A3 Intro: Raytracing & GLSL

COS 426 Spring 2019
Austin Le & Jiaqi Su

Adapted from Spring 2018
Originally created by Kyle Genova

What is Raycasting?

(1) Trace primary rays into the scene.
(2) Intersect with an object.
(3) Estimate the radiance by summing contribution from each unblocked light to that point.

Raycasting produces results that only account for **direct illumination**!

(see “Lighting & Reflectance” lecture for more details)

What is Ray Tracing?

Raycasting, but trace secondary rays for specular (mirror) reflection and refraction from point of intersection, if appropriate.

This is recursive!

(see "Lighting & Reflectance" lecture for more details)

What is GLSL?

It’s the Open

Graphics

Library

Shader

Language

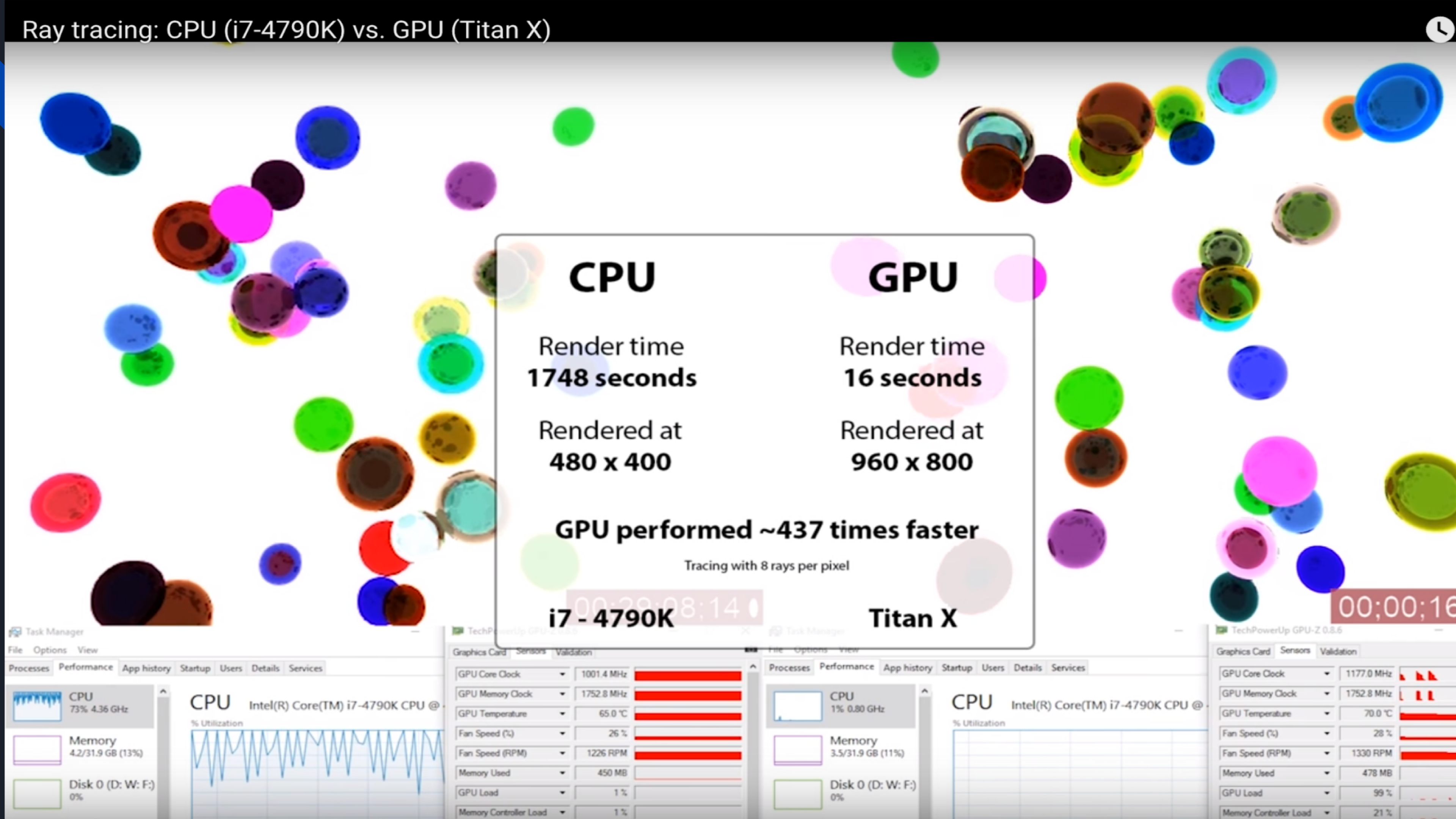
(Open**GL SL**)

