

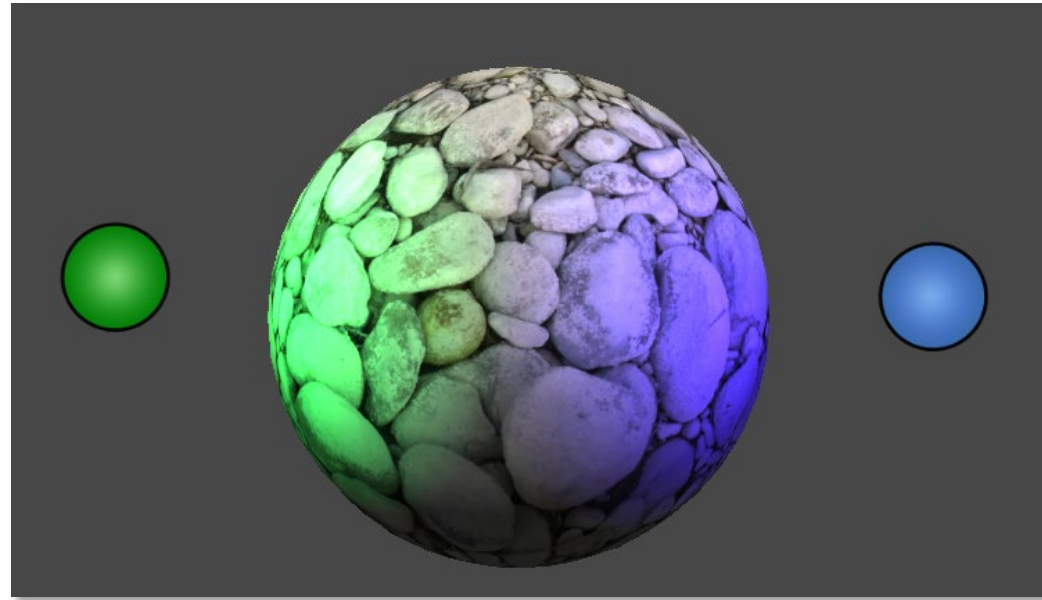
Normal Mapping

Review: Basic 3D lighting

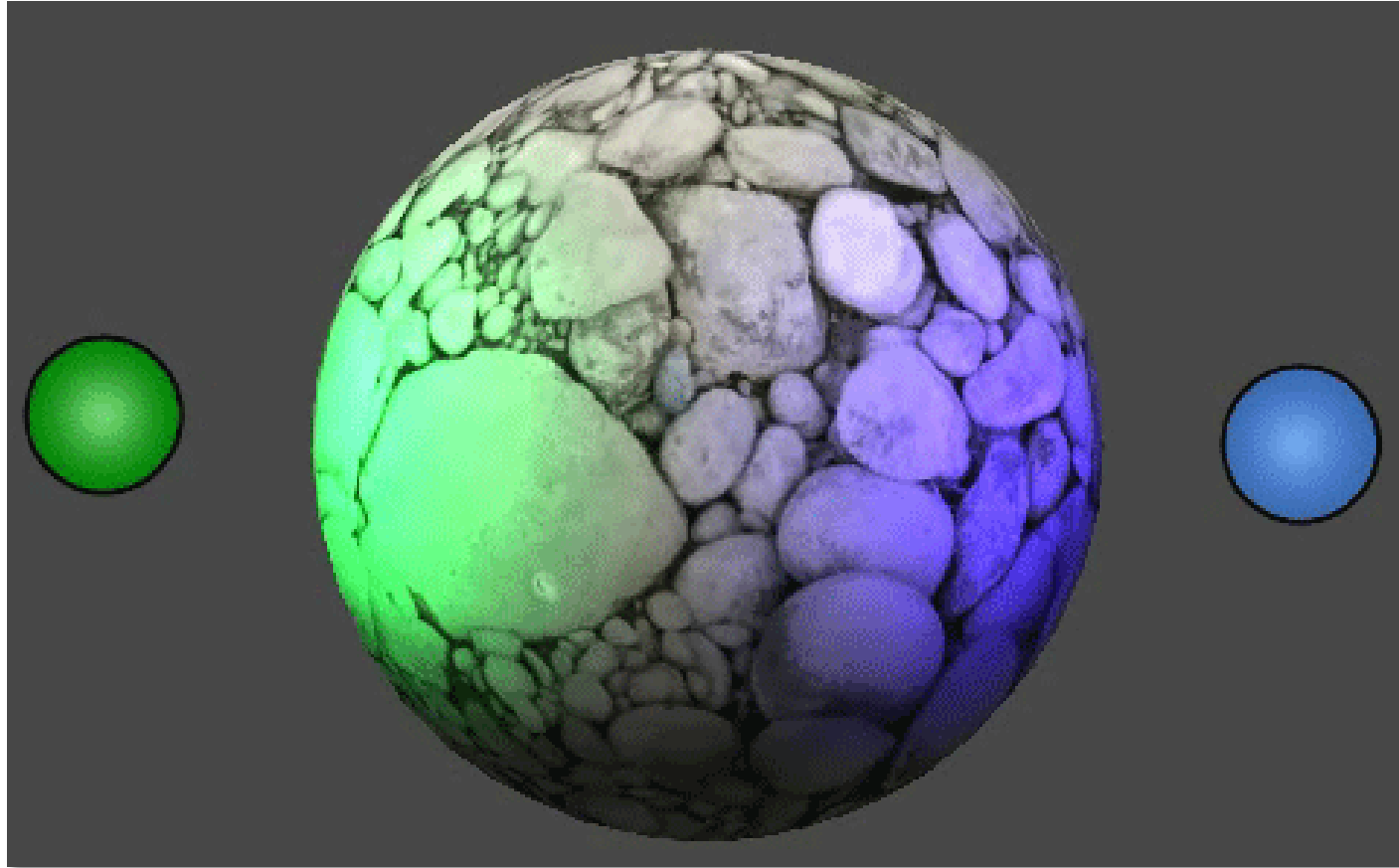
- ▶ Shading calculation compares two things:
 - Normal of the surface (triangle)
 - Direction of the light
- ▶ Results can be combined with textures
- ▶ Problems:
 - Equations only use surface normals
 - Doesn't "know" what the texture represents

