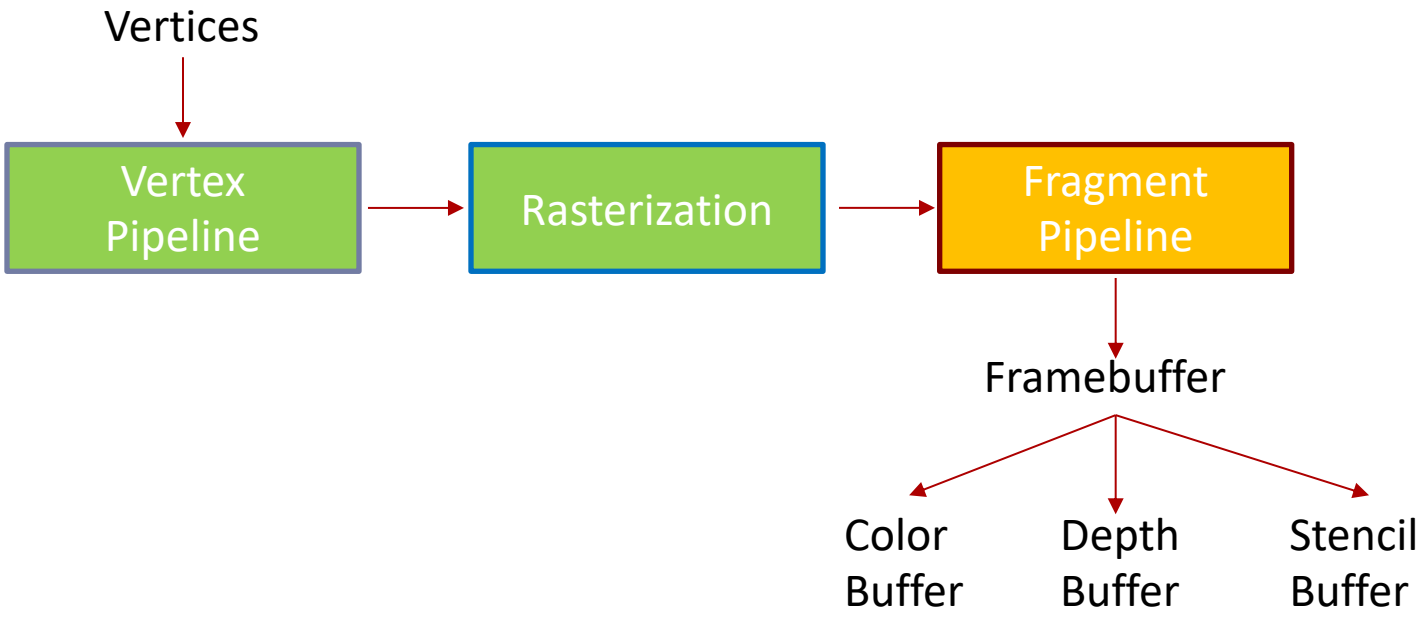
CENG – 477
Introduction to Computer Graphics

Fragment Processing

Fragment Processing

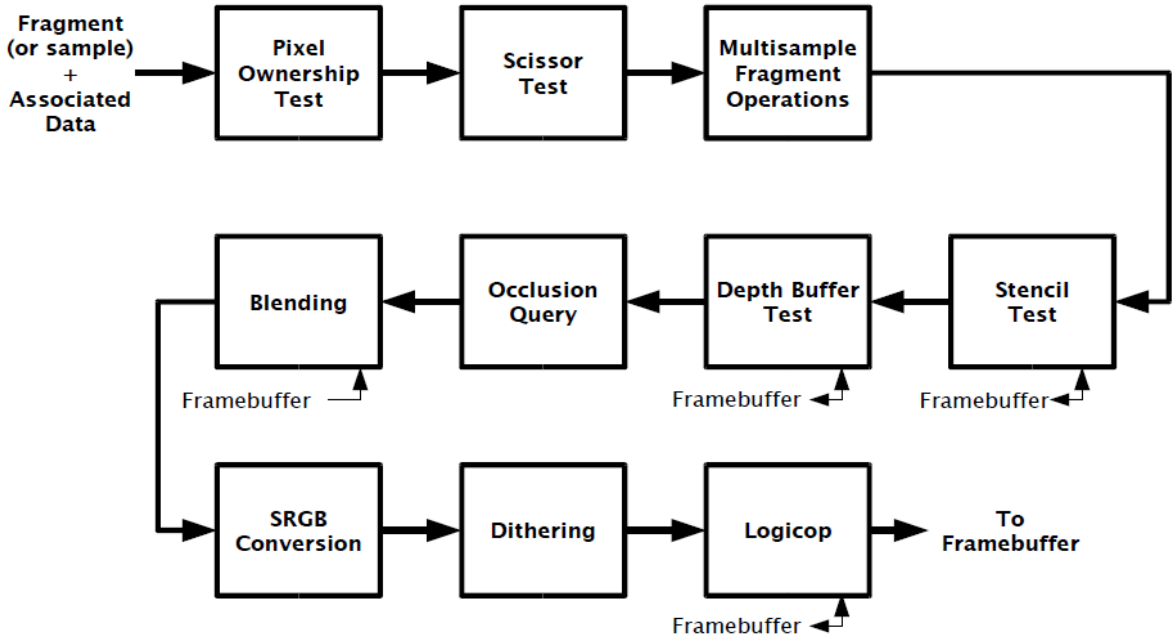
- The previous stages of the pipeline (up to rasterization) is generally known as the vertex pipeline
- Rasterization creates a set of fragments that make up the interior region of the primitive
- The rest of the pipeline which operates on these fragments is called the fragment pipeline
- Fragment pipeline is comprised of several operations
- The end result of fragment processing is the update of corresponding locations in the framebuffer

