Week4

Hooman Salamat

Normal Mapping

Objectives:

1. To understand why we need normal mapping.

2. To discover how normal maps are stored.

3. To learn how normal maps can be created.

4. To find out the coordinate system the normal vectors in normal maps are stored relative to and how it relates to the object space coordinate system of a 3D triangle.

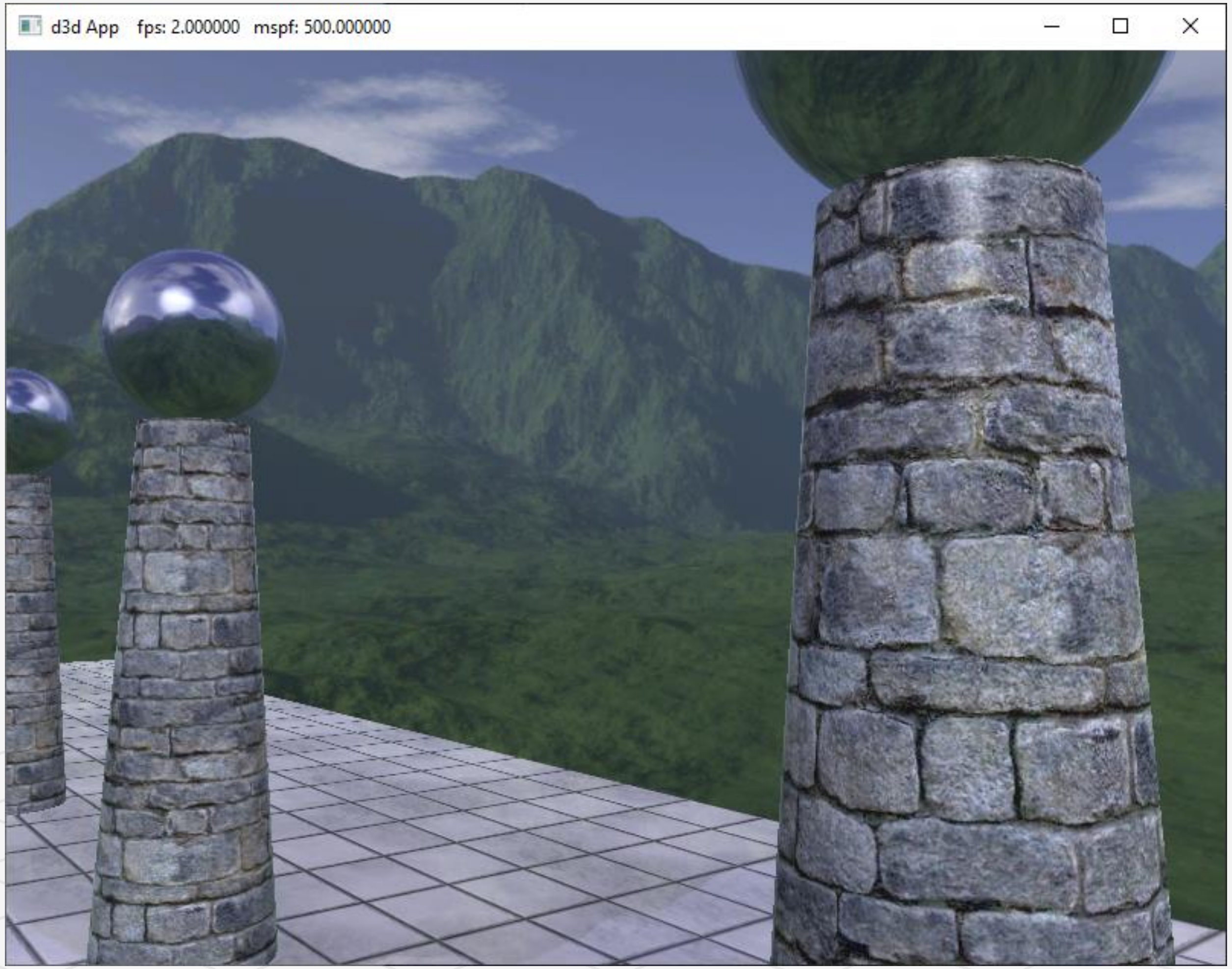
5. To learn how to implement normal mapping in a vertex and pixel shader.

MOTIVATION

The specular highlights on the cone shaped columns do not look right—they look unnaturally smooth compared to the bumpiness of the brick texture.

Because the underlying mesh geometry is smooth, and we have merely applied the image of bumpy bricks over the smooth cylindrical surface.

The lighting calculations are performed based on the mesh geometry (in particular, the interpolated vertex normals), and not the texture image.



NORMAL MAPS

A *normal map* is a texture, but instead of storing RGB data at each texel, we store a compressed *x*-, *y*-, and *z*-coordinates in the red, green, and blue components, respectively. These coordinates define a normal vector;

A normal map stores a normal vector at each pixel.

Normals stored in a normal map relative to a texture space coordinate system defined by the vectors **T** (*x*-axis), **B** (*y*-axis), and **N** (*z*-axis).

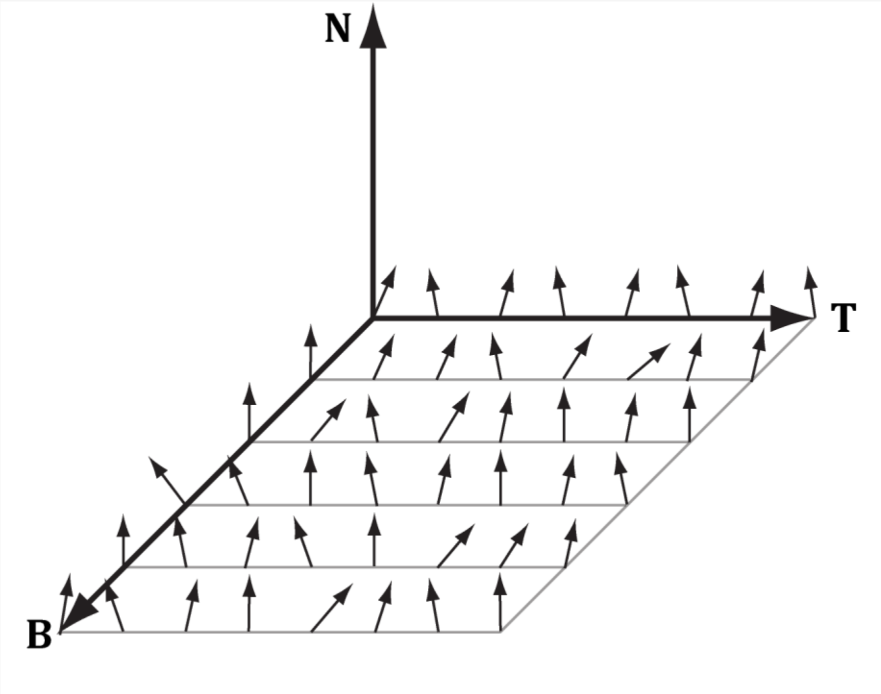
The **T** vector runs right horizontally to the texture image;

The **B** vector runs down vertically to the texture image;

N is orthogonal to the texture plane.

The **T**, **B**, and **N** vectors are commonly referred to as the *tangent*, *binormal* (or *bitangent*), and *normal* vectors, respectively.

For illustration, we will assume a 24-bit image format, which reserves a byte [0-255] for each color component



Compression Texture Coordinates

Normal vectors range between -1 and 1.

How do we compress a unit vector [-1,1] into this 24-bit or 32 bit format [0-255][0-255][0-255][0-255]?

If we shift and scale this range to [0, 1] and multiply by 255 and truncate the decimal, the result will be an integer in the range 0-255.

if x is a coordinate in the range $[-1, 1]$, then the integer part of $f(x)$

$$f(x) = (0.5x + 0.5) * 255$$

With normal vectors transformed to an RGB color component like this, we can store a per-pixel normal derived from the shape of a surface onto a 2D texture.

