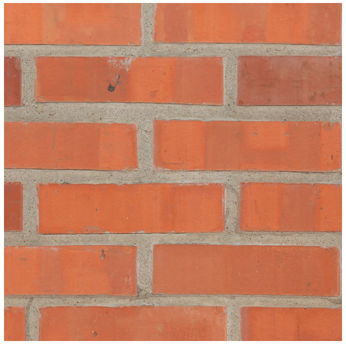
TA 8

- Texture types
- Displacement Mapping
- Bump Mapping
- Reflection Mapping
- Texturing in Unity ShaderLab

Texture Types

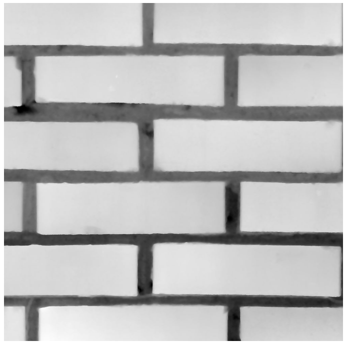
Computer Graphics 2020

Some Types of Textures

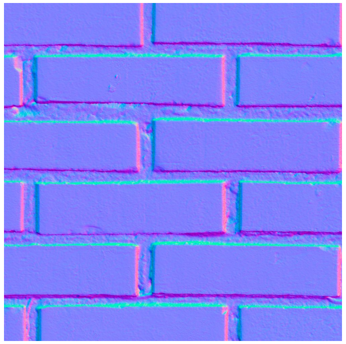


Albedo/Color Map

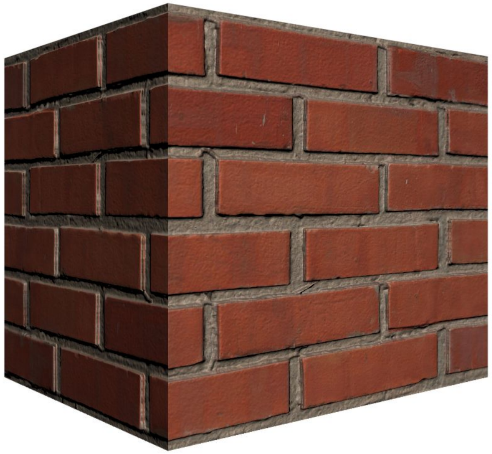
