Procedural Terrain Generation for Virtual Reality Environments

---

A Thesis
Presented to
The Division of Mathematics and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Josh E. Reiss

May 2020

Approved for the Division
(Computer Science)

Eric S Roberts

Acknowledgements

Thank you to Maddy, for helping me keep it together. I never would have graduated without you around.

Thank you to my friends, for always being there for me. I would be a far worse person without all of you.

Thank you to my family, for supporting me throughout my life. I am privileged and grateful to have grown up in your home.

List of Abbreviations

3DOF	Three Degrees of Freedom
6DOF	Six Degrees of Freedom
AR	Augmented Reality
API	Application Programming Interface
BAFTA	British Academy Film Awards
CPU	Central Processing Unit
CTO	Chief Technical Officer
DOS	Disk Operating System
GPU	Graphics Processing Unit
HMD	Head-Mounted Display
IR	Infrared
USB	Universal Serial Bus
VR	Virtual Reality

Contents

Abstract

Exploring novel worlds in virtual reality is endlessly compelling. To that end, this thesis explored methodology for generating infinite novel landscapes for VR exploration. Terrain generation was achieved using a parallelized implementation of the Marching Cubes algorithm in Unity using compute shaders written in HLSL. These terrains were then made navigable via hand-oriented smooth locomotion through the SteamVR API. The chapters herein are designed to work as a guide for others to build new procedural terrain techniques, for VR and otherwise, using similar methodology.

Introduction

This thesis explores the development of a VR application in which users can explore any infinite number of unique, procedurally generated environments.

Chapter 1 establishes my background, and why I wanted to build such an application.

Chapter 2 explores the VR aspect of this project, as well as a failed thesis idea which lead into this one.

Chapter 3 builds up from a basic understanding of procedural generation all the way to a full implementation of the marching cubes algorithm on an infinite space of xyz points.

Chapter 4 concludes with a reflection on this thesis, as well as a discussion of what could be done to expand upon it.

Links to all pertinent code are in the appendix.

Chapter 1

Motivation

1.1 2012

I still remember when the Oculus kickstarter page launched in 2012. At the time, I was just beginning to dip my toes into the world of programming by writing Half-Life 2 mods in my free time. Personal projects of that nature were the driving force behind my eventual decision to go into computer science — there's something incredible about hitting "Build Solution" and watching a folder full of text files come to life.

And yet, even at the age of 13, I couldn't shake the feeling that I had somehow missed the golden age of computing. When reading about the famous programmers of the 80s and 90s, it seemed to me that they had an entire world of opportunities open to them, and my generation had been left with the scraps. John Carmack built the early pillars of 3D graphics by writing the engines for DOOM and Quake as the sole engineer at a 6-man company. Steve Wozniak built the Apple I single-handedly in his free time. There are countless other indivudals from that era who shaped the future of my field, but what mark could *I* leave on it? I'm certainly not writing a 4D renderer, and my apartment can never hope to match an Intel research lab.

Enter Oculus, a small startup trying to build affordable, consumer-ready virtual reality headsets. I was immediately enamored with the idea that VR might be an opportunity for me to work on something truly important. I dreamt myself a future as a VR engineer fresh out of college, with the kind of näiveté only a 13-year-old can muster.

1.2 2016

The headsets themselves didn't hit the market for a long while, and when they did, they were entirely out of my reach. It was early 2016, and working retail jobs over the summers during high school wasn't enough for me to fund the purchase of a $600 toy.

In April that year, John Carmack (who had become the CTO of Oculus in the interim) received a British Academy of Film and Teleivison Arts (BAFTA) Fellowship

award, and his acceptance speech resonated with me deeply. He spoke of his own teenage years, when he "thought [he] had missed the Golden Age of 8-bit Apple-II [game programming]", just as I now spent my time yearning for the "Golden Age" of 90s DOS software. The knowledge that my generation was not the first to feel helplessly trapped in the shadow of past programmers was far more reassuring than anything I could have asked for.

1.3 2017-2018

By my sophomore year at Reed, consumer-ready virtual reality software was still taking its baby steps. But, by this point, it was clear that I wasn't alone in my belief that VR was going to become an important subset of computer science in the coming years. Facebook had acquired Oculus for 3 billion dollars, and had begun to slash hardware prices in the hopes of dominating the market. It was an effective strategy — the lowered prices were in a range I could actually afford.

Trying to use a VR setup in my tiny Foster dorm room resulted in a lot of missing paint on my hand controllers and more than a few broken nails from running my fists directly into the wall, but I couldn't have cared less. It took 8 years of waiting, but my expectations were thoroughly exceeded.

Of course, it was clear that we were still in the days of early adopter technology. In terms of the hardware, text was so blurry it was almost illegible, I had three 20-foot USB extensions duct-taped to my walls, and I was using command strips with random bits from Ace Hardware to mount cameras to my ceiling (sorry, Residence Life). In terms of software, most experiences were extremely short, and clearly developed on shoestring budgets by small teams. On the development front, actually learning how to build an application felt impossible. Documentation for tools was frequently out-of-date, self-contradictory, or entirely nonexistent.

The rough edges only made me love it more. Every broken piece of software was confirmation to me that this really *was* something young and unrefined for me to contribute to. Better yet, I knew exactly what I wanted to make.

When first booting up my headset after initial setup, I was dropped into a room overlooking a hilly landscape dotted with trees. I immediately wanted to leave the room to explore the world outside, and was deeply disappointed when I discovered I couldn't do so. I decided, then, that if they wouldn't let me walk out of the room, I'd just write my own application instead.

Despite my excitement, the obtuse nature of VR development combined with my courseload made diving into such a project feel like a very bad idea. I shelved the concept and told myself I'd come back to it when I had more free time.

1.4 2019-2020

In the two short years since I first made the plunge, VR has come a long way. Current headsets don't requires cables to be run all over the room, the sorts of high-budget software experiences seen on traditional platforms are finally making their way over to

VR, and independent development tools have become far more friendly to newcomers. What perfect timing, then, for my senior thesis. In fall of 2019, I was certain there couldn't be a better year for me to graduate than 2020.

My VR environment idea felt too small for a thesis, though. Instead, I chose to build a toolkit for porting board games into VR, since that seemed to me like an endeavor that better fit the scale of a Reed College Senior Thesis. I figured I'd come back to the environment concept at some point in the distant future.

Pictured above: The VR landscape that sparked my imagination.

Chapter 2

Virtual Reality in General

2.1 What is VR, anyway?

I'd like to take some time to explain how VR actually works, but before I do so, let's define a few terms.

Definition 2.1.1. Virtual Reality
The usage of computers to experience and influence an artificial environment through sensory stimuli.

Definition 2.1.2. Head-Mounted Display
HMD; the industry-standard term for a virtual reality headset.

Definition 2.1.3. 6DOF
"Six Degrees of Freedom"; used to refer to virtual reality technology that can be tracked both in terms of rotation and position in three-dimensional space.

Definition 2.1.4. 3DOF
"Three Degrees of Freedom"; used to refer to virtual reality technology that can be tracked in terms of rotation but *not* in terms of position in three-dimensional space.

Definition 2.1.5. Framerate
The number of images displayed per second; also referred to as "Frames per Second", or FPS.

With that out of the way, I can explain more specifically why Oculus and its competitors' HMDs were so exciting. Contrary to what one might think, VR isn't all that young of a technology. The first HMDs sold on the consumer market were made available in 1985! However — those headsets sold for $49,000; nearly $180,000 in 2020 dollars. It's hardly surprising that VR hasn't really had a place in consumer homes until more recently.

