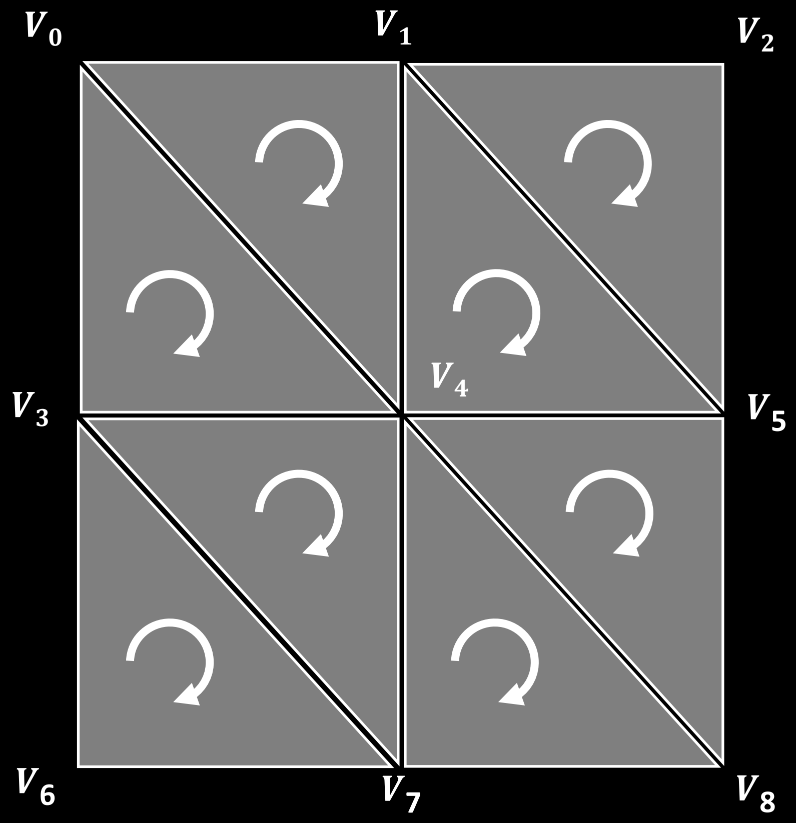
GRAPHICS PROGRAMMING I

SOFTWARE RASTERIZATION PART II

Rasterization: Quad [TriangleList]

- Last week you rendered two triangles with a different depths. Let's make this a quad!
- You could do this by using 4 vertices and 6 indices, thus defining two triangles. Instead, I want you to define it using 9 vertices and 24 indices (3 indices * 8 triangles).

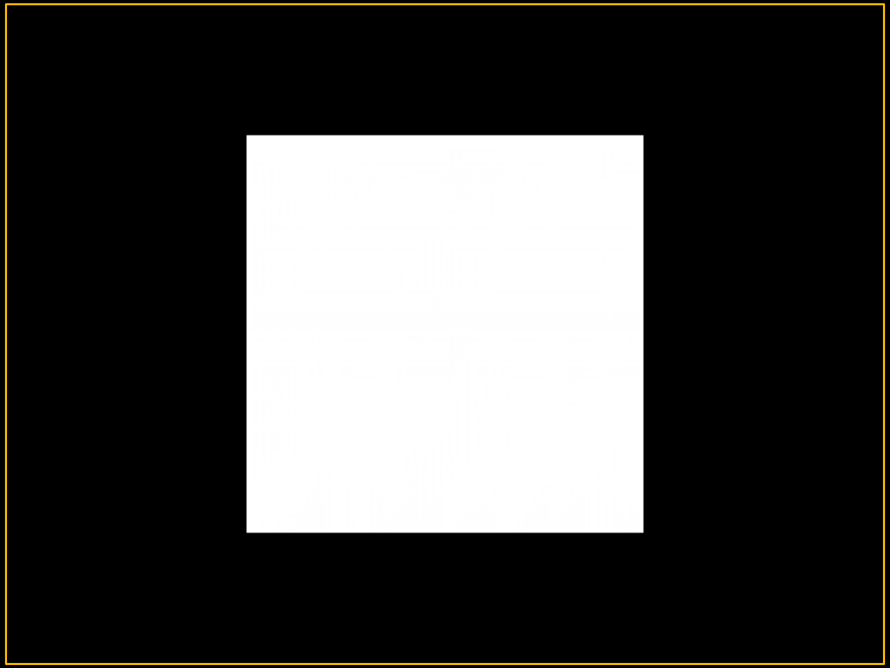


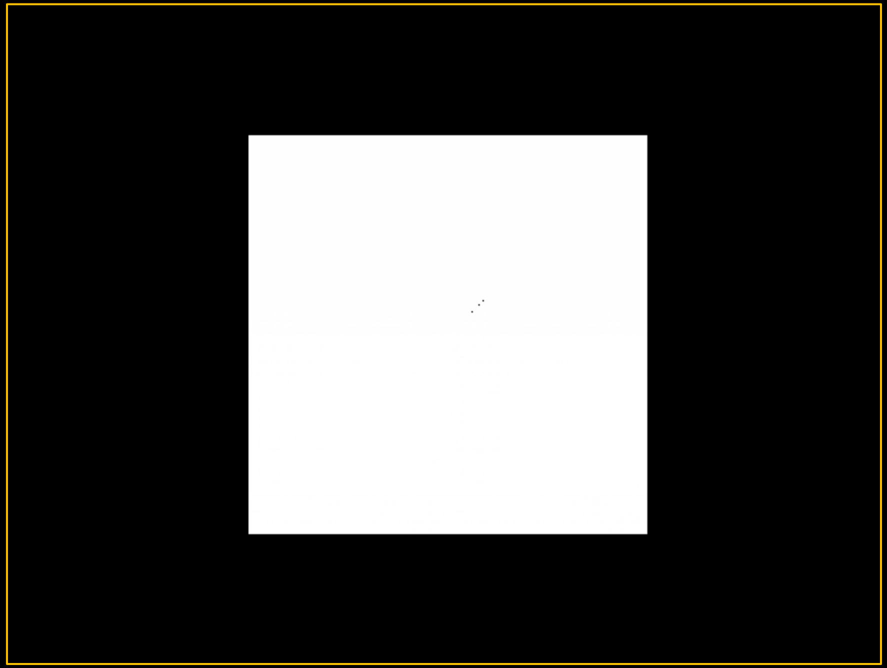
V_0 = (-3, 3, -2)
V_1 = (0, 3, -2)
V_2 = (3, 3, -2)
V_3 = (-3, 0, -2)
V_4 = (0, 0, -2)
V_5 = (3, 0, -2)
V_6 = (-3, -3, -2)
V_7 = (0, -3, -2)
V_8 = (3, -3, -2)

T_0 = (3, 0, 4)
T_1 = (0, 1, 4)
T_2 = (4, 1, 5)
T_3 = (1, 2, 5)
T_4 = (6, 3, 7)
T_5 = (3, 4, 7)
T_6 = (7, 4, 8)
T_7 = (4, 5, 8)

