Computer Graphics II
– Normal Mapping

Kai Lawonn

Introduction

- All our scenes are filled with polygons consisting of hundreds or maybe thousands of flat triangles
- Boosted the realism by pasting 2D textures on triangles
- Take a close look and it is easy to see the underlying flat surfaces
- Most real-life surface are not flat however and exhibit a lot of (bumpy) details

Introduction

- A brick surface is rough and not completely flat: contains sunken cement stripes and detailed little holes and cracks
- If we were to view such a brick surface in a lighted scene the immersion gets easily broken

Introduction

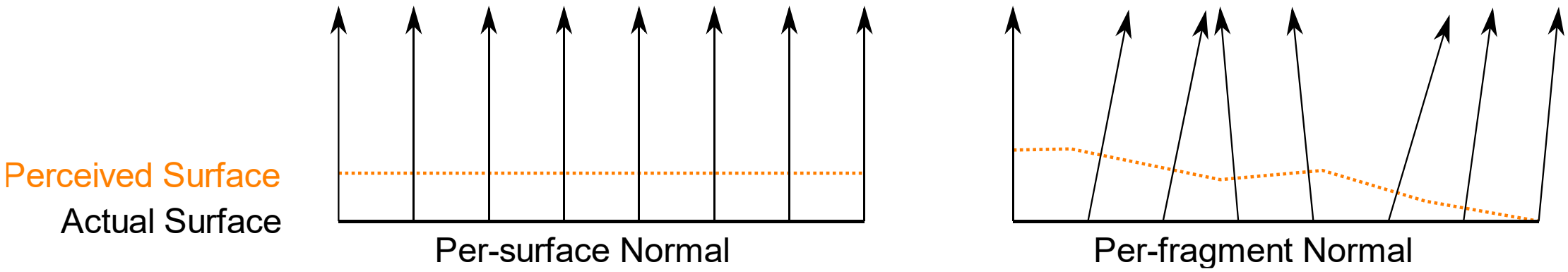
- Lighting not taking small cracks and holes into account, completely ignores the deep stripes between the bricks
- Surface looks perfectly flat
- Partly solve the flatness by using a specular map (pretend some surfaces are less lit due to depth or other details), that's more of a hack than a real solution
- Need a way to inform the lighting system about all the little depth-like details of the surface

Introduction

- From a light's perspective: how comes the surface is lit as a completely flat surface? → surface's normal vector
- From the lighting algorithm's: way it determines the shape of an object is by its perpendicular normal vector
- Brick surface has a single normal vector → surface is uniformly lit

Introduction

- What if, instead of a per-surface normal that is the same for each fragment, use a per-fragment normal that is different for each fragment
- Slightly deviate the normal vector based on a surface's little details; as a result this gives the illusion the surface is a lot more complex:



Introduction

- Using per-fragment normal leads to the illusion a surface consists of tiny little planes (perpendicular to the normal vectors) giving the surface an enormous boost in detail
- This technique to use per-fragment normals compared to per-surface normals is called normal mapping or bump mapping:

Normal Mapping

Normal Mapping

- Need a per-fragment normal
- Similar to diffuse maps and specular maps, can use a 2D texture to store per-fragment data
- Aside from color and lighting data we can also store normal vectors in a 2D texture
- This way we can sample from a 2D texture to get a normal vector for that specific fragment

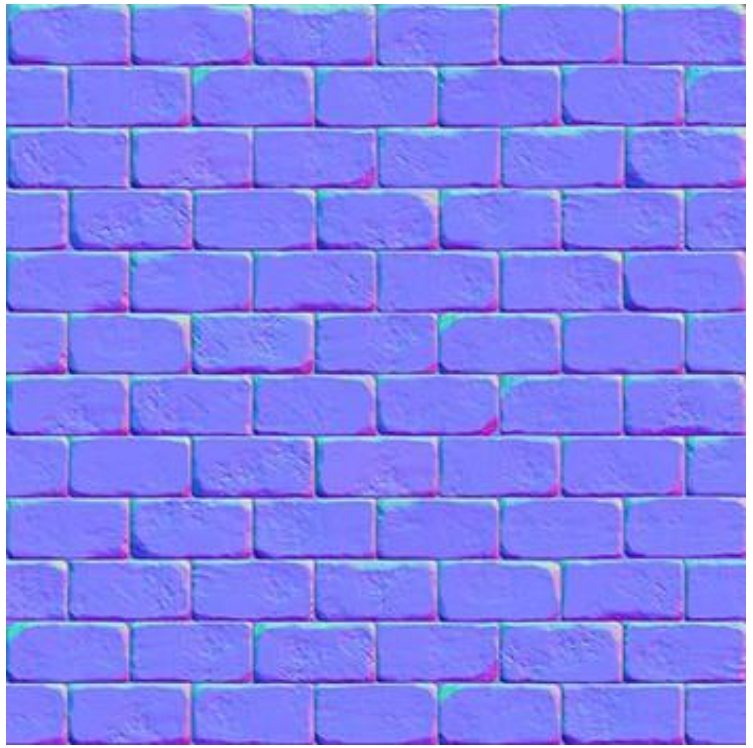
Normal Mapping

- Normal vectors are geometric entities, textures used for color information → storing normal vectors in a texture not obvious
- Color vectors in a texture represented as a 3D vector with an r, g and b component
- Can store a normal vector's x, y and z component in the respective color components
- Normal vectors range between -1 and 1 so they're first mapped to [0,1]:

```
vec3 rgb_normal = normal * 0.5 + 0.5; // transforms from [-1,1] to [0,1]
```

