Customizable Stereogram Creator

Sam O'Connor (oconns5@rpi.edu) and Sophie Richards (richas3@rpi.edu)

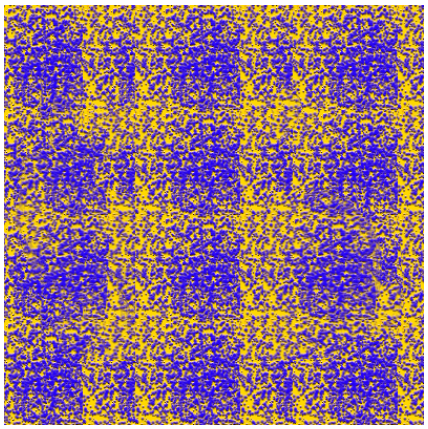

Figure 1: Stanford Bunny Custom Autostereogram

ABSTRACT

We present a customizable system that allows for users to create personalized autostereograms. We provide options for the construction of random-dot stereograms and custom pattern stereograms, alongside personalized user input for meshes, image size, oversampling, and eye convergence distance. This system outputs a single-image optical illusion wherein the user must adjust their vision to experience the appearance of three dimensions and depth.

Keywords: autostereogram, optical illusions, random-dot stereogram, stereoscopy, stereoscope, stereogram

1. INTRODUCTION

1.1 Background

Stereoscopy is defined as a technique for generating the illusion of three dimensional depth and space using two dimensional images or video, and it's something that's been researched for centuries. Its earliest documentation dates back to the early 19th century - in 1838, Sir Charles Wheatstone made note of the fact that the left and right eyes see slightly differently when it comes to perspective. Wheatstone discovered that "[there is] no difference between the visual appearance of an object in relief and its perspective projection on a plane surface…" [6] This physiological discovery led him to create the Wheatstone stereoscope. These were novel devices that contained two versions of the same image (as seen by each eye), and used reflective mirrors and refractive prisms to combine the two images into a composite. This composite was dubbed a stereogram, and within it contained the mirage of three dimensions; some elements of the picture would appear closer to the eye than others. These magical scopes are seen as some of the earliest forms of virtual reality, and stereoscopic technology has become much more commonplace nowadays. Modern examples of this are 3D movies

(popularized by studios like IMAX), and virtual reality goggles, like the Oculus Rift and the HTC Vive.

1.2 Related Work

As the computer graphics community began to grow, computer-generated stereograms became an area of interest in the late 1970s. Morland [1] produced the first known computer-generated stereoscopic space, using the GENIGRAPHICS system created by General Electric. From there, the stereogram niche was given room to blossom.

Around the same time, Christopher Tyler and Maureen Clarke were working to combine the pre-existing theories regarding wallpaper stereograms and random-dot separate-image stereograms. By 1990, Tyler and Clarke [5] had discovered the computer's ability to combine two stereograms created with random black and white pixel values, and created the first random-dot autostereogram. These autostereograms were able to create the illusion of space like previous stereograms, but with the distinct difference of being amalgamated into one singular image. The person viewing the image would have to do the work of stereoscope by unfocusing their eyes to see the hidden image. Their work would go on to inspire a series of books called *Magic Eye*, which featured autostereograms composed of custom patterns and colors rather than black and white dots. *Magic Eye* serves as the main inspiration for our implementation.

2. AUTOSTEREOGRAMS

Our implementation allows for the creation of two different types of autostereograms: random-dot stereograms (RDS), and custom patterned stereograms.

2.1 Random-Dot Stereograms (RDS)

To create random-dot stereograms, we provide a recreation of the algorithm developed by Thimbleby et al. [4]. We chose to base our work off of their algorithm due to its high performance and its symmetry regarding left eye and right eye bias. It produces little to no artifacts, and for its minimal drawbacks (depth stairs), we provide the solution of oversampling, described in Section 4.1 of this paper.

