Computer Graphics 1

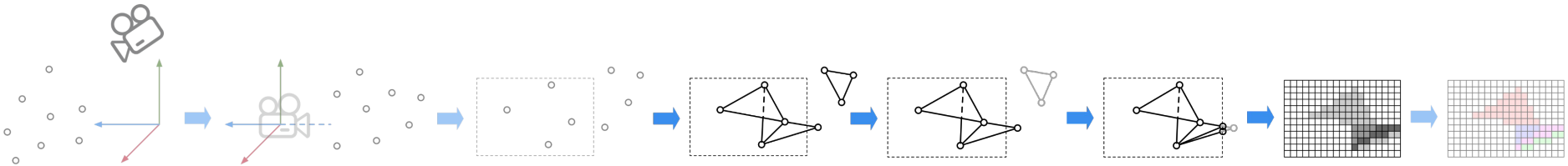
5 Rasterization

Summer Semester 2022

Ludwig-Maximilians-Universität München

Tutorial 5: Rasterization

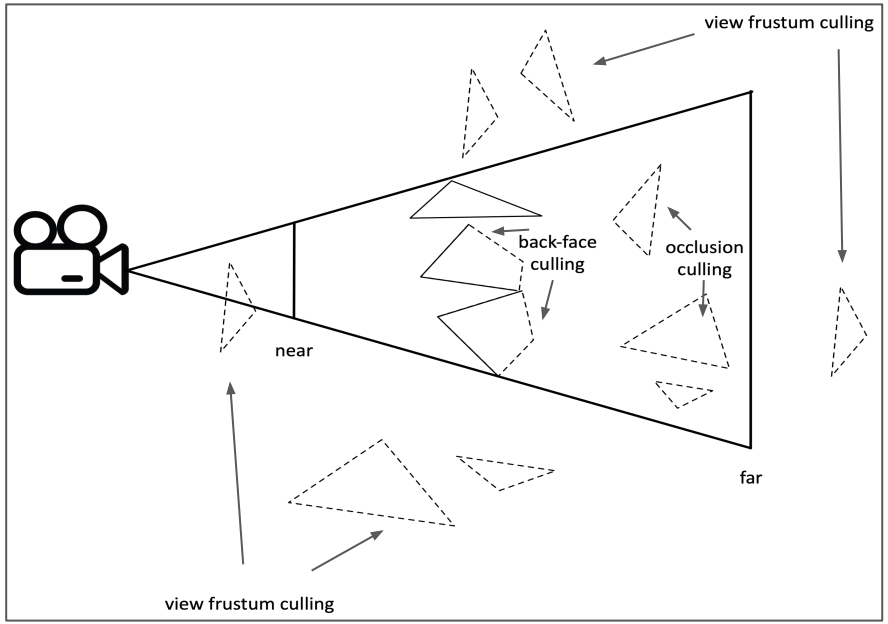
- Culling
  - Different Types of Culling
  - AABB and View Frustum Culling
- Screen-space Buffers
  - Frame and Depth Buffer
- Drawing and Sampling
  - Bresenham and Scan Line Algorithms
- Modern Rasterization Rendering Pipeline
  - Shaders



Types of Culling

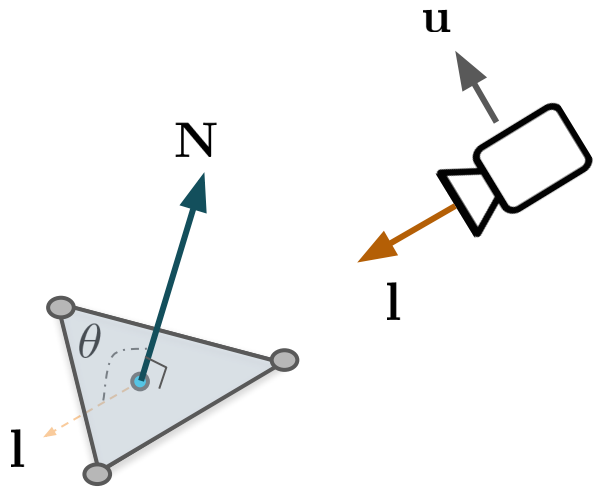
- Backface culling: do not render back faces
- View frustum culling: do not render objects outside of the view frustum
- Occlusion culling: do not render objects behind visible objects

What do we need in order to implement them all?



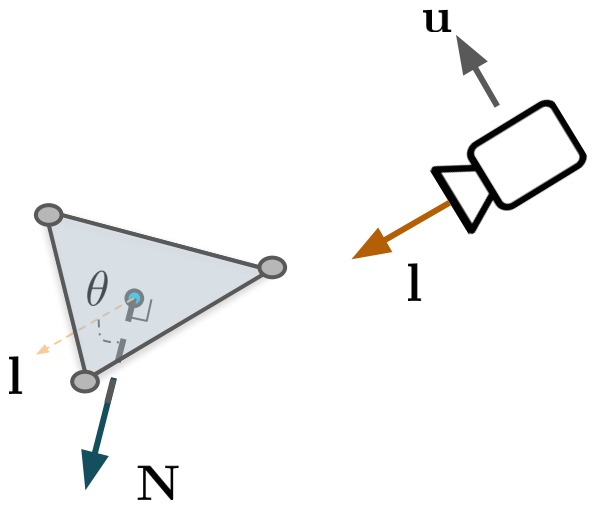
Backface Culling

Backface culling can be easily implemented by calculating the dot product* of face normal and camera look at direction.



$$\cos\theta = \mathbf{l} \cdot \mathbf{N} < 0$$

Frontface



$$\cos\theta = \mathbf{l} \cdot \mathbf{N} \geq 0$$

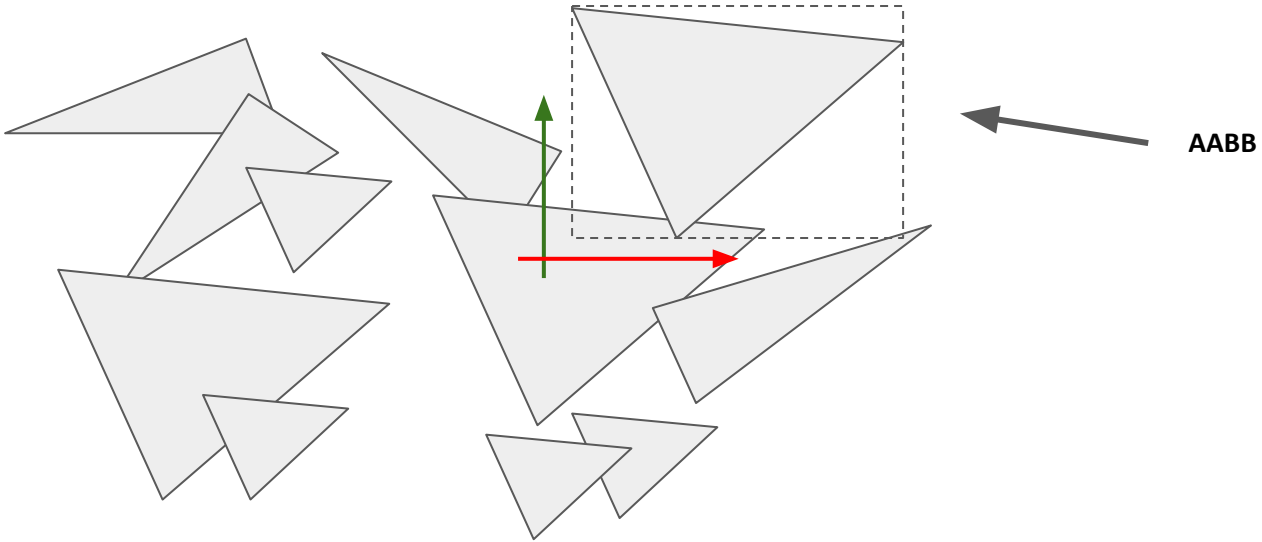
Backface

*Normal and look at vectors are assumed to be unit vectors

Axis-Aligned Bounding Box (AABB)

A *bounding volume* is a volume that encloses a set of objects. A possible (and the easiest to implement) BV is the *axis-aligned* bounding boxes (AABBs).

AABB can be represented via two points (xMin, yMin, zMin) and (xMax, yMax, zMax)



Q: How to compute a minimum AABB for a triangle?