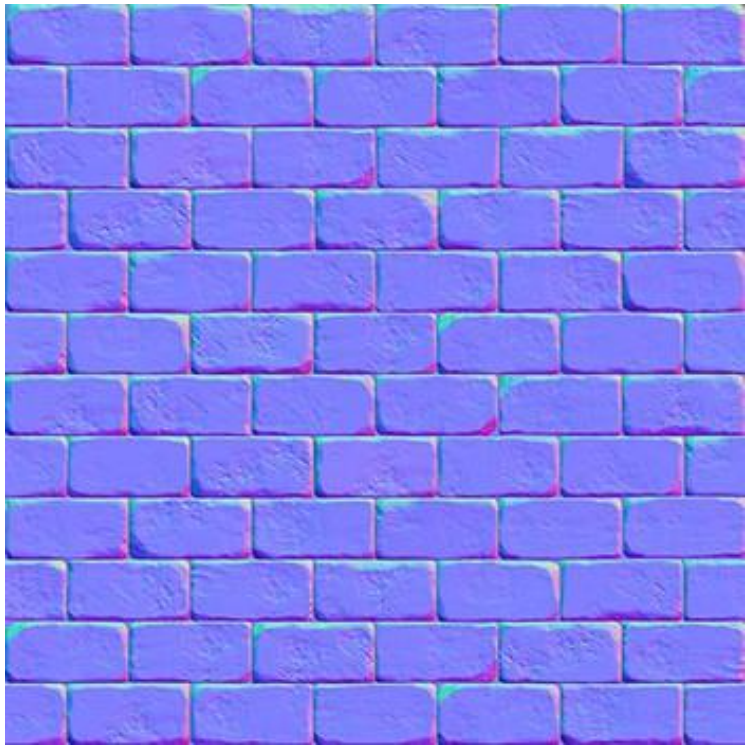
Normal Mapping

- With normal vectors transformed to an RGB color component like this, can store a per-fragment normal derived from the shape of a surface onto a 2D texture
- Example normal map of the brick surface:

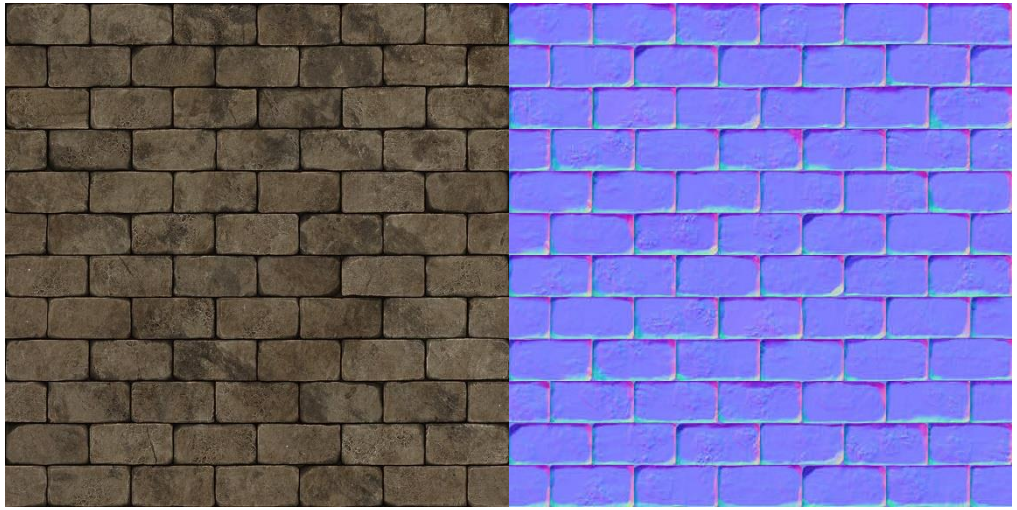


Normal Mapping

- Almost all normal maps have a blue-ish tint
- Normals pointing outwards (towards the positive z-axis) are (0, 0, 1): a blue-ish color
- Slight color deviations are normals that are slightly off from the general positive z direction
- E.g., top of each brick, color tends to green (top have normals pointing in the positive y direction (0, 1, 0) → green)

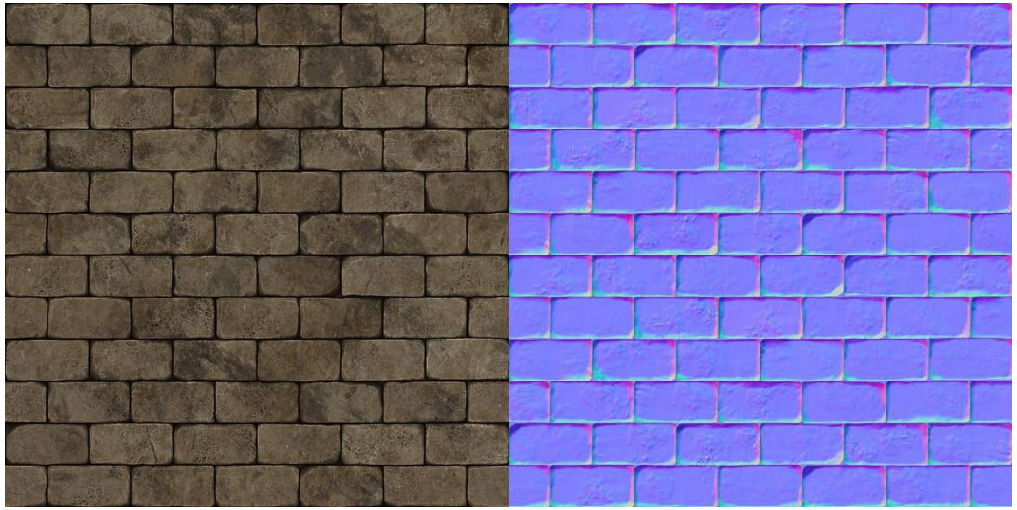


Normal Mapping



- With a plane looking at the positive z-axis, these diffuse texture and normal map can be used to apply normal mapping
- This normal map is different from the previous one (OpenGL reads texture coordinates with the y (or V) coordinates reversed) → y (or green) component reversed (green colors pointing downwards)
- If we forget this, lighting will be incorrect

