

Computer Graphics 1

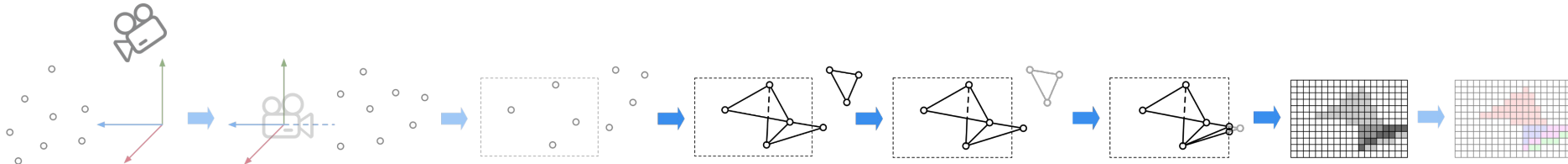
5 Rasterization I

Summer Semester 2021

Ludwig-Maximilians-Universität München

Tutorial 5: Rasterization I

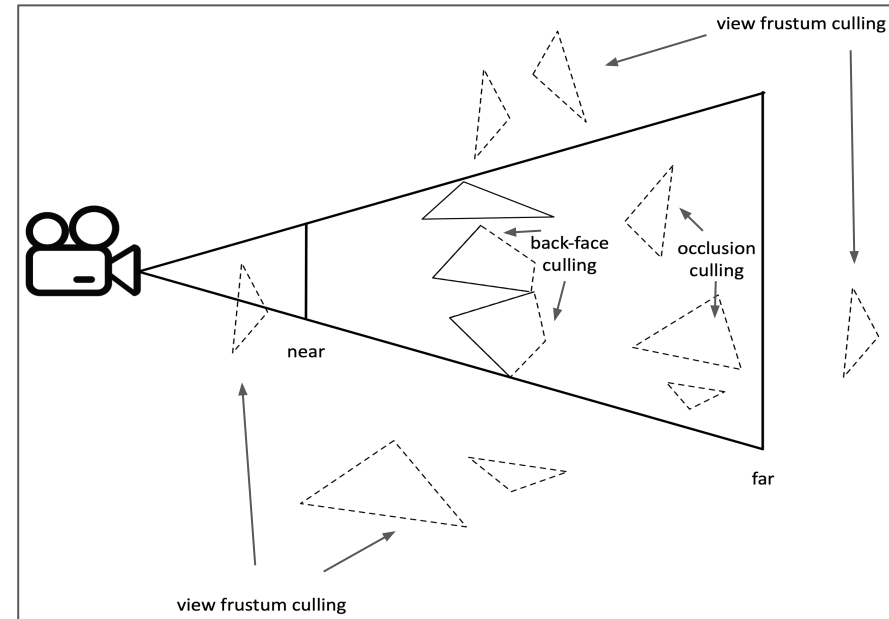
- Spatial Partitions
 - Different Types of Culling
 - AABB and BVH
- Screen-space Buffers
 - Frame Buffer
 - Depth Buffer
- Drawing
 - Bresenham and Scan Line Algorithms



Types of Culling

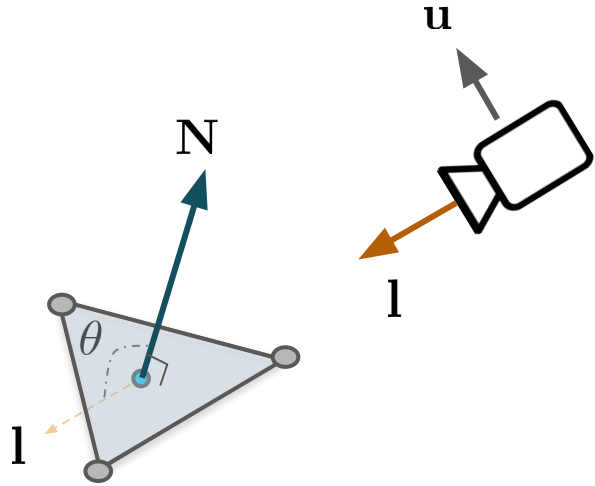
- Backface culling: do not render back faces
- View frustum culling: do not render objects outside of the view frustum
- Occlusion culling: do not render objects behind visible objects

What do we need in order to implement them all?



