2D Paletted Lighting

by Colin Neville

1. Introduction

   This paper attempts to devise a method of applying dynamic lighting to 2-dimensional, limited color images. Typically, 2D lighting does not take object depth into account and often ignores occlusion. Such lighting systems are also not suitable for art styles which use limited colors.

   The process contains three steps. The first is the preprocessing step, which consists of both automated preparation and a manual heightmap creation. Next, the occlusion of each pixel is checked, and simulated self-shadowing is applied. Lastly, the full-color shadowed image is adapted to the color restrictions.

2. Preprocessing

   To ensure the result image is limited to the desired color palette, the shader must be given a list of acceptable colors. A small program was designed to compact each unique color from a source image into an *n* by 1 texture to be passed as a sampler to the shader. The process can also be replicated by hand for specific palettes or to include additional colors.

   The standard RGB representation of color is not well-suited to determining the visual proximity of two colors. For example, a 25% difference in the hues of two colors would have more of an impact in perception than the same difference in raw red values. For this reason, the desired palette is passed in with a pre-converted HSV analog. The hue, saturation, and value levels are encoded the R, G, and B levels of a standard image. Pre-converting the palette avoids unnecessary recalculations on the GPU, and avoids the branching inherent in RGB to HSV conversion. Because the input image will be modified by the shader before color comparison, there is no reason to convert it beforehand.

   Palette and HSV generation can easily be automated when loading images. The image to the right shows the 56-color NES palette above, and the HSV encoding below.

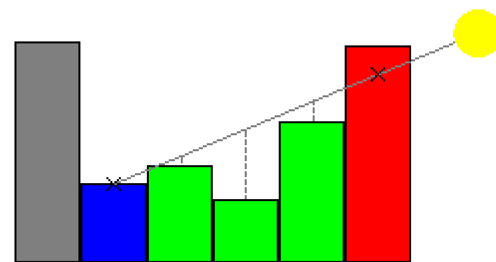
   Finally, a heightmap image must be manually generated for the input image. A heightmap is a grayscale image of the same dimensions as the input image. The pixel values of the heightmap represent z-dimension values ranging from 0.0 to 1.0 in arbitrary units. Shown are two images and their heightmaps (right, bottom).

3. Heightmap lighting

   Once preprocessing has occurred and the samplers have been sent to the shader, it must calculate self-shadowing. The initial approach was to generate a 3D mesh from the heightmap comprised of approximately two triangles per pixel. This approach, while simple to generate, had several downsides. The first, and most

obvious, was the large number of triangles resulting from even a small image. A non-transparent 64 by 64 pixel image creates a mesh of 8,712 triangles. While it is possible to simplify the mesh by combining regions with the same height values, the worst-case remains the same. Self-shadowing lighting techniques are generally costly, and tis would be compounded by the sheer number of triangles. Because image models had actual depth, clipping between multiple objects was also a potential issue.

Generating models was abandoned for a more efficient technique than takes advantage of the gridded nature of the heightmaps. Any given pixel can only be occluded by anything existing on the vertical plane which contains the ray from the pixel to the light source. Because each height is represented in a grid, the only ones that must be checked are the pixels along this ray. The pixel is then occluded if there are any heights in this plane that exist above the ray. A shader can calculate this by traversing the ray and ensuring all values in the heightmap at the point are less than the current height. The illustration demonstrates a single check: The blue is the pixel being lighting tested. Green shows heights that do not block the light source, and the red fails



The method is similar to a simplified portion of the process used by Timonen and Westerholm[i]. In this method, the maximum number of pixels sampled is the height plus the width of the image. In heightmaps composed of large flat areas, it may be possible to sample fewer pixels (e.g., every other pixel) with minimal loss in accuracy. Significant additional efficiency could be achieved by determining the convex hull of the heightmap and testing only the texels located the hull edges, as described in the Timonen paper.

This process can be repeated for any number of light sources, and neither the light sources nor the image must remain static. An ambient lighting shader parameter is also adjustable. The heightmap has no normal information, so the angle to the light source has no bearing on intensity. The lighting information for a scene with a soft light composed of five point lights is shown to the right.

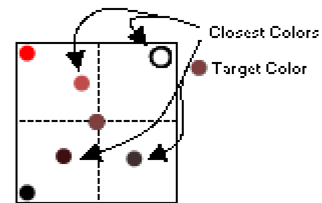
4. Color limiting

Once lighting has been determined, the resulting image must be limited to the specified palette. There are several ways to accomplish this, which each yielding the best results in different kinds of input images.

