Tutorial 5

Rasterization

Computer Graphics

Summer Semester 2020

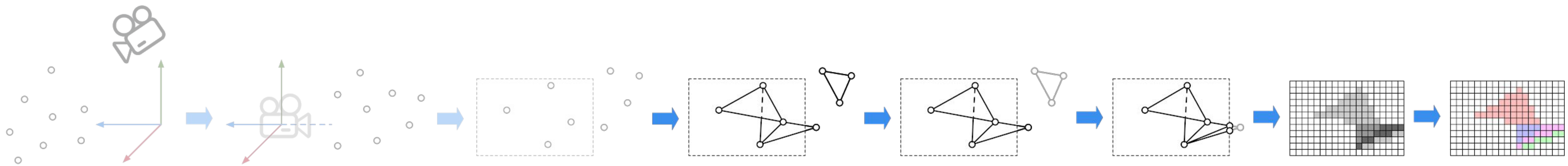
Ludwig-Maximilians-Universität München

Exam

- 3 "Online-Hausarbeiten", release in the Uni2Work
- Tasks are similar to the existing assignments. The schedule:
  - Abgabe 1 (Programming tasks, 50p) 06.07.-10.07.20 (5 days)
  - Abgabe 2 (Non-programming tasks, 50p) 13.07.-18.07.20 (6 days)
  - Abgabe 3 (Programming tasks, 100p) 20.07.-31.07.20 (12 days)
- You need 100 points to pass the exam and 190 points to get 1.0
- 10% Bonus are given in the Online-Hausarbeiten
- Please register yourself via Uni2Work

Agenda

- Culling
- Clipping
- Frame/Depth Buffer
- Drawing
- Antialiasing
- OpenGL Shading Language (GLSL)

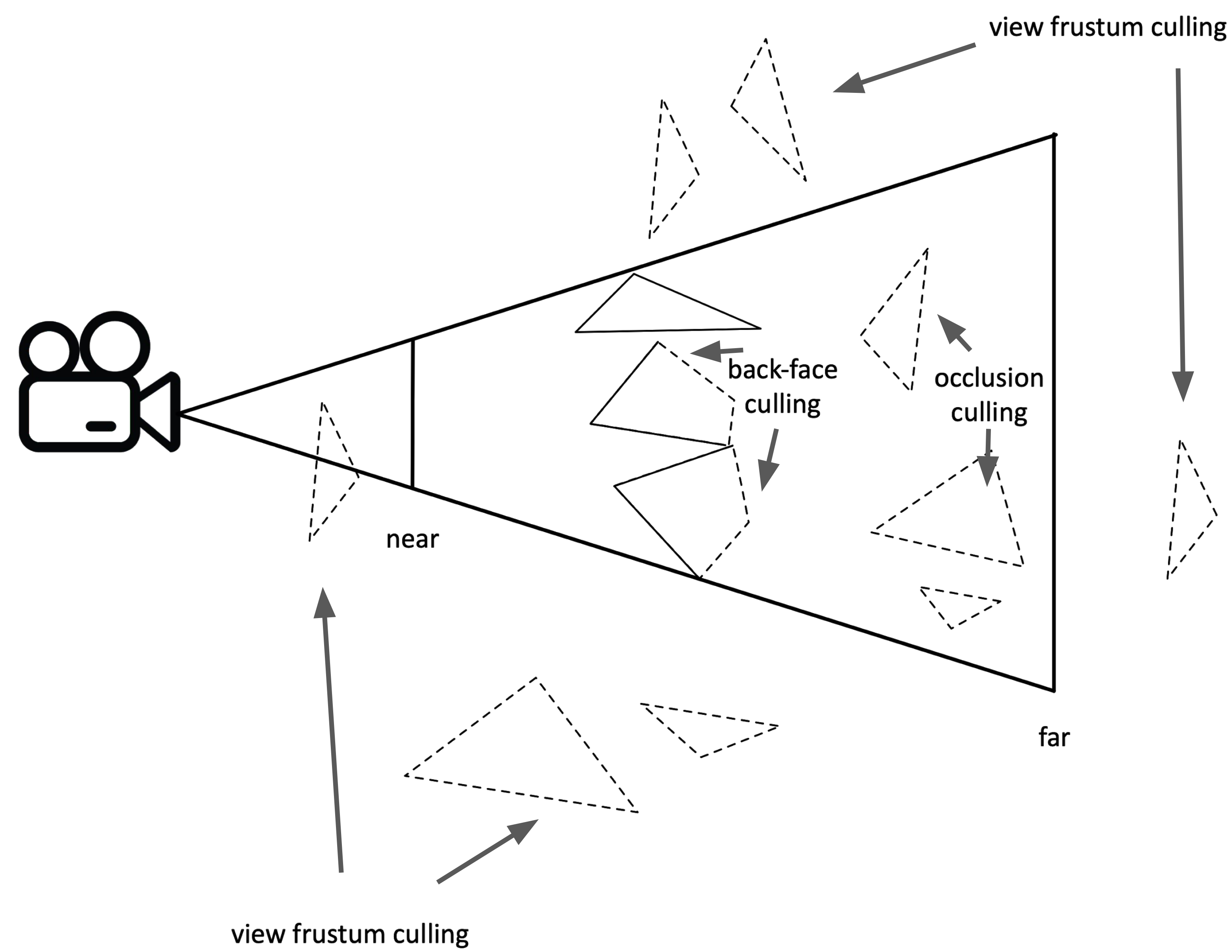


Tutorial 5: Rasterization

- Culling
- Clipping
- Frame/Depth Buffer
- Drawing
- Antialiasing
- OpenGL Shading Language (GLSL)

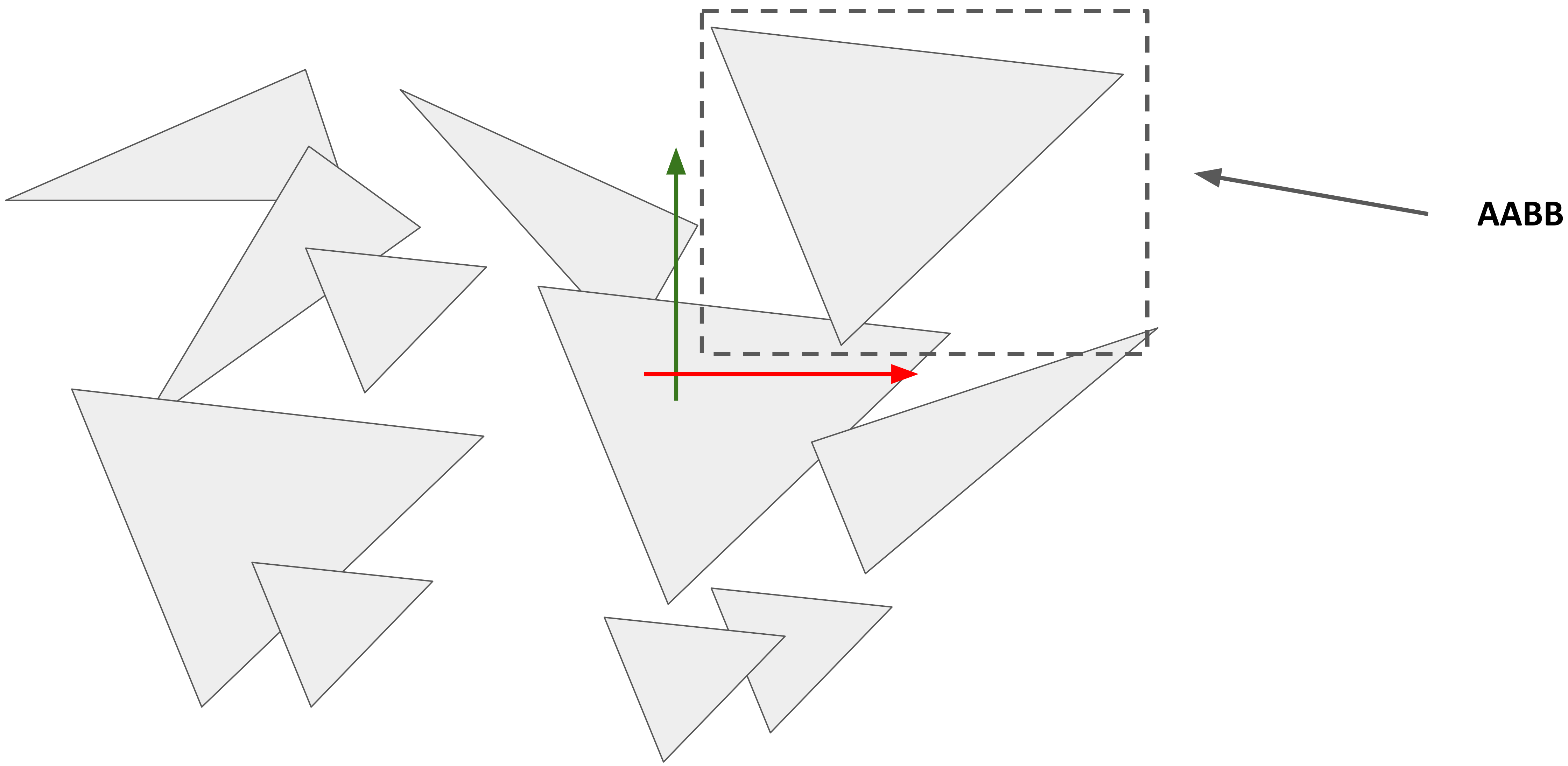
Task 1 a)