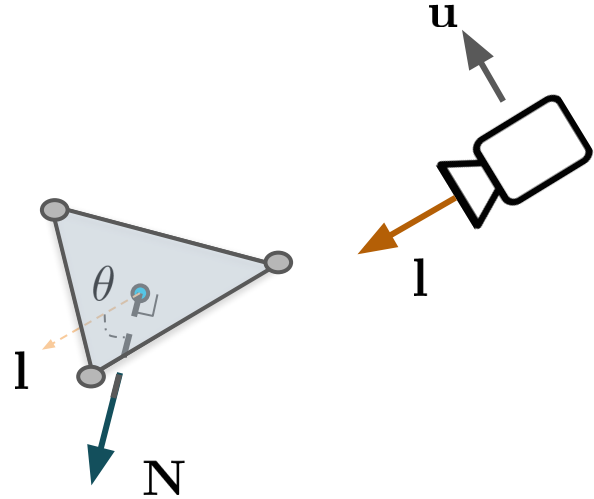
Backface Culling

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



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

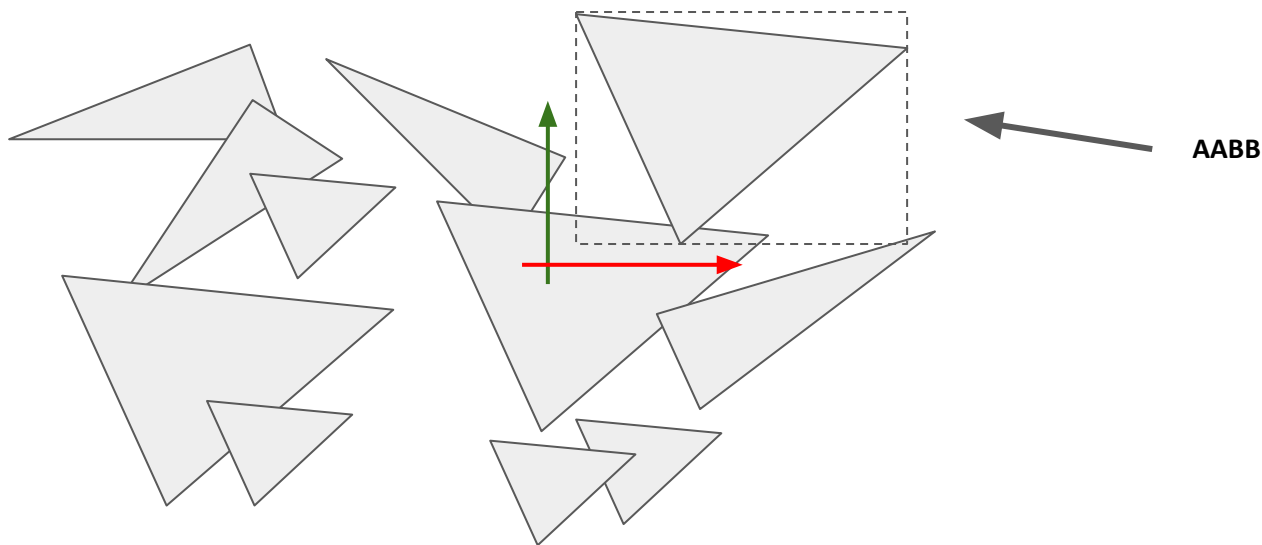
Backface

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

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).

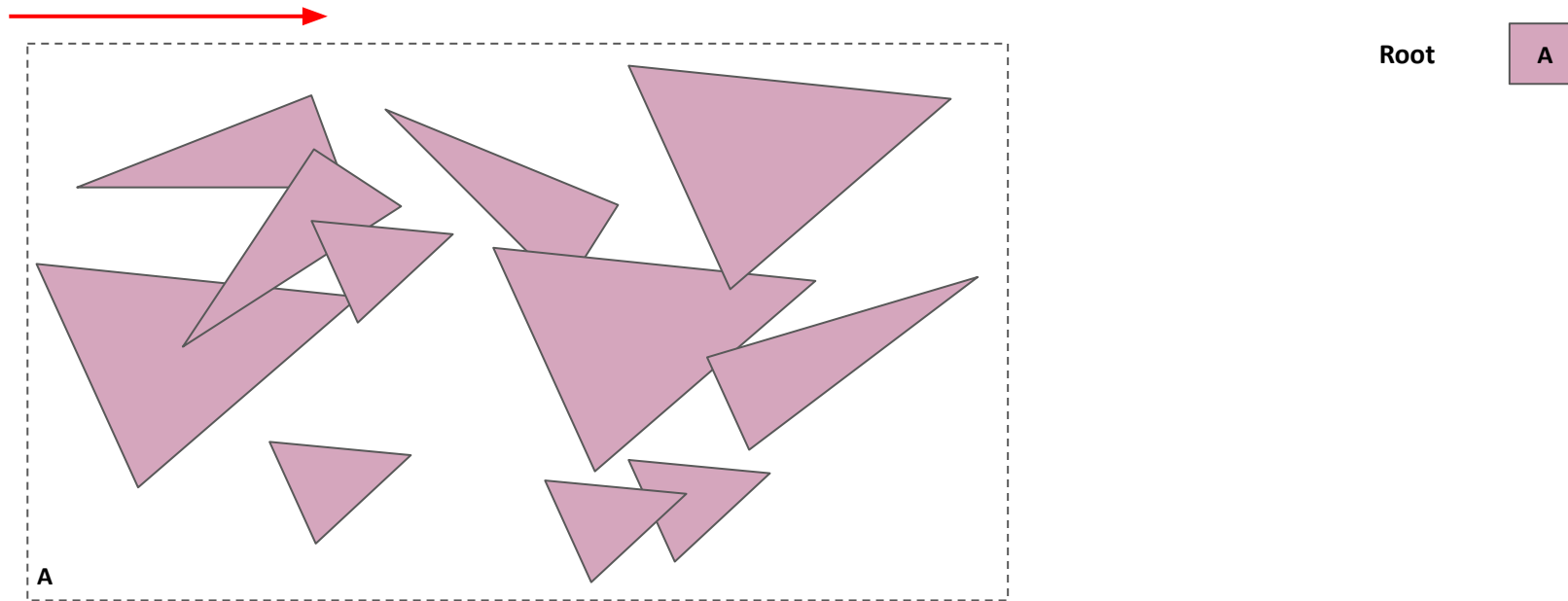
AABB can be represented via two points (x_{\min} , y_{\min} , z_{\min}) and (x_{\max} , y_{\max} , z_{\max})



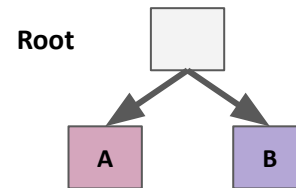
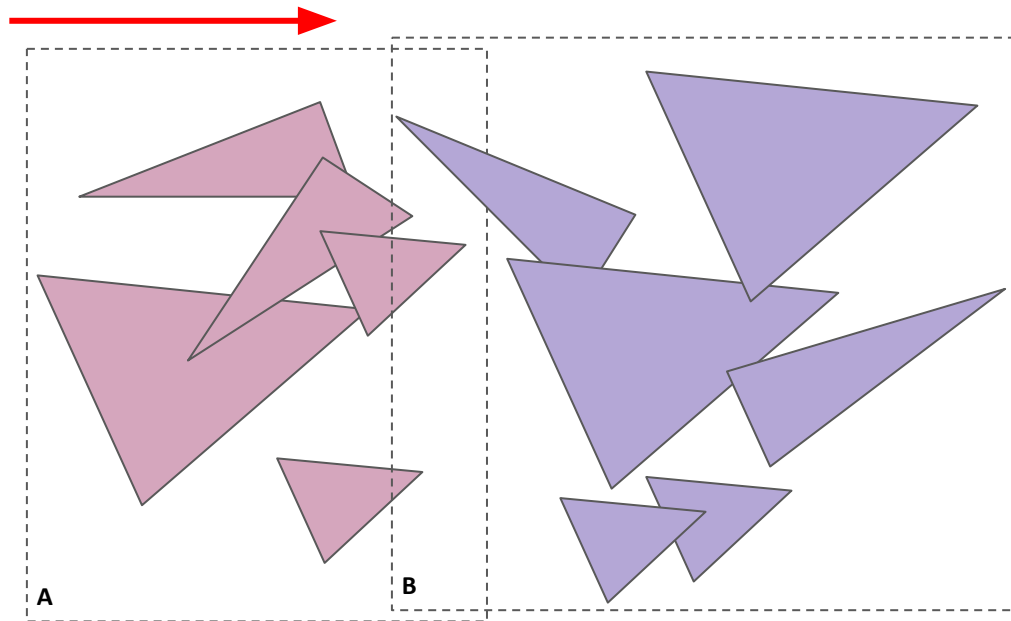
Q: How to compute a minimum AABB for a triangle?

