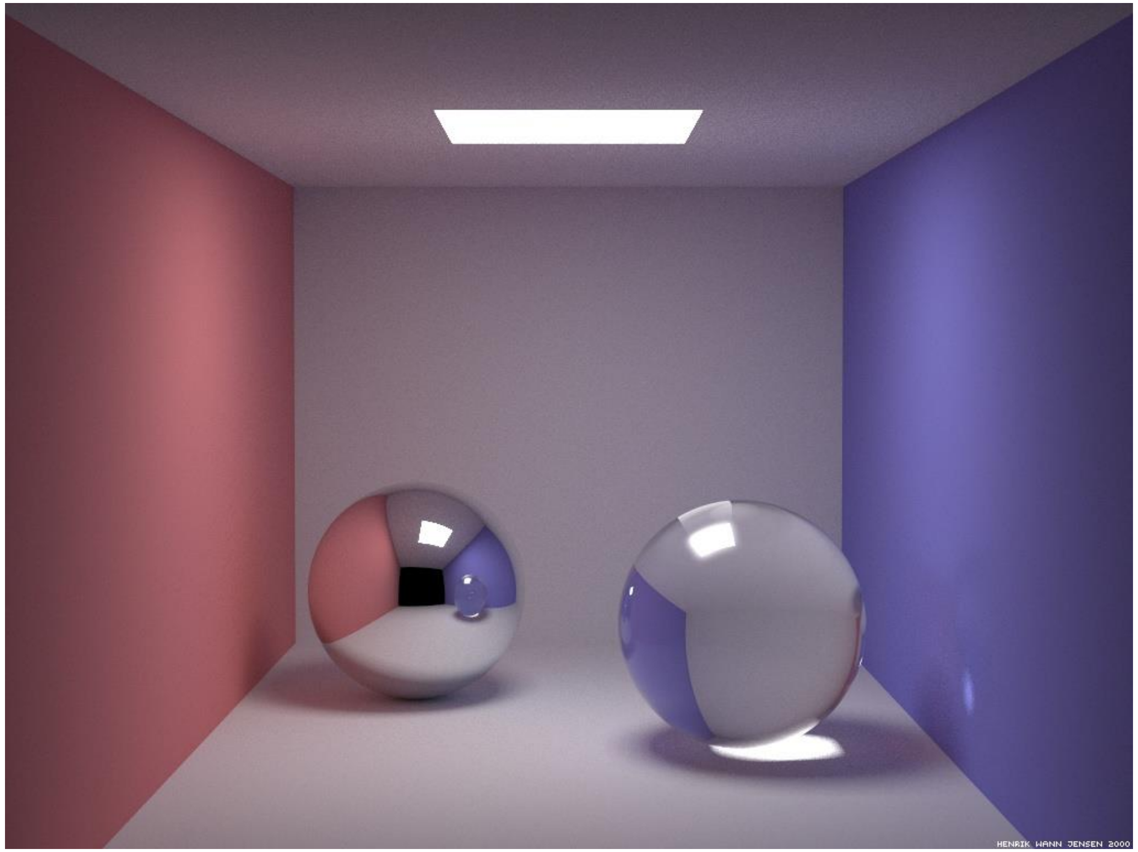
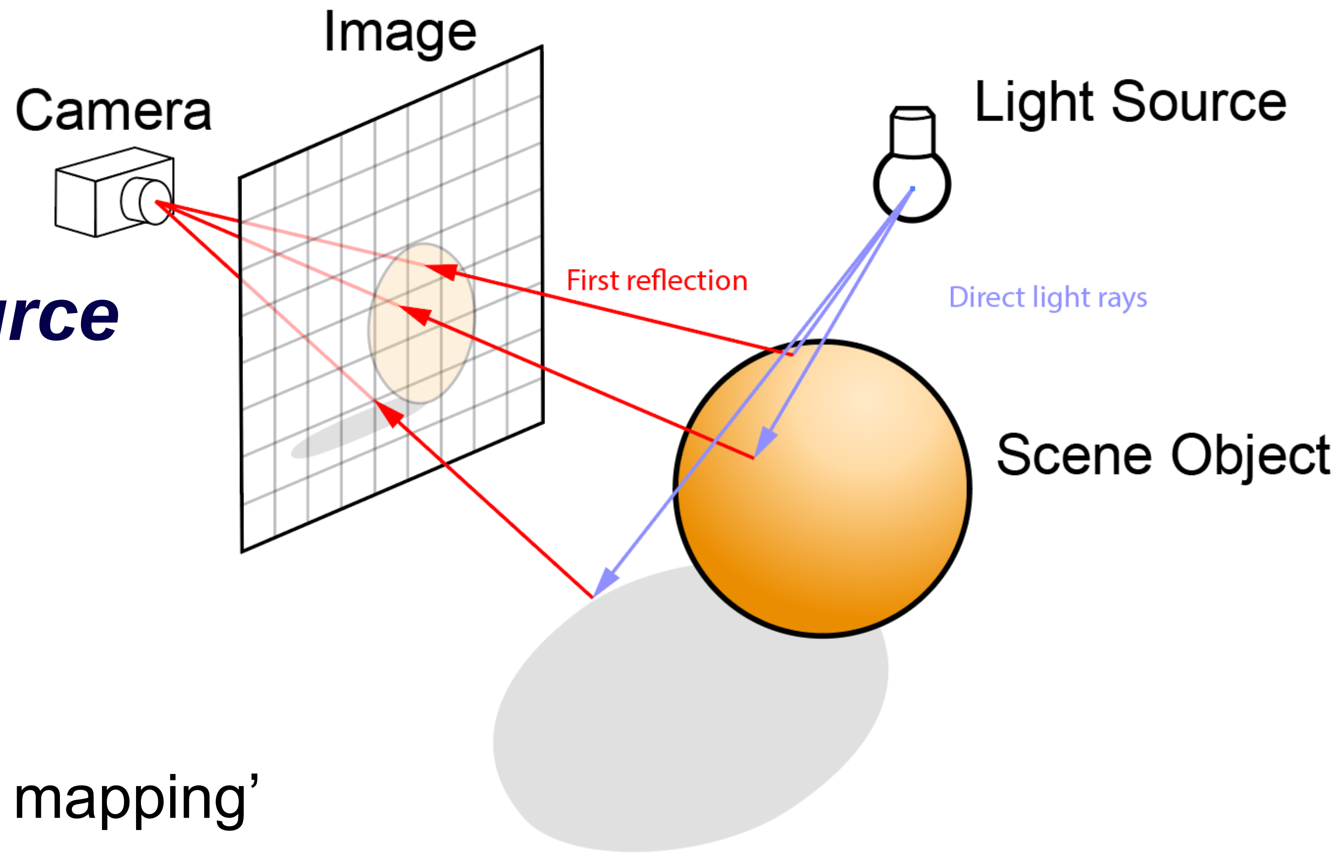
Course 3: Rendering Pipeline and OpenGL



Rendering – Photon Tracing

- ***simulate physical light transport from a source to the camera***
  - *the paths of photons*



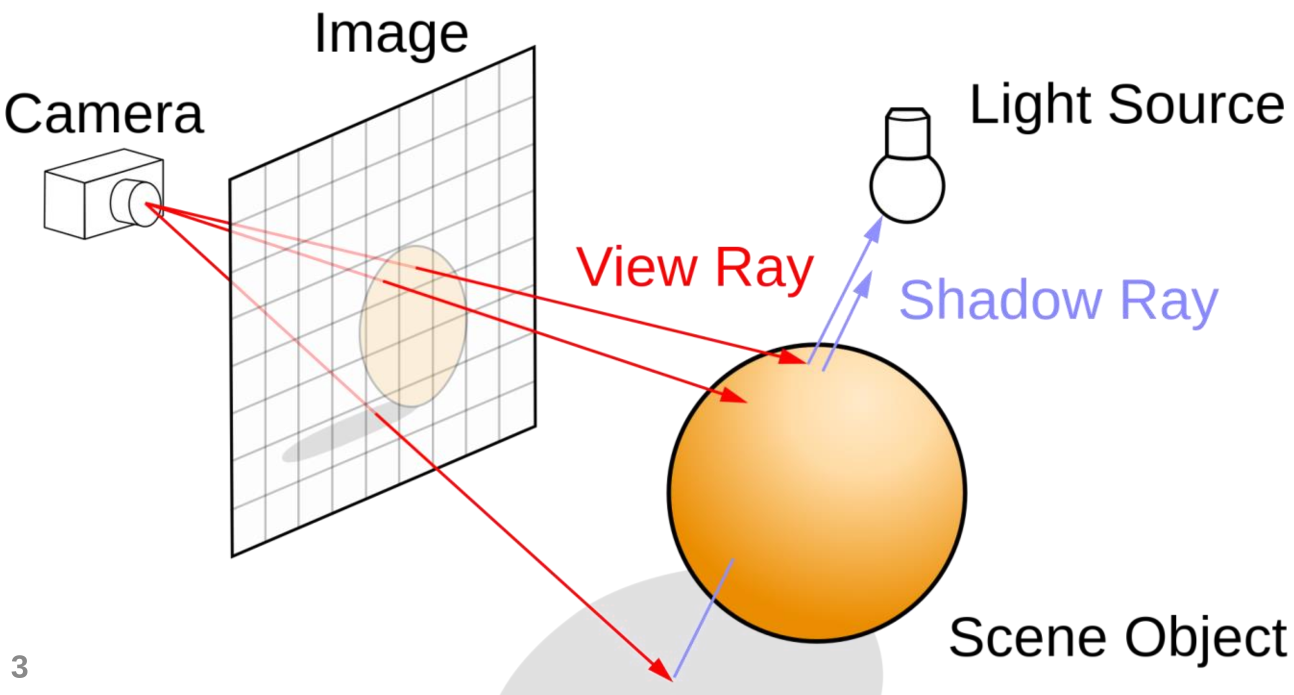
- ***shoot rays from the light source***
  - *random direction*
- ***compute first intersection***
  - *continue towards the camera*