By 2014, truly affordable headsets had begun to emerge. Products like Google Cardboard and Samsung Gear VR used a smartphone along with an inexpensive enclosure to jerry-rig a simple HMD. If you've walked into the technology section of

a big-box retailer in the few past years, you might have stumbled across extremely inexpensive HMDs from brands you've never heard of. These headsets are all fairly similar to Gear VR in terms of design and functionality. You might wonder, then, why I feel that 2016 was the real birth-year of VR.

The primary cause is the advent of reasonably-priced 6DOF tracking. Simply strapping a phone to your head can only allow for 3DOF tracking using the phone's gyroscope. Without 6DOF, you can't reach out to touch the world around you, or look under a table, or wave at a friend. When using a 3DOF HMD, it's hard not to feel like a passive observer viewing the world through fancy 3D glasses, but 6DOF offers a sense of true presence.

That sense of presence has endless implications, but among the more interesting to me is the effect it has on social interaction. While video chat applications like FaceTime are nice, they certainly don't make it feel like the people on the other end of the phone are there with you. I wouldn't claim that VR is just as good as real, in-person interaction (the inability to read facial expressions is a great loss), but I do find it far more compelling that video chat. You can't *feel* other people touching you, and you can only convey a certain degree of body language, but I find it far more personal than watching a loved one's face move on a screen.

This sense of shared space immediately got me thinking about tabletop games. There are already multiple ways to play board games with others over the internet on a flat screen, but doing so loses a lot of what makes board games special: spending time around a table with other people. It is for this reason that I initially sought to build a toolkit for making VR board games as my senior thesis.

2.2 Virtual Reality and Tabletop Games

I am by no means the first person to see the potential of VR tabletop experiences. One of the most popular applications for flat-screen online board gaming, Tabletop Simulator, has VR support. There are also applications that port specific tabletop games into VR, such as Catan VR. I find these solutions unsatisfying. Catan is just one game, and Tabletop Simulator is an unintuitive application that does not fully capitalize on the physicality of VR. I wanted to build a set of board game assets specifically designed for VR that could be used by other developers for their own projects.

The initial difficulty of learning how to develop in VR was quite fun to surmount. I used SteamVR's Unity plugin to get a basic scene cobbled together, and immediately started working on a deck of cards. The interactions I wanted to develop seemed simple enough — it's just a deck of cards, after all.

There are two problems to solve that make a realistic hand of cards extremely complicated; one in software, and one in hardware. On the software side, existing VR APIs like to assume you are either empty-handed or holding a single object. In fact, in SteamVR, when the user picks up an object, their hand "becomes" the object, and loses some of the default hand logic. This means that one would need to either rewrite portions of SteamVR or figure out some clever way to make all in-hand card

interactions stem from a single object.

On the hardware side, most VR controllers track only your index finger's position, whether or not your palm is open, and whether or not your thumb is raised. The only physical feedback VR software can give users is haptic vibration. Given this limited interactivity, it would be easy to draw a single card from a splayed hand of five cards, but how would you draw a particular card out of a hand of ten? Twenty? The fine movements required are extremely difficult when you can't physically feel the edges of the cards, or push groups of cards aside in your palm.

Consider, now, that any clever solution to the software problem must also bear this hardware issue in mind. It should be no surprise that some existing implementations of cards in VR replace the cards with large tiles, and do not allow the player to hold multiple tiles at once. Other implementations force the user to interact with decks of cards via a pop-up menu rather than physically grabbing them.

None of the solutions I could think of were satisfactory to me. I wanted a senior thesis I could be proud of, and a system that gave up on these problems by evading them could never be enough. Halfway through my allotted thesis time, I decided it was time to switch gears. I figured that, although I didn't have much time left to work, the seemingly small-scale nature of a procedural generation project made it a great idea to pursue. Throughout the rest of this paper, we will observe the degree to which I incorrectly evaluated its scale.

2.3 Virtual Reality and Environmental Navigation

We'll spend the next chapter talking about my implementation of procedural generation, but it's worth talking about the VR implementation as well. When designing any VR application with a large environment, one needs to plan out some sort of artificial locomotion. A given users' VR space can only be so large (most hardware distributors recommend a 2.5 meter square as the standard size), so it will be impossible for a user to physically walk across the space.

A common solution is hand-oriented teleportation, in which users point their hands where they want to go, press a button, and are instantly warped to that point in space. An alternative is stick-based "smooth locomotion", which is modeled after traditional video game movement with a joystick or a keyboard. Trying to program stick locomotion immediately presents an interesting question: which way is forward?

In flat-screen applications, pressing forward on the joystick will move the user in the direction the camera is facing. The VR equivalent would be considering "forward" to be wherever the HMD is facing towards. This is called "head-oriented smooth locomotion".

Head-oriented locomotion is easy to implement, but can be disorienting and inconvenient. If the user was pushing forward on their joystick, then suddenly looked to their left before looking forward again, their body would momentarily jolt sideways, since the meaning of "forward" was changed while they were looking away.

Unexpected movement in VR frequently results in nausea and/or migraines in most users. This discomfort is colloquially referred to as "VR sickness", though

the more formal term is "simulator sickness". While many users eventually develop "VR legs", just as a sailor gets their sea legs, it is nonetheless inadvisable to build applications which might make your users physically ill.

I consider "hand-oriented smooth locomotion" to be a more favorable alternative to head-oriented locomotion. The name says it all: instead of considering "forward" to be wherever the HMD is facing, we consider "forward" to be wherever the joystick hand is facing. When using hand-oriented locomotion, users can walk forward and look around simultaneously without altering their movement. While head-oriented locomotion has its place, seeing as this project is based around the observation of novel environments, I chose to opt for hand-oriented locomotion instead.

Now that we've agreed upon a locomotion method, we still need to figure out how to handle collision between the user and the terrain. Traditional interactive software considers the user to be a pill-shaped capsule, and slides them around along the environment like a cylinder on ice skates. VR collision is much more complicated.

It would be disorienting to push the user backwards when they try to put their head through a wall, so many VR applications opt to ignore HMD collision entirely. However, we still need some way to keep the player from falling through the floor or walking through walls. Herein lies the complication: we do not know for sure where the user's body is in space. Current consumer VR headsets consist of an HMD and two hand controllers, but there is no direct tracking of the feet or the torso. Therefore, if we want to use a traditional capsule collider, we need to develop some method of determining the player's body position mathematically.

It is tempting to just place a capsule collider directly below the user's head, but this can cause unexpected behavior when the user leans forward, since their head is no longer right above their torso. Furthermore, if the user decides to crouch or lie down, it is necessary to resize the capsule accordingly, lest they get pushed upwards by the assumption that the capsule is always a constant height.

We also need to note that the user can move both in the real world via physical movement as well as in the virtual world via stick movement. This means there needs to be some method of reconciling these two movement methods in order to accurately represent the user's position in space.

Now that we've established the complexities of locomotion, we ought to note that there is another trigger for VR sickness besides poorly implemented movement mechanics: unstable framerate. Any drops below the target framerate for a given HMD will result in an uncomfortable experience, even for those with strong VR legs such as myself.

Most VR headsets are targeting 90 frames per second rather than the software industry standards of 30 and 60 frames per second. This would make keeping a stable framerate hard enough on its own, but VR is also far more computationally intensive than traditional interactive software, since the GPU needs to render the scene once per eye, and the CPU is constantly occupied with tracking data. Given these two factors in tandem, VR software needs to be extremely well-optimized, lest it cause physical harm to its users.

By now, I hope it is clear why the "Virtual Reality" portion of this thesis' title boils down to "I let the user move around comfortably". VR programming makes

all traditional interactive software problems infinitely more difficult — but it also makes them infinitely more fun to work on. Tasks like player movement are no longer trivialities, but interesting problems with imperfect solutions. Even though I am quite fond of hand-oriented locomotion, it is by no means flawless. I look forward to seeing what new VR locomotion methods are proposed in the future.

