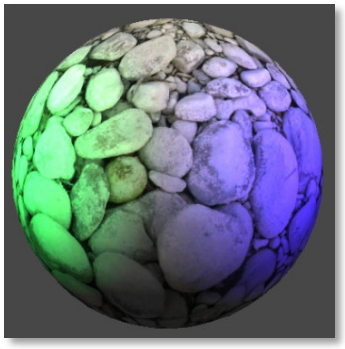
Normal Mapping

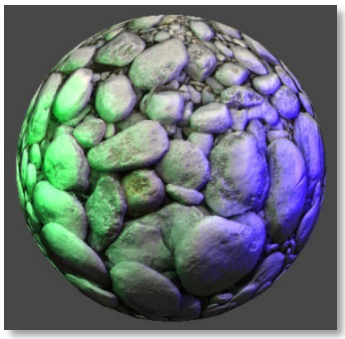
Using textures to apply per-pixel normals

Lighting & Textures

To make our geometry appear three dimensional, we use lighting calculations to apply shading. We can also apply textures to the surface of that geometry to give it more detail. Combining these techniques is trivial: the texture is simply used as surface's color.



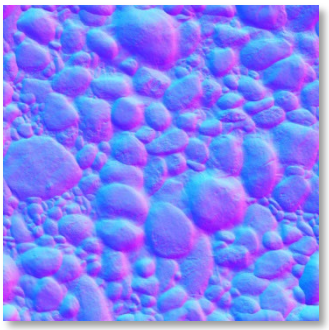
Unfortunately, no matter how much detail is in a texture, all of the lighting information comes from the geometry itself. A low-poly surface with a highly-detailed texture will still be lit like a low-poly surface.



In the top right image, you can see this phenomenon quite clearly. The sphere mesh has an image of rocks (above) as its texture. Instead of looking like the sphere is made of rocks, it looks more like an image of rocks has been painted onto a perfectly-smooth sphere.

In contrast, in the bottom right image, it looks as if there are individual rocks comprising the sphere. This is the exact same mesh – a smooth sphere – and the exact same rock texture. The difference is in the lighting: the normals use in the shading process are no longer only from the geometry. Instead, an extra texture holding *surface normal* information is used along with the geometry normals. This texture is known as a *normal map*.

Normal Maps



A normal map is a texture containing normals, as seen to the right. The red, green and blue channels in the image correspond to the x, y and z directions of our normals (with a slight modification).

Packing and Unpacking Normals

Standard image formats only store their color channels as positive values which, when converted to floats, are in the range [0, 1]. However, each component of a unit normal in 3D space could be in the range [-1, 1]. Any software that creates a normal map goes through a process of *packing* the normals into the [0,1] range. This is quite simple: add 1 to each component, then divide each by 2.

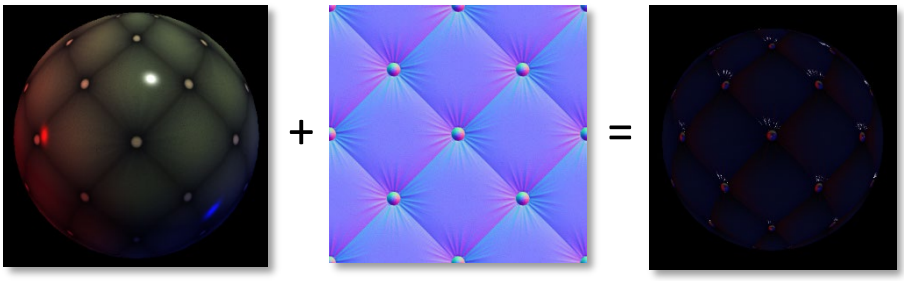
$$packedNormal = (normal + 1)/\,2$$

Whenever we sample from a normal map, we must *unpack* it by reversing this math (and normalizing):

$$normal = normalize(packedNormal * 2 - 1)$$

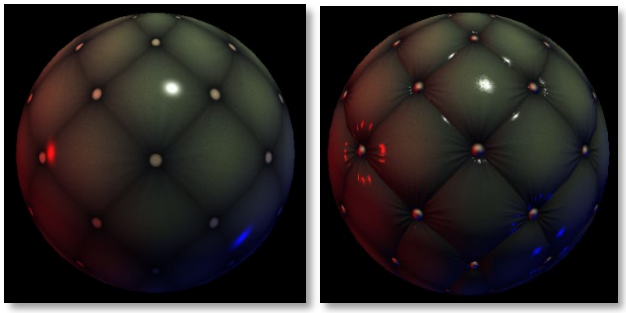
Using Normals from Normal Maps

So, we just sampling this texture, unpack and use this new normal during lighting? Unfortunately, it's not *that* easy. Below (left) is a sphere with a cushion texture on it. In the right image, the unpacked normal from the normal map is used *instead of* the geometry's normal.