How to reverse the compression process; that is, given a compressed texture coordinate in the range 0-255, how can we recover its true value in the interval $[-1, 1]$? Invert the function f.

$$f^{-1}(x) = \frac{2x}{255} - 1$$

We will not have to do the compression process ourselves, as we could use a Photoshop plug-in to convert images to normal maps.

However, when we sample a normal map in a pixel shader, we will have to do part of the inverse process to un-compress it. When we sample a normal map in a shader like this:

```
float3 normalT = gNormalMap.Sample(gTriLinearSam, pin.Tex);
```

The color vector normalT will have normalized components (r, g, b) such that $0 \le r, g, b \le 1$.

```
// Uncompress each component from [0,1] to [-1,1].

normalT = 2.0f * normalT - 1.0f;
```

The Photoshop plug-in is available at https://developer.nvidia.com/nvidia-texture-tools-adobe-photoshop

There are other tools available for generating normal maps such as http://www.crazybump.com/ and

http://shadermap.com/home/

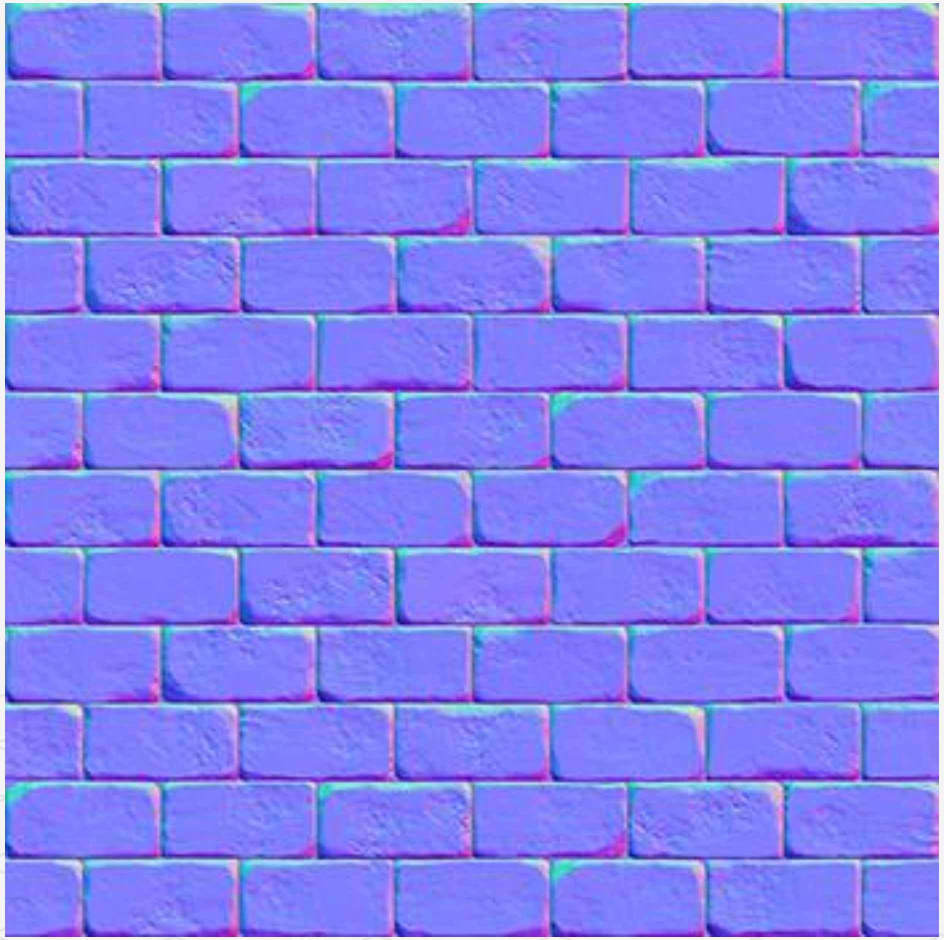
Normal Map Example

This (and almost all normal maps you find online) will have a blue-ish tint.

This is because all the normals are all closely pointing outwards towards the z-axis which is (0,0,1): a blue-ish color.

The slight deviations in color represent normal vectors that are slightly offset from the general z direction, giving a sense of depth to the texture.

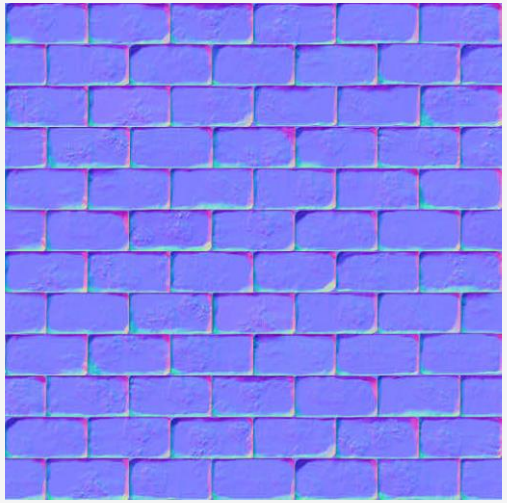
For example, you can see that at the top of each brick the color tends to get more green which makes sense as the top side of a brick would have normals pointing more in the positive y direction (0,1,0) which happens to be the color green!

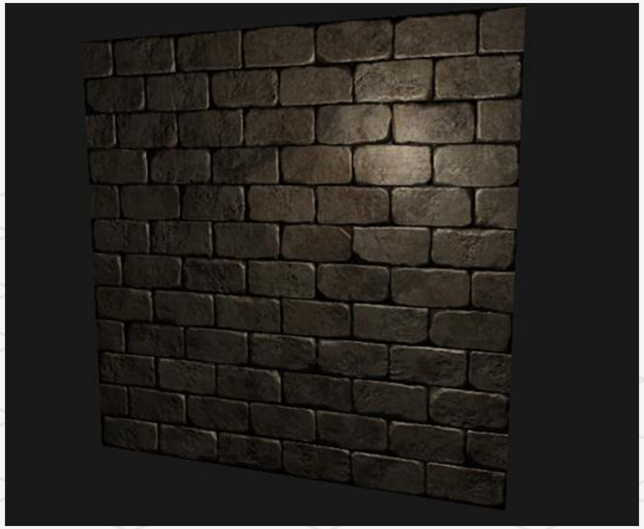


Normal Mapping

With a simple plane looking at the negative z-axis, we can take the diffuse texture and the normal map to render the image.

Load both textures, bind them to the proper texture units and render a plane for lighting in the pixel shader.





How to use normal map in the pixel shader

```
//This is pixel shader before using normal map
// Dynamically look up the texture in the array.
diffuseAlbedo *=
gDiffuseMap[diffuseTexIndex].Sample(gsamAnisotropicWrap,
pin.TexC);

// Interpolating normal can unnormalize it, so renormalize it.
pin.NormalW = normalize(pin.NormalW);

// Vector from point being lit to eye.
float3 toEyeW = normalize(gEyePosW - pin.PosW);

// Light terms.
float4 ambient = gAmbientLight*diffuseAlbedo;

const float shininess = 1.0f - roughness;
Material mat = { diffuseAlbedo, fresnelR0, shininess };
float3 shadowFactor = 1.0f;
float4 directLight = ComputeLighting(gLights, mat, pin.PosW,
pin.NormalW, toEyeW, shadowFactor);

float4 litColor = ambient + directLight;
```

```
// Dynamically look up the texture in the array.
diffuseAlbedo *= gTextureMaps[diffuseMapIndex].Sample(gsamAnisotropicWrap, pin.TexC);

// Interpolating normal can unnormalize it, so renormalize it.
pin.NormalW = normalize(pin.NormalW);

float4 normalMapSample = gTextureMaps[normalMapIndex].Sample(gsamAnisotropicWrap,
pin.TexC);
// Uncompress each component from [0,1] to [-1,1].
float3 bumpedNormalW = 2.0f*normalMapSample.rgb - 1.0f;


// Vector from point being lit to eye.
float3 toEyeW = normalize(gEyePosW - pin.PosW);

// Light terms.
float4 ambient = gAmbientLight*diffuseAlbedo;

const float shininess = (1.0f - roughness) * normalMapSample.a;
Material mat = { diffuseAlbedo, fresnelR0, shininess };
float3 shadowFactor = 1.0f;
float4 directLight = ComputeLighting(gLights, mat, pin.PosW,
    bumpedNormalW, toEyeW, shadowFactor);

float4 litColor = ambient + directLight;
```

The lighting doesn't look right!

