CPSC 427
Video Game Programming

Rendering and Transformations

Helge Rhodin

Today

- *UX and UI conclusion*
- *ECS conclusion*

- *Game loop*
- *2D Transformations*
- *Rendering*

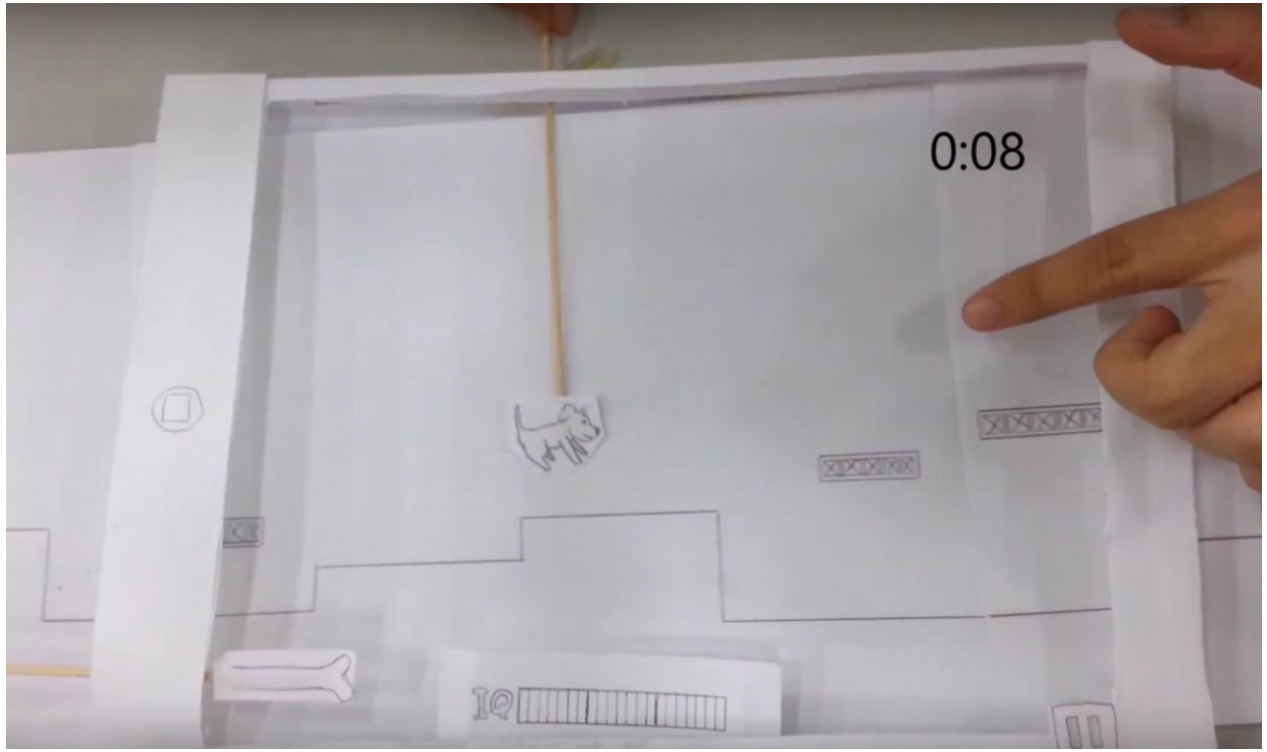
Recap: Design Concepts

- **Design concepts:** Basic ideas that help us understand & design what's happening in a user interface

- Norman's Design Concepts:
  - Affordances
  - Constraints
  - Mapping
  - Visibility
  - Feedback
  - Consistency

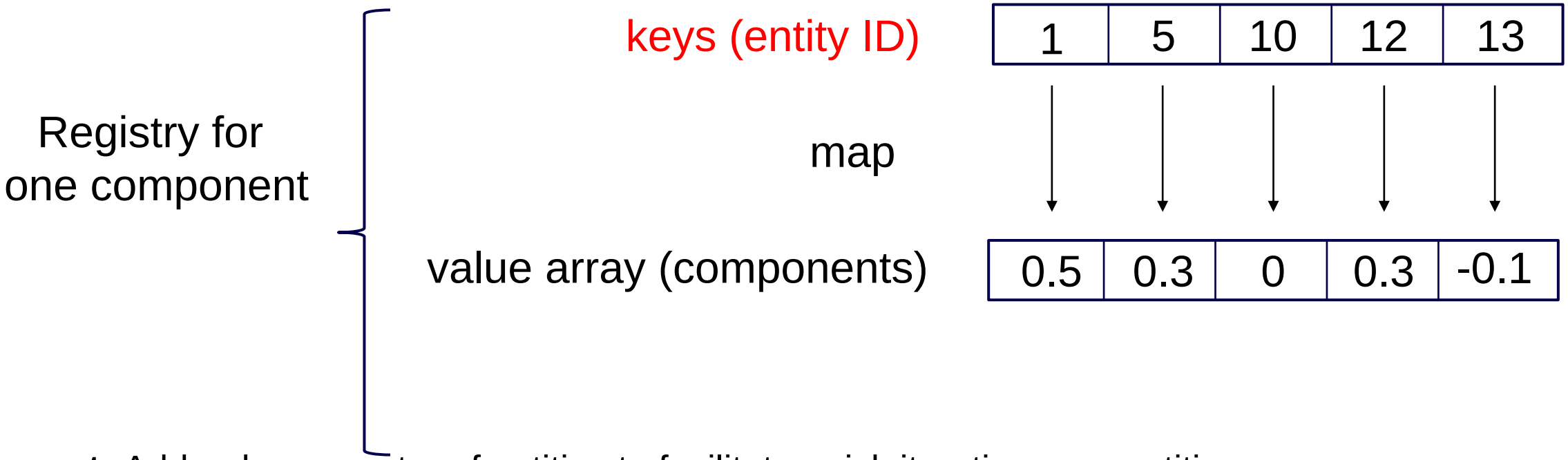
Recap: Low Fidelity Prototyping

- Used for early stages of design
  - *Quick & cheap to deploy*
  - *Easy to test*
- Iterate on story and core gameplay mechanics
- Sketches are a great way to start designing



ECS Conclusions

Map + Component Vector + Entity Vector



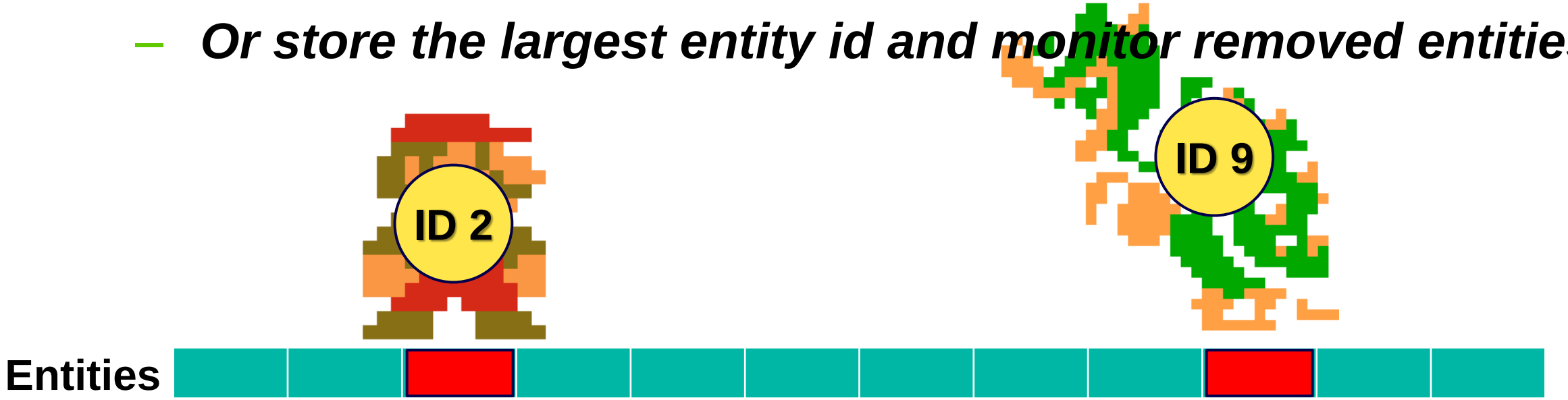
Concept: Add a dense vector of entities to facilitate quick iteration over entities
Implementation: std::vector<Entities>; std::vector<Component>; std::map<Entity,unsigned int>

Deletion of components

- **When we "delete" an entity we must delete corresponding components to.**
- **Different approaches to this,**
  - ***Fill deleted components in arrays with the last entities data***
    - Extra care must be taken when managing indices
  - ***Mark spots in arrays as rewritable***
    - Big systems will suffer from poor memory management

Lifetime of entities

- Each Entity is typically just a unique identifier to its components
  - *The tinyECS implementation can hold* 4,294,967,295 entities
- Store Entities in a big static array in the Entity Manager
  - *Or store the largest entity id and monitor removed entities*



How Does a System Find its Entities?

