Automatic Pixel Art Shading

Motivation

Pixel art is still popular today, and is used in popular games such as *Stardew Valley*, *Game Dev Story*, *Milkmaid of the Milkyway* and *Stranger Things 3: The Game*. An automatic pixel art shading program is believed to improve the productivity of pixel art designers so that pixel art games can be produced and updated faster.

Solution outline

The solution explored in this paper is adapted from the paper "Automatic Sprite Shading" by Bandeira, D., & Walter, M. (2009). In the solution discussed in this report, the following three steps from the paper are used:

1. Highlight spots approximation
2. Shading distribution
3. Final composition

Then a final pixelation step that is not discussed in the paper is used:

4. Pixelation

Implementation outline

The shading program is implemented in Python using the library “opencv.” The program is only a simple starting point, and is tested on simple shapes such as circles and ovals. It cannot shade more complex shapes such as hearts.

Introduction

The program discussed in this report attempts to automate the step of shading in pixel art design, which makes 2D art look more 3D. This report discusses a Python program which shades simple sprites, attempting to make them look more three dimensional.

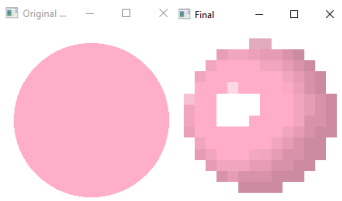

Figure 1: the program discussed in this report turns the sprite on the left to the shaded and highlighted pixel-art sprite on the right

Definitions

Highlight spot: spot of highest light exposure

Base color: the original color of a segment of the sprite before any shading

Base color intensity: the intensity of the base color in the HSI color model

Sprite: a small bitmap of an object or character in a game

Light position: position of the light source

Solution

Step 1: Highlight Spots Approximation (implemented as described in the paper)

In this step, we find the pixel on the sprite with the highest light exposure according to the light position, which is prespecified by the user. This step is implemented exactly as described in the paper.

1. The sprite is segmented in the x-direction and y-direction as shown in the second and third pop-up windows in Figure 1. The red lines are the center pixels. The center pixels are derived from the formula

   center = start pixel position + light position * (end pixel position – start pixel position)

   Where the start pixel is the first pixel inside the sprite in the respective row or column and the end pixel is the last pixel inside the sprite in the respective row or column. In Figure 2, the start pixels are in green and the end pixels are in blue.

2. The intersection of these two segments (red lines) is the highlight spot, i.e. the spot with the highest light exposure

This step is responsive to the light position. Changing the light position yields a different highlight spot as shown in Figure 3.

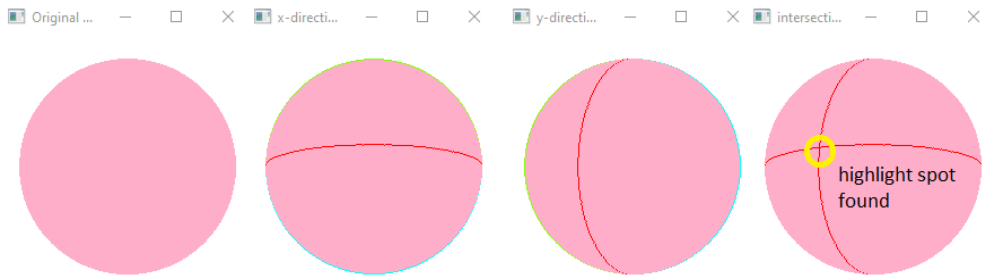

Figure 2: Finding highlight spot

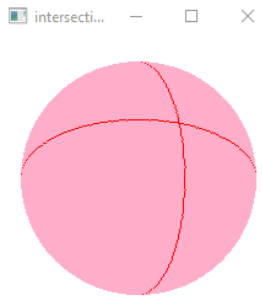

Figure 3: shifting the light position to be at the right of the sprite

Step 2: Shading distribution (implemented as described in the paper)

Here, we get a shading distribution, where we assign each pixel an intensity in the range [0.0, 1.0].

