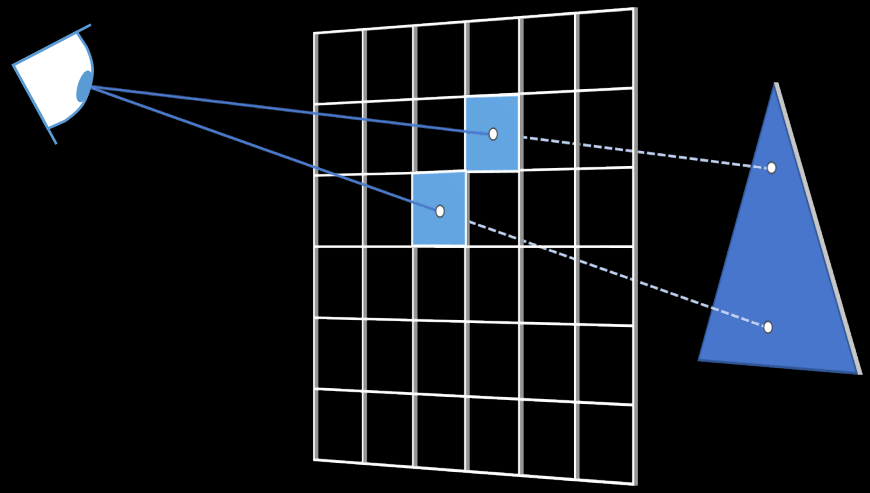
GRAPHICS PROGRAMMING I

SOFTWARE RASTERIZATION PART I

Rasterization vs Ray Tracing

Ray Tracing

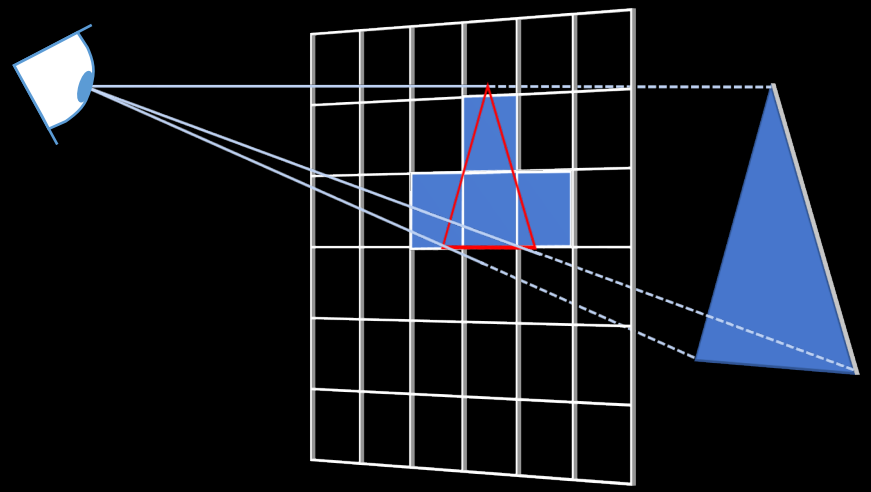
```
for each pixel (cast a ray)
    for each triangle
        does ray hit triangle?
```

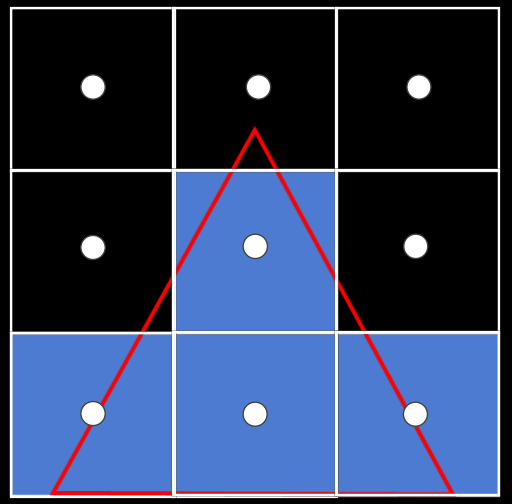


Rasterization vs Ray Tracing

Rasterization

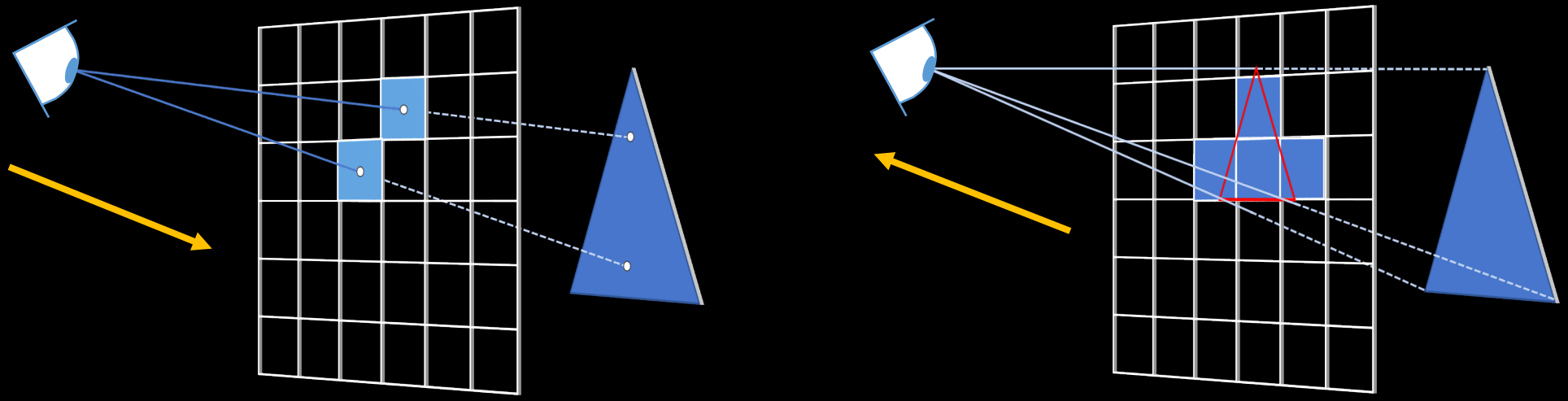
```
for each triangle (project)
    for each pixel
        is pixel in triangle?
```





Rasterization vs Ray Tracing

- A ray tracer is often called image centric, because for every pixel in the view plane we cast a ray into the scene and check for collision.
- A rasterizer is often called object centric, because we project the primitive onto the view plane and than check if a pixel overlaps with the primitive.
- So, in rasterization, we do the ”opposite” of ray tracing.



Rasterization: Why?

- As you've probably noticed, the loops of both techniques are swapped. Why would we do this?

Ray Tracing

```
for each pixel (cast a ray)
    for each triangle
        does ray hit triangle?
```

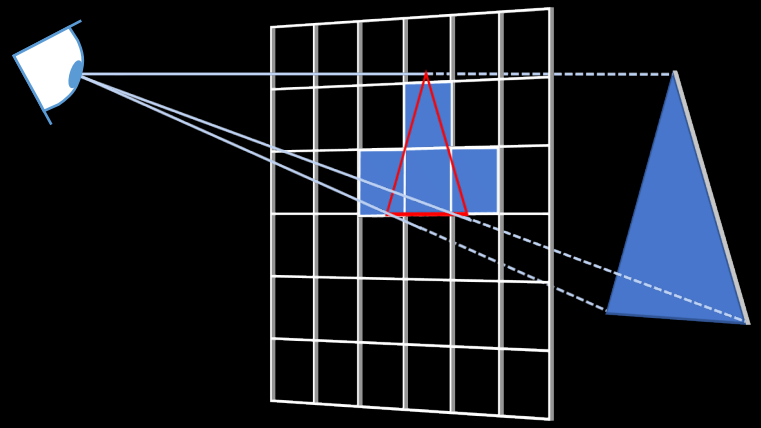
Rasterization

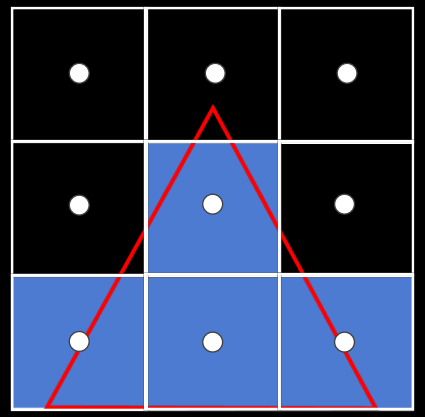
```
for each triangle (project)
    for each pixel
        is pixel in triangle?
```

- It's just a different technique to solve the visibility problem. A lot of people state that rasterization is always faster than ray tracing. This is not true! It heavily depends on the scene complexity, data organization and the optimization techniques used.
- In general, rasterization is much faster because:
  - better convergence when projecting.
  - GPU optimized to do these kind of operations. ☺

Rasterization: Algorithm

- Let's take a closer look at the rasterization algorithm.
- As mentioned before the algorithm exists out of two parts:
  - projecting 3D geometry onto our 2D view plane → PROJECTION STAGE
  - Check if a pixel of our 2D view plane overlaps with our projected geometry → RASTERIZATION STAGE

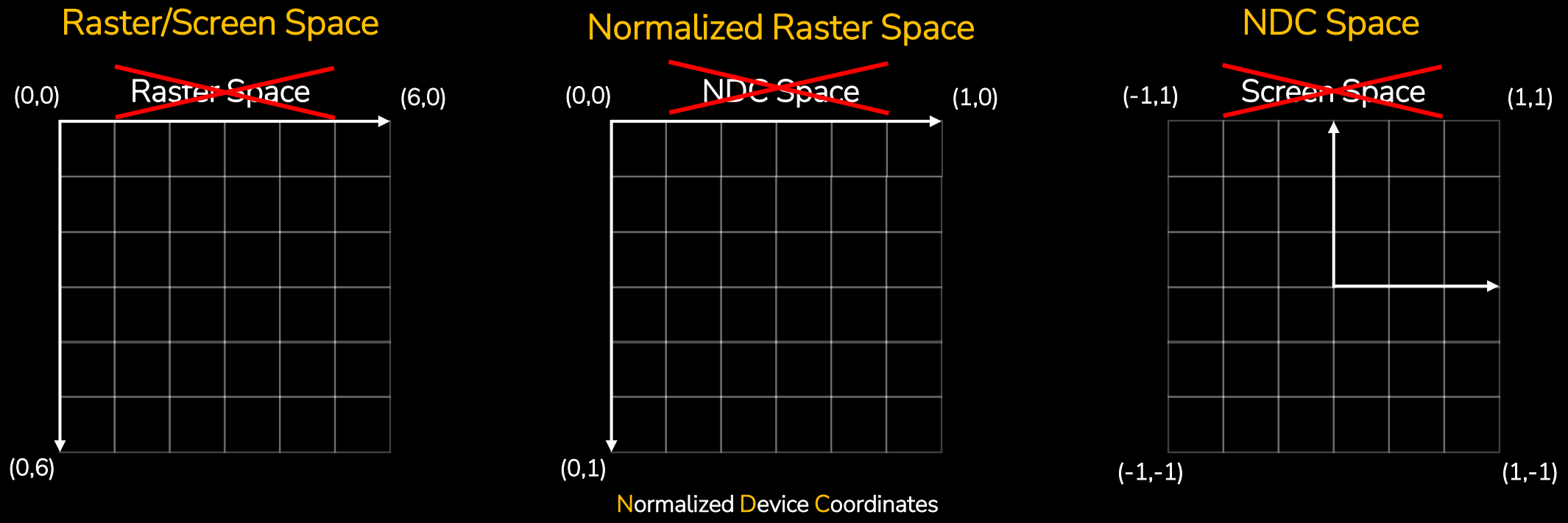




- A lot of books and online resources explain rasterization in that order. Let's do the opposite and start programming our rasterizer with coordinates that already have been projected.
- What coordinates are you talking about? Well, remember this?

