Trypophobia: A DirectX Application

Sam Drysdale

January 10, 2023

Contents

1 Summary **1**

2 User Controls **2**

3 Techniques **2**
3.1 Normal Mapping 2
3.2 Dynamic Textures 3
3.2.1 Case Study: Squares and Cells 3
3.2.2 Case Study: Quadrilateral Distortion 5
3.2.3 Case Study: Hexagonal Distortion 7
3.2.4 Case Study: Dynamic Pores 8
3.3 Skyboxes 9
3.4 Glass Containers 10
3.4.1 Reflection 11
3.4.2 Refraction 12
3.4.3 Specimens 13

4 Evaluation **13**

References **15**

1 Summary

Trypophobia is a short showcase of the DirectX 11 functionality. Alongside demonstrations of normal mapping, environment mapping, and other common techniques, the application aims to replicate two horror-adjacent effects: the distortion of light through a mad scientist's brine-filled specimen jars, and the trypophobia-inducing unpleasantness of a surface erupting with pores. While the resulting scene (a single cluster of spheres) appears almost deceptively simple, the code required to implement it is suitably complex.

2 User Controls

Trypophobia uses the following key mappings:

W ..... Move forward
A ..... Move left
S ..... Move backward
D ..... Move right
Spacebar ..... Move up
LShift ..... Move down
Esc ..... Quit

Move mouse to look around. Application must be run at a fixed 1280×720 resolution.

3 Techniques

The following in a technical discussion of the key features built into *Trypophobia*. Code snippets are kept to a minimum, edited for clarity rather than accuracy to the original application.

3.1 Normal Mapping

Normal mapping is, put simply, a means of creating high-poly lighting effects on low-poly models. Conceptualised by Jim Blinn (of Phong-Blinn lighting) as early as 1978, the trick is based on perturbing the normals of a smooth surface by some choice of "wrinkle function". Then, when lighting is applied, the shadows created by these normals produce the optical illusion of an actually bumpy model.

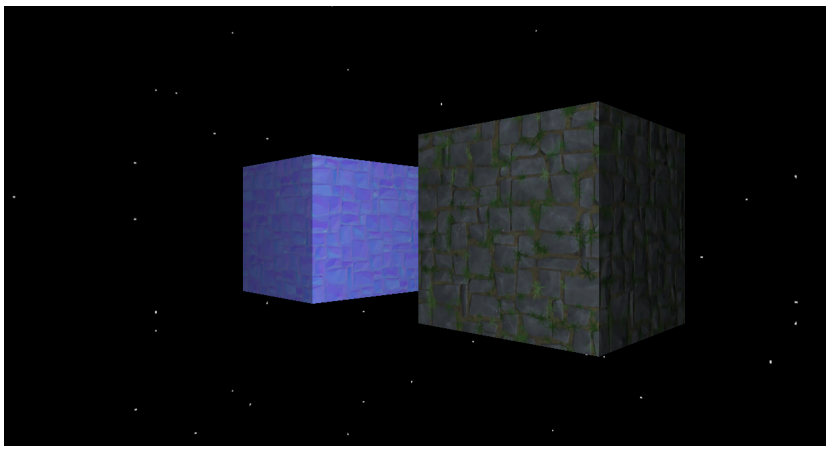

Figure 1: A normal map (left); lighting effect created by a normal map (right)

Of course, *Trypophobia*'s framework already uses normals to create lighting effects - it just does so with less sophistication. The class `modelclass` loads in a model and notes the normal data at each vertex, and the vertex shader `light_vs.hlsl` multiplies these normals by the model's world matrix, so that the pixel shader `light_ps.hlsl` can calculate light intensity by

```
float3 lightDirection = normalize(position - lightPosition);
float lightIntensity = saturate(dot(normal, -lightDirection));
```

Once reconciled with the existing vertex normals, normal maps create their lighting effects the same way; the challenge lies in that reconciliation.

First, `light_ps.hlsl` will need more data: a tangent $\mathbf{t}$ and binormal $\mathbf{b}$ to accompany every input vertex normal $\mathbf{n}$. Mathematically speaking, these could be any two vectors that satisfy conditions

$$\mathbf{t} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{n} = \mathbf{n} \cdot \mathbf{t} = 0, \;\; |\mathbf{t}| = |\mathbf{b}| = |\mathbf{n}| = 1$$

(that is to say, that form an orthonormal basis of 3D space). For ease of use with texture coordinates, though, Rastertek's normal mapping tutorial (n.d.*a*) sets "the tangent going along the x-axis and the binormal going along the y-axis," without loss of generality.

This tutorial also shows how the `modelclass` can determine tangents and binormals on loading. *Trypophobia* uses (a cleaned-up version of) the same code. The frame-by-frame calculations from here on are very simple: the vertex shader now multiplies all three direction vectors by the world matrix, and the pixel shader can at last decrypt a new normal

```
float4 normalMap = 2.0f * normalMapTexture.Sample(SampleType, tex) - 1.0f;
float3 normal = normalMap.x*tangent + normalMap.y*binormal + normalMap.z*normal;
```

to use in lighting calculations.

This is very efficient - from a technical perspective, at least. Whilst normals can be baked into maps and used to add incredible detail to low-poly models at little computational cost, a 3D artist will still need time to create the high-poly versions that the maps comes from. How then, could a programmer, devoid of any artistic merit, best showcase - show off, even - their DirectX application's normal mapping functionality?

3.2 Dynamic Textures

Not all textures are drawn by hand; some are coded. Pixel shaders are fundamental to any DirectX project, and one of their many uses in *Trypophobia* is in generating textures. By applying a pixel shader to a flat, square surface, pointing a camera at it (with the correct field of view and aspect ratio) and using the framework's built-in render to texture functionality, the camera can capture a static texture at runtime. Taking this one step further, by passing the in-game time into pixel shaders through a buffer and rendering to texture every frame, models can be assigned a dynamic texture that constantly updates!

