One-bit dithering

Olof Ekenberg, Michal Pomorski

***Abstract*—To achieve an aesthetic effect dithering was applied to a 3D scene. Ordered and blue noise dithering was investigated as well as their variants sampled from a cube map. Highlighting of edges was used to preserve details.**

I. Introduction

Our project was to implement a one-bit dithering shader, using different techniques to test their validity. We were inspired by the video games *Return of the Obra Dinn* and *Who's Lila*.

Dithering is a technique which uses patterns of dots in order to form gradients, with the aim of obtaining an image that looks like it has a wider range of colors than it actually has. This can be used both as a way of reducing the file size (as you need fewer colors to represent the image), to break up color banding, or as a stylistic choice. If only two colors are used with the dithering, it is referred to as one-bit dithering. In this work, we see the technique as a subjective stylistic choice, rather than a method of objectively improving the image quality.

It is trivial to apply dithering to a static image, but with a moving scene and/or moving camera it can result in certain unintended visual effects. This may be the reason why *Who's Lila* uses stationary cameras for most of the game, as moving objects (such as the player model) can sometimes "flicker" while moving. The game also uses lines around certain objects, such as non-player characters, possibly to give them a clear outline.

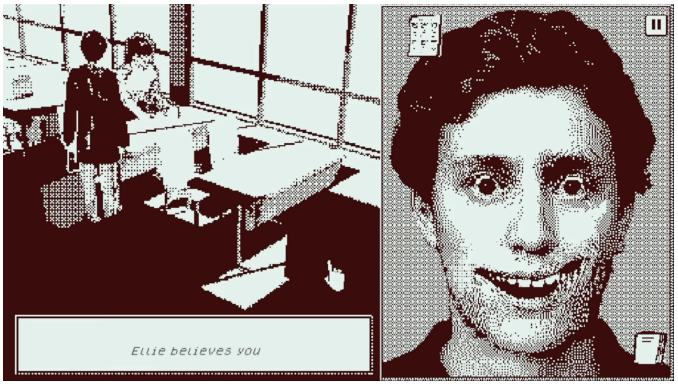

Figure 1. A screenshot from *Who's Lila*, taken from the YouTube channel ManlyBadassHero[1].

Return of the Obra Dinn meanwhile spent much time trying to iron out these unintended effects, as the developer wrote a blog post discussing the methods they used[2]. It was the basis for much of our research on dithering techniques. It also inspired us to use blue noise dithering as well as highlighting to make objects clearer.

II. Method: Algorithms and Application

A. Dithering

The first form of dithering we implemented was *ordered dithering*, which uses screen coordinates to sample from a threshold map. We followed the tutorial by Aeris[3], and used the same Bayer matrix.

```
mat4 dither_pattern = mat4(
  -0.5,     0.0,    -0.375,   0.125,
   0.25,   -0.25,    0.375,  -0.125,
  -0.3125,  0.1875, -0.4375,  0.0625,
   0.4375, -0.0625,  0.3125, -0.1875
) ;
```

Figure 2. The Bayer matrix used in our program. The values in the matrix span from minimum brightness to maximum brightness, with no repetitions, ordered such that large values are inter-spaced with small values. The idea is to compare each pixel in the original image with the one in the matrix and color white those that are brighter. As a consequence, a solid gray block, the size of the matrix will be scattered with white pixels in proportion to the brightness of the block.

There seems to be two general approaches to dither sampling; either adding the threshold and source image together before sampling with cutoff 0.5, or by using the threshold map as a cutoff. Aeris used the former approach, which seems to use more operations, but it does also make it easier to adjust the "intensity" of the threshold map. The general formula for this approach is:

$$out = getclosestvalue(in + intensity * map(screen_x, screen_y))$$

We also added the option to use a blue noise texture rather than a Bayer matrix. Calculating blue noise dynamically is quite intensive, so we instead used a pre-computed free-to-use texture from Christoph Peter's blog *Moments in Graphics* [4].

B. Application

The effect can be used as a post-processing step for the whole scene or as a color shader for each individual triangle. Applying it on each object individually allows the dithering pattern to remain permanently attached to each surface as it moves in the scene. This might be desirable because it reduces flickering and noise during movement. The difficulty is in achieving pixel-accuracy of the pattern, as much of the graphics pipeline relies on interpolation depending on depth and orientation. When the effect is applied on the whole screen, the result might be seen as a kind of static mesh held in front the screen - especially noticeable in a moving scene.

An intermediary solution, as applied by [2], is to make the pattern remain static with respect to camera rotation, relying on the relative stability of the scene otherwise.

