Sources

Half-Life 2 / Valve Source Shading by Gary McTaggart
http://www.moonpod.com/board/images/misc/D3DTutorial10_Half-Life2_Shading.pdf

Efficient Self-Shadowed Radiosity Normal Mapping by Chris Green
https://cdn.akamai.steamstatic.com/apps/valve/2007/SIGGRAPH2007_EfficientSelfShadowedRadiosityNormalMapping.pdf

Overview

RNM is a simple and straightforward way to encode irradiance in lightmaps for use with normal mapping:

- 3 constant basis vectors B1,B2,B3 in tangent space form an orthogonal basis.
- Irradiance is stored for all 3 basis into 3 separate RGB textures.
- For each bit of incoming light L along light direction D in tangent space,
  the light gets accumulated for each basis/texture separately:
  `LMi += L * dot(D, Bi)`

LM1 being the lightmap texture for basis 1, LM2 for basis 2, LM3 for basis 3.

To apply this at run-time, the shading normal N must be in tangent space:

```
W1 = dot(N, B1)
W2 = dot(N, B2)
W3 = dot(N, B3)
LM = LM1*W1*W1 + LM2*W2*W2 + LM3*W3*W3
```

Run-time algorithm

```
// lm[1|2|3] are the lightmap colors for the 3 basis vectors
// N is the tangent-space shading normal
// all vectors are in tangent-space
float3 RadiosityNormalMapping(float3 lm1, float3 lm2, float3 lm3, float3 N)
{
    const float3 Basis1 = float3( 0.904534,  0.000000, 0.426401);
    const float3 Basis2 = float3(-0.408248,  0.707107, 0.577350);
    const float3 Basis3 = float3(-0.408248, -0.707107, 0.577350);

    float3 weights;
    weights.x = saturate(dot(N, Basis1)); // without saturate, weights < 0 will
become > 0
    weights.y = saturate(dot(N, Basis2));
    weights.z = saturate(dot(N, Basis3));
    weights *= weights;
    float weightSum = dot(weights, float3(1, 1, 1)); // renormalization factor

    return (weights.x * lm1 + weights.y * lm2 + weights.z * lm3) / weightSum;
}
```

Issues with the first talk

The code presented on page 12 of the original PDF has the following issues:

- Using the dot products instead of the square of the dot products
- Not saturating the dot products: negative weights should be clamped to 0 but become non-zero positive weights instead
  - Clamping means that the sum of the weights isn't guaranteed to be 1 anymore, so renormalization is needed

The code above has the correct logic, unlike the code on page 12 of the original PDF, which uses:

```
return (dp1 * lightmap1) + (dp2 * lightmap2) + (dp3 * lightmap3);
```

In other words, the correct weights are the squares of the dot products, not the dot products.

To convince yourself, here's a really easy example to follow with very little math needed:

- Assume the shading normal is exactly half-way between the first 2 basis vectors: `N = normalize(B1 + B2)`
- Since basis vectors form an orthonormal basis, the angle between any 2 basis vectors is 90 degrees.
- The angles beteen N and B1,B2,B3 are therefore 45,45,90 degrees respectively.
- With `Wi = dot(N, Bi)`:

  ```
  SumW = cosd(45) + cosd(45) + cosd(90) = 2 * cos(45) + 0
       = 2 * sqrt(2)/2 = sqrt(2)
       ~ 1.4142...
  ```

  Well that's a bit of a problem as the sum of the trilinear weights should be 1.
- With `Wi = dot(N, Bi) * dot(N, Bi)`:

  ```
  SumW = cosd^2(45) + cosd^2(45) + cosd^2(90) = 2 * cosd^2(45) + 0
       = 2 * (sqrt(2)/2)^2 = 2 * (2/4) = 2 * (1/2) = 1
       = 1
  ```

  This works out.

In addition, the code of the original doesn't clamp the dot products ()

Alternatives

The angular resolution of RNM is extremely low, which makes it unsuitable for almost any use today. There are other ways to encode the irradiance over a hemisphere for lightmapping however:

- The main one is Spherical Harmonics (SH).

  Example: *Doom Eternal*[1] used 2-band SH.
- A few games have used Ambient Highlight Direction (AHD).

  Example: *The Last of Us*[2].

I have no knowledge of any game using Spherical Gaussians or wavelets for directional lightmaps.

