
Computer Graphics

13 – Scan Conversion, Visibility

Yoonsang Lee
Hanyang University

Spring 2026

Final Exam Announcement (Unchanged)

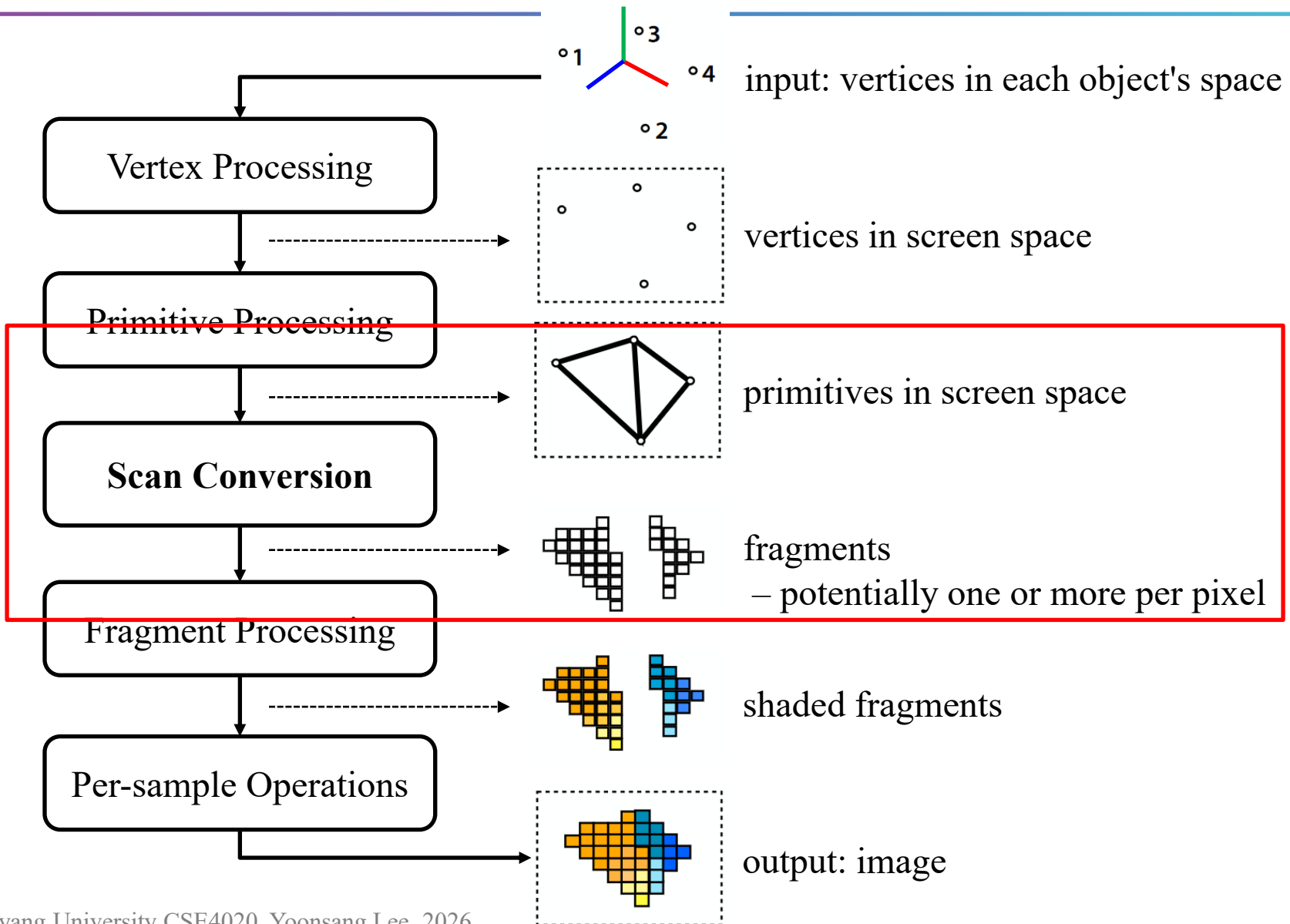
- Date & time: **June 22 (Mon), 10:00 - 11:30 AM**
- Place: IT.BT Room **508 & Room 510**
 - **Check the announcement in LMS** for your assigned exam room:
https://learning.hanyang.ac.kr/courses/201420/discussion_topics/501477
- Scope: Lecture, Lab, Lab Check **8 ~ 13, Project 2 & 3**
- All other exam policies remain unchanged.
- Bring your **student ID card**. It will be checked during the exam.

Outline

- Scan Conversion
- Visibility Problem
 - Clipping (Viewing frustum culling)
 - Back-face culling
 - Hidden surface removal
- Rendering Pipeline Again
- Course Wrap-up

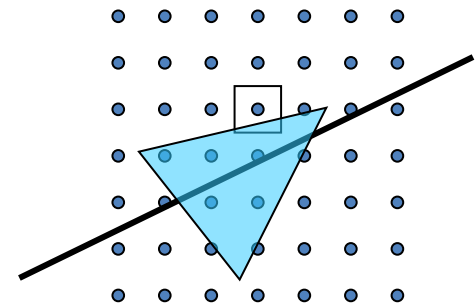
Scan Conversion

Recall: Rendering Pipeline



Scan Conversion

- Scan conversion converts a **vertex-based representation** into a **fragment-based representation**.



- First task: Determine **which fragments are covered** by a primitive.
 - A primitive is a basic geometric object such as a point, line, or triangle.
- Second task: **Interpolate vertex attributes** for each covered fragment.
 - e.g., color, normal, texture coordinates.

Scan Conversion

- Algorithms that determine which fragments are covered by a primitive are often called **drawing algorithms**.
 - In raster graphics (pixel-based graphics), "drawing" a primitive essentially means finding the pixels/fragments covered by that primitive.
- Line drawing algorithms
 - Digital differential analyzer (DDA)
 - Bresenham's line algorithm (1962)
 - Xiaolin Wu's line algorithm (1991)
 - ...
 - For details, refer to https://www.tutorialspoint.com/computer_graphics/line_generation_algorithm.htm

Scan Conversion

- Circle drawing algorithms
 - Midpoint circle algorithm
 - Bresenham's circle algorithm
 - Xiaolin Wu's circle algorithm
 - ...
 - For details, refer to https://www.tutorialspoint.com/computer_graphics/circle_generation_algorithm.htm
- Polygon drawing algorithms
 - Edge Function-Based Rasterization
 - Barycentric Coordinate Rasterization
 - Scanline Rasterization
 - ...
 - For details, refer to
 - <https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/rasterization-stage.html>
 - https://www.tutorialspoint.com/computer_graphics/polygon_filling_algorithm.htm

Scan Conversion

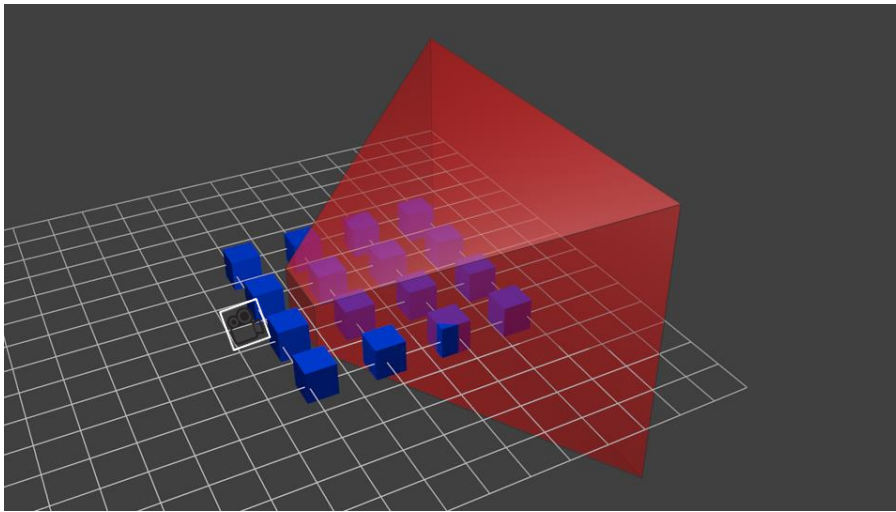
- We will not go into the details of these algorithms.
- In fact, these tasks are more complex than they may seem.
 - Computational efficiency, anti-aliasing, numerical robustness, etc.
- Modern graphics APIs (e.g., OpenGL) provide built-in support for these operations, which are efficiently handled by GPUs.
 - These algorithms are well-established and highly optimized today.
- Today, we can simply use graphics APIs to draw primitives without worrying about these low-level details.

Visibility Problem

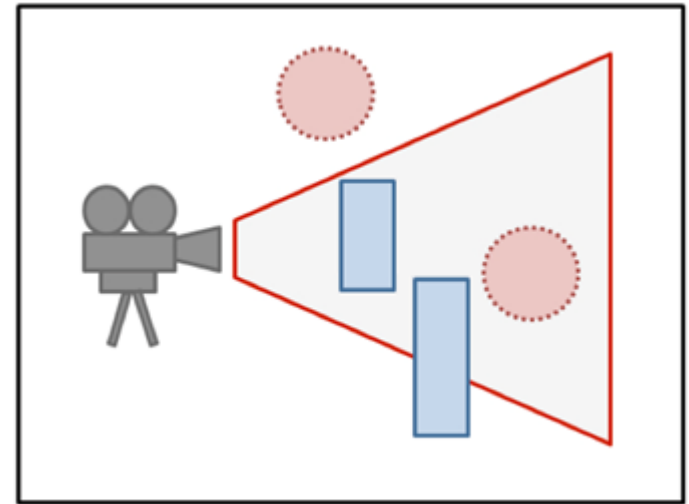
Visibility Problem

- What is VISIBLE to the camera?