Figure 4 depicts such a distribution that is found by the program.

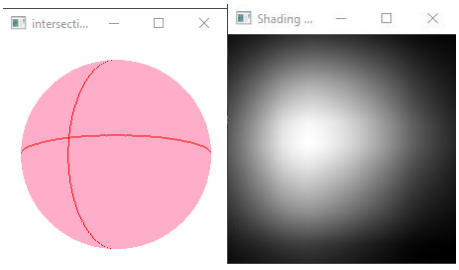

Figure 4: the shading distribution associated with the image on the left

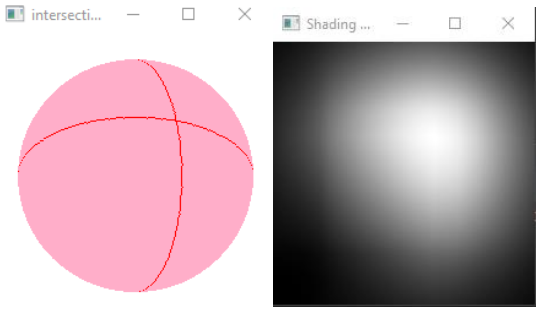

Figure 5: the shading distribution is affected by the light position

To get the shading distribution, we first assign each pixel in the sprite one of three weights: 0, 1 or 2 to make a weights matrix. A weight of 0 is given to all pixel that are not a part of one of the two segments (depicted by the red lines in the figures above), a weight of 1 is given to all pixels that are on one of the segments and a weight of 2 is given to the highlight spot. This weights matrix is then blurred with a large kernel size, depending on how smooth we want the shading distribution to be. The larger the kernel size, the smoother the shading distribution will be and so the smoother the shading in the resulting shaded sprite. An example of a kernel size is 121 x 121. We then blur the result a second time to reduce sharpness with a smaller kernel size this time, for example 99 x 99. The blurred weights matrix is then normalized so that its values are in the range [0.0, 1.0]. The higher the value, the brighter the pixel should be. As can be seen from figures 3 and 4, the order of brightness is: 1. the highlight spot and pixels nearest to it (white) 2. the pixels nearest the segments (greys) and 3. all other pixels (black).

Step 3: Final composition (not implemented as described in the paper)

Finally, we shade the sprite using the shading distribution obtained from the previous step. To do this, we convert the RGB sprite into HSI so that the intensity of each pixel can be directly

manipulated according to the shading distribution. The code assigns new intensities to each pixel as follows:

If the shading distribution value of a pixel is less than or equal to 0.3, the new intensity of that pixel becomes 80% of the base color intensity. If the shading distribution value of a pixel is less than or equal to 0.3, the new intensity of that pixel becomes 85% of the base color intensity. Similarly for values 0.4, 0.5, 0.6 and 0.7, with the percentage of the base color intensity increasing in increments of 5 (90%, 95%, 100%). If the shading distribution value for the pixel is less than or equal to 0.8, the new intensity becomes the minimum of 105% of the base color intensity and 255, and if the shading distribution value for the pixel is less than or equal to 0.9, the intensity becomes 255 (maximum intensity). Finally, to add a strong highlight effect near the highlight spot, any pixel with shading value greater than 0.9 has its saturation changed to 0 and its intensity changed to 255 which gives a highlight effect.

```
base_intensity = hsi_art[row, col][2]
if shading_distribution[row, col] <= 0.3:
    hsi_art[row, col][2] = base_intensity * 0.8
elif shading_distribution[row, col] <= 0.4:
    hsi_art[row, col][2] = base_intensity * 0.85
elif shading_distribution[row, col] <= 0.5:
    hsi_art[row, col][2] = base_intensity * 0.9
elif shading_distribution[row, col] <= 0.6:
    hsi_art[row, col][2] = base_intensity * 0.95
elif shading_distribution[row, col] <= 0.7:
    hsi_art[row, col][2] = base_intensity * 1
elif shading_distribution[row, col] <= 0.8:
    hsi_art[row, col][2] = min(255, base_intensity * 1.05)
elif shading_distribution[row, col] <= 0.9:
    hsi_art[row, col][2] = 255
else:
    hsi_art[row, col][1] = 0
    hsi_art[row, col][2] = 255
```