The random-dot algorithm cannot work without one argument: a two dimensional depth buffer with defined dimensions of the width times the height of the window. This buffer contains values from 0.0 - 1.0, and said values correspond to depth information about the inputted mesh in relation to the camera's position and perspective. Once this is obtained, pixel values are generated in scan-line order. At each pixel location, the stereoscopic separation is calculated using the depth buffer by the following formula:

$$separation = ((1 - \mu * z) * e / (2 - \mu * z))$$

where μ is the fraction of viewing distance, z is the depth buffer value, and e is the eye separation. $separation$ is then rounded to the nearest whole number. Left and right eye-values are determined from the separation value by:

$$left = x_p - separation / 2$$

$$right = left + separation$$

where x_p is the x position of the pixel. From there, we determine if the current point is obscured by a surface in the foreground, and perform hidden surface removal. Values to the left and right of the current x position are checked incrementally to see if they block the current *z* value. If the current position is visible, the *left* and *right* values are used to displace the current pixel. Otherwise, it is ignored. From there, random dots are assigned to the pattern as a whole. If the pixel has not been displaced, a random value is assigned. Otherwise, the pixel's color is constrained and defined by its *left* or *right* value. This culminates by compiling all of the rows of pixels into a two dimensional array of values of 0's and 1's. 0's correspond to a black pixel, and 1's correspond to a white pixel. This completes the RDS implementation, and the array of values is passed to our image generator.

2.2 Custom Stereograms

Custom stereograms can be differentiated by their random-dot predecessor due to their distinctive patterns. These patterns can be any size and can contain all sorts of shapes and colors. From our testing, we've found that certain patterns work better than others. Smaller patterns that contain lots of color variation and "noise" help disguise the displaced pixels best. Figure 2 and Figure 3 show the difference between a bad custom pattern and a good custom pattern, respectively. Figure 2 showcases a notable outline of the Stanford bunny, while it's indiscernible what Figure 3 is an autostereogram of at a glance.

In our implementation, the algorithm is fairly similar to the random-dot stereogram. We follow the algorithm laid out by Steer [3], which differs only slightly from RDS's. The major difference between the two relates to the pixel color assignment. Instead of a random black or white color, the pixel color is defined by a region of the original pattern image. The location is determined by the x and y positions modded by the pattern width and height, respectively. Another difference is that a two dimensional array of RGB values is compiled rather than single floats. This preserves color from the original pattern. Additionally, the custom pattern is tiled beginning with the pattern placed in the center of the image. Previous algorithms for generating custom patterns begin by creating a stripe of the custom pattern on one side of the image, and deriving pixel color values from that untouched stripe [2]. Our implementation avoids this artifact, and overall, generates a more pleasing stereogram.

3. IMAGE GENERATION

To generate the resulting image, first the values from the depth buffer from the modern graphics pipeline from OpenGL are read from the GPU to the CPU. Then, said values are passed to the stereogram algorithm. Next, the resulting values from the stereogram algorithm are taken, which holds a value between 0 to 1 for each pixel for RDS, or RGB values between 0 to 1 for custom stereograms. If there is no oversampling for the RDS, then the values will only be 0 for a black dot and 1 for a

white dot; oversampling will add in-between values. These values are used to set each color of each pixel of the resulting image to be between 0 to 255. These pixels are saved as a .ppm file. This file type was chosen due primarily to the ease of reading and writing from them. All of the .ppm files have a max color value of 255, meaning the pixel data for each color is 1 byte.

4. TECHNICAL ISSUES

4.1 Depth Buffer Precision

There were a few issues that arose from the depth values. One of these issues was that the values from OpenGL's depth buffer are not linearized, so the values had to be adjusted to be linear before being passed to the stereogram algorithms.

Another roadblock we faced early on was that the near and far planes were distant from each other compared to the models that we were loading. This resulted in stereograms that appeared to be rather flat. In Figure 4, it resulted in the stereogram having about three layers of depth. The near and far planes were brought closer to each other to get a greater range of depth values from the models. This made our stereograms appear to have more depth than before. This can be seen in Figure 5.

Finally, there was the “depth stairs” issue, which defines when there is a noticeable step between one layer of depth to another. At first, we thought that this was the result of the limited precision of the depth buffer, so we made a frame buffer that had greater precision for the depth buffer. We used that to get the depth buffer values, but the resulting images still had the same amount of depth stairs. Instead, the reason we were getting depth stairs was due to the limited resolution of the images we were creating. All of the images we created were set to be 72 DPI. To get smooth images, they would have to be approximately 300 DPI [3]. This is impractical for many reasons; one reason being that a lot of monitors have less than 300 DPI resolution. Instead, to get smooth images, we used oversampling. The oversampling in the stereogram algorithm causes it to treat the inputted image as a higher resolution than it actually is, and then averaging the results to get the desired lower resolution at the end, similar to antialiasing. This resulted in stereograms that had less noticeable depth stairs, even though they were created at 72 DPI. The effects of this oversampling can be seen by comparing Figure 5 and Figure 6.