Basic lighting example

- ▶ The triangles are flat
 - The texture looks rather flat when applied
 - Ideally each rock should look “rounded”



What we currently have



The only “shape” information is the sphere itself

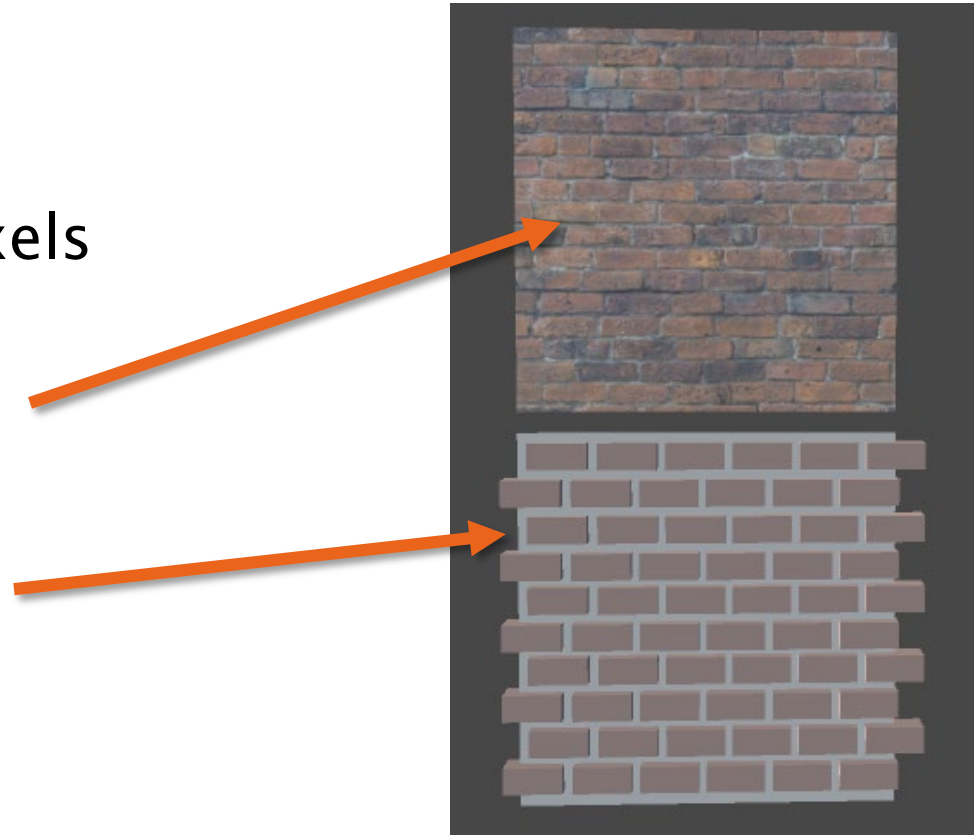
Flat rocks?

- ▶ Each rock should be lit as if it were actually round
- ▶ Problems:
 - The only “surface direction” information we have comes from the surface itself
 - That surface is “flat” (compared to texture)
 - Modeling and rendering individual rocks may be too costly
 - Especially if these are applied to an entire terrain



More detail with same geometry?

- ▶ Throwing more triangles at the problem can cause performance bottlenecks
 - ▶ Example: Walls
 - Could be handled in two ways
 - Both cover approx. equal # of pixels
1. Two triangles & texture(s)
 2. Actual geometry per brick
 - Lots of triangles & texture(s)

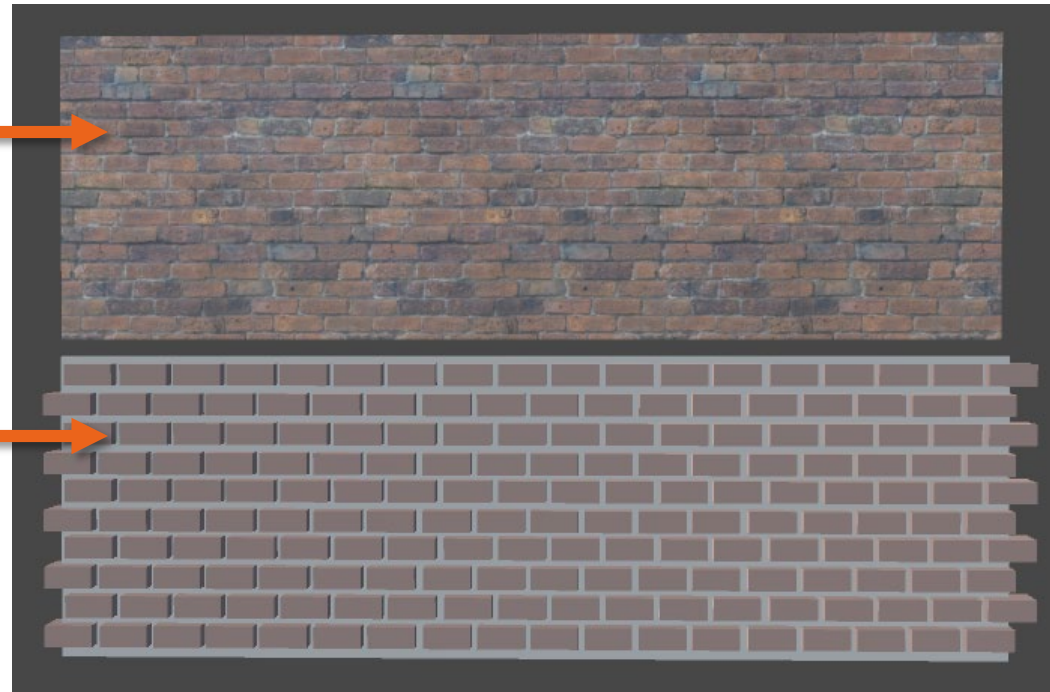


Extending the wall

- ▶ Let's say you want the wall to be 3x longer
 - Both methods still cover approx. equal # of pixels

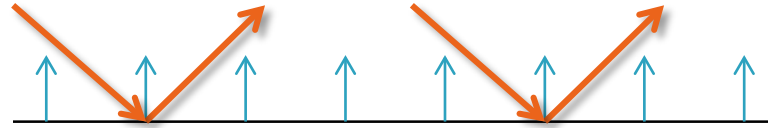
- ▶ Still 2 triangles
 - Just scale UV's

- ▶ Triple the geometry!