What is GLSL?

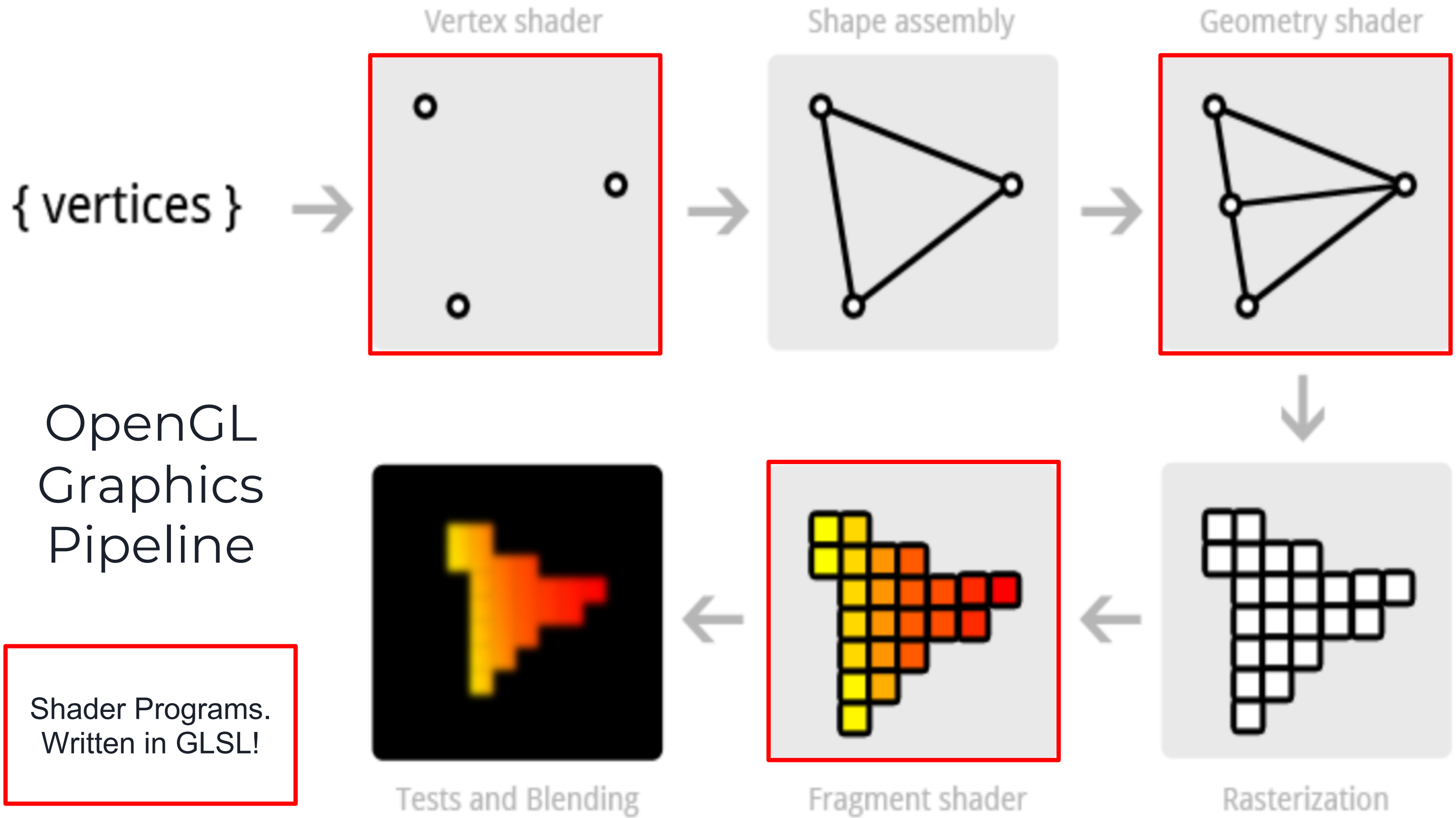
A C-like language (syntactically) with more type safety and no recursion that executes code directly on the GPU.

