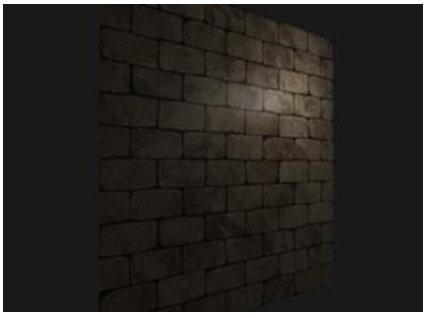
NORMAL MAPS



Textures help give detail to simple shapes with low triangle counts, but you can still easily tell that the underlying surfac e is flat.

- Example: A brick surface is very rough and not completely flat, with sunken cement stripes and lots of detailed holes and cracks. However, viewing such a brick surface in a lit scene breaks immersion due to the flatness of the surface when lit, as seen in the image to the right.
  - We can partially fix the flatness by using a specular map to pretend some surfaces are less lit due to depth or other details , but that's more of a hack than a real solution.

Below is the diffuse texture for the brick wall used in the image.

Set up your scene so that it appears similar to the image. You should be able to do this on your own by now. Just use the Blinn-Phong shader we made and modify the vertex shader slightly to utilize a quad instead of a 3D shape.

If we think about depth-based details from a light's perspective, the reason the surface is lit to look completely flat is because of the surface's normal vector.

- From the lighting technique's point of view, the only way it determines the shape of an object is by its perpendicular normal vector.
  - The brick surface only has a single normal vector, so the surface is uniformly lit based on this normal vector's direction.
- Instead of using a per-surface normal, we can use a per-fragment normal that is different for each fragment. This will allows us to slightly deviate the normal vector based on a surface's details, giving the illusion of the surface being more complex.
  - This technique is called **normal mapping** or **bump mapping**.
  - As you can see in the image to the right, this is incredibly powerful for making surfaces appear more complex, while also being relatively low cost.

Normal Mapping

Below is the normal map we'll be using.

We can sample the above 2D texture to get a normal vector for a specific fragment.

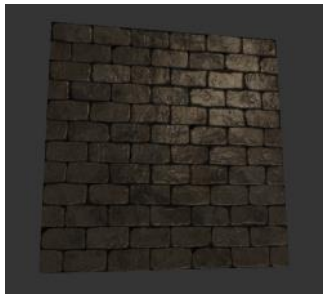
While normal vectors are geometric entities and textures are generally only used for color information, storing normal vectors in a texture may not be immediately obvious.

- If you think about color vectors in a texture, they are represented as a 3D vector with an r, g, and b component.
- We can similarly store a normal vector's x, y, and z component in the respective color components.
  - Normal vectors have a range of `[-1,1]`, which can easily be translated to RGB. Our normal map (the image to the right) represents such a translation.

Most normal maps you will find have a blue-ish tint.

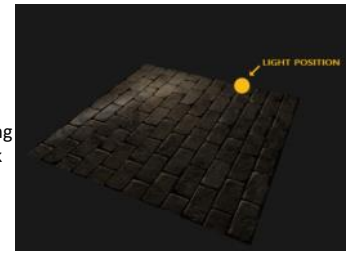
- This is because the normals are all closely pointing outwards towards the positive z-axis `(0, 0, 1)`: a blue-ish color.
- The deviations in color represent normal vectors that are slightly offset from the general positive z direction, giving a sense of depth to the texture.
  - Example: You can see in the image that the top of each brick is green-ish, which makes sense as the top side of a brick would have normals pointing more in the positive y direction `(0, 1, 0)`, which happens to be the color green.

NOTE: The supplied normal map is actually an upside-down version of the image above. This is because OpenGL reads texture coordinates with the y (or v) coordinate reversed from how textures are generally created. The linked normal map thus has its y (or green) component inversed.



