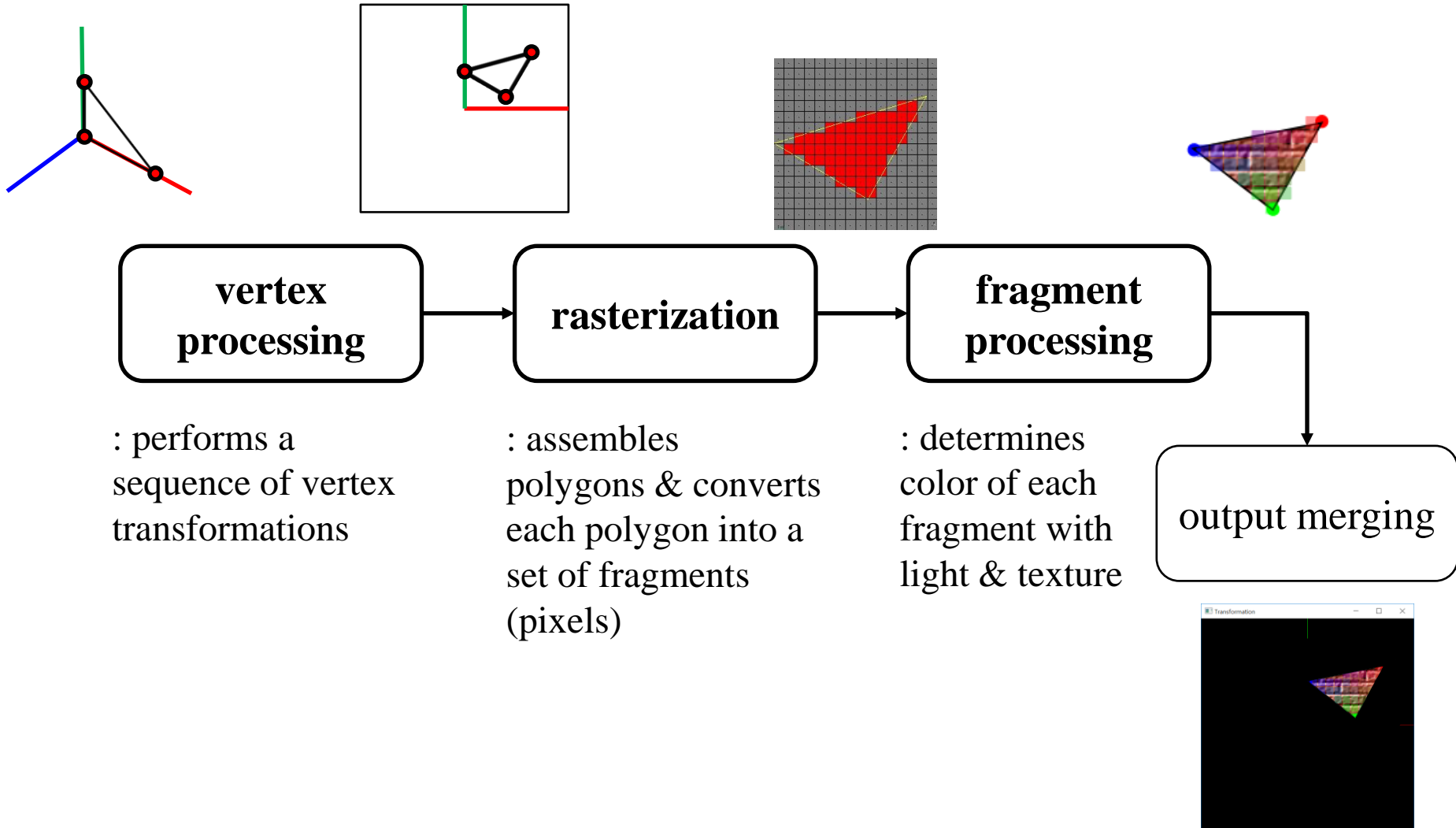# Computer Graphics

13 - Rasterization & Visibility

Yoonsang Lee

Spring 2022

Topics Covered

- Two Approaches for Rendering
  - Object-oriented (Rasterization)
  - Image-oriented (Raytracing)

- Rasterization (in a narrow sense)
  - Line / Polygon Drawing

- Visibility Problem
  - Clipping (Viewing frustum culling)
  - Back-face culling
  - Hidden surface removal

- Rendering (Graphics) Pipeline Again

- Course Wrap-up

Recall: Rendering (Graphics) Pipeline

vertex processing → **rasterization** → **fragment processing** → **output merging**

: performs a sequence of vertex transformations

: assembles polygons & converts each polygon into a set of fragments (pixels)

: determines color of each fragment with light & texture
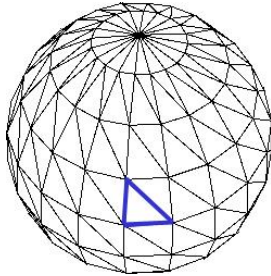
Two Approaches for Rendering - 1

```
for each object in scene
    transform the object to viewport     # vertex processing
    find pixels for the object     # rasterization (in a narrow sense)
    set color of the pixels based on texture and lighting
model                                          # fragment
                                                 processing
```
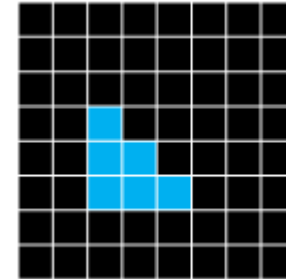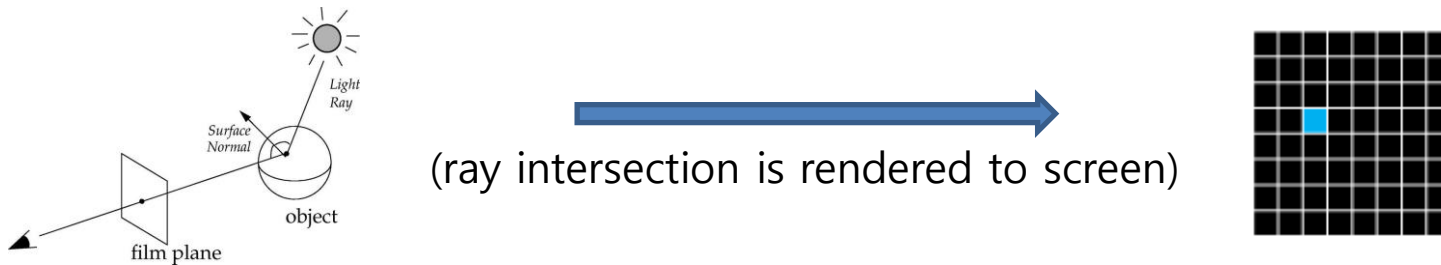
(triangle is rendered to screen)

- This is called **rendering (graphics) pipeline**
- or **rasterization** (in a broad sense)
- or **object-oriented rendering**.

Two Approaches for Rendering - 2

```
for each pixel in image(film plane)
    determine which object should be shown at the pixel
    set color of the pixel based on texture and lighting model
```


(ray intersection is rendered to screen)

- This is called **ray tracing**
- or **image-oriented rendering.**

- We'll skip ray tracing part, see *13 - reference-RayTracing.pdf* for more information about it.

Rasterization (in a broad sense) & Ray Tracing in this Course

- Most topics we've covered are *fundamental concepts* of computer graphics, regardless of these two rendering approaches.
  - Transformations, Hierarchical Modeling, Orientation & Rotation, Kinematics & Animation
  - Mesh, Lighting & Shading, Curves, Texture Mapping

- Except some topics:
  - Rendering Pipeline, Viewing / Projection / Viewport transformations
  - Rasterization & Visibility (today's topic)
- are specific to **rasterization** (in a broad sense).

Rasterization (in a narrow sense)

- Rasterization converts vertex representation to pixel representation (fragments)
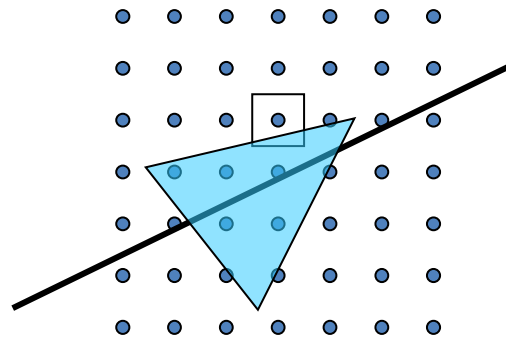
- First job: Compute which pixels belong to a primitive
  - to enumerate the pixels covered by the primitive

- Second job: Interpolate values across the primitive
  - e.g. colors computed at vertices
  - e.g. normals at vertices

Rasterization (in a narrow sense)

- A primitive can be a point, line, or polygon.

- Line drawing algorithms
  - Digital differential analyzer (DDA)
  - Bresenham's (a.k.a. Midpoint) (1962)
  - Xiaolin Wu's (1991)

- Polygon drawing algorithms
 - Scanline
 - Boundary fill
  - Flood fill

Rasterization (in a narrow sense)

- But, we'll just skip details of these algorithms.

- Actually, line drawing and polygon drawing are not so easy as one might think.
  - Computational efficiency, anti-aliasing, ...

- But most graphics APIs (including OpenGL) basically support these operations.
  - These algorithms were intensively studied in early days of computer graphics, so quite mature now.
  - Now basic algorithms are implemented in graphics hardware (GPU) and you can use them by calling such graphics APIs.