Specular Map



Height Map

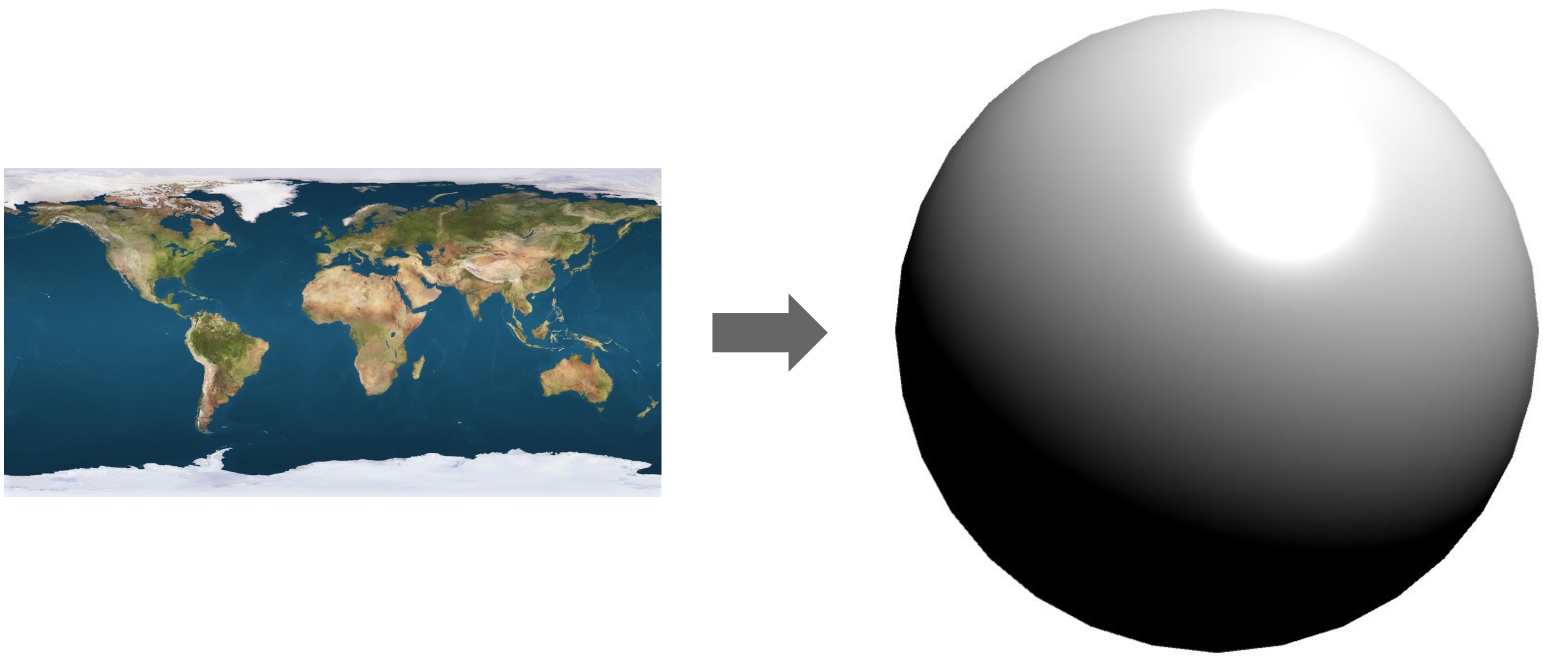


Normal Map



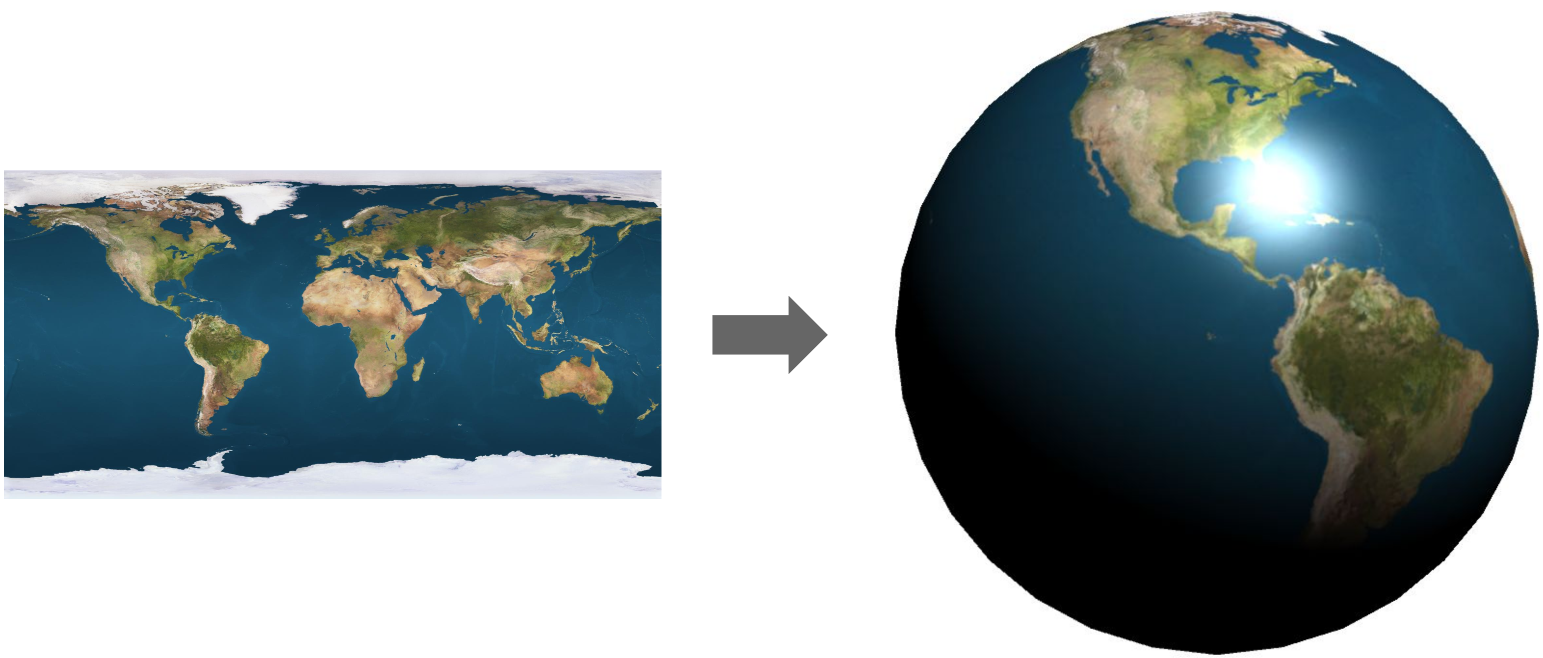
Albedo/Color Mapping

- Sets the basic diffuse color of the material
- 3 channels - RGB



Albedo/Color Mapping

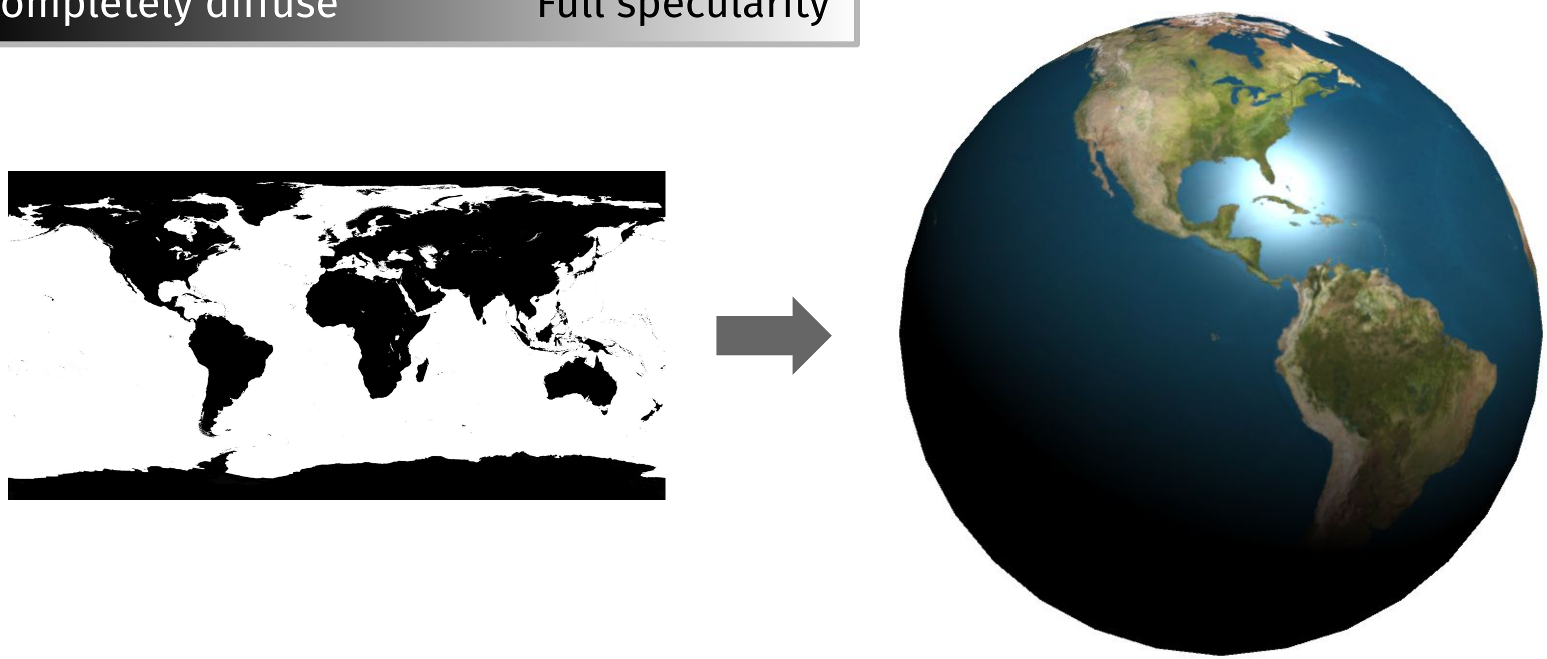
- Sets the basic diffuse color of the material
- 3 channels - RGB