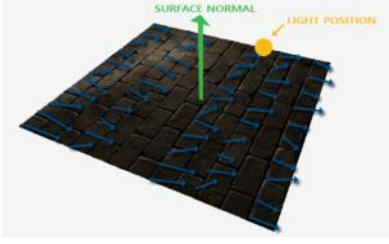
Load and bind both the diffuse and normal map and modify the fragment shader to use the normal map's per-fragment normals rather than the per-vertex normals of the vertex data.

- *Your result should look similar to the image to the right. It won't look like the previous images from earlier in the chapter because I pulled those straight from the learnopengl material rather than taking a snippet of my own.*
- If you can't tell if it's working, increase the material's shininess (I used a shininess value of 128.0) and it should be obvious whether it's working or not.
- Implementing a Normal Map



There is one issue that greatly limits this use of normal maps, though.

- The normal map we used had normal vectors that all pointed somewhat in the positive z direction, which worked because the surface normal was also pointing in the positive z direction.
- However, if we use the same normal map on the ground with a surface normal vector pointing in the y direction (by rotating the wall 90 degrees around the x-axis), the lighting doesn't look right, as seen in the image to the right.
  - This happens because the sampled normals of the plane still roughly point in the positive z direction, even though they should point in the positive y direction. As a result, the lighting thinks the surface's normals are the same as before when he plane was pointing towards the positive z direction.



The image to the right shows what the sampled normals approximately look like on the rotated brick wall surface.

- As you can see, all the normals point somewhat in the positive z direction, even though they should be pointing towards the positive y direction.
  - One solution to this problem is to define a normal map for each possible direction of the surface. In the case of a cube, we'd need 6 normal maps. However, with advanced meshes

that can have hundreds or more possible surface directions, this becomes an infeasible approach.
- A different solution exists that does all the lighting in a different coordinate space: a coordinate space where the normal map vectors always point towards the positive z direction; all other lighting vectors are then transformed relative to this positive z direction.
  - This way, we can always use the same normal map, regardless of orientation.
  - This coordinate space is called **tangent space**.

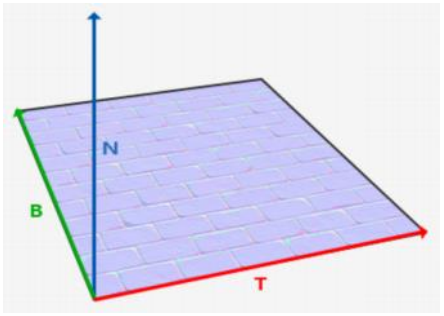
Tangent Space

Normal vectors in a normal map are expressed in tangent space, where normals always point roughly in the positive z direction.

- Tangent space is a space that is local to the surface of a triangle; they're all defined pointing in the positive z direction regardless of the final transformed direction, kind of like a local space of the normal map's vectors.
- 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.

As it stands, our normal map is looking in the positive y direction. The normal map is defined in tangent space, so one way to solve the problem is to calculate a matrix to transform normals from tangent space to a different space such that they're aligned with the surface's normal direction: the normal vectors are then all pointing roughly in the positive y direction.

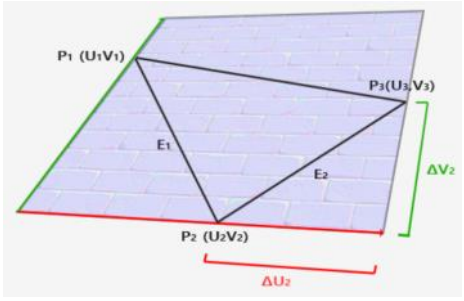
- The great thing about tangent space is that we can calculate this matrix for any type of surface so that we can properly align the tangent space's z direction to the surface's normal direction.
- Such a matrix is called a **TBN matrix**, where the letters depict a **Tangent**, **Bitanget**, and **Normal** vector.



The tangent, bitangent, and normal vector are needed to construct the TBN matrix.

- To construct such a change-of-basis matrix that transforms a tangent space vector to a different coordinate space, we need 3 perpendicular vectors that are aligned along the surface of a normal map: an up, right, and forward vector; similar to what we did in the camera chapter.
  - We already know the up vector, which is the surface's normal vector.
  - The right and forward vector are the tangent and bitangent vector, respectively.
    - We can see from the image to the right that the direction of the normal map's tangent and bitangent vector align with the direction in which we define a surface's texture coordinates. We can use this fact to calculate tangent and bitangent vectors for each surface.