Bounding Volume Hierarchy (BVH)

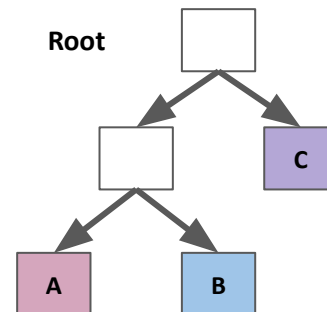
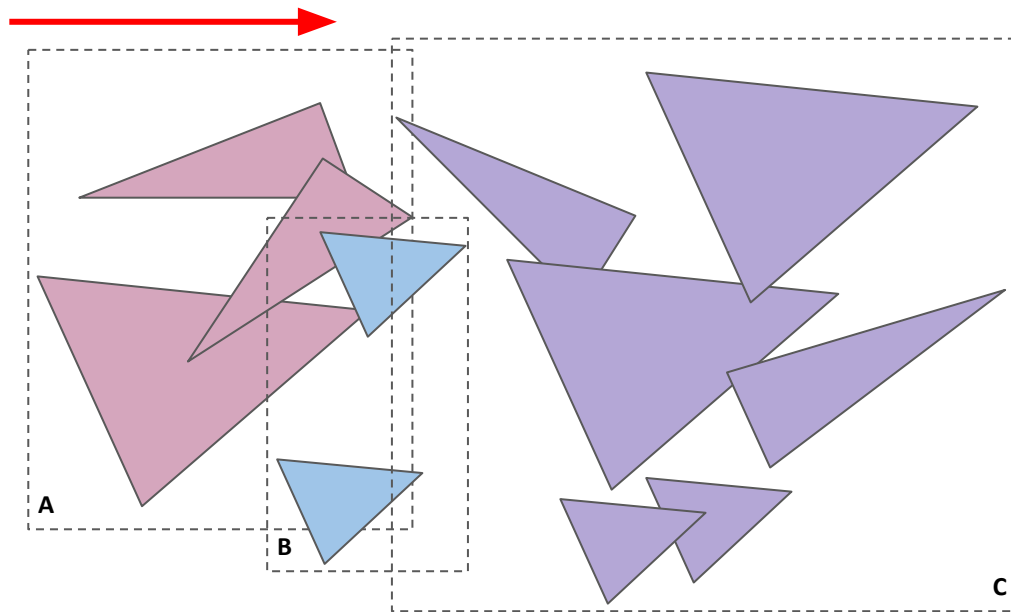
Core idea: split the scene along an axis and divide the number of triangles by density



Bounding Volume Hierarchy (BVH)

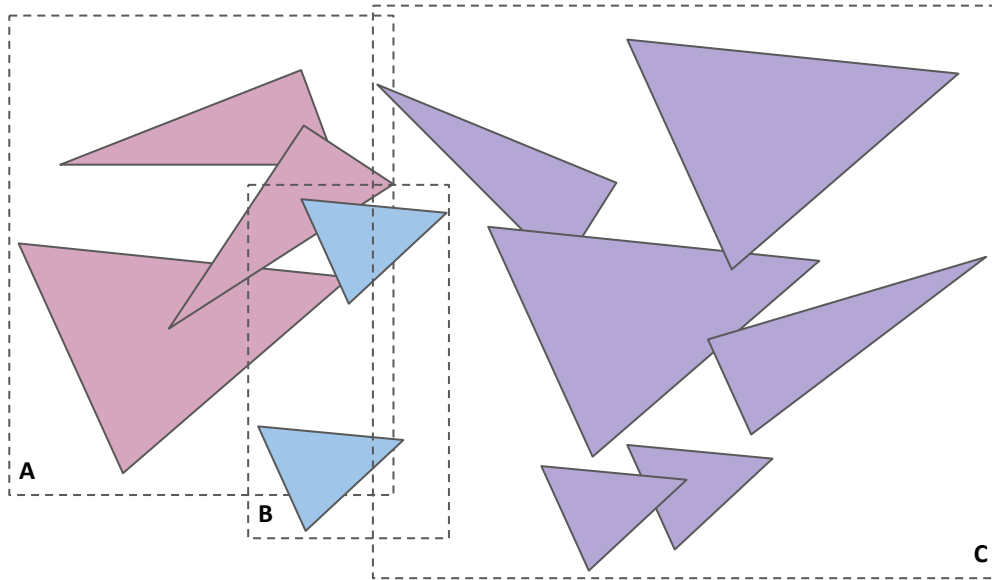


Bounding Volume Hierarchy (BVH)



- Process for constructing a BVH:
 - Compute the bounding box
 - Split the set of objects into two subsets
 - Recompute the bounding boxes
 - Stop when necessary
 - Store objects in each leaf node
 - Similar to a scene graph

Why do we use 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:
 - number of partitions can explode (8^3 ...)
 - An object may occur in multiple partitions
 - Requires an additional intersection check
 - ...
- A recent reason: BVH can be optimized in parallel for ray tracing (later)