1. *id Tech 8 Global Illumination* by Tiago Sousa
2. *Lighting Technology of The Last Of Us* by Michal Iwanicki

Why are directional lightmaps even needed?

If we had enough memory, we could just bake the final lighting of all static lights at full resolution into a single RGB texture. The problem is that in general, we just don't have enough memory for full-resolution lightmaps as they cannot be tiled.

Since lightmaps have a tendency to have low-frequency data, a lower spatial resolution is acceptable. However, if we try to bake the lighting using the normal maps, the low resolution of the bake means we lose all of the fine details. We therefore need a way to combine low-resolution lightmaps (unique texturing) with high-resolution material textures (repeated/tiled texturing) at run-time. The problem is that to be able to do normal mapping, some form of light direction data is needed in the first place.

Without any directionality, all that's available is a shading normal from the normal map and a radiant exitance value from the lightmap. You can write something like `C = LM * dot(N, float3(0, 0, 1))` assuming all the light in the lightmap came from the exact same angle as the surface normal, which is ridiculously wrong. All this is doing is applying a darkening factor that's stronger the more a shading normal is at a grazing angle. Parts of these dark silhouette edges should actually reflect more light when the micro-facts face a static light source, not less.

Using an existing path tracer for RNM baking

Blender doesn't give you access to the irradiance data during a texture bake. How can you get around that to bake RNM lightmaps?

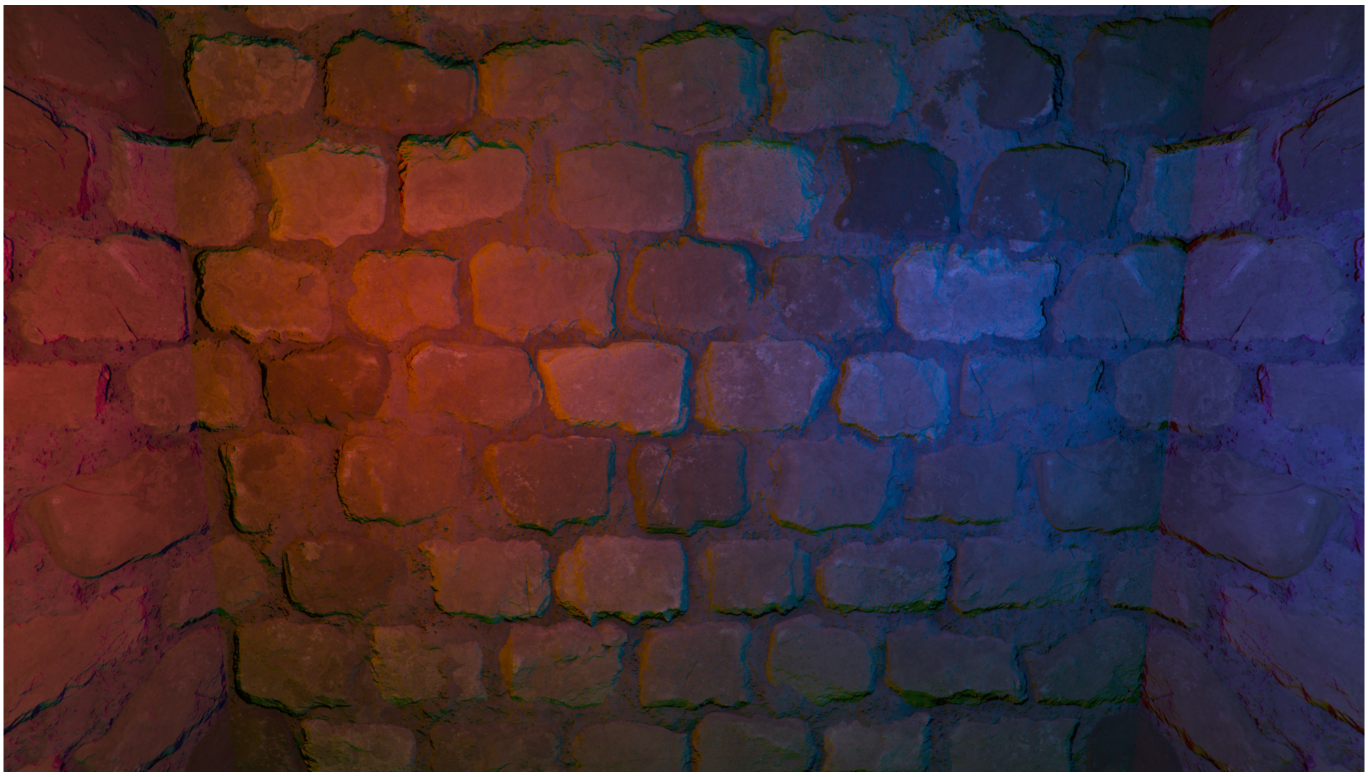
During encoding, we need to multiply the incoming light by the dot product of the light direction in tangent space and the basis vector: `L * dot(D, Bi)`. We can achieve this by feeding `Bi` to the renderer as the normal vector. In the renderer, the incoming light is multiplied by the Lambertian dot product, which we hacked to use the basis vector instead of a surface or shading normal vector.