Extension/Optimization:

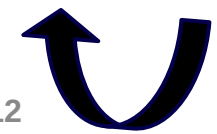
- **Each system has a list of entity IDs it is interested in**
- **Systems register their bitsets/bitmaps with the Entity Manager**
- **Whenever an Entity is added…**
  - ***Evaluate which systems are interested & update their ID lists***

The game loop

Can you imagine a game without?

A game is a simulator

1. ***AI and user input***
2. ***Environment reaction***
3. ***Equations of Motion***
   - sum forces & torques, solve for accelerations: $\vec{F} = ma$
4. ***Numerical integration***
   - update positions, velocities
5. ***Collision detection***
6. ***Collision resolution***



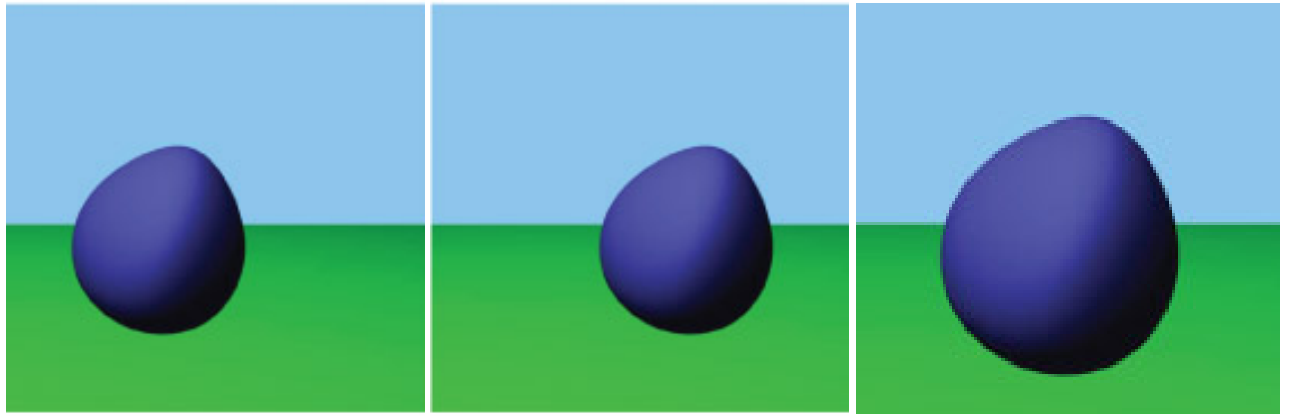
We will have a separate lecture on physics simulation!

Our game loop (A1, main.cpp)

```
// Set all states to default
world.restart();
auto t = Clock::now();
// Variable timestep loop
while (!world.is_over())
{
    // Processes system messages, if this wasn't present the window would become unresponsive
    glfwPollEvents();

    // Calculating elapsed times in milliseconds from the previous iteration
    auto now = Clock::now();
    float elapsed_ms = static_cast<float>((std::chrono::duration_cast<std::chrono::microseconds>(now - t)).count()) / 1000.f;
    t = now;

    DebugSystem::clearDebugComponents();
    ai.step(elapsed_ms, window_size_in_game_units);
    world.step(elapsed_ms, window_size_in_game_units);
    physics.step(elapsed_ms, window_size_in_game_units);
    world.handle_collisions();

    renderer.draw(window_size_in_game_units);
}

return EXIT_SUCCESS;
```