- So nowadays you can think lines and polygons as "primitives" that are basically rendered.
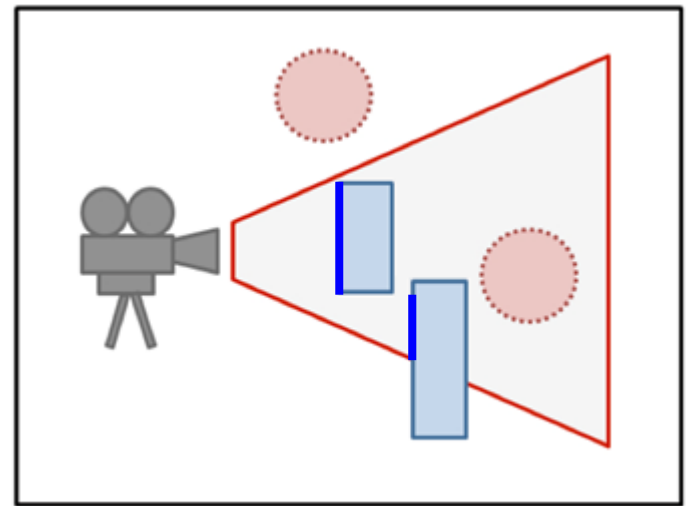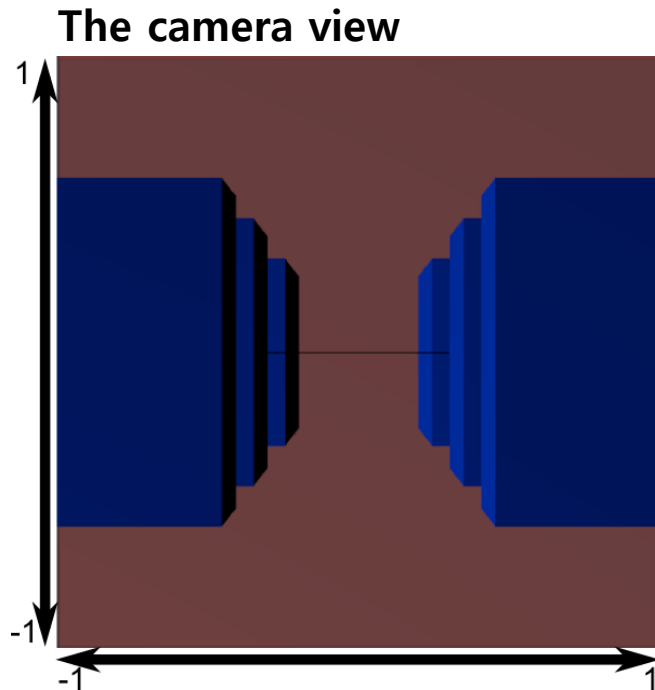
Visibility Problem

- What is VISIBLE?
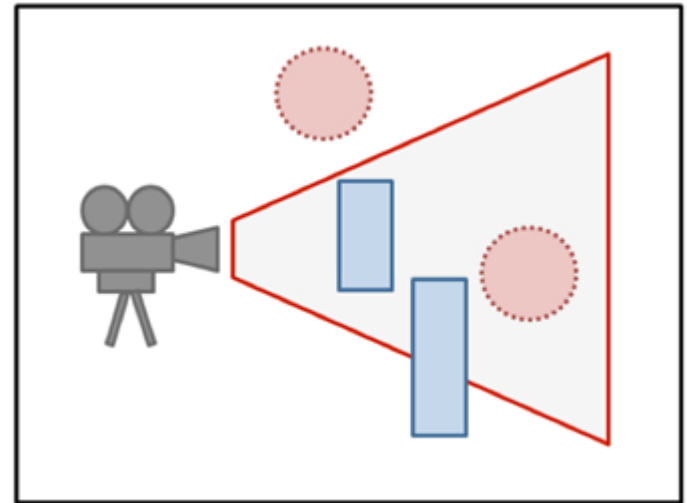
Red: viewing frustum, Blue: objects
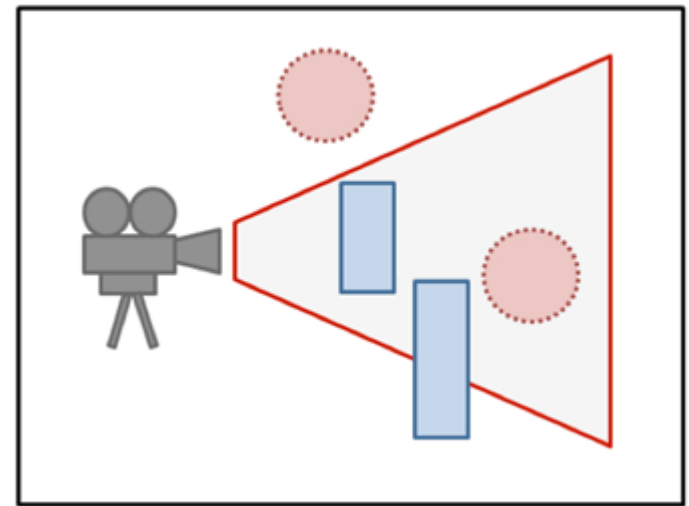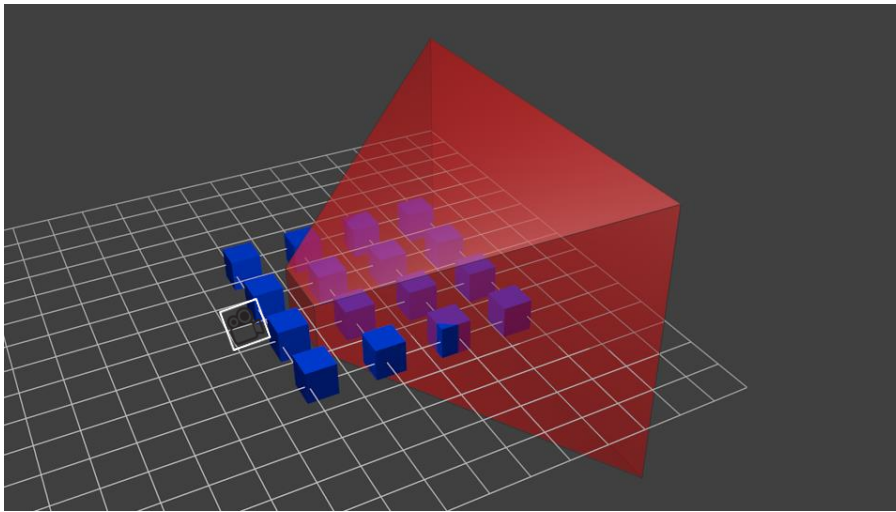
Visibility Problem

- The answer is:
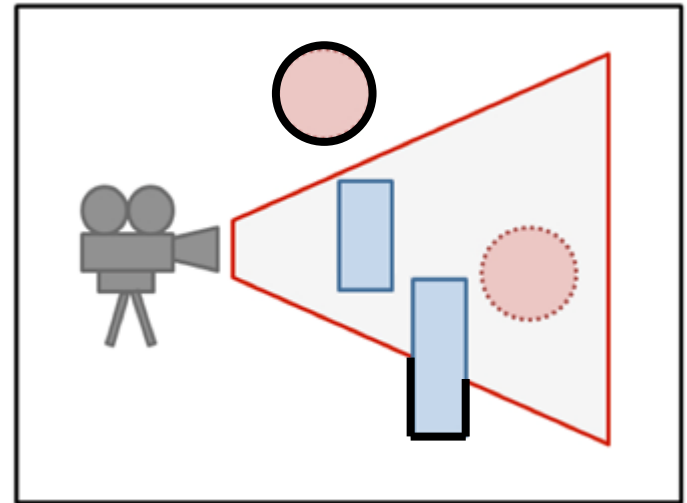
The camera view

Visibility Problem
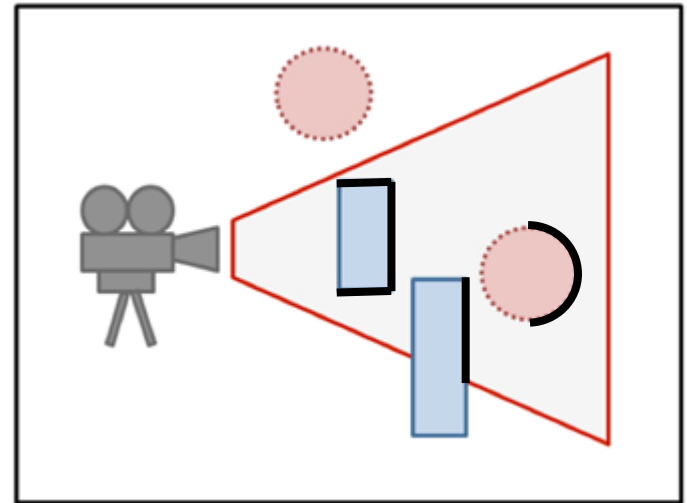
- What is NOT VISIBLE?

Visibility Problem

- What is NOT VISIBLE?

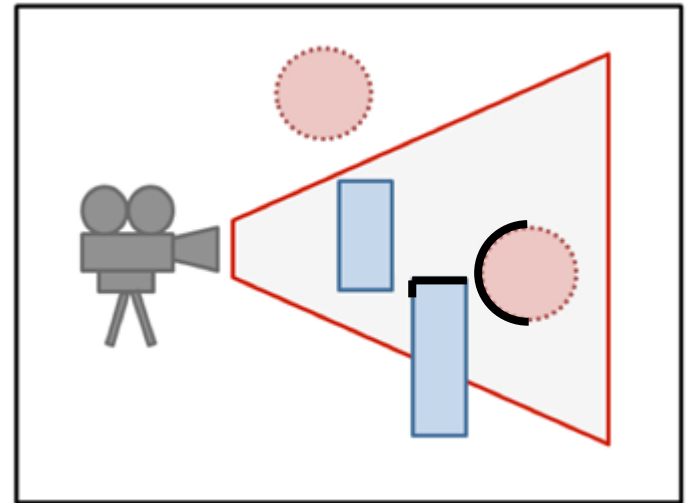- **Primitives outside of the viewing frustum**

Visibility Problem

- What is NOT VISIBLE?

- Primitives outside of the viewing frustum
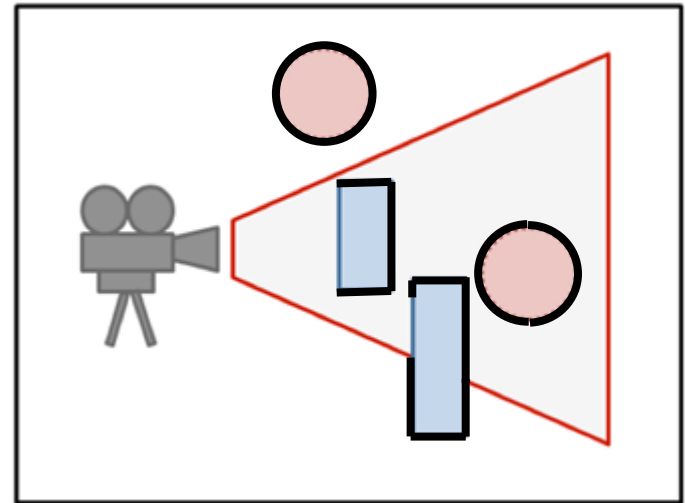
- **Back-facing primitives**

Visibility Problem

- What is NOT VISIBLE?

- Primitives outside of the viewing frustum

- Back-facing primitives

- **Primitives occluded by other objects closer to the camera**

Visibility Problem

- These **invisible primitives should be removed** because…

- No need to spend time to process invisible vertices and polygons.

- A close object must hide a farther one.

- So, removing these primitives is required for efficient and correct rendering.

Visibility Problem

- Removing...