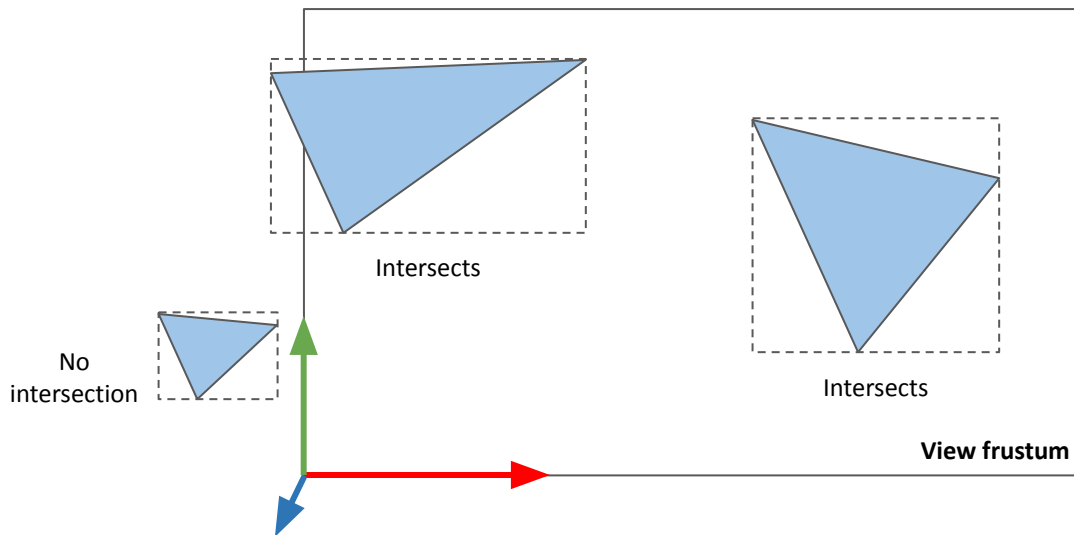
View Frustum Culling

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

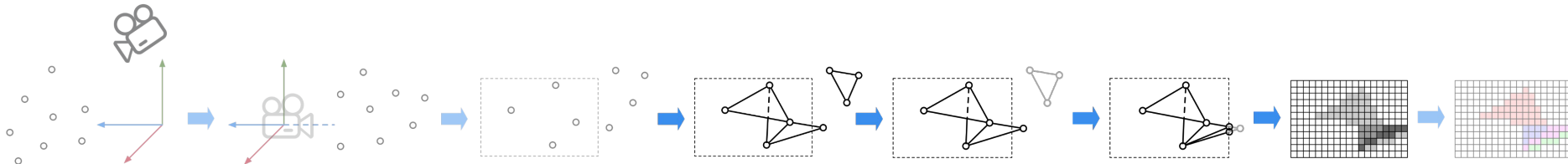
Implementation thinking:

- 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 BVH to test if a group triangles intersects with the view frustum (as optimization)



Tutorial 5: Rasterization I

- Spatial Partitions
 - Different Types of Culling
 - AABB and BVH
- Screen-space Buffers
 - Frame Buffer
 - Depth Buffer
- Drawing
 - Bresenham and Scan Line Algorithms



Screen-space Buffers

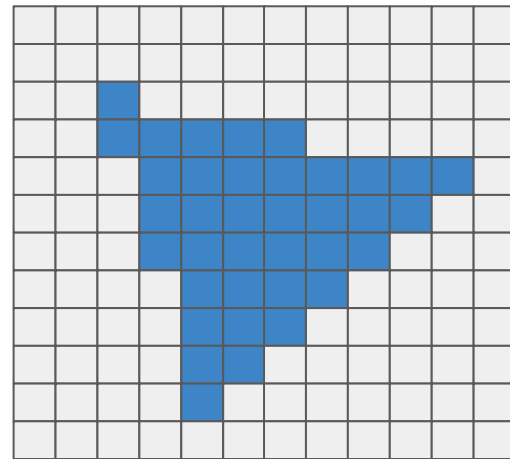
Screen-space (or Image-space) buffers are arrays for storing rendering information, such as pixel color. They are eventually flushed to a monitor.

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

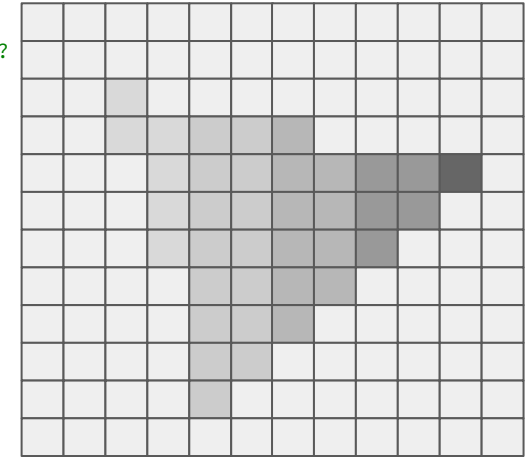
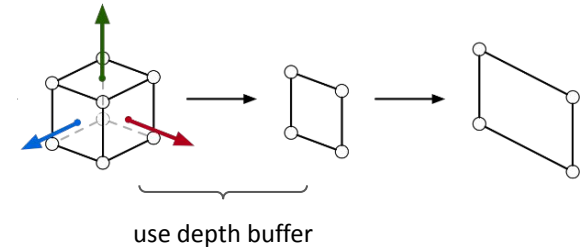
The Painter's algorithm cannot solve the occlusion issue.

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: 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:

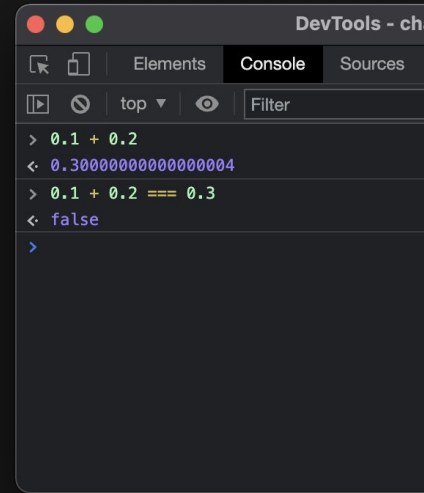
```
0.1+0.2 === 0.3 // false
```

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: 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.

