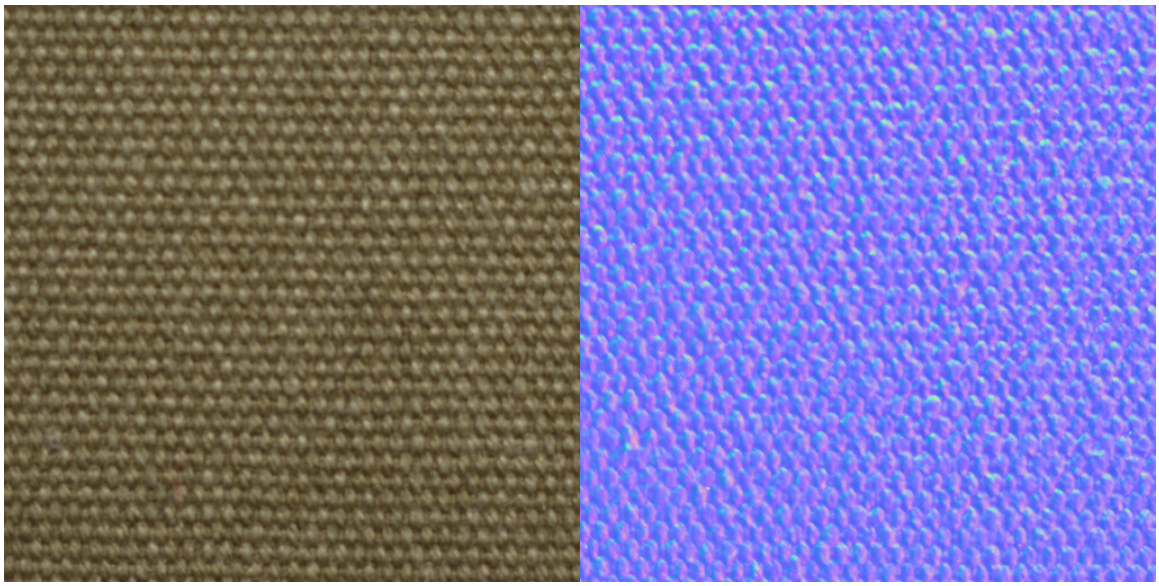
Normal and Diffuse Recovery using Photometric Stereo with Uncalibrated Light Sources

Brian Kehrer – Project Report

Georgia Institute of Technology
CS 6475 – Computational Photography Professor Irfan Essa



Introduction

Realtime computer graphics applications use high resolution normal maps to approximate surface detail, without requiring additional geometric tessellation. Normal maps encode surface normal direction (x,y,z) in the color channels of an image (r,g,b) for rapid access on graphics hardware *(for more information, see The CG Tutorial, by NVIDIA). As* the requirements for visual quality increase, so too does the need for high resolution normal maps. Generating these maps typically involves either computing them from high resolution 3D models, computing them from photographs, or creating them by hand with image editing or sculpting tools.

Some tools exist to create normal maps from photographs, such as CrazyBump - http://www.crazybump.com/

CrazyBump, as well as the other tools I'm familiar with attempt to generate a normal map from a single image, and consequently involve a fair amount of 'artistry' to achieve a desirable result. See a workflow here: https://iamsparky.wordpress.com/crazybump/

The goal of this project was to construct and demonstrate a simple workflow for scanning real world materials, and determining surface normal direction and surface pixel albedo, using varied light sources and a stationary camera, what is known as photometric stereo (Woodham n.d.). Furthermore, many of the systems require complex and or expensive calibration – so a further goal was to design a setup that could be reproduced by anyone with a camera. A tripod may also be useful.

Background

The original proposed solution solves for surface normal direction and albedo using the Lambertian lighting equation

$$I = k * (\hat{n} \cdot \hat{L})$$

where (I) is the final image intensity, (k) is the surface albedo, or color, (n) is the normalized 3 vector representing the surface direction in world coordinates, and (L) the normalized 3 vector representing the direction *toward* the light.

(N) and (L) are both normalized vectors, and are a substitution for the cosine of the angle between the surface direction, and the light direction.

When we take a photograph, we have created a map of image intensities (I), but the albedo (k), surface normal (n), and light direction (l) are all unknown to us.

The solution to this equation requires knowing the light directions ahead of time, typically by some kind of calibrated system, where light positions are fixed, and accurate measurements can be made. Once the light direction (L) is known for each image (I), we can solve for surface normal (n) and albedo (k) with 3 images (we only need 3 since the surface normal is normalized, and therefore only has 2 degrees of freedom, not 3). It is also possible to solve the equation using more than 3 images using a standard linear least squares solution – which is what we'll be doing.

Setup

The setup for the photography used a Nikon d600, a zeiss 50mm ZF.2 macro lens, a tripod, and a handheld shop light. The source for the first set of photographs is my laptop case, a canvas like material.

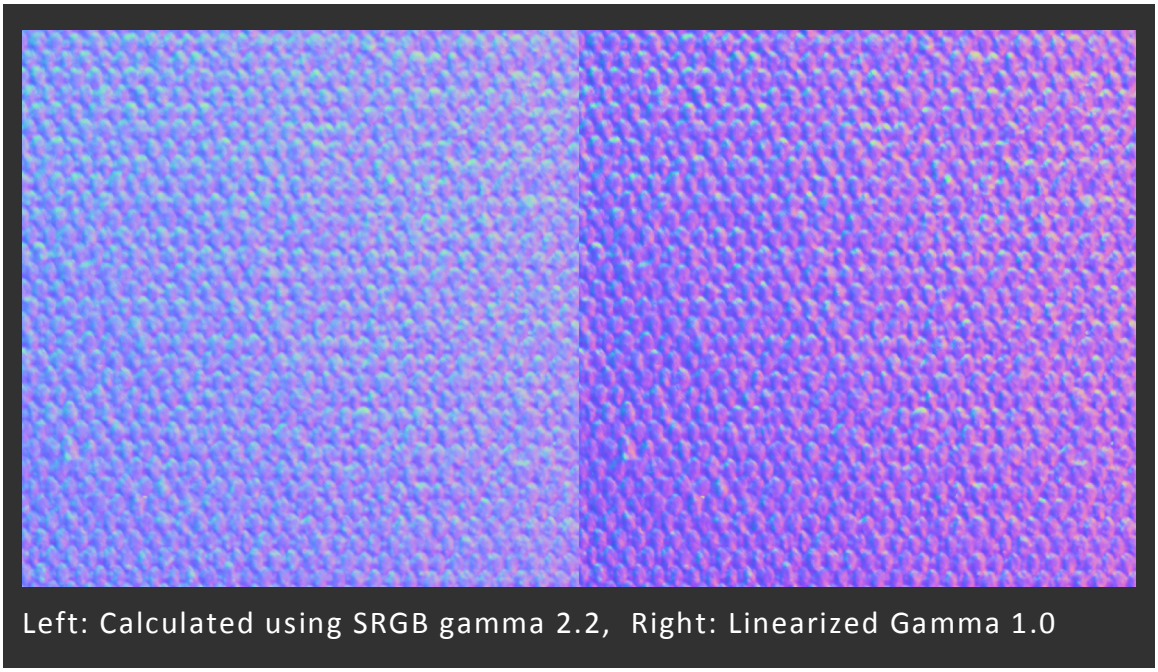
Gamma

As a first step all images were converted to linear space on import by applying a power of 2.2.

```
image = pow(image / 255.0 , 2.2)
```

This ensures all our images are linear, and in the range 0 to 1 in a floating point (32 bit) format.

What happens if you don't do this? The albedo looks almost the same, but the difference is most obvious in the normal map.