Normal Mapping



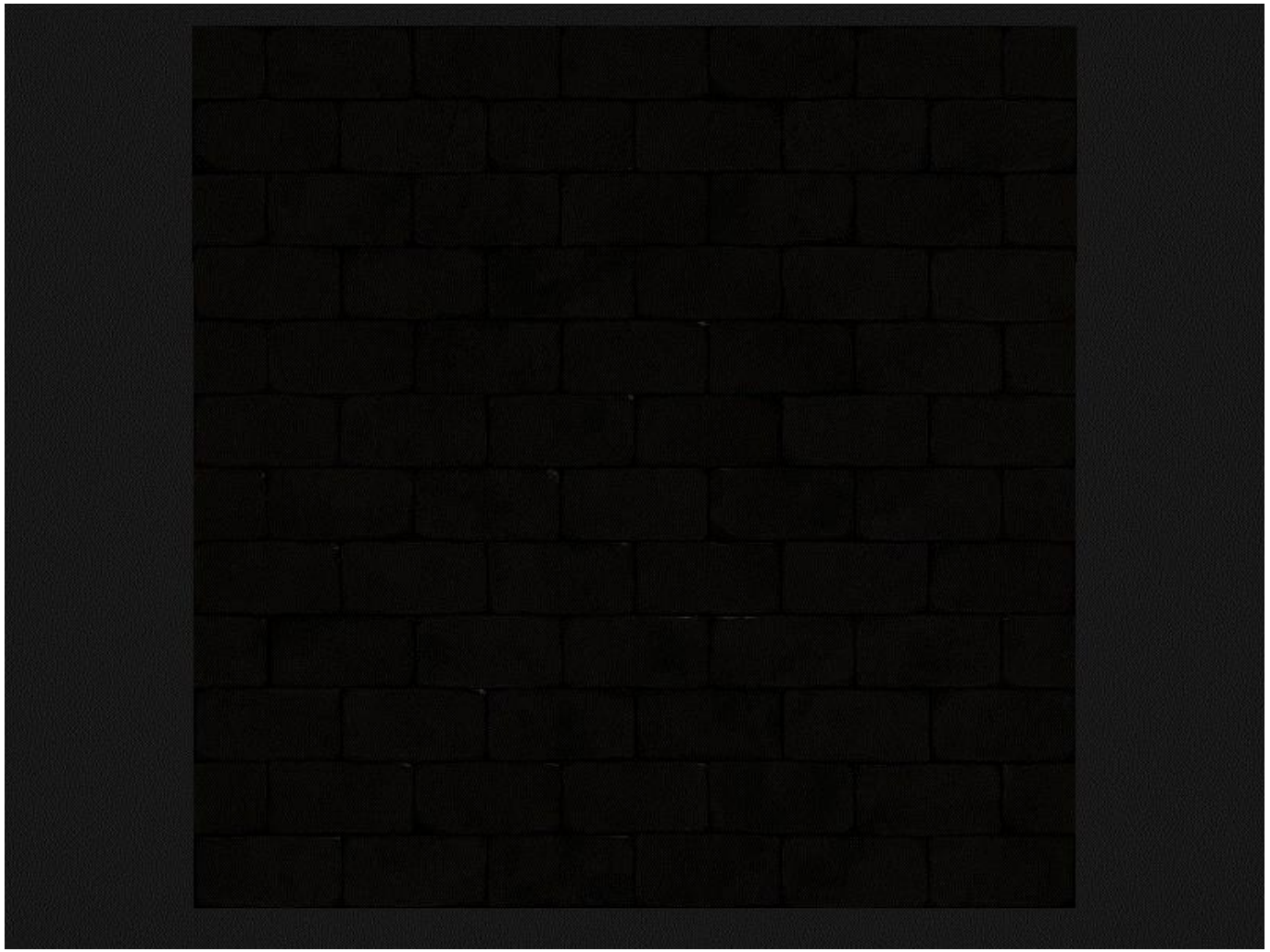
- Load both textures, bind them, and render a plane with the following changes in a lighting fragment shader:

```
uniform sampler2D normalMap;

void main()
{
    // obtain normal from normal map in range [0,1]
    normal = texture(normalMap, fs_in.TexCoords).rgb;
    // transform normal vector to range [-1,1]
    normal = normalize(normal * 2.0 - 1.0);
    [...]
    // proceed with lighting as normal
}
```

F5…

- … a brick wall with normal mapping

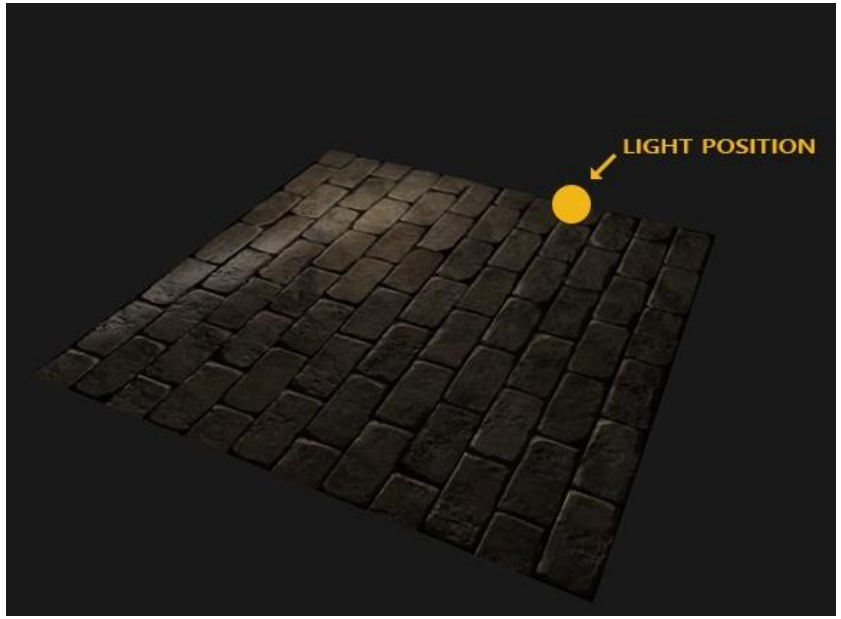


Issue

- Normal map had normal vectors that all roughly pointed in the positive z direction, just like the plane's surface normal
- What if we used the same normal map on a plane with a surface normal vector pointing in the positive y direction?

