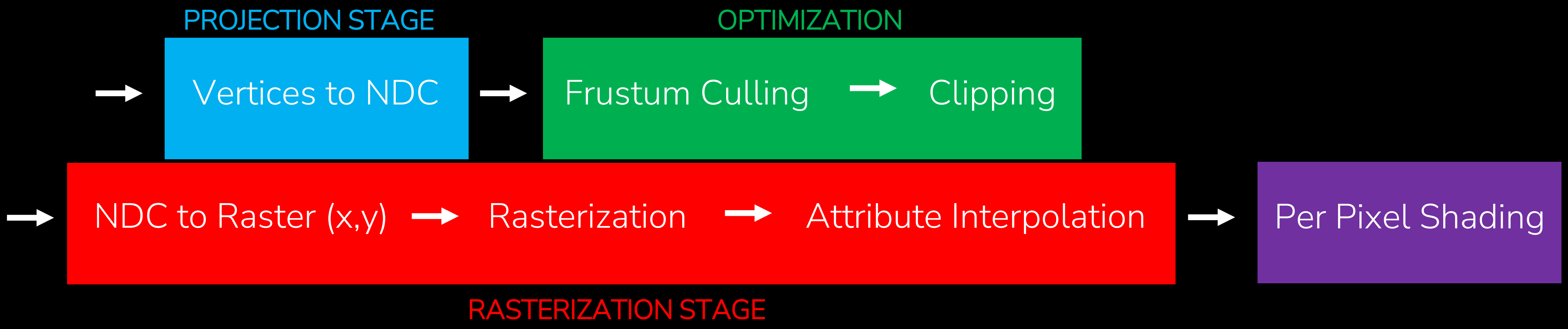
GRAPHICS PROGRAMMING I

SOFTWARE RASTERIZATION PART IV

Rasterization: Shading

- Last week you've implemented the "correct" depth buffer and did all the necessary transformations with the WorldViewProjection matrix.
- You've also sampled a texture map and used it to render a diffuse color on an 3D mesh.



- This week we'll dive deeper into the topic of shading!
- After the rasterization and the attribute interpolation we are left with data that "represents" a pixel.

Rasterization: Shading

- What do I mean with “data that “represents” a pixel”:
  - Vector4 position:
    - x = screen space (x-coordinate of pixel)
    - y = screen space (y-coordinate of pixel)
    - z = remapped depth in frustum (our w remapped in [0,1] range)
    - w = interpolated camera space z-coordinate.
  - Attributes:
    - All attributes have been determined for this single pixel. They got interpolated with correct depth interpolation.
    - Attributes used could be vertex color, vertex normal, vertex texture coordinates, etc.

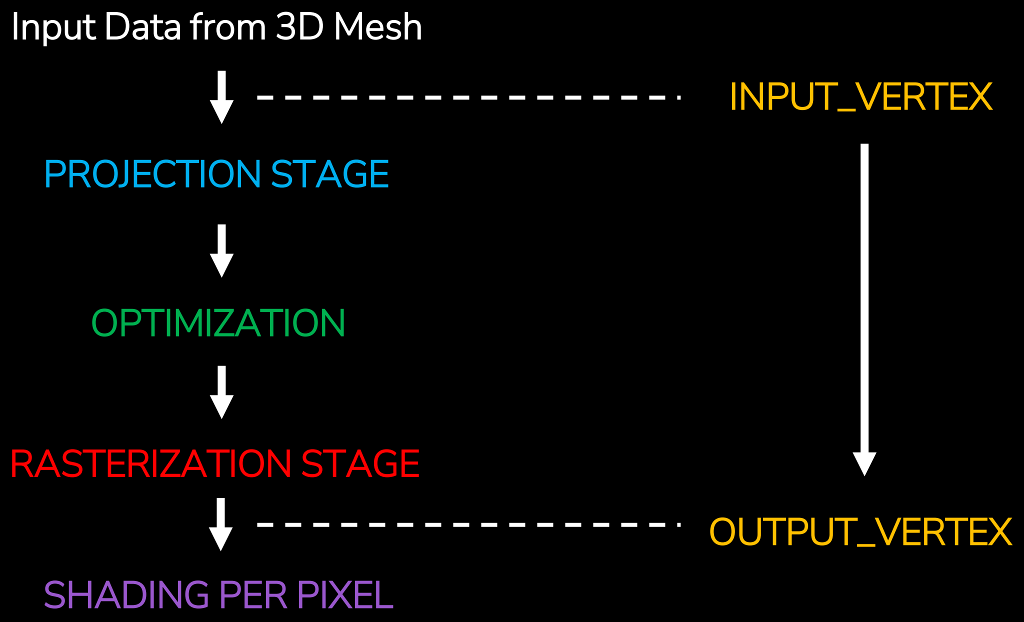
- So, how do we shade a pixel? In the ray tracer we did all our calculations in world space, not in screen space. So now what?
- Not all attributes are in screen space! They are in the coordinate system you defined them in. In the rasterizer we just interpolate them based on the depth values of the vertices!

Rasterization: Shading

- During the vertex transformation, so before the actual rasterization, we multiply our position with the WorldViewProjection matrix and do the perspective divide to put it in NDC.
- Normals in NDC on the other hand make no sense… We are interested in normals in world space when we do lighting calculations. This means, in the vertex transformation we multiply our normals with the World matrix, NOT the WorldViewProjection matrix.
- We also don't do a perspective divide on the normals. We only do it on the position. We only need the position to solve the visibility problem.
- You can store any attribute you want; in any space you want! After the rasterization our attributes will be interpolated for one single pixel and we can start doing some lighting calculations.
- Be careful though! Interpolating two normalized vectors does not guarantee a normalized result! Normalize vectors that need to be normalized after the interpolation.