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

The normal map we used had normal vectors that all roughly pointed in the negative z direction.

This worked because the plane's surface normal was also pointing in the negative z direction.

However, what would happen if we used the same normal map on a plane laying on the ground with a surface normal vector pointing in the positive y direction?

This happens because the sampled normals of this plane still point roughly in the negative z direction even though they should point somewhat in the positive y direction of the surface normal.

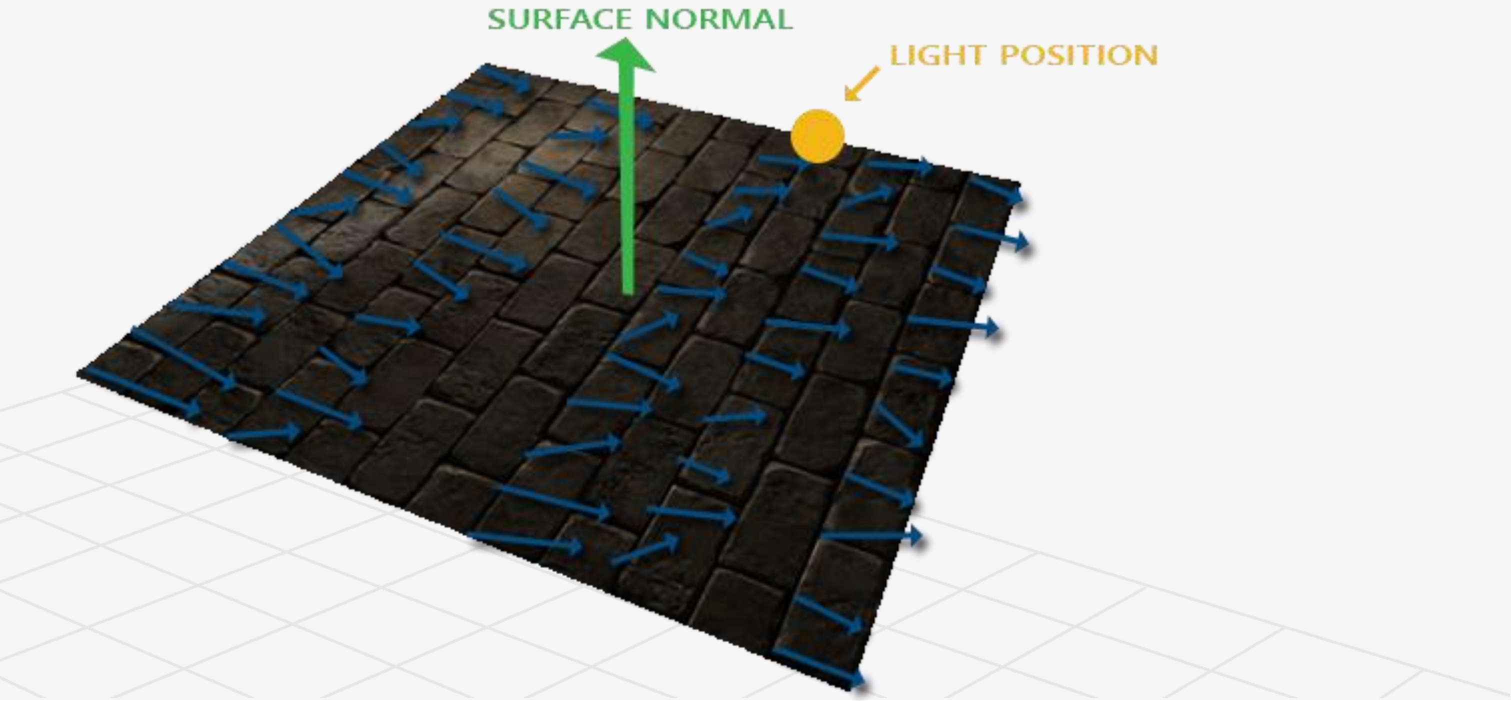
As a result the lighting thinks the surface's normals are the same as before when the surface was still looking in the negative z direction.

The image below shows what the sampled normals approximately look like on this surface.

You can see that all the normals roughly point in the negative z direction while they should be pointing alongside the surface normal in the positive y direction.

A possible solution to this problem is to define a normal map for each possible direction of a surface. In the case of a cube we would need 6 normal maps, but with advanced models that can have more than hundreds of possible surface directions this becomes an infeasible approach.

A different solution works by doing lighting in a different coordinate space: a coordinate space where the normal map vectors always point roughly in the negative z direction; all other lighting vectors are then transformed relative to this negative z direction. This way we can always use the same normal map, regardless of orientation. This coordinate space is called tangent space.



TEXTURE/TANGENT SPACE

Tangent space is a space that's local to the surface of a triangle: the normals are relative to the local reference frame of the individual triangles.

Think of it as the local space of the normal map's vectors; they're all defined pointing in the negative 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.

Consider a 3D texture mapped triangle.

The relationship between the texture space of a triangle and the object space.

The 3D tangent vector **T** aims in the *u*-axis direction of the texturing coordinate system.

The 3D tangent vector **B** aims in the *v*-axis direction of the texturing coordinate system.

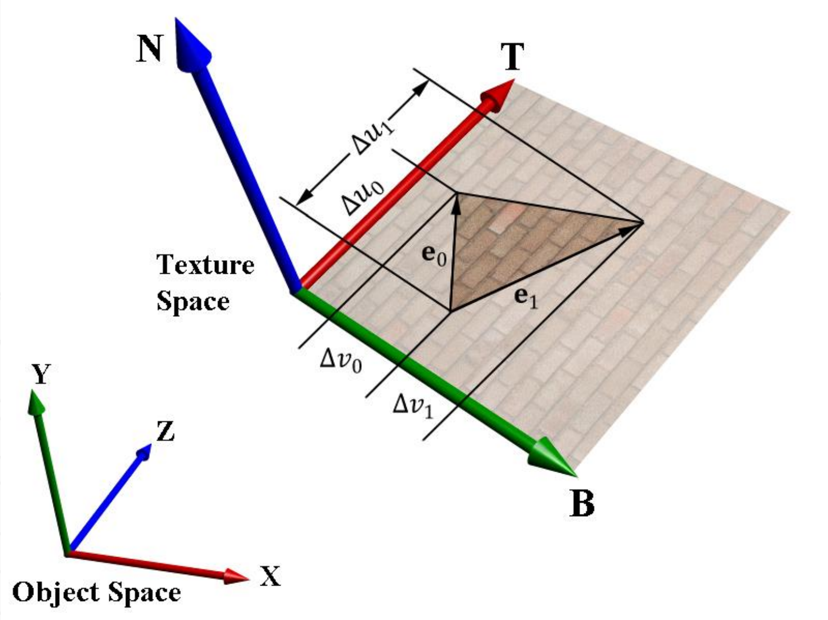
Figure shows how the texture space axes relate to the 3D triangle: they are tangent to the triangle and lie in the plane of the triangle.

Let $\mathbf{v}_0$, $\mathbf{v}_1$, and $\mathbf{v}_2$ define the three vertices of a 3D triangle with corresponding texture coordinates (u_0, v_0), (u_1, v_1), and (u_2, v_2) that define a triangle in the texture plane relative to the texture space axes (i.e., **T** and **B**).