- Primitives outside of the viewing frustum
- → **Clipping (Viewing frustum culling)**

- Back-facing primitives
- → **Back-face culling**

- Primitives occluded by other objects closer to the camera
- → **Hidden surface removal**

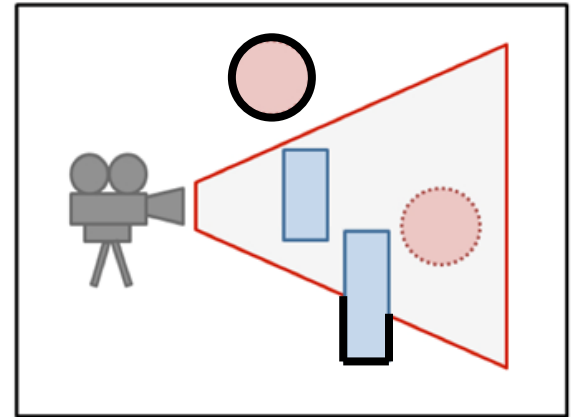# Clipping (Viewing Frustum Culling)

- Removing primitives outside of the viewing frustum

- Clipping is much easier with canonical view volume.
  - actually done in *clip space*
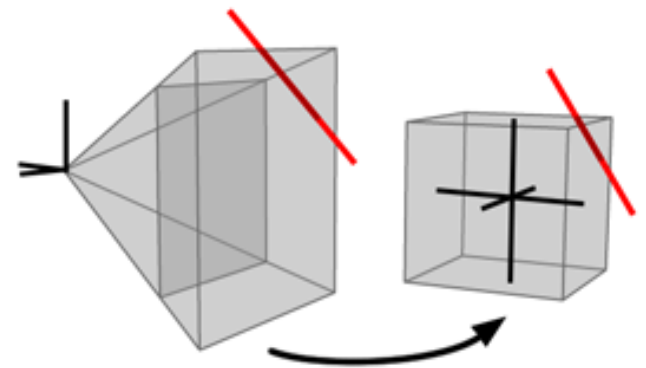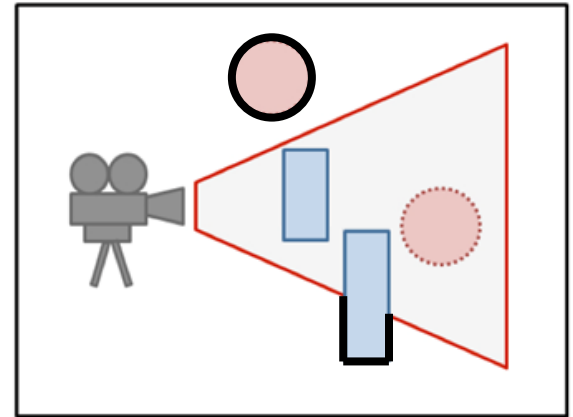
Clipping (Viewing Frustum Culling)

- Line clipping algorithms
  - Cohen–Sutherland (1967)
  - Cyrus–Beck (1978)
  - Liang–Barsky (1984)

- Polygon clipping algorithms
  - Sutherland–Hodgman (1974)
  - Weiler–Atherton (1977)

Clipping (Viewing Frustum Culling)

- Polygon clipping algorithms are more complicated.
  - Vertices may be added to or deleted from the triangle.

- Again, let's just skip details of these algorithms.

- Most graphics APIs (including OpenGL) performs clipping by default.
  - You just set the view frustum, then OpenGL will do clipping for you.

- *13 - reference-rasterization,clipping.pdf* has brief slides about DDA (line drawing) & Cohen-Sutherland algorithms (line clipping). If you're interested, please refer it.
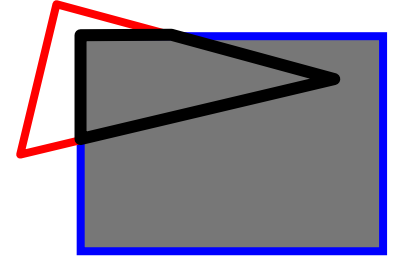
triangle → quad

Back-Face Culling

- Removing back-facing primitives

- Determined by the dot product of normal and view (camera) vectors.

$n_3 \cdot c_3 = 0$
edge-on face

Discard!

$n \cdot c = ||n|| ||c|| cos\theta$

$n_1 \cdot c_1 > 0$
back face

$n_2 \cdot c_2 < 0$
front face
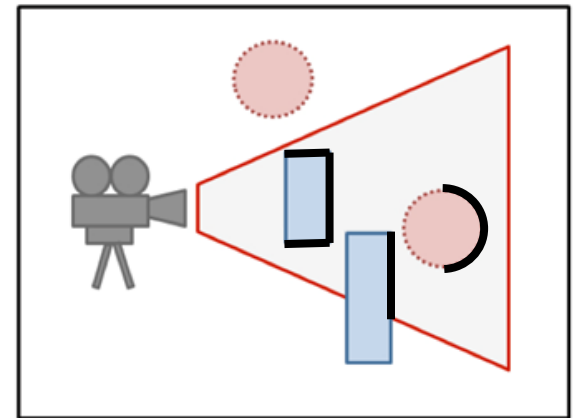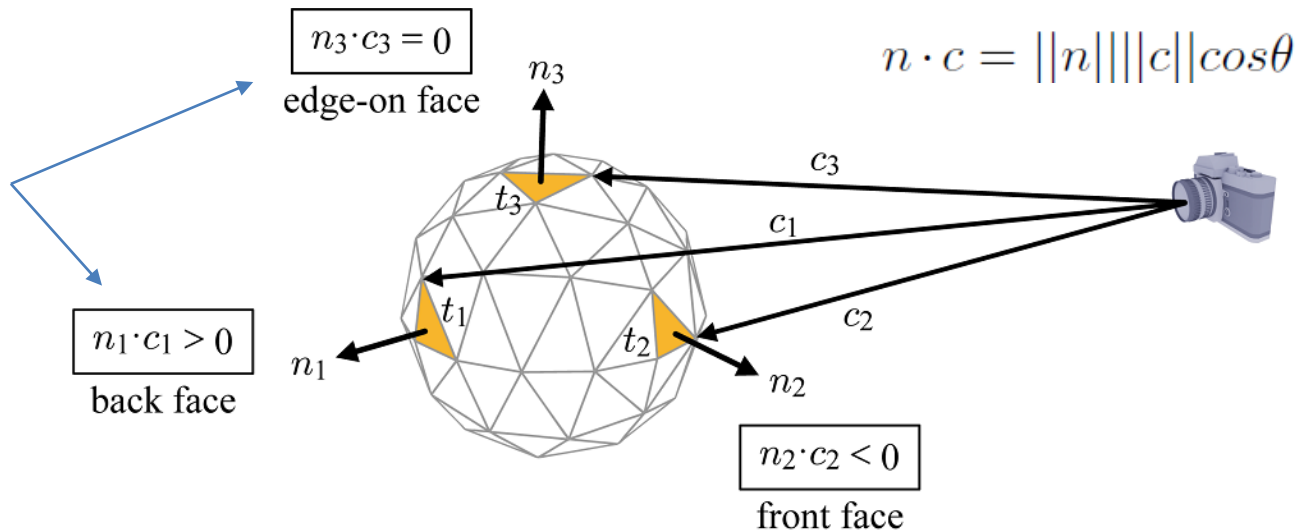
Back-Face Culling

- Back-face culling is much more efficient when performed in canonical view volume.

    - Because in canonical view volume, we can use a single view vector, (0,0,1).

camera space
(in RHS)
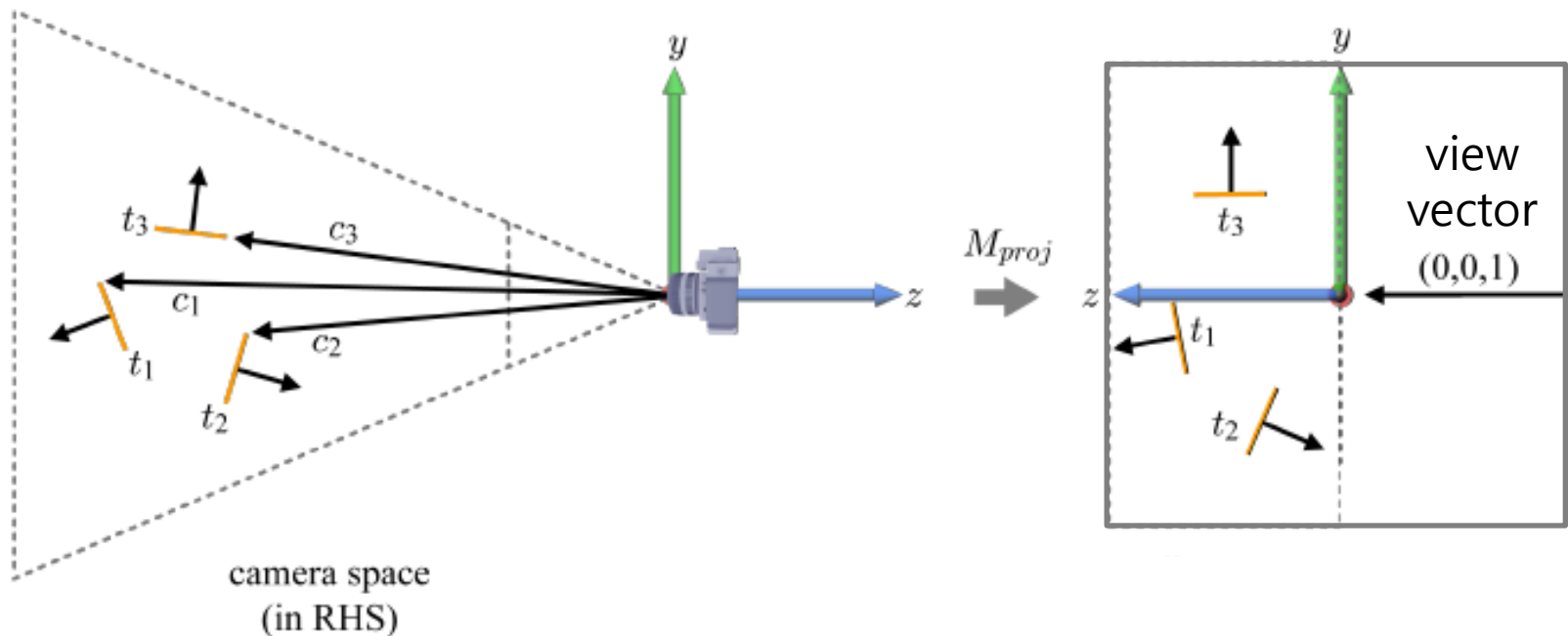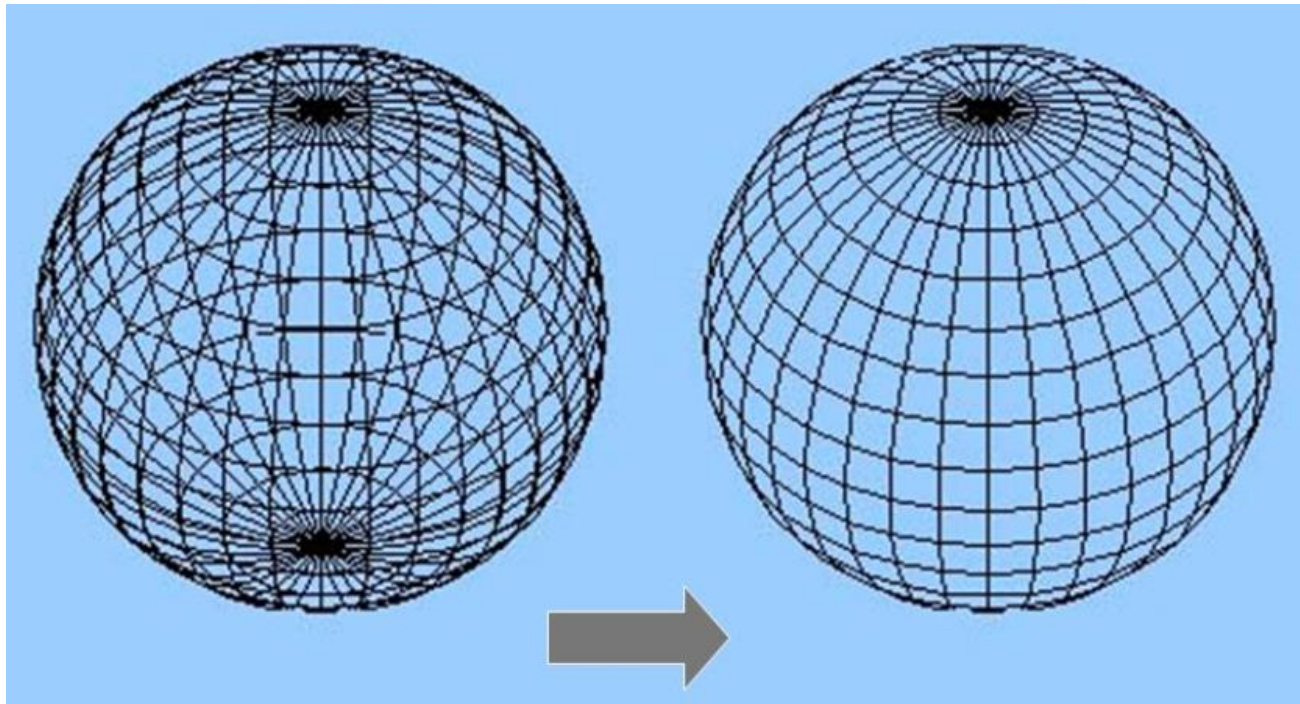
Back-Face Culling