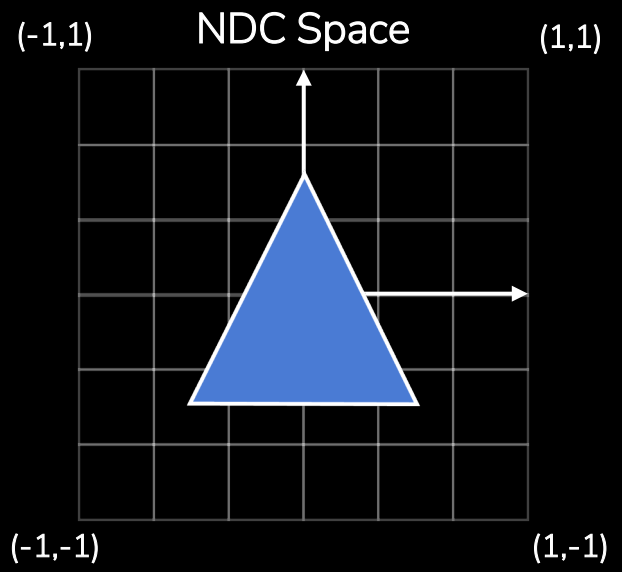
Rasterization: Coordinate Systems

- When we talked about sampling, we introduced some different coordinate systems.
- As it often happens, programmers use different conventions. This is no different! Let us take over the most common terminology used in rasterization!



Rasterization: Rasterization Stage

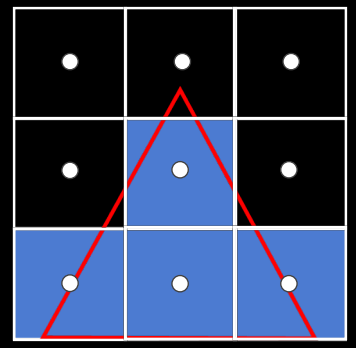
- To start with the rasterization stage, define a triangle with the following NDC coordinates:
  - Vertex 0: position(0.f, .5f, 1.f)
  - Vertex 1: position(.5f, -.5f, 1.f)
  - Vertex 2: position(-.5f, -.5f, 1.f)



- What we need to do in the rasterization stage, is finding out if a pixel is in the triangle.
- But before we can do this, our point should be, in what we call, screen space (often called raster space).
- We can easily convert our coordinates to screen space using the following formula:

$$ScreenSpaceVertex_x = \frac{Vertex_x + 1}{2} * ScreenWidth$$

$$ScreenSpaceVertex_y = \frac{1 - Vertexy}{2} * ScreenHeight$$

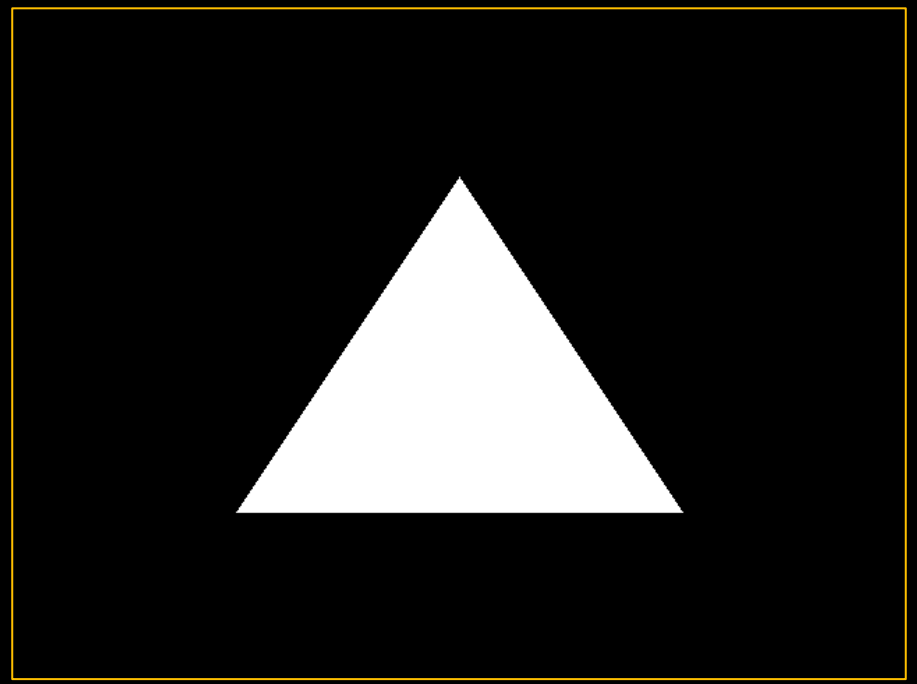


Rasterization: Rasterization Stage

- As you've might have noticed, that formula is the opposite calculation of what you did in your ray tracer!
- Now that our point is in the correct space, we can find out if a pixel is in the triangle. We've seen this before, right?
- Just as with the ray tracer you take the cross products and check the signs of the returned signed areas.
- This time you don't do the inside-outside test with an intersection point, but with the pixel itself! So, define the pixel as an 2D point: Vector2(p_x, p_y). Of course, don't use the pixel directly for the cross products. Remember we need the vector pointing from the current vertex to the pixel!
- You can take the cross product of 2D vectors as well. In contrast to the Vector3 implementation, it just returns the signed area, not a perpendicular vector scaled with the signed area. Different libraries might have different functions!
- We usually store our vertices in screen space in 3D (storing the initial depth value). But we are just interested in the signed area, so for now you can just use the 2D cross products and ignore the depth value!

Rasterization: Rasterization Stage

- So, loop over all your triangles (in this case, just one triangle) with its vertices already defined in NDC space. Go over all the pixels and check if a pixel is in the triangle, using the same technique as last week (Triangle Intersection Test).
  If it is inside the triangle, color the pixel white.
- If everything is done correctly, you should get the following result.

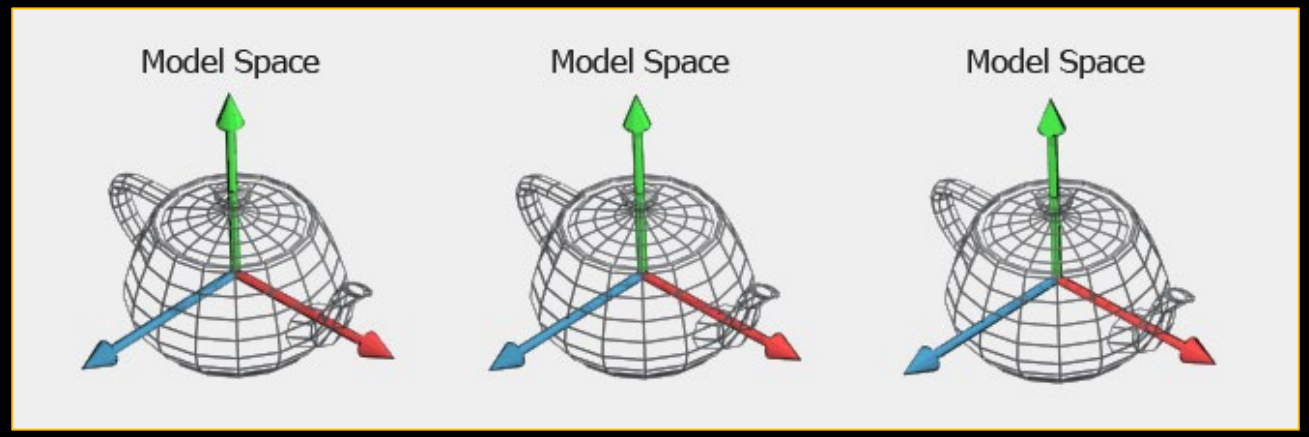


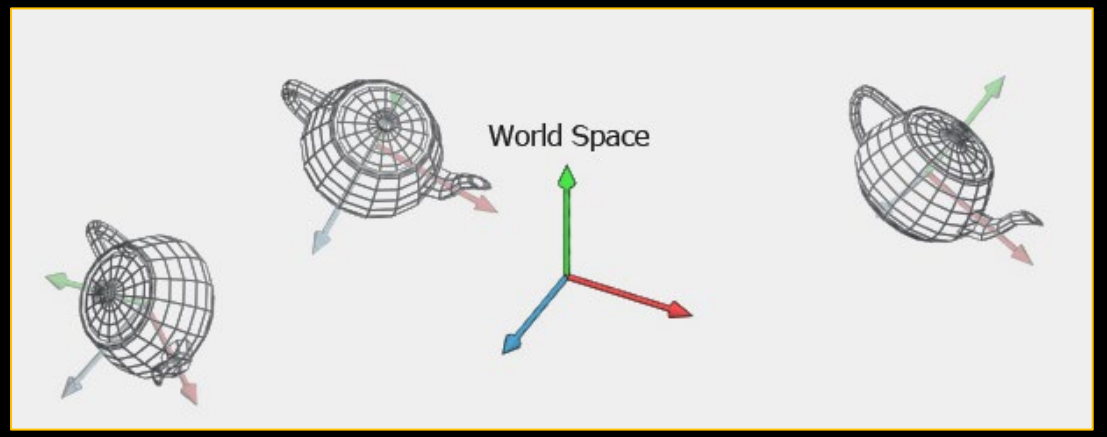
V_0 = (0, .5, 1)
V_1 = (.5, -.5, 1)
V_2 = (-.5, -.5, 1)

→ Notice **clockwise** order

Rasterization: Projection Stage

- That is the rasterization stage in essence. Of course, there are other things we should still do, but let's first look at the PROJECTION STAGE.
- Usually, you don't define a triangle in NDC, but in world space. In other words, the triangle is somewhere in the world, with its coordinates relative to the world axis.





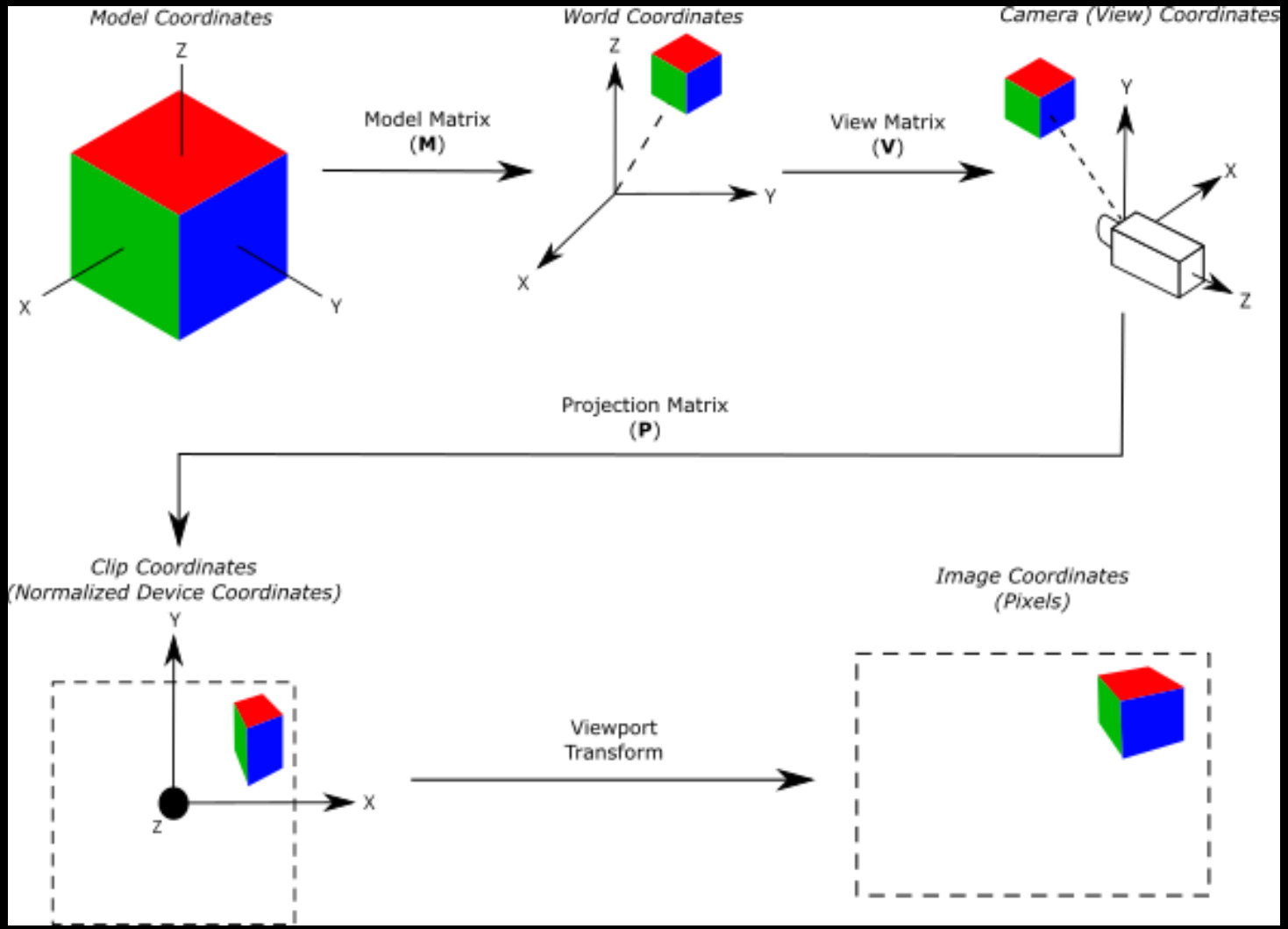
codinglabs.net

- In your ray tracer you probably defined the points in model space and transformed them in world space using a matrix.

