Lighting

...and of course, by lighting we mean shading.

Lighting Contents

1. Lighting Algorithms review
    a. Ambient + Diffuse + Specular
2. Light Mapping
    a. How to support precomputed lighting value.
3. Radiosity
    a. A more realistic lighting model.
4. Ambient Occlusion Mapping
    a. A means to create more realistic ambient lighting
5. Normal Mapping
    a. Providing additional surface detail without additional vertices.

Lighting Algorithm Review

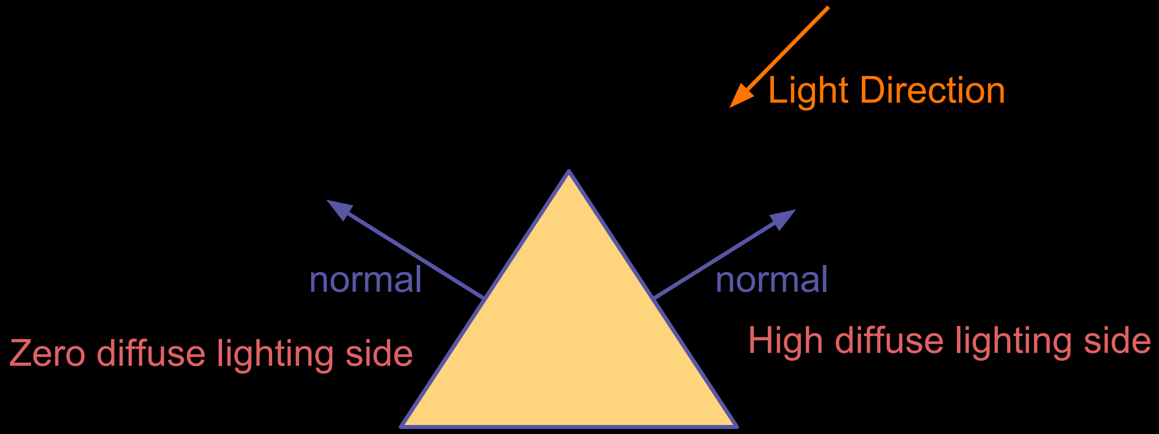
- We will support the three basic light types, directional, point and spot.
  - A Directional light is used to represent a light source that is so far away, that individual rays of light are effectively parallel, such as the sun.
    - Treating the sun as a point light may seem more correct, but the position values would be so large we would have precision issues storing the locations.
  - A Point is used to represent a light originating from a given point in space, and shines in all directions.
  - A Spot light is used to represent a directed light source, such as a flashlight or a ...spot light.

Properties of lights

- Each of our lights will apply three types of light, used to approximate what light actually does.
- return ambient + diffuse + specular;
    - Diffuse light represents the effect of a ray of light directly hitting the fragment.
    - Ambient light is a way to approximate the effect of bouncing rays of light hitting fragments that may not be hit directly.
    - Specular light is an approximation of how the fragment reflects the light. This is not bouncing rays lighting, but actually seeing the reflection on the surface of the light source.

Diffuse Lighting Algorithm

- Directional Light
  - `float nDotL = saturate(dot(normal, -lightDirection);`
  - `float4 finalDiffuse = nDotL * surfaceColor * lightDiffuseColor;`
  - nDotL will be highest when the normal and -lightDirection are parallel. When the normal and -lightDirection are at a 90 degree or greater angle nDotL will be 0.



Diffuse Lighting Algorithm

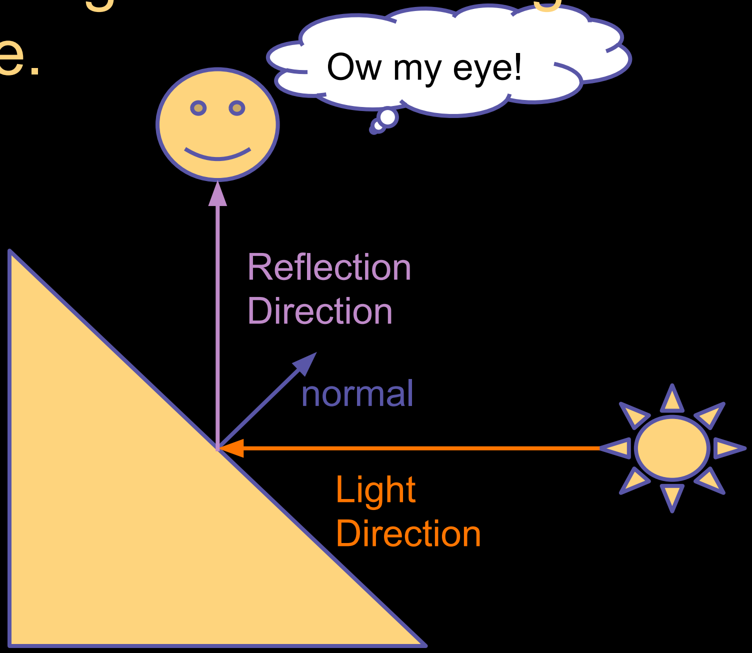
- Point and Spot lights do not cast parallel rays, so they do not use (have) lightDirection, instead we get a vector from the fragment to the light, typically called toLight.
    - We will want to save the length of this light for attenuation, but normalize it before we calculate nDotL
- Point and Spot lights diffuse algorithm
    - `float3 toLight = lightPosition - fragPosition;`
    - `float toLightLength = length(toLight); toLight /= toLightLength;`
    - `float nDotL = saturate(dot(normal, toLight));`
    - `float4 finalDiffuse = nDotL * surfaceColor * lightDiffuseColor;`

Ambient Lighting

- Ambient lighting is a simple, and far from perfect, approximation of the effect of bouncing rays of light.
  - We will improve the result in the Ambient Occlusion Mapping section of this presentation.
- Ambient lighting algorithm
  - `float4 finalAmbient = surfaceColor * lightAmbientColor;`
- Some ambient lighting models will include a global ambient light value.
  - We can also get global ambient lighting through any directional light.
  - In our deferred shading system, we can get global ambience by by clearing the buffer we use for rendering lights to our global ambience color.

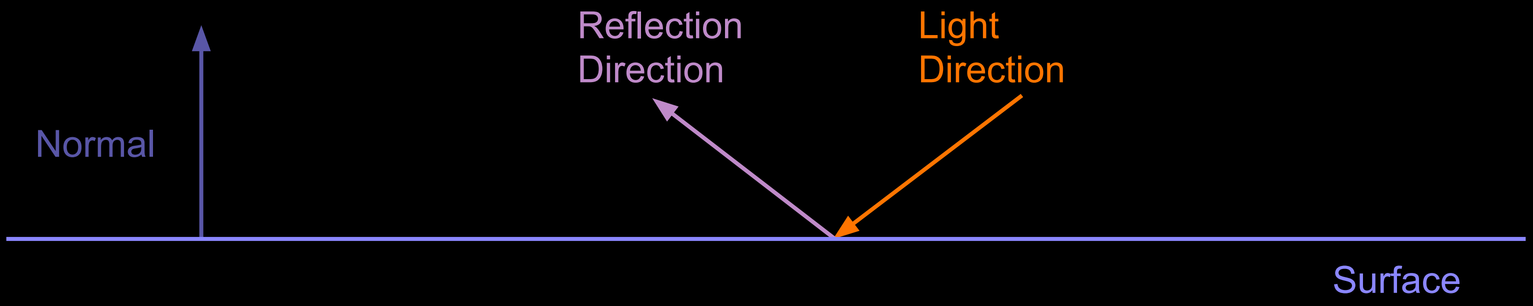
Specular lighting algorithm

- Where light would bounce off a surface and hit the viewer in the eye, is where the specular highlight will be the most intense. With that in mind, we begin by calculating the direction light would be reflected on a surface.



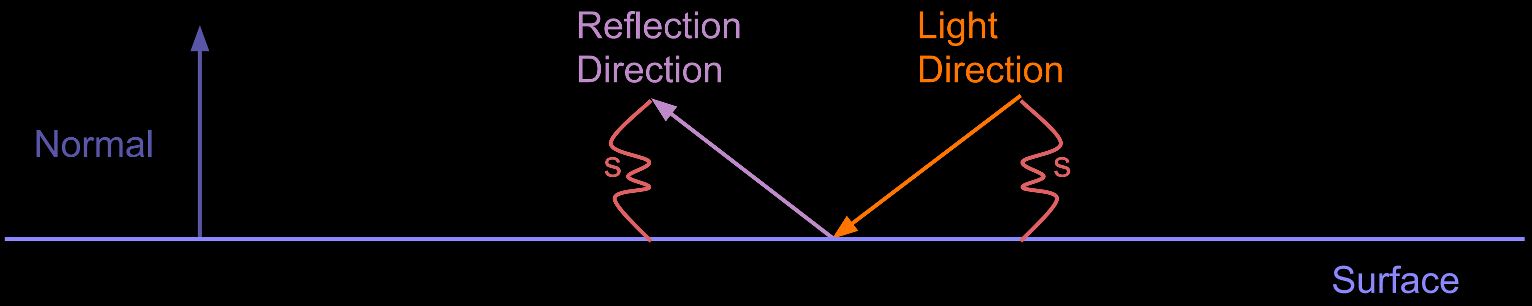
Reflection

- What we have:
  - Surface Normal
  - Light Direction
- What we want
  - Reflected light direction



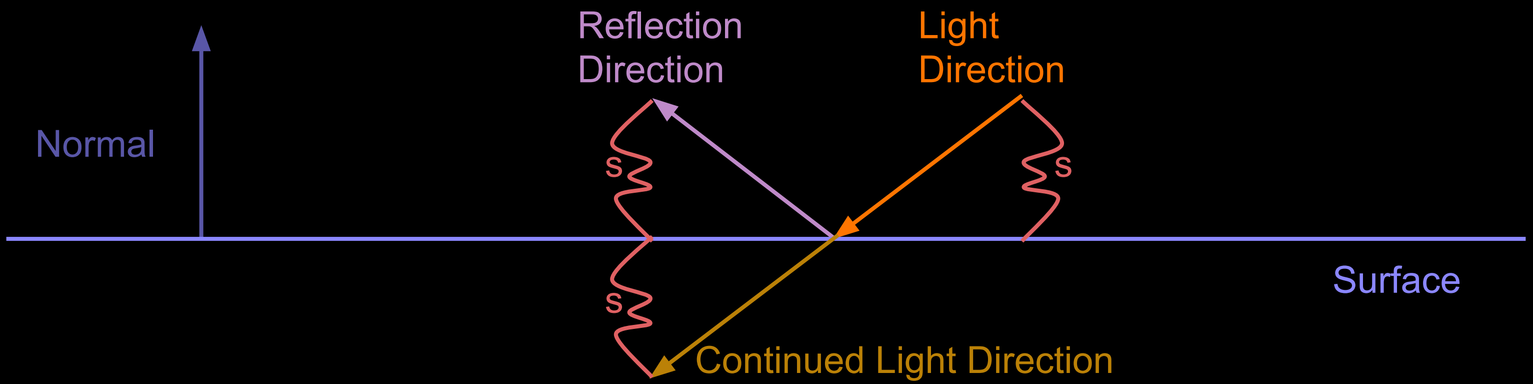
Reflection

- We begin by projecting our negative Light Direction onto our Normal.
  - s will be how far the negative Light Direction travels in our normal's direction.
  - s = Normal dot -LightDirection



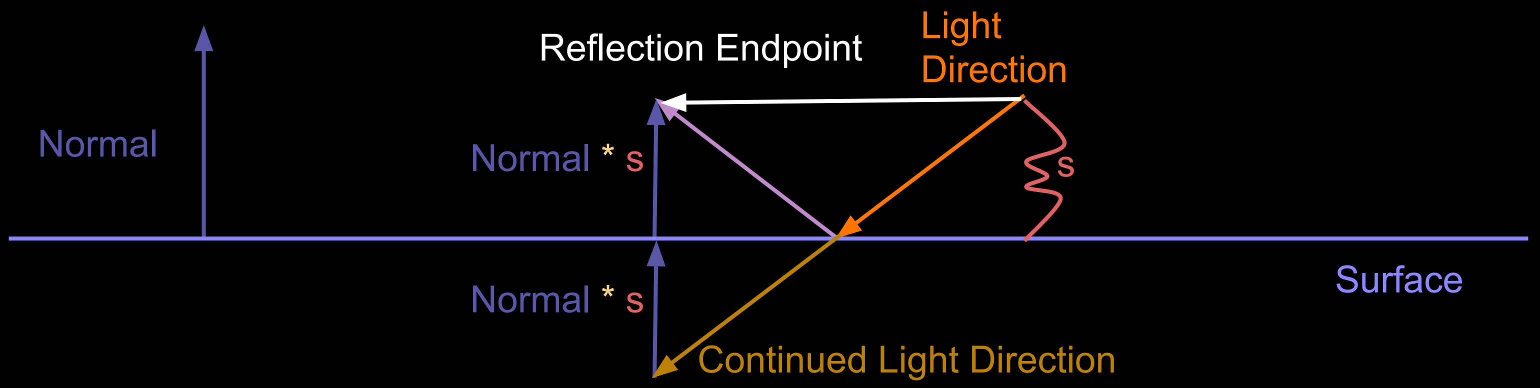
Reflection

- Next, we will find where the light would travel if a reflection did not occur.
  - Continued Light Direction = Light Direction + Light Direction
  - This new location will also be s units away from the plane of our surface.



