GRAPHICS PROGRAMMING I

SOFTWARE RASTERIZATION PART III

Rasterization: Projection Stage

- The previous weeks you've implemented a basic projection stage and rasterization stage. The projection stage required a few calculations:
  - Vertex from Model to World Space (if not defined in world space, which it was in our demo).
  - Vertex from World Space to View Space (using the inverse of our CameraToWorld (ONB/World), which makes a WorldToCamera (View) matrix).
  - Mapping our x and y coordinates to the camera settings (aspect ratio and field of view).
  - Doing the perspective divide on both the x and y component to put our vertex in Projection Space
    (x and y in [-1,1] range).
- Finally, we moved our vertex from NDC to Raster Space. This isn't part of the projection stage, but rather the rasterization stage (see culling soon).
- All the calculations defined above require a few lines of code. What if I tell you these can be combined! How?
  - Matrices! ☺

Rasterization: Projection Matrix

- Let’s see how we can combine these calculations.
- Moving a model from Model Space to World Space (putting it somewhere in the world) can be defined using a WorldMatrix. You’ve done this before in your Ray Tracer (rotating triangles)!
- Going from World Space to View Space is already defined by a matrix, the inverse of the camera ONB, called the ViewMatrix.
- So how can we combine the calculations of the projection in a matrix, called the ProjectionMatrix.

Rasterization: Projection Matrix

- Let’s start with step one.
  - We need to scale the x and y coordinates of our vertex with the camera settings.

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

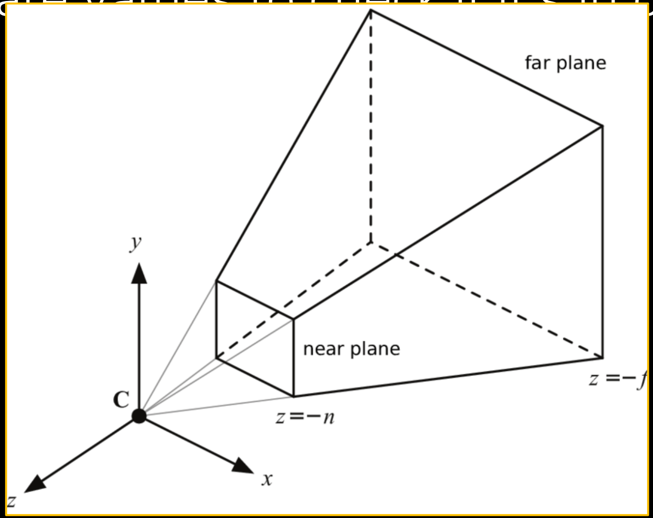
$$\boldsymbol{ProjectedVertex_y} = \frac{\boldsymbol{ProjectedVertex_y}}{\boldsymbol{FOV}}$$

  - Putting this in a matrix isn’t hard:

$$\begin{bmatrix} v_x & v_y & v_z \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} v_x & v_y & v_z \end{bmatrix} \begin{bmatrix} \frac{\mathbf{1}}{\boldsymbol{AspectRatio * FOV}} & 0 & 0 \\ 0 & \frac{\mathbf{1}}{\boldsymbol{FOV}} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rasterization: Projection Matrix

- You probably noticed already but there are still some problems when our camera gets to close to an object (crash). The reason for this is that we do not check if our vertices are inside our frustum.
- We can easily check if our x and y coordinates are inside that frustum.
  - Both are mapped into the [-1,1] range after the perspective divide. If a value is < -1.f or > 1.f it's outside the frustum, so we ignore it. In other words, we cull it, hence Frustum Culling.
- But what about the z coordinate? As you recall, the z value is a depth value defined in ViewSpace. What is the range? How can we compare values to check if it's in our frustum?
- Well, we are going to normalize the z value as well. We just must define to what range.
- In the camera we define a near plane and a far plane.
- Everything closer than the near plane or further than the far plane will be culled as well



researchgate.net – Stefan Diewald

Rasterization: Projection Matrix

- We must pick a range for the z coordinate. As usual there is no standard. For example:
  - DirectX = [0, 1]
  - OpenGL = [-1, 1]
- Because we are already following DirectX standards, we are going to map our z coordinate to the [0, 1] range.
- So how are we going to put this in our matrix?

$$\begin{bmatrix} v_x & v_y & v_z & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{AspectRatio * FOV} & 0 & 0 & 0 \\ 0 & \frac{1}{FOV} & 0 & 0 \\ 0 & 0 & A & 0 \\ 0 & 0 & B & 0 \end{bmatrix} = \begin{bmatrix} \frac{v_x}{AspectRatio * FOV} & \frac{v_y}{FOV} & Av_z + B & 0 \end{bmatrix}$$

- We know that we are going to do a perspective divide afterwards, so the z will be: $\boldsymbol{A + \frac{B}{v_z}}$
- $\boldsymbol{A}$ and $\boldsymbol{B}$ are unknown, while our near and far plane are known. We also know our range so:
  - If $\boldsymbol{v_z}$ is on the near plane: $\boldsymbol{A + \frac{B}{near} = 0}$ and if $\boldsymbol{v_z}$ is on the far plane: $\boldsymbol{A + \frac{B}{far} = 1}$

Rasterization: Projection Matrix

- Let’s solve the two unknown $\boldsymbol{A}$ and $\boldsymbol{B}$, knowing:
  - $A + \frac{B}{far} = 1$ and $A + \frac{B}{near} = 0$
  - Let’s rewrite the second formula to find $\boldsymbol{B}$: $A + \frac{B}{near} = 0 \longrightarrow B = -A * \text{near}$
  - Let’s find a solution for $\boldsymbol{A}$ by substituting $\boldsymbol{B}$ in our first formula:

$$A + \frac{B}{far} = 1 \longrightarrow A + \frac{(-A * near)}{far} = 1$$

  - Now let’s solve for $\boldsymbol{A}$:

$$A + \frac{(-A * near)}{far} = 1 \longrightarrow \frac{(A * far - A * near)}{far} = 1 \longrightarrow (A * far - A * near) = \text{far}$$

$$(A * far - A * near) = \text{far} \longrightarrow A\,(far - near) = \text{far} \longrightarrow A = \frac{far}{(far - near)}$$