Back-Face Culling in OpenGL

- **Can cull front faces or back faces**

- **Back-face culling can sometimes double performance**

```
if (cull):                          (initial value: GL_CCW)
    glFrontFace(GL_CCW)      # define winding order
    glEnable(GL_CULL_FACE)   # enable Culling(initially disabled)
    glCullFace(GL_BACK)      # which faces to cull
else:
    glDisable(GL_CULL_FACE)
```

You can also do front-face culling!

Trackball demo
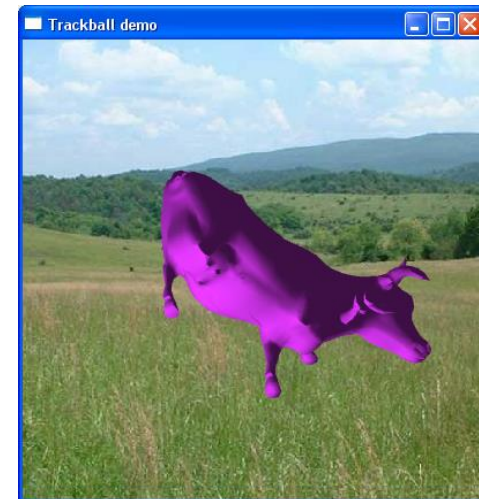
Hidden Surface Removal

- Removing primitives occluded by other objects closer to the camera

- Also known as
  - Hidden Surface Elimination
  - Hidden Surface Determination
  - Visible Surface Determination
  - Occlusion Culling
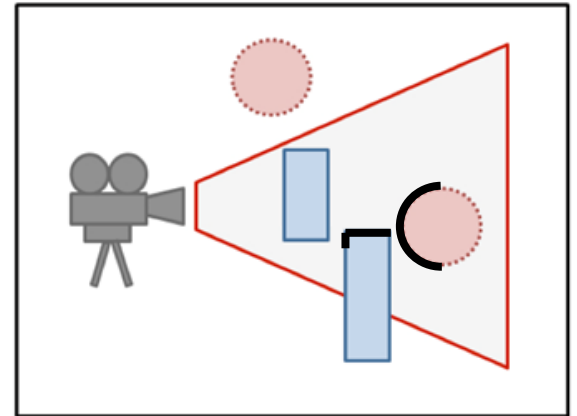
Hidden Surface Removal

- Many algorithms
  - Z-buffer (Depth buffer)
  - Painter's algorithm
  - BSP tree
  - ...

- Z-buffer is the standard method.

- Let's see the ideas of Painter's algorithm & Z-buffer.

Frame Buffer (background knowledge for understanding HSR algorithms)

- Frame buffer is the portion of memory to hold the bitmapped image that is sent to the (raster) display device.

- A frame buffer is characterized by its width, height, and depth.
  - E.g. The frame buffer size for 4K UHD resolution with 32bit color depth = 3840 x 2160 x 32 bits
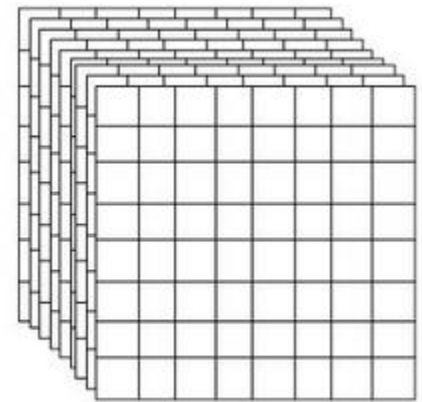
- Typically stored on the graphic card's memory.
  - But integrated graphics (e.g. Intel HD Graphics) use the main memory to store the frame buffer.

Painter's algorithm

- Simplest way to do hidden surfaces

- Draw from back to front, use overwriting in framebuffer

- Requires sorting all polygons by their depth
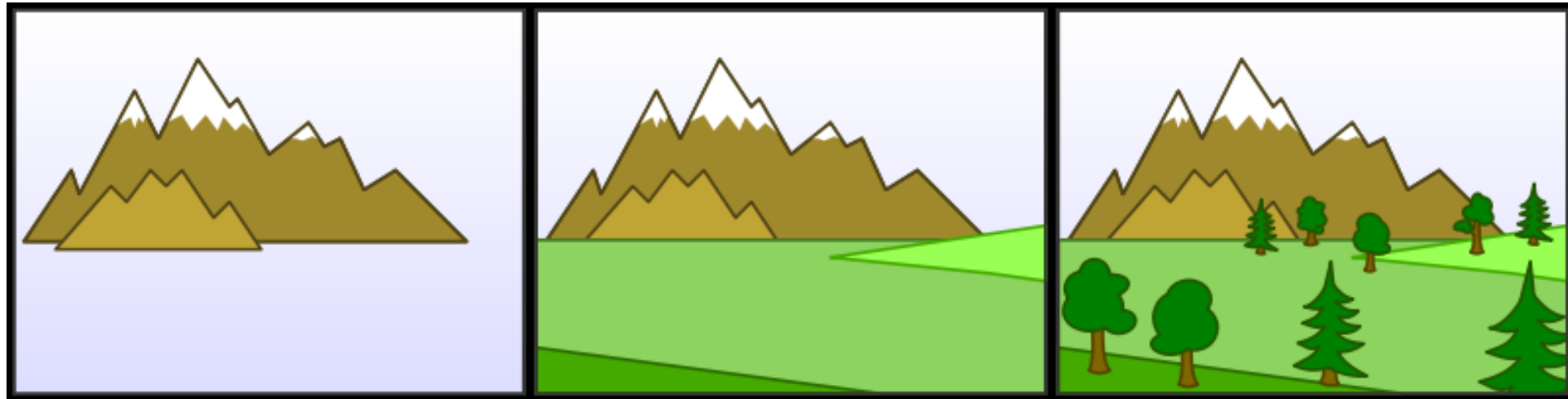
Weakness of Painter's Algorithm

- What if there are cycles in the sorted graph?
  - The only solution is dividing these polygons into small pieces.

- Need to update the sorted graph whenever camera or object location is changed.

- → Time-consuming!
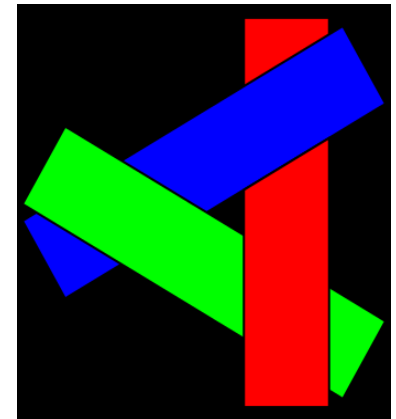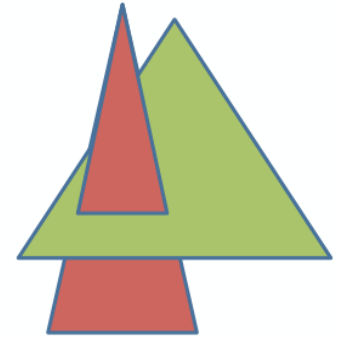
The *z* buffer

- In many (most) applications maintaining a *z* sort is too expensive
  - changes all the time when the view changes
  - many data structures exist, but complex
- Solution: draw in any order, keep track of closest
  - Z-buffer keeps track of closest depth so far
  - when drawing, compare object's depth to current closest depth and discard if greater

