Visual Computing Exercise 9: WebGL Shading Language and Blending

Lingchen Yang

Yingyan Xu

lingchen.yang@inf.ethz.ch

yingyan.xu@inf.ethz.ch

WebGL Shading Language

- Syntax similar to C/C++
  - Shader entry point: `void main()`
  - Vertex shader: executed once per vertex
  - Fragment shader: executed once per fragment (pixel)
- Shaders compiled at runtime
  - Passed to WebGL as string
  - Graphics hardware vendor provides compiler

Data types

Scalars	
float, int, bool	basic types
Vectors	
vec2, **vec3**, **vec4**	2D, 3D, 4D float vectors
ivec2, ivec3, ivec4	2D, 3D, 4D int vectors
bvec2, bvec3, bvec4	2D, 3D, 4D bool vectors
Matrices	
mat2, mat3, **mat4**	2D, 3D, 4D float matrices
Samplers	
sampler1D, **sampler2D**, sampler3D	1D, 2D, 3D textures

Vectors

- vec2, vec3, vec4 types
- Components can be selected by a "." (swizzling)
  - 3 sets of component names
    - x, y, z, w (treat vector as a position or direction)
    - r, g, b, a (treat vector as a color)
    - s, t, p, q (treat vector as texture coordinate)

```
vec4 a = vec4(1.0, 2.0, 3.0, 4.0);
float f = a.x;
vec2 v2 = a.xy;
vec3 v3 = a.rgb;
vec4 v4 = a.xgba;
// is illegal - component names
not from the same set

vec3 b = vec3(1.0, 2.0, 3.0);
vec3 flip = b.zyx
//flip = (3.0, 2.0, 1.0)
vec4 dup = b.rrgg
//dup = (1.0, 1.0, 2.0, 2.0)
```

Functions

- GLSL supports functions
- Similar to C functions, except:
  - function names can be overloaded
  - parameter qualifiers:
    - **in** (default): values copied into function at call time
    - **out**: values copied out of function
    - **inout**: values copied in and out of function
  - no recursion

```
void multAndAdd(in vec2 a,
                in vec2 b,
                out vec2 add,
                out vec2 mult);
```

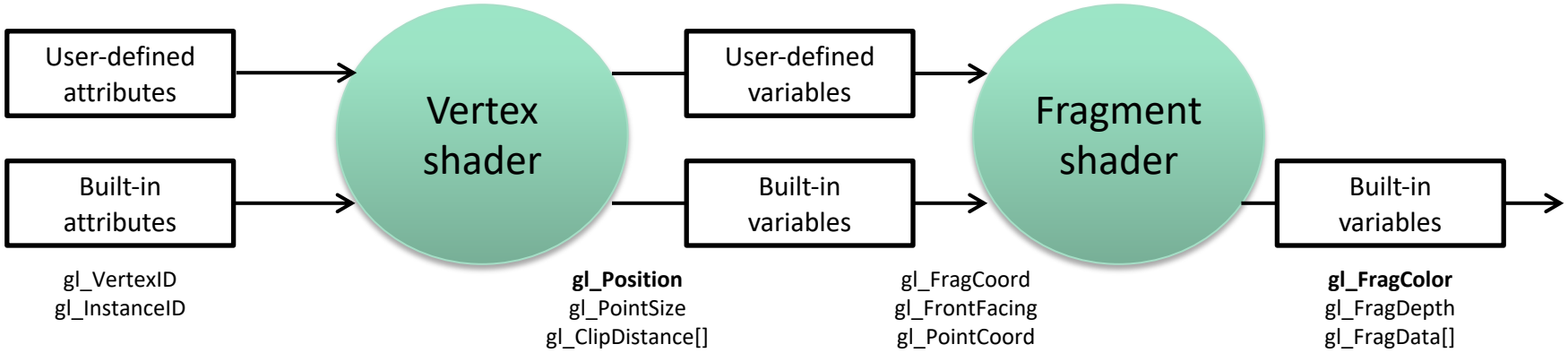
Built-in functions

- Angle, Trigonometry
  - radians, degrees, sin, cos, tan, asin, acos, atan
- Exponential functions
  - pow, exp, log, exp2, log2, sqrt
- Common functions
  - abs, sign, floor, ceil, fract, mod, min, max, clamp, mix

