Introducing Assignment 3: GLSL & Raytracing I

COS 426: Computer Graphics (Spring 2020)

Reilly Bova

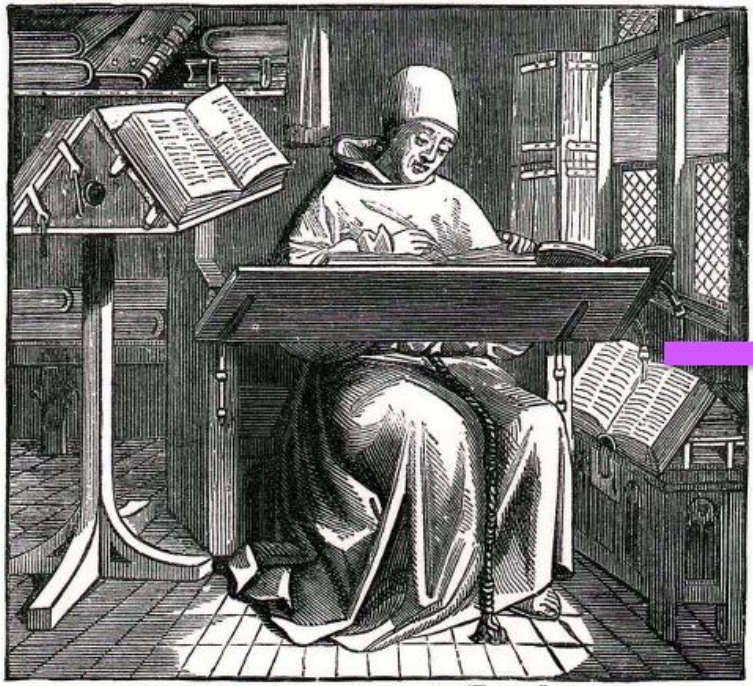
Agenda

- GLSL
  - What is a GPU?
  - What is a Shader?
  - What is GLSL?
  - GLSL Programming
  - GLSL Examples
- Raytracing
  - Background & Theory
  - Raytracing in Assignment 3
- Ray Intersections

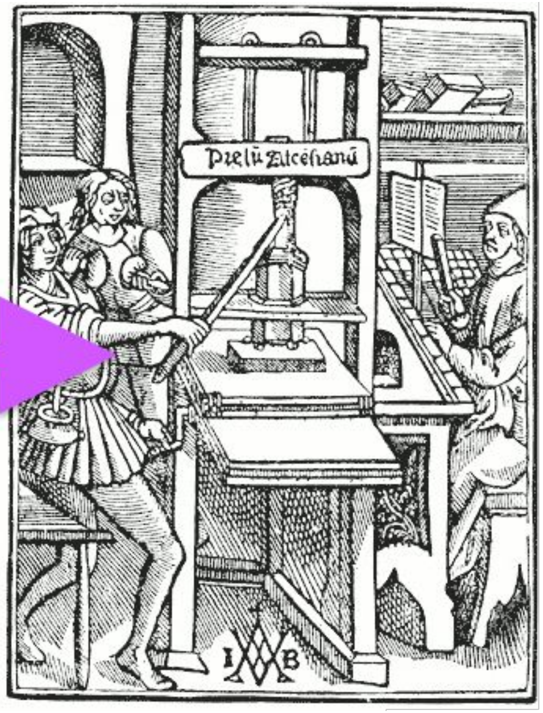
What is a GPU?

What is a GPU?

- A CPU is to a GPU, as a writer is to a printing press:



SCRIPTORIUM MONK AT WORK. (From *Lacroix.*)