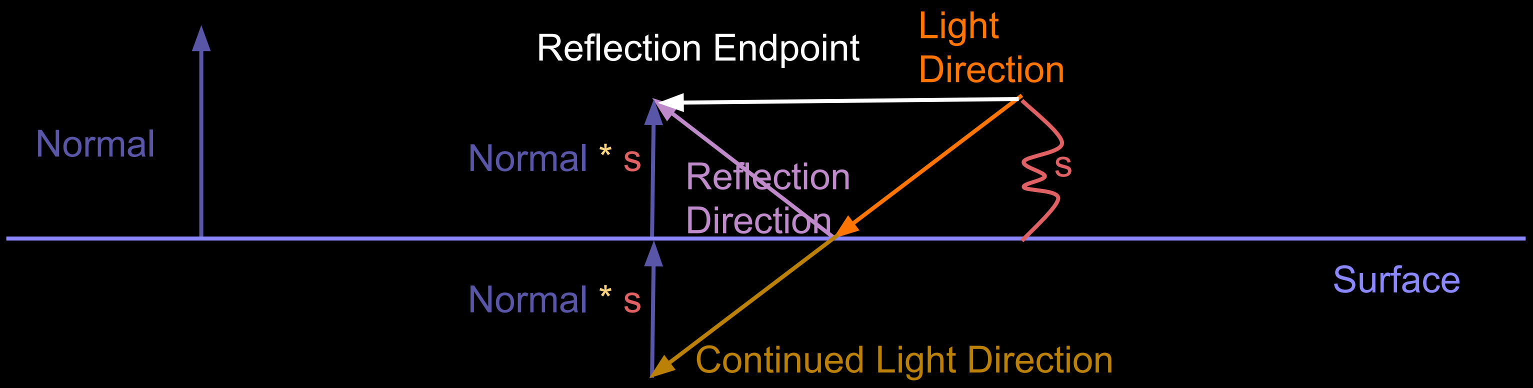
Reflection

- We now have enough data to calculate our Reflection Endpoint.
  - Reflection Endpoint = Continued Light Direction + Normal * 2s



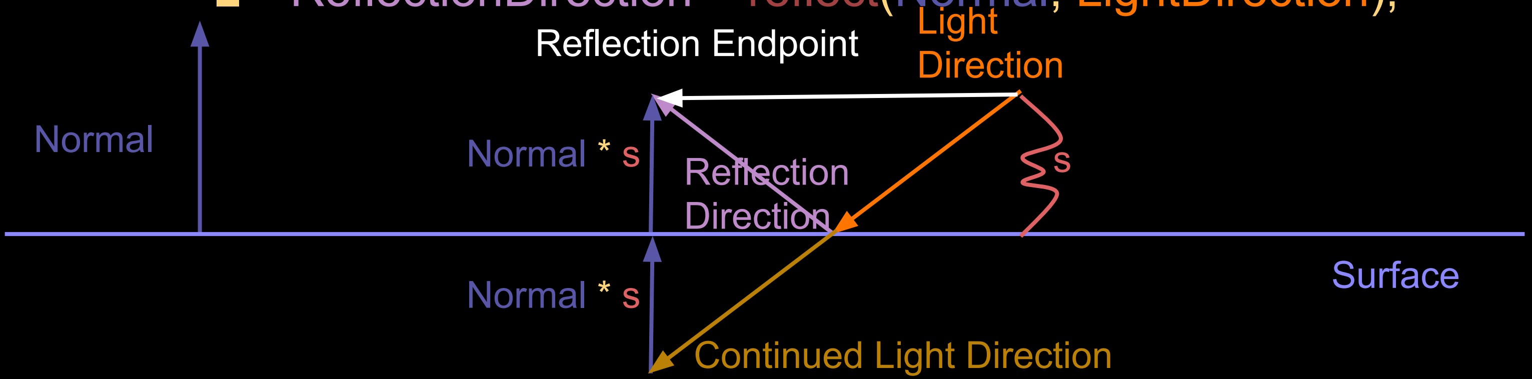
Reflection

- The Reflection Endpoint is **not** the Reflection Vector.
- Reflection Direction = Reflection Endpoint - Light Direction



Reflection

- Final formula:
  - Reflection Direction = Light Direction - ((Light Direction + Light Direction) + (Normal * 2 * (Normal dot -Light Direction)))
  - Or you could just call the HLSL intrinsic reflect method
    - ReflectionDirection = reflect(Normal, LightDirection);



Specular Formula 1/3

- float3 directionToCamera = normalize (camera position - fragment position);

- We need to calculate the direction the user is looking to see if we are shining the light in their eyes. Make sure to normalize this, as the distance the user is away should have no effect.
  - Our specular result will be scaled by attenuation

Specular Formula 2/3

- float specScale = specular intensity * pow(saturate(dot(reflectionDirection, directionToCamera)), specular power);

- We now have enough information to calculate how intense the specular highlight should be.
- Specular intensity and power should be properties of a light.

Specular Formula

- float4 finalSpecColor = fragment spec material * specScale * (nDotL > 0) * specularLightColor;
- The “fragment spec material” refers to how reflective the surface should be. In our engine all objects have a specular texture that defines their reflectiveness.
- We scale by (nDotL > 0) to prevent reflections on surfaces facing away from the light.

Attenuation

- The farther light has to travel, the less effect it will have on a surface. This is referred to as attenuation.
- This will only apply to lights with positions, such as point and spot lights.
- We typically support three types of attenuation, constant, linear and quadratic.

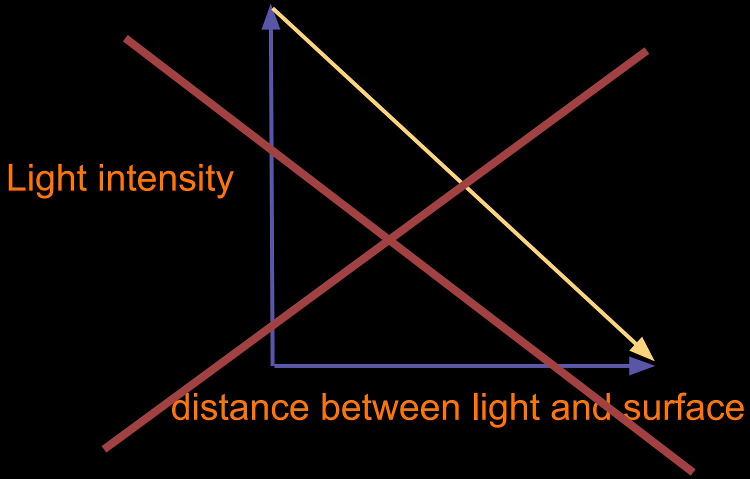
Constant Attenuation

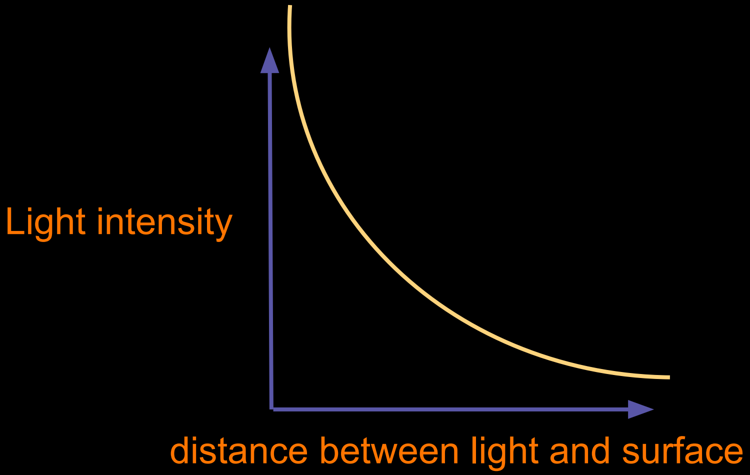
- Constant attenuation reduces the light intensity a constant amount, and is unaffected by distance.

- `float attenuation = 1 / constantAttenuationFactor;`

Linear Attenuation

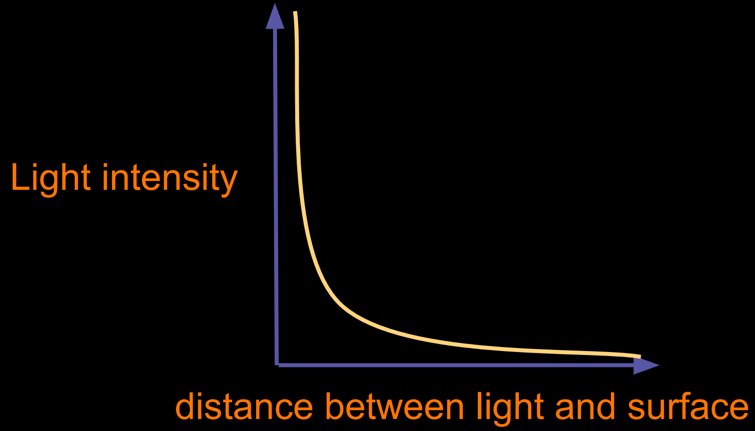
- Linear attenuation creates a linearly increasing attenuation factor, not linear reduction in light intensity.
- attenuation = 1 / (linearAttenuationFactor * lightDistanceToFragment);





Quadratic Attenuation

- Quadratic attenuation reduces light intensity at a quadratic rate. Of the three, this model best matches what light actually does, but light intensity never reaches 0, which can be problematic.
- `float attenuation = 1 / (quadAttenuationFactor * lightDistanceToFragment * lightDistanceToFragment);`



Attenuation

- We can combine the three types of attenuation into one formula.
- float attenuation = 1 / (constant + linear * distance + quadratic * distance * distance);
- Our deferred shading renderer will render a sphere for a point light. We will want the edges of the sphere to be where attenuation reaches 0...

Attenuation over a defined range

- Given a fixed range, or radius, we can ensure attenuation maps smoothly.
- We start with our general attenuation formula.
- float attenuation = 1 / (constant + linear * distance + quadratic * distance * distance); //Map to range next
- float distanceRatio = distance / range;
- float dampeningFactor = saturate(1 - pow (distanceRatio, power));
- attenuation = attenuation * dampeningFactor;
- “Power” adjusts the shape of the dampening, where power = 1 will be linear, power = 2 will be quadratic, ect. Power = 2, is the typical nice looking case.

