Computer Graphics

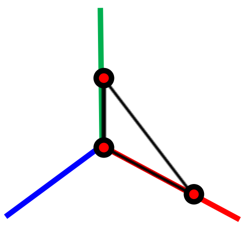
13 - Rasterization & Visibility

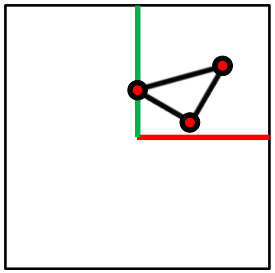
Yoonsang Lee
Spring 2019

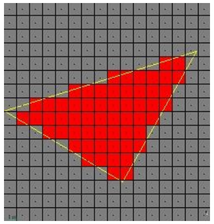
Topics Covered

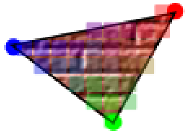
- Two Approaches for Rendering
  - Object-oriented (Rasterization)
  - Image-oriented (Raytracing)
- Rasterization (in a narrow sense)
  - Line / Polygon Drawing
- Visibility Problem
  - Clipping (Viewing frustum culling)
  - Back-face culling
  - Hidden surface removal
- Rendering(Graphics) Pipeline Again

Recall: Rendering(Graphics) Pipeline









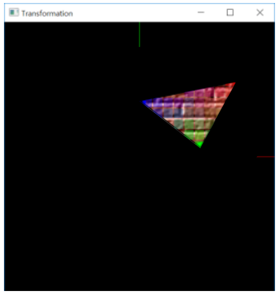
vertex processing → **rasterization** → **fragment processing** → output merging

vertex processing: performs a sequence of vertex transformations

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

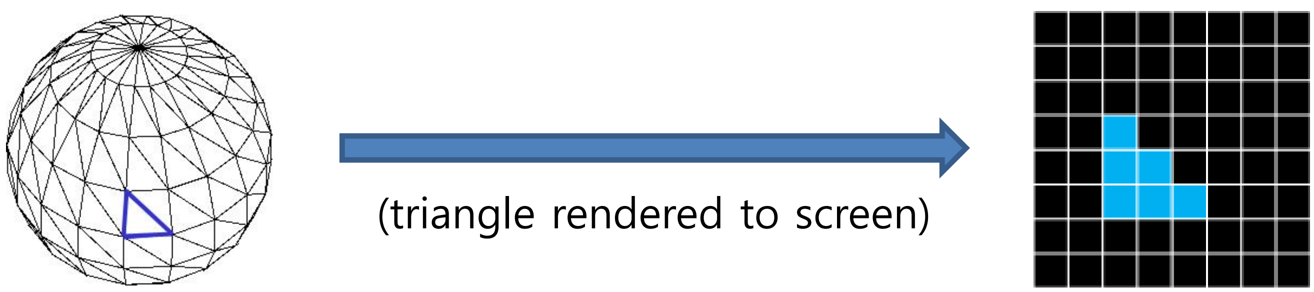
fragment processing: determines color of each fragment with light & texture

output merging



Two Approaches for Rendering - 1

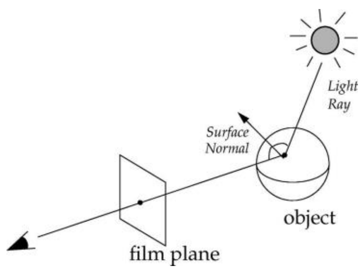
```
for each object in scene
    transform the object to viewport   # vertex processing
    find pixels for the object   # rasterization (in a narrow sense)
    draw these pixels based on texture and lighting model
                                                          # fragment
                                                           processing
```



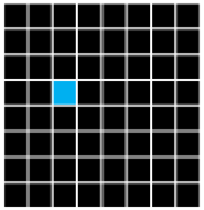
- Called **object-oriented rendering** or **rendering(graphics) pipeline** or just **rasterization**(in a broad sense)

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 rendered to screen)



- Called **image-oriented rendering** or **ray tracing**

- We'll skip ray tracing part, see *14-reference-RayTracing.pdf* for more information about it.

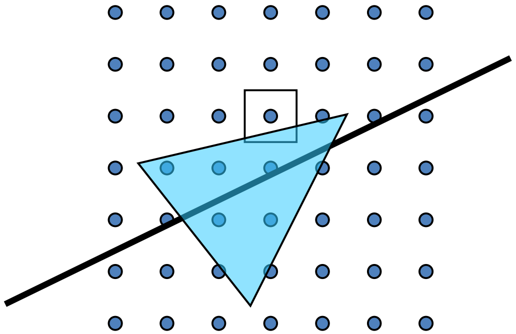
Rasterization(in a broad sense) & Ray Tracing in this Course

- Most topics we’ve covered are *fundamental concepts* of computer graphics, regardless of two rendering methods.
  - Transformations, mesh, lighting, shading, texture, rotation, curves, ...

- 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)
  - Xiaolin Wu's

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

Rasterization(in a narrow sense)

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

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

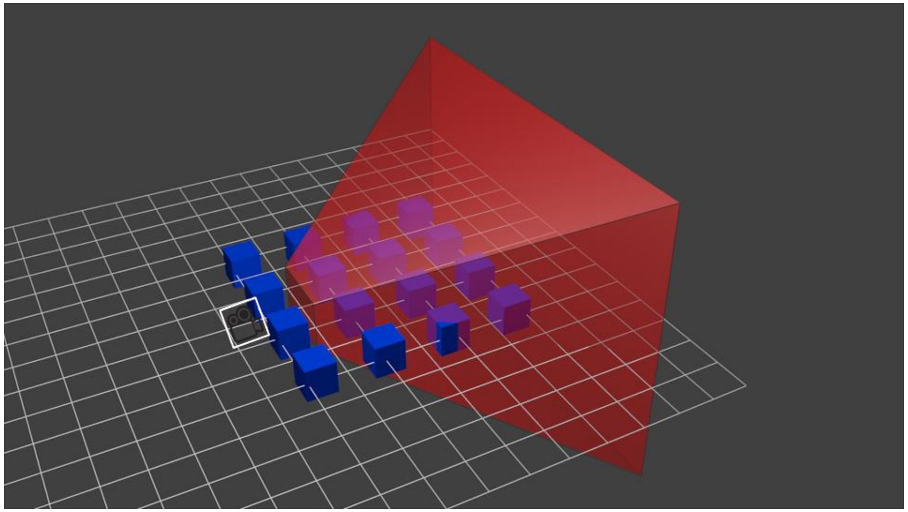
- But graphics hardware take care of them!
  - These algorithms were intensively studied in early days of computer graphics, so quite mature now.
  - Now basic algorithms are implemented in graphics hardware (GPU).

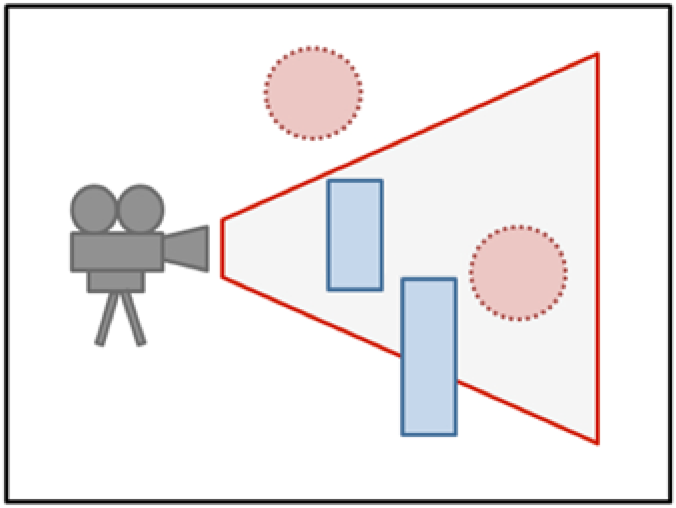
- So nowadays you can think lines and polygons as "primitives" that are basically rendered.

Visibility Problem

- What is VISIBLE?

Red: viewing frustum, Blue: objects

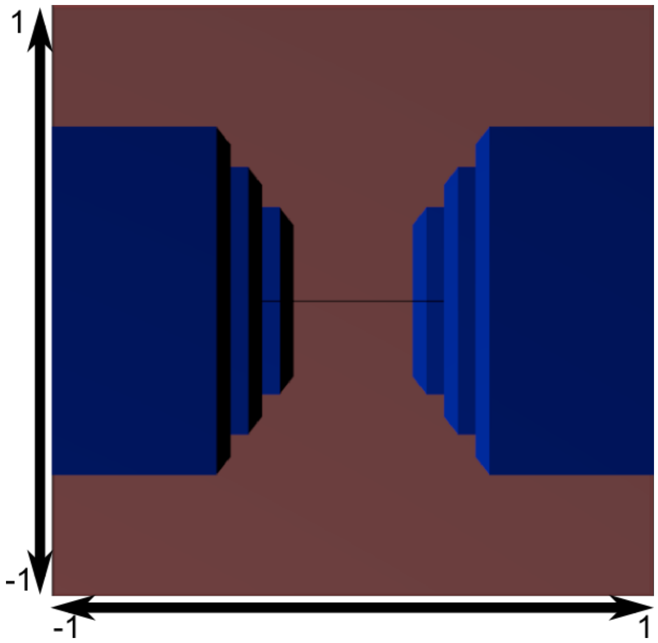


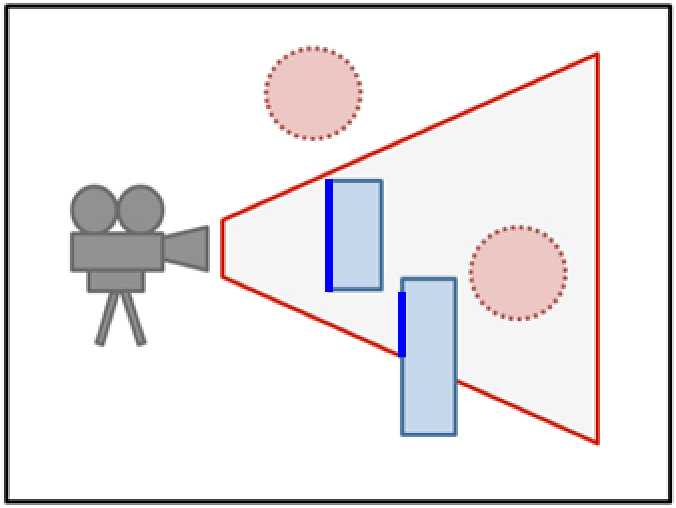


Visibility Problem

- The answer is:

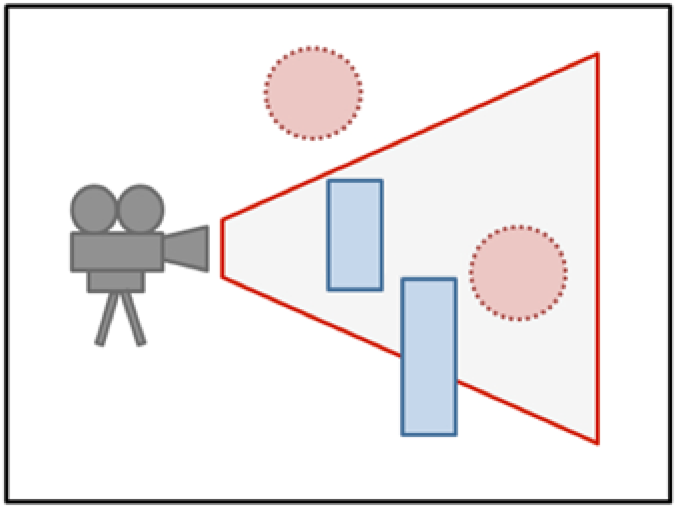
The camera view





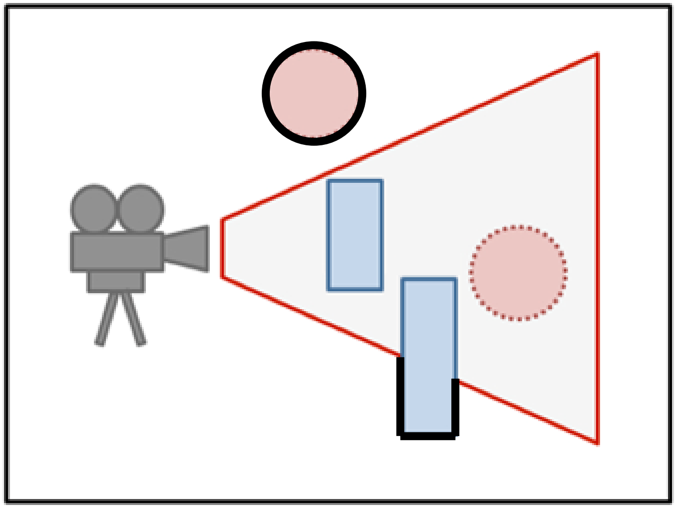
Visibility Problem

- What is NOT VISIBLE?