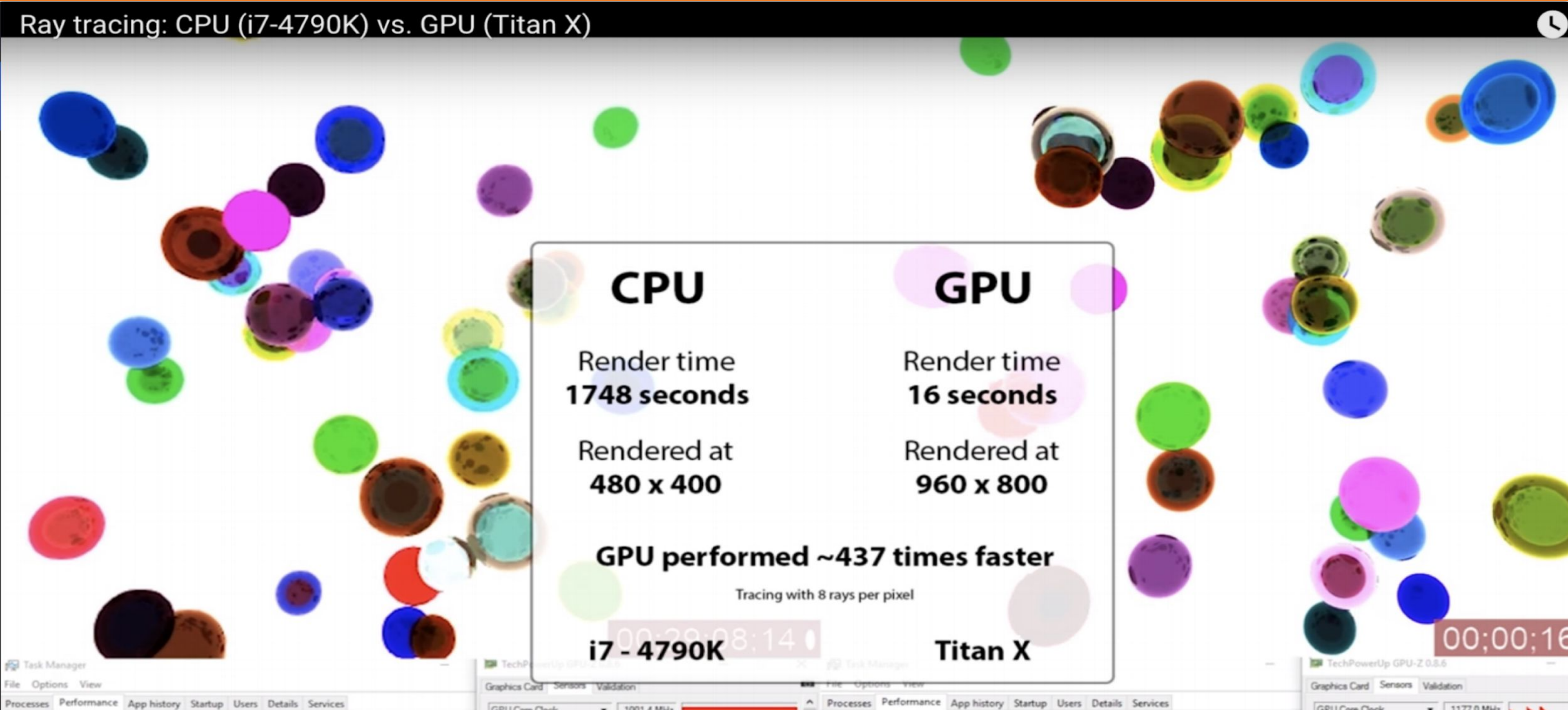
What is a GPU?

- A CPU contains a few powerful general processors that can each perform complex tasks.
    - CPU cores have a large memory bank (RAM)
    - CPU cores can execute complex machine instructions
    - CPUs can support modest parallelization via multithreading
    - Threads can communicate with each other via RAM, but this can cause trouble (take COS 318 for more)

What is a GPU?

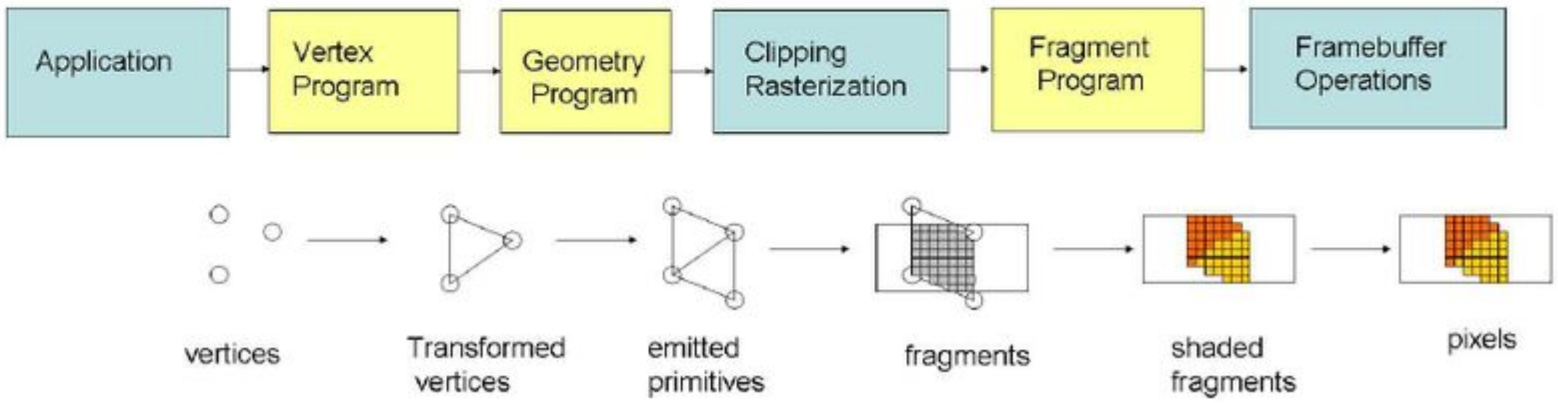
- A GPU can contain thousands of microprocessors that can only perform simple tasks.
  - GPU cores have a limited memory bank (VRAM)
    - VRAM has to store the frame buffer, textures, and processing data for each of the 1K+ cores (it's crowded). Thus, cores have limited memory.
  - GPU cores can only execute simpler instructions
  - GPU cores are **blind**: they cannot communicate with each other
  - GPU cores **forget**: they cannot remember previous frames
  - GPUs are designed for massive parallelization

What is a GPU?