Rasterization: Shading

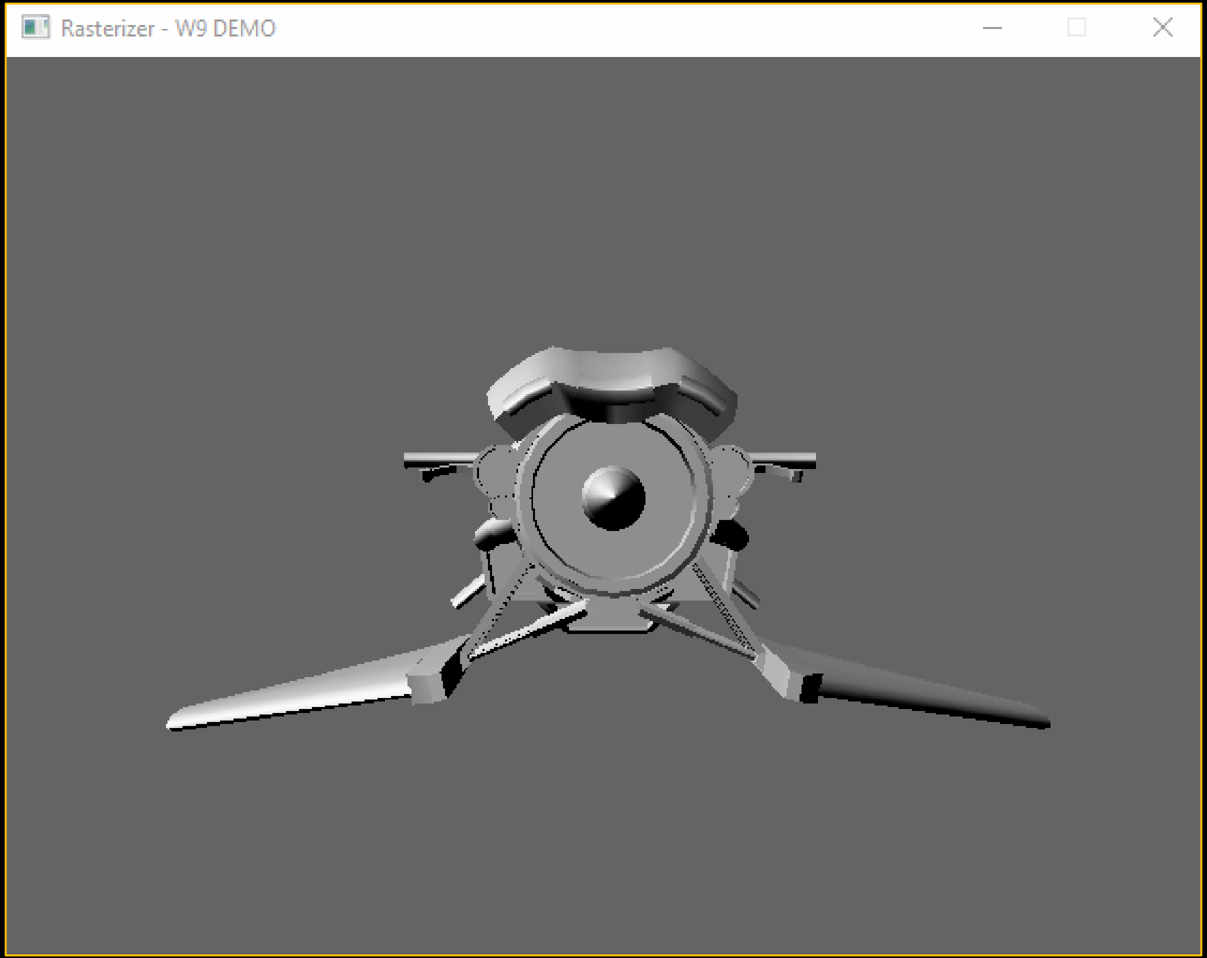
- Hopefully, you wrote your code in similar steps as defined in one of the previous slides.
- One important note, make sure you keep on **passing your vertex information during each stage**. If you don't, you just loose the information!



Rasterization: Shading

- Transform the normals from the vertices in the correct space (world space) in your transformation function.
- Create a separate function called PixelShading(const Vertex_Out& v) in your renderer. This function will be responsible for shading a single pixel. This function only gets triggered when the DepthTest succeeds (rasterization stage).
- Once we have the normals and the positions, we need one more thing before we can start doing some basic shading. What do we still need?
- Lights! ☺ You can implement point and directional lights if you want, but to focus on the actual shading we'll hardcode a directional light in our PixelShading function. Add the following local variable which represents a direction of a directional light (so no fall-off):
  - Vector3 lightDirection = { .577f, -.577f, .577f }
- Lighting in a rasterizer is almost exactly the same as in the ray tracer. The only difference is the way we handle input parameters.
- Add basic diffuse lambert + the observed area (lambert's cosine law) and output the color
  - Same logic as Lambert BFRD from the RayTracer