Red: viewing frustum, Blue: objects

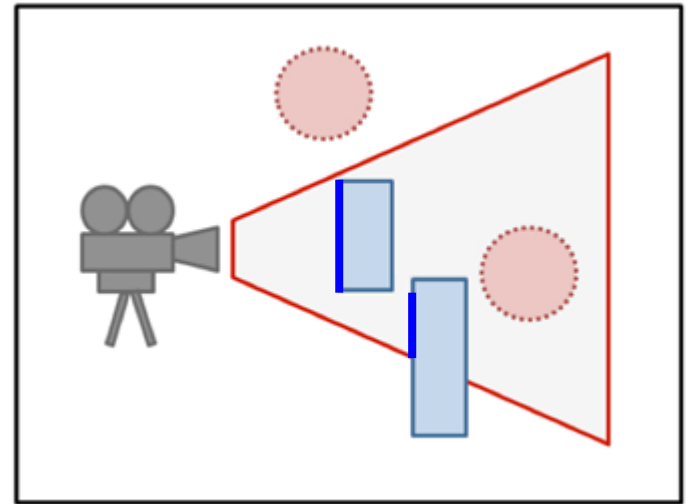
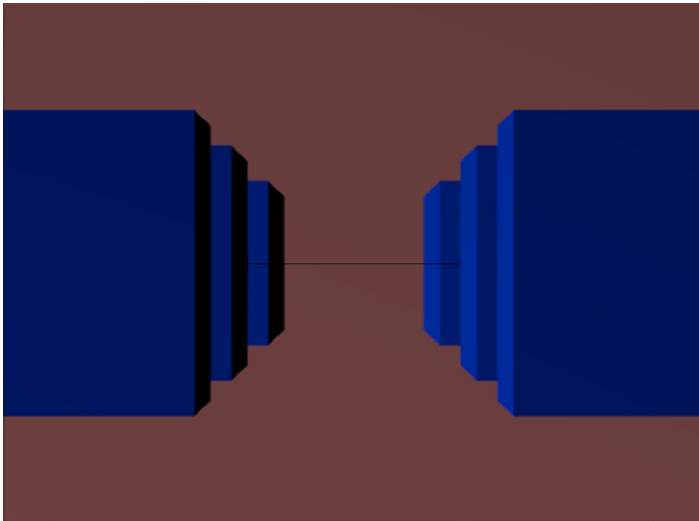


Red solid: viewing frustum
Red dotted & Blue: objects



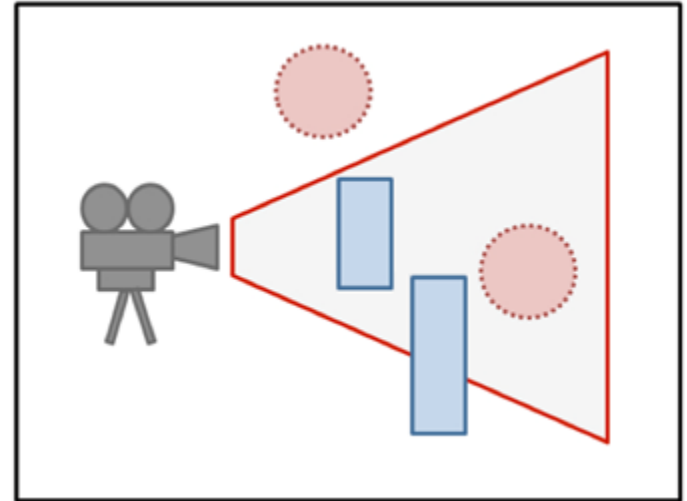
Visibility Problem

- The answer is:



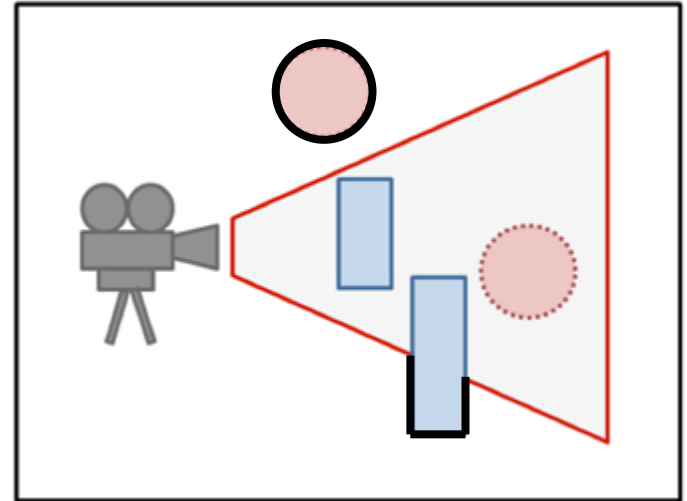
Visibility Problem

- What is NOT VISIBLE to the camera?



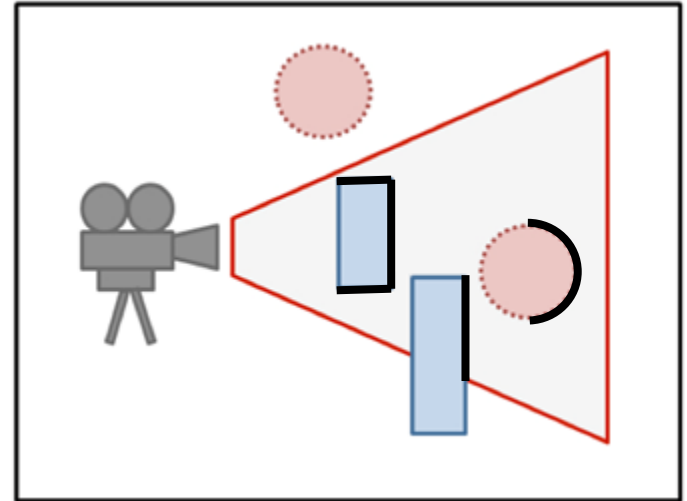
Visibility Problem

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



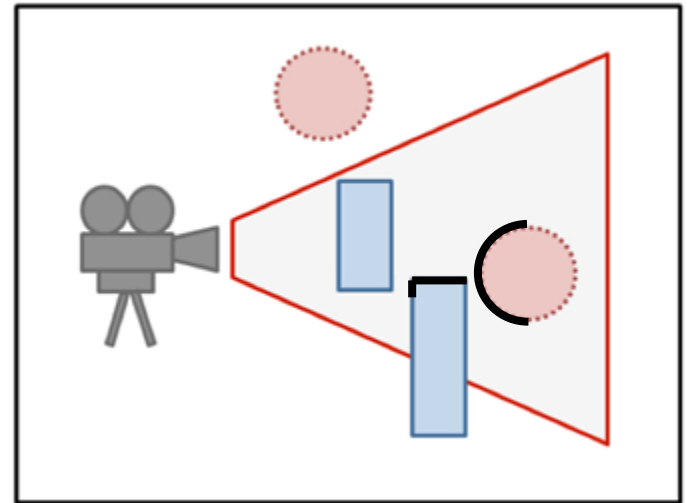
Visibility Problem

- What is NOT VISIBLE to the camera?
- Primitives outside the viewing frustum
- **Back-facing primitives**



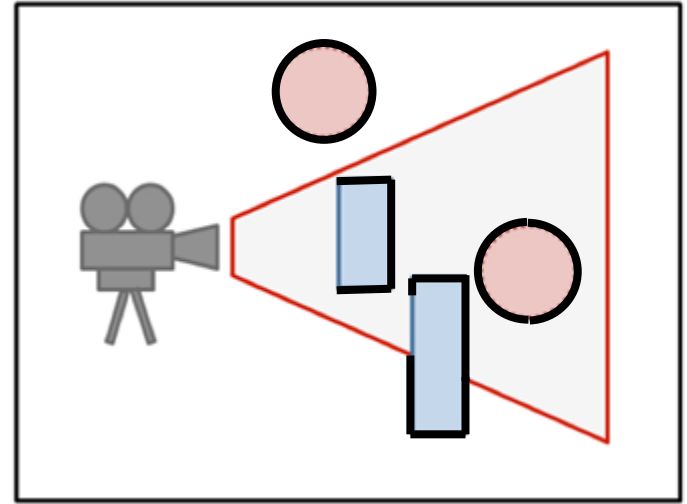
Visibility Problem

- What is NOT VISIBLE to the camera?
- Primitives outside the viewing frustum
- Back-facing primitives
- **Primitives occluded by other objects closer to the camera**



Visibility Problem

- These invisible primitives should be removed because:
 - Rendering invisible primitives wastes computation.
 - Closer objects should hide farther objects.
- Therefore, visibility determination is essential for both:
 - Efficiency
 - Correctness

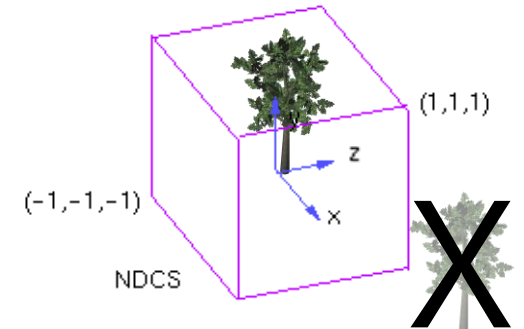
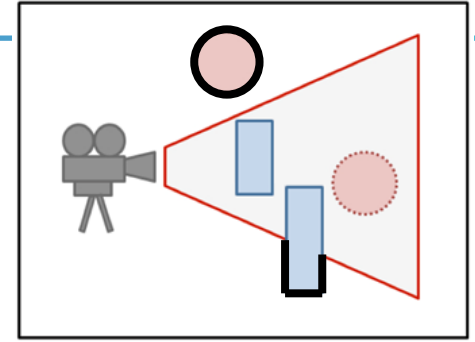


Visibility Problem

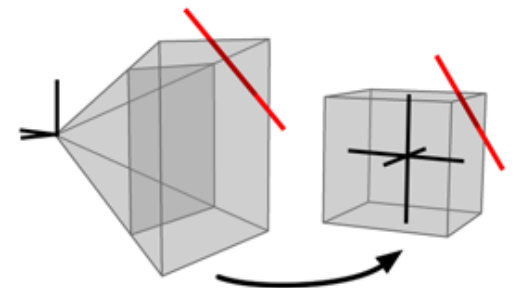
- Removing...
- Primitives outside the viewing frustum
- → **Clipping (Viewing frustum culling)**
- Back-facing primitives
- → **Back-face culling**
- Primitives occluded by other objects closer to the camera
- → **Hidden surface removal**

Clipping (Viewing Frustum Culling)

- Removing primitives outside the viewing frustum
- Clipping is performed in **clip space**.
 - Recall: A point (x',y',z') in **NDC space** is inside the canonical view volume if $-1 \leq x',y',z' \leq 1$.
 - Equivalently, a point (x,y,z,w) in **clip space** is inside the view volume if $-w \leq x,y,z \leq w$.
- → Clipped primitives can be discarded before perspective division.
- Clipping is simpler than in view space, where the view volume is a frustum.