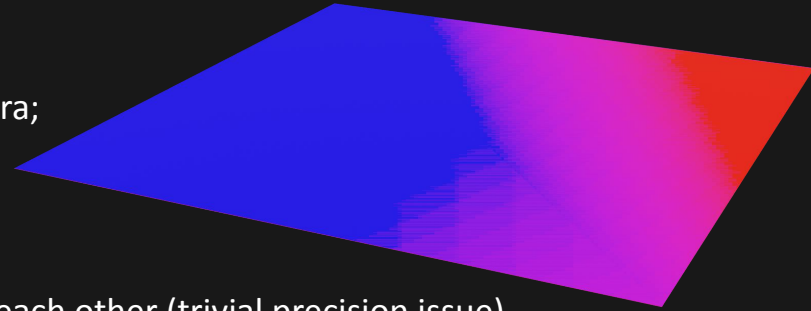
In demos/05-raster1/1-zfighting:

1. Open `planes-case1.blend` in **Blender**

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

2. Open `planes-case2.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




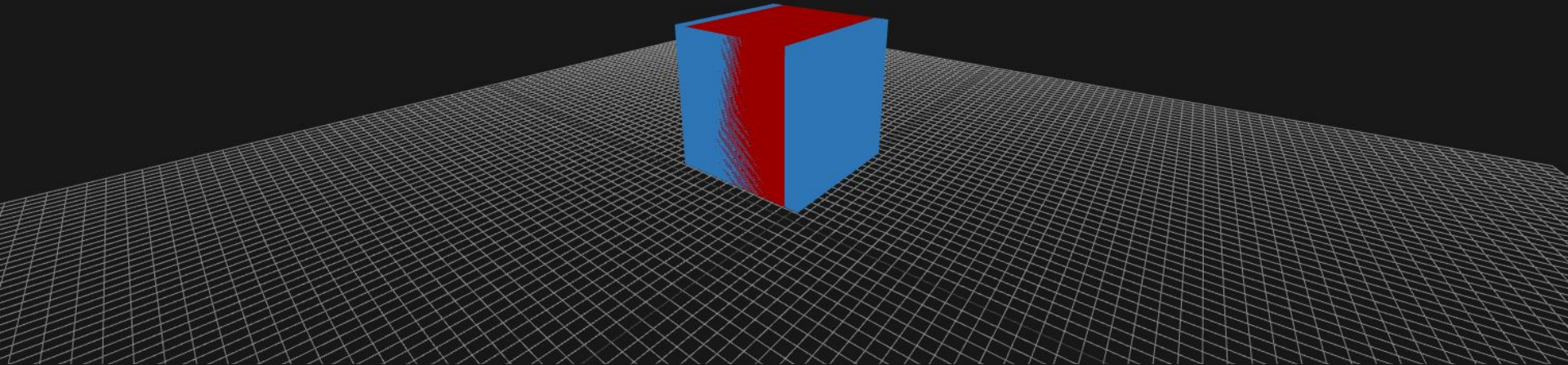
Case 1: two objects are too close, and their z values are fighting with each other (trivial precision issue)

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

Breakout: Experiment Z-fighting Effects

Run code in demos/05-raster1/1-zfighting ([live demo](#))

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? → 

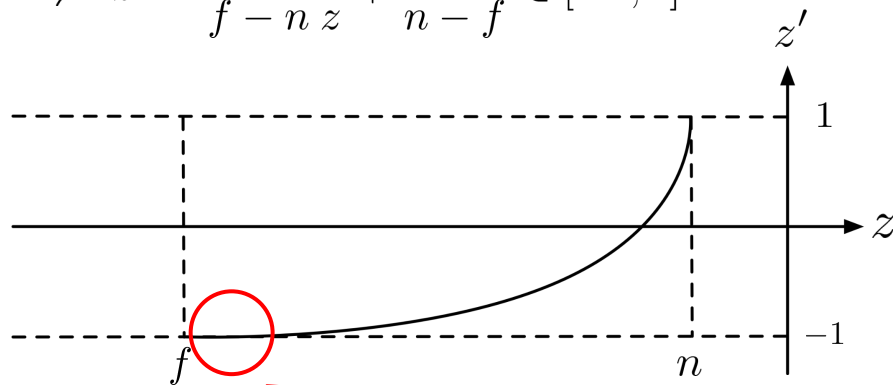


Z-fighting: Far from Viewport (Case 2)

Recall the perspective projection matrix (see [Tutorial 4](#)):

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

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



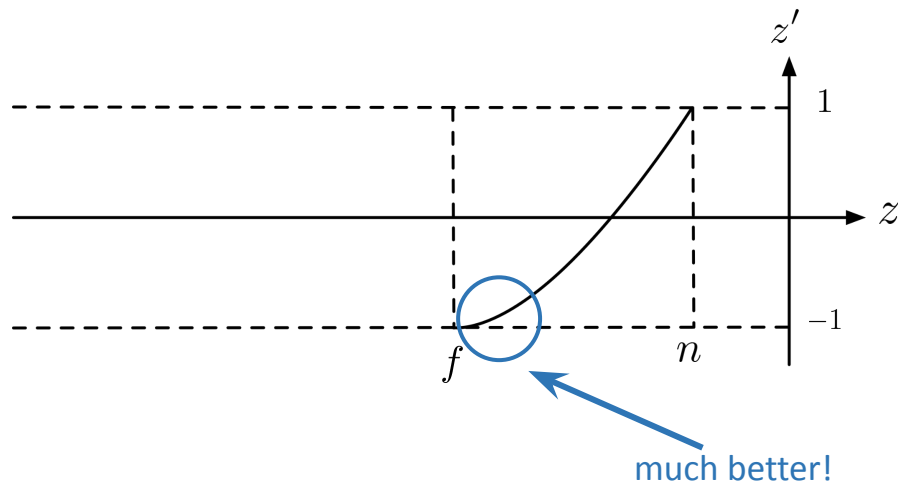
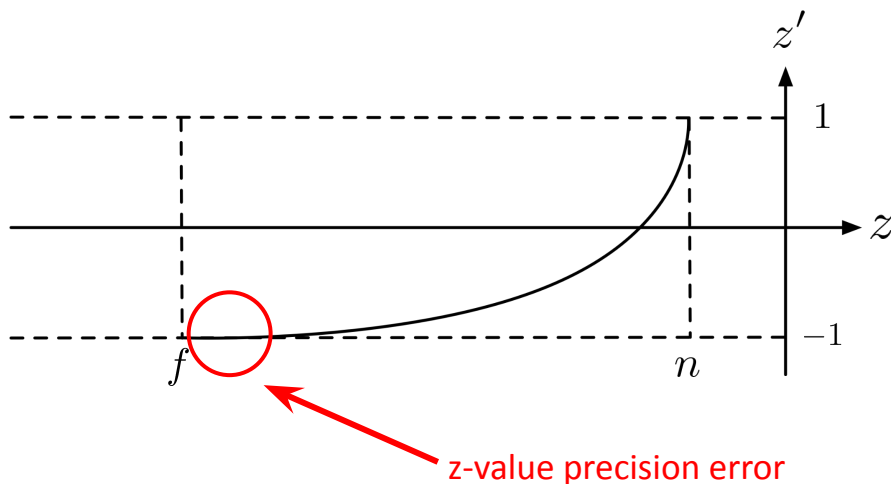
Depth values (after projection) are less accurate when the object is further away from the viewport.