Now that we've established a locomotion method with which to navigate environments, it's time to give our users an environment to navigate. In the next chapter, we will build up a method by which to offer infinite VR environments algorithmically.

Chapter 3

Procedural Terrain

Generating a landscape is going to require some kind of procedural generation algorithm. This chapter will get increasingly more and more technical as we build our way towards such an algorithm. We will begin with a more conceptual overview of procedural generation.

3.1 Procedural Generation vs. The World

Before describing specific techniques, it is necessary to first describe what *procedural generation* means.

Definition 3.1.1. Random Level Generation
Building an environment using prebuilt assets and some algorithm.

Definition 3.1.2. Procedural Level Generation
Building an environment strictly algorithmically.

Put simply: where random level generation uses prebuilt assets and some algorithm to cobble a world together, procedural level generation uses some algorithm to generate worlds entirely from scratch. Let's try thinking about this via an example.

Imagine we draw 64 rooms, each 8x8 in square units, on pieces of graph paper, and throw them into a hat. To randomly generate a level from these rooms, we'll shake the hat around, then use an algorithm such as the following:

```
fun GraphRooms(numRoomsDesired, ourRooms):
1. ourLevel = new Level();                          //define our level
2. ourHat = new Hat(ourRooms);                      //put our rooms in a hat
3. startRoom = ourHat.getRandomRoom();              //draw a room from the hat
4. ourLevel.place(startRoom, (0, 0));               //place a room at origin
5. while numRoomsDesired ≥ 0:                       //for num of rooms desired:
       wall = ourLevel.getLonelyWall();             //get a wall with no neighbor
       newRoom = ourHat.getRandomRoom();            //draw a room from the hat
       ourLevel.placeAdj(newRoom, wall);            //put newRoom next to wall
       ourLevel.createDoor(newRoom, wall);          //make a door in wall
       numRoomsDesired -= 1;
6. return ourLevel;
```

This would be *random* level generation, since all our algorithm did was staple together prebuilt content from the hat. A procedural generation algorithm, on the other hand, would generate our world completely from scratch.

For an extremely simple procedural algorithm, we could write something that looks like:

```
for all (x, y) in world:
    if (x < -10 || x > 10 || y < -10 || y > 10):
        (x, y) = Black;
    else
        bw = random(0 or 1);
        if bw == 0:
            (x, y) = Black;
        else
            (x, y) = White;
```

Or, if we wanted something less boring than a square full of points, we could always try the following:

fun `FunLevel(`*numRoomsDesired*`)`:
1. room[64] roomSet;
2. **for** *(int i = 0; i < 64; i++):*
   roomSet[i] = *emptySquare*(8); //creates an empty 8x8 square room
3. return *GraphRooms*(16, *roomSet*);

While it is true that this is making use of the same function as our random generation algorithm, we still consider this to be a procedural generation algorithm, since we created our rooms in code via some function *emptySquare*(), rather than building them by hand.

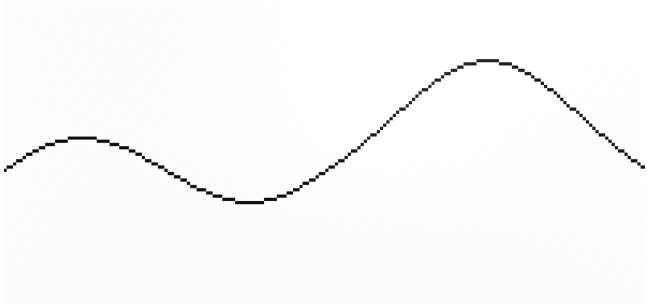
Now that we've made the distinction between these methods, it might initially seem like procedural generation is much easier. After all, in order to build our basic random generation algorithm, we had to draw 64 rooms ourselves! That takes a lot of creative energy.

However, true as that may be, it is much harder to make procedural generation result in something interesting. A grid full of points or a maze of empty rooms probably isn't a very fun environment to explore, especially when compared to a random set of hand-built rooms. So, what can we do?

3.2 What's all that noise!?

We'll use Perlin Noise! The Perlin Noise algorithm, written by Dr. Ken Perlin for use in VFX on the original TRON film, is incredibly useful. Conceptually, this algorithm is quite simple: Perlin Noise takes any pair of values (*x,y*) and gives back some $w \in$ {*-1,1*} that feels relatively random, but for any set of points $\{(x_1, y_1), (x_2, y_2), ...(x_n, y_n)\}$, the resulting points $\{w_1, w_2...w_n\}$ look relatively "smooth." More rigorously: the Perlin Noise function is continuous, and the gradient at some given $perlin(x, y)$ is quite similar to the gradient at $perlin(x + 1, y + 1)$.

To demonstrate what I'm talking about, let's try graphing some one-dimensional Perlin results.



We're just mapping $perlin(x) = y$ here, and we can very easily observe the "smoothness" of the noise. It looks like if we drew this out infinitely, we would get something that resembled a hillside.

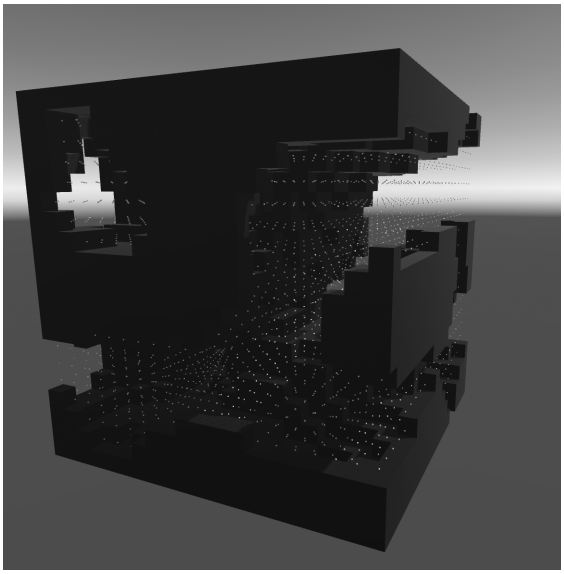
But what if we want something more complicated than hills? We could try mapping $perlin(x, y) = w$, and then set some threshold $t \in$ {-1,1} like so:

```
for all(x,y) ∈ world:
    w = perlin(x, y);
    if (w > t):
        (x, y) = Black;
    else
        (x, y) = White;
```

That looks more like an environment to me.

Building something similar in 3D isn't too hard. We can map $perlin(x, y, z) = w$, and then try the following:

```
for all(x,y,z) ∈ world:
    w = perlin(x, y, z);
    if (w > t):
        (x, y, z) = Black;
    else
        (x, y, z) = White;
```



Now we have something resembling a 3D voxel world! (Those small, floating white dots represent voxels for which $w < t$ for visualization purposes). By "voxel", I just mean a pixel with volume. Or, more formally:

Definition 3.2.1. Voxel
A single cubic unit with some volume that defines a point in 3D space.

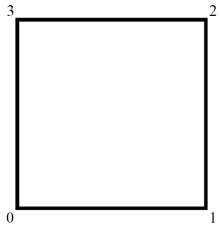
All we've done so far is place these 3D points. If we wanted to use this method to generate worlds that don't look blocky, we would need an absurdly large number of voxels. It would still look a little blocky if we looked real closely at it. So, what can we do?

3.3 Marching Squares

We're going to use an algorithm called Marching Cubes. First, though, we're going to ease our way into it by explaining Marching Squares.