The following case studies in dynamic texture generation have been greatly influenced by *The Book of Shaders* (n.d.*c*). While drawing textures with code might seem like a very basic technique, Vivo and Lowe make a compelling case for the complexity that emerges simply by overlaying different mathematical functions. It is that complexity that *Trypophobia* aims to replicate here.

3.2.1 Case Study: Squares and Cells

First, the basics. Colouring point $(s, t) \in [0, 1) \times [0, 1)$ by $(f_n(s), f_n(t), 0.0, 1.0)$, where $f_n(x) = nx \pmod 1$, $n \in \mathbb{N}$, gives the tiled pattern seen in Figure 2. In other words, on multiplying texture coordinates `st` by n, the resulting fractional part `fst` can create repeating patterns over an $n \times n$ square grid - with the integer part `ist` uniquely indexing each tile (Vivo & Lowe n.d.*b*).

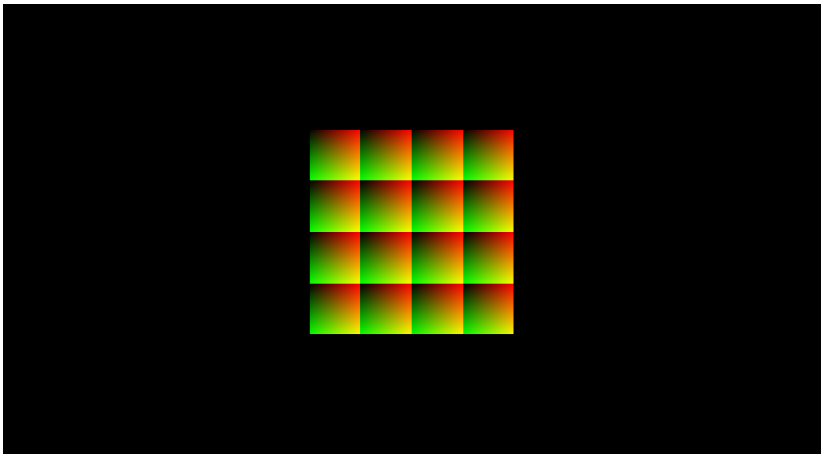

Figure 2: Colouring of a square, using f_4.

The Book of Shaders uses this underlying square grid to create Voronoi noise (Vivo & Lowe n.d.*a*). Starting with a pseudorandom function `random2`, the following returns a moving point `ivertex` guaranteed to lie somewhere on square `ist`:

```
float2 randomness = random2(ist);
randomness = float2(0.5, 0.5)+0.5*sin(time+6.2831*randomness);
float2 ivertex = ist+randomness;
```

For each point on square `ist`, the algorithm then checks which `ivertex` it is closest to,[1] and indexes the point by the square that that closest `ivertex` lies on.

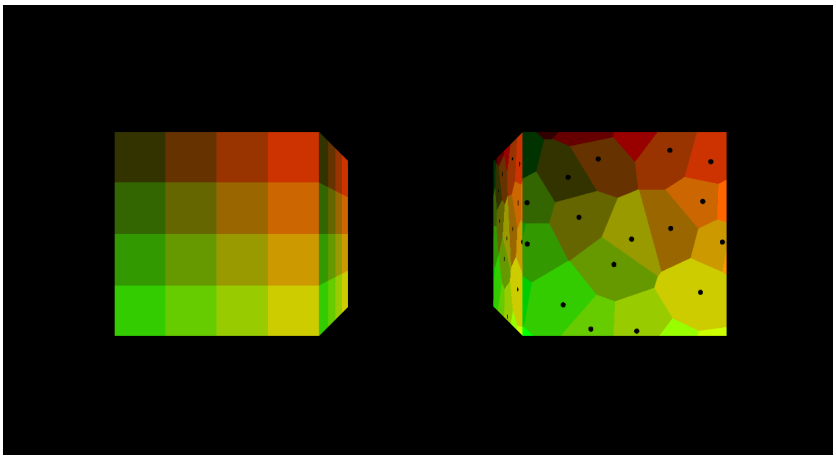

Figure 3: Square tiling (left); Voronoi noise, with a black dot representing each `ivertex` (right)

[1]Note that what is 'closest' depends on the choice of metric. Instead of 'as the crow flies' Euclidean distance, the algorithm may use Manhattan distance $d(x, y) = |x_1 - y_1| + |x_2 - y_2|$, named for the borough's grid system (Lamb 2016). It might even use the (terrible) French Railway metric, where $d_P(x, y) = \|x - y\|_2$ if points x, y, P are colinear, and $d_P(x, y) = \|x - P\|_2 + \|P - y\|_2$ otherwise - because "if you want to go by train from x to y in France, the most efficient solution is to reduce the problem to going from x to Paris and then from Paris to y" (Planetmath n.d.)

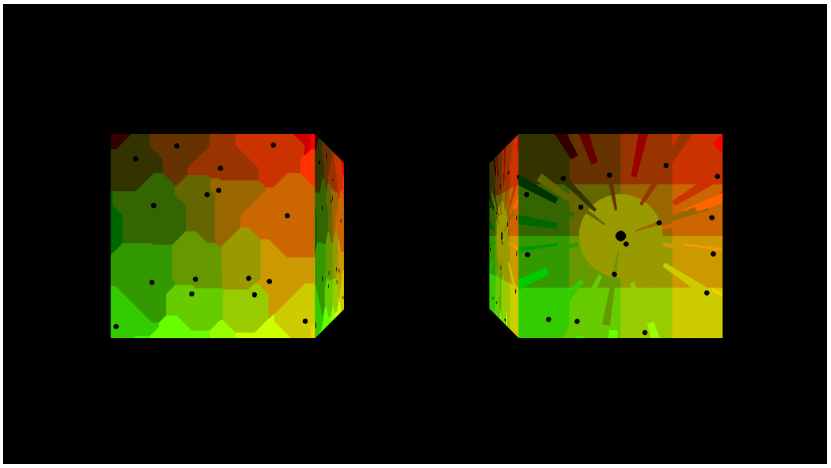

Figure 4: Voronoi noise using Manhattan distance (left); French Railway distance (right). Neither texture displays 'smooth' changes over time; neither metric is differentiable.

