TA 3

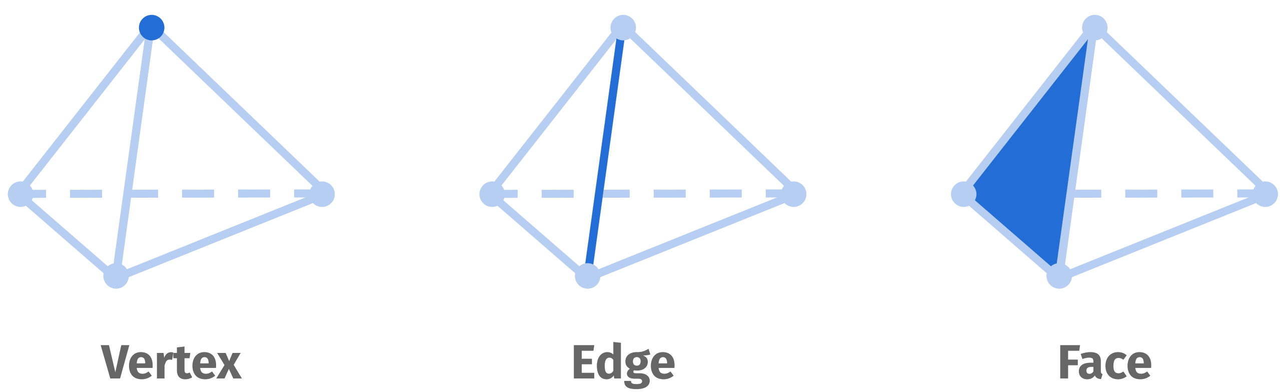
- 3D meshes
- The OBJ format
- Rasterization

Meshes

Computer Graphics 2020

Meshes

- A polygon ***Mesh*** is a collection of vertices, edges and faces (usually triangles) that defines the shape of a polyhedral object



Meshes

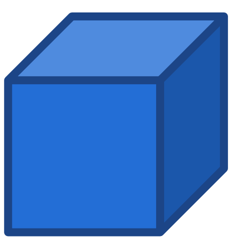
- Using meshes we can represent many objects in 3D to create complex scenes

Vertices

- At each ***Vertex*** of a mesh we can store data about its structure and properties, for example:
    - *Position* - coordinates in 3D space
    - *Color* - color of the mesh at this point
    - *Normal Vector*
    - *Texture/UV Coordinates*
- We will learn more about them later in the course

Faces

- A ***Face*** is a closed set of edges
- A ***Triangle Face*** has three edges, and a ***Quad Face*** has four edges
- A ***Polygon*** is a coplanar set of faces. Usually the terms are used interchangeably



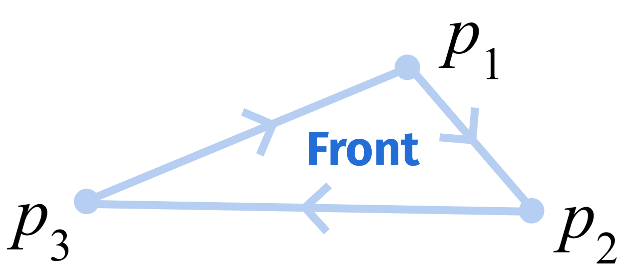
Cube Mesh

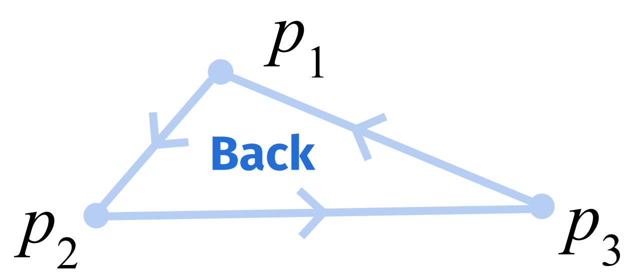
Triangle Face

Polygon

Faces

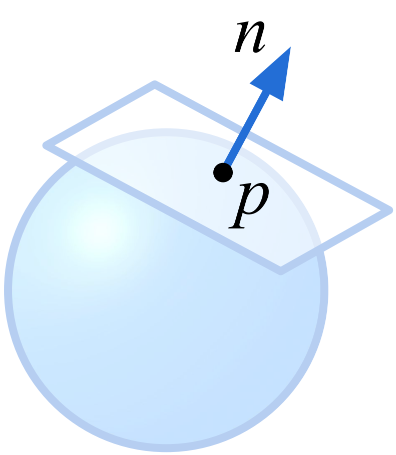
- A face is defined by a series of vertices
- Ordering matters! $F_1 = (p_1 \ p_2 \ p_3) \neq F_2 = (p_3 \ p_2 \ p_1)$
- In Unity (left-handed coordinate system), Vertices are assumed to be in a clockwise order. This determines which side of a face is the front





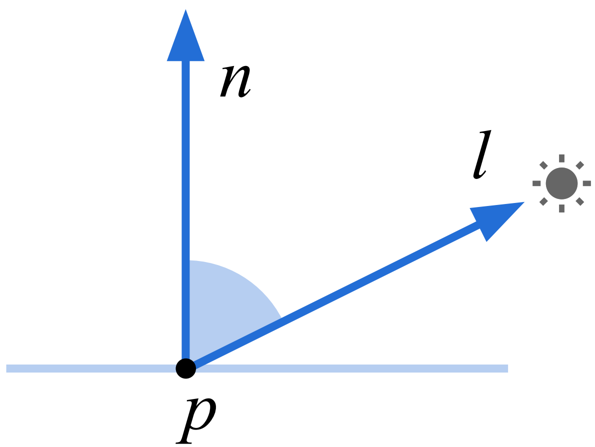
Surface Normals

- A ***Surface Normal*** n to a surface at point p is a vector perpendicular to the tangent plane of the surface at p
- n is usually normalized, $\|n\| = 1$