Rasterization: Projection Stage

- So how to we go from these World Space Coordinates to Normalized Device Coordinates?
- We do this in several step:
  - First, we want to know how our triangle in world space relates to the camera. (VIEW SPACE)
    It is our camera that defines what we see in our scene!
  - To know the coordinates relative to another coordinates system (axis), we multiply the data with a matrix that defines that coordinate system. (VIEW MATRIX)
  - If you remember, when we created the ONB of our camera, we created a matrix that defines our camera. To be accurate, it's a matrix that tells us how we go from camera space to world space. If you recall, when we multiplied our rays with this matrix, we moved them to a certain location (and with a certain rotation) in world space based on the position and rotation of our camera.
  - Now, we want to do the opposite! We want to go from these world space coordinates of our triangle to camera space coordinates (coordinates relative to our camera). How do we do this?
  - Remember from math that we can do the opposite transformation by multiplying any point or vector with the inverse of that transformation matrix? Well, same applies here! If we want to go to camera space, we multiply every point with the inverse of our ONB!
    (Camera ONB / Camera WORLD Matrix > Transforms from View to World Space)
    (Camera VIEW Matrix > Inverse of Camera ONB > Transforms from World to View Space)

Rasterization: Projection Stage



- Image resembles the FULL projection stage (this is also what we want to implement in the end)

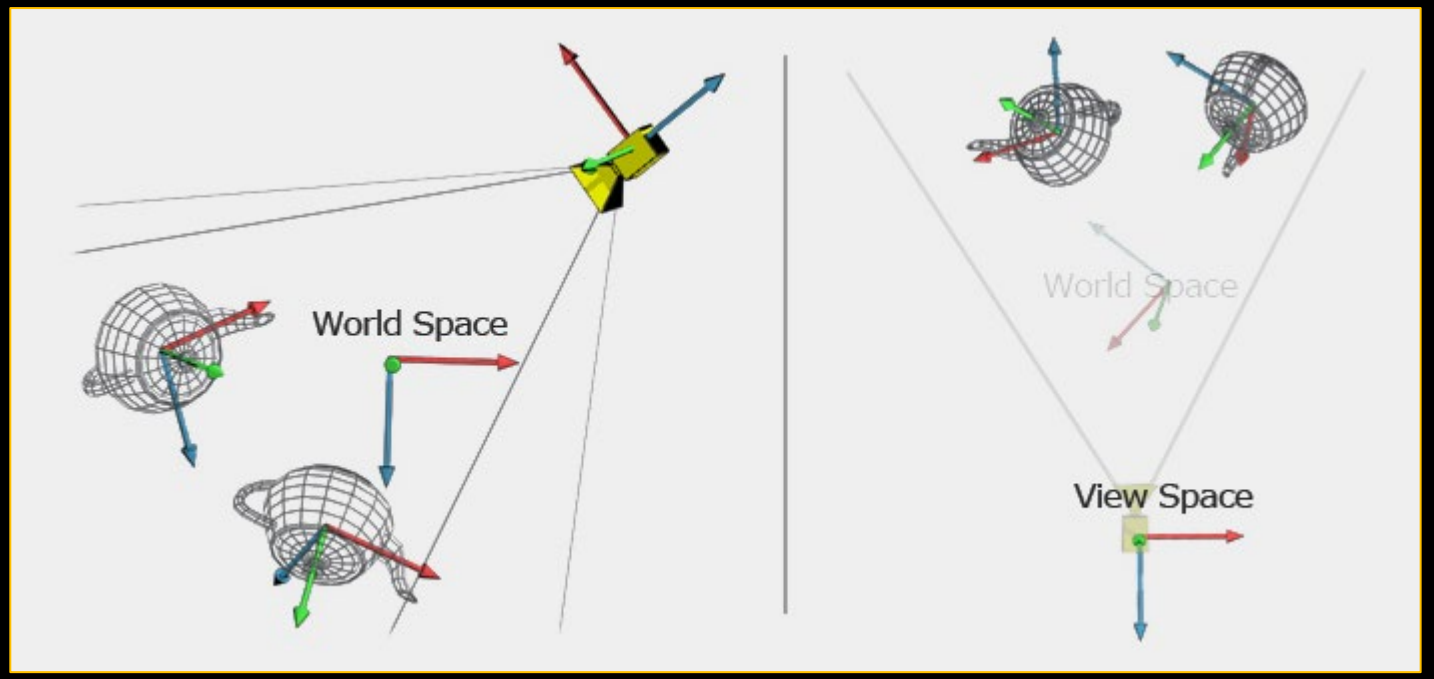
- Model > World Space (World Matrix)
- World > View Space (View Matrix)
- View > Clipping Space (NDC) (Projection Matrix)

Using a WorldViewProjection (WVP) Matrix:
Model > Clipping Space
(WVP Matrix > World*View*Projection)

- NDC > Raster Space

Rasterization: Projection Stage

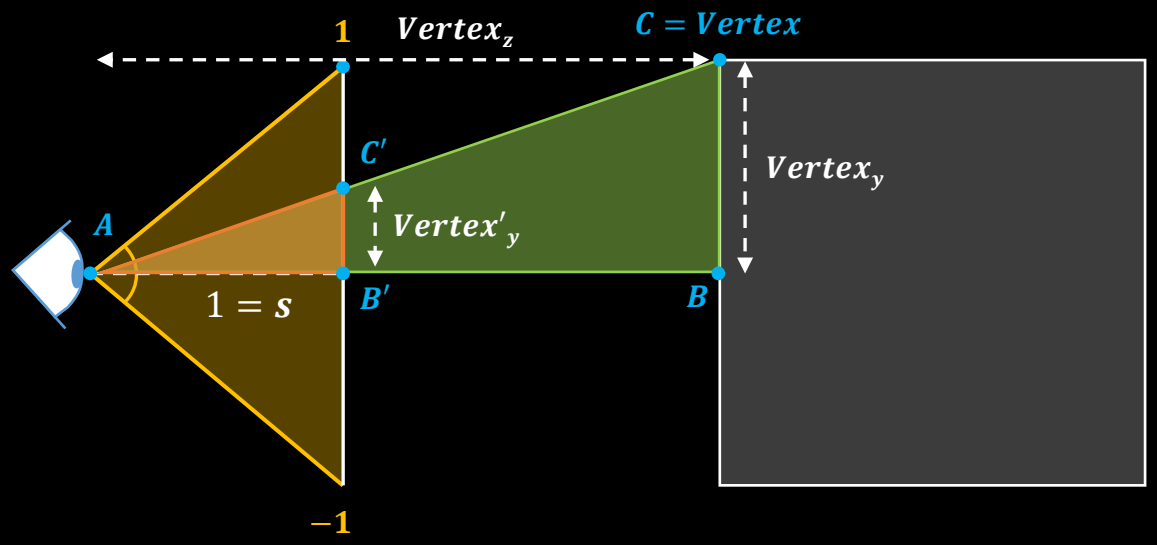
- Steps:
  1. Transform all vertices from world space to camera space, often called view space, by multiplying them with the inverse of the camera matrix (ONB).



codinglabs.net

Rasterization: Projection Stage

- Now that our vertices are in view space, what is next:
  - These coordinates are not in the required [-1, 1] range of NDC. Let's see how we go from this view space to these NDC coordinates.
  - What we want to do is project our 3D point onto our 2D view plane. So how do we project a 3D point onto a 2D plane?



$$\frac{BC}{AB} = \frac{B'C'}{AB'}$$

$$\downarrow$$

$$\frac{Vertex_y}{Vertex_z} = \frac{Vertex'_y}{1}$$

$$\downarrow$$

$$\frac{Vertex_y}{Vertex_z} = Vertex'_y$$

- This is how you project a 3D point onto the view plane. This is called the PERSPECTIVE DIVIDE!

Rasterization: Projection Stage

- We are using a left-handed system, which means that our positive Z-axis is pointing into the screen. The $ViewSpaceVertex_z$ currently resembles the distance of a vertex relative to the camera (in world units).

$$ProjectedVertex_x = \frac{ViewSpaceVertex_x}{ViewSpaceVertexz}$$

$$ProjectedVertex_y = \frac{ViewSpaceVertex_y}{ViewSpaceVertexz}$$

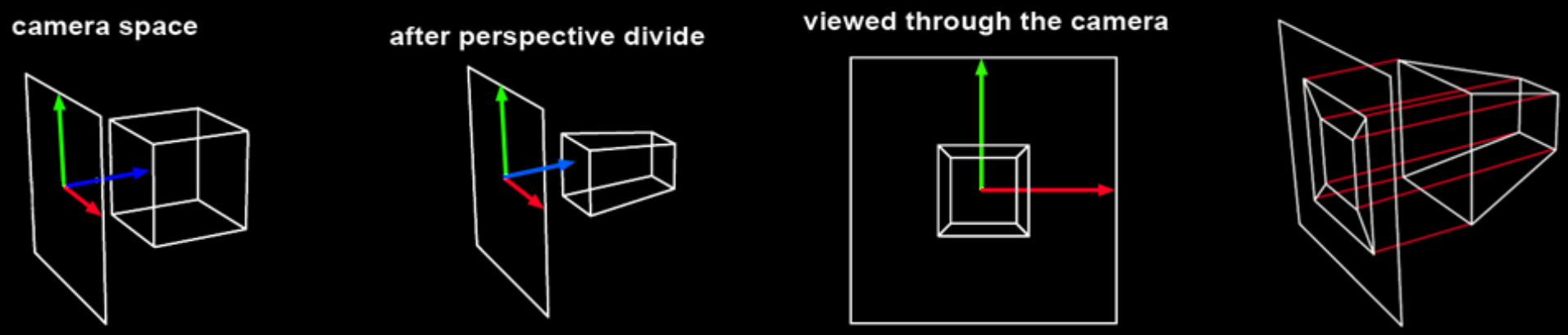
- What about our 'projected' z-component? For now, you can just store ViewSpace Z. ☺

$$ProjectedVertex_z = ViewSpaceVertex_z$$

- So, after this perspective divide your 3D point is in what we call PROJECTION SPACE, with the z-axis point into the screen.

Rasterization: Projection Stage

- The following images might help you understand what is happening!
- As you'll see the only thing you do is you project the point onto the screen. This is just an orthographic projection. But because we divide by the z-component (which is different for points in the distance), we mimic perspective distortion!
- To quote Edwin Catmull: "Screen-space is also 3D, but the objects have undergone a perspective distortion so that an orthogonal project of the object onto the x-y plane, would result in the expected perspective image."



Rasterization: Projection Stage

- There is one thing missing. Can you guess what? ☺
- We didn't consider the camera settings! As you might remember from your ray tracer, most of the time our screen isn't a nice square. We need to consider the settings (size) of our camera!
- And as it turns out, we can just use the same logic, though inverted, as in our ray tracer! So, to make sure our point is projected relative to the size of our screen we just divide it with the aspect ratio (only x-component) and the field of view (FOV).

$$ProjectedVertex_x = \frac{ProjectedVertex_x}{AspectRatio * FOV}$$

$$ProjectedVertex_y = \frac{ProjectedVertex_y}{FOV}$$

- The order of this and the previous operation doesn't matter. You can first put the point relative to the camera and then do the perspective divide. It should give you the same result.