Z-Buffering: Algorithm

```
allocate depth_buffer;            // Allocate depth buffer → Same size as viewport.

for each pixel (x,y)              // For each pixel in viewport.
  write_frame_buffer(x,y,backgrnd_color);    // Initialize color.
  write_depth_buffer(x,y,farPlane_depth);    // Initialize depth (z) buffer.

for each polygon                 // Draw each polygon (in any order).
  for each pixel (x,y) in polygon   // Rasterize polygon.
    color = polygon's color at (x,y);
    p_z = polygon's z-value at (x,y);// Interpolate z-value at (x, y).
    if (p_z < read_depth_buffer(x,y))      // If new depth is closer:
      write_frame_buffer(x,y,color);        //  Write new (polygon) color.
      write_depth_buffer(x,y,p_z);          //  Write new depth.
```
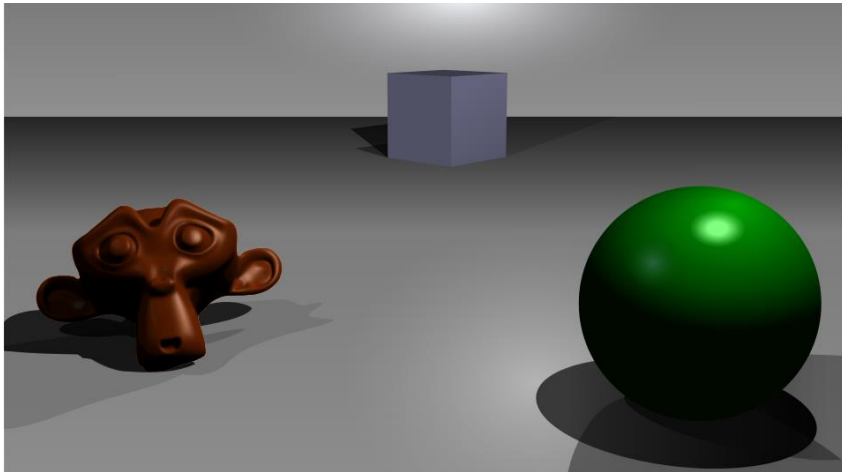

Frame buffer

Z-buffer (Depth buffer)

Example: rendering three opaque triangles
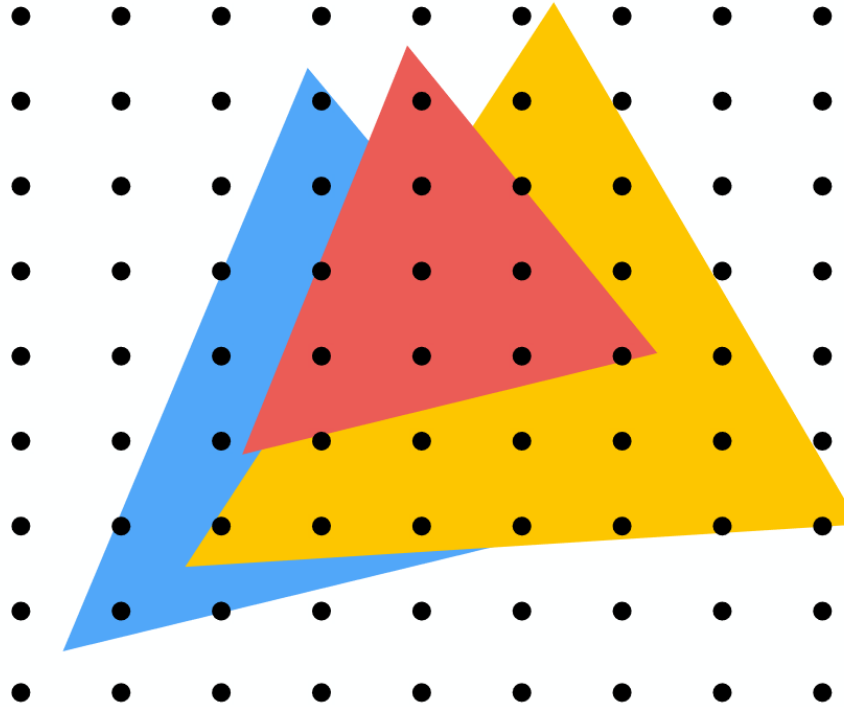
Occlusion using the depth-buffer (Z-buffer)

Processing yellow triangle:

depth = 0.5
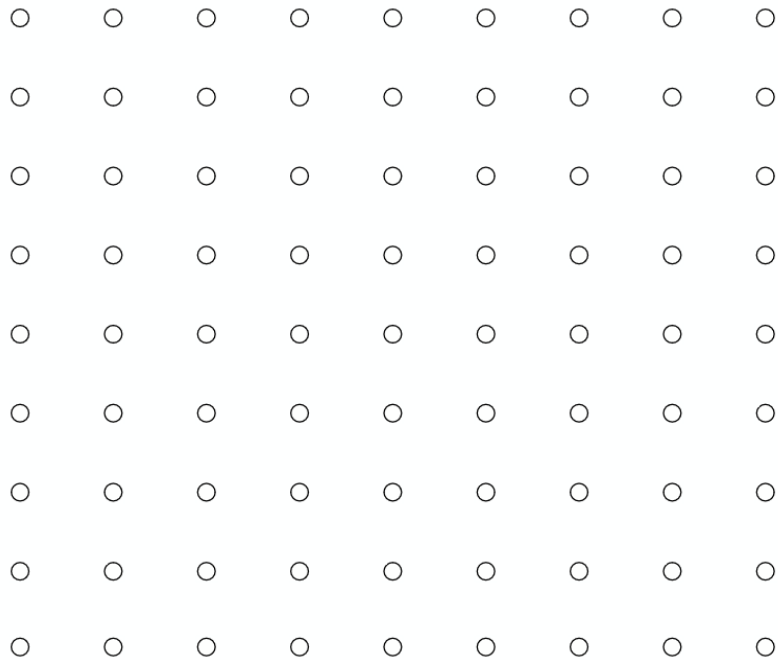
Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

Red = sample passed depth test

Color buffer contents

Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing yellow triangle:
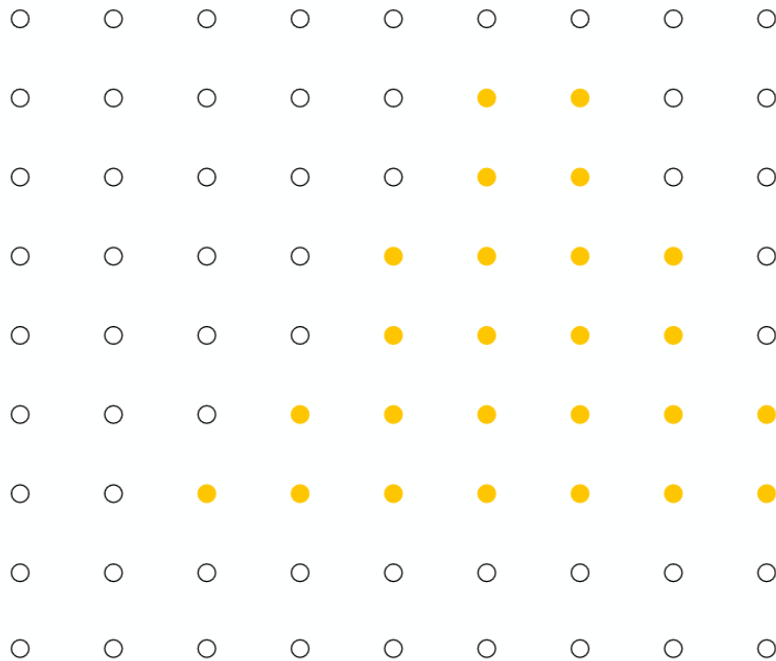
Grayscale value of sample point used to indicate distance

White = large distance
Black = small distance
Red = sample passed depth test

Color buffer contents

Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

Processing blue triangle:

$depth = 0.75$
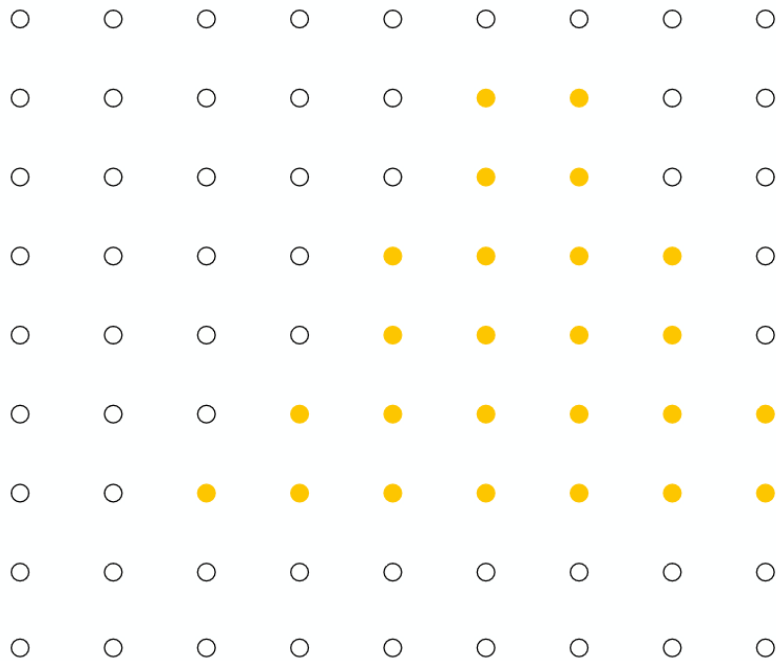
Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

Red = sample passed depth test

Color buffer contents
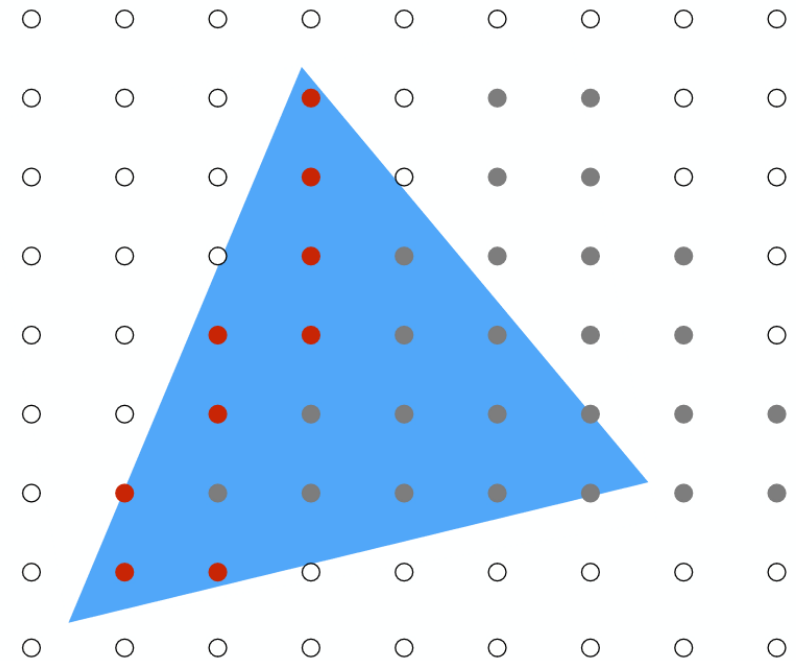
Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing blue triangle:
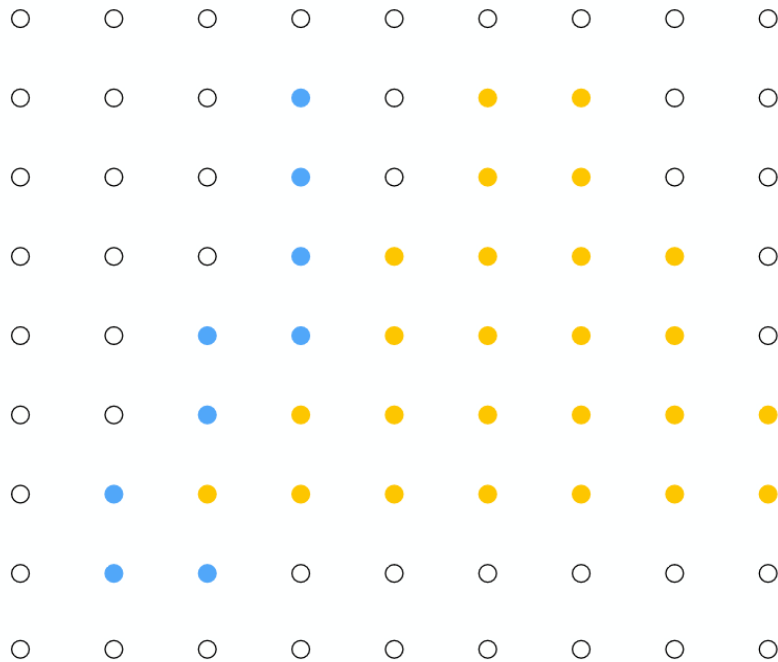
Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