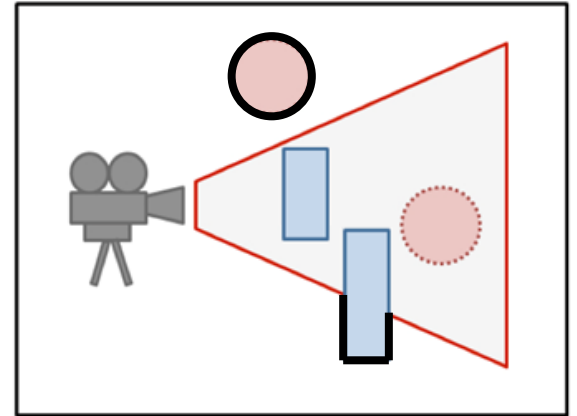


$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x/w \\ y/w \\ z/w \\ 1 \end{bmatrix} \sim \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$



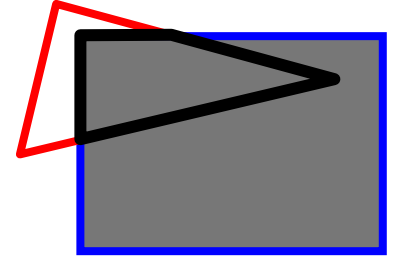
Clipping (Viewing Frustum Culling)

- Line clipping algorithms
 - Cohen–Sutherland (1967)
 - Cyrus–Beck (1978)
 - Liang–Barsky (1984)
 - ...
- Polygon clipping algorithms
 - Sutherland–Hodgman (1974)
 - Weiler–Atherton (1977)
 - ...
- For details, refer to
https://www.tutorialspoint.com/computer_graphics/viewing_and_clipping.htm



Clipping (Viewing Frustum Culling)

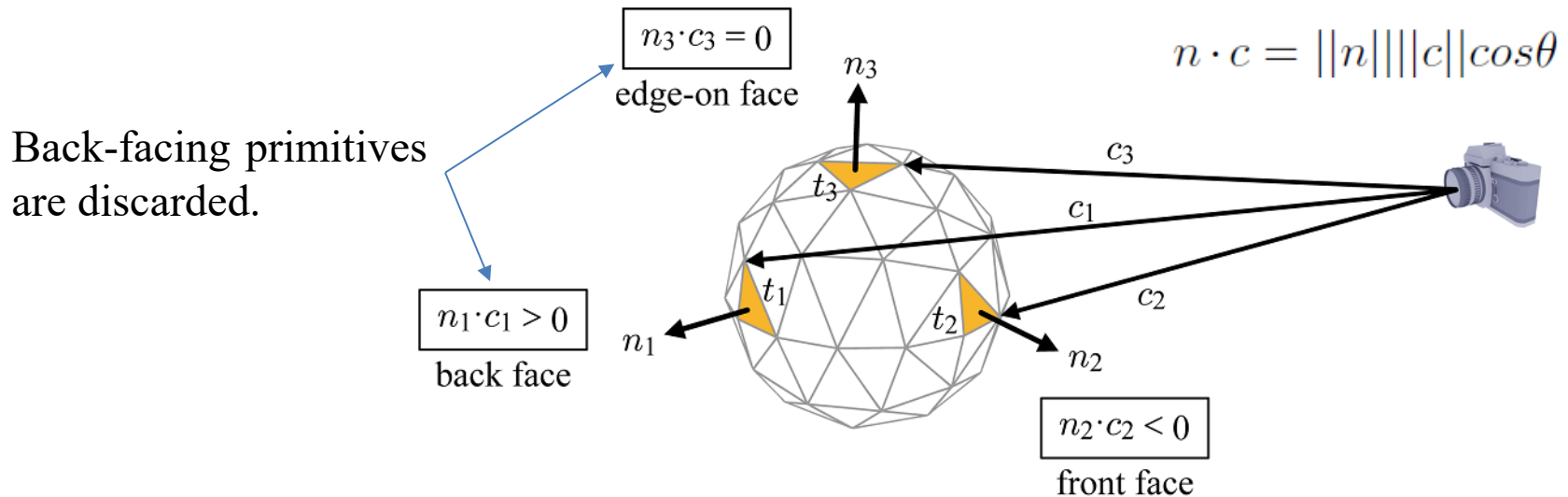
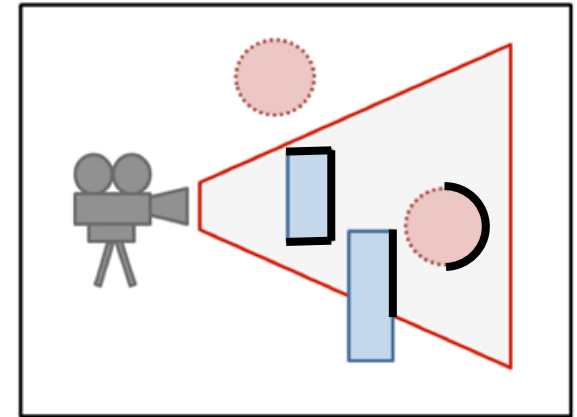
- Clipping, especially polygon clipping, is more complex than it may seem.
 - Vertices may be added or removed during clipping.
- We will not go into the details of clipping algorithms.
- Clipping is automatically handled by modern GPUs once the view frustum is defined through the projection matrix.



triangle → quad

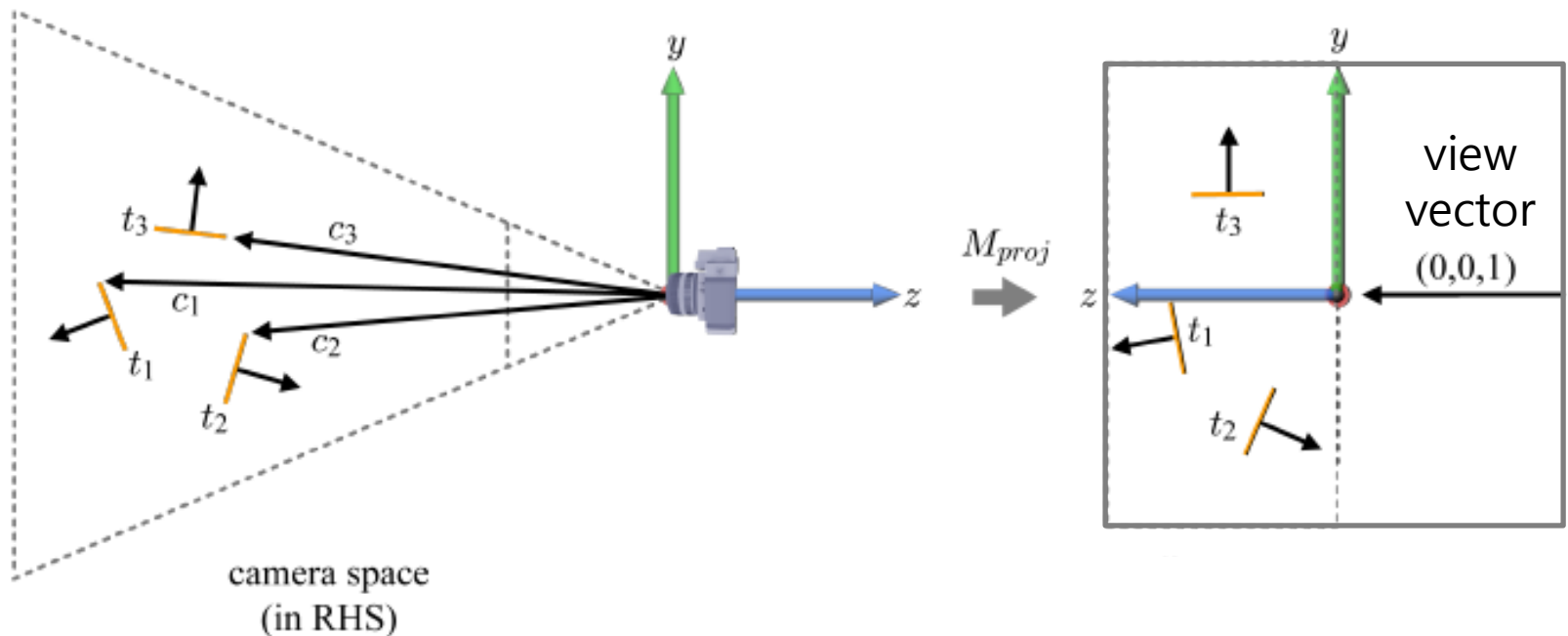
Back-Face Culling

- Removing back-facing primitives
- A face is back-facing when its normal points away from the camera.
- This can be determined by the sign of the dot product between the normal and view vector.

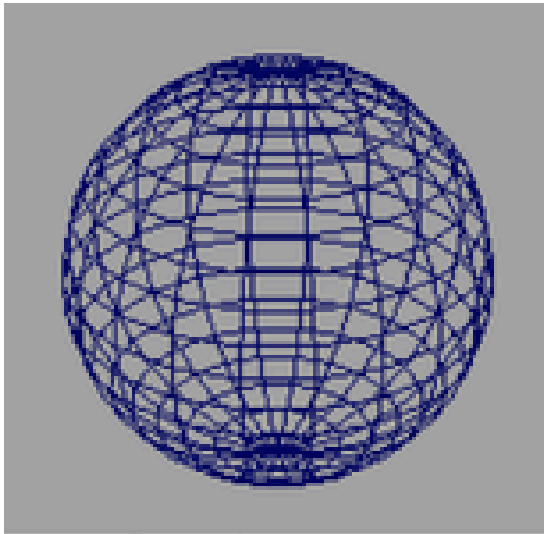


Back-Face Culling

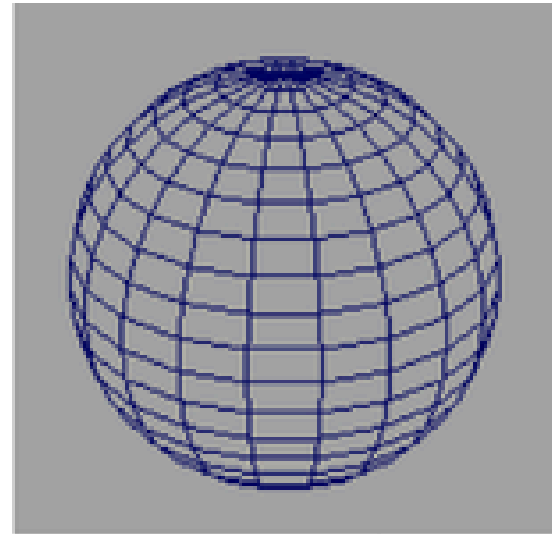
- Back-face culling is performed after projection and before rasterization.
 - After projection, all primitives can be tested against **the same viewing direction**, making back-face culling efficient.



