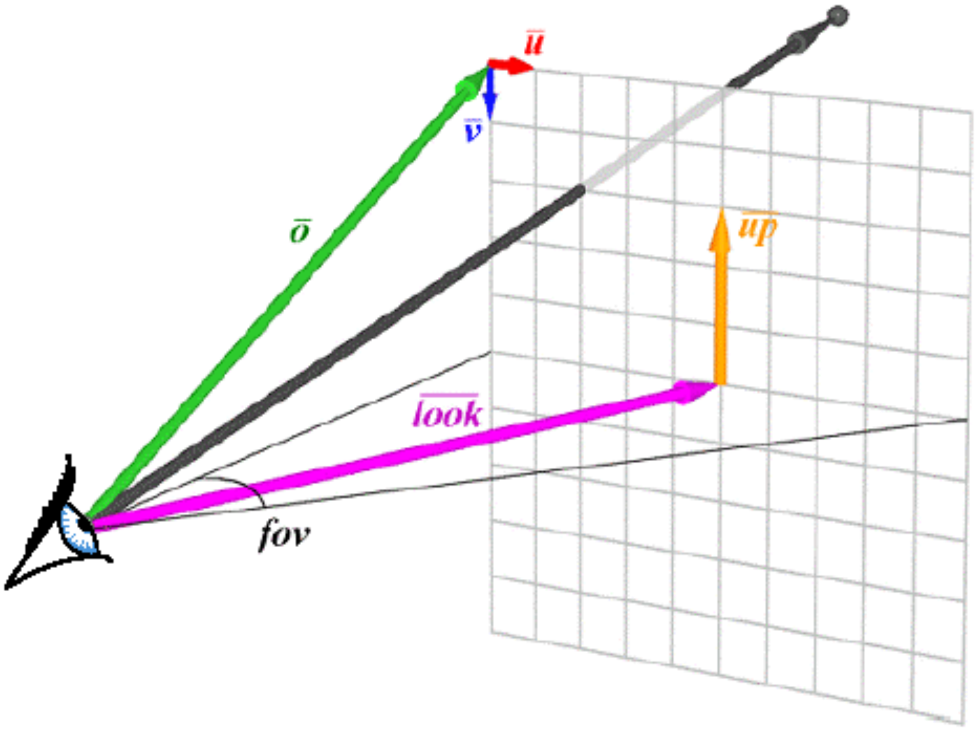
Raytracing Implementation

- Raytracing breaks down into two tasks:
  - Constructing the rays to cast
  - Intersecting rays with geometry
- The former problem is simple vector arithmetic
- Intersection is essentially root finding (as we will see)
  - Any root finding technique can be applied
- Intersection calculation can be done in world coordinates or model coordinates

Constructing Rays

- Define rays by an initial point and a direction: $\mathbf{x}(t)=\mathbf{x}_0+t\mathbf{d}$
- Eye rays: Rays from the eye through a pixel
  - Construct using the eye location and the pixel's location on the image plane. $\mathbf{X}_0$ = **eye**
- Shadow rays: Rays from a point on a surface to the light.
  - $\mathbf{X}_0$ = point on surface
- Reflection rays: Rays from a point on a surface in the reflection direction
  - Construct using laws of reflection. $\mathbf{X}_0$ = surface point
- Transmitted rays: Rays from a point on a transparent surface through the surface
  - Construct using laws of refraction. $\mathbf{X}_0$ = surface point

From Pixels to Rays



$$\vec{u} = \frac{look \times up}{|look \times up|}$$

$$\vec{v} = \frac{look \times \vec{u}}{|look \times \vec{u}|}$$

$$\Delta\vec{x} = \frac{2\tan(fov_x / 2)}{W}\vec{u}$$

$$\Delta\vec{y} = \frac{2\tan(fov_y / 2)}{H}\vec{v}$$

$$\vec{d}(i,j) = \frac{look}{|look|} + \frac{(2i+1-W)}{2}\Delta\vec{x} + \frac{(2j+1-H)}{2}\Delta\vec{y}$$