Rasterization: Quad

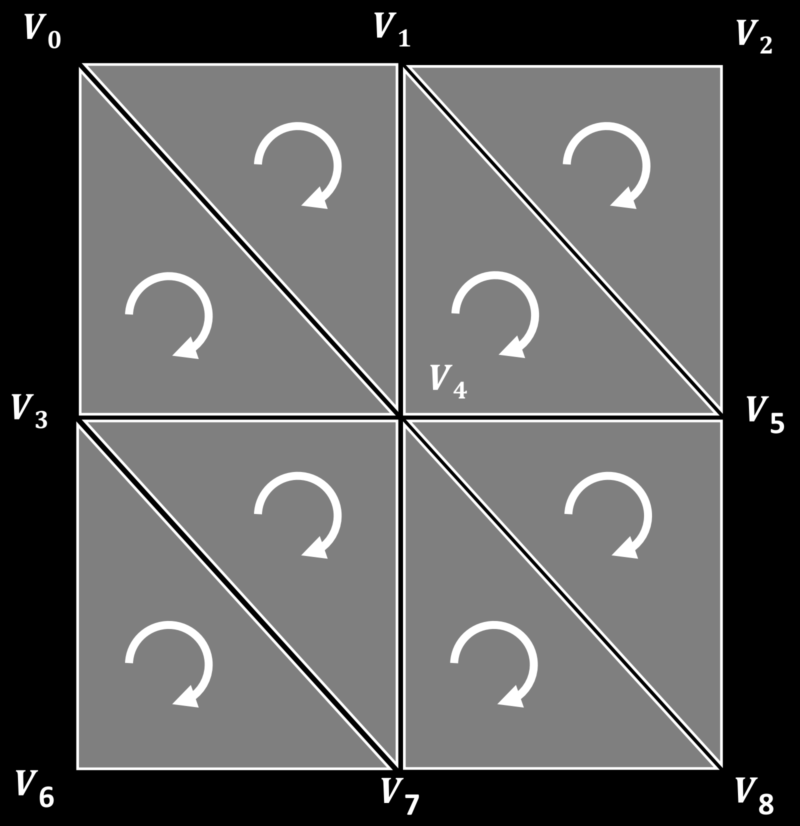
- Using the code from last week, with a white color for all vertices, you should get the following result.
- Start moving the camera, and you **might** encounter the following issue! Check types & Bounds ☺



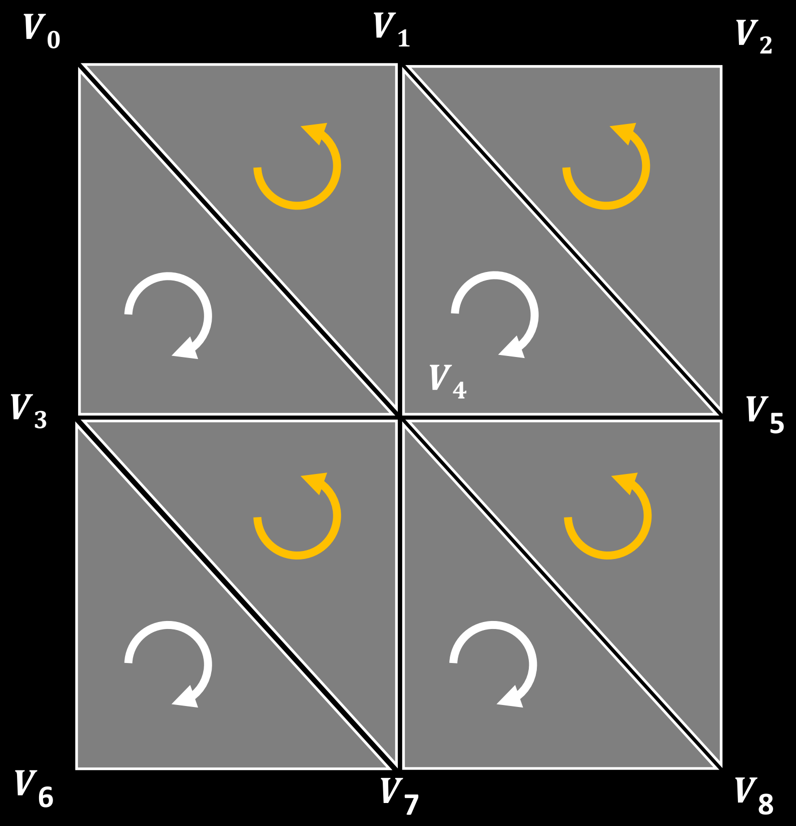


- What if I tell you we can optimize our data even more…

Rasterization: Quad [TriangleStrip]

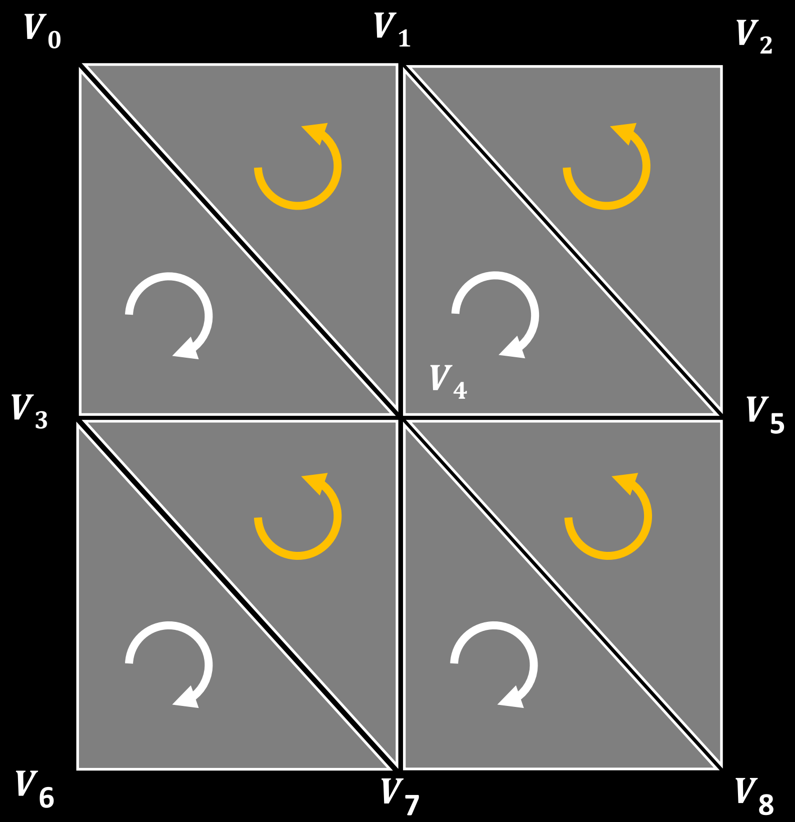


$T_0 = (3, 0, 4)$
$T_1 = (0, 1, 4)$
$T_2 = (4, 1, 5)$
$T_3 = (1, 2, 5)$
$T_4 = (6, 3, 7)$
$T_5 = (3, 4, 7)$
$T_6 = (7, 4, 8)$
$T_7 = (4, 5, 8)$