Fragment Processing



Fragment Processing

- Fragment pipeline is comprised of many stages:
  - Following is OpenGL's handling of the fragment pipeline
  - Different renderers may implement a different set of stages

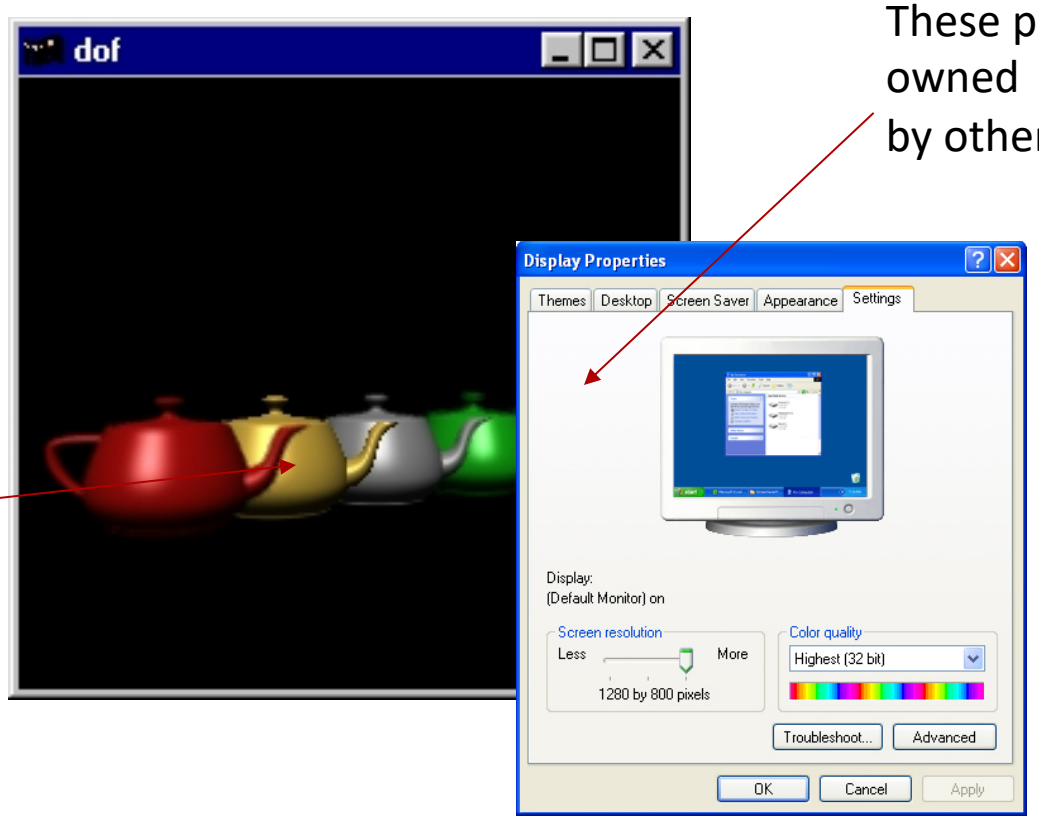


See: https://www.khronos.org/opengl/wiki/Per-Sample_Processing

Pixel Ownership Test

framebuffer is the entire monitor screen

- Does the renderer (e.g. OpenGL) has the ownership of the current framebuffer pixel?

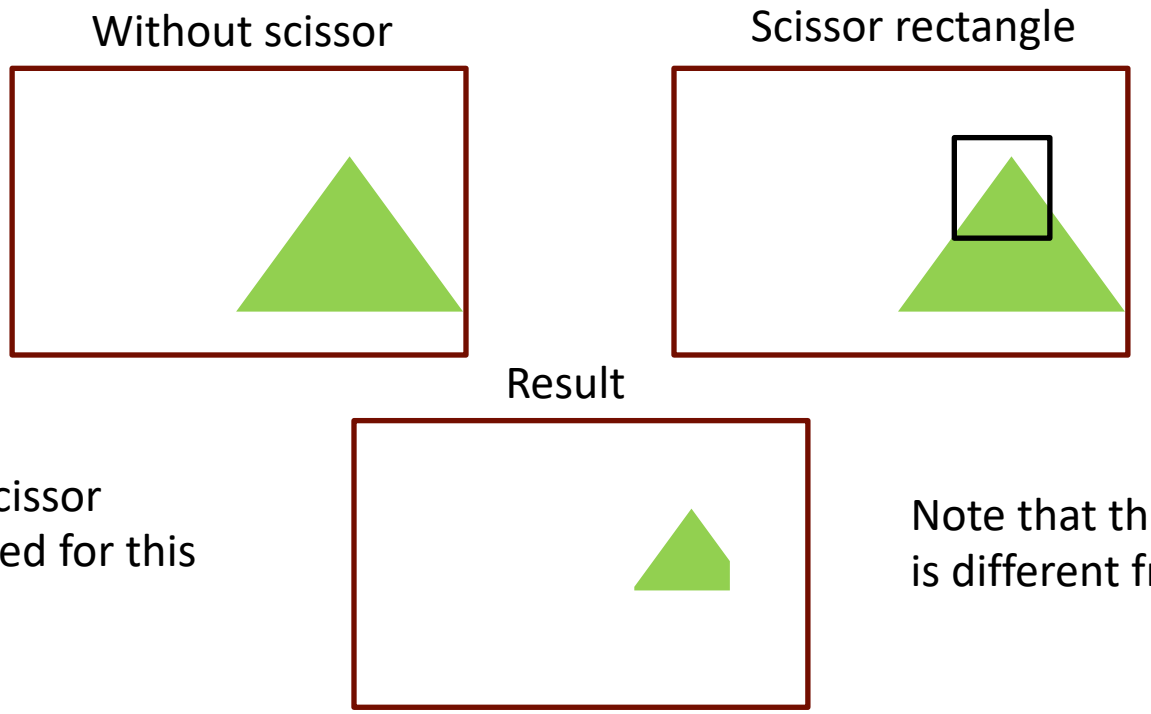


These pixels are owned by other windows

These pixels are owned by OpenGL

Scissor Test

- Scissor test is a per-fragment operation that discards fragments outside a certain rectangular region