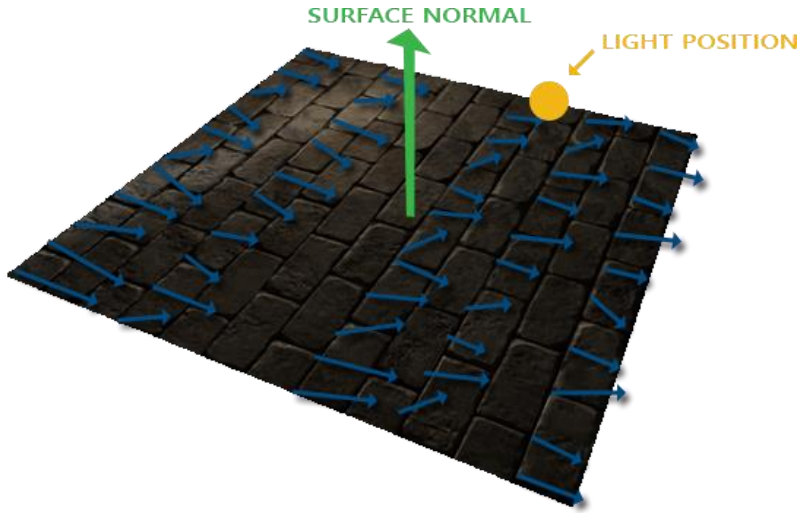
Issue

- Lighting doesn't look right → Sampled normals point roughly in the positive z direction, but should point in the positive y direction
- Result is lighting thinks the surface's normals are the same as before



Issue

- Sampled normals approximately look like on this surface:
- Normals roughly point in the positive z direction, but should be along the positive y direction



Issue

- Possible solution: define a normal map for each possible direction of a surface
- For a cube, need 6 normal maps, but with advanced models with a lot possible surface directions this becomes an infeasible approach
- Another solution: lighting in a coordinate space where the normal map vectors always point roughly in the positive z direction
- Other lighting vectors are then transformed relative to this
- Then, can always use the same normal map, regardless of orientation

Tangent Space

Introduction

- Vectors in a normal map are expressed in tangent space (normals point roughly in the positive z direction)
- Tangent space is local to the surface of a triangle: normals are relative to the local reference frame of the individual triangles
- Local space of the normal map's vectors; they're all defined pointing in the positive z direction regardless of the final transformed direction
- Using a specific matrix we can then transform normal vectors from this local tangent space to world or view coordinates, orienting them along the final mapped surface's direction

Introduction

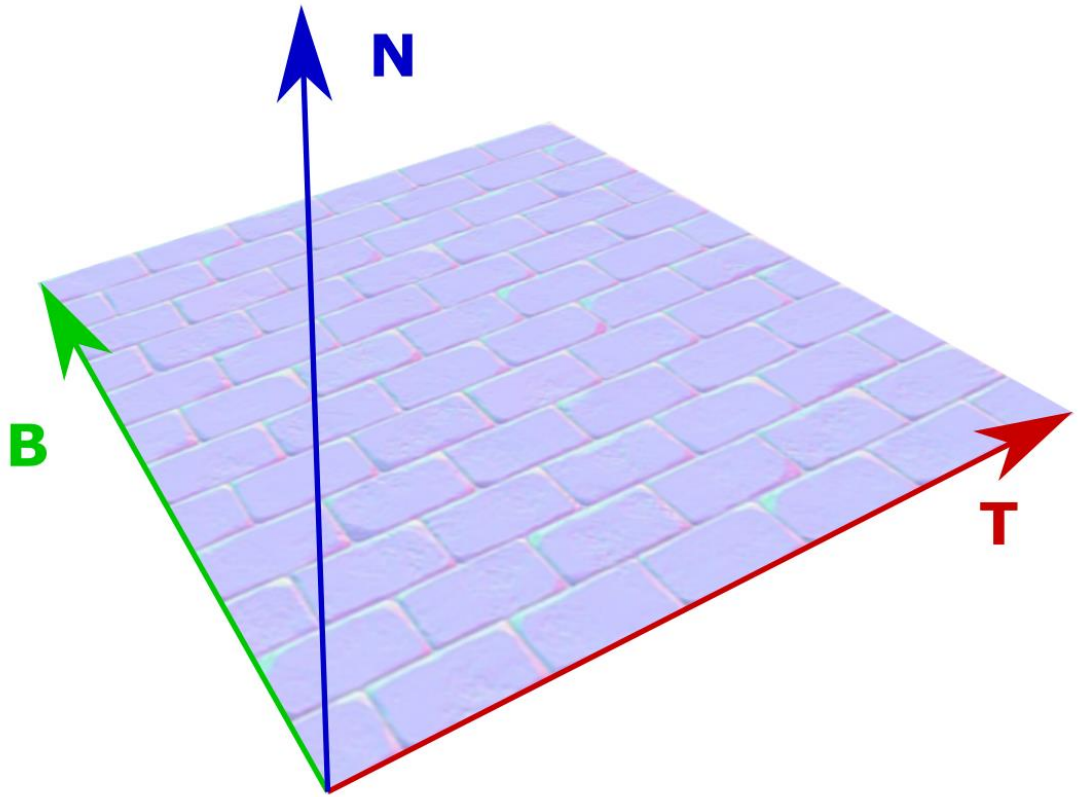
- Assume having incorrect normal mapped surface (previous example)
- Normal map is defined in tangent space → calculate a matrix to transform normals from tangent space to a different space (aligned with the surface's normal direction)
- This case: normal vectors pointing roughly in the positive y direction
- Calculate such a matrix for any type of surface → properly align the tangent space's z direction to any surface's normal direction

TNB Matrix

- Matrix is called TBN matrix: tangent, bitangent and normal vector
- These are the vectors we need to construct this matrix
- Change-of-basis matrix that transforms tangent-space vector to different coordinate space needs three perpendicular vectors (aligned along the surface of a normal map: up, right and forward vector - remember camera lecture)