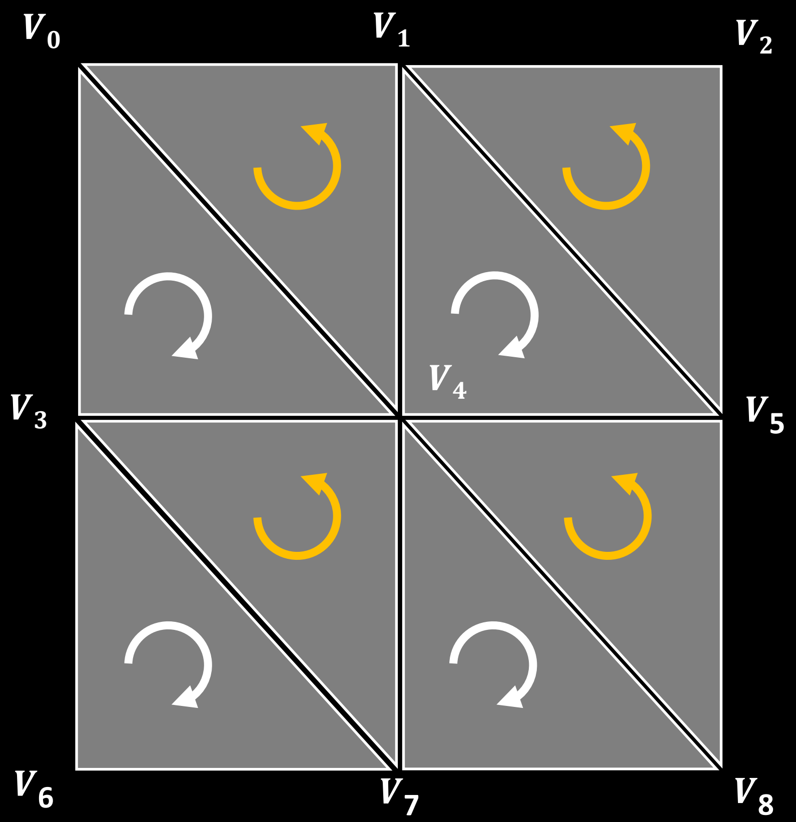


$T_0 = (3, 0, 4)$
$T_1 = (0, 4, 1)$
$T_2 = (4, 1, 5)$
$T_3 = (1, 5, 2)$
$T_4 = (6, 3, 7)$
$T_5 = (3, 7, 4)$
$T_6 = (7, 4, 8)$
$T_7 = (4, 8, 5)$

Rasterization: Quad [TriangleStrip]



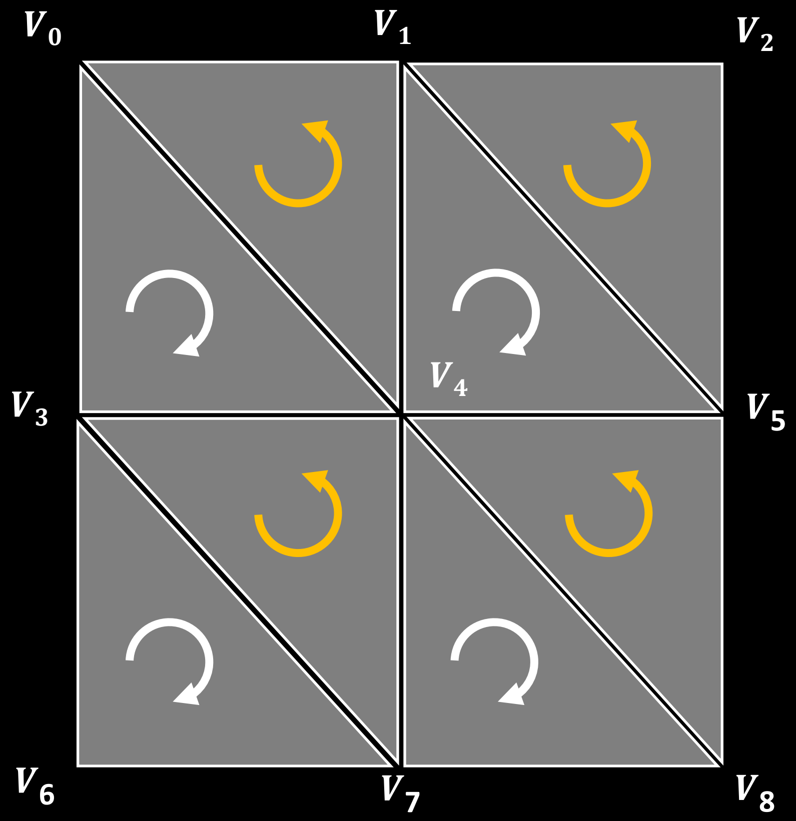
$T_0 = (3, 0, 4)$
$T_1 = (0, 4, 1)$
$T_2 = (4, 1, 5)$
$T_3 = (1, 5, 2)$
$T_4 = (6, 3, 7)$
$T_5 = (3, 7, 4)$
$T_6 = (7, 4, 8)$
$T_7 = (4, 8, 5)$



$T_0 = (3, 0, 4)$
$T_1 = (0, 4, 1)$
$T_2 = (4, 1, 5)$
$T_3 = (1, 5, 2)$
$T_4 = (6, 3, 7)$
$T_5 = (3, 7, 4)$
$T_6 = (7, 4, 8)$
$T_7 = (4, 8, 5)$

Rasterization: Quad [TriangleStrip]

- We can get rid of these **duplicate** pieces of information!



$T_0 = (3, 0, 4)$
$T_1 = (0, 4, 1)$
$T_2 = (4, 1, 5)$
$T_3 = (1, 5, 2)$
$T_4 = (6, 3, 7)$
$T_5 = (3, 7, 4)$
$T_6 = (7, 4, 8)$
$T_7 = (4, 8, 5)$

{ 3, 0, 4, 0, 4, 1, 4, 1, 5, 1, 5, 2, 6, 3, 7, 3, 7, 4, 7, 4, 8, 4, 8, 5 }

↓

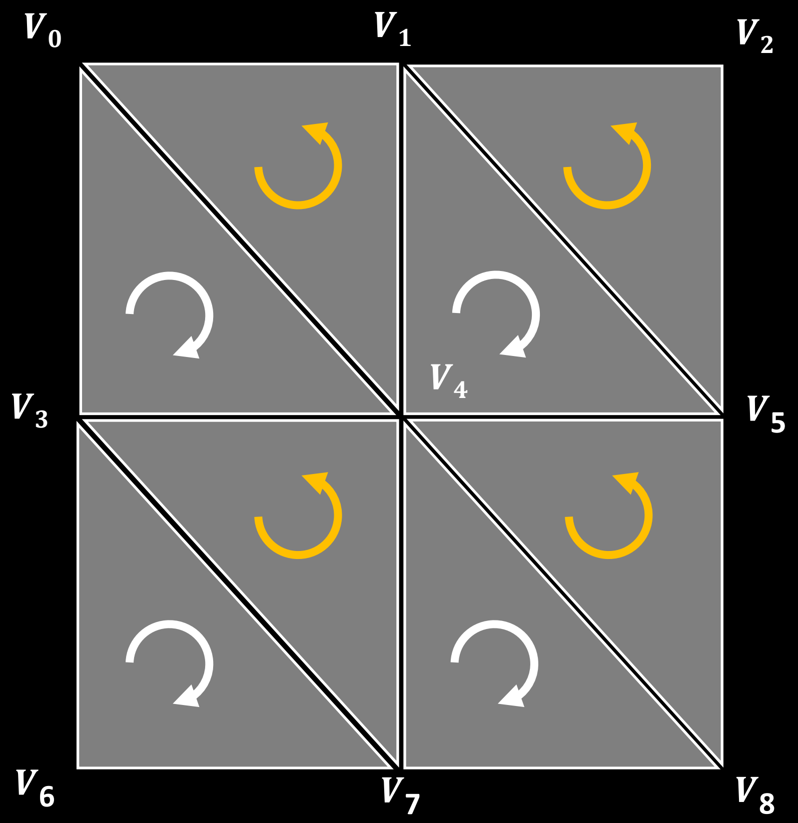
{3, 0, 4, 0, 4, 1, 4, 1, 5, 1, 5, 2, 6, 3, 7, 3, 7, 4, 7, 4, 8, 4, 8, 5 }