Left: Calculated using SRGB gamma 2.2, Right: Linearized Gamma 1.0

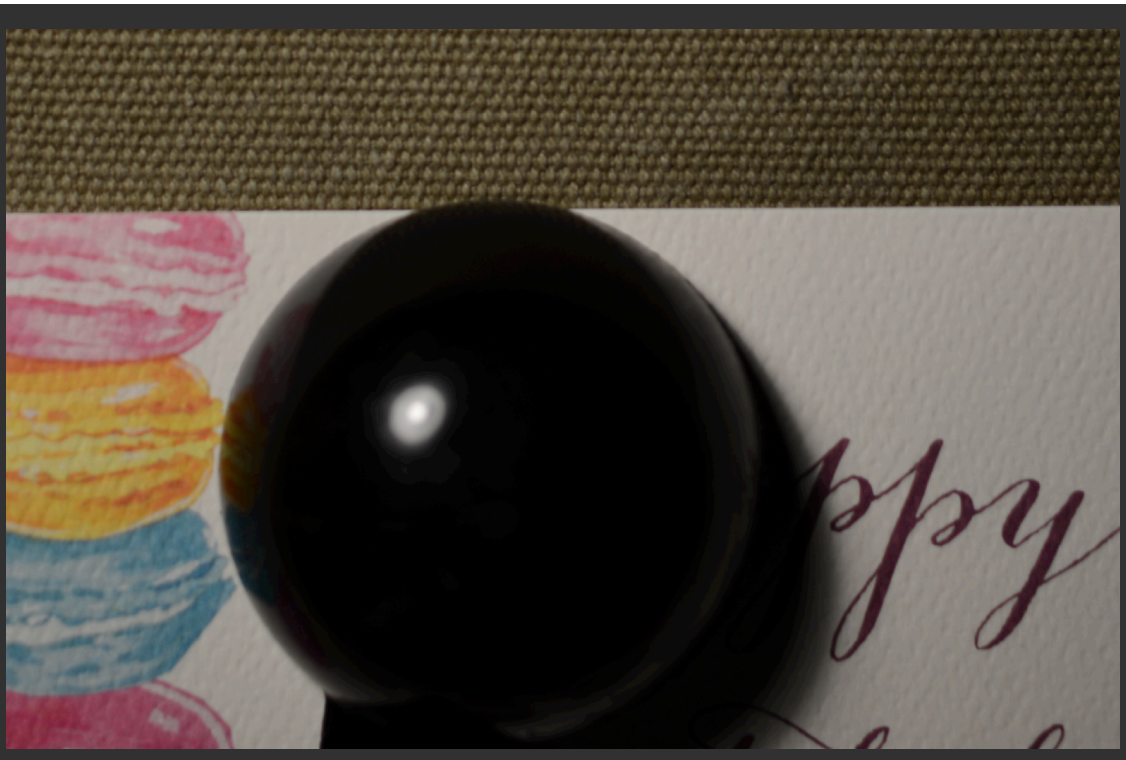
Solving for Light Direction

Ready at Dawn studios propose a number of useful techniques for calibration, including per-pixel light falloff (Pettineo n.d.). All of their techniques involve a stationary set-up with measured light positions – what I wanted to achieve was a completely portable solution, capable of fitting small or large volumes, where the lights could be handheld, and therefore completely unreliable in terms of calibration.

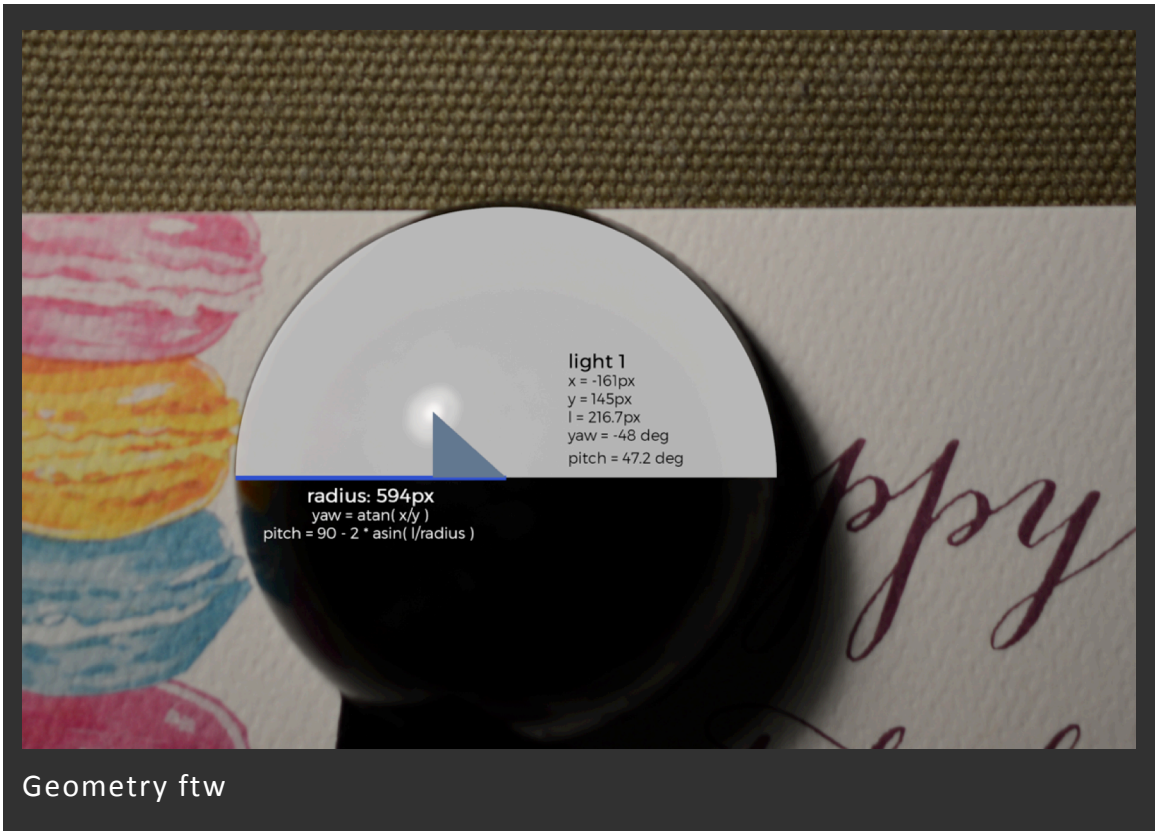
I've implemented two methods for solving for light direction using only the images taken from the camera. Both involve utilizing some of the frame for calibration – but both can also be fully automated, as I will demonstrate.

Hemisphere Method

The first method I attempted involved using a reflective hemisphere to record the light angle. Finding a reflective hemisphere proved challenging, so I made use of a plastic tablespoon. I have no shame in noting, while it certainly looked like a hemisphere – far better results could probably be obtained with something more precise.



Tablespoon as measurement device



Geometry ftw

To calculate the light position, we need only perform some simple geometry. First, we need the radius of the hemisphere in pixels, as well as the coordinates of the light, which should be the brightest point on the hemisphere.

$$\boldsymbol{x = light.x - center.x}$$

$$\boldsymbol{y = light.y - center.y}$$

$$\boldsymbol{length = \sqrt{x^2 + y^2}}$$

$$\boldsymbol{light\ yaw = atan2(x, y)}$$

$$\boldsymbol{light\ pitch = \frac{\pi}{2} - 2 * \mathrm{asin}\left(\frac{length}{radius}\right)}$$

Yaw is straightforward. X and Y are reversed from normal in the arctan so the central axis is pointed up. This was only a matter of convenience.

Pitch is only slightly more complex. Since we are looking at a sphere, the arcsin is only the half angle between the light and camera. We must multiply by 2 to get the actual light angle. We subtract the result from pi / 2 for convenience again, so that 90 degrees is straight up, and 0 degrees represents the horizon.

Linear Least Squares

Just a refresher on linear algebra that makes this possible. See http://mathworld.wolfram.com/LeastSquaresFittingPolynomial.html

$$I = k * (\hat{n} \cdot \hat{L})$$
$$\hat{L}^T \cdot I = k * (\hat{n} \cdot (\hat{L}^T \cdot \hat{L}))$$
$$(\hat{L}^T \cdot \hat{L})^{-1} \cdot \hat{L}^T \cdot I = k * (\hat{n})$$