Specular Mapping

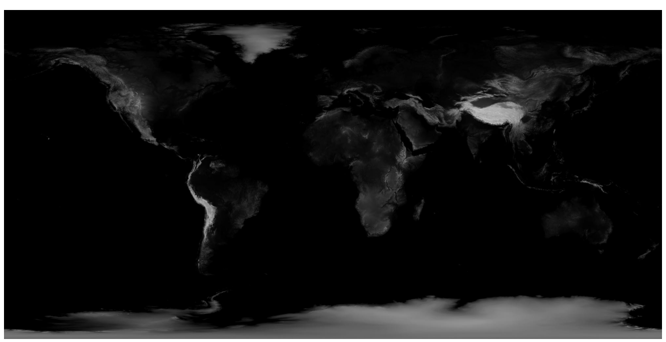
- Sets the specularity of the material at each point, allows for more and less reflective areas

Completely diffuse — Full specularity



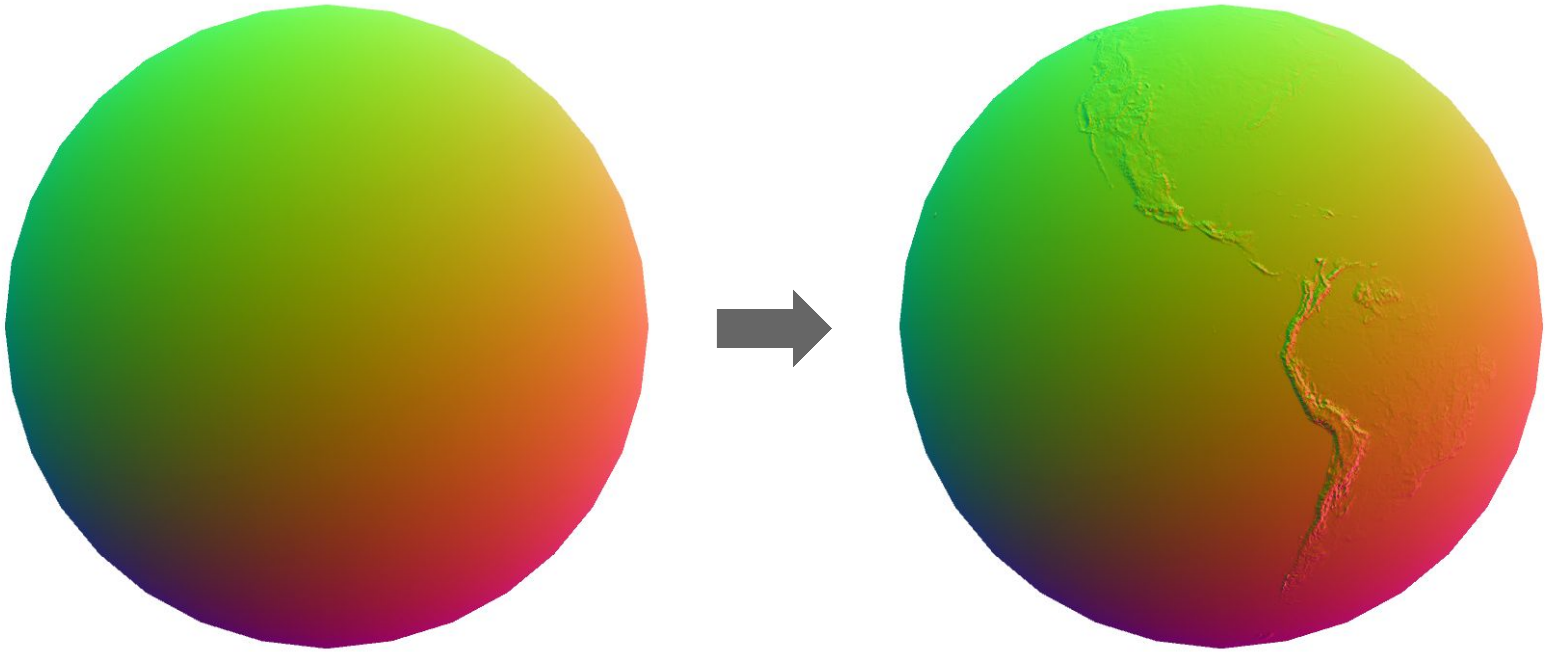
Bump Mapping

- “Fakes” details on the surface by changing the surface normals according to a height map

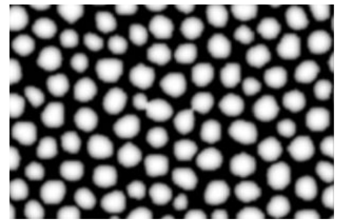


Bump Mapping

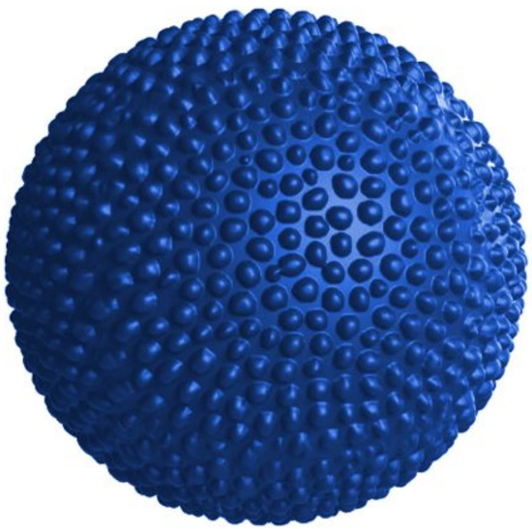
- We can visualize the surface normals at each point to see the effect better:



Displacement vs. Bump Mapping



Height Map



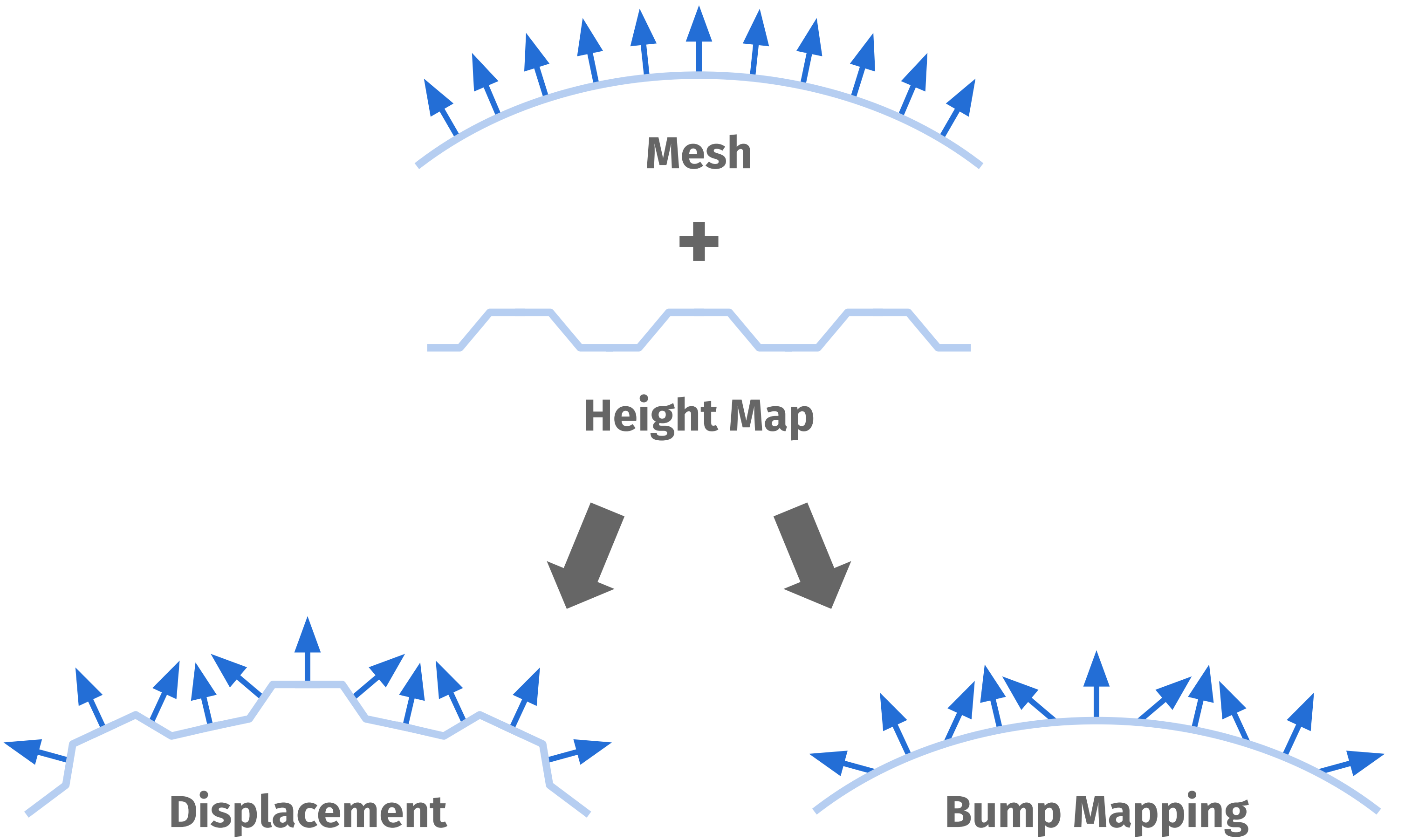
Displacement Mapping

Bump Mapping

Displacement Mapping

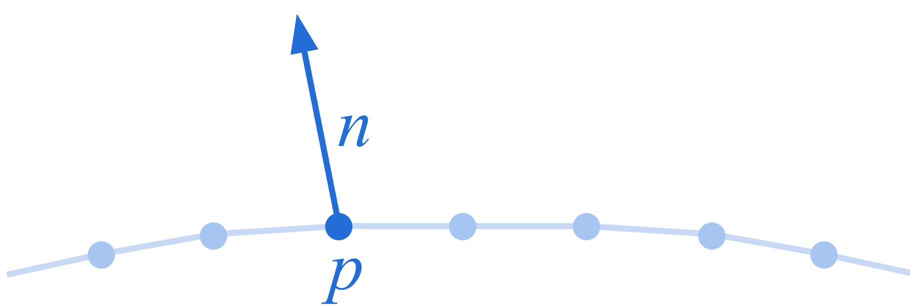
- Also uses a height map
- Affects the actual geometry - moves vertices according to the height values
- Visually more accurate than bump mapping because we actually change the geometry
- Requires a high vertex count, very inefficient when compared to bump mapping

Displacement vs. Bump Mapping



Displacement Mapping

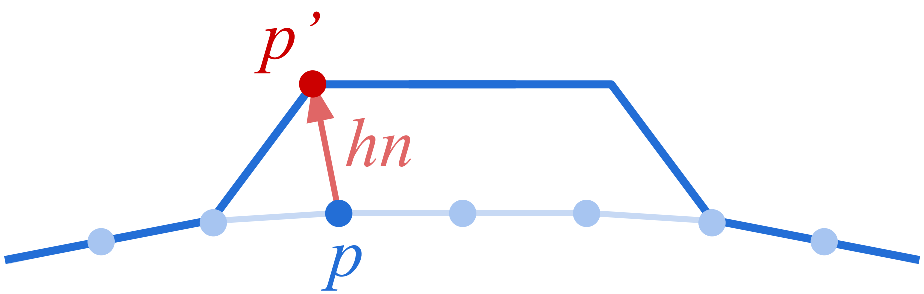
- Say we have a height map $f(u,v) = h$ and some mapping function $m : \mathbb{R}^3 \to [0,1]^2$
- We have a vertex at position $p \in \mathbb{R}^3$ with normal n to which we want to apply the displacement map