Ray tracing: CPU (i7-4790K) vs. GPU (Titan X)
CPU
Render time
1748 seconds
Rendered at
480 x 400
i7 - 4790K
GPU
Render time
16 seconds
Rendered at
960 x 800
Titan X
GPU performed ~437 times faster
Tracing with 8 rays per pixel

What is a Shader?

- A **shader** is a program that executes on the GPU
- The yellow boxes in the following diagram of the OpenGL graphics pipeline are programmable shaders:



What is a Shader?

- **Vertex Shader**:
  - Automatically runs once per vertex
  - Project a vertex from 3D space to 2D space with a Z-depth using the camera
  - Must output the final vertex position and any attributes the fragment shader needs
- **Fragment Shader:**
  - Automatically runs once per rasterization fragment (think of this as a pixel)
  - Has access to certain attributes provided by the GPU and vertex shader
  - Must output a final pixel color
- **Geometry Shader:**
  - Optional, but it can modify geometries and even add vertices

What is GLSL?

- GLSL = Open **G**raphics **L**ibrary **S**hader **L**anguage
  - Part of the OpenGL specification
  - Adapted for browsers as WebGL
- GLSL is a **C/C++ flavoured language** with more type safety and no recursion; it executes on the GPU
- GLSL is used to write **shader programs**, which are used by OpenGL applications to render graphics

What is GLSL?

- What’s missing from C in GLSL syntax: “C \ GLSL”
  - **No Recursion** => You must unroll recursive functions into loops
  - **No Implicit Casting** => You must explicitly cast everything
  - **No Libraries** => You must write/provide all the code yourself
  - **No Dynamic Memory** => No heap! All memory is static
  - **No Pointers** => Yay?
  - **No Objects** (but there are structs)
  - **No char**
  - **No string**
  - **No I/O** => No trace statements!

What is GLSL?

- GLSL syntax extensions: “GLSL \ C”
  - **Storage qualifiers: `varying`, `uniform`, & `attribute`**
  - **Parameter qualifiers: `in`, `out`, & `inout`**
  - **Variable types: `vecN`, & `matN`**
    - Vectors and Matrices, respectively, e.g: `vec2`, `vec3`, `mat4`, ...
    - Standard math operators (`+`, `-`, `*`, `/`) are applied component-wise.
    - **swizzling:** `vec3 yxz_comp = some_vec3.yxz;`
  - **Polymorphic builtins: `max`, `min`, `sqrt`, `dot`, `cross`, ...**
  - **Predefined variables: `gl_*`**
    - `gl_Position`
    - `gl_FragCoord`
    - `gl_FragColor`, `gl_FragData[]`

GLSL Programming

- **`uniform`** (i.e. *Dynamically Uniform*):
  - **Read-only** and statically **shared** between all vertices and fragments
  - Similar to global variables in C; **set by the application** and then passed into the vertex and fragment shaders
  - Common use: informing the shaders of the lights and objects in the scene
- **`varying`**:
  - Variables **set by the GPU** (so it does the heavy lifting)
  - **Per-vertex outputs** in the vertex shader
  - **Automatically interpolated** between triangle vertices by the GPU and passed as per-pixel inputs to the fragment shader
  - Varying variables are **written by the vertex shader** and **read by the fragment shader**
  - Used to pass information from the vertex shader to the fragment shader

GLSL Programming

- `attribute`:
  - Values that are **unique per-vertex** and are **passed into the vertex shader**
  - Common uses: providing a vertex its position, color, and material

GLSL Programming

- The `in` parameter qualifier:
  - Argument value is **copied** into the function
  - This is the **default** if no qualifier is specified
  - "Copy and pass by value"
- The `out` parameter qualifier:
  - The function **cannot read** the argument, but it can **write** to the argument
  - Changes to the variable are visible (to the caller) **outside** of the function
  - "Pass by reference, but write-only"
- The `inout` parameter qualifier:
  - The function can **both read and write** to the argument
  - Changes to the variable are visible (to the caller) **outside** of the function
  - "Pass by reference"

GLSL Programming

- Parameter qualifiers example I:

```
void multiplyByTwo(inout float value) {
    value *= 2;
}

void main() {
    float t = 2;
    multiplyByTwo(t);
    // t is now 4
}
```

- `value` is an **inout** variable
- Function can **read** the variable
- Function can **modify** the variable