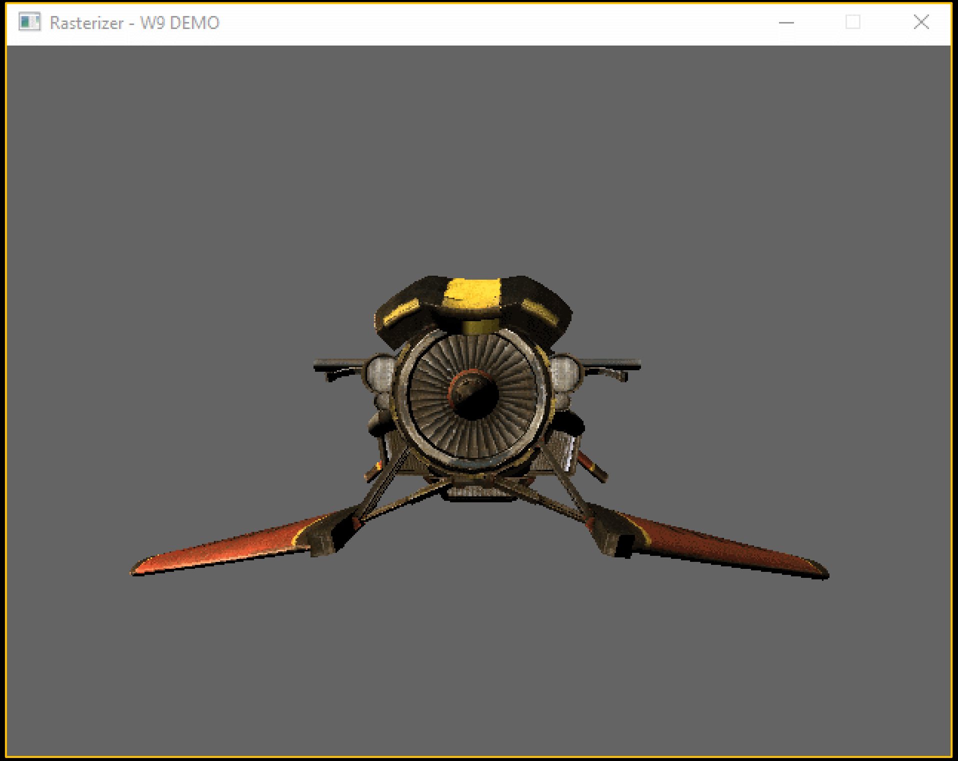
Rasterization: Shading



Observed Area

Rasterization: Shading

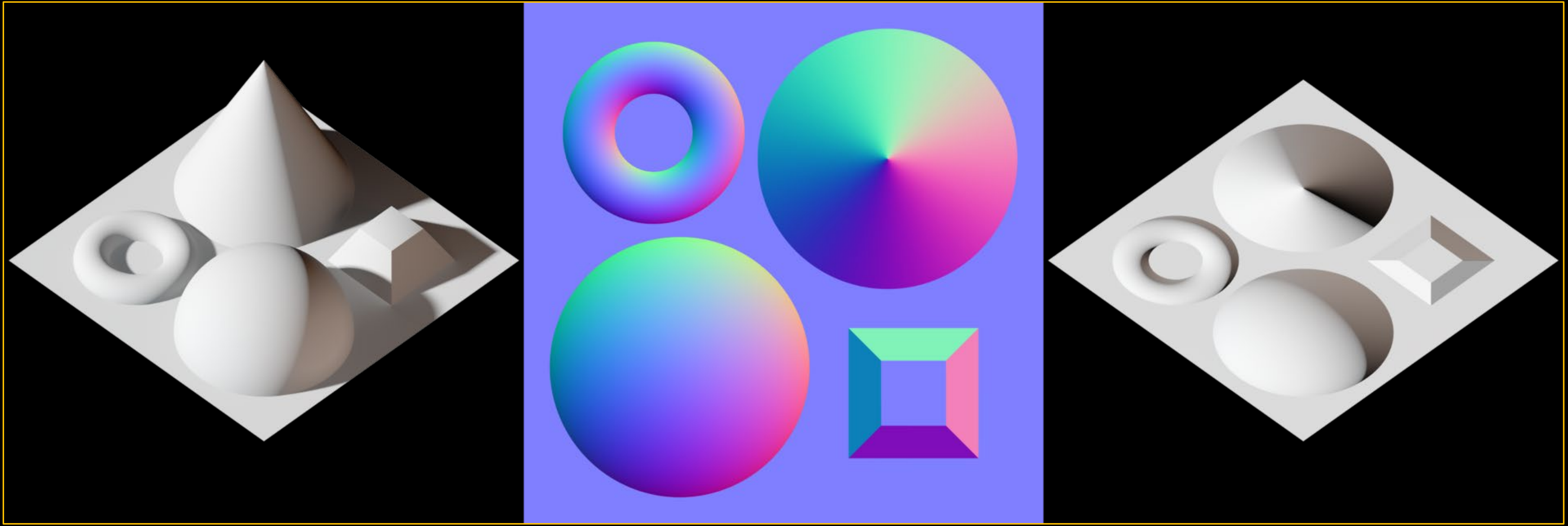
- Adding the diffuse texture color isn't hard.
  You already know how to sample from the diffuse texture map.
  So, use that color for the Lambert Diffuse color component



Diffuse (Light Intensity = 7.f)