As seen in the image to the right, acquiring the tangent and bitangent vector requires a bit of math.



- From the image, we can see that the texture coordinate differences of an edge $\boldsymbol{E_2}$ of a triangle (denoted as $\boldsymbol{\Delta U_2}$ and $\boldsymbol{\Delta V_2}$) are expressed in the same direction as the tangent vector $\boldsymbol{T}$ and bitangent vector $\boldsymbol{B}$.
- Because of this, we can write both displayed edges $\boldsymbol{E_1}$ and $\boldsymbol{E_2}$ of the triangle as a linear combination of the tangent vector $\boldsymbol{T}$ and the bitangent vector $\boldsymbol{B}$:
  $E_1 = \Delta U_1 T + \Delta V_1 B$
  $E_2 = \Delta U_2 T + \Delta V_2 B$
  which we can also write as:
  $(E_{1x}, E_{1y}, E_{1z}) = \Delta U_1 (T_x, T_y, T_z) + \Delta V_1 (B_x, B_y, B_z)$
  $(E_{2x}, E_{2y}, E_{2z}) = \Delta U_2 (T_x, T_y, T_z) + \Delta V_2 (B_x, B_y, B_z)$
- We can calculate $\boldsymbol{E}$ as the difference vector between two triangle positions, and $\boldsymbol{\Delta U}$ and $\boldsymbol{\Delta V}$ as their texture coordinate differences.
  - We're then left with two unknowns (tangent $\boldsymbol{T}$ and bitangent $\boldsymbol{B}$) and two equations. This allows us to solve for $\boldsymbol{T}$ and $\boldsymbol{B}$.
- The last equation allows us to write it in a different form: that of matrix multiplication:
  $$\begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$
- Try to visualize the matrix multiplications in your head and confirm that this is indeed the same equation.
  - An advantage of rewriting the equations in matrix form is that solving for $\boldsymbol{T}$ and $\boldsymbol{B}$ is easier to understand.
- If we multiply both sides of the equation by the inverse of the $\Delta U \Delta V$ matrix, we get:
  $$\begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix}^{-1} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$
- This allows us to solve for $\boldsymbol{T}$ and $\boldsymbol{B}$. This does require us to calculate the inverse of the delta texture coordinate matrix. We won't go into the matematical details of calculating the matrix's inverse, but it roughly translates to 1 over the determinant of the matrix, multiplied by its adjugate matrix:
  $$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \frac{1}{\Delta U_1 \Delta V_2 - \Delta U_2 \Delta V_1} \begin{bmatrix} \Delta V_2 & -\Delta V_1 \\ -\Delta U_2 & \Delta U_1 \end{bmatrix} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix}$$
- This final equation gives us a formula for calculating the tangent vector $\boldsymbol{T}$ and the bitangent vector $\boldsymbol{B}$ from a triangle's two edges and its texture coordinates.

Don't worry if you don't fully understand the mathematics behind this; you can mostly treat the mathematics as a black box. A s long as you understand that we calculate the tangents and bitangents from a triangle's vertices and its texture coordinates (since texture coordinates are in the same space as tangent vectors), you're halfway there.

Manual Calculation of Tangents and Bitangents

We want to implment normal mapping using tangent space so we can orient our brick wall plane however we want and normal mapping would still work.

Using the previously discussed mathematics, manually calculate the tangent and bitangent vectors for the brick wall plane.

- Because a triangle is a flat shape, we only need to calculate a single tangent/bitangent pair per triangle as they will be the same for each of the triangle's vertices.
- Calculating Tangents and Bitangents

In the code above, the resulting tangent and bitangent vector should have a value of `(1,0,0)` and `(0,1,0)`, respectively. These, together with the normal `(0,0,1)`, forms an orthogonal TBN matrix. Visualized on the plane, the TBN vectors would look like the image to the right.