GLSL Programming

- Parameter qualifiers example II:

```
float findIntersectionWithPlane(Ray ray, vec3 norm, float dist,
                                out Intersection intersect) {
  float a = dot(ray.direction, norm);
  float b = dot(ray.origin, norm) - dist;

  if (a < EPS && a > -EPS)
    return INFINITY;

  float len = -b / a;
  if (len < EPS)
    return INFINITY;

  intersect.position = rayGetOffset(ray, len);
  intersect.normal = norm;
  return len;
}
```

- `intersect` is an **`out`** variable
- Function **cannot read** the variable
- Function can **modify** the struct directly (e.g. its `position` and `normal` fields)

GLSL Programming

- **vecN**: easy vector math

```
vec3 a = vec3(1.0, 2.0, 3.0);   // make a vec3
vec4 b = vec4(a, 1.0);          // make vec4 from vec3
vec3 c = b.xyz + a.zyx;         // add two vec3 together
vec3 d = 2.0 * c;               // mult vec3 by scalar
vec4 e; e.xyz = c; e[3] = b.w; // can use index or .{xyzw}
```

GLSL Programming

- Important built-in **`gl_*`** values:
  - **`gl_Position`**
    - The key vertex shader output (the vertex position)
  - **`gl_FragColor`**
    - The key fragment shader output (the pixel color)
  - **`gl_FragCoord`**
    - The pixel location in window space

GLSL Examples

- A Simple Vertex Shader

```
attribute vec2 my_position;
void main() {
    gl_Position = vec4(my_position, 0, 1);
}
```

GLSL Examples

- A Simple Fragment Shader

```
// What does this draw? (assume entire screen is rendered)
void main() {
    gl_FragColor = vec4(gl_FragCoord.x / canvas_width,
                        gl_FragCoord.y / canvas_height,
                        0, 1
                    );
}
```

GLSL Examples

- A (Less) Simple Fragment Shader

```
bool inArea(float cX, float cY) {
     return (sqrt(cX*cX + cY*cY) < 80.0);
}

// What does this draw? (assume entire screen is rendered)
void main() {
     float cX = gl_FragCoord.x - width/2.0;
     float cY = gl_FragCoord.y - height/2.0;
     if (inArea(cX, cY)) {
          gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
     } else {
          gl_FragColor = vec4(0.0, 0.0, 0.0, 1.0);
     }
}
```

GLSL Examples

- Here are some cool examples of complex shaders:
    - An Ocean
    - A Flame
    - A Snail
    - Intra-nebular Space
    - Voxels
    - A Rainforest
    - Zoom's #1 Profit Driver This Quarter and the Source of My Despair
    - Raytraced Cornell Box with Global Illumination*
    - Raytraced Scene with Advanced Materials*

**These are advanced versions of A3.*

DO NOT INSPECT "BUFFER A" CODE UNTIL FINISHED ASSIGNMENT 3

Raytracing

Raytracing: A Background

- Traced back to techniques of 16th century artist Albrecht Dürer:

Raytracing: A Background

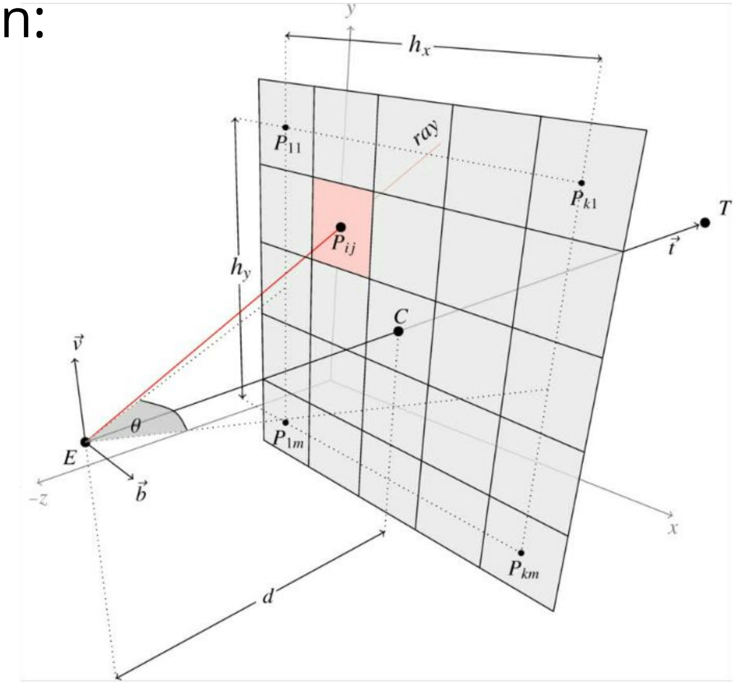
- Now the standard technique for rendering CGI and 3D animations
  - First fully raytraced film was *Monster House* (2006)
  - Earlier 3D feature films (like *Toy Story*) only used rasterization (next assignment)
- Video games, which are generally rasterized, are also now incorporating raytracing
  - See Nvidia's "RTX on" videos