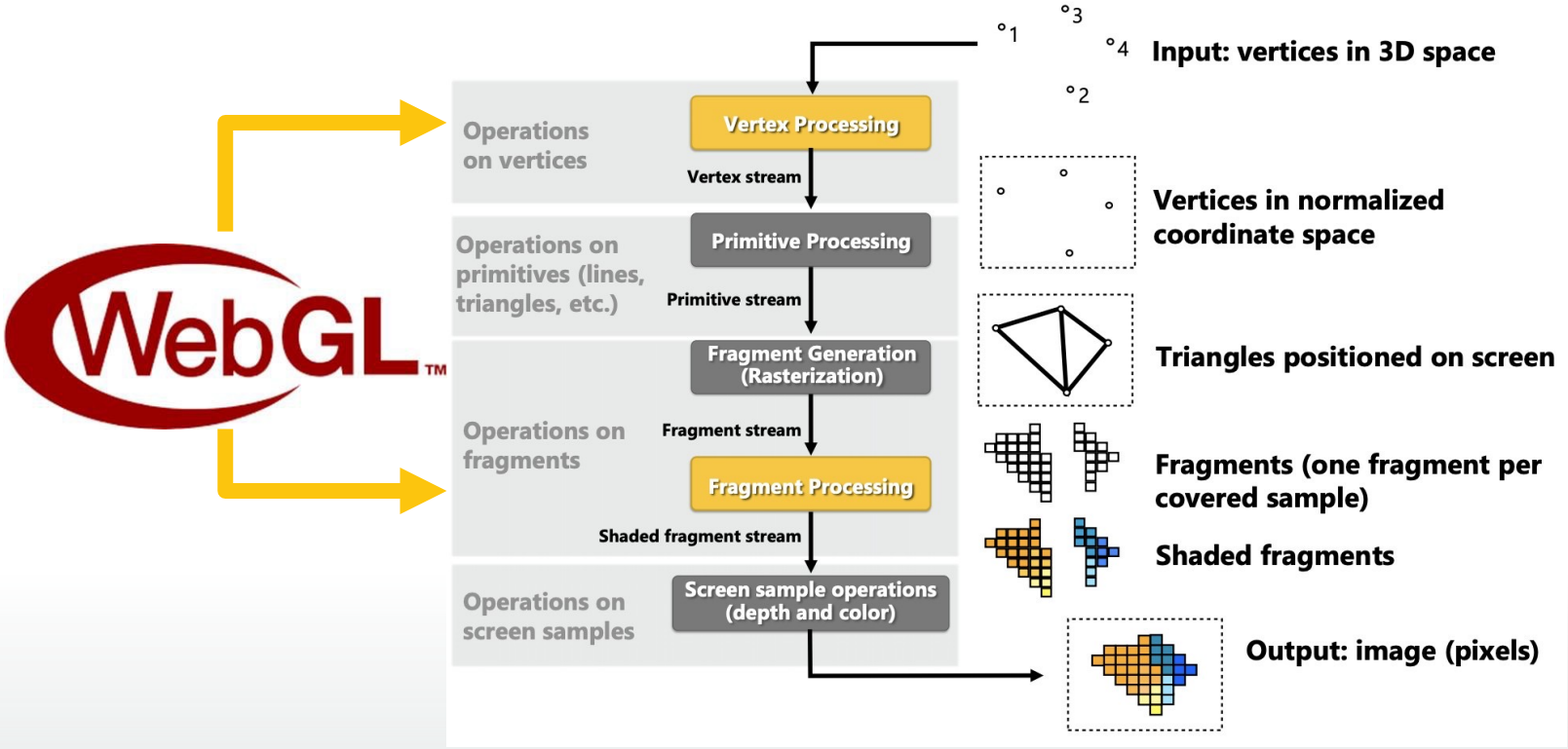
Built-in functions

- Geometric functions
  - length, distance, dot, cross, normalize
- Matrix functions
  - matrixCompMult, outerProduct, transpose
- many more...

