

# Integration of Environment and Realtime Weather in an Executable Program

---

*Using Adaptive Shaders to Construct a Conduit of Nature*

BY

Zoey Schlemper

THESIS

Submitted in partial fulfillment of the requirements  
for the degree of Master of Fine Arts in Digital Arts  
awarded by DigiPen Institute of Technology  
Redmond, Washington  
United States of America

June  
2018

Thesis Advisor: David Longo

© 2018, DigiPen Institute of Technology. All Rights Reserved.

The material presented within this document does not necessarily reflect the opinion of the Committee, the Graduate Study Program, or DigiPen Institute of Technology.

## Abstract

Dynamic weather systems are utilized in a wide variety of videogames, and are made up of various techniques including particle systems and graphic shaders. The wide assortment of elements must interconnect and be driven by controllable variables in the game engine logic. They are traditionally controlled by code in the service of game logic. This Thesis presents a dynamic weather system that connects to the internet and retrieves weather data from stations across the United States in order to drive this suite of effects. It also features a graphic user interface for controlling the elements manually. This is both an artistic statement through surrender of technologic control to nature and a prototype for a system that could be implemented in a variety of products, including videogames, real estate, and virtual training applications.

## Acknowledgements

I owe the opportunity to complete this thesis to a few important people. First and foremost, I thank my parents, Carol Ann and Michael. They supported my creative endeavors since I was a child. We are all born with a love of art and creativity, but it was my parents who fostered that joy within me. They also encouraged me to pursue the insane idea of being an artist in the 21<sup>st</sup> century, and in fact supported me in that pursuit as well. Without their financial, emotional, and academic support, I would not be writing this.

Thank you, Mom and Dad!

I thank select Digipen Faculty (you know who you are) for the time and interest you have shown in my project. Digipen seems to demand more from their teachers (everyone, really) than other schools I have attended. I am thankful for the extra effort beyond your job description that has impacted me in positive ways. The late-night classes and meetings, the extra emails and opinions, were essential to finishing during my two short years here.

Lastly, I thank my wife, Marisa. Her support means the world to me. She has believed in me when I didn't believe in myself.



## Table of Contents

Abstract .....	iii
Acknowledgements .....	iv
List of Figures .....	vii
Introduction .....	1
Chapter 1: Artistic Direction and Development of Style .....	4
Concept Development and Aesthetic Style .....	12
Chapter Two: Techniques and Creative Process .....	29
3D Modeling Workflow .....	29
Texturing Workflow .....	34
Shader Development in Unreal Engine 4.16 .....	39
UX Design Process for Menu Systems .....	46
Chapter 2.5: Anatomy of the Weather System .....	47
Chapter Three: Coding Key Features of the Weather System .....	49
The XML parser: How Weather Data is collected from the Internet .....	49
One-Way Data Manipulation in Unreal Engine 4 .....	52
Structure of the Python Weather Station Code Compiler .....	53
Can the System adapt to Additional Databases? .....	54
Setting the Final Executable as a Desktop Background .....	54
Chapter Four: User Experience Design .....	54
Designing Based on Established Methods .....	56
Chapter 4.5: Weather Effects Compilation .....	60
Examples of Weather Changes .....	60
Chapter Five: Conclusion and Looking Forward .....	69
Possible Critiques of the Work .....	73
References .....	79
Appendix A .....	81
Introduction .....	83
Prerequisites .....	83

Software Information .....	84
Resources and How-To's.....	85
Creating Cloud Textures in Substance Designer and Exporting to Unreal .....	85
Expose Parameters in Substance Designer .....	89
Exporting a Substance for Unreal .....	95
The Substance Plug In for Unreal Engine .....	96
Installation .....	96
The Plug In Explained.....	97
The Parameter Window.....	99
Essential Nodes in Unreal Material Editor .....	100
Add and Subtract: the Value Shifters.....	100
Multiply: The Mask Node.....	101
Clamp: Keeping values under control.....	102
Simple, Yet Powerful.....	104
The Core Node Network .....	104
Section one: The Sun .....	105
Section Two: The Clouds.....	106
On your Own.....	108
Learning Resources .....	108
UV Warping A Red and Green Texture Map .....	109
Packing Textures to RGB in Photoshop.....	110
The Main Problem to Solve: Translating Vector Direction into Degree Rotations .....	112
The Whole Graph.....	113
Glossary .....	123

## List of Figures

Image Description	Figure Code
Pre-Production Sketch Collection	1.1.1 – 1.1.4
Pre-Production Concept Painting Collection	1.2.1 – 1.2.4
Miyazaki film environment	1.3
Legend of Zelda: Breath of the Wild	1.5
RiME Environment	1.4
Alexandre Diboines Image	1.6
Alex Konstad Image	1.7
Luke Mancini Image	1.8
Triumphal Arch	1.9
Comparison of hero asset silhouette to generic object	1.10
Comparison of jagged silhouette to smooth one	1.11
Balck Widow Image	1.12.A
Crystal Lake Mountains	1.12.B
Mushroom Simplification Drawing	1.13
Prickly Pear Cactus Reference	1.14.A
Prickly Pear Cactus in Chromniview	1.14.B
Mario + Rabbids: Kingdom Battle	1.14.C
Call of Duty Screenshot	1.15
Legend of Zelda: Breath of the Wild VFX	1.16
Explosion Reference	1.17.A
Landscape Study in Gouache	1.17.B
Chromniview Fog VFX	1.18
Hard and Soft Vertex Normal Comparison	2.1
High Poly and Low Poly Comparison	2.2
Infographic: RGB = Vector Angle in a Normal Map	2.3