Back-Face Culling



Backfaces

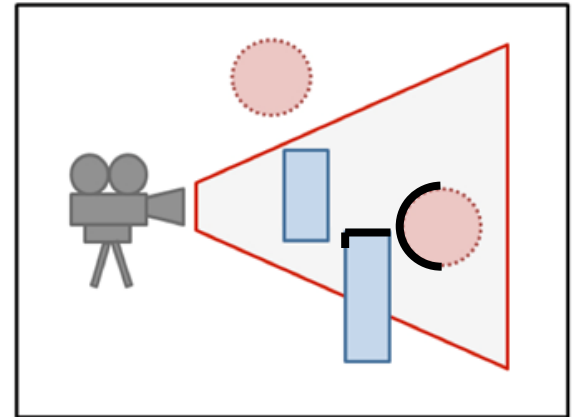


No backfaces

* This image is from <https://help.autodesk.com/view/MAYAUL/2024/ENU/?guid=GUID-B7F70ACE-0F3F-483B-83B5-D9711D6CBAAC>

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

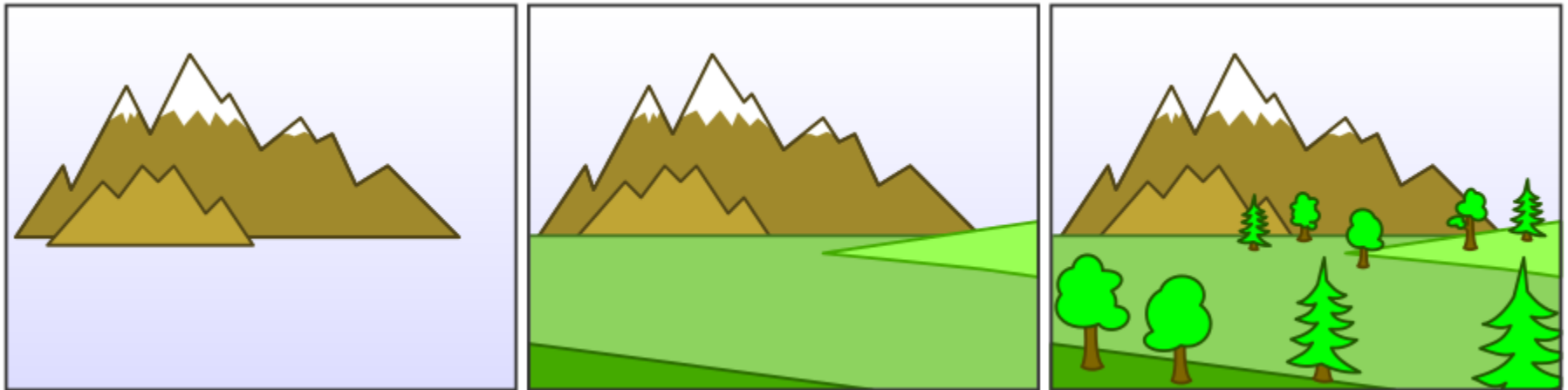


Hidden Surface Removal

- Historically, many approaches have been proposed:
 - Z-buffering (a.k.a. Depth buffering)
 - Painter's algorithm
 - BSP tree traversal
 - ...
- Z-buffering has become the standard technique in modern graphics systems.
- Now, let's explore the basic concepts behind the Painter's algorithm and Z-buffering.

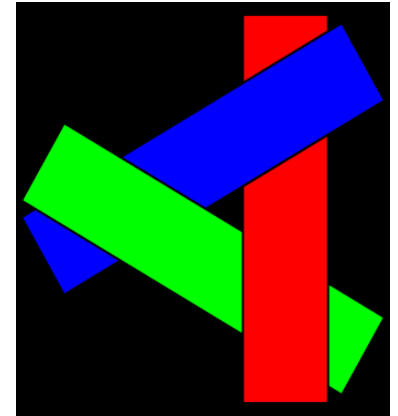
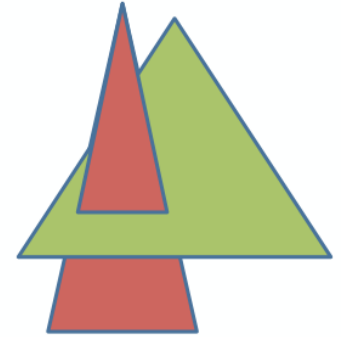
Painter's Algorithm

- Sorts polygons by their *depth* (distance from the viewer) and renders them from farthest to nearest.
- Polygons that are closer to the viewer will be drawn on top of polygons that are farther away.
- Works on a polygon-by-polygon basis.



Weakness of Painter's Algorithm

- What if visibility relationships form a cycle?
 - Polygons may need to be split into smaller pieces.
- Requires sorting all polygons by their depth whenever the viewer's perspective or object placement changes.
- → Expensive for dynamic scenes

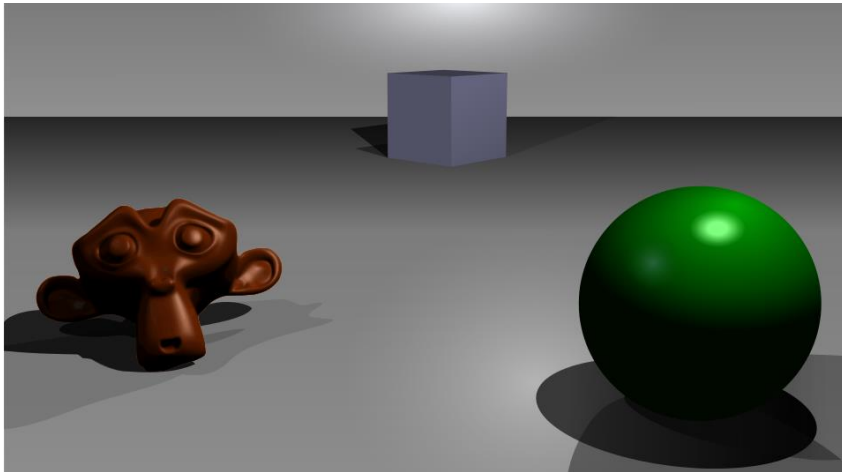


Z Buffering

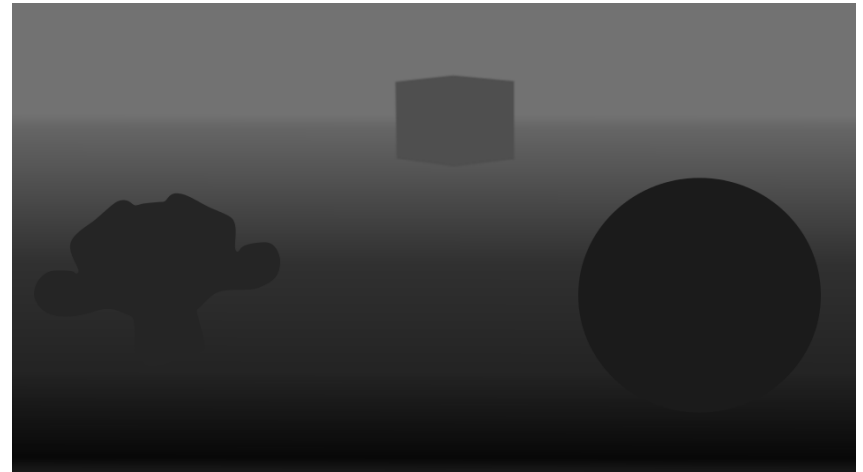
- A separate buffer called the *z-buffer* (a.k.a. *depth buffer*) stores the “depth” value of each fragment.
 - Often described as operating per pixel, but technically depth testing is performed per fragment.
- During rendering, when a newly rendered primitive covers a pixel, the depth of its fragment is compared against the depth currently stored in the z-buffer.
 - If the fragment (for the new primitive) is closer, the fragment is drawn and its depth value is stored in the z-buffer.
 - If the fragment (for the new primitive) is farther away, the fragment is discarded and the existing pixel color remains unchanged.
- Works on a fragment-by-fragment basis.