Attenuation Factors

- So many factors, what do I set them all to?
- In practice, whatever looks the best and works for you.
- To model something close to a realistic average attenuation of light, Constant = 1, Linear = 2/Range and Quadratic = 1/Range * Range.
- You will likely need to adjust, but these make for a good starting point.
- Check out the attenuation excel sheet at:
- https://docs.google.com/spreadsheet/pub?key=0AkOa1G1tyhlKdDhmbHZEZlNuSFZ5cmVCSlhQR3FDb1E&output=xls

Light Mapping

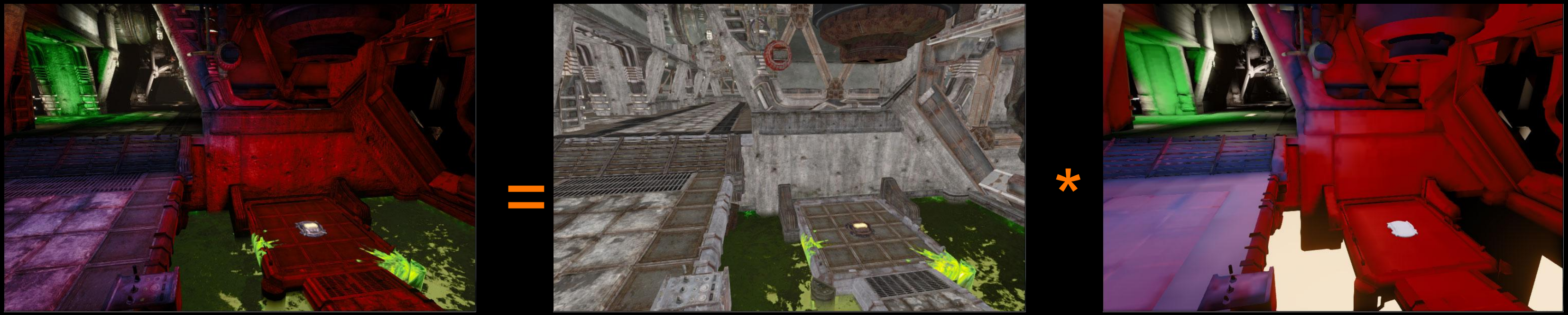
The fastest lighting algorithm around.

Light Mapping

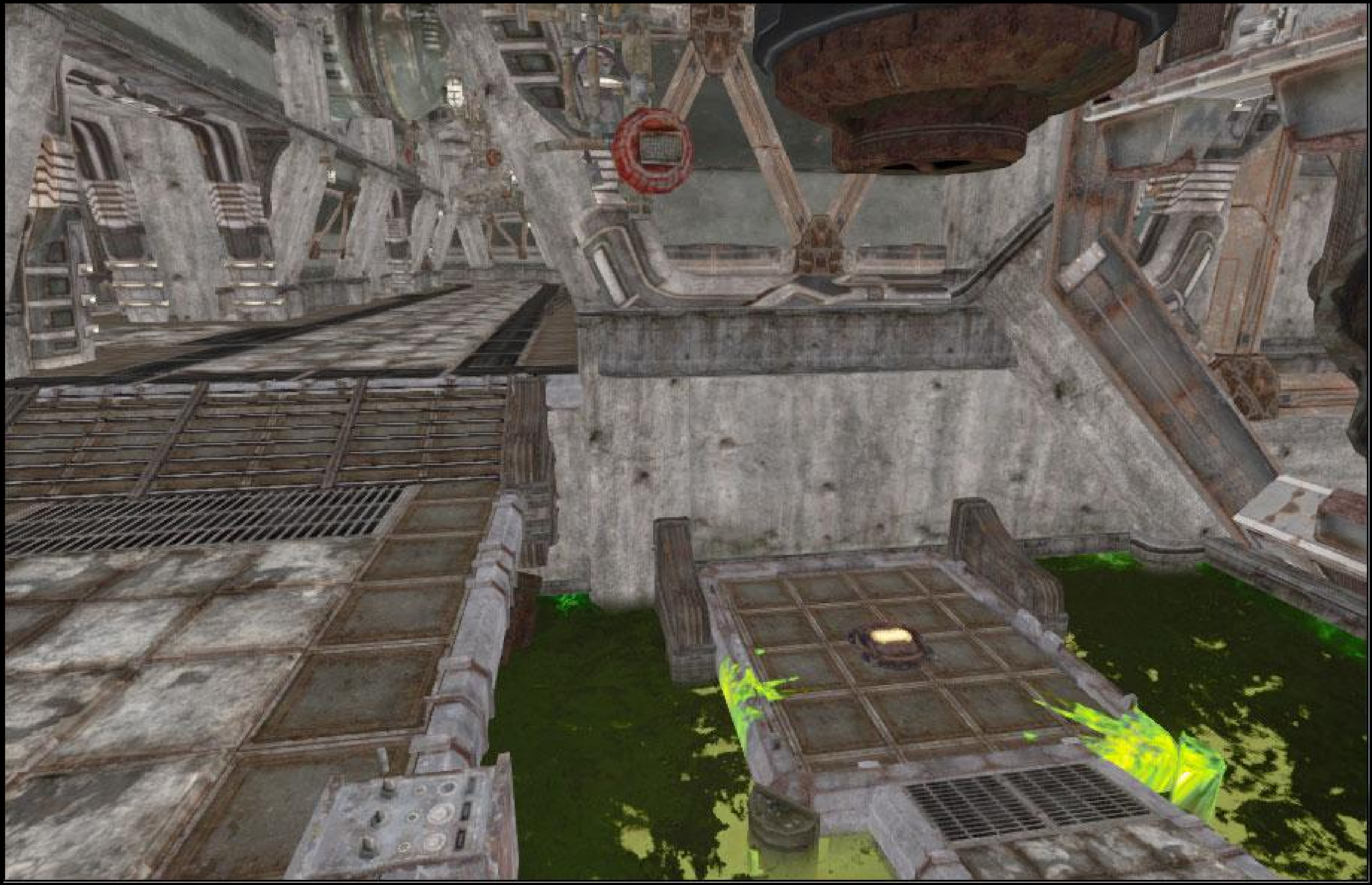
- There are other lighting models that can create more realistic results, but may not be fast enough to process for real-time rendering. Radiosity is one such model.
- These lighting models can be precomputed by an artist or tool running an expensive algorithm, but how can the results be applied in real-time?
- The answer is Light Maps.

Light Map

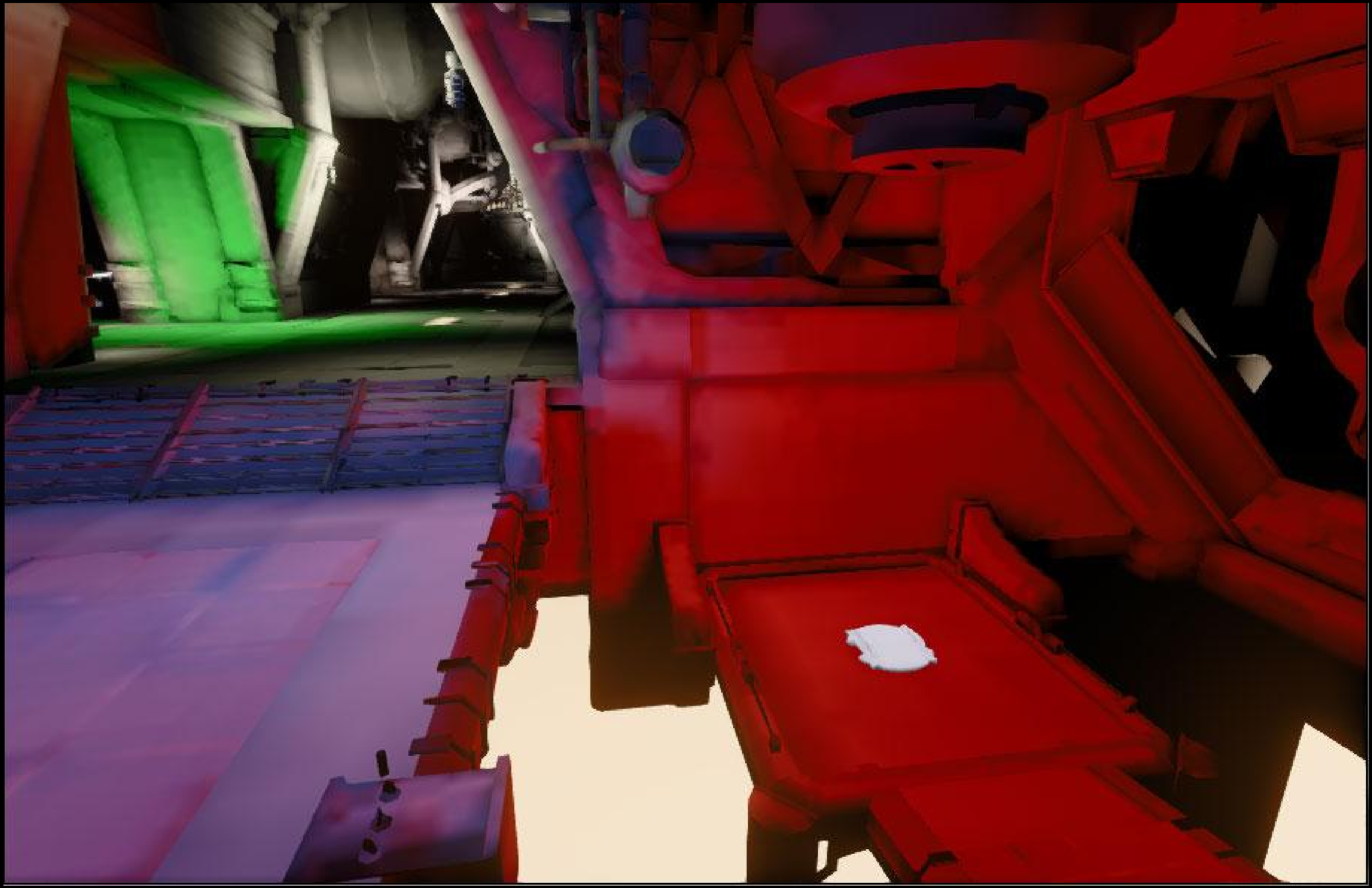
- A light Map is simply a texture containing lighting values for a given surface.
- float4 finalColor = surfaceColor * lightMapColor;