As you might suspect, this is not correct. We cannot just *replace* the geometry's normal. If the texture is on the *right* side of the object, the normals should generally point to the *right*. However, if the same texture is on the *left* side of the object, the normals should generally point to the *left* instead. Thus, we cannot rely on the normal map normals *alone*; we have to orient them to match the geometry!

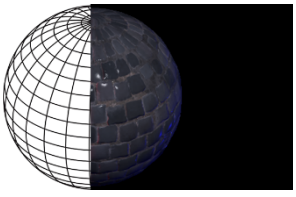
We'll dive into the details of proper normal mapping in a moment, but here's what it should look like:



What About More Triangles?

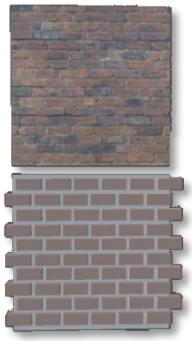
Before we get into the nitty gritty of properly using a normal map, it's worth discussing the alternative. If the problem is a lack of geometric detail, can't we just have more geometry? Why not just add more triangles? While it's not impossible to simply use higher-poly models, there are several issues.

First, each vertex needs to pass through the vertex shader. Even though these are handled in parallel, if you have more vertices than GPU cores it will take more time to transform them all to screen space. More complex vertex work, as with skeletal animation, will take even longer.

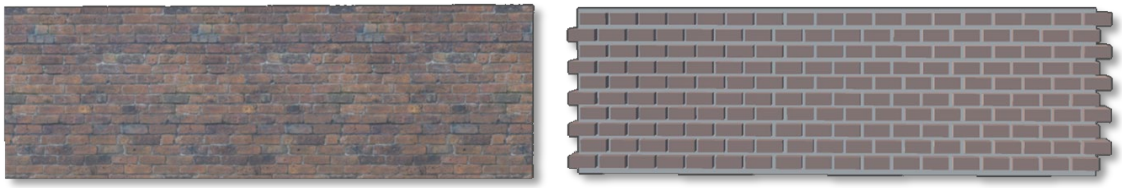


Additionally, if part of the mesh hangs off the edge of the screen, we still need to process each and every vertex to make this determination. The same is not true for pixels; the pixel shader does *not* run for pixels outside the viewport. Similarly, if a mesh is highly-detailed but extremely far away, we may end up processing more vertices than pixels!

Another issue with high-poly models has to do with scaling. As an example, take the two "brick walls" to the right, each of which takes up *approximately* the same number of pixels (overhanging bricks aside). The top one is simply two triangles with a texture. The bottom is created with actual brick geometry (and could also have a texture if we wanted). The advantage of the bottom version is that we have the geometry, and therefore the normals, of every single brick.



But what happens when we want to make these walls three times wider? Scaling the top wall is easy: scale the mesh and the corresponding uvs to repeat the texture. To make the bottom wall 3x wider, we need three times as much geometry! If we instead applied a normal map to the first wall, we could have highly-detailed normals without any extra geometry, making it far more scalable.

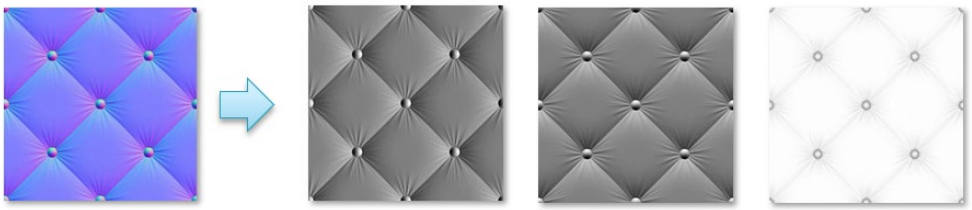


Tangent Space vs. World Space

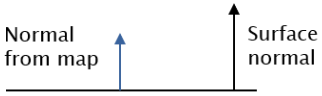
Back to the details of normal mapping: why can't we just use normals straight from the normal map? Because they're not in *world space*, as our light vectors and geometry normals are. Normals in a normal map are stored in *tangent space*. To use them for lighting calculations, we first need to transform them.

Tangent Space

The colors in a normal map generally appear magenta, purple and cyan. This is due to the meaning of each color channel. The red channel, corresponding to the x component of the normal, tells us whether the normal points left or right. The green channel, or y, means up or down. The blue channel, or z, means *away from the surface*. Thus, the blue channel of most normal maps is almost entirely 1.0, or very close to it, since normals always point away from the surface.