- used for indirect illumination: ‘photon mapping’

Rendering – Ray Tracing

Start rays from the camera (opposes physics, an optimization)

- *View rays: trace from every pixel to the first occlude*
- *Shadow ray: test light visibility*



Nvidia RTX does ray tracing

Recap: Rendering – Rasterization

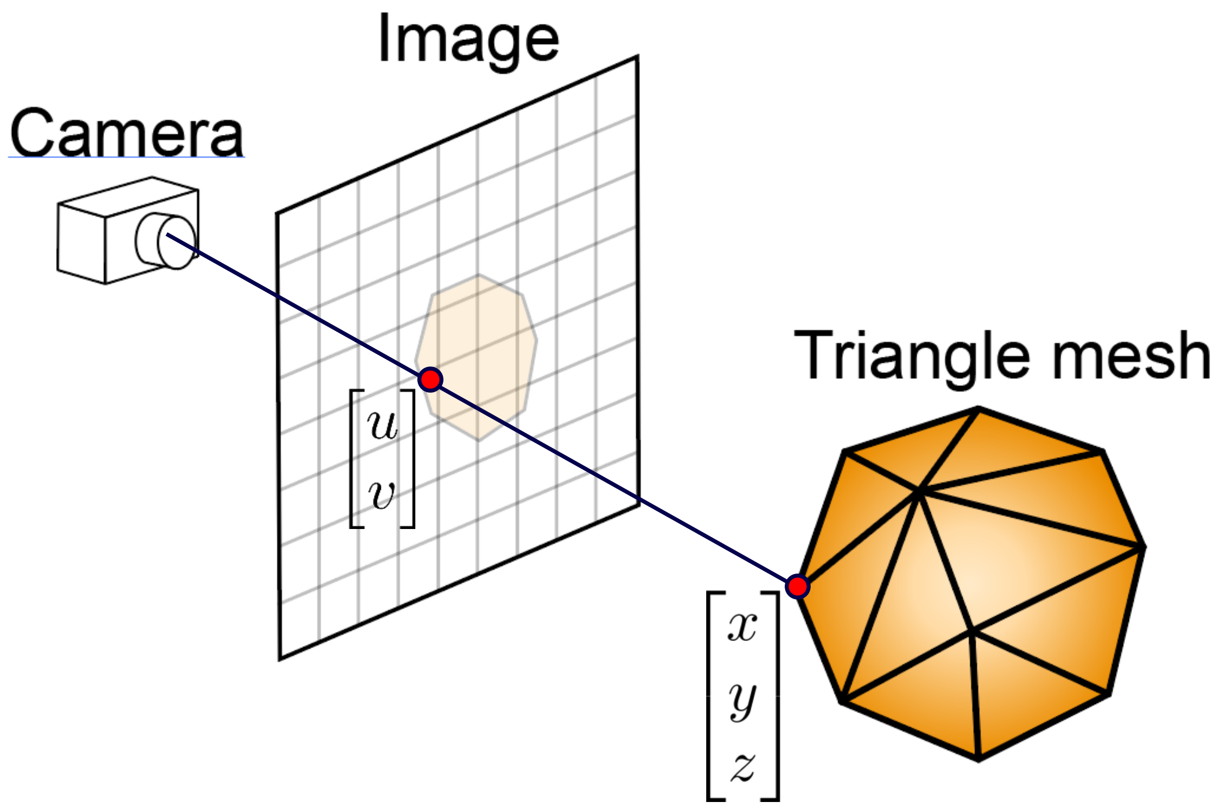
Approximate objects with triangles

1. *Project each corner/vertex*
   - projection of triangle stays a triangle

$$\begin{bmatrix} u \\ v \end{bmatrix} = \frac{1}{z} \begin{bmatrix} x \\ y \end{bmatrix}$$

   - O(n) for n vertices

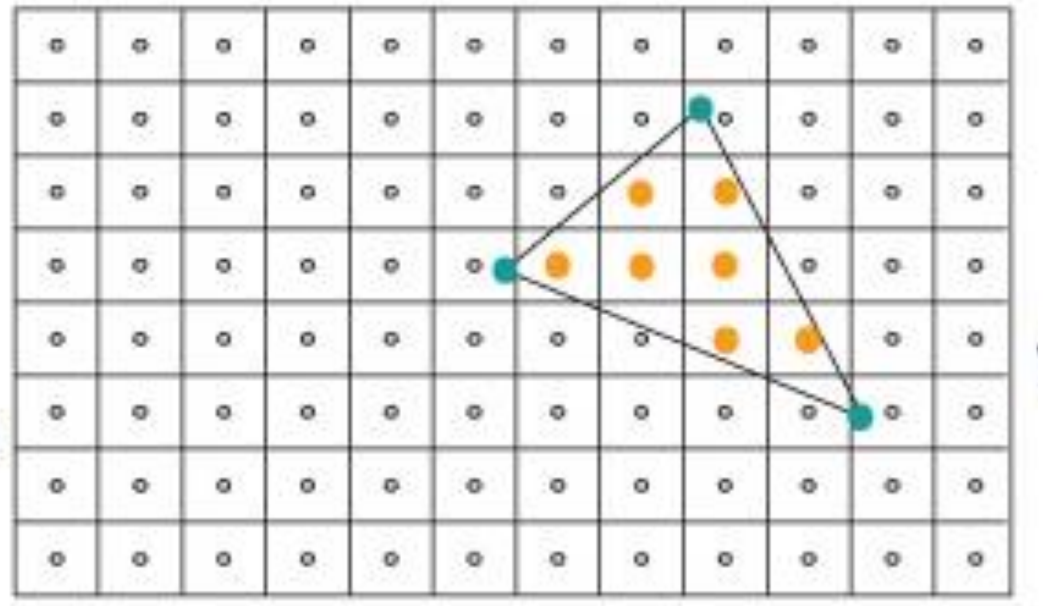
2. *Fill pixels enclosed by triangle*
   - e.g., scan-line algorithm



Recap: Rasterizing a Triangle

- ***Determine pixels enclosed by the triangle***
- ***Interpolate vertex properties linearly***

Vertices



Fragments

(for every pixel; color or attributes to compute color: texture coordinate, direction, …)

Graphics processing unit (GPU)

Specialized hardware designed for rendering

- highly parallel architecture
- dedicated instructions
- hardware pipeline (parts are not programmable)

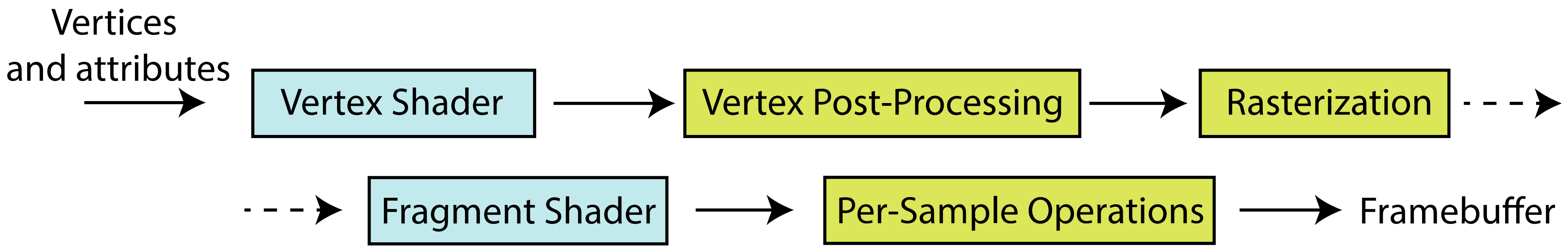
Proved useful for high-performance computing

- machine learning
- bitcoin mining
- …

OpenGL Rendering Pipeline

Input:

- *3D vertex position*
- *Optional vertex attributes: color, texture coordinates,…*



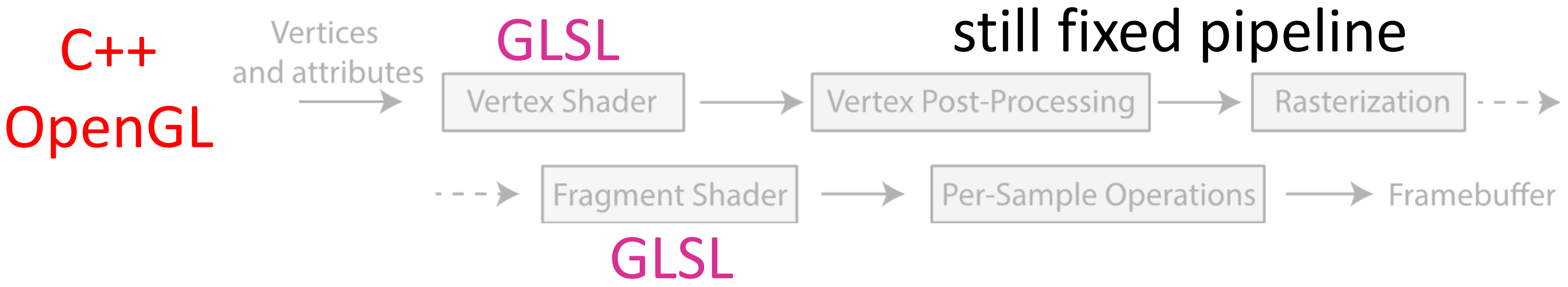
Output:

- ***Frame Buffer :*** *GPU video memory, holds image for display*
- *RGBA pixel color (Red, Green, Blue, Alpha / opacity)*

Programming languages

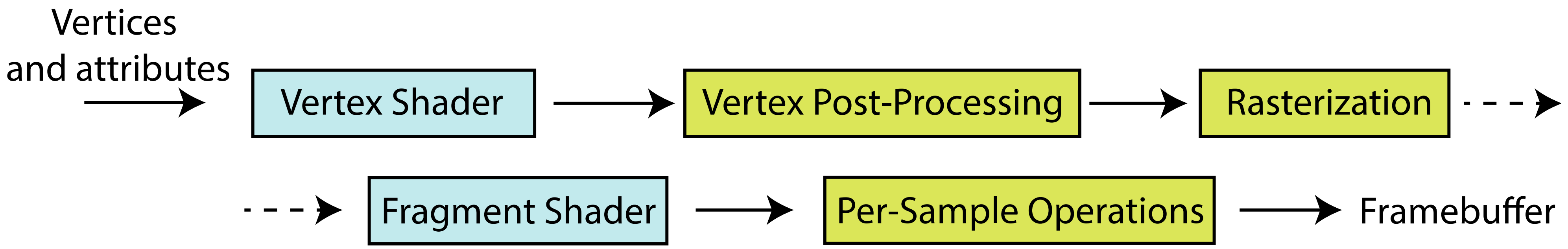
Traditionally, the entire pipeline was fixed (until ~2004)

- *vertex and fragment shaders now programmable with GLSL*



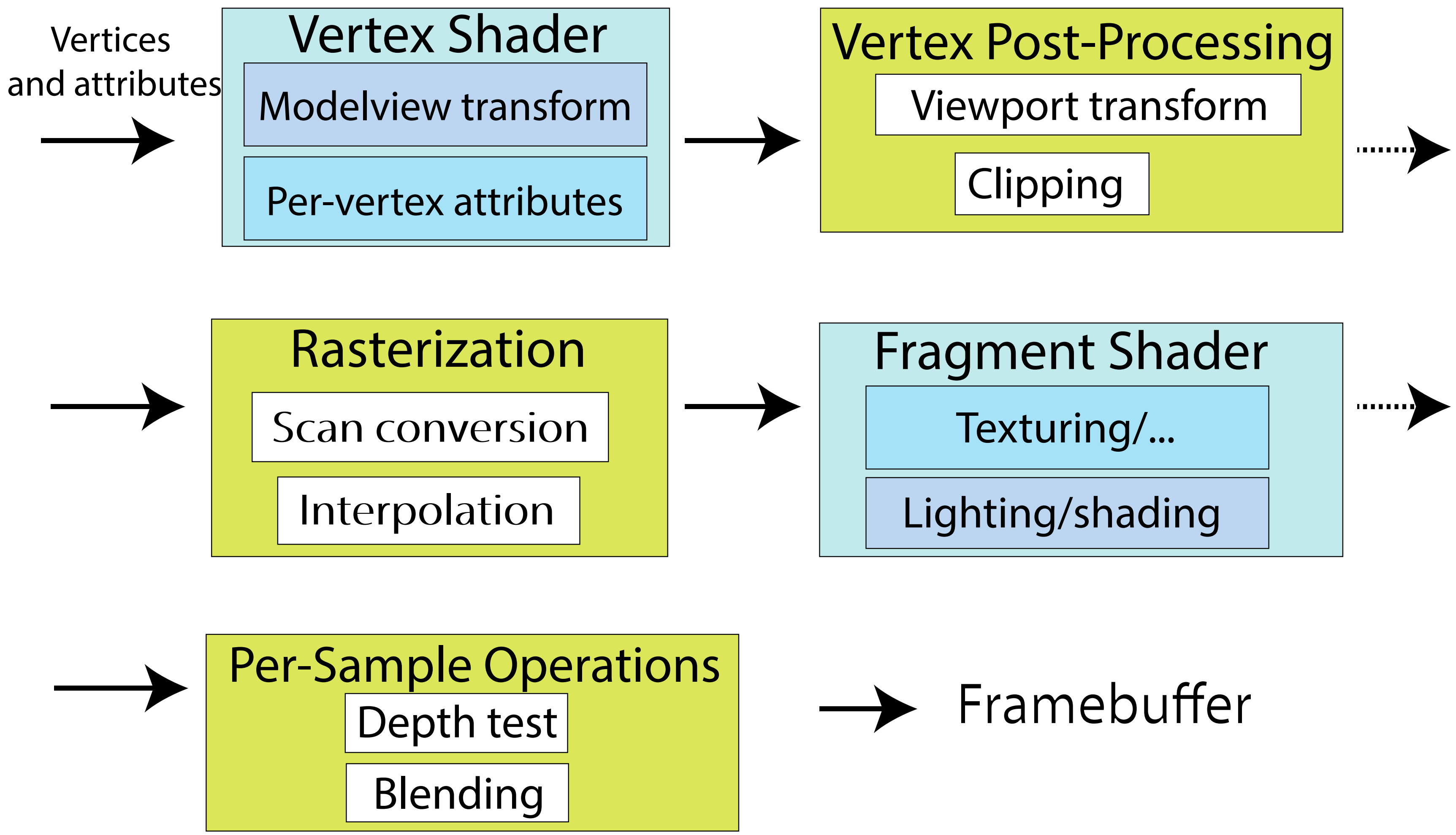
Recap: Coordinate transformations

World+Object Coordinates | Camera Coordinates | Window Coordinates | Pixel-wise attributes*



*usually multiple fragments for every pixel (fragment != pixel)

OpenGL Rendering Pipeline (detailed)



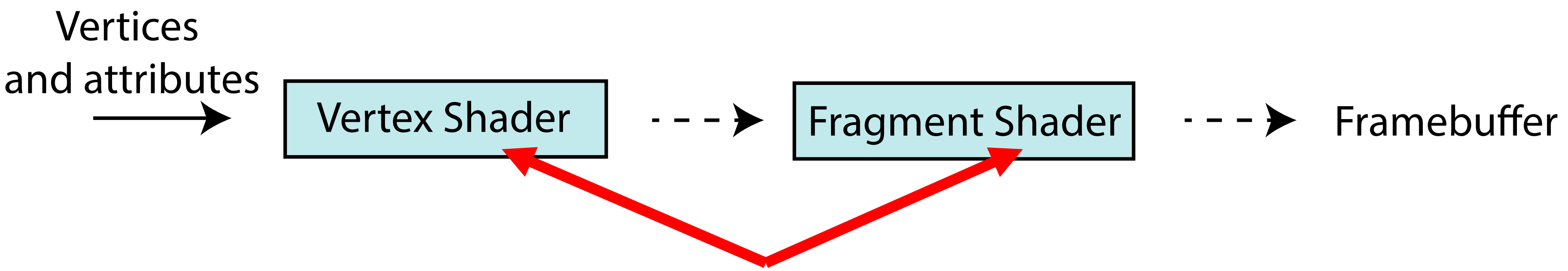
OpenGL Rendering Pipeline (simplified)

1. ***Vertex shader: geometric transformations***
2. ***Fragment shader: pixel-wise color computation***