  - Now we can substitute in $\boldsymbol{A}$ our $\boldsymbol{B}$ formula:

$$B = -A * \text{near} \longrightarrow B = \frac{-far}{(far - near)} * \text{near} \longrightarrow B = \frac{-(far * near)}{(far - near)}$$

Rasterization: Projection Matrix

- And there we have it. We solved $\boldsymbol{A}$ and $\boldsymbol{B}$. Now let's plug this into our matrix:

$$A = \frac{far}{(far - near)}$$

$$B = \frac{-(far * near)}{(far - near)}$$

$$\begin{bmatrix} \frac{1}{AspectRatio * FOV} & 0 & 0 & 0 \\ 0 & \frac{1}{FOV} & 0 & 0 \\ 0 & 0 & \frac{far}{(far - near)} & \mathbf{0} \\ 0 & 0 & \frac{-(far * near)}{(far - near)} & 0 \end{bmatrix}$$

- We have one problem now! Can you spot it?
  Hint: what happens after this step?

Rasterization: Projection Matrix

- We just lost our original z value, which we need for our perspective divide and for our vertex attribute interpolation!
- We solve this by storing our original z component in the w component of our 4D vector:

$$\begin{bmatrix} v_x & v_y & v_z & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{AspectRatio * FOV} & 0 & 0 & 0 \\ 0 & \frac{1}{FOV} & 0 & 0 \\ 0 & 0 & \frac{far}{(far - near)} & 1 \\ 0 & 0 & \frac{-(far * near)}{(far - near)} & 0 \end{bmatrix} = \begin{bmatrix} \frac{v_x}{AspectRatio * FOV} & \frac{v_y}{FOV} & \frac{far * vz}{(far - near)} + \frac{-(far * near)}{(far - near)} & \boldsymbol{v_z} \end{bmatrix}$$

Rasterization: Projection Matrix

- The implementation of a Projection Matrix also depends on the coordinate system you're using. (Left-Handed > +Z vs. Right-Handed > -Z)

$$\begin{bmatrix} \frac{1}{AspectRatio * FOV} & 0 & 0 & 0 \\ 0 & \frac{1}{FOV} & 0 & 0 \\ 0 & 0 & \frac{far}{(far - near)} & 1 \\ 0 & 0 & \frac{-(far * near)}{(far - near)} & 0 \end{bmatrix} \leftrightarrow \begin{bmatrix} \frac{1}{AspectRatio * FOV} & 0 & 0 & 0 \\ 0 & \frac{1}{FOV} & 0 & 0 \\ 0 & 0 & \frac{\mathbf{far}}{(\mathbf{near} - \mathbf{far})} & -1 \\ 0 & 0 & \frac{(\mathbf{far} * \mathbf{near})}{(\mathbf{near} - \mathbf{far})} & 0 \end{bmatrix}$$

Left-Handed Coordinate System

Right-Handed Coordinate System

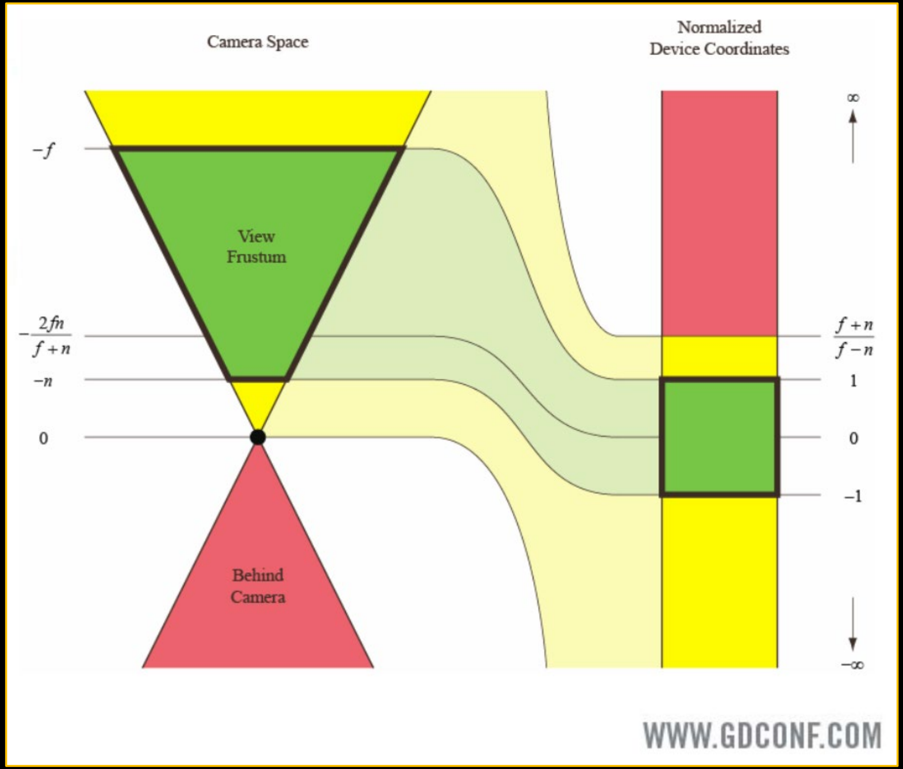
- We are of course using the left-handed version

Rasterization: Projection Matrix

- We can now combine all our space transformation into one matrix:
  - WorldViewProjectionMatrix = WorldMatrix * ViewMatrix * ProjectionMatrix
- So, in our projection stage, we can multiply every vertex with this matrix, which is the **same** for all vertices **within one mesh**!
- After that, we must perform perspective divide to put them in NDC. Watch out though! Now use the **w component** of your vector (which has the correct depth value in ViewSpace):
  - $v_x \mathrel{/}= v_w$
  - $v_y \mathrel{/}= v_w$
  - $v_z \mathrel{/}= v_w$
  - $v_w = v_w$ → Warning: some rasterizers store $\frac{1}{v_w}$ because you need this value for attribute interpolation.
- Now our coordinates are defined in:
  - $v_x = [-1, 1]$
  - $v_y = [-1, 1]$
  - $v_z = [0, 1]$
  - $v_w = v_z$ in ViewSpace