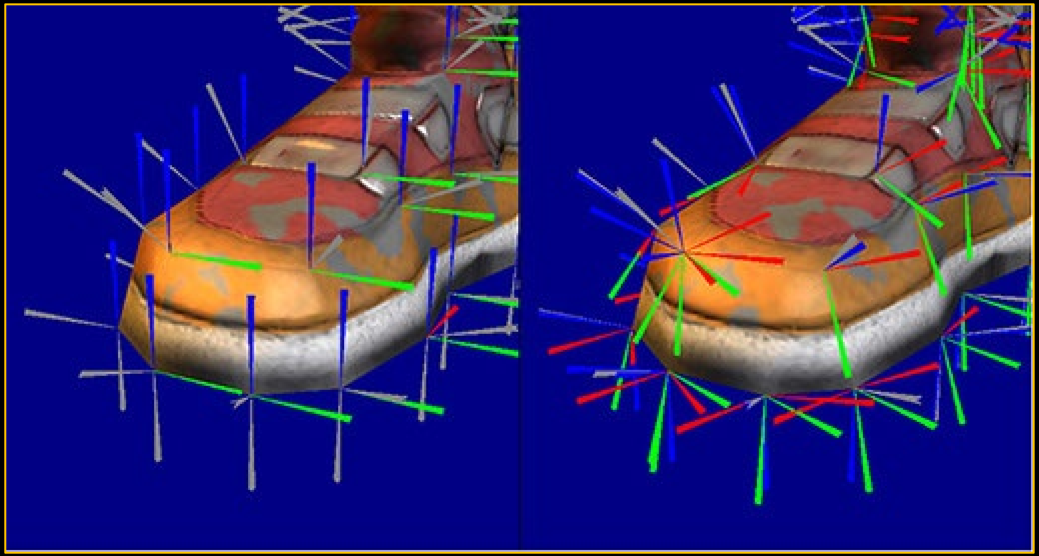
Rasterization: Normal Maps

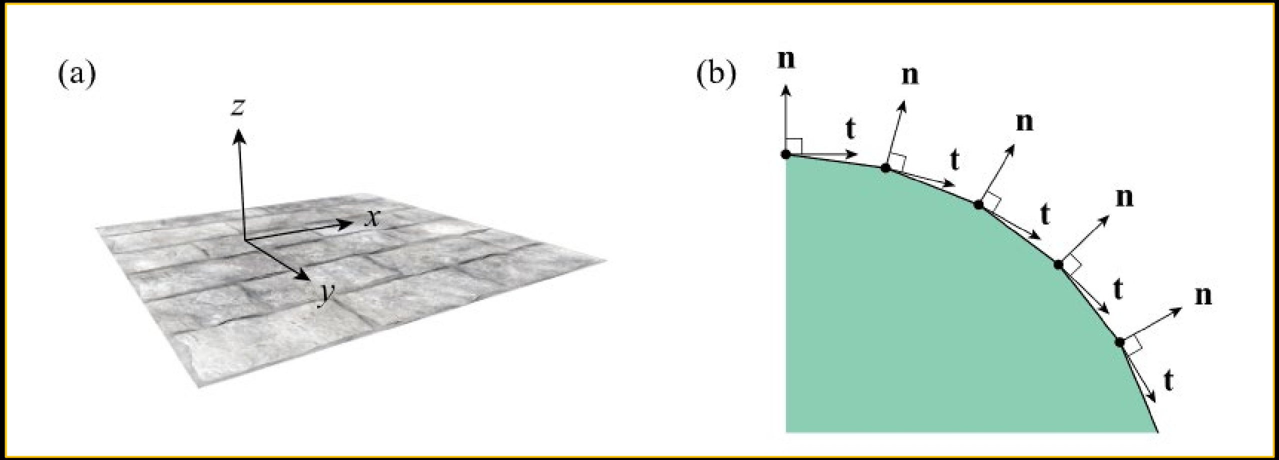
- Normal maps are used to fake more surface details on a low-poly mesh. Usually they are created by baking information onto a low-poly mesh, but there are also tools that allow you to directly “paint” information onto a mesh.
- Normal maps represent per pixel vectors! So, we don’t interpret them as colors.



Rasterization: Normal Maps

- Why do normal maps look purplish?
- They represent the surface normal in the **tangent space**. In tangent space the **z-component points upwards**. Most normals point upwards along the actual surface normal, thus it usually holds the largest values in the z-component.