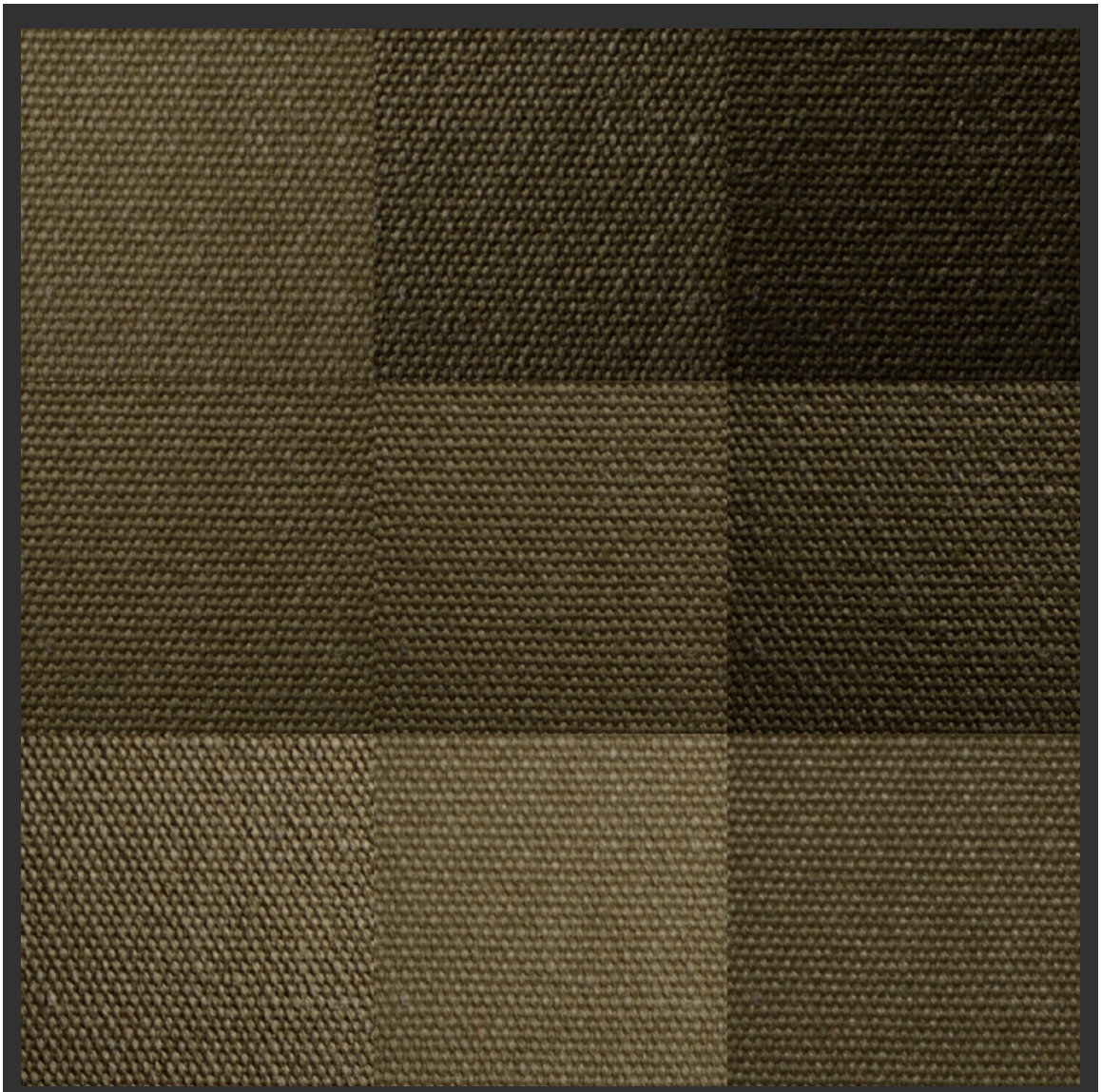
L is the list of light vectors. Compute the dot product of the light vectors with itself, and then invert the matrix.

The end result is a scaled vector. The magnitude of that vector is k, the normalized version is the surface normal.

First Image Set

The first set of images was solved using the hemisphere method. The light position was measured manually in photoshop, but this could be automated.

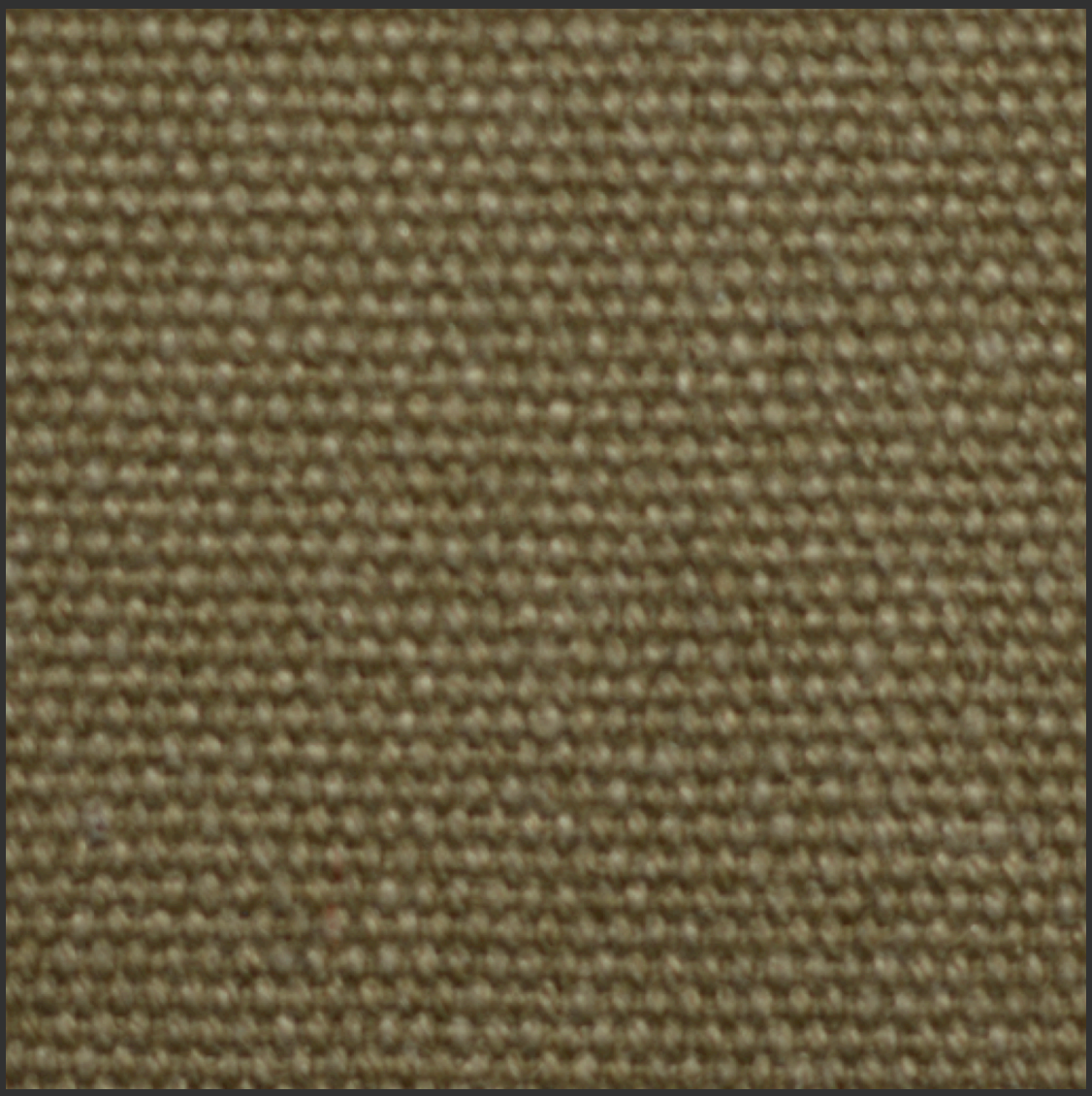
These are the 9 images that were used to compute the normal and albedo, and the corresponding calculated angles, in degrees.