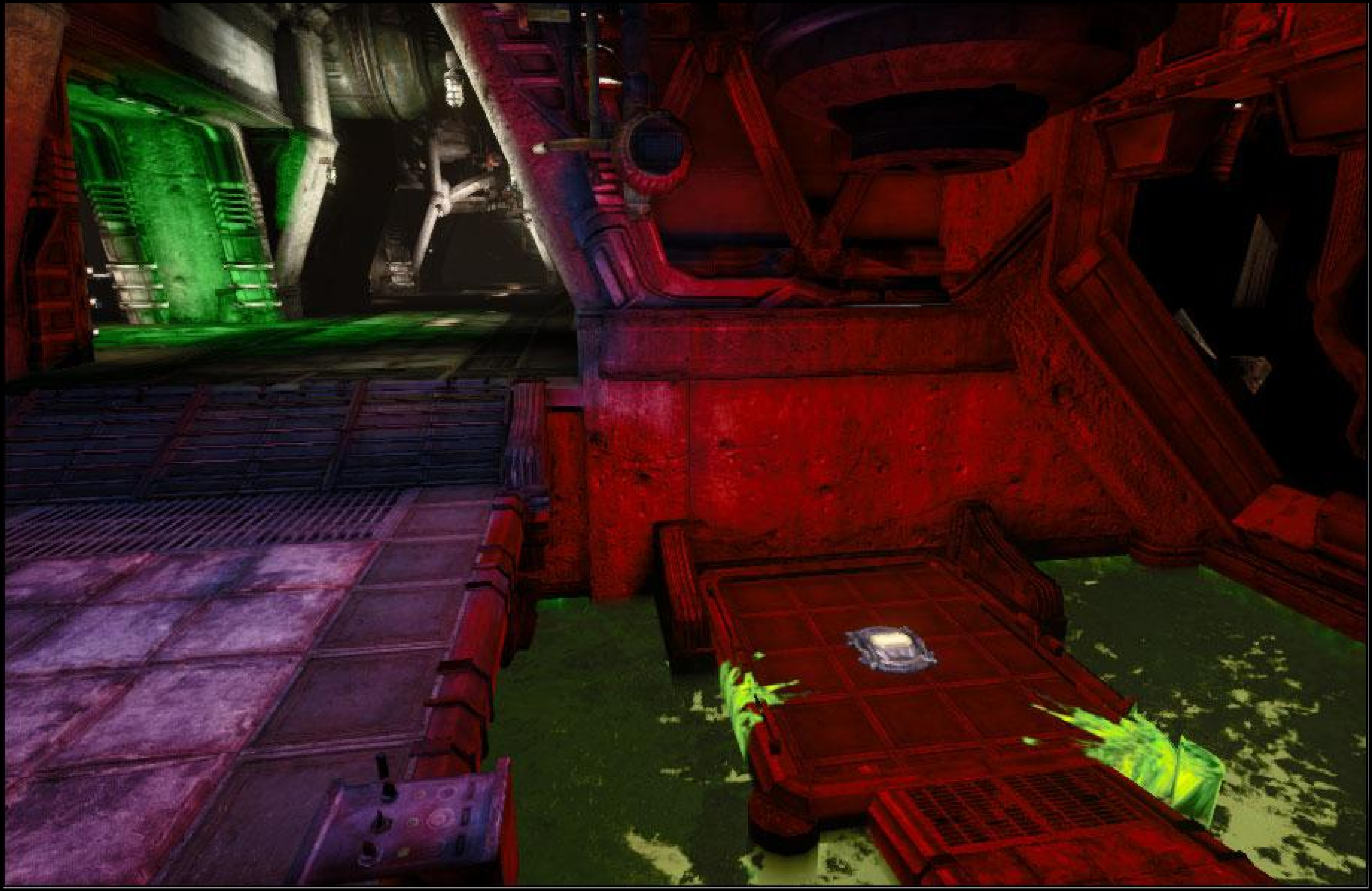
Light Mapping - Surface Colors



Light Mapping - Light Map Colors



Light Mapping - Final Colors



Light Mapping

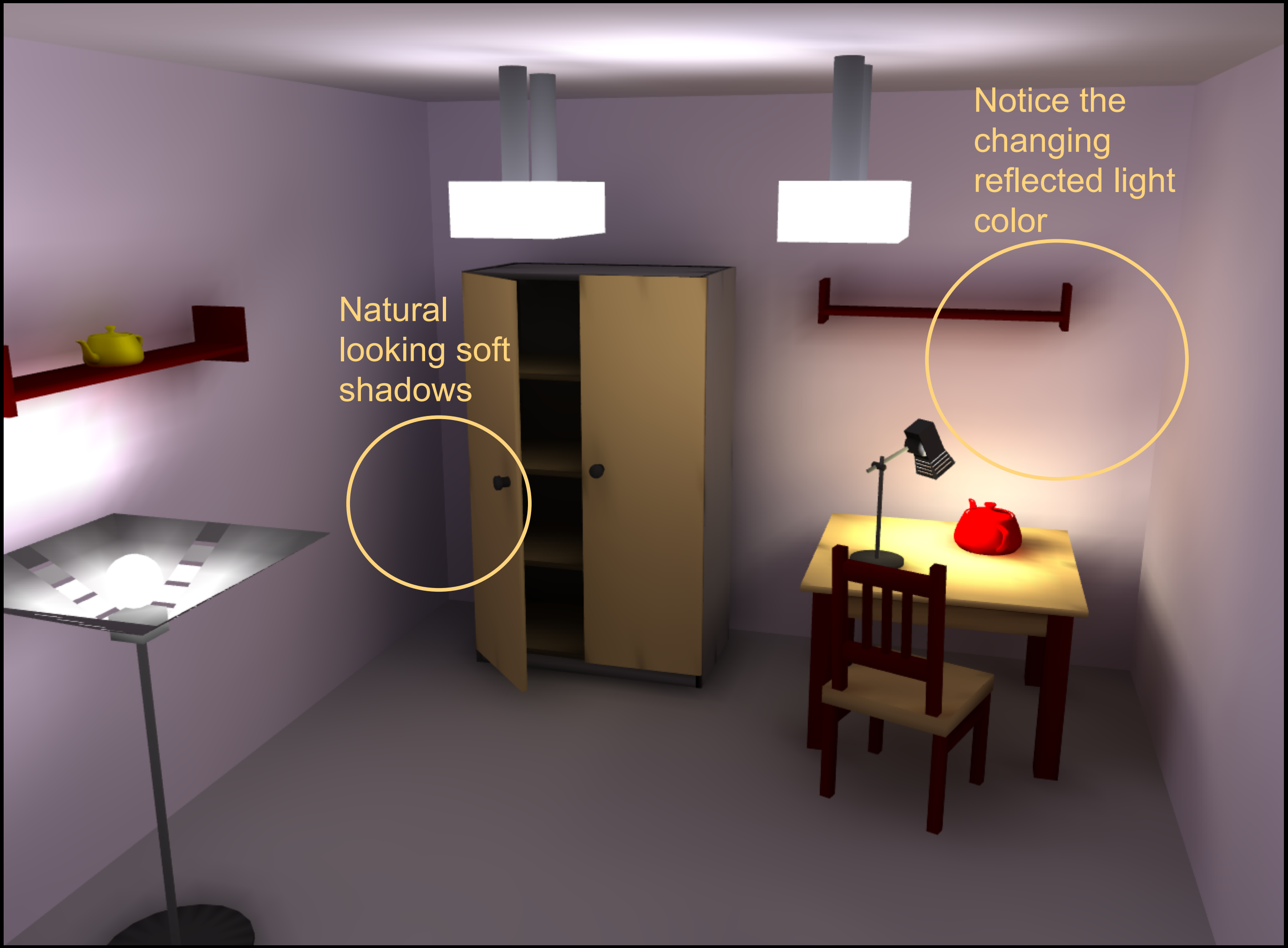
- The main pro and con for this technique is the same, it is precomputed.
- Since it is precomputed it is very cheap to use at run time.
- Since it is precomputed the results are static.
- Since the results are static, this model is only appropriate for static objects. It can be combined with our standard lighting algorithms.

Radiosity

Radiosity

- Our standard ambient + diffuse + specular lighting model is only an approximation of how light actually acts.
  - We have no reflection of light, which can change the color of the light being cast.
  - We have no shadows.
  - We only light faces semi-correctly that are hit directly by rays from a light source.
- The Radiosity lighting model will create a much more believable result, at a cost.

Scene lit using Radiosity Model


Natural looking soft shadows
Notice the changing reflected light color

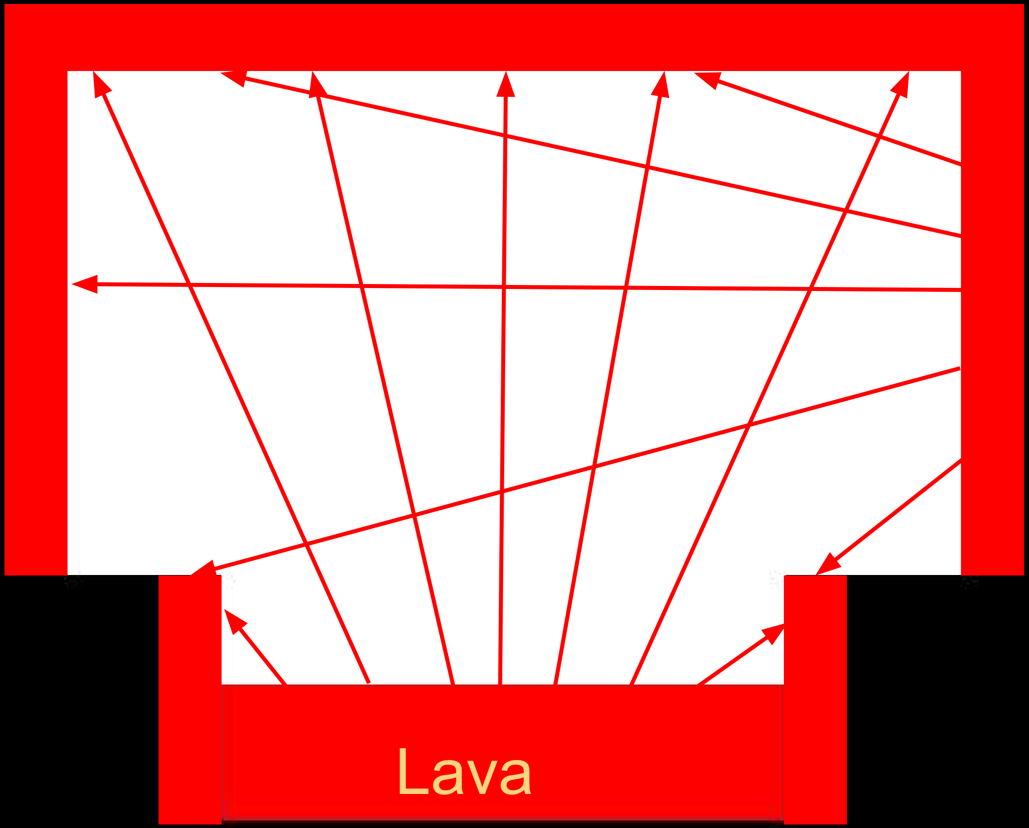
Radiosity

So how does it work?

1. For each light in the scene
   a. Cast a ray from the light to every other point in space
      i. For each point hit, calculate lighting, and turn that point into a light source.
      ii. Start the process again at step 1.

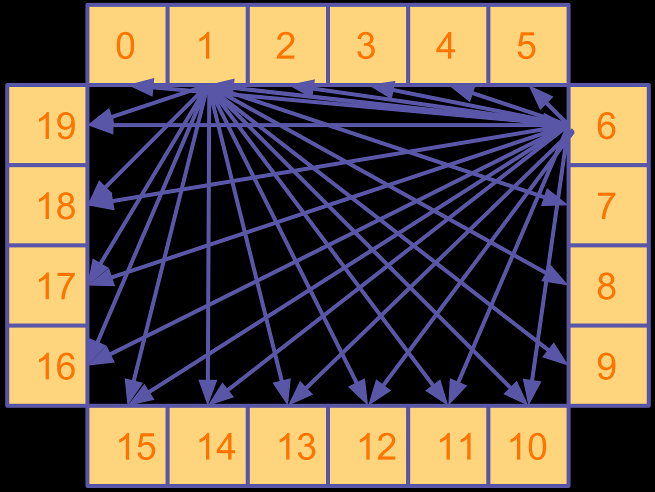
Did you notice the two different infinite sets in that process?