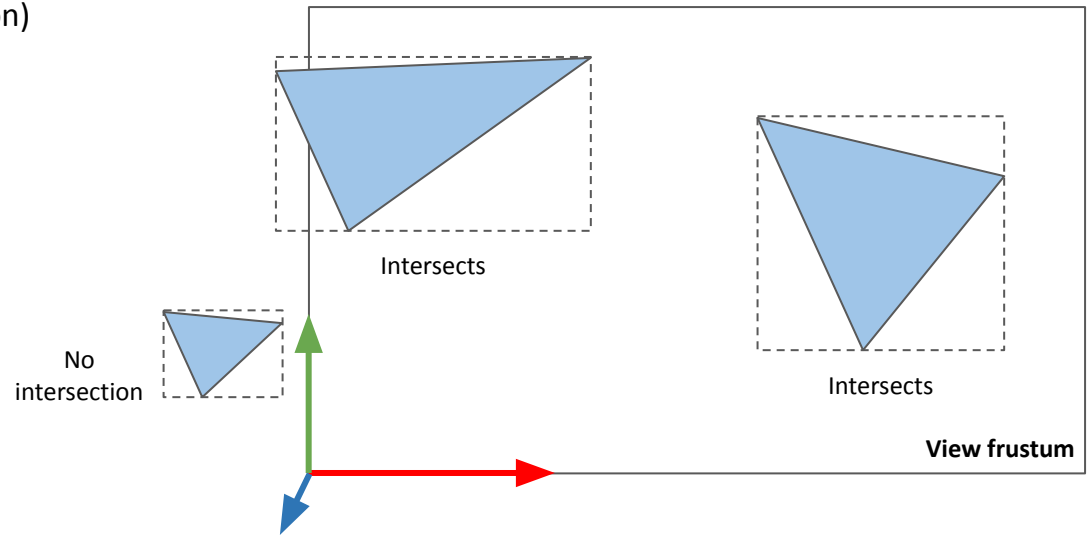
View Frustum Culling

View frustum culling can be easily done via AABB.

Basic idea: If an AABB does not intersect with view frustum (also an AABB), then the object should be culled

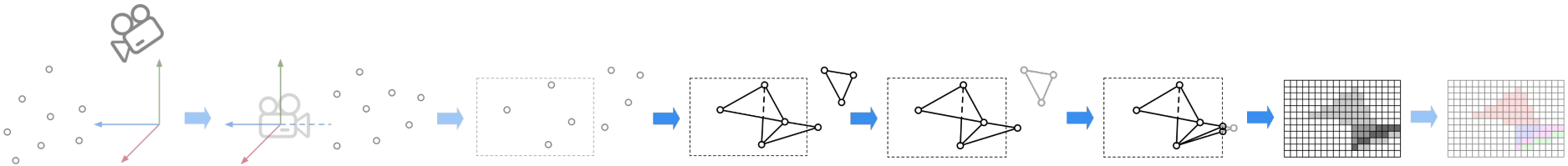
Implementation thinking:

- For a very small scene, one could just loop all existing triangles then test their AABB with the view frustum
- For larger scenes, one can construct a bounding volume hierarchy (BVH) to test if a group triangles intersects with the view frustum (as optimization)



Tutorial 5: Rasterization

- Culling
  - Different Types of Culling
  - AABB and View Frustum Culling
- Screen-space Buffers
  - Frame and Depth Buffer
- Drawing and Sampling
  - Bresenham and Scan Line Algorithms
- Modern Rasterization Rendering Pipeline
  - Shaders

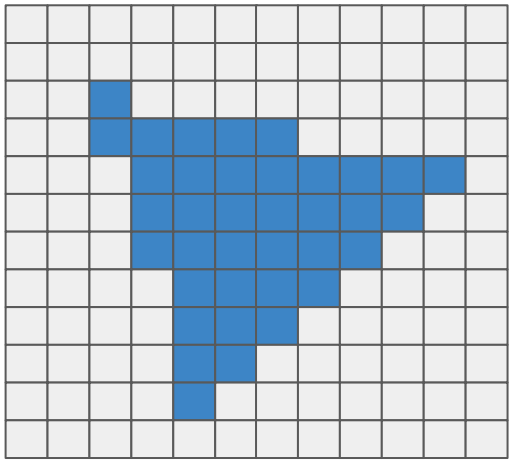


Screen-space Buffers

Screen-space (or Image-space) buffers are arrays for storing rendering information, such as pixel color.

The two most important buffers for a minimum rasterization pipeline are:

- **Frame Buffer**
  - frame buffer stores the pixel color values, which are directly sent to the display
  - frame buffer enables high performance graphics computation:
    - flushing an entire buffer at once is much faster than rendering pixel by pixel
    - enables parallelization by caching multiple frames if we have enough memory
    - ...
- **Depth Buffer** (or Z-buffer)



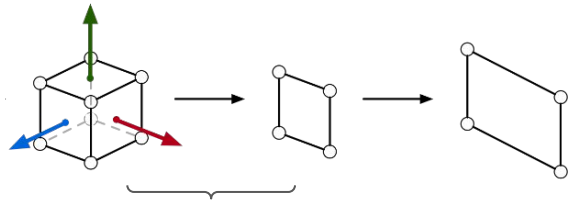
frame buffer

Depth Buffer (Z Buffer)

The Painter's algorithm cannot solve the occlusion issue.

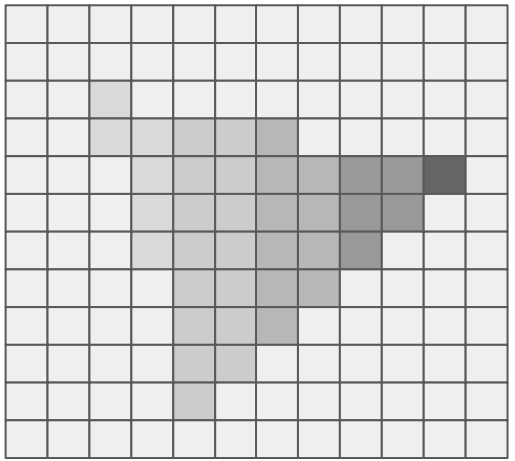
The idea behind a depth buffer for *occlusion culling*:

- Store the current maximum (why?) z-value for each pixel
- Needs an additional buffer for depth values to test fragment visibility



Pseudo code:

```
let frameBuffer: number[][] = new Array<number[]>(width * height).fill([0, 0, 0]); // [r, g, b]
let depthBuffer: number[] = new Array<number>(width * height).fill(-1); // why -1? How about 0, or -2?
triangles.forEach(tri => { // triangles is a list of triangles
 tri.project().fragments.forEach((x: number, y: number, z: number, color: number[]) => {
   // do depth test for each projected fragments (pixels)
   if (z < depthBuffer[x + y*width]) { // is the closest pixel?
     return
   }
   // update frame and depth values
   frameBuffer[x + y*width] = color;
   depthBuffer[x + y*width] = z;
 })
})
```



depth buffer

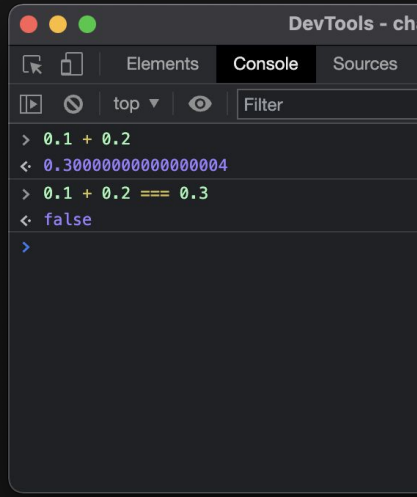
Breakout 1: Experiment Z-fighting Effects

Numeric issues (and floating point numbers) are very tricky to handle in all graphics applications.

Example: try 0.1+0.2 in a browser console (check the reason from Google)

To check if 0.1+0.2 is equal to 0.3, we cannot do:

```
0.1+0.2 === 0.3 // false
```



Instead, as we have seen this approach already:

```
function approxEqual(v1: number, v2: number, epsilon = 1e-7): boolean {
 return Math.abs(v1 - v2) <= epsilon;
}
```

The function compares two numbers approximately with given precision:

```
approxEqual(0.1+0.2, 0.3) // true
```

Breakout 1: Experiment Z-fighting Effects

The most common numeric issue in a graphics application: If two planes have the same depth value, then a Z-buffer might randomly pick a fragment to render because of the depth value precision.

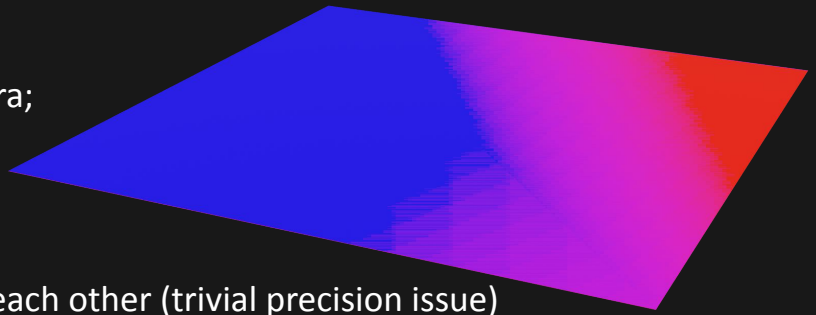
1. Open `plane1.blend` in Blender

Rotate the camera and see if the plane is displayed in one color (blue or red)