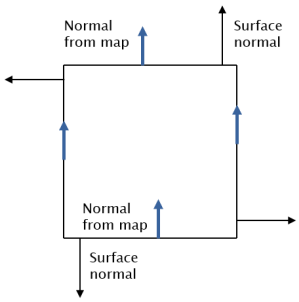


What do "left", "right", "up", "down" and "away" mean? If we sample a pixel from a normal map and, after unpacking, get the vector (0, 0, 1), this normal should point directly *away from the surface*. Sticking straight out of the geometry. That's in *tangent space*.



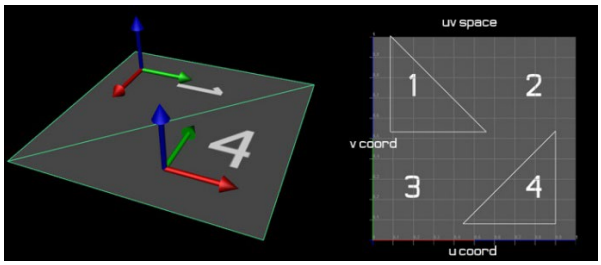
However, our lighting calculations assume vectors are in *world space*. So, what does the vector (0, 0, 1) mean in world space? Along the world's Z axis. Is that actually the way the geometry is facing? If our model's surface happens to be facing that direction at this exact point, everything matches!

But what if some of the sides of the model *aren't* facing the Z axis? Sampling the same pixel in the normal map gives us the same vector, (0, 0, 1), which always means *Z axis*. That probably isn't correct for most of our triangles!



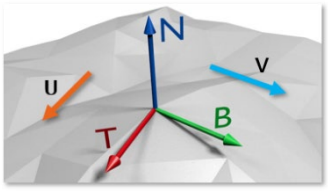
Tangent Space to World Space

We convert normals from a normal map into world space by rotating them to match the orientation of the triangle they're on. This means not only do we need to take into account which direction the triangle is facing (its normal), but we also need to know the *orientation of the texture* on that triangle.

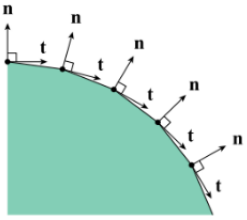


As seen above, the UV mapping on each triangle may be wildly different. "Up" on the texture doesn't always face the same way! We need to know how the texture is oriented on *each and every triangle*. Since we store vertices instead of triangles, we'll really need this information for *each vertex*.

We need two vectors in 3D space that match the U and V directions from the texture at each vertex. These vectors are known as the tangent and bitangent, respectively. Tangents point *along* the surface, in the same direction as the U dimension of the texture. Bitangents also point *along* the surface, but following the texture's V dimension.



These must be orthogonal (perpendicular) to each other, and orthogonal to the geometry's normal. These three vectors together (normalized, of course) form an *orthonormal basis*, called the *tangent frame*, which we can use to rotate the normal map's normal into world space.



> *How do these three vectors help us rotate the normal map's normal? Remember that we can rotate a 3D vector using a 3x3 rotation matrix.*
>
> *The rows of a rotation matrix represent the axes of a coordinate system. The tangent, bitangent and normal give us the three axes of such a coordinate system: the normal points away from the surface and the tangent & bitangent point along the surface, but oriented to match the U and V directions of the normal map. Exactly what we need!*

Calculating Tangents

We'll need to calculate one or both of these vectors and store them along with the rest of our vertex data. This can be done as we load our meshes, before sending them to the GPU. This will require us to store more one or both of these vectors with each vertex our meshes, which will necessitate an update to our Vertex definitions in both C++ and HLSL.

Do we calculate and store both new vectors? Or just one? If we store just the tangent, we can calculate the bitangent "live" in our shaders with a simple cross product, which will work quite well.

> *While storing both may seem like it would save us some work, we'll still need to apply the entity's world matrix to these vectors in the vertex shader and re-normalize them in the pixel shader, which is perhaps even more work than just calculating a cross product.*

Ok, so how do we calculate tangents? This sample chapter from the book Foundations of Game Engine Development, Volume 2: Rendering contains the full derivation of the math to do exactly that, including sample C++ code!

> *I'll be **providing tangent-calculating code** compatible with our current mesh class in a future assignment, allowing you to drop it right into your existing code base with very few changes.*

The gist of the tangent calculation is that we first calculate vectors along two edges of a triangle. We then calculate how the U and V coordinates change along those same triangle edges. We can use this information to figure out the tangent and bitangent with the following equations:

$$edge_1 = (u_1 - u_0) * tangent + (v_1 - v_0) * bitangent$$

$$edge_2 = (u_2 - u_0) * tangent + (v_2 - v_0) * bitangent$$

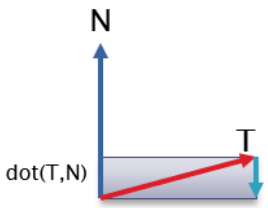
We can then solve for tangent and bitangent (or, in our case, just the tangent). Again, see the sample chapter link above for the full derivation, and a future assignment for code compatible with our engines.