Q: What about orthographic projection?

z-value precision error

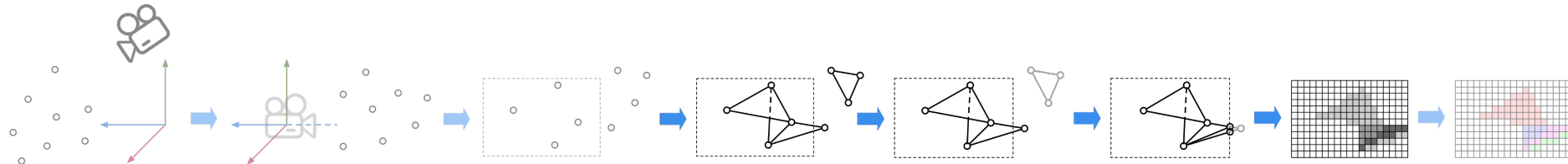
How to avoid Z-fighting?

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 I

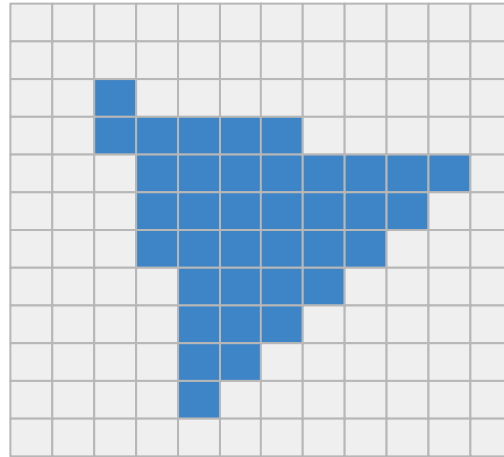
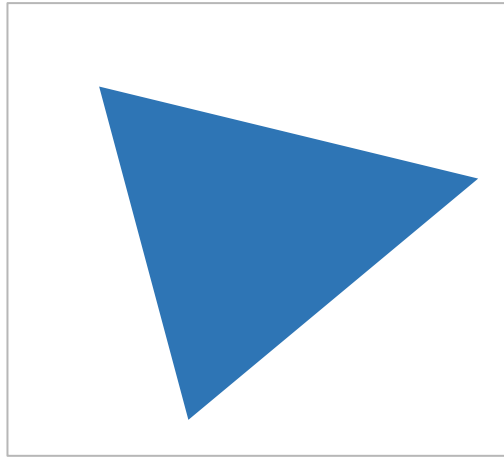
- Spatial Partitions
 - Different Types of Culling
 - AABB and BVH
- Screen-space Buffers
 - Frame Buffer
 - Depth Buffer
- Drawing
 - Bresenham and Scan Line Algorithms



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*



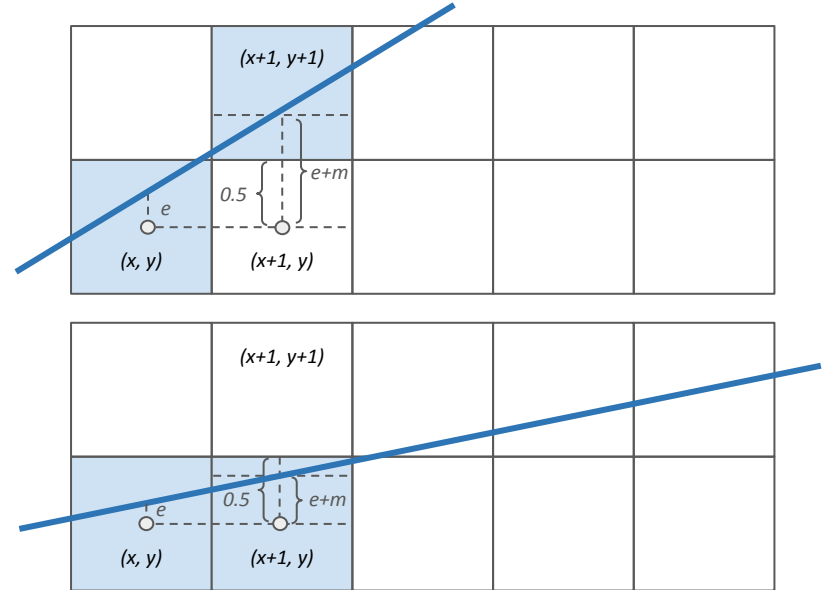
Line Drawing: Bresenham Algorithm

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 $e + m \leq 0.5$ then draw $(x+1, y)$
 - The line is closer to y than to $(y+1)$



Draw A Line from (x0, y0) to (x1, y1), $0 \leq \text{slope} \leq 1$

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 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

```
let e=0, 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 (2*e+dx+D > 0) {
    y += 1
    e -= 1
  }
  e += dy/dx
}
```

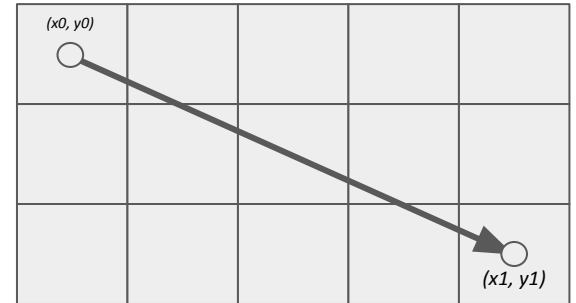
```
let e = 0, dy = y1-y0, dx = x1-x0
for (let x = x0, y = y0; x <= x1; x++) {
  draw(x, y)
  // how to update x and y?
  if (2*e+dx+2*dy-dx > 0) {
    y += 1
    e -= 1
  }
  e += dy/dx
}
```