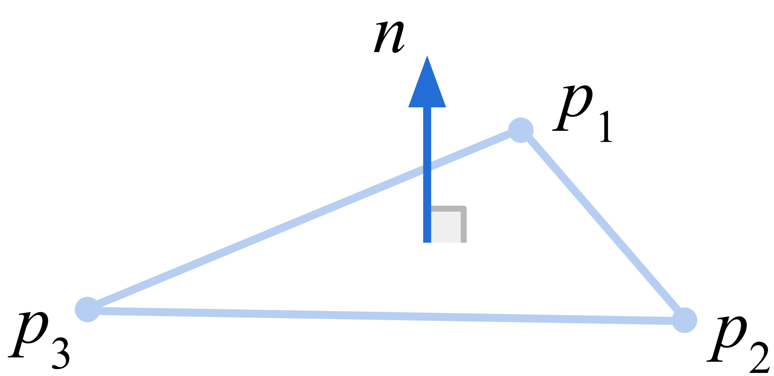
Surface Normals

- The normal can be used to determine a surface's orientation toward a light source at p
- Using this angle we can calculate the *shading* at p and draw its color accordingly



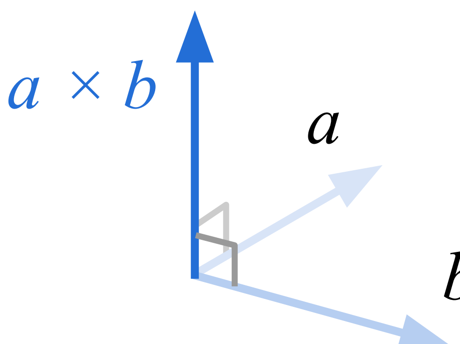
Surface Normals

- Given a polygon with vertices p_1 p_2 p_3, how do we calculate its surface normal n?



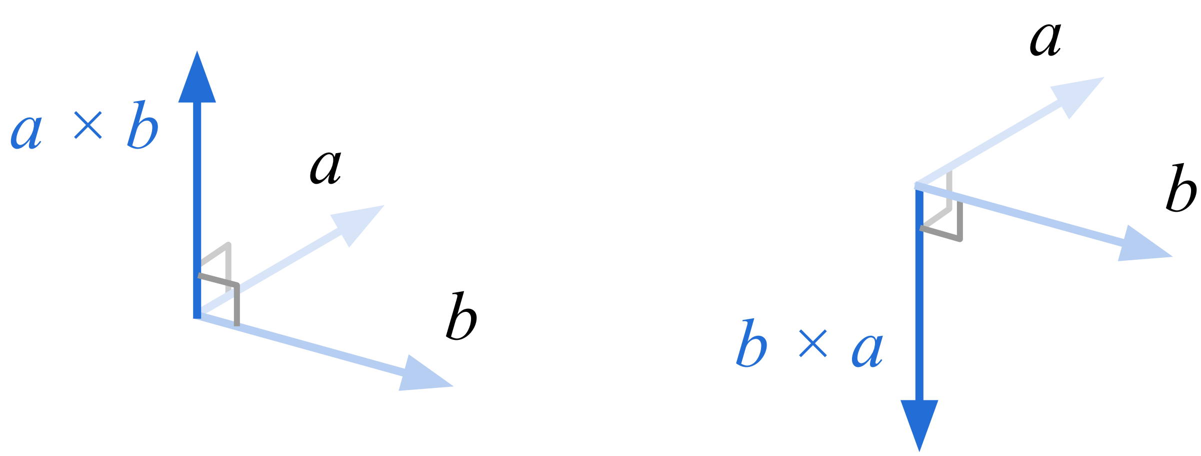
Cross Product

- Given two linearly independent vectors a and b, $a \times b$ is a vector that is perpendicular to both of them
- $a \times b$ is a normal to the plane spanned by a and b



Cross Product

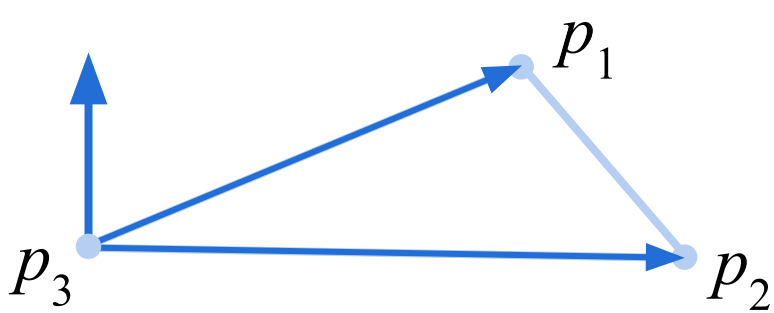
- $b \times a$ is also a normal to the plane, in the negative direction: $a \times b = -b \times a$
- The positive direction is determined by the handedness of the coordinate system



Surface Normals

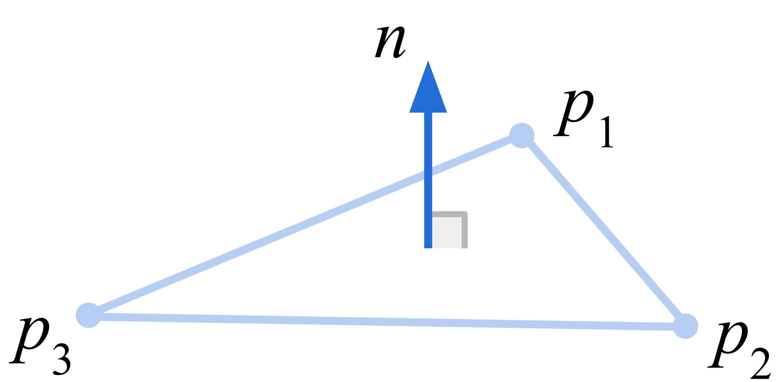
- So how do we calculate the surface normal using the cross product?