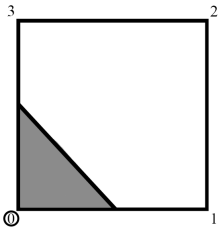
We imagine a square space of points on an *(x,y)* plane of size *(m,m)*. We call each of these points a pixel, with values *(x,y,w)*, where x and y are its Cartesian coordinates, and $w = perlin(x,y)$. Within our square space of points, we will consider every 2x2 square to be a "cell".

Let us consider a single cell. We will number each pixel 0-3, and number each edge 0-3.



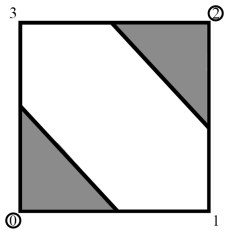
Given this cell, we want to generate some mesh of triangles. We will consider a given pixel to be a "1" if its w is greater than some threshold – we'll use 0.5. We will consider a given pixel to be a "0" otherwise.

Let's consider the simplest cell, where only one pixel is a "1". From now on, we will mark "1" pixels with a circle. If we wanted to generate a triangle, we might do something like this:

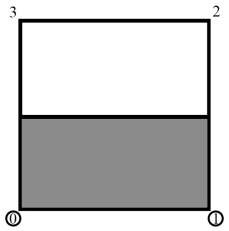


We merge the edges that are directly attached to the pixel at their midpoints, then shade the resulting triangle. It follows that, for any rotation of this cell, we would do the same.

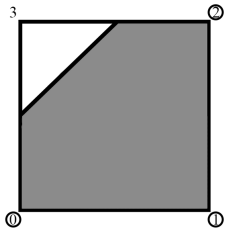
Let us now consider a cell where two opposite corners are "1"s.



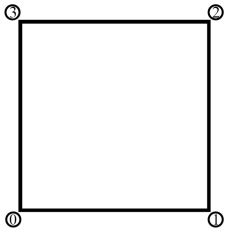
Once again, we just attach the edges adjacent to our pixels at their midpoints.
What, then, if we have two adjacent pixels set to "1"?



We join the two edges that are adjacent to exactly one "1" pixel.
What about three pixels set to "1"?

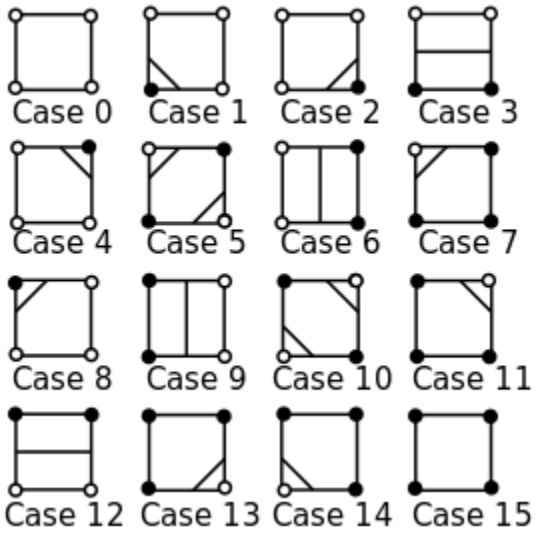


Once again, we join the edges that are adjacent to exactly one "1" pixel.
We are beginning to see a pattern emerge.
Finally, what if we set all four to "1"?



We try to follow our pattern of joining the two edges that are adjacent to exactly one "1" pixel, but no such edge exists! Therefore, we do nothing.

So, we've visually intuited what we think one ought to do with a given cell. Once we consider the various symmetries of the cases we've seen so far, we discover that the total number of possible cases is 16. These cases are shown below.



Now that we've established our cases and our methodology, we can try formalizing this as an algorithm. Let's make an attempt at doing just that.

fun `Join(`*edge1, edge2*`)`: //returns triangle with legs edge1 and edge2
1. chosenCorner = selectCorner(edge1,edge2); //selects corner for triangle
2. tri = new Triangle{$edge1, edge2, chosenCorner$};
3. return tri;

fun `Triangulate(`*cell*`)`:
1. tris = [ ];
2. **for** *edge* ∈ *cell.Edges:*
   edge.edgeValue = 0;
3. **for** *pixel* ∈ *cell.Pixels:*
   if *pixel.w >threshold:*
   neighbor1.edgeValue++;
   neighbor2.edgeValue++;
4. **while** ∃*(edge1,edge2)* | *(edge1.edgeValue = edge2.edgevalue = 1):*
   edge1.edgeValue = 0;
   edge2.edgeValue = 0;
   tris.append(Join(edge1,edge2));
5. return tris;

This strategy does work, conceptually. We use an edgeValue attribute to determine whether an edge is adjacent to exactly one "1" pixel, and if so, we join it to an appropriate neighboring edge.

How would we actually make this work in a computer, though? What is an "edge"? What is a "tri"? How do we perform a "Join"? How does "selectCorner" work? It's starting to seem like this kind of algorithm won't work as well as we'd hoped. So we'll do something clever.

We know that our cell has 4 pixels, and we know that there are 16 cases. We can easily map any given cell to its case by building a half-byte, or nibble, from its pixels.

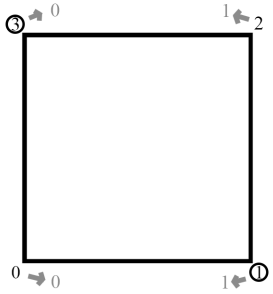
Definition 3.3.1. Byte
A group of 8 bits (binary digits).

Definition 3.3.2. Nibble
A group of 4 bits; half of a byte.

Earlier, we agreed to number our pixels 0-3. If all our pixels are "0"s, then we have the nibble 0000. If our 0th pixel is a "1", then we have the nibble 0001. If our 1st pixel is a "1", then we have the nibble 0010, and so on.

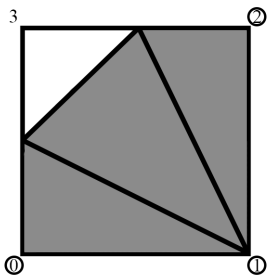
Since $2^4 = 16$, we know that a nibble can be used as a counter that increments from 0 to 16. Therefore, if we build a 16-item long table, where each entry contains instructions for the case that corresponds to that nibble, then we just need an algorithm that reads and executes those instructions!

Let's try building out our table. Consider the following cell:

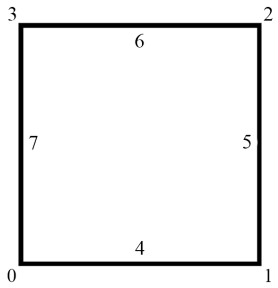


This cell's nibble is 0110, which is 6 in decimal. Therefore, we know that this ought to be resolved by case 6.

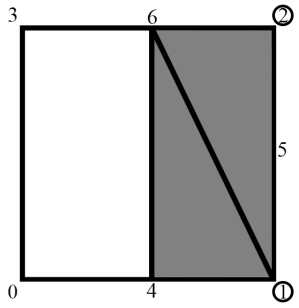
What, exactly, should our instructions be, then? Well, we know that we want our final mesh to be built from triangles, and we know that the most complicated mesh (the three-corner case) is composed of three triangles (visualized here):



So, we could make each entry in our table a 3-entry array, where each entry is a triangle (pixel/edge, pixel/edge, pixel/edge). If a triangle entry is all -1s, we'll ignore it. This method will require us to number our edges. We'll do this like so:



So, for case 6, we would have an entry that looks like $\{(1, 6, 4), (2, 1, 6), (-1, -1, -1)\}$. Let's draw those triangles!