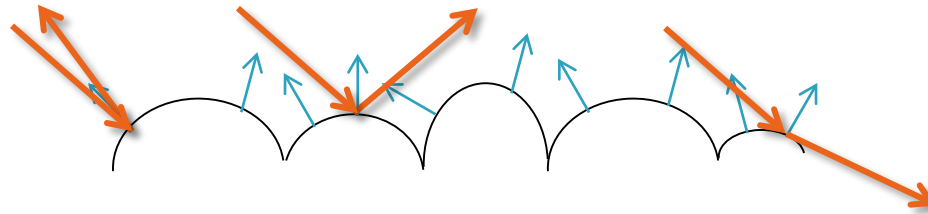


Problem visualization

- ▶ The surface we have:



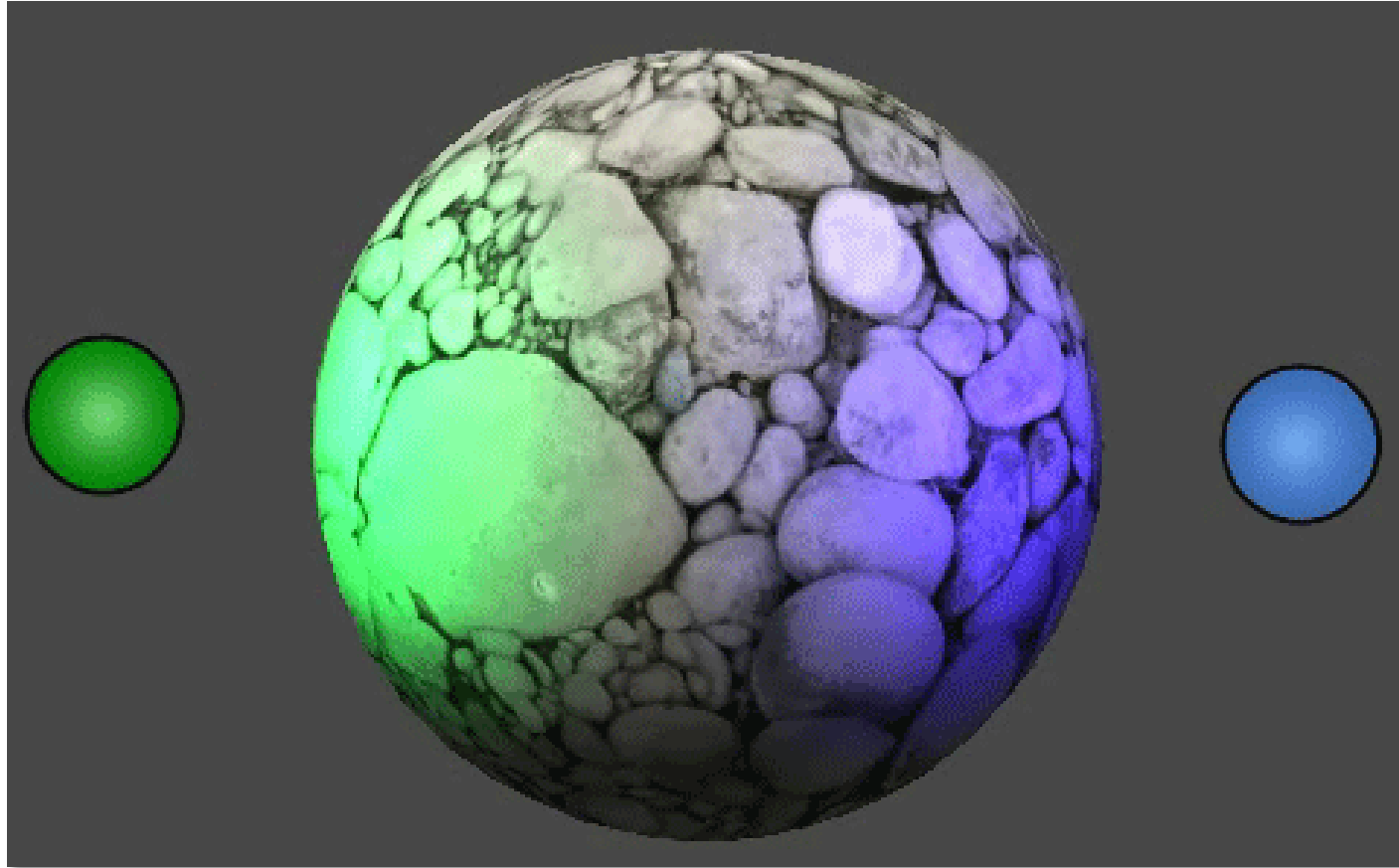
- ▶ The surface we want:



- ▶ Our ideal situation:

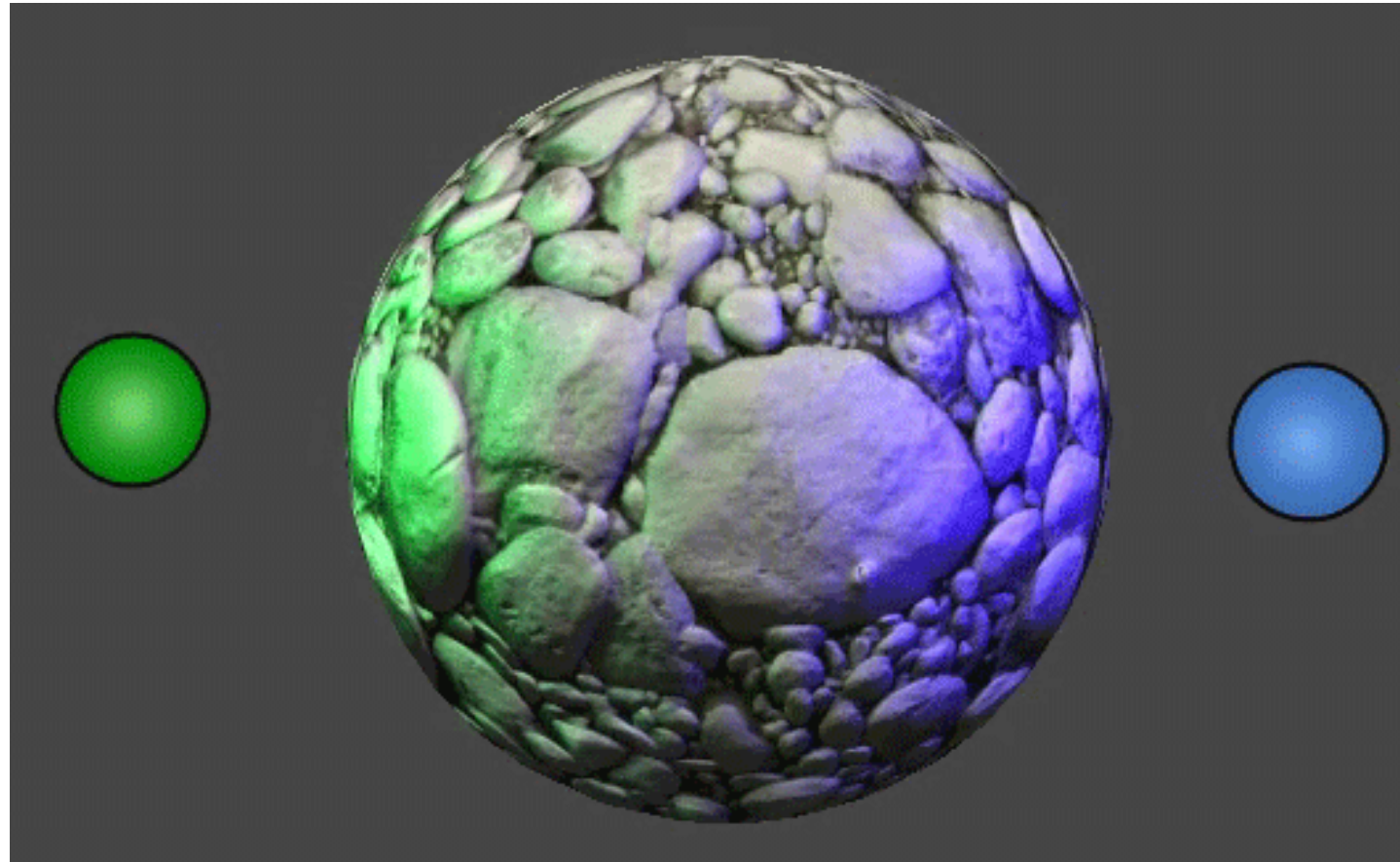


What we have now



The only “shape” information is the sphere itself

What we want

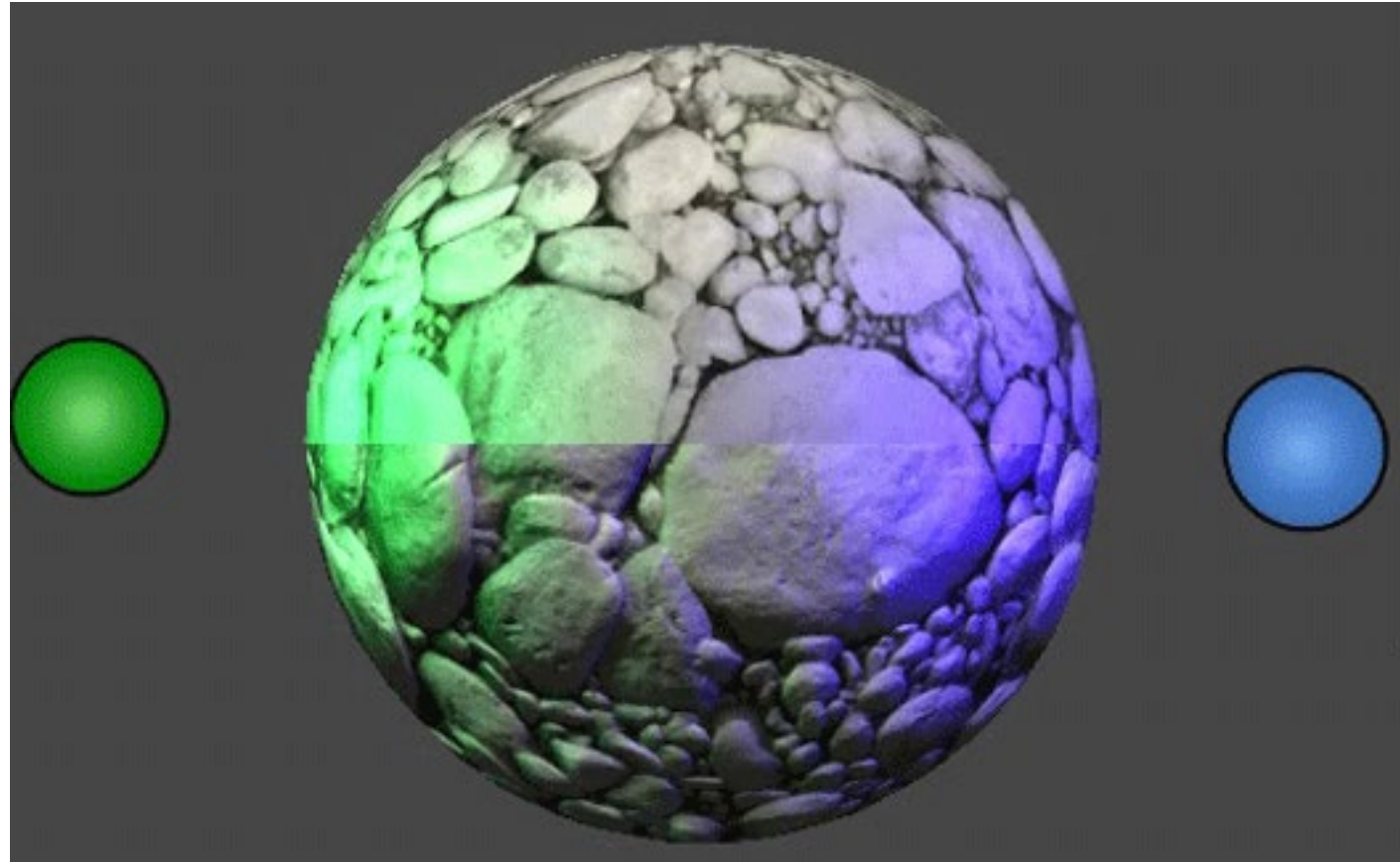


We want the shape of the rocks to also affect lighting

Comparison

What we have

What we want

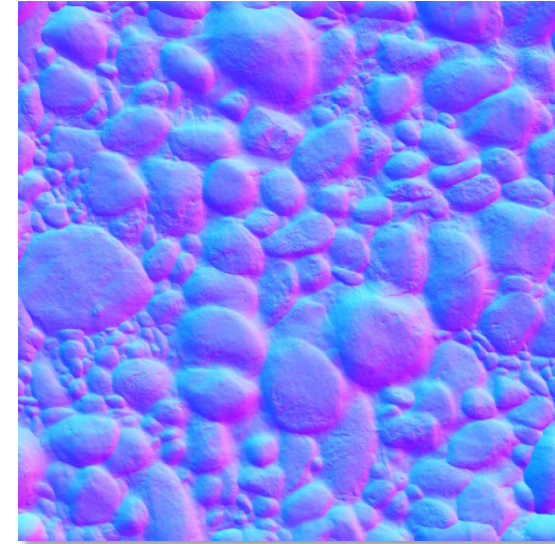


Per-pixel normals

- ▶ Need to alter normals at a sub-triangle level
- ▶ How do we alter colors at the pixel level?
 - A texture
 - “Per-pixel color”
 - Instead of gradients across each triangle
- ▶ How can we do that with normals?
 - Another texture
 - Specifically: A **normal map**

Normal maps