Let $\mathbf{e}_0 = \mathbf{v}_1 - \mathbf{v}_0$ and $\mathbf{e}_1 = \mathbf{v}_2 - \mathbf{v}_0$ be two edge vectors of the 3D triangle with corresponding texture triangle edge vectors:

$(\Delta u_0, \Delta v_0) = (u_1 - u_0, v_1 - v_0)$ and $(\Delta u_1, \Delta v_1) = (u_2 - u_0, v_2 - v_0)$

$e_0 = \Delta u_0 T + \Delta v_0 B$ and $e_1 = \Delta u_1 T + \Delta v_1 B$



TEXTURE/TANGENT SPACE

Representing the vectors with coordinates relative to object space, we get the matrix equation:

$$\begin{bmatrix} e_{0,x} & e_{0,y} & e_{0,z} \\ e_{1,x} & e_{1,y} & e_{1,z} \end{bmatrix} = \begin{bmatrix} \Delta u_0 & \Delta v_0 \\ \Delta u_1 & \Delta v_1 \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

Note that we know the object space coordinates of the triangle vertices; therefore we know the object space coordinates of the edge vectors:

$$\begin{bmatrix} e_{0,x} & e_{0,y} & e_{0,z} \\ e_{1,x} & e_{1,y} & e_{1,z} \end{bmatrix}$$

We know the texture coordinates:

$$\begin{bmatrix} \Delta u_0 & \Delta v_0 \\ \Delta u_1 & \Delta v_1 \end{bmatrix}$$

Solving for the **T** and **B** object space coordinates we get:

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \begin{bmatrix} \Delta u_0 & \Delta v_0 \\ \Delta u_1 & \Delta v_1 \end{bmatrix}^{-1} \begin{bmatrix} e_{0,x} & e_{0,y} & e_{0,z} \\ e_{1,x} & e_{1,y} & e_{1,z} \end{bmatrix}$$

$$= \frac{1}{\Delta u_0 \Delta v_1 - \Delta v_0 \Delta u_1} \begin{bmatrix} \Delta v_1 & -\Delta v_0 \\ -\Delta u_1 & \Delta u_0 \end{bmatrix} \begin{bmatrix} e_{0,x} & e_{0,y} & e_{0,z} \\ e_{1,x} & e_{1,y} & e_{1,z} \end{bmatrix}$$

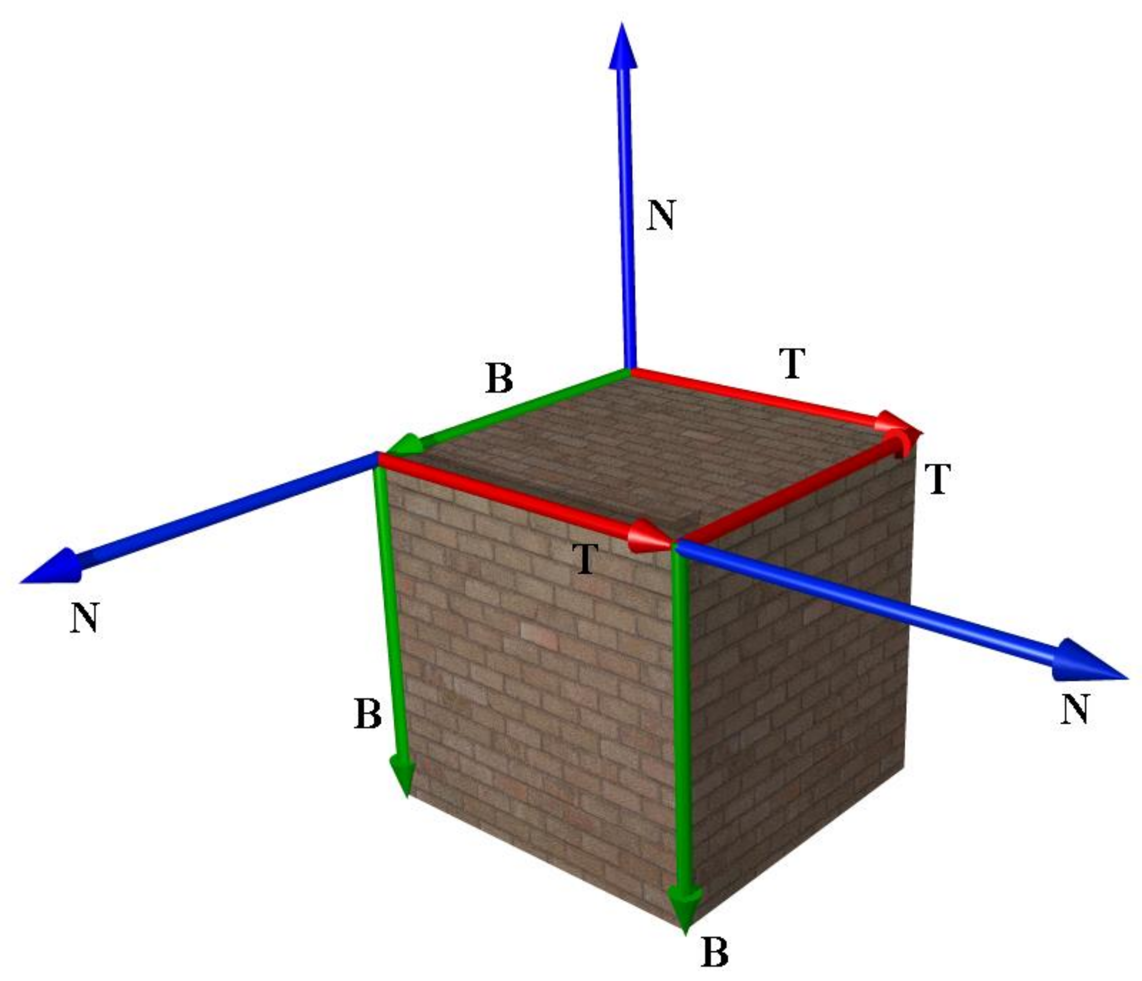
TEXTURE/TANGENT SPACE

The texture coordinates of the triangle are relative to the texture space coordinate system.

Incorporating the triangle face normal **N**, we obtain a 3D *TBN-basis* in the plane of the triangle that we call *texture space* or *tangent space*.

Note that the tangent space generally varies from triangle-to-triangle.

The normal vectors in a normal map are defined relative to the texture space. But our lights are defined in world space. In order to do lighting, the normal vectors and lights need to be in the same space. So our first step is to relate the tangent space coordinate system with the object space coordinate system the triangle vertices are relative to.



VERTEX TANGENT SPACE

If we use this texture space for normal mapping, we will get a triangulated appearance since the tangent space is constant over the face of the triangle.

We specify tangent vectors per vertex, and we do the same averaging trick that we did with vertex normals to approximate a smooth surface:

1. The tangent vector **T** for an arbitrary vertex **v** in a mesh is found by averaging the tangent vectors of every triangle in the mesh that shares the vertex **v**.

2. The bitangent vector **B** for an arbitrary vertex **v** in a mesh is found by averaging the bitangent vectors of every triangle in the mesh that shares the vertex **v**.

After averaging, the TBN-bases will generally need to be orthonormalized, so that the vectors are mutually orthogonal and of unit length.

This is usually done using the Gram-Schmidt procedure.