World + Object Coordinates

Pixel-wise attributes

RGBA image



Shader: Programmable functions
to define object appearance locally
(vertex wise or fragment wise)

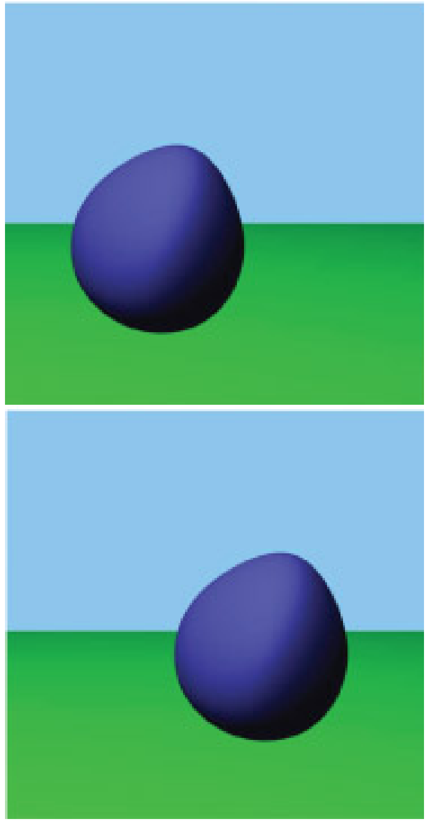
Recap: Vertex shader examples

Object motion & transformation

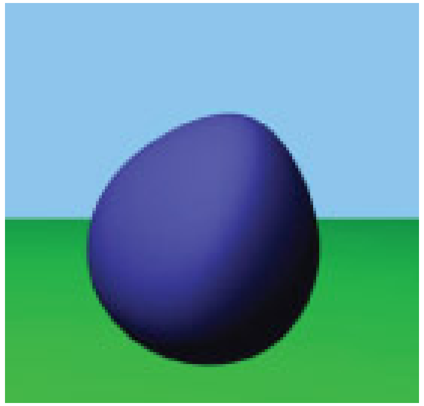
- translation
- rotation
- scaling

Projection

- Orthographic
  - *simple, without perspective effects*
- Perspective
  - *pinhole projection model*



Translation



Scaling

Matrices

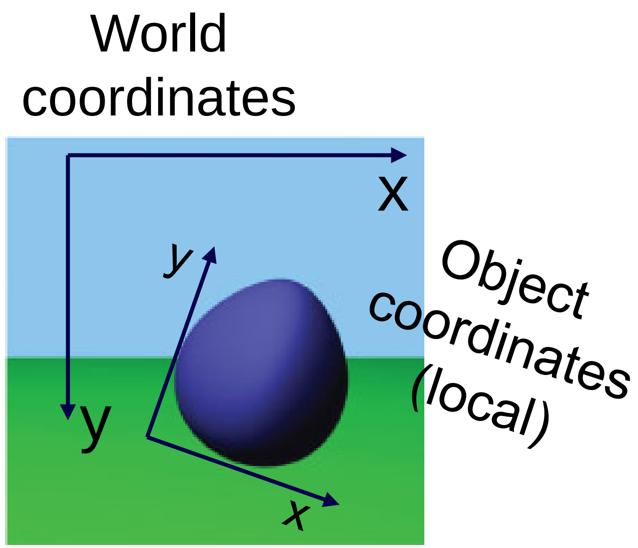
Object coordinates -> World coordinates

- **Model Matrix**
- One per object

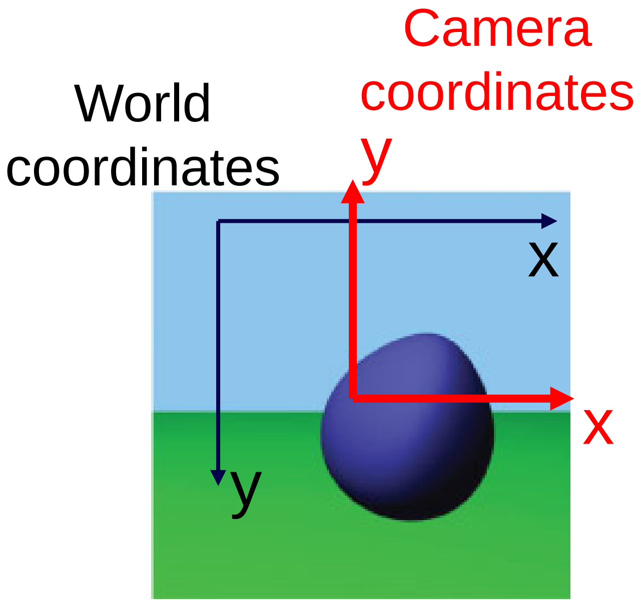
World coordinates -> Camera coordinates

- **View Matrix**
- One per camera

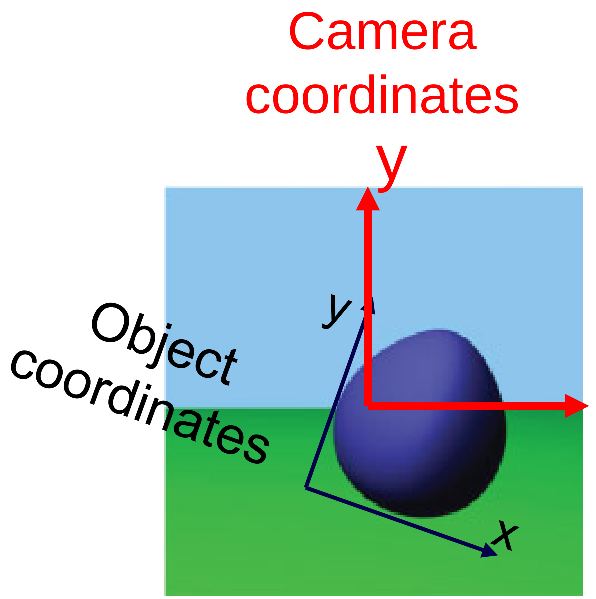
From local object to camera coordinates



object -> world



world -> camera



object -> camera

`transform`

`projection`

`projection * transform`

Recap: GLSL Vertex shader

The OpenGL Shading Language (GLSL)

- Syntax similar to the C programming language
- Build-in vector operations
  - functionality as the GLM library

```
void main()
{
    // Transforming The Vertex
    vec3 out_pos = projection * transform * vec3(in_pos.xy, 1.0);
    gl_Position = vec4(out_pos.xy, in_pos.z, 1.0);
}
```