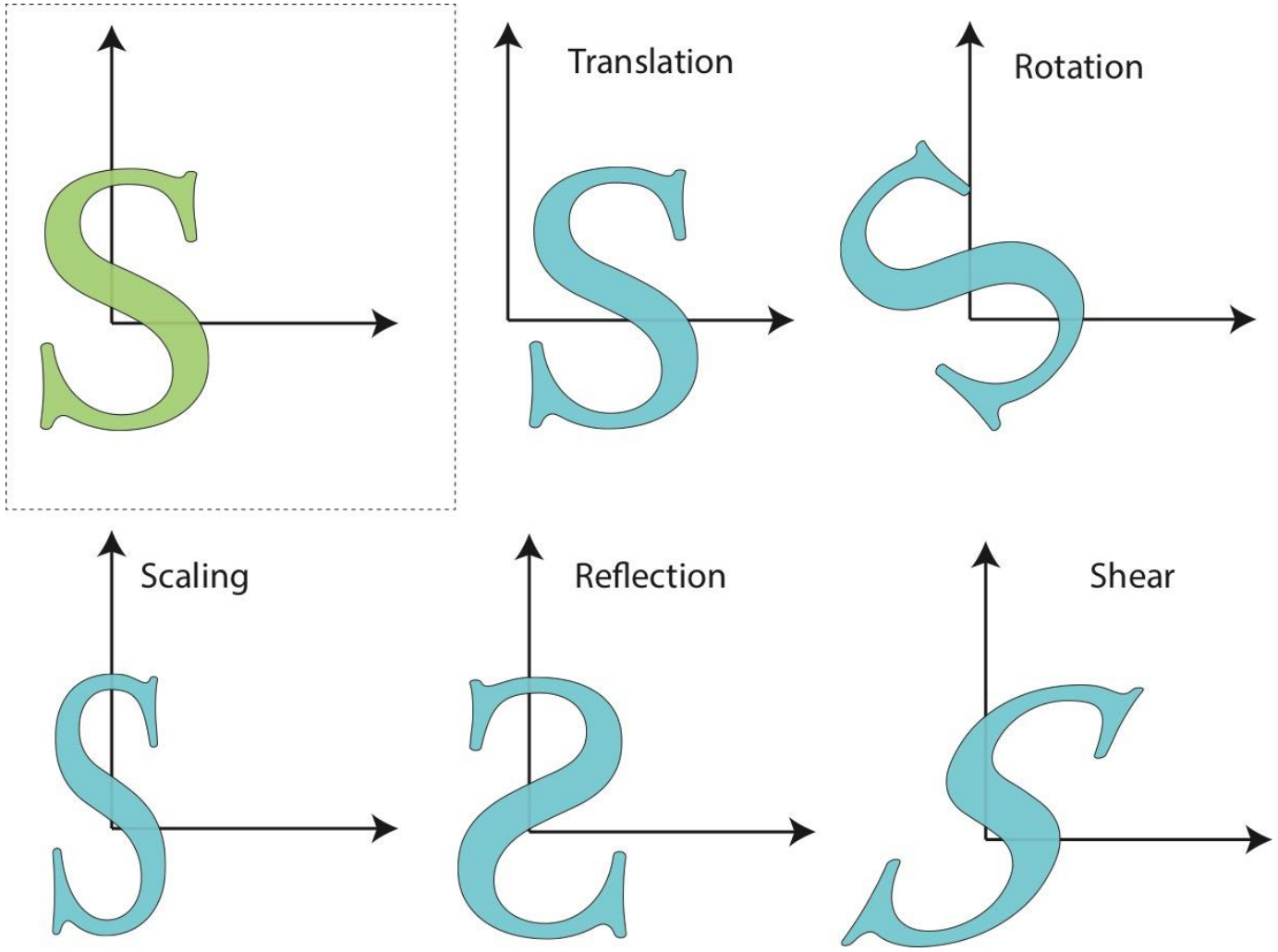
CPSC 427
Video Game Programming

Transformations

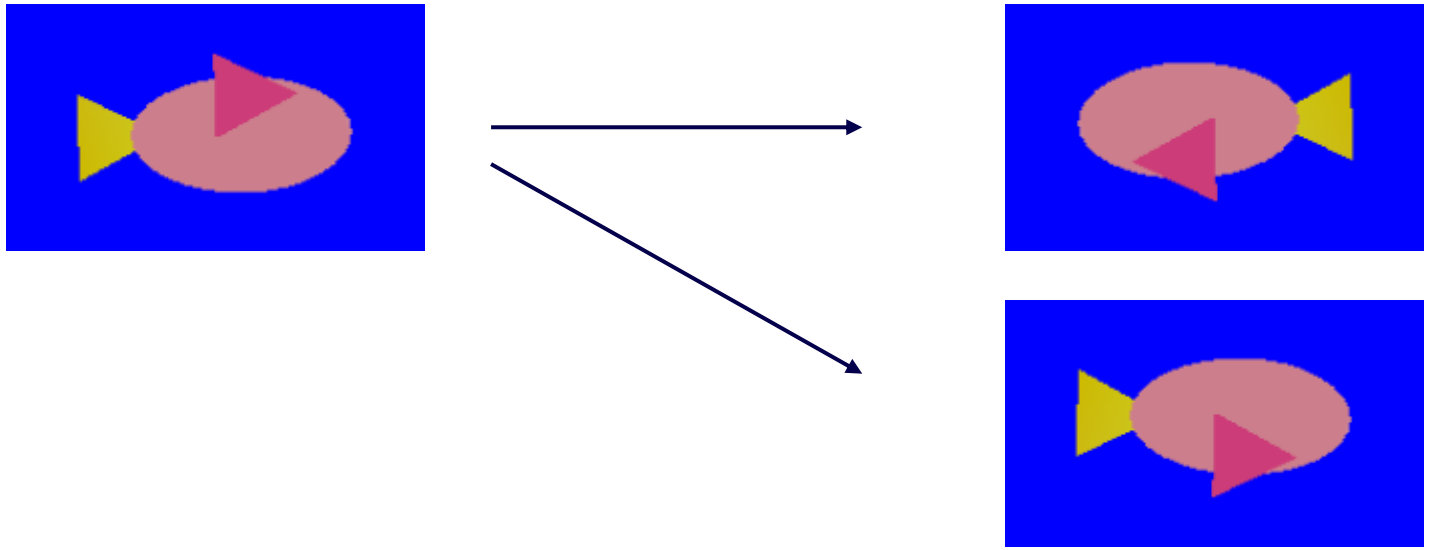


Helge Rhodin

Modeling Transformations



How to turn the fish?



Both versions are fine for Assignment 1 (A1)!

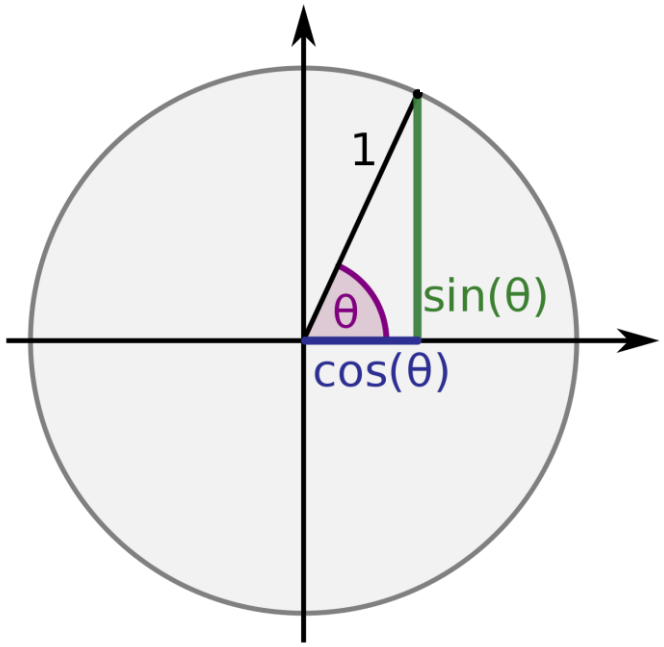
Linear transformations

- Rotations, scaling, shearing
- Can be expressed as 2x2 matrix (for 2D points)
- E.g.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

- or a rotation

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$



Rotation angle θ, cos, and sin

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

Affine transformations

- Linear transformations + translations
- Can be expressed as 2x2 matrix + 2 vector
- E.g. scale+ translation:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

Modeling Transformation

Adding a third coordinate

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix} \quad \rightarrow \quad \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \\ 0 \end{pmatrix}$$

$$= \begin{pmatrix} 2 & 0 & t_x \\ 0 & 2 & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Affine transformations are now linear

- one 3x3 matrix can express: 2D rotation, scale, shear, and translation

Combination of Transformations?

- ***How can we combine***
  - translation
  - rotation
  - scaling

… into one matrix?

Self study: Homogeneous coordinates

- Homogeneous coordinates are defined as vectors, with equivalence

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x/z \\ y/z \\ 1 \end{pmatrix} = \begin{pmatrix} x\lambda \\ y\lambda \\ z\lambda \end{pmatrix}$$

- Can also represent projective equations
- homogeneous matrix becomes 4x4

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 & t_x \\ 0 & 2 & 0 & t_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

CPSC 427
Video Game Programming

Rendering basics

Helge Rhodin

What is rendering?

Generating an image from a (3D) scene

Let's think how!

Scene

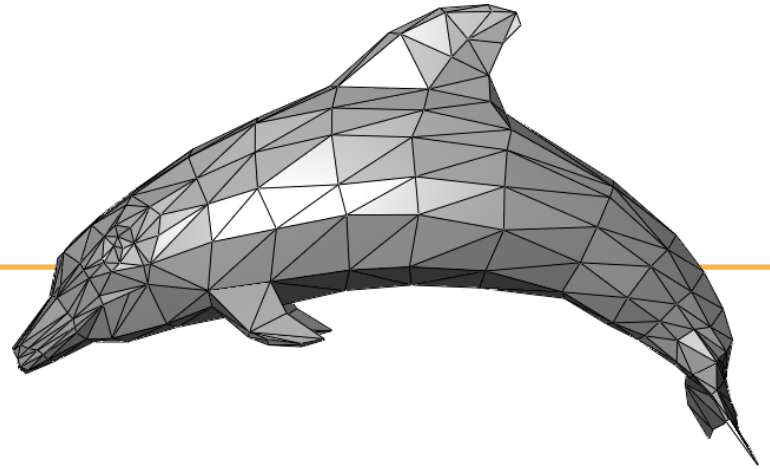
- A coordinate frame
- Objects
- Their materials
- (Lights)
- (Camera)

Object

Most common:

- ***surface representation***

GEOMETRY



Triangle meshes

- Set of vertices
- Connectivity defined by indices
  - `uint16_t indices[] = {vertex_index1, vertex_index2, vertex_index3, …}`