Red = sample passed depth test

Color buffer contents
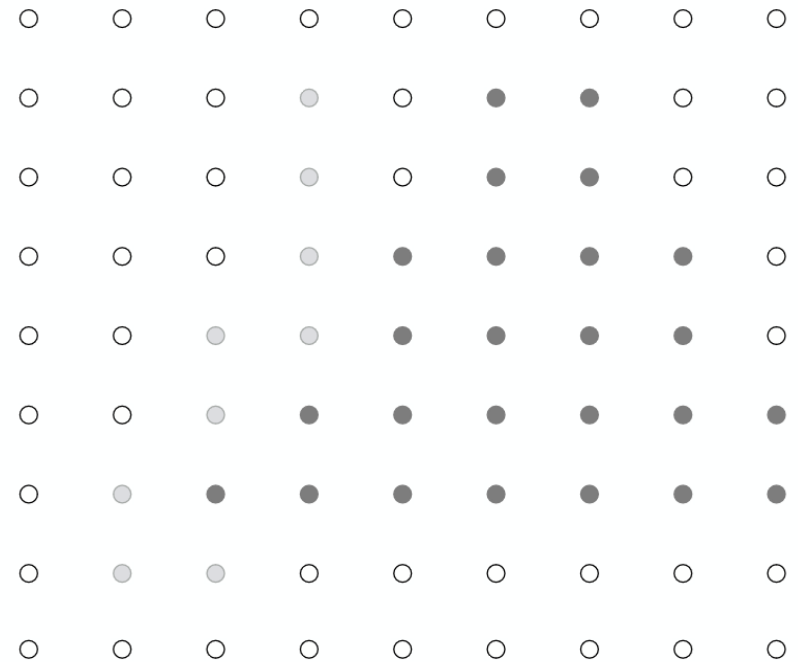
Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

Processing red triangle:

$depth = 0.25$
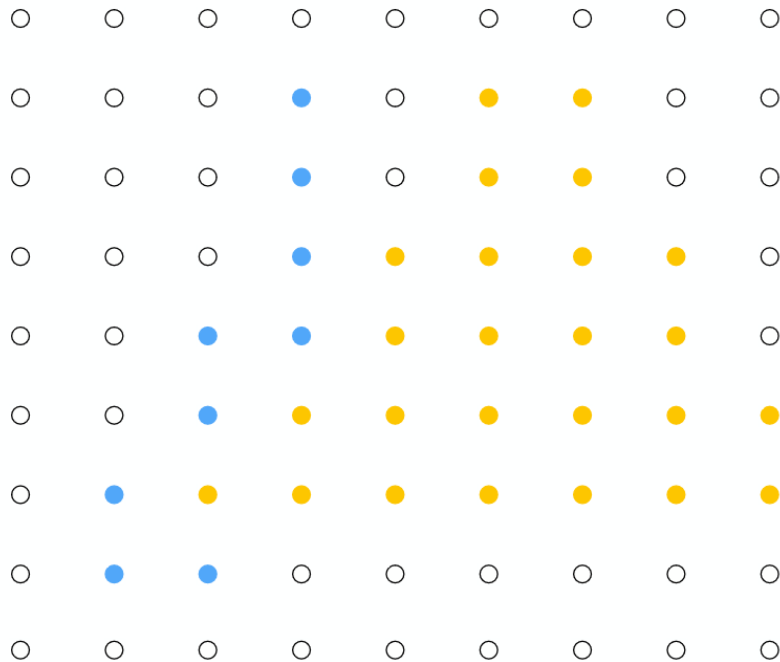
Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

Red = sample passed depth test

Color buffer contents
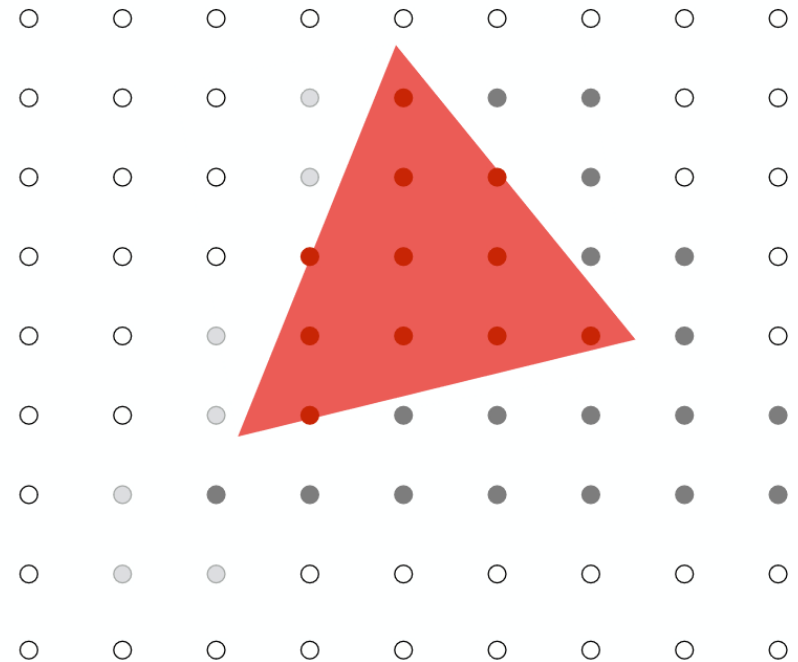
Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing red triangle:
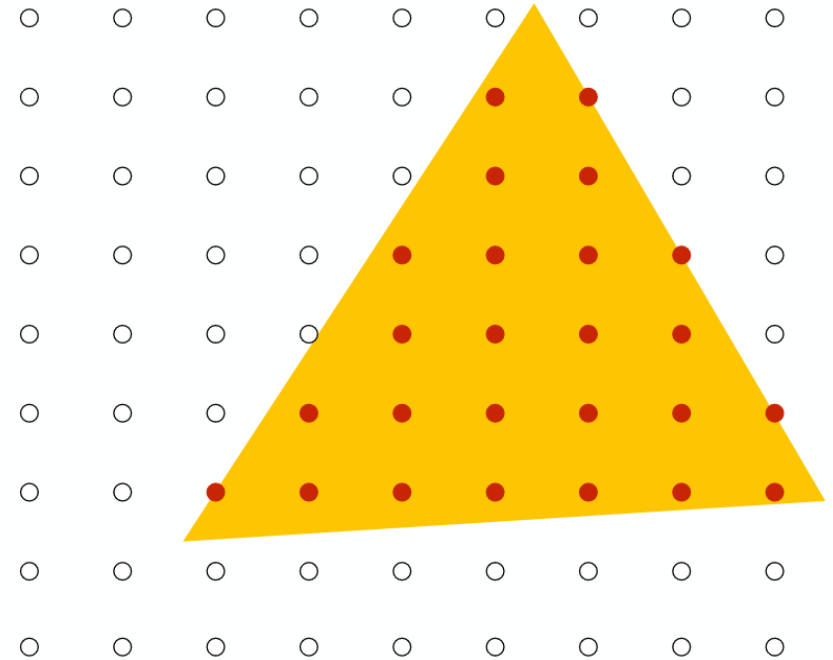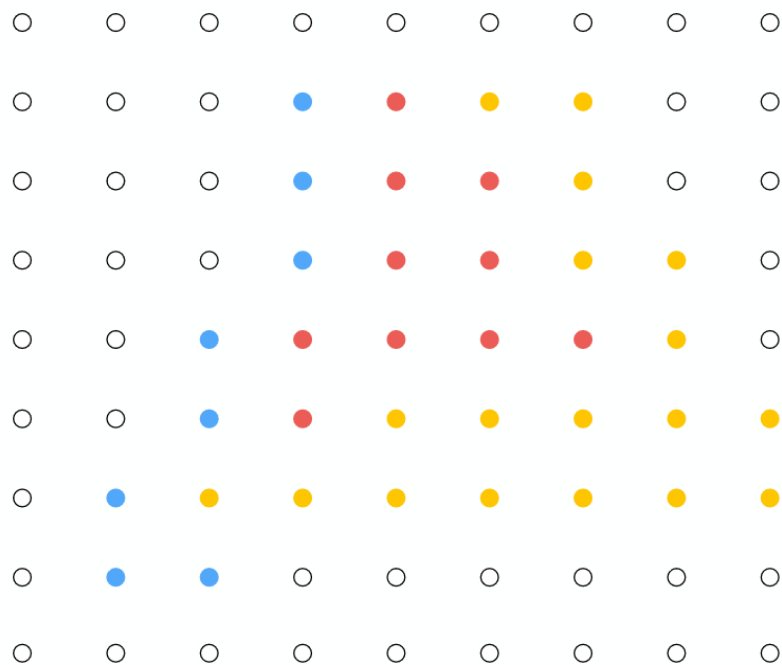
Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