Visibility Problem

- What is NOT VISIBLE?
- **Primitives outside of the viewing frustum**

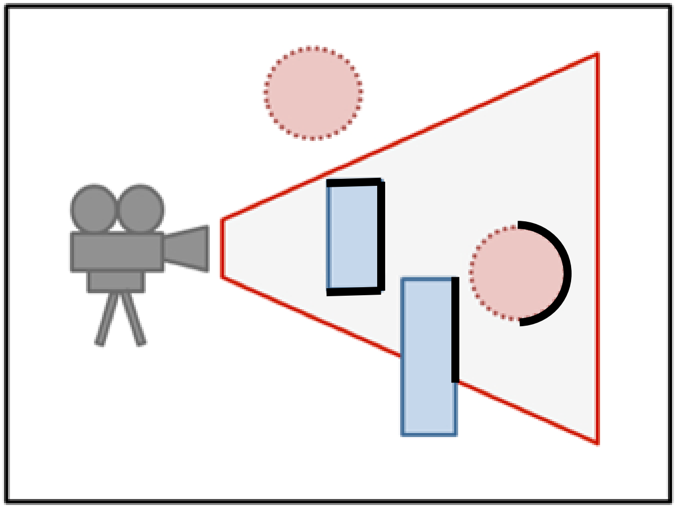


Visibility Problem

- What is NOT VISIBLE?

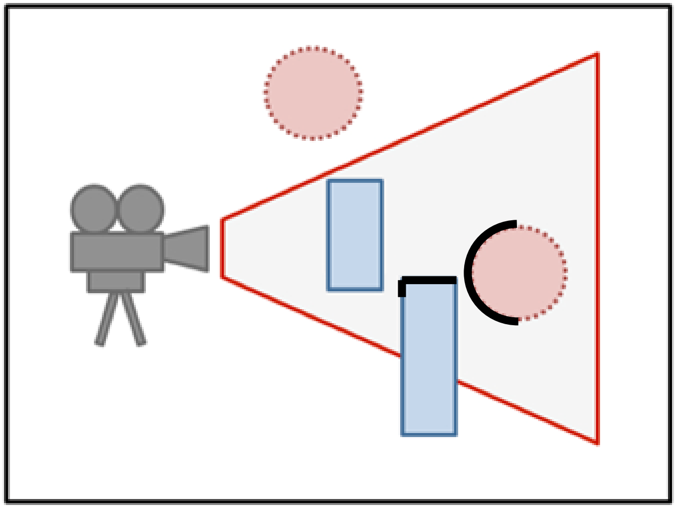
- Primitives outside of the viewing frustum

- **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**



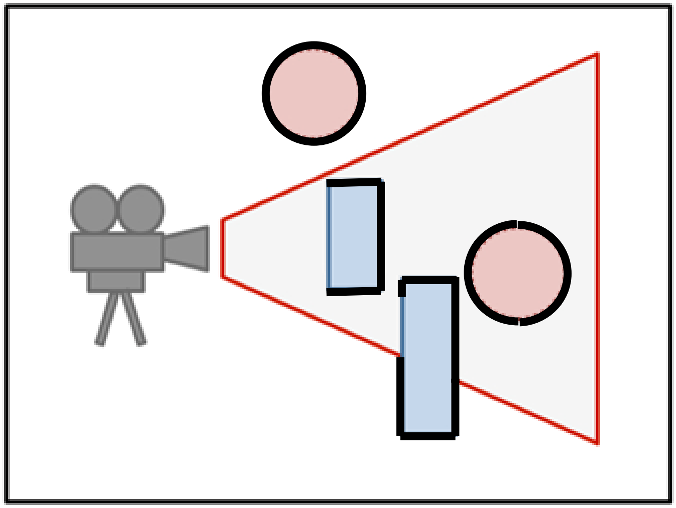
Visibility Problem

- 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...

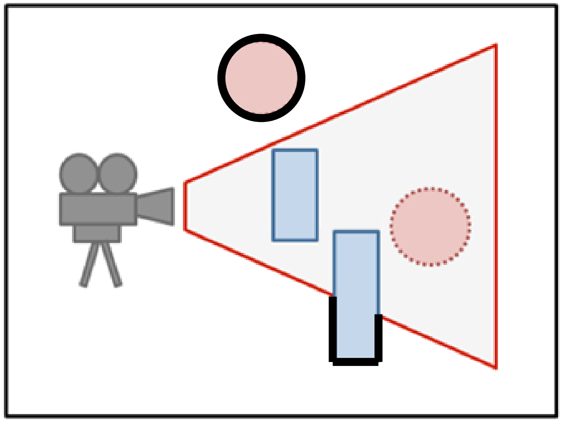
- Primitives outside of the viewing frustum
- → **Clipping (Viewing frustum culling)**

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

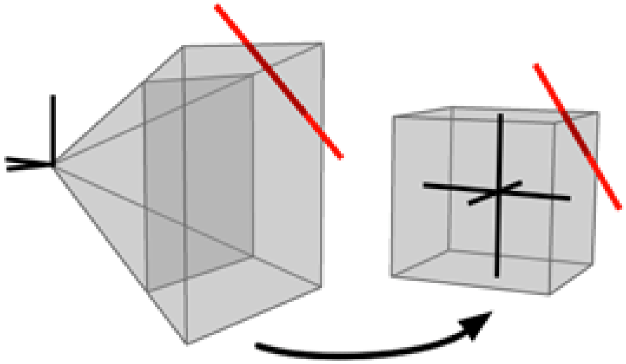
- Primitives occluded by other objects closer to the camera
- → **Hidden surface removal**

Clipping (Viewing Frustum Culling)

- Removing primitives outside of the viewing frustum

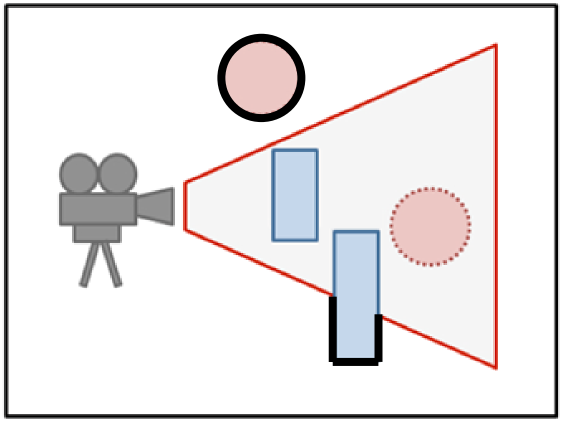


- Clipping is much easier with canonical view volume.
  - actually done in *clip space*



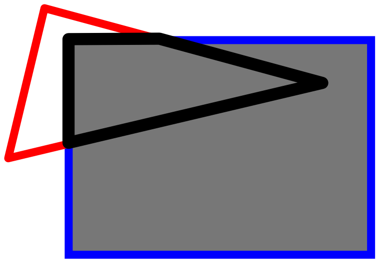
Clipping (Viewing Frustum Culling)

- Line clipping algorithms
  - Cohen–Sutherland
  - Liang–Barsky
  - Cyrus–Beck



- Polygon clipping algorithms
  - Sutherland–Hodgman
  - Weiler–Atherton

Clipping (Viewing Frustum Culling)



triangle → quad

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

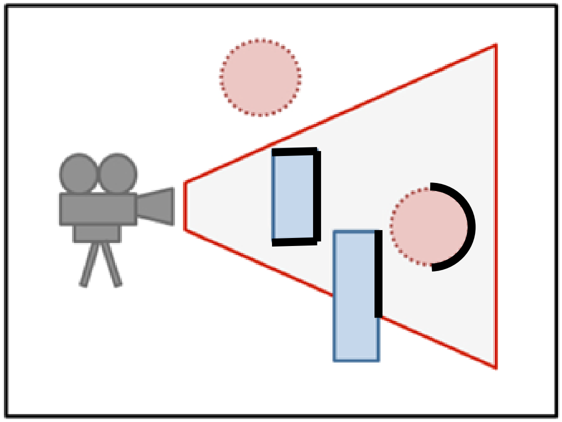
- Again, let’s just skip details of these algorithms.

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

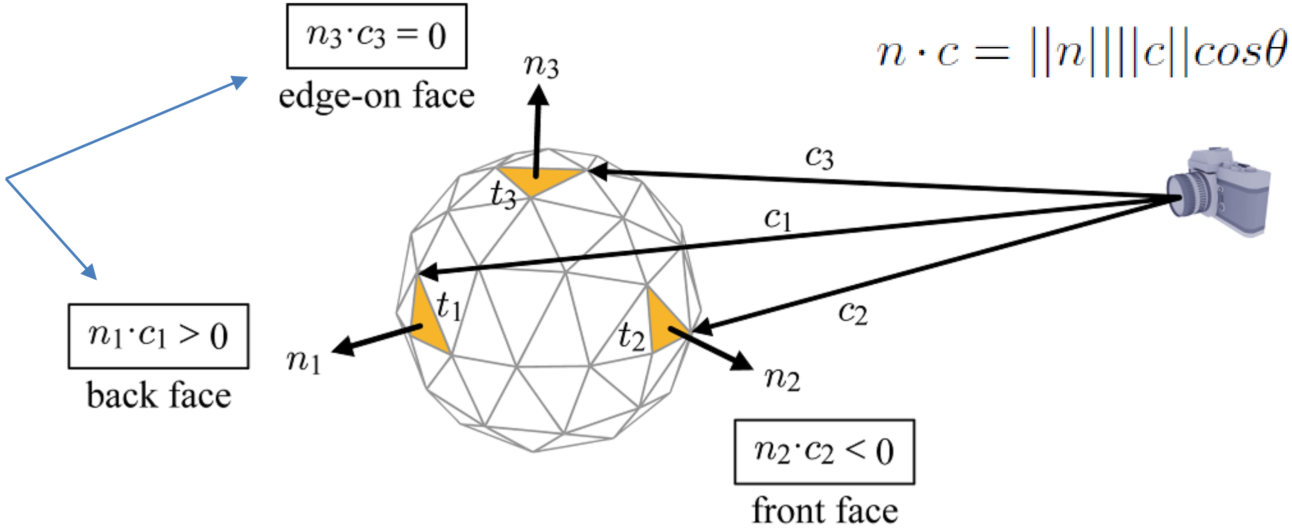
- *13-reference-rasterization,clipping.pdf* has brief slides about DDA & Cohen-Sutherland algorithms. If you're interested, please refer it.