Displacement Mapping

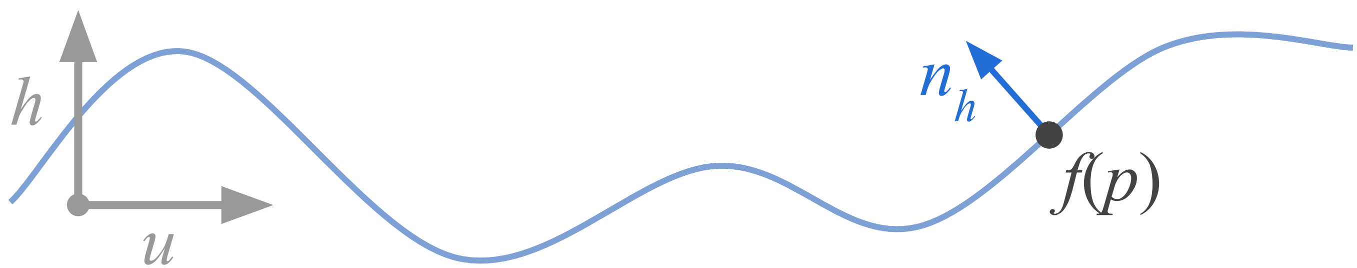
- We get the new position p' by displacing along the surface normal, according to the height value at the point $h = f(m(p))$:

$$p' = p + hn$$



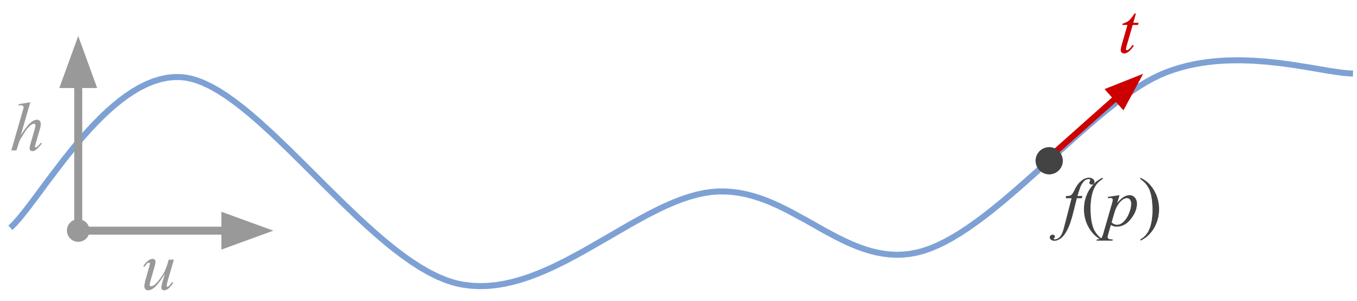
Bump Mapping

- To apply a bump map we need to figure out what the normal n_h should be at each point - using the heightmap data
- First, let's limit the height map to 1 dimension: $f(u) = h$. How do we find the normal at $f(p)$?



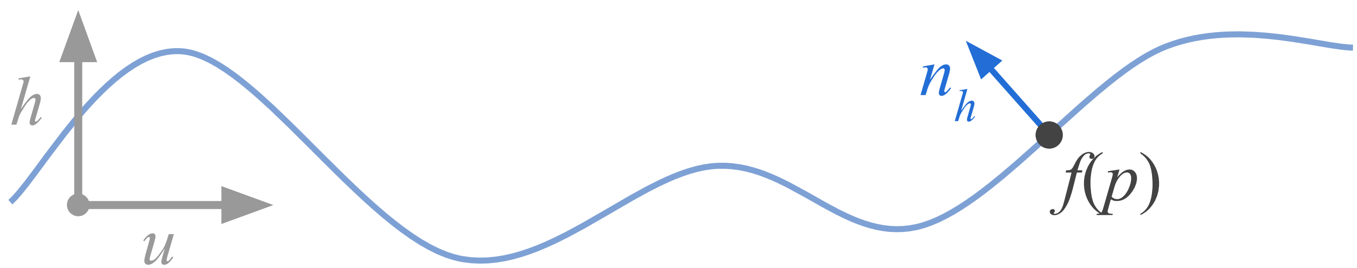
Bump Mapping

- If we knew the slope of the function, we could use it to compute its normal at any point
- The slope is given by the derivative $f'(u)$
- From the slope we can calculate a tangent vector to the function at p: $t = (1, 0, f'(p))^\top$



Bump Mapping

- The normal direction is a 90° rotation of the tangent, so we get $n_h = (-f'(p), 0, 1)^\top$
- Usually we will us a heightmap that isn't a simple differentiable function, so we can't calculate $f'(p)$ directly



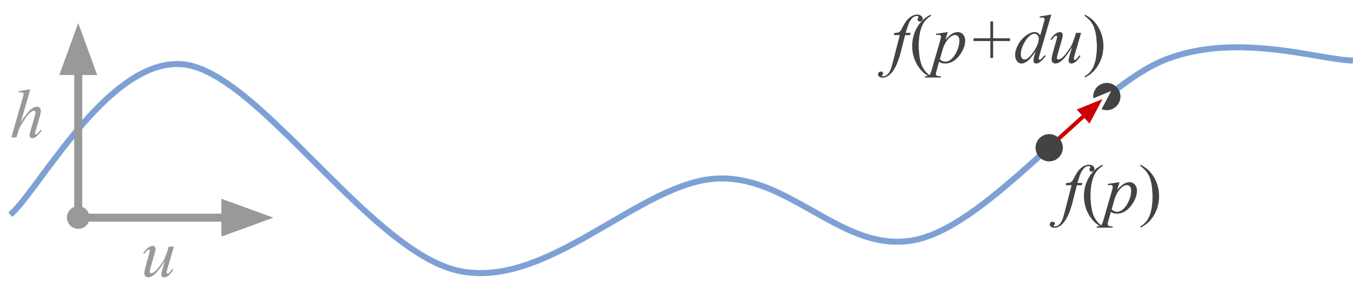
Bump Mapping

- Recall the derivative definition:

$$f'(p) = \lim_{du \to 0} [f(p + du) - f(p)] / du$$

- We can approximate the derivative by using a small step size *du*:

$$f'(p) \approx [f(p + du) - f(p)] / du$$



Bump Mapping

- A heightmap is actually a 2D function $f(u,v) = h$
- We've been using f_u' - the partial derivative of f with respect to u
- t_u is the surface tangent in the u direction