The recipe for Blender:

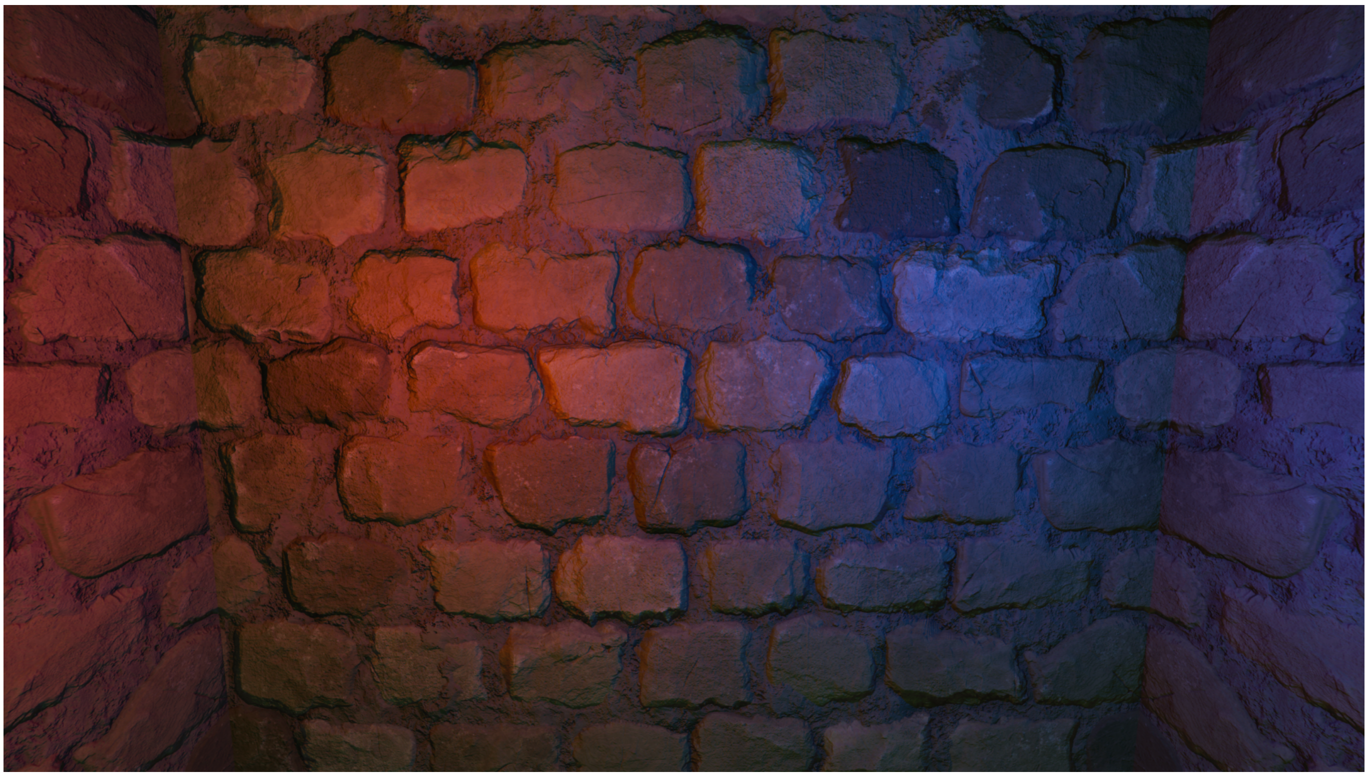
- Use Python code to programmatically dispatch a bake per object/mesh/sub-mesh.