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.
```

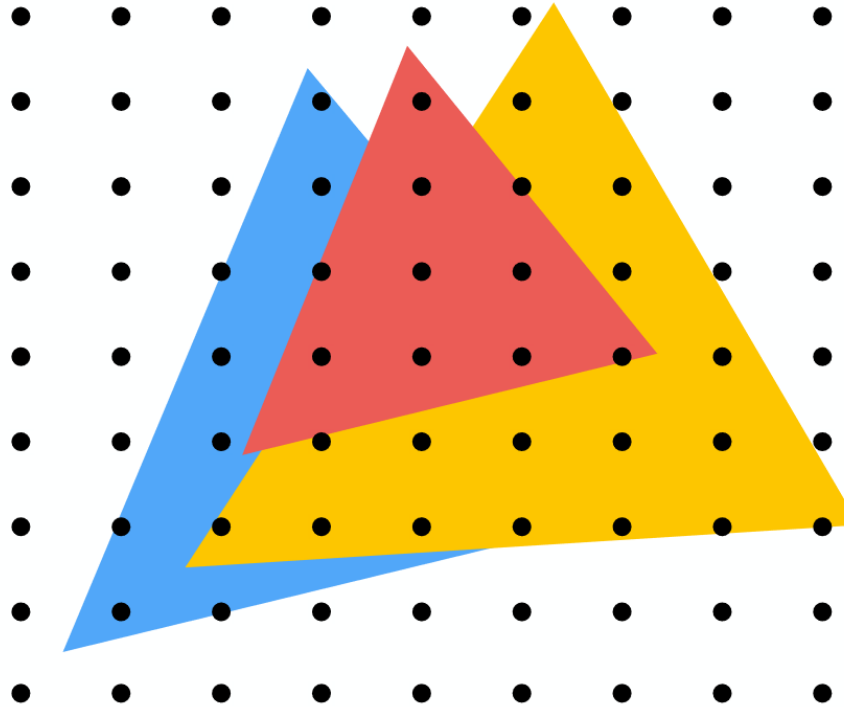


Frame buffer



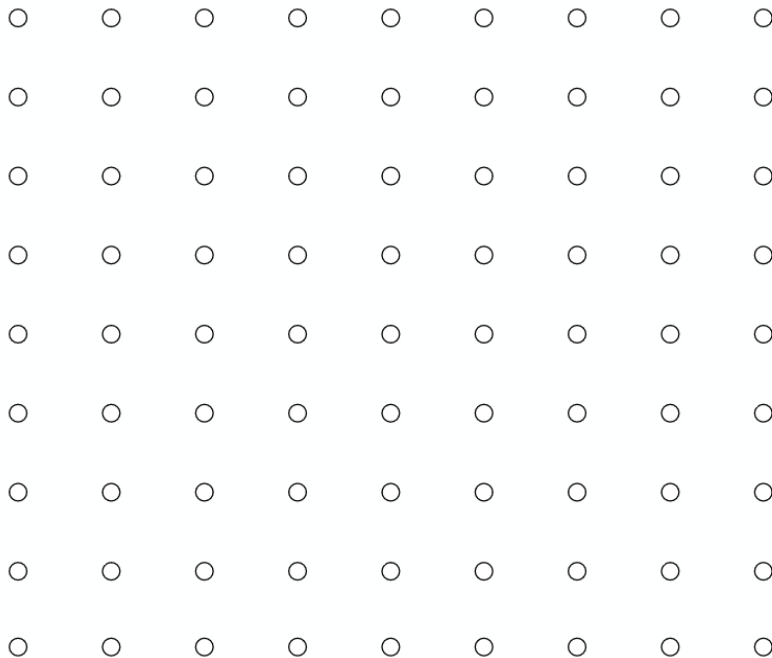
Z-buffer (Depth buffer)

Example: rendering three opaque triangles



Occlusion using the depth-buffer (Z-buffer)

Processing yellow triangle:
depth = 0.5



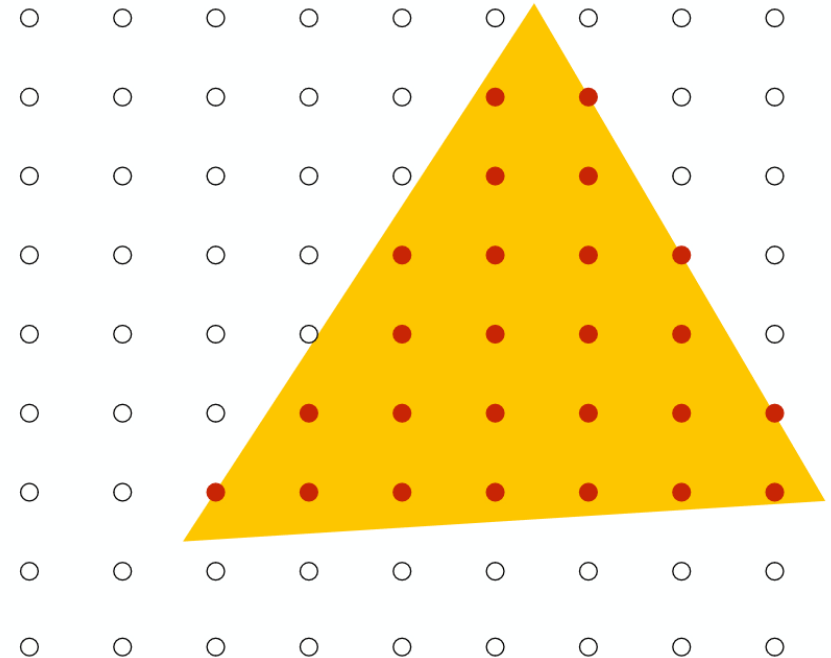
Color buffer contents
= Frame buffer

Grayscale value of sample point
used to indicate distance

White = large distance

Black = small distance

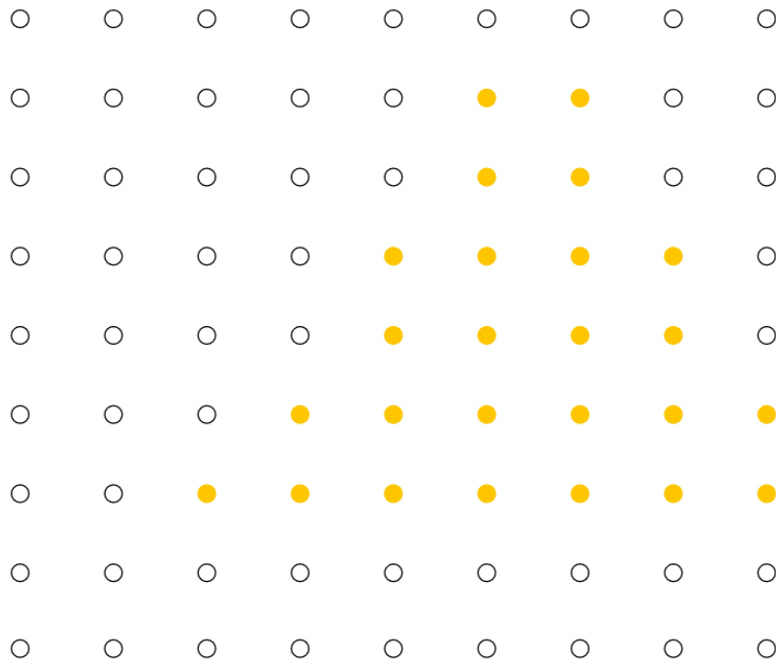
Red = sample passed depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing yellow triangle:



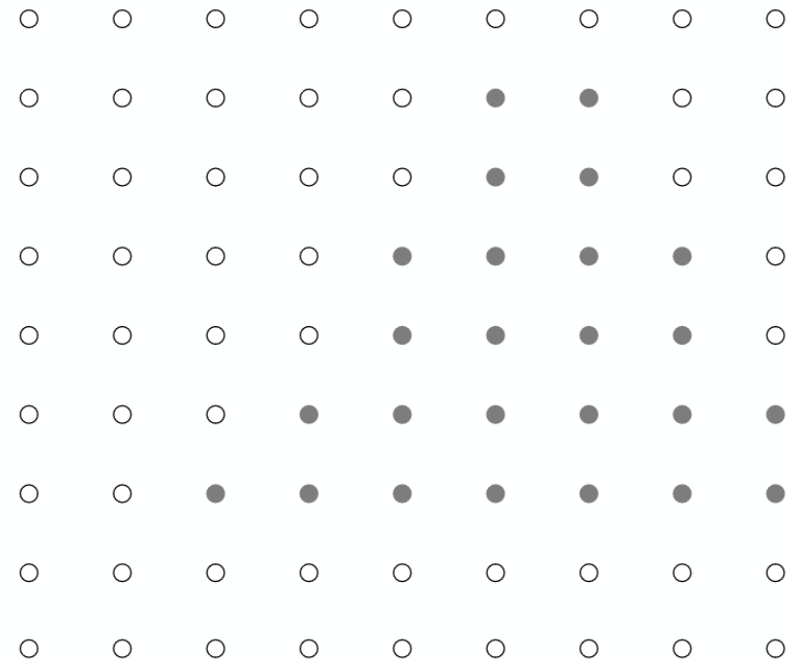
Color buffer contents

Grayscale value of sample point
used to indicate distance

White = large distance

Black = small distance

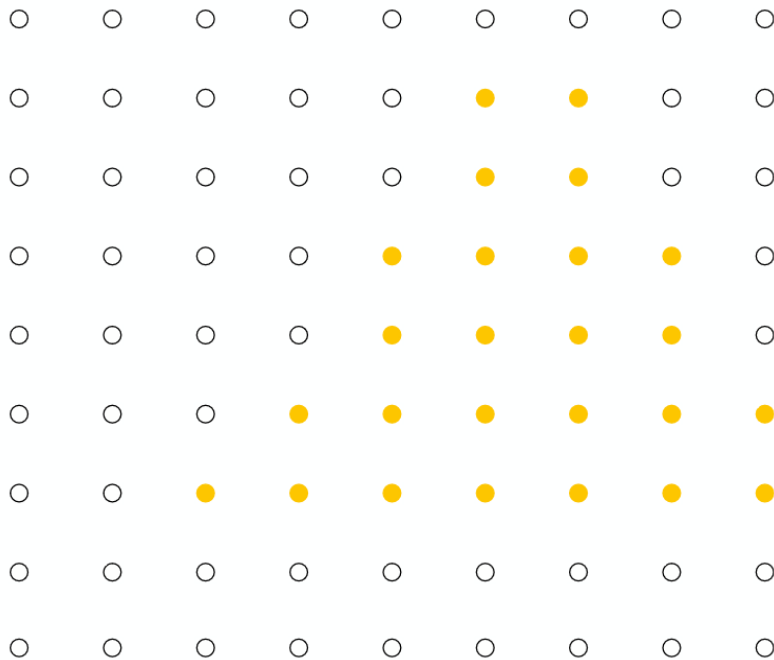
Red = sample passed depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

Processing blue triangle:
depth = 0.75



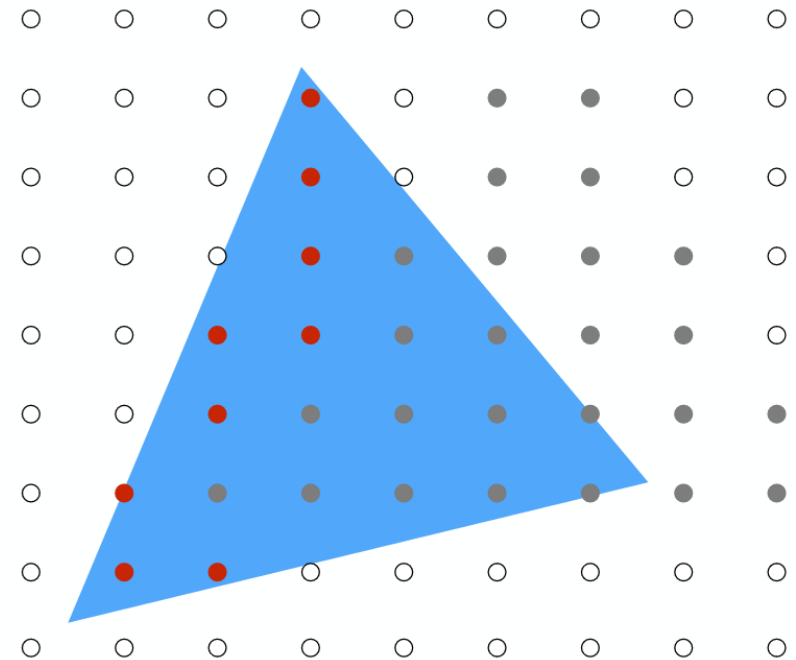
Color buffer contents

Grayscale value of sample point
used to indicate distance

White = large distance

Black = small distance

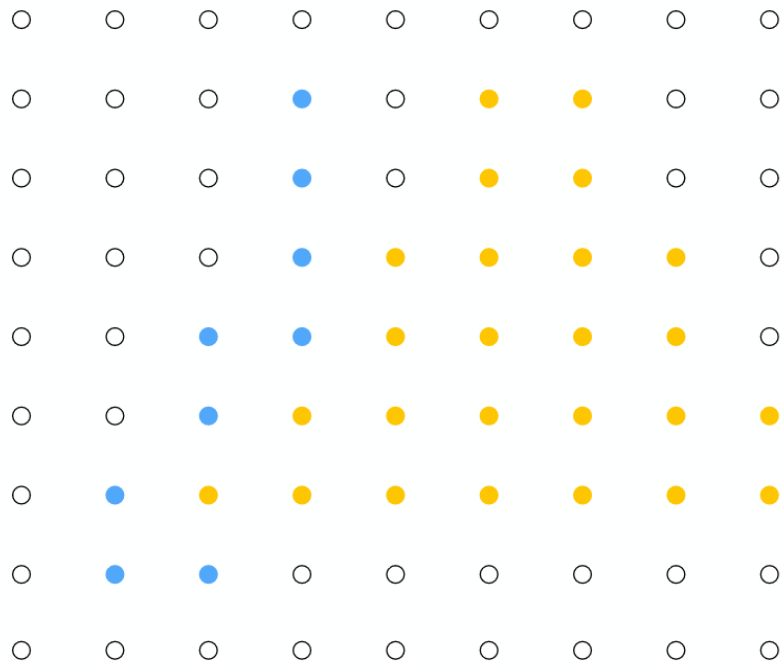
Red = sample passed depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing blue triangle:



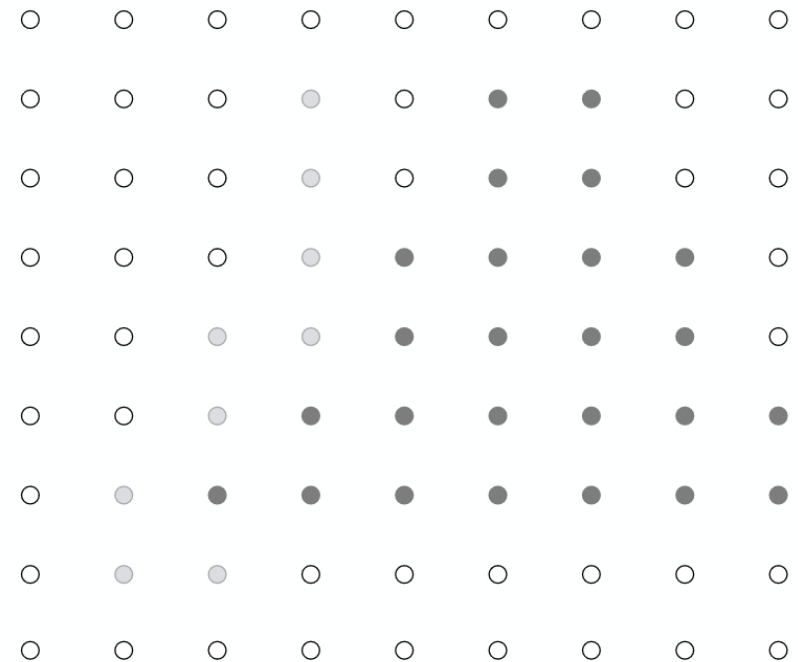
Color buffer contents

Grayscale value of sample point
used to indicate distance

White = large distance

Black = small distance

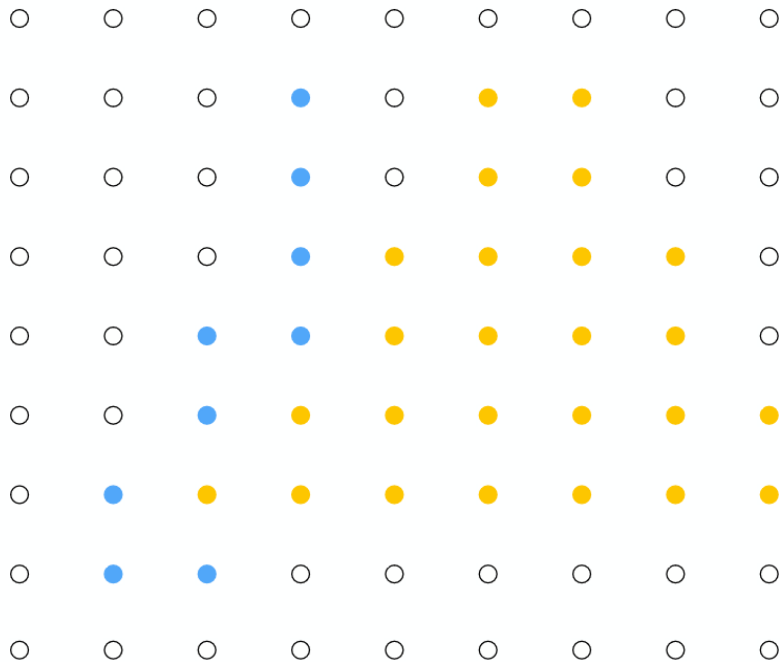
Red = sample passed depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

Processing red triangle:
 $\text{depth} = 0.25$



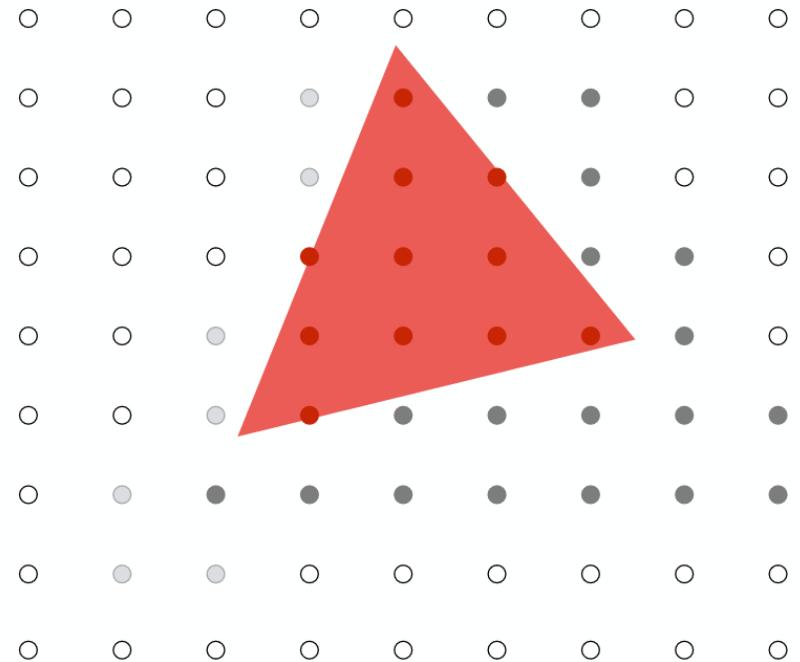
Color buffer contents

Grayscale value of sample point