One last thing to note: we will need to perform a *orthonormalization* step at the end to ensure the vectors are, in fact, orthogonal and normalized, or our final lighting calculations will be imperfect. If we assume one of the vectors is ideal (the normal), we can "bend" or "push" the other vector (the tangent) just enough so that it is exactly 90 degrees from the first.

The following pseudocode applies the Gram-Schmidt orthonormalization process:

```
tangent = normalize(tangent - dot(tangent, normal) * normal);
```

It works by subtracting from the tangent in the direction opposite the normal, by an amount denoted by the dot product between the two.



Converting From Tangent to World Space

Now that our vertices store both a normal and a tangent, we need to get them down to the pixel shader. The vertex struct in the vertex shader needs to be updated to include a tangent (matching the new C++ definition). The VertexToPixel struct in both shaders also needs to be updated to pass that tangent vector through the rasterizer.

Just as we've done with the normal, we need to rotate the tangent to account for the entity's overall rotation. Unlike the normal, however, you'll use the *world matrix* instead of the *inverse-transpose-world-matrix*, as non-uniform scales won't result in incorrect tangents (after normalization). Be sure to normalize the vectors on the way out of the vertex shader.

Down in the pixel shader is where the magic happens. See below for the pseudocode of this process.

You should already be renormalizing the normal from the vertex shader (due to the rasterizer's linear interpolation). You'll also need to orthonormalize the tangent here again (damn you, rasterizer!) Lastly, cross those two vectors to calculate the bitangent. Create a 3x3 matrix from these vectors where the tangent represents the X axis, the bitangent represents the Y axis and the normal represents the Z axis. This is the aptly named TBN matrix! Rotate your unpacked (and normalized) normal from the normal map and use that result in your lighting calculations.

```
// 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);
```

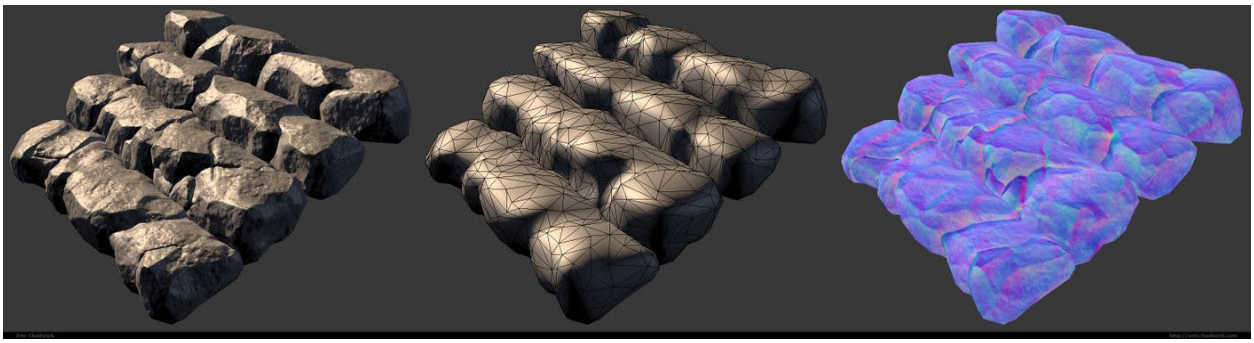
All in all, not a ton of code!

Obtaining (or Creating) Normal Maps

In general, you'll need a normal map that corresponds to a particular surface color texture. Mixing and matching normal maps and color maps doesn't usually work too well. Most websites that distribute free texture assets include normal maps these days, so if you're looking for any old texture, check for a corresponding normal map!

One common workflow for artists generating normal maps is to create a high-poly sculpt of their model, including every wrinkle, scar, skin pore and other minute detail. This highly detailed surface is used to

bake (create) a normal map containing these tiny details. Then that normal map is applied to a low-poly version of the model. This retains all of the lighting details even though there are fewer triangles.



However, if you need a normal map for an existing texture, there are programs that can generate them. Photoshop, for instance, has a “Normal Map” filter that can create a normal map from any image. It does this by looking at color differences in neighboring pixels and making a best guess as to how the surface would change. A large change in color means the pixels have different “depths” and therefore represent a very extreme normal.

> *Note that this is not the usual workflow for artists and will not result in accurate normal maps for most images! But it’s often good enough for simple demos and educational purposes.*