Rasterization: Projection Matrix

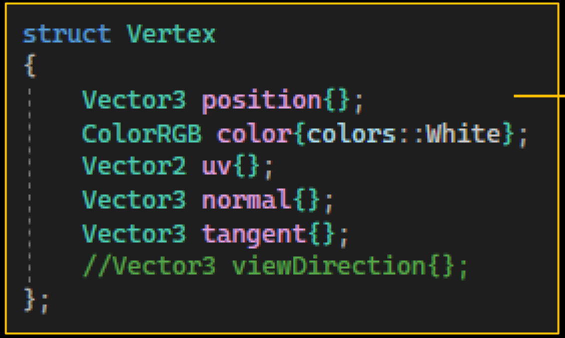
- But why don't we divide (x,y,z) by **z in ProjectionSpace** instead of **z in ViewSpace**?
- Turns out, if we put our z component in the [near, far] range, the value is no longer linear! Dividing by this value would result in incorrect values. So don't forget to change your perspective divide to use the "original" **z in ViewSpace** (our w component).



Rasterization: Projection Matrix

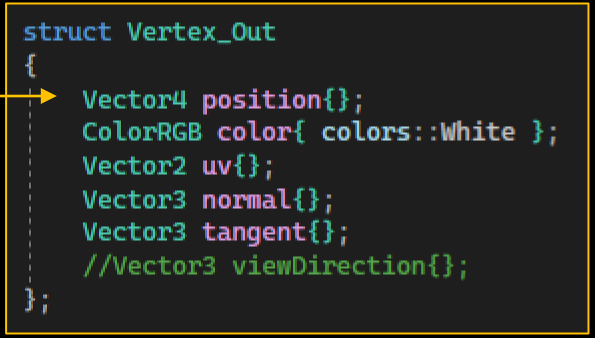
- For what other reasons do we need to keep the w component?
  - If we want to correctly interpolate our vertex attributes, we want to do this by a value that is linear. Remember: $\frac{1}{z}$, well this becomes: $\frac{1}{w}$
  - You cannot interpolate with the new z value, because it is not linear. Even not when you would try to do: $\frac{1}{z}$
- So, to sum this up, after using the matrix and the perspective divide:
  - z = not linear → [0, 1] range
  - w = not linear → perspective distortion: projection does not preserve distances!
  - $\frac{1}{z}$ = not linear → even countering perspective distortion, we still have a non-linear value! ☺
  - $\frac{1}{w}$ = linear!
- Again, this means, interpolating vertex attributes in your rasterization stage now happens with: $\frac{1}{w}$
- This also means, after the projection stage and the perspective divide, the position of our vertices are no longer stored in a Vector3, but a Vector4!

Rasterization: Projection Matrix

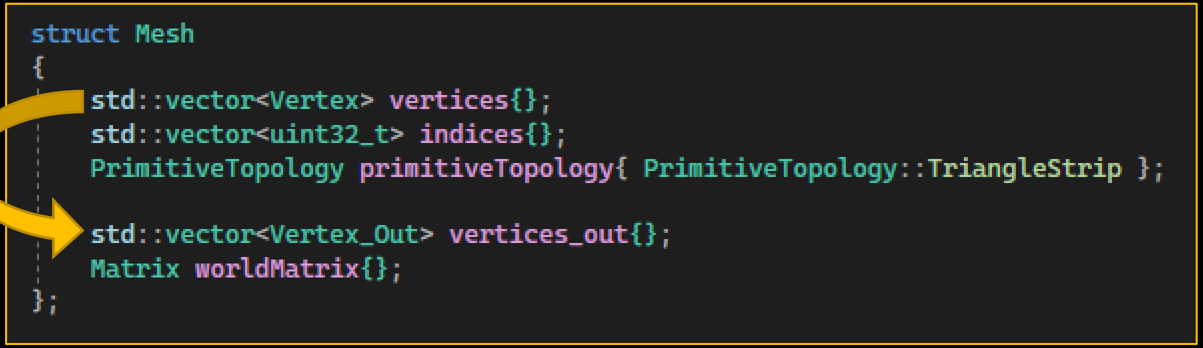


```cpp
struct Vertex
{
    Vector3 position{};
    ColorRGB color{colors::White};
    Vector2 uv{};
    Vector3 normal{};
    Vector3 tangent{};
    //Vector3 viewDirection{};
};
```

After Projection Stage



```cpp
struct Vertex_Out
{
    Vector4 position{};
    ColorRGB color{ colors::White };
    Vector2 uv{};
    Vector3 normal{};
    Vector3 tangent{};
    //Vector3 viewDirection{};
};
```



```cpp
struct Mesh
{
    std::vector<Vertex> vertices{};
    std::vector<uint32_t> indices{};
    PrimitiveTopology primitiveTopology{ PrimitiveTopology::TriangleStrip };

    std::vector<Vertex_Out> vertices_out{};
    Matrix worldMatrix{};
};
```

Rasterization: Depth Buffer

- Didn't we store the v_z in ViewSpace in our Depth Buffer last week?
- Yes, and now the same value is stored in v_w. If we would store this value, we would be storing the w value (which is the z in ViewSpace). We would be creating a w-buffer and not a z-buffer.
- In hardware accelerated rasterizers, the depth buffer is a z-buffer. I also want you to store the depth as a z-buffer. Why?

- Frustum clipping! We can check if a vertex is inside the frustum.
  So, our Depth Test becomes:
  - Is our interpolated depth value inside [0,1] range?
  - If so, is this value closer than the one stored in our depth buffer?
  - If so, this pixel is closer → color pixel and store the depth in the depth buffer.

Rasterization: Depth Buffer

- WARNING! For the Depth Buffer, we are not using those values to interpolate vertex attributes. We are only interested in comparing depth values. If those values are not linear, that is fine, if both values are non-linear with the same function!

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

(= Interpolated Depth)
Non-Linear!