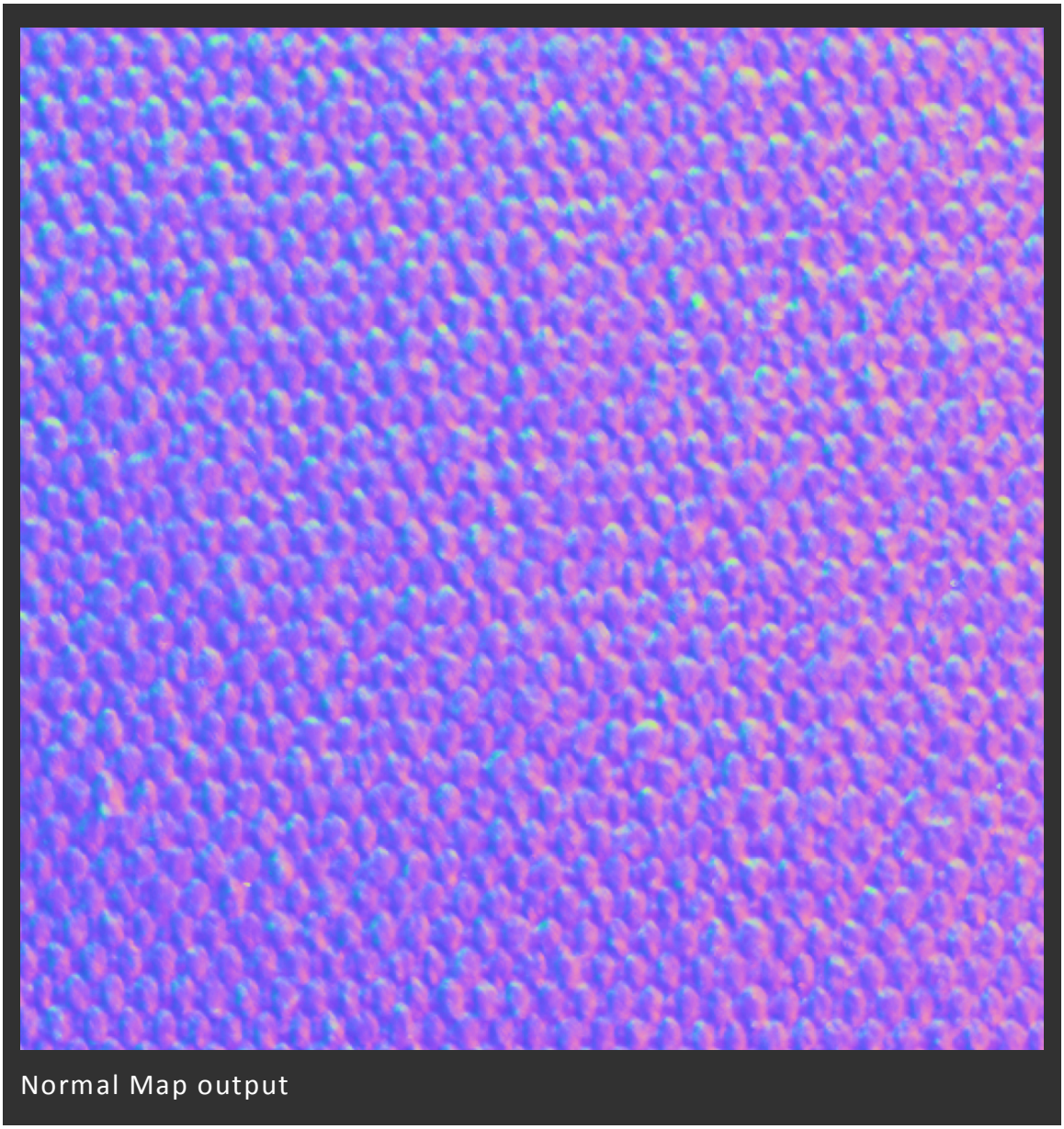
The 9 source images

-48° yaw, 47° pitch	-46° yaw, 21° pitch	-23° yaw, 18° pitch
-21° yaw, 41° pitch	14° yaw, 37° pitch	18° yaw, 19° pitch
52° yaw, 23° pitch	48° yaw, 51° pitch	-2° yaw, 60° pitch

First Image Results



Albedo output



Normal Map output

The first results show a good amount of detail in both the albedo and normal map, although unfortunately I don't have a control to compare against, so it's up to one's own judgment.

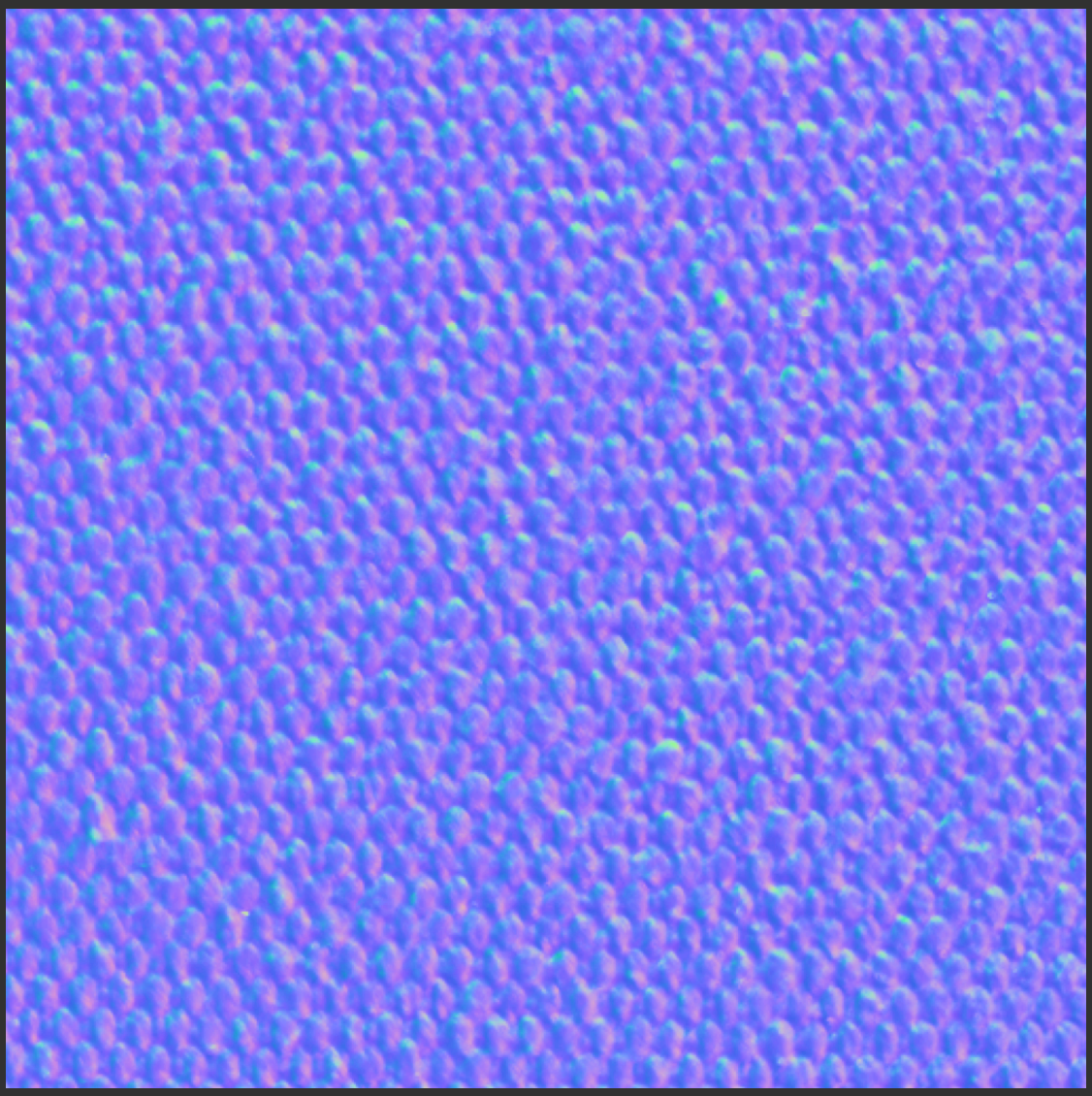
As Ready at Dawn mentioned, one apparent failing of this technique arises from the fact that the lighting equations take only light direction, and assume there to be no falloff. However, in reality, lights function more like point sources, and there is attenuation and falloff. Since I do not have steady lights, I cannot first build a luminance map to scale the images.

This error creates low frequency error in the resulting normal map. It may be hard to see in the above image, however when we apply a Gaussian blur, the distortion becomes more apparent.