Rasterization: Projection Stage

- Our point is now in projection space with NDC [-1,1] for the x- and y-component.
  We can now rasterize it as well.
- Recap, transforming an 3D point to NDC requires the following steps:
  1. Transform the point from world space to camera space, often called View Space, by multiplying it with the inverse of the camera matrix (ONB).
  2. Take the camera settings/size into account by dividing the point with the aspect ratio (only x-component) and the field of view.
  3. Apply the perspective divide to get the correct perspective distortion. To do this you just divide both the x- and y-component by the z-component.
     For now, you just store the viewSpace z-component in the z-component of the projected point.
     After the divide, the point is in Projection Space.
- Once our point is in projection space, we still need to put it in Screen Space before we can start comparing it with actual pixels (rasterization).
- So, we do the following transformations before rasterization:
  - World Space → View Space → Projection Space → Screen Space

Rasterization: Projection Stage

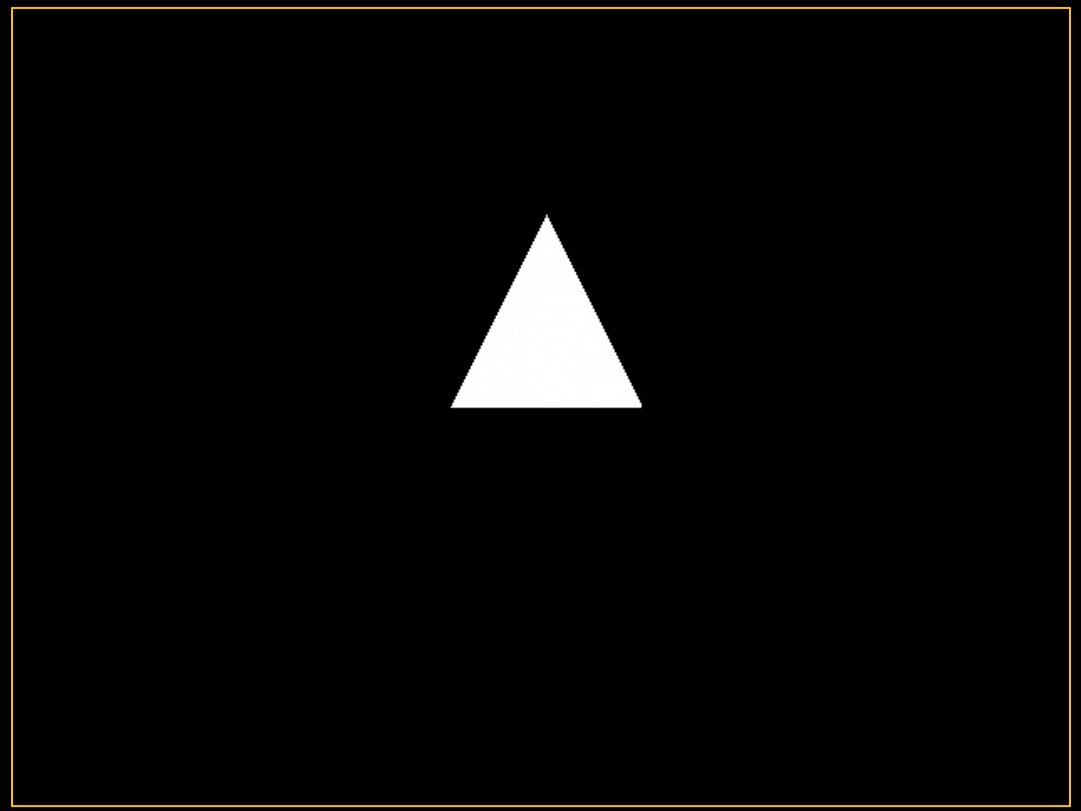
- Thus, all we do in the projection stage is doing the opposite of what we did with our ray tracer.
- As a reminder, applying a matrix just puts the coordinates of a point relative to this new axis system. That's why we call it spaces. We represent a point relative to a certain space / coordinate system.
- Next step is to implement the camera. (Same 'update' logic you used for the RayTracer ☺) We still need the ONB to do the inverse transformation. Moving the camera doesn't change.
- Define a triangle in world space, transform its vertices accordingly and just use the previously written rasterization stage. Hint: put the transformation logic in a function.

```
//Function that transforms the vertices from the mesh from World space to Screen space
void VertexTransformationFunction(const std::vector<Vertex>& vertices_in, std::vector<Vertex>& vertices_out) const;
```

- Define a triangle with the coordinates: v0(0.f,2.f,0 .f), v1(1.f,0 .f,0 .f), v2(-1.f,0 .f,0 .f)
- Our camera with an origin(0.f, 0.f, -10.f) and FOVTotalAngle(60.f)

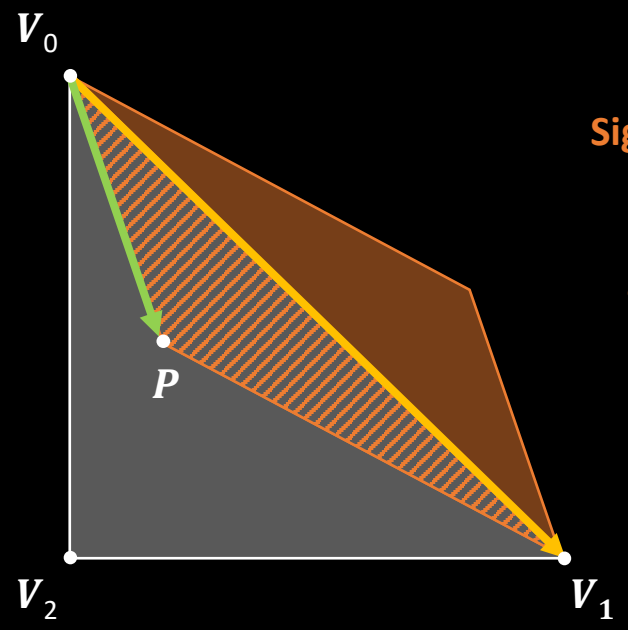
Rasterization: Projection Stage

- If you've done everything correctly you should see the following triangle. You should also be able to move and rotate around the triangle, because you already have a working camera, right?! ☺



Rasterization: Barycentric Coordinates

- When we did the inside-outside test, we checked if a point was on the “right” side of the edges. As you recall, when we take the cross product of 2D vectors we get a scalar value, representing the signed area of the parallelogram.
- This scalar value can also be interpreted in another way! Let’s have a look.



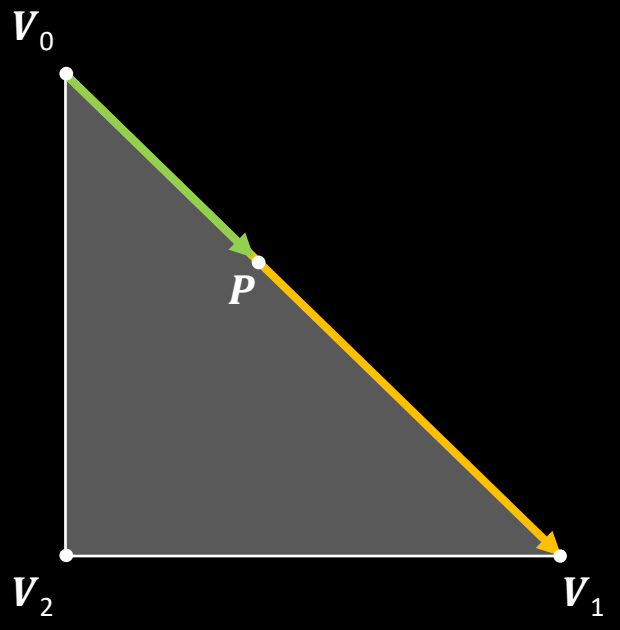
$\text{SignedAreaParallelogram}_{P,V0,V1} = \text{Cross}(V_1 - V_0, P - V_0)$

↓

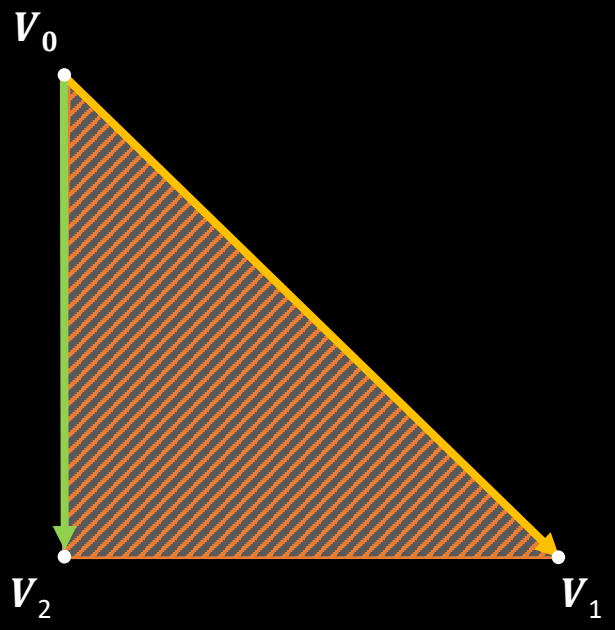
$\text{SignedAreaTriangle}_{P,V0,V1} = \text{Cross}(V_1 - V_0, P - V_0) / 2$

Rasterization: Barycentric Coordinates

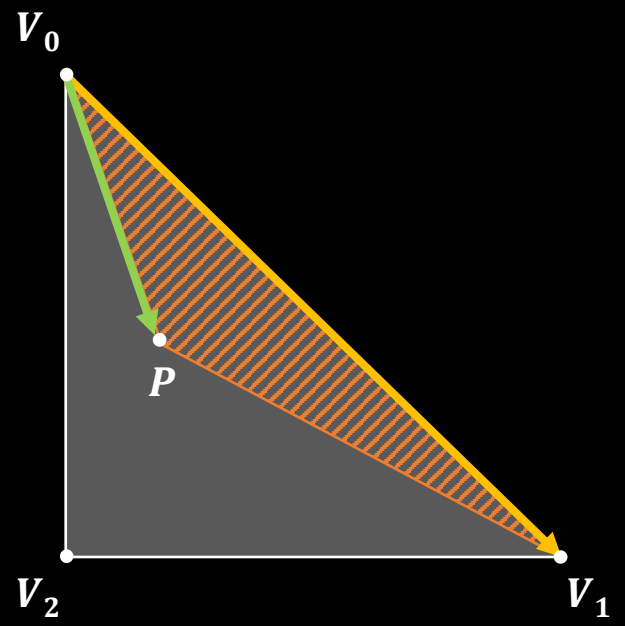
- If we divide this signed area with the total area of the entire triangle, we can have different scenario's:



$\text{SignedAreaTriangle}_{P,V0,V2} = 0$



$\text{SignedAreaTriangle}_{P,V0,V2} = 1$

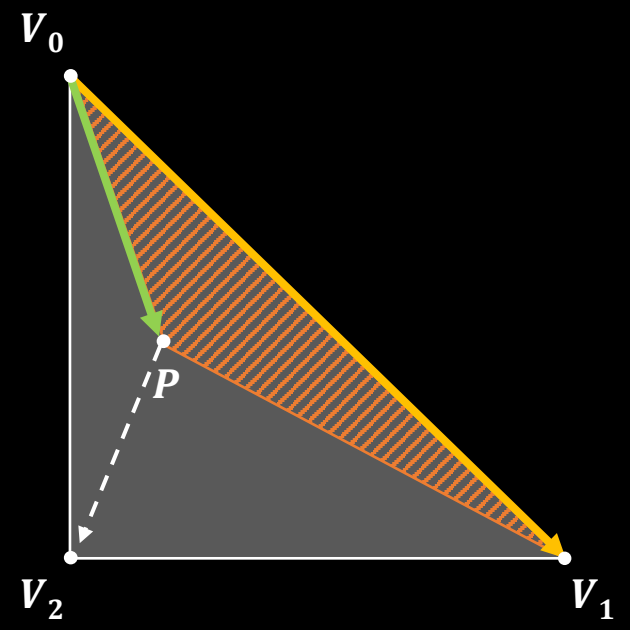


$\text{SignedAreaTriangle}_{P,V0,V2} = 0.4$

- So, as you can see, the returned value says how much of the triangle's area, formed by the two vectors, covers the area of the total triangle. It's a sort of ratio.