Back-Face Culling

- Removing back-facing primitives
- Determined by the dot product of normal and view (camera) vectors.

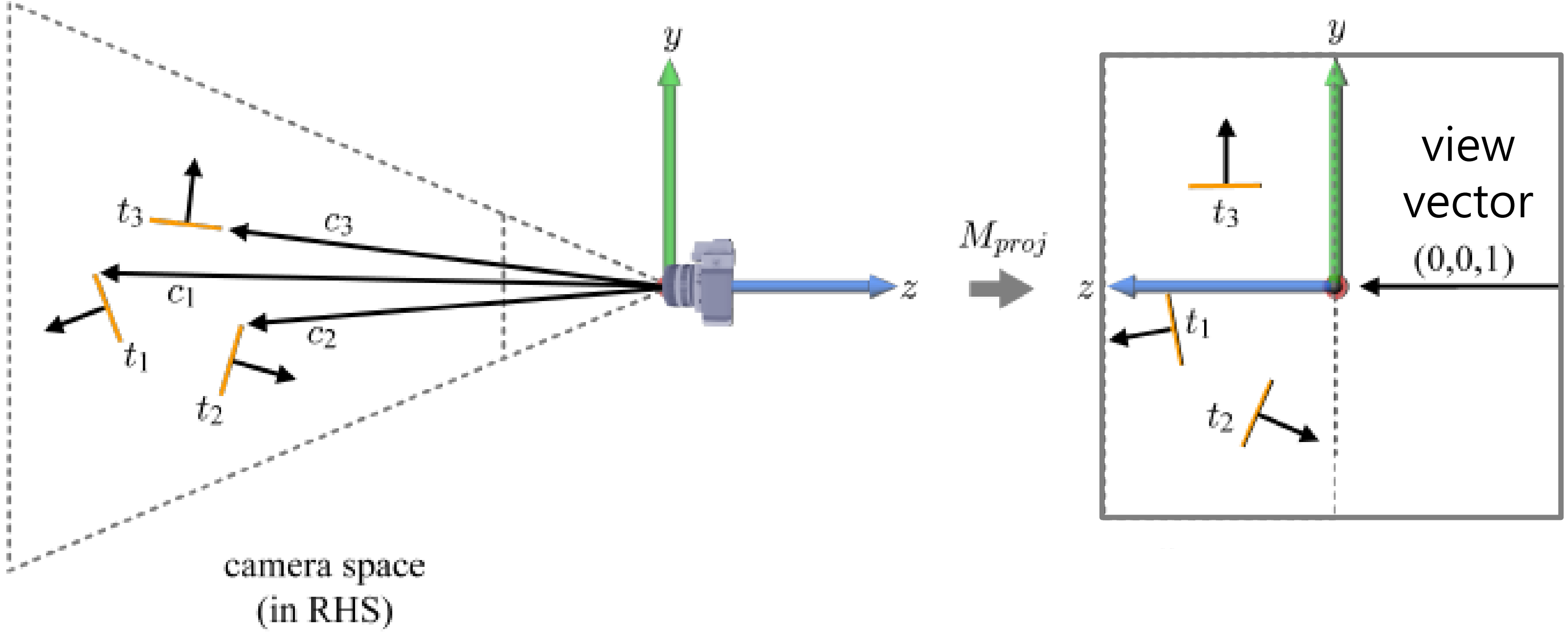


Discard!

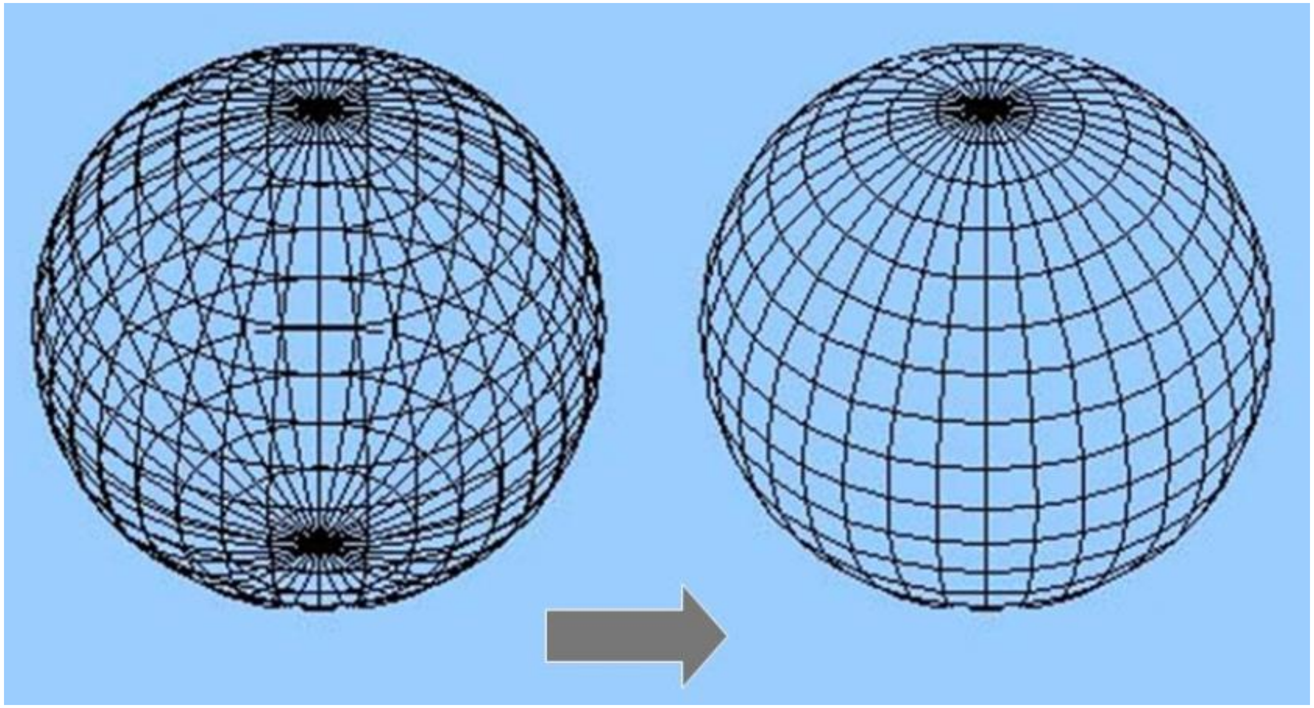


Back-Face Culling

- Back-face culling is much more efficient with canonical view volume
  - Because in canonical view volume, we can use a single view vector, (0,0,1).



Back-Face Culling



Back-Face Culling in OpenGL

- Can cull front faces or back faces
- Back-face culling can sometimes double performance

(initial value: GL_CCW)

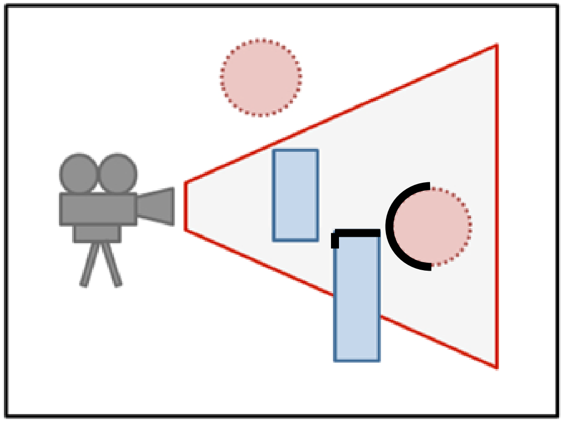
```
if (cull):
    glFrontFace(GL_CCW)          # define winding order
    glEnable(GL_CULL_FACE)       # enable Culling
    glCullFace(GL_BACK)          # which faces to cull
else:
    glDisable(GL_CULL_FACE)
```

(initially disabled)

You can also do front-face culling!

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
  - ...

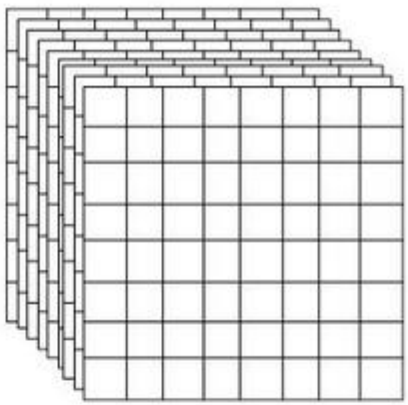
- Z-buffer is the standard method.

- 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.

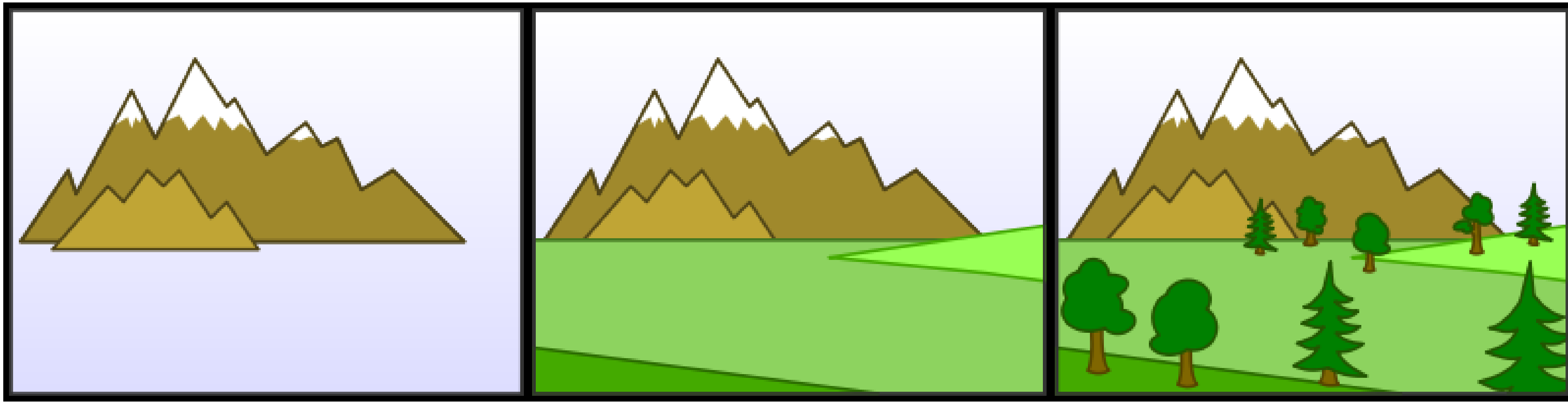
- A frame buffer is characterized by its width, height, and depth.
  - E.g. The frame buffer size for 4K UHD resolution with 32bit color depth = 3840 x 2160 x 32 bits



- 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.