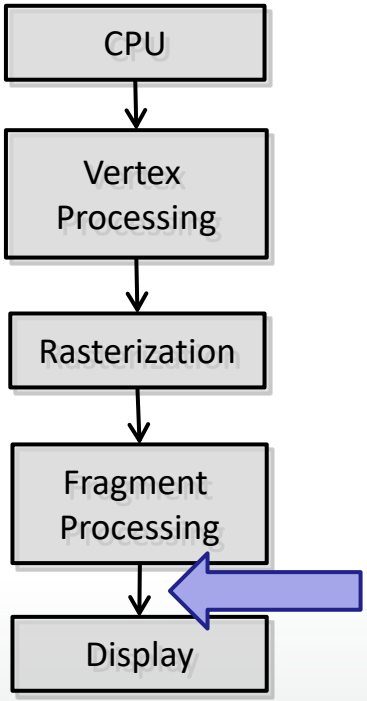
Shader inputs/outputs



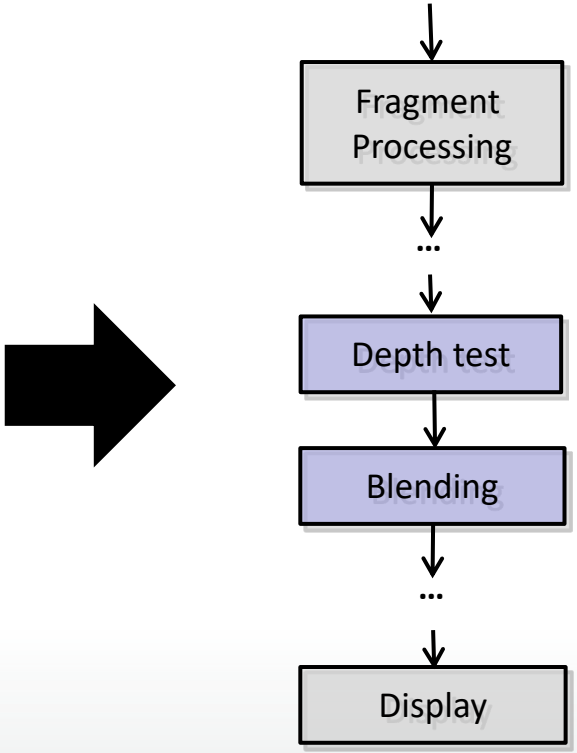
Graphics Pipeline



Graphics pipeline

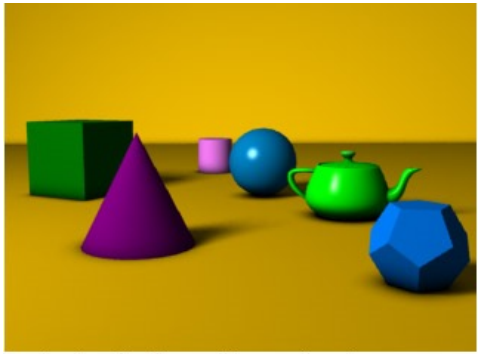


Steps after fragment shader



Depth buffer test (z-test)

- Depth stored for every pixel in framebuffer
  - Depth = distance from camera
- Depth buffer also called z-buffer
- Used to resolve occlusions

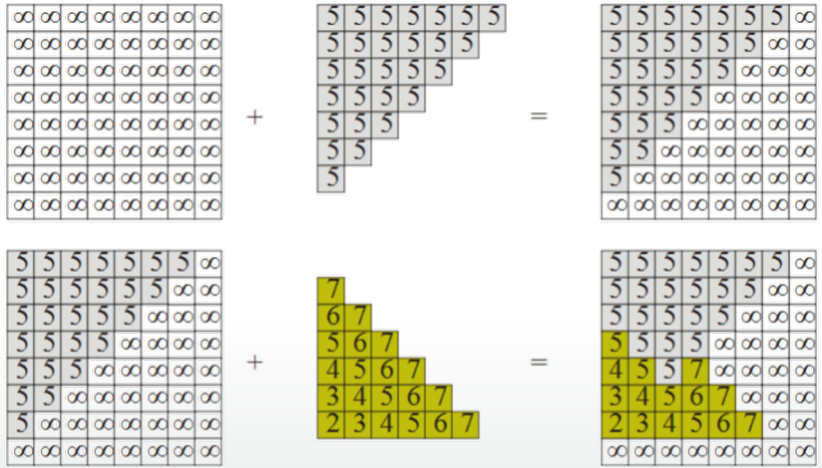


A simple three dimensional scene

Z-buffer representation

Depth buffer test (z-test)

- Compare depth of current fragment with depth of already drawn pixel
- Pass if smaller → only draw if fragment is closer to camera than already drawn pixel