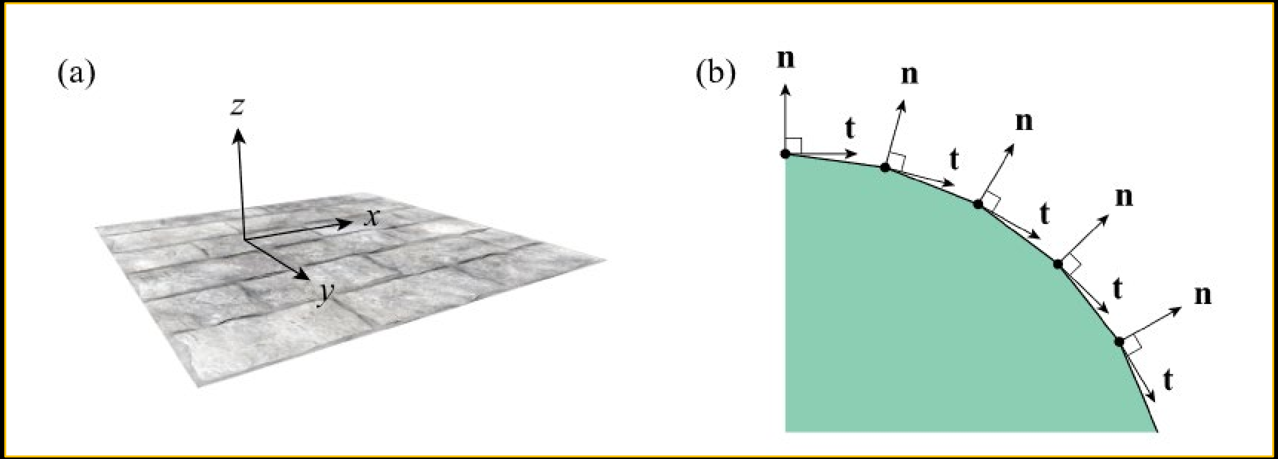


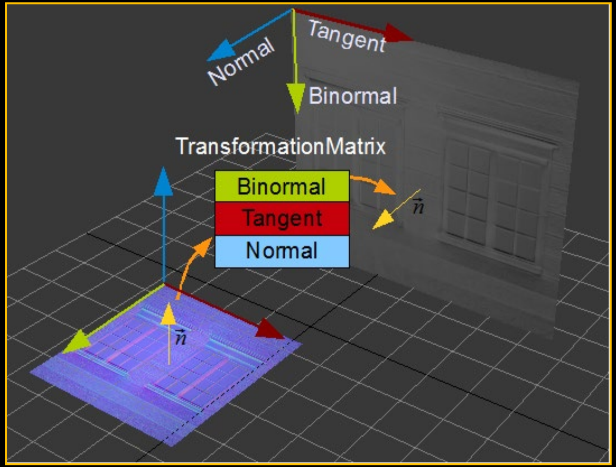
1.

1. Foundations of Game Engine Development Volume 2 – Eric Lengyel

Rasterization: Normal Maps

- This means, we will have to **transform** our sampled normal of the normal map into the tangent space of our current "triangle".





- Current triangle? But we are calculating the color at the pixel level. How can we do this?
- Just as the ONB we can create a **matrix** that makes us able to transform the sampled normal into the correct space. We need **two vectors** to create the axis:
  - Interpolated vertex **normal**
  - Interpolated vertex **tangent**

Rasterization: Normal Maps

- With these two vectors we can create the tangent space transformation matrix:
  - Vector3 binormal = Cross(normal, tangent)
  - Matrix tangentSpaceAxis = Matrix{ tangent, binormal, normal, zero }

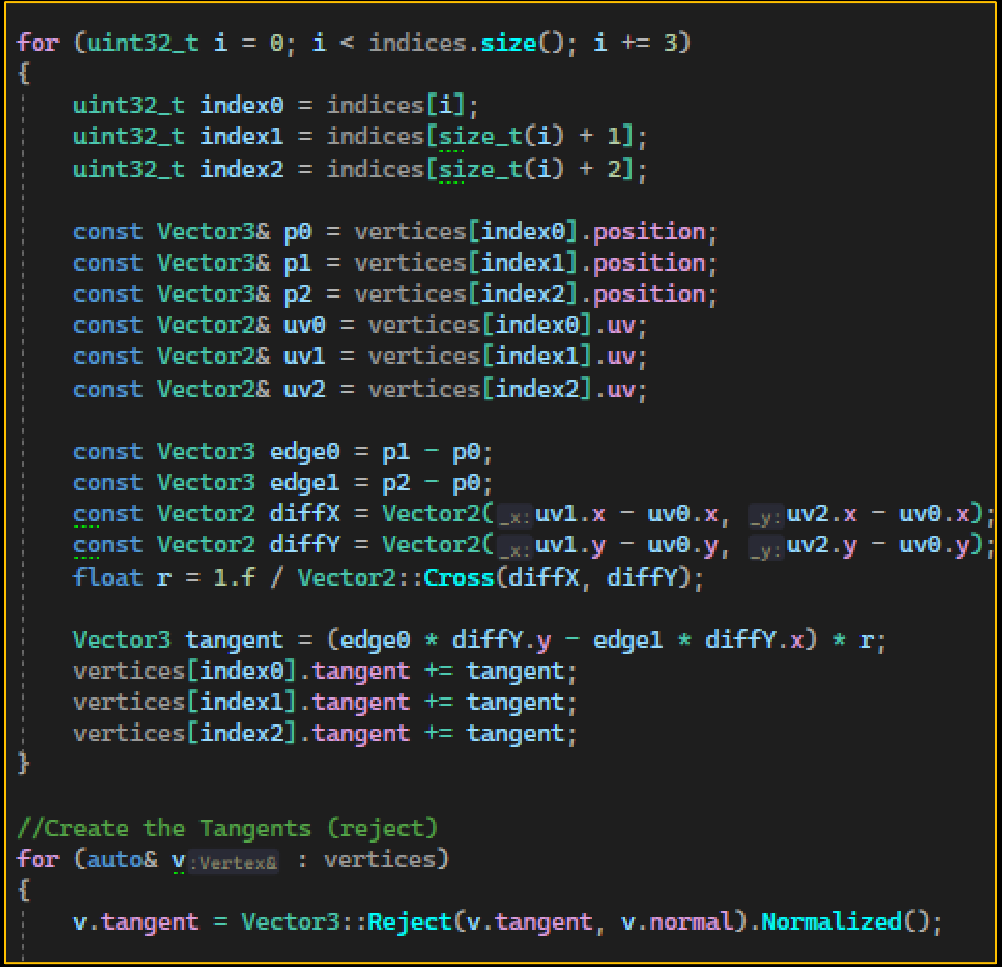
- We can now sample from our normal map texture and **multiple** that normal with this matrix to put it in the correct tangent space (where the normal and tangent of our vertex are defined in world space).
- There is one catch though… When we sample our normal from the normal map, the value of the vector is in range [0, 255]. As we all know, a normalized vector has the range [-1, 1]. So, after sampling, remap to the correct range:
  - sampledNormal /= 255.f → [0, 255] to [0, 1]
  - sampledNormal = 2.f * sampledNormal – 1.f → [0, 1] to [-1, 1]

- Instead of the original vertex normal you can now use the sampled normal for the diffuse lighting calculation!