While this algorithm does provide an integer index, it leaves no obvious analogue to the fractional coordinates `fst`. This sets out a clear challenge for *Trypophobia*: draw inspiration from Voronoi noise to create a distorted grid on which the fractional coordinates *remain well-defined*.

3.2.2 Case Study: Quadrilateral Distortion

The attempt this application offers is, conceptually, very simple. Starting with a square grid, imagine displacing the corner of each square such that the initial pattern is stretched and squeezed. Since the grid started with well-defined fractional coordinates, it makes sense that these too would - somehow - become distorted, yet remain well-defined. The source's `tiling_irregular_quad.hlsl` shader is based on this intuition.

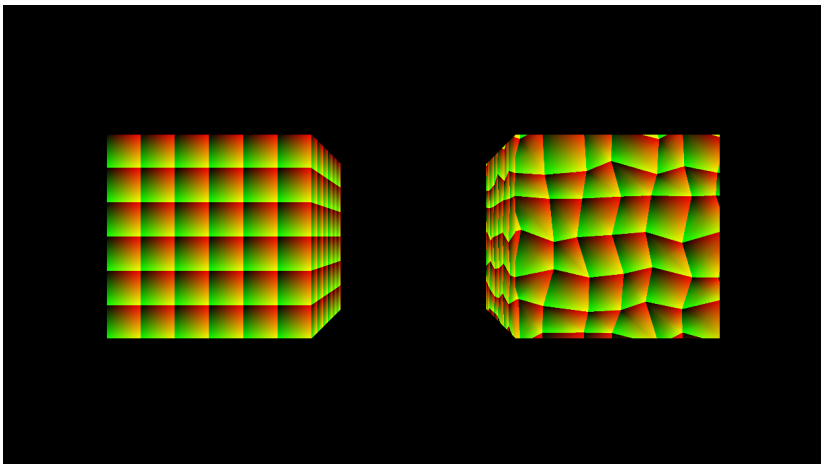

Figure 5: Initial quadrilateral grid (left); distorted quadrilateral grid (right).

The first step is to define integer coordinates for each quadrilateral. Starting on a square grid, the shader takes a tile and the four tiles immediately adjacent to it, and for each calculates an `ivertex` as in the Voronoi algorithm above. Drawing lines from the centre square's `ivertex` to the four

others will partition it into four sections. The integer coordinates of any point on this centre tile can then be determined from the section on which it lies, as in Figure 6.

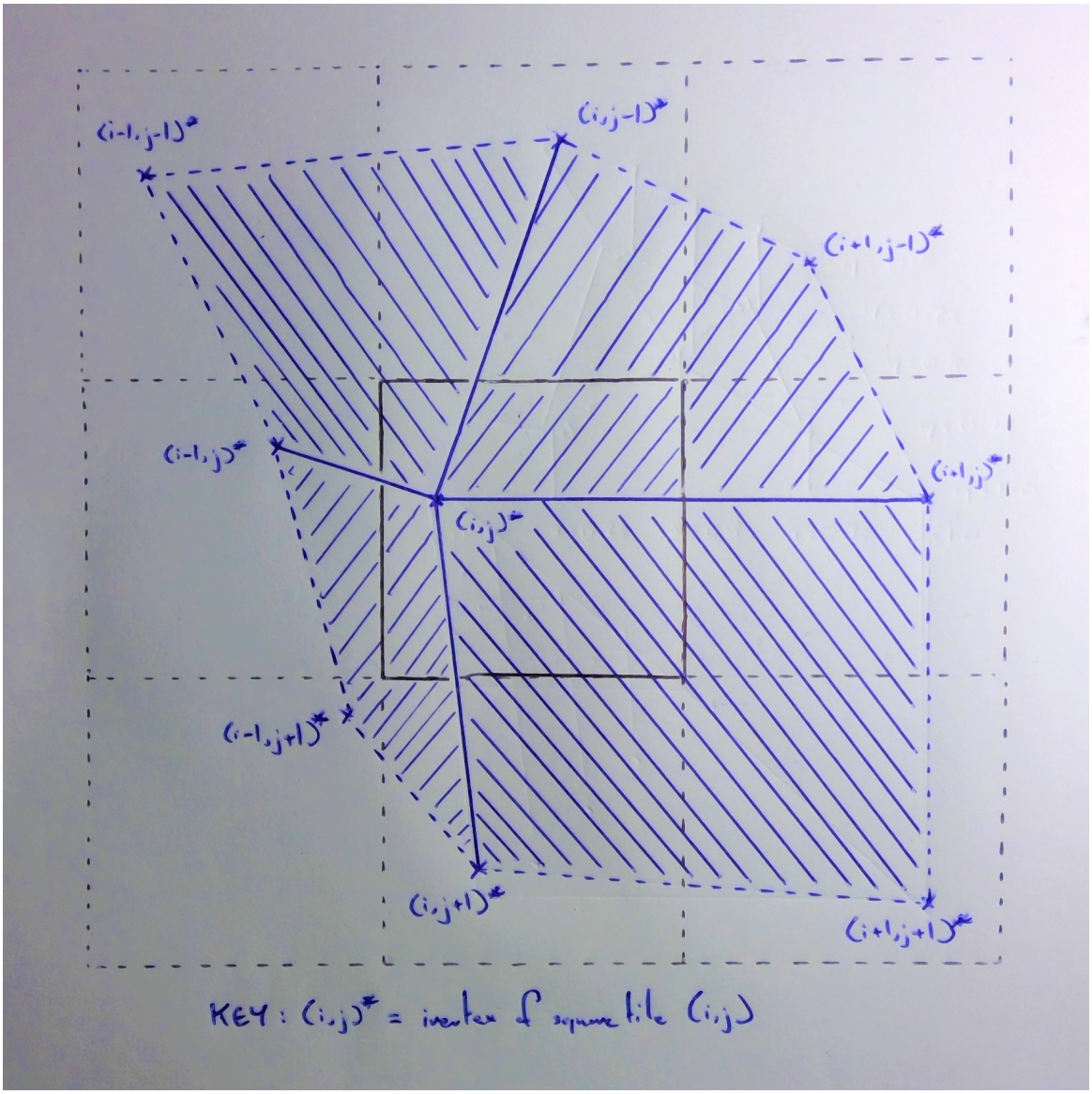

Figure 6: Integer coordinates for quadrilaterals: each point on the central square (i, j) lies on one of four quads, labelled $(i-1, j-1)$, $(i, j-1)$, (i, j), $(i-1, j)$ (clockwise, from upper left).