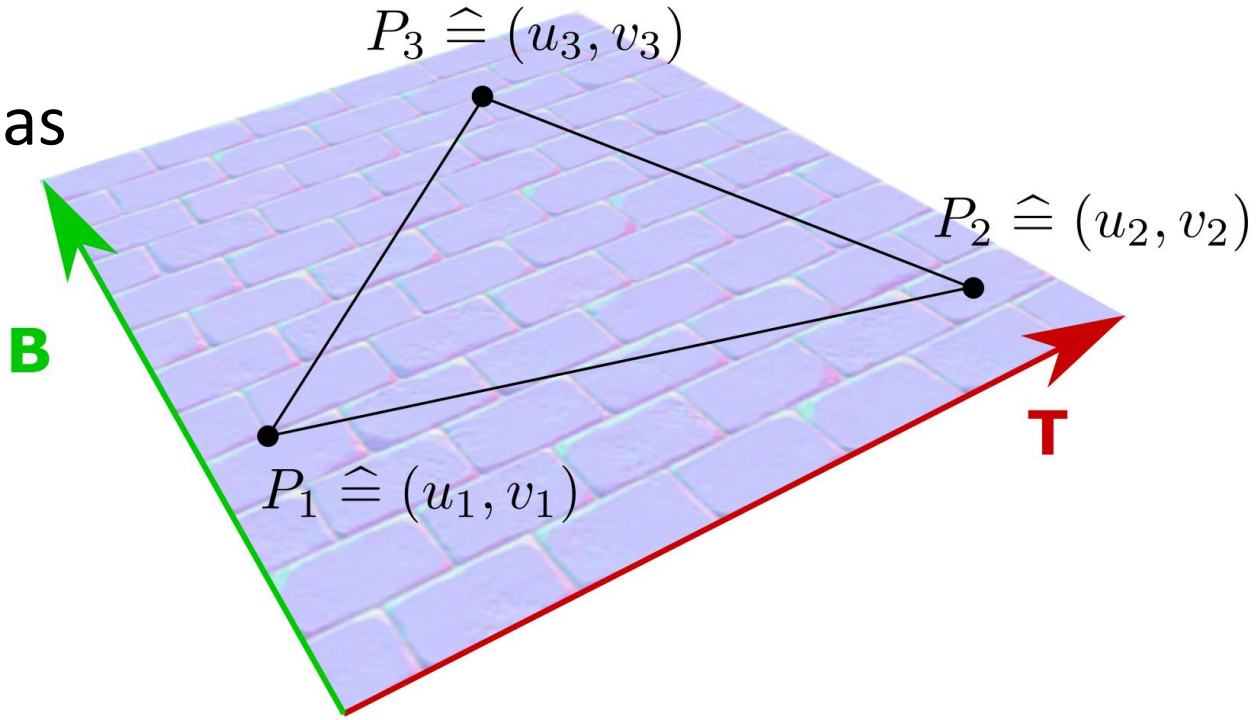
TNB Matrix

- Up vector is the surface's normal vector
- The right and forward vector are the tangent and bitangent vector respectively:



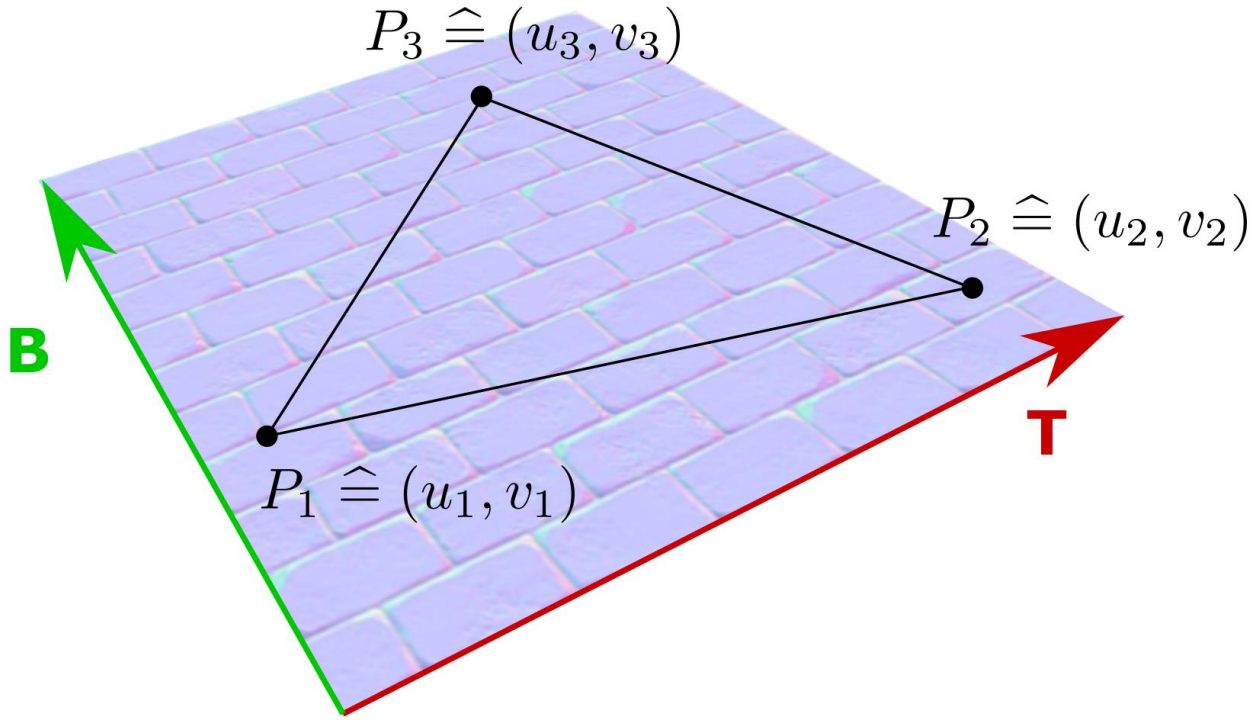
TNB Matrix

- Calculating the tangent and bitangent is not as straightforward as the normal vector
- Direction of the normal map's tangent and bitangent vector align with surface's texture coordinates
- Use this to calculate tangent and bitangent for each surface