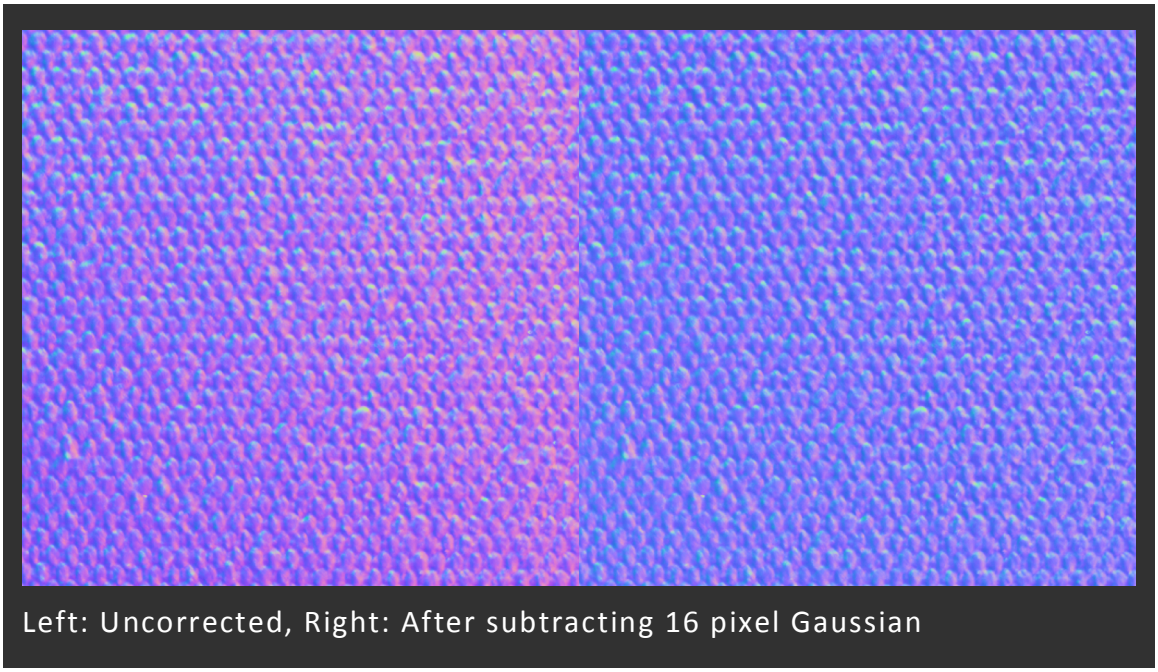
Normal map after applying 16 pixel Gaussian blur

This normal map displays a clear gradient of color from top left to bottom right, indicating a curvature. While compensating directly for light intensity is not trivial, it is possible to remove this low frequency error simply by subtracting it from the normal map. To correct for this, the blurred normal map was subtracted from the original – and (0,1,0) added back, representing an up vector. The normal map is then renormalized per pixel.

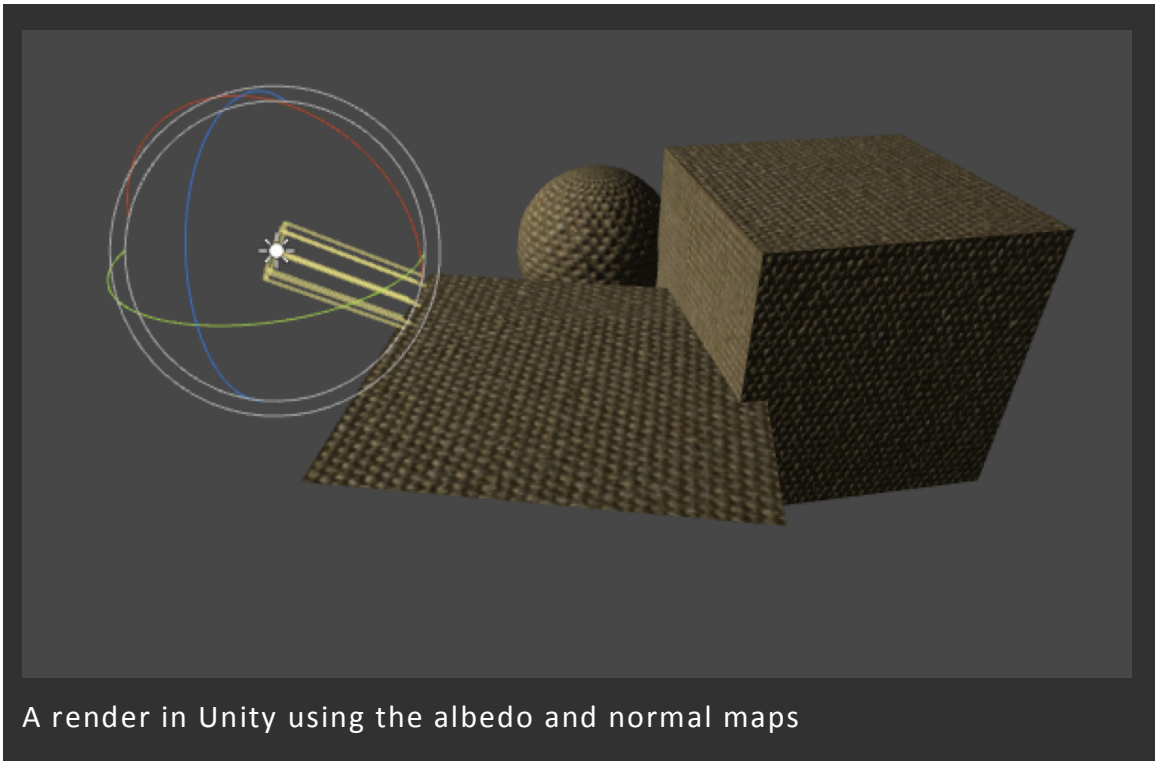
Below is the final corrected normal map, as well as side by side comparison.



The final normal map.



Left: Uncorrected, Right: After subtracting 16 pixel Gaussian



A render in Unity using the albedo and normal maps

Automating the Process

The first method showed promising results, but there are drawbacks – the most obvious being my only reflective hemisphere was a small tablespoon measure – and I'm not even sure it was spherical!

Furthermore, hand calculating the center of the hemisphere and light positions is time consuming.

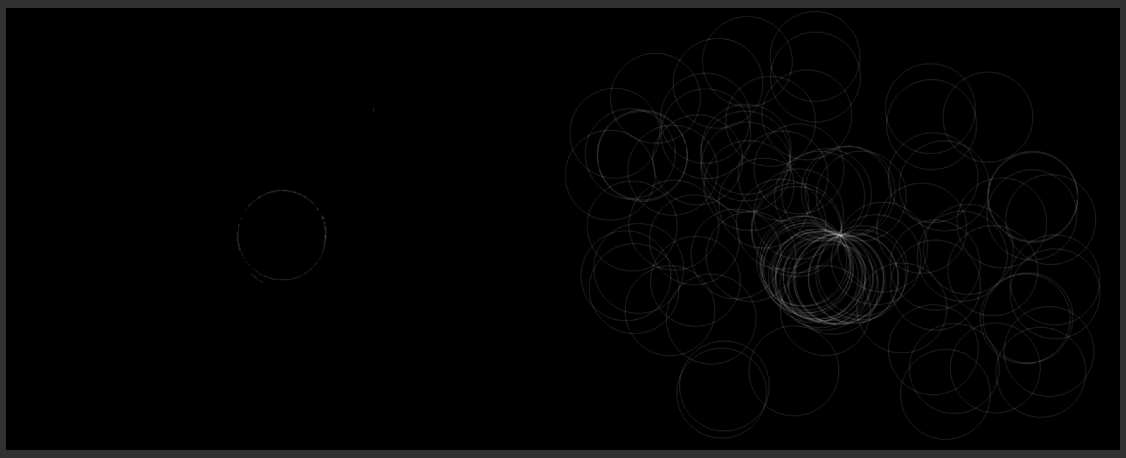
Shadow Method

The second method requires only a cylinder, and for the user to measure the diameter and height of the cylinder once.

Once we know the height of the cylinder, we can use the length of the shadow it casts to determine both the light yaw and pitch.

Cylinder with orange post-it note stuck on to remove any reflections, and improve Hough circle detection.

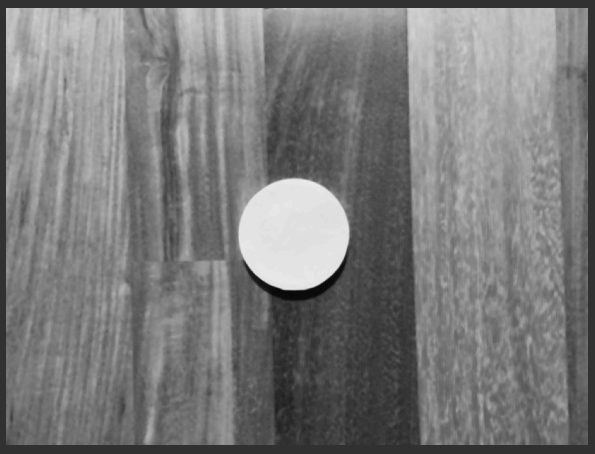
The first step in calibration is to identify the circular region. To do this, I used a custom Hough transform. In particular, to improve performance the detector looks only for large circles, and samples 200 points at random to perform the transform.