C. Cube mapping

The *Return of the Obra Dinn* blog post[2] mentions cube mapping as a way to improve pattern stability, so we also added options to sample based on a cube map rather than screen coordinates. Since the Bayer matrix used for ordered dithering is hardcoded into the shader, we added another texture from Peter's blog as a Bayer texture when using cube map.

GLSL has a built-in cube sampler, which we use. For this sampler to work, we need a direction vector from the camera position to the pixel position expressed in world coordinates. By using the depth of the pixel and its screen coordinates we calculate the clip-space coordinates, and then multiply the clip-space coordinates with the inverse-view-matrix to get the world-space coordinates.

When doing cube-mapped dithering, the resolution of the texture used plays an important role. Ideally, we want one pixel of the cube-map texture to correspond to one pixel on the screen, but since the edges of the cube are further away than the faces, this can be quite tricky. Having a too high resolution can also cause problems, as this can lead to flickering.

D. Outlining

Because pixels are thresholded against values taken from a specific pattern, some details will be lost due to them not aligning with the dithering pattern. One way to make these visible is to outline the edges. This is much more feasible in a 3D scene than in a photograph, due to access to the geometry of the objects in the scene. In particular, information about the normal vectors of surfaces as well as the distance from the camera can be used instead of the final pixel color to achieve a cleaner result.

The method we use for finding edges in a image, *the Sobel filter*, is based on highlighting differences between pixels in the x and y directions separately. Mathematically this is achieved by applying differentiating convolution kernels to the image.

$$s_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}, s_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \quad (1)$$

The two kernels used to find edges in the scene are presented in equation 1. The result of the operation is the L^2 norm of the two convolutions.

In order for the highlighting to naturally blend with the dithering, the lines should not be at full brightness and the dithering should be applied after the outlining. We found that fading the intensity of the highlight with the depth of the scene to be particularly pleasing.

III. Results

A selection of screenshots and the settings used to achieved them are shown below. Note that the images may be compressed which may result in artefacts not seen in the actual shader.