Notice how in this context, `ivertices` represent the corners of irregular quadrilateral tiles, rather than the centres of a cell. A pixel with integer coordinates $(i, j-1)$, say, will therefore lie on the quadrilateral created by the `ivertices` for $(i, j-1)$, $(i+1, j-1)$, $(i+1, j)$, and (i, j). Defining these points as having fractional coordinates $(0.0, 0.0)$, $(1.0, 0.0)$, $(1.0, 1.0)$, $(0.0, 1.0)$, and their mean `ivertexMean` as $(0.5, 0.5)$, the piecewise coordinate system in Figure 7 emerges.

The above explanation is by no means mathematically rigourous. Referring back to the start of 3.2.2, though, this vague notion of a 'distorted yet well-defined' coordinate system should now be far more convincing.

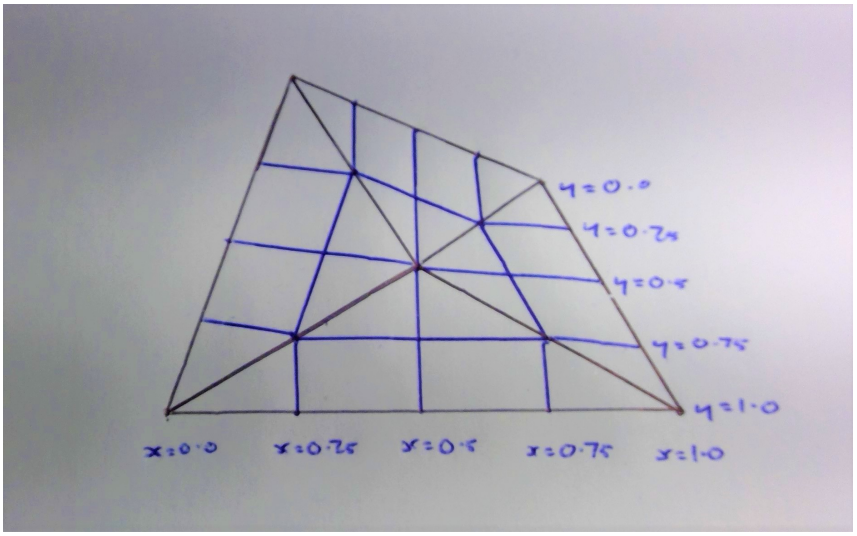

Figure 7: Fractional coordinates for quadrilateral $(i, j-1)$ (from Figure 6): splitting apart along the lines from the `ivertexMean` to each `ivertex`, a separate fractional coordinate system is constructed on each triangle; sewing back together gives a continuous, but not necessarily smooth, grid.

3.2.3 Case Study: Hexagonal Distortion

Likewise, a well-defined coordinate system can be constructed for a grid of distorted *hexes*. The shader `tiling_irregular_hexes.hlsl` is at clear generalisation of `tiling_irregular_quad.hlsl`,[2] so much so that explaining any nuances between the two would require retreading most of 3.2.2 in the process; for the sake of brevity, this report asks the reader to take it at face value that such a generalisation exists.

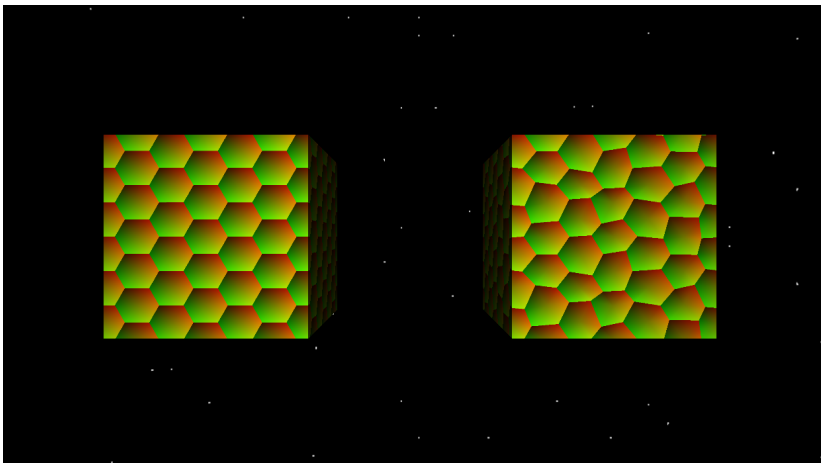

Figure 8: Hexagonal grid (left); distorted hexagonal grid (right).

At any rate, these coordinate systems only exist as means to an end - the programmer must now write visually interesting shaders on top of them.

[2]The curious reader is encouraged to compare the two shaders line-by-line, to satisfy themself that the main distinction between the two is that the hexagonal case uses far more trigonometry.

3.2.4 Case Study: Dynamic Pores

This application uses dynamic textures to create its central porous effect. The construction is simple enough: the shader `pores.hlsl` uses a series of circles with centres $(0.5, 0.5)$ and radii of the form

$$r_i(t) = R_i + R_i' \sin(t/T_i)$$

to create a set of nested holes, expanding and contracting over different periods T_i. These are tiled onto a distorted hexagonal grid. Instead of idealised mathematical objects the pores look irregular, no longer a series of perfect Euclidean circles, but a warped, cellular pattern that could have far more convincingly emerged from nature. The opinion of this report is that it is that wholly natural quality in which horror of trypophobia lies.