That all works! If we check our cases diagram from earlier, we can see that this matches up with Case 6. We'll build out our whole table, and then we can define our algorithm rigorously:

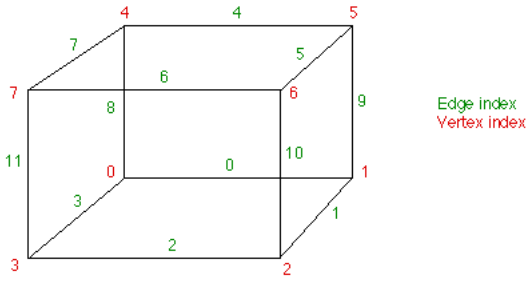
fun `Triangulate(`*cell, threshold*`)`:
1. cellType = 0;
2. pixelArray = (cell.corner0, cell.corner1, ..., cell.corner3);
3. **for** *i in pixelArray:*
   if *pixelArray[i].w >threshold:*
      cellType |= 2^i;
4. vertSet = triTable[cellType];
5. vertSet = vertSet[s] where s[0] != 0;
6. RenderTriangles.Append(vertSet);

fun `MarchingSquares(`*xyGrid, threshold*`)`:
1. **for** *square in xyGrid:*
   Triangulate(square, threshold);

Here, "RenderTriangles" refers to the set of triangles that will be rendered to our screen. For now, we can assume that our computer handles this for us. That's marching squares (at least conceptually)!

3.4 Marching Cubes

Now it's time to talk about the real deal: Marching Cubes, the 3D implementation of Marching Squares. We'll call our points in space "voxels" instead of "pixels", since they're 3D points now, and a "cell" is no longer a square of four points; rather, a "cell" refers to a cube of 8 points. Just like with our 2D "cell", we'll number the vertices and edges of our 3D "cell", like so:



With marching squares, we used each cell to form a nibble (0000 - 1111), and used this to map any given cell to one of 16 possible arrangements.

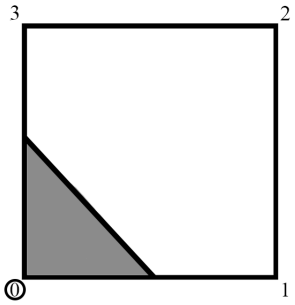
For marching cubes, we will use each cell to form a byte (00000000 - 11111111), since we now have 8 voxels instead of 4 pixels. This results in $2^8 = 256$ possible arrangements, where each arrangement is 5 (edge, edge, edge) arrangements.

Your first thought might be "oh, well, we won't have to actually hardcode all 256 arrangements, there must be a smarter way!" – and you might even be right about that. But if we want our algorithm to work quickly, and we do, then the most efficient way to get this all to come together is if we actually do go case-by-case and precompute our table.

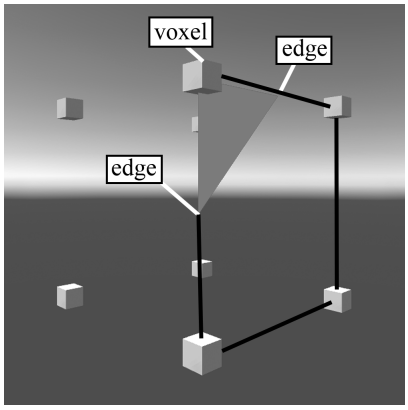
Fortunately for my sanity, and for this paper's brevity, this table is easily found on the internet written as a C array (Appendix A). Let's talk about a few cases anyway, since it will be helpful for understanding our algorithm.

You might have already noticed that for Marching Squares each entry was a (pixel/edge, pixel/edge, pixel/edge), whereas here, entries are written (edge, edge, edge) instead of (voxel/edge, voxel/edge, voxel/edge). This allows us to number the edges separately from the vertices, rather than numbering them together as we did with marching squares. Why is this?

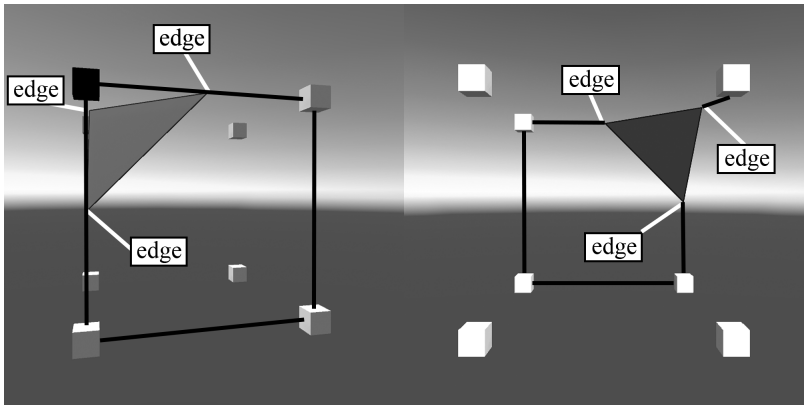
For the case where only one pixel was a "1" on a square cell, we did the following:



For the case where only one voxel is a "1" on a cube cell, we form a (voxel, edge, edge) triangle whose vertices are a voxel and the midpoints of two edges on the cell.



What we want instead is the following (edge, edge, edge) triangle, whose vertices are the midpoints of three edges on the cell.

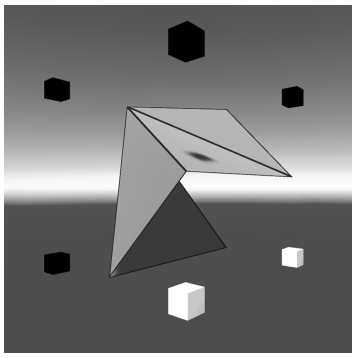


Note that, in all of the above diagrams, each voxel in our cell is represented by a cube. These cubes are not a part of the mesh, just as the numbers marking pixels in our square diagrams were not a part of the mesh. Similarly, the black lines on these diagrams represent a projection of a Marching Squares-style 2D cell, solely for the sake of visualization.

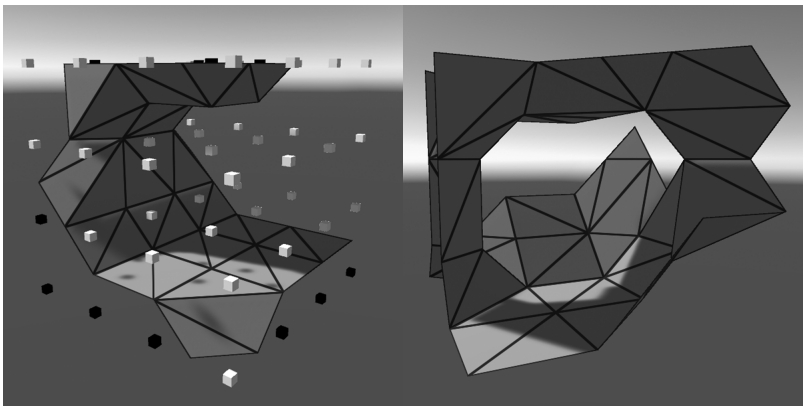
The difference between these triangles in 3D space makes sense once we recognize the pattern that's present here. With Marching Squares, for a given cell, we joined

the edges that were adjacent to strictly one "1" pixel such that all shapes formed by doing so contained a "1" pixel. Here, we're doing something similar – it's just that, to form a triangle by joining edges across a square, we need to use a pixel as the third vertex, whereas here, we form a triangle by joining three edges across our cube.

It's worth noting that this leads to some fairly strange shapes:



However, once we march across multiple cells and stitch together the resulting mesh, we get nice-looking results:



We could, if we wanted, rigorously define our marching cubes algorithm right here and now. However – in order to do so, we're going to want to step out of the abstracted world of pseudocode for a moment, and really think about how we would write this as real code. Doing that is going to require some groundwork first.

3.5 ...What about the real world?

I would completely understand if, by this point, you're starting to get a little restless. After all, we've done a whole lot of talking about algorithms, but we haven't addressed an important question: how would we implement this in a real application? What kind of assumptions that we've been making will get in our way? Will there be any serious problems we need to solve?