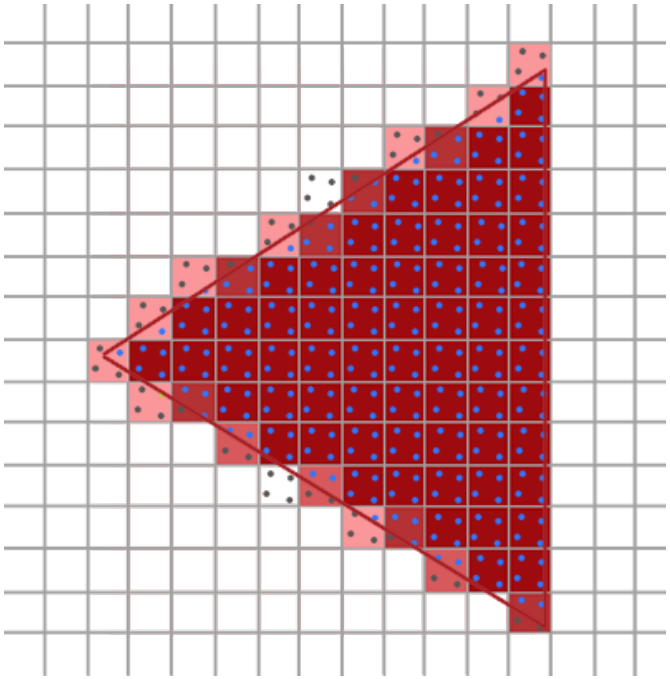


In OpenGL, glScissor command is used for this purpose

Note that this operation is different from clipping

Multi-Sample Operations

- Each framebuffer pixel may contain multiple samples. This allows a smoother border of primitives at the cost of extra processing and memory



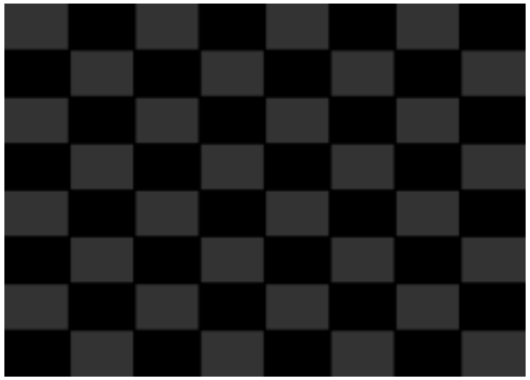
https://learnopengl.com/Advanced-OpenGL/Anti-Aliasing

Stencil Test

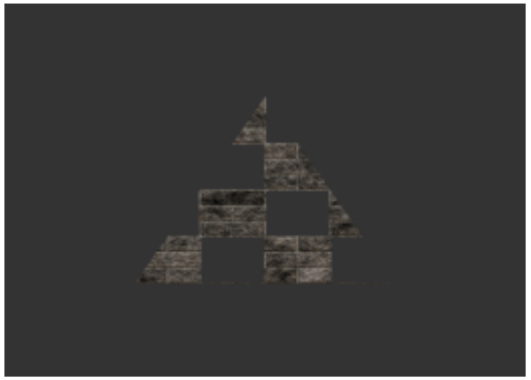
- While scissor test can be used to mask out rectangles, stencil test can be used to mask arbitrary fragments
- Requires a different buffer known as the stencil buffer