↓

{3, 0, 4, 1, 4, 1, 5, 2, 6, 3, 7, 4, 7, 4, 8, 5 }

Rasterization: Quad [TriangleStrip]

- We can get rid of these **duplicate** pieces of information!
- We just saved 12 * 4 bytes = 48 bytes! (24 > 12 Indices)



T_0 = (3, 0, 4)
T_1 = (0, 4, 1)
T_2 = (4, 1, 5)
T_3 = (1, 5, 2)
T_4 = (6, 3, 7)
T_5 = (3, 7, 4)
T_6 = (7, 4, 8)
T_7 = (4, 8, 5)

{ 3, 0, 4, 0, 4, 1, 4, 1, 5, 1, 5, 2, 6, 3, 7, 3, 7, 4, 7, 4, 8, 4, 8, 5 }

24 Indices

↓

{3, ~~0, 4~~, 0, 4, 1, 4, ~~1, 5~~, 1, 5, 2, 6, ~~3, 7~~, 3, 7, 4, 7, ~~4, 8~~, 4, 8, 5 }

↓

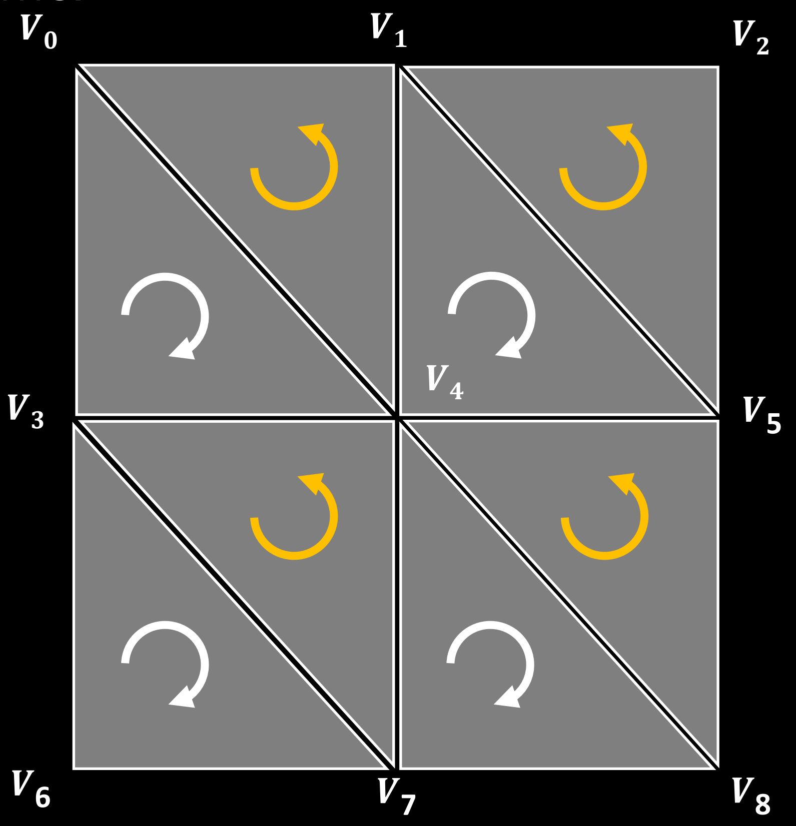
{3, 0, 4, 1, 4, 1, 5, 2, 6, 3, 7, 4, 7, 4, 8, 5 }

↓

{3, 0, 4, 1, 5, 2, 6, 3, 7, 4, 8, 5 } 12 Indices

Rasterization: Quad [TriangleStrip]

- How do we read this data now?
- We read them shifting one index at the time, instead of 3 at a time!



$T_0 = (3, 0, 4)$
$T_1 = (0, 4, 1)$
$T_2 = (4, 1, 5)$
$T_3 = (1, 5, 2)$
$T_4 = (6, 3, 7)$
$T_5 = (3, 7, 4)$
$T_6 = (7, 4, 8)$
$T_7 = (4, 8, 5)$

$\{\underbrace{3, 0, 4}_{T_0}, 1, 5, 2, 6, 3, 7, 4, 8, 5\}$

ClockWise

$\{3, \underbrace{0, 4, 1}_{T_1}, 5, 2, 6, 3, 7, 4, 8, 5\}$

CounterClockWise

$\{3, 0, \underbrace{4, 1, 5}_{T_2}, 2, 6, 3, 7, 4, 8, 5\}$

ClockWise

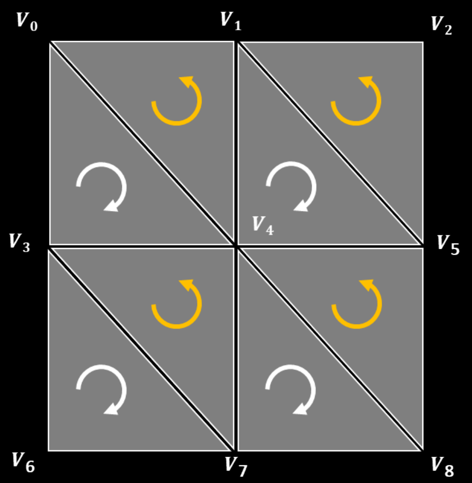
$\{3, 0, 4, \underbrace{1, 5, 2}_{T_3}, 6, 3, 7, 4, 8, 5\}$

CounterClockWise

$\{3, 0, 4, 1, \underbrace{5, 2, 6}_{T_{??}}, 3, 7, 4, 8, 5\}$

Rasterization: Quad [TriangleStrip]

- When you want to start a new **strip** (going to the next "row" or "band" of our primitive), we need to add some additional data to make sure our loop is correct.
- We can fix this by adding a **degenerate triangle**. A degenerate triangle is triangle with **no surface area**. Because it has no area it will **not** be rendered! The only reason we use it, is to fix our index buffer loop!
- The idea is simple, when starting a new strip, add the last vertex of the current "row" and the first vertex of the second "row".



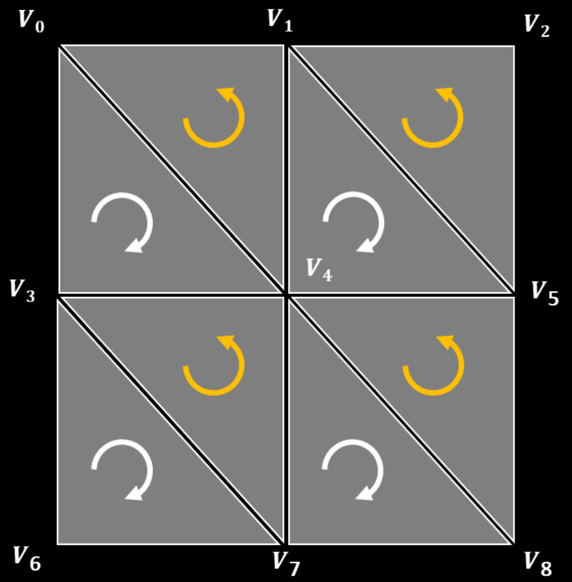
V_2 and V_6 in this particular case.