Bump Mapping

- We can approximate the partial derivative with respect to v similarly: $f_v\text{'}(p) \approx [f(p + dv) - f(p)] \,/\, dv$
- The tangent vector in the v direction is given by:

$$t_v = (0,\ 1, f_v\text{'})^\top$$

Bump Mapping

- Finally, the normal direction can be calculated using the cross product:

$$n_h = t_v \times t_u = (0,\ 1, f_v\text{'})^\top \times (1,\ 0, f_u\text{'})^\top =$$

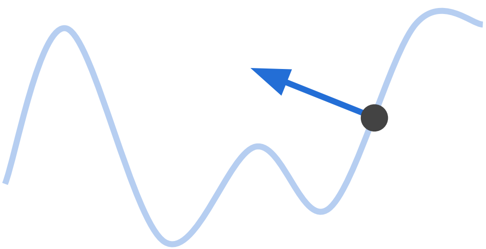
$$(-f_u\text{'}, -f_v\text{'}, 1)$$

Bump Mapping

- As always, remember to **normal**ize!
- Because the height value is some arbitrary number between 0 and 1, we use a "bump scale" parameter s that controls the "steepness":

$$n_h = (-sf_u', -sf_v', 1) \,/\, \|(-sf_u', -sf_v', 1)\|$$

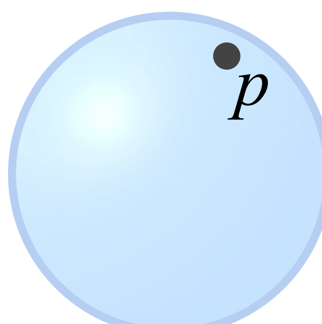
Small bump scale



Large bump scale

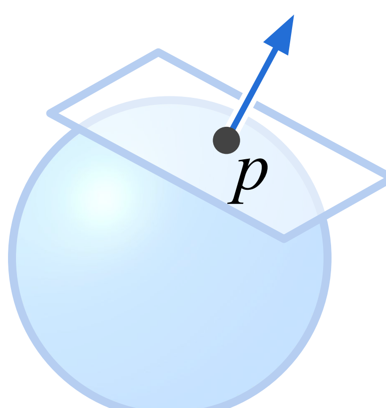
Bump Mapping

- We now know how to find the normal n_h at each point of the heightmap, but we are using the texture coordinate system!
- Remember we mapped the 2D texture to a 3D object somehow, so p is actually a point in 3D space:



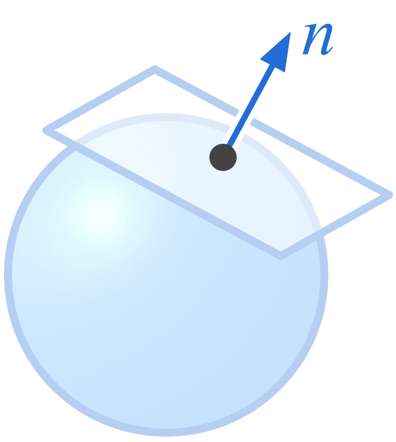
Bump Mapping

- We have to transform the results of our bump mapping so it matches the orientation of p in the scene
- In other words, we need to translate ***tangent-space*** u, v, h to world-space coordinates



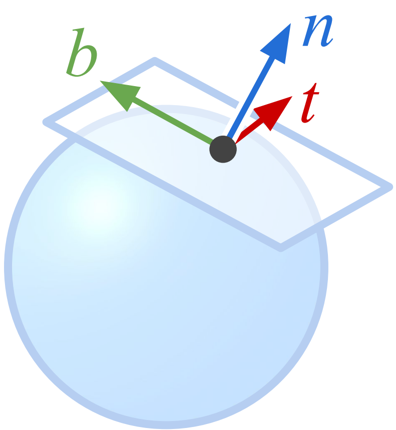
Tangent Space

- We already have the normal vector n that corresponds to the “height” axis h
- Remember that n_h is a different vector!
- We need vectors that define the direction of the u, v axes at each point p



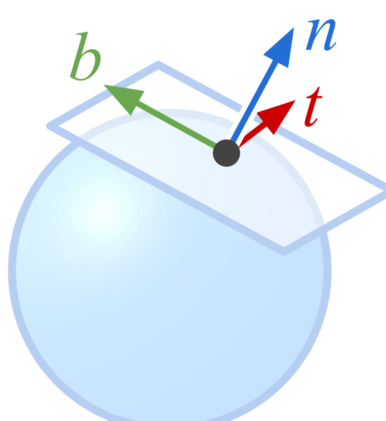
Tangent Space

- The vector in the direction of the u axis is called the ***tangent vector*** t
- The vector in the direction of the v axis is called the ***bitangent*** or the ***binormal vector*** b



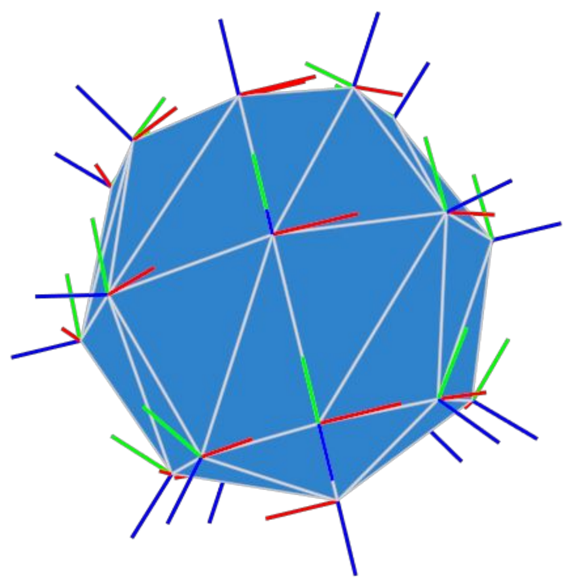
Tangent Space

- Usually the tangent vector t is stored for each vertex of a mesh, along with the surface normal n and the uv coordinates
- The binormal can be derived using the cross product: $b = t \times n$