Original

Stencil Buffer



Result

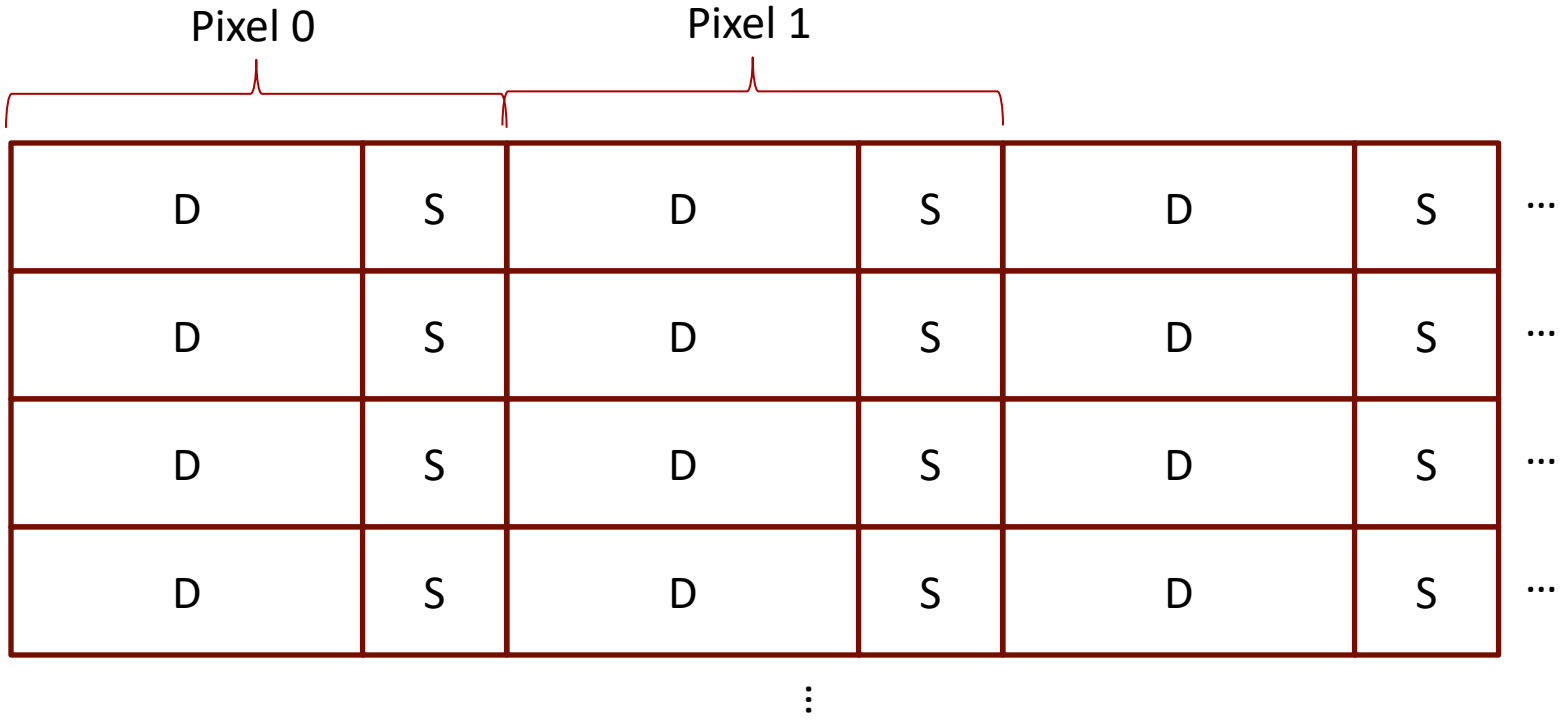


From research.ncl.ac.uk

Stencil Test

normally, 1 bit (boolean) is enough for stencil, but in shadows for example, we count certain things. so 8 bit -> 0-255

- Typically depth and stencil buffers are combined to produce single buffer made of 24-bit depth and 8-bit stencil information for each pixel



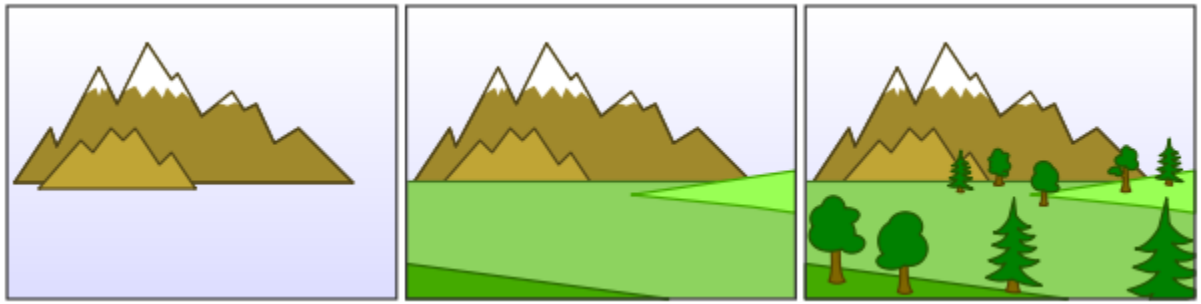
Stencil Test

- Stencil buffer and stencil test can also be used to implement one type of shadowing algorithms (we'll learn this later)

Doom 3

Depth Buffer Test

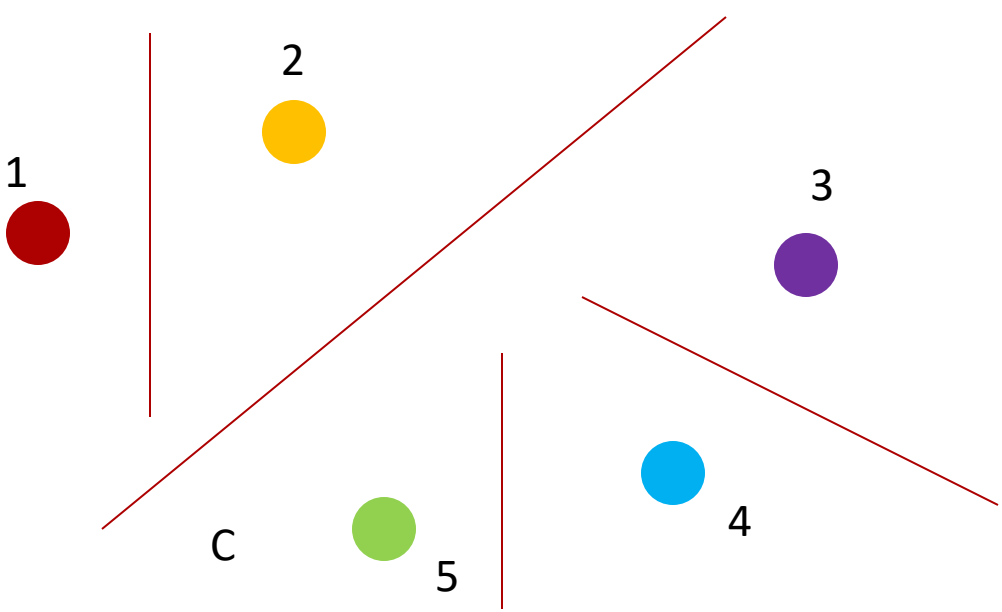
- Among these, the depth buffer test is very important to render primitives in correct order
- Without depth buffer, the programmer must ensure to render primitives in a back to front order
  - Known as painter's algorithm:

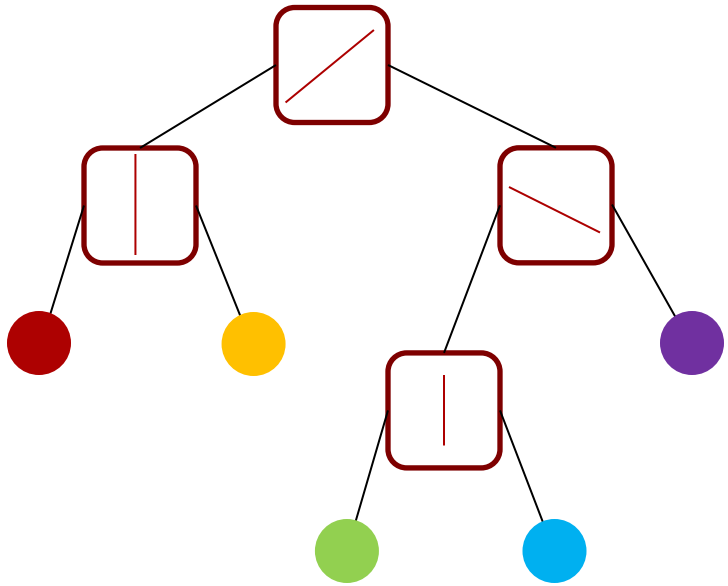


From wikipedia.com

Depth Buffer Test

- Binary space partitioning (BSP) trees may be used for this purpose

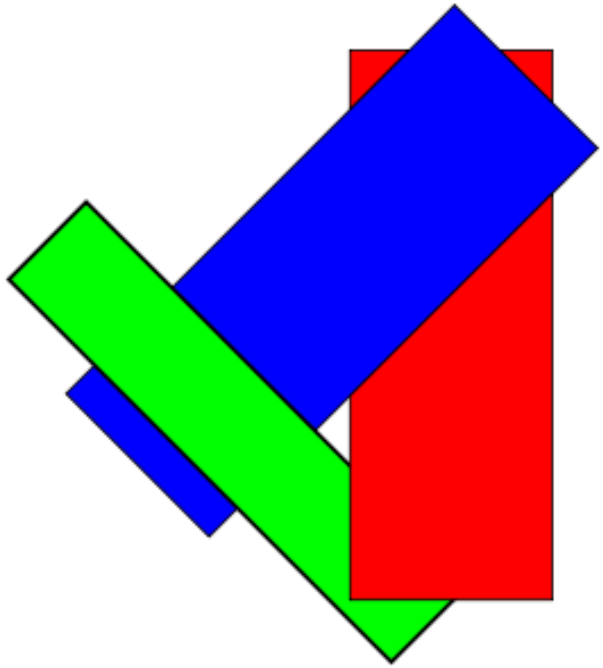




Corresponding BSP tree

Depth Buffer Test

- However, they are costly to generate and may require splitting primitives due to impossible ordering cases:



From wikipedia.com

Depth Buffer Test

- When memory was a very valuable resource, such algorithms were implemented
- Quake3 was one of the main games that used painter's algorithm using BSP trees
- Each game level was stored as a huge BSP tree
  - Read more at: https://www.bluesnews.com/abrash/chap64.shtml

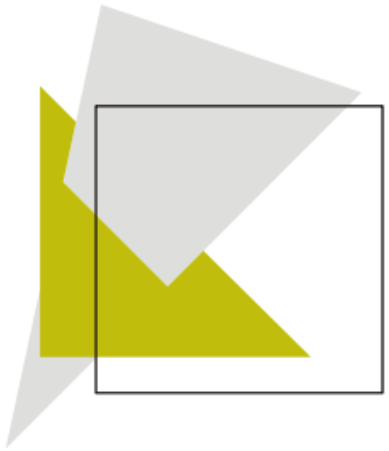
Depth Buffer Test

after transformations etc, z values are normalized to 0,1 and stored here

- **Main Idea:**
  - At each pixel, keep track of the distance to the closest fragment that has been drawn in a separate buffer
  - Discard fragments that are further away than that distance
  - Otherwise, **draw** the fragment and **update** the z-buffer value with the z value of that fragment
- Requires an extra memory region, called the depth buffer, to implement this solution
- At the beginning of every frame, the depth buffer is reset to infinity (1.0f in practice if the depth range is [0.0f,1.0f])
- Depth buffer is also known as z-buffer

Example

color buffer is also updated (not shown here)