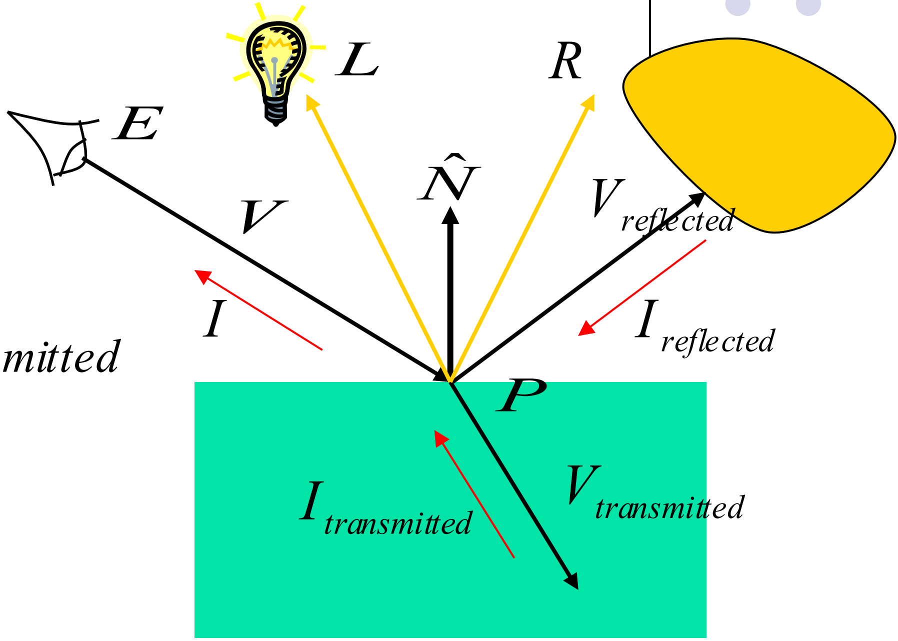
Ray Tracing Illumination

Recursive

$$I(E,V) = I_{direct} + I_{reflected} + I_{transmitted}$$

$$I_{reflected} = k_r I(P, V_{reflected})$$

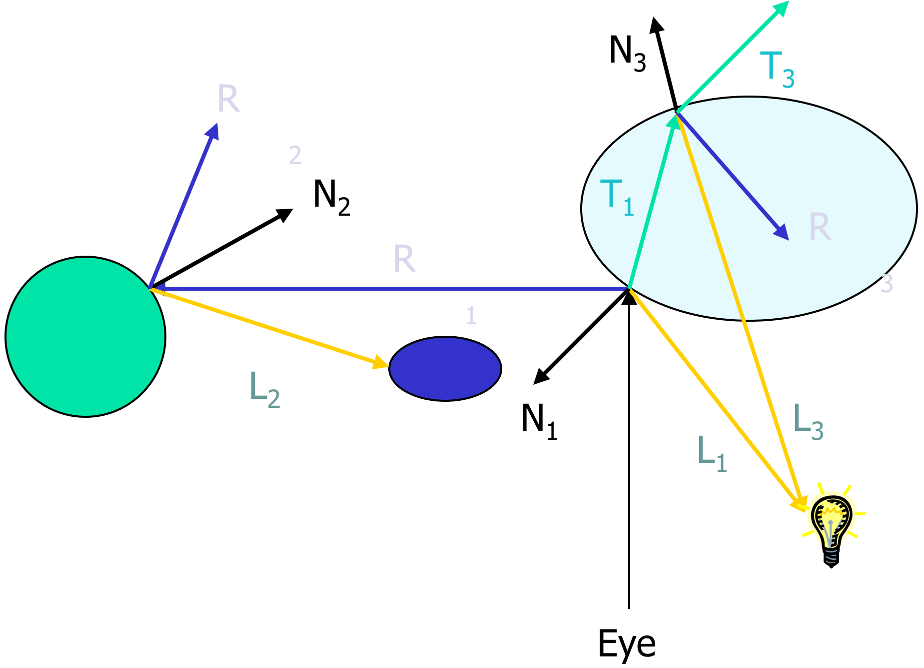
$$I_{transmitted} = k_t I(P, V_{transmitted})$$