In our system, we will not store the bitangent vector **B** directly in memory. Instead, we will compute **B** = **N** × **T** when we need **B**, where **N** is the usual averaged vertex normal. Hence, our vertex structure looks like this:

```
struct VertexIn

{

float3 PosL    : POSITION;

float3 NormalL : NORMAL;

float2 TexC    : TEXCOORD;

float3 TangentU : TANGENT;

};
```

GeometryGenerator computes the tangent vector **T** corresponding to the *u*-axis of the texture space. The object space coordinates of the tangent vector **T** is easily specified at each vertex for box and grid meshes

TRANSFORMING BETWEEN TANGENT SPACE AND OBJECT SPACE

We have the coordinate of the TBN-basis relative to the object space coordinate system, we can transform coordinates from tangent space to object space with the matrix:

$$\mathbf{M}_{object} = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{bmatrix}$$

An orthogonal matrix TBN is a square matrix whose columns and rows are orthogonal unit vectors (i.e., orthonormal vectors), i.e. $MM^T=I$. Therefore, its inverse is its transpose. Therefore, the change of coordinate matrix from object space to tangent space is:

$$\mathbf{M}_{tangent} = \mathbf{M}_{object}^{-1} = \mathbf{M}_{object}^{T} = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$

In our shader program, we will actually want to transform the normal vector from tangent space to world space for lighting. One way would be to transform the normal from tangent space to object space first, and then use the world matrix to transform from object space to world space:

$$n_{world} = (n_{tangent} M_{object}) M_{world}$$

$$n_{world} = n_{tangent} (M_{object} M_{world})$$

So to go from tangent space directly to world space, we just have to describe the tangent basis in world coordinates, which can be done by transforming the TBN-basis from object space coordinates to world space coordinates.

$$\mathbf{M}_{object}\mathbf{M}_{world} = \begin{bmatrix} \leftarrow \mathbf{T} \rightarrow \\ \leftarrow \mathbf{B} \rightarrow \\ \leftarrow \mathbf{N} \rightarrow \end{bmatrix} \mathbf{M}_{world} = \begin{bmatrix} \leftarrow \mathbf{T}' \rightarrow \\ \leftarrow \mathbf{B}' \rightarrow \\ \leftarrow \mathbf{N}' \rightarrow \end{bmatrix} = \begin{bmatrix} T'_x & T'_y & T'_z \\ B'_x & B'_y & B'_z \\ N'_x & N'_y & N'_z \end{bmatrix}$$

where $\mathbf{T}' = \mathbf{T} \cdot \mathbf{M}_{world}$, $\mathbf{B}' = \mathbf{B} \cdot \mathbf{M}_{world}$, and $\mathbf{N}' = \mathbf{N} \cdot \mathbf{M}_{world}$

NORMAL MAPPING SHADER CODE

1. Create the desired normal maps from some utility program and store them in an image file. Create 2D textures from these files when the program is initialized.

2. For each triangle, compute the tangent vector **T**. Obtain a per-vertex tangent vector for each vertex **v** in a mesh by averaging the tangent vectors of every triangle in the mesh that shares the vertex **v**. (we use simply geometry and are able to specify the tangent vectors directly, but this averaging process would need to be done if using arbitrary triangle meshes made in a 3D modeling program.)

3. In the vertex shader, transform the vertex normal and tangent vector to world space and output the results to the pixel shader.

```
vout.NormalW = mul(vin.NormalL, (float3x3)gWorld);
vout.TangentW = mul(vin.TangentU, (float3x3)gWorld);
```

4. Using the interpolated tangent vector and normal vector, we build the TBN-basis at each pixel point on the surface of the triangle. We use this basis to transform the sampled normal vector from the normal map from tangent space to the world space. We then have a world space normal vector from the normal map to use for our usual lighting calculations.

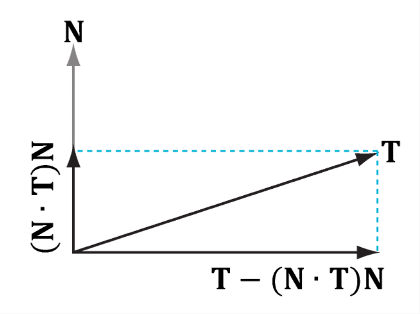
```
//---------------------------------------------------------------------------
-
// Transforms a normal map sample to world space.
//---------------------------------------------------------------------------
--------
float3 NormalSampleToWorldSpace(float3 normalMapSample, float3 unitNormalW,
float3 tangentW)
{
// Uncompress each component from [0,1] to [-1,1].
float3 normalT = 2.0f*normalMapSample - 1.0f;

// Build orthonormal basis➔look at the figure
float3 N = unitNormalW;
float3 T = normalize(tangentW - dot(tangentW, N)*N);
float3 B = cross(N, T);

float3x3 TBN = float3x3(T, B, N);

// Transform from tangent space to world space.
float3 bumpedNormalW = mul(normalT, TBN);

return bumpedNormalW;
}
```



This function is used like this in the pixel shader:

```
float3 normalMapSample = gNormalMap.Sample(samLinear,pin.Tex).rgb;

float3 bumpedNormalW = NormalSampleToWorldSpace(normalMapSample,pin.NormalW,
pin.TangentW);
```

TBN

After the interpolation, the tangent vector and normal vector may not be orthonormal.

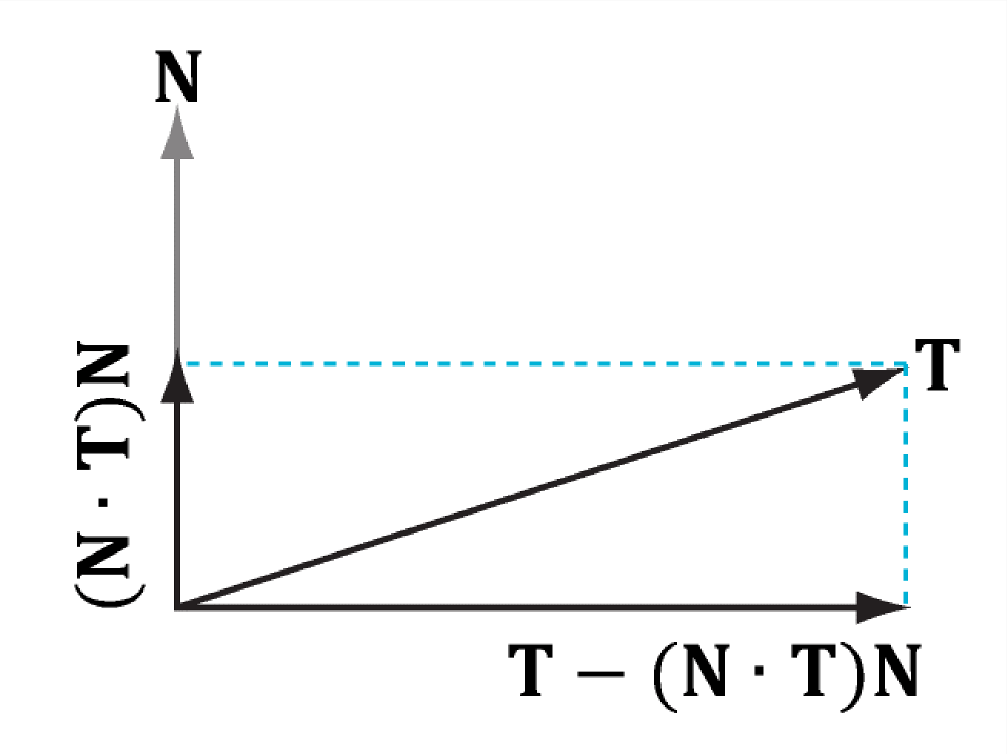
This code makes sure **T** is orthonormal to **N** by subtracting off any component of **T** along the direction **N.**

Note that there is the assumption that unitNormalW is normalized.

```
// Build orthonormal basis➔look at the figure
float3 N = unitNormalW;
float3 T = normalize(tangentW - dot(tangentW, N)*N);
float3 B = cross(N, T);
```

Since $||\mathbf{N}|| = 1$, $\text{proj}_{\mathbf{N}}(\mathbf{T}) = (\mathbf{T}\cdot\mathbf{N})\mathbf{N}$.