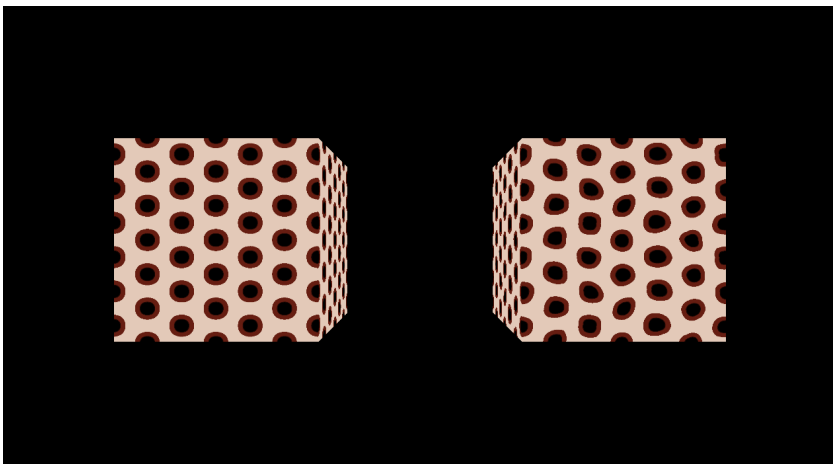

Figure 9: Pores on a regular hexagonal grid (left); irregular hexagonal grid (right).

To tie back to 3.1 for a moment, now imagine points at radius r on the surface around the ith hole have a height

$$h_i(r) = H_i \left(1 - \frac{r - r_i(t)}{r_{i-1}(t) - r_i(t)}\right)^3.$$

Taking the derivative with respect to r,

$$h_i'(r) = -\frac{3H_i}{r_{i-1}(t) - r_i(t)} \left(1 - \frac{r - r_i(t)}{r_{i-1}(t) - r_i(t)}\right)^2,$$

which by radial symmetry and angle of elevation

$$\phi = \arctan\left(-\frac{1}{h_i'(r)}\right)$$

produces a surface normal. With a few lines of calculations, *Trypophobia*'s central dynamic texture now has its own dynamic normal map.

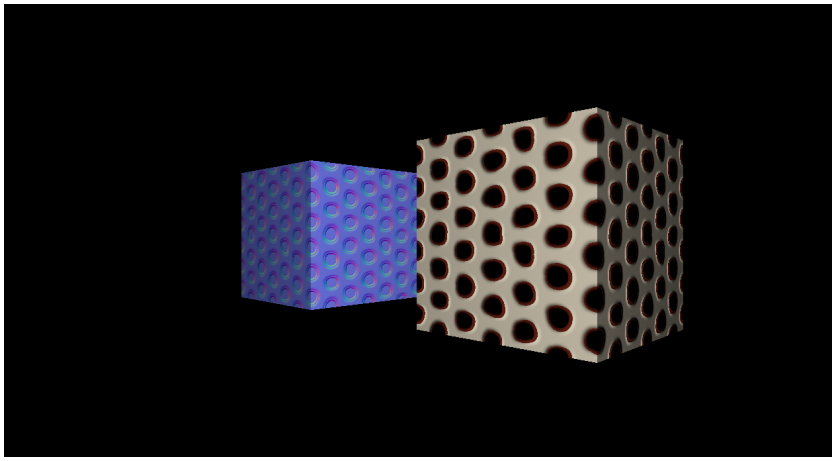

Figure 10: `pores_nm.hlsl` (left); lighting effect created by `pores_nm.hlsl` (right)

3.3 Skyboxes

Many techniques are less involved. While it is not worth litigating every minor quality-of-life improvement made to the original framework, this report will briefly touch on the skybox: a cube rendered with back-face culling turned on (i.e. 'inside out'), at the player's current position (i.e. always enclosing them), and with its depth set to none (i.e. infinitely far away, behind all other objects).

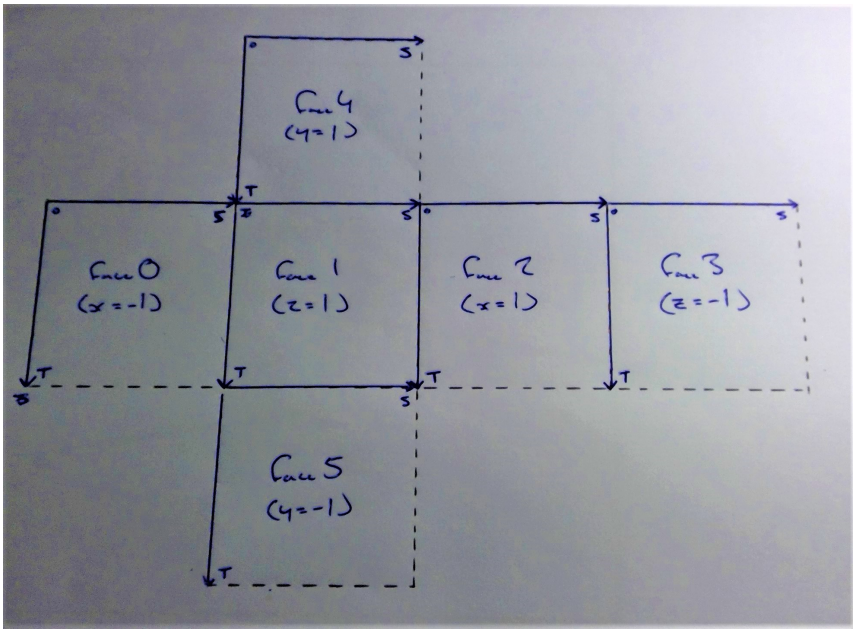

Figure 11: Coordinate system for environment mapping; faces are labelled by their index (and the equation of the plane they represent), and axes represent the orientation of each face's texture coordinates.

Skyboxes are textured by environment maps. It is easy to define a function `find_environment_st()` that takes in a pixel's 3D position relative to a camera - in this case, the player's camera - identifies the face it maps to by the position and sign of its largest coordinate, then calculates st coordinates

on a case-by-case basis according to the above. This sampling technique is as explained in the lecture material.