Raytracing: Theory

- The goal of raytracing is to approximate the physics of light as closely as possible (just need to trick the eye)
  - See also: electromagnetism and quantum electrodynamics
  - A full simulation will never be feasible, and many real-world effects have to be ignored; the only known simulator of all known electromagnetic effects at all wavelengths at all positions in time is the Universe
- Key insight: a photon's path obeys **time-symmetry**
  - Shooting a ray from where a photon expires will bounce back **along the photon's path** back to where it originated
  - Raytracing: shoot rays from the "eye/camera" to retrace photons

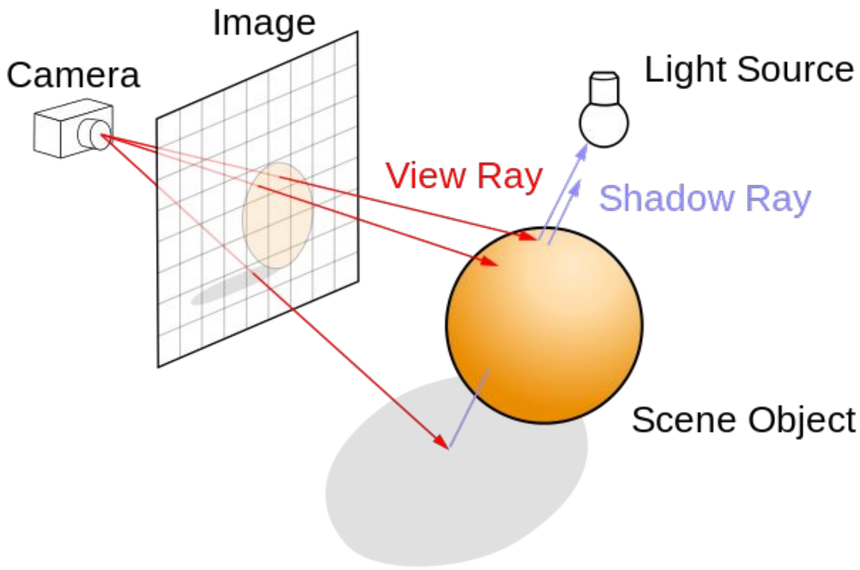
Raytracing: Theory

- *Raycasting* analogy: your eye "looking" through the pixels of your computer screen:



Raytracing: Theory

- A common optimization is to only look at the first intersection of each ray in the scene:
  - Photons lose a lot of energy after the first bounce
  - Assume almost all radiance at an intersection comes directly from the light
  - “Direct Illumination”

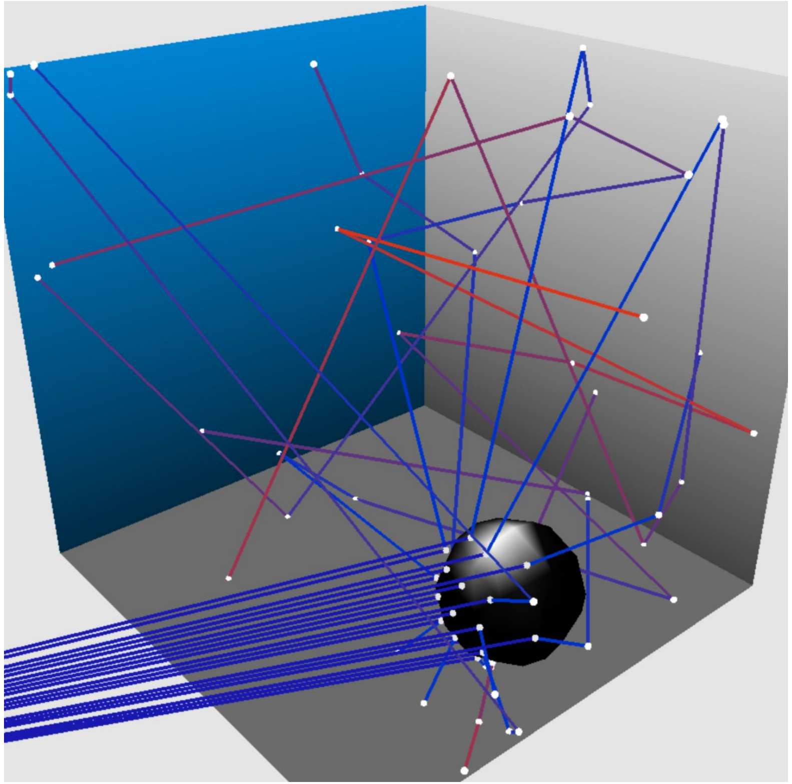


Raytracing in Assignment 3

- You will implement *Direct Illumination* (DI) for your Assignment 3 raytracer
  - Scenes won't look photorealistic, but they'll be fast and sharp
  - Your eye will be somewhat tricked
  - Some advanced techniques (not required for A3) next week
- Certain DI intersections still need raycasting recursion
  - Reflections (mirror bounce)
  - Refractions (refractive bounce)
  - Formulae for bounces drawn from electromagnetism (optics)