- Run a bake per basis vector / lightmap texture: `bake_count = bake_objects * 3`.
- Make the material of the current bake target use the basis vector as a flat normal map.
  - All other objects must use their designated/correct material.
  - Because objects can reflect light onto themselves, this hack can negatively affect the baked lighting quality of large non-flat objects.
  - Why? A bake target object must use the basis vector normal to encode the irradiance correctly but the light that is reflected off itself first should have used the "real" normal map, not the basis vector normal.
- Larger bake objects mean:
  - fewer bakes/dispatches and thus better performance
  - more self-reflections and thus more rendering errors due to wrong normal being used

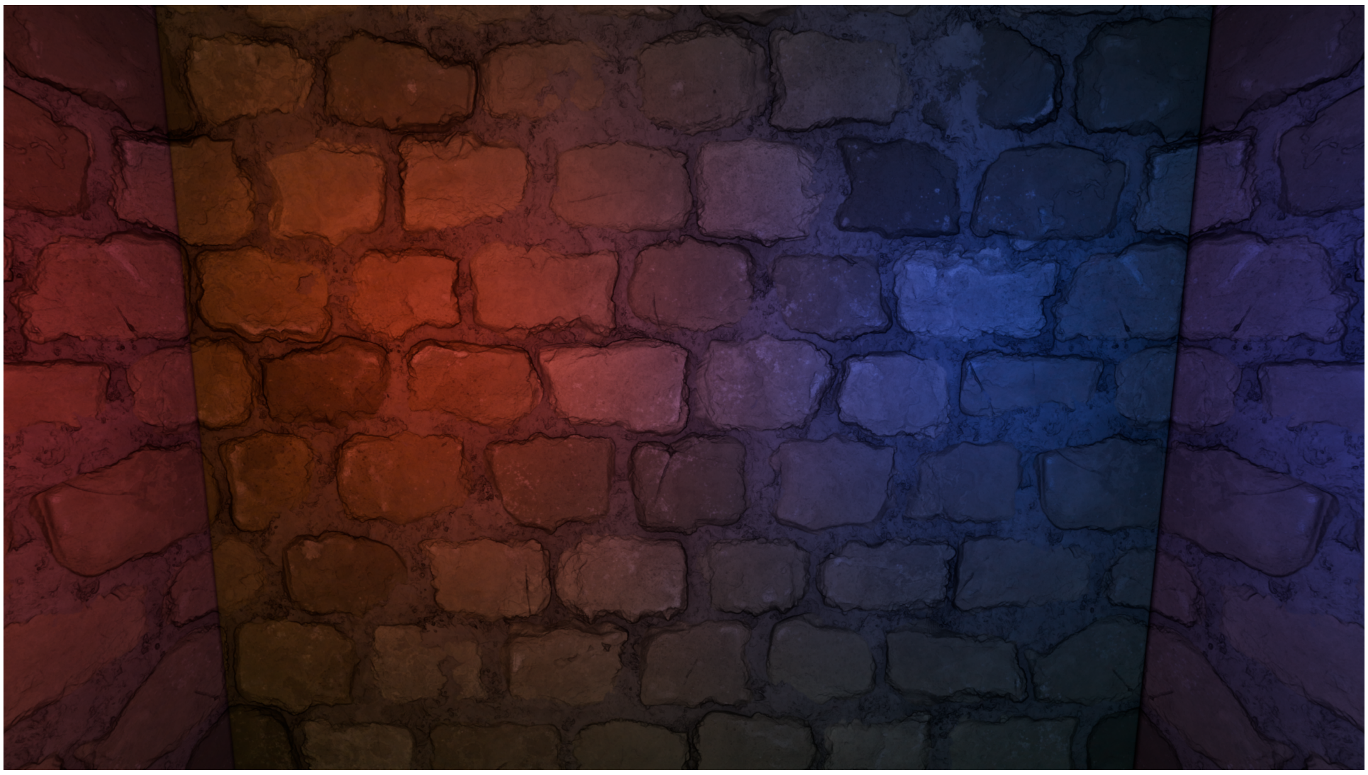
Blender Cycles



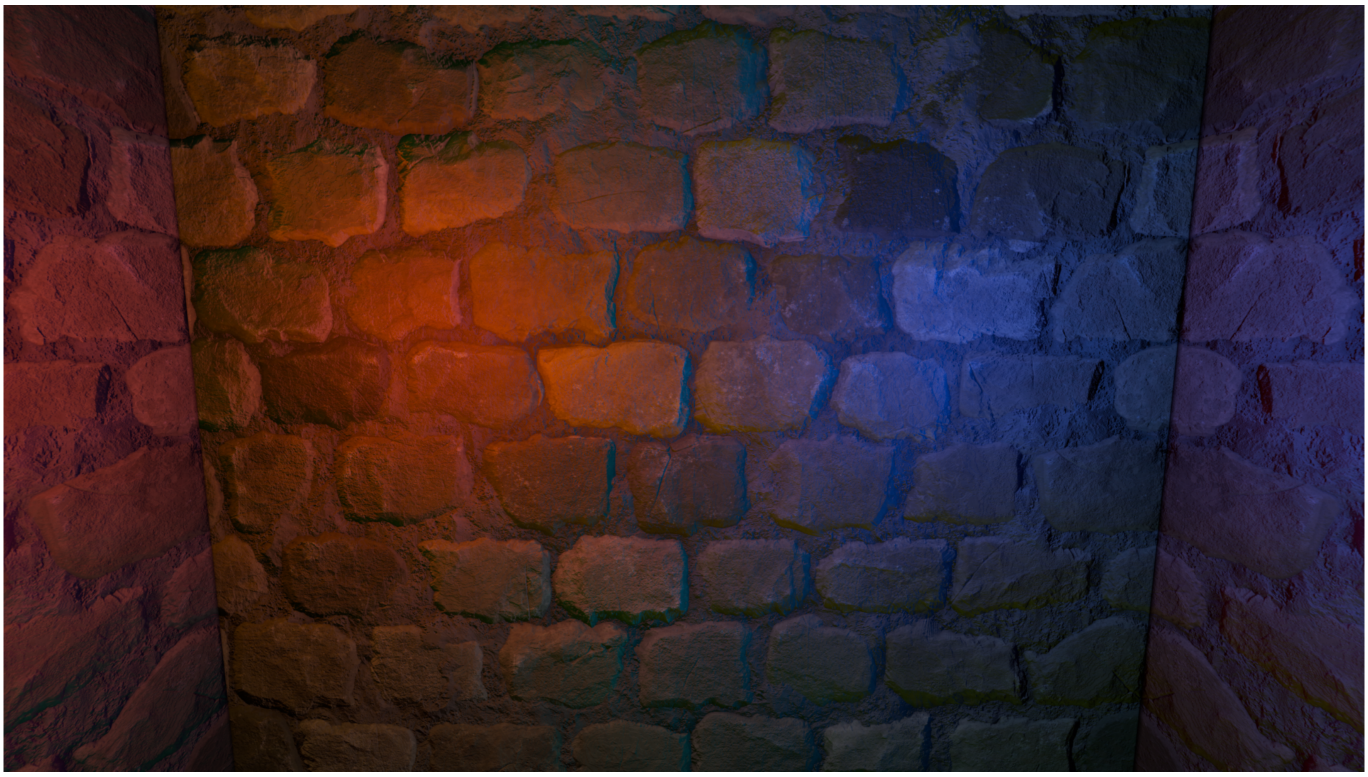
Blender Eevee



Baked with a single lightmap: `LM * dot(N, float3(1, 1, 1))`



Baked with a RNM lightmap



Self-shadowing

To have a better sense of depth, self-occlusion is needed. This can be solved by multiplying the per-basis weights (computed in the code above at run-time) by new per-basis directional occlusions factors (computed offline). This could of course be stored into more texture data but Valve did something different for *Half-Life 2 Episode 2*. They replaced the standard tangent-space normal maps with pre-computed lightmap blend weights. Since occlusion factors are normalized weights too, the original weights can be pre-multiplied by them in the new "ssbump" (self-shadowed bump) textures directly.

To compute the directional occlusion factors, one could do the followin for each basis and each texel:

- Define a 120-degree wide cone in the direction of the basis vector.
- Trace many rays inside the cone against the height map and gather the hit results.
- Compute a (weighted) average of all the hit results.