- ▶ Each pixel represents a particular direction
 - 3 colors = 3-component vector!
- ▶ XYZ = RGB? (Almost)
 - Images store colors 0 to 1
 - Normals could be -1 to 1
 - Unpack: $XYZ = RGB * 2 - 1$
- ▶ Sample this texture and use the resulting normal as part of your lighting calculations

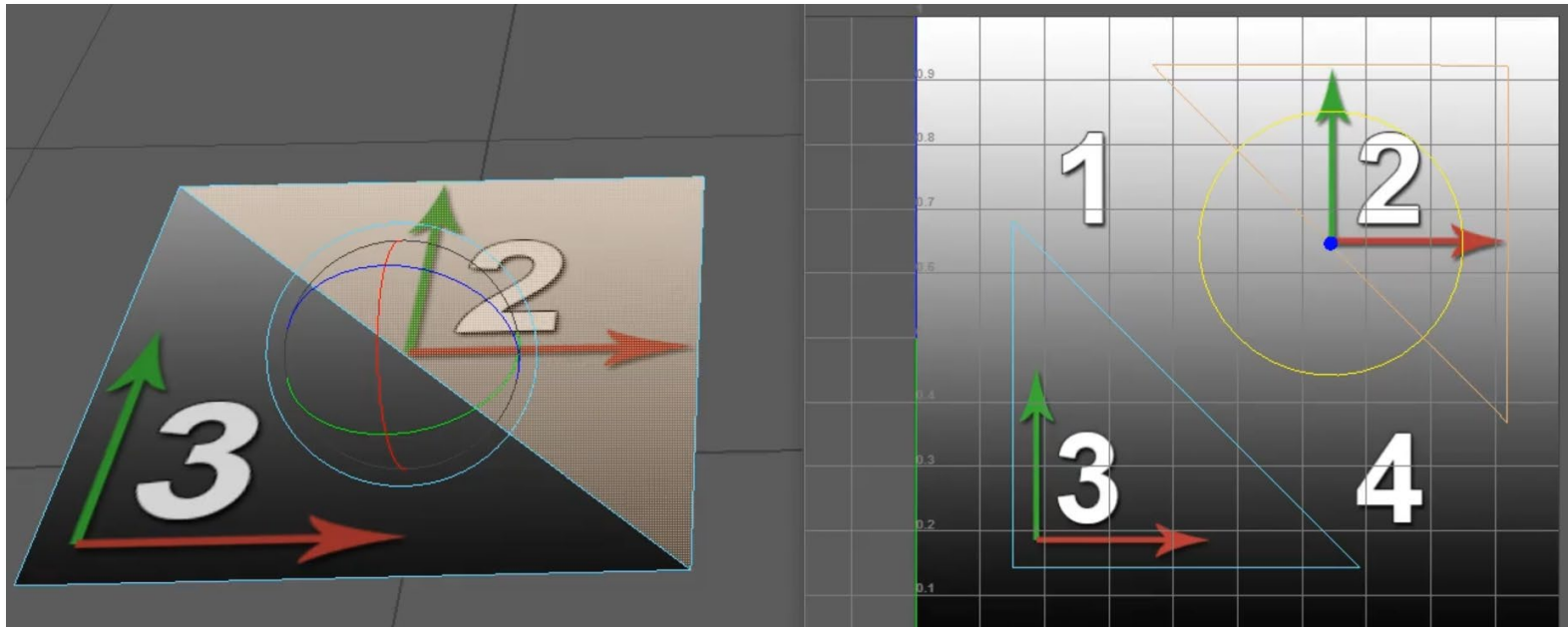


Normal maps

- ▶ Seems easy enough
 - Sample extra texture
 - Use that color as a normal
 - Same lighting calculation
- ▶ Unfortunately it's not *that* easy
- ▶ Normal map vectors are usually in *tangent space*
 - Relative to the UV mapping of each triangle
 - Need a little more information to convert