2. Open `plane2.blend` in Blender

Do the same as above, then move the object further away from camera;

see what happens when the object is closed to the far plane



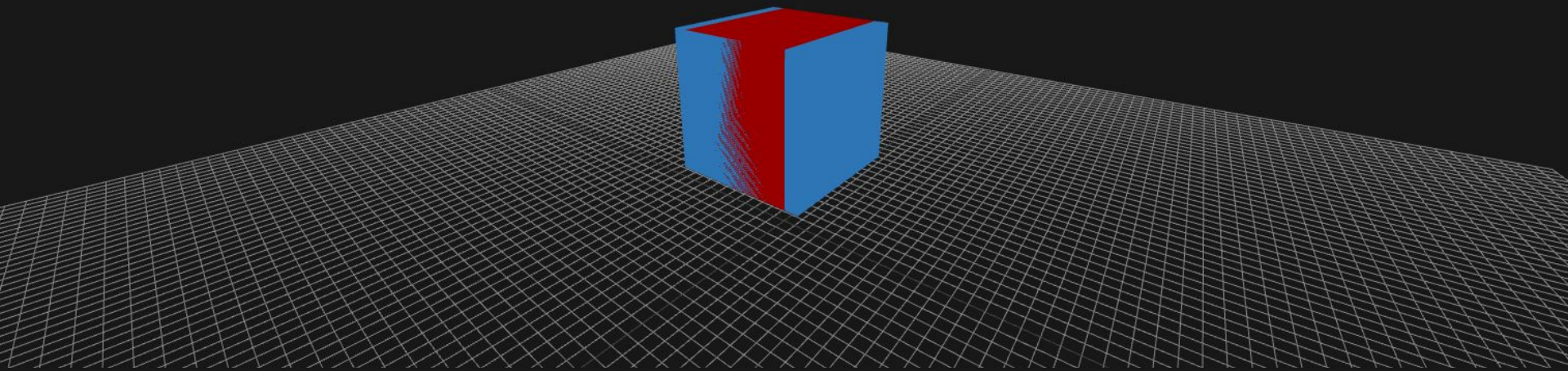
Case 1: two objects are too close, and their z values are fighting with each other (trivial precision issue)

Case 2: two objects are not close, but their z values are fighting when they further away from camera (why?)

Breakout 1: Experiment Z-fighting Effects

Open provided example "zfighting"

1. Rotate the scene and try to display the red cube in pure red
2. Find the TODO comment, uncomment the given parameters, and see if the problem still exist
3. Move the object further away from the camera, what will happen? → 

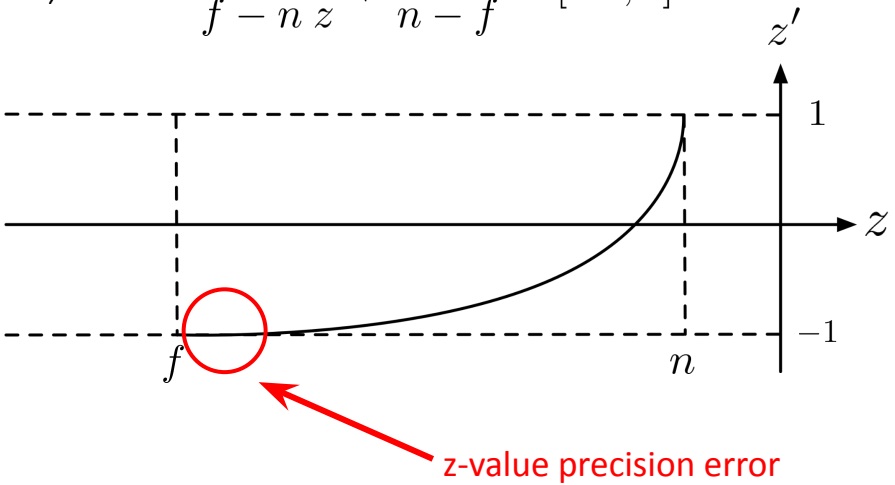


Z-fighting: Far from Viewport (Case 2)

Recall the perspective projection matrix:

$$P' = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \mathbf{T}_{\text{persp}} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} -\frac{1}{\lambda \tan\frac{\theta}{2}} & 0 & 0 & 0 \\ 0 & -\frac{1}{\tan\frac{\theta}{2}} & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-1} & \frac{2nf}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cdots \\ \cdots \\ \frac{n+f}{n-f}z + \frac{2nf}{f-n} \\ z \end{pmatrix} = \begin{pmatrix} \cdots \\ \cdots \\ \frac{n+f}{n-f} + \frac{2nf}{f-n}\frac{1}{z} \\ 1 \end{pmatrix}$$

$$\Rightarrow \quad z' = \frac{2nf}{f-n}\frac{1}{z} + \frac{n+f}{n-f} \in [-1, 1]$$



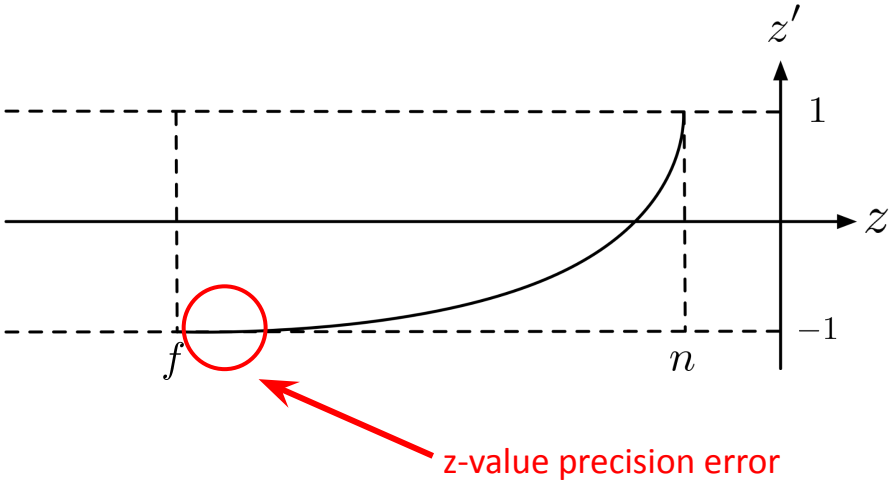
Depth values (after projection) are less accurate when the object is further away from the viewpoint.

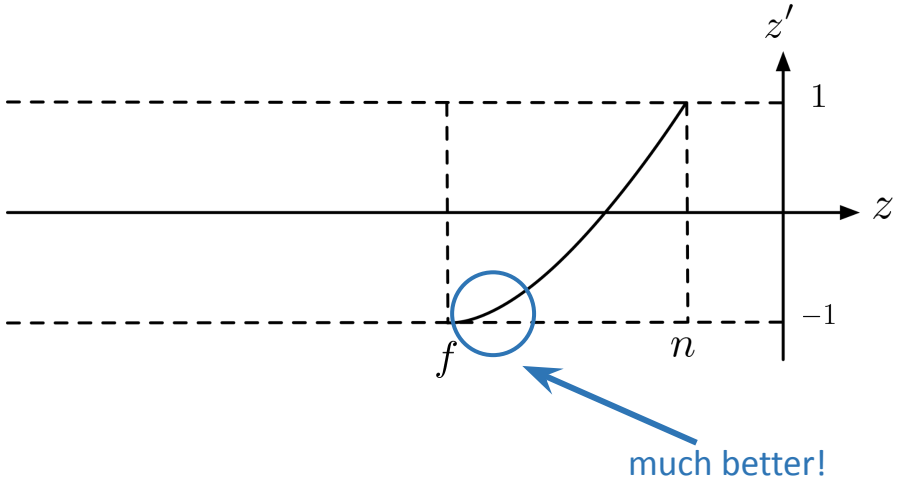
Q: What about orthographic projection?

How to avoid Z-fighting?

1. (Properly) make near and far planes closer
2. Use a higher precision depth buffer or use a logarithmic depth buffer
3. Use a fog effect to avoid objects close to the far plane, and move objects away from each other
4. Use polygonOffset

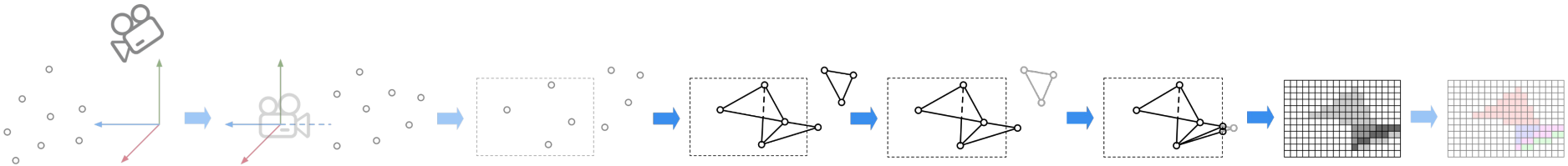
... and more :)





Tutorial 5: Rasterization

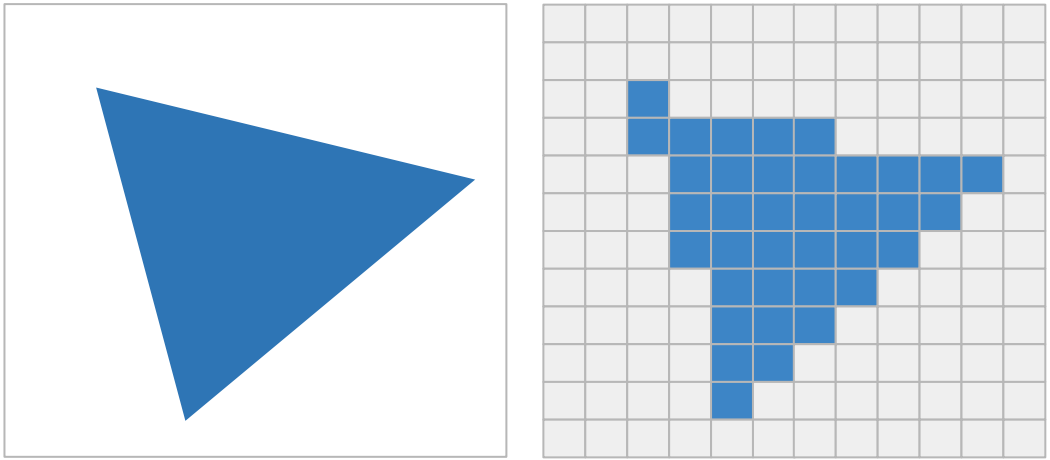
- Culling
  - Different Types of Culling
  - AABB and View Frustum Culling
- Screen-space Buffers
  - Frame and Depth Buffer
- Drawing and Sampling
  - Bresenham and Scan Line Algorithms
- Modern Rasterization Rendering Pipeline
  - Shaders



How to draw a triangle on a rasterized screen?