- This $Z_{\text{BufferValue}}$ is the one we compare in the Depth Test and the value we store in the Depth Buffer (uses V_z).
- When we want to interpolate vertex attributes with a correct depth (color, uv, normals, etc.), we still use the View Space depth (V_w):

$$W_{\text{Interpolated}} = \frac{1}{\frac{1}{V0_w}w0 + \frac{1}{V1_w}w1 + \frac{1}{V2_w}w2}$$

(= Interpolated Depth)
Linear!

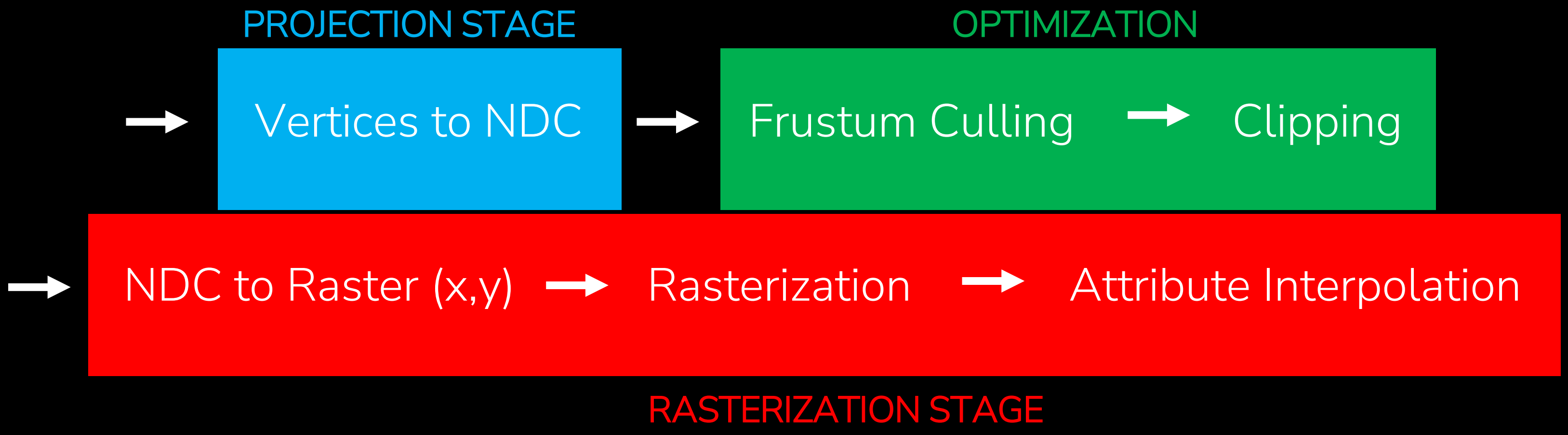
$$UV_{\text{Interpolated}} = \left(\frac{V0_{uv}}{V0_w}w0 + \frac{V1_{uv}}{V1_w}w1 + \frac{V2_{uv}}{V2_w}w2\right) w_{\text{Interpolated}}$$

(= Interpolated UV)
Same applies to other vertex attributes!

Rasterization: Depth Buffer

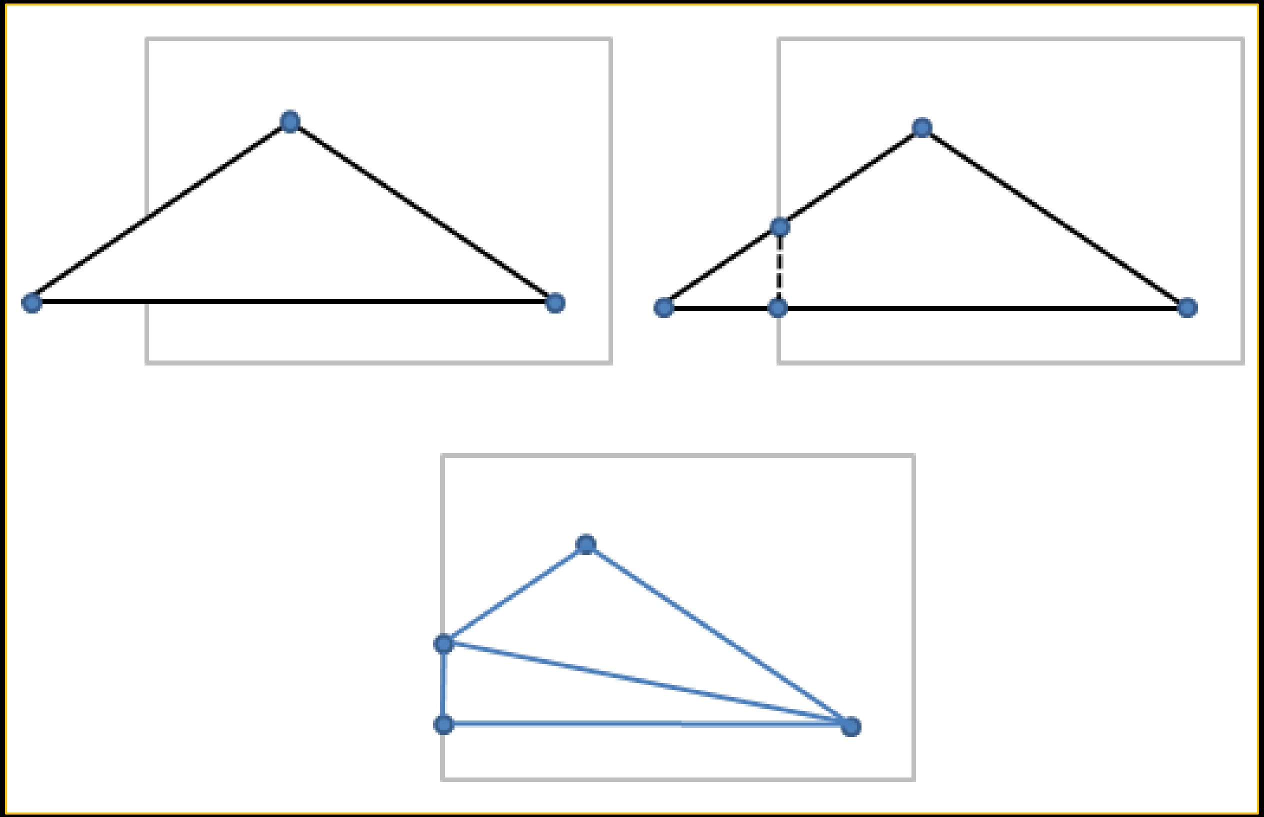
- What changes this week:
  - Use a (World)ViewProjectionMatrix for transforming our vertices.
  - Store the result in a Vector4 instead of a Vector3.
  - Use $\boldsymbol{V_z}$ to interpolate the depth and use the result for our DepthTest and DepthBuffer.
  - Use $\boldsymbol{V_w}$ to interpolate all other vertex attribute using correct depth interpolation.
  - Now that we have our x,y and z in NDC, perform **culling**.