Raytracing in Assignment 3

- Here is a visualization of paths traced for a scene with a mirror ball in a mirror box
  - Paths are terminated when they leave through the open face of the box
  - Color of ray warms with each bounce



Raytracing in Assignment 3

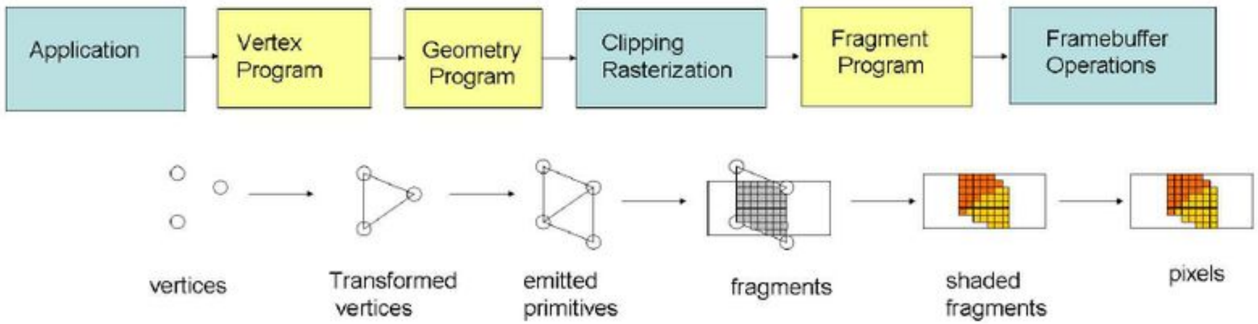
- How do we recur without recursion?

```
#define MAX_RECURSION 10

function g() {
	float x = 0.0, weight = 1.0, res = 0.0;
	float cur_contrib;

	for (int i = 0; i < MAX_RECURSION; i++) {
		cur_contrib = f();
		res = res + weight * cur_contrib;
		weight = weight * 0.8;
	}

	return res;
}
```

- Use a loop!
- This is known as "unrolling" recursion
- Any recursive function can be unrolled into a tail-recursive procedure like this

Raytracing in Assignment 3

- How are we **raytracing** with a **shader** program?
    - Think of the rendered scene as a large rectangle made up of 2 triangles
        - There are 4 vertices in total (2 are shared between the 2 triangles)
    - The fragment shader operates on each of the pixels inside this rectangle and computes that pixel's color
        - NB: each pixel's position was interpolated from the original 4 vertices!
    - The resulting color for each pixel is what we get from tracing a ray for the corresponding ”pixel” in the camera!



Raytracing in Assignment 3

- Raytracing in a Fragment Shader

```
void main() {
  float cameraFOV = 0.8;
  vec3 direction = vec3(v_position.x * cameraFOV * width / height, v_position.y * cameraFOV, 1.0);

  Ray ray;
  ray.origin = vec3(uMVMatrix * vec4(camera, 1.0));
  ray.direction = normalize(vec3(uMVMatrix * vec4(direction, 0.0)));

  // trace the ray for this pixel
  vec3 res = traceRay(ray);

  // paint the resulting color into this pixel
  gl_FragColor = vec4(res.x, res.y, res.z, 1.0);
}
```

Raytracing in Assignment 3

- No console I/O or breakpoints makes traditional debugging techniques ineffective
- Instead, you must do **visual debugging** which is simply creative use of the one shader output you have: **the pixel color**
- Some simple suggestions:
  - Output red for sphere, yellow for triangle, green for cylinder, etc.
  - Output the normal vector of the surface directly.
  - `if` (some_condition) then GREEN `else` normal shading.
    - This can track down which pixels are problematic.
  - Move around in the scene! The real-time performance of the raytracer for A3 is a huge asset and real treat. Leverage it!

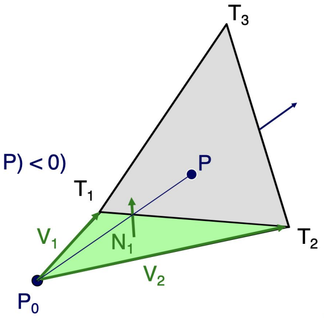
Ray Intersections: Triangle

- There are many algorithms for testing ray intersections with a triangle
  - The industry standard is Möller-Trumbore. **Do not read code for this algorithm if you choose to attempt it**.
  - Other algorithms use a plane-intersection test, and then check if the point of intersection lies within the provided triangle (recommended).
  - Lecture 11 gives three algorithms — use any!