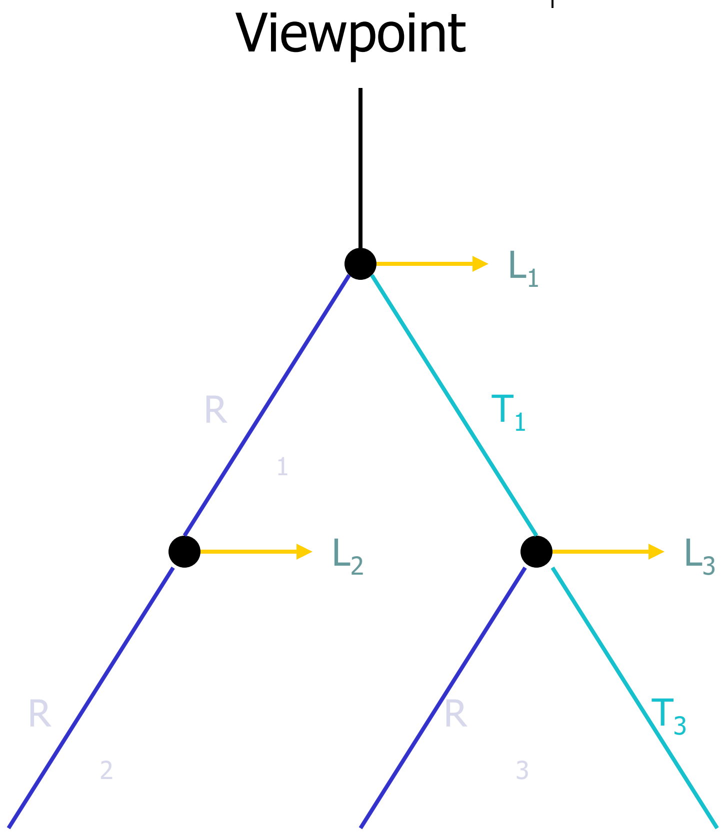


$$I_{direct} = k_a I_{ambient} + I_{light}\left[k_d\left(\hat{N}\cdot\hat{L}\right) + k_s\left(-\hat{V}\cdot\hat{R}\right)^{n_{shiny}}\right]$$

Check for shadowing (intersection with object along ray (P,L))