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



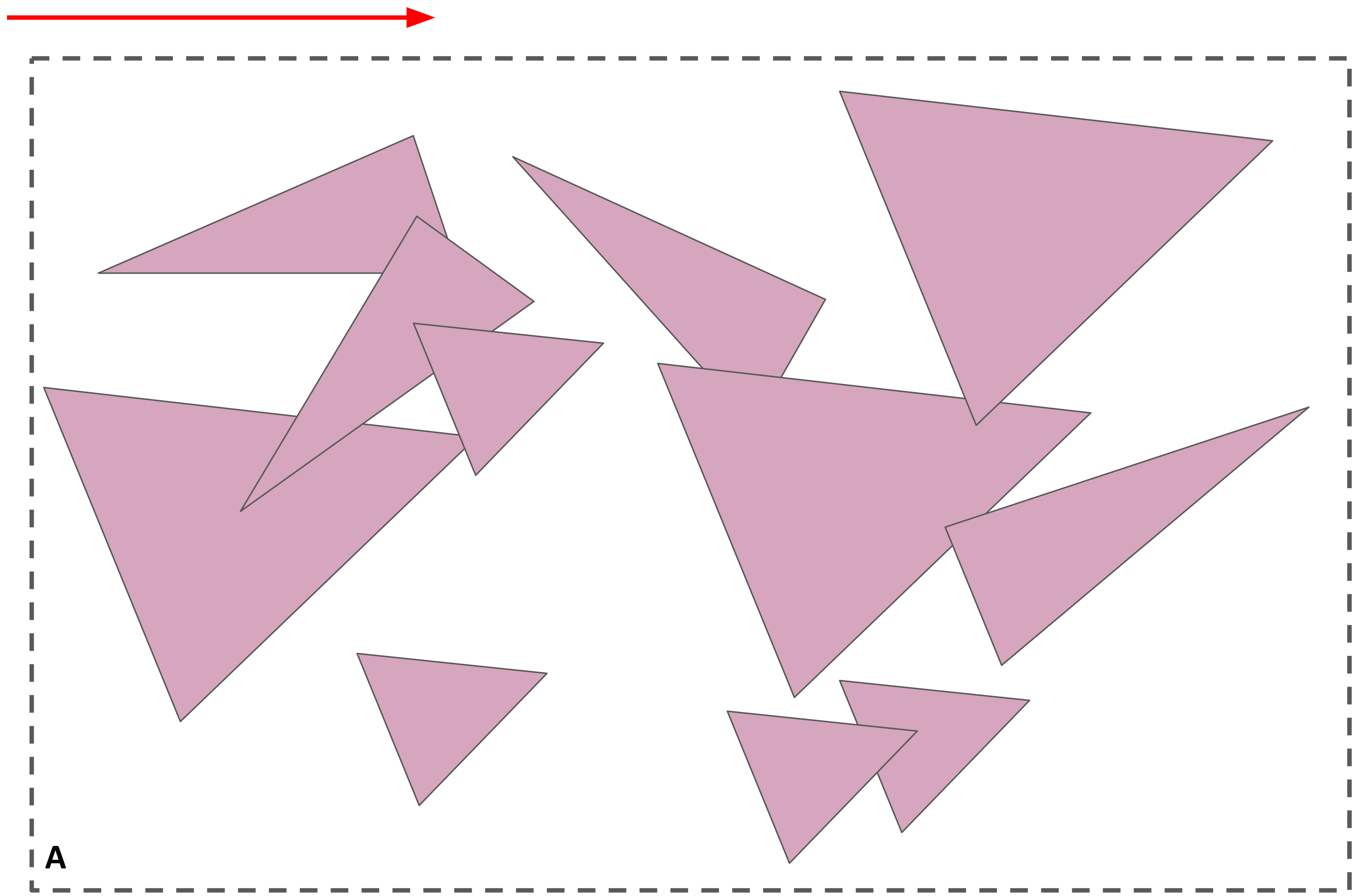
Bounding Volume Hierarchy **(BVH)**

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

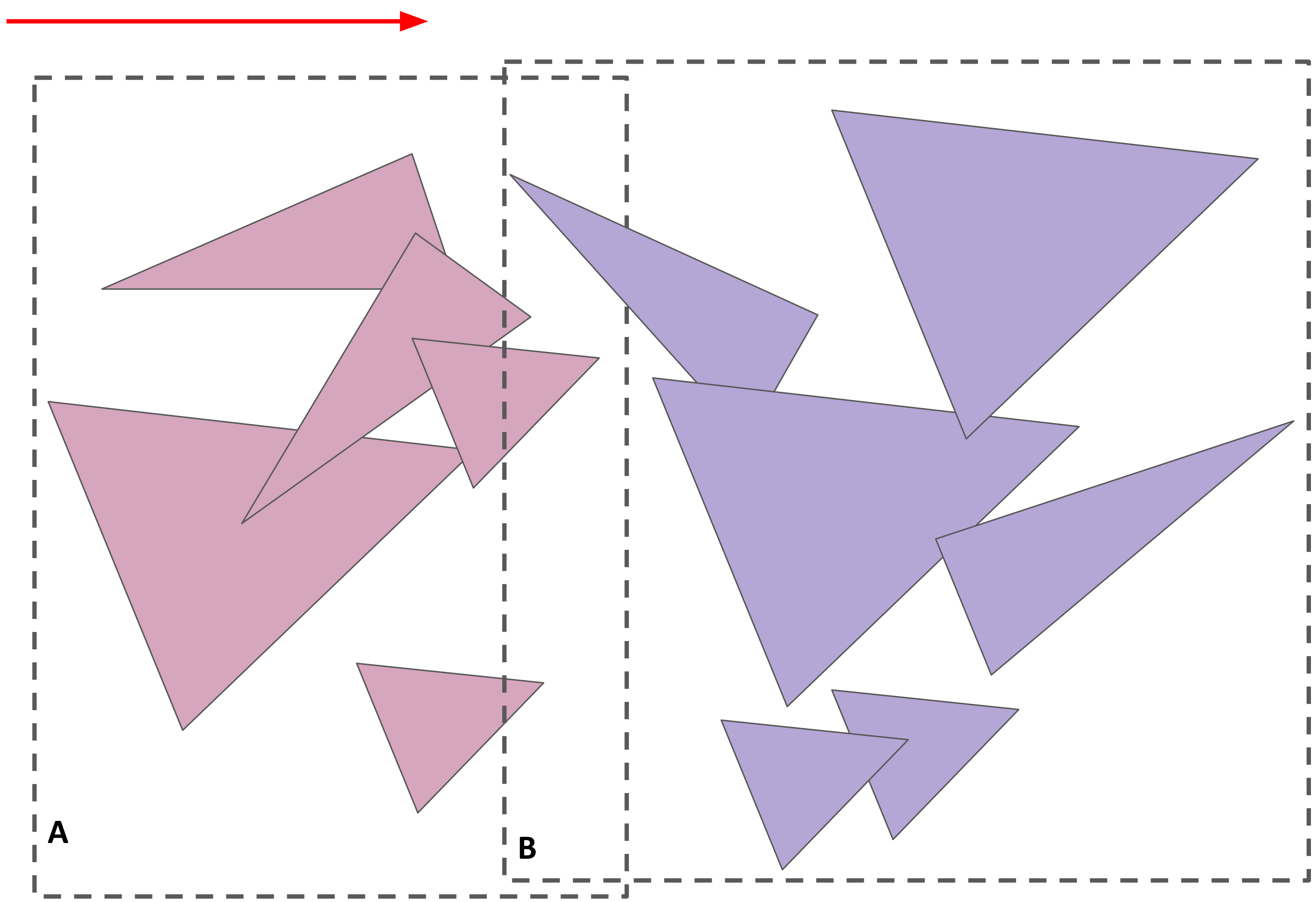


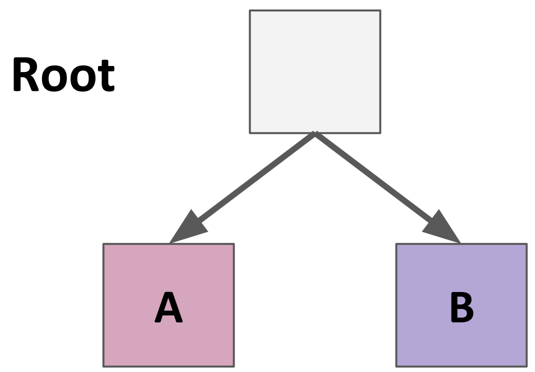
Bounding Volume Hierarchy (BVH)

Core idea: split along an axis and divide number of triangles by density

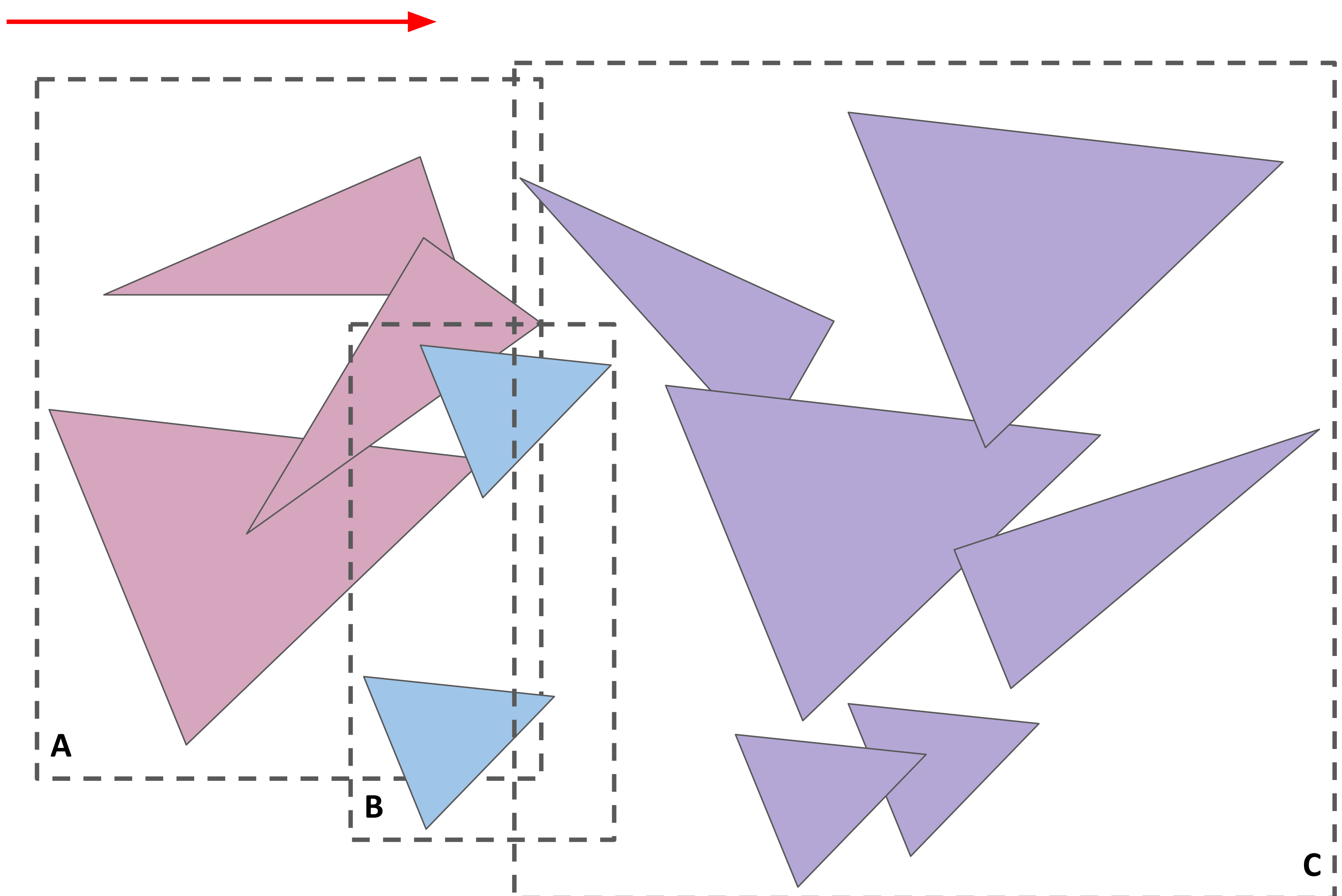


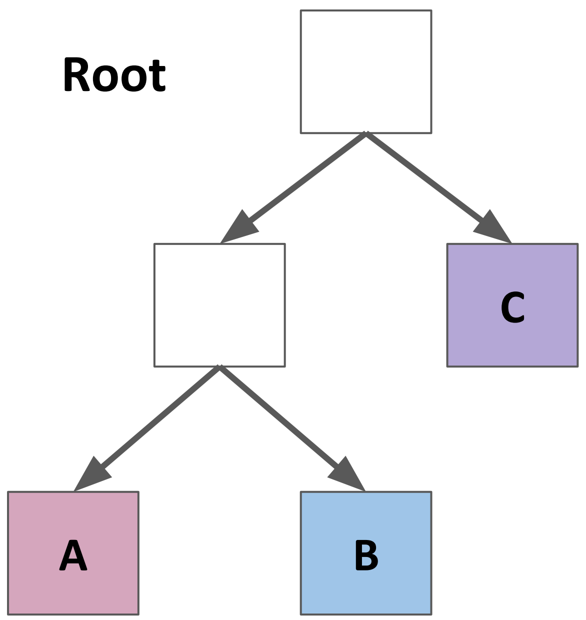
Bounding Volume Hierarchy (BVH)





Bounding Volume Hierarchy (BVH)

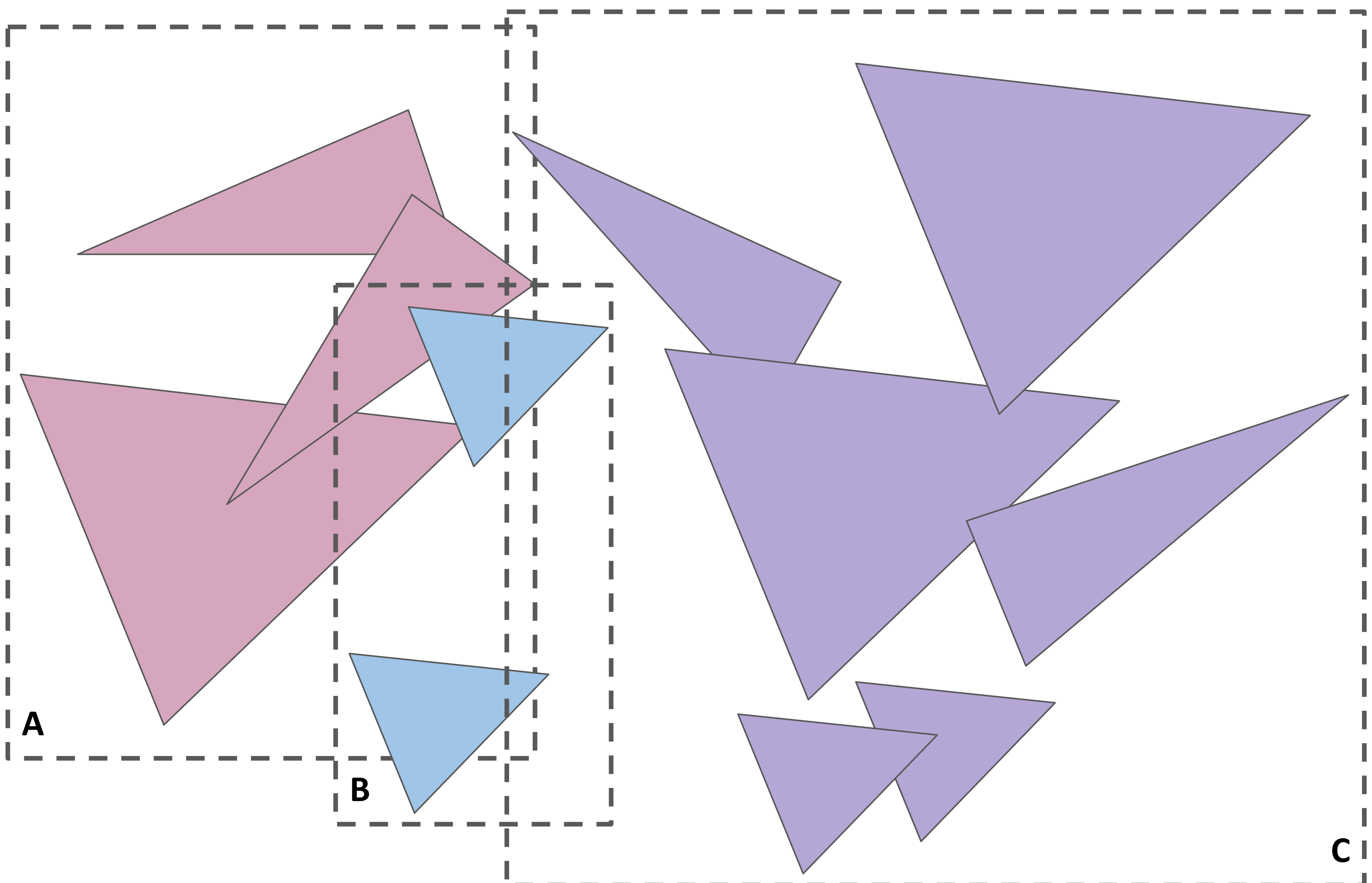




Process:

- Compute bounding box
- Split set of objects into two subsets
- Recompute bounding boxes
- Stop when necessary
- Store objects in each leaf node
- Similar to scene graph

Why BVH with AABB?



- Very efficient and practical for culling!
  - An object can only appear in one node
  - Easy to compute axis-aligned bounding volume
  - No additional intersection check between triangles and bounding volume
  - Low memory footprint
  - …
- Comparing to Octree? Octree:
  - #partitions explode (*8)
  - An object may occur in multiple partitions
  - Requires additional intersection check
  - ...

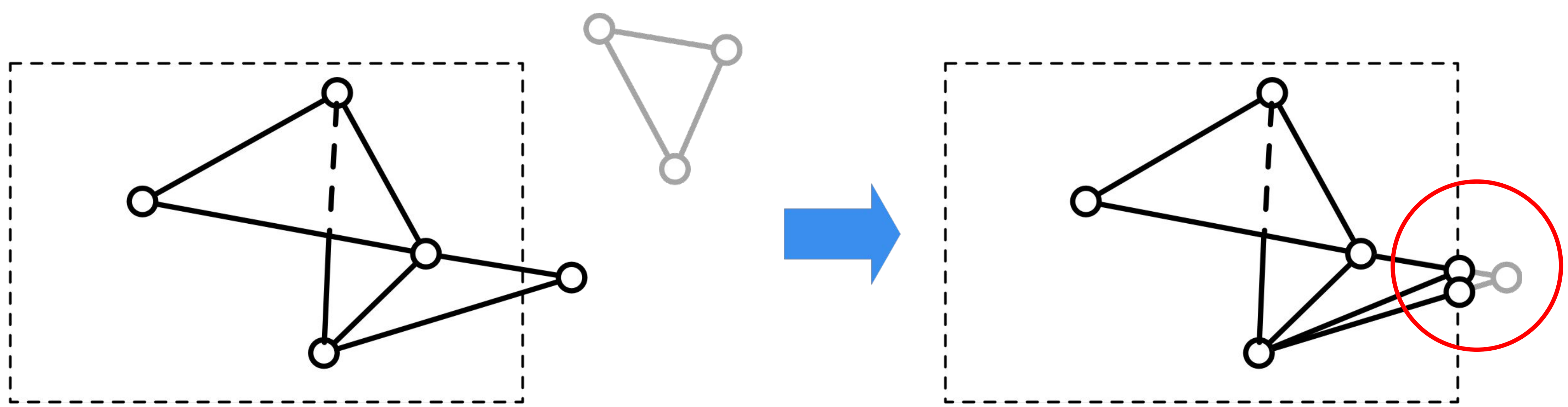
Tutorial 5: Rasterization

- Culling
- **Clipping**
- Frame/Depth Buffer
- Drawing
- Antialiasing
- OpenGL Shading Language (GLSL)

Task 1 b) Clipping

Purpose: before drawing, make sure the mesh is completely inside the [-1, 1]^3 unit cube

Issue: Creates more triangles



Task 1 c) How?