Depth test in WebGL

```
glEnable(GL_DEPTH_TEST);
glDisable(GL_DEPTH_TEST);
glDepthFunc( enum func );
```

glDepthFunc() functions

```
GL_NEVER
GL_LESS
GL_EQUAL
GL_LEQUAL
GL_GREATER
GL_NOTEQUAL
GL_GEQUAL
GL_ALWAYS
```

Blending

- Combine pixel color (source) with color that is already in framebuffer (destination)
- Without blending: New fragment replaces old color

```
final.rgb = src.rgb * src.a + dest.rgb * (1.0 - src.a)
```

Blending in WebGL

```
glEnable(GL_BLEND);
glDisable(GL_BLEND);

glBlendEquation( enum mode );
glBlendFunc( enum src, enum dst);
```

```
final.rgb = src.rgb * src.a +
            dest.rgb * (1.0 - src.a)
```

glBlendEquation() modes

```
GL_FUNC_ADD
GL_FUNC_SUBTRACT
GL_FUNC_REVERSE_SUBTRACT
```

glBlendFunc() functions

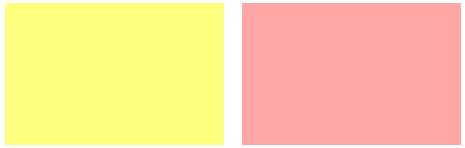
```
GL_ZERO
GL_ONE
GL_SRC_COLOR
GL_ONE_MINUS_SRC_COLOR
GL_DST_COLOR
GL_ONE_MINUS_DST_COLOR
GL_SRC_ALPHA
GL_ONE_MINUS_SRC_ALPHA
GL_DST_ALPHA
GL_ONE_MINUS_DST_ALPHA
GL_CONSTANT_COLOR
GL_ONE_MINUS_CONSTANT_COLOR
GL_CONSTANT_ALPHA
GL_ONE_MINUS_CONSTANT_ALPHA
GL_SRC_ALPHA_SATURATE
GL_SRC1_COLOR
GL_ONE_MINUS_SRC1_COLOR
GL_SRC1_ALPHA
GL_ONE_MINUS_SRC1_ALPHA
```

Depth Buffer Bug

- Depth test doesn't consider transparency
  - Front object is drawn first
  - Back object fails the z-test and is not drawn at all
  - No blending happens

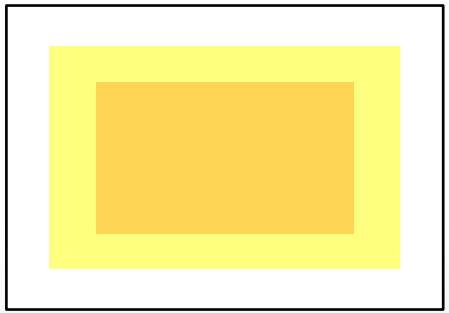
Simple illustration

- Drawing two semi-transparent squares
  - Assuming the red square is rendered first

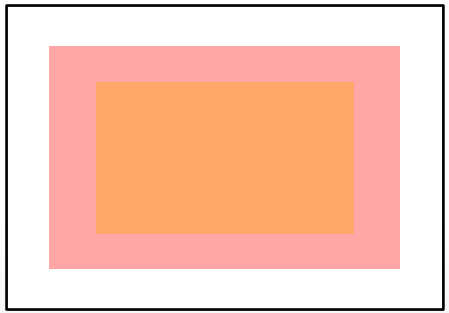


Correct results:

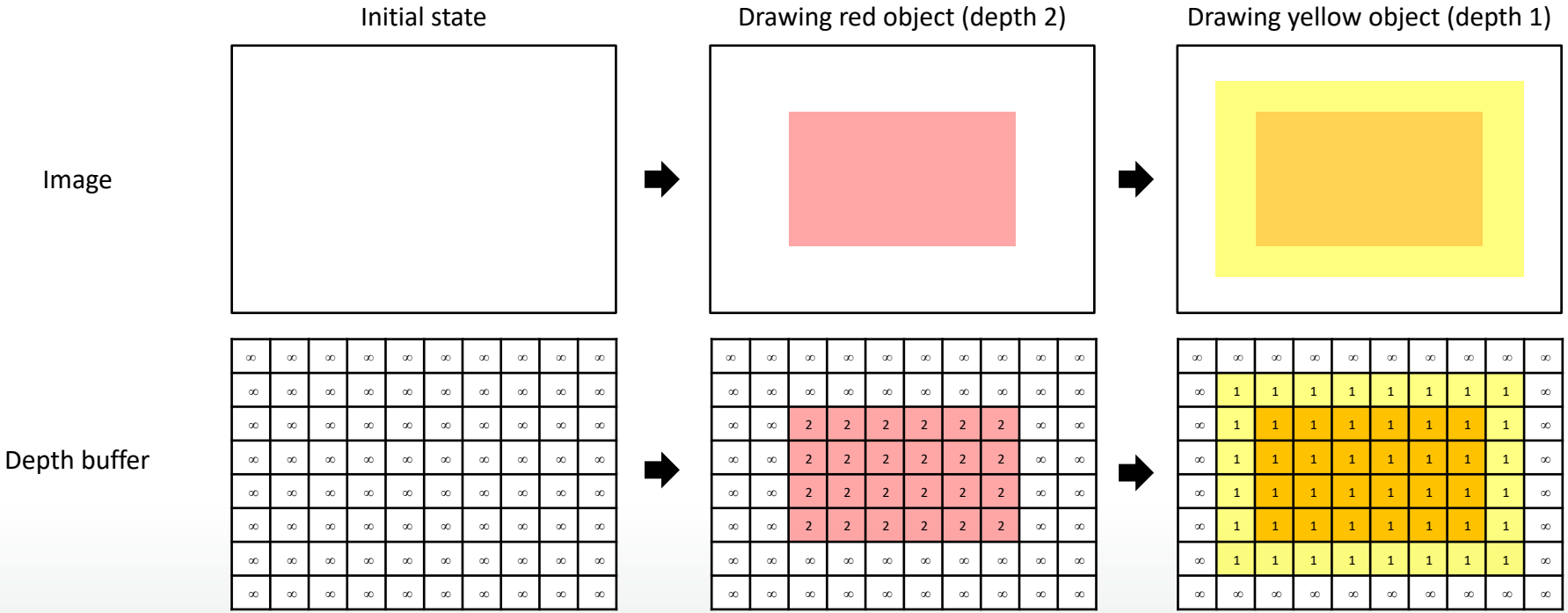
Case 1:
Yellow in front of red



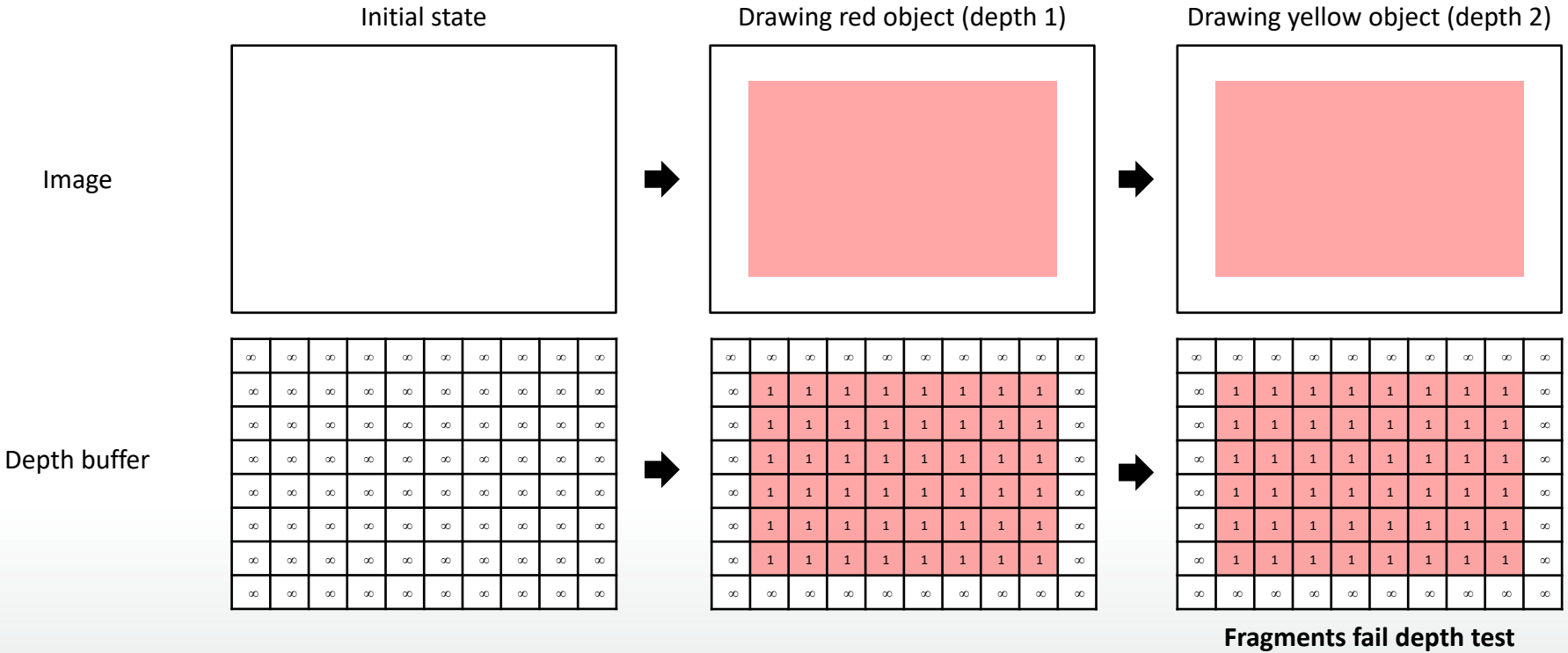
Case 2:
Red in front of yellow



Simple illustration: Case 1

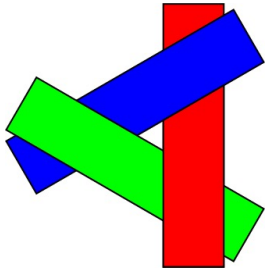


Simple illustration: Case 2



Solutions