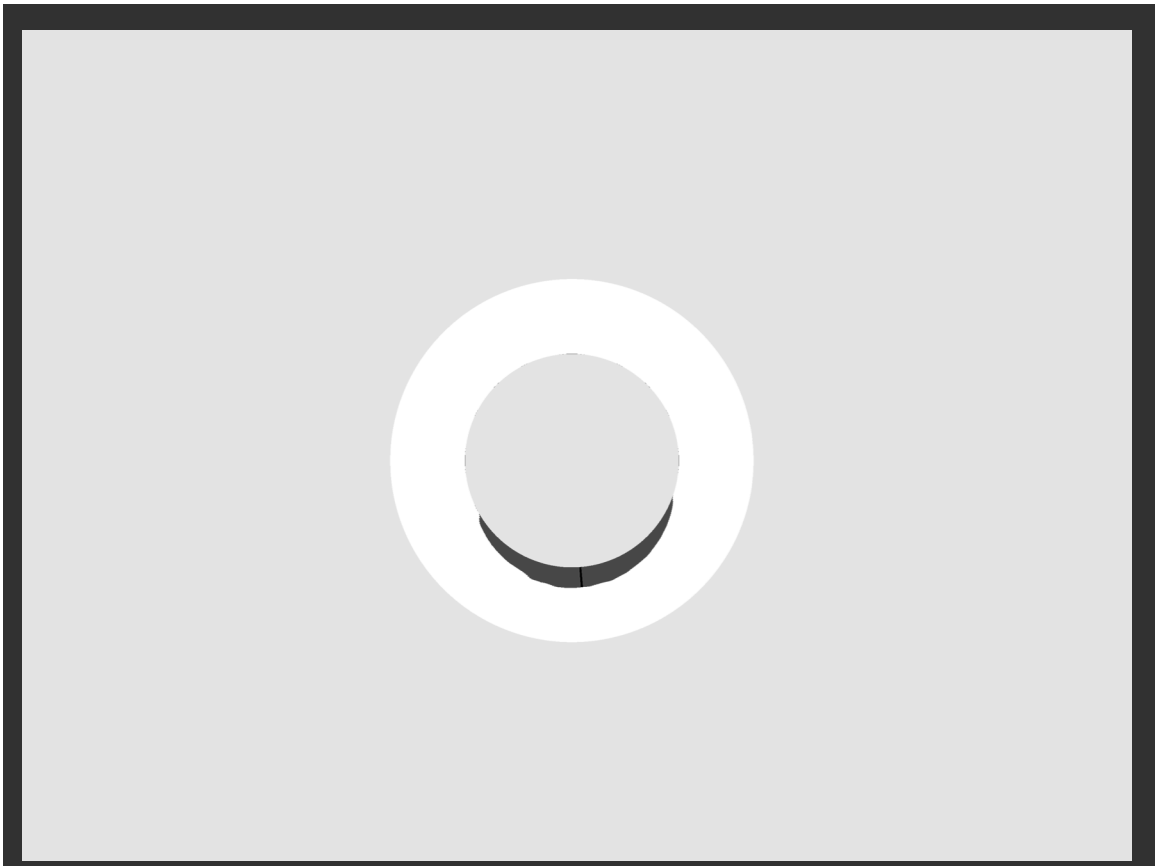
Left: Canny edge detection, Right: 200 point Hough transform

After locating the position and radius of the circular region, the goal is to identify the region of shadow, and from that shadow, determine the light angle.

This was accomplished by transforming the image into polar coordinates, such that 1 degree of rotation represented 1 pixel in the x dimension, and 1 pixel-unit of length along that rotation represented 1 pixel in the y dimension. These bucket sizes are arbitrary, but represent the precision of the measurement system.



After median filtering



The isolated shadow region

Each region in polar coordinates will consist of multiple pixels – and so those regions are summed, and averaged to generate our polar coordinate transform.

Each polar pixel is compared to the image average, and if it is less than some threshold, it is called a shadow pixel. This can result in a very noisy image – so I found it best as a first step to perform a very large median filter on the entire image, prior to performing the polar transformation. Median filtering should not alter the shadow boundaries significantly, but will greatly reduce the noisy.

The final step is to count the shadow pixels along each radial bucket. The entire algorithm looks something like this (pseudocode)

```
int degrees = 360
int totalLength = maximumShadowDistance - radius
bool[] shadow = [degrees,totalLength]
int[] shadowCounts =[degrees]

for (int degree < degrees)
  bool inShadow = true
  for (int t < totalLength)
    if ( shadow[degree, t] )
```

```
  if( inShadow )
    shadowCounts[degree] ++
else
  if(shadow[degree, t-1])
    inShadow = false
```

This iterates outward over each degree from the center of the image toward the edge. It counts shadow pixels until it determines we've left a shadow region. However, it assumes we are in a shadow until it finds a shadow pixel, followed by a non-shadow pixel.

Why not just terminate once we find a non-shadow pixel? The Hough circle detection is not perfect, and sometimes a few bright pixels will slip by. Still, we only are counting the shadow pixels, so a long line of non-shadow pixels, followed by some noise will not cause a problem. It will appear to this algorithm to be a 1 pixel shadow. Error, yet, but very small.

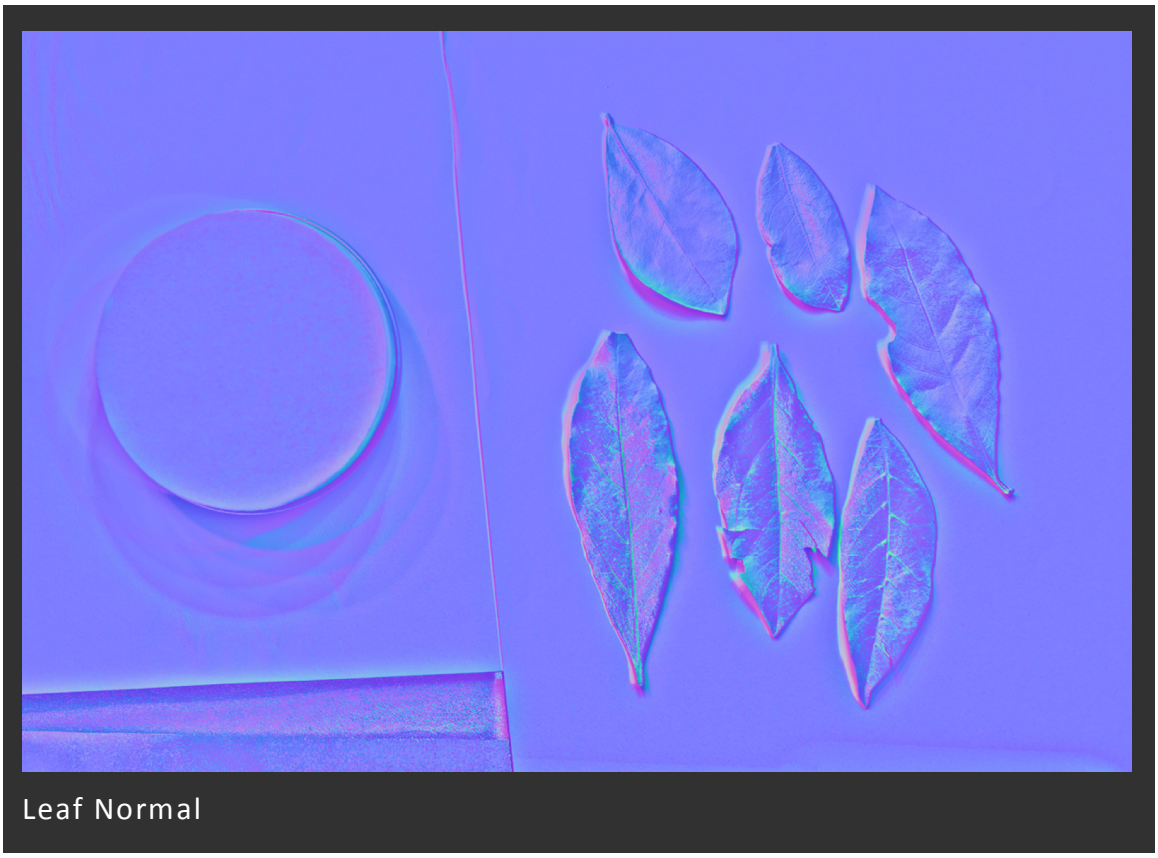
We have to be careful the shadow array is initialized to 'not shadow', so in the event anything goes wrong (such as some of the potential shadow region falling outside the frame), this algorithm does not decide there is a giant shadow just off screen. A common sense threshold (such as $1/8^{th}$ of the total distance) can also be applied to stop tracking shadow regions.