Radiosity



Radiosity

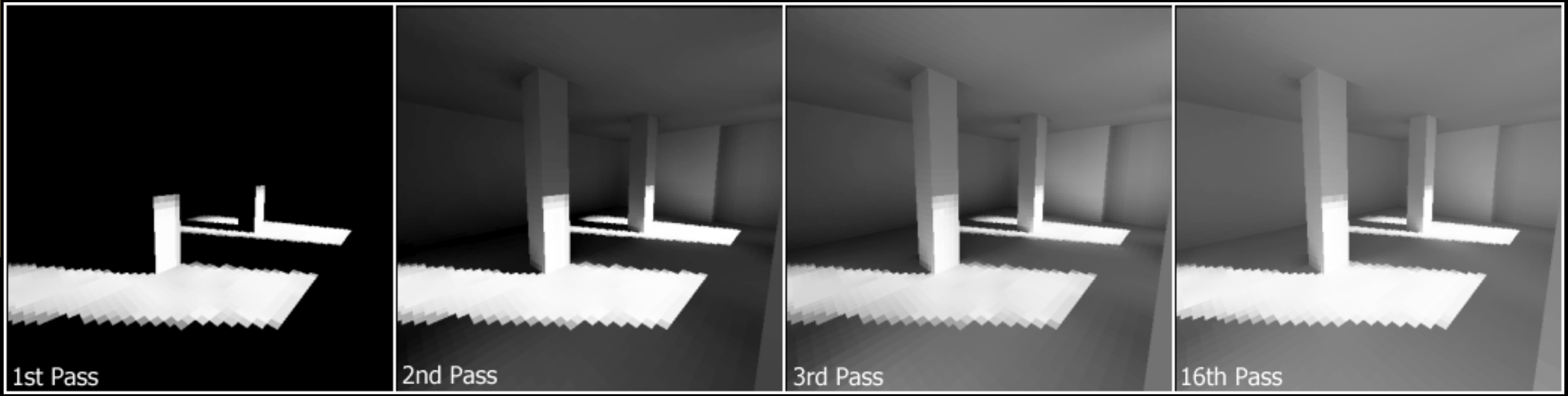
- Instead of lighting an infinite set of points in space, we break the world into small sections that get lit evenly, called patches.
- Patches do not have to be a uniform size. Smaller patches give better results, but take longer to process as there will be more of them.



Radiosity Passes

- The concept of patches gives us a finite set of things to be lit in the scene, but we still have to decide when to stop doing additional passes.
- Each pass will give us diminished changes from the previous pass due to light attenuation.
- We could look for a minimum threshold of light energy, go by eye, or just hard code a number of passes.

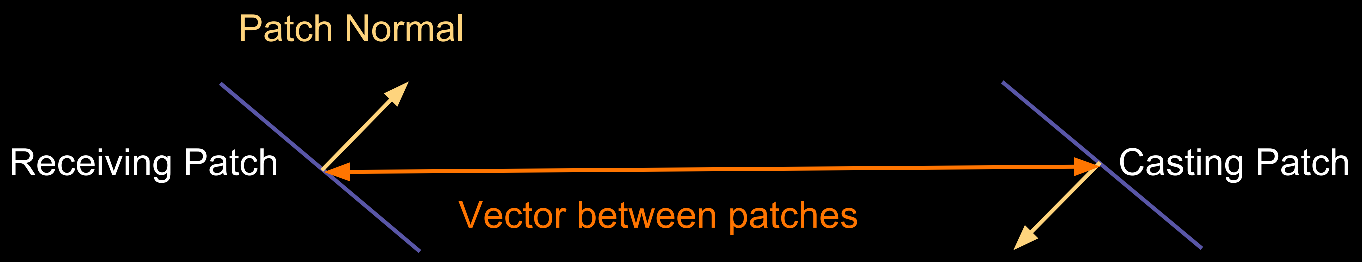
Radiosity Passes



Radiosity Energy Formula

$$P_i = P_i + P_j * F_{ij} * A_i / A_j$$

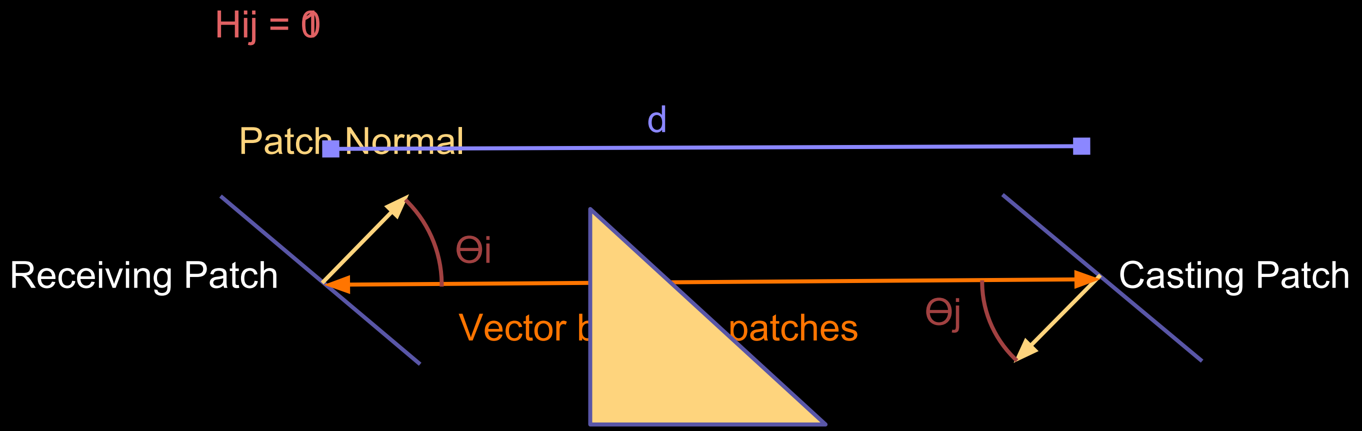
- Pi is the current energy level of the receiving patch
- Pj is the current energy level of the casting patch
- Fij is the form factor of the two patches, explained on the next slide
- Ai is the area of the receiving patch
- Aj is the area of the casting patch



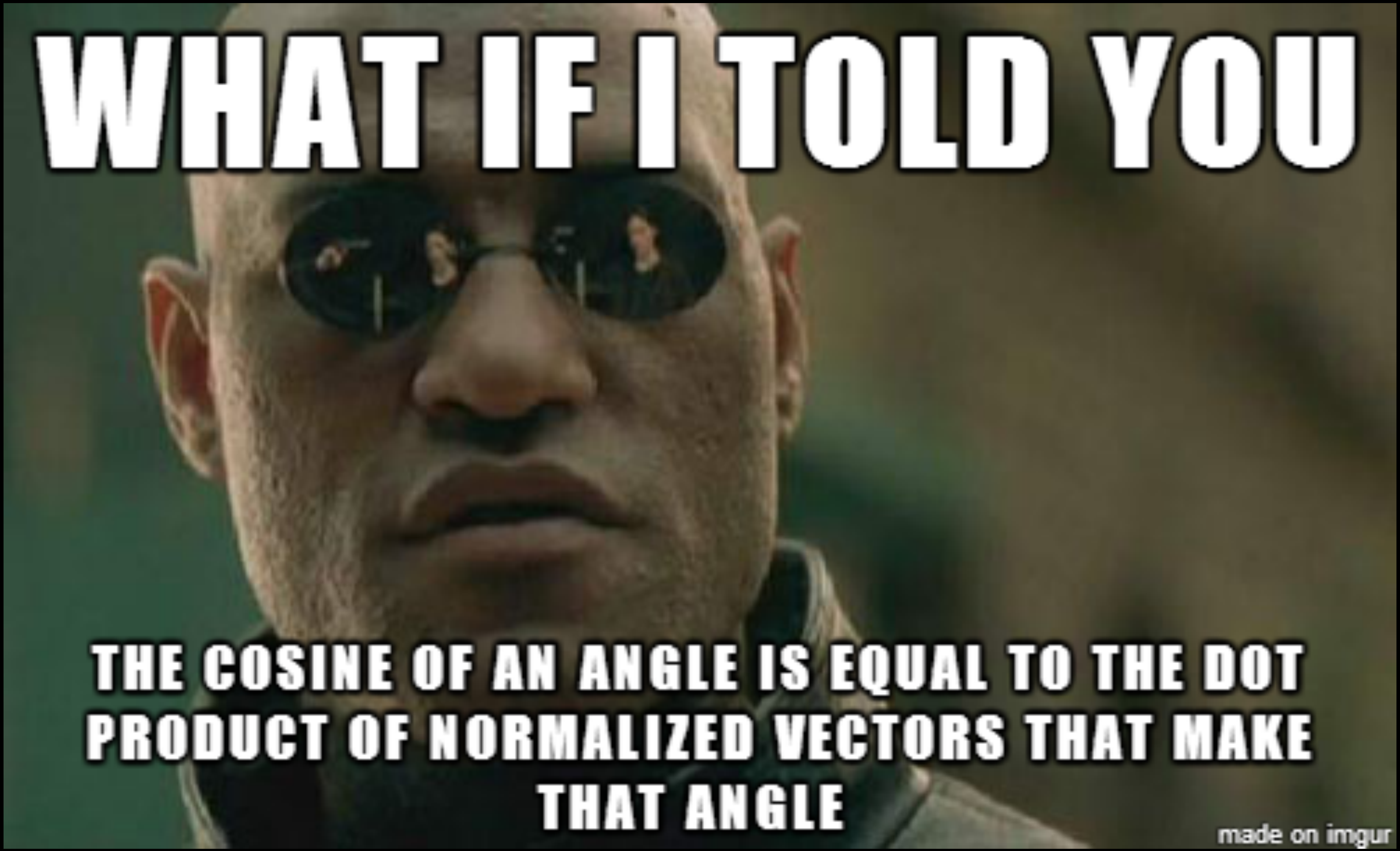
Radiosity Form Factor Formula

Fij = (cos Өi * cos Өj) / d^2 * Hij

- Өi, Өj is the angle between the patch normal and the vector to the other patch.
- d is the distance between patches
- Hij is just a 0 or 1 value representing whether a ray from one patch to the other is unobstructed.



Cosine...


WHAT IF I TOLD YOU
THE COSINE OF AN ANGLE IS EQUAL TO THE DOT PRODUCT OF NORMALIZED VECTORS THAT MAKE THAT ANGLE
made on imgur

Cosine and dot product

- Cosine of 90 degrees == dot(float3(1, 0, 0), float3(0, 1, 0)) == 0
- Cosine of 0 degrees == dot(float3(1, 0, 0), float3(1, 0, 0)) == 1
- Angle Formula:
  - angle = acos((dot(A, B)) / magnitude(A) * magnitude(B))
  - If A and B are normalized:
    - angle = acos((A dot B));
    - acos is the inverse cosine, it “removes” a cosine
    - angle = A dot B “- cosine”
    - A dot B = angle “+ cosine”

Radiosity

- Radiosity is currently too slow for real-time rendering.
- The formulas discussed are not terribly expensive, the issue is the very large number of times we have to do them.
- Likely the biggest cost is doing collision tests to see if rays can travel between patches unobstructed.
- Spatial Partitioning can help speed this up...

Radiosity and Spatial Partitioning

- Spatial Partitioning refers to a system used to divide up the simulation world into logical chunks that can be used to speed up collision queries.
  - Example, if I am standing in Florida, and I have not grown quite large over night, we can safely and quickly decide I am not colliding with anything in Ohio.
  - Spatial Partitioning will be one of the main topics of Engine Development 3
- Spatial Partitioning can be used to get a set of objects that might be in view from a given location.
- We will need to optimize the process of calculating Radiosity even for use in tools as we do not want to have to wait a great deal of hours everytime we export a level.

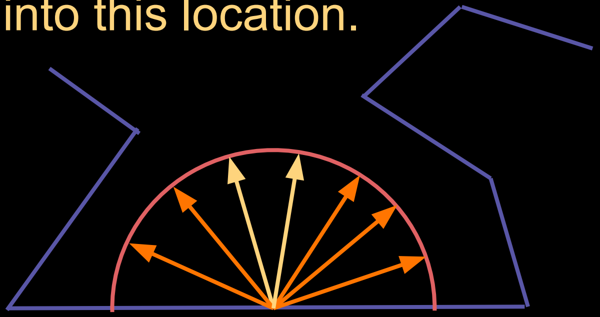
Ambient Occlusion Mapping

Ambient Occlusion Mapping

- The ambient portion of our lighting model is a fairly poor approximation of the effect of reflected light rays.
- Ambient Occlusion Mapping (AO mapping) is a technique we will use to improve the ambient portion of our lighting.
- AO mapping works by providing a texture map whose data represents how likely bouncing rays of light are to hit the location.
- We will use this map to modify our ambient light value.