$$(p_1 - p_3) \times (p_2 - p_3)$$



Surface Normals

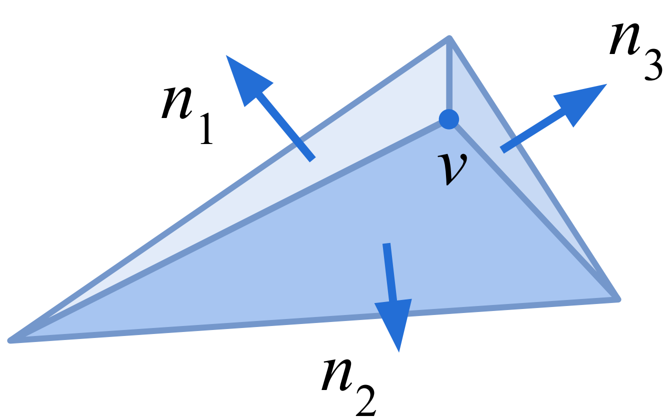
- We get: $n = (p_1 - p_3) \times (p_2 - p_3)$
- Remember to normalize! $n \leftarrow n \,/\, \|n\|$



Vertex Normals

- Usually normals are stored per-vertex rather than per-face

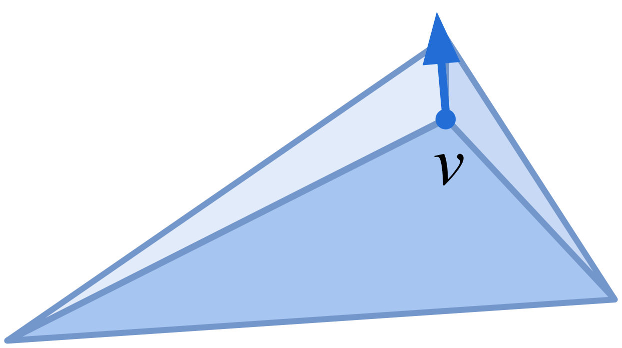
- Given a vertex v on the intersection of 3 faces with normals n_1, n_2, n_3 what should its normal be?



Vertex Normals

- v's normal will be the normalized average direction of the 3 surface normals:

$$\frac{(n_1 + n_2 + n_3)}{||(n_1 + n_2 + n_3)||}$$



The OBJ Format

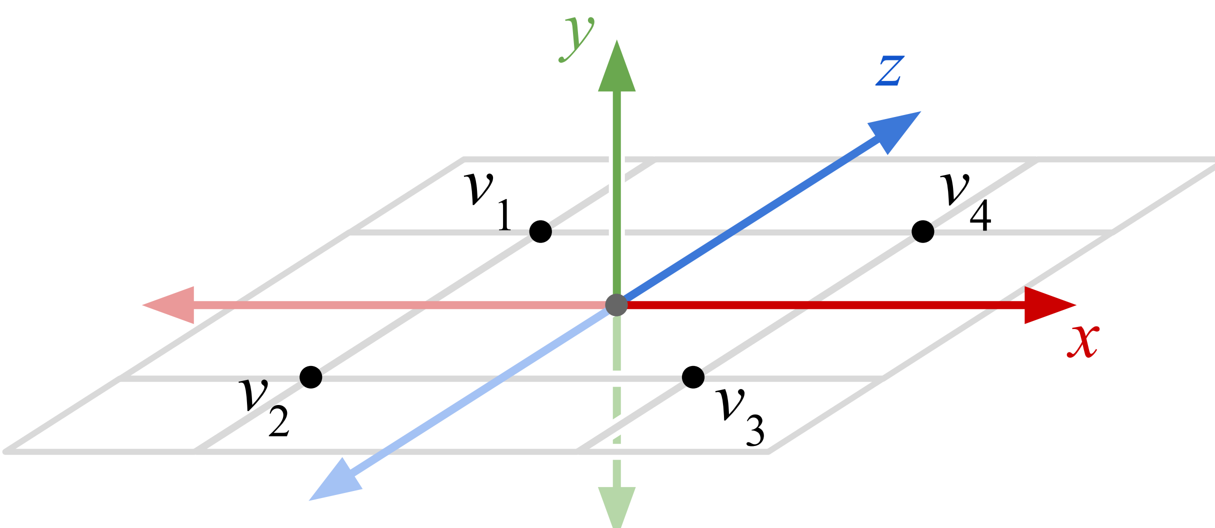
- Common data format that represents 3D meshes
- Text-based - lists of vertices, faces and other properties:

```
# List of vertices
v -1 0 1
v -1 0 -1
v 1 0 -1
v 1 0 1

# List of faces
f 1 3 2
f 1 4 3
```