Arcane Scale	2.4
Example of Vegetation Cards	2.5
Modified Normals on Grass Card Example	2.6
Substance Designer Results Example	2.7
A Sculpted Trim Sheet	2.8
Curvature Map Example	2.9
Chromniview Shader Utilizing Curvature Map	2.9.A
Procedural Copper : Base Mat and Details	2.10
The Order: 1886 Texture Examples	2.11
Rime VFX Fire Technique	2.12
Final product: Rime Flames	2.13
Black and White Cloud Shader from Chromniview Beta	2.14
Deformed Gradient Clouds Shader from Chromniview Final	2.15
Snippet of Shader Illustrating a Material Function	2.16
A Full Material Function Shader Graph	2.17
Relationship Between Compass Direction, UV, and Vector Space	3.1
Menu System image	4.1
Old Menu Design	4.2
Paper Prototype	4.3
Digital Prototype	4.4
Weather Effects Compilation	4.5

Outline Shader in Unreal Engine	5.1
Cel-Shading and Shadow Manipulation in Unreal Engine	5.2
Borderlands 2D/ Handdrawn shaders	5.3
Guilty Gear XRD Outline Effect	5.4
DragonBall Outline Effect	5.5
Original Thesis Statement	5.6
Original Method for Enshrining Nature	5.7
Mock-Up of Desktop Integration	5.8

## Introduction

Culture and technology in the modern age are deeply entwined. As our lives become more “high tech,” the norms and expectations surrounding our entertainment experiences grow in complexity and connection. Although contrived, the idea that “the world is at your fingertips” has never been more true. But the irony of such intoxicating power at a personal level is the tunnel vision that coincides. The immense opportunity for micro control forces our attention into the artificial and minute—away from the cosmic mystery that surrounds us. And as the human race’s collective spine slowly cranes towards the dim glow of an entirely digital age, the wondrous miracle of mother earth roils just beyond the window, slipping into corruption and irrelevance.

This dynamic weather system (dubbed Chromnview) utilizes particle systems, polygon shaders, and code/logic to create a system that can respond to weather data including word-based descriptions, temperature levels, windspeed amounts, and time of day. One of the first conceptual versions of this project included the ability to run the program on the desktop as an interactive wallpaper, behind the icons. The system is self-contained within the engine, and can be transferred between levels and even entire projects. It can simulate mid-day rainstorms, quiet snowfall at night, and clear, sunny mornings.

The implementation of the myriad and complex features included in the initial pitch of this project was very successful, but not absolute. Some features evolved over

time, such as the user interface. The initial concept of using Rainmeter for the interface was traded out for Unreal Engine's UMG Designer, producing a more integrated, visually pleasing, and game-like experience. Other features were too resource-heavy to complete for this proof-of-concept, such as formatting the application as an interactive desktop wallpaper. Still others were discovered and frantically pursued out of pure necessity, such as the JSON request builder for querying the user's desired weather-station.

In my studies I have harnessed various technologies and achievements of other humans and created meaningful translations between them in order to craft a new avenue for us to remember and enshrine the glory of Nature. Chromnview is merely a messenger of Nature's perfect design. It reflects Her beauty, authenticity and meaning in the inherently redundant existence of digital media.

Chromnview offers a novel passive connection with others. Though many of us may be chained to our desks during the average work week, or we chase those elusive jobs across the country, this application offers a non-intrusive method for "being present" in the experience of faraway friends and family. By seeing the weather and general time update based on their earthly position throughout the day, we can keep our parents, children, or best friends at the forefront of our minds.

Even if Chromnview is slightly inaccurate due to a lack of weather stations in less populous areas or long refresh periods, it offers a fresh perspective on "talking about the weather." It gives you a reason to call up a friend to confirm that it is actually

snowing in the middle of summer at their location. The artistic value in this project comes from the opportunity for passive connection it creates by being a medium for Nature to act upon. It encourages a very real connection by faithfully interpreting one of the few things all humans have in common: the earth's ecosystem.

In Chapter One, I explain the artistic style and approach to design used for the entire project. I discuss and examine specific inspirations from recent media, and point out the key areas that the style of Chromnview emphasizes in order to glorify nature. Chapter Two is an overview of the many resources, tools, and techniques that went into the creation of the weather system and environments. In Chapter Three, I break down some key scripting and the technical integrations between shaders. Chapter Four details the User Experience, and the tenants that drove the design of the usability. Finally, Chapter Five offers a conclusion and possible future directions from this point.