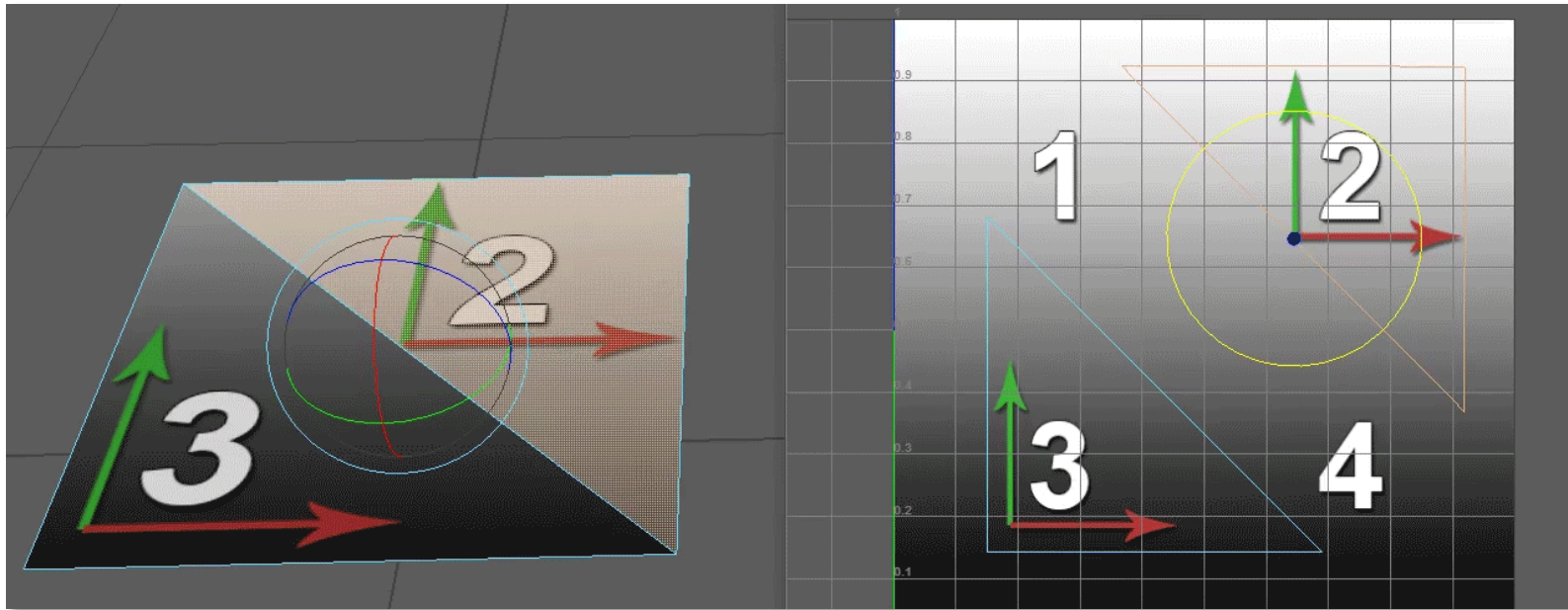
Textures on a surface

- ▶ Textures are mapped relative to the surface
 - But what if the UV mapping changes?



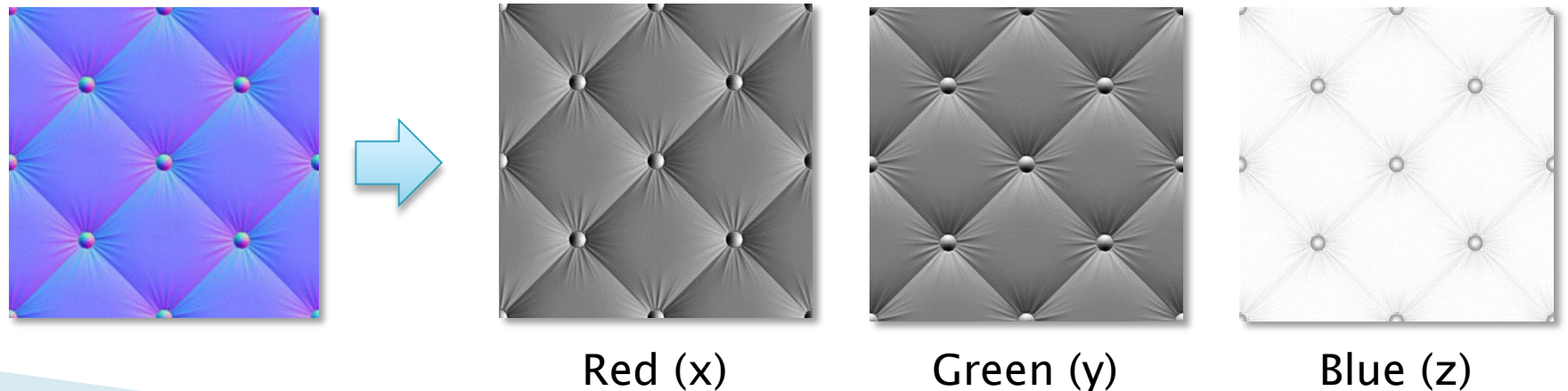
Which way is up?

- ▶ “Up” on the texture != “up” on the model
- ▶ Each triangle could even have a different “up”



Dealing with tangent space

- ▶ Relative to the underlying object's UV mapping
 - X on normal map is along U texture direction
 - Y on normal map is along V texture direction
 - Z on normal map points “away” from surface
- ▶ We need to convert to *world space*

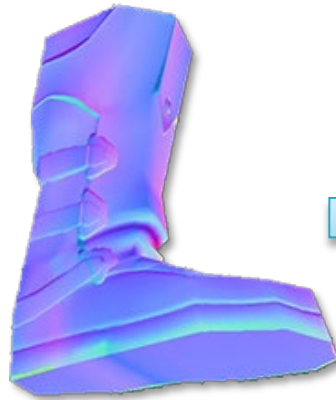


World vs. tangent space

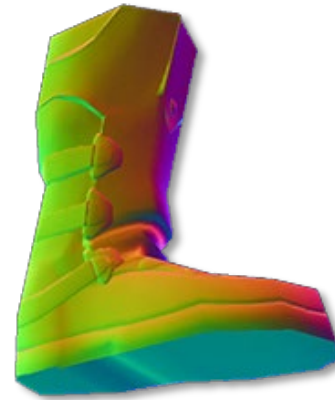
- ▶ Normal maps are stored in tangent space
- ▶ Our lighting equations are all in world space
 - We need to convert from tangent to world space
 - Or our lighting is wrong