{3, 0, 4, 1, 5, 2, 6, 3, 7, 4, 8, 5 }

Last index of top strip, first index of bottom strip

Rasterization: Quad [TriangleStrip]

- So, our index buffer becomes: {3, 0, 4, 1, 5, 2, 2, 6, 6, 3, 7, 4, 8, 5 }
- Let's see if we fixed it?



T_0 = (3, 0, 4)
T_1 = (0, 4, 1)
T_2 = (4, 1, 5)
T_3 = (1, 5, 2)
T_4 = (6, 3, 7)
T_5 = (3, 7, 4)
T_6 = (7, 4, 8)
T_7 = (4, 8, 5)

$\{3, 0, 4, 1, \underbrace{5, 2, 2}_{T_{NoSurface}}, 6, 6, 3, 7, 4, 8, 5\}$

$\{3, 0, 4, 1, 5, \underbrace{2, 2, 6}_{T_{NoSurface}}, 6, 3, 7, 4, 8, 5\}$

$\{3, 0, 4, 1, 5, 2, \underbrace{2, 6, 6}_{T_{NoSurface}}, 3, 7, 4, 8, 5\}$

$\{3, 0, 4, 1, 5, 2, 2, \underbrace{6, 6, 3}_{T_{NoSurface}}, 7, 4, 8, 5\}$

$\{3, 0, 4, 1, 5, 2, 2, 6, \underbrace{6, 3, 7}_{T_4}, 4, 8, 5\}$

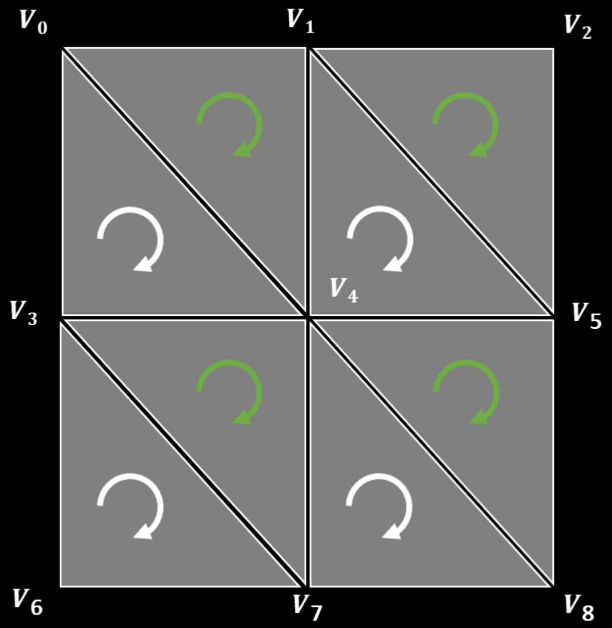
$\{3, 0, 4, 1, 5, 2, 2, 6, 6, \underbrace{3, 7, 4}_{T_5}, 8, 5\}$

$\{3, 0, 4, 1, 5, 2, 2, 6, 6, 3, \underbrace{7, 4, 8}_{T_6}, 5\}$

$\{3, 0, 4, 1, 5, 2, 2, 6, 6, 3, 7, \underbrace{4, 8, 5}_{T_7}\}$

Rasterization: Quad [TriangleStrip]

- Yes, we've fixed it! We had to add 2 indices though. This still gave us an optimization of 40 bytes!
- There is one last problem. Didn't we say we need to render our triangles in clockwise order?
- Yes, you can easily fix this by swapping the last two indices if the current triangle is odd!



$\{\underbrace{3, 0, 4}_{T_0}, 1, 5, 2, 2, 6, 6, 3, 7, 4, 8, 5\}$ $\downarrow$ $T_0 = (3, 0, 4)$

$\{3, \underbrace{0, 4, 1}_{T_1}, 5, 2, 2, 6, 6, 3, 7, 4, 8, 5\}$ $\downarrow$ $T_1 = (0, 4, 1)$

$T_1 = (0, 1, 4)$

Rasterization: Quad

- These two methods are usually supported (amongst others) by hardware accelerated rasterizers. The method used is defined by the Primitive Topology.
- The first method is called a TriangleList, while the latter is called a TriangleStrip.
- What to do:
  - An enum for the Primitive Topology is already defined – a mesh can use one of these.

```
enum class PrimitiveTopology
{
    TriangleList,
    TriangleStrip
};
```

(DataTypes.h, double check, it is possible your version contains a typo… ‘TriangeList <> TriangleList’)

  - Change your index loop accordingly(not pixel loop!). Considering if it’s an odd or even triangle if using the triangle strip technique. Hint: odd or even? → modulo or bit masking
  - Make two different index buffers to test both techniques. The vertex buffer doesn’t change! Use the buffers defined in the next slide.
  - You’ll also have to change your VertexTransformationFunction
    - Accepting a vector or Meshes, instead of a vector of Vertices

Rasterization: Quad

```cpp
//Define Mesh
std::vector<Mesh> meshes_world
{
    Mesh{
        .vertices: {
            Vertex{.position: {_x: -3, _y: 3, _z: -2}},
            Vertex{.position: {_x: 0, _y: 3, _z: -2}},
            Vertex{.position: {_x: 3, _y: 3, _z: -2}},
            Vertex{.position: {_x: -3, _y: 0, _z: -2}},
            Vertex{.position: {_x: 0, _y: 0, _z: -2}},
            Vertex{.position: {_x: 3, _y: 0, _z: -2}},
            Vertex{.position: {_x: -3, _y: -3, _z: -2}},
            Vertex{.position: {_x: 0, _y: -3, _z: -2}},
            Vertex{.position: {_x: 3, _y: -3, _z: -2}}
        },
        .indices: {
            3,0,4,1,5,2,
            2,6,
            6,3,7,4,8,5

        },
        PrimitiveTopology::TriangleStrip
    }
};
```

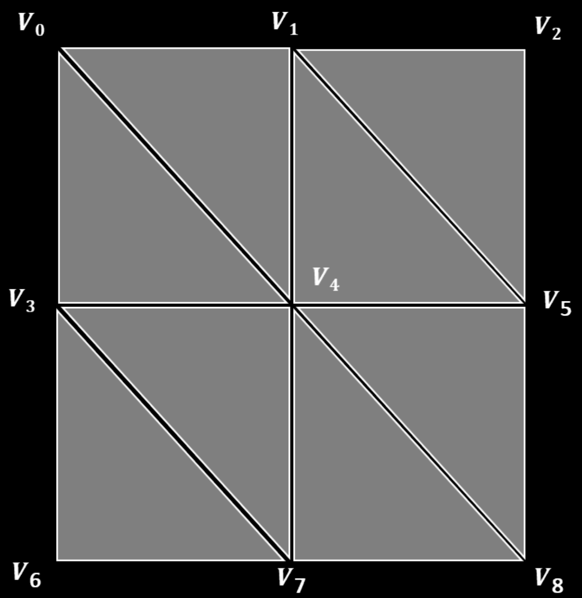
```cpp
//Define Mesh
std::vector<Mesh> meshes_world
{
    Mesh{
        .vertices: {
            Vertex{.position: {_x: -3, _y: 3, _z: -2}},
            Vertex{.position: {_x: 0, _y: 3, _z: -2}},
            Vertex{.position: {_x: 3, _y: 3, _z: -2}},
            Vertex{.position: {_x: -3, _y: 0, _z: -2}},
            Vertex{.position: {_x: 0, _y: 0, _z: -2}},
            Vertex{.position: {_x: 3, _y: 0, _z: -2}},
            Vertex{.position: {_x: -3, _y: -3, _z: -2}},
            Vertex{.position: {_x: 0, _y: -3, _z: -2}},
            Vertex{.position: {_x: 3, _y: -3, _z: -2}}
        },
        .indices: {
            3, 0, 1,    1, 4, 3,    4, 1, 2,
            2, 5, 4,    6, 3, 4,    4, 7, 6,
            7, 4, 5,    5, 8, 7

        },
        PrimitiveTopology::TriangleList
    }
};
```