three indices make one triangle

OpenGL resources

- vertex buffer

Creation

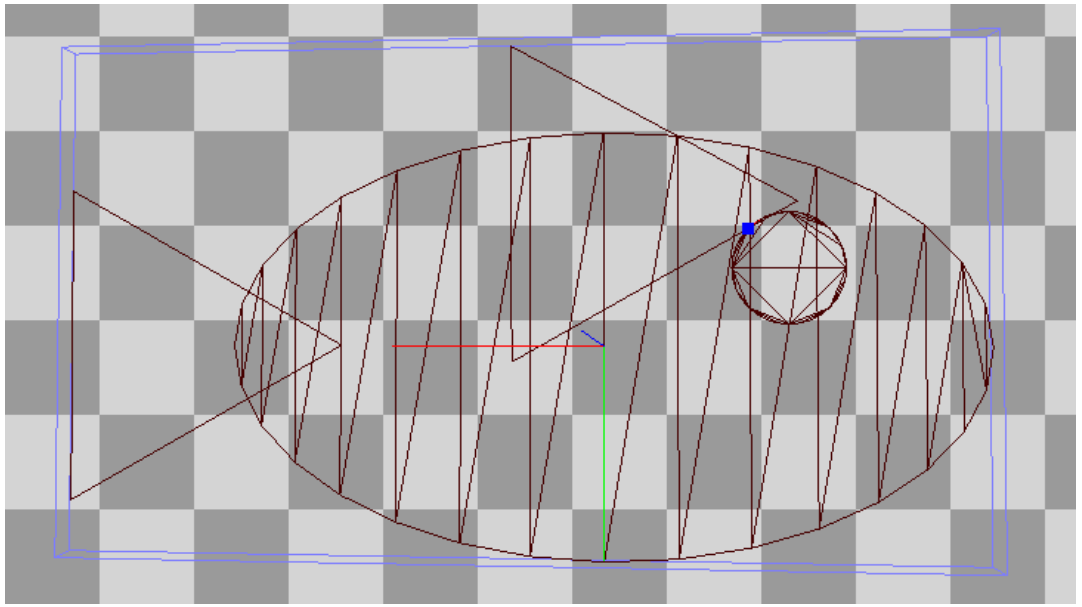
```
Gluint vbo;
glGenBuffers(vbo);
```

- index buffer

```
Gluint ibo;
glGenBuffers(ibo);
```

Programmatic geometry definition

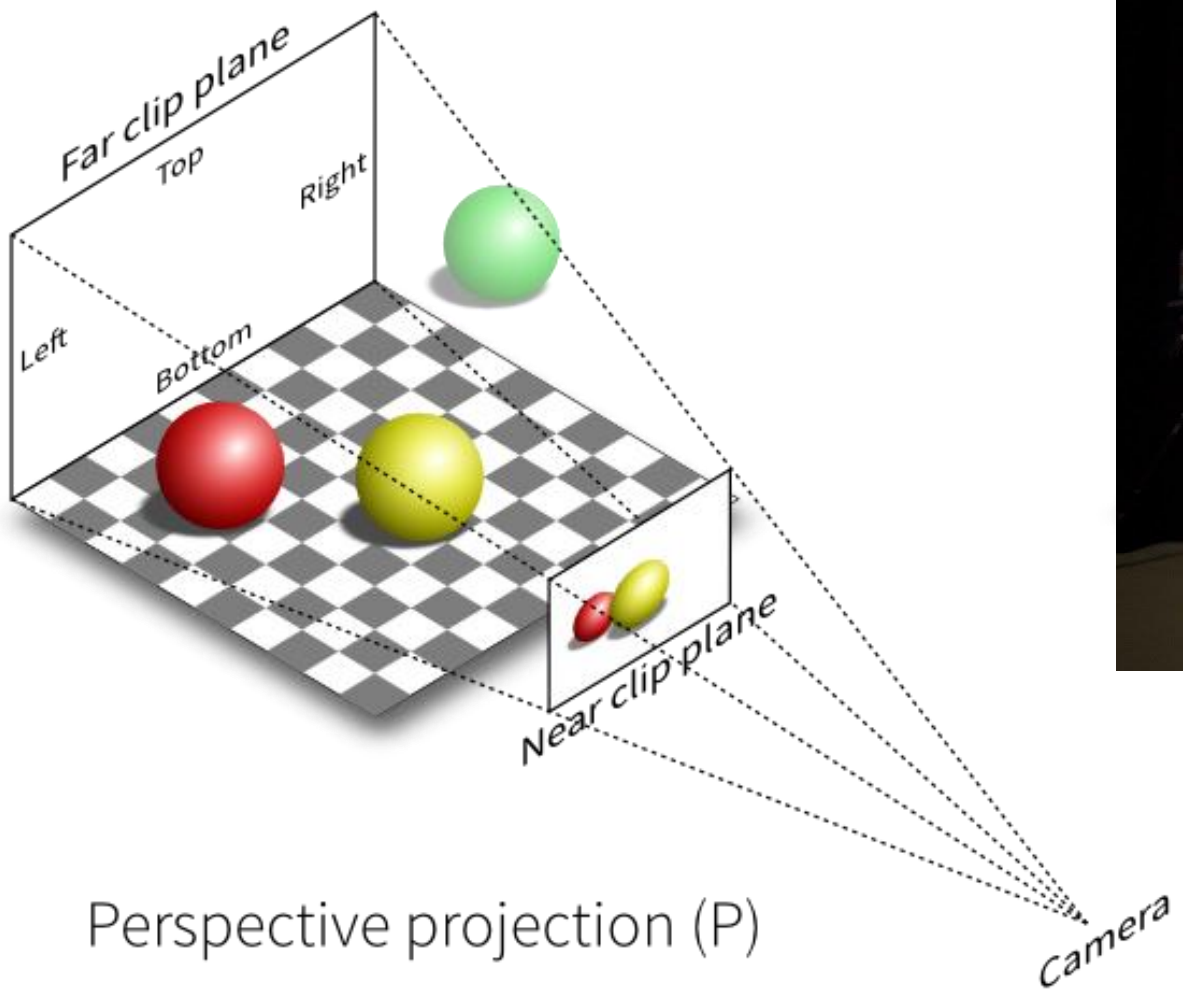
```
vec3 vertices[153];
vertices[0].position = { -0.54, +1.34, -0.01 };
vertices[1].position = { +0.75, +1.21, -0.01 };
…
vertices[152].position = { -1.22, +3.59, -0.01 };

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);
```



Image

A grid of color values

Virtual Camera

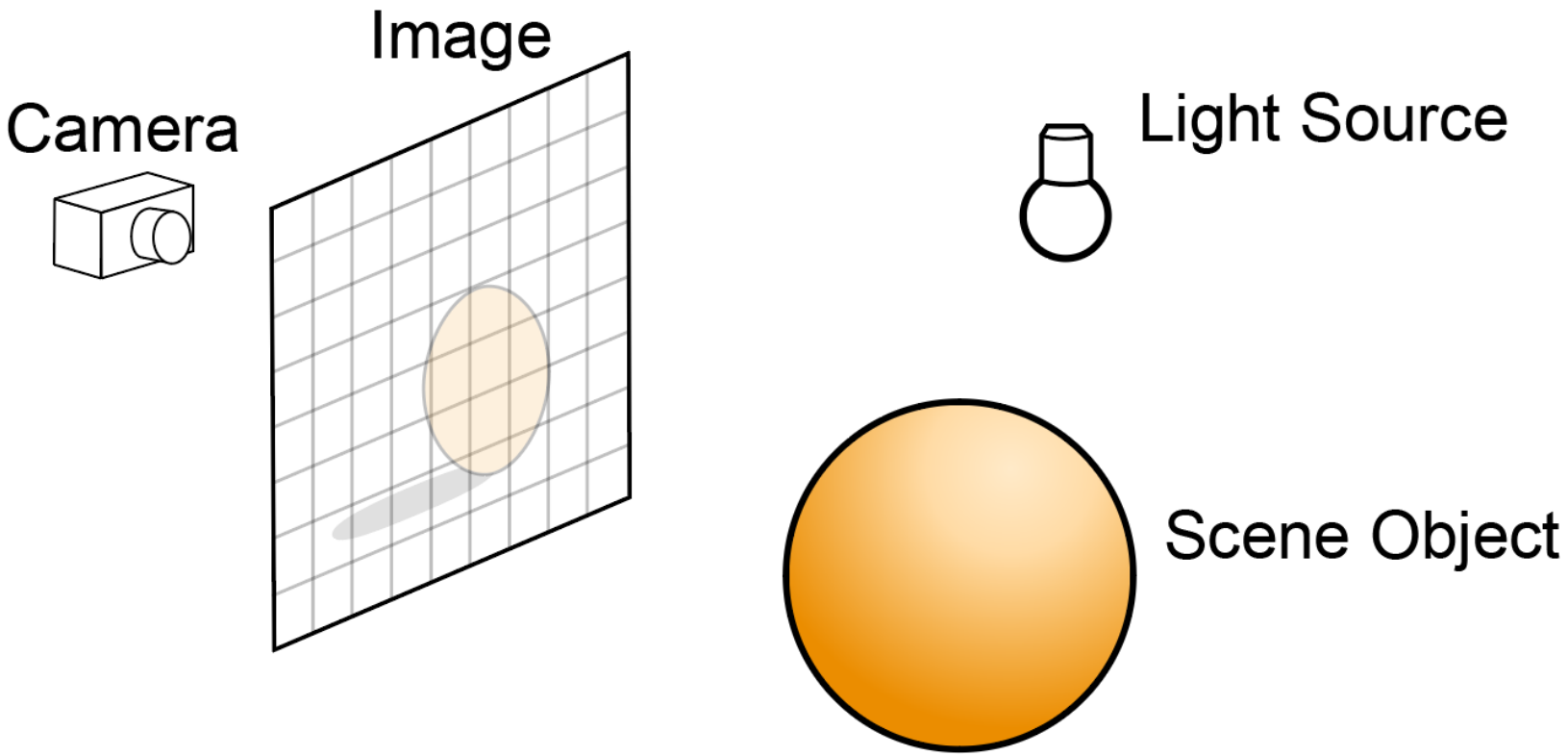


Perspective projection (P)