- Cohen & Sutherland algorithm
  - Check the lecture slides
  - Less efficient
- *Liang-Barsky algorithm*
  - significantly more efficient
  - Very practical in conjunction with AABBs

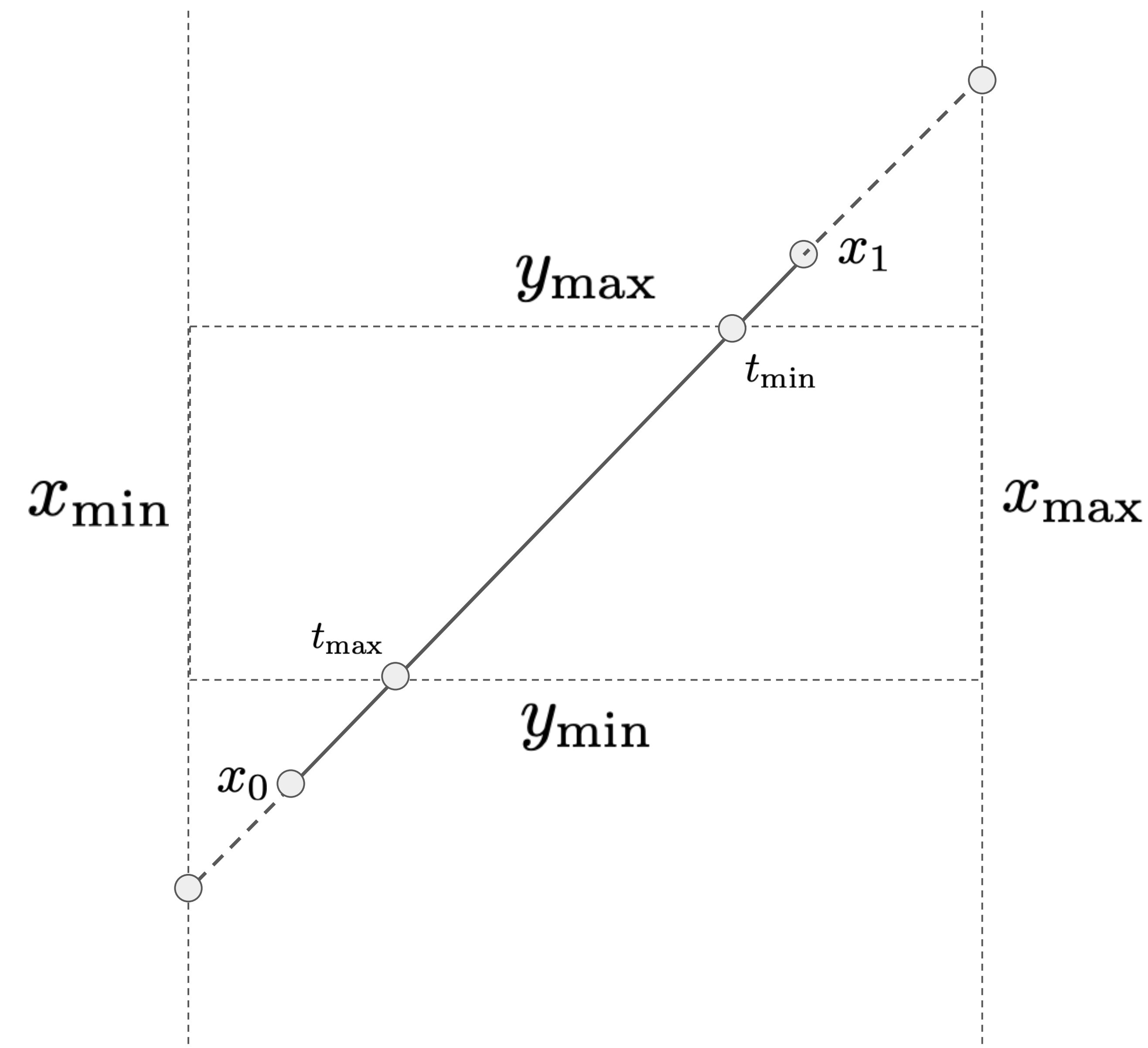
Liang-Barsky Algorithm

Line parametric equation:

$$x_{\min} \leq x_0 + t(x_1 - x_0) \leq x_{\max}$$
$$y_{\min} \leq y_0 + t(y_1 - y_0) \leq y_{\max}$$

Expressed by $tp_i \leq q_i, i = 1, 2, 3, 4$

$p_1 = -(x_1 - x_0),$	$q_1 = x_0 - x_{\min}$	(left)
$p_2 = x_1 - x_0,$	$q_2 = x_{\max} - x_0$	(right)
$p_3 = -(y_1 - y_0),$	$q_3 = y_0 - y_{\min}$	(bottom)
$p_4 = y_1 - y_0,$	$q_4 = y_{\max} - y_0$	(top)



1. Parallel to viewport edge ⇒ p_i=0
2. iq_i < 0 ⇒ outside
3. p_i < 0 ⇒ outside to inside, p_i > 0 inside to outside
4. $t_i = q_i/p_i$ are intersection points (with boundaries or boundary extensions)
5. t_{min} = min(t_i, 1), t_{max} = max(0, t_i). Line intersect with viewport if and only if $t_{max} \leq t_{min}$

Tutorial 5: Rasterization

- Culling
- Clipping
- **Frame/Depth Buffer**
- Drawing
- Antialiasing
- OpenGL Shading Language (GLSL)

Frame and Depth Buffers

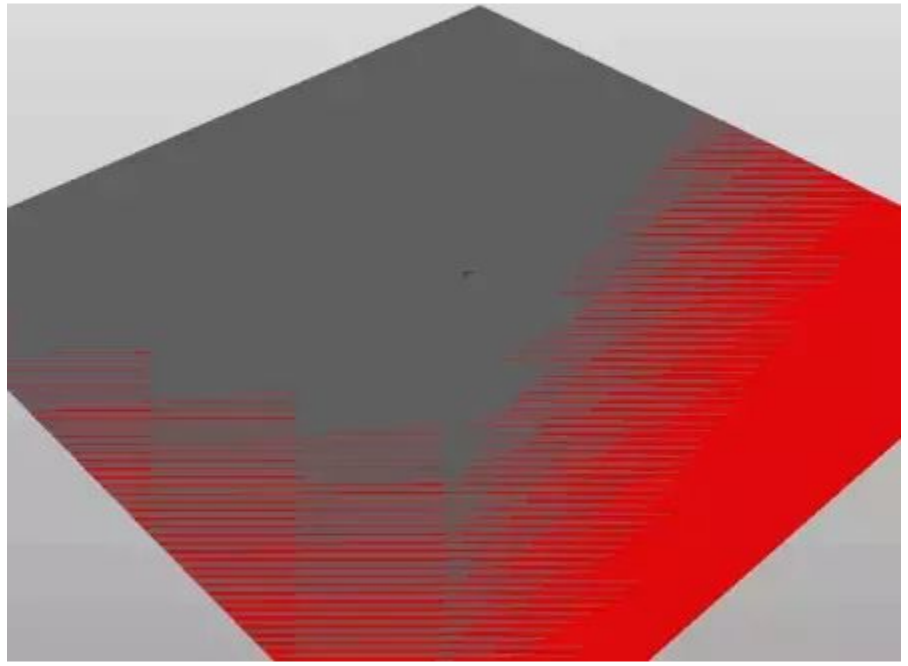
The Painter's algorithm cannot solve the occlusion issue. Z-buffer idea:

- Store current minimum z-value for each pixel
- Needs an additional buffer for depth values
  - frame buffer stores color values, directly sent to display (Task 1 f)
  - depth buffer stores depth, for visibility test
- Pseudocode:

```
let frameBuffer = [...]
let depthBuffer = [...]
triangles.forEach(tri => {
 tri.project().fragments.forEach((x, y, z, color) => {
   if (z < depthBuffer[x][y])    // depth test: check closest pixel
     frameBuffer[x][y] = color   // update color in frame buffer
     depthBuffer[x][y] = z       // update depth in depth buffer
 })
})
```

Task 1 d) Z-fighting: Case 1 - Depth values are very close

If two planes have same depth value, Z-buffer might randomly pick a fragment to render because of the depth value precision (try 0.1+0.2 in your browser console):

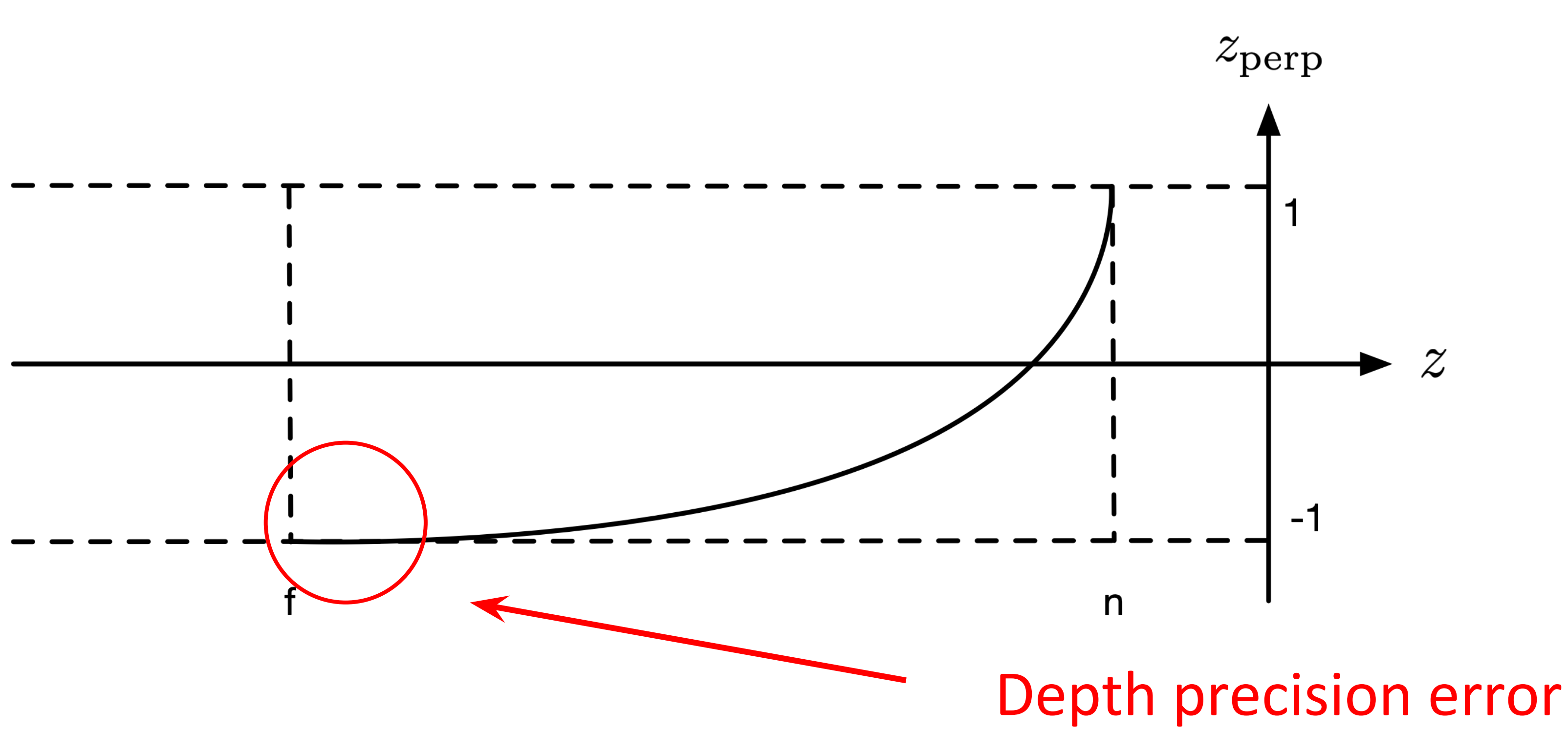


Task 1 d) Z-fighting: Case 2 - Close to far plane

Recall the perspective projection matrix (see Assignment 4):

$$P' = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = T_{\text{persp}} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} -\frac{1}{\lambda \tan \frac{\theta}{2}} & 0 & 0 & 0 \\ 0 & -\frac{1}{\tan \frac{\theta}{2}} & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cdots \\ \cdots \\ \frac{n+f}{n-f} z + \frac{2nf}{f-n} \\ z \end{pmatrix} = \begin{pmatrix} \cdots \\ \cdots \\ \frac{n+f}{n-f} + \frac{2nf}{f-n} \frac{1}{z} \\ 1 \end{pmatrix}$$

$$\implies z_{\text{perp}} = \frac{2nf}{f-n} \frac{1}{z} + \frac{n+f}{n-f} \in [-1, 1], 0 > n \geq z \geq f$$



Z-values are less accurate when the object is further away from the viewpoint.

Q: What about orthographic projection?

Task 1 e) How to avoid Z-fighting?