When compared to the more advanced technique of the skydome, one common criticism of skyboxes is how the perspective seems off around the edges. As seen in Figure 12, `skybox_ps.hlsl` tries to smooth over these, but the results are limited at best.

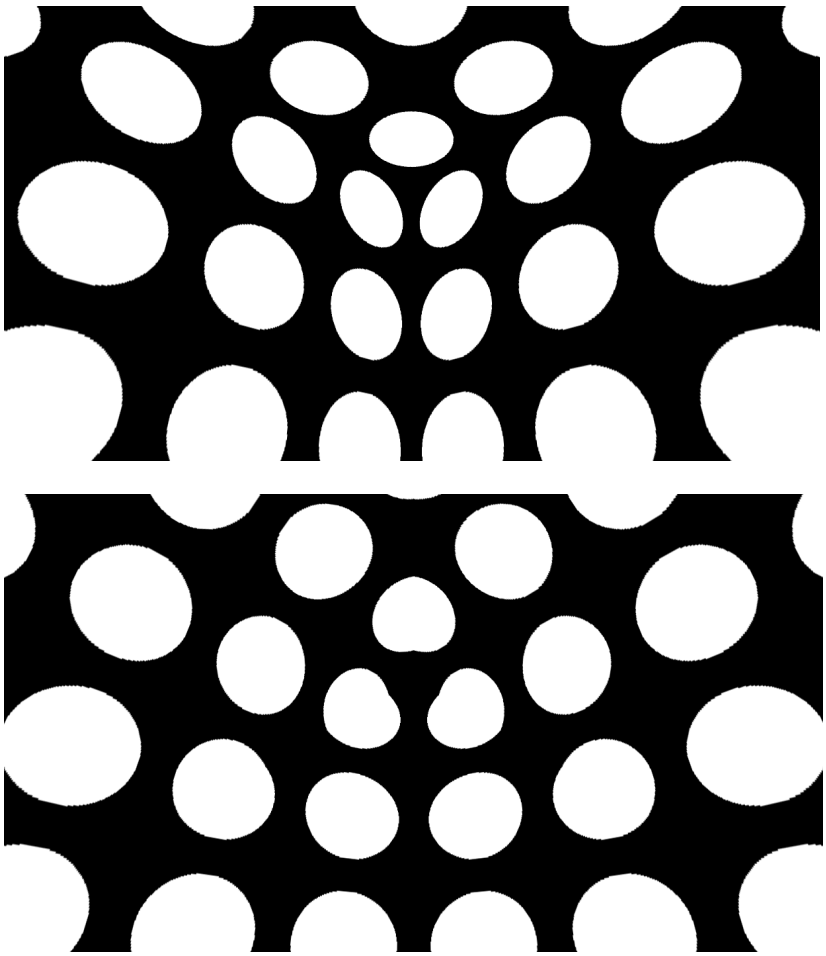

Figure 12: Skybox (above); skybox with edges 'smoothed' (below).

As a direct consequence of this inherent limitation, the actual skybox used in *Trypophobia* is deliberately simple; it exists only to orient the player. Far more important for the purposes of this report are the concepts introduced above - environment mapping, back-face culling - as these are essential in creating...

3.4 Glass Containers

The original idea for this application was to present a twitchy, porous hand in a glass specimen jar. While this ultimately proved too ambitious (see 'Evaluation'), the goal of *Trypophobia* has been to achieve the imagined visual effects independent of the imagined models. With that context in mind: what effects are necessary to create a glass container?

3.4.1 Reflection

Reflection is a textbook use of environment mapping. By setting up six cameras pointed in the six cardinal directions (which the report will refer to as a single 'environment camera') at the centre, *Trypophobia* can render an environment of a reflective object's surroundings. Taking any point X on the model, finding relative position $\mathbf{v} = \texttt{input.position3D} - \texttt{cameraPosition}$ and surface normal $\mathbf{n} = \texttt{input.normal}$, the function

$$f_{\mathbf{n}}(\mathbf{v}) = 2(\mathbf{n} \cdot \mathbf{v}) - \mathbf{v}$$

returns a reflection vector - which, using `find_environment_st()` from 3.3, allows the application to sample the reflection on the environment map and display it back on the model.

This is a cheap trick, and it certainly has its limitations. The player is not seeing reflection off X, rather reflection off centre C using the normal of X; this becomes all-too-apparent from certain angles, as illustrated in Figure 13. The real interest here lies in the similarity to skybox mapping; outside of what the environment map looks like and where it is being sampled to, the only substantive difference is that relative position has been run through some function f before being entered into `find_environment_st()`.