It is used to write shader programs, which are used by OpenGL applications to render graphics.

Why do we want it for Ray Tracing?


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
00;00;16
Task Manager
Processes
Performance
App history
Startup
Users
Details
Services
CPU
73% 4.36 GHz
Memory
4.2/31.9 GB (13%)
Disk 0 (D: W: F:)
0%
CPU
Intel(R) Core(TM) i7-4790K CPU @
% Utilization
Graphics Card
Sensors
Validation
GPU Core Clock
1001.4 MHz
GPU Memory Clock
1752.8 MHz
GPU Temperature
65.0 °C
Fan Speed (%)
26 %
Fan Speed (RPM)
1226 RPM
Memory Used
450 MB
GPU Load
1 %
Memory Controller Load
1 %
CPU
1% 0.80 GHz
Memory
3.5/31.9 GB (11%)
Disk 0 (D: W: F:)
0%
CPU
Intel(R) Core(TM) i7-4790K CPU @
% Utilization
TechPowerUp GPU-Z 0.8.6
Graphics Card
Sensors
Validation
GPU Core Clock
1177.0 MHz
GPU Memory Clock
1752.8 MHz
GPU Temperature
70.0 °C
Fan Speed (%)
28 %
Fan Speed (RPM)
1330 RPM
Memory Used
478 MB
GPU Load
99 %
Memory Controller Load
21 %

OpenGL Graphics Pipeline



Shader Programs. Written in GLSL!

Two Critical Components: Vertex & Fragment Shaders

- **Vertex Shader**: Runs automatically once per vertex. Must output the final vertex position and any attributes the fragment shader needs.
- **Fragment Shader**: Runs automatically once per pixel (AKA fragment). Runs after the vertex shader. Must output the final pixel color.

- (**Note:** Geometry Shader is an optional stage)

What’s Missing in GLSL Syntax: C \ GLSL

- Recursion
- Implicit Casting
- Libraries
- Dynamic memory allocation
- Pointers
- Objects
- Char, String
- Console I/O ?!

GLSL Syntax Extensions: GLSL \ C

- `varying`
- `uniform`
- `attribute`
- Parameter qualifiers: `in, out, inout`
- `vecN`
  - 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[]`

Uniform (AKA Dynamically Uniform)

Uniform variables are read-only and statically shared between all vertices and fragments.

Similar to global variables in C, which can be modified and set by the application and then passed into the vertex and fragment shaders.

Common uses: informing the vertex and fragment shaders of the lights and objects in the scene.

Varying: The GPU does the heavy lifting

Varying variables are per-vertex outputs in the vertex shader.

They are **automatically interpolated** between triangle vertices by the GPU and passed as per-pixel inputs to the fragment shader.

Varying variables are <u>written by the vertex shader</u> and <u>read by the fragment shader</u>. Used to pass information from the vertex shader to the fragment shader.

Attribute: Vertex Shader Only

Attributes are values that are unique per-vertex and are passed into the vertex shader.