Some custom patterns can also hide the depth stairs as well. The floral pattern from [2] (used in Figure 2) resulted in smooth stereograms, despite not being the best pattern due to its color gradients. The depth stairs in the colorful dots pattern (used in Figures 3 and 9) are somewhat noticeable, but not as bad as an RDS with no oversamples.

4.2 Drawing from Custom Images

We ran into a couple of issues when trying to create the custom patterned stereograms. Namely, the resulting stereograms had colors not present in the original image. Additionally, the resulting stereogram turned out rather skewed and warped, as shown in Figure 7. This was the result of two things: reading in the image incorrectly, and disparities with different parts of the program using row major matrices and

others using column major matrices. For fixing the image reading issues, we made sure we were reading things in the right order and setting things to be the right data type by writing a copy of the image and making sure that they looked the same. Image reading was mostly fixed by doing this, with custom stereograms with no oversampling looking correct. Custom stereograms with oversampling had noticeable errors though, with pixels of different colors from the original pattern appearing around the edges of a depth layer. This was due to reading in color values in the .ppm file as a char instead of an unsigned char, resulting in negative values for colors. This would cause values that didn't make sense to pop up when adding and averaging colors together. The reason the non-oversampled custom stereograms seemed to be fine was due to the values being cast as chars when being written to the file, so those color values were the same as the values in the original image.

The issue regarding row vs. column major matrices had also turned up earlier in our implementation, with the depth buffer values being in opposite orientation compared to the stereogram algorithms. This resulted in the models in the stereogram to be rotated 90 degrees compared to their actual positioning on the screen. Both the depth buffer and the custom image data were reoriented to match the stereogram algorithm's orientation, which finally resulted in stereograms matching the perspective seen in the viewport window. This was important to us, as we wanted the stereograms to match the user-defined orientation of the mesh.

.

5. CONCLUSION

5.1 Results and Discussion

All of the resulting images were created at 72 DPI, with an eye separation of about 2.5 inches. These same values were used in the algorithm by Thimbleby et al. [4]. The actual separation needed to view these pictures may be more or less due to the scaling of these images compared to the original size they were created at. To account for this, our implementation allows this variable to be user-defined in the command-line inputs, with a default value of 2.5 inches.

Our stereogram creator can be used with a variety of meshes: closed meshes, open meshes, and even complex meshes, such as the knot model we used that overlaps itself (see Figure 8). It can use any type of pattern (though the choice of a good pattern is left up to the user), with tips of what makes a good pattern discussed earlier in Section 2.2.

5.2 Performance and Limitations

The runtime of each of our stereogram algorithms is approximately O(x*y*number of oversamples), where x and y are the width and height of the stereogram in pixels, respectively. The runtime of the image reading for custom patterns is O(x*y) where x and y are the width and height of the image provided in pixels. The image is only read once in the beginning and stored so that it does not have to be reread if the same model is used to create multiple stereograms. In practice, our stereogram algorithm may be slower than the runtime implies. This is due to two main reasons: reading the depth values from the GPU to the CPU may be slow, and the stereogram algorithms loop

through the width times the number of samples multiple times.

There are a few limitations of our stereogram creator. One limitation is that small details of a model may be lost in the created stereogram due to the limited resolution. The stereogram creator works well to capture the general shape of the model. The custom pattern selection also affects how much of the detail of the model is preserved. Another hindrance is that the stereogram creator can be noticeably slow for creating larger stereograms.

5.3 Future Work

There are numerous avenues for future additions to this system, including a few stretch goals we didn't get to implement before we ran out of time. A stereogram solver would be a good thing to add, especially since it may be difficult for people to view autostereograms if they have no experience with them. Furthermore, there exists the possibility for animated stereograms, though our current system does not allow for motion capture or live depth buffer updating. This would require some reworking of the image generation system.

5.4 Work Breakdown

Sam set up the stereogram class and implemented all algorithms for each type of stereogram generation. He spent about 12 hours on this project.

Sophie did the set up, adjustment, and visualization of the depth buffer values, the image reading and writing, and the implementation for oversampling. She spent about 14 hours on this project.

ACKNOWLEDGEMENTS

We would like to thank professor Barb Cutler for providing all the necessary starter code and meshes. A special shout out to Cutler's two cats: Licorice and Nutmeg, both of whom provided emotional support at office hours.

REFERENCES

[1] Morland, D.. (1976). Computer-generated stereograms: a new dimension for the graphic arts. ACM SIGGRAPH Computer Graphics. 10. 19-24. 10.1145/965143.563279.