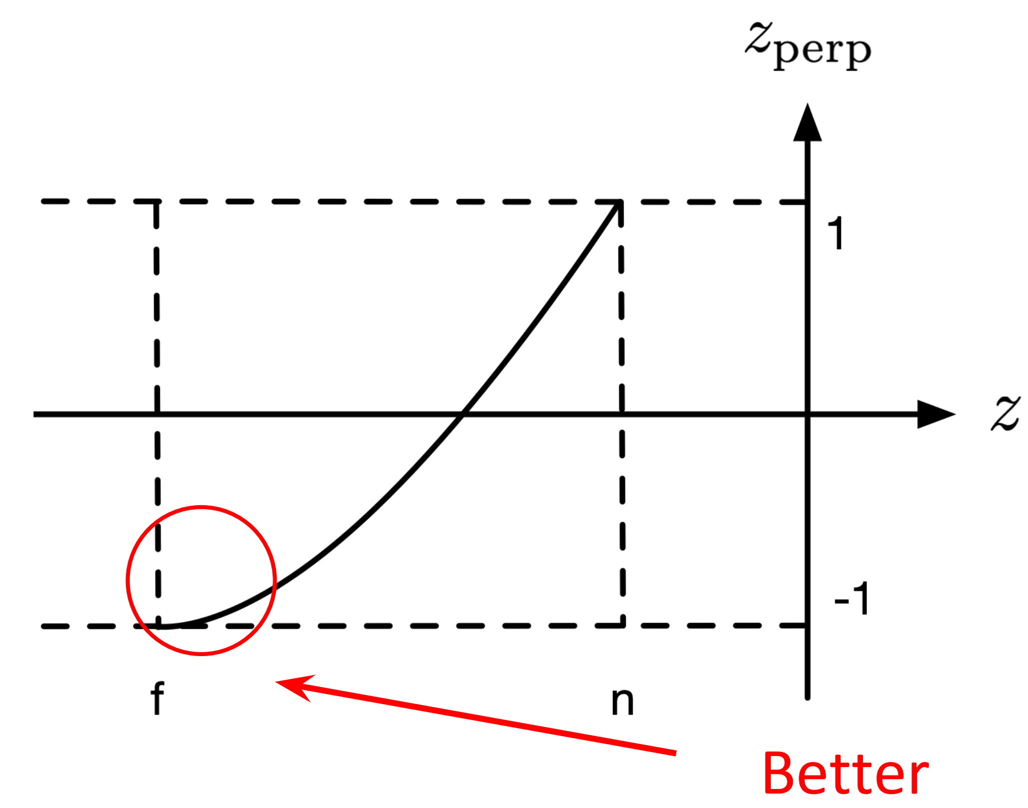
1. (Properly) make near and far planes closer

2. Use higher precision depth buffer

3. Use a fog effect to avoid objects close to far plane,

and move objects away from each other

...



Task 1 f) Why do we need a frame buffer?

Performance!

- Flushing an entire buffer at once is much faster than rendering pixel by pixel
- Enables CPU/GPU pipelining and we are able to cache multiple frames if we have enough memory
- ...

Tutorial 5: Rasterization

- Culling
- Clipping
- Frame/Depth Buffer
- **Drawing**
- Antialiasing
- OpenGL Shading Language (GLSL)

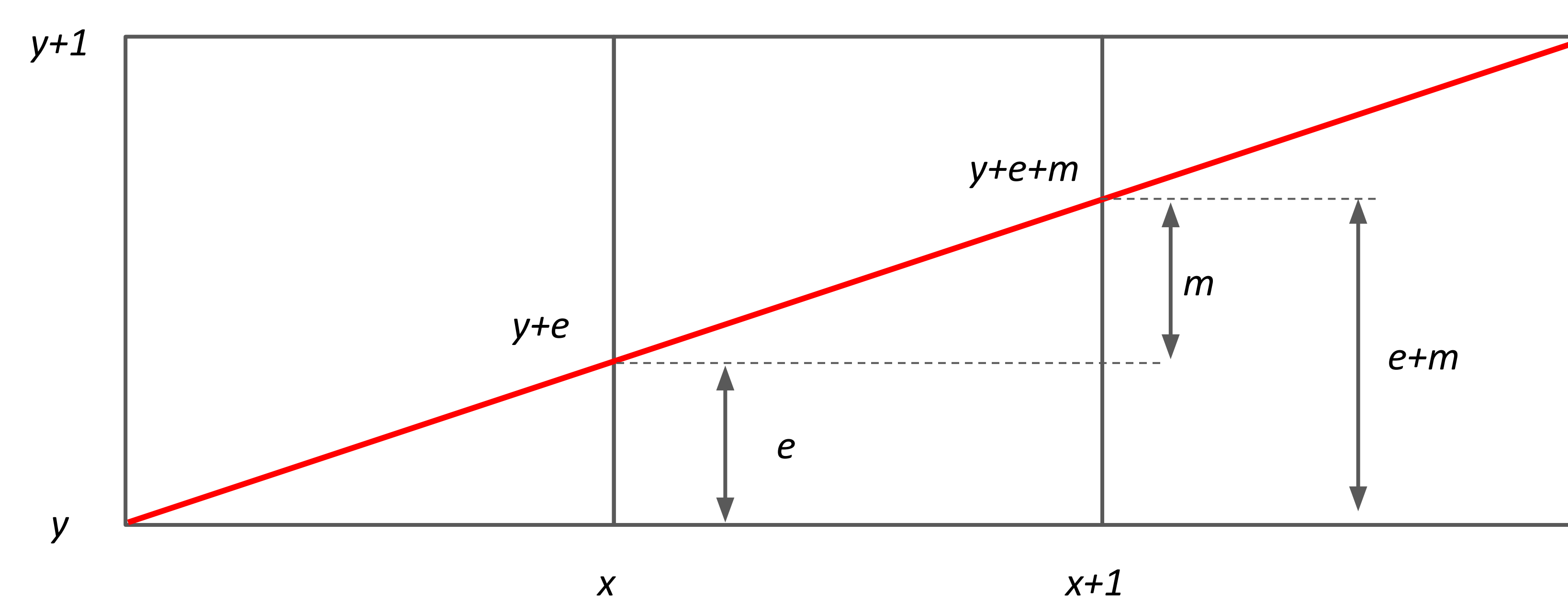
Bresenham Algorithm

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

Consider a line with slope in range [0, 1]

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)
- if e + m <= 0.5 then draw (x+1, y)



J. E. Bresenham. 1965. *Algorithm for computer control of a digital plotter*. IBM Syst. J. 4, 1 (March 1965), 25–30. DOI:https://doi.org/10.1147/sj.41.0025

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

```
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

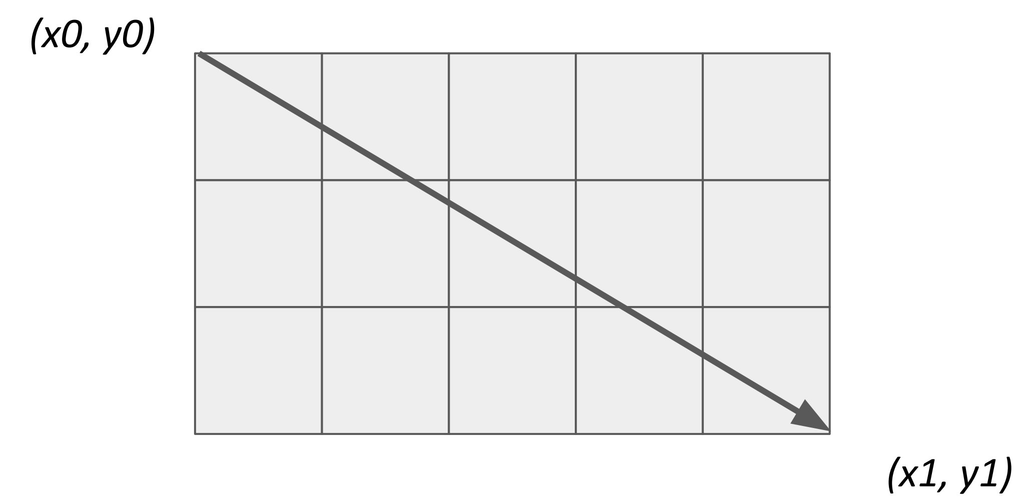
Why? Blazing fast: No floating points; multiply 2 can be done by left-shift (<<)

Bresenham Algorithm (cont.)

There are other cases, but same idea can be applied.

For instance: `dy < 0`

```
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
}
```



Task 1 g) Bresenham (complete version)

Case 1: 0 <= |slope| <= 1

```
drawLineLow(x0, y0, x1, y1, color) {
  let dx = x1 - x0
  let dy = y1 - y0
  let yi = 1
  if (dy < 0) {
    yi = -1
    dy = -dy
  }
  let D = 2*dy - dx
  let y = y0
  for (let x = x0; x <= x1; x++) {
    this.drawPoint(x, y, color)
    if (D > 0) {
      y += yi
      D -= 2*dx
    }
    D += 2*dy
  }
}
```

Case 2: |slope| >= 1, include dx === 0

```
drawLineHigh(x0, y0, x1, y1, color) {
  let dx = x1 - x0
  let dy = y1 - y0
  let xi = 1
  if (dx < 0) {
    xi = -1
    dx = -dx
  }
  let D = 2*dx - dy
  let x = x0
  for (let y = y0; y <= y1; y++) {
    this.drawPoint(x, y, color)
    if (D > 0) {
      x += xi
      D -= 2*dy
    }
    D += 2*dx
  }
}
```

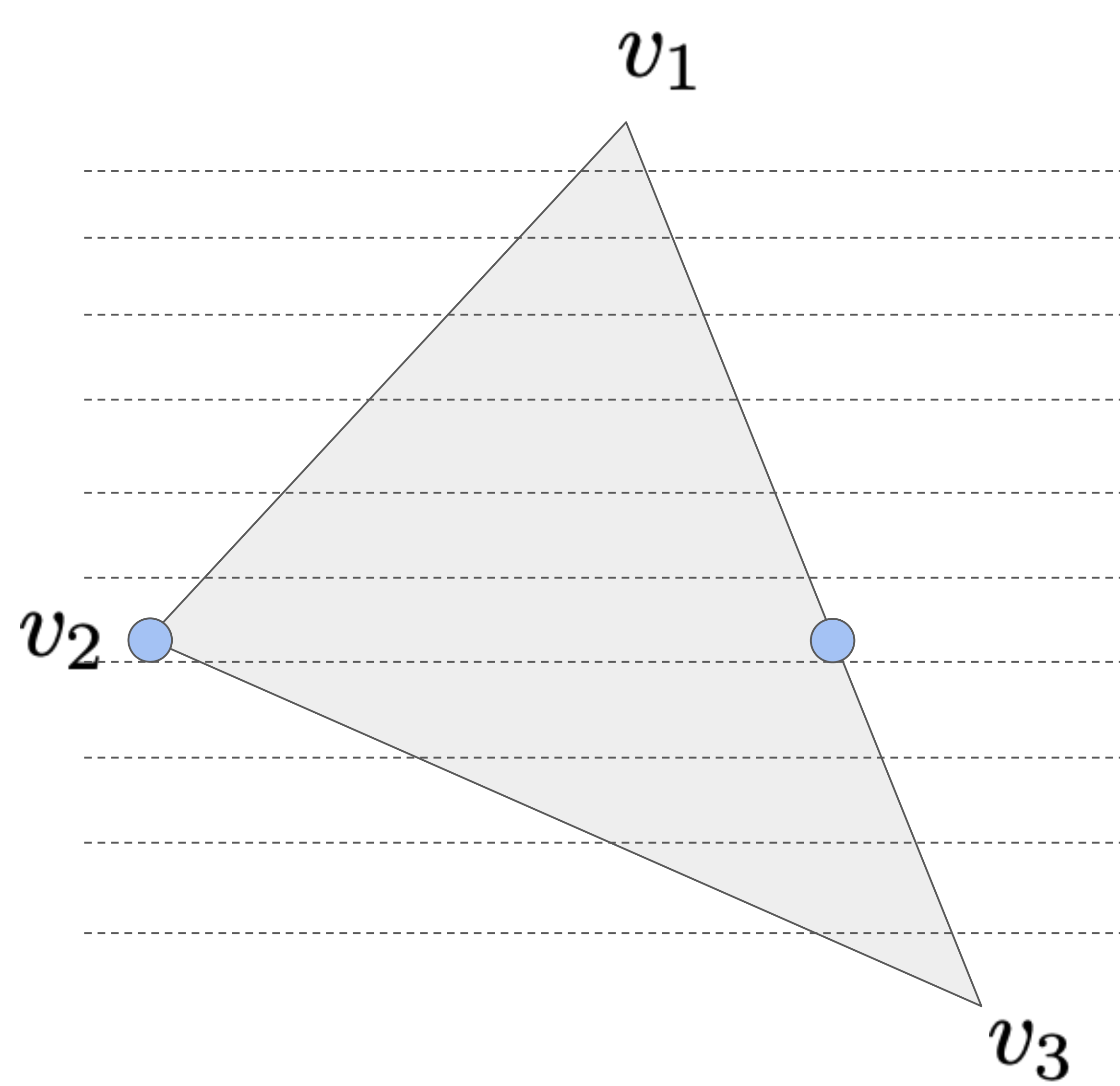
Task 1 g) Bresenham (complete version, cont.)

Putting it all together, draw from left to right:

```
drawLine(p1, p2, color) {
  // TODO: implement Bresenham algorithm
  if ( Math.abs(p2.y - p1.y) < Math.abs(p2.x - p1.x) ) {
    if (p1.x > p2.x) {
      this.drawLineLow(p2.x, p2.y, p1.x, p1.y, color)
    } else {
      this.drawLineLow(p1.x, p1.y, p2.x, p2.y, color)
    }
  } else {
    if (p1.y > p2.y) {
      this.drawLineHigh(p2.x, p2.y, p1.x, p1.y, color)
    } else {
      this.drawLineHigh(p1.x, p1.y, p2.x, p2.y, color)
    }
  }
}
```

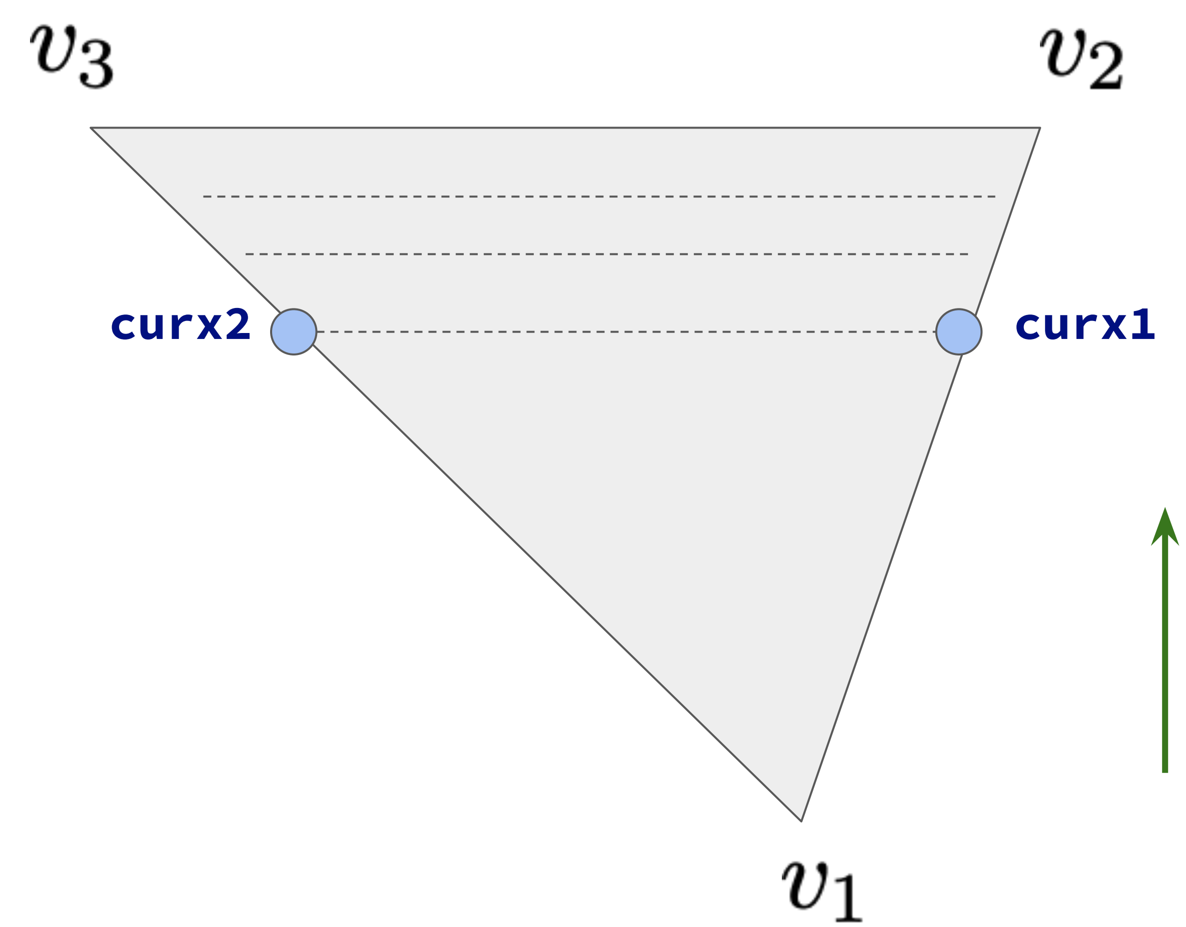
Scan Line Algorithm for Triangles

Basic idea: fill a polygon line by line horizontally or vertically



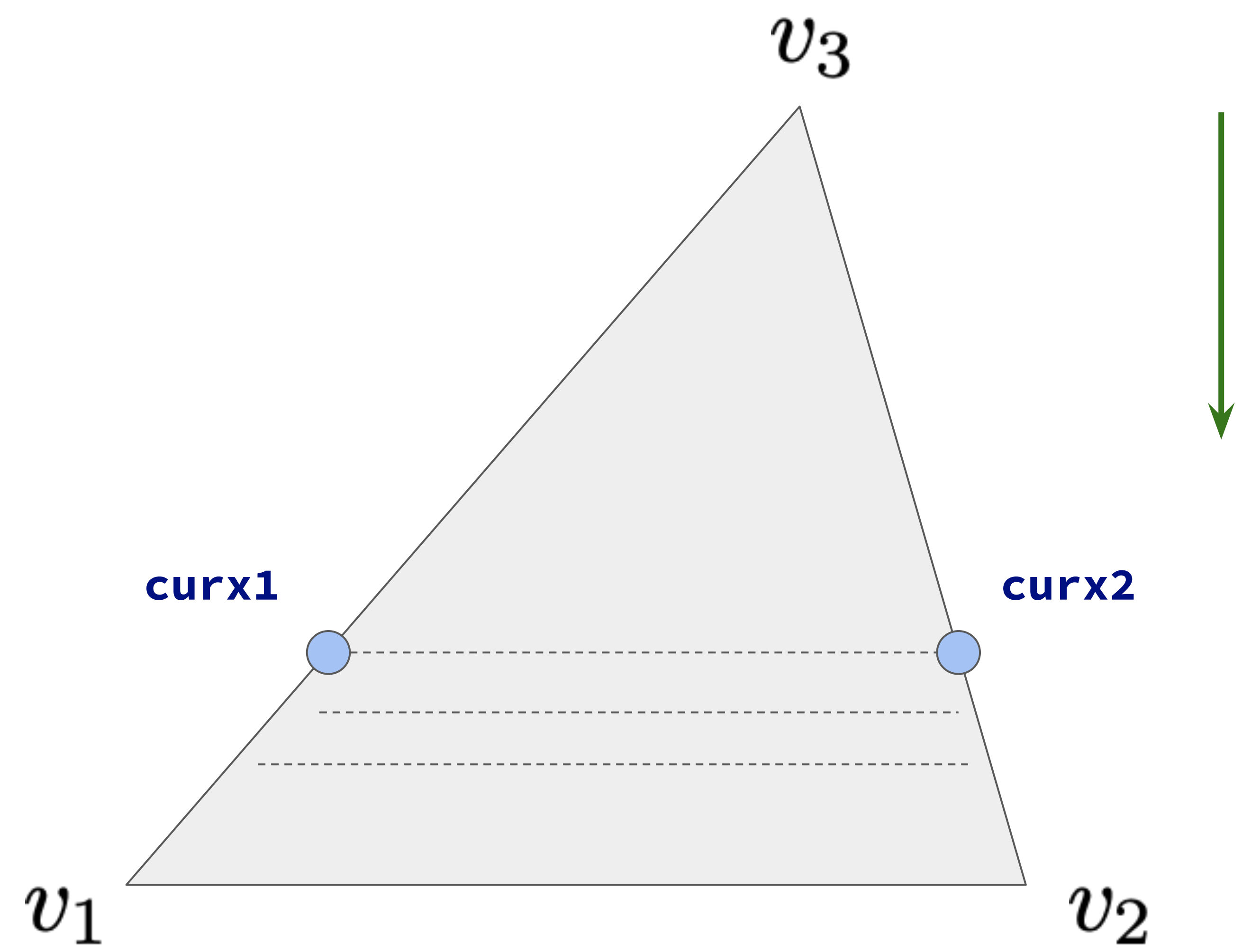
Task 1 g) Scan Line Algorithm for Triangles

```
drawTriangleBottom(v1, v2, v3, color) {
  const invsploe1 = (v2.x - v1.x) / (v2.y - v1.y)
  const invsploe2 = (v3.x - v1.x) / (v3.y - v1.y)

  let curx1 = v1.x
  let curx2 = v1.x

  for (let scanlineY = v1.y; scanlineY <= v2.y; scanlineY++) {
    this.drawLine(
      new Vector2(Math.round(curx1), scanlineY),
      new Vector2(Math.round(curx2), scanlineY), color)
    curx1 += invsploe1
    curx2 += invsploe2
  }
}
```



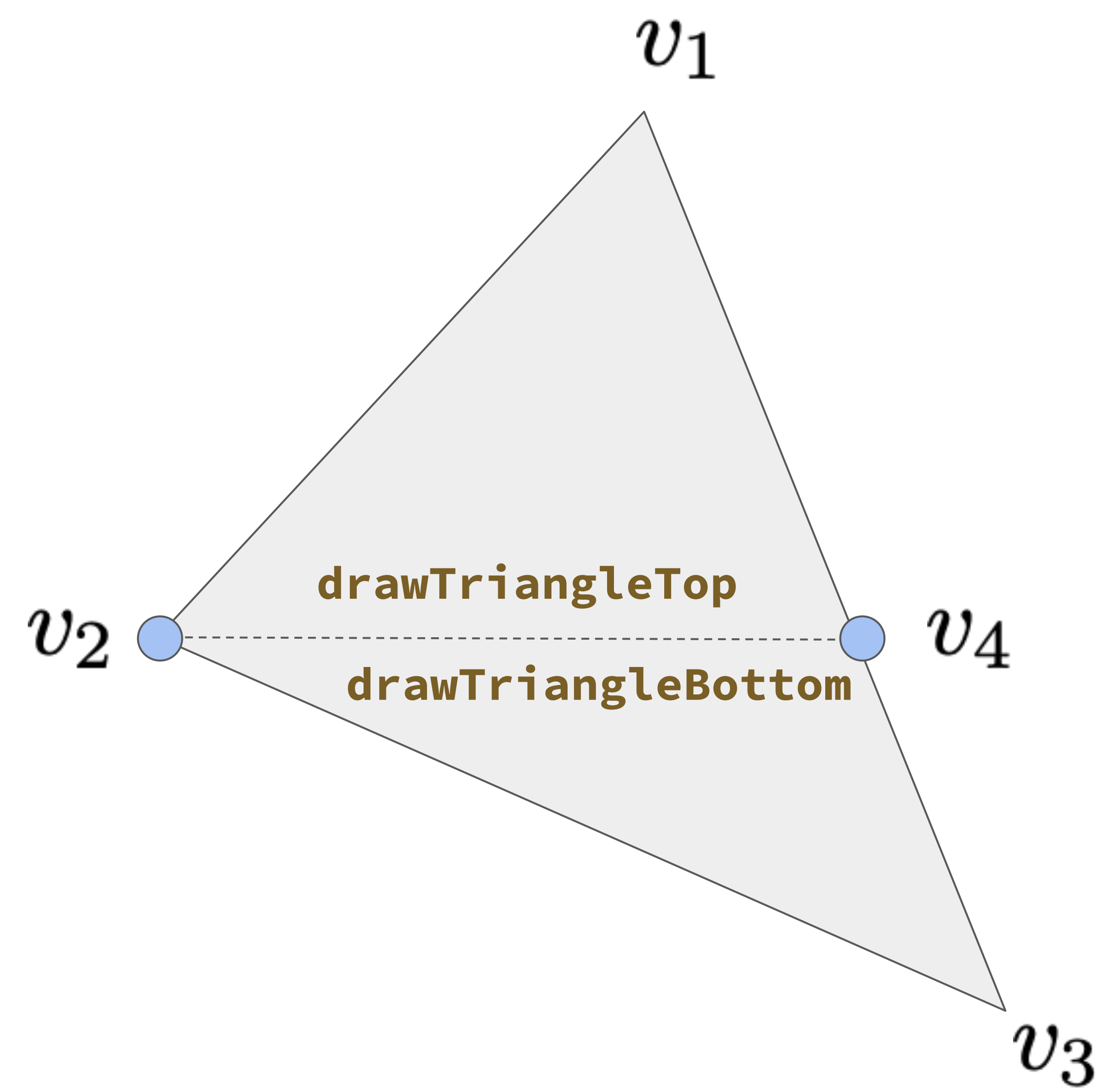
Task 1 g) Scan Line Algorithm for Triangles (cont.)

```
drawTriangleTop(v1, v2, v3, color) {
  const invsploe1 = (v3.x - v1.x) / (v3.y - v1.y)
  const invsploe2 = (v3.x - v2.x) / (v3.y - v2.y)

  let curx1 = v3.x
  let curx2 = v3.x

  for (let scanlineY = v3.y; scanlineY > v1.y; scanlineY--) {
    this.drawLine(
      new Vector2(Math.round(curx1), scanlineY),
      new Vector2(Math.round(curx2), scanlineY), color)
    curx1 -= invsploe1
    curx2 -= invsploe2
  }
}
```



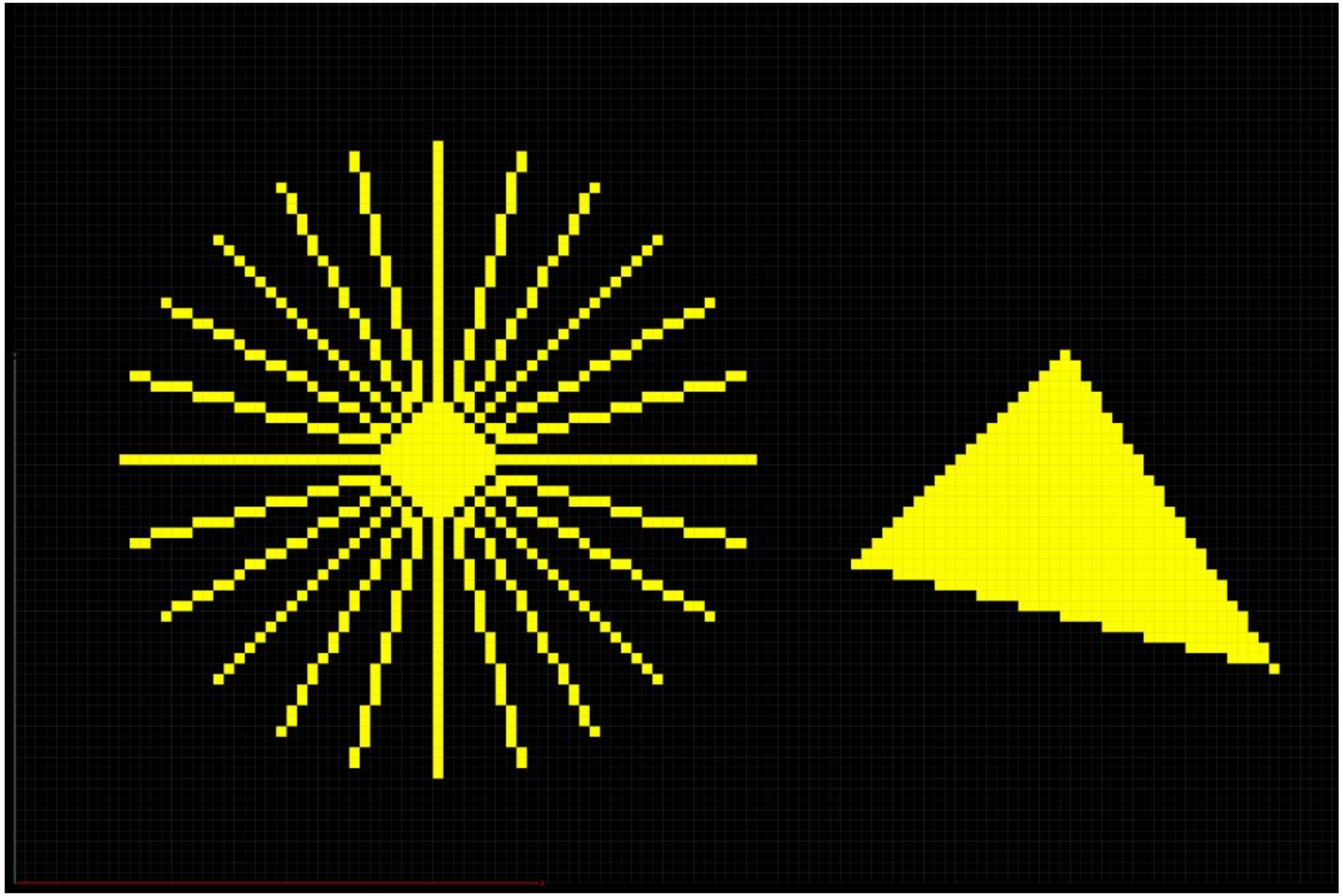
Task 1 g) Scan Line Algorithm for Triangles (cont.)