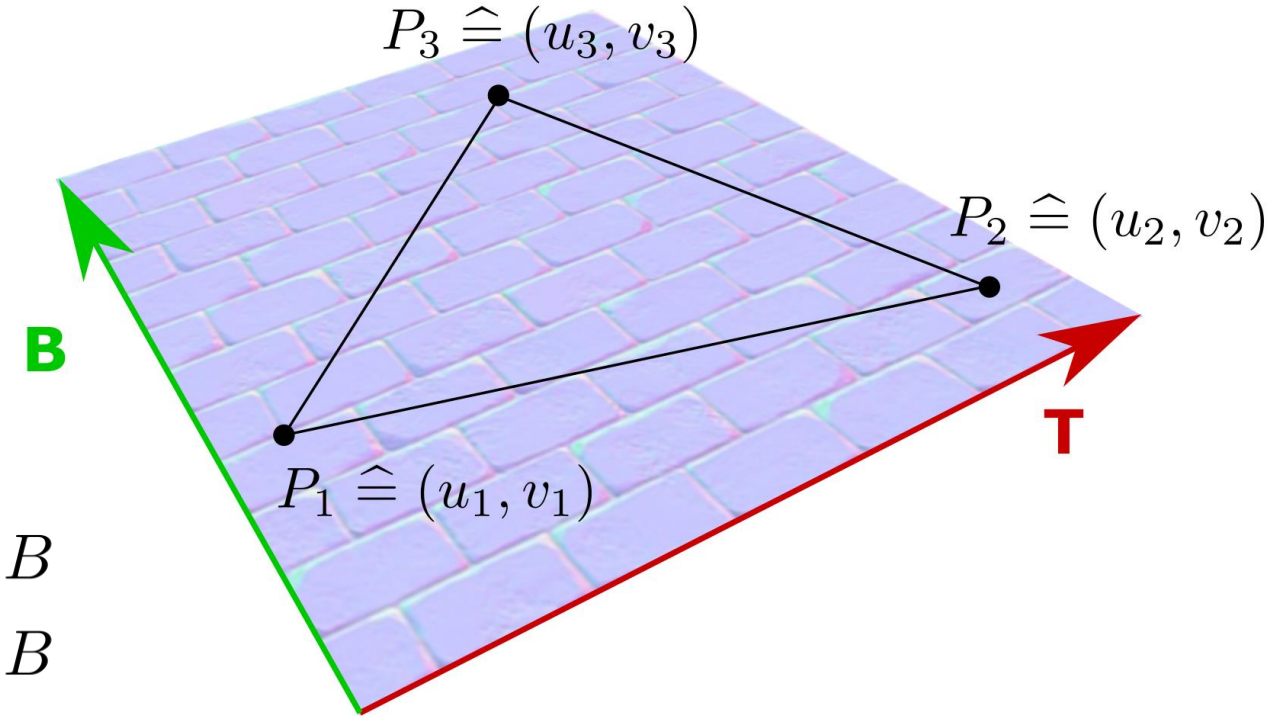
TNB Matrix

$$P_1 = u_1 T + v_1 B$$
$$P_2 = u_2 T + v_2 B$$
$$P_3 = u_3 T + v_3 B$$



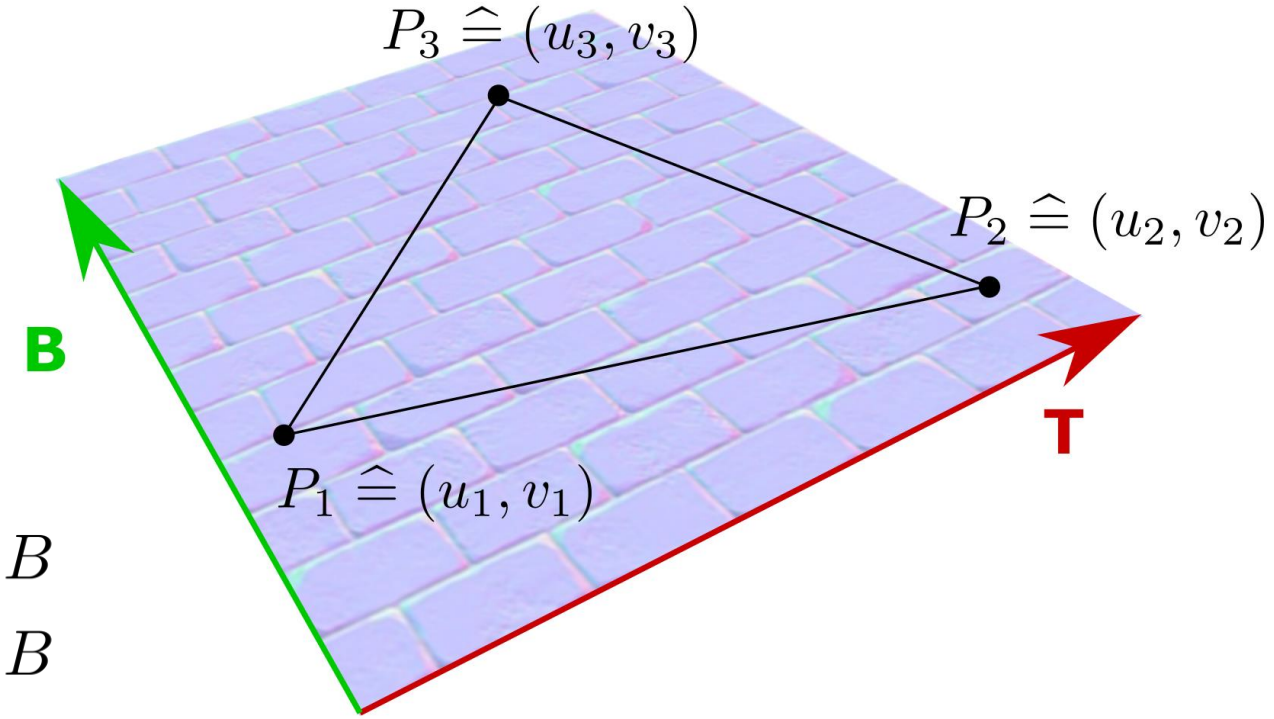
TNB Matrix

$$P_1 = u_1 T + v_1 B$$
$$P_2 = u_2 T + v_2 B$$
$$P_3 = u_3 T + v_3 B$$
$$\Rightarrow$$
$$P_2 - P_1 = (u_2 - u_1)T + (v_2 - v_1)B$$
$$P_3 - P_2 = (u_3 - u_2)T + (v_3 - v_2)B$$