Rasterization: Barycentric Coordinates

- More importantly, it's a scalar value that can be interpreted as how close the pixel is to the vertex that not part of this smaller triangle.



- In other words, it's a weight value we can use together with the vertex to determine, relative inside the triangle, how close our pixel is to the actual vertex.
- We do this for all three vertices!

Rasterization: Barycentric Coordinates

- If you do this for every vertex, and thus every small triangle, you will have three weights that map relative to the triangle. We can use these to figure out where the pixel is inside the triangle (in Barycentric Coordinates), using the following formula:

$$P_{insideTriangle} = W_0 * V_0 + W_1 * V_1 + W_2 * V_2$$

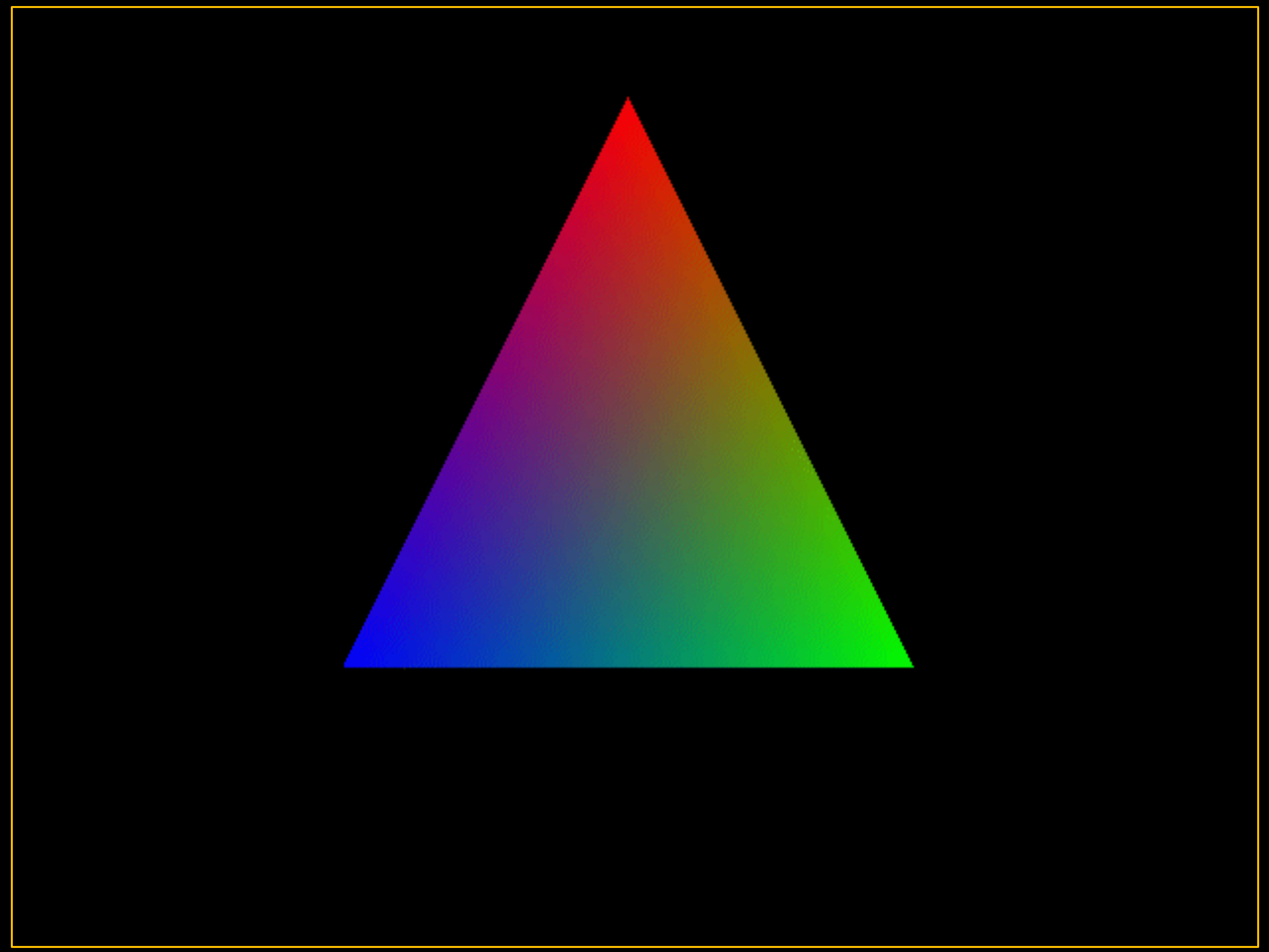
- It's important to keep track which cross product represents which weight (vertex opposite to the small triangle):
  - Weight of V_0 (W_0) is defined by: Vector($P - V_1$) and Vector($V_2 - V_1$) [weight = cross product]
  - Weight of V_1 (W_1) is defined by: Vector($P - V_2$) and Vector($V_0 - V_2$)
  - Weight of V_2 (W_2) is defined by: Vector($P - V_0$) and Vector($V_1 - V_0$)
  - Total Triangle Area = $W_0 + W_1 + W_2$

- When you add all the weights (that have been divided by the total triangle area) you should get a total value of 1! If not, there is something wrong!
- So why do we care about these coordinates?

Rasterization: Barycentric Coordinates

- Imagine every vertex has a color value encoded. If you check if a pixel is inside triangle, and it is, which color will the pixel have?
- Using barycentric coordinates you know the weighting relative to each vertex. What you can do is interpolate all three colors (from the three vertices) using the weights to get to your final color!
- So, the next step in your rasterizer:
  - Use the predefined Vertex Struct (It contains multiple attributes, but for now we are only interested in the position & color attributes).
  - When checking if the pixel is inside the triangle, store the results of the cross products. If it is inside the triangle, calculate the final weights!
    - You do this by dividing the area of current parallelogram (result cross product) by the area of total parallelogram (cross product of $\boldsymbol{V_1 - V_0}$ and $\boldsymbol{V_2 - V_0}$). (Or accumulate the individual weights)
    - There is no need to divide both areas by 2 (making it a triangle area), because we only care about the ratio.
  - Calculate the final color by interpolating every vertex color by its corresponding weight:
    - Interpolated Color = $\boldsymbol{VertexColor_0 * W_0 + VertexColor_1 * W_1 + VertexColor_2 * W_2}$
- Using the following triangle, you should get the result from the next slide:
  - V0: position(0,4,2) color(1,0,0); V1: position(3,-2,2), color(0,1,0); V2: position(-3, -2, 2) color(0,0,1)

Rasterization: Barycentric Coordinates

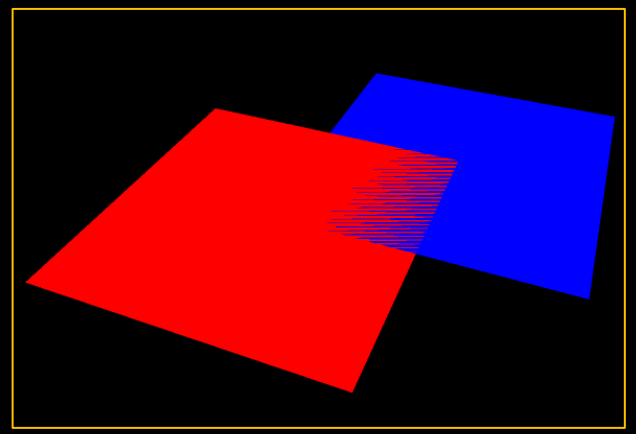


Rasterization: Depth Buffer

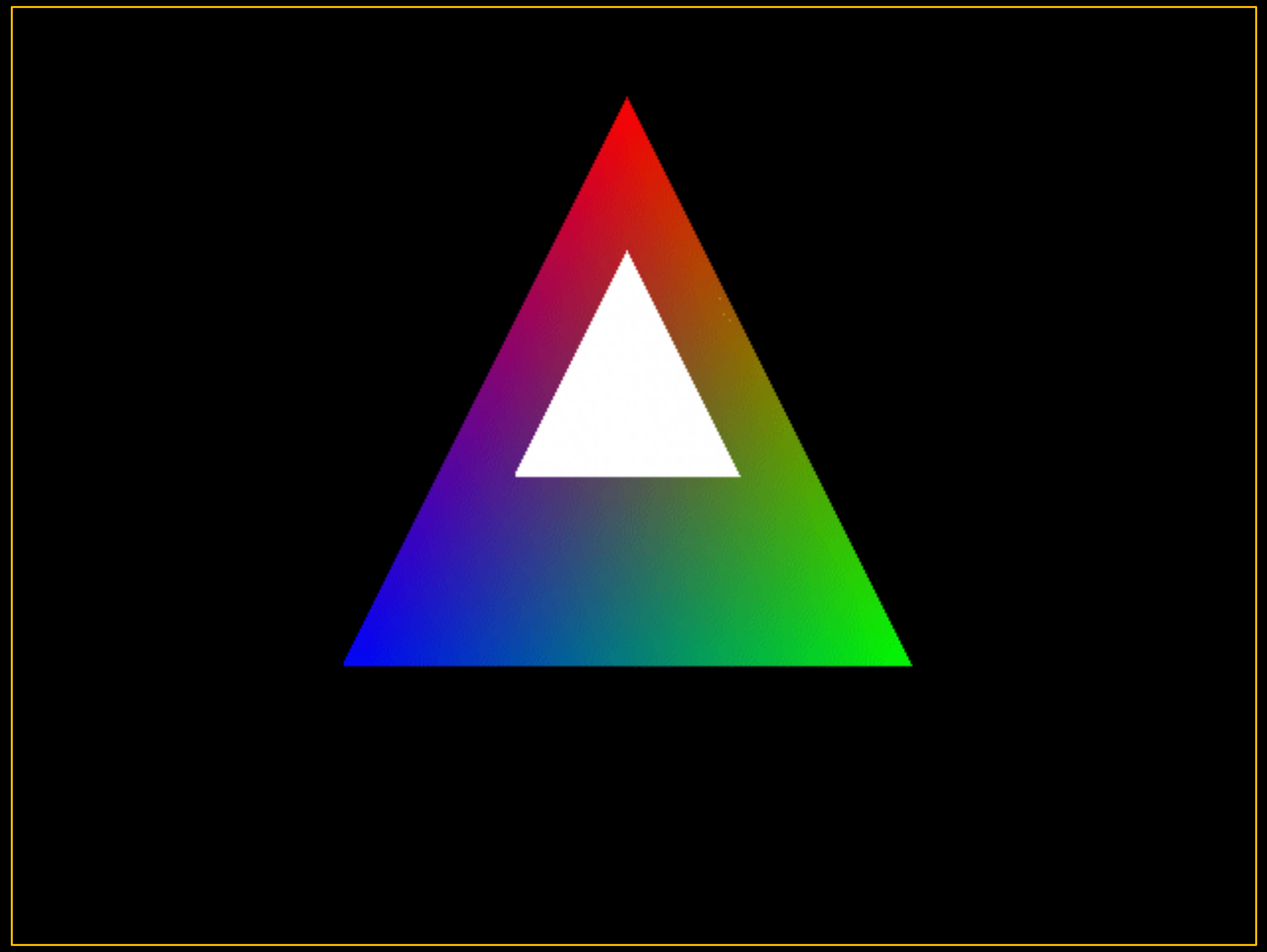
- One of the things we didn't consider when rendering is the depth of a triangle.
- If you want to render two triangles, which triangle is closer to our camera?
- Remember we just store the z-component in view space in our 3D projected point. We need this information when we rasterize other triangles. So, we need a way to store it. This is quite easy!
- We will create an array with the same size as the array of pixels. Instead of it being an array of RGB Colors, we just store floats (depth value / z-component). Call this array the DEPTHBUFFER.
- Whenever a pixel is inside the current triangle, we read from the depth buffer (which should be initialized with the maximum value of a float or in other words, points that are infinitely far away). If a pixels' depth value is smaller than the one in the depth buffer (so closer to the viewer), we render the pixel and we store its depth value in the depth buffer. This test is called the DEPTH TEST, writing the depth is called DEPTH WRITE.
- If we write or compare a depth value, which depth value should we take (vertex 0, 1 or 2)?
  - Well, we just interpolate between the different depth values again using the barycentric weights!