The Ray Tree





N_i surface normal

R_i reflected ray

L_i shadow ray

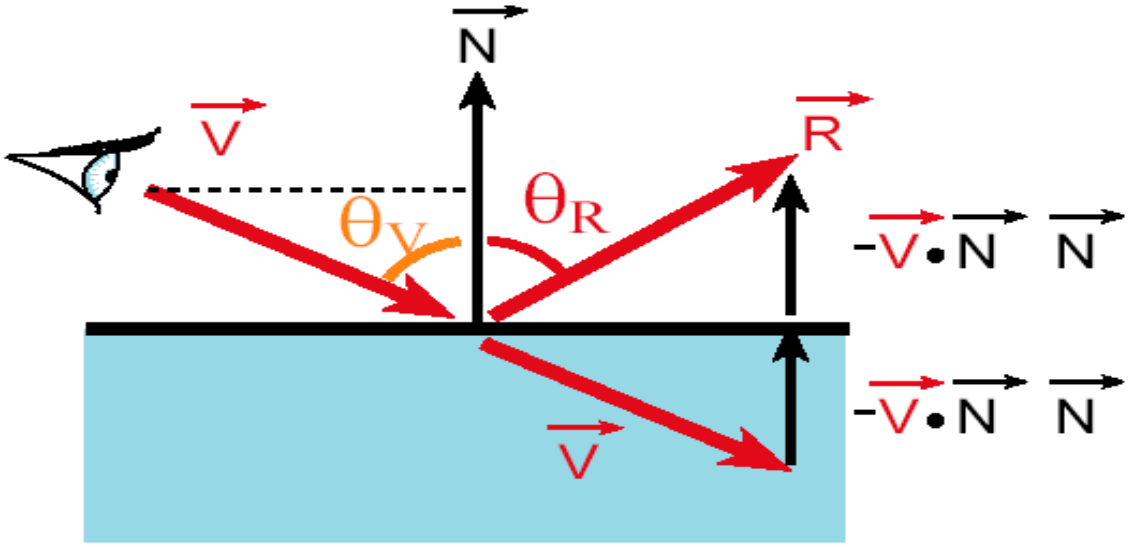
T_i transmitted (refracted) ray

Psuedo-code

Reflection

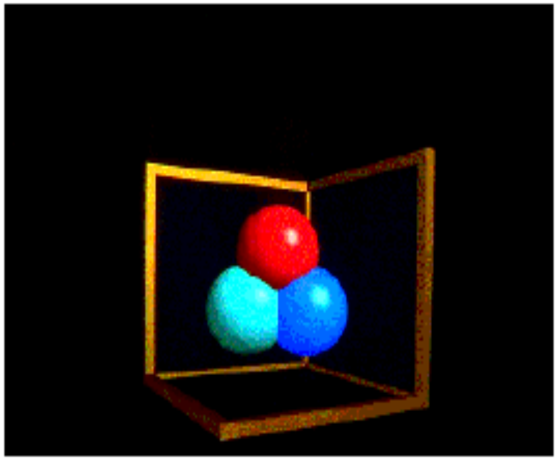
- Reflection angle = view angle

$$\vec{R} = \vec{V} - 2\left(\vec{V} \bullet \vec{N}\right)\vec{N}$$

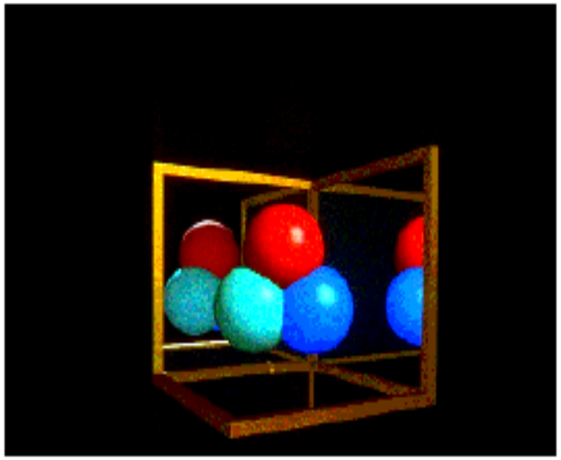


Reflection

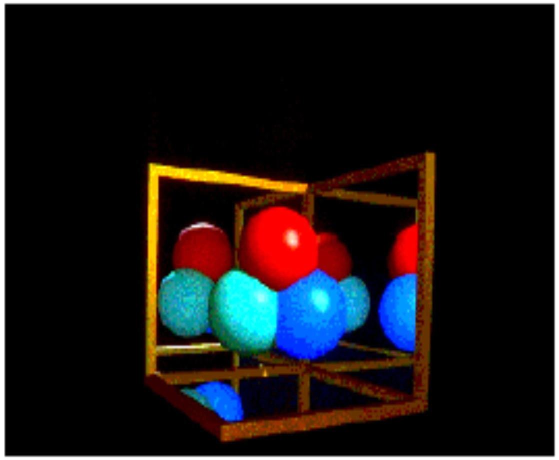
- The maximum depth of the tree affects the handling of refraction
- If we send another reflected ray from here, when do we stop? 2 solutions (complementary)
  - Answer 1: Stop at a fixed depth.
  - Answer 2: Accumulate product of reflection coefficients and stop when this product is too small.