The flow of your program should now be:



Rasterization: Clipping?

- I don’t expect you to do this but be my guest to try it! ☺
- Right now, just don’t render the triangle if the value is out of the frustum range.

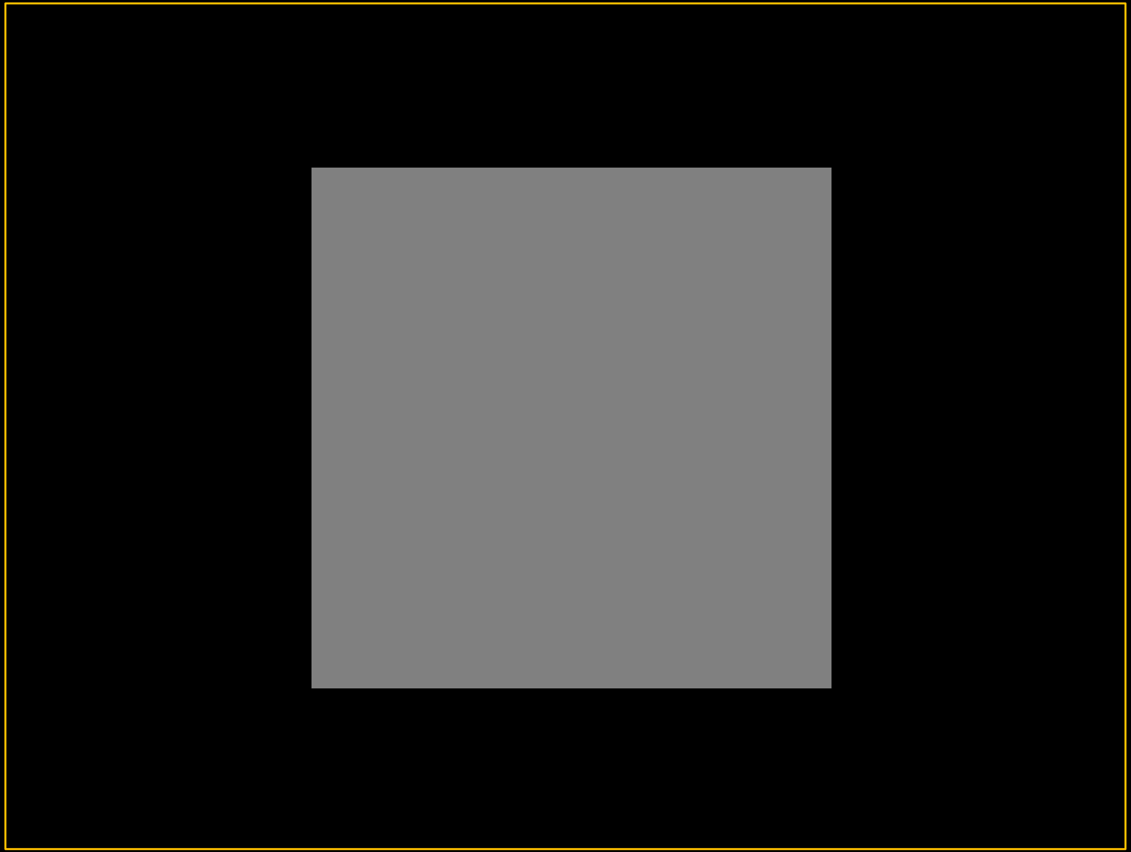


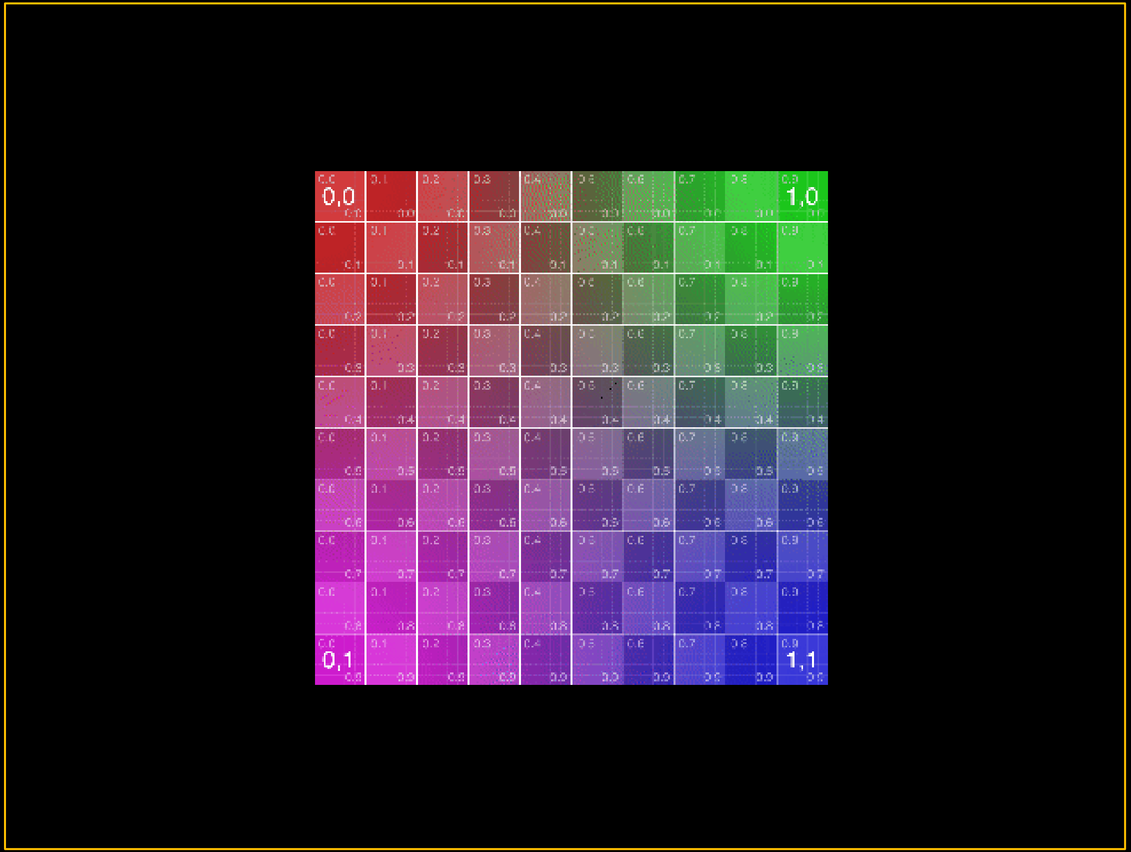
Rasterization: What to do?