Rasterizing a triangle consists of two parts:

- Drawing the exterior boundary: *Bresenham algorithm*
- Drawing the interior area: *Scanline algorithm*



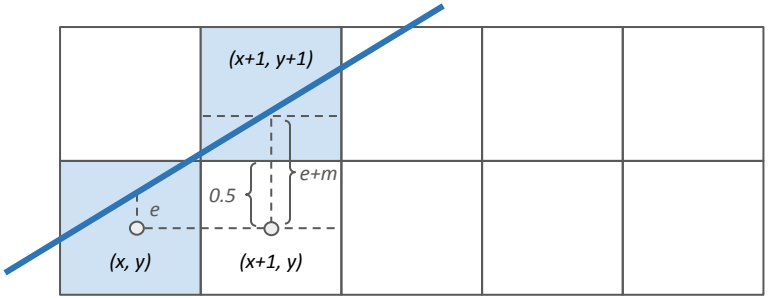
Line Drawing: Bresenham Algorithm

Basic idea: Proceed step by step and accumulate errors up to the ideal line

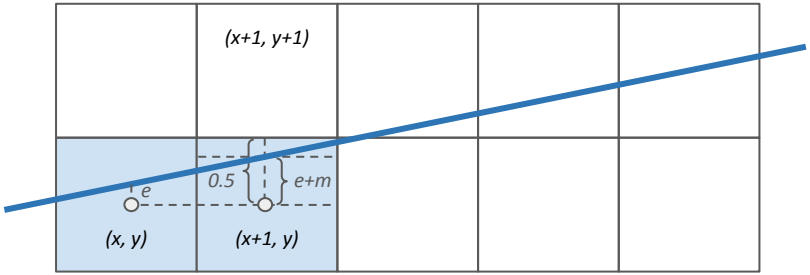
Consider a line with a slope in the range [0, 1], i.e. a line going up less than it goes right.

Having plotted a point at `(x,y)`, the next point on the line can only be `(x+1, y)` or `(x+1, y+1)`

- If `e + m > 0.5` then draw `(x+1, y+1)`
  - The line is closer to `(y+1)` than to `y`



- If `e + m <= 0.5` then draw `(x+1, y)`
  - The line is closer to `y` than to `(y+1)`



J. E. Bresenham. 1965. *Algorithm for computer control of a digital plotter*. IBM Syst. J. 4, 1 (March 1965), 25–30. [PDF]

Draw A Line from (x0, y0) to (x1, y1), 0<= slope <=1

We can reformulate the code for the algorithm introduced on the last slide. With this we reach a fast and computationally easy version of the algorithm. The final version only multiplies with 2, which can be done by left-shift (<<).

```
let e = 0, m = (y1-y0)/(x1-x0)
for (let x = x0, y = y0; x <= x1; ) {
   draw(x, y)
   // how to update x and y?
   if (e+m <= 0.5) {
       x += 1
       e += m
   } else {
       x += 1
       y += 1
       e += m-1
   }
}
```

naive version

```
let e = 0, m = (y1-y0)/(x1-x0)
for (let x = x0, y = y0; x <= x1; x++) {
   draw(x, y)
   // how to update x and y?
   if (e+m <= 0.5) {
   } else {
       y += 1
       e -= 1
   }
   e += m
}
```

```
let e = 0, m = (y1-y0)/(x1-x0)
for (let x = x0, y = y0; x <= x1; x++) {
   draw(x, y)
   // how to update x and y?
   if (e+m > 0.5) {
       y += 1
       e -= 1
   }
   e += m
}
```

```
let e = 0, dy = y1-y0, dx = x1-x0
for (let x = x0, y = y0; x <= x1; x++) {
   draw(x, y)
   // how to update x and y?
   if (2*e*dx+2*dy-dx > 0) {
       y += 1
       e -= 1
   }
   e += dy/dx
}
```

```
let e=0, dy=y1-y0, dx=x1-x0, D=2*dy-x
for (let x = x0, y = y0; x <= x1; x++) {
   draw(x, y)
   // how to update x and y?
   if (2*e*dx+D > 0) {
       y += 1
       e -= 1
   }
   e += dy/dx
}
```

```
let dy = y1-y0, dx = x1-x0, D=2*dy-dx
for (let x = x0, y = y0; x <= x1; x++) {
   draw(x, y)
   // how to update x and y?
   if (D > 0) {
       y += 1
       D -= 2*dx
   }
   D += 2*dy
}
```

final version

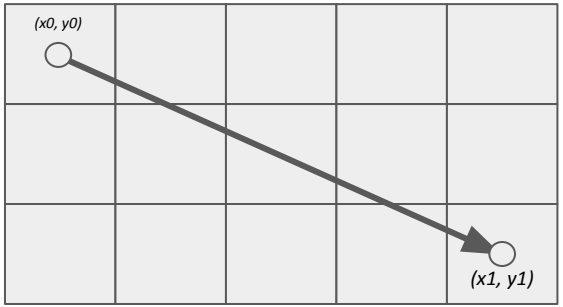
Bresenham Algorithm

What happens, if the slope is not between 0 and 1?

We can apply the same idea and adjust the algorithm accordingly.

- If the slope is between -1 and 0: Change the sign of two operations
- If the magnitude of the slope is larger than 1: exchange x and y
- If the vector between the points is directed left or downwards:

  change the direction



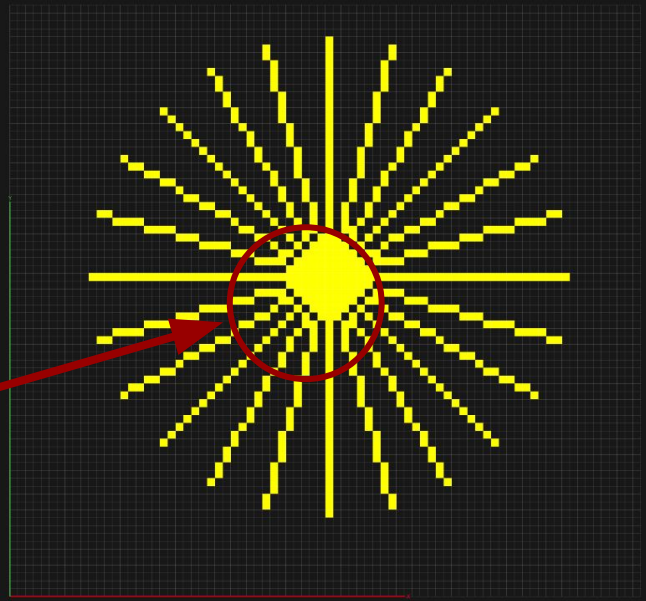
Breakout 2: Implement Bresenham Algorithm

Open the provided example "bresenham".

Look for the TODO comment in the `src/main.ts` and implement the Bresenham algorithm.