0 recursion

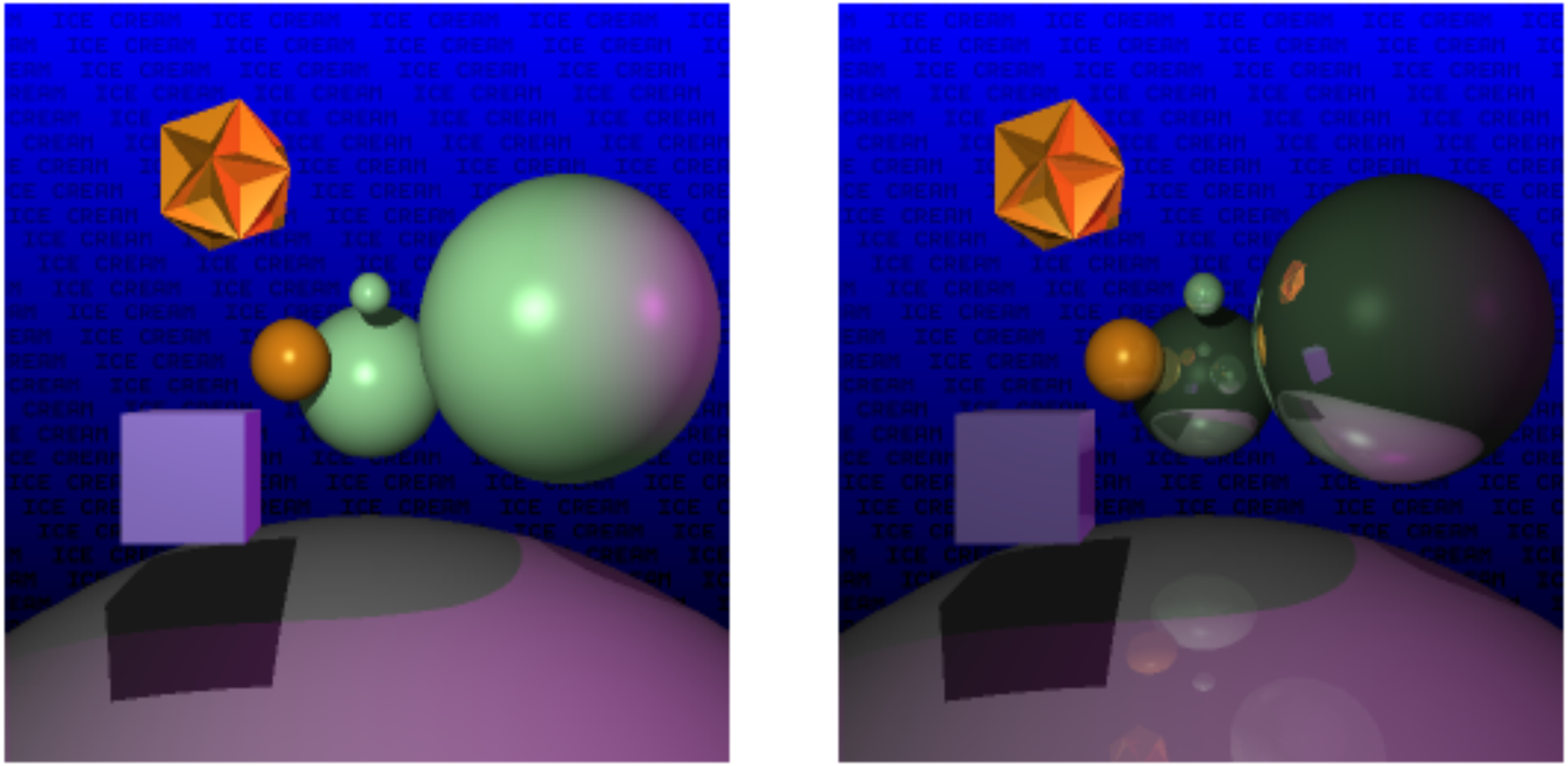


1 recursion



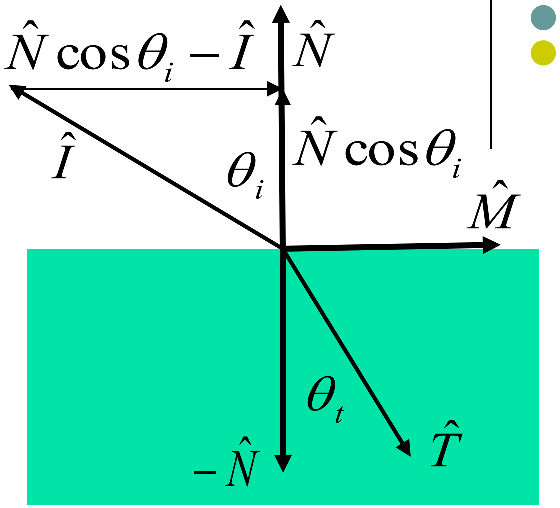
2 recursions

Reflection



Refraction

Snell's Law $\frac{\sin\theta_t}{\sin\theta_i} = \frac{\eta_i}{\eta_t} = \eta_r$



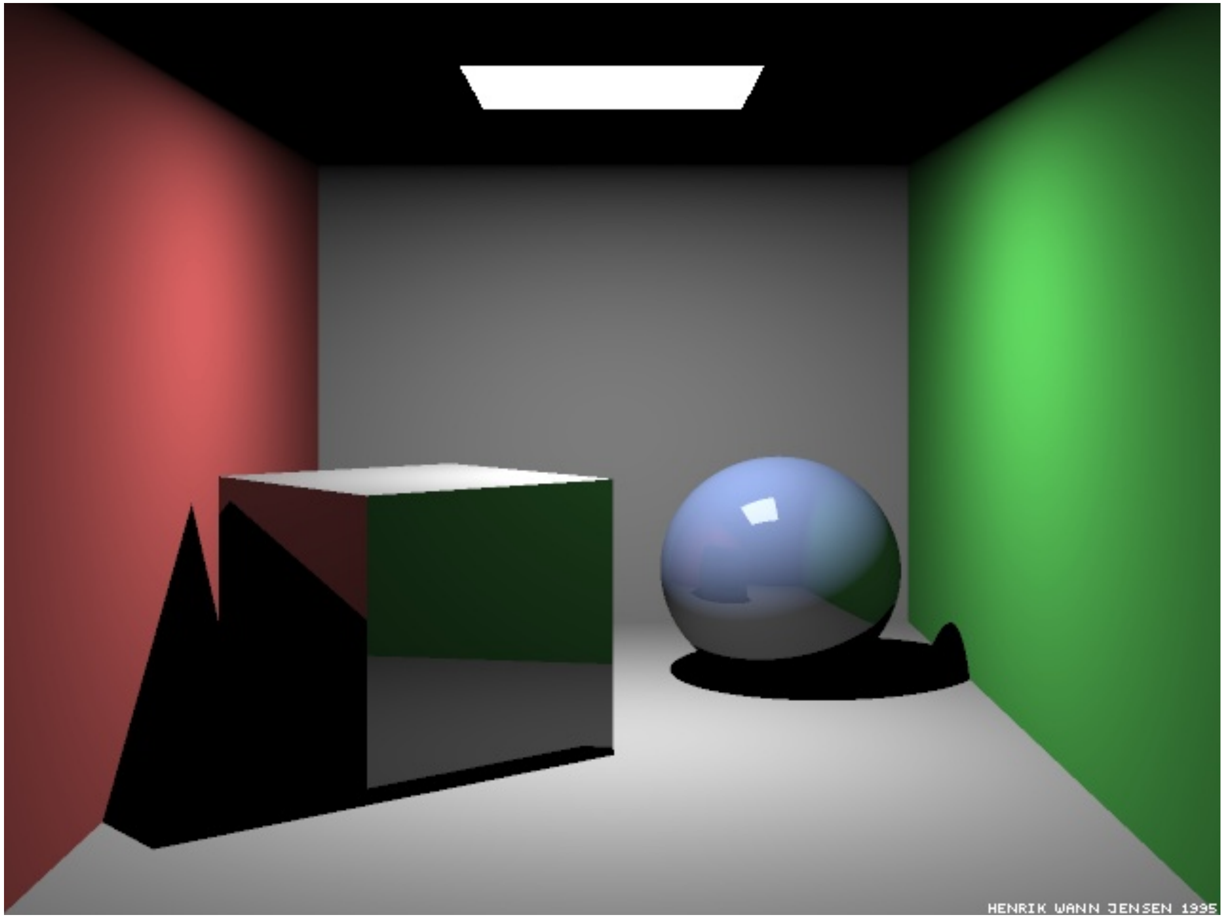
Note that I is the negative of
the incoming ray

Pseudo Code for Ray Tracing

```
rgb lsou;        // intensity of light source
rgb back;        // background intensity
rgb ambi;        // ambient light intensity

Vector L         // vector pointing to light source
Vector N         // surface normal
Object objects [n] //list of n objects in scene
float Ks [n]     // specular reflectivity factor for each object
float Kr [n]     //  refractivity index for each object
float Kd [n]     // diffuse reflectivity factor for each object
Ray r;

void raytrace() {
   for (each pixel P of projection viewport in raster order) {
        r = ray emanating from viewer through P
        int depth = 1; // depth of ray tree consisting of multiple paths
        the pixel color at P = intensity(r, depth)
   }
}
```

```
rgb intensity (Ray r, int depth) {
   Ray  flec, frac;
   rgb   spec, refr, dull, intensity;

    if (depth >= 5) intensity = back;
    else {
        find the closest intersection of r with all objects in scene
        if (no intersection) {
            intensity =back;
        } else {
            Take closest intersection which is object[j]