Rasterization: Depth Buffer

- Adjust your rasterizer to take depth into account and render two triangles:
  - Create depth buffer (array with the total number of pixels, holding just float values) – initialize all values with the maximum value of a float. The depth buffer holds the depth values of all the closest pixels.
  - When the pixel is inside the triangle, calculate its depth by interpolating between the different depth values.
  - Do the depth test (is this pixel closer than the one stored in the depth buffer). If the test succeeds, render the pixel and store the pixels' depth value as the new depth buffer value.
  - If two values are identical you should choose to either keep the current value/pixel and replace it with the next value/pixel. Generally, this is a bad thing, and it should be avoided. The effect is called depth or z-fighting.



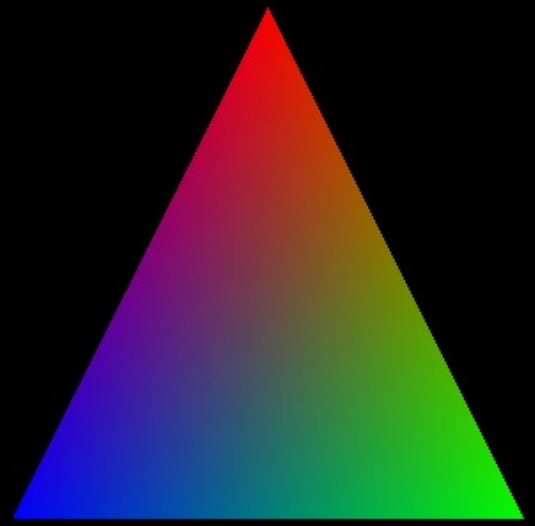
Rasterization: Depth Buffer



Rasterization: Optimizations

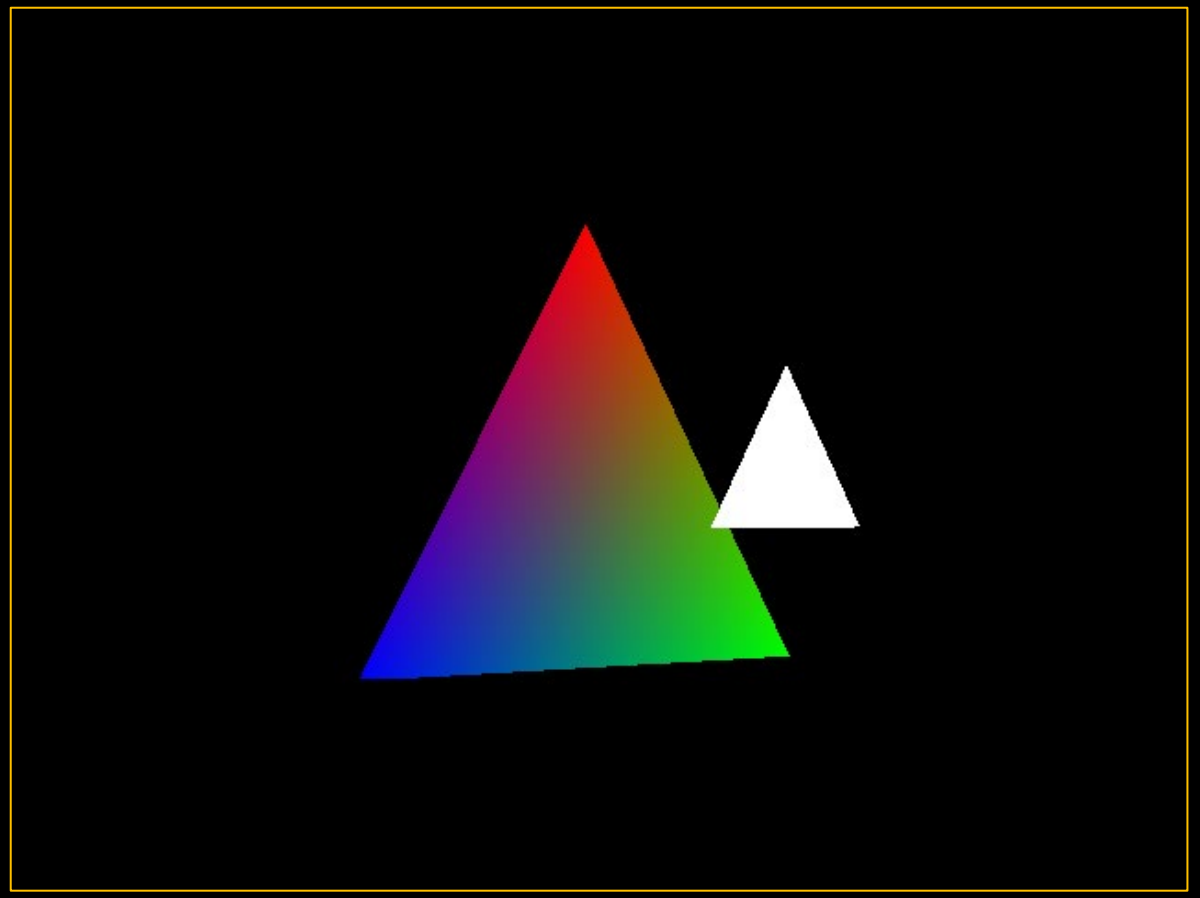
- Finally, right now we loop over **all the pixels** for every triangle. That is a lot of unnecessary pixels, **depending how big the triangle is**. We are only interested in the pixels that are **(potentially) covered** by our triangle. With our triangle being in screen space, we can easily find out which pixels might be rendered. In other words, we can eliminate a bunch of pixels that will **never** be covered by our triangle! How?
- We just calculate the bounding box of our triangle in screen space and only loop over those pixels. ☺
- Calculating the bounding box is not hard:
  - Find the **top left** point (smallest x-value and smallest y-value) and the **right bottom** point (largest x-value and largest y-value), based on the three vertices. (Hint: use std::max, std::min).
  - Make sure the two points defining the bounding box **don't exceed the screen boundaries**:
    - 0 >= xValues <= (width – 1) → 0 >= yValues <= (height - 1)
  - Think about the **types** of these points stored: float vs int vs uint.
- Instead of looping over all the pixels, you use those two values in your double for-loop. That's it! This small optimization will give you a lot of performance gain (if triangles are not too big or covering the entire screen).

Rasterization: Optimizations



Rasterization: What to do?

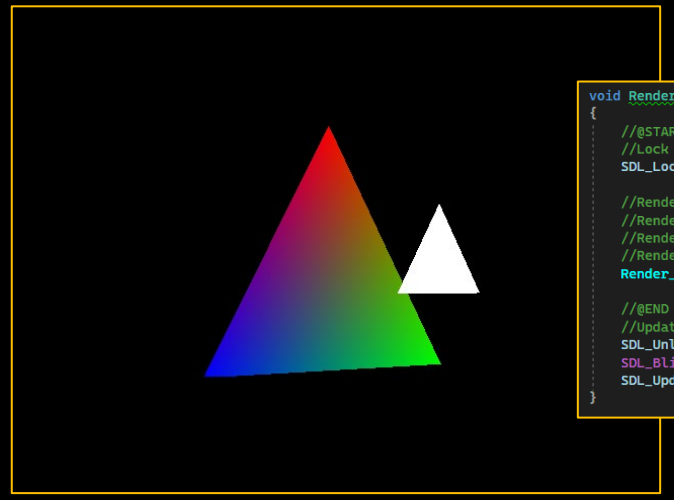
- And that's it for this week.



Rasterization: Lab 6 - Todos

- A lot of topics were covered during this lab, use the steps below to implement the topics one by one. Each step will add another portion to your rasterization pipeline, make sure that you understand each individual step before moving to the next one.
- The next slide also contains an overview of the pipeline for this week.

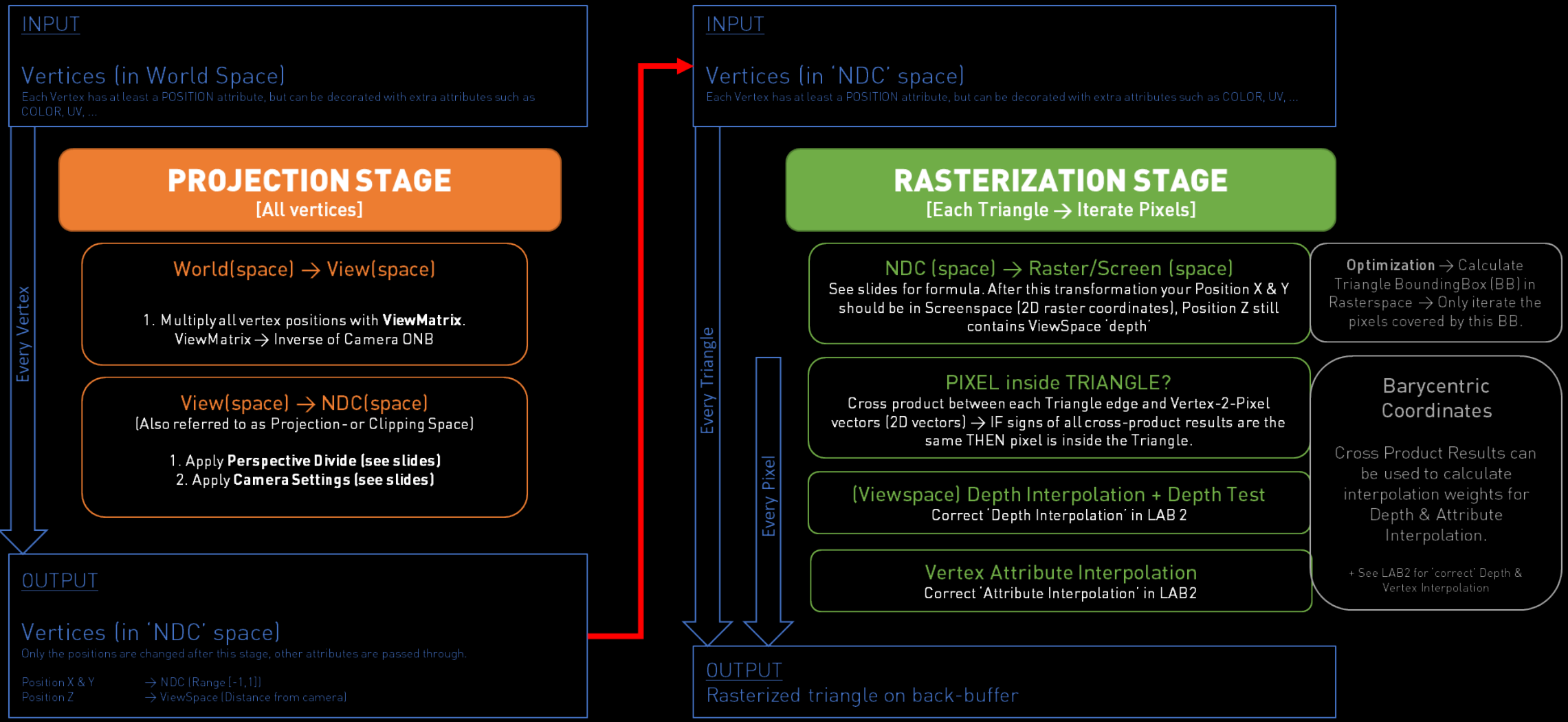
1. Rasterizer Stage Only
2. Projection Stage (Camera)
3. Barycentric Coordinates
4. Depth Buffer
5. BoundingBox Optimization