Before we get into talking about the implementation I've built, I will begin with a warning: we have made many assumptions that will get in our way, and some of them are directly tied to truly heinous problems.

We're going to use C# with the Unity game engine, and we'll talk our way from start to finish using real code, but before we do so, let's lay a little foundation.

Since marching cubes is splicing triangles together, we're going to build our environment as a solid mesh. When building a mesh in Unity from code, we will consider two attributes of the mesh: mesh.vertices and mesh.triangles.

mesh.vertices is a list of all vertices in the mesh, stored as Vector3s. mesh.triangles is a little less intuitive: it is a list of length $numTriangles*3$, where $numTriangles$ is the total number of triangles in the mesh. For some $i = 3n$ where $n < numTriangles$,

$mesh.triangles[i, i+1, i+2]$

represents a triangle composed of the vertices

$mesh.vertices[mesh.triangles[i], mesh.triangles[i+1], mesh.triangles[i+2]]$.

To make this clearer, let's consider an example mesh of two triangles.

```
//consider the following square:
//point2 - - - point3
//   ||            ||
//point0 - - - point1
mesh.vertices = new Vector3[ ](point0, point1, point2, point3);
mesh.triangles = [0,2,1,  3,1,2];
```

The resulting mesh will consist of the triangles (point0, point2, point1) and (point3, point1, point2). Note that the spacing in mesh.triangles does not do anything and has been added for readability — Unity reads this in as an arbitrary array of integers. We also must note that triangles are *one-sided*. If we wanted to view our triangles from any perspective, we would need to change mesh.triangles like so:

```
mesh.triangles = [0,2,1,  3,1,2,  0,1,2,  3,2,1];
```

This way, both of our triangles are drawn once clockwise and once counterclockwise, thus ensuring that they have two surface normals facing in opposite directions. Since we want our terrain to look like one solid mesh, we're going to be drawing every triangle in both directions like this.

You might have already noticed that this method of drawing a mesh is going to work quite nicely for our marching cubes algorithm. For a given cell, we used our lookup table to determine what triangles need to be joined, and stored the result as an array of cell edges. If we write some function that takes an array of numbered vertices in a cell, returns the corresponding entry in our lookup array, then adds the resulting edges to *mesh.triangles*, everything should work just fine.

Thus far, we've assumed that there's an *xyzGrid* composed of cubes (or, in the Marching Squares case, an *xyGrid* composed of squares) that magically know the *xyz* coordinates and w values for all 8 of their voxels. If you haven't tried programming in a game engine before, you might suggest we take the naive approach and actually generate every cube in our space, use 3D Perlin Noise to assign their w values,

then access a given cube's attributes from an array whenever we need to. In practice, generating that many cube primitives is an incredibly effective way to kill your framerate.

Instead, we could try building an array of Vector4s, where the *xyz* attributes are a set of coordinates, and the w attribute is *Perlin3D(xyz)*. This works fairly well! We would need to give our *Triangulate* method access to some function $FindCorners(xyz)$ since it will no longer be receiving "cells" as some structure, but will instead receive only a reference to one corner of the cell. It turns out that writing such a function is straightforward – we just use our knowledge of the xyzGrid's size to locate the neighboring voxels, then return them as an array.

Let's try writing out our Marching Cubes algorithm to reflect these changes.

```
fun Triangulate(xyzGrid, threshold):
1. allTris = new Triangle[ ];
2. for cubeCorner in xyzGrid:
       cellType = 0;
       voxels = FindCorners(cubeCorner);
       for i in voxels: //determine cell type
           if voxels[i].w >threshold:
               cellType |= 2^i;
       edges = LookupTable[cellType]; //get list of edges to join
       for (i = 0; edges[i] > 0; i+=3):
           tri = new Triangle; //Triangle is a struct of three xyz coords
           tri.vertexA = lookupVert[edges[i]]; //lookupVert gets xyz of edge
           tri.vertexB = lookupVert[edges[i+1]];
           tri.vertexC = lookupVert[edges[i+2]];
           allTris.append(tri);
           tri.vertexB = lookupVert[edges[i+2]];
           tri.vertexC = lookupVert[edges[i+1]];
           allTris.append(tri2); //triangle is rendered from both sides
3. return allTris;

fun Density(xSize, ySize, zSize):
1. Vector4[ ] xyzGrid = new Vector4[xSize * ySize^2 * zSize^3];
2. for k,j,i in zSize, ySize, xSize:
       for j in ySize:
           for i in xSize:
               xyzGrid[i + j^2 + k^3] = {i, j, k, Perlin3D(i,j,k)};
3. return xyzGrid;
fun MarchingCubes(xSize, ySize, zSize, threshold):
1. xyzGrid = Density(xSize, ySize, zSize);
2. Triangle[ ] tris = Triangulate(xyzGrid, threshold);
3. int numTris = tris.Length()/3;
4. (mesh.triangles, mesh.vertices) = numTris.extractData;
```

A few things here have been simplified for the sake of brevity, but some chunks of this code have been copy-pasted directly from one of my early attempts at a real-world implementation. If you have moderate experience with writing C# in Unity, the block above should be enough of a guide to get Marching Cubes working.

However: if you were to do so, you might be disappointed. If you take the time to actually read over that code, you will note that there are *many* nested loops. The runtime on our *Triangulate* and *Density* methods are very bad in practice. In fact, they're exponential – building a world of 2000 blocks takes *substantially* more than twice as much time than generating a world of 1000 blocks.

The solution to this problem is not immediately apparent, and it's the reason I used the word "heinous" a few pages ago. We're going to move both *Triangulate* and *Density* into the GPU by rewriting them as compute shaders. It's time to write parallelized code!

3.6 Compute Shaders

In our current solution, we're processing each cell one at a time. If the handling of any one cell was dependent on the handling of another, this would make sense. However, each cell is computed completely independently of all others. It is very easy to imagine, then, that if we could somehow work on multiple cells at once, we could run our algorithm much, much faster. Thankfully, there's an entire chunk of hardware in our modern machines specifically designed for these sorts of problems: the GPU (Graphics Processing Unit).

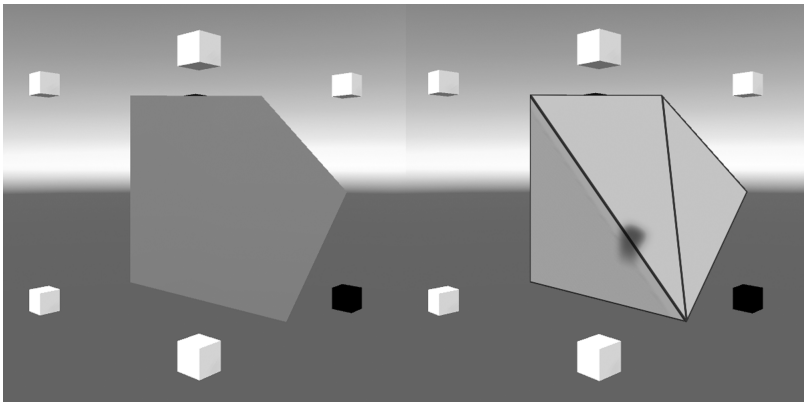
We don't need to get too far into the specifics of GPU hardware, but it's worth understanding why we prefer using a GPU over a CPU (Central Processing Unit) for our algorithm. Both CPUs and GPUs are partitioned into cores, and each core can handle one computation at a time. CPU cores have far more versatile instruction sets than GPUs, and tend to be orders of magnitude faster – for example, my GPU cores can handle 1733 million computations per second, whereas my CPU can handle 4.4 *billion* per second.