Figure 6: part of the implementation of the final composition step in Python

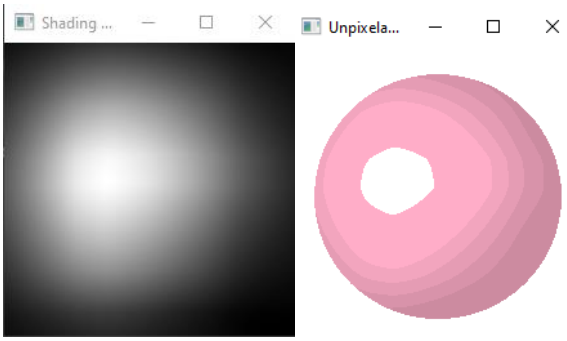

Figure 7: result after the final composition step

Step 4: Pixelation

Without pixelating the sprite, the shading will not have jagged edges between the shades, so the result would not look like typical pixel art. So in this step we pixelate the result of the final composition step. This is done by first reducing the size of the sprite, for example to a size of 16 x 16 and using linear interpolation. Then the sprite is again resized to its original size using nearest neighbor interpolation. This is done with the OpenCV “resize” method.

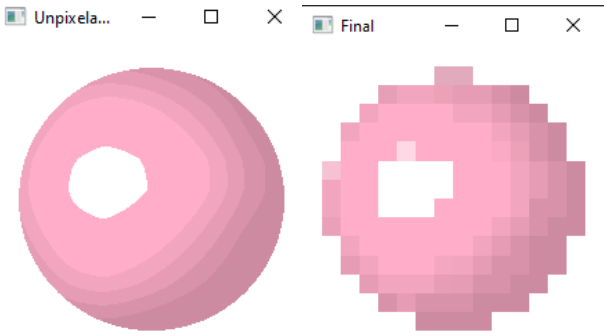

Figure 8: final result

Limitations

The pixelation step distorts the outline of sprites that have an outline around them. It also does not work on more complex shapes such as hearts or torus shapes.

Advantages

The program not only shades pixel art, but it also turns art that is not already in pixel-art style into shaded pixel art (see figure 6). Also, the results of the program are highly detailed as can be seen in figures 7-9, with several different intensity levels of shades and highlights. Shading in this way could be too tedious and time consuming for humans.

Results from running the program

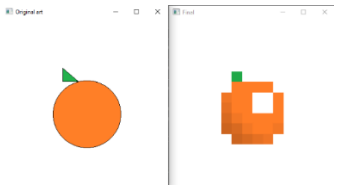

Figure 9: Turning an orange which is not in pixel art style into shaded pixel art

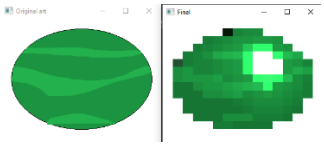

Figure 10: Watermelon; black outline almost disappears in the result

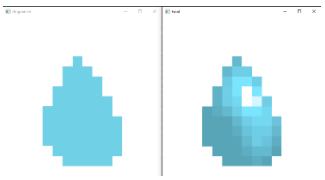

Figure 11: water drop

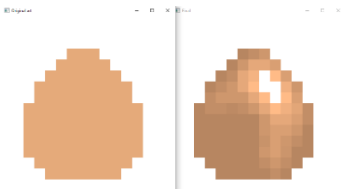

Figure 12: Egg

References

Bandeira, D., & Walter, M. (2009). Automatic Sprite Shading. *2009 VIII Brazilian Symposium on Games and Digital Entertainment*, 27–31. https://doi.org/10.1109/SBGAMES.2009.12

Contribution

All the images in this report have been produced by Mariam Ahmed

Presentation – Mariam Ahmed 100%

Report – Mariam Ahmed 100%

Source code – Mariam Ahmed 100%