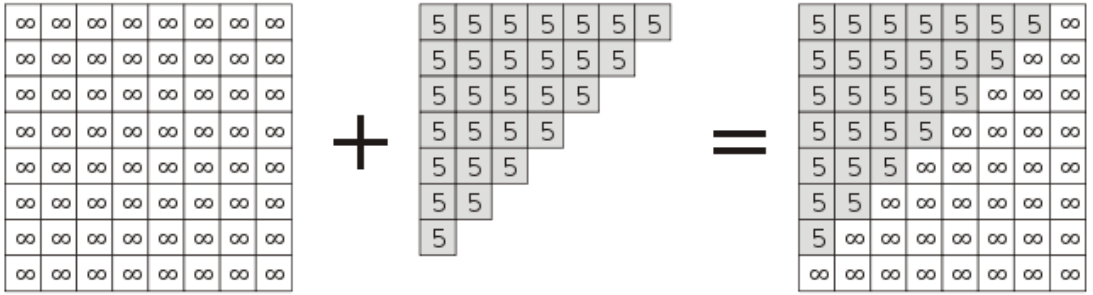


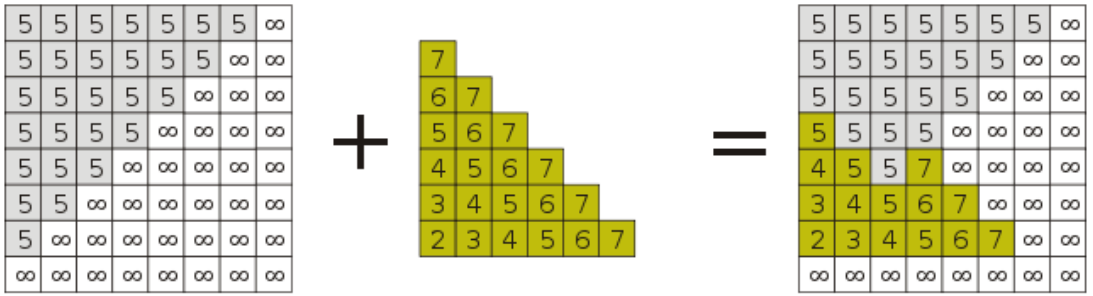
wikipedia.com

Initial state of depth buffer

z-values of the first triangle

Resulting depth buffer





z-values of the second triangle

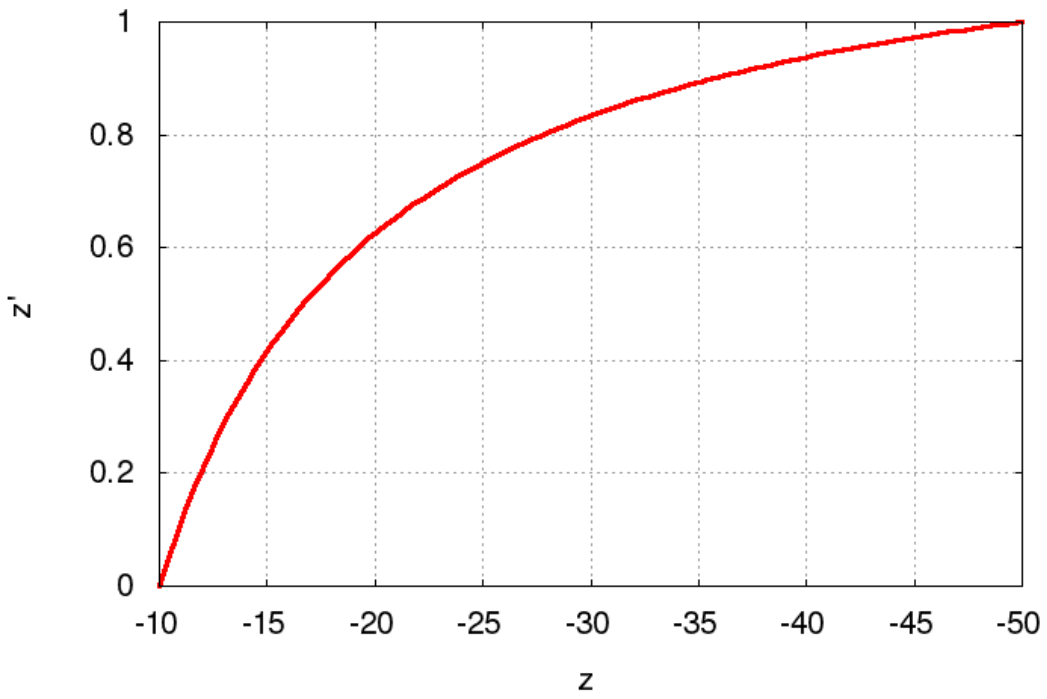
Resulting depth buffer

Depth Range

- The range of values written to the depth buffer can generally be controlled by the programmer
- In OpenGL, the command **`glDepthRange(zMin, zMax)`** is used
- The default depth range is [0, 1]
- The z-value in the canonical viewing volume (CVV), which is in range [-1, 1] is scaled to this range during the viewport transform
- **`glDepthRange`** is to the z-values what **`glViewport(x, y, width, height)`** is to the x- and y-values

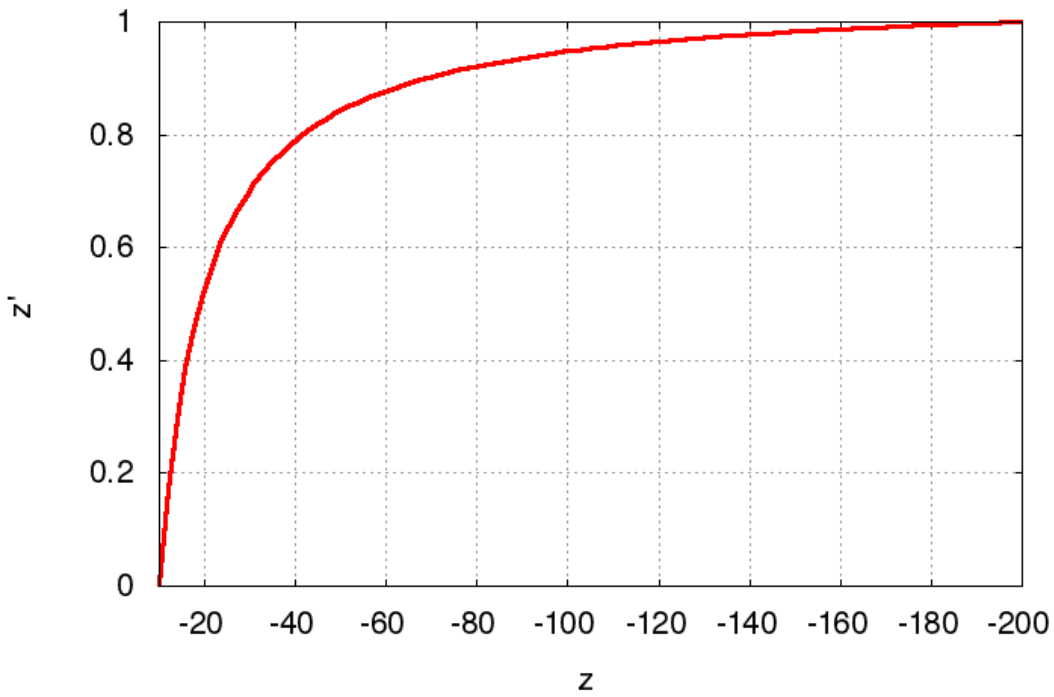
Z-Fighting

- Remember that the z-values get compressed to [0, 1] range from the [-n:-f] range after projection and viewport transforms
- Observe