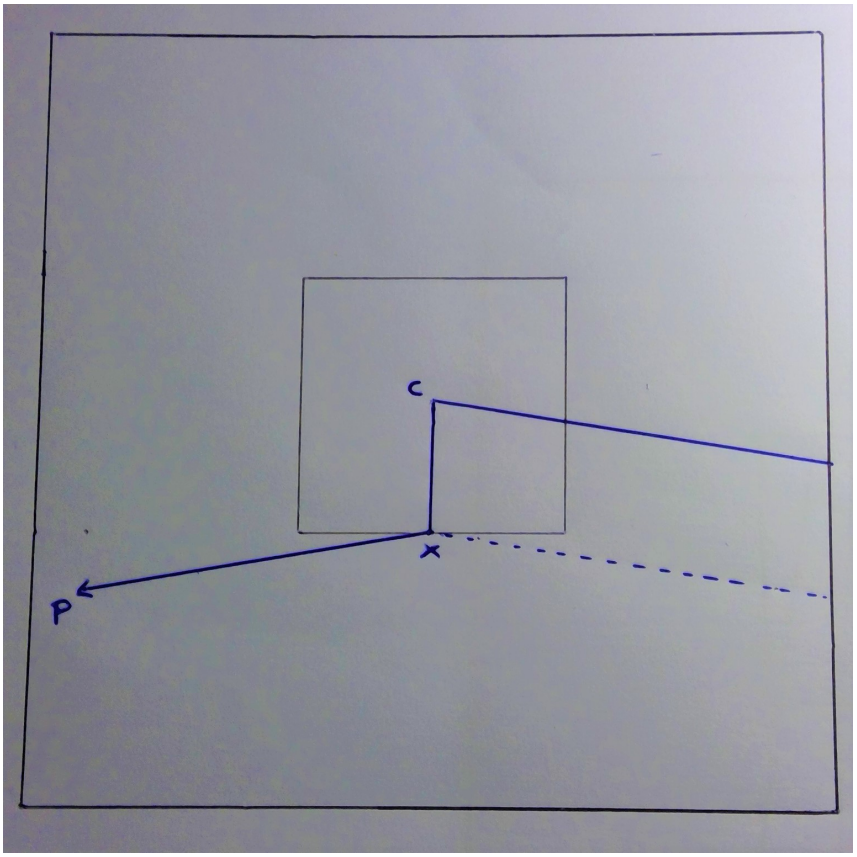

Figure 13: Limitations of 'centroid reflection': solid blue line represents the path of light from the environment map to the player; dashed blue line corresponds to the physically-accurate path.

3.4.2 Refraction

Suppose, then, that f represented refraction, rather than reflection.[3] To refract through a glass container, the environment camera is now set to the *player's* position, and renders all surroundings *except* the glass onto an environment map. Then, the process is much as before: when the glass is rendered, point X's relative position is taken, refracted by f, mapped by `find_st_environment()`, with the resulting sample from the environment map used to color X in.

The above explanation is intentionally concise, glossing over a key point: *this isn't how refraction works*. The technique described does recreate the phenomenon of light changing speed as it passes between mediums, but it assumes only *one* change of medium. While this would be perfectly appropriate for looking into, say, a deep lake (light from the bed transitions water-to-air), it won't be sufficient for *Tryphobia*'s purposes (light through glass transitions air-to-glass-to-air, not just glass-to-air; see Figure 14).

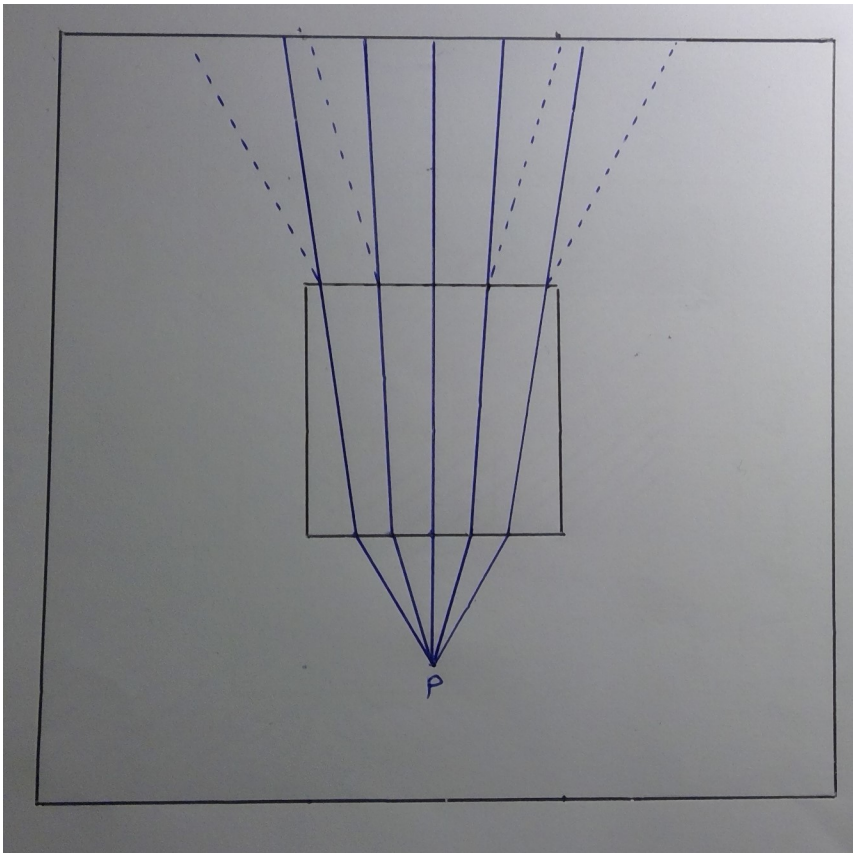

Figure 14: Limitations of 'single refraction': solid blue lines represent the paths of light from the environment map to the player; dashed blue lines correspond to the physically-accurate paths.

The solution here is surprisingly elegant. On a first pass, the environment camera again generates a map of the player's surroundings without the glass object, and again renders this onto the glass object. This time, however, the glass is rendered using back-face culling, to create the effect of an inward air-to-glass transition. The environment camera then renders a second environment map *that*

[3] HLSL - mercifully - comes with a built-in `refract()` function, though for the purists out there, there are tutorials that work through the same calculations from first principles (Scratchapixel n.d.).

includes the back-face-culled glass, and uses this second map to refract glass-to-air as above. Note that the refractive indices in the two calculations will be different: since the refractive index of the glass with respect to air is ≈ 1.50, the second refraction pass uses 1.00/1.50, the index of air with respect to glass.

3.4.3 Specimens

To turn these glass models into specimen jars, *Trypophobia* further makes use of alpha maps: grayscale textures that encode a pixel's alpha value. If some `texture` has some `overlayTexture` placed in front of it, an `overlayAlphaMap` can be sampled to linearly interpolate the two:

```
float overlayAlpha = overlayAlphaMap.Sample(SampleType, pixel);
float4 color = (1.0-overlayAlpha)*textureColor+overlayAlpha*overlayColor;
```

In render to texture passes, the application creates images (and alpha maps) of opaque specimens floating in translucent liquid. These are then overlaid between refraction passes - the specimens will not only mask the air-to-glass effects, but undergo refraction back from glass to air.

Since all glass containers are rendered simultaneously, they will not be be visible in one anothers' reflection and refraction passes. Note that alpha maps allow *Trypophobia* to render 'pseudoglass' in their place: a cheap version of glass that does not reflect or refract, but at least appears translucent.