x and y coordinates of a vec2, vec3 or vec4

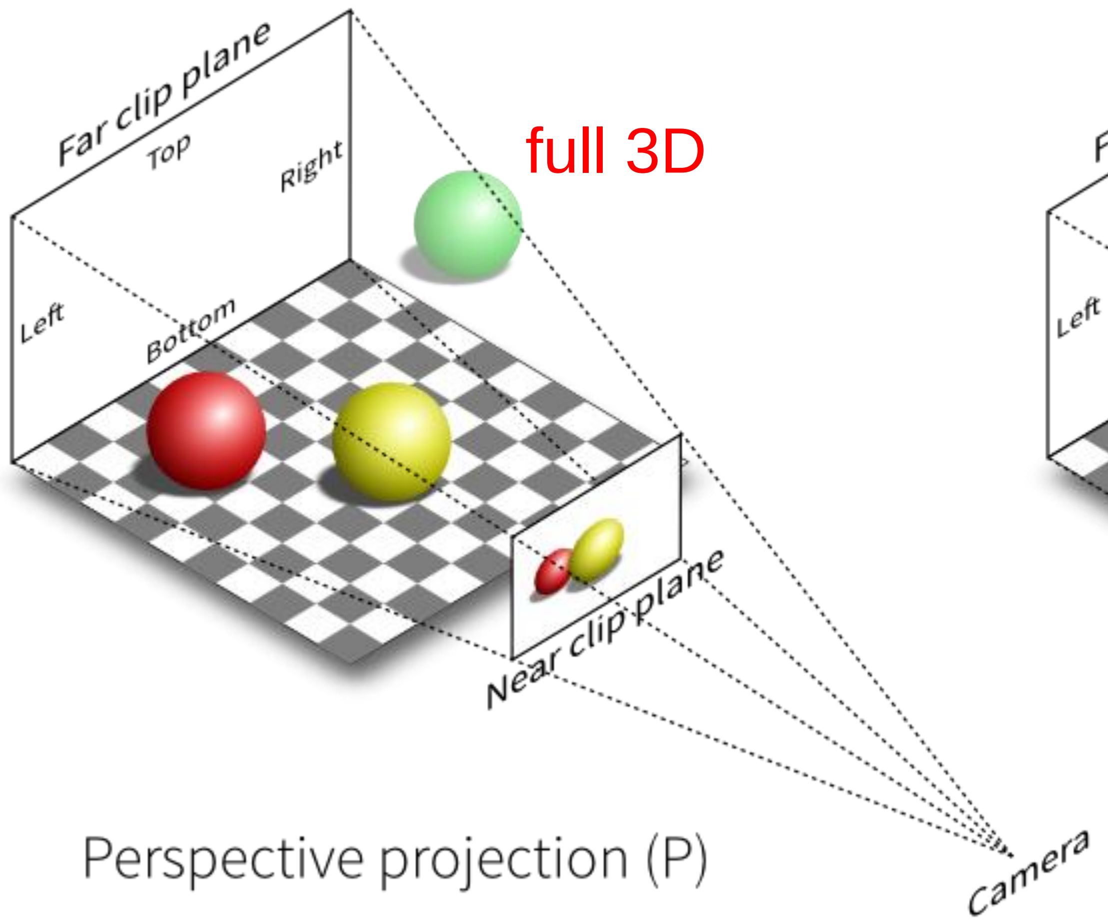
world -> camera

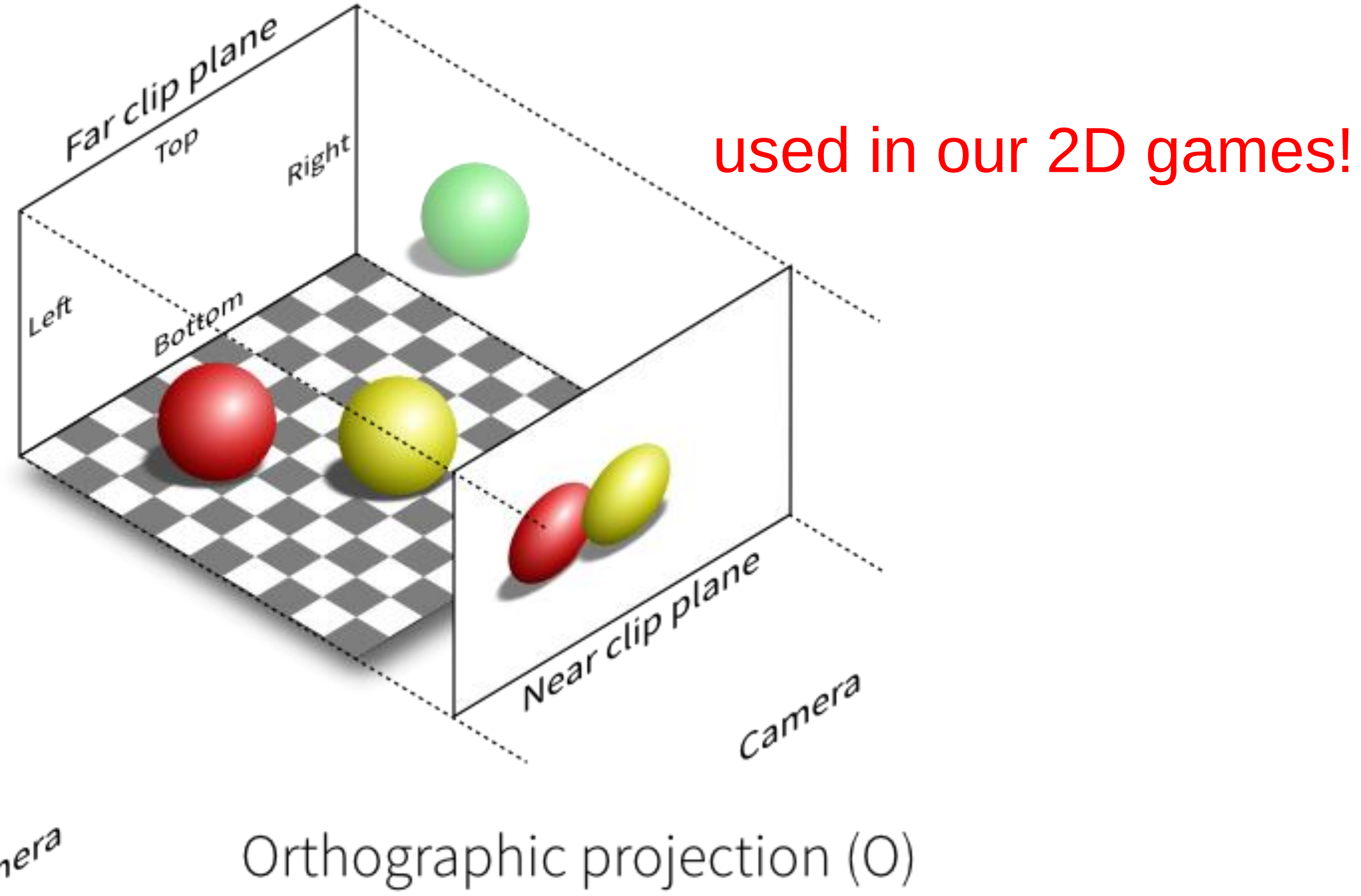
object -> world

vector of 3 (vec3) and 4 (vec4) floats

float (32 bit)

Camera types

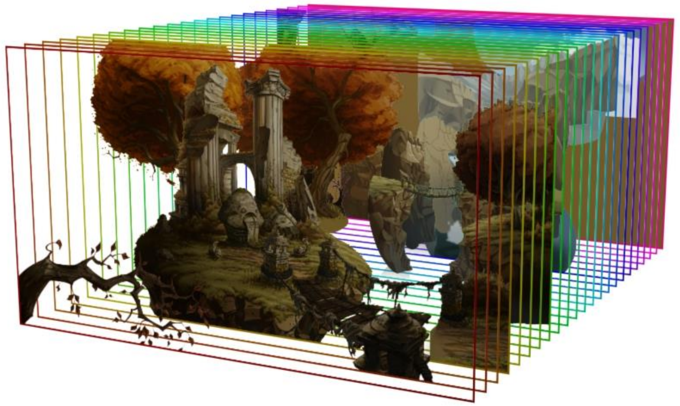




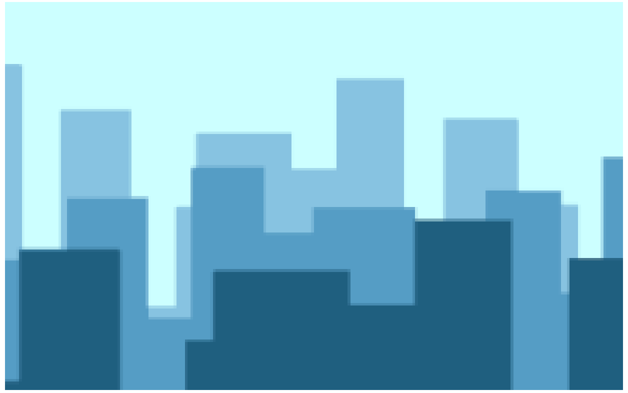
https://www.youtube.com/watch?v=zit45k6CUMk

Parallax Scrolling Background

Side view:



Depth effect:



Frontal view:

Formula?

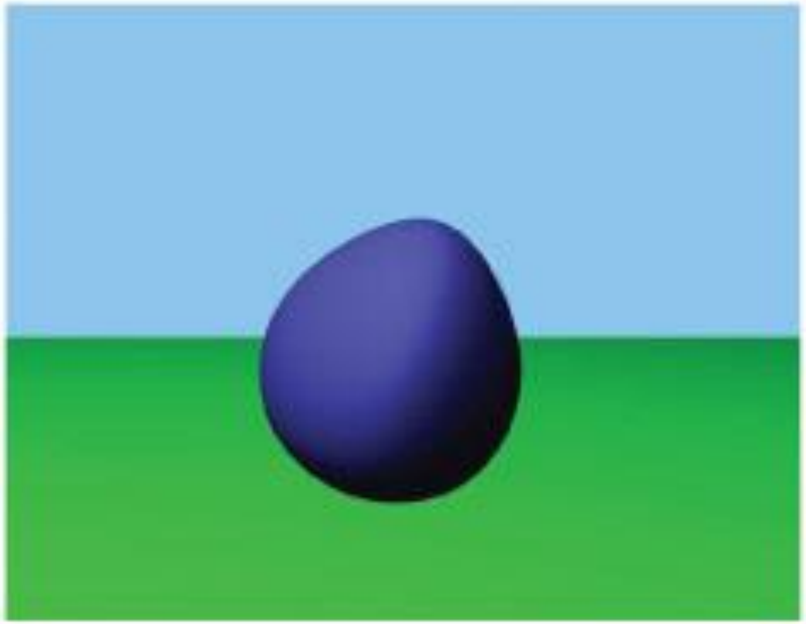
$$x = t \ * ??$$

What is x and t?