```
drawTriangle(v1, v2, v3, color) {
  // TODO: implement the scan line algorithm for filling triangles

  // sort three vertices to guarantee v1.y > v2.y > v3.y
  if (v1.y > v2.y && v2.y > v3.y) {}
  else if (v1.y > v3.y && v3.y > v2.y) [v2, v3] = [v3, v2]
  else if (v3.y > v1.y && v1.y > v2.y) [v1, v2, v3] = [v3, v1, v2]
  else if (v2.y > v1.y && v1.y > v3.y) [v1, v2] = [v2, v1]
  else if (v2.y > v3.y && v3.y > v1.y) [v1, v2, v3] = [v2, v3, v1]
  else if (v3.y > v2.y && v2.y > v1.y) [v1, v3] = [v3, v1]

  if (v2.y == v3.y) {
    this.drawTriangleBottom(v1, v2, v3, color)
    return
  }
  if (v1.y == v2.y) {
    this.drawTriangleTop(v1, v2, v3, color)
    return
  }

  const v4 = new Vector2(v1.x + ((v2.y - v1.y) / (v3.y - v1.y)) * (v3.x - v1.x), v2.y)
  this.drawTriangleTop(v2, v4, v1, color)
  this.drawTriangleBottom(v3, v4, v2, color)
}
```



Caution: order matters (why?)

Task 1 g) Final



Q: What's wrong with this picture??

Tutorial 5: Rasterization

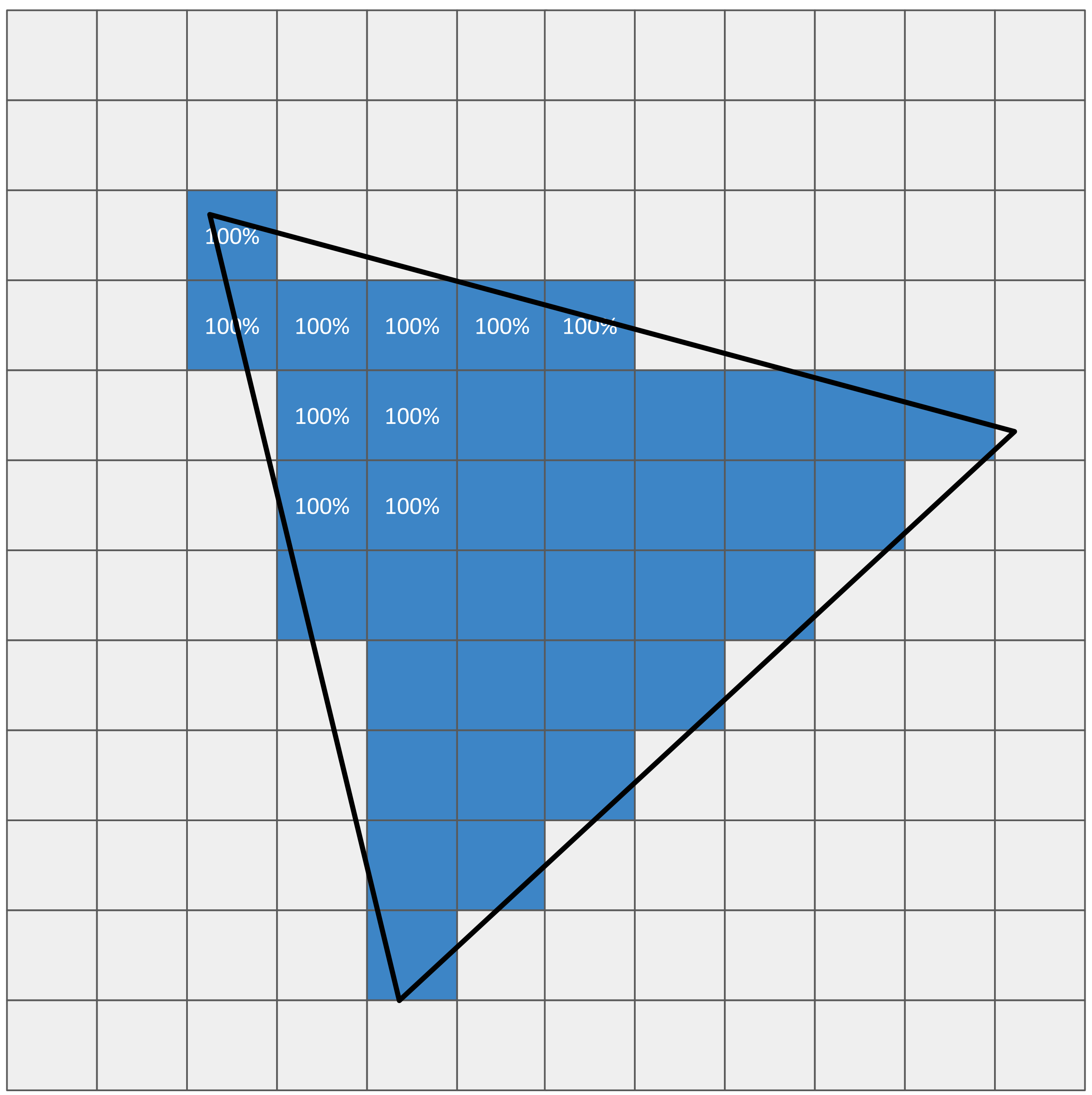
- Culling
- Clipping
- Frame/Depth Buffer
- Drawing
- **Antialiasing**
- OpenGL Shading Language (GLSL)

Task 1 h) and i) Point *Aliasing*

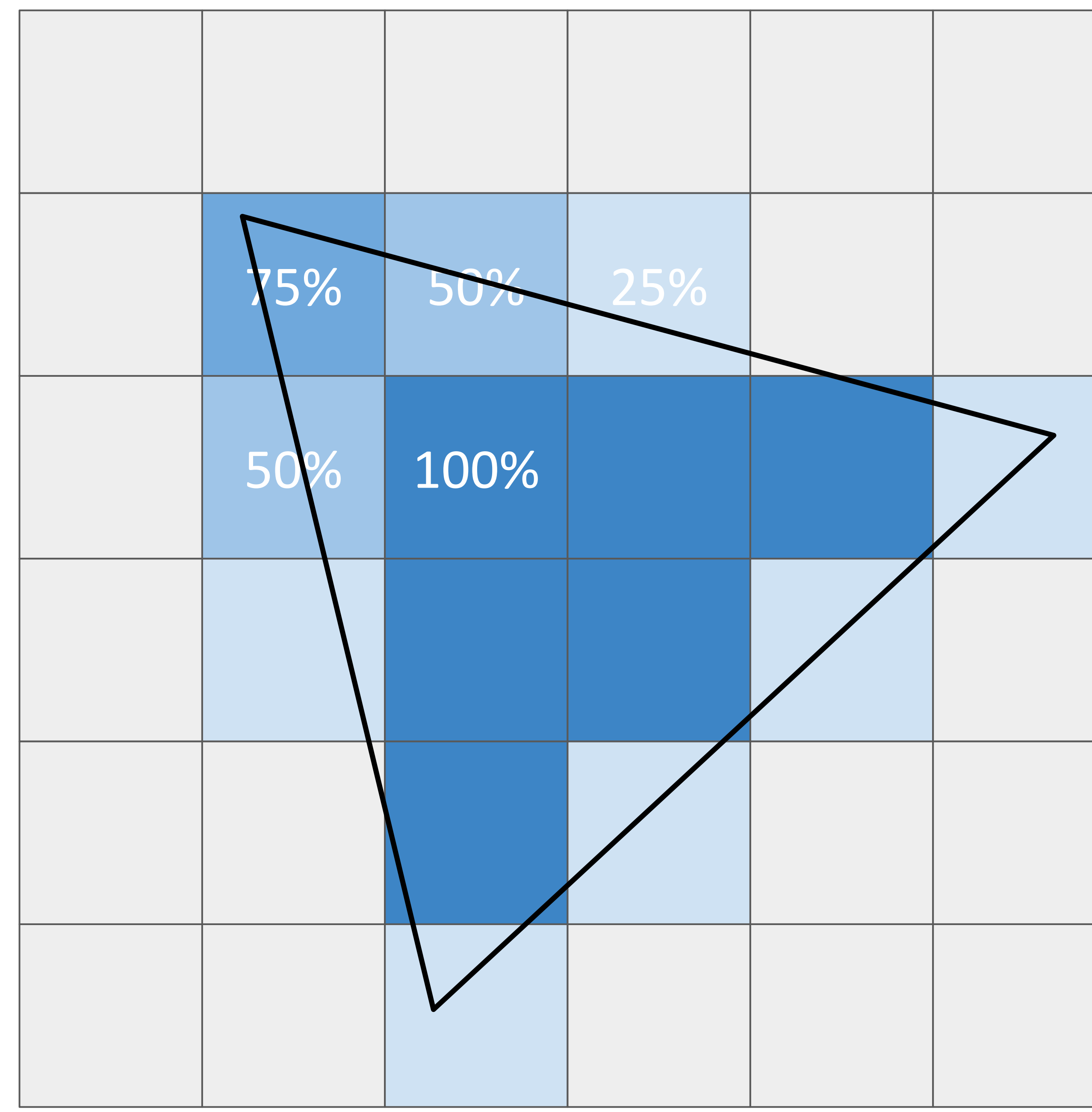
- Bresenham algorithm introduces the fragment *aliasing* issue
- Xiaolin Wu's Antialiasing Approach
    - Check lecture slides
    - A replacement of Bresenham for antialiasing
    - Much slower compare to the Bresenham

Super Sampling Antialiasing (SSAA)

Super sampling antialiasing (SSAA): Sampling high resolution samples then render in a lower resolution, e.g. Multisample Antialiasing (MSAA):



4x4 Super sampling



Averaging down

Antialiasing Today

Q: What's the cost of using MSAA?

The antialiasing methods that appear in many video games:

- Fast Approximate Antialiasing (FXAA, 2009)
- Temporal Antialiasing (TXAA, 2012)

Antialiasing Today (cont.)

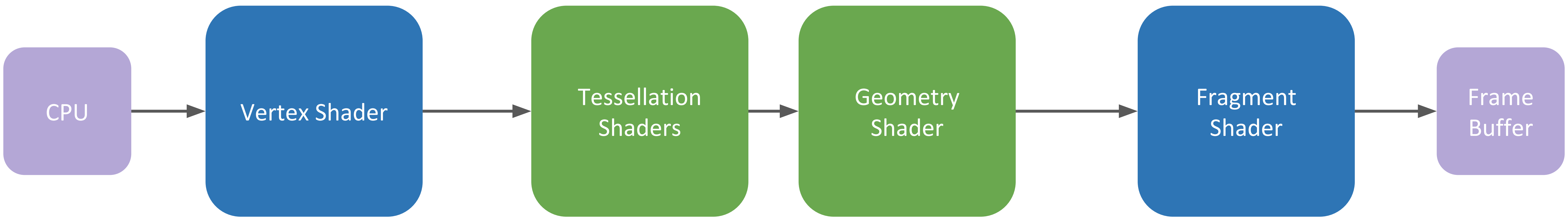
Deep Learning Super Sampling (DLSS 2.0, 2020)

Tutorial 5: Rasterization

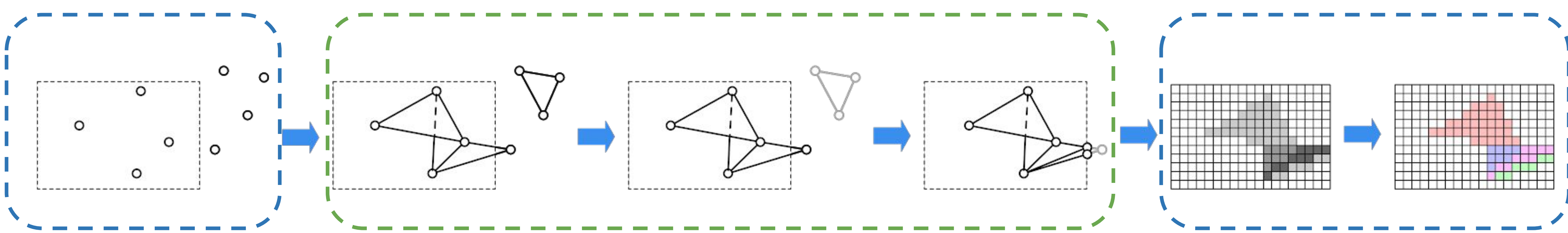
- Culling
- Clipping
- Frame/Depth Buffer
- Drawing
- Antialiasing
- OpenGL Shading Language (GLSL)

OpenGL Shading Language (GLSL)

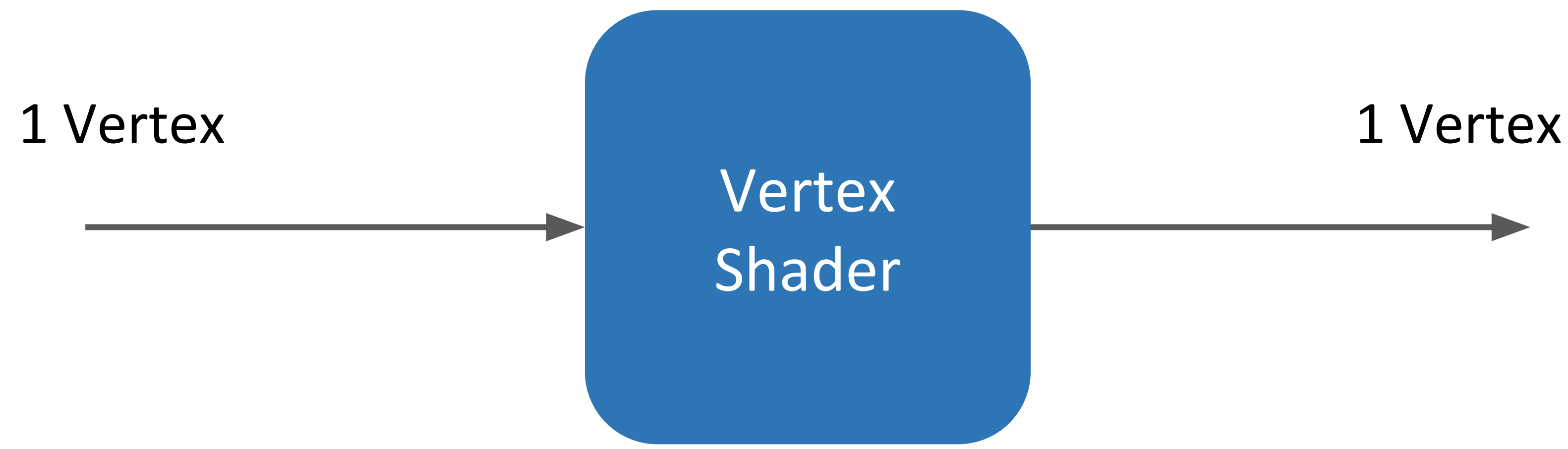
- High-level language for programming programmable stages of graphics pipeline
- Vertex and fragment shaders
  - Manipulation of the rendering pipeline for vertices and fragments



Example:



Task 2 a) Vertex Shader



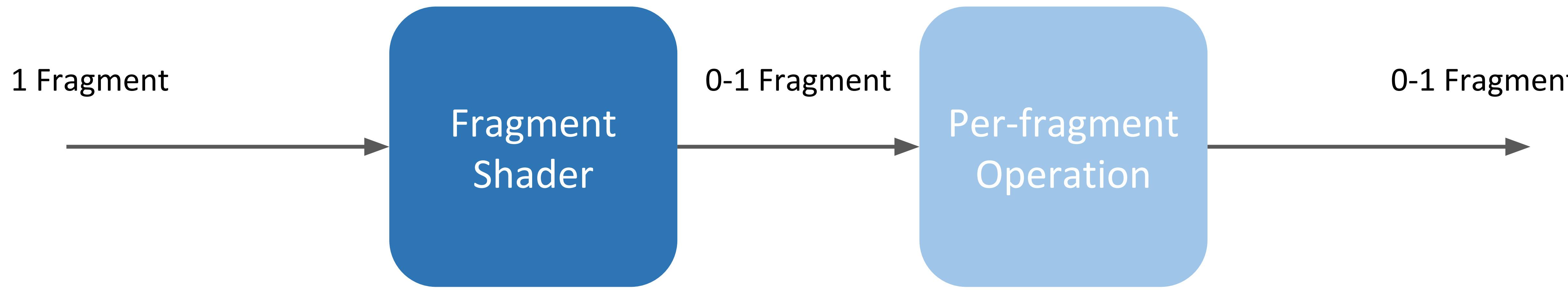
- Transformation of single vertices and their attributes (e.g. normals, ...)
    - No vertex generation
    - No vertex destruction (handled by clipping)
- Calculation of all attributes that remain constant per vertex
    - Saves computing time compared to the Fragment Shader
    - e.g. lighting by vertex (old-fashioned)
- Set attributes to be interpolated per fragment
    - e.g. normals for per-pixel lighting

Minimum Vertex Shader (WebGL 2)