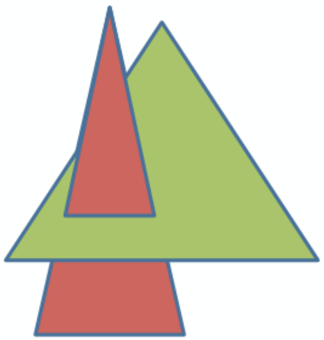
Painter's algorithm

- Simplest way to do hidden surfaces
- Draw from back to front, use overwriting in framebuffer
- Requires sorting all polygons by their depth

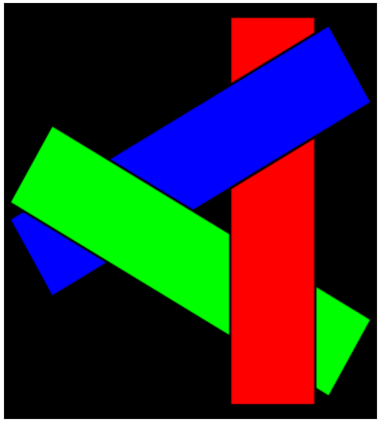


Weakness of Painter’s Algorithm

- 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

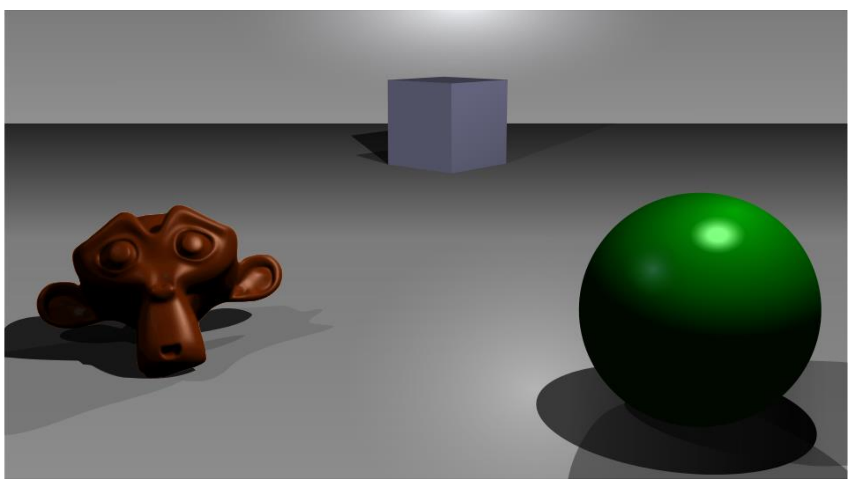
- In many (most) applications maintaining a *z* sort is too expensive
  - changes all the time when the view changes
  - many data structures exist, but complex
- Solution: draw in any order, keep track of closest
  - Z-buffer keeps track of closest depth so far
  - when drawing, compare object's depth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

Z-Buffering: Algorithm

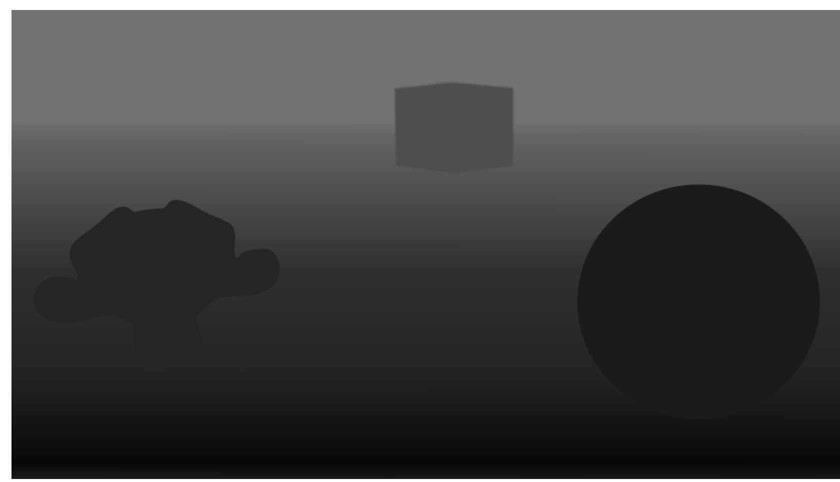
```
allocate depth_buffer;           // Allocate depth buffer → Same size as viewport.

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.
```

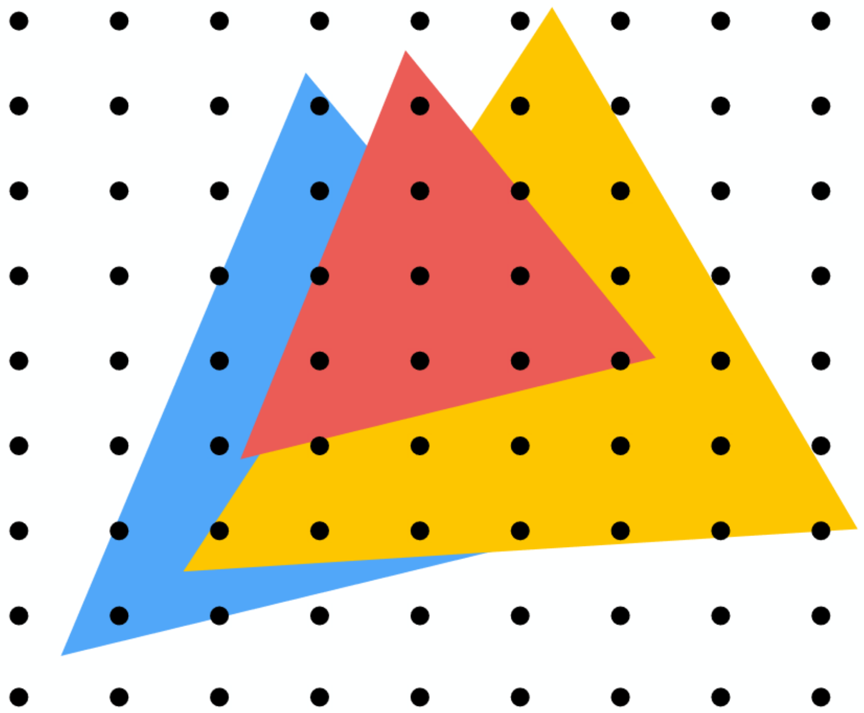


Frame buffer



Z-buffer (Depth buffer)

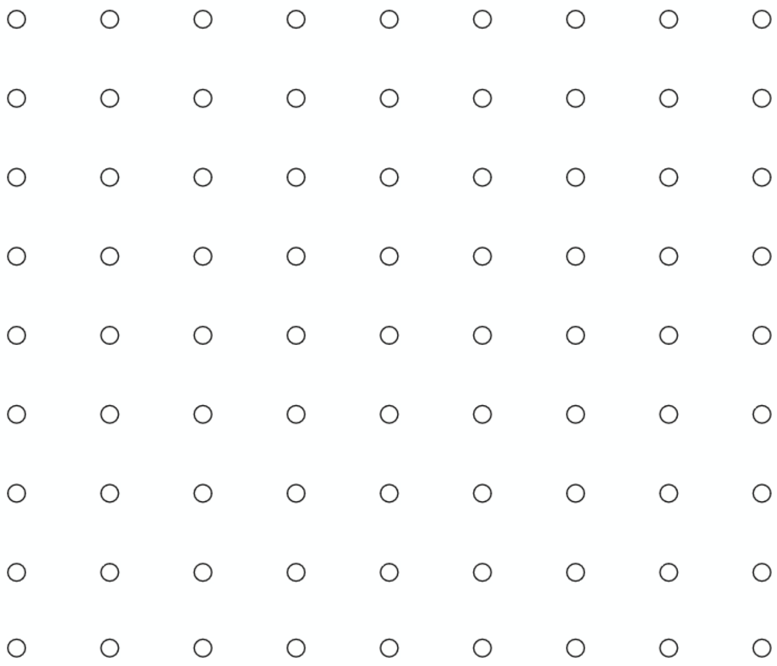
Example: rendering three opaque triangles

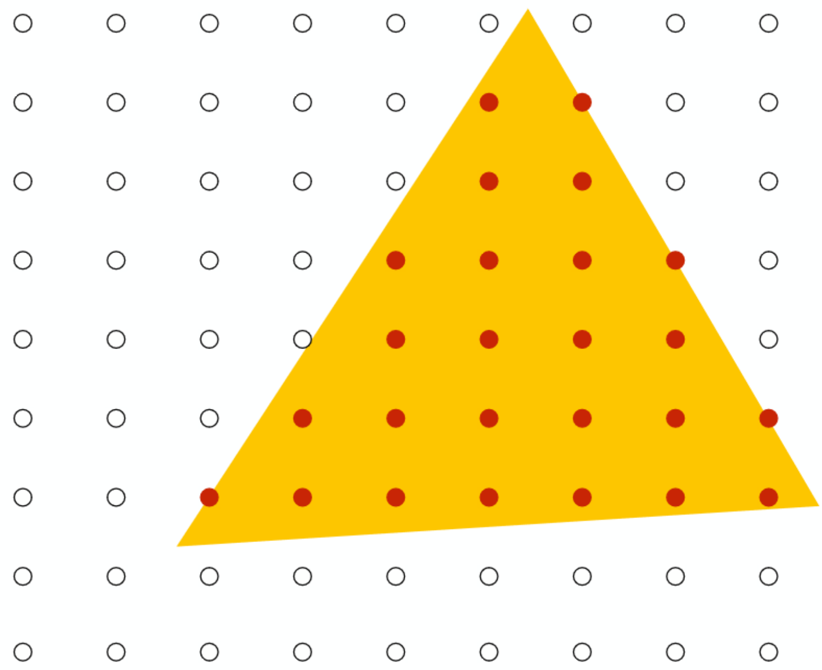


Occlusion using the depth-buffer (Z-buffer)

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)

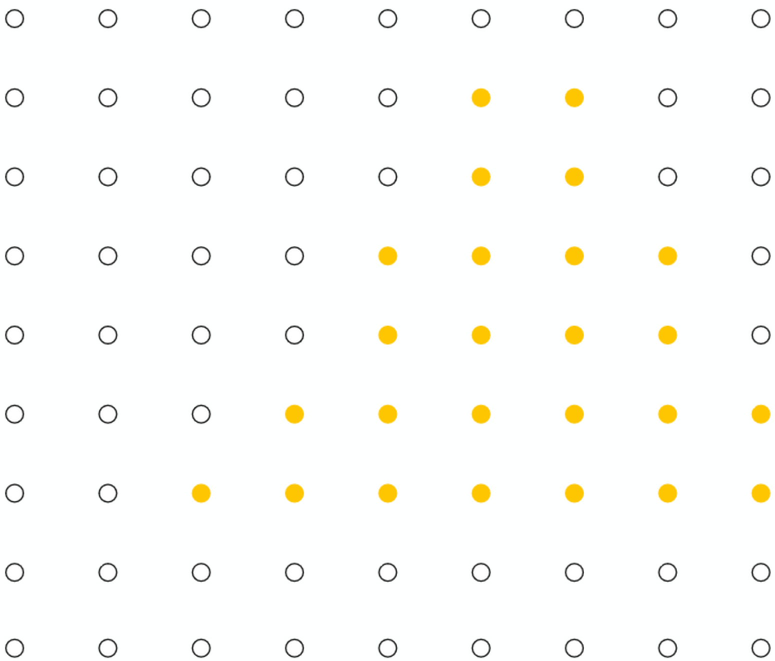
After processing yellow triangle:

Grayscale value of sample point used to indicate distance

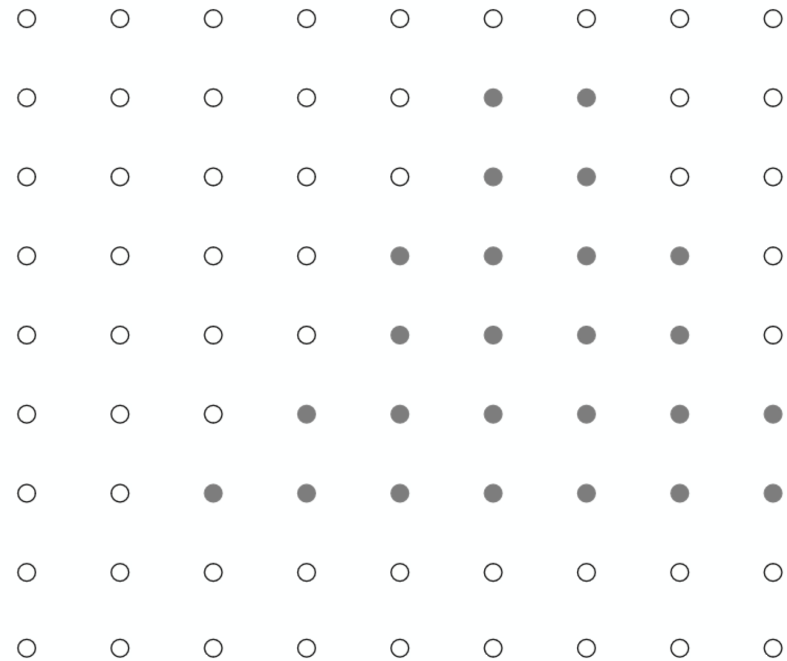
White = large distance

Black = small distance

Red = sample passed depth test



Color buffer contents

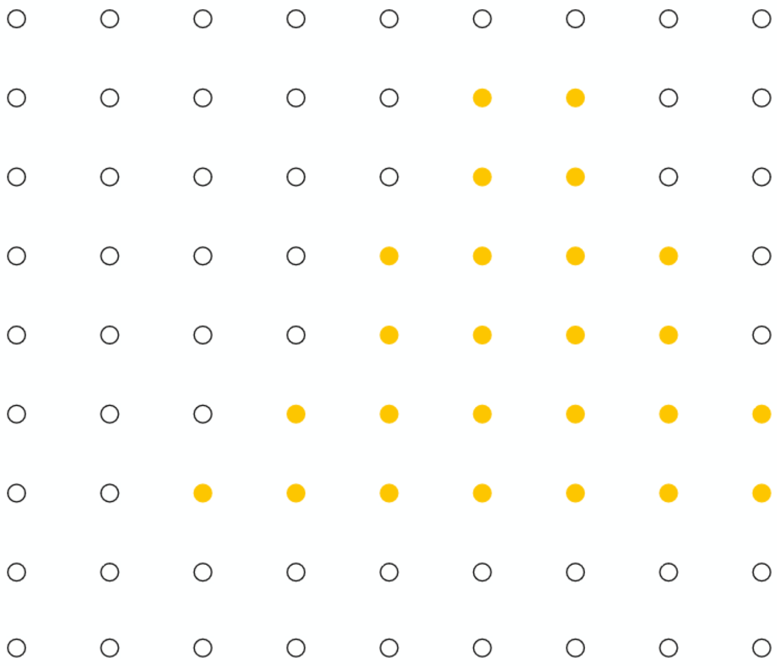


Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

Processing blue triangle:

depth = 0.75



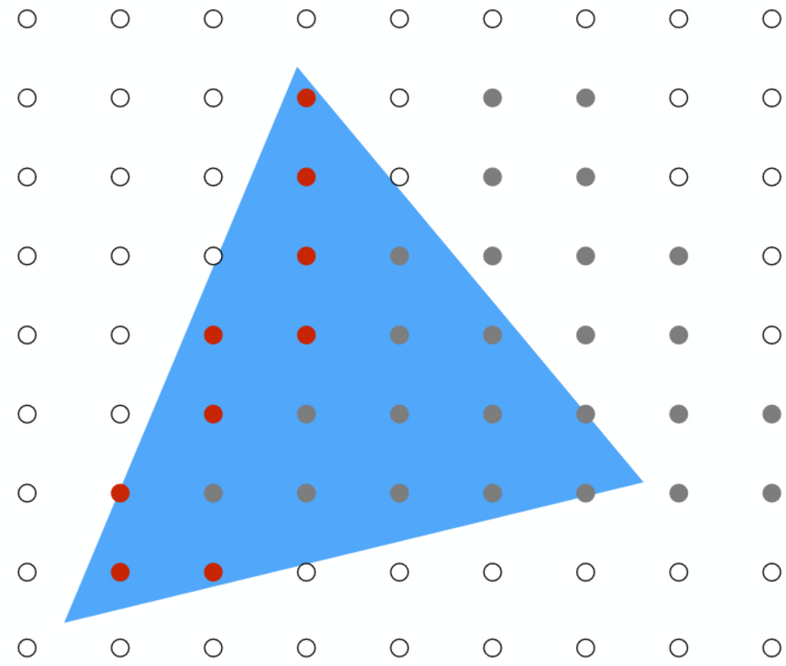
Color buffer contents

Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

Red = sample passed depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

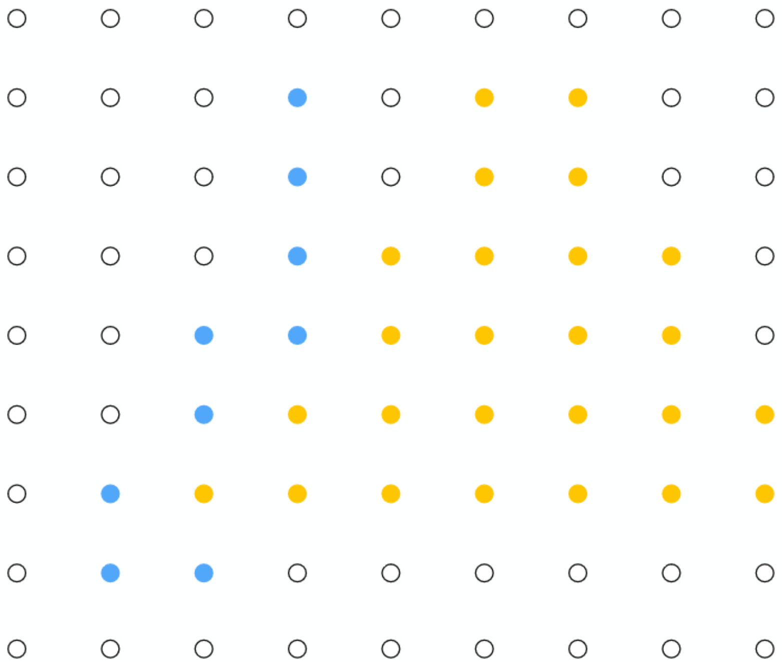
After processing blue triangle:

Grayscale value of sample point used to indicate distance

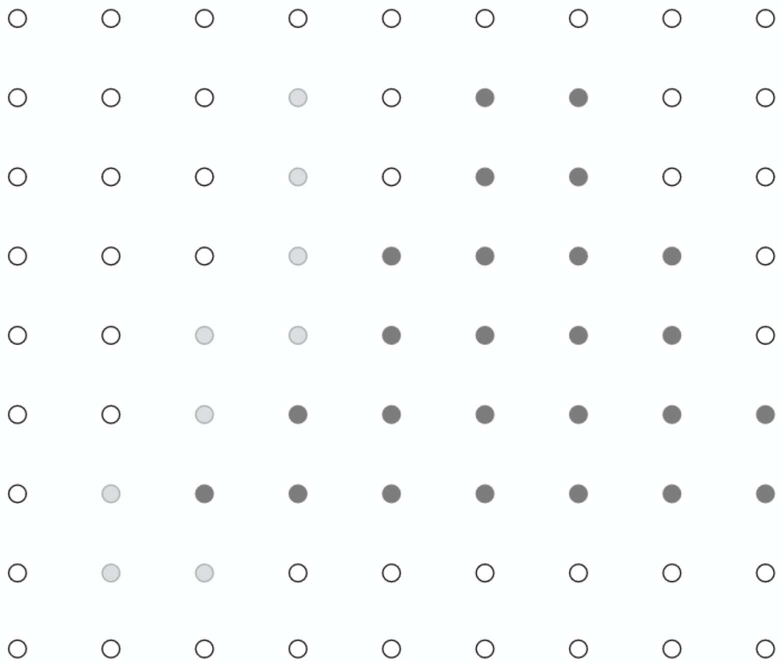
White = large distance

Black = small distance

Red = sample passed depth test



Color buffer contents



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

Processing red triangle:

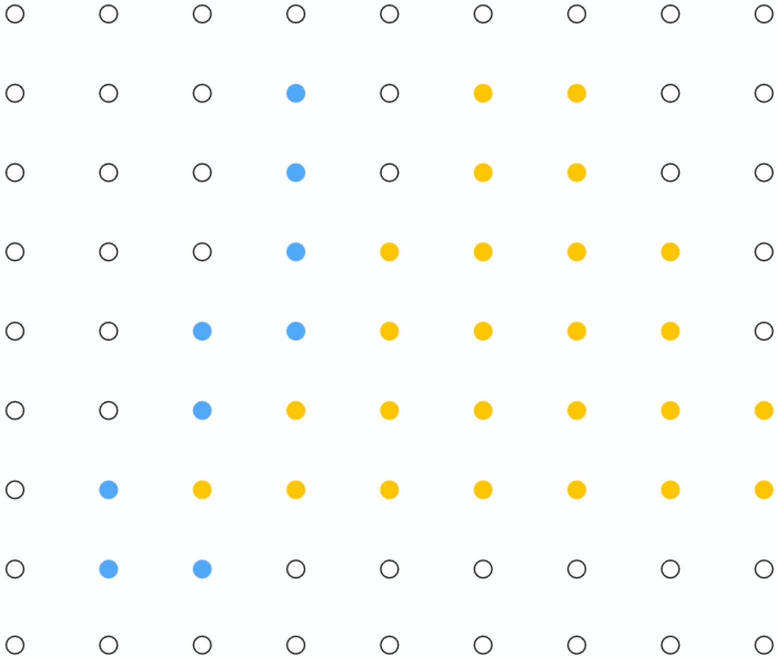
depth = 0.25

Grayscale value of sample point used to indicate distance

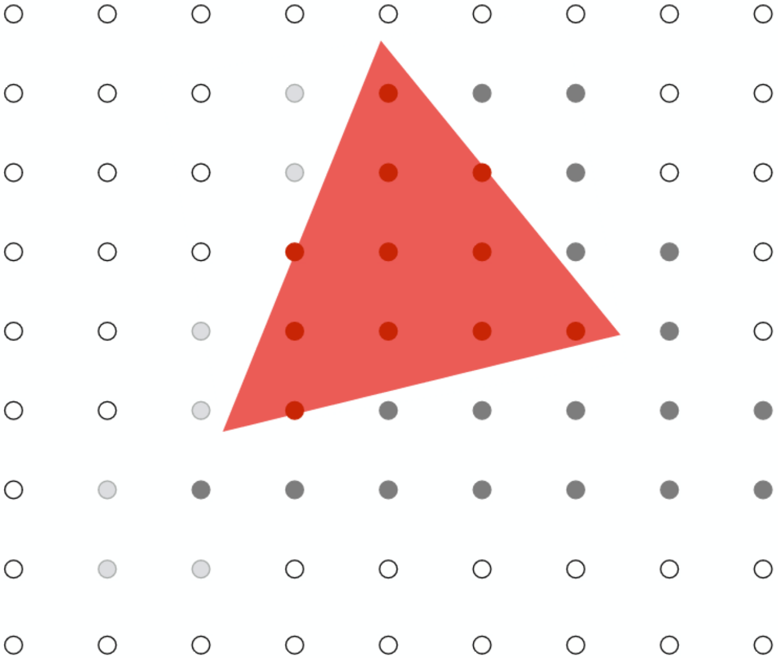
White = large distance

Black = small distance

Red = sample passed depth test



Color buffer contents



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

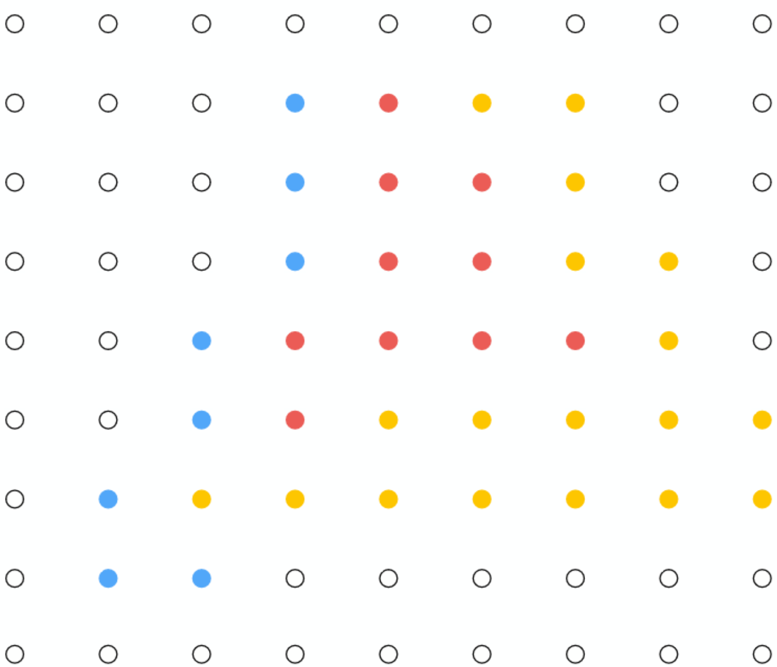
After processing red triangle:

Grayscale value of sample point used to indicate distance

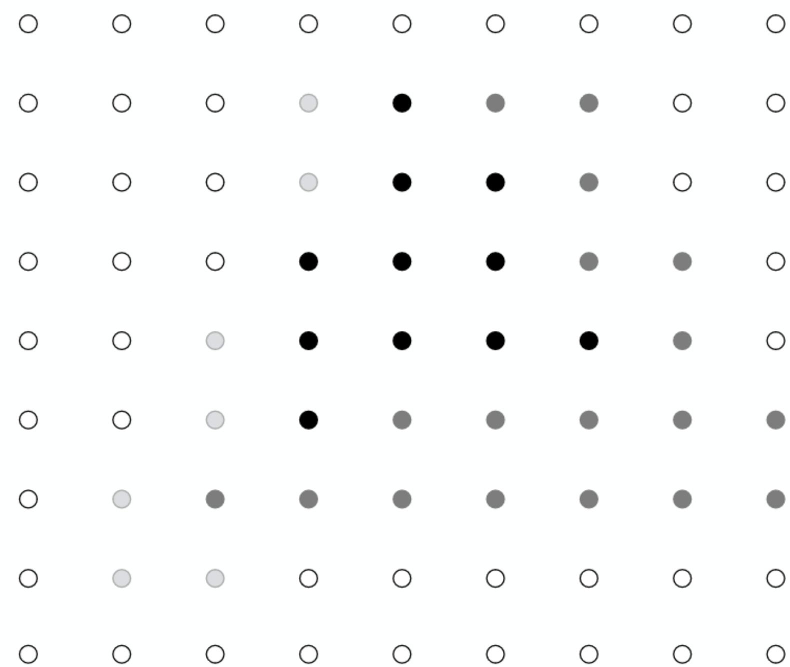
White = large distance

Black = small distance

Red = sample passed depth test



Color buffer contents

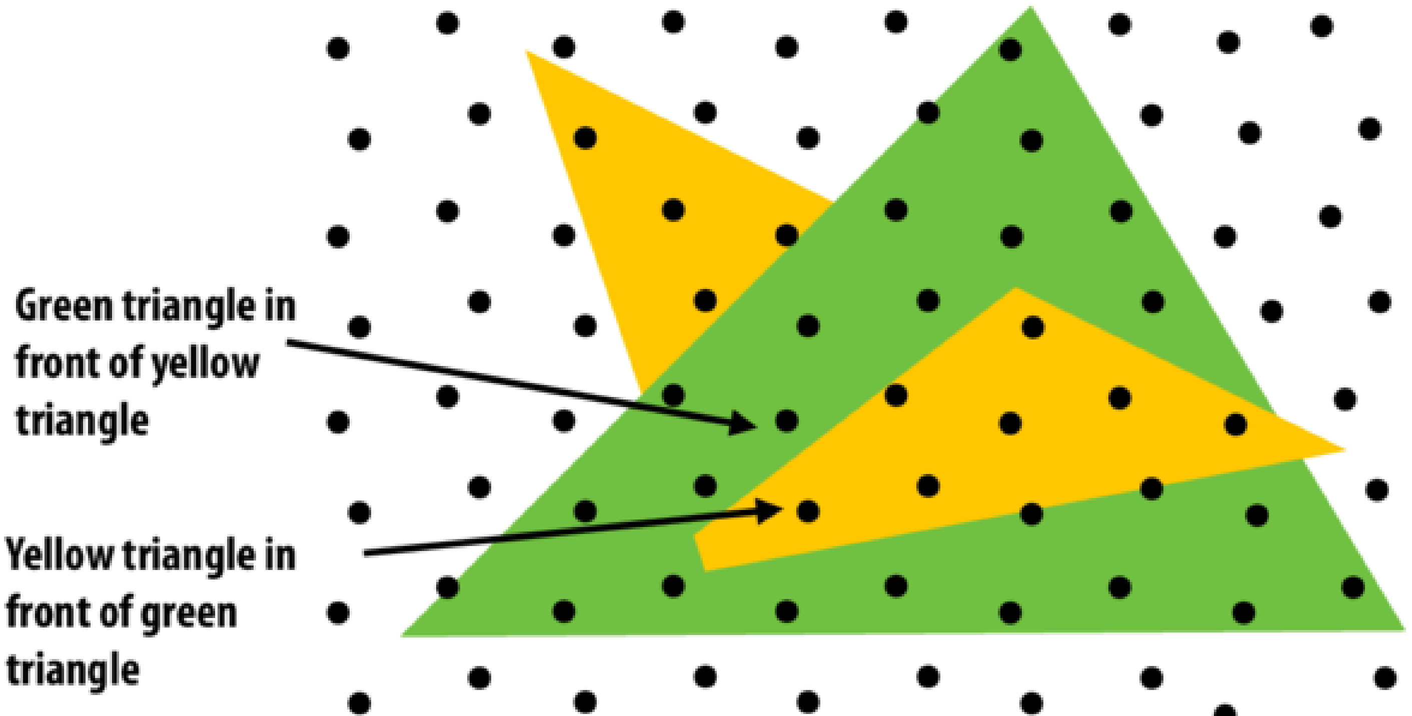


Depth buffer contents

Does depth-buffer algorithm handle interpenetrating surfaces?

Of course!

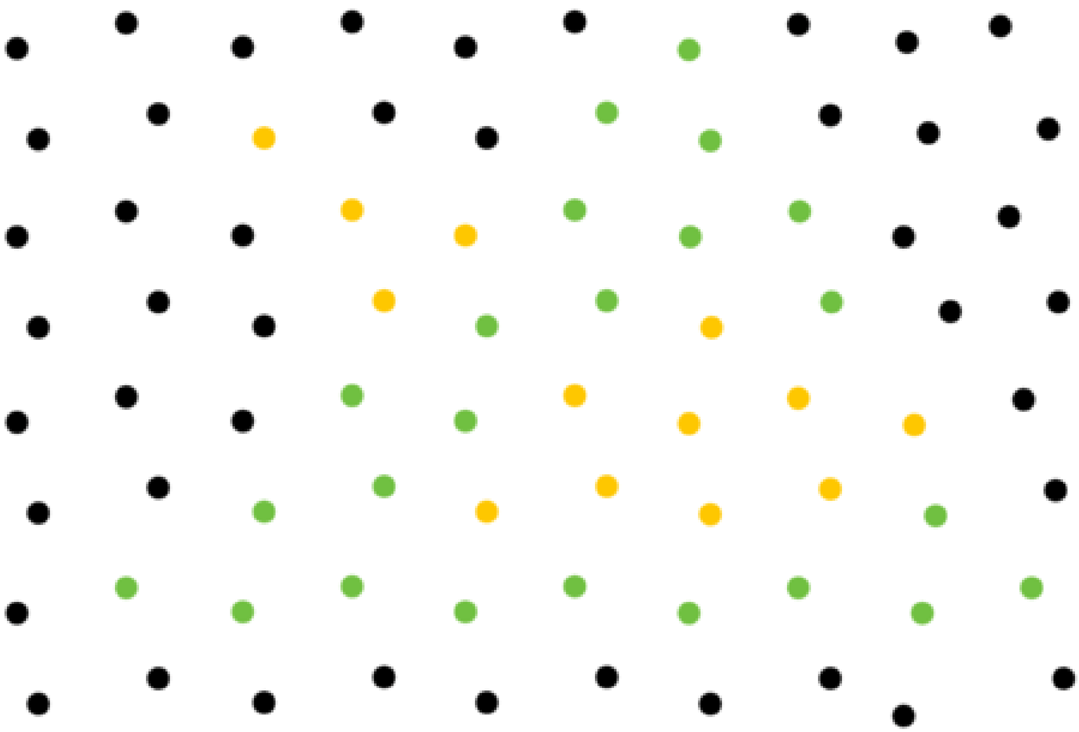
Occlusion test is based on depth of triangles at a given sample point. The relative depth of triangles may be different at different sample points.



Does depth-buffer algorithm handle interpenetrating surfaces?

Of course!