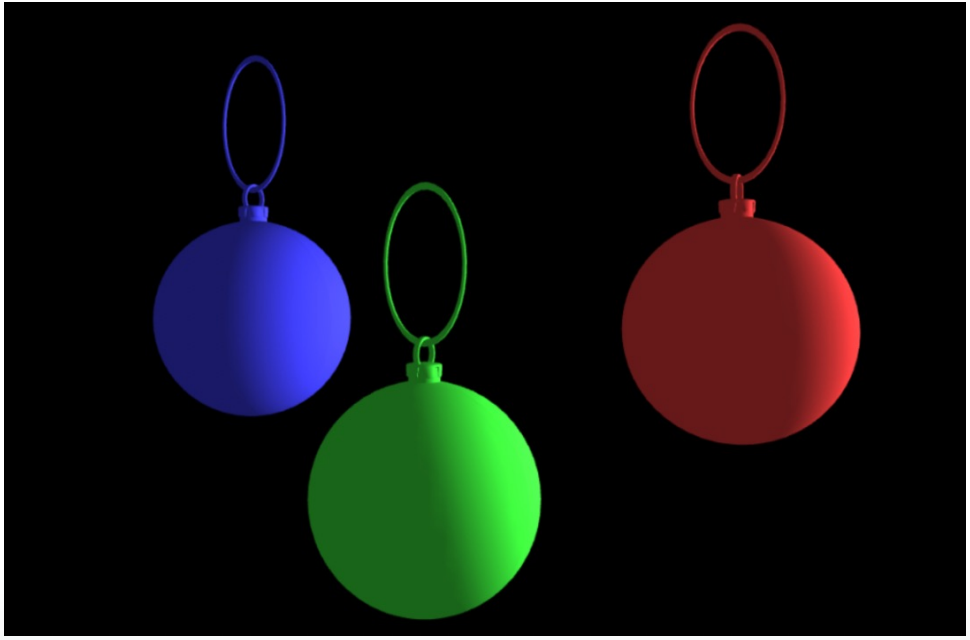
- z-sorting (depth sorting)
  - Draw transparent objects from back to front
  - Might still fail in certain cases



- Order-independent transparency techniques
  - Depth peeling
  - Store and sort fragments on the GPU