## Chapter 1: Artistic Direction and Development of Style

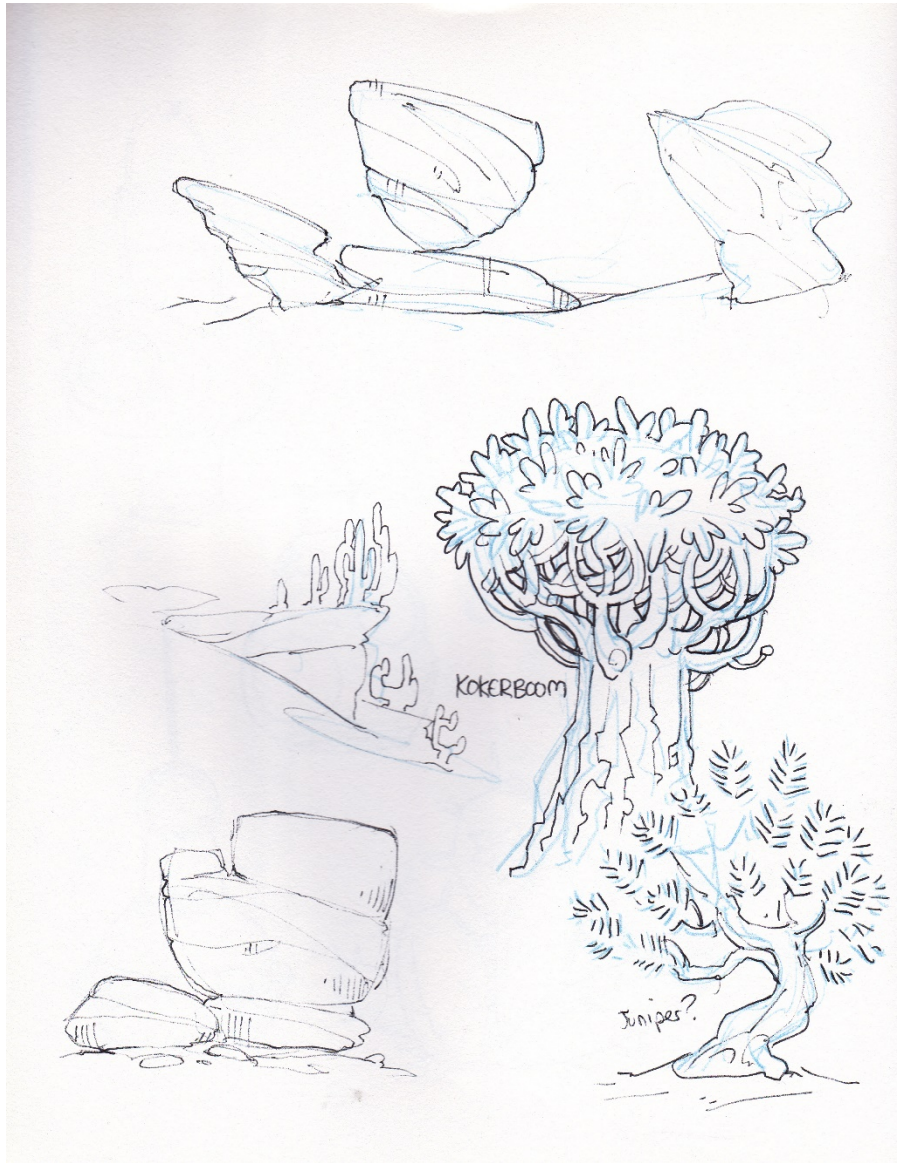


Figure 1.1.1 A Collection of PreProduction Sketches for Chromniveu. Zoey Schlemper. Blue Pencil and Pen. 2017.

1

---

<sup>1</sup> These drawings begin to dissect the shapes and structure of natural elements, including rocks and trees. They explore without complicating. The modest line-count and emphasis on outline serve this end as well.