With tangent and bitangent vectors defined per vertex, we can start implementing proper normal mapping.

Tangent Space Normal Mapping

Create the TBN matrix in the vertex shader.

- Pass the previously calculated tangent and bitangent vectors into the vertex shader as vertex attributes.
- Sending Tangent Space Data to Vertex Shader

Create the TBN matrix in the vertex shader:

- Pass the previously calculated tangent and bitangent vectors into the vertex shader as vertex attributes.
- Sending Tangent Space Data to Vertex Shader

Now that we have a TBN matrix, there are two ways we can use it for normal mapping:

1. We take the TBN matrix that transforms any vector from tangent space to world space, give it to the fragment shader, and transform the sampled normal and tangent space to world space using the TBN matrix; the normal is then in the same space as the other lighting variables.
2. We take the inverse of the TBN matrix that transforms any vector from world space to tangent space, and use this matrix to 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.

Let's review the first case.

- The normal vector we sample from the normal map is expressed in tangent space, whereas the other lighting vectors (light and view direction) are expressed in world space.
- By passing the TBN matrix to the fragment shader, we can multiply the sampled tangent space normal with this TBN matrix to transform the normal vector to the same reference space as the other vectors. This way, all the lighting calculations (specifically the dot product) make sense.

Pass the TBN matrix to the fragment shader and use it to transform the `normal` variable from tangent space to world space.

- Transforming Normal from Tangent Space to World Space

Now, let's review the second case.

- Here, we take the inverse of the TBN matrix to transform all relevant world space vectors to the space the sampled normal vectors are in (tangent space).

Pass the inverse TBN matrix to the fragment shader and use it to transform the `lightDir` and `viewDir` variables from world space to tangent space.

- Transforming viewDir and lightDir from World Space to Tangent Space

The second approach requires matrix multiplications in the fragment shader and looks like more work. So, why would you use this apporach?

- Transforming vectors from world to tangent space has an added advantage in that we can transform all the relevant lighting vectors to tangent space in the vertex shader instead of in the fragment shader.
  - *In our case, we'll need to remove the position value from the `PointLight` uniform struct in the fragment shader and set it as a uniform `lightPos` in the vertex shader.*
  - This works because `lightPos` and `viewPos` don't update every fragment shader run. Also, for `fragPos`, we can calculate its tangent space position in the vertex shader and let the fragment interpolation do its work.
- There is effectively no need to transform a vector to tangent space in the fragment shader, while it is necessary with the first approach as sampled normal vectors are specific to each fragment shader run.

Modify your vertex and fragment shader so to use the second approach, but only transform varibles to tangent space in the vertex shader.

- You should now have a proper normal mapping for the brick wall plane. Make sure to test different rotations to make sure your implementation is working correctly.
- Second Approach Efficiency

Complex Objects

While you can manually implement tangent and bitangent vectors (like above), it is not something we do too often. Usually, you will implement it once in a custom model loader, or i n our case, use an existing model loader (which we set up in the Assimp chapter).

Assimp has a configuration bit called `aiProcess_CalcTangentSpace` that, when supplied to Assimp's `ReadFile` function, can be used to calculate the smooth tangent and bitangent vectors for each of the loaded vertices, similarly to how we did it in this chapter.

Modify the Assimp model loader implementation (from the Assimp chapter) to also load normal maps for wavefront (.obj) models.

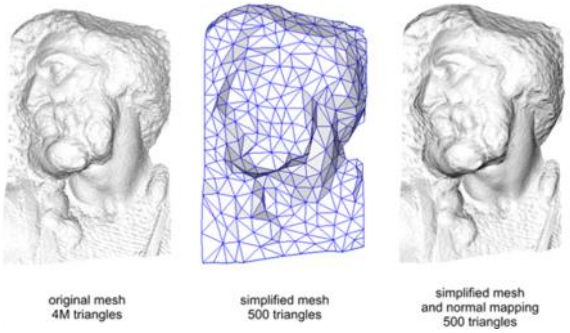
- Including Normal Maps in Assimp Implementation

Running the application on a model with specular and normal maps, using an updated model loader, gives us the result seen in the image to the right.