Red = sample passed depth test
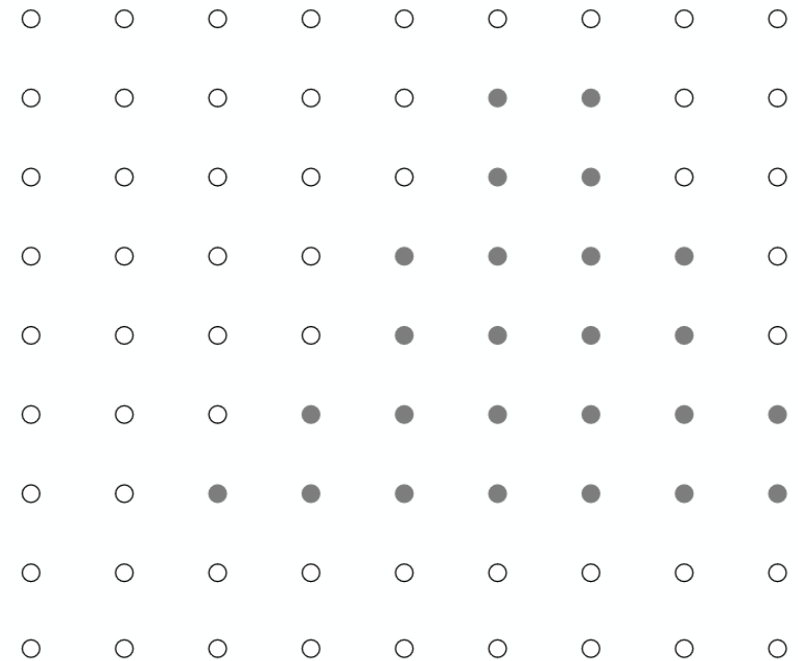
Color buffer contents
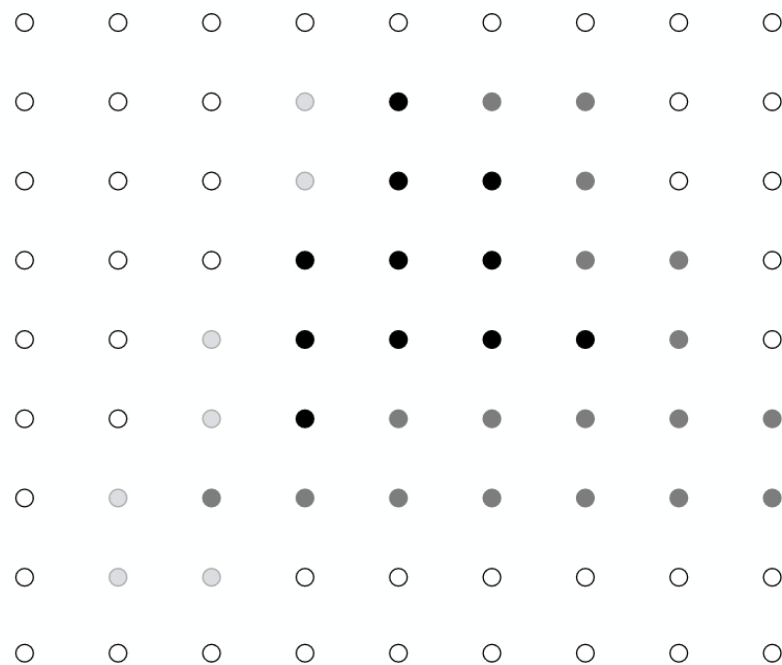
Depth buffer contents

Does depth-buffer algorithm handle interpenetrating surfaces?

Of course!

Occlusion test is based on depth of triangles at a given sample point. The relative depth of triangles may be different at different sample points.

Green triangle in front of yellow triangle

Yellow triangle in front of green triangle
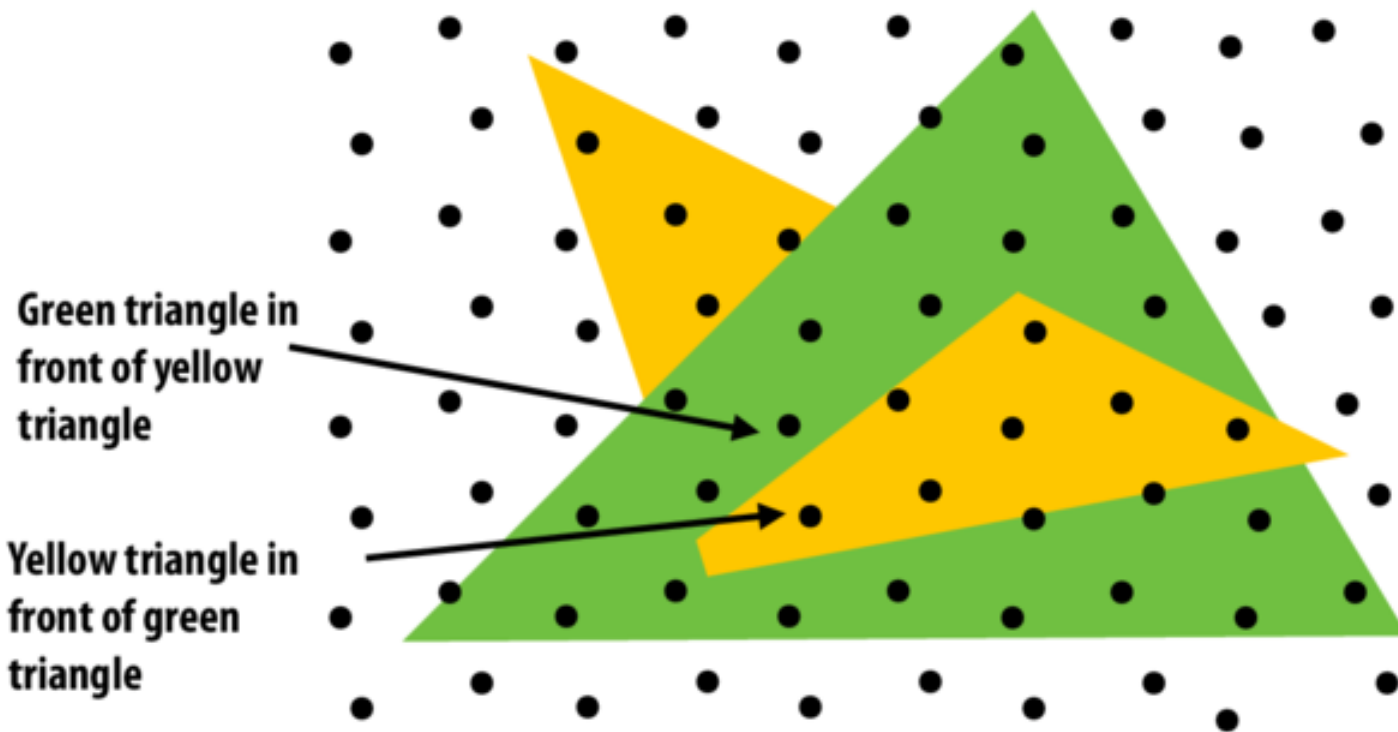
Does depth-buffer algorithm handle interpenetrating surfaces?

Of course!

Occlusion test is based on depth of triangles at a given sample point. The relative depth of triangles may be different at different sample points.
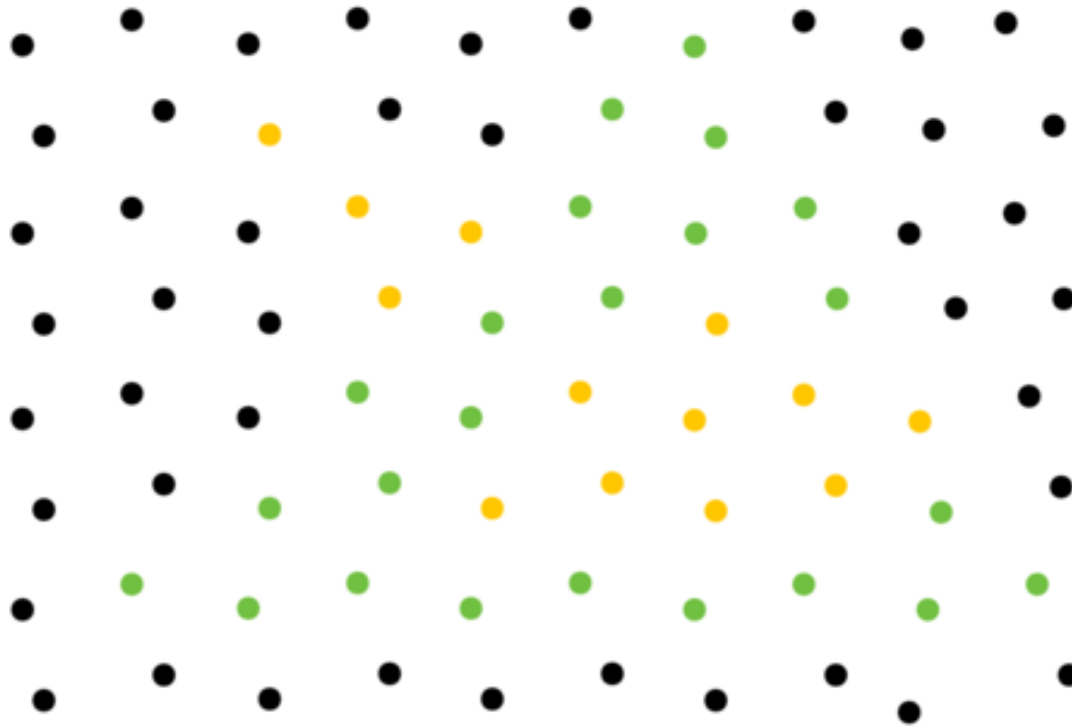
Z-Buffering : **Summary**

- Current standard algorithm that is implemented on all graphics hardwares

- Advantages / Disadvantages:
    - Easy to implement
    - Fast with hardware support → Fast depth buffer memory
    - Polygons can be drawn in any order
    - Extra memory required for z-buffer
        - not a problem anymore

Rendering (Graphics) Pipeline Again

| **vertex processing** | → | **rasterization** | → | **fragment processing** | → | output merging |

: performs a sequence of vertex transformations

: assembles polygons converts each polygon into a set of fragments (pixels)

: determines color of each fragment with light & texture

OpenGL/Direct3D graphics pipeline *

Structures rendering computation as a series of operations on vertices, primitives, fragments, and screen samples

Input: vertices in 3D space

Operations on vertices

Vertex Processing

Vertex stream

Vertices in positioned in normalized coordinate space

Operations on primitives (triangles, lines, etc.)

Primitive Processing

Primitive stream

Triangles positioned on screen

Operations on fragments

Fragment Generation (Rasterization)

Fragment stream

Fragments (one fragment per covered sample)

Fragment Processing

Shaded fragment stream

Shaded fragments

Operations on screen samples

Screen sample operations (depth and color)

Output: image (pixels)

* Several stages of the modern OpenGL pipeline are omitted

OpenGL/Direct3D graphics pipeline *

Structures rendering computation as a series of operations on vertices, primitives, fragments, and screen samples

Input: vertices in 3D space

Operations on vertices

Vertex Processing

Vertex stream

Operations on primitives (triangles, lines, etc.)