Rasterization: Normal Maps

- OBJ does not support tangents (in contrast to other formats like FBX).

- Tangents are automatically calculated by the ParseOBJ function

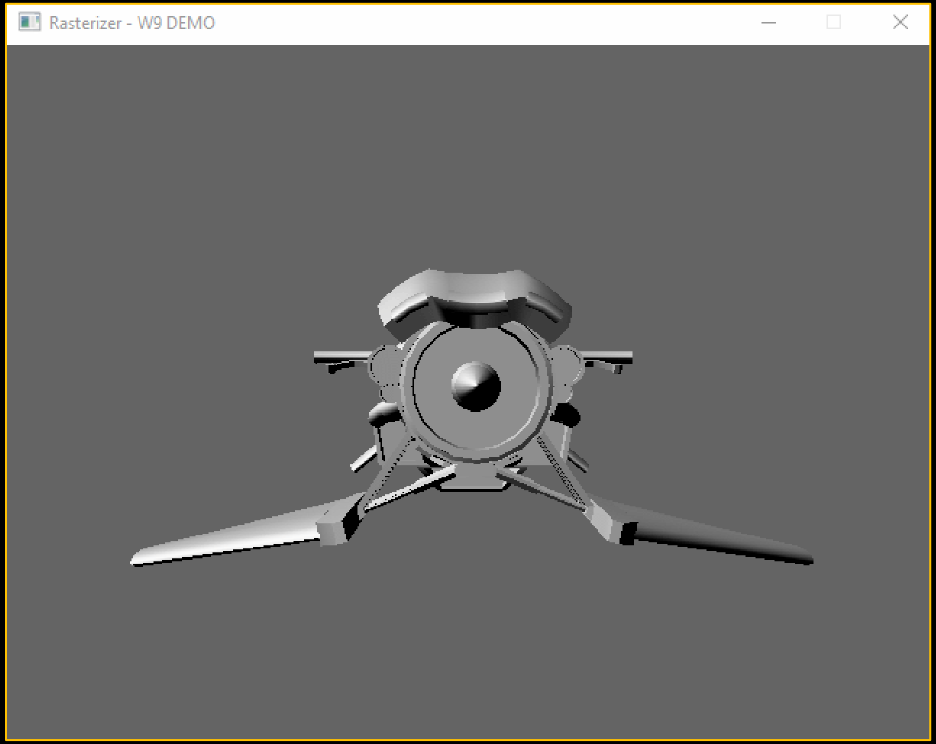


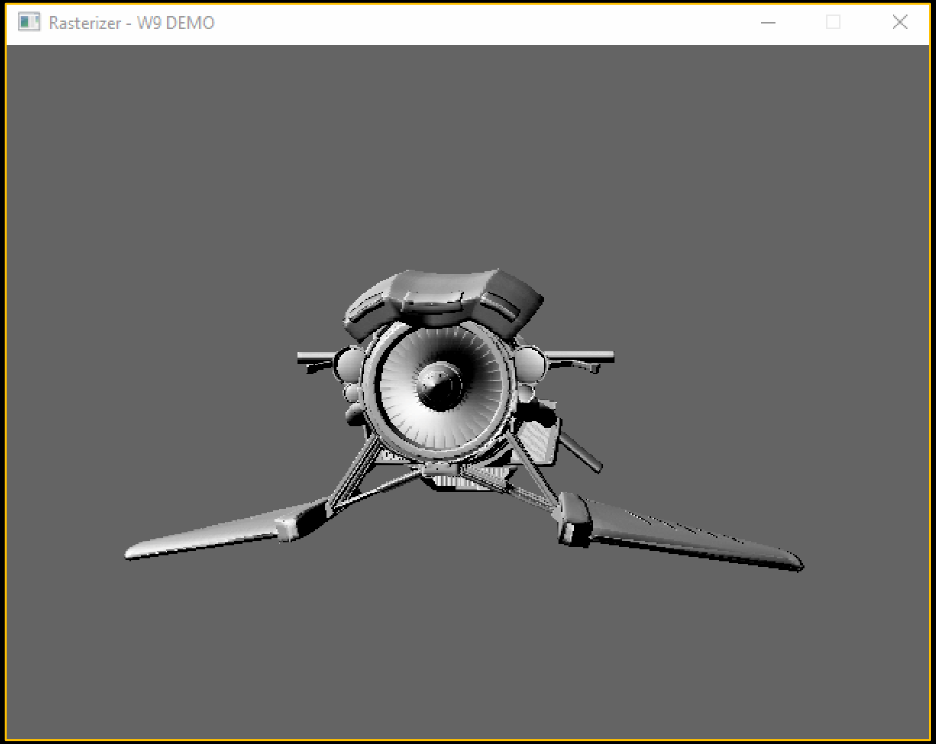
```
for (uint32_t i = 0; i < indices.size(); i += 3)
{
    uint32_t index0 = indices[i];
    uint32_t index1 = indices[size_t(i) + 1];
    uint32_t index2 = indices[size_t(i) + 2];

    const Vector3& p0 = vertices[index0].position;
    const Vector3& p1 = vertices[index1].position;
    const Vector3& p2 = vertices[index2].position;
    const Vector2& uv0 = vertices[index0].uv;
    const Vector2& uv1 = vertices[index1].uv;
    const Vector2& uv2 = vertices[index2].uv;

    const Vector3 edge0 = p1 - p0;
    const Vector3 edge1 = p2 - p0;
    const Vector2 diffX = Vector2(_x: uv1.x - uv0.x, _y: uv2.x - uv0.x);
    const Vector2 diffY = Vector2(_x: uv1.y - uv0.y, _y: uv2.y - uv0.y);
    float r = 1.f / Vector2::Cross(diffX, diffY);

    Vector3 tangent = (edge0 * diffY.y - edge1 * diffY.x) * r;
    vertices[index0].tangent += tangent;
    vertices[index1].tangent += tangent;
    vertices[index2].tangent += tangent;
}

//Create the Tangents (reject)
for (auto& v :Vertex& : vertices)
{
    v.tangent = Vector3::Reject(v.tangent, v.normal).Normalized();
```