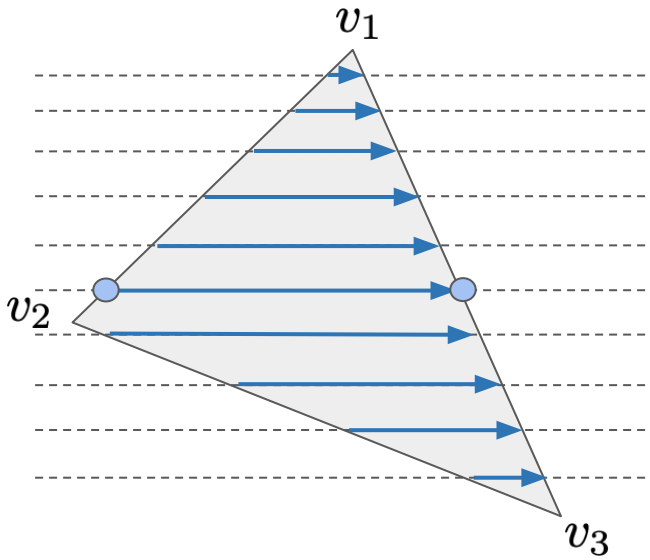
Step 1: Implement function `drawLineLow` and `drawLineHigh`

Step 2: Implement the function `drawLine` to sort out the line



Triangle Drawing: Scan Line Algorithm

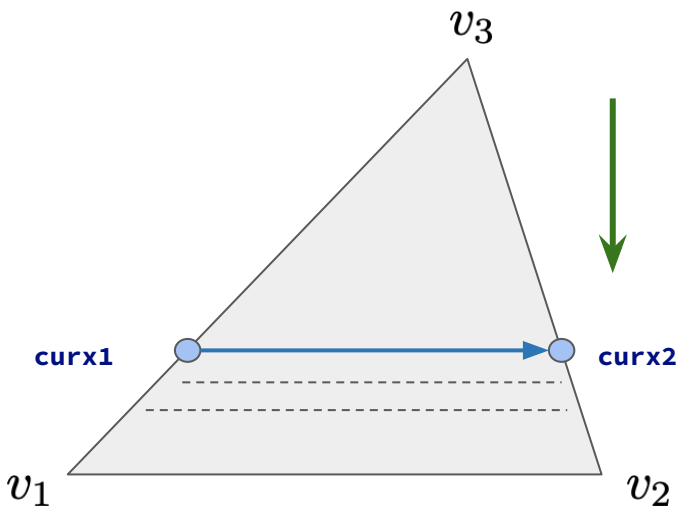
Any polygon can be considered as a group of triangles. To draw a triangle, we can utilize the Bresenham algorithm for the interior of the triangle. Basic idea: fill the triangle line by line horizontally or vertically

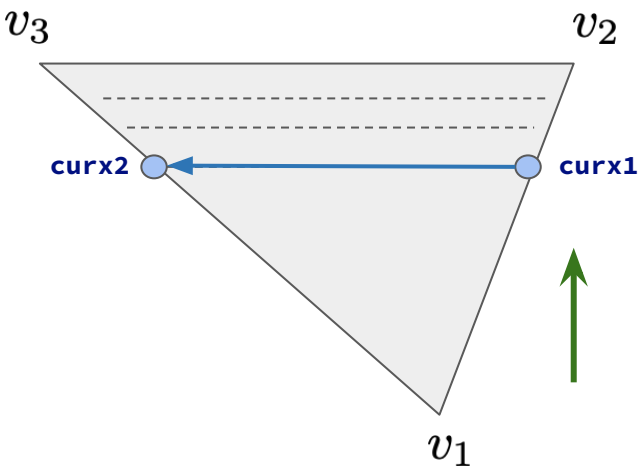


Scan Line Algorithm for Triangles

If one side of the triangle is parallel to the x-axis, we can simply parameterize the scanline.

Depending on where the third vertex lies, the scanline moves in different y-directions.

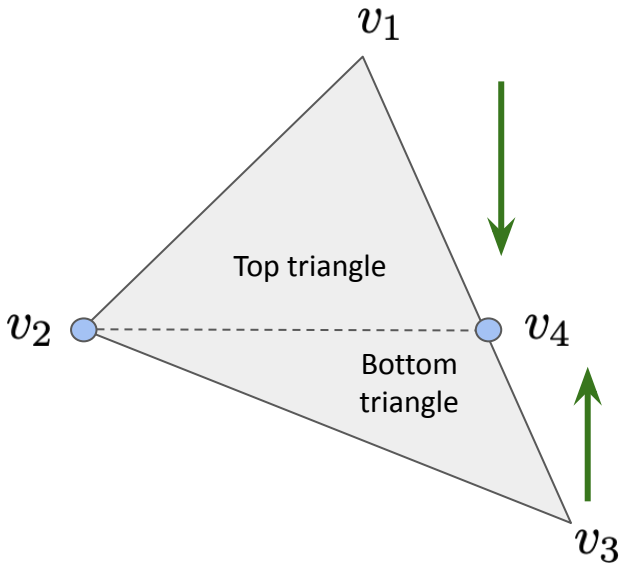




Scan Line Algorithm for Triangles

For an arbitrary triangle, we can calculate a new vertex v_4 with the same y-coordinate as v_2 and on the edge between the other two vertices of the triangle.

This results in two new triangles, one top triangle and one bottom triangle, which we can scan according to the previous slide.



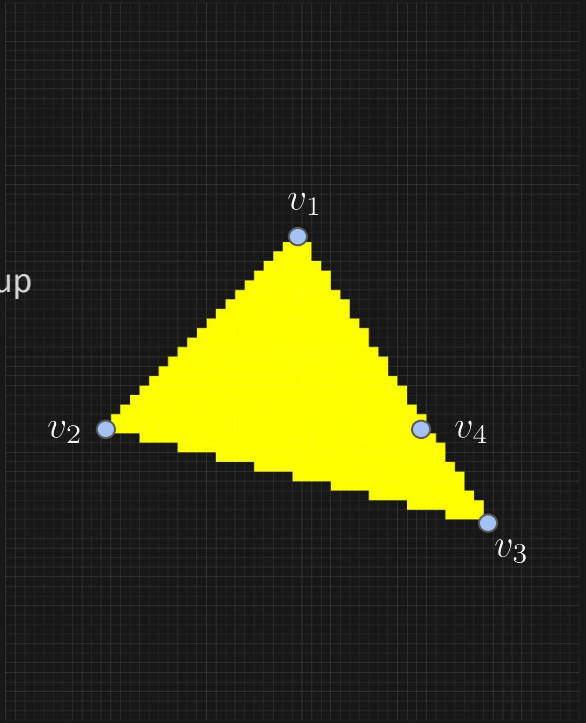
Breakout 3: Implement Scan Line Algorithm

Open the provided example "bresenham".

We now want to use our code from the last breakout session to draw a triangle.

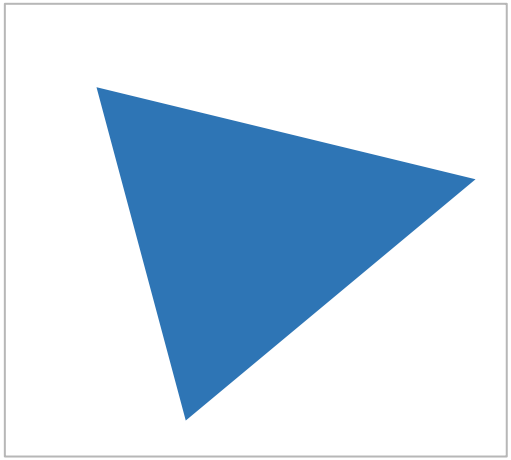
Step 3: Implement the function `drawTriangleTop` and `drawTriangleBottom`

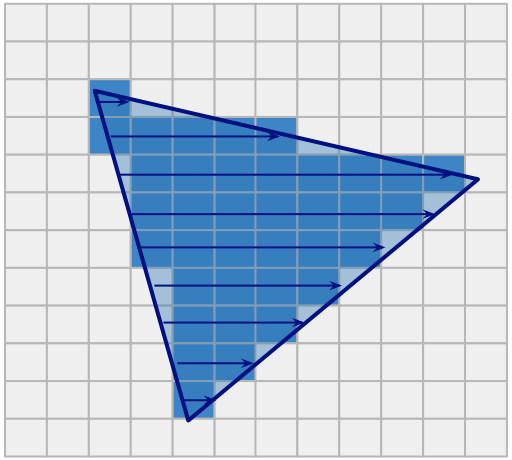
Step 4: Implement the function `drawTriangle` to check the triangle and split it up

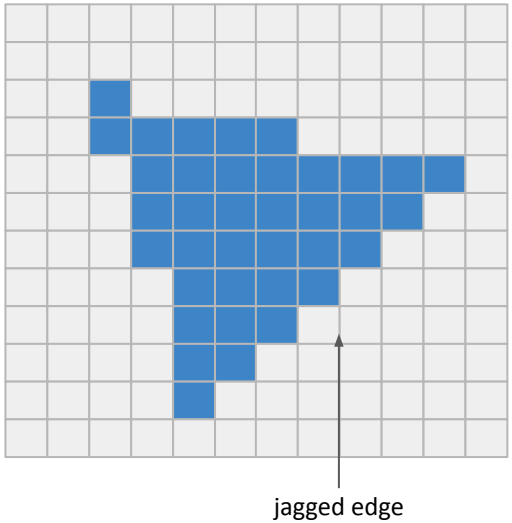


Issues with Bresenham and Scan Line Algorithms

- *Performance*: Desire parallelized execution for all pixels but drawing a line from left to right is sequential
- *Aliasing*: scan converted objects exhibit discretization artifacts (staircase effect)







Point-in-Triangle Assertion

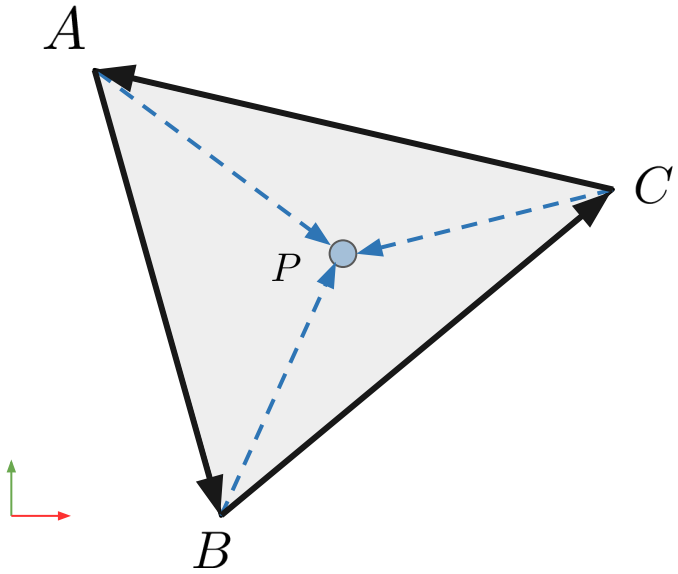
Basic idea: If P is always on the left of all edges

P is on the left side of AB: $\langle \overrightarrow{AB} \times \overrightarrow{AP}, (0,0,1,0)^\top \rangle$

P is on the left side of BC: $\langle \overrightarrow{BC} \times \overrightarrow{BP}, (0,0,1,0)^\top \rangle$

P is on the left side of CA: $\langle \overrightarrow{CA} \times \overrightarrow{CP}, (0,0,1,0)^\top \rangle$

⇒ P is inside triangle ABC



Alternative to scan line algorithm for triangle drawing:

For all pixels in the AABB of a given ABC, if a pixel is inside the triangle, then draw the pixel.