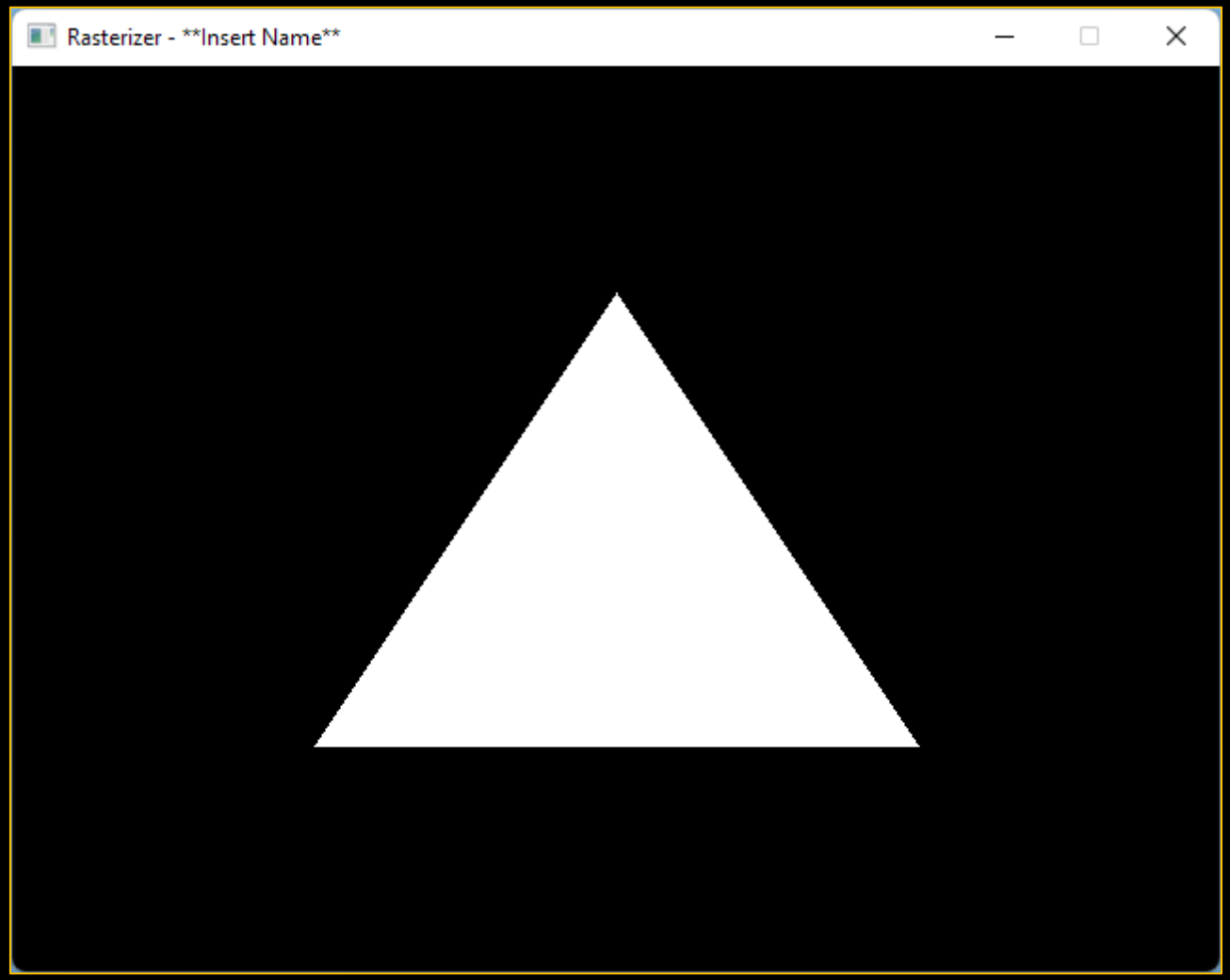
```
void Renderer::Render()
{
    //@START
    //Lock BackBuffer
    SDL_LockSurface(m_pBackBuffer);

    //Render_W1_Part1(); //Rasterizer Stage Only
    //Render_W1_Part2(); //Projection Stage (Camera)
    //Render_W1_Part3(); //Barycentric Coordinates
    //Render_W1_Part4(); //Depth Buffer
    Render_W1_Part5(); //BoundingBox Optimization

    //@END
    //Update SDL Surface
    SDL_UnlockSurface(m_pBackBuffer);
    SDL_BlitSurface( src: m_pBackBuffer,  srcrect: 0,  dst: m_pFrontBuffer,  dstrect: 0);
    SDL_UpdateWindowSurface(m_pWindow);
}
```

Tip: For each step/extension you could create a separate Render function, that way you can easily go back to a previous step. This is of course optionally but could help you with keeping things structured. (Each time you start a new step you can create a copy of the result of the previous step which you can use to expand on) – Once everything is working you can safely remove the intermediate Render functions.

Rasterization: Lab 6 - Todos



Rasterization: (1) Rasterization Only

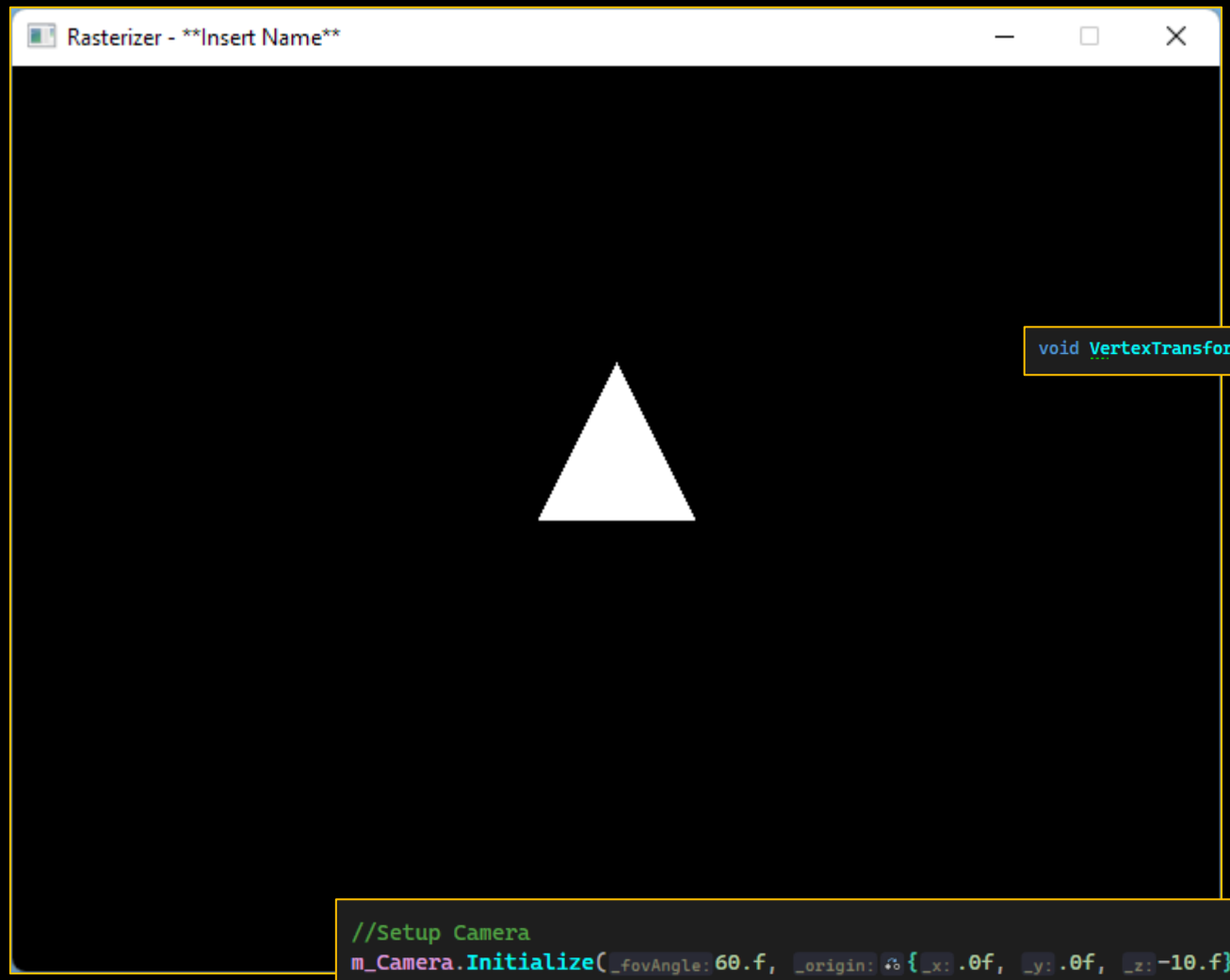


- **Objective**
  Rasterize the given triangle (each vertex is already defined in NDC space)

- You should convert to each pixel to Screen Space (pixel coordinates)
- Loop over all pixels, and check if the triangle covers the given pixel.
- Vertex vector can be defines in the render function itself

```
//Define Triangle - Vertices in NDC space
std::vector<Vector3> vertices_ndc
{
    {0.f, .5f, 1.f},
    {.5f, -.5f, 1.f},
    {-.5f, -.5f, 1.f},
};
```

Rasterization: (2) Projection Stage




```
//Setup Camera
m_Camera.Initialize(_fovAngle:60.f, _origin:{_x:.0f, _y:.0f, _z:-10.f});
```

- **Objective**
  First Project the given triangle (now, the vertices are defined in WORLD space). Rasterization stays the same as the previous step.
- Create a separate function that transforms a vector of WORLD space vertices to a vector of NDC space vertices (or directly to SCREEN space).

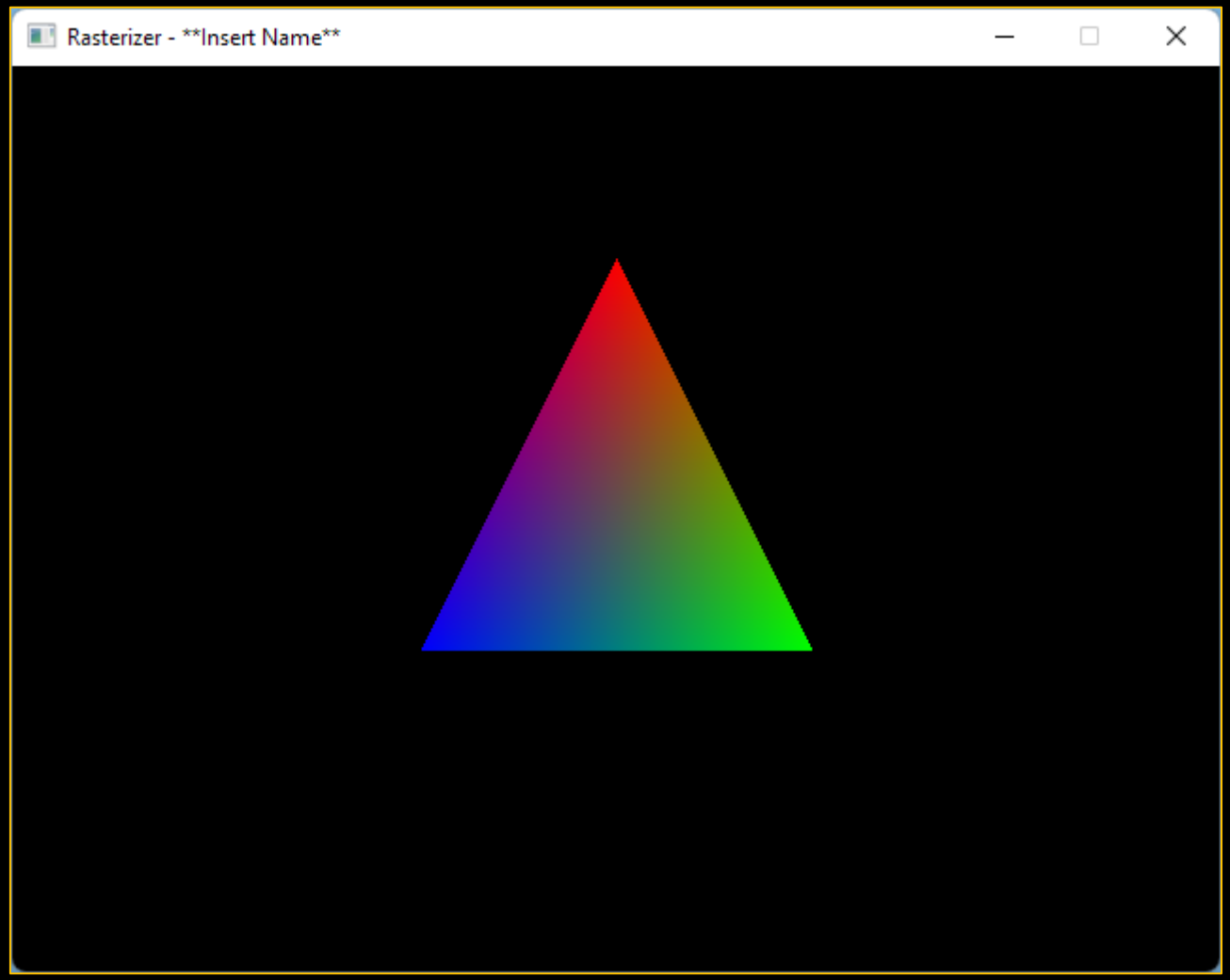
```
void VertexTransformationFunction(const std::vector<Vertex>& vertices_in, std::vector<Vertex>& vertices_out) const;
```