Tangent Space

Blue = away
from surface

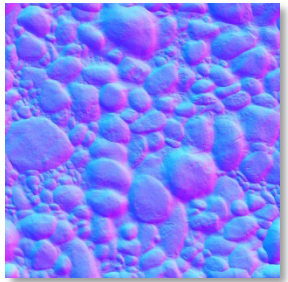


World space



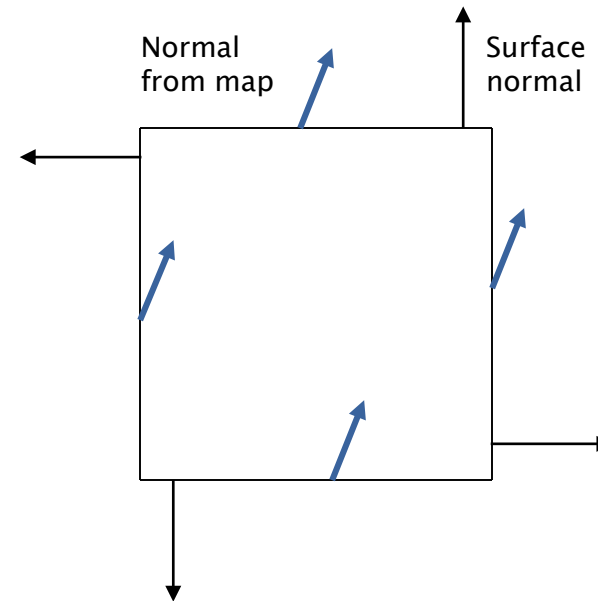
Tangent space – Problem example

Plane with normal map



Each pixel “points” in a specific direction

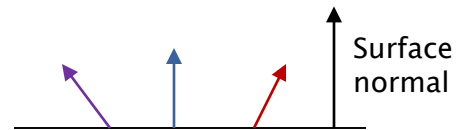
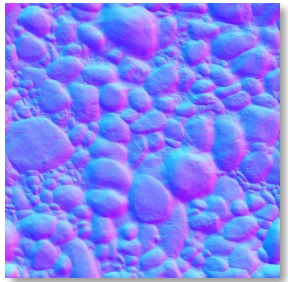
Cube with normal map
(Same map on each face)



Each pixel still “points” in the same direction regardless of triangle’s normal

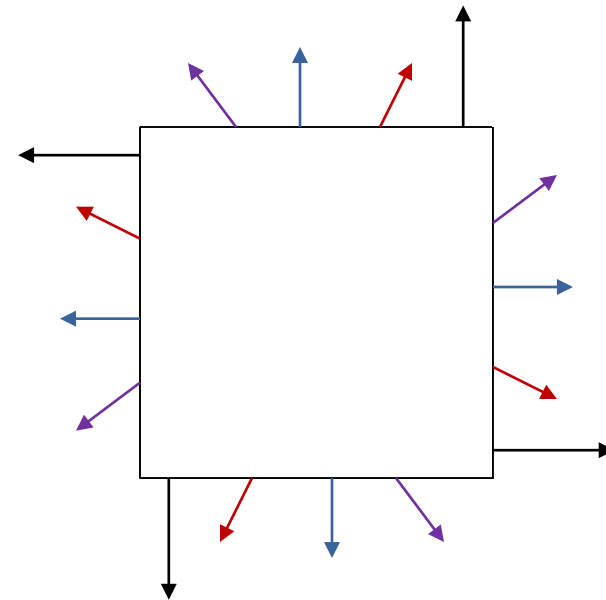
Tangent space – Correct example

Plane with normal map



Each pixel "points" in a specific direction

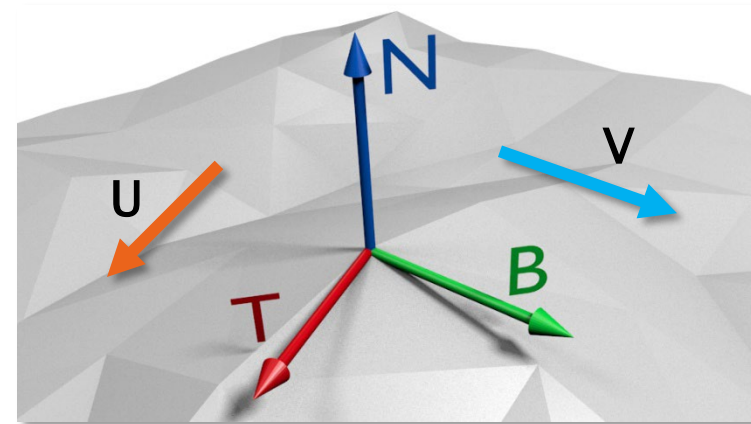
Cube with normal map
(Same map on each face)



Correct – After adjusting normal map from tangent to world space

Tangent to world space

- ▶ Need 3 vectors to convert a tangent space to a world space:
 - Normal of the surface
 - Tangent of the surface
 - Bi-tangent of the surface
 - All 3 are orthonormal
- ▶ Tangent points along texture's U direction
- ▶ Bi-tangent points along texture's V direction



Calculating tangent & bi-tangent

- ▶ **Tangent & Bi-Tangent** are vertex data
 - Like the position, UV and normal
- ▶ **Tangent** calculation:
 - Based on UV's of each triangle
 - Done when loading model in C++
 - Store in vertex buffer along with normal, UV, etc.
- ▶ **Bi-tangent** calculation:
 - Cross the normal and tangent
 - Can be pre-calc'd and stored, or calculated in shader

Calculating tangents in C++

- ▶ What does the tangent tell us?
 - Which way does the texture's u direction point?
 - In 3D space (as it lays on the triangle)
- ▶ Bi-tangent is the same, but for v direction
- ▶ Derivation and sample code:
 - <http://foundationsofgameenginedev.com/FGED2-sample.pdf>
 - See listing 7.4 in section 7.5 (page 9 of the PDF)

I have some vectors – Now what?