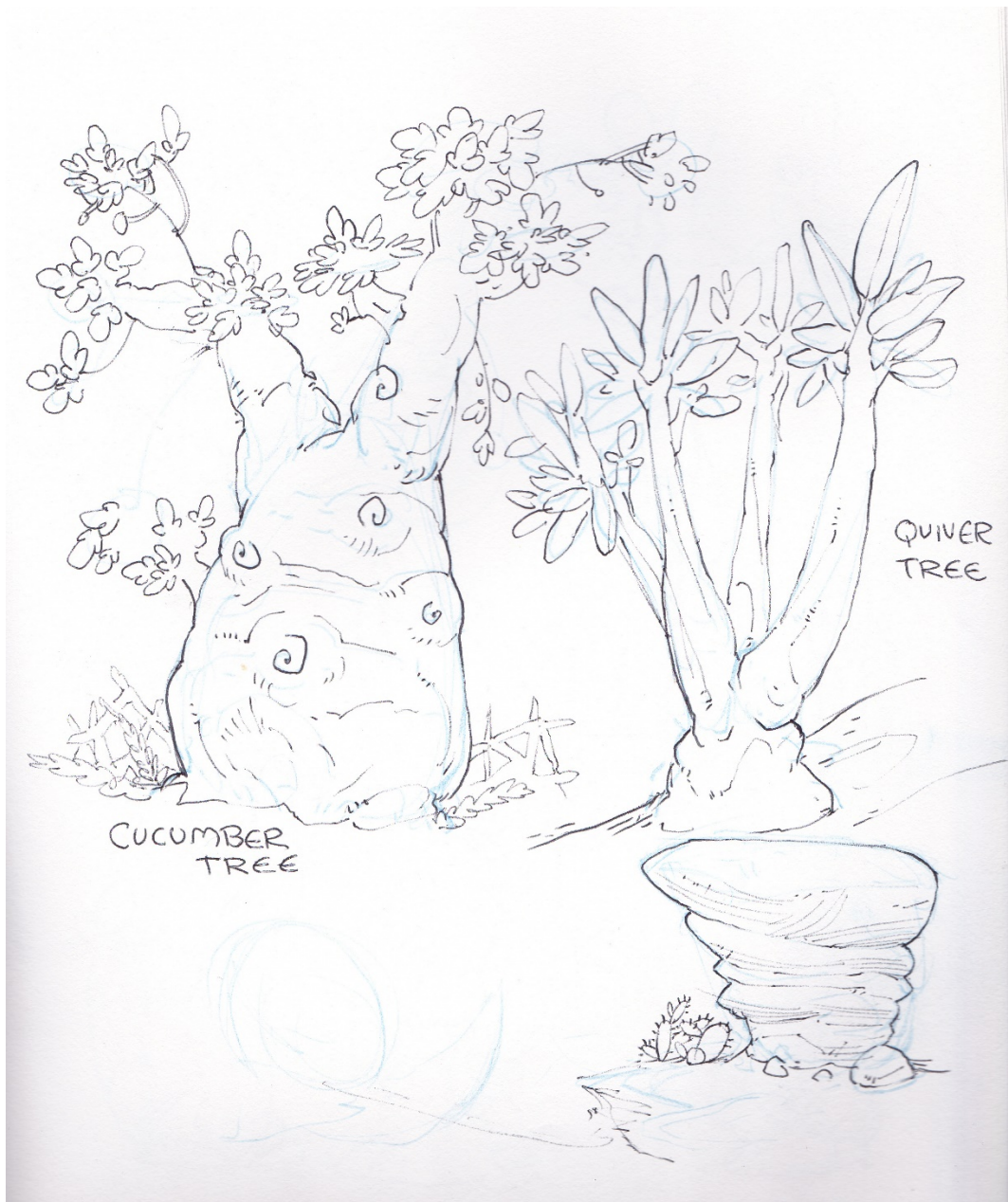


Figure 1.1.2 A Collection of PreProduction Sketches for Chromnivism. Zoey Schlemper. Blue Pencil and Pen. 2017.



Figure 1.1.3 A Collection of PreProduction Sketches for Chromnview. Zoey Schlemper. Blue Pencil and Pen. 2017.





Figure 1.1.4 A Collection of PreProduction Sketches for Chromnview. Zoey Schlemper. Blue Pencil and Pen. 2017.

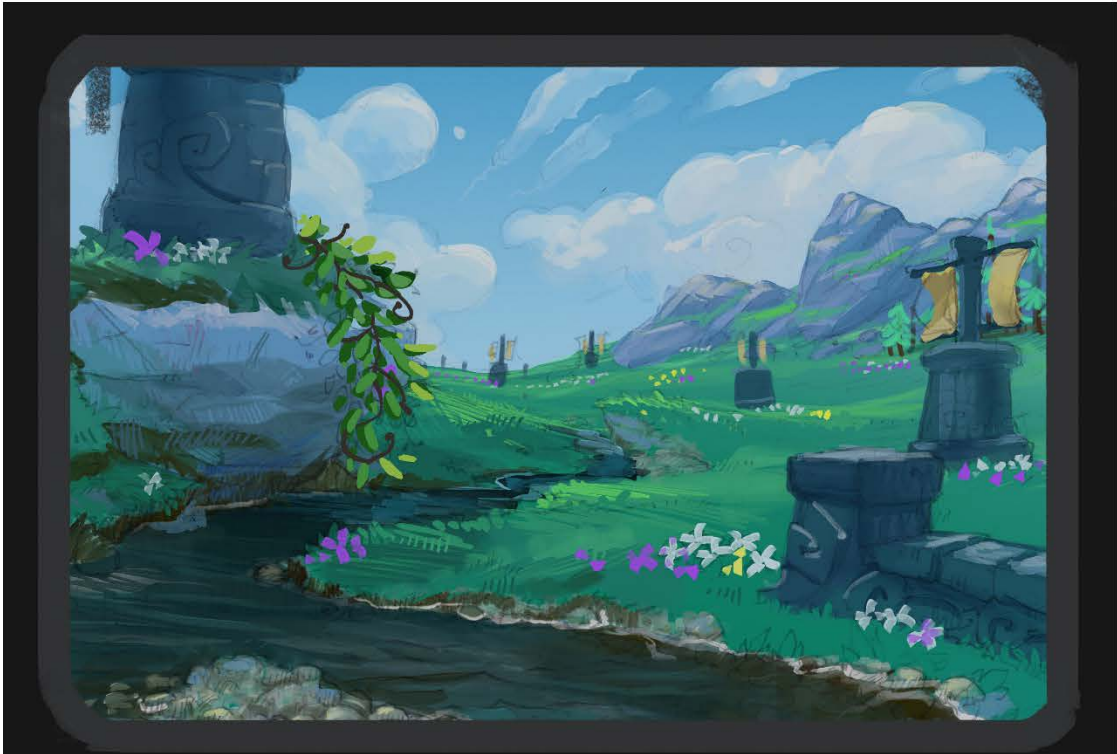


Figure 1.2.1 A Collection of PreProduction Concepts for Chromnview. Zoey Schlemper. Pencil and Digital. 2018



Figure 1.2.2 A Collection of PreProduction Concepts for Chromnview. Zoey Schlemper. Pencil and Digital. 2018





Within the Windy River Valley



Atop the Stairway at the Forest's Edge



Figure 1.2.3 A Collection of PreProduction Concepts for Chromnview. Zoey Schlemper. Pencil and Digital. 2018



Figure 1.2.4 A Collection of PreProduction Concepts for Chromniview. Zoey Schlemper. Pencil and Digital. 2018



Figure 1.3 Tonari no Totoro. Hayao Miyazaki. Studio Ghibli. 1988.