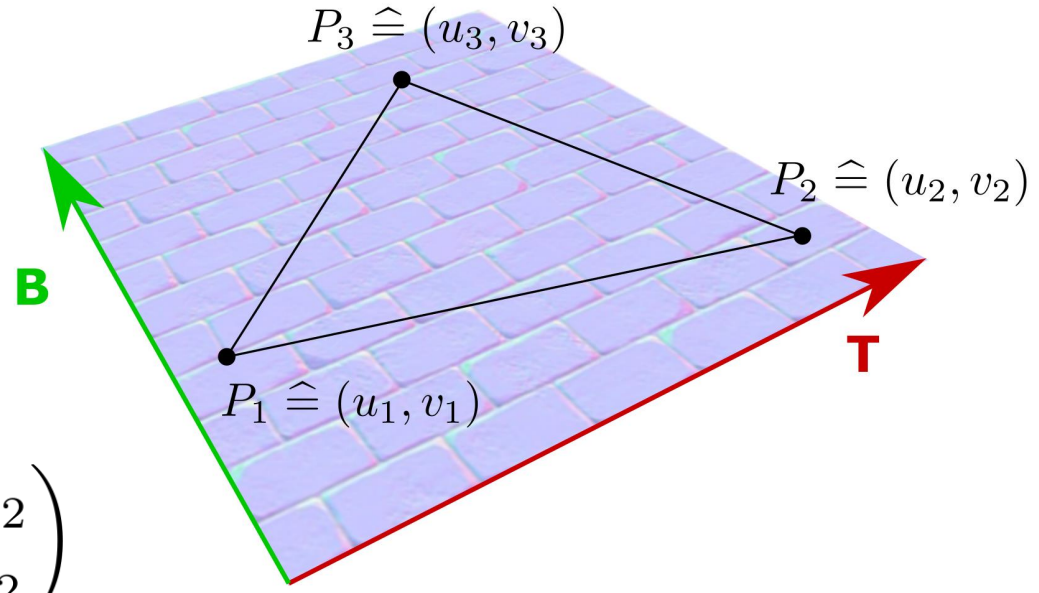


TNB Matrix

$$P_1 = u_1 T + v_1 B$$
$$P_2 = u_2 T + v_2 B$$
$$P_3 = u_3 T + v_3 B$$
$$\Rightarrow$$
$$P_2 - P_1 = (u_2 - u_1)T + (v_2 - v_1)B$$
$$P_3 - P_2 = (u_3 - u_2)T + (v_3 - v_2)B$$
$$\Rightarrow$$
$$\begin{pmatrix} P_2 - P_1 & P_3 - P_2 \end{pmatrix} = \begin{pmatrix} T & B \end{pmatrix} \begin{pmatrix} u_2 - u_1 & u_3 - u_2 \\ v_2 - v_1 & v_3 - v_2 \end{pmatrix}$$



TNB Matrix



$$\begin{pmatrix} P_2 - P_1 & P_3 - P_2 \end{pmatrix} = \begin{pmatrix} T & B \end{pmatrix} \begin{pmatrix} u_2 - u_1 & u_3 - u_2 \\ v_2 - v_1 & v_3 - v_2 \end{pmatrix}$$

$$\Rightarrow$$

$$\begin{pmatrix} P_2 - P_1 & P_3 - P_2 \end{pmatrix} \begin{pmatrix} u_2 - u_1 & u_3 - u_2 \\ v_2 - v_1 & v_3 - v_2 \end{pmatrix}^{-1} = \begin{pmatrix} T & B \end{pmatrix}$$

$$\begin{pmatrix} T & B \end{pmatrix} = \frac{\begin{pmatrix} P_2 - P_1 & P_3 - P_2 \end{pmatrix} \begin{pmatrix} v_3 - v_2 & -u_3 + u_2 \\ -v_2 + v_1 & u_2 - u_1 \end{pmatrix}}{(u_2 - u_1)(v_3 - v_2) - (v_2 - v_1)(u_3 - u_2)}$$

Tangents and Bitangents

- Previously, had a simple 2D plane looking at the positive z direction
- Now, normal mapping using tangent space → independent of orientation
- Going to manually calculate this surface's tangent and bitangent vectors

Tangents and Bitangents

- Assuming the plane is built up from the following vectors (with 1, 2, 3 and 1, 3, 4 as its two triangles):

```
// positions
glm::vec3 pos1(-1.0f,  1.0f, 0.0f);
glm::vec3 pos2(-1.0f, -1.0f, 0.0f);
glm::vec3 pos3( 1.0f, -1.0f, 0.0f);
glm::vec3 pos4( 1.0f,  1.0f, 0.0f);
// texture coordinates
glm::vec2 uv1(0.0f, 1.0f);
glm::vec2 uv2(0.0f, 0.0f);
glm::vec2 uv3(1.0f, 0.0f);
glm::vec2 uv4(1.0f, 1.0f);
// normal vector
glm::vec3 nm(0.0f, 0.0f, 1.0f);
```

Calculate

$$\begin{pmatrix} T & B \end{pmatrix} = \frac{\begin{pmatrix} P_2 - P_1 & P_3 - P_2 \end{pmatrix} \begin{pmatrix} v_3 - v_2 & -u_3 + u_2 \\ -v_2 + v_1 & u_2 - u_1 \end{pmatrix}}{(u_2 - u_1)(v_3 - v_2) - (v_2 - v_1)(u_3 - u_2)}$$

- First calculate the first triangle's edges and delta UV coordinates:

```
// triangle 1
// ----------
glm::vec3 edge1 = pos2 - pos1;
glm::vec3 edge2 = pos3 - pos1;
glm::vec2 deltaUV1 = uv2 - uv1;
glm::vec2 deltaUV2 = uv3 - uv1;
```

Calculate

$$\begin{pmatrix} T & B \end{pmatrix} = \frac{\begin{pmatrix} P_2 - P_1 & P_3 - P_2 \end{pmatrix} \begin{pmatrix} v_3 - v_2 & -u_3 + u_2 \\ -v_2 + v_1 & u_2 - u_1 \end{pmatrix}}{(u_2 - u_1)(v_3 - v_2) - (v_2 - v_1)(u_3 - u_2)}$$

- Start following the equation:

```
float f = 1.0f / (deltaUV1.x * deltaUV2.y - deltaUV2.x * deltaUV1.y);

tangent1.x = f * (deltaUV2.y * edge1.x - deltaUV1.y * edge2.x);
tangent1.y = f * (deltaUV2.y * edge1.y - deltaUV1.y * edge2.y);
tangent1.z = f * (deltaUV2.y * edge1.z - deltaUV1.y * edge2.z);

bitangent1.x = f * (-deltaUV2.x * edge1.x + deltaUV1.x * edge2.x);
bitangent1.y = f * (-deltaUV2.x * edge1.y + deltaUV1.x * edge2.y);
bitangent1.z = f * (-deltaUV2.x * edge1.z + deltaUV1.x * edge2.z);
```

Tangents and Bitangents

- Triangle flat shape → only calculate a single tangent/bitangent pair per triangle (same for each of the triangle's vertices)
- Note, most implementations (model loaders, terrain generators) have triangles that share vertices with other triangles
- Then, usually average the vertex properties like normals and tangents/bitangents for each vertex to get a smoother result
- Here, plane's triangles also shares some vertices, but both triangles are parallel → no need to average