Virtual camera registered in the real world
(using marker-based motion capture)

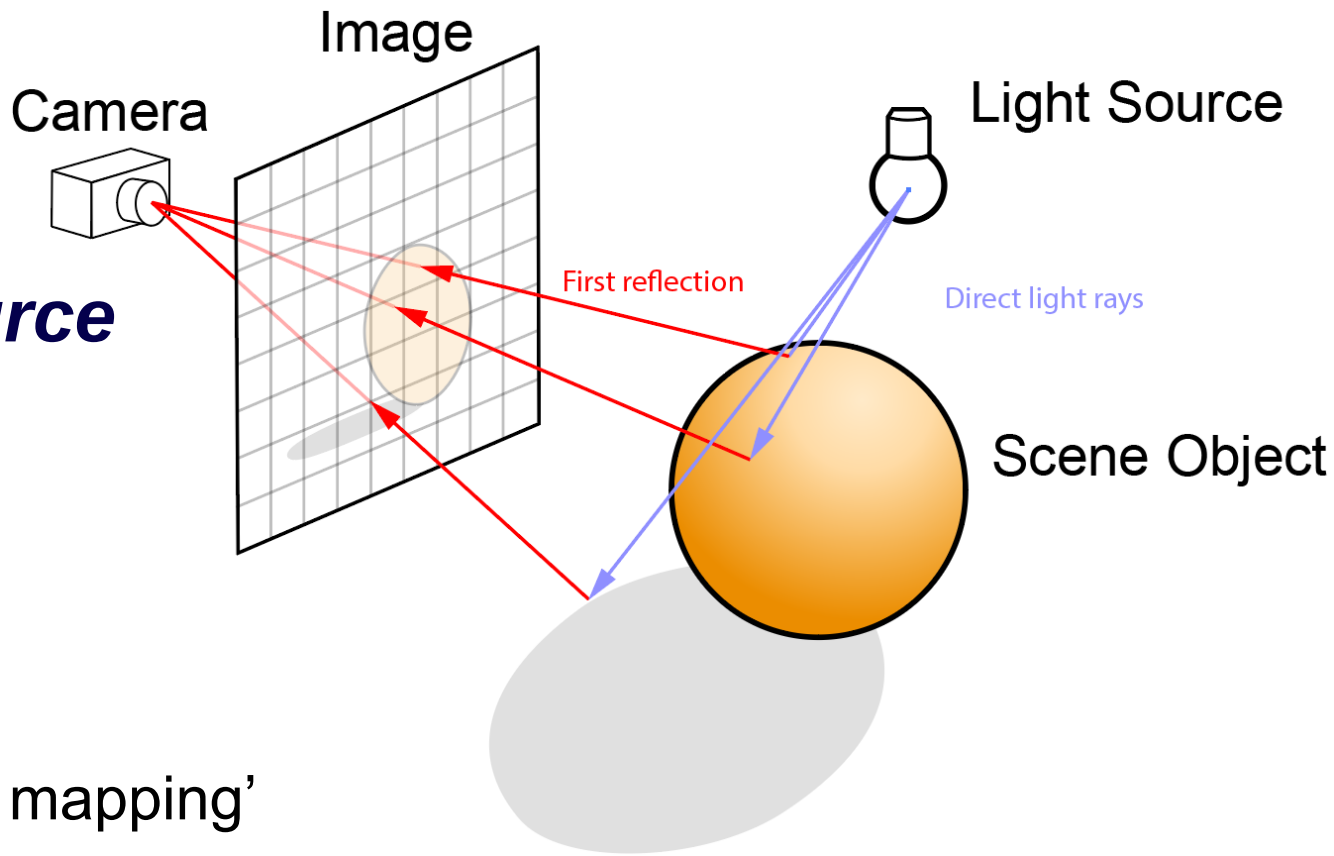
Rendering?

- ***Simulating light transport***
  - How to simulate light efficiently?



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*

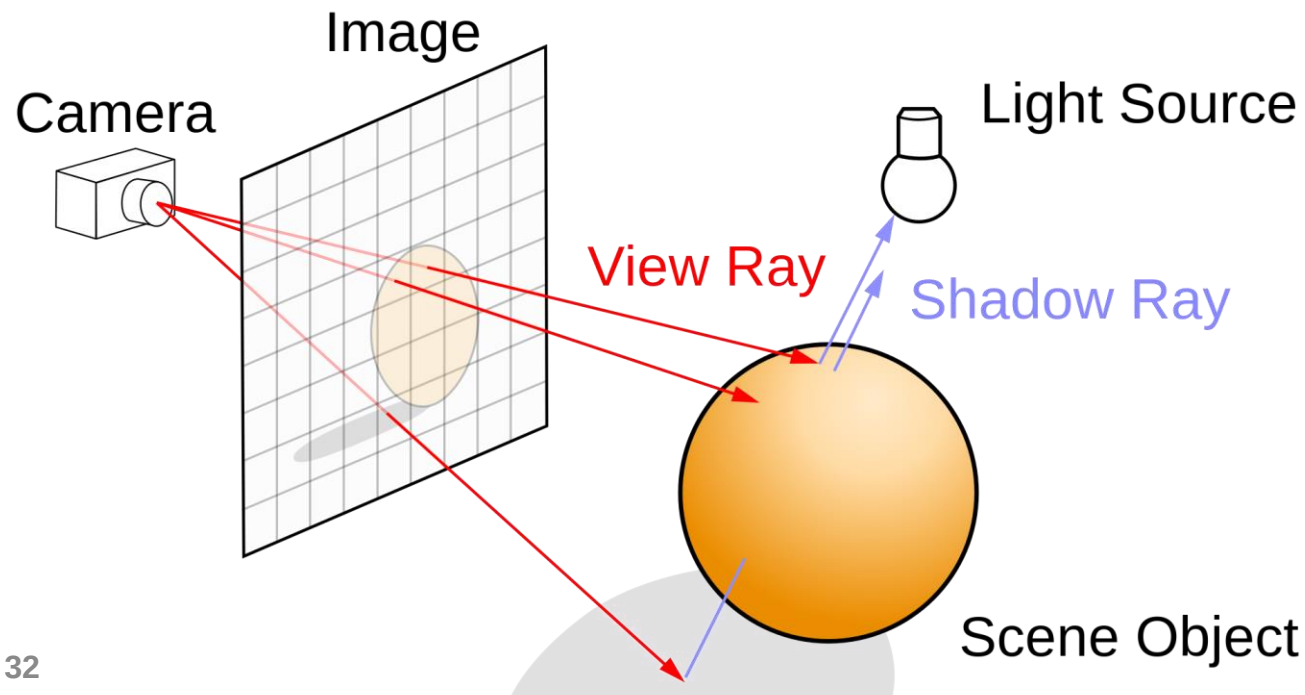


- used for indirect illumination: ‘photon mapping’

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

Problems of ray tracing

- ***the collision detection is costly***
  - ray-object intersection
    - *n objects*
    - *k rays*
    - *naïve: O(n*k) complexity*