Figure 1.4. RiME. Tequila Softworks. 2017. <<https://www.nintendo.com/games/detail/rime-switch>>

## Concept Development and Aesthetic Style

The aesthetic style of this project is heavily influenced by hand-drawn and two-dimensional animations. The works of Miyazaki (fig 1.3), Tequila Works' "RiME" (fig 1.4), and certain Nintendo titles (fig 1.5) have been important. Concept Artists like Alexander Diboines (fig 1.6), Alex Konstad (fig 1.7), and Luke Mancini (fig 1.8) also influenced the aesthetic language of this project. Finally, I hope that my own voice (fig 1.9) has peeked through the cracks, as I've developed a stronger personal style over the past 6 years. These influences share a number of things in common, including an emphasis on shape language, vibrant color palettes, and reductionist representation. The most important similarity is the emphasis on nature through these devices. I will consider two

dimensions of these artistic principles: that concerning the enshrinement of nature, and that concerning game art.



Figure 1.5. *Breath of the Wild*. Nintendo. 2017. < <https://80.lv/articles/the-legend-of-zelda-breath-of-the-wild-environment-change-the-game/>>



EVERYWHERE HE WAS GOING, PEOPLE WOULD REJOICE WEEKS IN ADVANCE,  
AWAITING THE VENUE OF THE ULTIMATE CHEF, THE COOK OF LEGEND ...

Figure 1.6. *Everywhere He Would Go*. Alexandre Diboinés. Digital. 2017. < <http://alexandrediboine.tumblr.com/>>





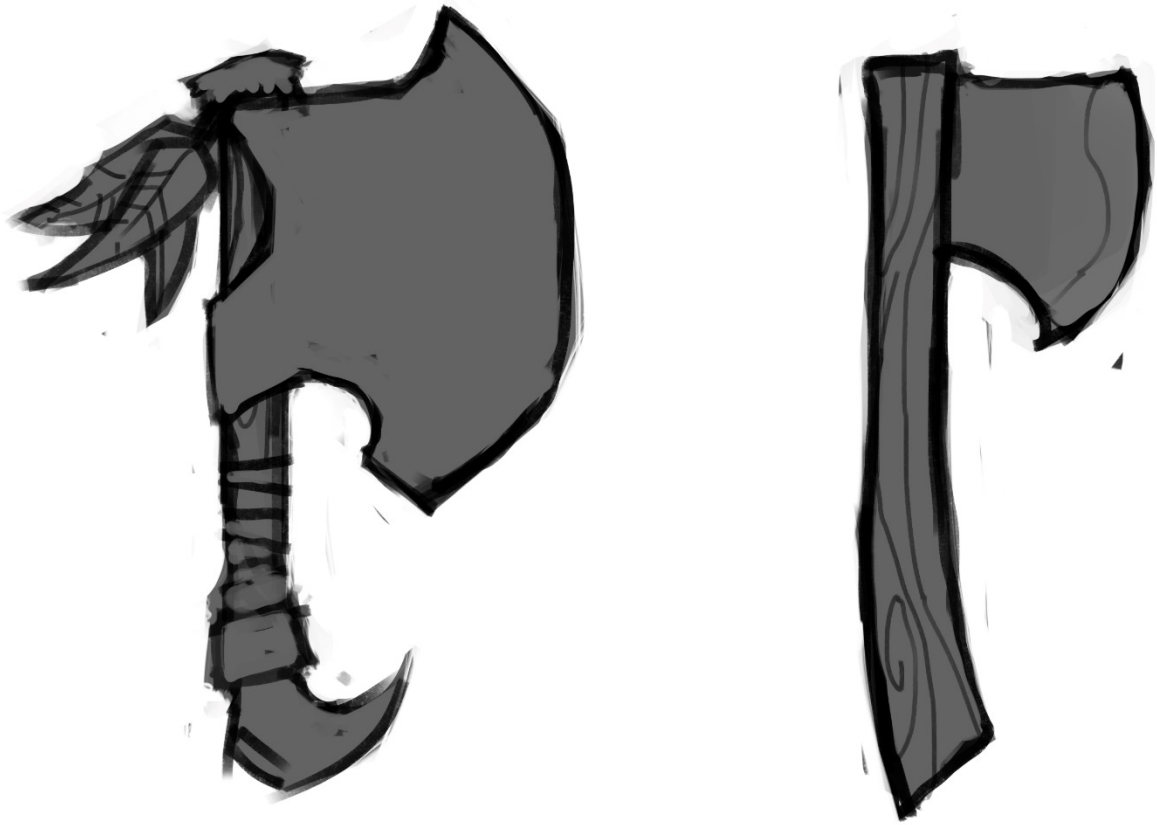
Figure 1.7. *Obliskura: Forest*. Alex Konstad. Digital. 2015. < <https://www.artstation.com/artwork/6LRBO>>



Figure 1.8. *Hilltop Encounter*. Luke "Mr. Jack" Mancini. Digital. 2012. < <https://mr--jack.deviantart.com/art/Hilltop-Encounter-285881079>>



Figure 1.9. Triumphal Arch. Zoey Schlemper. Digital. 2016.



*Figure 1.10. Hero Asset VS Generic Asset. Zoey Schlemper.*

Shape language is essential in both videogame development and our understanding of nature, but for different reasons. For videogames, shape language can signal many forms of information to the player. It can indicate the importance of something. For example, “Hero Assets” tend to have more break-up in their silhouette than generic pieces (fig 1.10). It can also show the general mood of a game. Compare the general shapes of Mario enemies (fig 1.11.A) to Dark Souls (fig 1.11.B). The enemies in Mario are mostly “safe” looking, with rounded, simplistic designs. The Dark Souls enemies are complex and angular. Shape language seems to be considered for many



reasons, including the relative importance of something and the over-all tone of the game.