- You'll have to
  - Implement the Camera First!
  - Transform them with a VIEW Matrix (inverse ONB)
  - Apply the Perspective Divide
  - Convert to NDC > SCREEN space

Once all vertices are transformed, you can apply the Rasterization logic from the previous step

```
//Define Triangle - Vertices in WORLD space
std::vector<Vertex> vertices_world
{
    {{.position:{_x:0.f, _y:2.f, _z:0.f}},
    {{.position:{_x:1.f, _y:.0f, _z:.0f}},
    {{.position:{_x:-1.f, _y:0.f, _z:.0f}},
};
```

Rasterization: (3) Barycentric Coordinates

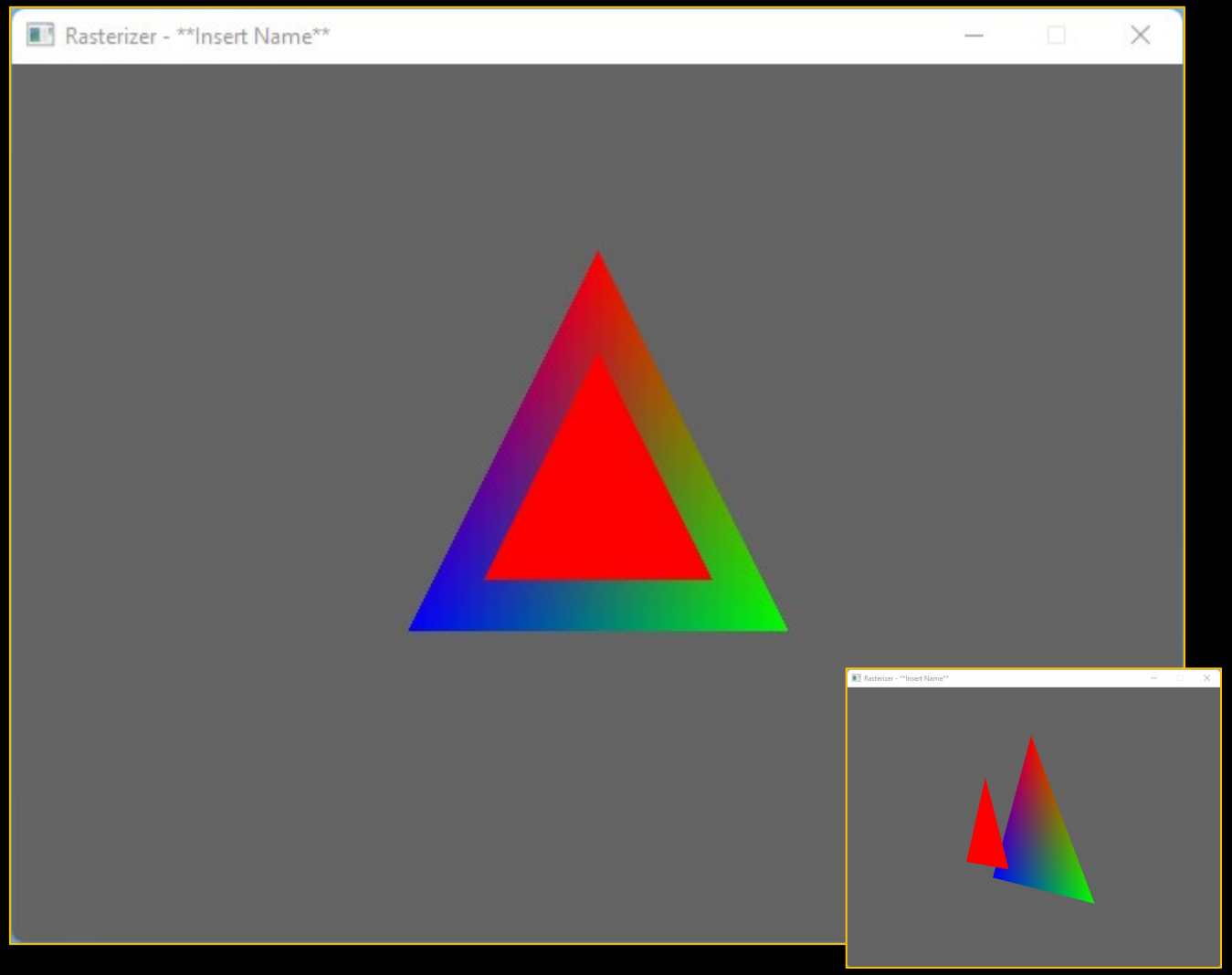


- **Objective**
  Extend the Rasterization Stage with Barycentric Coordinates. Interpolate the color for each pixel.

- See slides for implementation

```
//Define Triangle - Vertices in WORLD space
std::vector<Vertex> vertices_world
{
    {{.position={_x:0.f, _y:4.f, _z:2.f}, .color={.r:1, .g:0, .b:0}},
    {{.position={_x:3.f, _y:-2.f, _z:2.f}, .color={.r:0, .g:1, .b:0}},
    {{.position={_x:-3.f, _y:-2.f, _z:2.f}, .color={.r:0, .g:0, .b:1}}
};
```

Rasterization: (4) Depth Buffer

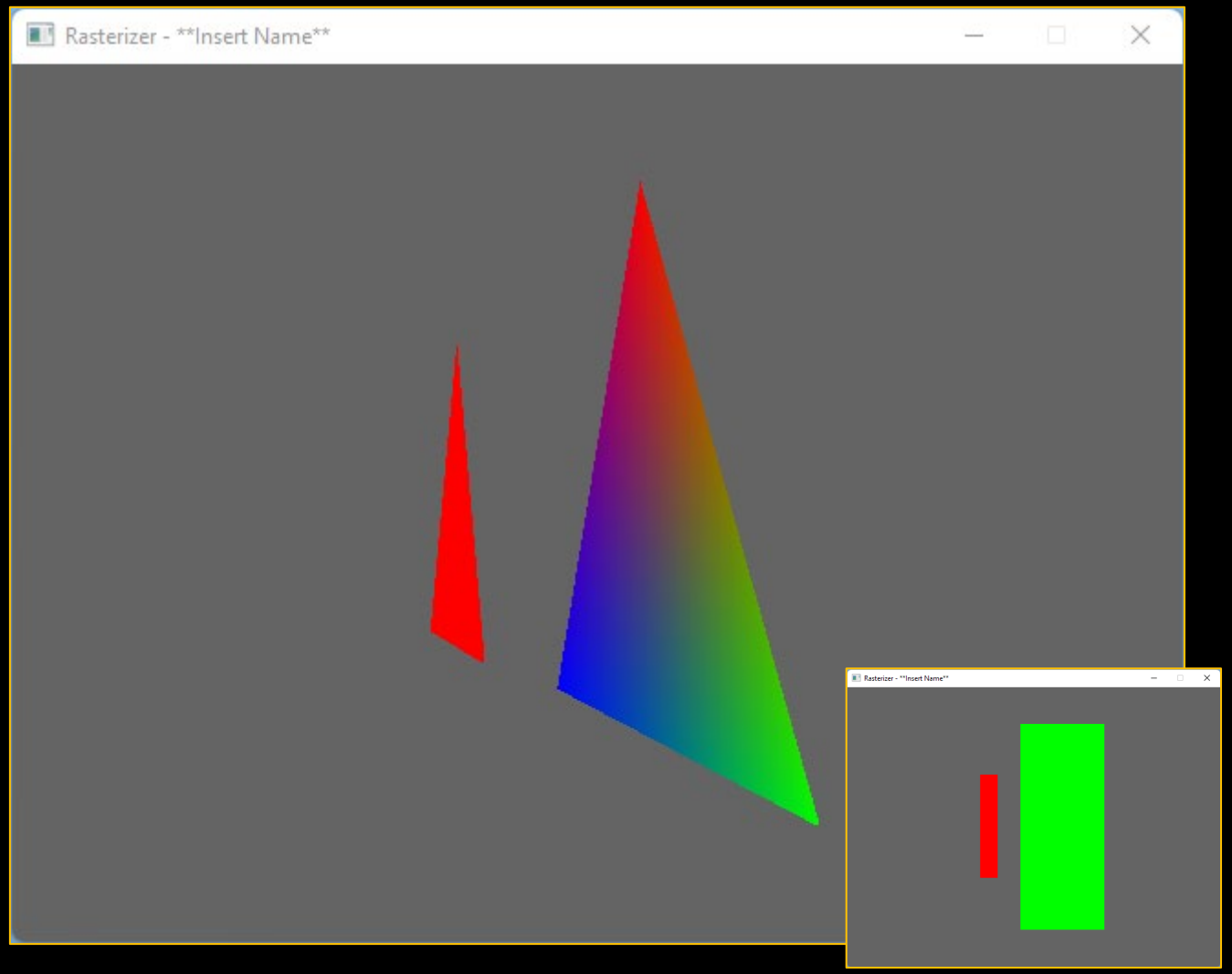


- **Objective**
  Implement a Depth Buffer. Perform a Depth Test to check whether a pixel is visible or covered by a previous rendered primitive. (This is part of the Rasterization Stage) + Adjust the render loop to support multiple triangles (each set of 3 vertices defines a separate triangle)

- Create a float array, initialize each value with the maximum value of a float. You need to do this every frame before rendering! (std::fill_n)
- Also clear the BackBuffer (SDL_FillRect, clearColor = {100,100,100}
- You'll have to interpolate the depth value, and perform a depth test for that pixel.
- Do not forget to also update the Depth Buffer if the Depth Test succeeds!
- See slides for implementation

```
//Define Triangle - Vertices in WORLD space
std::vector<Vertex> vertices_world
{
    //Triangle 0
    {{.position: {{_x: 0.f, _y: 2.f, _z: 0.f}, .color: {{.r: 1, .g: 0, .b: 0}},
    {{.position: {{_x: 1.5f, _y: -1.f, _z: 0.f}, .color: {{.r: 1, .g: 0, .b: 0}},
    {{.position: {{_x: -1.5f, _y: -1.f, _z: 0.f}, .color: {{.r: 1, .g: 0, .b: 0}},

    //Triangle 1
    {{.position: {{_x: 0.f, _y: 4.f, _z: 2.f}, .color: {{.r: 1, .g: 0, .b: 0}},
    {{.position: {{_x: 3.f, _y: -2.f, _z: 2.f}, .color: {{.r: 0, .g: 1, .b: 0}},
    {{.position: {{_x: -3.f, _y: -2.f, _z: 2.f}, .color: {{.r: 0, .g: 0, .b: 1}}
};
```

Rasterization: (5) BoundingBox Optimization



- **Objective**
  Once you have your pixels in Screen Space (pixel coordinates), define a closest fitting boundingbox. Instead of iterating all the pixels of the screen, only iterate over the pixels defined by the boundingbox

- Depending on how much screenspace each triangle covers, you should see a difference in performance with/without using a boundingbox

```
//Define Triangle - Vertices in WORLD space
std::vector<Vertex> vertices_world
{
    //Triangle 0
    {{.position: {_x: 0.f, _y: 2.f, _z: 0.f}, .color: {.r: 1, .g: 0, .b: 0}},
    {{.position: {_x: 1.5f, _y: -1.f, _z: 0.f}, .color: {.r: 1, .g: 0, .b: 0}},
    {{.position: {_x: -1.5f, _y: -1.f, _z: 0.f}, .color: {.r: 1, .g: 0, .b: 0}},

    //Triangle 1
    {{.position: {_x: 0.f, _y: 4.f, _z: 2.f}, .color: {.r: 1, .g: 0, .b: 0}},
    {{.position: {_x: 3.f, _y: -2.f, _z: 2.f}, .color: {.r: 0, .g: 1, .b: 0}},
    {{.position: {_x: -3.f, _y: -2.f, _z: 2.f}, .color: {.r: 0, .g: 0, .b: 1}}
};
```

GOOD LUCK!