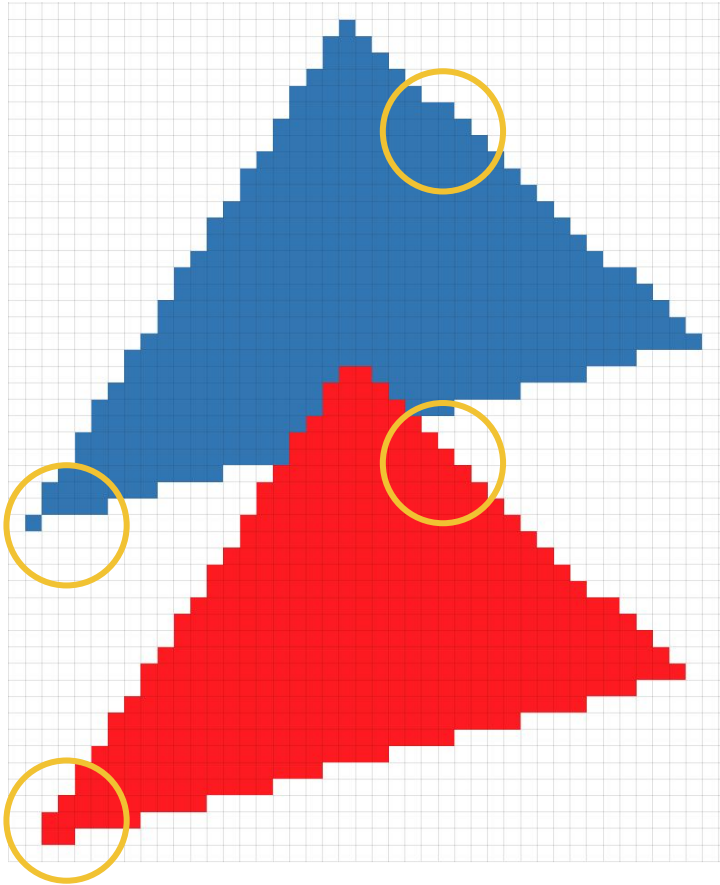
Point-in-triangle assertion is implemented on the GPU as fixed, specialized function. The GPU executes this test for all pixels parallelly and efficiently. Testing point-in-triangle is the most practical and efficient approach to draw a triangle.

Scan line vs. Point-in-Triangle Assertion based Drawing

- Scanline algorithm embeds numeric issue inside the algorithm design: when should the coordinates of a vertex position be numerically rounded (i.e. which pixel to initiate the drawing)?
- Point-in-triangle assertion is a boolean assertion to check if pixel center is inside the triangle, and can be easily optimized and executed in parallel

Scan Line Drawing

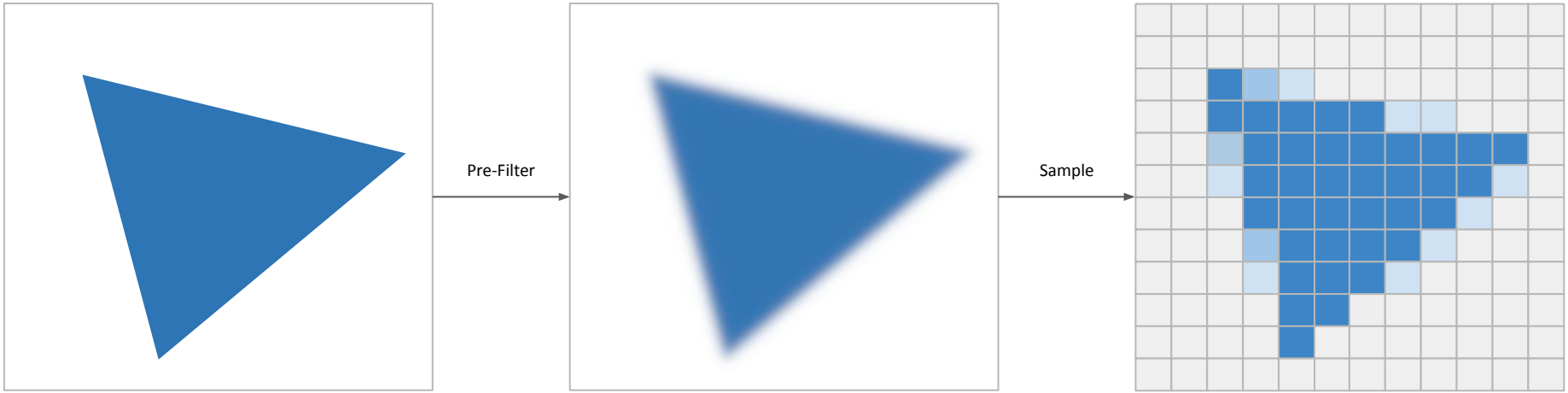
Point-in-Triangle Drawing



*Corner case: if a pixel center is exactly at the edge of the triangle: decide yourself in the implementation

Aliasing and Antialiasing

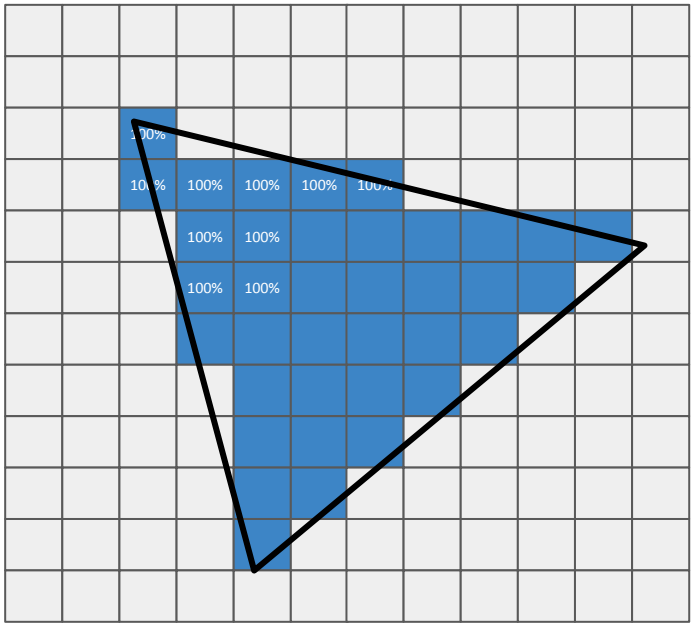
- Both scan line algorithm, and point-in-triangle assertion based drawing introduces line *aliasing* issue
- How to reduce aliasing issue?
  - Higher resolution display (therefore higher frame buffer) i.e. + ~~€€€~~
    - Disadvantage: adds more computation cost to software, and needs high resolution on hardware
  - **Antialiasing**



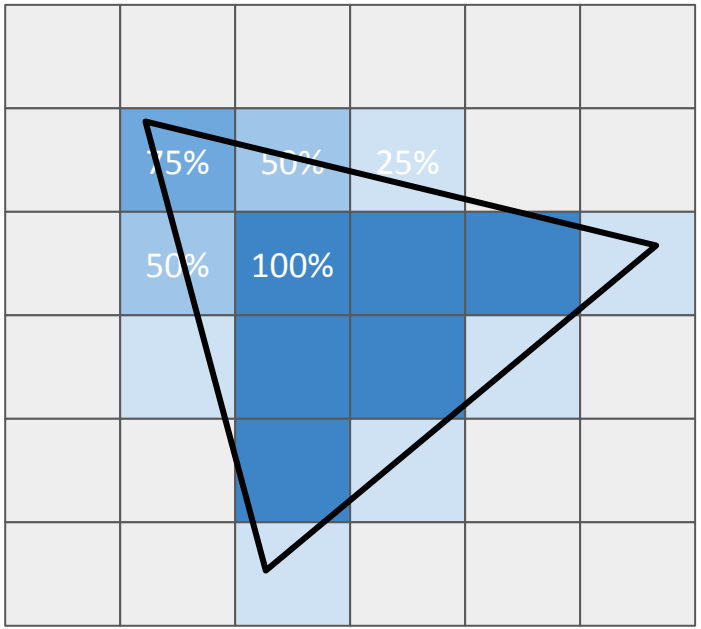
Multi-Sample Anti-aliasing (MSAA)