Ray-Triangle Intersection I

- Check if point is inside triangle algebraically

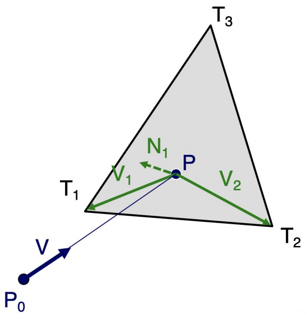
For each side of triangle
  $V_1 = T_1 - P_0$
  $V_2 = T_2 - P_0$
  $N_1 = V_2 \times V_1$
  Normalize N_1
  Plane $p(P_0, N_1)$
  if (SignedDistance(p, P) < 0)
    return FALSE
end
return TRUE



Ray-Triangle Intersection II

- Check if point is inside triangle algebraically

For each side of triangle
  $V_1 = T_1 - P$
  $V_2 = T_2 - P$
  $N_1 = V_2 \times V_1$
  if ($V \cdot N_1 < 0$)
    return FALSE
end
return TRUE



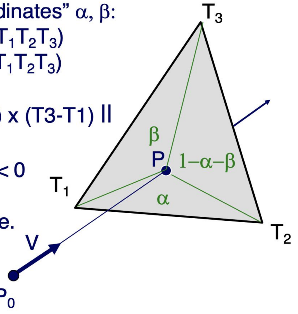
Ray-Triangle Intersection III

- Check if point is inside triangle parametrically

Compute "barycentric coordinates" α, β:
  $\alpha = Area(T_1T_2P) / Area(T_1T_2T_3)$
  $\beta = Area(T_1PT_3) / Area(T_1T_2T_3)$

$Area(T_1T_2T_3) = ½ \| (T2-T1) \times (T3-T1) \|$
  check if backfacing:
  $((T2-T1) \times (T3-T1)) \cdot N < 0$

Check if point inside triangle.
  $0 \leq \alpha \leq 1$ and $0 \leq \beta \leq 1$
  and $\alpha + \beta \leq 1$

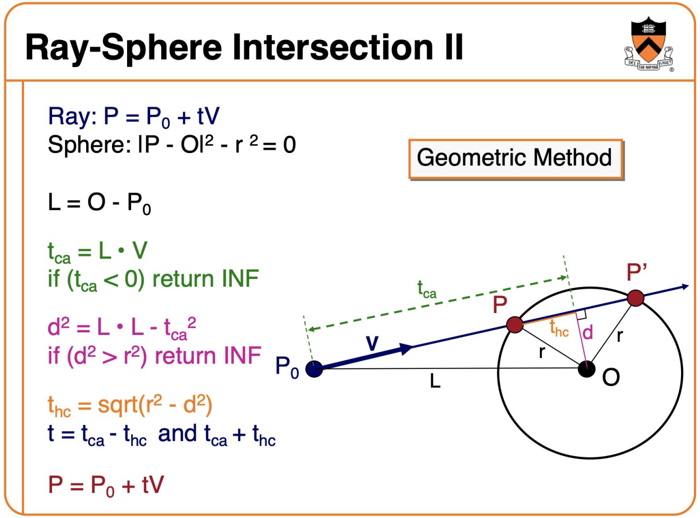


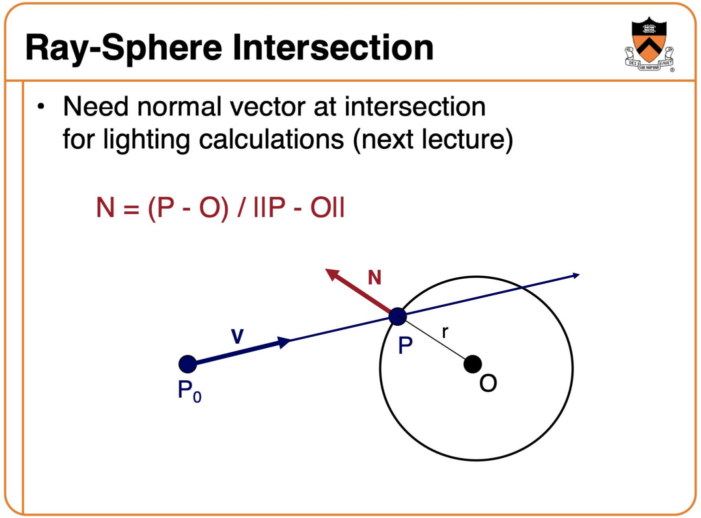
Ray Intersections: Sphere

- Need to be careful to return the *nearest* closest intersection

```
t1 = tca - thc; t2 = tca + thc;
if (t1 > 0) return t1; else if (t2 > 0) return t2;
else return INFINITY;
```