https://stackoverflow.com/questions/5255806/how-to-calculate-tangent-and-binormal

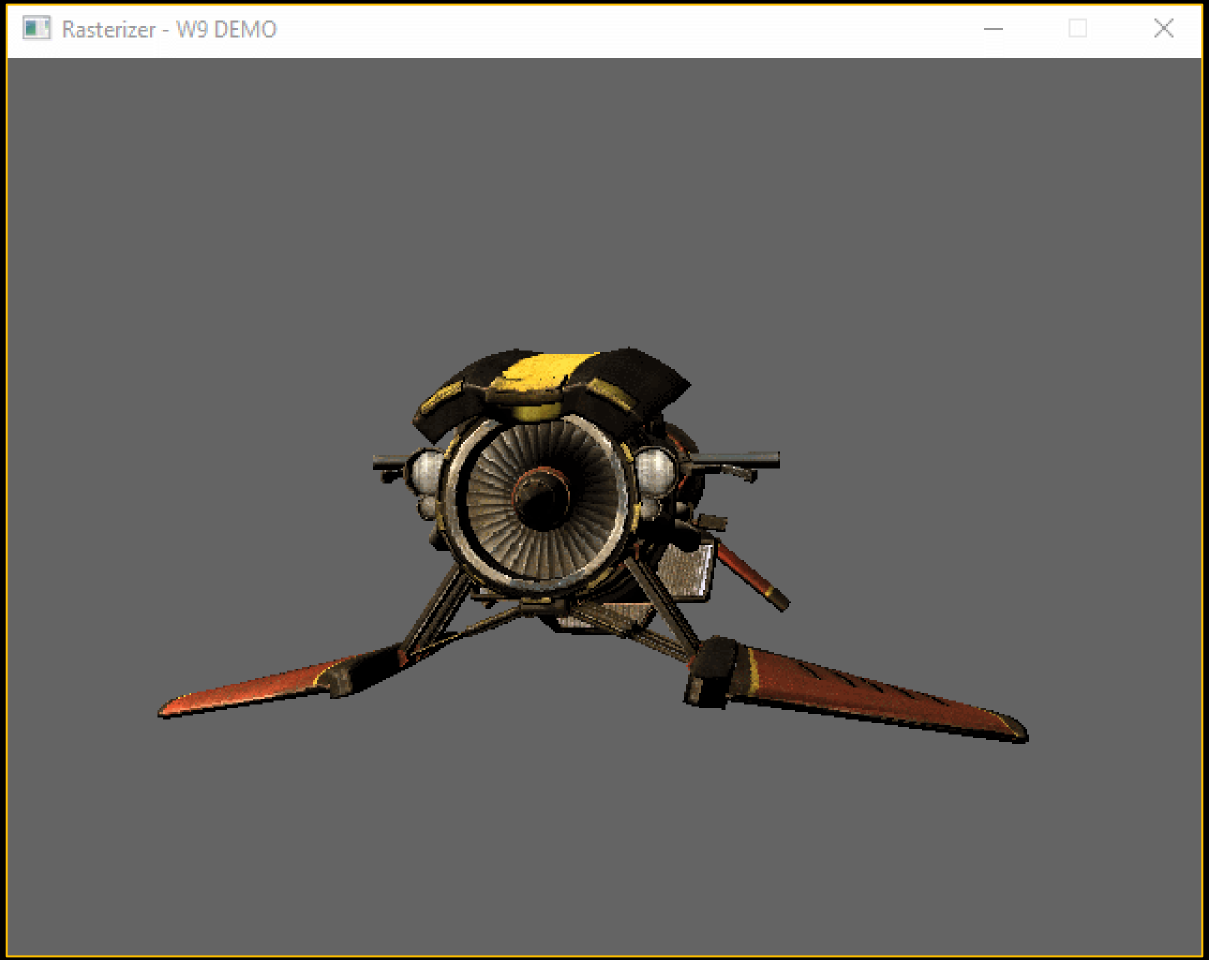
Rasterization: Normal Maps

- So, after parsing and storing the necessary information, transform the tangent into world space (just as the normal) in your vertex transformation function.
- In your PixelShading function, sample the normal from the normal map, transform it in the correct tangent space and use it in your lambert's cosine law calculation.