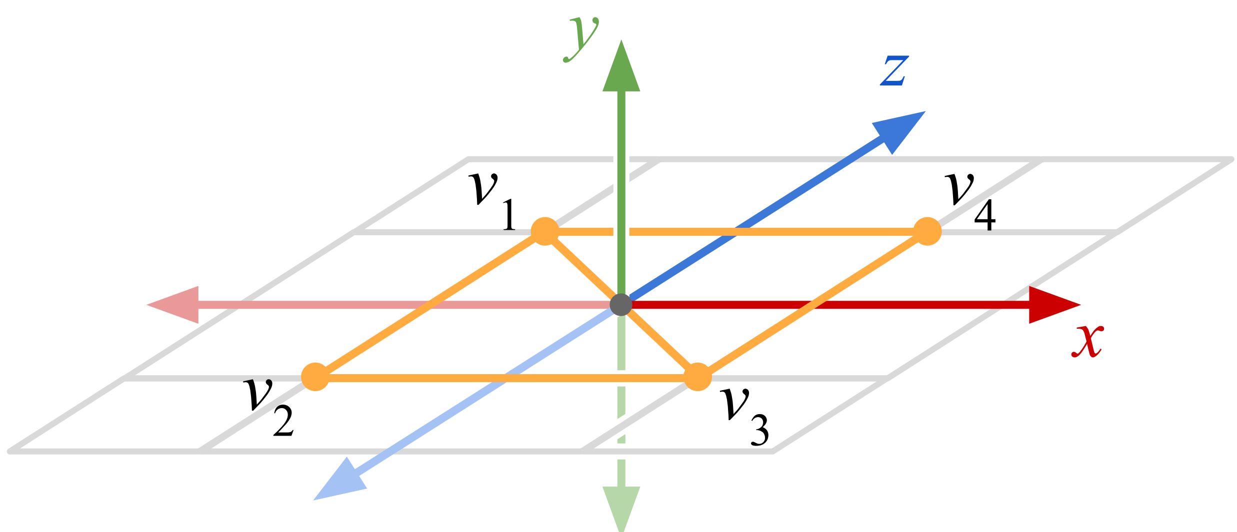
The OBJ Format

```
# List of vertex positions
v -1  0  1  # v1
v -1  0 -1  # v2
v  1  0 -1  # v3
v  1  0  1  # v4
```



The OBJ Format

```
# Each face is a list of indices
f 1 3 2
f 1 4 3
```



Unity Mesh Class

- A Unity `Mesh` contains vertex data and face data
- All vertex data is stored in separate arrays of the same size
- Complete `Mesh` documentation:
  docs.unity3d.com/ScriptReference/Mesh.html

Unity Mesh Class

```
public class Mesh
{
    Vector3[] vertices; // Vertices
    int vertexCount;    // Number of vertices
    int[] triangles;    // Faces (indices of vertices)
    Vector3[] normals;  // Surface normals per vertex
    Color[] colors;     // Colors per vertex
    Vector2[] uv;       // Texture coords per vertex

    // More properties and methods...
}
```