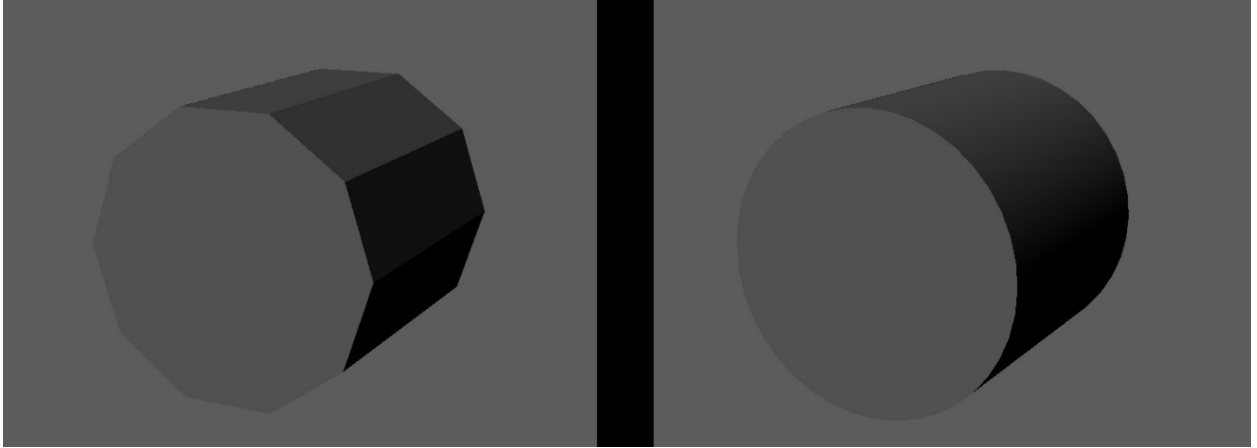


*Figure 1.11.A. Goomba and Koopa Troopa. Nintendo. < <http://twinfinite.net/2016/03/top-10-most-iconic-video-game-enemies/5/>>*



*Figure 1.11.B. Sewer Centipede. Capcom. Dark Souls 3. <[http://darksouls.wikia.com/wiki/Sewer\\_Centipede](http://darksouls.wikia.com/wiki/Sewer_Centipede)>*

A smooth silhouette is a luxury afforded by the ever-increasing capacity of computer technology, and is also important to help facilitate the suspension of disbelief (fig 1.9). Ironically, simple shapes like those of Mario enemies are difficult to create, as polygons can only be made of straight lines.



*Figure 1.12. Jagged Silhouette from low poly count compared to smooth silhouette from high polycount. (provided by author, rendered in Maya 2016)*

Due to the commercial nature of video games, it could be argued that all choices concerning shape language are dictated by users. Nature, on the other hand, is her own master. In some cases, we either respond to her dictation, or we die. “Nature” is an umbrella term that includes countless phenomenon, creatures and scientific properties. Nature is immutable and older than the mind of man. It dictates the meaning of its shapes by killing us (fig 1.12.A) and awing us (fig 1.12.B) with them.





Figure 1.12.A. Black Widow Spider.<sup>2</sup> Photography. National Geographic. 2015. <  
<https://www.nationalgeographic.com/animals/invertebrates/group/black-widow-spiders/>>



Figure 1.12.B. Crystal Lake Mountains. Zoey Schlemper. Photography. 2017.

My primary inspirations of shape language include Alexander Deboines and Alexander Konstad. In two words: curvilinear and voluminous. I deconstructed natural shapes into exaggerated primitive volumes in order to invoke a sense of “alien” and

---

<sup>2</sup> The design and general shape of the Black Widow illicit a response based on the actions it takes against humanity and other creatures. There is no human origin for the meaning behind the design.

“new perspective” on archetypes such as mushroom (fig 1.13), mountain, and cactus that we take for granted. This allowed the essence of a shape to take control over its net impact, such as the way that 3 giant leaf shapes can constitute a “bush” just as well as 100 can. This cactus has been reduced to a few spines on an inflated body, straying heavily from a naturalistic representation but retaining the defining qualities (fig 1.14.A-B). Consider the sketch of the cactus from figure 1.1.4 from page 7 for the treatment of the subject in the concept as well. A great example of this technique is in *Mario + Rabbids: Kingdom Battle* (fig 1.14.C)



Figure 1.13. *Reduction of Shapes in Mushrooms Exercise*. Zoey Schlemper. Graphite on Paper. 2017.



Figure 1.14.A. Prickly Pear Cactus. El Charco del Ingenio. Photography. 2008. <  
[http://www.elcharco.org.mx/boletines/not\\_vol3no06.html](http://www.elcharco.org.mx/boletines/not_vol3no06.html)>



Figure 1.14.B. Prickly Pear Cactus. Zoey Schlemper. 3D Render. 2017.



*Figure 1.14.C. Mario + Rabbids: Kingdom Battle. Nintendo & Ubisoft. 2017.*

The colors of this project are vibrant and focused on creating a “hypersensitive” experience in the viewer. Imagine if, instead of only having 3 color-sensitive cones in your eye, that you had 6. Drawing parallels between the sense of taste and sight were my primary vehicle for achieving this. I tend to think of visuals in terms of how “tasty” they are. Are the colors sweet or sour? Does the shape make me salivate? This type of free-association was an exciting way to approach creation.