used to indicate distance

White = large distance

Black = small distance

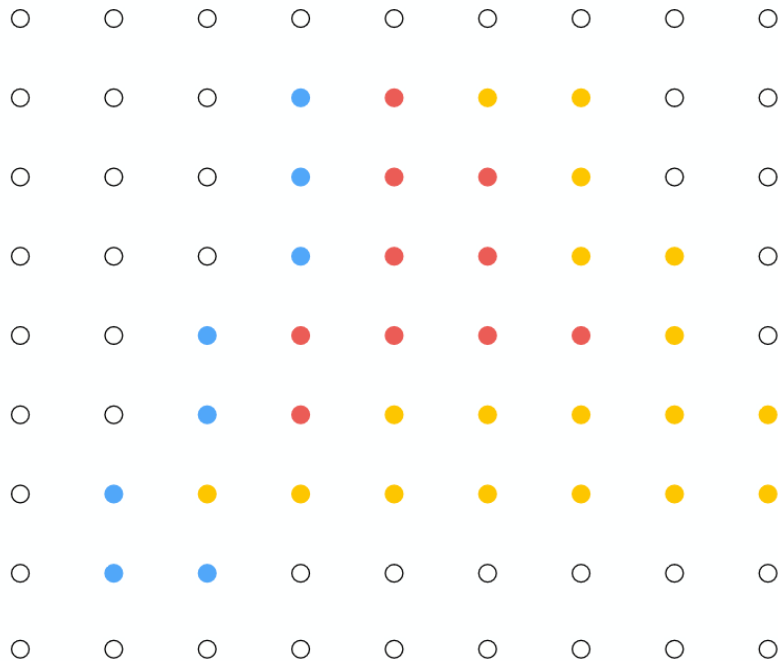
Red = sample passed depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing red triangle:



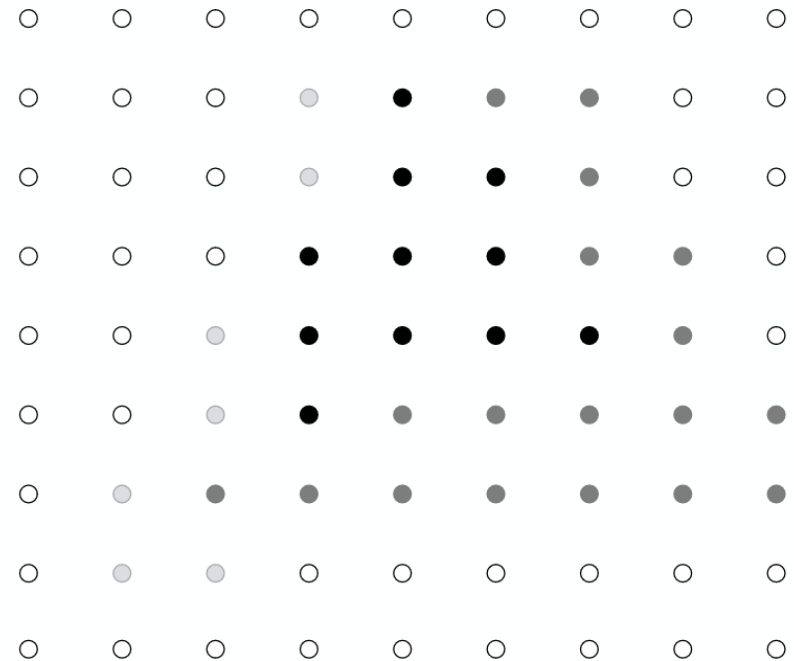
Color buffer contents

Grayscale value of sample point
used to indicate distance

White = large distance

Black = small distance

Red = sample passed depth test

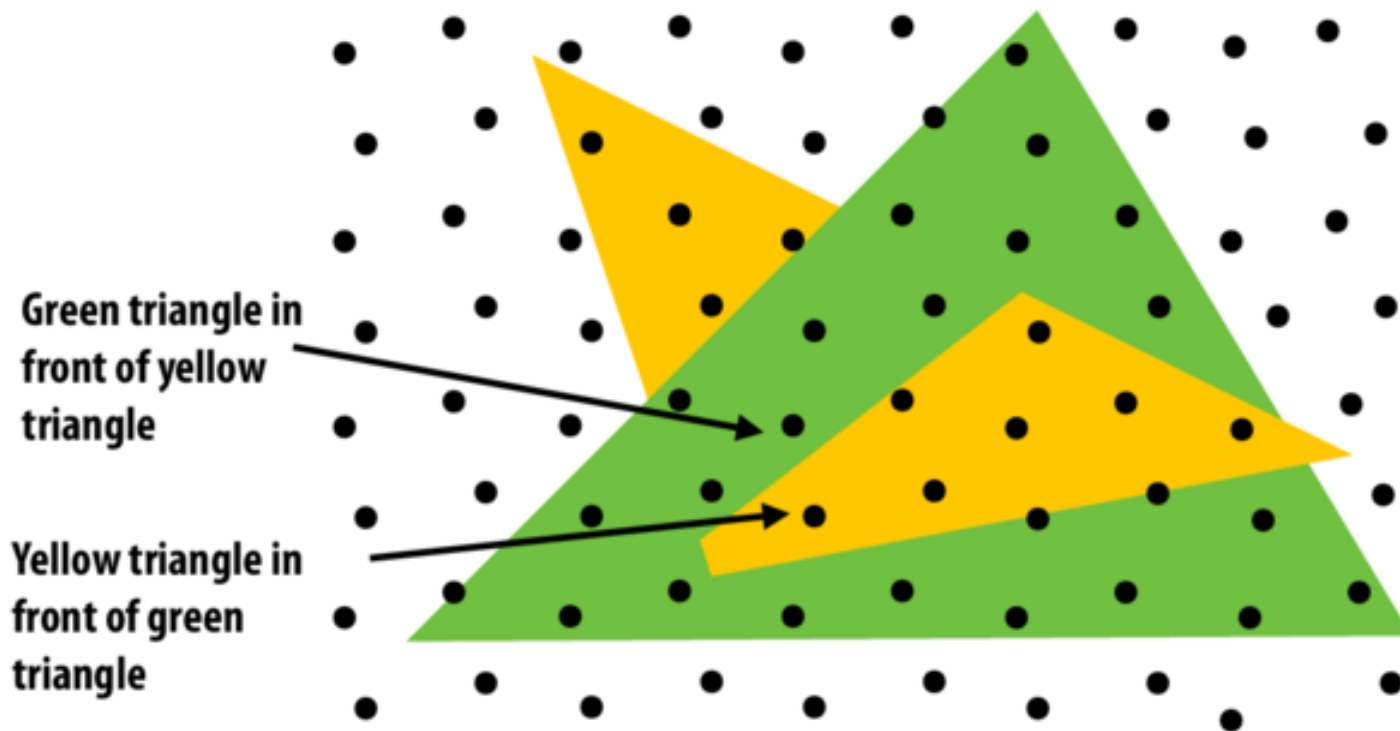


Depth buffer contents

Does depth-buffer algorithm handle interpenetrating surfaces?

Of course!

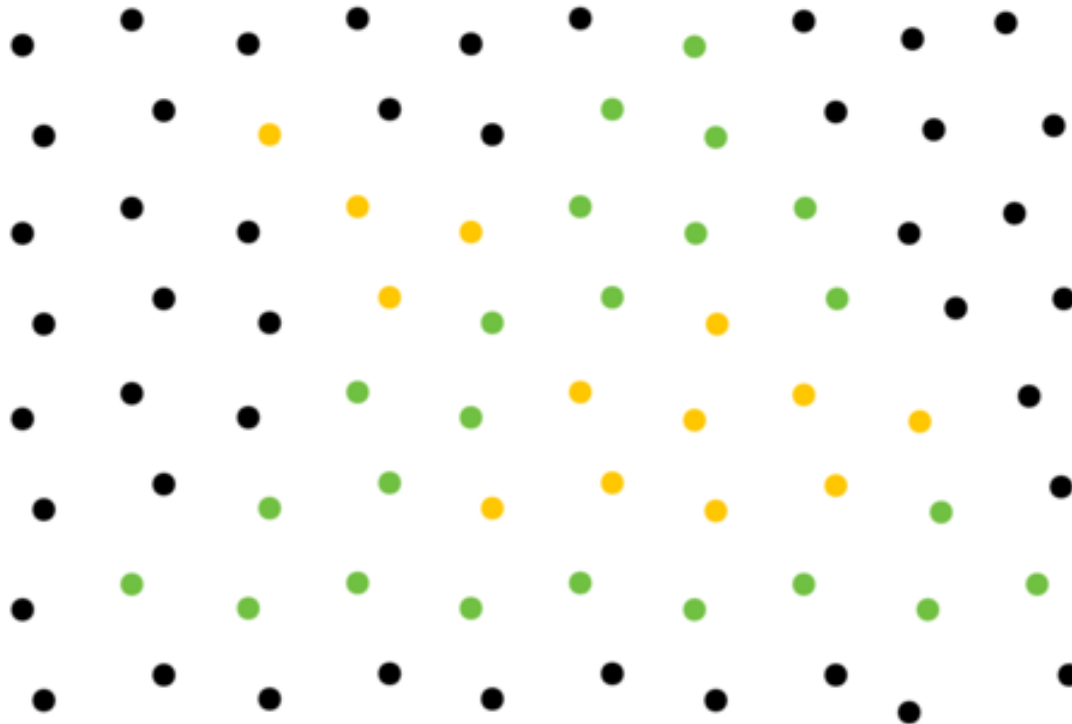
Occlusion test is based on depth of triangles at a given sample point. The relative depth of triangles may be different at different sample points.



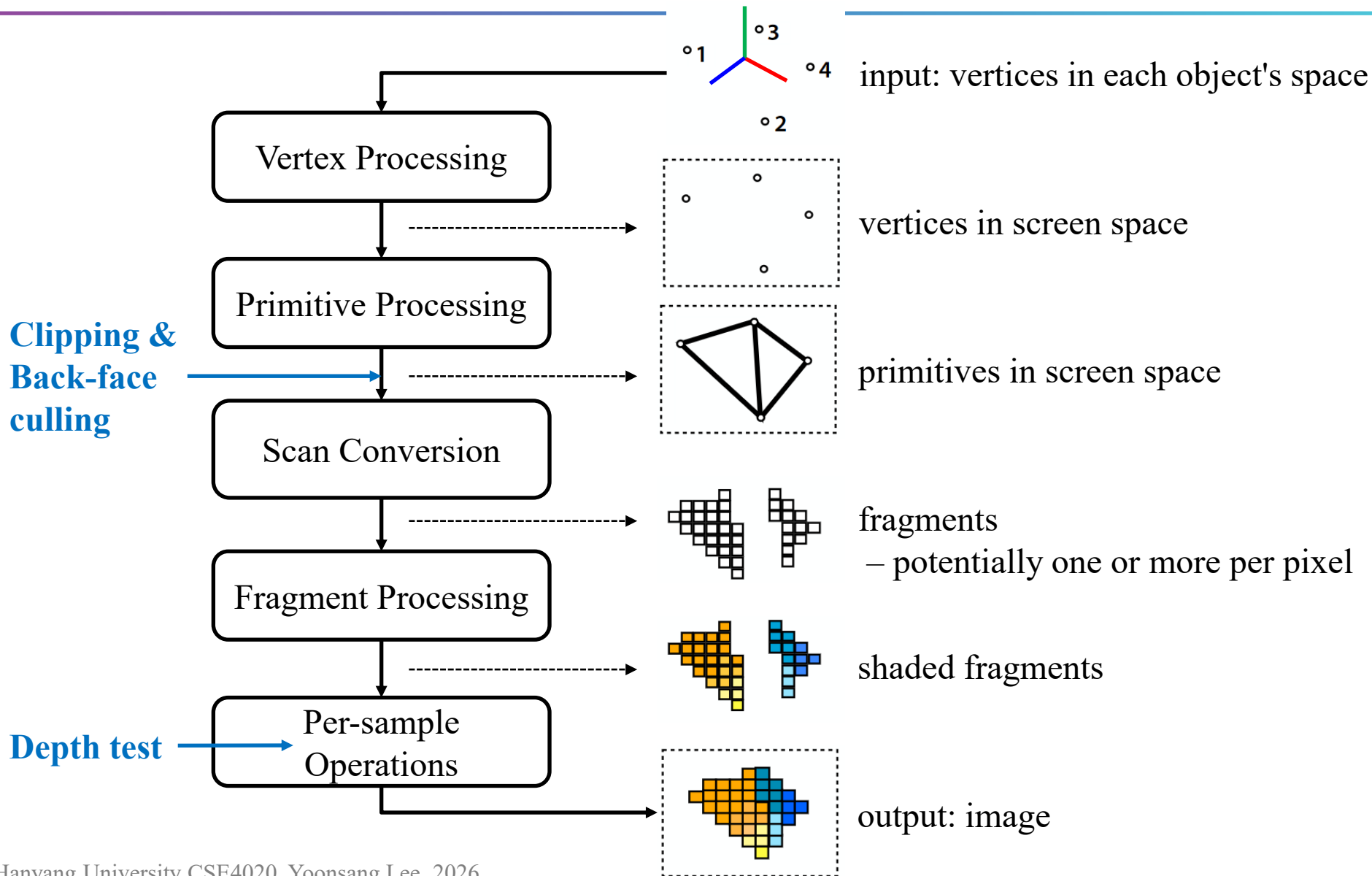
Does depth-buffer algorithm handle interpenetrating surfaces?

Of course!

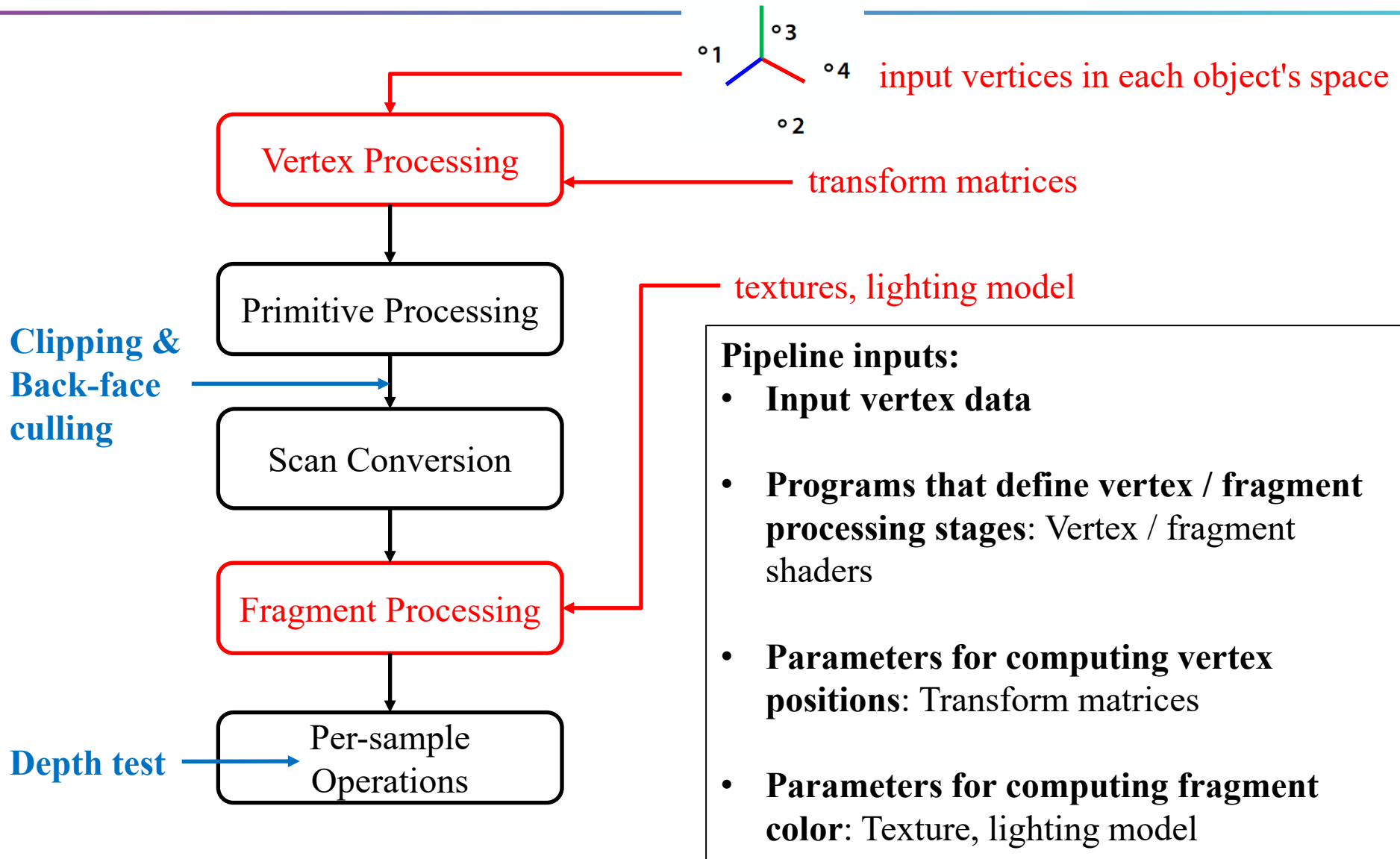
Occlusion test is based on depth of triangles at a given sample point. The relative depth of triangles may be different at different sample points.



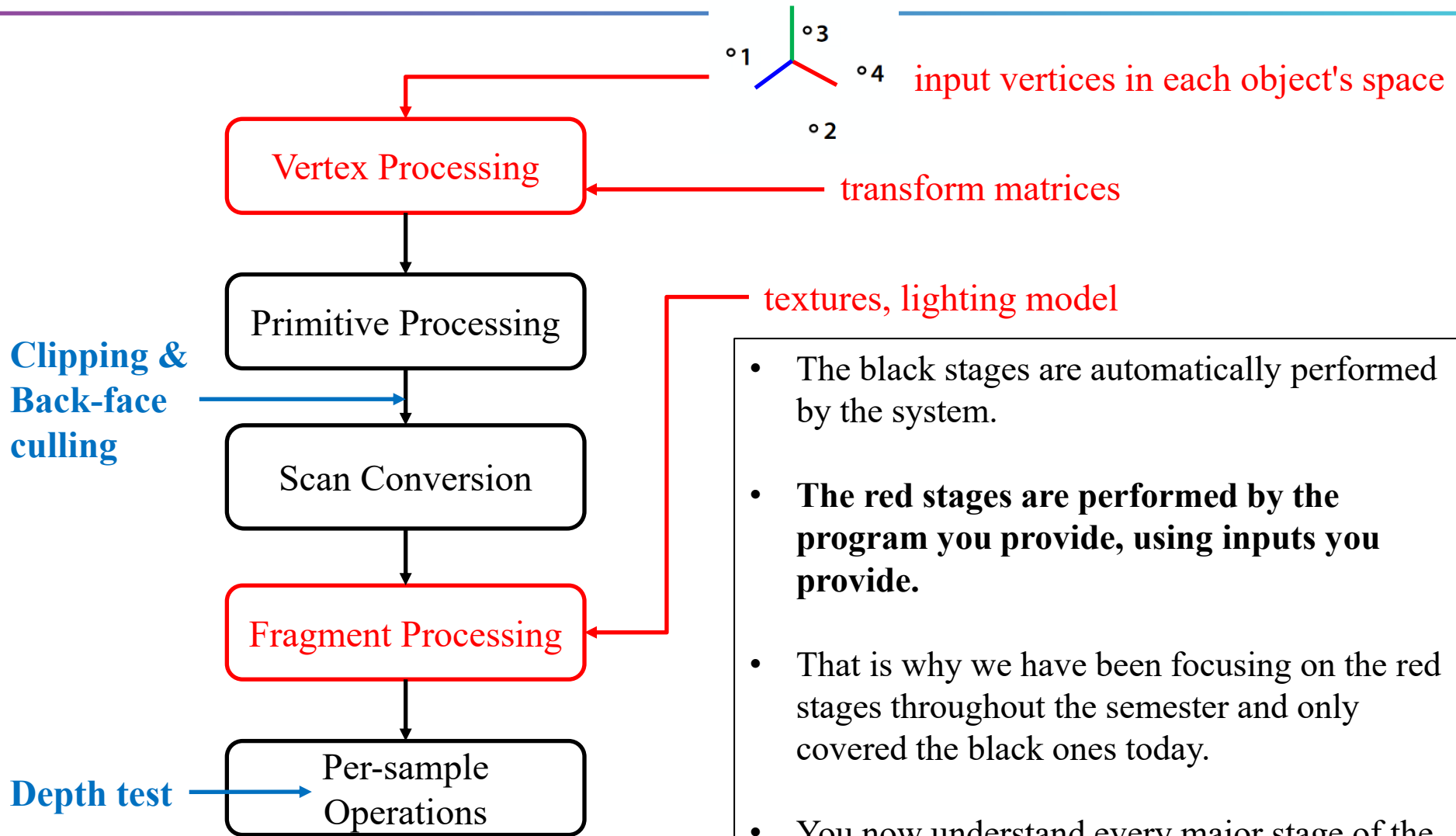
Rendering Pipeline Again



Pipeline Input



Pipeline Input



- The black stages are automatically performed by the system.
- **The red stages are performed by the program you provide, using inputs you provide.**
- That is why we have been focusing on the red stages throughout the semester and only covered the black ones today.
- You now understand every major stage of the rendering pipeline and have the knowledge to build a basic software renderer from scratch.

Lab Session

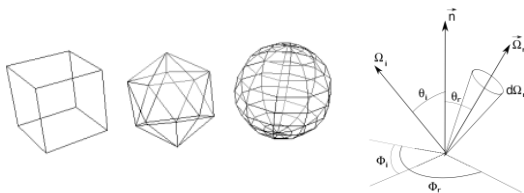
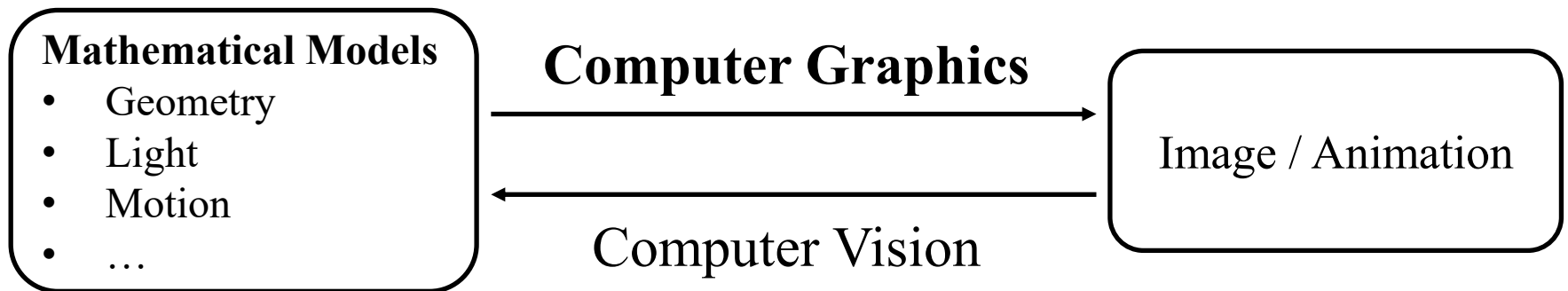
- We don't have a lab today!