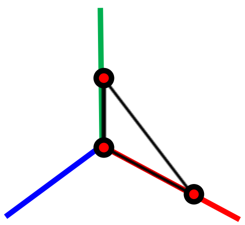
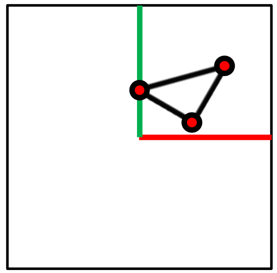
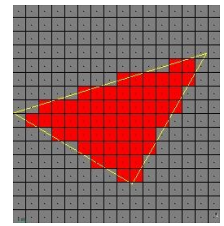
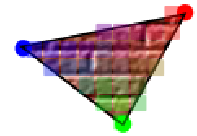
Occlusion test is based on depth of triangles at a given sample point. The relative depth of triangles may be different at different sample points.



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

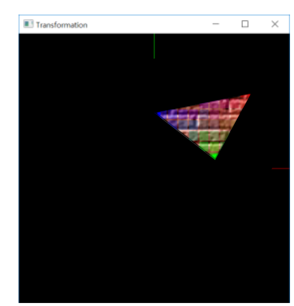
vertex processing → **rasterization** → **fragment processing** → output merging

vertex processing: performs a sequence of vertex transformations

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

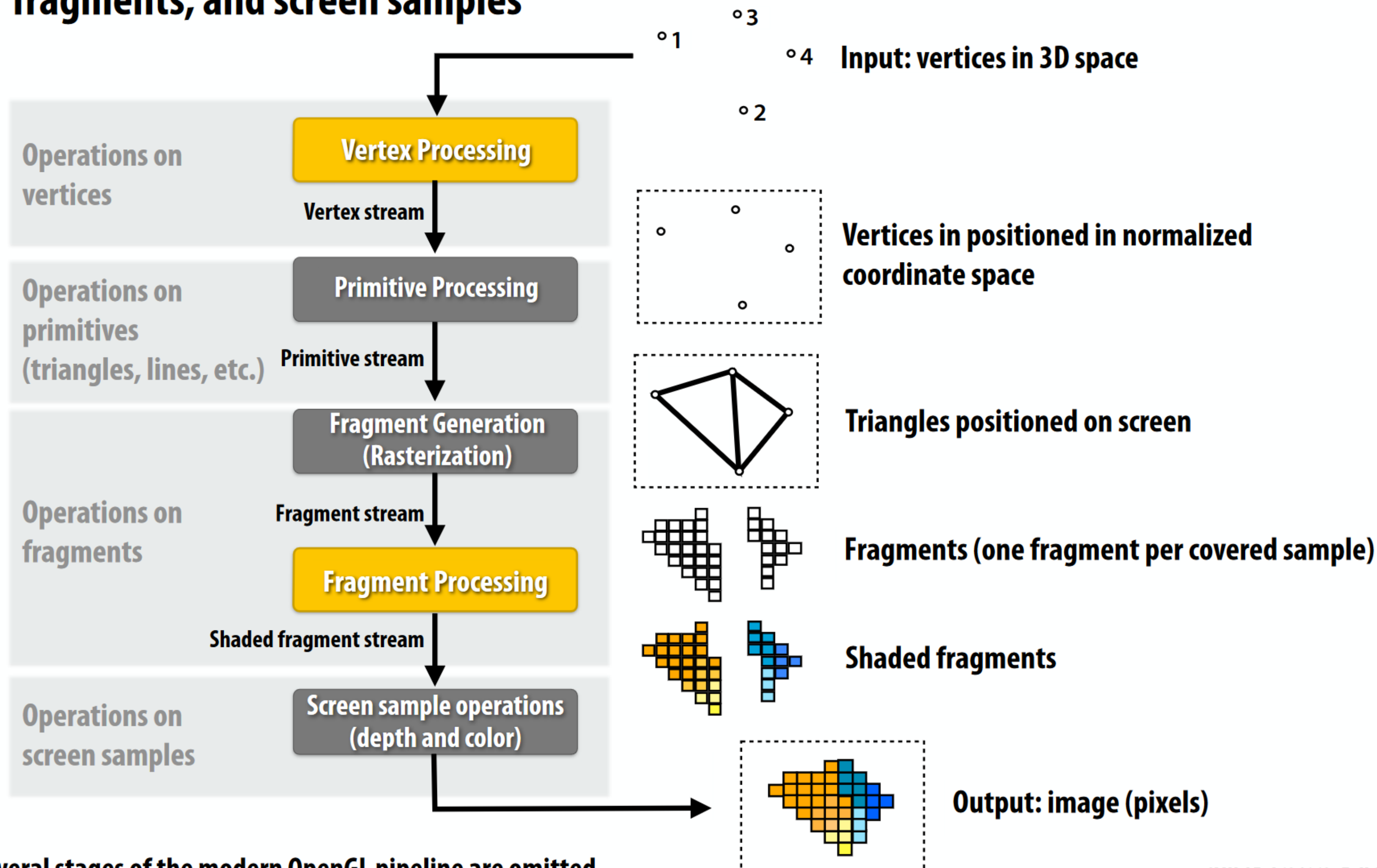
fragment processing: determines color of each fragment with light & texture

output merging



OpenGL/Direct3D graphics pipeline *

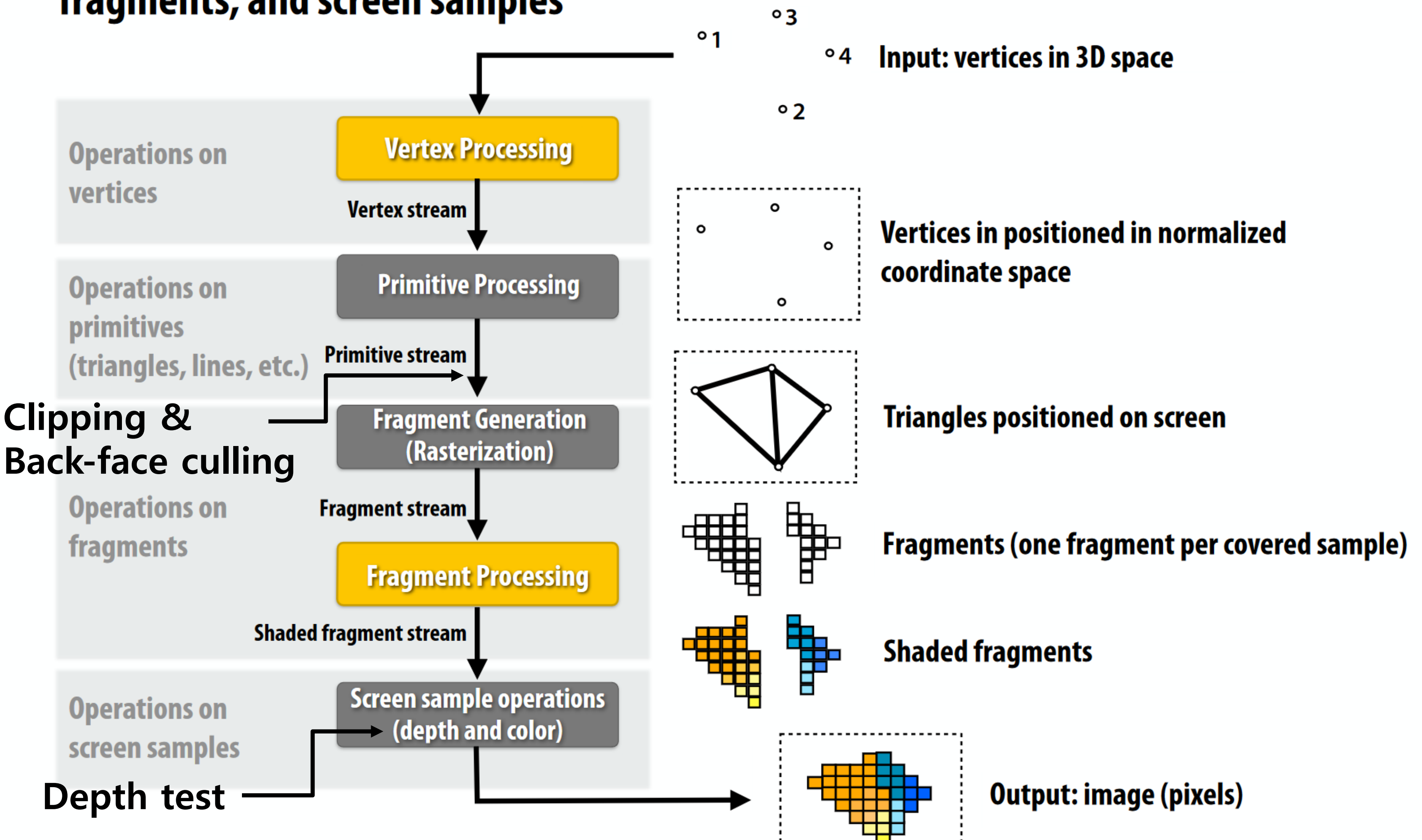
Structures rendering computation as a series of operations on vertices, primitives, fragments, and screen samples