The final step is simple. Once we have shadow counts, we know the distance along each radius. To further reduce noise, I performed a 5 pixel Gaussian blur – and then performed a weighted sum of unit vectors, where each unit vector was weighted by the shadow length. This yields the average shadow direction, which we can use to look up into shadowCounts array to find the length. With direction and length, we can trivially calculate the light direction, provided we know the ratio of width to height of our cylinder, since we already know the pixel width of the cylinder from the Hough transform, we can compute the pixel 'height' – and proceed with geometry from there.

Automated Results

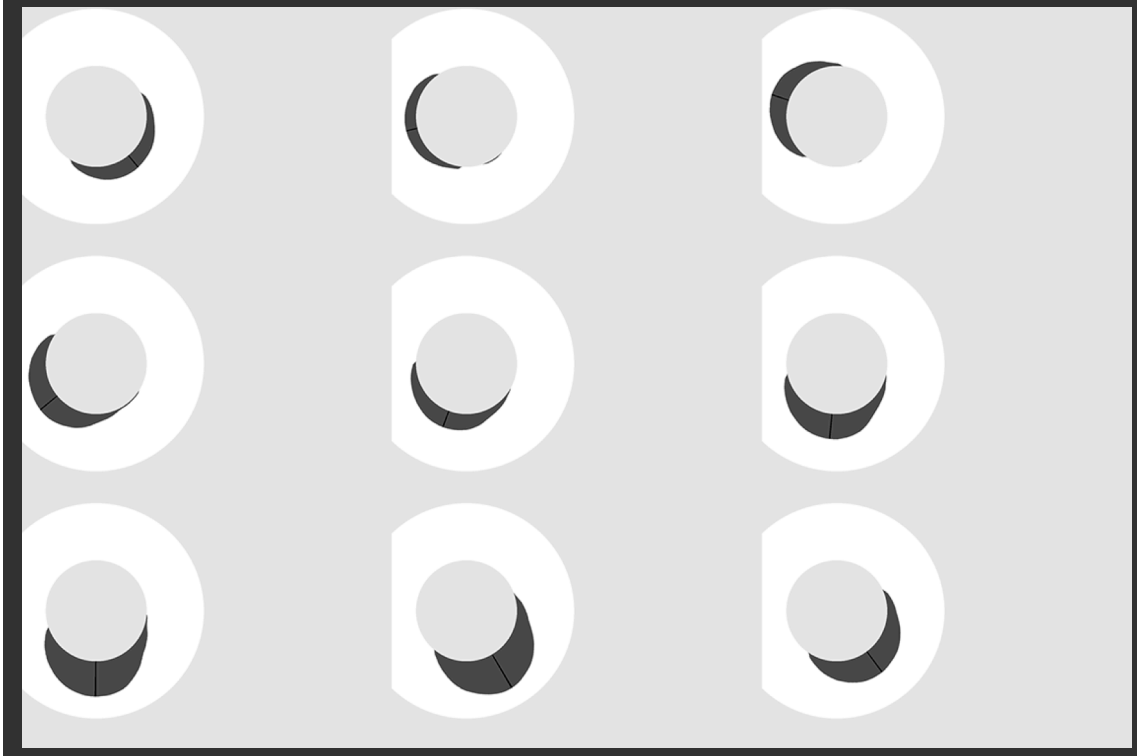
Leaves

Leaf Albedo



Leaf Normal

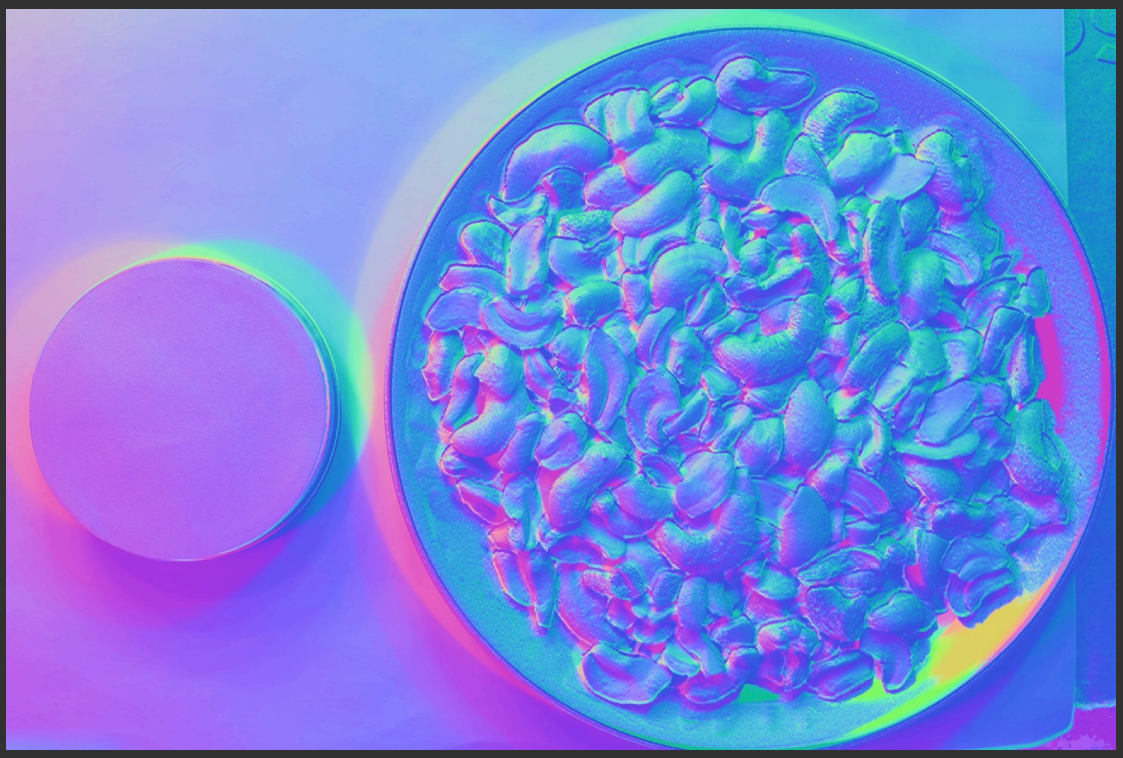
Leaf Images at 9 different lighting setups