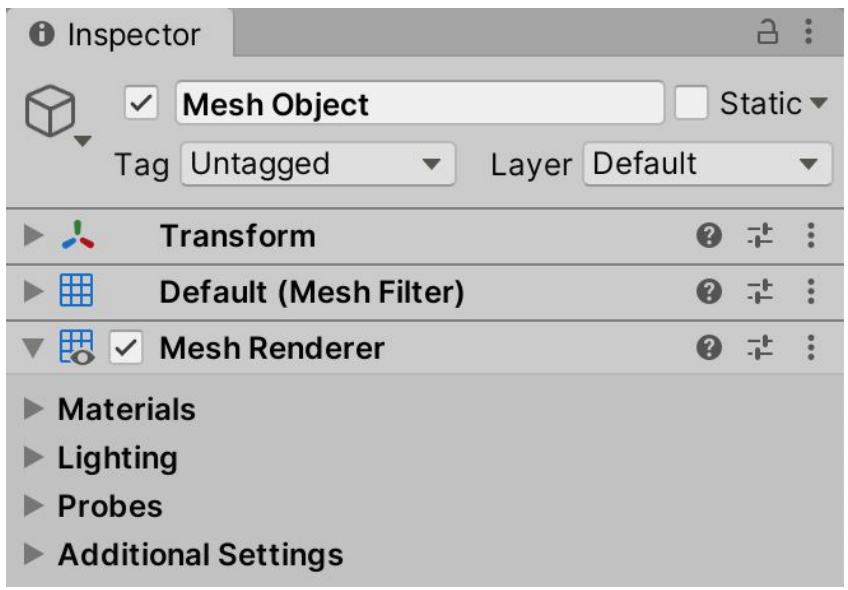
Mesh Filter Component

- In order to display a mesh in our scene, we must attach it to a GameObject

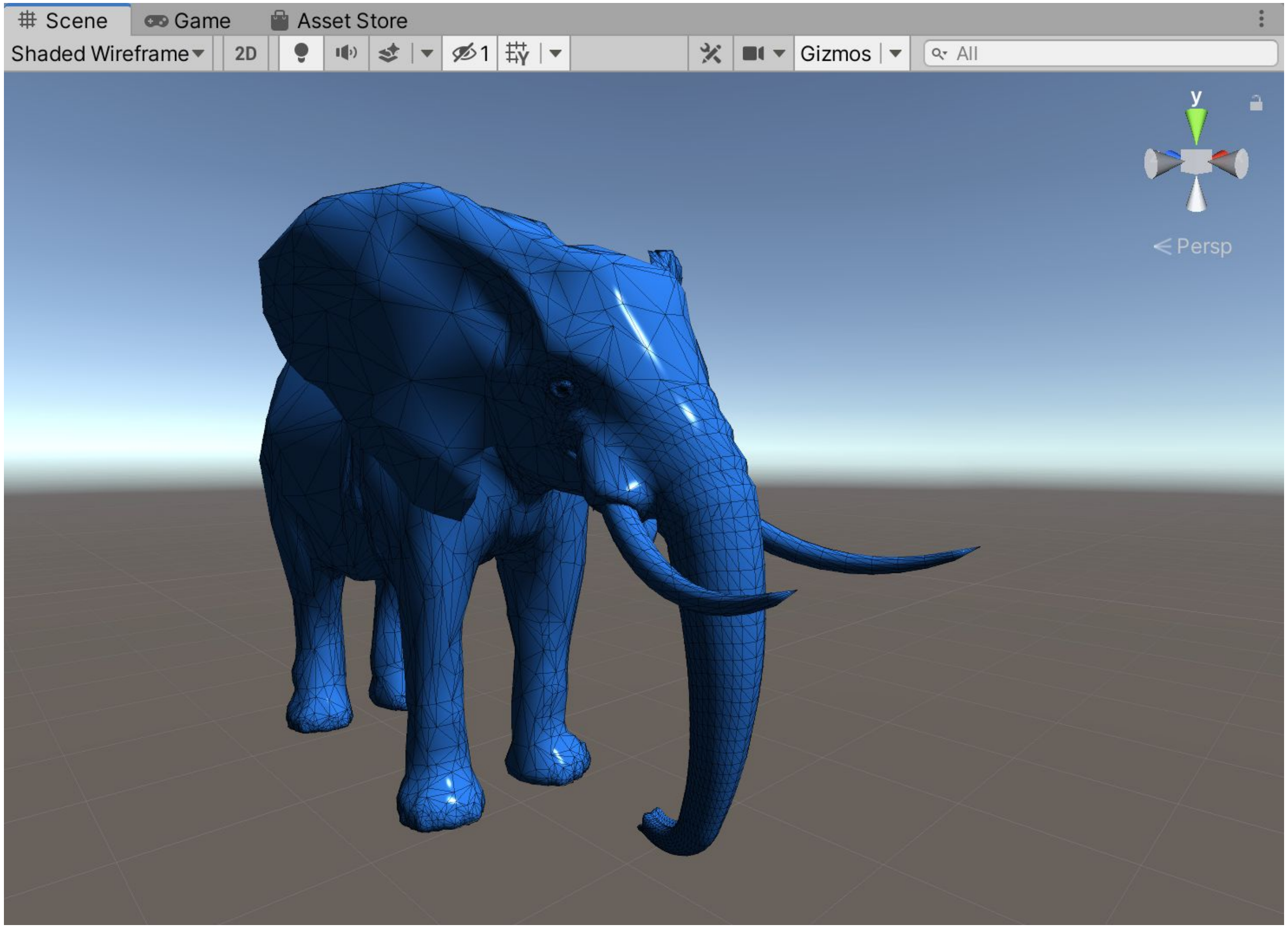
- The ***Mesh Filter*** component does just that - it has a Mesh field that can be assigned from the inspector or programmatically

Mesh Renderer Component

- The ***Mesh Renderer*** takes the geometry from the *Mesh Filter* and renders it at the position defined by the GameObject's Transform component



Unity Mesh



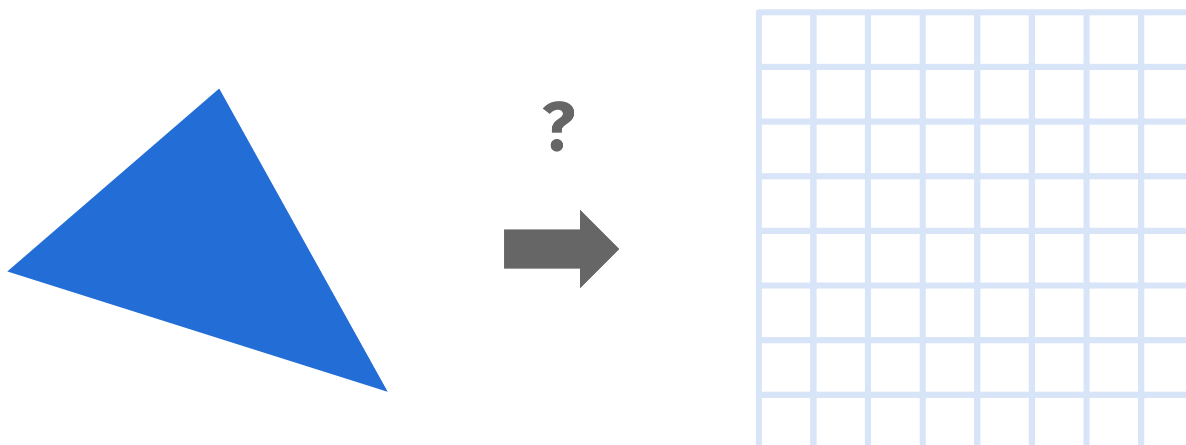
Scene
Game
Asset Store
Shaded Wireframe
2D
1
Gizmos
All
y
Persp

Rasterization

Computer Graphics 2020

Rasterization

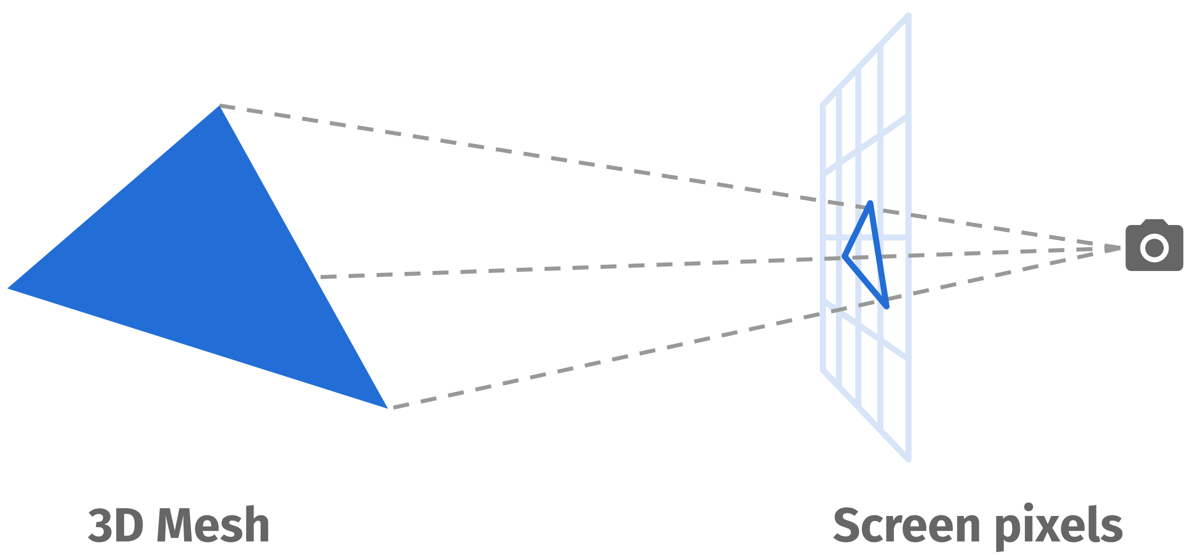
- We have a triangle representation of our 3D scene, but how do we draw it to our screen?
- Remember, a screen is a 2D grid of pixels