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



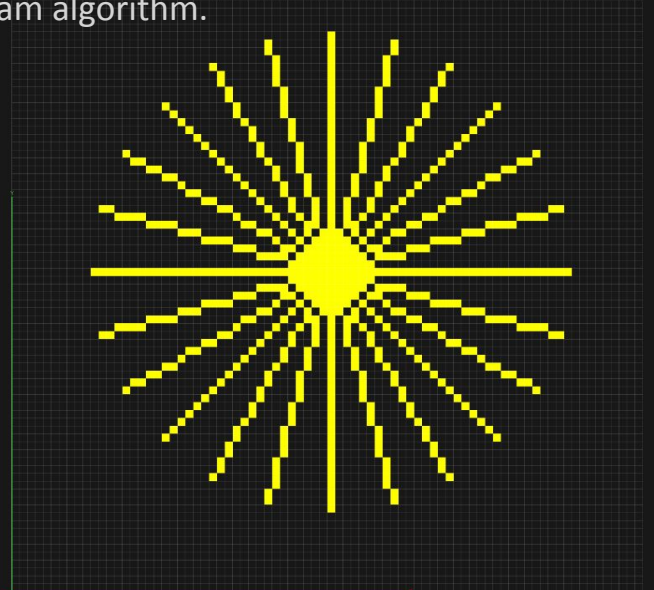
Breakout: Implement Bresenham Algorithm

Enter the folder `demos/05-raster1/2-bresenham` ([live demo](#)).

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

Step 1: Implement the function `drawLine` to sort out the line

```
drawLine(p1: Vector2, p2: Vector2, color: number) {  
  // TODO: implement Bresenham algorithm  
  if (Math.abs(p2.y - p1.y) < Math.abs(p2.x - p1.x)) { // slope less than 1?  
    if (p1.x > p2.x) [p1, p2] = [p2, p1]; // is draw from left to right?  
    this.drawLineLow(p1.x, p1.y, p2.x, p2.y, color);  
  } else {  
    if (p1.y > p2.y) [p1, p2] = [p2, p1]; // is draw from bottom to top?  
    this.drawLineHigh(p1.x, p1.y, p2.x, p2.y, color);  
  }  
}
```



Breakout: Implement Bresenham Algorithm

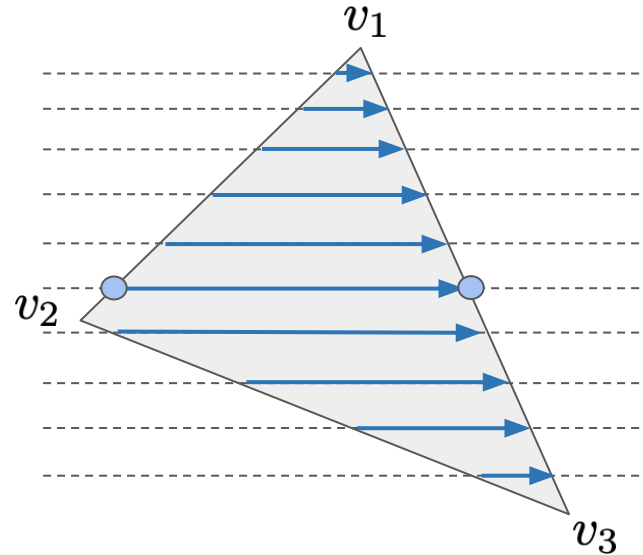
Step 2: Implement the two mentioned functions for drawing the line:

```
// 0 <= |slope| < 1
drawLineLow(
  x0: number, y0: number,
  x1: number, y1: number, color: number) {
  let yi = 1; // default: 0 <= slope < 1
  const dx = x1 - x0;
  let dy = y1 - y0;
  if (dy < 0) { // deal with -1 < slope < 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;
  }
}
```

```
// |slope| >= 1 && dx !== 0
drawLineHigh(
  x0: number, y0: number,
  x1: number, y1: number, color: number) {
  let xi = 1; // default: slope >= 1
  let dx = x1 - x0;
  const dy = y1 - y0;
  if (dx < 0) { // deal with slope <= -1
    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;
  }
}
```

Triangle Drawing: Scan Line Algorithm

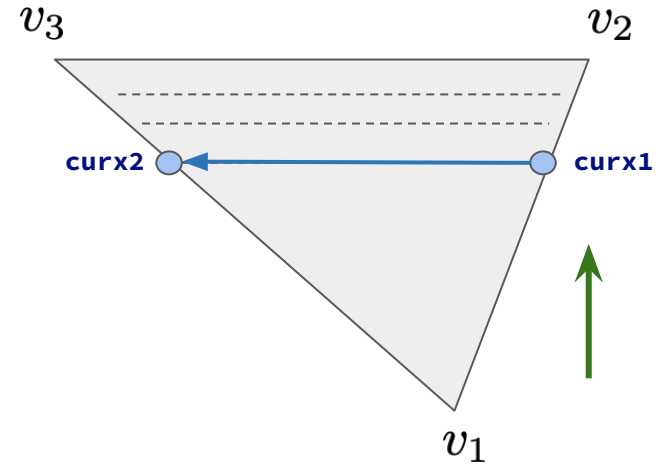
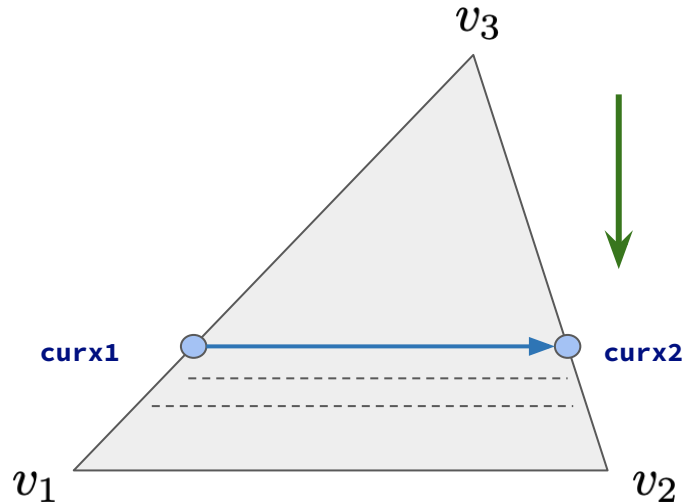
Any polygon can be considered as a group of triangles. To draw a triangle, we can utilize the Bresenham algorithm for the interior of the triangle. Basic idea: fill the triangle line by line horizontally or vertically