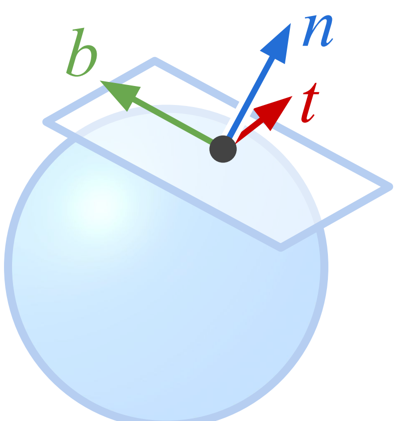
Tangent Space

- Usually the tangent vector t is stored for each vertex of a mesh, along with the surface normal n and the uv coordinates



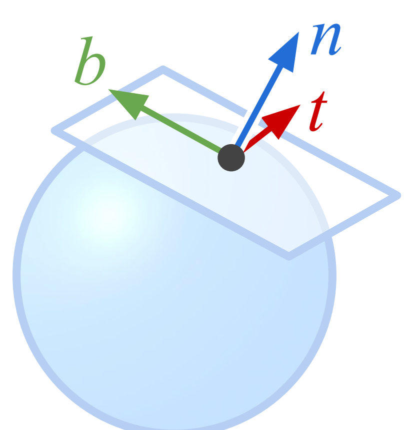
Tangent Space

- Remember that n and t are given in object-space, so we must transform them to world-space before using them



Tangent Space

- We can now translate our tangent-space normal $n_h = (n_x, n_y, n_z)^\top$ to world-space coordinates
- Multiply each component n_x, n_y, n_z by its corresponding axis vector t, b, n

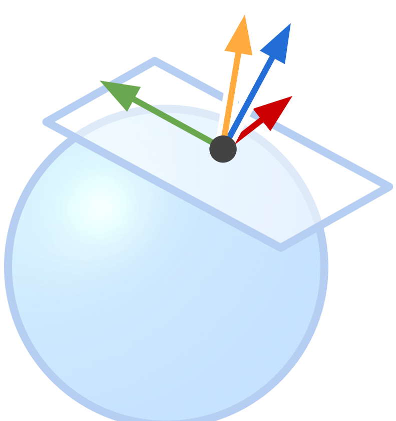


Tangent Space

- Finally we have our world-space bump-mapped normal which we can use for shading!

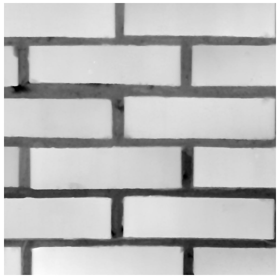
$$n_{world} = t \cdot n_x + n \cdot n_z + b \cdot n_y$$

- Remember that in Unity, y rather than z is "up"

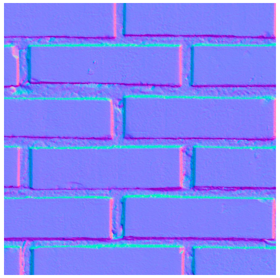


Normal Mapping

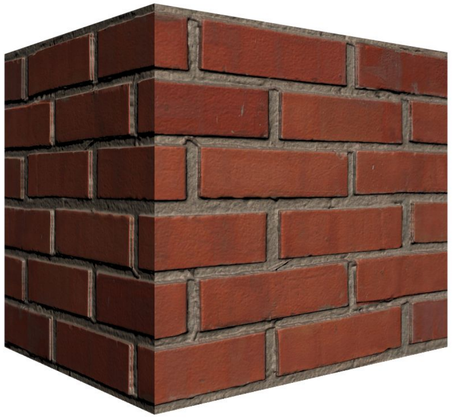
- The computation of tangent-space normals can be done in advance and saved to an rgb texture
- Requires less computation at runtime
- We can also save object-space normals



Height Map



Normal Map



Reflection Mapping

- ***Reflection Mapping***, also called *Environment Mapping*, is a method for simulation environment reflections on a reflective surface

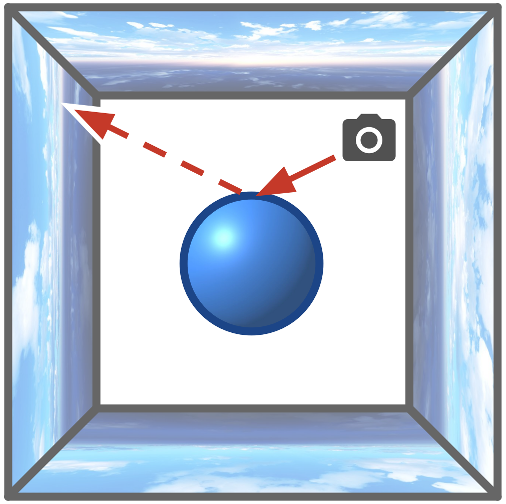
Reflective Bunny

Reflection Mapping

- We use a texture to store the image of the distant environment surrounding the rendered objects
- Usually a ***Cube Map*** with 6 faces representing up, down, left, right, front and back directions:

Reflection Mapping

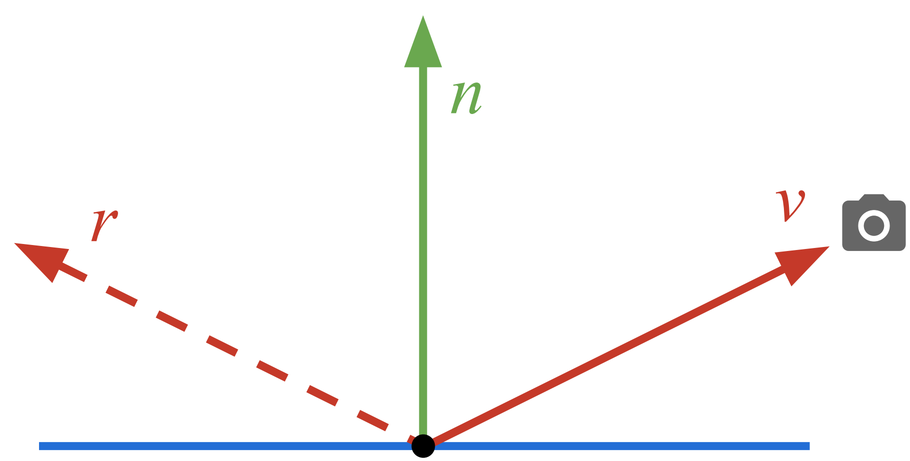
- At each point we calculate the reflection of the view direction
- Use this reflection vector to sample the environment cube map texture