The vector $\mathbf{T}\text{-proj}_{\mathbf{N}}(\mathbf{T})$ is the portion of **T** orthogonal to **N**.



bumpedNormalW vector

Observe that the “bumped normal” vector is used in the light calculation, but also in the reflection calculation for modeling reflections from the environment map.

```
float4 PS(VertexOut pin) : SV_Target
{
// Fetch the material data.
MaterialData matData = gMaterialData[gMaterialIndex];
float4 diffuseAlbedo = matData.DiffuseAlbedo;
float3 fresnelR0 = matData.FresnelR0;
float  roughness = matData.Roughness;
uint diffuseMapIndex = matData.DiffuseMapIndex;
uint normalMapIndex = matData.NormalMapIndex;

// Interpolating normal can unnormalize it, so renormalize it.
pin.NormalW = normalize(pin.NormalW);

float4 normalMapSample =
gTextureMaps[normalMapIndex].Sample(gsamAnisotropicWrap, pin.TexC);
float3 bumpedNormalW =
NormalSampleToWorldSpace(normalMapSample.rgb, pin.NormalW,
pin.TangentW);

// Dynamically look up the texture in the array.
diffuseAlbedo *=
gTextureMaps[diffuseMapIndex].Sample(gsamAnisotropicWrap, pin.TexC);

// Vector from point being lit to eye.
float3 toEyeW = normalize(gEyePosW - pin.PosW);

// Light terms.
float4 ambient = gAmbientLight*diffuseAlbedo;

const float shininess = (1.0f - roughness) * normalMapSample.a;
Material mat = { diffuseAlbedo, fresnelR0, shininess };
float3 shadowFactor = 1.0f;
float4 directLight = ComputeLighting(gLights, mat, pin.PosW,
        bumpedNormalW, toEyeW, shadowFactor);

float4 litColor = ambient + directLight;

// Add in specular reflections.
float3 r = reflect(-toEyeW, bumpedNormalW);
float4 reflectionColor = gCubeMap.Sample(gsamLinearWrap, r);
float3 fresnelFactor = SchlickFresnel(fresnelR0, bumpedNormalW, r);
litColor.rgb += shininess * fresnelFactor * reflectionColor.rgb;

// Common convention to take alpha from diffuse albedo.
litColor.a = diffuseAlbedo.a;

return litColor;
}
```

Shininess

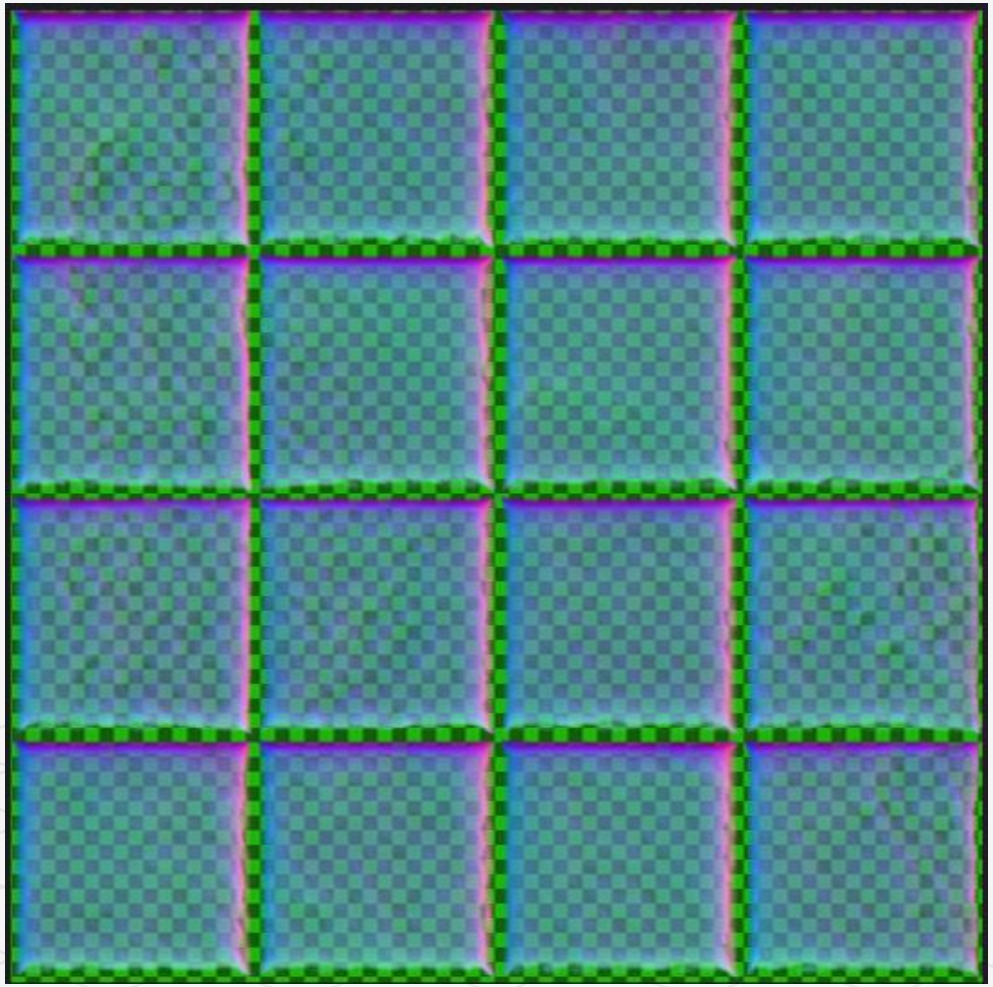
In addition, in the alpha channel of the normal map we store a shininess mask, which controls the shininess at a per-pixel level.

The alpha channel of the *tile_nmap.dds* image under "Textures" folders.

The alpha channel denotes the shininess of the surface.

White values indicate a shininess value of 1.0 and black values indicate a shininess value of 0.0.

This gives us per-pixel control of the shininess material property.



SUMMARY

The strategy of normal mapping is to texture our polygons with normal maps.

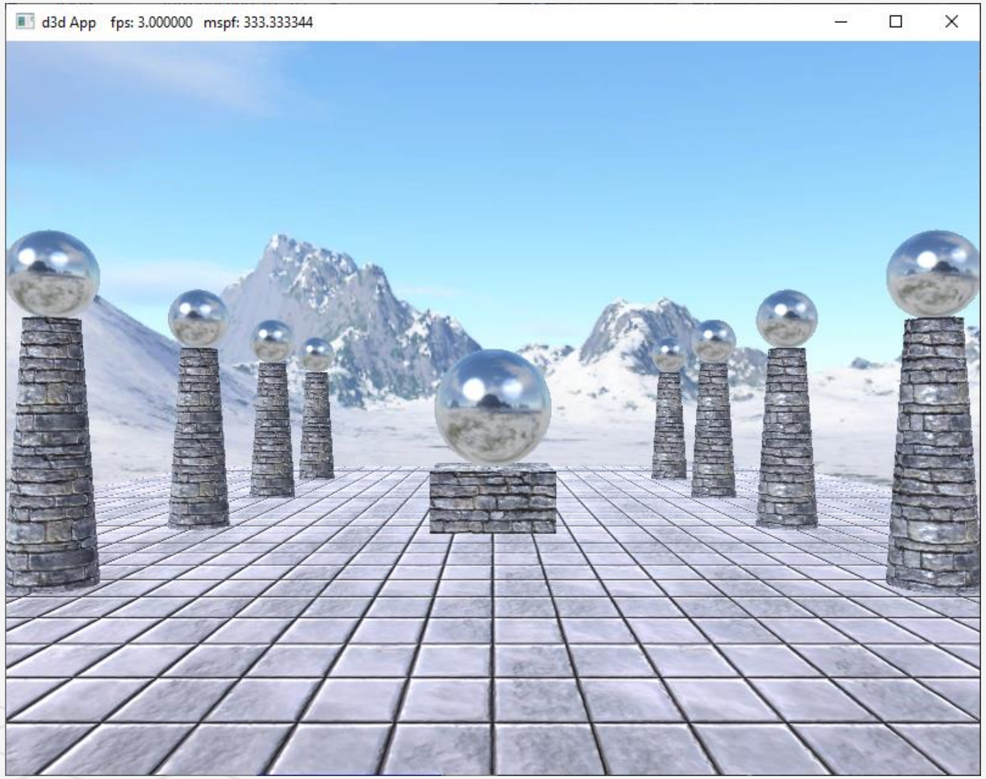
We then have per-pixel normals, which capture the fine details of a surface like bumps, scratches, and crevices.