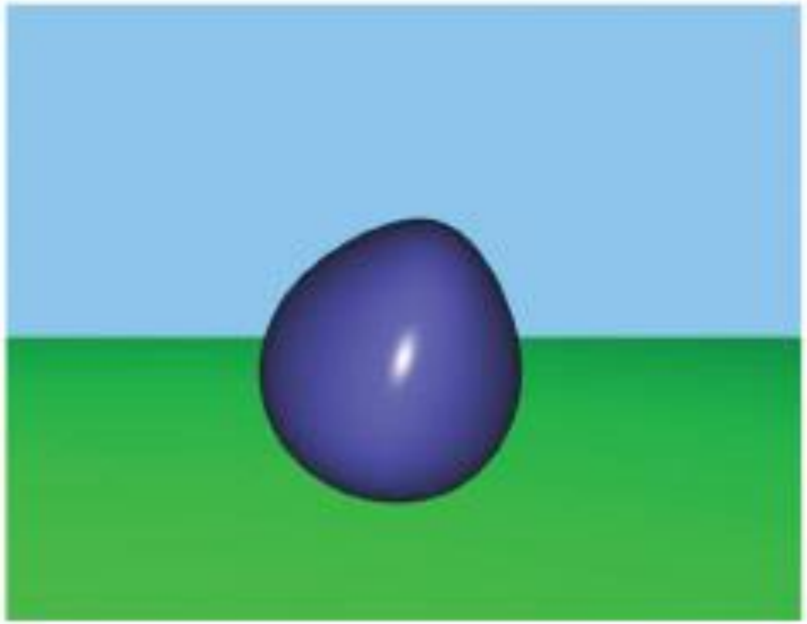
https://en.wikipedia.org/wiki/Parallax_scrolling

Fragment shader examples

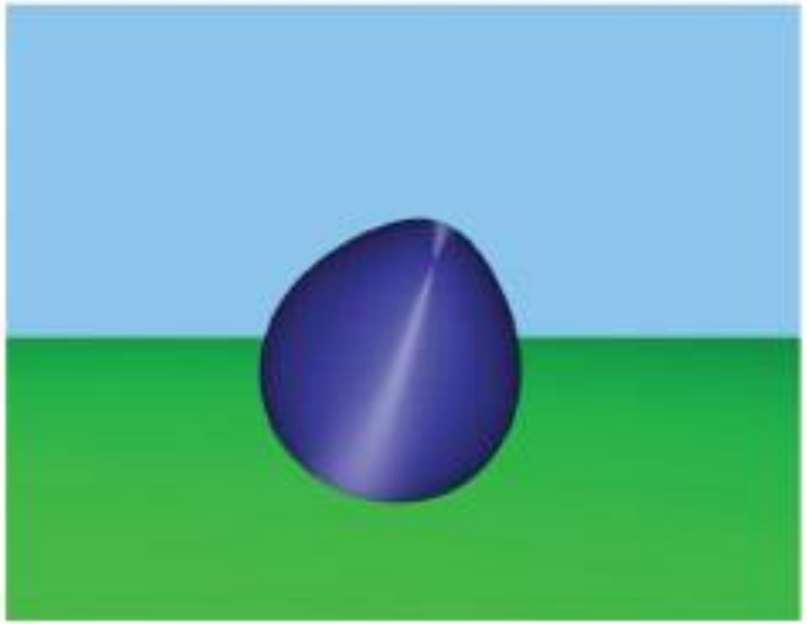
- *simulates materials and lights*
- *can read from textures*



Diffuse



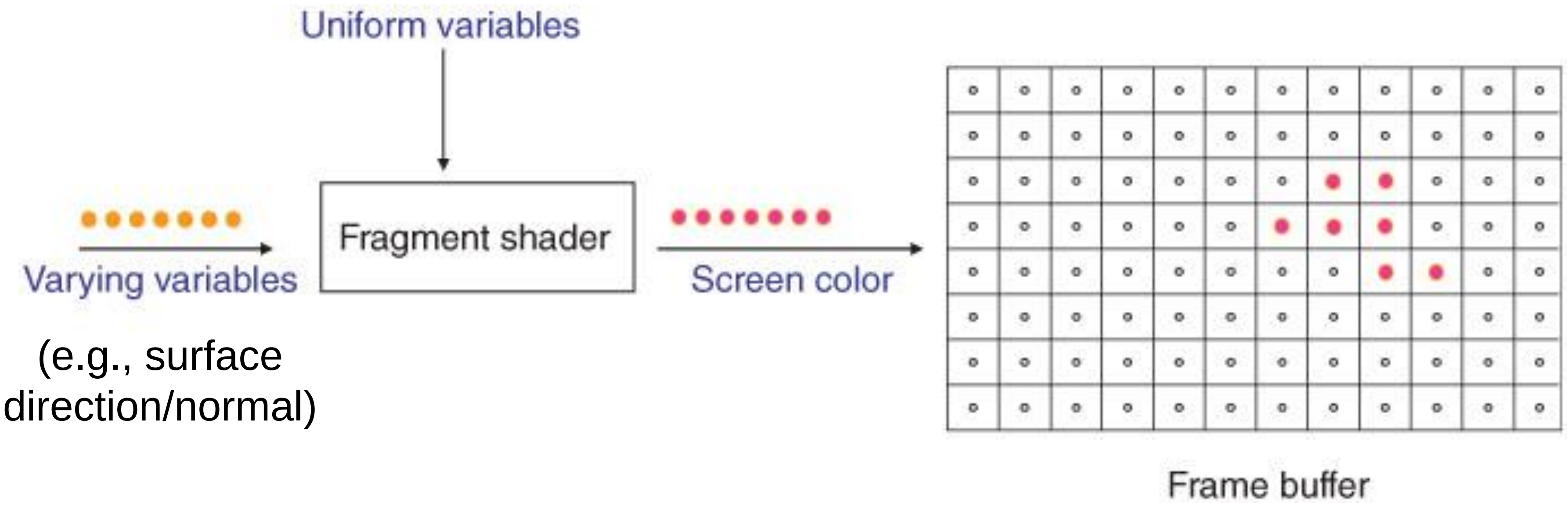
Specular



Directional

Fragment shader overview

(e.g., light direction)



(e.g., surface direction/normal)

GLSL fragment shader examples

Minimal:

```
out vec4 out_color;
void main()
{
    // Setting Each Pixel To ???
    out_color = vec4(1.0, 0.0, 0.0, 1.0);
}
```

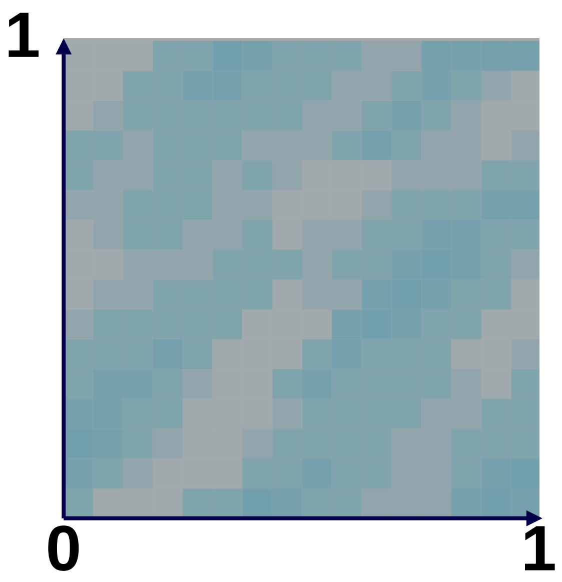
Specify color output

Red, Green, Blue, Alpha

Texture and Texture sampler

- ***A grid of pixels/colors***

- ***Parametrized from 0…1***



- ***The sampler returns the color/value at 2D position (u,v)***
  - How to determine the value between two pixels?

Shader demo

- go to https://www.shadertoy.com/view/ttKcWR
- lets play together
- Mental image of a fragment shader?
  - A function?
    - Input?
    - Output?

The OpenGL library

- Low-level graphics API
- C Interface accessed from C++

- ***How to***
- create textures
- set shaders
- set shader inputs
- start rendering

How to create a texture?