Scan Line Algorithm for Triangles

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

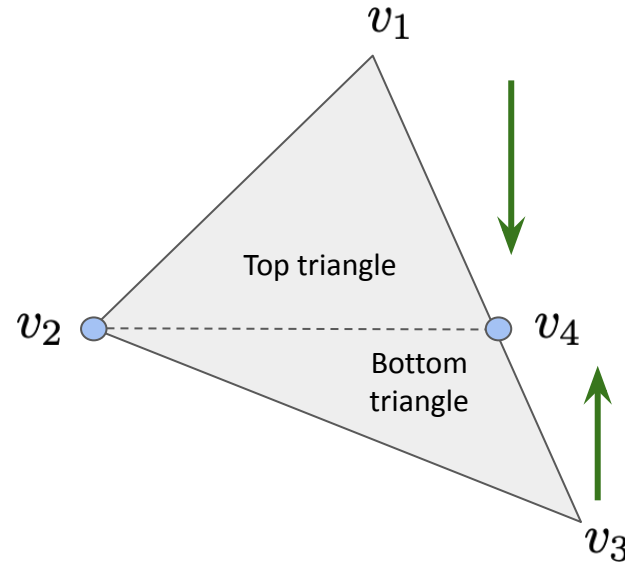
Depending on where the third vertex lies, the scanline moves in different y-directions.



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.

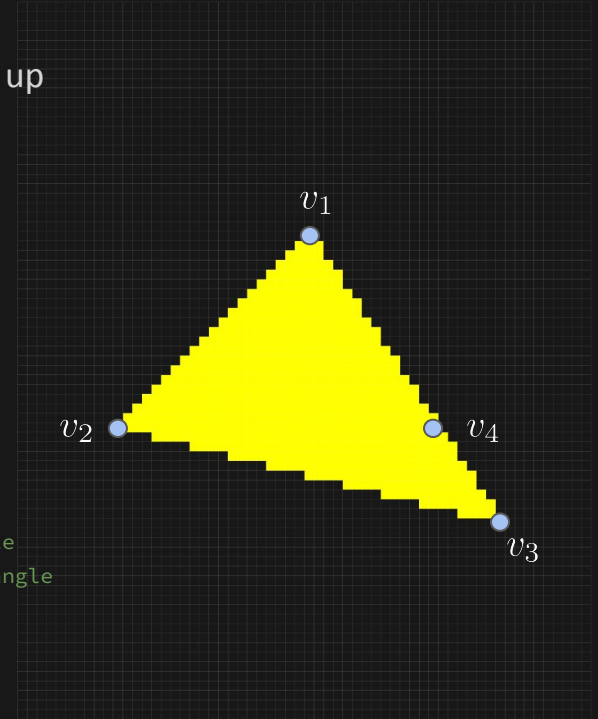


Breakout: Implement Scan Line Algorithm

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

Step 3: Implement the function `drawTriangle` to check the triangle and split it up

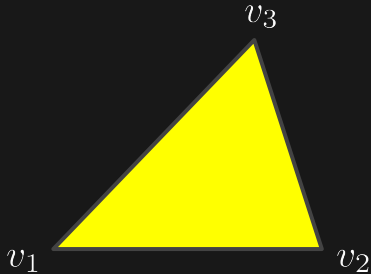
```
drawTriangle(v1: Vector2, v2: Vector2, v3: Vector2, color: number) {  
  // TODO: implement the scan line algorithm for filling triangles  
  
  // sort three vertices to guarantee v1.y > v2.y > v3.y  
  if (v2.y >= v1.y && v2.y >= v3.y) [v1, v2] = [v2, v1];  
  if (v3.y >= v1.y && v3.y >= v2.y) [v1, v3] = [v3, v1];  
  if (v3.y > v2.y) [v2, v3] = [v3, v2];  
  
  const v4 = new Vector2(  
    v1.x + ((v2.y - v1.y) / (v3.y - v1.y)) * (v3.x - v1.x),  
    v2.y  
  );  
  if (v4.x !== v1.x) this.drawTriangleTop(v2, v4, v1, color);    // if top or arbitrary triangle  
  if (v4.x !== v3.x) this.drawTriangleBottom(v3, v4, v2, color); // if bottom or arbitrary triangle  
}
```



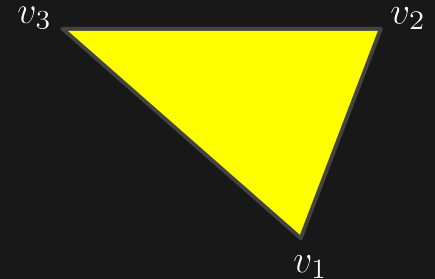
Breakout: Implement Scan Line Algorithm

Step 4: Implement the two mentioned functions for drawing the triangles:

```
drawTriangleTop(  
  v1: Vector2, v2: Vector2, v3: Vector2, color: number) {  
  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;  
  }  
}
```



```
drawTriangleBottom(  
  v1: Vector2, v2: Vector2, v3: Vector2, color: number) {  
  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;  
  }  
}
```



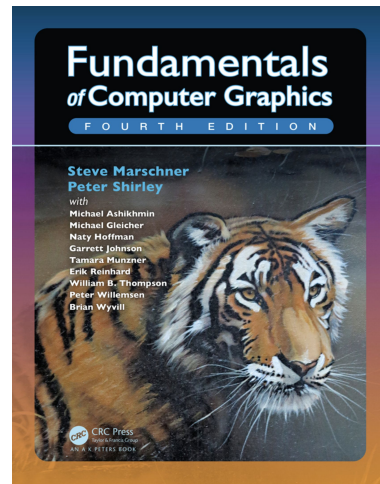
Summary

We covered:

- AABB and BVH as the acceleration data structure for different types of culling
- 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

Milestone: We have enough knowledge to implement the first rasterizer (without using any Graphics APIs)

- **Remember: You don't need a graphics API to do graphics!**
- **Check out this book for more fundamentals and optimizations**



Next

Rasterization II