It's beginning to sound like CPU cores are outright better than GPU cores (and, to be fair, they are). However, there is one key difference between CPU and GPU architecture that leads us to prefer the latter here: GPUs have far more cores. My GPU has 2560 cores, but my CPU only has 6. That's ∼427 times as many cores!!

In other words, if each core is handling one cell at a time, my CPU would need to handle each cell 427 times faster than my GPU in order to outpace it. If we do some napkin math (4.4 billion / 1733 million), we see that my CPU is only ∼2.5 times faster per computation. Therefore, if we could distribute our algorithm across the GPU so that each core is handling a single cell, we would expect to see a gargantuan performance increase.

When getting into independent game development programming, it doesn't take long before one starts to stumble upon discussion threads in which people complain about the complexity of "shaders." These are, usually, folks who want to make their GPU apply some graphical effect for them. Maybe they want to apply custom lighting

to a model, maybe they want their world to look like a cartoon, or maybe they want to draw gridlines across some mesh. I use a custom surface shader to handle both lighting and gridlines, as shown below.



The term "shader" is a result of their original use: applying lighting and shadows to 3D models. It is unsurprising, then, that discussion threads about shaders are almost always referring either to surface shaders, unlit shaders, or image effect shaders, which all directly effect graphical rendering in a Unity application. However, Unity also has a fourth shader type: compute shaders.

The name is a little misleading here, in that compute shaders do not explicitly involve graphics at all. A compute shader takes some buffer (or buffers) and performs a series of computations on each element of the buffer (or some subset of the buffers). This sounds perfect for us! We can build a compute shader that runs our algorithm on a buffer of points, then returns a buffer of triangles.

3.7 The Density Shader

As with everything in this project, it turns out that conceptual simplicity does not imply ease of implementation. I might just have to get that printed on a T-shirt by the end of this. It'd be great to run our algorithm on a buffer of points, but where will that buffer of points come from? Our current implementation is three nested for() loops that go over our xyz space and store the corresponding $xyzw$ entries in an array.

This is very slow. In fact, these nested for() loops take just as long as the entire marching cubes algorithm does on the CPU, according to my benchmarks. Once we move the marching cubes algorithm into a compute shader, we will find the generation of this point buffer to be a restrictive bottleneck on our runtime.

What if we built this buffer using a compute shader, then? We can use this as an opportunity to practice writing compute shaders before we move on to the meat of the algorithm. We'll call this our "Density Shader," since the generation of our w component can be expressed by a density function.

```
1. static const int numThreads = 8;
2. RWStructuredBuffer<float4> voxels;
3. int voxRes;
4. float voxelSize;
5. fun indexFromCoord(int x, int y, int z):
       return (z * voxRes^2 + y * voxRes + x);
6. fun saturate(float x):
       return max(0, min(1,x));

//the following line defines the structure of our threads for Density()
 [numthreads(numThreads, numThreads, numThreads)]
7. fun Density(int3 id: SV_DispatchThreadID):
   1. if (id.x ≥ voxRes || id.y≥ voxRes || id.z≥ voxRes):
          return;
   2. float hard_floor_y = 10;
   3. double vSize = voxelSize / 10;
   4. double3 ws = (id) * vSize;
   5. double density = -ws.y;
   6. density += Perlin3D(ws.xyz).x;
   7. density += saturate((hard_floor_y)*3);
   8. int i = indexFromCoord(id.x, id.y, id.z);
   9. voxels[i].xyz = ws.xyz;
   10. voxels[i].w = ws.w;
```

This is a slightly simplified version of a prototype density shader I wrote for this project – and it works! The "hard_floor_y" and the saturate function force our point space to have a "bottom" so that we are generating actual terrain instead of arbitrary floating shapes (although that does look pretty cool).