Multi sample antialiasing (MSAA): Sampling high resolution samples then render in a lower resolution

MSAA computes the coverage of a triangle area on a pixel



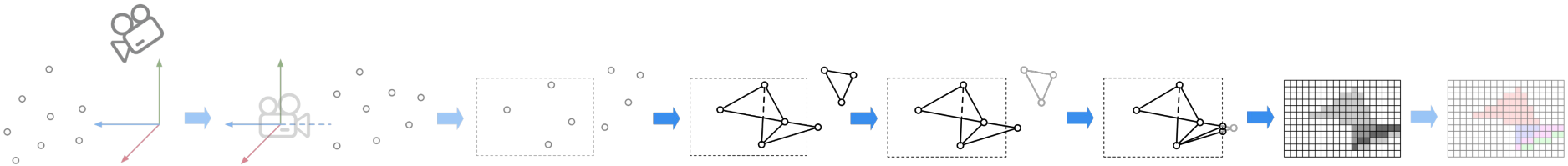
2x2 Super sampling



Averaging down

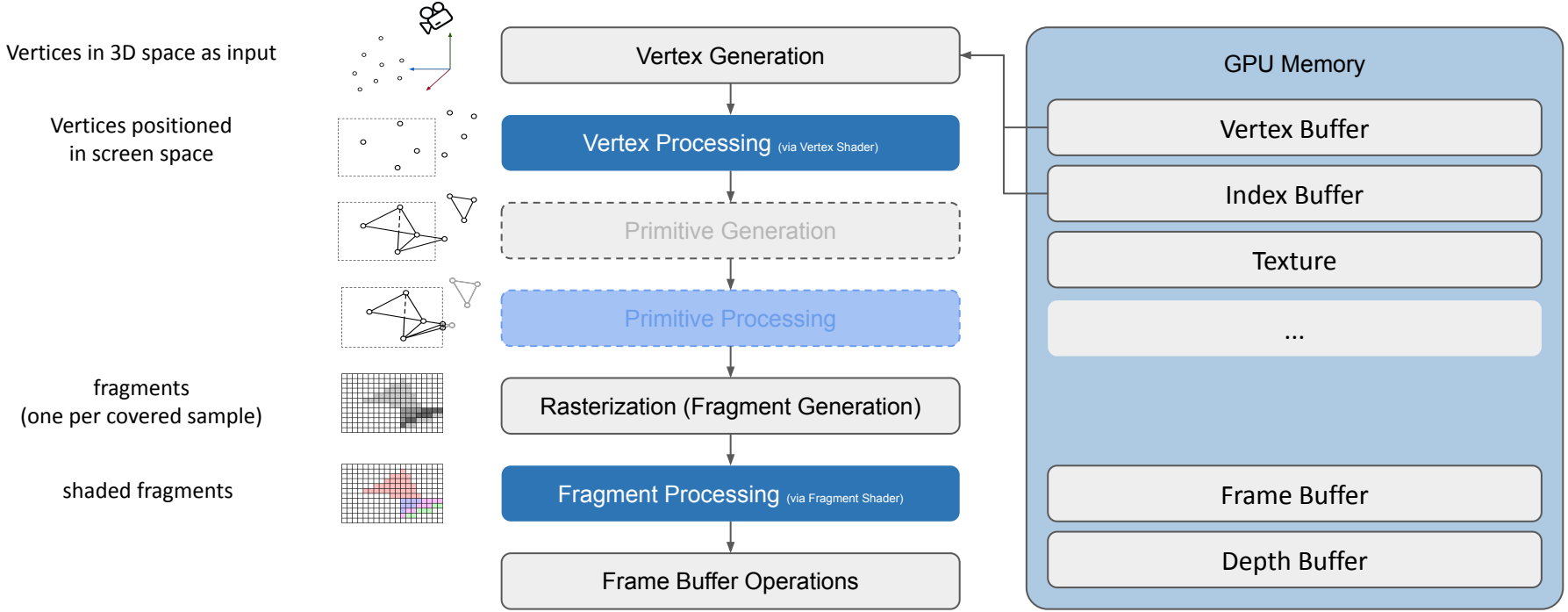
Tutorial 5: Rasterization

- Culling
  - Different Types of Culling
  - AABB and View Frustum Culling
- Screen-space Buffers
  - Frame and Depth Buffer
- Drawing and Sampling
  - Bresenham and Scan Line Algorithms
- Modern Rasterization Rendering Pipeline
  - Shaders



Modern Rasterization Rendering Pipeline (on GPU)

The pipeline can be executed for multiple ***pass***es, and one rendering ***pass*** means: 1) create a frame buffer, 2) specify one or more buffers as output, and 3) render content from an output buffer



OpenGL *Deprecated!*

OpenGL is a standardized set of APIs that describes the previous rasterization rendering pipeline on a GPU.

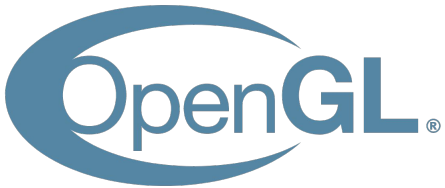
Advantage:

- Cross platform

Disadvantages:

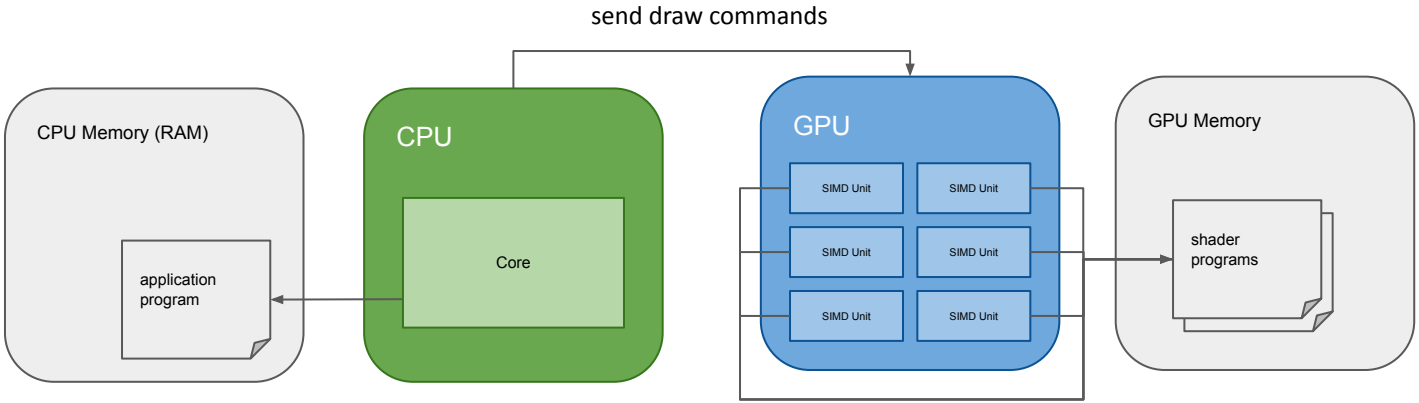
- Compatibility: different versions have different set of APIs or different API behaviors
- State-machine programming model, C-style and not easy to use
- Debugging is (or was) non-trivial

Fore more, see http://docs.gl/. We will not discuss OpenGL in detail. Instead...