Course Wrap-up

Do you remember?

- What is Computer Graphics?

- Computer graphics is the study of creating, manipulating, and displaying visual content.
- Core idea:
Computer graphics generates **images** from **mathematical models of a scene**.



Questions about Computer Graphics

- To achieve this, we need to be able to answer the following questions:
- (General)
- How can we represent the movement, placement, shape, and appearance of objects?
- (Rasterization-specific)
- How do we project 3D objects onto a 2D screen?
- How is the entire rendering process carried out?

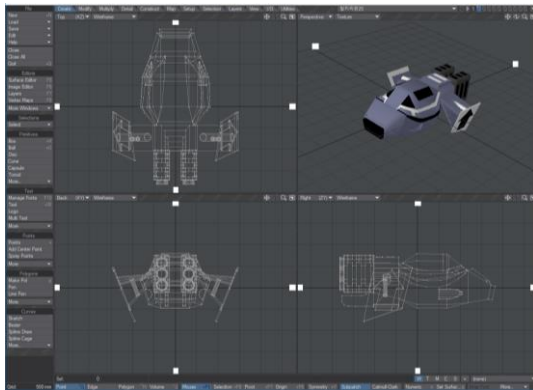
Movement & placement	3 - Transformations 4 - Affine Space Frame Matrix 7 - Hierarchical Modeling, Mesh 9 - Orientation & Rotation 10 - Character Animation 11 - Curves
Mapping to 2D screen	5 - Vertex Processing 1 6 - Vertex Processing 2
Shape	7 - Hierarchical Modeling, Mesh 11 - Curves
Appearance	8 - Lighting 12 - More Lighting, Texture
Rendering Pipeline	2 - Rendering Basics 13 - Scan Conversion, Visibility

How Was It for You?

- Please share your thoughts on the Slido.com Q&A.
- This course may have been challenging,
- but I hope you had **fun** and felt a **sense of reward**—especially during the lab checks and the project.
- I also understand that some of you might feel discouraged.
 - maybe because of low exam scores,
 - or because your project was not successful.

It's Okay to Feel That Way

- What really matters is not your grade, but whether this subject sparked your curiosity.
- If you've **had more fun** in this course than other courses, then you already have **the potential** to do exciting research in computer graphics!