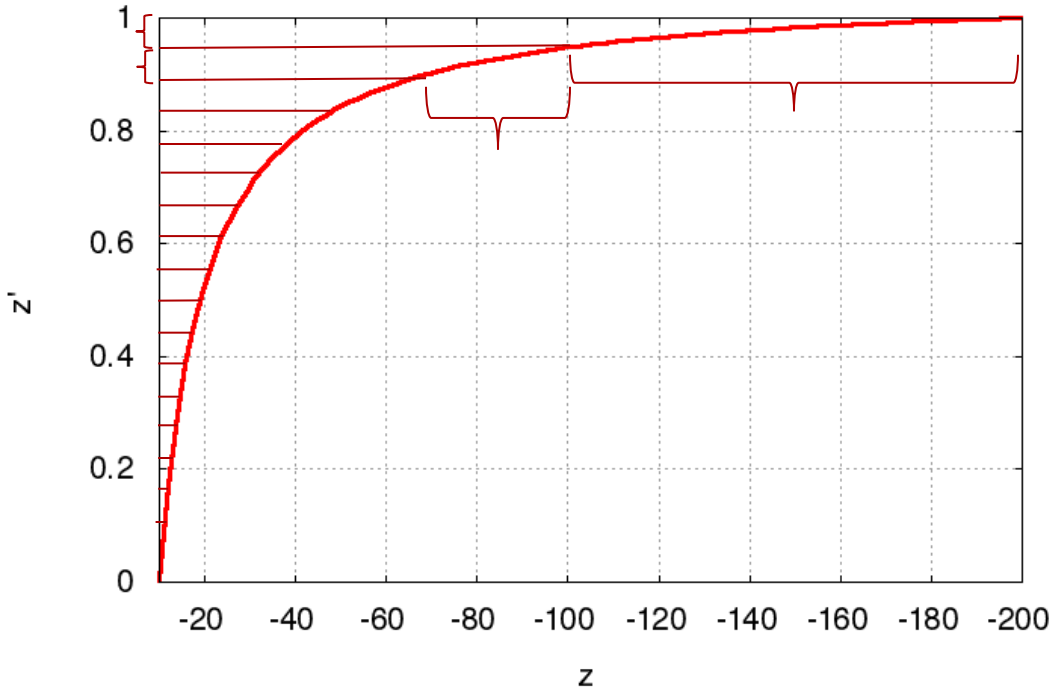
Z-Fighting

- Remember that the z-values get compressed to [0, 1] range from the [-n:-f] range after projection and viewport transforms
- Observe



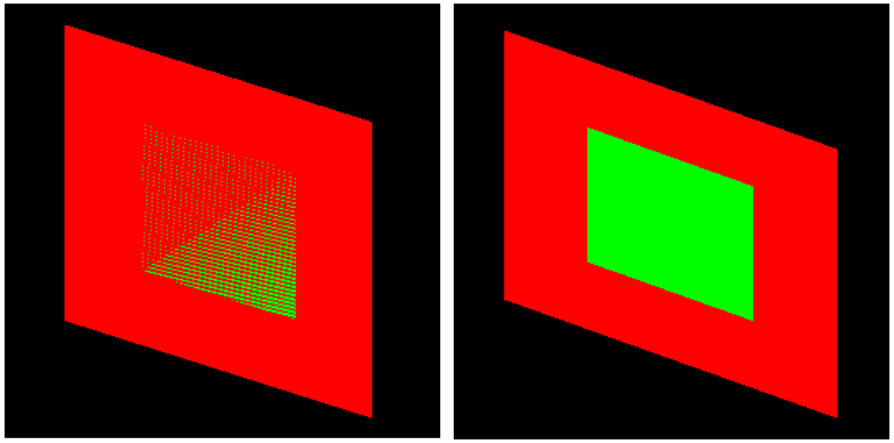
Z-Fighting

- With a limited precision depth buffer, fragments that are close in depth may get mapped to the same z-value