- ▶ Form a 3x3 matrix
 - The “Tangent / Bi-tangent / Normal” (TBN) Matrix
 - Converts (rotates) from tangent to world space
- ▶ Sample the normal map
 - Don’t forget to unpack the normal!
- ▶ Transform unpacked normal by TBN matrix
 - Puts the normal into “world” space
 - Like the rest of the model’s normals

Tangent to world space – Vertex Shader

- ▶ Already have this:

```
output.normal = mul((float3x3)worldInverseTranspose, input.normal);
```

- ▶ Similar step for tangent:

```
output.tangent = mul((float3x3)world, input.tangent);
```

- ▶ Bi-Tangent can be calculated in pixel shader
 - Simple cross product


Tangent to world space – Pixel Shader

```
// Unpack normal from texture sample – ensure normalization!
float3 unpackedNormal = normalize(normalFromTexture * 2.0f - 1.0f);

// Create TBN matrix
float3 N = normalize(normalFromVS);
float3 T = normalize(tangentFromVS - dot(tangentFromVS, N) * N); // Orthonormalize!
float3 B = cross(T, N);

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

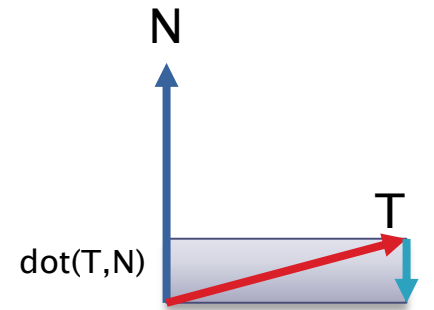
// Transform normal from map
float3 finalNormal = mul(unpackedNormal, TBN);
```



Orthonormalize

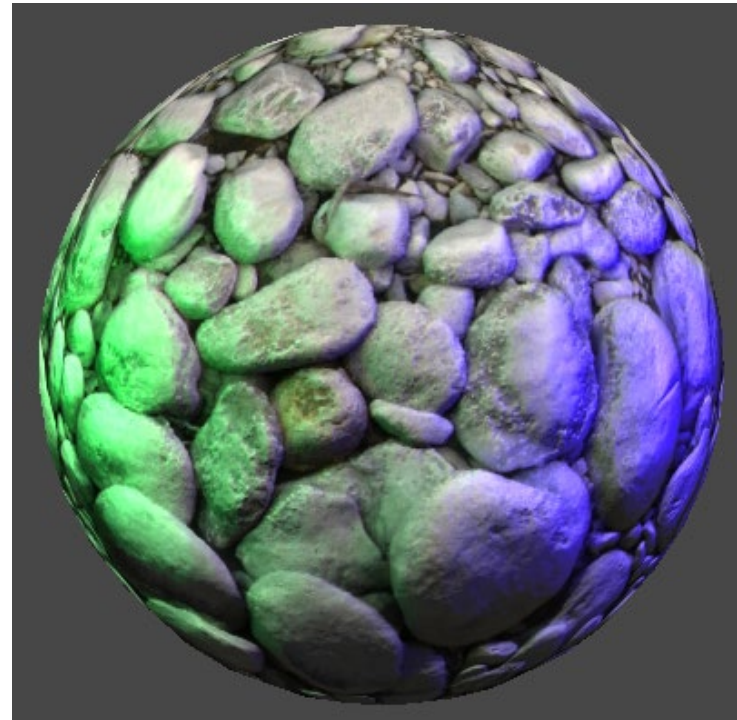
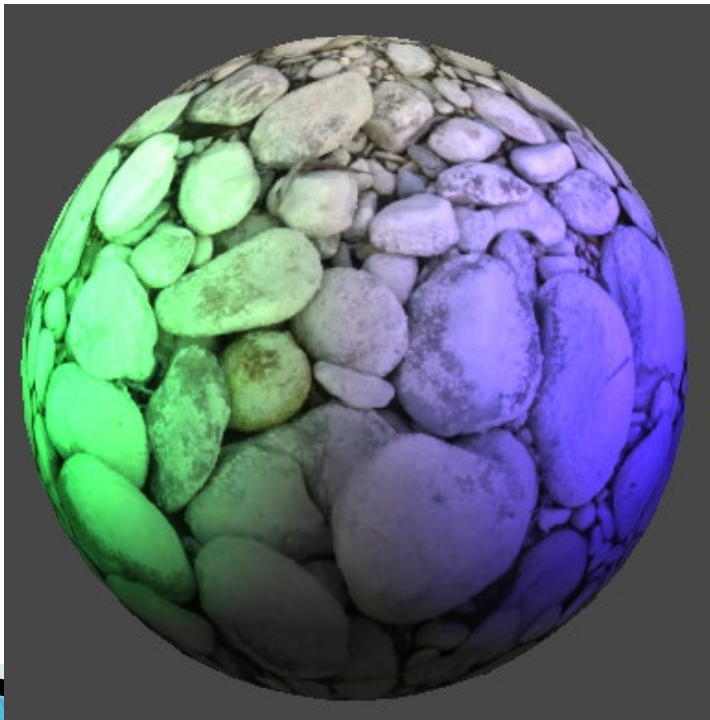
- ▶ Interpolation from VS to PS requires vector re-normalization
- ▶ Results may no longer be ortho-normal (90 degrees apart)
- ▶ We can perform an orthonormalization to fix this
 - Subtract a portion of the normal from the tangent to “bend” it
 - That “portion” defined by the dot product

```
// Gram-Schmidt orthonormalize (make tangent exactly 90° from normal)  
float3 T = normalize(tangentFromVS - dot(tangentFromVS, N) * N);
```



Final steps

- ▶ You now have a per-pixel normal – in world space!
- ▶ Use the same lighting calculations as before
- ▶ Impress your friends!



Making Normal Maps

- ▶ Usually created by an artist
 - Rendered from high-poly version of model
 - Applied to low-poly version
- ▶ Can be generated from surface color textures
 - Note: *Not* how real normal maps for games are made!
 - Fine for a quick and dirty map
 - Photoshop: Filter > 3D > Generate Normal Map

Generating in Photoshop

- ▶ Options and live preview!
 - Filter > 3D > Generate Normal Map