- In computer graphics, **passion** drives creativity and progress — it's more important than anything else.
 - That was the starting point for me on this path.



Take Time to Discover What Feels Right for You

- I truly hope that each of you takes time to reflect—
 - on what you enjoy, what you're curious about, and what kind of work brings you energy.
- If this course — or perhaps another one — can support you even a little along that journey, then it will have been more than worthwhile.

If You Enjoyed This Course, What's Next?

- If you're interested in computer graphics research, especially in areas such as character animation or robotics,
- Feel free to email me if you'd like to get involved in research in these areas:
yoonsanglee@hanyang.ac.kr

Characteristics of Computer Graphics Research

- Requires a lot of programming, but you don't have to be an expert to get started.
 - If you enjoy solving problems and are willing to learn, you'll do just fine.
- Research in this field is **highly visual**.
 - Ideas are always demonstrated through images, animations, and videos, which makes your work easy to share and exciting to watch.
- It combines mathematics, physics, and creativity.
 - We build models and algorithms that create or understand visual worlds.

Computer Graphics and Robotics Lab.

- Website:

<https://cgrhyu.github.io/>

- Youtube channel:

<https://www.youtube.com/@cgrlab>

- Feel free to visit these sites anytime and take a look around.

One Last Thing...

- I hope this course has been meaningful and helpful
— not only in learning computer graphics,
but also in finding what drives you,
what matters to you,
and maybe even becoming a small turning point in
your twenties.

**Thanks for
being a great
class!**