Tangent Space Normal Mapping

- Have to create a TBN matrix in the shaders
- For this, pass the calculated tangent and bitangent vectors to the vertex shader as vertex attributes:

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;
layout (location = 3) in vec3 aTangent;
layout (location = 4) in vec3 aBitangent;
```

Tangent Space Normal Mapping

- Then, create the TBN matrix:

```
void main()
{
   …
   vec3 T = normalize(vec3(model * vec4(aTangent,   0.0)));
   vec3 B = normalize(vec3(model * vec4(aBitangent, 0.0)));
   vec3 N = normalize(vec3(model * vec4(aNormal,    0.0)));
   mat3 TBN = mat3(T, B, N)
   …
}
```

No need for the bitangent variable in the vertex shader

All three TBN vectors are perpendicular, so calculate the bitangent in the vertex shader by: vec3 B = cross(T, N);

Tangent Space Normal Mapping

- What now with the TBN matrix?
- Two ways to use a TBN matrix for normal mapping:
  - 1. Take TBN matrix, give it to the fragment shader → transform sampled normal from tangent space to world space with TBN; the normal is then in the same space as the other lighting variables
  - 2. Take inverse of TBN matrix: transform not the normal, but the other relevant lighting variables to tangent space; the normal is then again in the same space as the other lighting variables

Tangent Space Normal Mapping

- 1. Pass the TBN matrix to the fragment shader
- Multiply the sampled tangent space normal with this TBN matrix to transform the normal vector to the same reference space as the other lighting vectors
- This way all the lighting calculations (specifically the dot product) make sense

Tangent Space Normal Mapping

- Sending the TBN matrix to the fragment shader:

```
out VS_OUT {
    vec3 FragPos;
    vec2 TexCoords;
    mat3 TBN;
} vs_out;
…
void main()
{
   …
   vs_out.TBN = mat3(T, B, N);
}
```

Tangent Space Normal Mapping

- Fragment shader input variable:

```
in VS_OUT {
    vec3 FragPos;
    vec2 TexCoords;
    mat3 TBN;
} fs_in;
```

- TBN matrix update the normal mapping code (tangent-to-world space transformation):

```
normal = texture(normalMap, fs_in.TexCoords).rgb;
normal = normalize(normal * 2.0 - 1.0);
normal = normalize(fs_in.TBN * normal);
```

Tangent Space Normal Mapping

- 2. Take the inverse of TBN to transform relevant world-space vectors to the space the sampled normal vectors are in: tangent space
- The construction of the TBN matrix remains the same, but we first inverse the matrix before sending it to the fragment shader:

```
Vs_out.TBN = transpose(mat3(T, B, N));
```

- TBN orthogonal matrix: $TBN^{-1} = TBN^{T}$

Tangent Space Normal Mapping

- Within fragment shader, do not transform the normal vector, but transform other relevant vectors to tangent space (lightDir, viewDir)
- Then, each vector is in the same coordinate system: tangent space

```
void main()
{
   vec3 normal = texture(normalMap, fs_in.TexCoords).rgb;
   normal = normalize(normal * 2.0 - 1.0);

   vec3 lightDir = fs_in.TBN * normalize(lightPos - fs_in.FragPos);
   vec3 viewDir = fs_in.TBN * normalize(viewPos - fs_in.FragPos);
   …
}
```

Tangent Space Normal Mapping

- 2. approach more work and requires more matrix multiplications in the fragment shader (which are slightly expensive), why bother with this?
- Transforming vectors from world to tangent space advantage: can transform relevant vectors to tangent space in the vertex shader (instead of in the fragment shader)
- Works, because lightPos and viewPos do not change each fragment run
- fs_in.FragPos can also calculate its tangent-space position in the vertex shader (let fragment interpolate it)
- Basically, no need to transform any vector to tangent space in the fragment shader (necessary with the first approach as sampled normal vectors are specific to each fragment shader run)

Tangent Space Normal Mapping

- Instead sending the inverse of TBN to fragment shader, send tangent-space light position, view position and vertex position to the fragment shader (saves matrix multiplications in the fragment shader)
- Nice optimization as the vertex shader runs considerably less often than the fragment shader
- This is also the reason why this approach is often the preferred approach

Tangent Space Normal Mapping

```
out VS_OUT {
    vec3 FragPos;
    vec2 TexCoords;
    vec3 TangentLightPos;
    vec3 TangentViewPos;
    vec3 TangentFragPos;
} vs_out;

uniform vec3 lightPos;
uniform vec3 viewPos;

void main()
{
    …
    mat3 TBN = transpose(mat3(T, B, N));
    vs_out.TangentLightPos = TBN * lightPos;
    vs_out.TangentViewPos  = TBN * viewPos;
    vs_out.TangentFragPos  = TBN * vs_out.FragPos;
}
```

Tangent Space Normal Mapping

- Fragment shader: use these new input variables to calculate lighting in tangent space
- As the normal vector is already in tangent space the lighting makes sense
- Can orient plane in any way and the lighting would still be correct:

```
glm::mat4 model = glm::mat4(1.0f);
model = glm::rotate(model, glm::radians((float)glfwGetTime() * -10.0f),
glm::normalize(glm::vec3(1.0, 0.0, 1.0))); // rotate the quad
…
renderQuad();
```

F5…

- …nice wall

One Last Thing

One Last Thing

- Last technique to slightly improve its quality without too much extra cost
- When tangent vectors are calculated on larger meshes, sharing vertices the tangent vectors are generally averaged to give nice and smooth results
- Problem is that TBN vectors could end up non-perpendicular → resulting TBN matrix would no longer be orthogonal
- Normal mapping will be only slightly off with a non-orthogonal TBN matrix

One Last Thing

- With the Gram-Schmidt process, can re-orthogonalize the TBN vectors such that each vector is again perpendicular to the other vectors (vertex shader):

```
vec3 T = normalize(vec3(model * vec4(aTangent, 0.0)));
vec3 N = normalize(vec3(model * vec4(aNormal, 0.0)));
// re-orthogonalize T with respect to N
T = normalize(T - dot(T, N) * N);
// then retrieve perpendicular vector B with the cross product of T and N
vec3 B = cross(N, T);

mat3 TBN = mat3(T, B, N)
```

Questions???