* Several stages of the modern OpenGL pipeline are omitted

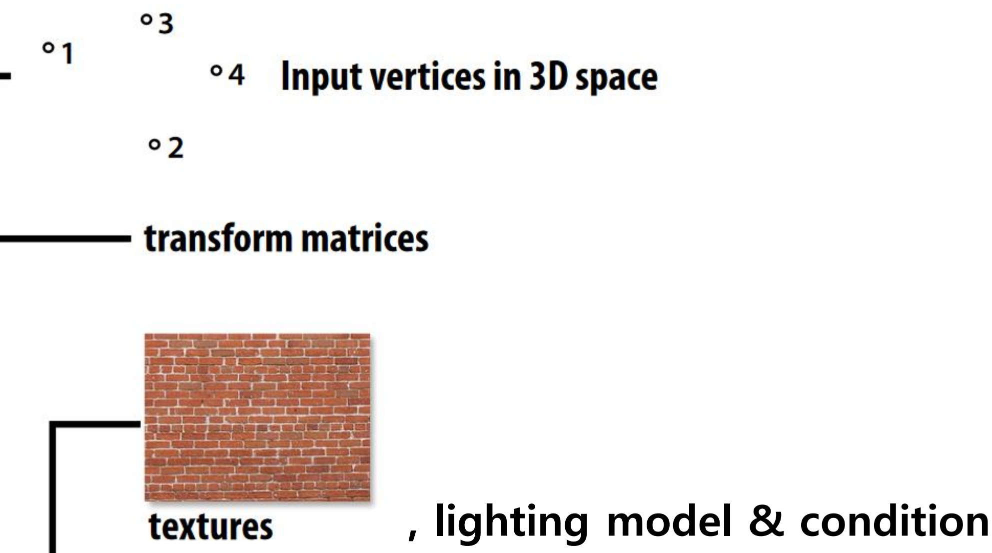
OpenGL/Direct3D graphics pipeline *

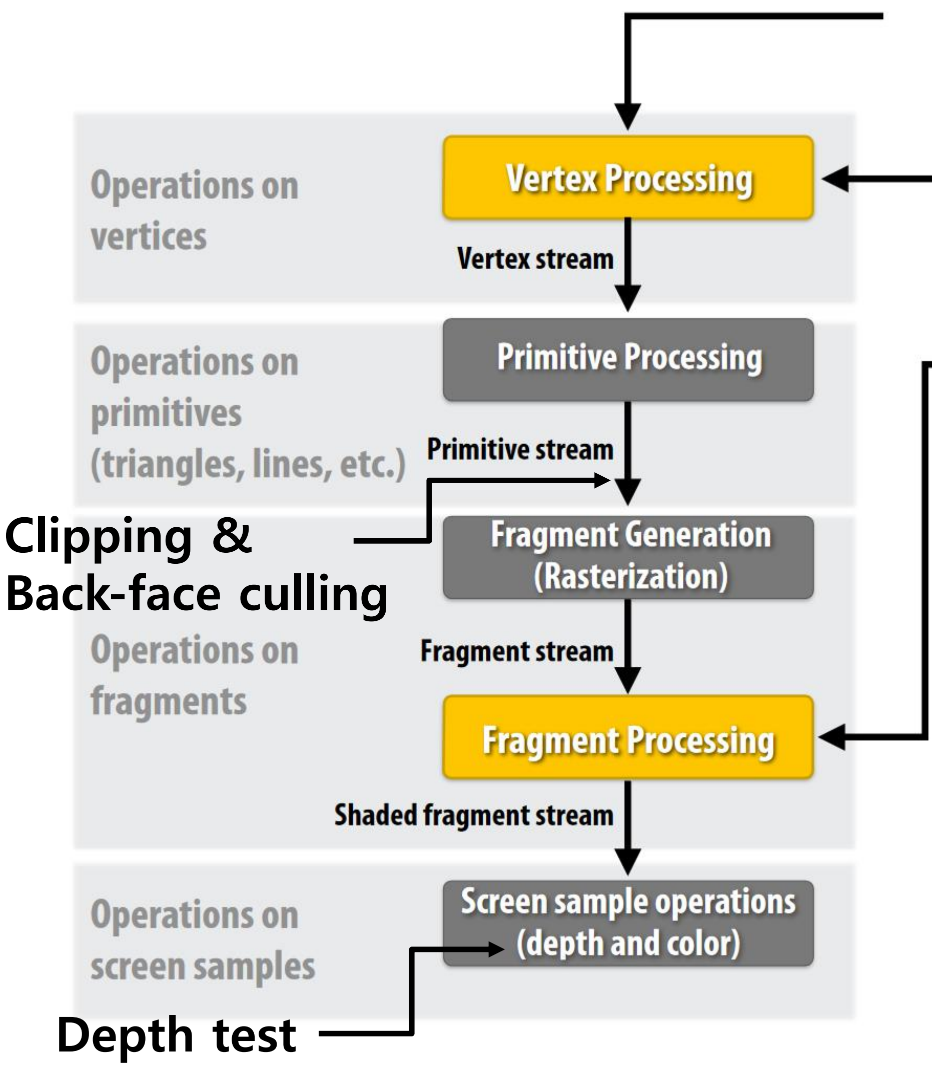
Structures rendering computation as a series of operations on vertices, primitives, fragments, and screen samples



* Several stages of the modern OpenGL pipeline are omitted

OpenGL/Direct3D graphics pipeline *



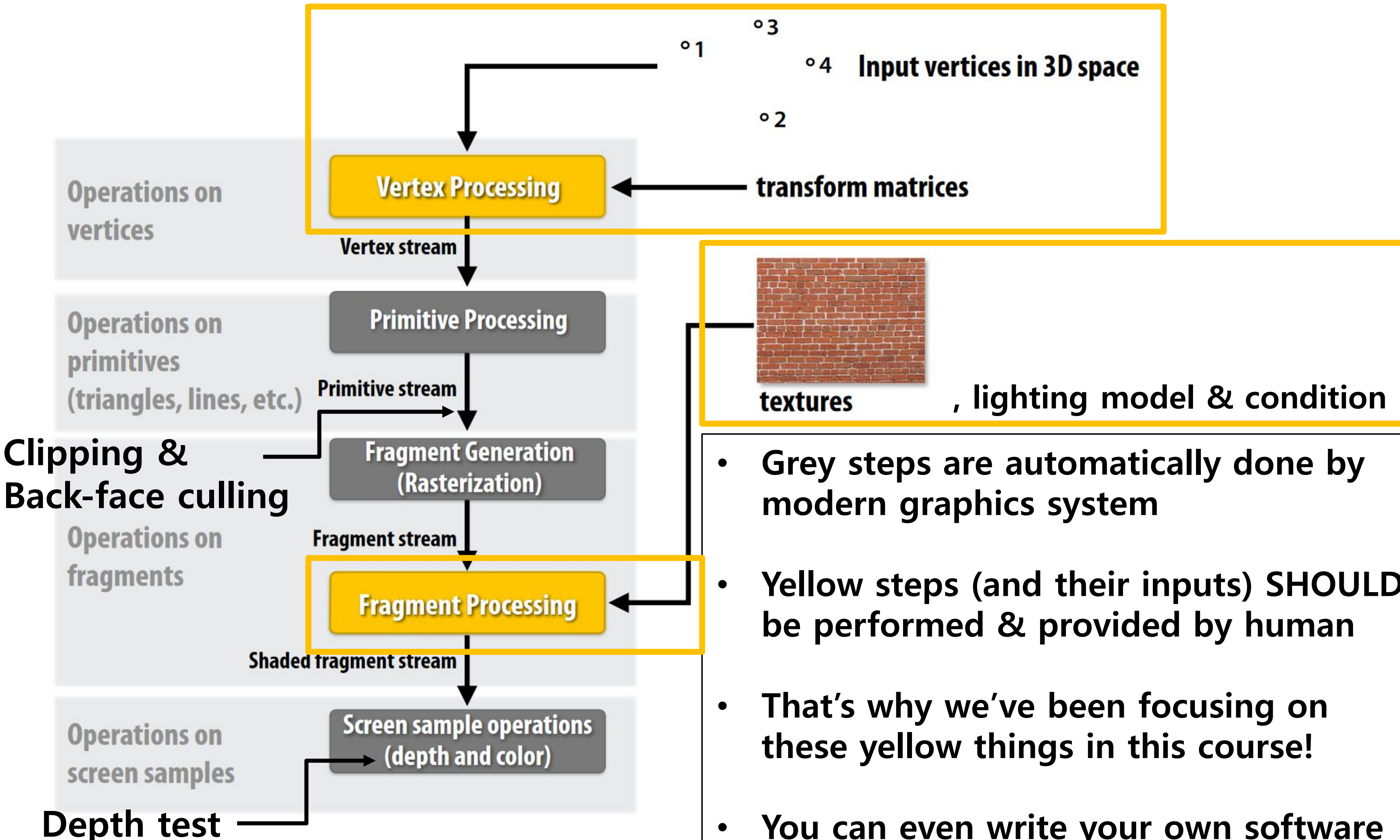


Pipeline inputs:

- 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 *



- Grey steps are automatically done by modern graphics system
- Yellow steps (and their inputs) SHOULD be performed & provided by human
- That's why we've been focusing on these yellow things in this course!
- You can even write your own software renderer that covers whole process!

* several stages of the modern OpenGL pipeline are omitted

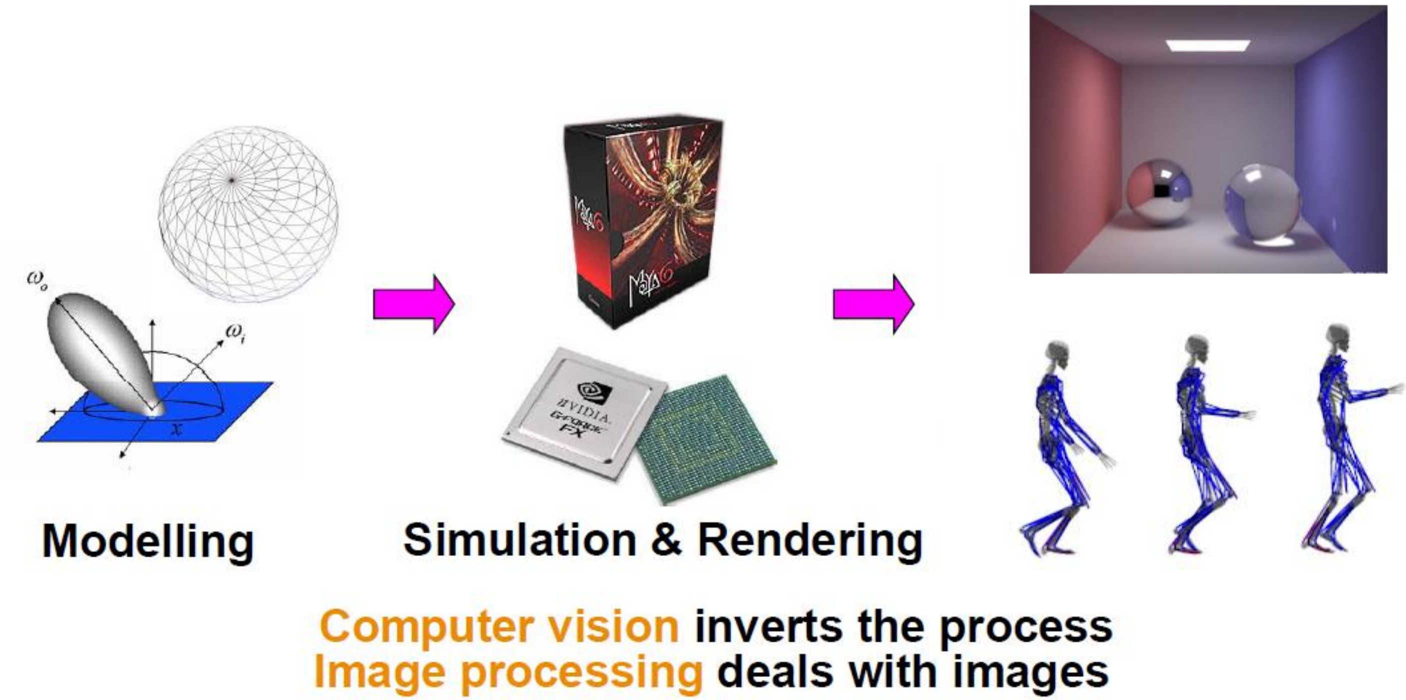
Acknowledgement

- Acknowledgement: Some materials come from the lecture slides of
  - Prof. Sung-eui Yoon, KAIST, https://sglab.kaist.ac.kr/~sungeui/CG/
  - Prof. JungHyun Han, Korea Univ., http://media.korea.ac.kr/book/
  - Prof. Taesoo Kwon, Hanyang Univ., http://calab.hanyang.ac.kr/cgi-bin/cg.cgi
  - Prof. Steve Marschner, Cornell Univ., http://www.cs.cornell.edu/courses/cs4620/2014fa/index.shtml
  - Prof. Kayvon Fatahalian and Prof. Keenan Crane, CMU, http://15462.courses.cs.cmu.edu/fall2015/

Course Wrap-up

Do you remember?

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



Image

Animation (a series of 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 Geometry, Rendering Pipeline 7 - Hierarchical Modeling, Mesh 9 - Orientation & Rotation 10 - Animation 11 - Curves
Mapping to 2D screen	5 - Affine Geometry, Rendering Pipeline 6 - Viewing, Projection
Shape	7 - Hierarchical Modeling, Mesh 11 - Curves
Appearance	8 - Lighting & Shading 12 - More Lighting, Texture
Rendering Pipeline	5 - Affine Geometry, Rendering Pipeline 13 - Rasterization & Visibility

How do you feel?

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

- If you think "that's me!" and are interested in doing some "research", please do not hesitate to mail to me.
  - yoonsanglee@hanyang.ac.kr

Thanks for being a great class!

(No lab in this week!)