Rasterization: Textures & Vertex Attributes

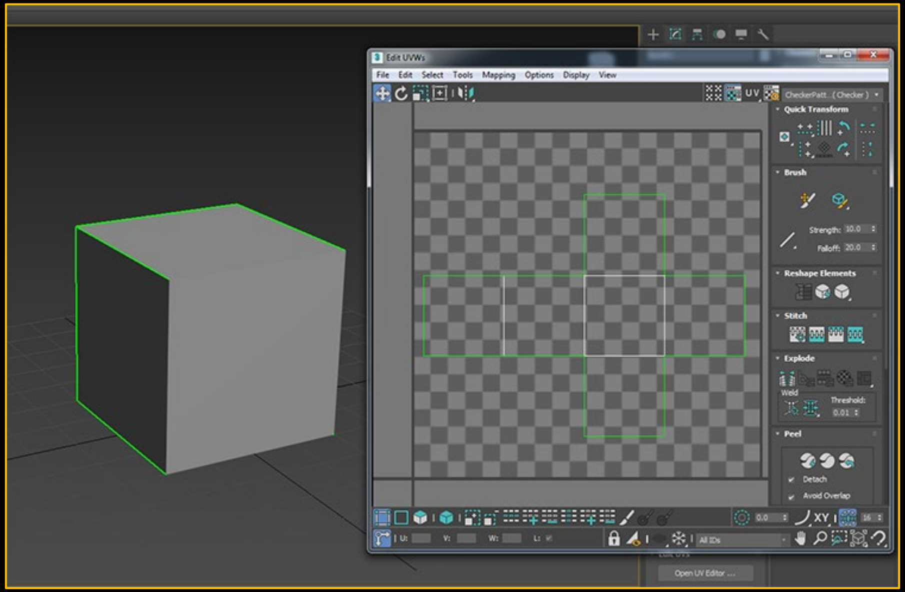
- Once you have both primitive topology techniques working, change your vertex struct by adding UV coordinates.

```
struct Vertex
{
    Vector3 position{};
    ColorRGB color{colors::White};
    Vector2 uv{};
    //Vector3 normal{};
    //Vector3 tangent{};
    //Vector3 viewDirection{};
};
```

- As you all know, the UV space is (typically) defined in [0,1] range. You usually get this data from your mesh file. For now, let's add the coordinates manually to our quad.



V_0 = (-3, 3, -2) – (0, 0)
V_1 = (0, 3, -2) – (.5, 0)
V_2 = (3, 3, -2) – (1, 0)
V_3 = (-3, 0, -2) – (0, .5)
V_4 = (0, 0, -2) – (.5, .5)
V_5 = (3, 0, -2) – (1, .5)
V_6 = (-3, -3, -2) – (0, 1)
V_7 = (0, -3, -2) – (.5, 1)
V_8 = (3, -3, -2) – (1, 1)



Rasterization: Textures & Vertex Attributes

- Now our quad is ready to be rendered using a texture.
- There are two things we need:
  - Have a class that reads in a texture file.
  - A way to sample the texture using our UV coordinates.

- Let's start by reading in a texture file:
  - Compared to the ray tracing framework, we added the necessary libraries to make your framework able to read in .png files.
    - We've added SDL2_image.lib with the other necessary libraries. Only .png is supported because we take over a limited amount of functionality. If you want to be able to read in other formats you can add the necessary libraries.

  - Complete the Texture Class functions.
    - Texture::LoadFromFile (static) > Loads the image, Creates & Returns a Texture Object
    - Texture::Sample > Samples a specific pixel from the SurfacePixels

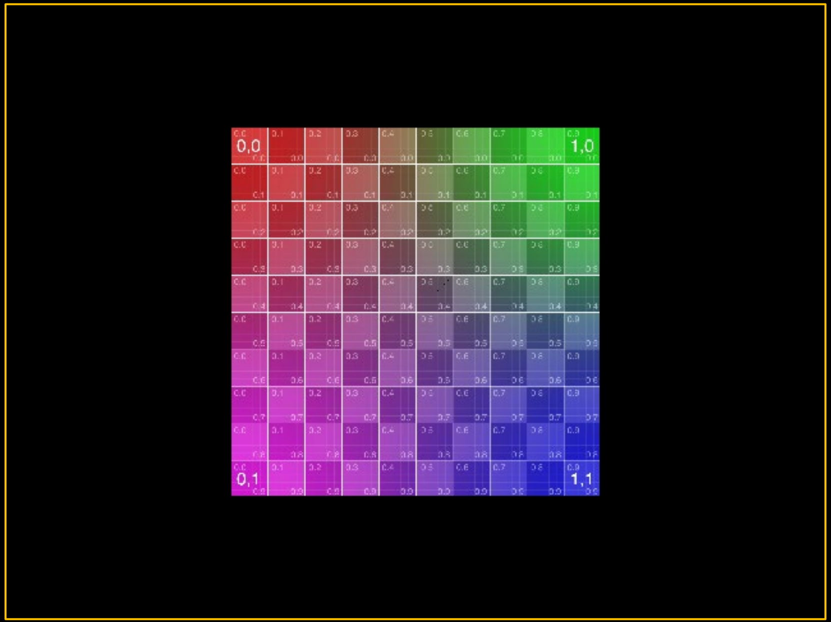
Rasterization: Textures & Vertex Attributes

- Texture::LoadFromFile
  - Hint: Once you have the SDL_Surface, you can access it just as the back buffer.
    See the constructor in Renderer.cpp how to do that. (use m_pSurfacePixel to store the pixel array)