- So, what should you do now:
  - Add a near and far plane variable to your camera. (default: near = .1f, far = 100.f)
  - Use the (World)ViewProjectionMatrix
  - Use the correct depth buffer for the depth test (z-buffer, not w-buffer).
  - Add frustum culling.
- If everything works, you should get the exact same result, but you'll finally have a "correct" depth buffer! ☺
- Toggle between FinalColor and DepthBuffer with key 'F4'. In other words, make sure you can visualize your depth buffer!

  Hint: values will probably be close to 1, showing a lot of white.
  Remap the values before rendering using: Remap(depthValue, 0.985f, 1.f)

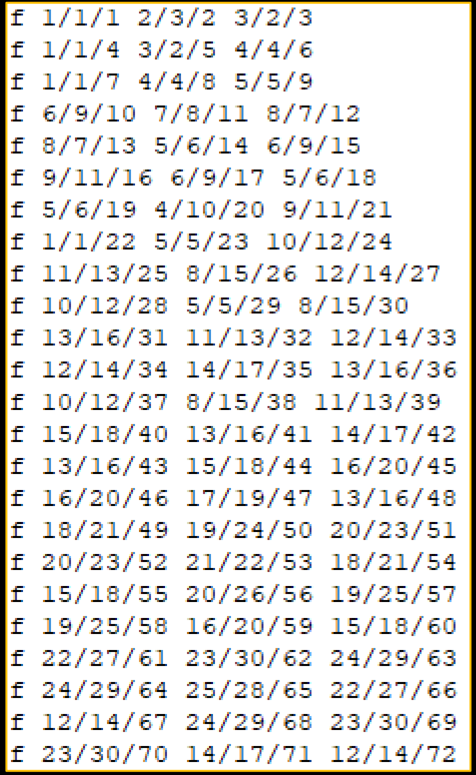
Rasterization: What to do?





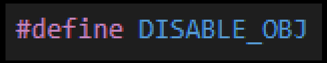
Rasterization: Meshes

- Finally, I want you to render an 3D Mesh again.
- With the ray tracer we just read in the vertex positions and calculate the normal of the primitive (triangle) ourselves. We didn't have an UV coordinate.
- With our rasterizer we don't do this. We just read in per vertex attributes and interpolate. This means you will have to update your OBJParser to read in:
  - UV coordinates: prefix vt
  - Normals: prefix vn



```
f 1/1/1 2/3/2 3/2/3
f 1/1/4 3/2/5 4/4/6
f 1/1/7 4/4/8 5/5/9
f 6/9/10 7/8/11 8/7/12
f 8/7/13 5/6/14 6/9/15
f 9/11/16 6/9/17 5/6/18
f 5/6/19 4/10/20 9/11/21
f 1/1/22 5/5/23 10/12/24
f 11/13/25 8/15/26 12/14/27
f 10/12/28 5/5/29 8/15/30
f 13/16/31 11/13/32 12/14/33
f 12/14/34 14/17/35 13/16/36
f 10/12/37 8/15/38 11/13/39
f 15/18/40 13/16/41 14/17/42
f 13/16/43 15/18/44 16/20/45
f 16/20/46 17/19/47 13/16/48
f 18/21/49 19/24/50 20/23/51
f 20/23/52 21/22/53 18/21/54
f 15/18/55 20/26/56 19/25/57
f 19/25/58 16/20/59 15/18/60
f 22/27/61 23/30/62 24/29/63
f 24/29/64 25/28/65 22/27/66
f 12/14/67 24/29/68 23/30/69
f 23/30/70 14/17/71 12/14/72
```

- Use Utils::ParseOBJ
  - Make sure to uncomment the required vertex attributes (DataTypes::Vertex) and comment out the 'DISABLE_OBJ' directive (Utils::ParseOBJ)