Building an AO map

- How to build AO map data
  - Iterate through locations on a given model
    - For each location, cast a hemisphere of rays away from the surface
      - For each ray that does not collide with the model, add 1 to a count variable
      - Divide the count variable by the number of rays cast, and you get a 0 to 1 value representing how likely light is to bounce into this location.



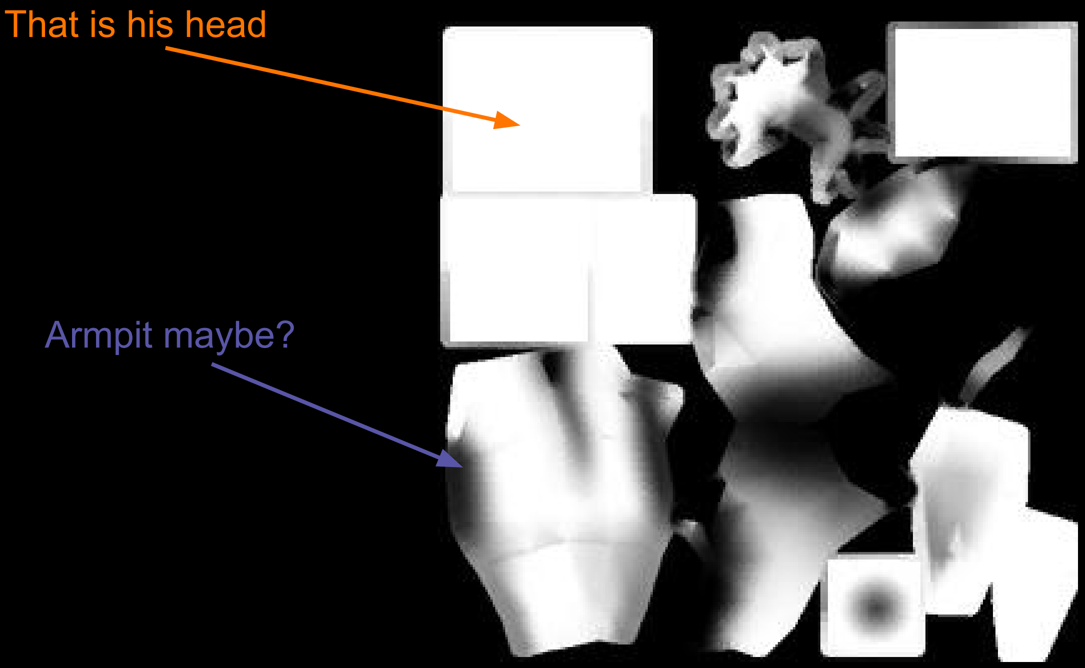
Colliding rays = 5

Unobstructed rays = 2

Ambient Occlusion amount = 2/7

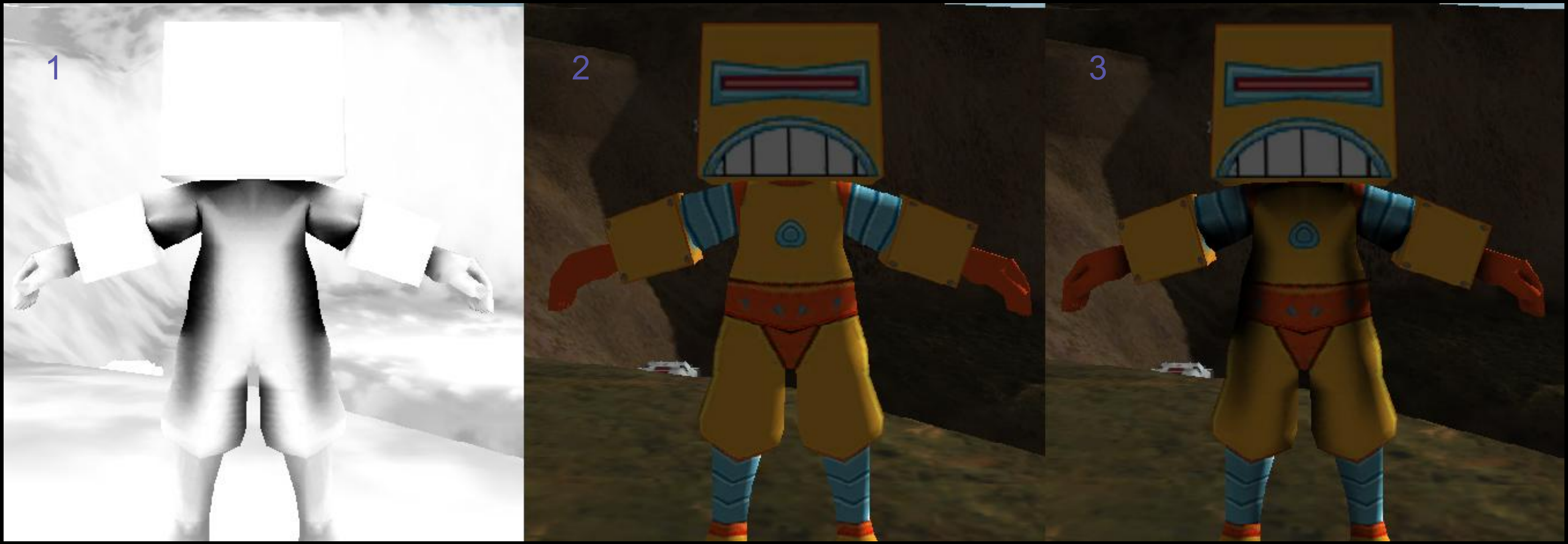
Building an AO Map

- AO map creation is a fairly standard process, so much so that Maya can do it for you.
- This is the ambient occlusion map "RobotKid" uses:



Ambient Occlusion Mapping

1. Scene rendered with AO Map as diffuse
    a. Don’t do this, it is just here for an example.
2. Scene rendered without AO Mapping
3. Scene rendered with ambient scaled by AO Map value. The AO Map is exaggerated to show results.



How to implement AO Mapping

These instructions assume a deferred shading model is used.

1. Store the AO Map value in the GBuffers.
   a. We will use our diffuse buffers alpha channel to store this, diffuse.a = AOMapSample.
2. During light rendering, scale the ambient color portion by the AO Map value stored in diffuse.a.
3. …
4. Profit

Screen Space Ambient Occlusion

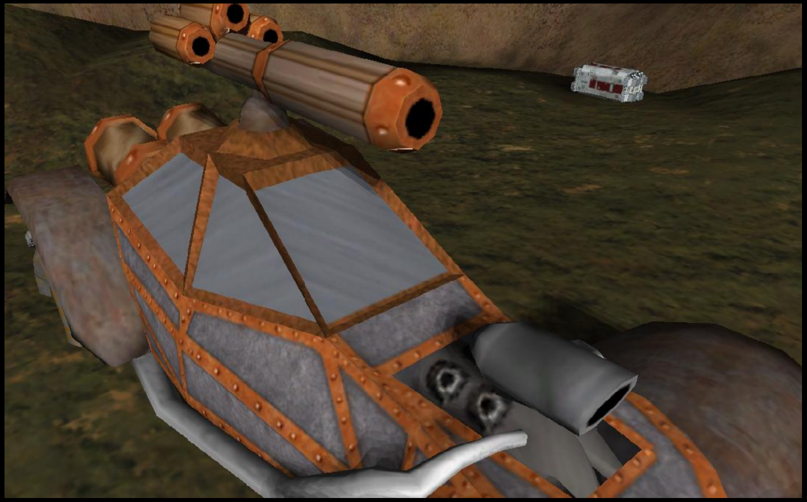
- Ambient Occlusion Mapping uses a preprocessed texture, making it both fast at run-time and only correct for static results…
- Screen Space Ambient Occlusion, or SSAO, is a real-time version of the same technique.
  - Instead of precomputing concave sections of our geometry, we calculate them in real-time, based on what the view currently has present.
  - This is a post-process technique that uses pixel normal and depth values to determine where the ambient should be darker.
  - Check out the nVidia Graphics SDK 11 for more information at : nVidia Graphics SDK

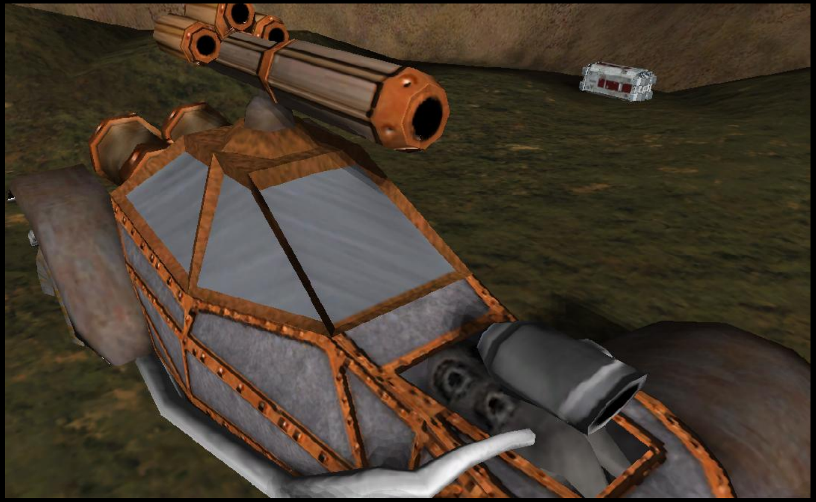
Normal Mapping

We want mo’ bumps.

Normal Mapping

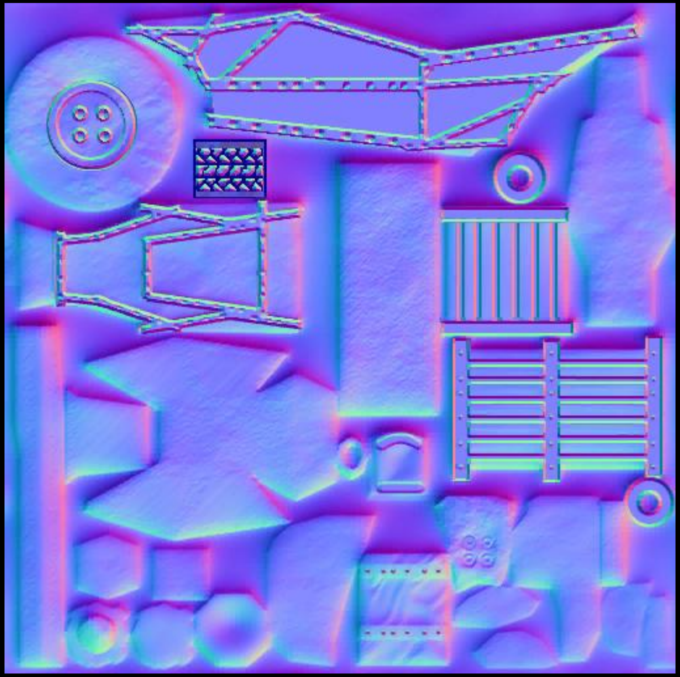
- Normal mapping is a technique we use to provide more normals than there are vertices.
- When lit, these additional normals can give the impression that the surface is more varied or bumpy.





Normal Mapping

- This is the normal map used with the buggy model from the previous example:
  - Yes it is weird that it is mostly blue, we will come back to that.



Bump Mapping

- We often use the terms bump mapping and normal mapping interchangeably.
- To be more precise, bump mapping is a technique that uses a height map to define surface properties.
- This height map can be used to define normals.
- Normal mapping is a subset of bump mapping that skips the height map, and just creates the normals.