Look at our template:

```
glGenTextures((GLsizei)texture_gl_handles.size(), texture_gl_handles.data());

for(uint i = 0; i < texture_paths.size(); i++)
{
    glBindTexture(GL_TEXTURE_2D, texture_gl_handles[i]);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, dimensions.x, dimensions.y, 0, GL_RGBA, GL_UNSIGNED_BYTE, data);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
}
gl_has_errors();
```

Loading and compiling shaders

load from data/shaders/salmon.vs.glsl

CREATING SHADER OBJECTS

```
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, sourceCode, sourceCodeLength);
GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, sourceCode, sourceCodeLength);
```

load from data/shaders/salmon.fs.glsl

COMPILING

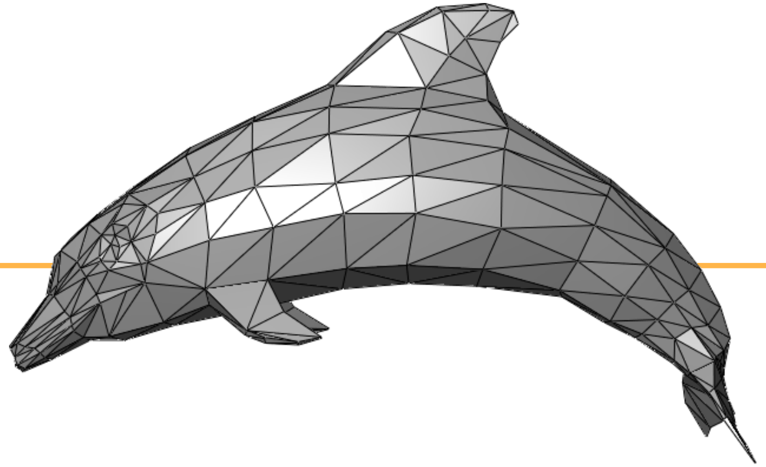
```
glCompileShader(vertexShader);
glCompileShader(fragmentShader);
```

Linking vertex and fragment shaders together

LINKING

```
program = glCreateProgram();
glAttachShader(program, vertexShader);
glAttachShader(program, fragmentShader);
glLinkProgram(program);
```

Recap: GEOMETRY



Triangle meshes

- Set of vertices
- Connectivity defined by indices
  - `uint16_t indices[] =` $\underbrace{\{\textit{vertex\_index1, vertex\_index2, vertex\_index3,}}_{\text{three indices make one triangle}}\ \ldots\}$

OpenGL resources

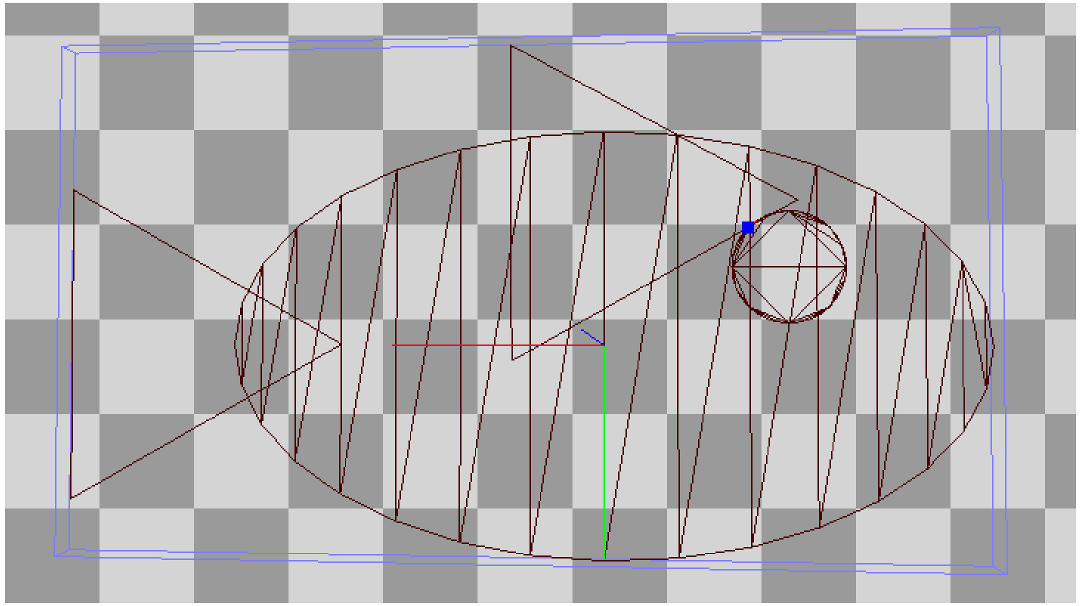
- vertex buffer
- index buffer

Creation

```
Gluint vbo;
glGenBuffers(vbo);

Gluint ibo;
glGenBuffers(ibo);
```

Recap: Programmatic geometry definition

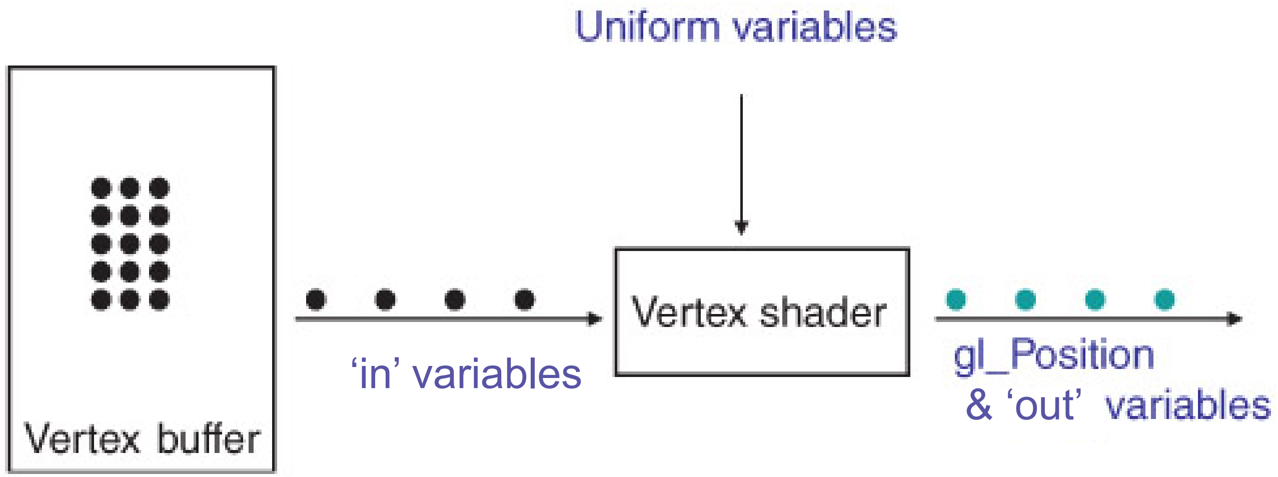
```
vec3 vertices[153];
vertices[0].position = { -0.54, +1.34, -0.01 };
vertices[1].position = { +0.75, +1.21, -0.01 };
…
vertices[152].position = { -1.22, +3.59, -0.01 };

uint16_t indices[] = { 0,3,1, 0,4,1,... , 151,152,150 };
Gluint vbo;
glGenBuffers(vbo);
glBindBuffer(vbo);
glBufferData(vbo, vertices);
Gluint ibo;
glGenBuffers(ibo);
glBindBuffer(ibo);
glBufferData(ibo, indices);
```



Vertex Shader

- VS is run for each vertex SEPARATELY
  - *doesn't know connectivity (by default)*
- Input:
  - *vertex coordinates in Object Coordinate System*
  - *vertex attributes: color, normal, …*
  - *uniform/global variables*
- It's primary role is to transform

Object coordinates
-> WORLD coordinates
-> VIEW coordinates



Shader Variable Types

More detail: GLSL Vertex shader

The OpenGL Shading Language (GLSL)

- Syntax similar to the C programming language
- Build-in vector operations
  - functionality as the GLM library

```
uniform mat3 transform;
uniform mat3 projection;
in vec3 in_pos;
void main() {
    // Transforming The Vertex
    vec3 out_pos = projection * transform * vec3(in_pos.xy, 1.0);
    gl_Position = vec4(out_pos.xy, in_pos.z, 1.0);
}
```

world -> camera

object -> world

vertex-specific input position

mandatory to set

Setting (Vertex) Shader Variables in C++

Uniform variable (same for all vertices/fragments)

```
mat3 projection_2D{ { sx, 0.f, 0.f },{ 0.f, sy, 0.f },{ tx, ty, 1.f } }; // affine transformation as introduced in the prev. lecture
GLint projection_uloc = glGetUniformLocation(texmesh.effect.program, "projection");
glUniformMatrix3fv(projection_uloc, 1, GL_FALSE, (float*)&projection);
```

In variable (attribute for every vertex)

```
// assuming vbo contains vertex position information already
GLint vpositionLoc = glGetAttribLocation(program, "in_pos");
glEnableVertexAttribArray(vpositionLoc);
glVertexAttribPointer(vpositionLoc, 3, GL_FLOAT, GL_FALSE, sizeof(vec3), (void*)0);
```

Variable Types

Uniform

- same for all vertices/fragments

Out (vertex shader) connects to In (fragment shader)

- computed per vertex, automatically interpolated for fragments
  - *E.g., position, normal, color, …*

In (attribute, vertex shader)

- values per vertex
- available only in Vertex Shader

Out (fragment shader)