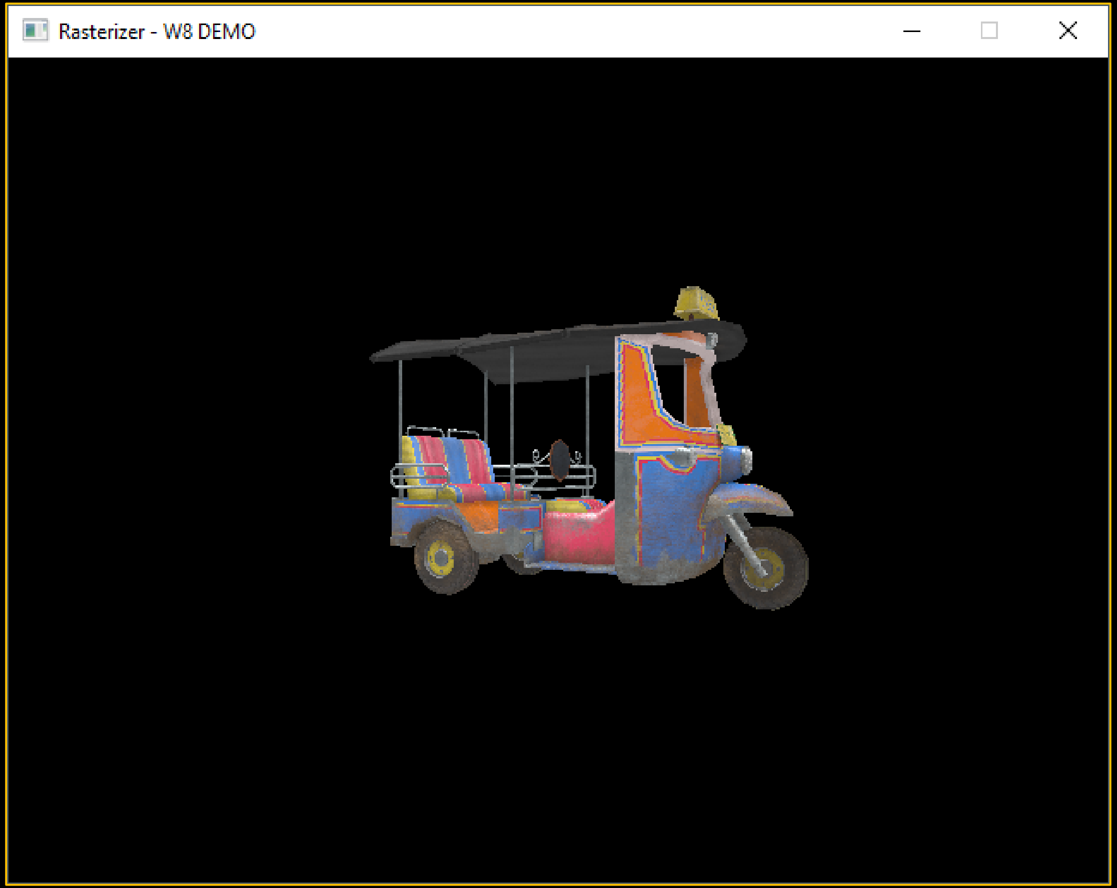


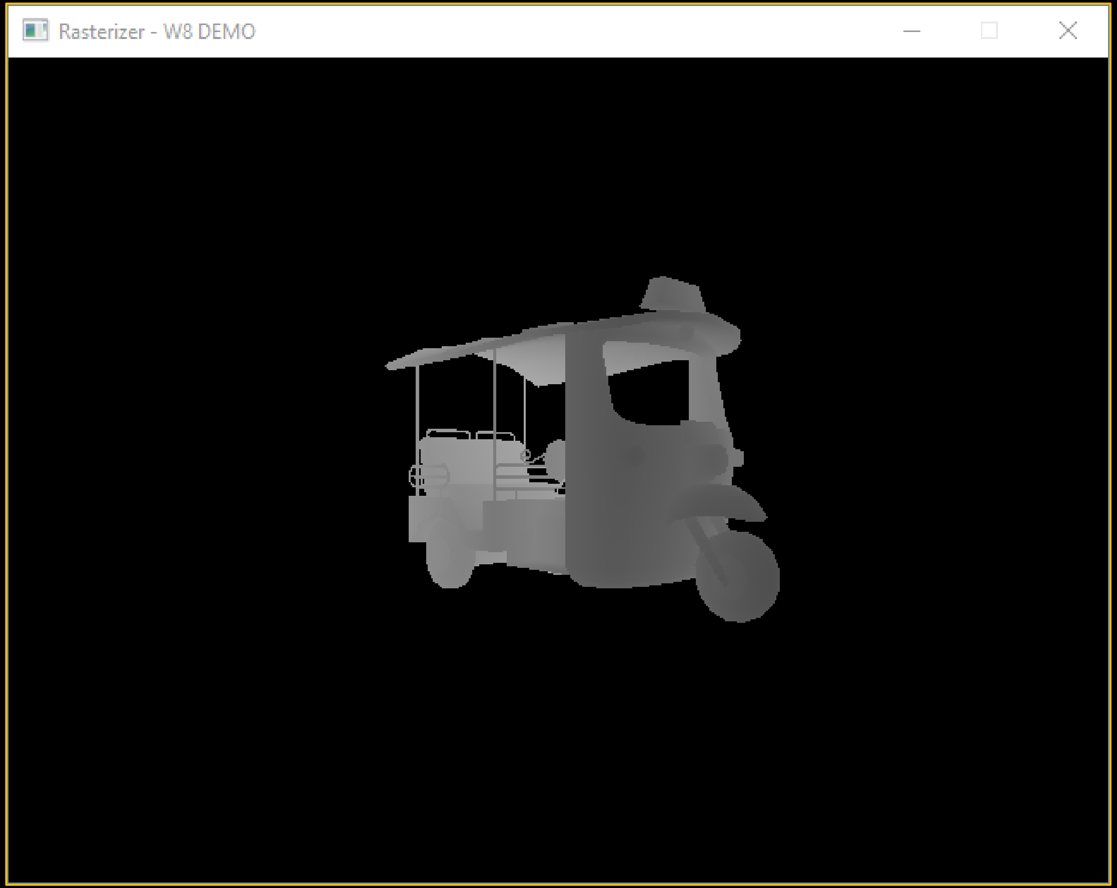
```
#define DISABLE_OBJ
```

  - This function will read out an OBJ file, and populate the vertex (position, uv, normal & tangent) and index array