The simplest method is to assign each pixel the closest color in the palette. “Closest” in this case is determined by assigning a weight to each hue, saturation, and value and determining a total distance based on the differences. The weighting scheme used in this case was H = 0.65, S = 0.15, V = 0.20. Although the simplest, this method still requires iterating through all palette values and calculating distance. Pre-arranging the palette into a kd-tree could reduce execution time for large palettes. Ultimately, this technique fares best with both small images and limited palettes. In small

images, any added detail may obscure the object. In limited palettes, using more distant colors may distort the color scheme of the image too much.

The second technique used is to find the closest colors in each of the eight octants (in the HSV color cube) surrounding the target color. The process is illustrated for a two-dimensional example to the right. The colored circles each represent a color in the palette, and are positioned on value and saturation axes (in practice, a hue axis is used as well, dividing the cube into octants). The target color divides the space into quadrants. The closest color appearing in each quadrant is collected. For each pair of opposite octants, the sum of the distances is calculated. The pair with the shortest distance is chosen. The shader calculates the ratio between the distances of the chosen colors. It then determines which color the current pixel would be if the entire image were filled according to this ratio. For example, if color A was 0.25 from the target color, and color B was 0.75 from the target color, the fill pattern would be AAAB so that the total distances in both octants are as close as possible. The process works with any ratios, but the fill pattern is not guaranteed to be constant throughout the image. In images with specifically crafted palettes, this method works very well and applies a dithering-like effect to non-representable colors. With more general palettes though, this method may select two moderately distant colors that do not successfully represent the target color. In the example image, odd coloration is visible in the windows.



The final method is very similar to the second, but instead of selecting the two octants based on the total distance, it selects them based on the minimum distance of any octant. For example, an octant has a color with a distance of 0.01, less than any other octant. The color gets selected, as well as the color in the opposite octant, regardless of its distance. Because the frequency of each color is based on the distance, if the opposite color is very distant from the target, it will rarely ever appear. This alteration improves results on general palettes and appears to be the most accurate overall.

5. Results

Running times proved to be real-time for reasonably sized images, palettes, and light counts. Tests were run on a beat-up and poorly cooled W500 with a Mobility FireGL V5700. The examples shown all used the 56 color NES palette.

The “Troy Teapot,” a 64 by 64 pixel image, ran in the range of 400 – 1000 fps with a single light source, and dropped to about 200 fps with five point light sources for soft shadows (shown right). Without any normal information, however, the curvature of the teapot is not very evident. The process does not seem well-suited to rounded objects. Not only is the depth information poorly conveyed, generating gradient heightmaps is also much more time-consuming.

A building scene (256 by 128 pixels) which utilizes sharp edges in its heightmap showed promising results. This scene, eight times larger than the teapot, ran at about 50 fps with a single light, and around 20 with five lights. Shadows looked much better on the gray building, thanks to the multiple shades of gray in the palette. Because no colors were similar to the left building, sparse noise pixels appear where there should be shadow. When the left building’s color was substituted for a light green (one of several greens in the palette), shadows appeared properly, at the expense of a tacky-colored building.

6. Conclusion

The technique is fast enough to use for select elements in a real-time application such as a game, but not fast enough for an entire background. Unfortunately, a background would be where this technique would be most appropriate, so performance improvements are necessary. The shader still has much room for optimization, including many branches which should be removable. As mentioned earlier, encoding the palette’s HSV values into a data structure like a kd-tree could reduce a linear lookup time to a logarithmic one.

This method offers real-time dynamic self-shadowing on small scenes. In its current state, the quality of results vary depending on the scene geometry, but future additions and improvements could rectify this. In particular, attempting to estimate a normal based on local heights may improve soft shadowing on curved surfaces. Further refinements in color selection could mitigate instances of “noise” shadows. Currently images only self-shadow, but it is possible to combine separate images if their heightmaps are scaled relative to each other and overlayed. Given enough time, the entire scene could be flattened and processed at once.

References

James F. Blinn. 1978. Simulation of wrinkled surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques* (SIGGRAPH '78). ACM, New York, NY, USA, 286-292. DOI=10.1145/800248.507101 http://doi.acm.org/10.1145/800248.507101

Panagiotis Metaxas. 2003. Parallel Digital Halftoning by Error-Diffusion. In *Proceedings of FCRC2003 Paris C. Kanellakis Workshop*. http://cs.wellesley.edu/~pmetaxas/pck50-metaxas.pdf

Fábio Policarpo, Manuel M. Oliveira, Jõao L. D. Comba. 2005. Real-Time Relief Mapping on Arbitrary Polygonal Surfaces.
http://www.inf.ufrgs.br/%7Eoliveira/pubs_files/Policarpo_Oliveira_Comba_RTRM_I3D_2005.pdf

Ville Timonen and Jan Westerholm. 2010. Scalable Height Field Self-Shadowing. In *Proceedings of Eurographics 2010 Computer Graphics Forum*. http://wili.cc/research/hfshadow/