- As you can see, normal mapping boosts the detail of an object by an incredible amount without too much extra cost.

Using normal maps is also a great way to boost performace.

- Before normal mapping, you had to use a large number of vertices to get high detail on a mesh.
- With normal mapping, we can get the same level of detail on a mesh using a lot less vertices, as seen in the image to the right.



One Last Thing

There is one last trick left to discuss that slightly improves quality without too much extra cost.

When tangent vectors are calculated on larger meshes that share a considerable amount of vertices, the tangent vectors are generally average to give nice and smooth results.

- A problem with this approach is that the three TBN vectors could end up non-perpendicular, which means the resulting TBN matrix would no longer be orthogonal. Normal mapping would only be slightly off with a non-orthogonal TBN matrix, but it's still something we can improve.
- Using a mathematical trick called the **Gram-Schmidt process**, we can re-orthogonalize the TBN vectors such that each vector is again perpendicular to the other vectors.
  - Within the vertex shader, we would do it like this.
  - This generally improves the normal mapping results with a little extra cost. Take a look at the end of the Normal Mapping Mathematics video for a great explanation of how this process actually works.

EXERCISES

1. Try loading a wavefront (.obj) model that has a normal map into your program using our model loading implementation.

Implementing a Normal Map

Tuesday, June 7, 2022 5:47 PM

normal_map.frag

```
// Grabs the normal vector from the normal map, transforms it to range [-1, 1], and normalizes it
vec3 normal = normalize(texture(material.normal, texCoords).rgb * 2.0 - 1.0);
```

Make sure you update you VAO, remove ALL normal-related code from the vertex shader, and set the `normal` variable above as a global variable in the fragment shader.

Calculating Tangents and Bitangents

Wednesday, June 8, 2022 4:23 PM

main()

```
// Positions
// 1,2,3 is a triangle and 1,3,4 is another triangle. The code here is just for the 1,2,3 triangle.
glm::vec3 pos1(-1.0,  1.0, 0.0);
glm::vec3 pos2(-1.0, -1.0, 0.0);
glm::vec3 pos3( 1.0, -1.0, 0.0);
glm::vec3 pos4( 1.0,  1.0, 0.0);
// Texture coordinates
glm::vec2 uv1(0.0, 1.0);
glm::vec2 uv2(0.0, 0.0);
glm::vec2 uv3(1.0, 0.0);
glm::vec2 uv4(1.0, 1.0);
// Normal vector
glm::vec3 nm(0.0, 0.0, 1.0);

// Calculates triangle's edges
glm::vec3 edge1 = pos2 - pos1;
glm::vec3 edge2 = pos3 - pos1;
// Calculates triangle's delta UV coordinates
glm::vec2 deltaUV1 = uv2 - uv1;
glm::vec2 deltaUV2 = uv3 - uv1;

// Following the equation...
float f = 1.0f / (deltaUV1.x * deltaUV2.y - deltaUV2.x * deltaUV1.y);
// Calculates the tangent vector
glm::vec3 tangent1;
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);
// Calculates the bitangent vector
glm::vec3 bitangent1;
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);
// Similar procedure for calculating tangent/bitangent for plane's second triangle (pos1,pos3,pos4).
edge1 = pos3 - pos1;
edge2 = pos4 - pos1;
deltaUV1 = uv3 - uv1;
deltaUV2 = uv4 - uv1;

// Following the equation...
f = 1.0f / (deltaUV1.x * deltaUV2.y - deltaUV2.x * deltaUV1.y);
glm::vec3 tangent2;
tangent2.x = f * (deltaUV2.y * edge1.x - deltaUV1.y * edge2.x);
tangent2.y = f * (deltaUV2.y * edge1.y - deltaUV1.y * edge2.y);
tangent2.z = f * (deltaUV2.y * edge1.z - deltaUV1.y * edge2.z);
glm::vec3 bitangent2;
bitangent2.x = f * (-deltaUV2.x * edge1.x + deltaUV1.x * edge2.x);
bitangent2.y = f * (-deltaUV2.x * edge1.y + deltaUV1.x * edge2.y);
bitangent2.z = f * (-deltaUV2.x * edge1.z + deltaUV1.x * edge2.z);
```