We then use these per-pixel normals from the normal map in our lighting calculations, instead of the interpolated vertex normal.

The coordinates of the normals in a normal map are relative to the texture space coordinate system.

Consequently, to do lighting calculations, we need to transform the normal from the texture space to the world space so that the lights and normal are in the same coordinate system.

The TBN-bases built at each vertex facilitates the transformation from texture space to world space.



Input Handling

Objectives

Examine SFML events and explore their purpose as input

Assess real-time input and evaluate its difference from events

Analyze and reproduce a command-based communication system to deliver events

Explore how to dynamically bind keys at runtime

Win32 Keyboard Input Model

https://docs.microsoft.com/en-us/windows/win32/inputdev/about-keyboard-input

Polling Events

Events are objects that are triggered when something happens

- E.g., user input

Behind the scenes, the OS reports an event to the application

- SFML processes such a report
- Converts it into a corresponding SFML event type

Polling Events (cont'd.)

Specifically, we extract events using the `sf::Window::pollEvent()` function

It's signature is:

```
bool sf::Window::pollEvent(sf::Event& event);
```

Polling Events (cont'd.)

Generally we want to poll an event with an event parameter as well as a bool that will tell us to keep polling the event or not

- If there are no more of that event type to poll

Events Thus Far

In the examples up to now, we’ve handled events in SFML thus:

```
sf::Event event;
while (window.pollEvent(event))
{
    // Handle the event
}
```

Events

We can group events to four different categories:

window, **joystick**, **keyboard** and **mouse**

The next few slides outline these events

Window Events

Window events concern windows directly

`sf::Event::Closed`

- Occurs when the user requests that the window be closed
- Pressing the [X] or Alt-F4 for example
- No data associated with this event

Window Events (cont'd.)

`sf::Event::Resized`

- Occurs when the window is resized
- User drags on edges to manually resize it
- Window must be enabled to resize
- Data type is `sf::Event::SizeEvent` that is accessed through `event.size`

Window Events (cont'd.)

`sf::Event::LostFocus`

`Sf::Event::GainedFocus`

Window is active or inactive (clicked away from)

No extra data for event

Joystick Events

Whenever a joystick or gamepad changes its state

Each input device has an ID number

`sf::Event::JoystickButtonPressed`

`sf::Event::JoystickButtonReleased`

Data structure associated is `sf::Event::JoystickButtonEvent` with the member `event.joystickButton`

Joystick Events (cont'd.)

`sf::Event::JoystickMoved`

Triggered when analog stick or D-pad moves

Data is `sf::Event::JoystickMoveEvent` and accessible through member `event.joystickMove`

Joystick Events (cont'd.)

`sf::Event::JoystickConnected`

`Sf::Event::JoystickDisconnected`

Data is `sf::Event::JoystickConnectEvent` and accessible through member `event.joystickConnect`

Keyboard Events

Generates event as the primary input device for computers

`sf::Event::KeyPressed`

- Data structure associated is `sf::Event::KeyEvent` with the member `event.key.code`
- `event.key.control` are Booleans that state whether a modifier is pressed
- Key repetition can be deactivated using `sf::Window::setKeyRepeatEnabled()`

Keyboard Events (cont'd.)

`sf::Event::KeyReleased`

- Counterpart to `KeyPressed`
- Similar in function

`sf:Event:TextEntered`

- Designed for receiving formatted text from the user
- Data is `sf::Event::TextEvent` and accessible through `event.text`

Mouse Events (cont'd.)

Events generated when the state of the cursor, mouse buttons or mouse wheel changes

`sf::Event::MouseEntered`

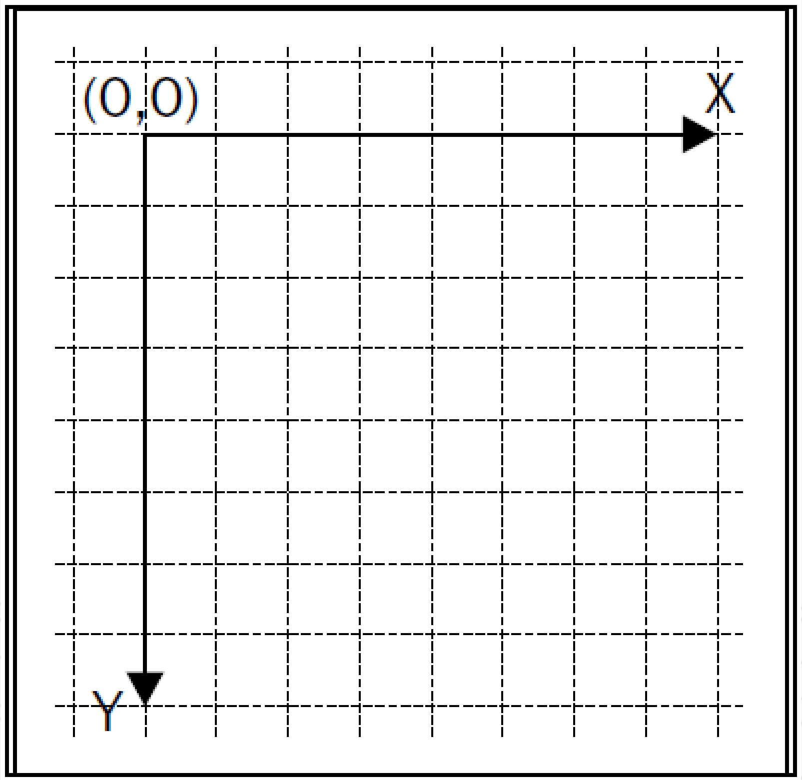
`sf::Event::MouseLeft`

`Sf::Event::MouseMoved`

- Data structure for `MouseMoved` is `sf::MouseMoveEvent` and can be accessed via `event.mouseMove`

Mouse Orientation

As most platforms, coordinates measures in window pixels



Mouse Events (cont'd.)

`sf::Event::MouseButtonPressed`

`sf::Event::MouseButtonReleased`

Data structure is `sf::MouseButtonEvent` and can be accessed via `event.mouseButton` member

`sf::Event::MouseWheelMoved`

Data structure is `sf::MouseWheelEvent` and can be accessed via `event.mouseWheel` member

Handling Input

```
void Game::handlePlayerInput(sf::Keyboard::Key key, bool isPressed)
{
    if (key == sf::Keyboard::W)
        mIsMovingUp = isPressed;
    else if (key == sf::Keyboard::S)
        mIsMovingDown = isPressed;
    else if (key == sf::Keyboard::A)
        mIsMovingLeft = isPressed;
    else if (key == sf::Keyboard::D)
        mIsMovingRight = isPressed;
}
```

Handling Input (cont'd.)

```
void Game::update()
{
   sf::Vector2f movement(0.f, 0.f);
   if (mIsMovingUp)
       movement.y -= 1.f;
   if (mIsMovingDown)
       movement.y += 1.f;
   if (mIsMovingLeft)
       movement.x -= 1.f;
   if (mIsMovingRight)
```

Combining Into Update

```
void Game::update(sf::Time elapsedTime)
{
   sf::Vector2f movement(0.f, 0.f);
   if (sf::Keyboard::isKeyPressed(sf::Keyboard::W))
       movement.y -= PlayerSpeed;
   if (sf::Keyboard::isKeyPressed(sf::Keyboard::S))
       movement.y += PlayerSpeed;
   if (sf::Keyboard::isKeyPressed(sf::Keyboard::A))
       movement.x -= PlayerSpeed;
   if (sf::Keyboard::isKeyPressed(sf::Keyboard::D))
```

Events vs. Real-Time Input

If a state has changed, you should use events

However, if you want to know the current state, then of course you must check using a function

```
// WHEN the left mouse button has been pressed, do something

if (event.type == sf::Event::MouseButtonPressed)



// WHILE the left mouse button is being pressed, do something

if (sf::Mouse::isButtonPressed(sf::Mouse::Left))
```

So the second method is good for sustained input

Delta Movement

The different in cursor position between two frames

```
sf::Vector2i mousePosition = sf::Mouse::getPosition(mWindow);

sf::Vector2i delta = mLastMousePosition - mousePosition;

mLastMousePosition = mousePosition;
```

Applying the Focus

```
void Game::run()

{

    while (mWindow.isOpen())

    {

        if (!mIsPaused)
          update();
        render();
        processEvents();
    }

}

void Game::processEvents()

{

    sf::Event event;

    while(mWindow.pollEvent(event))
```

Commands

Commanding the Entities

- Some example commands might be as follows:

```
// One-time events
sf::Event event;
while (window.pollEvent(event))
{
    if (event.type == sf::Event::KeyPressed
    && event.key.code == sf::Keyboard::X)
     mPlayerAircraft->launchMissile();
}
```

```
// Real-time input
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left))
  mPlayerAircraft->moveLeft();
else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right))
  mPlayerAircraft->moveRight();
```

Commanding (cont'd.)