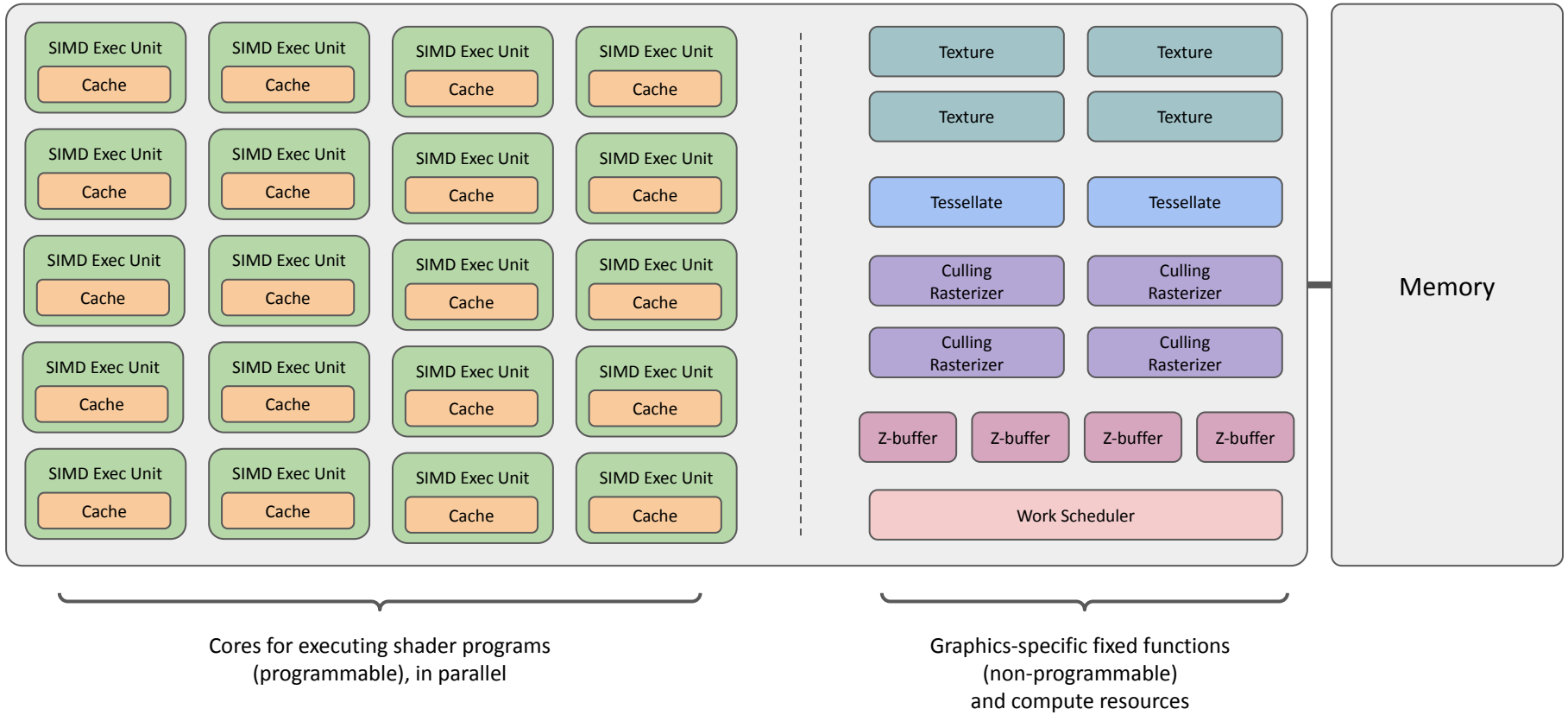


Shader Program and Shading Language

- Shader is a small program that runs on GPU instead of CPU
- Shader programs are written in language similar to C but with restrictions, called *shading language*
- To run a shader program (on GPU), similar to CPU programs, one must:
  1. create shaders for compilation
  2. compile shaders for execution
  3. link shader programs together and the application
  4. use shader program when necessary

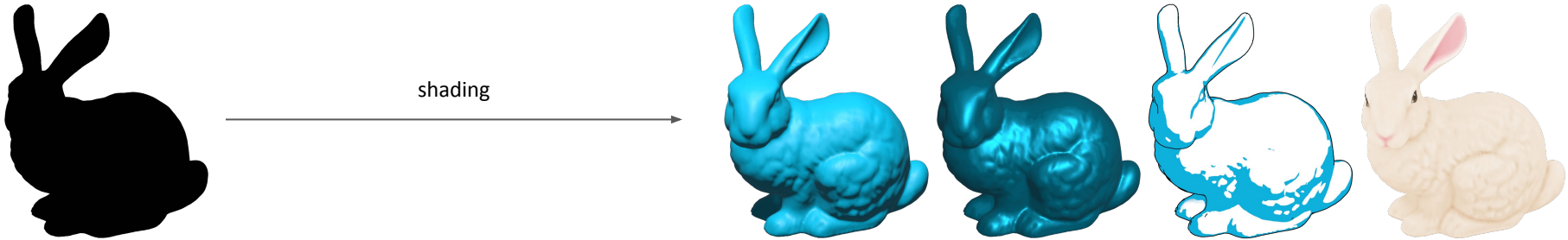


Executing Shaders on a Multi-core Processor (GPU)



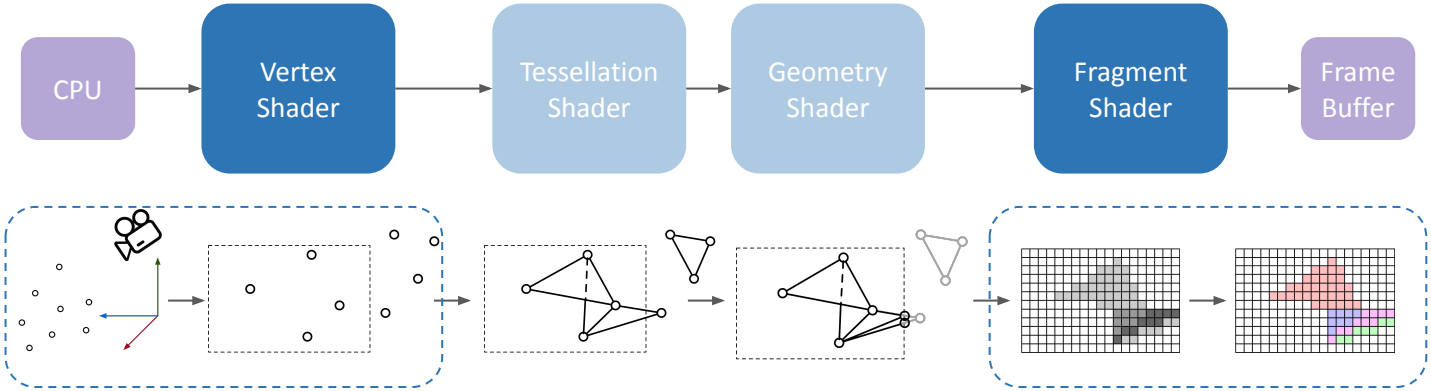
Why A Language?

- High-level, domain-specific language to describe *shading behavior*
  - Better utilize GPU and can customize
  - In ancient times: assembly on GPUs
  - e.g. *GLSL* in OpenGL, *HLSL* in DirectX
- Shading is a *local* behavior for a specific material
- A rasterizer turns geometries into pixels via sampling but does not include the process of how to figure out what is the "correct" color of a pixel, e.g. different shading behavior



OpenGL ES Shading Language (GLSL ES)

- GLSL ES (shortly GLSL) enables programmable stages of graphics pipeline computing using GPU in WebGL
- Different shader stages
  - ***vertex shader***
  - ***tessellation shader***
  - ***geometry shader***
  - ***fragment shader***
  - ***compute shader***
  - ***...***

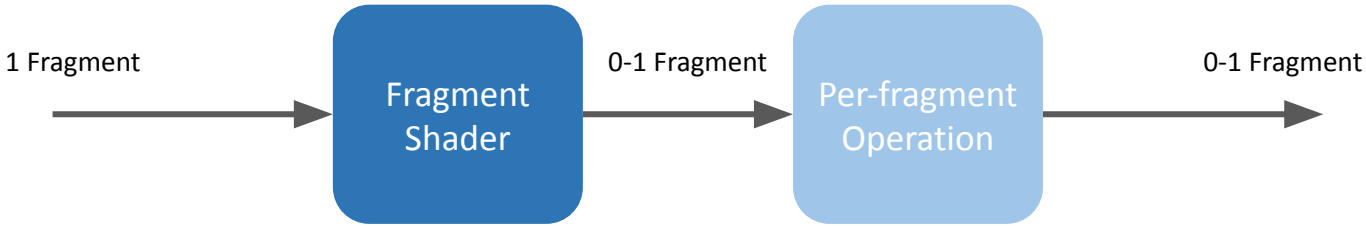


Vertex Shader

- Transformation of single vertices and their attributes (e.g. normals, ...)
  - No vertex generation
  - No vertex destruction (handled by clipping)
- Calculation of all attributes that remain constant per vertex
  - Saves computing time compared to the Fragment Shader
  - e.g. lighting by vertex (old-fashioned)
- Set attributes to be interpolated per fragment
  - e.g. normals for per-pixel lighting
- Determines the position of the vertices, otherwise cannot continue to the subsequent stages of the pipeline.

Fragment Shader

- Allows calculation per result pixel that ends up in the output buffer
  - Per-pixel lighting/shading
  - Sampling of data within the primitive, e.g. for
    - volume rendering
    - Implicit surfaces, glyphs
- The **input** attributes are *interpolated* (discussed later) within the primitive (can be turned off)
- Fragments can be discarded: `discard`
- Fragment operations: Tests, blending and etc.



Compute Shader

- Compute shaders allows general purpose, parallel computing on the GPU (with many many cores)
  - Examples: Physics calculations, particle systems, fluid or substance simulations
- Compute shader is located outside the rendering pipeline
  - No input from inside the pipeline and no output to the pipeline
- Can read and write textures, images and shader buffers
- WebGL Support
  - No support, and will not be supported :(
  - (Yet) very early alpha support in WebGPU and requires Chromium nightly builds

Summary

We discussed:

- Different types of culling techniques for the acceleration of rendering
- The two most important screen space buffers and related issues
- Rasterization process that renders a triangle from the 3D world space to 2D screen space using Bresenham and Scanline algorithm
- Issues with Bresenham and scan line algorithms and an alternative drawing approach that using point-in-triangle assertion
- Aliasing and antialiasing sampling issue in drawing
- The modern rasterization rendering pipeline and its components
- GLSL as a programming language for writing shader programs that execute on a GPU

How to create pictures like this?

Source: The Elder Ring