This project also utilized the full range of colors available on the modern screen, unlike the drab and grim realism of many AAA titles (fig 1.15). In order to properly emulate nature, increasing the potency of color was an imperative. Notice the difference between figure 1.15 and 1.14.C. Both of them are outside and include human



constructions, but the balances of nature vs civilization are different. The absence of mankind is synonymous with vibrancy in this example.

In order to glorify nature, I use the same correlation between mood and existence of mankind, and describe this by increasing the vibrancy of the non-man-made. The compilation of my work in chapter 4 helps illustrate this. In those images, I have made the physical materials of wood, metal, stone and kept their colors lower in vibrancy, and lacking in hues that are commonly associated with energy and happiness, such as highly saturated, high value greens, reds, and blues. Instead, the colors either skew towards earthy, or at the very least tend to be highly desaturated, similar to figure 1.15 below.



*Figure 1.15. Call of Duty, Beach Invasion. Activision & Infinity Ward. 2013.*

The VFX are inspired heavily by Miyazaki films and *The Legend of Zelda, Breath of the Wild* (fig 1.13). My interpretation of the elements focused on a reduction of sheer number, as in moving from sand grains to pebbles. In figure 1.13, we can see about 4 large cloud bursts in the explosion and 3 debris comets. The colors have been cell shaded to resemble something like gouache (fig 1.13.A). Whereas in figure 1.13.B we see at least twice as many of all of those, and fractal-like details on each smaller area, as well as a wide gamut of color. In my project, I reduce and simplify my VFX to reflect the same methods of simplification (fig 1.14). I took these simplified forms and emphasized their fluidity and rhythm. Hopefully this creates a sense of whimsy and imagination.



Figure 1.16. Legend of Zelda: Breath of the Wild. Nintendo. 2016. < <https://realtimevfx.com/t/cartoon-explosions-fire-effects-w-ref-legend-of-zelda/1490>>



Figure 1.17.A. Landscape Study in Kansas. Matthew Cook. Gouache. 2012.



Figure 1.17.B. Titleshot from Biggest Explosion Compilation 2015. InstaVids. Video. 2015.  
<<https://www.youtube.com/watch?v=zhBxfNm3rJE>>





*Figure 1.18. FogVFX. Zoey Schlemper. 2018.*

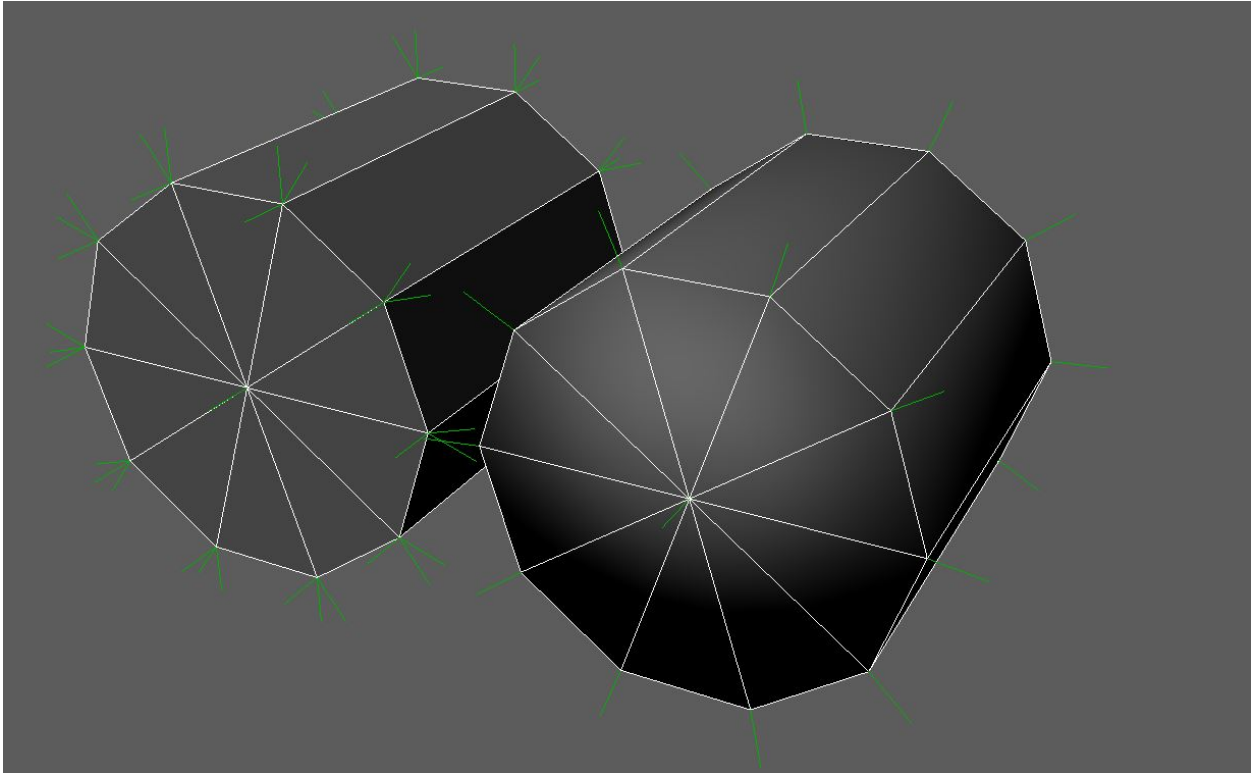
In terms of lighting the environments, I made certain each shadow was colorful. I followed the current trends of realism in lighting that PBR materials are so well-known for, but hiked up the floor of the value range. Also, metallic surfaces have had their effects reduced for the sake of subtlety and so as not to overpower the other aspects of each scene. In order to achieve this, I increased the roughness parameter greatly, and reduced the metallic parameter slightly.