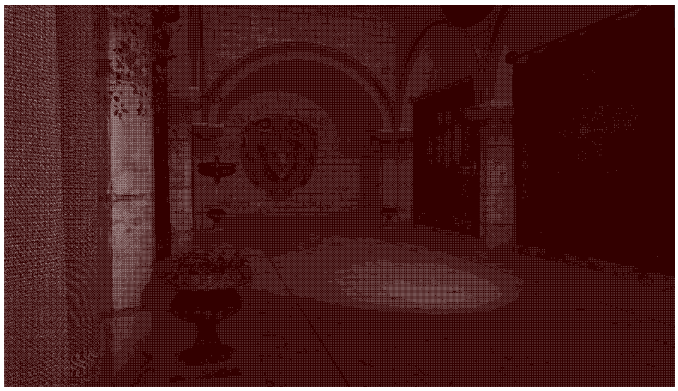

Figure 3. Ordered dithering

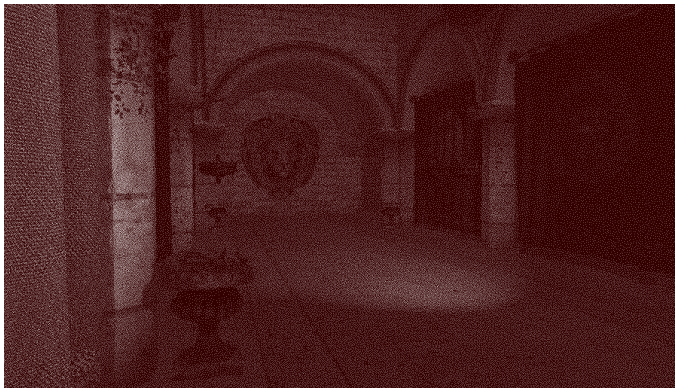

Figure 4. Blue-Noise dithering

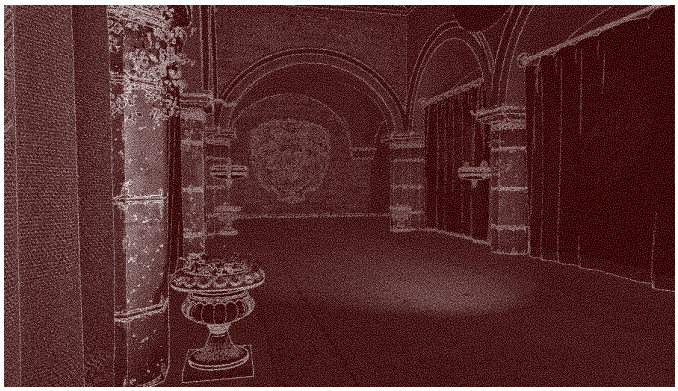

Figure 5. Blue-Noise dithering with Sobel

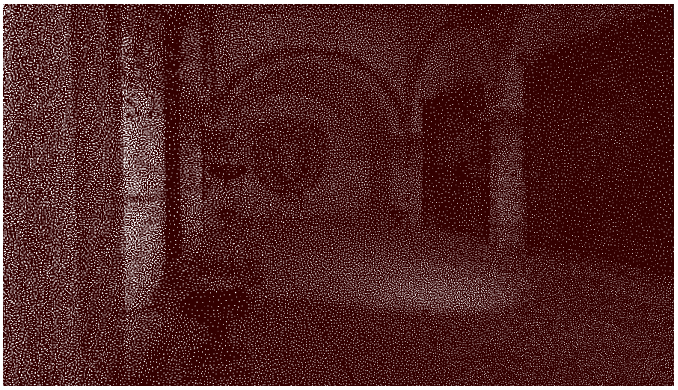

Figure 6. Cubemapped blue-noise dithering

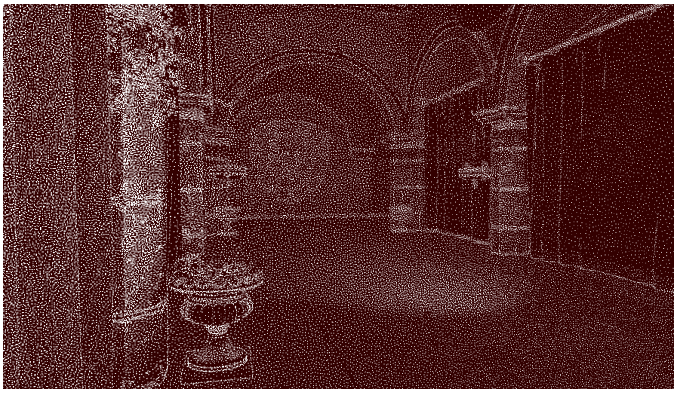

Figure 7. Cubemapped blue-noise dithering with Sobel

IV. Discussion

As one-bit dithering is a non-photorealistic shader, each of these shaders have their benefits and drawbacks depending on what is desired.

Ordered dithering results in a very "geometric" image where the cross-hatch pattern is very evident. This kind of dithering was widely used in 8 and 16-bit graphics and thus provides for a "retro" aesthetic. It may be less desirable if a more smooth image is wanted, and the pattern becomes very obvious when the camera is moved or an object on the screen moves.

Blue-noise dithering results in a clearer image overall, and has a less obvious pattern. This can make the image seem blurry or noisy, but paradoxically also makes more details visible compared to ordered dithering. Even though the dithering pattern is still glued to the screen, the more irregular pattern makes this harder to notice.

Cube-mapped dithering turned out to be quite tricky. Ideally, each pixel on the screen should map to one unique pixel, which is hard to do since the pixels near the corners occupy a smaller portion of the screen compared to pixels on the cube's faces. This meant we had to make a tradeoff between having a thicker dither pattern, resulting in a blurrier image, and having the pattern flicker when turning the camera, depending on the resolution of the cube texture. Overall the effect isn't that great, and without edge highlighting it is hard to tell what is going on.

Our favorite shader combo we developed is probably the one with blue dithering and Sobel filter. This gives a clear image without too much visual clutter or noticeable flickering, even if it could be better. The edge highlighting really helps detail pop, especially in areas of low dynamic range.

V. Conclusion

Our main takeaway from this project is that dithering is a technique that is easy to implement, but hard to master. The core principles of dithering is just sampling from a texture or matrix and thresholding, which are both very simple processes to implement in shading. However, the naive implementation of dithering also causes possibly unwanted artefacts, which are hard to counteract, and the dithering effect can obscure certain aspects of the image.

We have gone over some of the methods used to counteract these artefacts, namely blue noise, cube mapping, and sobel outlining. Out of these, blue noise was the most straightforward to implement, while cube mapping was far more tricky to fine-tune.

References

[1] *Who's Lila? - Horror Game Where You Use Your Face To Talk smile plz [1]*, ManlyBadassHero, screenshot taken from 17:39,https://www.youtube.com/watch?v=HVRkMNP_Luk

[2] Dukope, forum post, retrieved December 13, 2023 from https://forums.tigsource.com/index.php?topic=40832.msg1363742#msg1363742

[3] Aeris, *Dithered Shading Tutorial*, retrieved December 13, 2023, from https://medium.com/the-bkpt/dithered-shading-tutorial-29f57d06ac39

[4] Christoph Peter, *Moments in Graphics*, retrieved December 13, 2023, from https://medium.com/the-bkpt/dithered-shading-tutorial-29f57d06ac39