3D Mesh **Screen pixels**

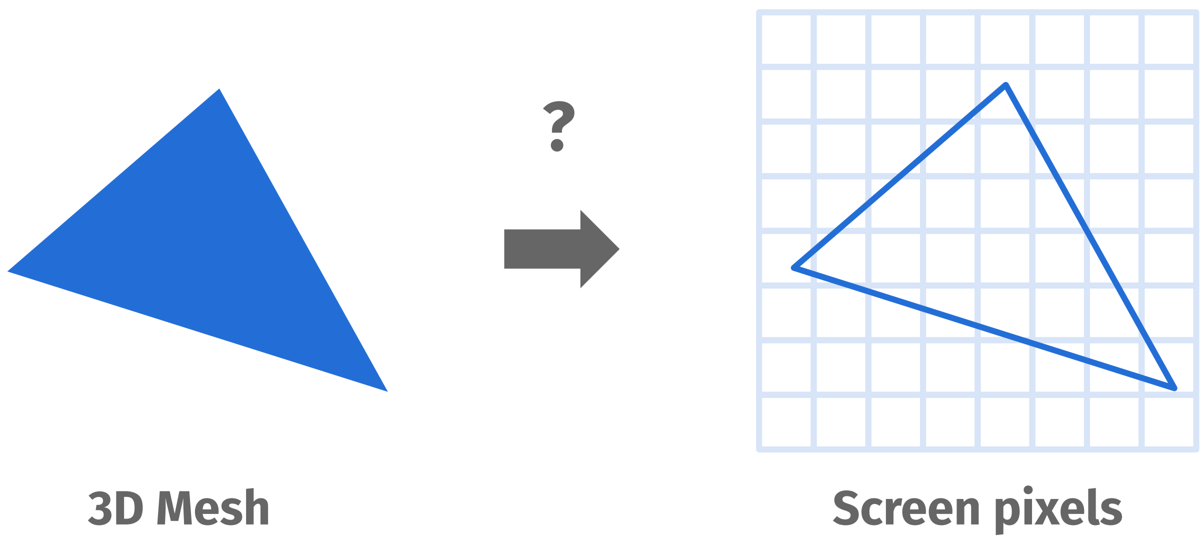
Projection

- First we project our 3D scene onto a 2D plane
- You will learn more about this process and the rendering pipeline in the lecture



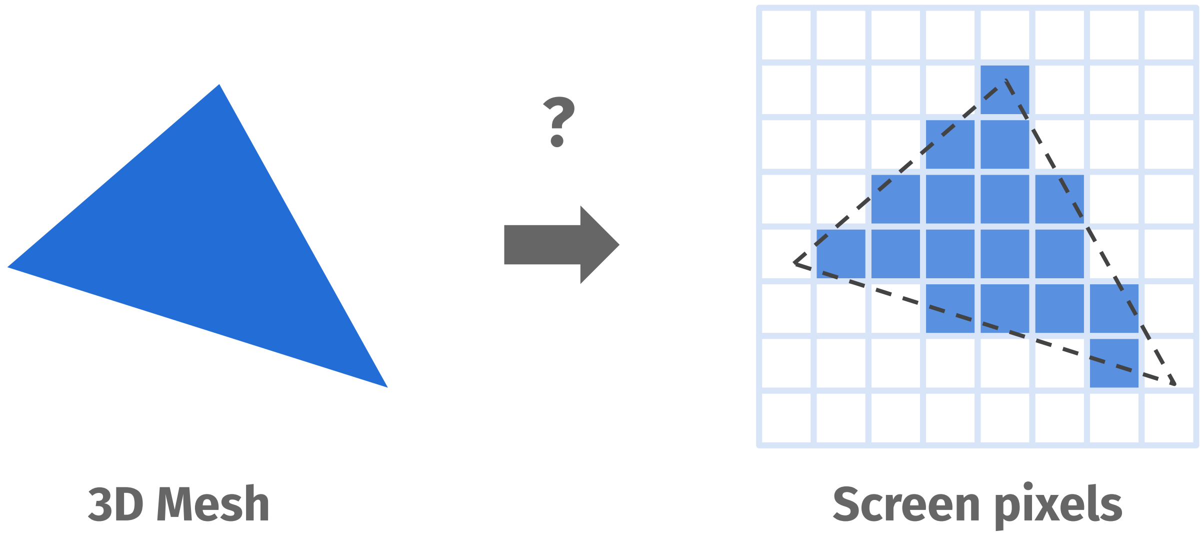
Rasterization

- ***Rasterization*** is the process of converting 2D primitives into a discrete pixel representation, known as a *Raster Image*



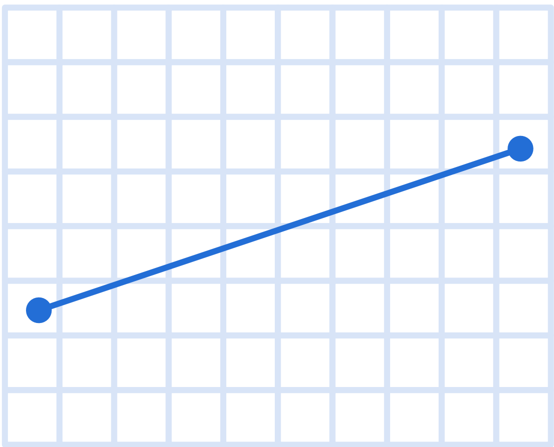
Rasterization

- ***Rasterization*** is the process of converting 2D primitives into a discrete pixel representation, known as a *Raster Image*