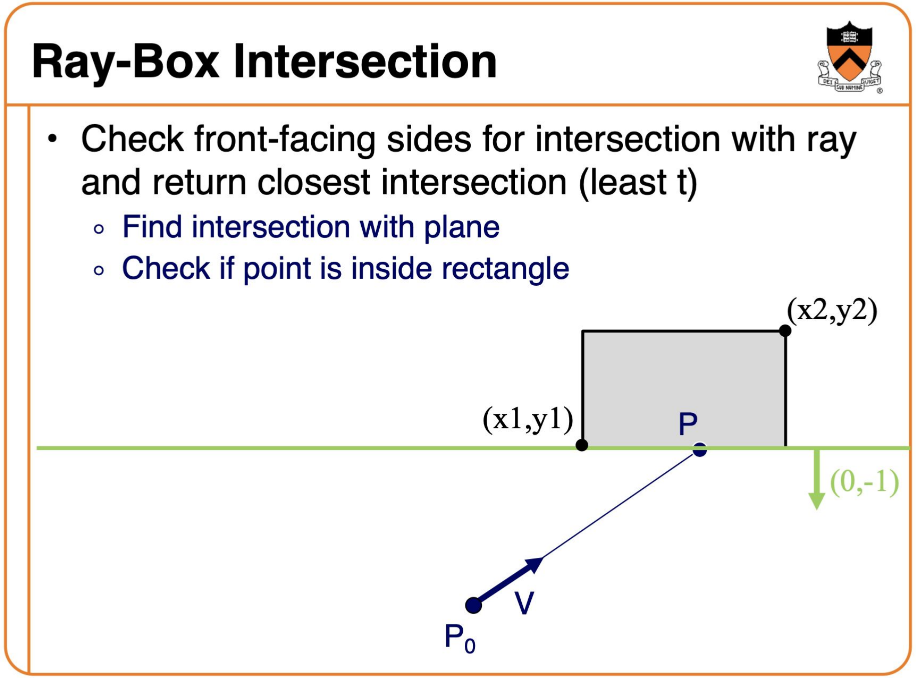
- Also need to compute the normal at the intersect for lighting





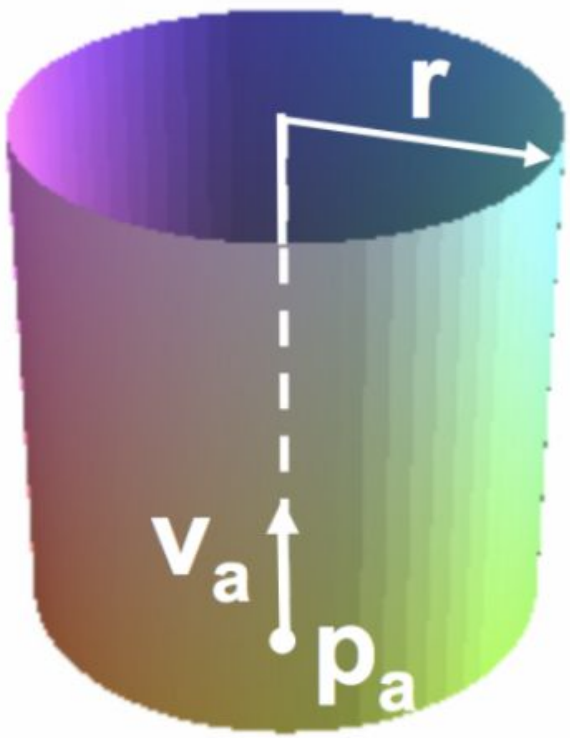
Ray Intersections: Box

- Treat each side of the face as a plane
- Intersect the ray with each plane separately
- Filter out intersections that do not lie on the box
  - This is easy because the box is axis-aligned
- Return the closest intersection, if one exists



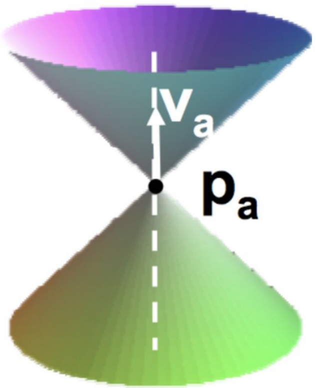
Ray Intersections: Closed Cylinder

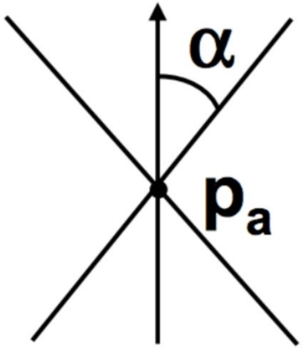
- A closed cylinder is an open cylinder with two caps (discs)
- First intersect an open cylinder of fixed height
- Then intersect the two discs
- Out of all intersections, choose the nearest
- Refer to the assignment specs to guide your solution (and math)



Ray Intersections: Closed Cone

- Similar to a closed cylinder
- A closed cone is an open cone with one cap
- First intersect an open cone (half of a finite double cone)
- Then intersect the cap (disc)
- Out of all intersections, choose the nearest
- Refer to the assignment specs to guide your solution (and math)





Q&A