Where do we get normal maps?

- Well, it is just a texture so conceivably an artist could make it by hand, but in practice that rarely if ever happens.
- Another approach is to create a very high poly count version of the model then use an algorithm to bake out the normals to a texture. The artist then creates a low polygon version of the model the normals are used on.
    - This path had lost some traction as what artist wants to create high poly models that do not get used.
    - Programs like Z-Brush being used for the first stage in art development have made this path common again. Z-Brush creates stupidly high poly count models.

Where do we get normal maps continued...

- Image based methods
  - There are actually formulas for creating normal maps based of off diffuse texture information.
  - These formulas look for changes in color and radiance of the diffuse texture to represent changes in direction the normal should point.
  - One popular, not quite free, program for doing this is called Crazy Bump.
  - A free plugin for Photoshop is available from nVidia.
  - For those truly on a budget, The Gimp, also has a normal map plugin.

Using Normal Maps

Here is what will not quite work out…

1. In special GBuffer pixel shader for normal mapping, sample normal map color
2. Convert color (0 -> 1) to normal range (-1 -> 1)
3. Apply objects World Matrix transform to normal
4. Write out normal for use in lighting shaders later

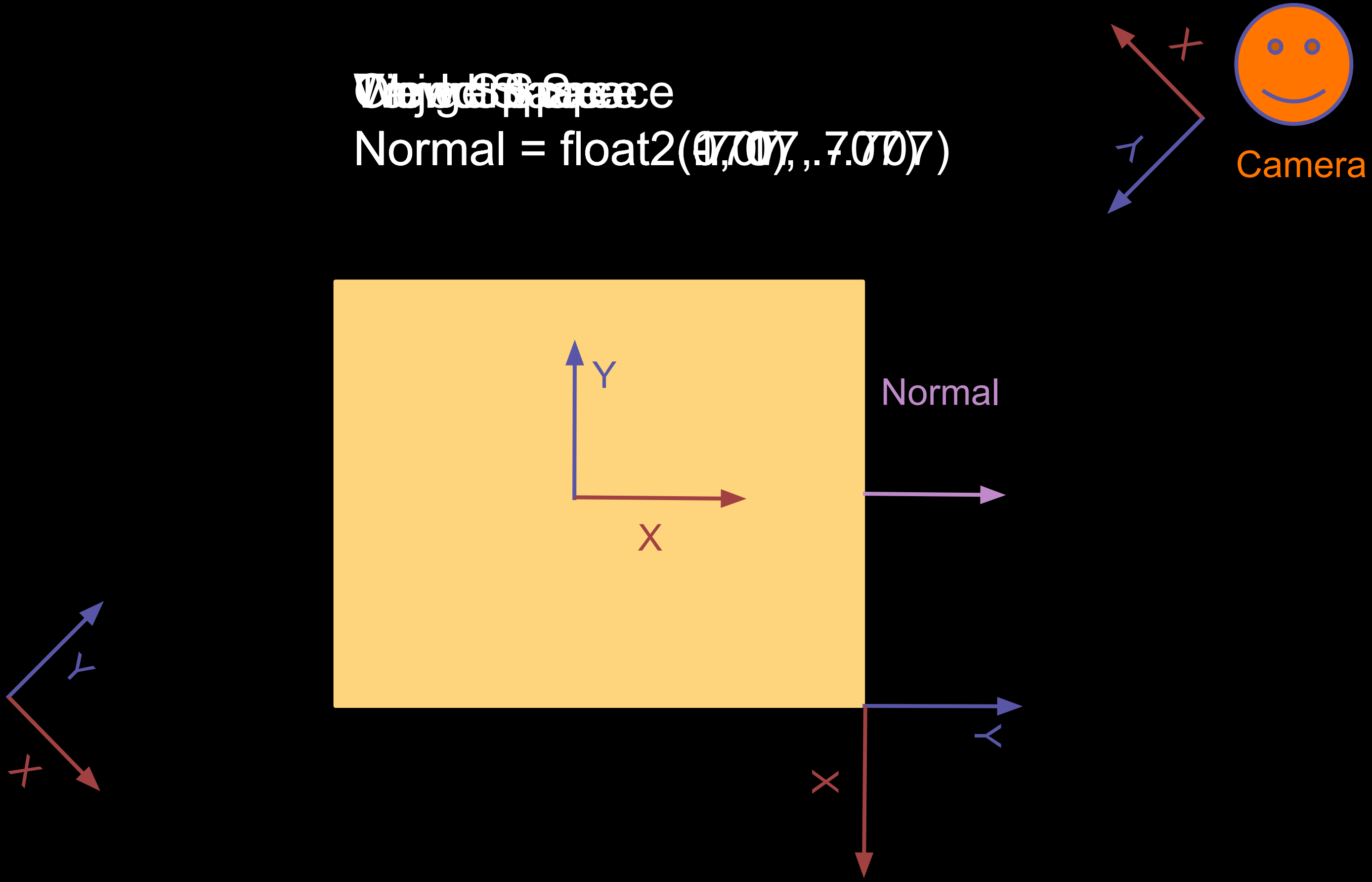
- This is fairly close to what we will do, but will not work for our normals that are in **tangent space**.
  - This technique would work for “Object Space Normals”, which will be part of our additional materials section at the end of the slides.

Tangent Space

- With Object Space, we think of the origin as being placed at and oriented to the object.
- With World Space, we think of the origin being placed at and oriented well… where the origin is.
- With View Space we think of the origin being placed at and oriented to the camera.
- With Tangent Space, the origin is oriented with a specific surface.
    - Tangent space matrices typically do not have translation (position)

Spaces

Object Space
World Space
View Space
Normal = float2(1,0)
Normal = float2(.707,.707)
Normal = float2(0,-1)



Converting spaces

1. Normals we receive are typically in object space
2. We multiply these normals by the object's world matrix to put them into world space
3. We multiply the world space normals by the camera's view and projection matrices to put them in view-projection space
4. We multiply the world-view-projection space normals by a tangent matrix to put them in tangent space.
   - Since tangent space is defined by the orientation of the surface, each vertex will have its own tangent space matrix
   - This is just a 3x3 orientation matrix. No translation is present, and scale would be weird to have.

Tangent Matrices

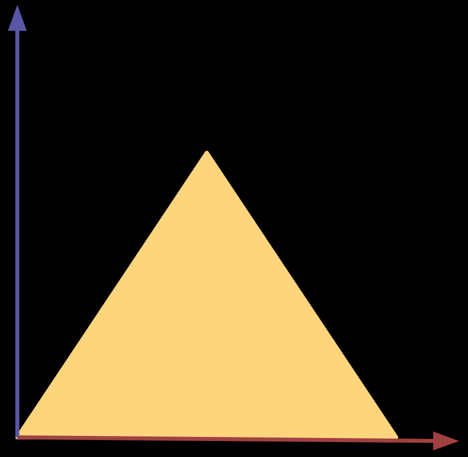
- Our vertex shaders will have the job of constructing the tangent matrix, which is sometimes called the TBN based on what each axis is storing.
- T for Tangent, B for Bitangent and N for Normal

$$\begin{array}{ccc} X & Y & Z \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array} \qquad \begin{array}{ccc} T & B & N \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}$$

TBN Matrix

- The normal in the TBN, is the normal of the surface.
- The tangent and bitangent are vertices parallel to the surface, but perpendicular to each other and the normal.

The normal is pointing at you.



TBN Matrix

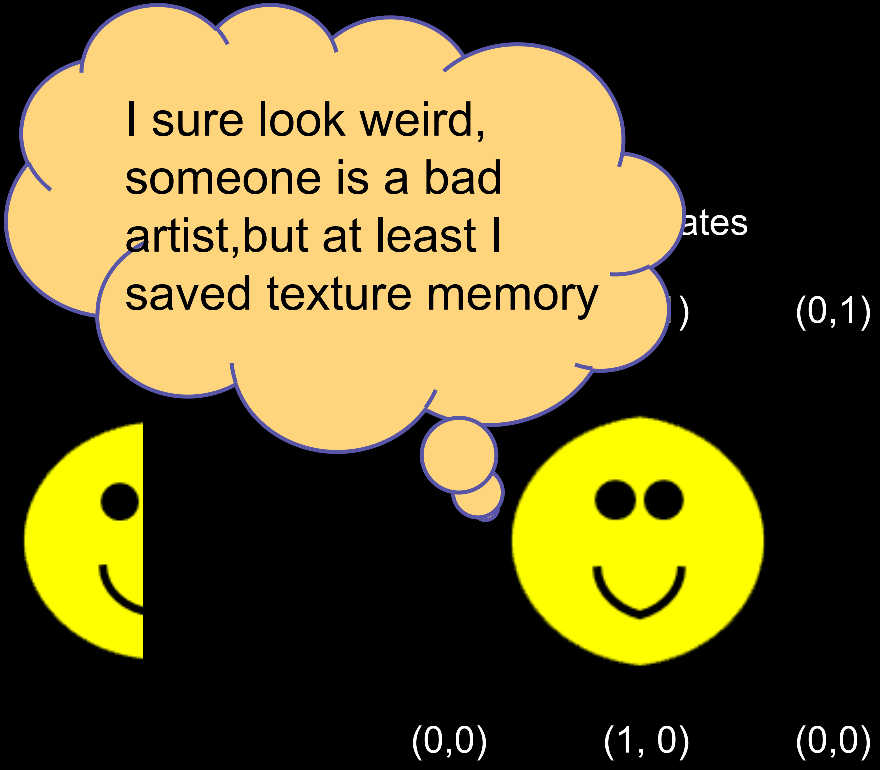
- Given a triangle, we could compute the whole TBN matrix ourselves.
- One edge of the triangle could be our tangent
- The cross product of the two edges would be our normal
- The cross product of our normal and tangent would be our bitangent

TBN Matrix

- Most (all) modeling programs like Maya will provide tangent, bitangent and normals for each vertex.
  - So we will not construct them ourselves
- Passing a 3x3 matrix with each vertex can be rather costly…
- To reduce the cost, we will only pass the normal and the tangent vectors.
- We can then cross the normal and tangent vectors received in our vertex shader to build the bitangent part of our matrix.
  - Remember, the order we do this cross product in is important, doubly important if we are supporting texture mirroring...

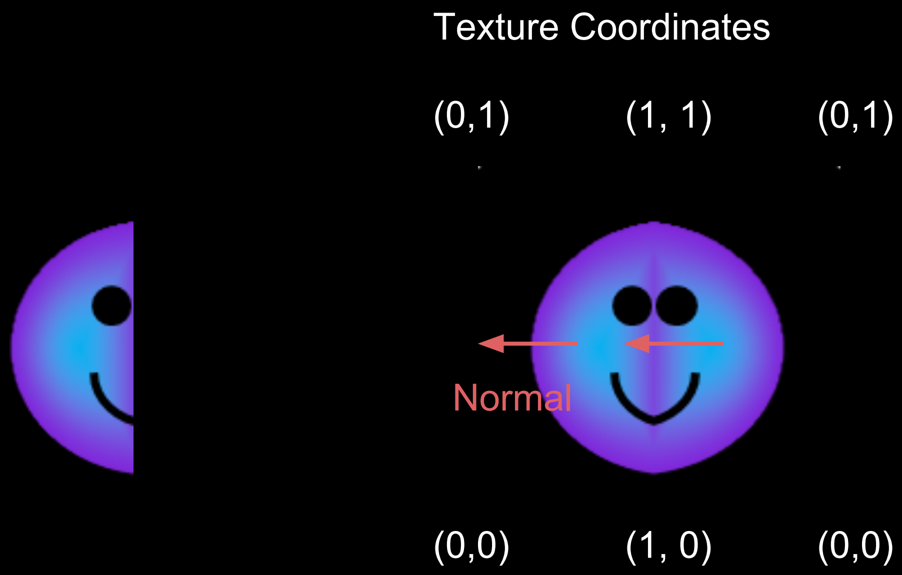
Normal Mapping, UV Mirroring

- It is a very common practice when texture mapping to mirror a texture across a surface.