Rendering – Splatting

Approximate scene with spheres

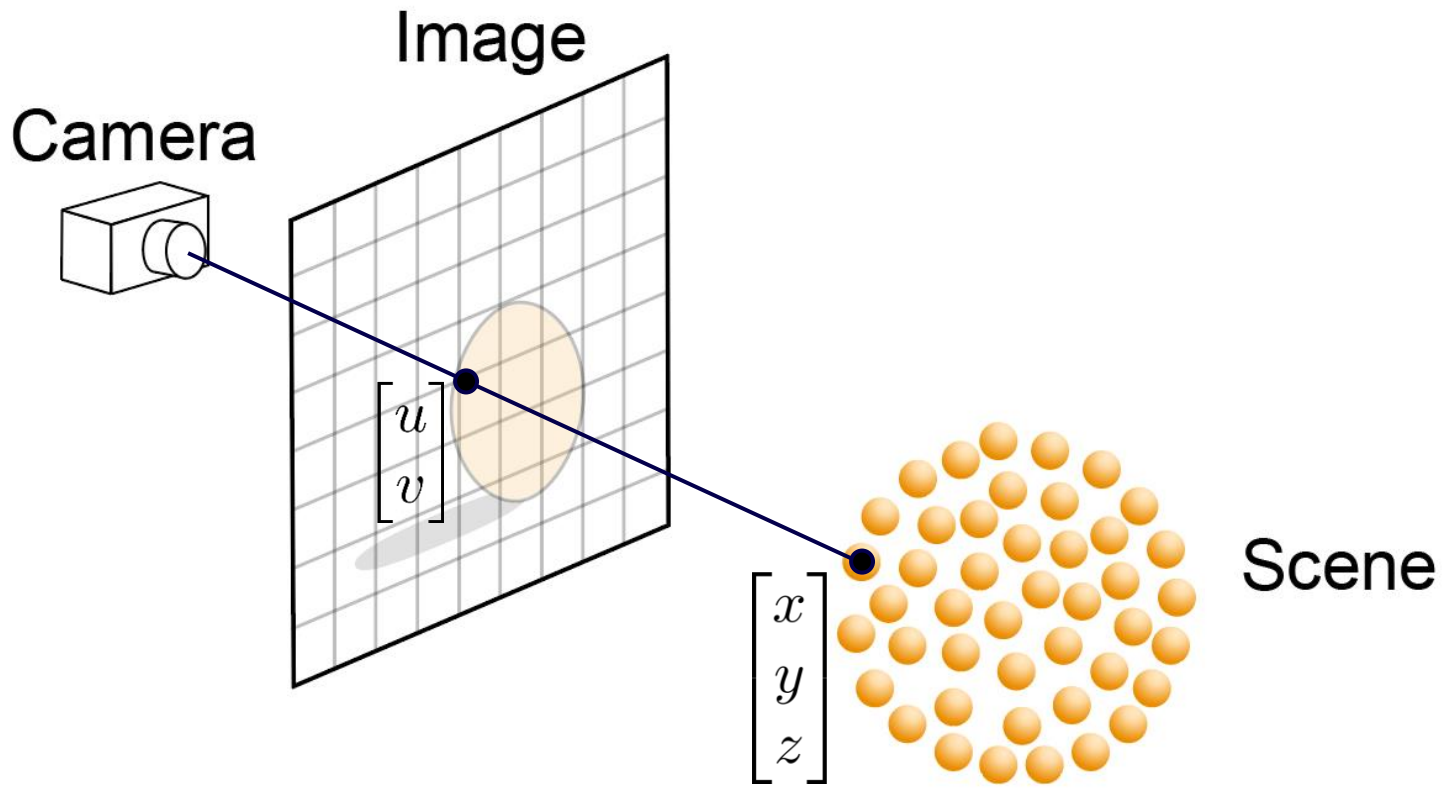
- *sort spheres back-to front*
- *project each sphere*
  - simple equation

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

  - O(n) for n spheres

Many spheres needed!

Shadows?



Rendering – Rasterization

Approximate objects with triangles

1. *Project each corner/vertex*

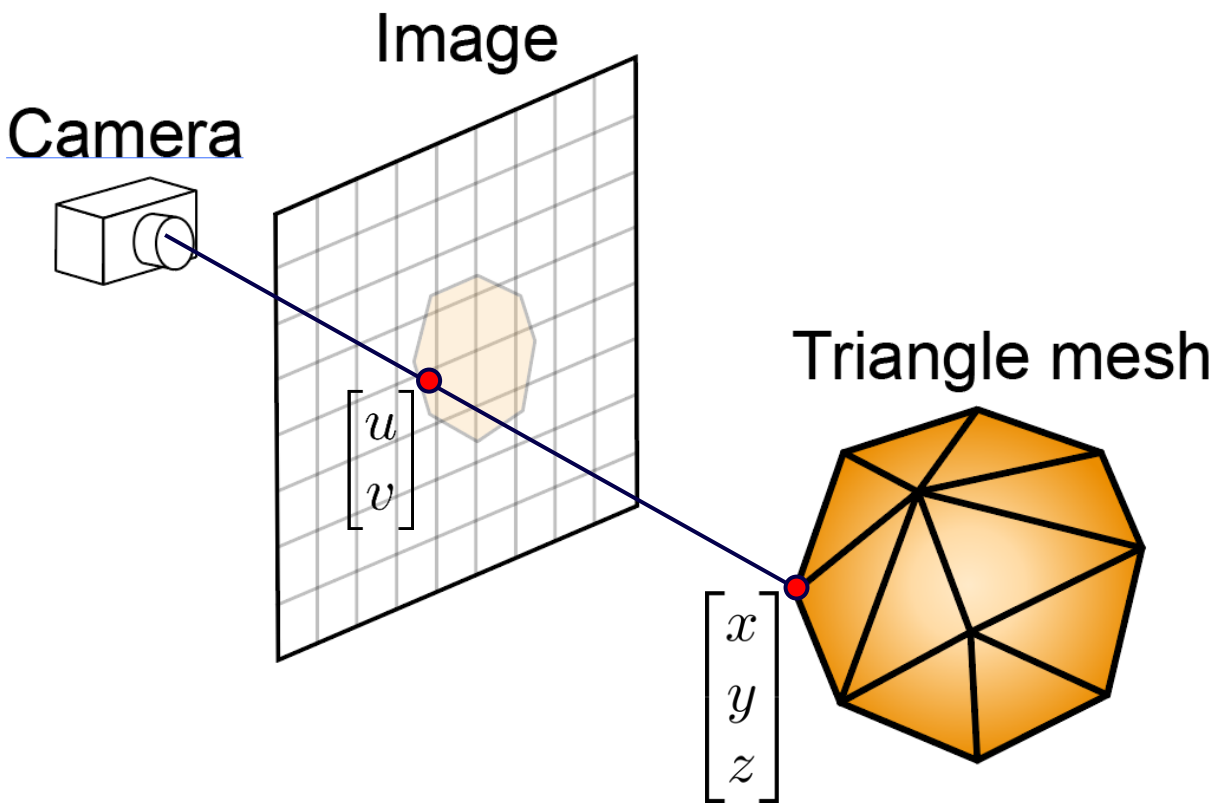
- projection of triangle stays a triangle