- Also, let it rotate using a WorldMatrix! ☺

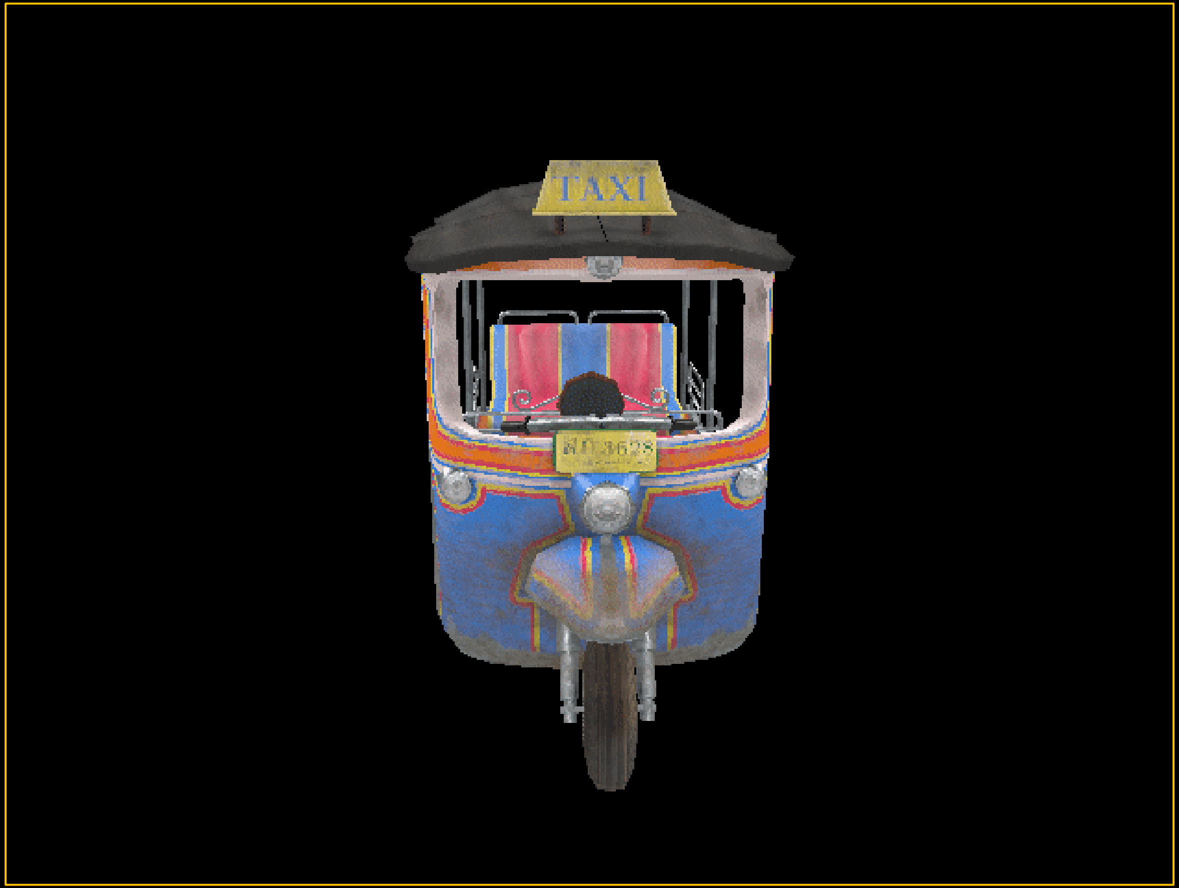
Rasterization: Meshes

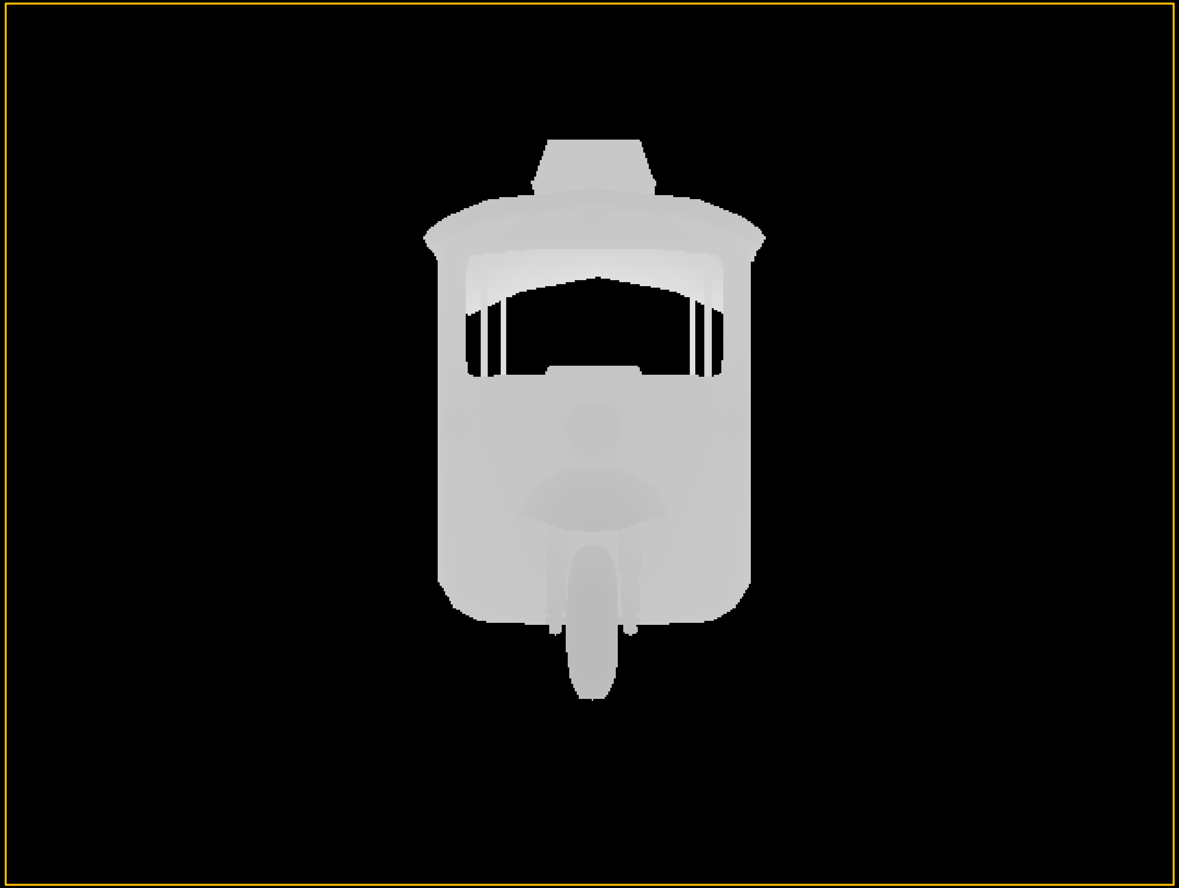




Camera Position: 0.f, 5.f, -30.f (60 FovAngle) || DepthRemap(0.995f – 1.f)

Rasterization: Meshes





GOOD LUCK!