Primitive Processing

Primitive stream

Vertices in positioned in normalized coordinate space

Clipping & Back-face culling

Fragment Generation (Rasterization)

Triangles positioned on screen

Operations on fragments

Fragment stream

Fragment Processing

Fragments (one fragment per covered sample)

Shaded fragment stream

Shaded fragments

Operations on screen samples

Screen sample operations (depth and color)

Depth test

Output: image (pixels)

* Several stages of the modern OpenGL pipeline are omitted

OpenGL/Direct3D graphics pipeline *

°3
°1
°4   **Input vertices in 3D space**
°2

Operations on vertices

Vertex Processing ← **transform matrices**

Vertex stream

Operations on primitives (triangles, lines, etc.)

Primitive Processing

Primitive stream

Clipping & Back-face culling

Fragment Generation (Rasterization)

Operations on fragments

Fragment stream

textures   **, lighting model & condition**

Fragment Processing

Shaded fragment stream

Operations on screen samples

Screen sample operations (depth and color)

Depth test

Pipeline inputs:

- Input vertex data
- Parameters needed to compute position on vertices in normalized coordinates (e.g., transform matrices)
- Parameters needed to compute color of fragments (e.g., textures)
- "Shader" programs that define behavior of vertex and fragment stages

* several stages of the modern OpenGL pipeline are omitted

OpenGL/Direct3D graphics pipeline *

°3
°1
°4   **Input vertices in 3D space**
°2

Operations on vertices

Vertex Processing ← **transform matrices**

Vertex stream

Operations on primitives (triangles, lines, etc.)

Primitive Processing

Primitive stream

Clipping & Back-face culling

Fragment Generation (Rasterization)

Fragment stream

Operations on fragments

Fragment Processing

Shaded fragment stream

textures , lighting model & condition

- Gray steps are automatically done by modern graphics system.

- Yellow steps are done by what you provide.

- That's why we've been focusing on these yellow steps in this course!

- You can even write your own software renderer that covers whole process!

Operations on screen samples

Screen sample operations (depth and color)

Depth test

* several stages of the modern OpenGL pipeline are omitted

Acknowledgement

- The scope of final exam is upto here.

Course Wrap-up

Do you remember?

- Computer graphics: The study of creating, manipulating, and using visual images in the computer.

Modelling   Simulation & Rendering

Image

Animation (a series of images)
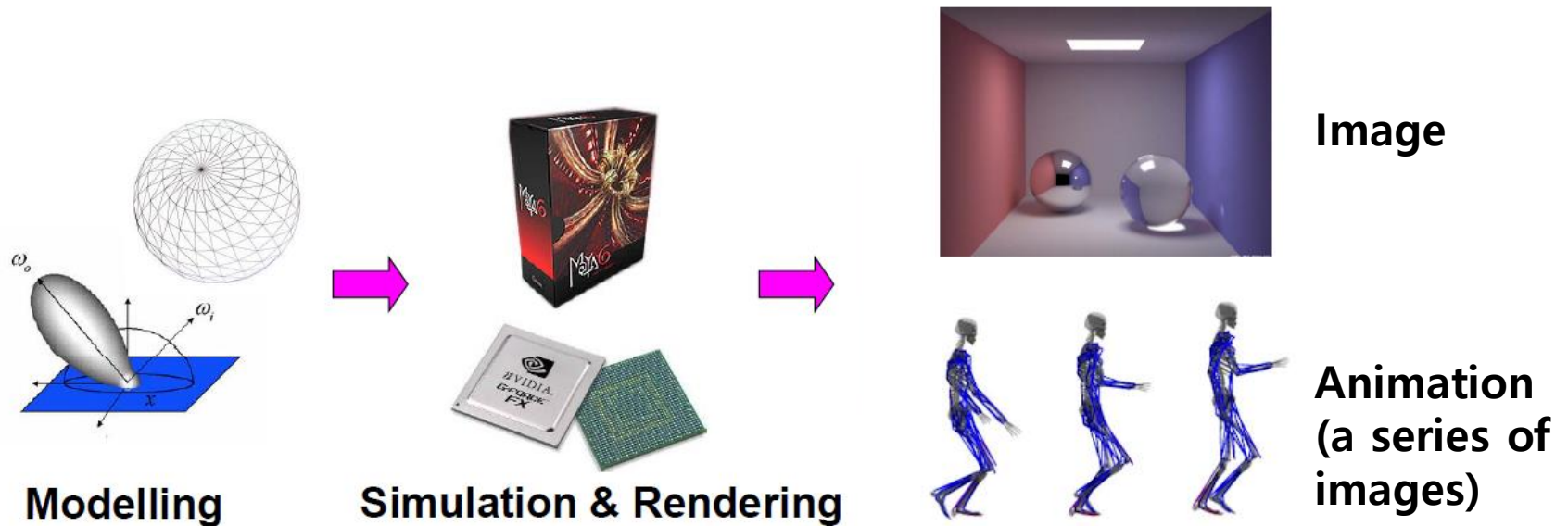
Computer vision inverts the process
Image processing deals with images

Questions about Computer Graphics

- To do this, we should be able to answer:

- How to express movement, placement, shape, and appearance of objects

- How to map 3D objects into 2D screen

- How the whole rendering process is performed

Movement & placement	3 - Transformation 1 4 - Transformation 2 5 - Affine Transformation Matrix, Rendering Pipeline, Viewing 8 - Lighting & Shading 2, Hierarchical Modeling 9 - Orientation & Rotation 10 - Kinematics & Animation 11 - Curves
Mapping to 2D screen	5 - Affine Transformation Matrix, Rendering Pipeline, Viewing 6 - Projection, Mesh 1
Shape	6 - Projection, Mesh 1 7 - Mesh 2, Lighting & Shading 1 11 - Curves
Appearance	7 - Mesh 2, Lighting & Shading 1 8 - Lighting & Shading 2, Hierarchical Modeling 12 - More Lighting, Texture
Rendering Pipeline	5 - Affine Transformation Matrix, Rendering Pipeline, Viewing 13 - Rasterization & Visibility

How do you feel?

- If you've **had more fun** in this course than other courses, you already have **the potential** to do interesting research in computer graphics!

- I think, **passion** is the most important thing in computer graphics.
  - That was the starting point for me on this path.
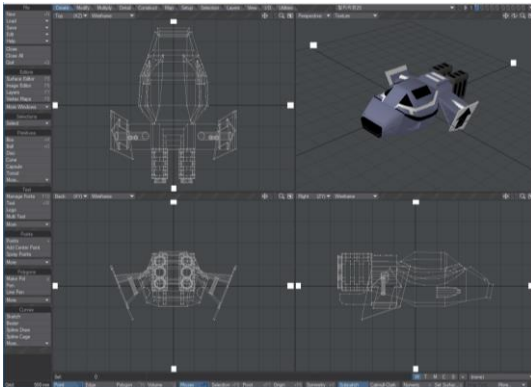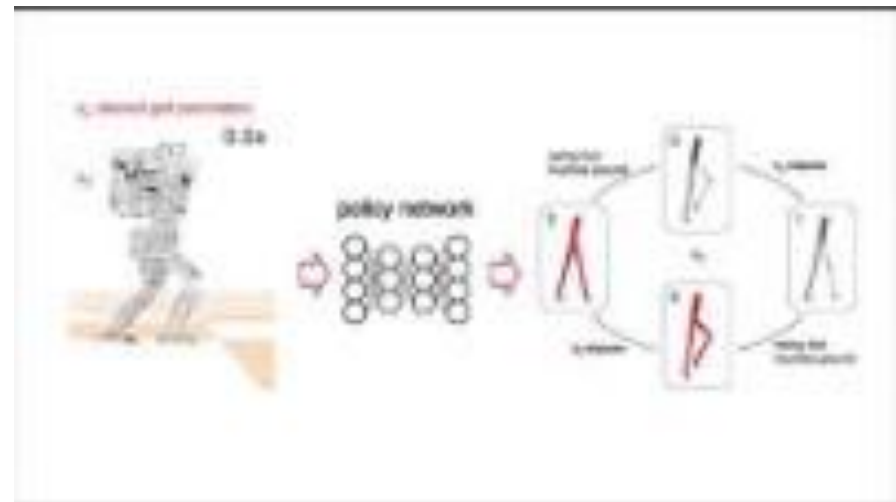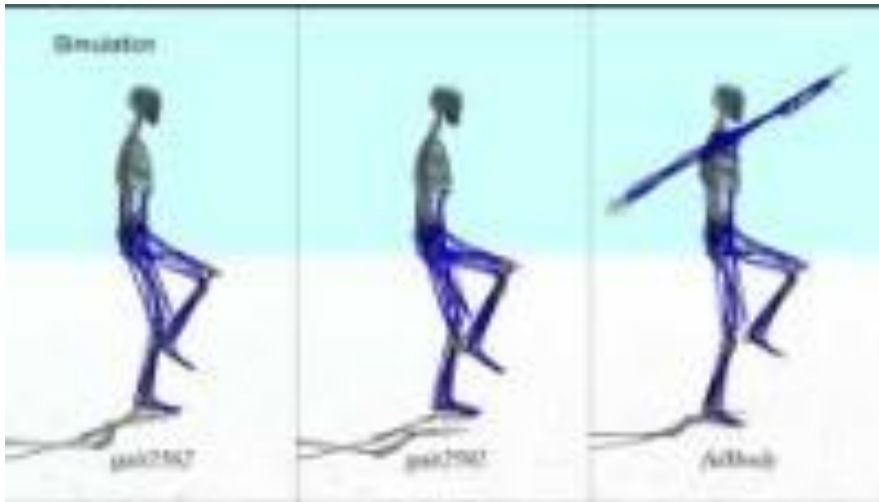
If you think "that's me!" and ...

- "I want to do a project related to character animation":
  - (When you are in 4th grade) Let's do a **graduation project** together.

- "I want to study more character animation and program a lot!":
  - I'd like to recommend to take "COMPUTER SCIENCE Capstone PBL (Character Motion Synthesis and Character Control)" (4th grade course)

- "I want to do something before I even enter 4th grade!":
  - Please email me: yoonsanglee@hanyang.ac.kr

Characteristics of Computer Graphics Research

- Requires a lot of **programming**.
  - If you like programming and have confidence, you are likely to do well.

- It is **interesting** because all the research results come out in a **visible** form.
  - In the field of computer graphics, a **video** is always attached when submitting a paper.

Computer Graphics and Robotics Lab.

- https://cgrhyu.github.io/

Thanks for being a great class!