To summarise, then, the steps by which glass container i is rendered are as follows:

1. *At runtime, render specimens to texture from i's perspective.*
2. *At runtime, render i's surroundings to texture, rendering containers $j \neq i$ as pseudoglass.* This gives the environment map later used for reflection.
3. *Each frame, render specimens to texture from the player's perspective*
4. *Each frame, render the player's surroundings to texture, rendering containers $j \neq i$ as pseudoglass, and not rendering i at all.* This gives the environment map used for the air-to-glass render pass.
5. *Each frame, render the player's surroundings to texture, adding a back-face-culled i showing air-to-glass refraction*
6. *Each frame, render i's specimens over the player's surroundings.* This gives the environment map used for the glass-to-air render pass.
7. *Finally, render the player's surroundings, adding a front-face-culled i showing both reflection and glass-to-air refraction.* The two visual effects are mixed according to Fresnel's equations (Scratchapixel n.d.).

This is a fairly dry set of instructions, included mainly as context for `Game.cpp`. For the purposes of what follows, it will suffice to say that *Trypophobia* has successfully combined reflection, refraction and alpha mapping into a nuanced visual effect - at some computational cost.

4 Evaluation

On reflection, *Trypophobia* has turned out very well. As with any project, there's plenty of features that could have been implemented with a little more time - shadow mapping, skeletal animation and 3D sound all fell by the wayside at one point or another - but frankly, Section 3 is long enough as is.

The Book of Shaders has been a major source of inspiration since the very start of this assignment, hence the determination to design compelling, dynamic shader textures. While a deeply unpleasant exercise in trigonometry, possibly the most gratifying part of the whole experience has been creating

the coordinate system for distorted hexes from scratch, in no small part because it's the one part of *Trypophobia* that could genuinely save someone else a lot of time and energy.

Despite this solid foundation, `pores.hlsl` is somewhat lacking. The 'what went wrong' here is more of a cause for self-reflection than anything else. Having reached the milestone of a well-defined coordinate system, I got ahead of myself and started trying to implement new techniques without an understanding of the underlying mathematics.[4] Returning to this project after the deadline, I'm confident I'd have the breathing space to build up the texture in a more appropriate, incremental fashion.

The glass containers look far more impressive. Having done a lot of reading into how to achieve something more advanced than single refraction, I'm quite surprised I didn't come across this back-face culling method anywhere. In hindsight it feels like such an obvious trick, but it was only pointed out to me in one of the later CMP502 labs - not only has double refraction been successfully integrated into *Trypophobia*, but in doing so added a level of realism that couldn't be found in any online tutorial.

In the sprit of rigourous self-criticism, though, I won't pretend the containers are perfectly accurate: outside of the issues with centroid reflection (recall Figure 13), the specimens being suspended in brine means we really should refract air-to-glass-to-water-to-glass-to-air! The effect is already too expensive, however. While the glass containers are great in theory, the sheer amount of rendering to texture involved in creating even one comes with a large computational cost. Working backwards from here, maximising efficiency while minimising the amount of detail lost, would be an appropriate course of action.[5]

There's also a fun bug to be fixed: notice how *all* objects in the player's surroundings are rendered into the environments used for refraction. This leads to the unexpected consequence of objects in front of a glass container suddenly being refracted through the glass as if they were in the background. While we can obscure this by repositioning models and lowering the lighting, the 'true' fix is, of course, incorporating a depth buffer - this would allow us to render only those objects behind the glass at this stage. Rastertek's tutorial on the technique (n.d.*b*) also leads me to believe I could decrease the cost of my glass shaders, by turning certain renders to texture on/off based on distance, but I'm not such how much of an improvement that will make.

Ultimately, this is just a vertical slice of the application's technical issues, and there will always be more optimisations and oversights to discuss. Ending on an artistic note, however, the biggest improvement to be made perhaps lies in the scene itself. While I was able to integrate normal mapping into the framework, I couldn't easily source maps for the models I wanted to use, leaving them looking low in detail and generally unappealling. To that extent I stand by the aesthetic decision to only use clean, geometric objects, but the scene never quite coheres into something more than the sum of its parts. *Trypophobia* is best considered simply as a showcase; a curious collection of graphical specimens, as it were.

References

Blinn, J. (1978), 'Simulation of Wrinkled Surfaces', *ACM SIGGRAPH computer graphics* **12**(3), 286–292.

Lamb, E. (2016), 'A Few of My Favorite Spaces: The SNCF Metric', Available at:

[4] Eagle-eyed readers may have noticed a hastily-commented out attempt at using Perlin noise in one of the pixel shaders...

[5] I believe implementing ray tracing would also be an option, although this appears far more advanced.

https://blogs.scientificamerican.com/roots-of-unity/a-few-of-my-favorite-spaces-the-sncf-metric/. (Accessed: 30 December 2022).

Planetmath (n.d.), 'SNCF Metric', https://planetmath.org/sncfmetric. (Accessed: 30 December 2022).

Rastertek (n.d.*a*), 'Tutorial 20: Bump Mapping', Available at: https://www.rastertek.com/dx11tut20.html. (Accessed: 28 December 2022).

Rastertek (n.d.*b*), 'Tutorial 35: Depth Buffer', https://www.rastertek.com/dx11tut35.html. (Accessed: 8 January 2023).

Scratchapixel (n.d.), 'Introduction to Shading: Reflection, Refraction and Fresnel', https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel. (Accessed: 8 January 2023).

Vivo, P. G. & Lowe, J. (n.d.*a*), 'The Book of Shaders: More Noise', Available at: https://thebookofshaders.com/12/. (Accessed: 30 December 2022).

Vivo, P. G. & Lowe, J. (n.d.*b*), 'The Book of Shaders: Patterns ', Available at: https://thebookofshaders.com/09/. (Accessed: 30 December 2022).

Vivo, P. G. & Lowe, J. (n.d.*c*), 'The Book of Shaders: Shaping Functions', Available at: https://thebookofshaders.com/05/. (Accessed: 28 December 2022).