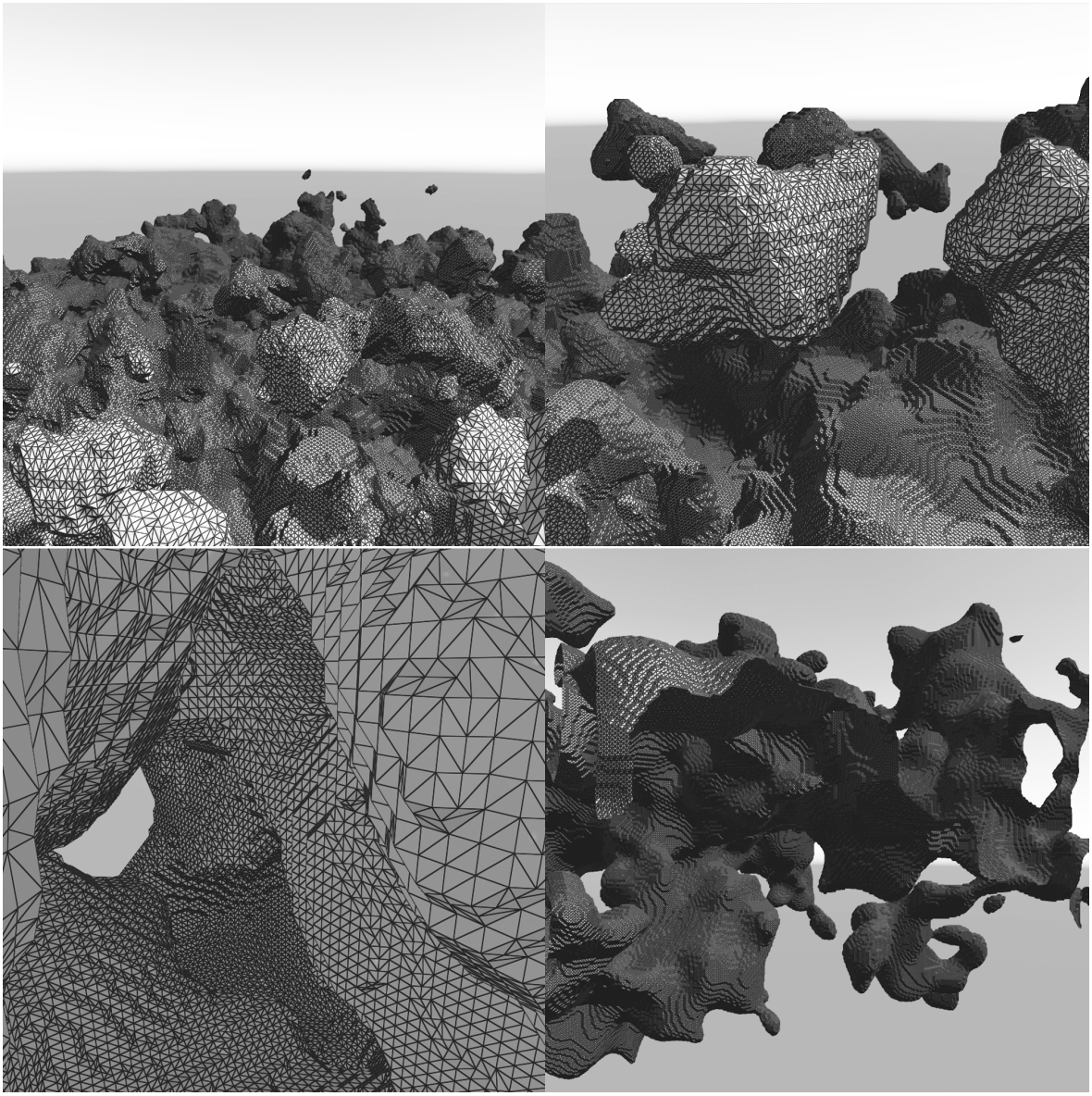
3.8 Marching Cubes as a Shader

Now that we have our points buffer, it's time to start marching our cubes, so let's write out our compute shader! This is the grand finale we've been building towards for 15 pages or so! Let's read one last giant block of code together!

```
1. static const numThreads = 8;
2. AppendStructuredBuffer<Triangle> tris;
3. RWStructuredBuffer<float4> voxels;
4. int res; //resolution
5. float isoLevel, vHSize;
6. fun vertMidpoint(float4 v1, float4 v2):
       return v1.xyz + 0.5 * (v2.xyz - v1.xyz);
7. fun edgeVertfromCornerVert(int cVert, bool firstCell):
       //returns the corresponding edge index given a vert;

//the following line defines the structure of our threads for March()
 [numthreads(numThreads, numThreads, numThreads)]
8. fun March(int3 id : SV_DispatchThreadID):
   1. if (id.x ≥ res − 1 || id.y ≥ res − 1 || id.z ≥ res − 1):
          return;
   2. float4 CubeVerts[8] = {select all 8 voxels for cell};
   3. int cubeIndex = 0;
   4. for (int currV = 0; currV 8; currV++):
          if CubeVerts[currV].w >isoLevel:
              cubeIndex |= 2^currV;
   5. for (int i = 0; triTable[cubeIndex][i] != -1; i += 3):
          1. float triArr[3];
          2. for (int currTri = 0; currTri ¡ 3; currTri ++):
                 id = triTable[cubeIndex][i+currTri];
                 v1 = edgeVertFromCornerVert(CubeVerts[id],true);
                 v2 = edgeVertFromCornerVert(CubeVerts[id],false);
                 triArr[currTri] = vertMidpoint(v1,v2);
          3. Triangle tri;
          4. tri.vertexA = triArr[0];
          5. tri.vertexB = triArr[1];
          6. tri.vertexC = triArr[2];
          7. tris.Append(tri);
          8. triVertC = triArr[0];
          9. triVertA = triArr[2];
          10. tris.Append(tri);
```

... That's it. I'm not going to make you look at code anymore, I promise. If you copy-pasted that code into a compute shader in Unity, wrote out "edgeVertfromCornerVert", and included a marching cubes lookup table (Appendix A), that would run perfectly fine. Let's marvel at our work.



That looks pretty great, doesn't it? We can achieve quite varied results by altering the computation of our "density" variable in the Density Shader from the last section. There are all sorts of interesting patterns that emerge from small changes.

3.9 Overlooked Simplifications

Before we close this chapter, I'll admit that we're skipping over three significant elements in my actual implementation.

Firstly, the implementation I've described works with one singular mesh. Therefore, if one wished to alter even one single vertex in the points buffer, one would then need to rerun our Marching Cubes shader from scratch to recalculate the mesh. The solution is simple: partition our mesh into many sub-meshes called "chunks", each with their own buffers, and run our shaders on them individually.

This leads us into our second simplification: we haven't really touched on driver code. I don't think it's interesting or complicated enough to discuss, frankly. It's just a few for loops and a bunch of simple math that accounts for the size and shape of our cells, though you can feel free to read it if you like (Appendix A).

Our final simplification is perhaps the most egregious: we've been treating 3D Perlin Noise as a magical black box, and we've never talked about the code implementation. Unfortunately, we're not going to fix that either. There are popular libraries for 3D noise all over the internet (I specifically used a GLSL shader found at github.com/keijiro/NoiseShader), and I feel that our conceptual understanding developed earlier in this chapter is more than enough to get by on.

With that out of the way, we've covered all our bases, and we can leave Marching Cubes behind us.

Chapter 4

I'm Not Leaving Marching Cubes Behind Me

I imagine most thesis students come to the end of their time at Reed and think the same thought I'm thinking now: "I wish I had more time to work on this". There are so many things I wanted to implement that just never got finished.

The entire genesis of this project came from my desire to walk through a procedurally generated forest in VR, but I haven't even gotten as far as modelling a single tree yet. My terrain is explorable in VR, but there's nothing to actually interact with. I can generate infinite chunks of terrain, but I have no way to despawn unnecessary ones, so the framerate drops further and further as you venture outward from the origin. I wanted my terrain to feel like a real place worth exploring, but right now it's only a red mesh with black lines along the edges.

If I were graduating in 2019, I might have followed that paragraph with "unfortunately, I don't know if I'll have time to do those things". However, as a 2020 graduate, I suddenly have a *lot* of free time on my hands. The COVID-19 pandemic has affected all of us differently. For me (and some of my friends), it's led to job offers being unceremoniously rescinded in the wake of our job market's complete collapse.

In my time at Reed College, I've certainly learned a great many things, but prior to starting my thesis I didn't feel like I'd made much of anything. I had a transcript that said I knew how to code, but nothing to really show for it. It didn't just feel difficult to convince employers I was worth hiring – it felt difficult to convince myself. That sounds a little dramatic, but I mean it genuinely!

Now that I'm coming to the end of my senior year, however, I've completely changed my tune. There were countless points in this project where I figured something out and thought to myself "I could never have solved that four years ago", even if it was completely unrelated to anything ever learned in the classroom. When my shaders finally gave back real results, it felt like a fitting payoff to an entire education's-worth of hard work. It was like summiting an enormous mountain of poorly documented code and dead links to tutorials that haven't been online since 2008. I suppose actually writing out this document has been the climb back down.

All that is to say: this is the first time I've had something that proves to me (and I think will prove to others) that I really do know a thing or two about computers. It

may not be perfect, and it may not be done, but it's mine, and I'm passionate about it. And now that I'm suddenly quarantined in my parents' garage, maybe I'll finish what I've started. Maybe I'll learn more about surface shaders. Maybe I'll pick up 3D modeling for the first time since high school. Maybe I'll make some trees. So, how about you swing by my desk a year from now? We'll explore a forest in VR together.

Appendix A

Data and Code Availability

Marching Cubes and Density Shaders:
https://github.com/reiss-josh/Learning_Cubes/blob/master/Assets/Shaders

World Generation Scripts:
https://github.com/reiss-josh/Learning_Cubes/blob/master/Assets/Scripts/Vox

VR Repository:
https://github.com/reiss-josh/VR_Procedural

Marching Cubes Tables:
http://paulbourke.net/geometry/polygonise/

Bibliography

Bourke, P. (1994). Polygonising a scalar field (Marching Cubes). http://paulbourke.net/geometry/polygonise/

Flick, J. (2017). Flat and Wireframe Shading. https://catlikecoding.com/unity/tutorials/advanced-rendering/flat-and-wireframe-shading/

Geiss, R. (2007). Chapter 1. Generating Complex Procedural Terrains Using the GPU. Library Catalog: developer.nvidia.com. https://developer.nvidia.com/gpugems/gpugems3/part-i-geometry/chapter-1-generating-complex-procedural-terrains-using-gpu

Halladay, K. (2014). Kyle Halladay - Getting Started With Compute Shaders In Unity. http://kylehalladay.com/blog/tutorial/2014/06/27/Compute-Shaders-Are-Nifty.html

IGI Consulting Incorporated (1992). *Emerging Markets for Virtual Reality*. Information Gatekeepers Inc. Google-Books-ID: QEK5Yy2BiY4C.

Lague, S. (2019). SebLague/Marching-Cubes. Original-date: 2019-05-05T21:44:26Z. https://github.com/SebLague/Marching-Cubes

UnityList (2017). Compute Shader. Library Catalog: unitylist.com. https://unitylist.com/p/yw/Compute-Shader

u/Zolden (2018). Drawing Mandelbrot fractal using GPU. Compute shader tutorial, easy level : Unity3D. https://www.reddit.com/r/Unity3D/comments/7pa6bq/drawing_mandelbrot_fractal_using_gpu_compute/

Valve Software (2018). SteamVR Unity Plugin | SteamVR Unity Plugin. https://valvesoftware.github.io/steamvr_unity_plugin/