I sure look weird, someone is a bad artist,but at least I saved texture memory
ates
)
(0,1)
(0,0)
(1, 0)
(0,0)

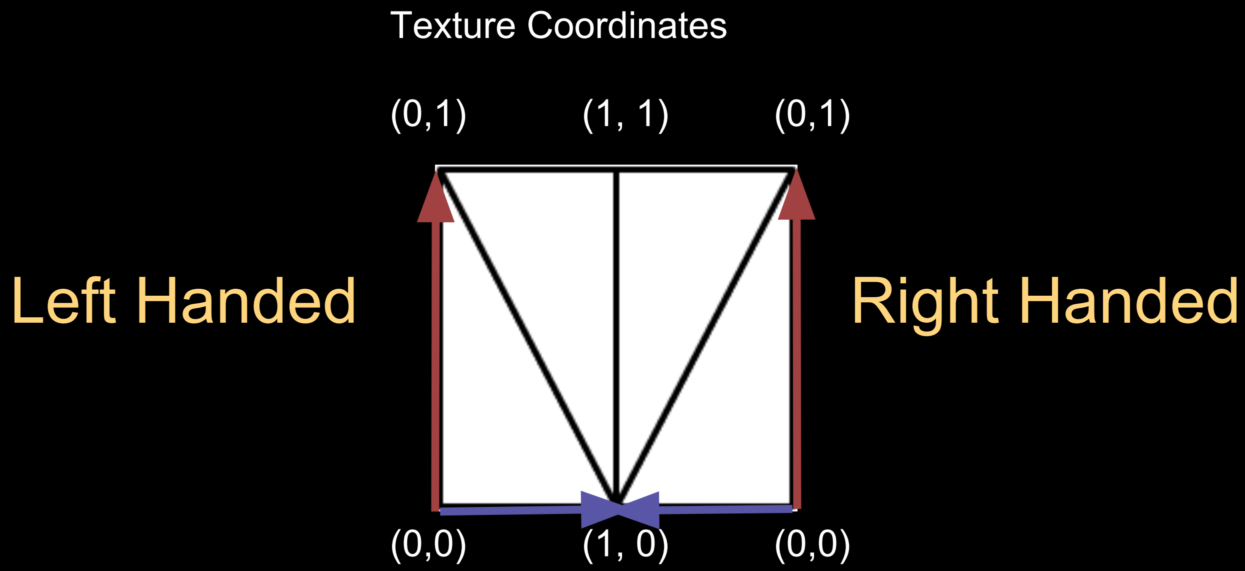
Normal Map Mirroring

- When we mirror a normal map, the individual normal's position gets mirrored, not their direction…
  - ...but we can fix this with the power of math!



Normal map mirroring

- We need to first decide if the normal needs to be flipped.
- Where the texture is mirrored, we will flip the normal.
- Where the handedness of the texture coordinates changes is where the mirroring is happening.

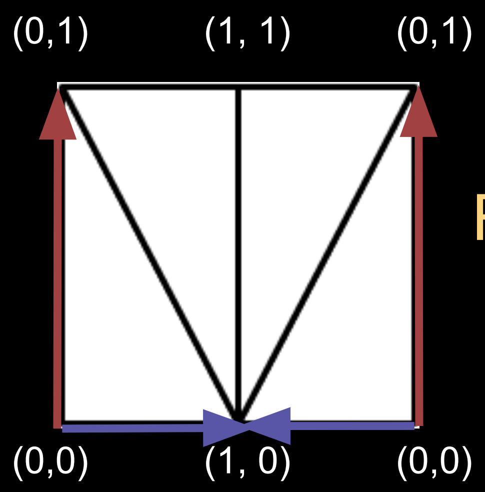


Normal map mirroring

- We will store the handedness of texture coordinates in a determinate value with each vertex. (+1 or -1)
- When the determinant is negative, we want to flip the normal, by flipping the TBN
    - Bitangent = cross(Normal, Tangent) * Determinant

Texture Coordinates

Left Handed



Right Handed

Normal Mapping Application

- For our Deferred Shading system we will need to write specialty geometry shaders for normal mapping support, GbuffersBump_VS and GBuffersBump_PS
- In the vertex shader:

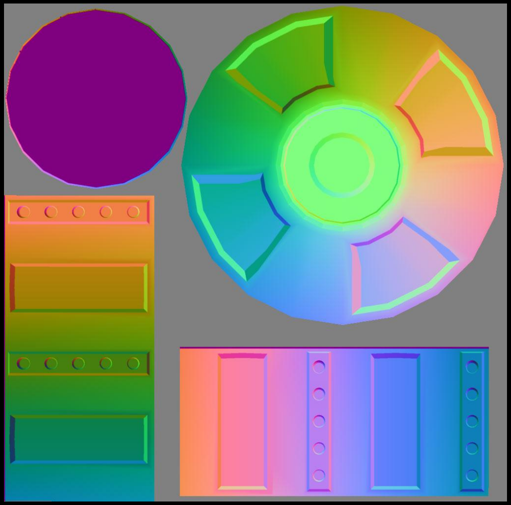
1. Transform the input normal by the world matrix and store in TBN[2].
2. Transform the input tangent by the world matrix and store in TBN[0].
3. Cross the transformed normal and tangent to compute the bitangent and store in TBN[1]
4. Scale the bitangent by the determinant value passed in to support UV mirroring.

Normal Mapping Application

- In our fragment shader:
  1. Sample the normal map to get the **color** version of our normal
  2. Convert sample result from color to normal range
  3. Transform sampled normal by passed in TBN matrix
  4. Convert transformed normal to color range
  5. Store result in normal color buffer output.

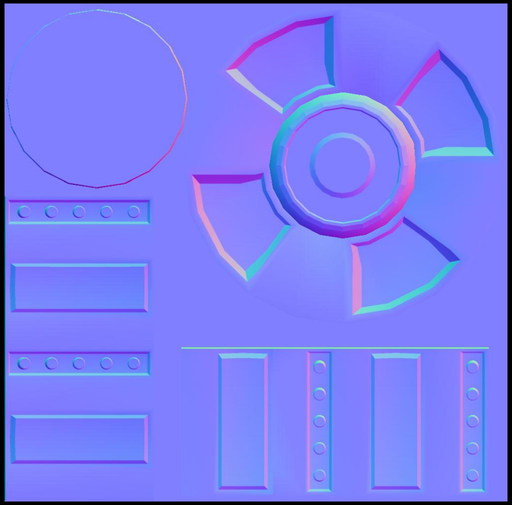
Object Space Normal Maps

- There are actually two ways to store normal maps, tangent space which we are using or object space.
- Where tangent space normals need to be transformed to match the orientation of their surfaces, object space normals already have this transform included.

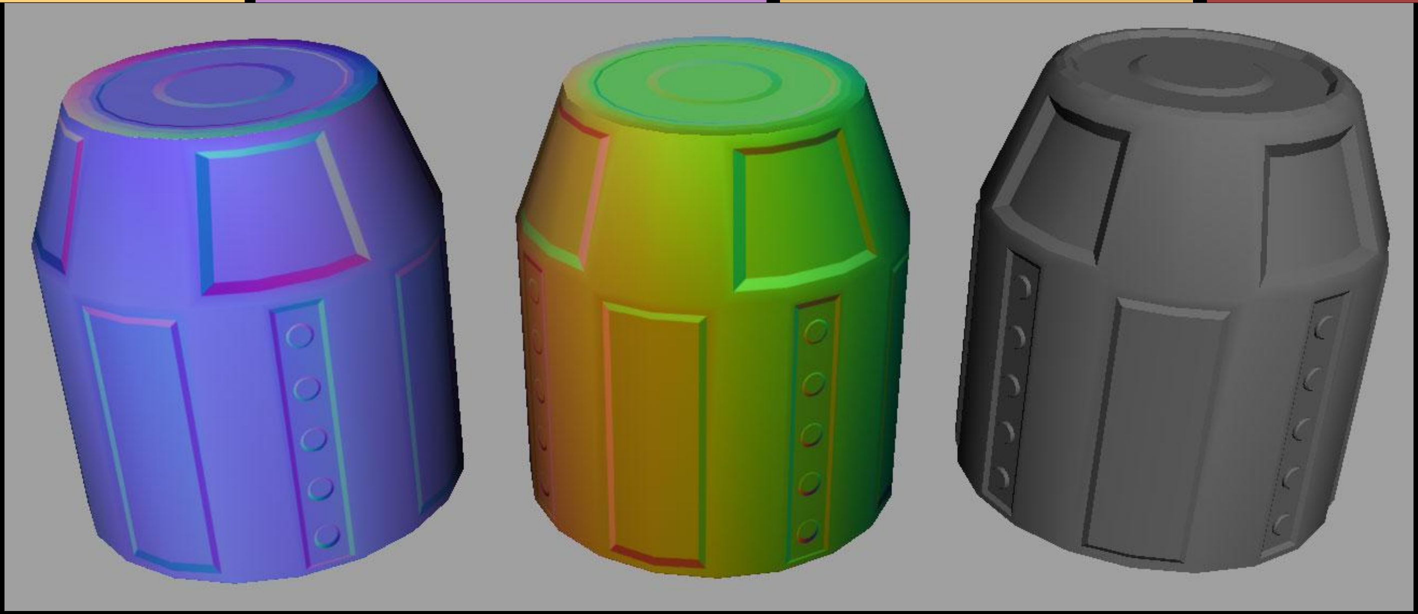


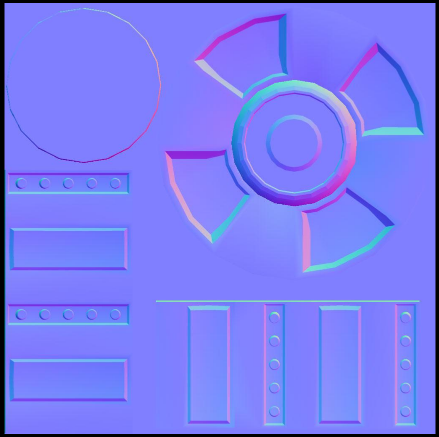
Object Space

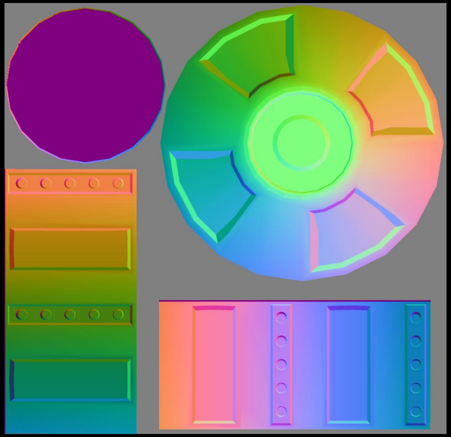
Tangent Space



Object Space Normals







This model is two smooth cylinders. The extra detail is from the normal map, tangent or object give the same result

Object Space Maps

- Benefits:
  - Easy to use. Just sample the texture, convert to normal range and apply the world matrix transform.
- Drawbacks:
  - Cannot reuse texture space. A normal map for a tire will either point to the right or the left, and cannot be easily flipped for the other.
  - Cannot support texture mirroring.
  - Poor support for deformable geometry… next slide

Object Space Maps

- A lack of support for deformable geometry is the main reason object space maps are rarely used.
  - Organic animations involve mesh deformation, making this a very common thing to need to support.
  - Changing the position of a vertex should change the normal associated with it as well. In tangent space we can do this by adjusting the TBN matrix. With Object space we would be required to find all the affected normals in the map and rewrite them.