- RGBA value per fragment

Salmon Vertex shader

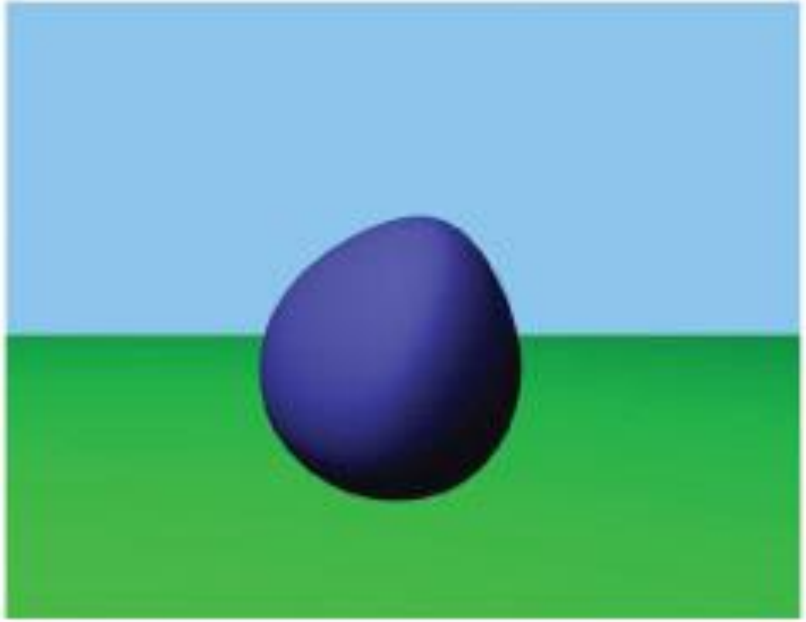
```
#version 330
// Input attributes
in vec3 in_position;
in vec3 in_color;

out vec3 vcolor;
out vec2 vpos;

// Application data
uniform mat3 transform;
uniform mat3 projection;

void main() {
        vpos = in_position.xy; // local coordinated before transform
        vcolor = in_color;
        vec3 pos = projection * transform * vec3(in_position.xy, 1.0);
        gl_Position = vec4(pos.xy, in_position.z, 1.0);
}
```

pass on color and position in object coordinates

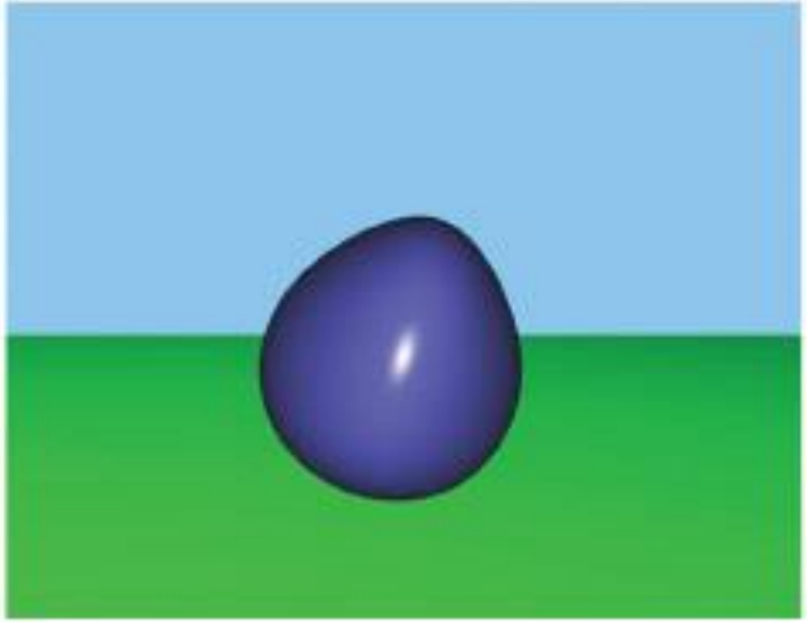
as before

Recap: Fragment shader examples

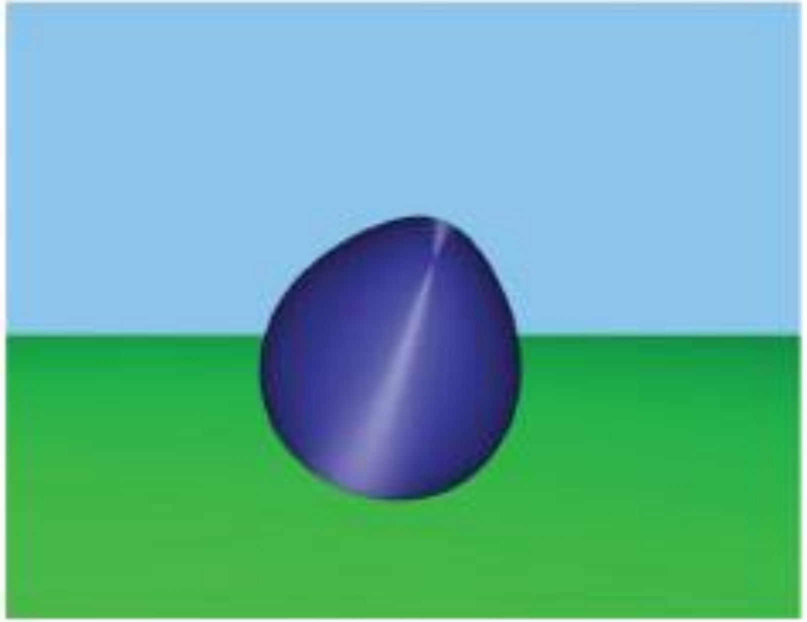
- ***simulates materials and lights***
- ***can read from textures***



Diffuse



Specular



Directional

Salmon Fragment shader

```
#version 330
// From Vertex Shader
in vec3 vcolor;
in vec2 vpos; // Distance from local origin

// Application data
uniform vec3 fcolor;
uniform int light_up;

// Output color
layout(location = 0) out vec4 color;

void main() {
	color = vec4(fcolor * vcolor, 1.0);

	// Salmon mesh is contained in a 1x1 square
	float radius = distance(vec2(0.0), vpos);
	if (light_up == 1 && radius < 0.3) {
		// 0.8 is just to make it not too strong
		color.xyz += (0.3 - radius) * 0.8 * vec3(1.0, 1.0, 1.0);
	}
}
```

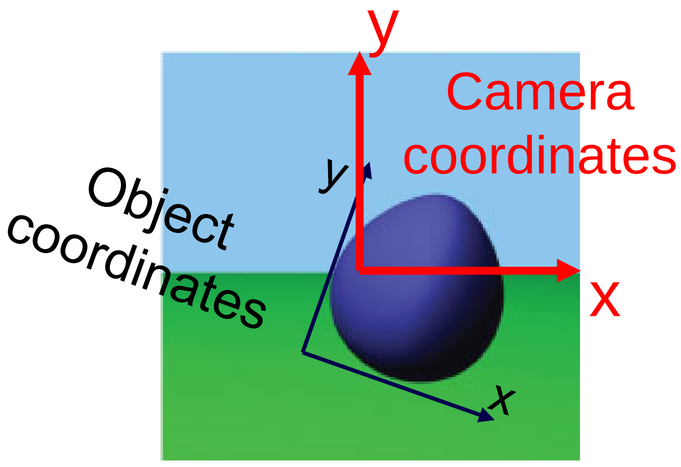
interpolated vertex color, times global color

create a spherical highlight around the object center

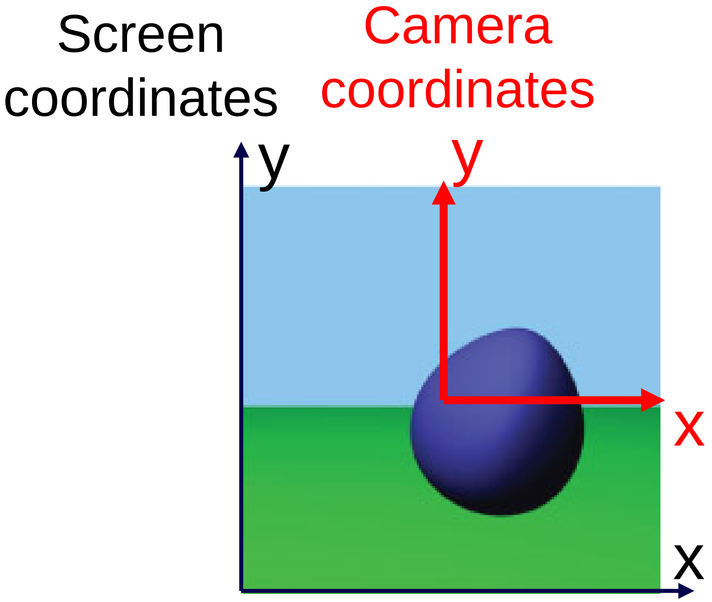
(Hidden) Vertex Post-Processing

Vertex Shader → Vertex Post-Processing →

- Viewport transform: camera coordinates to screen/window coordinates
  - set with `glViewport(0, 0, w, h);`



object -> camera



camera -> screen

- Clipping: Removing invisible geometry (outside view frame)

Rendering

Draw

```
glDrawElements(GL_TRIANGLES, 6, ..); // 6 is the number of indices
```