Sending Tangent Space Data to Vertex Shader

Wednesday, June 8, 2022 4:58 PM

normal_map.vert

```
layout (location = 0) in vec2 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;

...

void main() {
   ...

   // We want to work in world space, so we multiply with the model matrix
   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);
}
```

Since we only care about the orientation of the vectors, you can (and should) multiply them by the normal matrix.

main()

```
float quadVertices[] = {
   // Positions            // Normals        // Texcoords  // Tangents                          // Bitangents
   pos1.x, pos1.y, pos1.z, nm.x, nm.y, nm.z, uv1.x, uv1.y, tangent1.x, tangent1.y, tangent1.z, bitangent1.x, bitangent1.y, bitangent1.z,
   pos2.x, pos2.y, pos2.z, nm.x, nm.y, nm.z, uv2.x, uv2.y, tangent1.x, tangent1.y, tangent1.z, bitangent1.x, bitangent1.y, bitangent1.z,
   pos3.x, pos3.y, pos3.z, nm.x, nm.y, nm.z, uv3.x, uv3.y, tangent1.x, tangent1.y, tangent1.z, bitangent1.x, bitangent1.y, bitangent1.z,

   pos1.x, pos1.y, pos1.z, nm.x, nm.y, nm.z, uv1.x, uv1.y, tangent2.x, tangent2.y, tangent2.z, bitangent2.x, bitangent2.y, bitangent2.z,
   pos3.x, pos3.y, pos3.z, nm.x, nm.y, nm.z, uv3.x, uv3.y, tangent2.x, tangent2.y, tangent2.z, bitangent2.x, bitangent2.y, bitangent2.z,
   pos4.x, pos4.y, pos4.z, nm.x, nm.y, nm.z, uv4.x, uv4.y, tangent2.x, tangent2.y, tangent2.z, bitangent2.x, bitangent2.y, bitangent2.z
};

// Brick Wall VAO
[...]
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 14 * sizeof(float), (void*)0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 14 * sizeof(float), (void*)(sizeof(float) * 3));
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 14 * sizeof(float), (void*)(sizeof(float) * 6));
glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, 14 * sizeof(float), (void*)(sizeof(float) * 8));
glVertexAttribPointer(4, 3, GL_FLOAT, GL_FALSE, 14 * sizeof(float), (void*)(sizeof(float) * 11));

glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
glEnableVertexAttribArray(2);
glEnableVertexAttribArray(3);
glEnableVertexAttribArray(4);
```

NOTE: Technically you don't have to send the bitangent data to the vertex shader; since all 3 TBN vectors are perpendicular to each other, you can calculate the bitangent in the vertex shader by taking the cross product of the T and N vectors.

Transforming Normal from Tangent Space to World Space

Wednesday, June 8, 2022 5:49 PM

normal_map.vert

```
...
out mat3 tbn;

void main() {
   ...
   tbn = mat3(t, b, n);
}
```

normal_map.frag

```
void main() {
   // The only thing we changed was multiplying by the TBN matrix before normalizing
   normal = texture(material.normal, texCoords).rgb;
   normal = normal * 2.0 - 1.0;
   normal = normalize(tbn * normal);
   ...
}
```

Because the resulting normal is now in world space, there is no need to change any other fragment shader code as the lighting code assumes the normal vector to be in world space already.