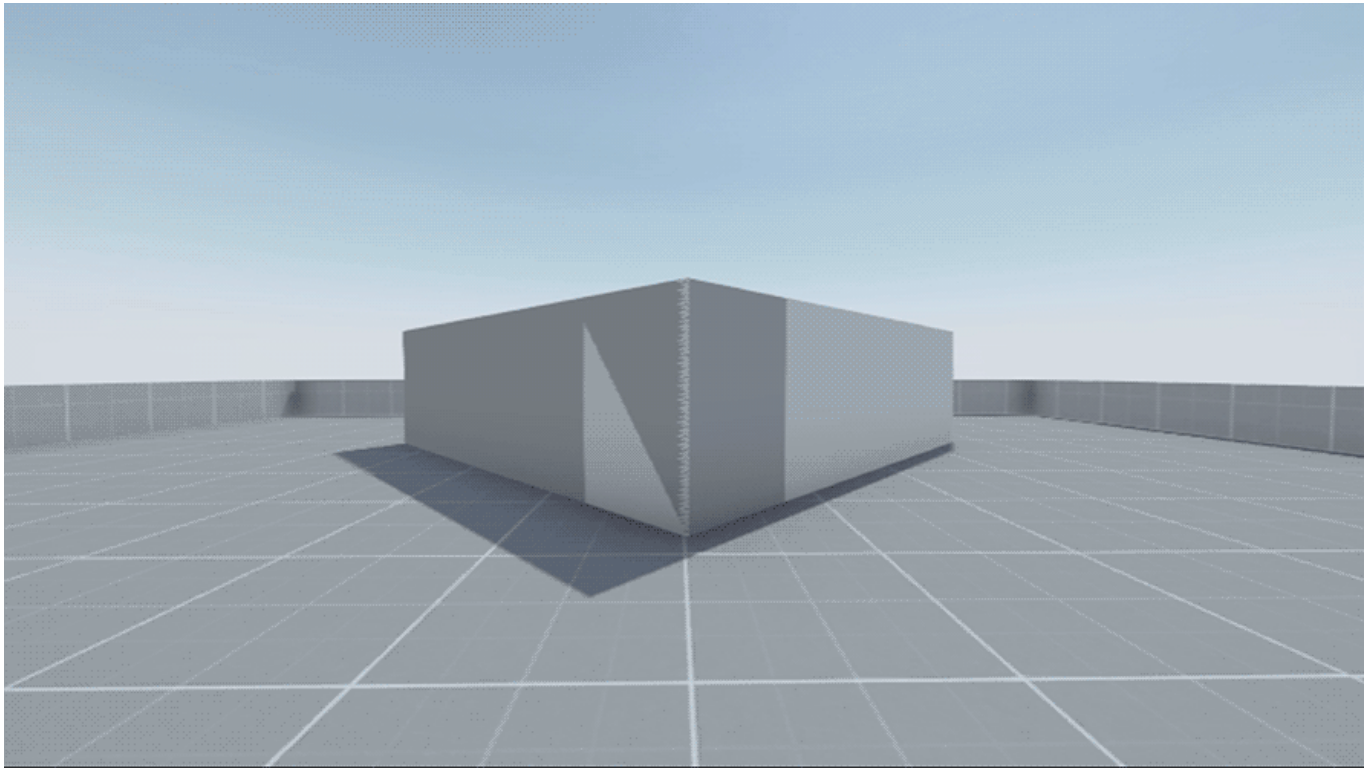


Z-Fighting

- The compression is more severe for with larger depth range
- This may cause a problem known as z-fighting:
  - Objects with originally different (but close) z-values get mapped to the same final z-value (due to limited precision) making it impossible to distinguish which one is in front and which one is behind



Z-Fighting



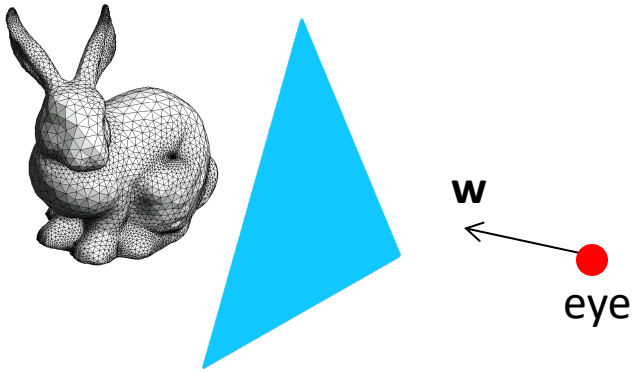
http://wiki.reflexfiles.com/

Z-Fighting

- To avoid z-fighting, the separation of the far and near planes should be kept as small as possible for keeping the compression less severe
- Alternatively, a floating point depth buffer can used
  - Unavailable in older hardware
  - Supported in all modern GPUs
- Finally, the command **`glPolygonOffset`** can be used to push and pull polygons a little to avoid z-fighting

Occlusion Query

- The removal of geometry that is within the view volume but is occluded by other geometry closer to the camera:



Bunny is occluded
by the blue triangle

- OpenGL supports occlusion queries to assist the user in occlusion culling
- By a fast rendering pass, it counts how many pixels of the tested object will be rendered
- This is commonly used in games

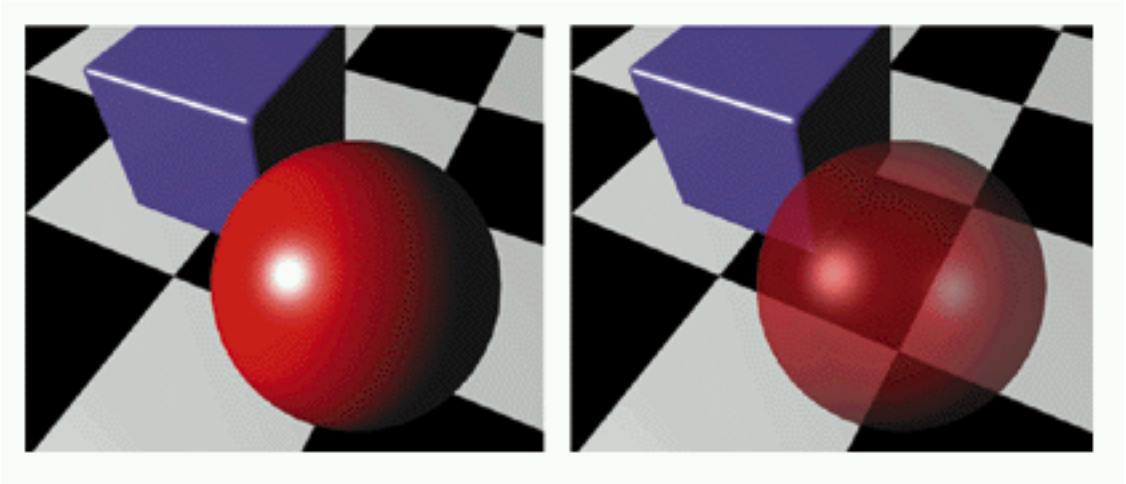
Occlusion Culling in OpenGL

1. Create a query.
2. Disable rendering to screen (set the color mask of all channels to False).
3. Disable writing to depth buffer (just test against, but don't update, the depth buffer).
4. Issue query begin (which resets the counter of visible pixels).
5. "Render" the object's bounding box (it'll only do depth testing; pixels that pass depth testing will not be rendered on-screen because rendering and depth writing were disabled).
6. End query (stop counting visible pixels).
7. Enable rendering to screen.
8. Enable depth writing (if required).
9. Get query result (the number of "visible" pixels).
10. If the number of visible pixels is greater than 0 (or some threshold),
    - Render the complete object.

For details see: http://http.developer.nvidia.com/GPUGems/gpugems_ch29.html

Alpha Blending

- Alpha blending is another fragment operation in which new objects can be blended with the existing contents of the color buffer for a variety of effects



Without blending With blending

sRGB Conversion

- The fragment values are converted to the sRGB color space before being written to the framebuffer. Serves the purpose of gamma correction

Summary

- At the end of the pipeline, input vertices with connectivity information end up populating certain regions of the framebuffer
- This pipeline can be implemented on the software (CPU), hardware (GPU) or both (CPU + GPU)