Rasterization: Normal Maps



Diffuse + Sampled Normals

Rasterization: Phong

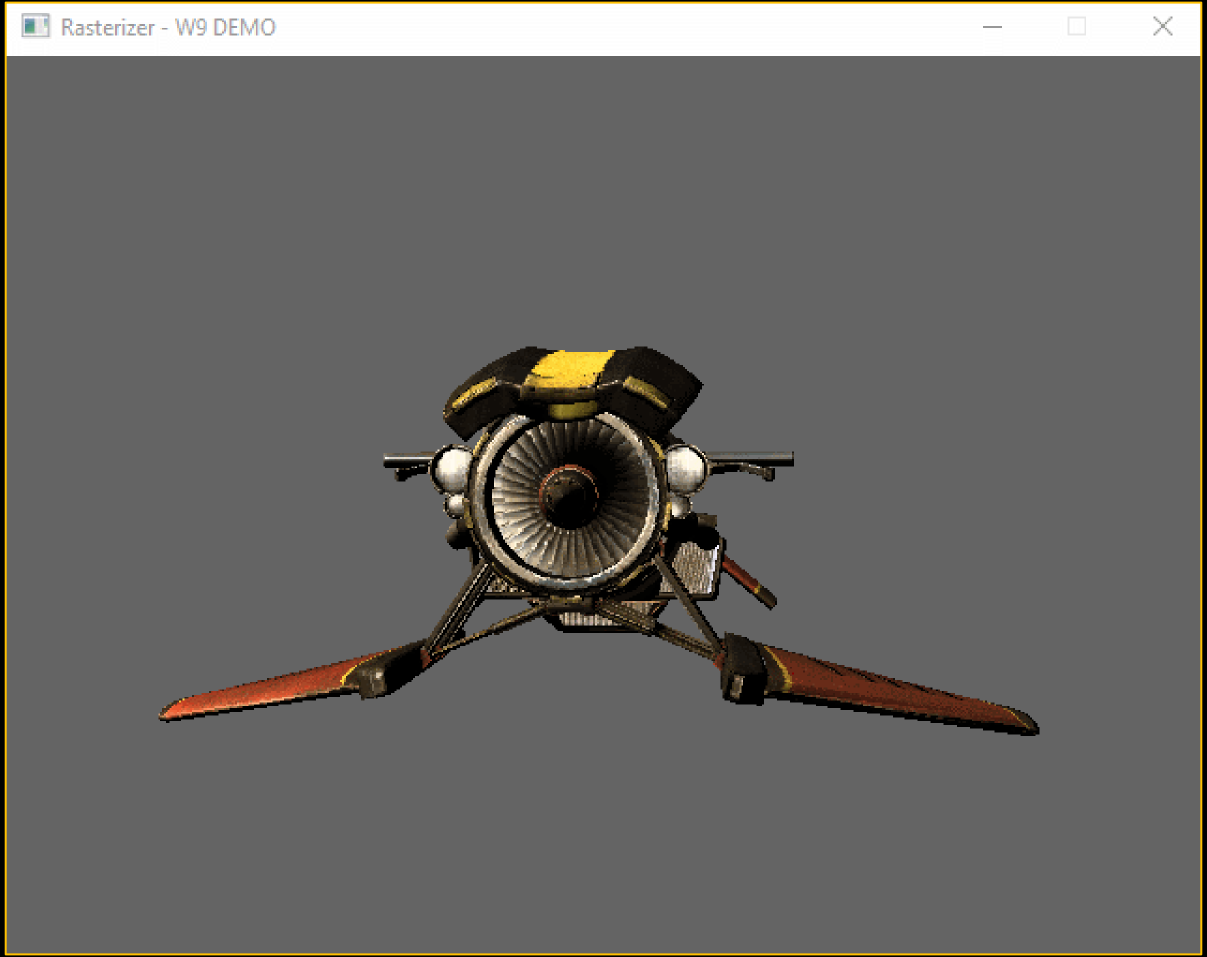
- And to get an even better result, you can also add Phong specular to the pixel shading.
- The math is the same as the Raytracer! ☺
- Some remarks though:
    - Find a way to pass your viewDirection to the PixelShading function.
        - Hint: create viewDirection in vertex transformation function per vertex and interpolate in the rasterizer OR calculate it in the PixelShading function. In both cases you need the cameraPosition and the position of the vertex or pixel in world space!
    - Make sure your vectors for lighting calculations point in the correct direction. This might be different compared to the ray tracer…
    - Instead of the using hardcoded SpecularColor and Phong Exponent values, sample from the SpecularMap and the Glossiness Map.
    - Every component in the glossiness map is the same as it is a greyscale map. Usually, we optimize this!
    - Values of the glossiness map are clamped and don't give you control over the shininess. Often, we create a variable called 'shininess' and multiply our sampled exponent with this value. Add the specular value, just like with the ray tracer, to the final color. Don't forget to MaxToOne!

Rasterization: Final Result

Camera
Pos: { 0.f, 0.f, 0.f }
FovAngle: 45.f

Vehicle
Pos: { 0.f, 0.f, 50.f }
Rot: 1 Radians / Sec

Shading
LightDir: {.577f, -.577f, .577f}
LightIntensity: 7.f
Shininess: 25.f
Ambient: {.025f, .025f, .025f}



Diffuse + Specular + Ambient

Rasterization: What to do?

- Transform all necessary attributes accordingly, interpolate them (correct depth interpolation) in the rasterization stage and store them in the vertex output.
- Shade your model with Lambert Diffuse, using a diffuse texture map, in a separate function called PixelShading(const Vertex_Out& v).
- Implement tangents in your .obj parser, transform (just as normals, in world space) and interpolate them as well.
- Shade your model using the normals from the normal map.
- Add Phong specular using the provided SpecularColor and Glossiness maps (scale with Shininess value).
- Key-Bindings!
  - F5 > Toggle Rotation
  - F6 > Toggle NormalMap
  - F7 > Cycle ShadingMode [ObservedArea (OA) > Diffuse (incl OA) > Specular (incl OA) > Combined]

GOOD LUCK!