Common use: providing a vertex its position or color

vecN: Easier vector math

```
// N = {2, 3, 4}

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}
```

Parameter qualifiers: **in**, **out,** and **inout**

Qualifier	Meaning
`in`	Variable value is copied into the function. This is the default if no qualifier is specified. (“copy and pass by value”)
`out`	Function cannot read the variable, but can write to the variable. The changes are visible outside of the function. (“pass by reference, but write-only”)
`inout`	Function can both read and write to the variable. The changes are visible outside of the function. (“pass by reference”)

Parameter qualifiers: **in**, **out,** and **inout**

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

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

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

Parameter qualifiers: **in**, **out,** and **inout**

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

```
// Plane
// this function can be used for plane, triangle, and box
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;
}
```

gl_Position and other **gl_*** values: Built-ins

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.

A Simple Vertex Shader

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

A Simple Fragment Shader

```
void main() {
  gl_FragColor = vec4(gl_FragCoord.x / canvas_width,
                      gl_FragCoord.y / canvas_height,
                      0, 1);
}
```

A (Less) Simple Fragment Shader

```
void main() {
  float cX = gl_FragCoord.x – width/2.0;
  float cY = gl_FragCoord.y – height/2.0;
  if (sqrt(cX*cX + cY*cY) < 80.0){
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
  } else {
    gl_FragColor = vec4(0.0, 0.0, 0.0, 1.0);
  }
}
```

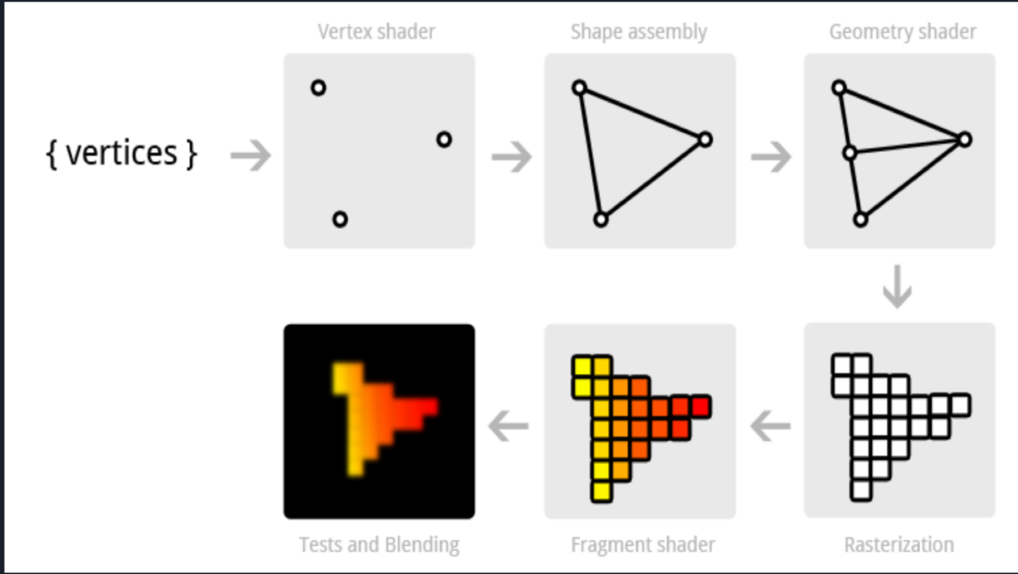
How to Avoid Recursion in a Recursive Ray Tracer

```
#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;
}
```

So how are we doing raytracing with a shader program?

- Think of the rendered scene in your browser 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.**
- (Note that each pixel's position was interpolated from the original 4 vertices!)

- … What is that color?
- **It's what we get from tracing a ray for the corresponding ”pixel” in the camera!**



The OpenGL Graphics Pipeline

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);
}
```

Visual Debugging

No console IO or breakpoints makes traditional debugging techniques ineffective. Instead, you must do “visual debugging,” which is simply creative use of the one fragment 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 is a huge asset.

Additional Learning Resources

https://www.opengl.org/wiki/Core_Language_%28GLSL%29

http://www.shaderific.com/glsl-qualifiers

See assignment FAQ for more!

We use WebGL (which is an implementation of the OpenGL ES 2.0 specification) to run our raytracer in the browser.

It uses GLSL ES 1.00!