[2] Randima Fernando. (2004) GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics. Pearson Higher Education.

[3] Steer, W. A. (2006, July 15). Andrew's Stereogram Pages.

[4] Thimbleby, Harold & Inglis, Stuart & Witten, Ian. (1994). Displaying 3D Images: Algorithms for Single Image Random Dot Stereograms. IEEE Computer. 27. 38-48. 10.1109/2.318576.

[5] Tyler, C.W. and Clarke, M.B. (1990) The autostereogram. SPIE Stereoscopic Displays and Applications 1258: 182–196.

[6] Wheatstone, C. (1838) Contributions to the Physiology of Vision. Part the First. On Some Remarkable, and Hitherto Unobserved, Phenomena of Binocular Vision. Philosophical Transactions of the Royal Society of London (1776-1886). 1838-01-01. 128:371–394

Figure 2: Stanford Bunny Floral Custom Pattern

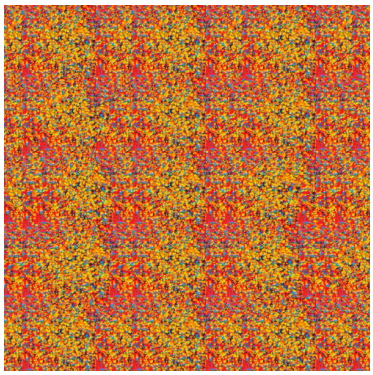

Figure 3: Stanford Bunny Colorful Dot Custom Pattern

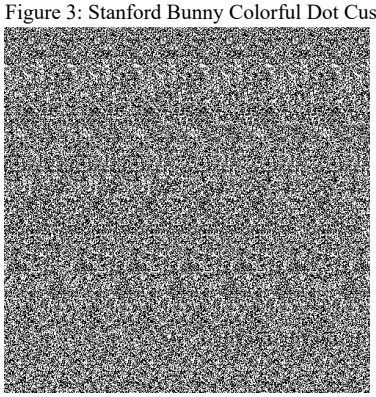

Figure 4**:** Stanford Bunny RDS (with little range in depth)

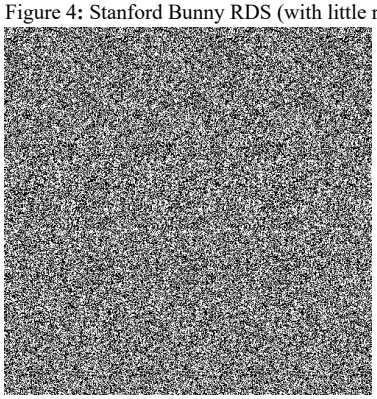

Figure 5: Stanford Bunny RDS (with no oversampling)

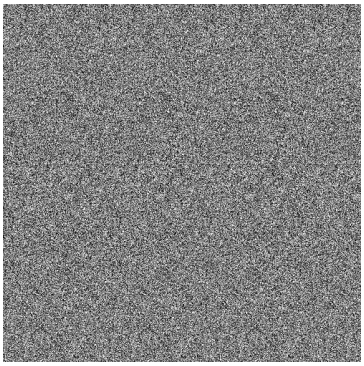

Figure 6: Stanford Bunny RDS (with 4 samples per pixel)

Figure 7: Stanford Bunny Floral Custom Pattern (with wrong colors)

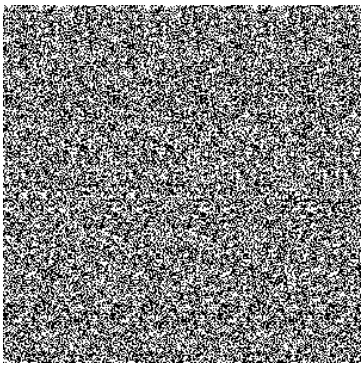

Figure 8: Complex Knot Model RDS

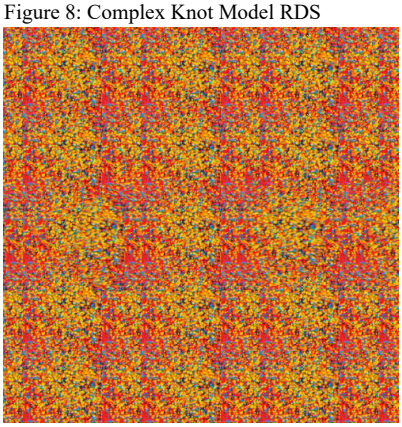

Figure 9: Custom Pattern Stereogram of a Teapot