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

- O(n) for n vertices

2. *Fill pixels enclosed by triangle*

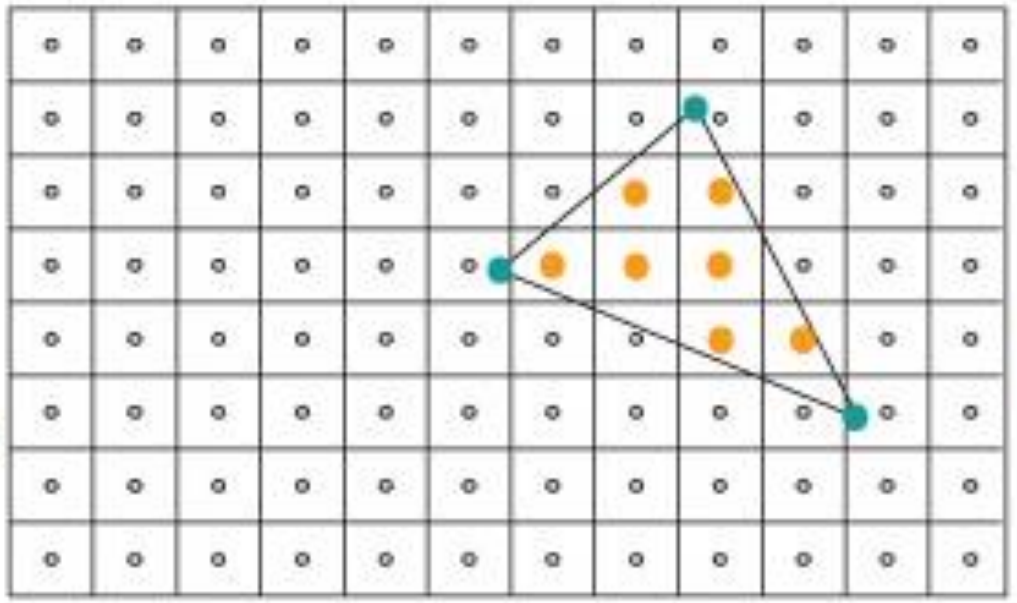
- e.g., scan-line algorithm



Rasterizing a Triangle

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

Vertices



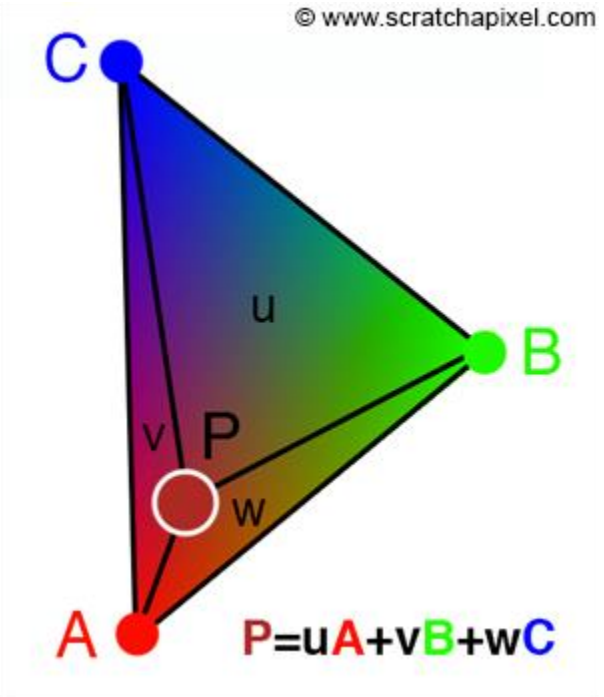
Fragments
(for every pixel; color or attributes to compute color: texture coordinate, direction, …)

Self study:

Interpolation with barycentric coordinates

- *linear combination of vertex properties*
  - *e.g., color, texture coordinate, surface normal/direction*

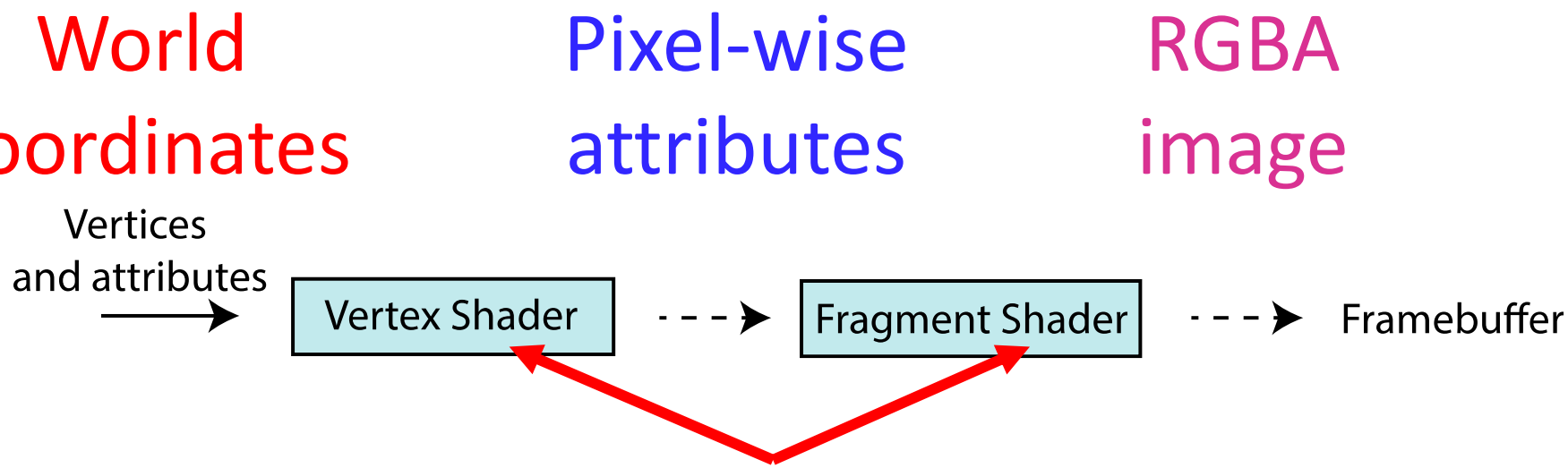
- *weights are proportional to the areas spanned by the sides to query point P*



Backup

OpenGL Rendering Pipeline (simplified)

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

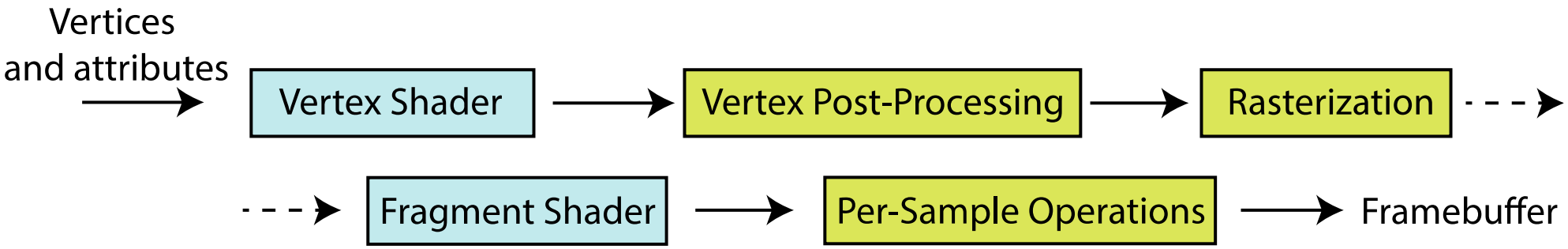


Shader: Programmable functions
to define object appearance locally
(vertex wise or fragment wise)

OpenGL Rendering Pipeline

Input:

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



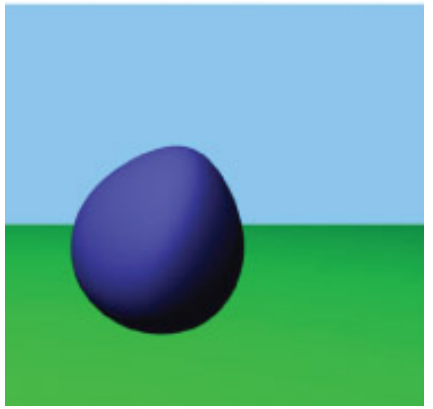
Output:

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

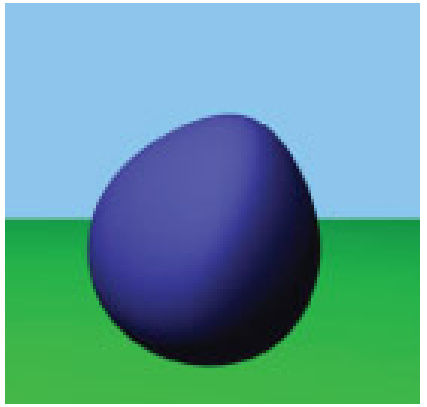
Vertex shader examples

Object motion & transformation

- translation
- rotation
- scaling



Translation



Scaling

Projection

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

GLSL Vertex shader

The OpenGL Shading Language (GLSL)