Transforming viewDir and lightDir from World Space to Tangent Space

Wednesday, June 8, 2022 6:02 PM

normal_map.vert

```
...
void main() {
   ...
   tbn = transpose(mat3(t, b, n));
}
```

normal_map.frag

```
void main() {
   // Make sure to change the normal vector calculations back
   normal = texture(material.normal, texCoords).rgb;
   normal = normalize(normal * 2.0 - 1.0);

   // View direction in tangent space
   vec3 viewDir = tbn * normalize(viewPos - fragPos);
   ...
}

vec3 CalcPointLight(PointLight light, vec3 viewDir) {
   ...
   // Light direction in tangent space
   vec3 lightDir = tbn * normalize(light.position - fragPos);
   ...
}
```

Second Approach Efficiency

Wednesday, June 8, 2022 6:25 PM

normal_map.vert

```
...
out vec2 texCoords;
out vec3 tangentLightPos;
out vec3 tangentViewPos;
out vec3 tangentFragPos;
...
uniform vec3 lightPos;
uniform vec3 viewPos;

void main() {
   ...
   tangentLightPos = tbn * lightPos;
   tangentViewPos = tbn * viewPos;
   tangentFragPos = tbn * vec3(model * vec4(aPos, 0.0, 1.0));;
}
```

normal_map.frag

```
...
struct PointLight {
   //vec3 position // Moved to lightPos uniform in vertex shader
   vec3 ambient;
   vec3 diffuse;
   vec3 specular;
};
uniform PointLight pointLight;

in vec2 texCoords;
in vec3 tangentLightPos;
in vec3 tangentViewPos;
in vec3 tangentFragPos;

...

void main() {
   // Remember, the normal is already in tangent space because normal map texture coordinates are in tangent space
   normal = texture(material.normal, texCoords).rgb;
   normal = normalize(normal * 2.0 - 1.0);

   vec3 viewDir = normalize(tangentViewPos - tangentFragPos);
   ...
}

vec3 CalcPointLight(PointLight light, vec3 viewDir) {
   ...
   vec3 lightDir = normalize(tangentLightPos - tangentFragPos);
   ...
}
```

All we did was swap every instance of `fragPos`, `lightPos`, and `viewPos` with `tangentFragPos`, `tangentLightPos`, and `tangentViewPos`, respectively.

Including Normal Maps in Assimp Implementation

Wednesday, June 8, 2022 7:08 PM

Mesh.h

```
struct Vertex {
   glm::vec3 Position;
   glm::vec3 Normal;
   glm::vec2 TexCoords;
   glm::vec3 Tangent; // Don't forget to add this!
};
```

Model.cpp

```
void Model::loadModel(std::string path) {
    ...
    // Creates the assimp scene object (with calculated tangents and bitangents)
    const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate | aiProcess_FlipUVs | aiProcess_CalcTangentSpace);
    ...
}

Mesh Model::processMesh(aiMesh* mesh, const aiScene* scene) {
    ...

    // 1. Store the vertex data from the assimp mesh
    for (int i = 0; i < mesh->mNumVertices; ++i) {
        ...
        vertex.Tangent = glm::vec3(mesh->mTangents[i].x, mesh->mTangents[i].y, mesh->mTangents[i].z);
        ...
    }

    ...

    // 3. Store the materials (textures) from the assimp mesh
    if (mesh->mMaterialIndex >= 0) {
        ...
        // Pushes all the normal maps into the textures vector
        std::vector<Texture> normalMaps = loadMaterialTextures(material, aiTextureType_HEIGHT, "texture_normal");
        textures.insert(textures.end(), normalMaps.begin(), normalMaps.end());
    }

    ...
}
```

Gram-Schmidt Process

Wednesday, June 8, 2022 9:40 PM

normal_map.vert

```
void main() {
   ...
   // Gram-Schmidt Process
   vec3 t = normalize(vec3(model * vec4(aTangent, 0.0)));
   vec3 n = normalize(vec3(model * vec4(aNormal, 0.0)));
   // Re-orthogonalizes t with respect to n
   t = normalize(t - dot(t, n) * n);
   // Calculates the perpendicular vector b using the cross product of n and t
   vec3 b = cross(n, t);
   mat3 tbn = mat3(t, b, n);
   ...
}
```