            compute normal N at the intersection point
            if (Ks[j] >0)   {  // non-zero specular reflectivity
                  compute reflection ray flec;
                  refl = Ks[j]*intensity(flec, depth+1);
            } else refl =0;
            if (Kr[j]>0)  {     // non-zero refractivity
                 compute refraction ray frac;
                 refr = Kr[j]*intensity(frac, depth+1);
            } else refr =0;
            check for shadow;
            if (shadow) direct = Kd[j]*ambi
            else direct = Phong illumination computation;
            intensity = direct + refl +refr;
       } }
     return intensity; }
```

Raytraced Cornell Box



Which paths
are missing?

Ray-traced Cornell box, due to Henrik Jensen, http://www.gk.dtu.dk/~hwj

Paths in RayTracing

- Ray Tracing
  - Captures LDS*E paths: Start at the eye, any number of specular bounces before ending at a diffuse surface and going to the light
- Raytracing cannot do:
  - LS^*D^+E: Light bouncing off a shiny surface like a mirror and illuminating a diffuse surface
  - LD^+E: Light bouncing off one diffuse surface to illuminate others
- Basic problem: The raytracer doesn't know where to send rays out of the diffuse surface to capture the incoming light
- Also a problem for rough specular reflection
  - Fuzzy reflections in rough shiny objects
- Need other rendering algorithms that get more paths

A Better Rendered Cornell Box