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

```
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

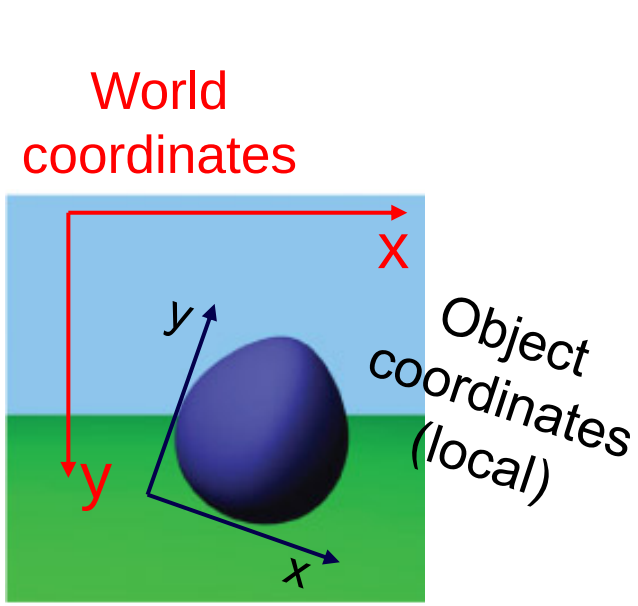
object -> world

x and y coordinates of a vec2, vec3 or vec4

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

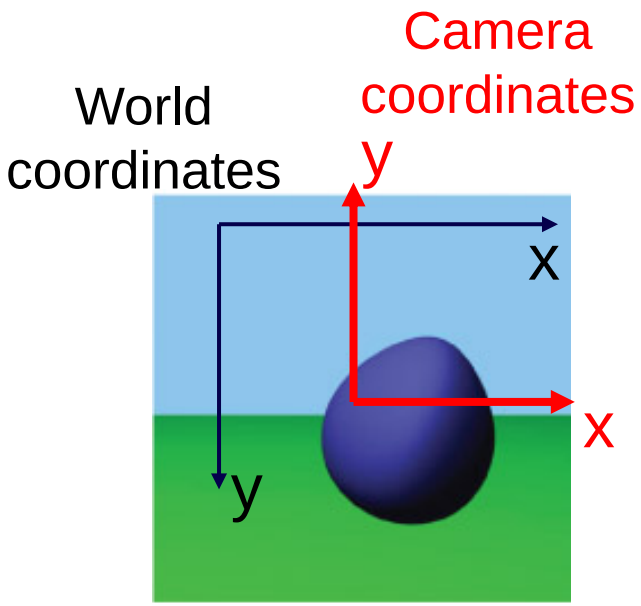
float (32 bit)

From local object to camera coordinates



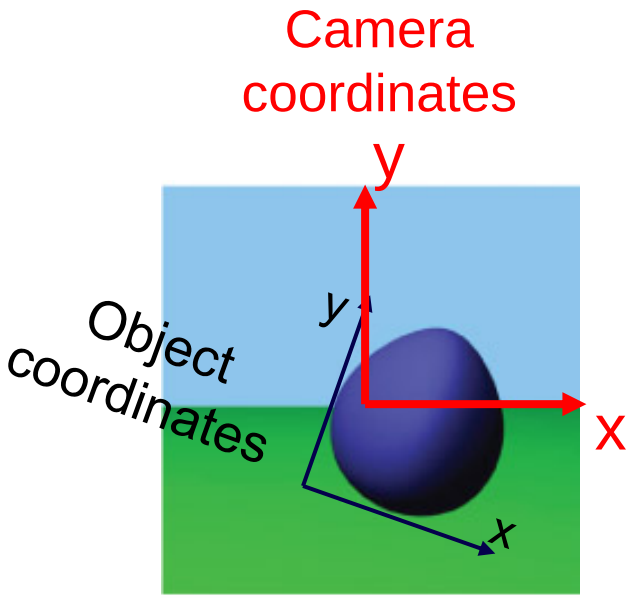
object -> world

`transform`



world -> camera

`projection`

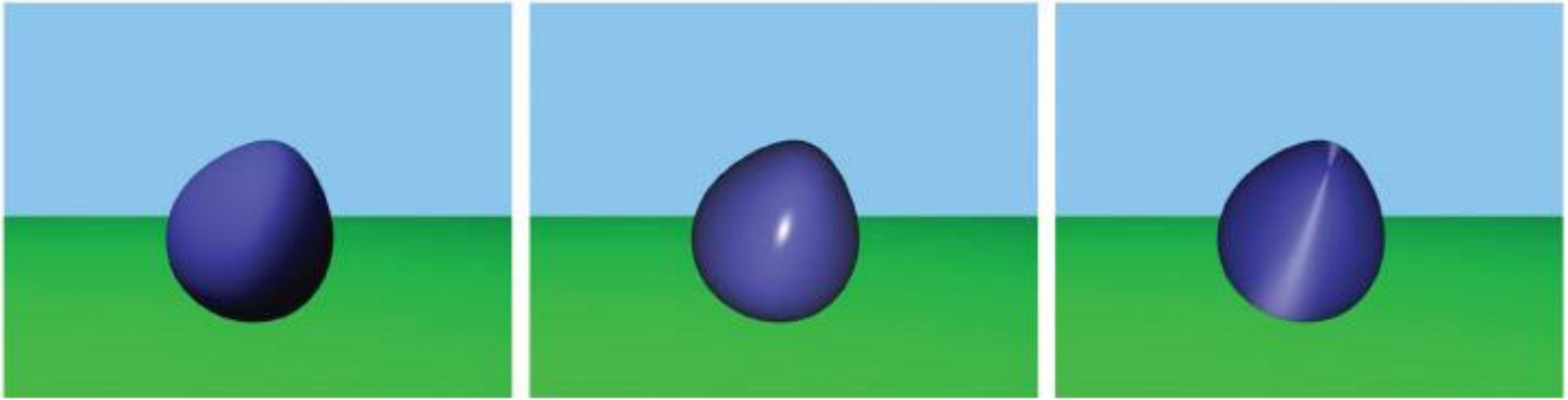


object -> camera

`projection * transform`

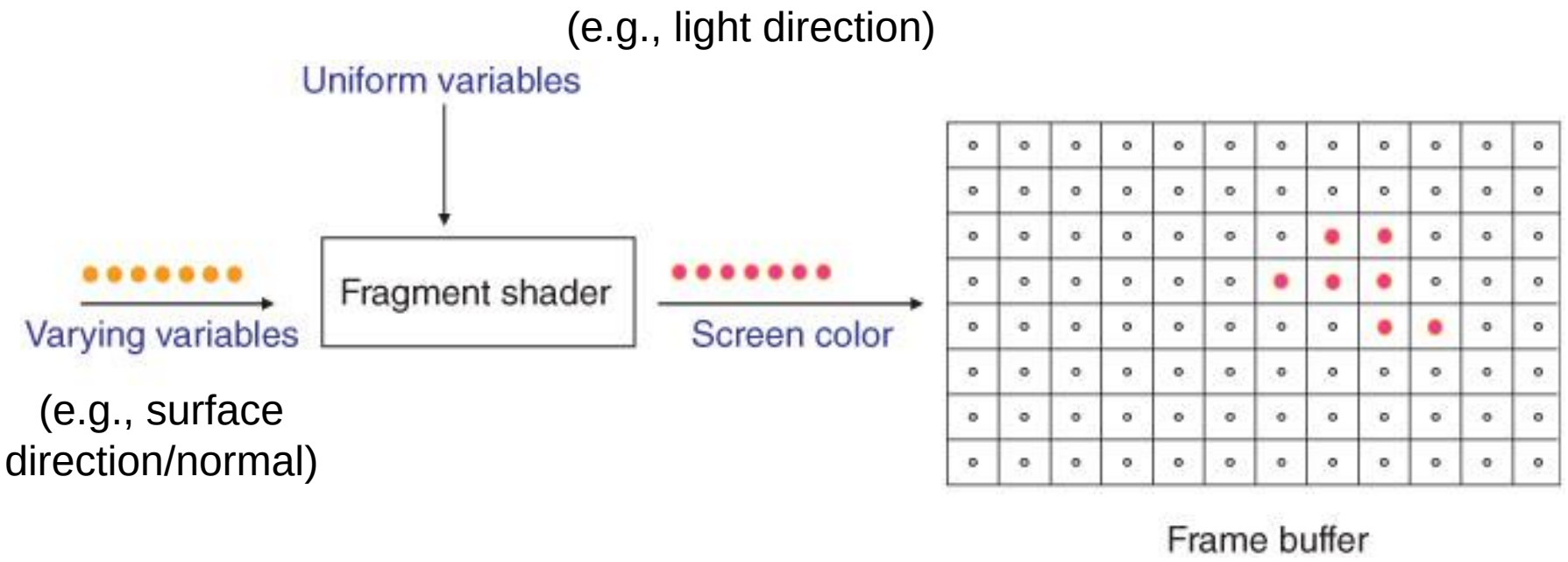
Fragment shader examples

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



Diffuse **Specular** **Directional**

Fragment shader overview



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