```
#version 300 es
precision highp float;

in vec3 position;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;

void main() {
   gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
}
```

- **gl_Position**: Built-in output attribute for Vertex Shader (required)
- **projectionMatrix**: Perspective/Orthographic Projection
- **modelViewMatrix**: Model and View Transformation
- **vec4(position, 1.0)**: Model Position

Task 2 a) 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
- Input attributes are interpolated within the primitive (can be turned off)
- Fragments can be discarded: `discard`
- Fragment operations: Tests, blending and etc.

Minimum Fragment Shader (WebGL 2)

```
#version 300 es
precision highp float;

out vec4 out_frag_color;

void main() {
   out_frag_color = vec4(1.0, 1.0, 0.0, 1.0); // yellow
}
```

- `out_frag_color` (self-defined) specifies the color (rgba) of a fragment
- The same color is applied to each pixel

Task 2 a) Compute Shader

- Allows general, parallel calculations on the GPU
  - Examples: Physics calculations, particle systems, fluid or substance simulations.
- 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.

Communication with Shaders

- In one direction: OpenGL→Shader
- Shaders have access to parts of the OpenGL state (e.g. lighting parameters)
- User-defined variables: Uniforms, Attributes, IN/OUT

Task 2 b) Uniforms

- Parameters that are the same for many/all vertices/primitives are defined, they are identified via their GLSL variable names (analogous to attributes)
- Each variable is assigned a "location" (index)
  - compare strings more efficiently than with every change
- Can be read in vertex and fragment shaders (read-only)

Task 2 c) Attributes

- Global variables that can be different for each vertex (e.g. normal vector)
- Read-only, only available in Vertex Shader
- Definable in program code

Task 2 d) Out variables

- Set by the Vertex Shader (per vertex) as output
- They are interpolated by the rasterizer
- If they are read by the fragment shader (per fragment, IN variable): Access to interpolated vertex data (e.g. color)
- Starting with OpenGL 3.0 or WebGL 2.0 previously "varyings" (WebGL 1.0)
  - Safari doesn't support WebGL 2.0 (see Appendix)

Task 2 e) and f)

e) **`gl_Position`**: *must* be written in the vertex shader.

Determines the position of the vertices, otherwise cannot continue to the subsequent stages of the pipeline.

f) **`out`** (in Fragment Shader): stores the color of a fragment.

Task 2 g) three.js construction

```
export default class Shader extends Renderer {
 constructor() {
   super()
   // TODO: 1. create a geometry, then push three vertices
   const tri = new Geometry()         0                    1                    2
   tri.vertices.push(new Vector3(-5, -3, -10), new Vector3(0, 5, -10), new Vector3(10, -5, -10))
   // TODO: 2. create a face for the created geometry
   const face = new Face3(0, 2, 1)
   face.vertexColors = [
     new Color(0x3399ff),
     new Color(0x00ffff),
     new Color(0x5500ee)
   ]
   tri.faces = [face]

   // TODO: 3. create a mesh with the geometry that you created in above,
   // then pass the loaded vertex and fragment shader to ShaderMaterial.
   // Enable vertexColor parameter to pass color from threejs to
   // the fragment shader.
   const mesh = new Mesh(tri, new ShaderMaterial({
     vertexShader: vert, fragmentShader: frag, vertexColors: true,
   }))
   // TODO: 4. add the created mesh to the scene
   this.scene.add(mesh)
 }
}
```

Caution: Back-face culling
Q: Where is the camera?
Q: What if you set 0, 1, 2?

Task 2 g) GLSL shaders

vert.glsl

```
#version 300 es

precision highp float;

// define the out to transmit the vertex color to
// the subsequent shaders
out vec3 vColor;

void main() {
   // TODO: scale x by 1.5, y by 0.5, and z by 2.0
   gl_Position = projectionMatrix * modelViewMatrix * vec4(
       position.x*1.5,
       position.y*0.5,
       position.z*2.0,
       1.0
   );
   // TODO: set the vColor out to the color we recieved
   // from the three.js code
   vColor = color;
}
```

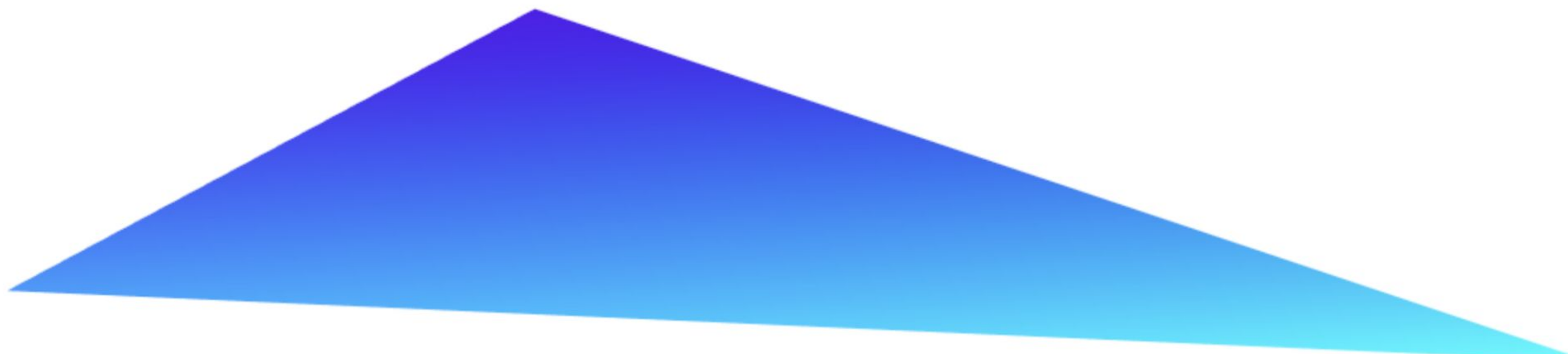
ShaderMaterial built-in
Not in RawShaderMaterial

frag.glsl

```
#version 300 es

precision highp float;

out vec4 outColor;

// TODO: define the in to receive
// the (interpolated) vertex color
// from the previous shaders
in vec3 vColor;