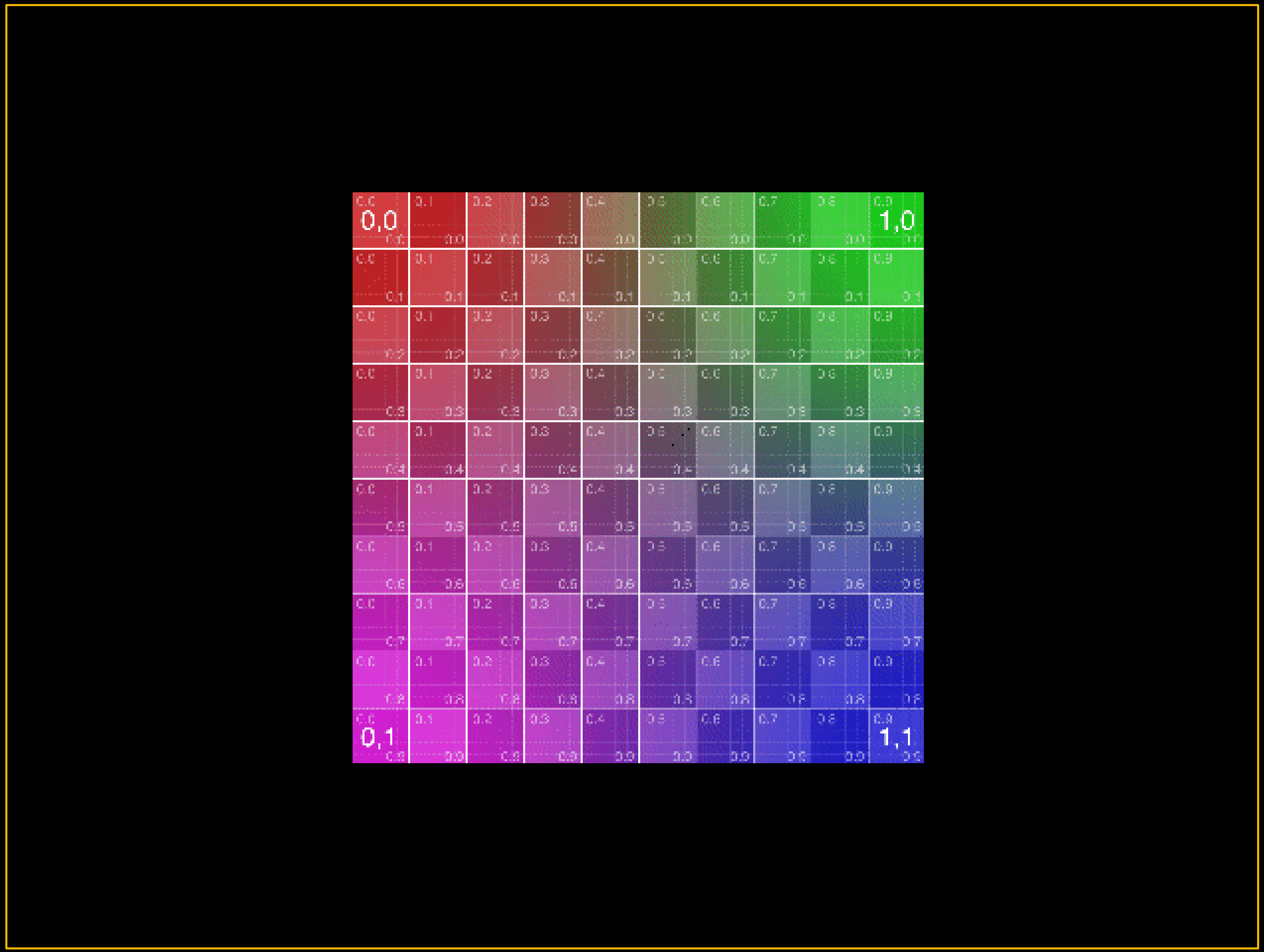
- Texture::Sample, how to sample the correct pixel
  - You can read a color from an SDL_Surface using SDL_GetRGB(…).
    - One the parameters is a pointer to the pixel you want to read → figure out how to get the index from the UV coordinates.
      - Hint: convert from [0,1] range to [0, width/height range] for U and V. The SDL_Surface holds both the width and height of the texture (w and h).
      - Hint: the U and V coordinate can be used for double arrays, but the pixels in the surface are defined in a single array. Find a way to convert from U and V coordinates to a single index → see how we reference a pixel in the back buffer ☺
    - The color you get is in the range [0,255]. Remap the color to [0,1] range for further calculations. Remember we multiple each component with 255 in the end when we write to the back buffer!

Rasterization: Textures & Vertex Attributes

- If finished the Texture class, **load** the given texture in your **renderer** (for now).
- When a pixel is inside our triangle, again **interpolate the UV coordinates**, just as you did with the color attribute. Use the **interpolated UV coordinate to sample** from the texture. Output the sampled color as your final color.
- If everything done right, you should get the following result:

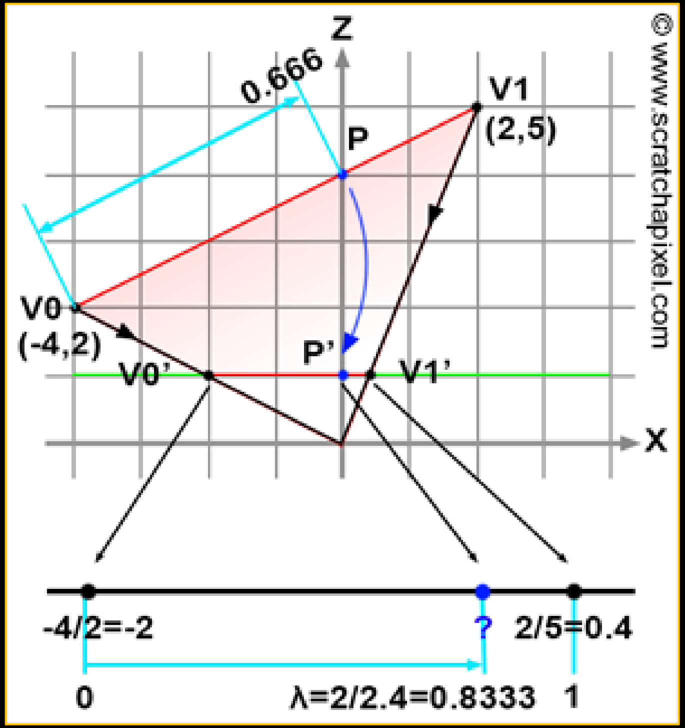


Rasterization: Depth Interpolation



Rasterization: Depth Interpolation

- What is happening? This doesn't look correct!
- Well, perspective projection preserves lines, but does not preserve distances! Let's look what this means.



$$P - V_0 = (0,4) - (-4,2) = (4,2)$$

$$V_1 - V_0 = (2,5) - (-4,2) = (6,3)$$

$$\downarrow$$

$$\frac{||P - V_0||}{||V_1 - V_0||} = \mathbf{0.666}$$

$$P' - V'_0 = (0,1) - (-2,1) = (2,\mathbf{0})$$

$$V'_1 - V'_0 = (0.4,1) - (-2,1) = (2.4,\mathbf{0})$$

$$\downarrow$$

$$\frac{||P' - V'_0||}{||V'_1 - V'_0||} = \mathbf{0.833}$$

Rasterization: Depth Interpolation

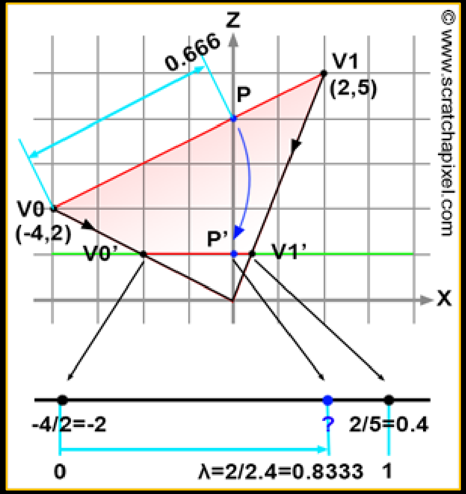
- As you can see, the ratio is not preserved! If we want to use linear interpolation, we need to make sure we take this into account!
- This also means our depth buffer is NOT correct (remember we also linearly interpolate the values of the vertex while this is not the correct interpolation after projection: $\boldsymbol{W_0 * V_0 + W_1 * V_1 + W_2 * V_2}$!
- The fix is easy:

$$P - V_0 = (0,4) - (-4,2) = (4,2)$$
$$V_1 - V_0 = (2,5) - (-4,2) = (6,2)$$

$$\frac{||P - V_0||}{||V_1 - V_0||} = \mathbf{0.666} \longrightarrow 2 * 0.333 + 5 * 0.666 = \mathbf{4}$$

$$P' - V'_0 = (0,1) - (-2,1) = (2,\mathbf{0})$$
$$V'_1 - V'_0 = (0.4,1) - (-2,1) = (2.4,\mathbf{0})$$

$$\frac{||P' - V'_0||}{||V'_1 - V'_0||} = \mathbf{0.833} \longrightarrow 2 * 0.167 + 5 * 0.833 = \mathbf{4.5} \longrightarrow \frac{\mathbf{1}}{\mathbf{2}} * 0.167 + \frac{\mathbf{1}}{\mathbf{5}} * 0.833 = \mathbf{0.25} \rightarrow \frac{\mathbf{1}}{\mathbf{0.25}} = \mathbf{4}$$



P_x = (-4*0,333 + (2*0,666)
= 0

P_z = (2*0,333) + (5*0,666)
= 4

Rasterization: Depth Interpolation

- The mathematical proof can be daunting, so let's just ignore that ☺. So, to correctly interpolate values we:
  - We calculate the correct interpolated depth by using (**don't forget to store this value in the depth buffer**):

$$Z_{\text{Interpolated}} = \frac{1}{\frac{1}{V0_z}w0 + \frac{1}{V1_z}w1 + \frac{1}{V2_z}w2}$$

  - Divide each attribute by the original vertex depth and interpolate as well (almost like the depth interpolation:

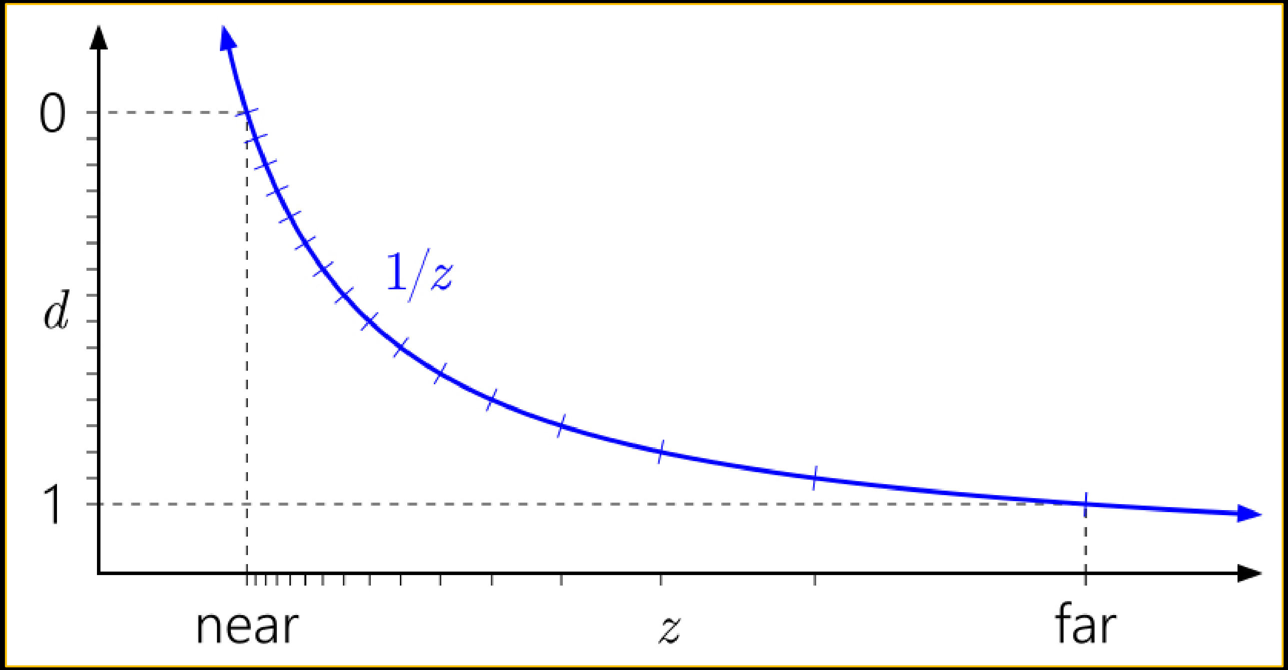
$$UV_{\text{Interpolated}} = \frac{V0_{uv}}{V0_z}w0 + \frac{V1_{uv}}{V1_z}w1 + \frac{V2_{uv}}{V2_z}w2$$

  - Take the correct depth into account by multiplying the interpolated value with the correct depth.

$$UV_{\text{Interpolated}} = \left(\frac{V0_{uv}}{V0_z}w0 + \frac{V1_{uv}}{V1_z}w1 + \frac{V2_{uv}}{V2_z}w2\right) Z_{\text{Interpolated}}$$

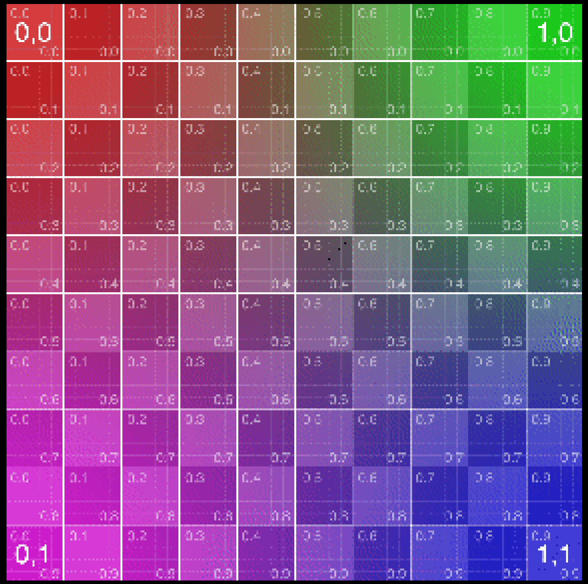
Rasterization: Depth Interpolation

- We need to do this for every attribute we interpolate (color, uv, normal, etc.)! Don't forget to consider the correct depth though.
- Doing this fixes our problem, but it creates an interesting side effect.
  - The depth buffer is no longer linear!



https://developer.nvidia.com/content/depth-precision-visualized

Rasterization: Depth Interpolation



Rasterization : What to do?

- Add the option to rendering using two different primitive topologies:
  - TriangleList
  - TriangleStrip

- Implement the texture class, which makes you able to load an .png and sample from that texture using UV coordinates.

- Use correct depth interpolation, for every attribute!

- Optional: fix the black lines between your triangles ☺

GOOD LUCK!