The calculated shadows

Cashews

Cashew Albedo

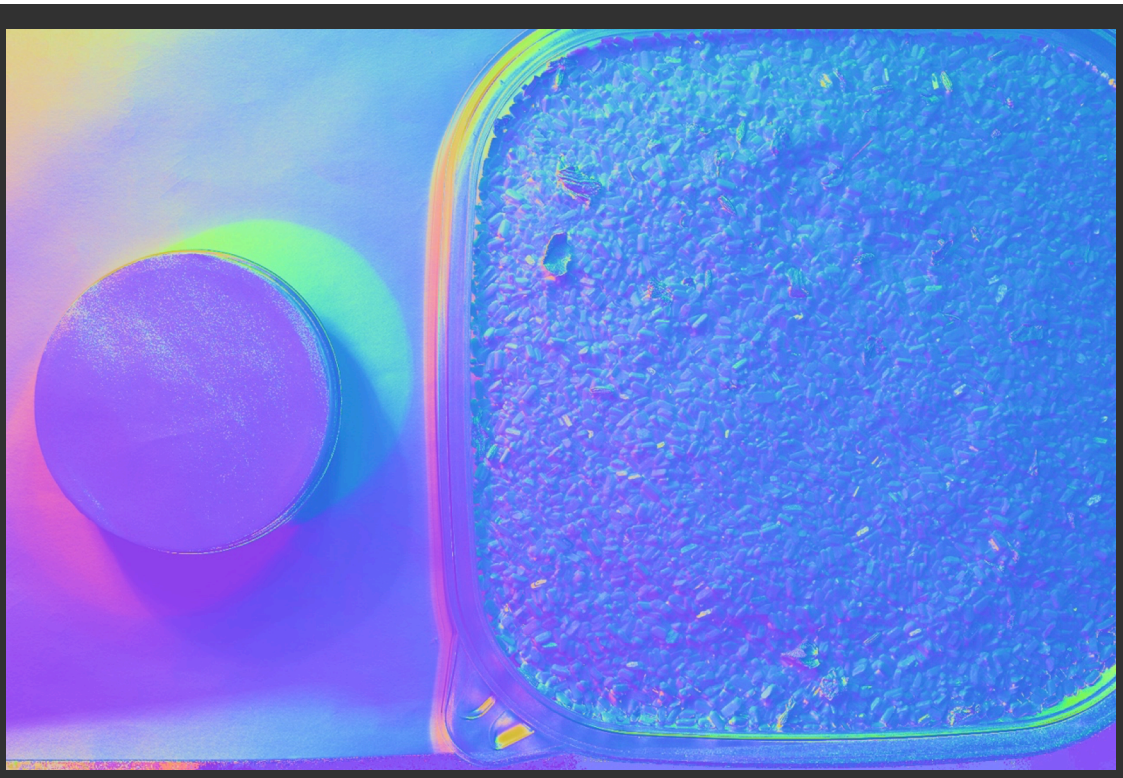


Cashew Normal

Cashew Albedo + Normal Relit in Unity

Salt

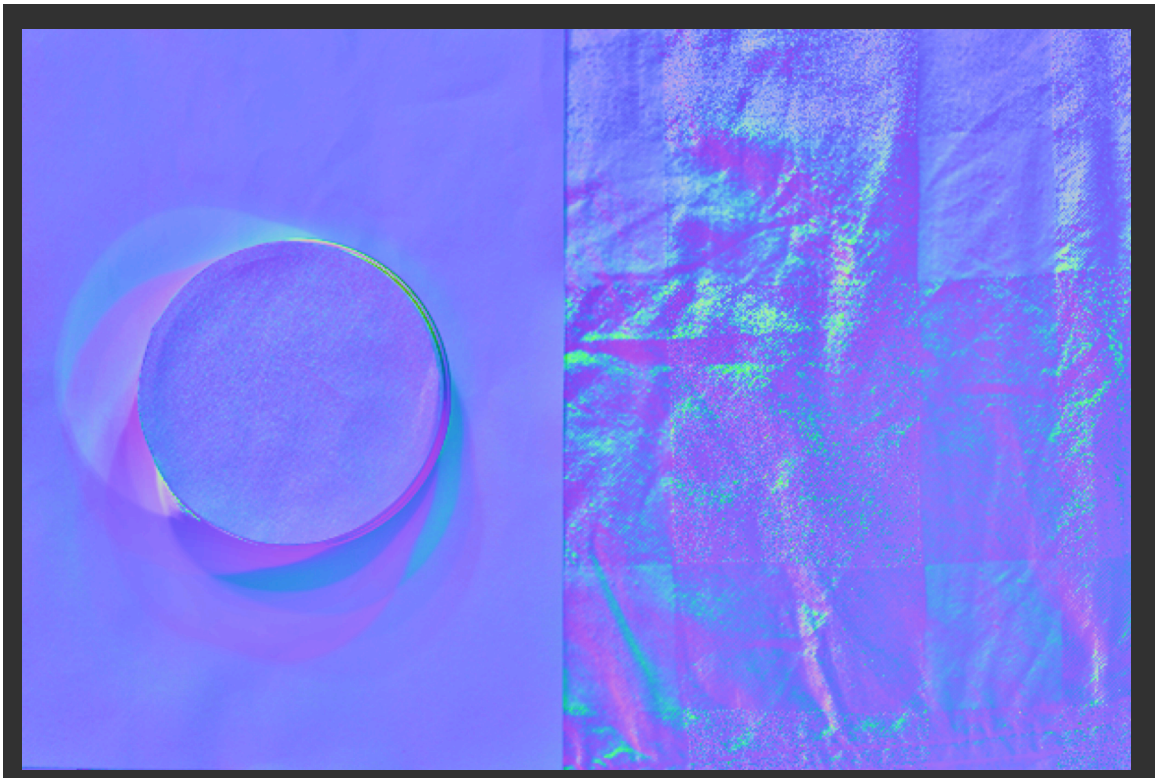
Salt Albedo



Salt Normal

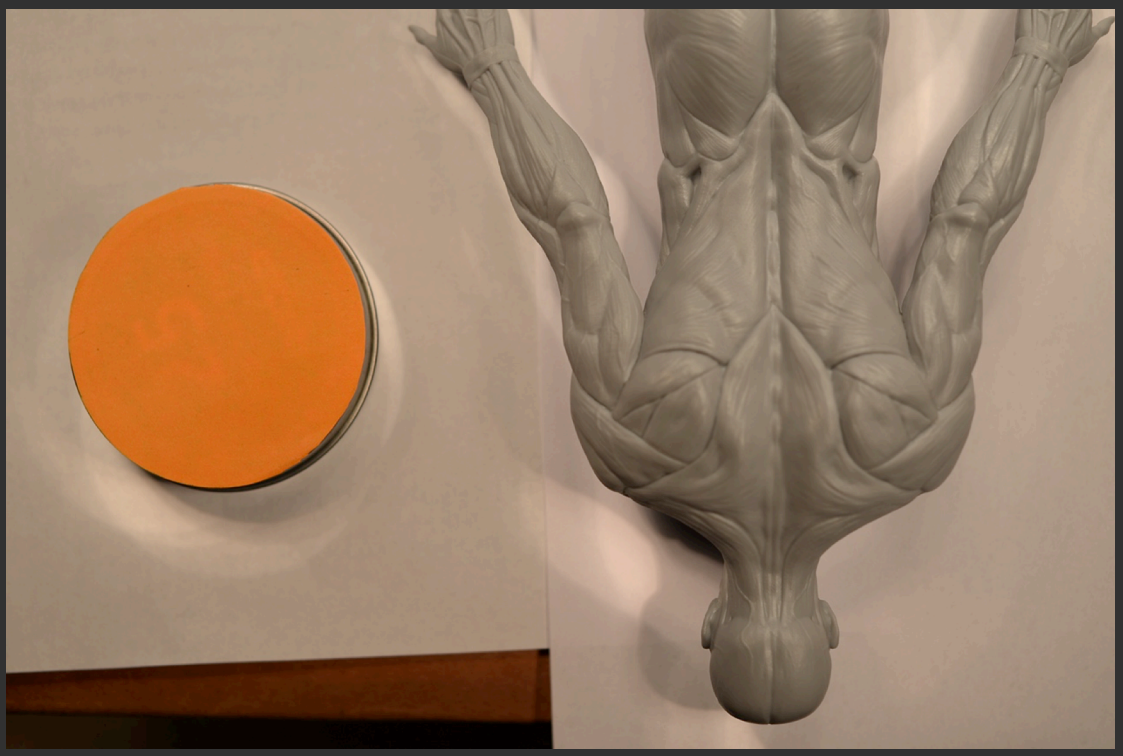
Shirt

Shirt Albedo



Shirt Normal

Man



Man Albedo

Man Normal

Issues

In practice, getting the Hough circle detection to be reliable is a bit tricky, and the shadow based algorithm is very dependent on clean results, as the light direction is calculated from the average. To improve results without overfitting to a single image set, I ended up averaging the circle radius and position between all the images. There is definitely room for improvement here, in particular, taking advantage of more sophisticated techniques.

Areas of the albedo which are actually black break the algorithm, since when we attempt to divide the normal by the magnitude, we end up dividing by zero. For that reason, slight overexposure is helpful, and black textures are not very suitable. For small flecks of black, it should be possible to average out the normal map into the offending regions. The shirt exhibited a number of these issues, and consequently, the normal map is rough where it should appear smooth.

Specular highlights are not handled by this setup, and cause errors.

Room for Improvement

Ready at Dawn makes mention of a few techniques I have not applied, all of which are good, and compatible with this method.
The input data from the camera should be linearized (gamma removed) and calibrations for the specific sensor made.
Second, lens distortion should be corrected. I chose a very low distortion macro lens and used the center of the image area to mitigate that issue.

References

- NVIDIA. *The CG Tutorial.* http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter08.html.

- Pettineo, David Neubelt and Matt. *Crafting a Next-Gen Material Pipeline for The Order: 1886.* Ready at Dawn Studios. http://blog.selfshadow.com/publications/s2013-shading-course/rad/s2013_pbs_rad_notes.pdf.
- Woodham, Robert J. *Photometric method for determining surface orientation from multiple images.* https://classes.soe.ucsc.edu/cmps290b/Fall05/readings/Woodham80c.pdf.

Thanks

Professor Irfan Essa, Professor Aaron Bobick, and the Georgia Institute of Technology