void main() {
   outColor = vec4(vColor, 1.0);
}
```

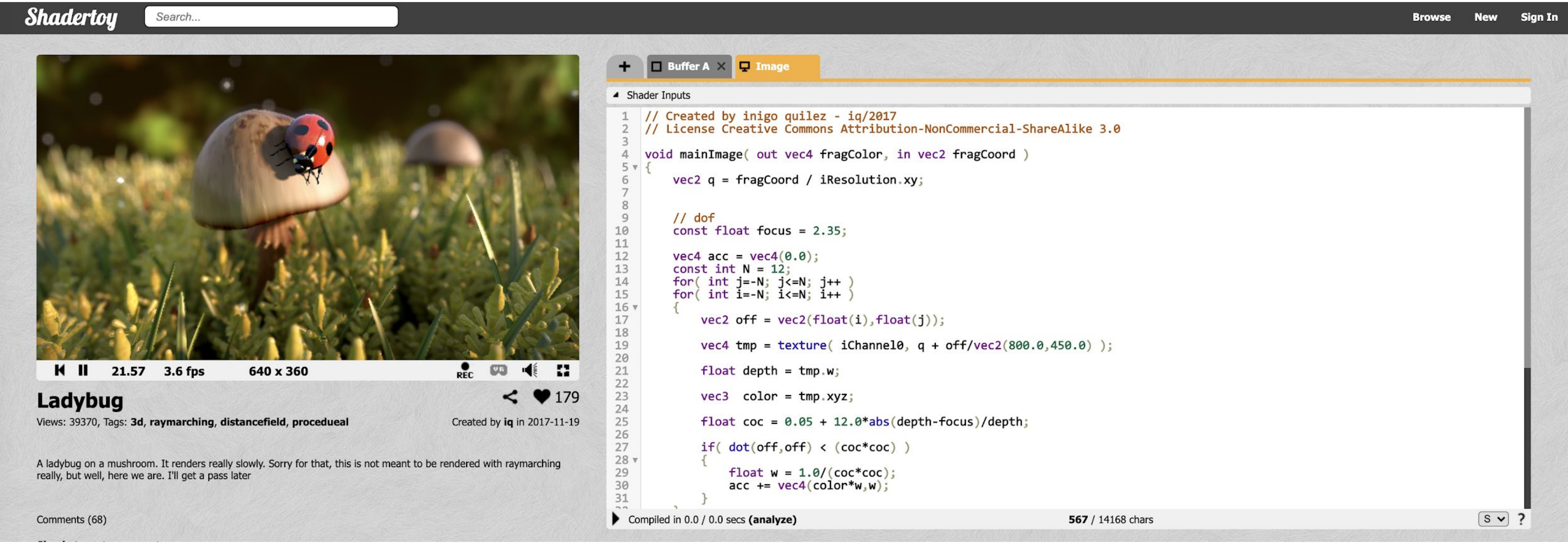
Color flow: **THREE.Color → ShaderMaterial color → VertexShader vColor → Fragment Shader vColor → Fragment Shader outColor → Display**

Task 2 g) Final



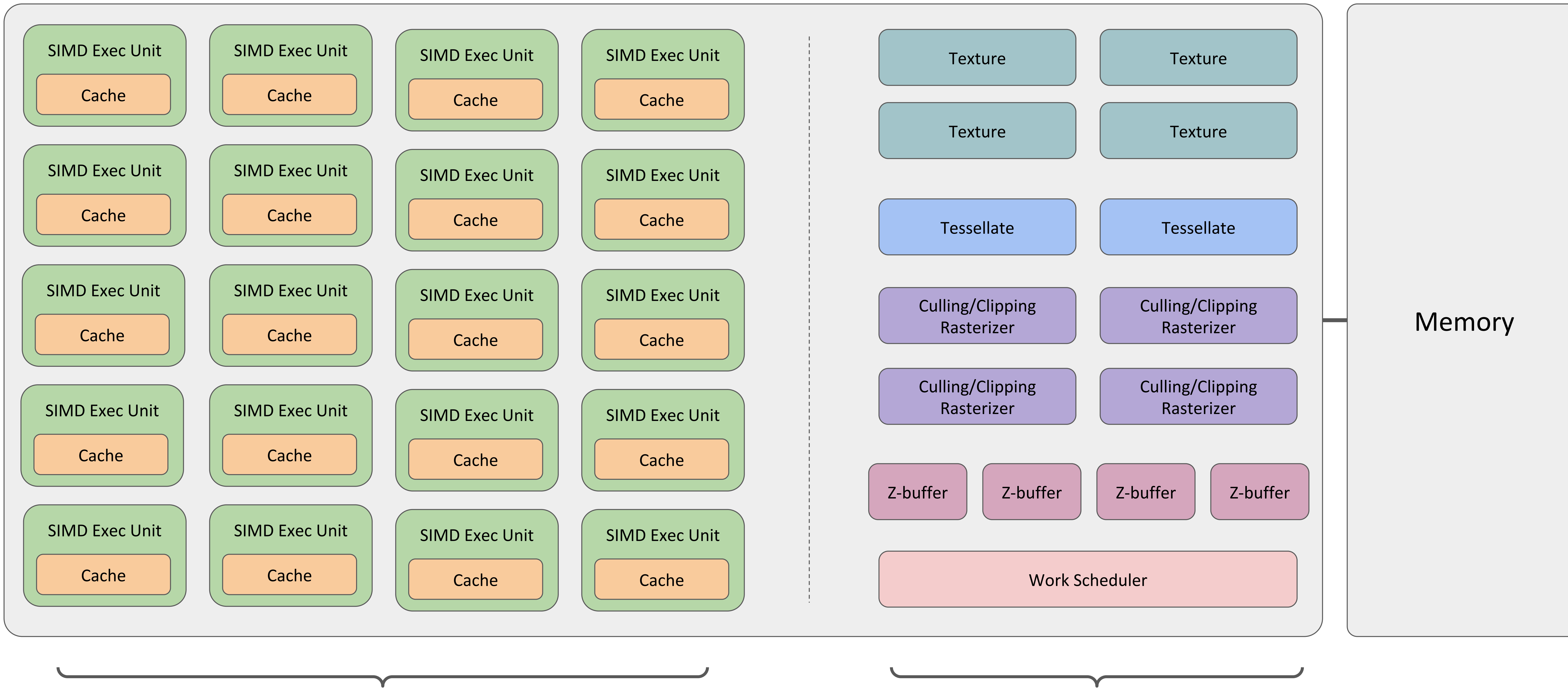
Shaders are powerful!

- Shaders can do more than you might think
- ~800 lines of code:



https://www.shadertoy.com/view/4tByz3

Executing Shaders on a Multi-core Processor (GPU)

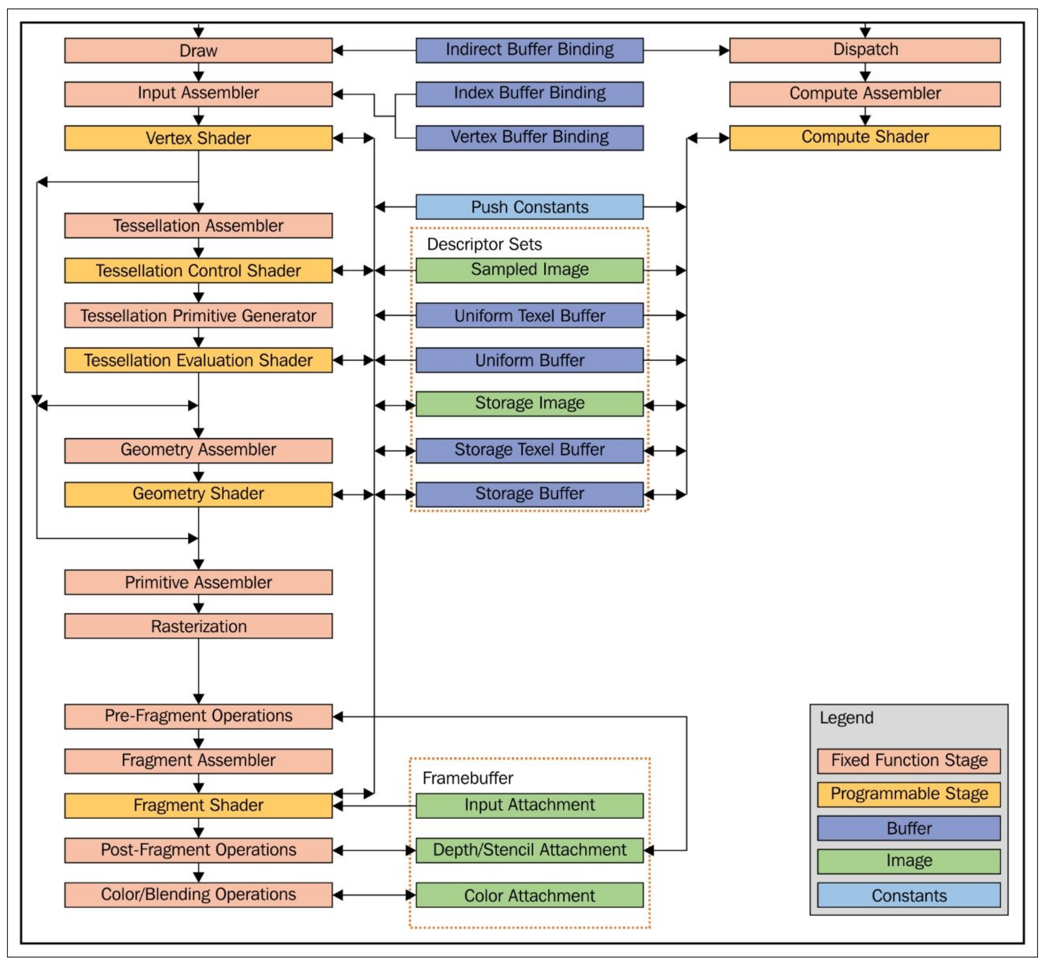


Cores for executing shader programs, in parallel

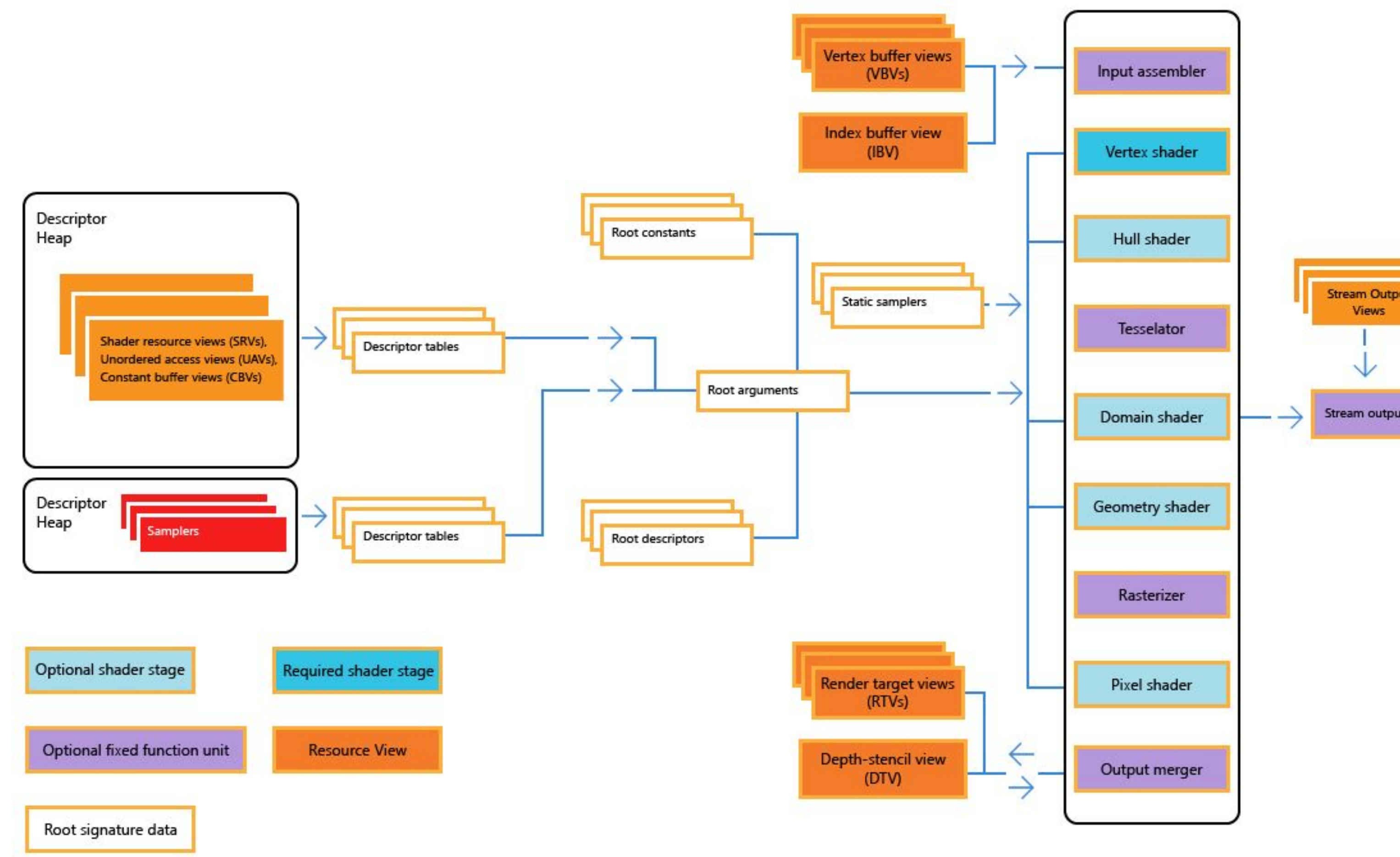
Graphics-specific fixed functions and compute resources

(Modern) Graphics APIs/Pipelines

- Modern graphics APIs are much more complex than what you learned from this course
- API changes fast but fundamental principles live long (Think about the Bresenham)



https://subscription.packtpub.com/book/application_development/9781786469809/8/ch08lvl1sec50/getting-started-with-pipelines

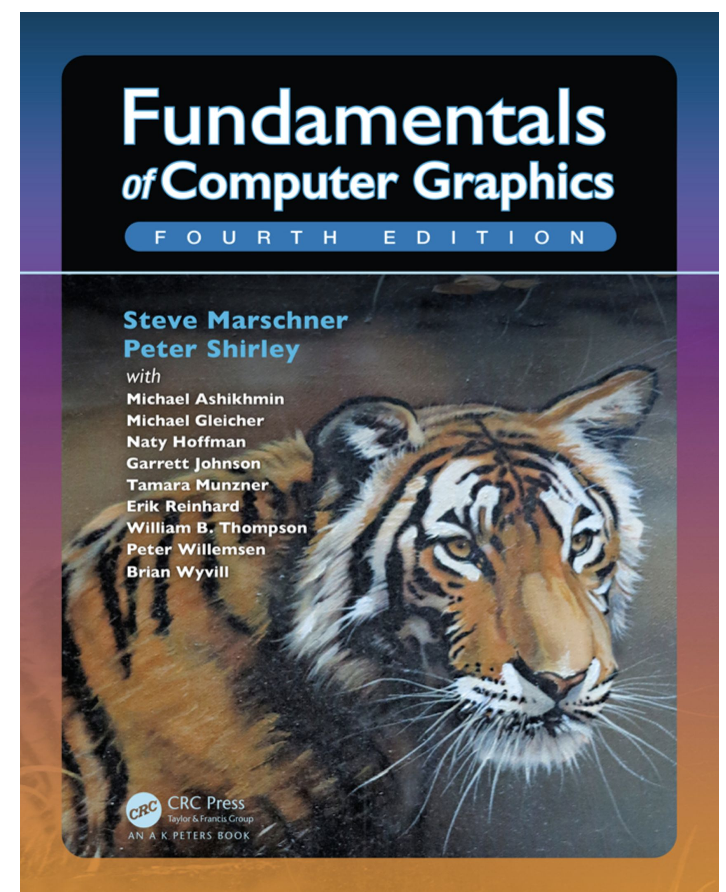


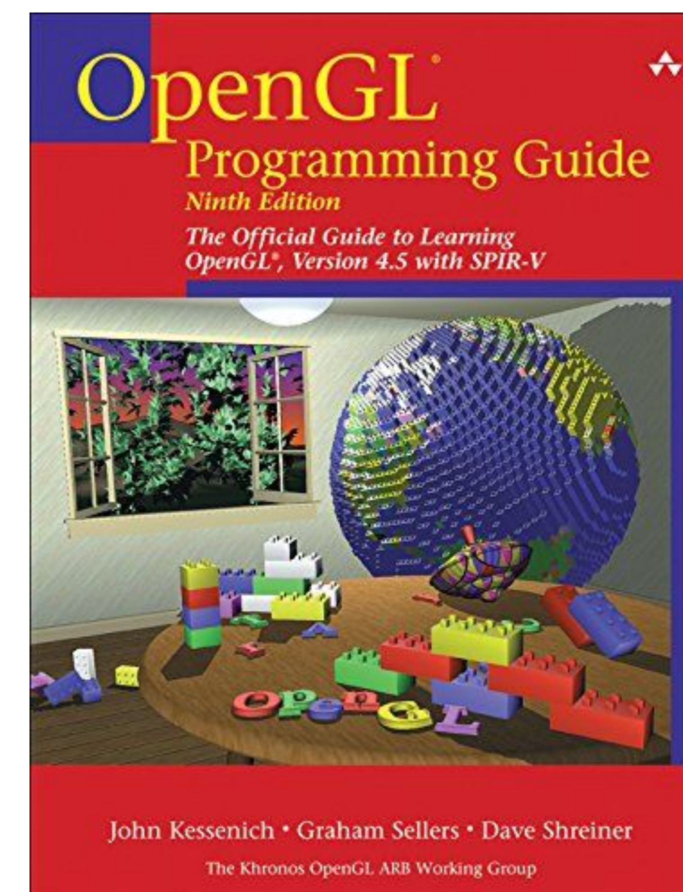
https://docs.microsoft.com/en-us/windows/win32/direct3d12/pipelines-and-shaders-with-directx-12

- How much do I have to know about graphics APIs (e.g. OpenGL) for this course?
  - You should be able to write GLSL shaders that can work with three.js.

Take Away

- The rasterization pipeline is the most important concept in classic computer graphics
- Almost all real-time rendering (e.g. video games) applications benefit from it
- Graphics APIs (e.g. OpenGL) evolve more lightweight over the years and empower end users to write programmable shaders with the reusable internal rasterizer
- You have enough knowledge to implement your own rasterizer (without Graphics APIs)
  - **You don't need a graphics API to do graphics!**
- Check books for the more fundamental details:





Thanks!
What are your questions?

Appendix

If you cannot work with shaders… - Browsers

- Safari doesn't work with WebGL2 (why Apple? why?)
- Use Firefox/Chrome

https://webglreport.com/?v=2

If you cannot work with shaders... - Hardware

- Maybe your graphics card driver is not set properly
- Maybe your hardware is too old

This is very unfortunate :(

```
#if _FP_W_TYPE_SIZE < 32
#error "Here's a nickel kid.  Go buy yourself a real computer."
#endif
```

from https://github.com/torvalds/linux/blob/v5.5/include/math-emu/double.h#L29