Commands are messages that are sent to game objects

- Alter the object
- Issue orders:
  - Movement
  - Firing weapons
  - Triggering state changes

Command struct

```
struct Command
{
      std::function<void(SceneNode&, sf::Time)> action;
};
```

`std::function` is a C++11 class template to implements callback mechanisms. It treats functions as objects and makes it possible to copy functions or to store them in containers. The `std::function` class is compatible with function pointers, member function pointers, functors, and lambda expressions. The template parameter represents the signature of the function being stored.

std::function Example

```
int add(int a, int b) { return a + b };

std::function<int(int, int)> adder1 = &add;


std::function<int(int, int)> adder2

= [] (int a, int b) { return a + b; };
```

Then it can be used thusly:

```
int sum = adder1(3, 5); // same as add(3, 5)
```

Movement Example

```
void moveLeft(SceneNode& node, sf::Time dt)
{
    node.move(-30.f * dt.asSeconds(), 0.f);
}
Command c;
c.action = &moveLeft;
```

Using Lambda expression, the equivalent being:

```
c.action = [] (SceneNode& node, sf::Time dt)
{
    node.move(-30.f * dt.asSeconds(), 0.f);
};
```

Commanding (cont'd.)

- The different game objects should each receive their appropriate commands
- So they are divided into different categories
- Each category has one bit set to 1 and rest are set to 0

```
namespace Category
{
    enum Type
    {
        None = 0,
        Scene = 1 << 0,
        PlayerAircraft = 1 << 1,
        AlliedAircraft = 1 << 2,
        EnemyAircraft = 1 << 3,
    };
}
```

Commanding (cont'd.)

- A bitwise OR operators allows us to combine different categories, for example all airplanes:

```
unsigned int anyAircraft =
 Category::PlayerAircraft
                    |
 Category::AlliedAircraft
                    |
 Category::EnemyAircraft;
```

The SceneNode class gets a new virtual method that returns the category of the game object. In the base class, we return Category::Scene by default:

```
unsigned int SceneNode::getCategory()
const
{
return Category::Scene;
}
```

Commanding (cont'd.)

- `getCategory()` can be overridden to return a specific category
- an aircraft belongs to the player if it is of type Eagle, and that it is an enemy otherwise:

```
unsigned int Aircraft::getCategory()
  const
{
  switch (mType)
  {
      case Eagle:
              return
  Category::PlayerAircraft;
      default:
              return
  Category::EnemyAircraft;
  }
}
```

Command struct Revisited

- we give our Command class another member variable that stores the recipients of the command in a category:

```
struct Command
{
 Command();
 std::function<void(SceneNode&, sf::Time)> action;
 unsigned int category;
};
```

- The default constructor initializes the category to Category::None. By assigning a different value to it, we can specify exactly who receives the command. If we want a command to be executed for all airplanes except the player's one, the category can be set accordingly:

```
Command command;
command.action = ...;
command.category = Category::AlliedAircraft
| Category::EnemyAircraft;
```

Command Execution

- Commands are passed to the scene graph
- Inside, they are distributed to all scene nodes with the corresponding game objects
- Each scene node is responsible for forwarding a command to its children
- SceneNode::onCommand() is called everytime a command is passed to the scene graph

```
void SceneNode::onCommand(const
  Command& command, sf::Time dt)
 { //check if the current scene node is a receiver
 of the command
  if (command.category & getCategory())
        command.action(*this, dt);


  FOREACH(Ptr& child, mChildren)
        child->onCommand(command, dt);
}
```

Command Queues

- A way to transport commands to the world and the scene graph
- A class that is a very thin wrapper around a queue of commands

```
class CommandQueue
{
public:
  void push(const Command& command);
  Command pop();
  bool isEmpty() const;

private:
  std::queue<Command> mQueue;
};
```

Command Queues (cont'd.)

The `World` class holds an instance of `CommandQueue`:

```
void World::update(sf::Time dt)
{
    ...

    // Forward commands to the scene graph
    while (!mCommandQueue.isEmpty())
        mSceneGraph.onCommand(mCommandQueue.pop(), dt);

    // Regular update step
    mSceneGraph.update(dt);
}
```

Player and Input

Together now we're going to look at how the player's input is handled

We will look at the following:

- The `Player` class
- The `processInput` function from `Game`

class Game

```
class Game : private sf::NonCopyable
{
public:
Game();
voidrun();

private:
voidprocessEvents();
voidupdate(sf::Time elapsedTime);
voidrender();
voidupdateStatistics(sf::Time elapsedTime);

private:
Player mPlayer;
static const sf::TimeTimePerFrame;
```

Game::processEvents()

```
void Game::processEvents()
{
CommandQueue& commands = mWorld.getCommandQueue();
sf::Event event;
while (mWindow.pollEvent(event))
{
switch (event.type)
{
mPlayer.handleEvent(event, commands);
case sf::Event::Closed:
mWindow.close();
break;
}

mPlayer.handleRealtimeInput(commands);
}
}
```

class Player

```
class Player

{

public:

Player();

static const float PlayerSpeed;

Void handleEvent(const sf::Event& event, CommandQueue& commands);

Void handleRealtimeInput(CommandQueue& commands);

};
```

Player::handleEvent

```
void Player::handleEvent(const sf::Event& event, CommandQueue& commands)

{

if (event.type == sf::Event::KeyPressed && event.key.code == sf::Keyboard::P)

{

Command output;

output.category = Category::PlayerAircraft;

output.action = [](SceneNode& s, sf::Time) {

std::cout << s.getPosition().x << ","

<< s.getPosition().y << "\n";

};

commands.push(output);

}

}
```

AircraftMover

```
struct AircraftMover
{
AircraftMover(float vx, float vy)
: velocity(vx, vy)
{
}

void operator() (Aircraft& aircraft, sf::Time) const
{
aircraft.accelerate(velocity);
}

sf::Vector2f velocity;
};
```

Player::handleRealtimeInput

```
void Player::handleRealtimeInput(CommandQueue& commands)
{
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left))
{
Command moveLeft;
moveLeft.category = Category::PlayerAircraft;
moveLeft.action = derivedAction<Aircraft>(
AircraftMover(-PlayerSpeed, 0.f)
);
commands.push(moveLeft);
}

if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right))
{
Command moveRight;
moveRight.category = Category::PlayerAircraft;
moveRight.action = derivedAction<Aircraft>(
```

derivedAction

```
template <typename GameObject, typename Function>

std::function<void(SceneNode&, sf::Time)> derivedAction(Function fn)

{

return [=](SceneNode& node, sf::Time dt)

{

// Check if cast is safe

assert(dynamic_cast<GameObject*>(&node) != nullptr);



// Downcast node and invoke function on it

fn(static_cast<GameObject&>(node), dt);

};

}
```