Line Rasterization

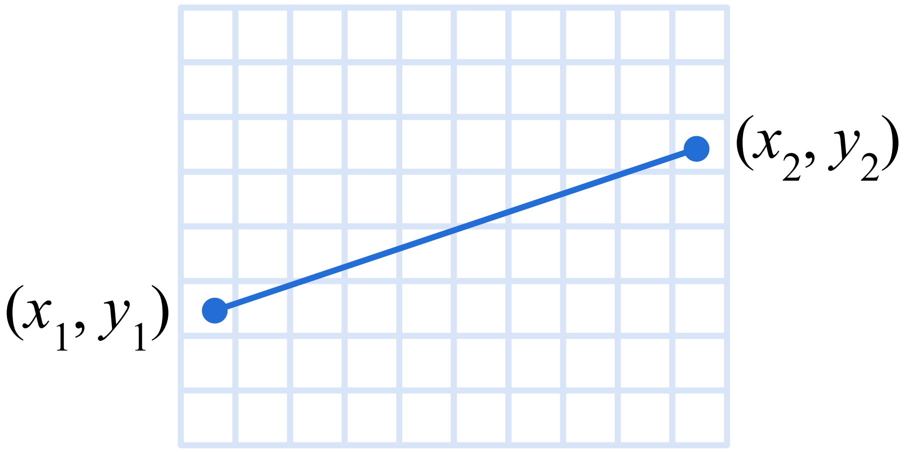
- The most basic primitive is a line
- Given a line from (x_1, y_1) to (x_2, y_2) how can we rasterize it?



Line Rasterization

- A naive line-drawing algorithm:

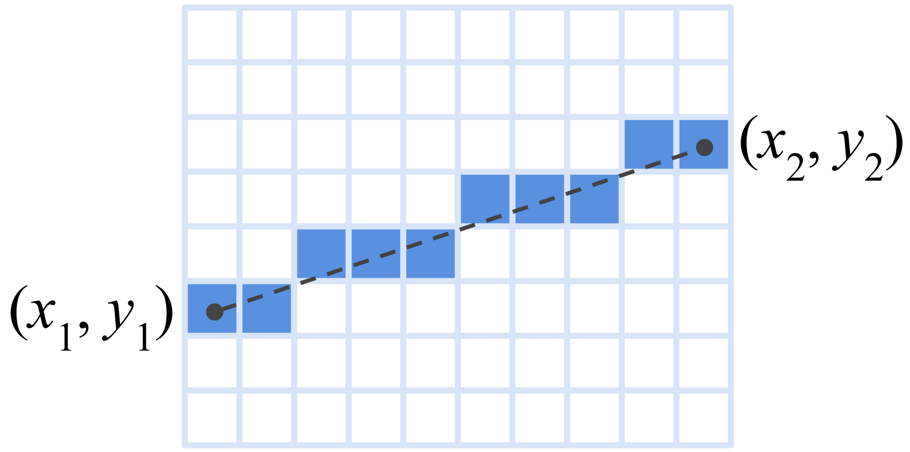
```
dx = x2 - x1
dy = y2 - y1
for x from x1 to x2 do:
    y = y1 + dy × (x - x1) / dx
    fill(x, y)
```



Line Rasterization

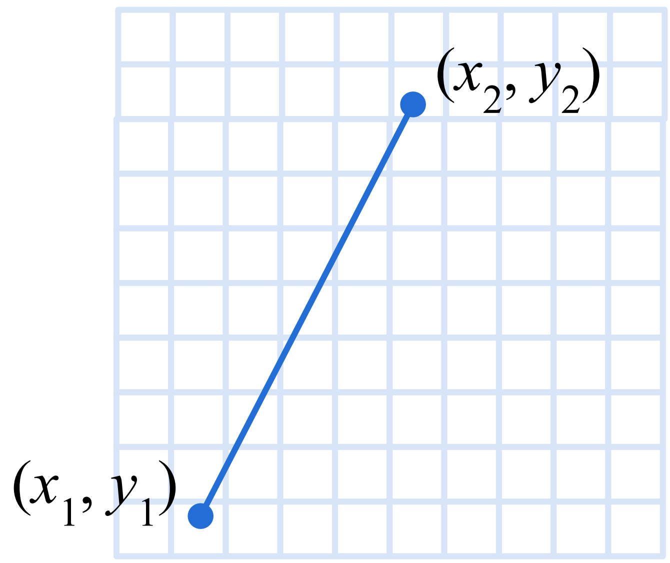
- A naïve line-drawing algorithm:

```
dx = x2 - x1
dy = y2 - y1
for x from x1 to x2 do:
    y = round(y1 + dy / dx × (x - x1))
    fill(x, y)
```



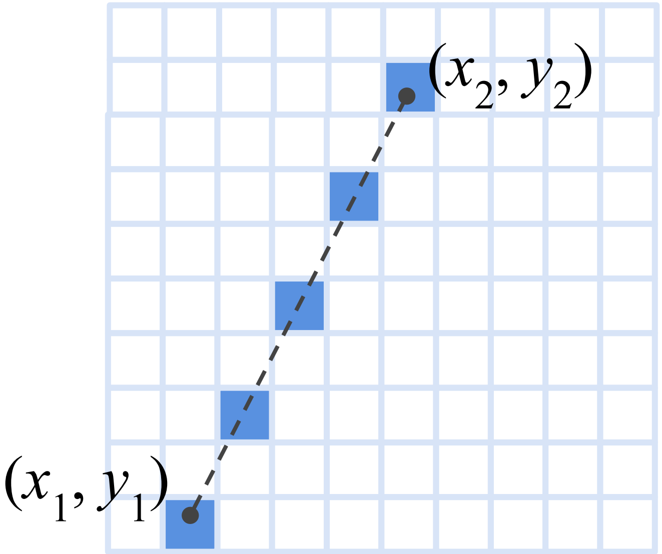
Line Rasterization

- What if the slope is greater than 1, i.e. `dy > dx`?