Reflection Mapping

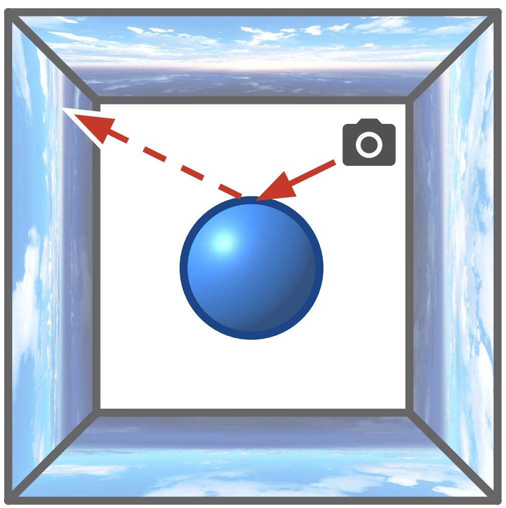
- Reminder - to find the reflection direction:

$$r = 2(v \cdot n)n - v$$



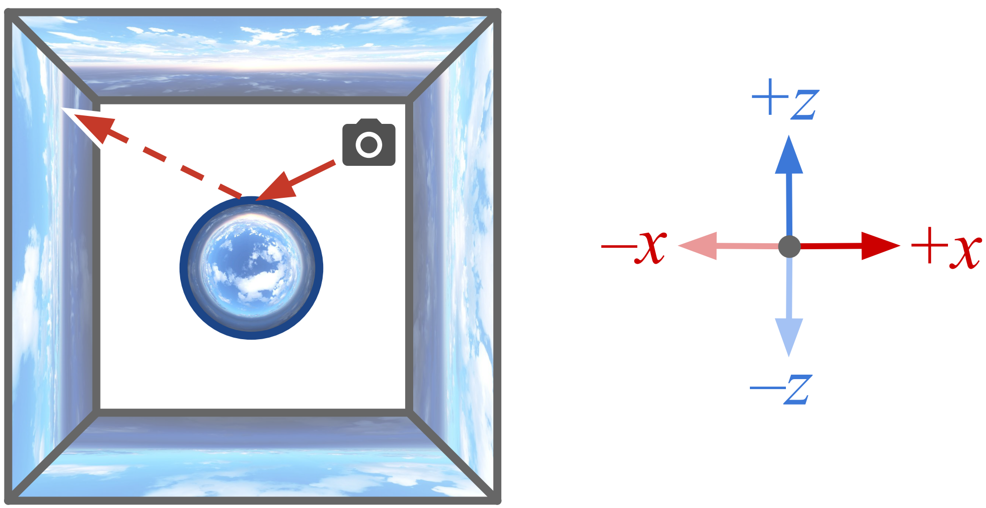
Reflection Mapping

- We assume the environment is infinitely far away, and each point is in the center of the cube
- Given a reflection direction $r = (x, y, z)$ how do we know which face to sample and where?



Reflection Mapping

- The component of the reflection vector with the greatest *magnitude* determines the face it will hit
- The exact uv coordinates will be the other two components scaled to [0, 1]



Reflection Mapping

- This works really well if our scene is static and we do not have many objects in our scene
- To get reflections of other objects in the scene we can *render* a cubemap by positioning additional cameras at each reflective object
- To get dynamic reflections we can *update* the cubemaps each frame - costly!

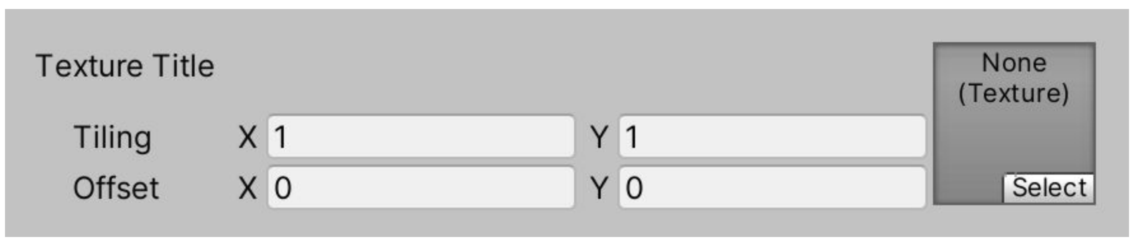
Unity Texturing

Computer Graphics 2020

ShaderLab Textures

- To use a texture in our shader, we first need to declare a property that will allow us to select an image in the inspector UI:

```
_MainTex ("Texture Title", 2D) = "defaulttexture" {}
```



- The texture type can be `2D`, `3D` or `Cube` if we want to use a cube map

ShaderLab Textures

- Like any ShaderLab property, we need to declare a uniform to access it

- We declare a ***Texture Sampler*** that will be associated with the texture:

```
uniform sampler2D _MainTex;
```

- The sampler needs to match the texture type: `sampler2D`, `sampler3D` or `samplerCUBE`

ShaderLab Textures

- We can also declare a `TexelSize` uniform:

```
uniform sampler2D _MainTex;
uniform float4 _MainTex_TexelSize;
```

- Its first two components contain the texel sizes, as fractions of U and V, the other two components contain the number of pixels
- For example, in case of a 256×128 texture:

$$(1/256 \approx 0.004,\ 1/128 \approx 0.008,\ 256,\ 128)$$

ShaderLab Textures

- Finally, to sample the texture in our vertex or fragment shaders, we need to pass some UV coordinates along with the sampler:

```
half4 color = tex2D(_MainTex, uv);
```

- As before, the sampling function matches the texture type: `tex2D`, `tex3D` or `texCUBE`
- By default, Unity generates MipMaps for every texture and handles the sampling logic behind the scenes

ShaderLab Textures

- When using a cubemap we sample using a direction vector rather than UV coordinates:

```
uniform samplerCUBE _CubeMap;

// ...

float3 dir = float3(0.21, 0.45, 0.86);
half4 color = texCUBE(_CubeMap, dir);
```