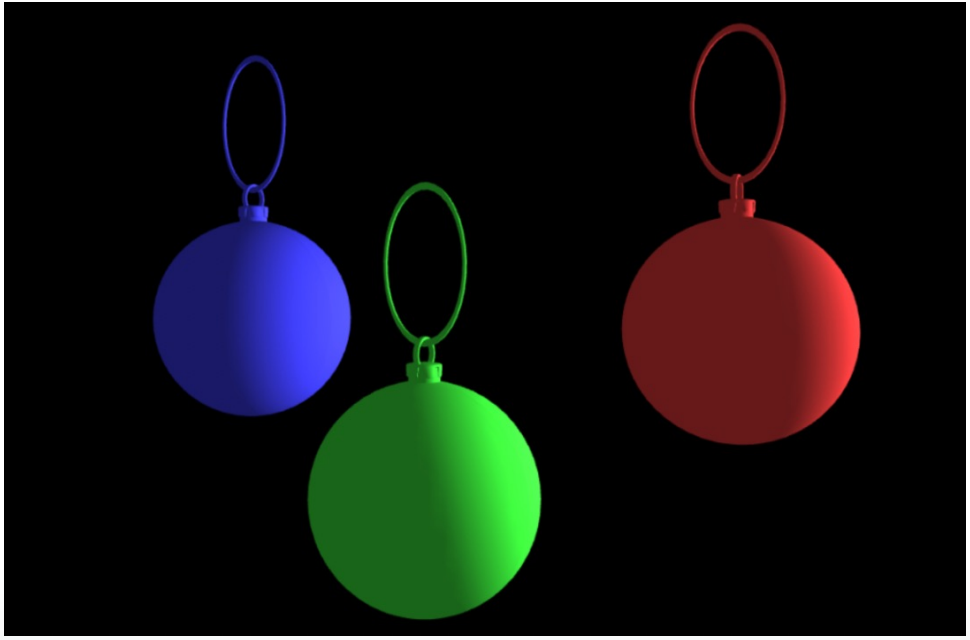
1) Separating Decorative Balls

- Separating the ball based on their relative positions.
- Rearrange the face indices if necessary.



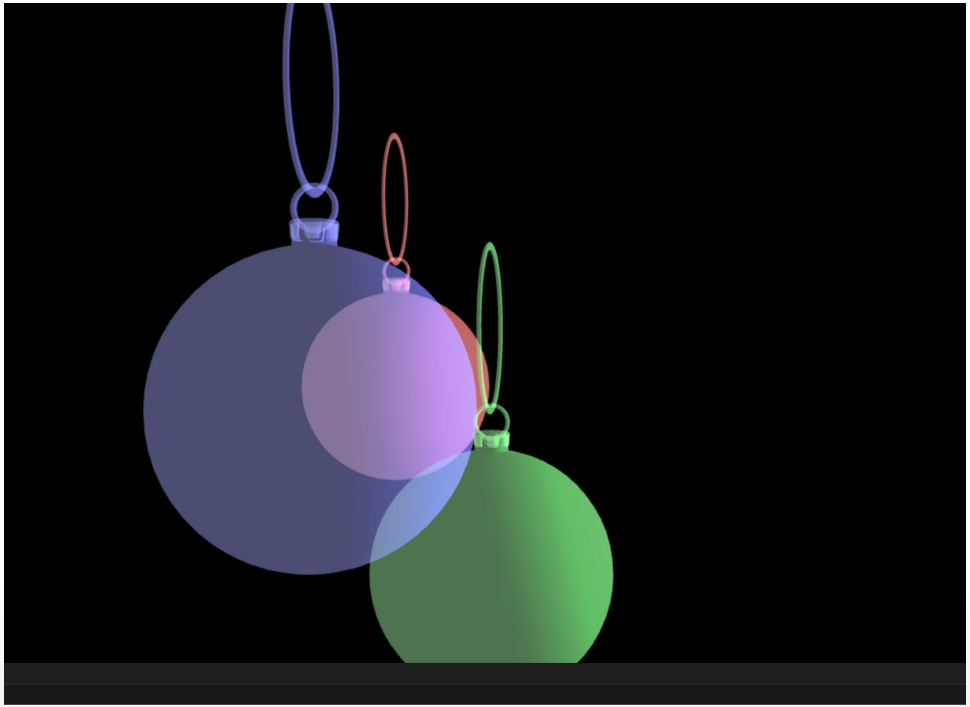
2) Depth Sorting

- For each time step, sort the balls according to their center depth values.



3) Blending

- Add objects to the scene according to the depth order.
- Transparency: use alpha value to blend objects and background



Questions

yingyan.xu@inf.ethz.ch