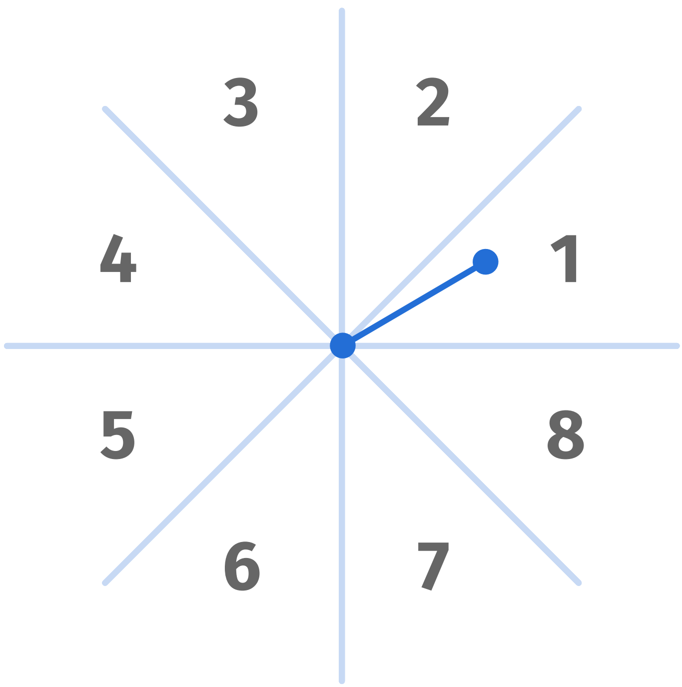
Line Rasterization

- The algorithm doesn't allow for more than one pixel per column - we get gaps!



Line Rasterization

- We need to adjust the algorithm for each octant:

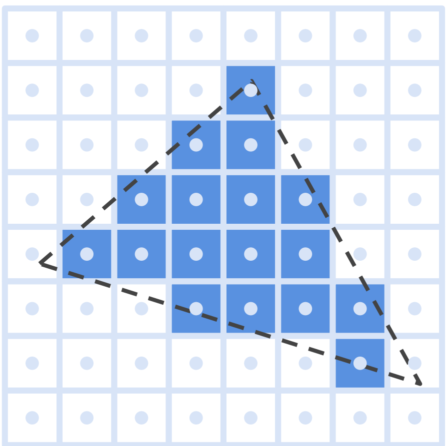


Line Rasterization

- This naïve line drawing algorithm we saw is inefficient
- It uses a large number of operations and floating-point calculations
- ***Bresenham's Line Algorithm*** is a better alternative - it uses only integer addition, subtraction and bit shifting
- Explanation of Bresenham's Line Algorithm

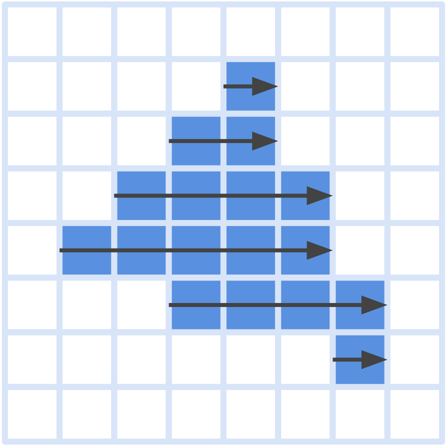
Triangle Rasterization

- To rasterize a triangle, we can iterate over the pixels, and check weather it is inside or outside the triangle (how?)
- We use center points of pixels for calculating



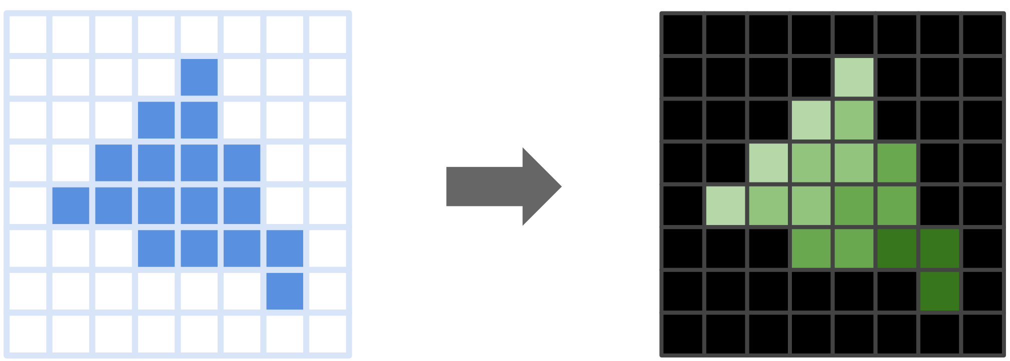
Triangle Rasterization

- A more efficient approach:
  - Bresenham line rasterization to find the edges
  - Fill inside in scan-line order



Rasterization

- Once we have rasterized our 3D shapes, we know which pixel contains which object
- To decide what color each pixel should be, we need to implement *lighting & shading*



Rasterization

- Next week we'll start talking about lighting and shading
- In tomorrow's lecture you will learn about the rendering pipeline in more detail

Geometry (vertices) → Vertex processing → Clipping & assembling → Rasterization → Fragment processing → Raster (pixels)

The Rendering Pipeline