The lighting set-up is simplified to a uniform color fill light and a directional light. This fill light emulates the sky and is comprised of two effects in-engine: a skylight, which adds color and reduces shadow by emanating light from a dome across the entire scene; and the environment color, a core setting of the entire level that tints the absolute black of shadows to a color and brightness specified by the user. The



directional light, unlike a spotlight, has no visible end of influence. There are no additional spot or edge lights to ensure computational efficiency and staying loyal to natural lighting.

## Chapter Two: Techniques and Creative Process



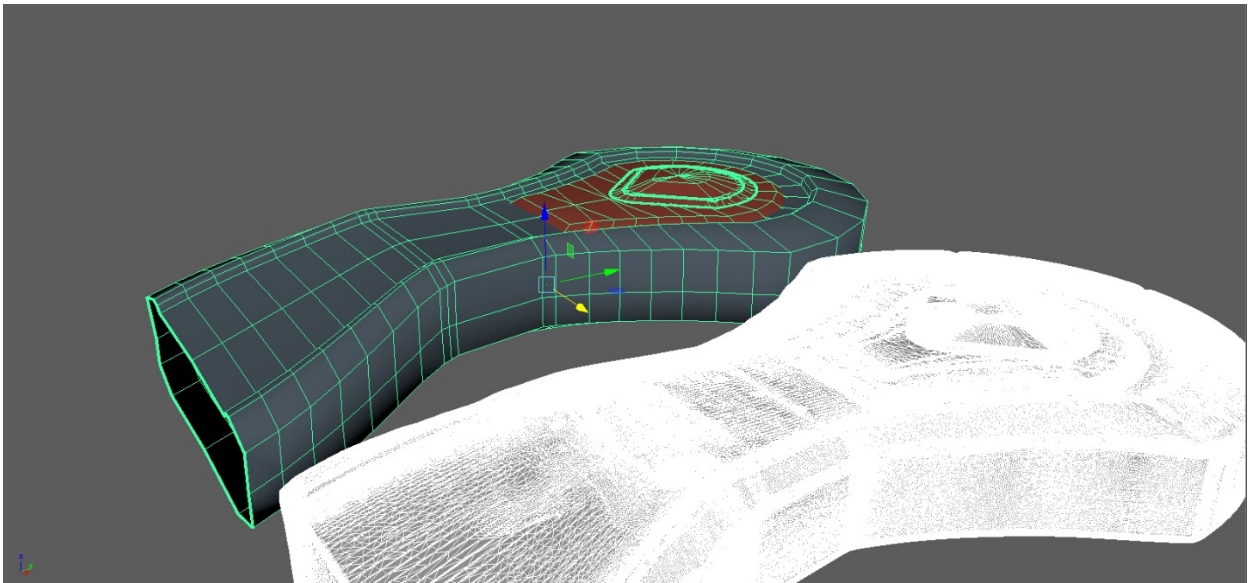
*Figure 2.1. Hard Normals Versus Soft Normals. Provided by Author, Rendered in Maya 2016.*

### 3D Modeling Workflow

Video game environments are made of points in Cartesian coordinates. These points are called vertices, and are connected to one another by edges (the white lines in figure 2.1) and when edges enclose an area, they form polygon primitives that have an image (or “texture”) mapped to their surface area using UV coordinates. Each vertex can have a soft or hard transition between itself and its neighboring vertex. This is a result of the “normal” of the vertex, indicating the perpendicular direction from the polygon surface. Notice in fig 2.1 how the hard transitions have multiple green lines protruding, while the soft transitions have only one. This is because on a hard edge, a vertex simulates an angle change by having separate facing directions for each polygon

it helps construct. Vertices can store many other additional data types, like a color that is not rendered known as “Vertex Color” or a measure of influence from a joint known as a “Skin Weight.” Using all of these attributes and extra storage options is necessary for creating complex and convincing 3D models. In this chapter, I will show a general workflow for making assets, and the considerations I take in order to maximize utility and flexibility when importing to a game engine.

The general workflow is a close relationship between the software packages called Maya and Z Brush. Many artists use a workflow referred to as “high to low”, including myself. This is when you create a detailed virtual sculpture, often millions of polygons in size, and then you create another model with an extremely low polycount using the detailed sculpture as reference (fig 2.2).



*Figure 2.2. High Poly (right) and Low Poly (left) with wireframes showing. Provided by Author, Rendered in Maya 2016.*

From there you can bake a Normal texture map. This texture encodes the refined surface angles (a vector in XYZ space) from the high polycount model as a color in an image (fig 2.3) mapped to the low polygon model. The game engine then loads the low polygon model, and when computing the lighting, takes into account that image in order to reproduce high fidelity detail for a fraction of the processing power.<sup>3</sup>

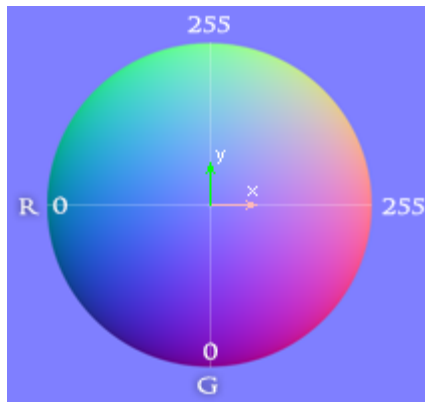
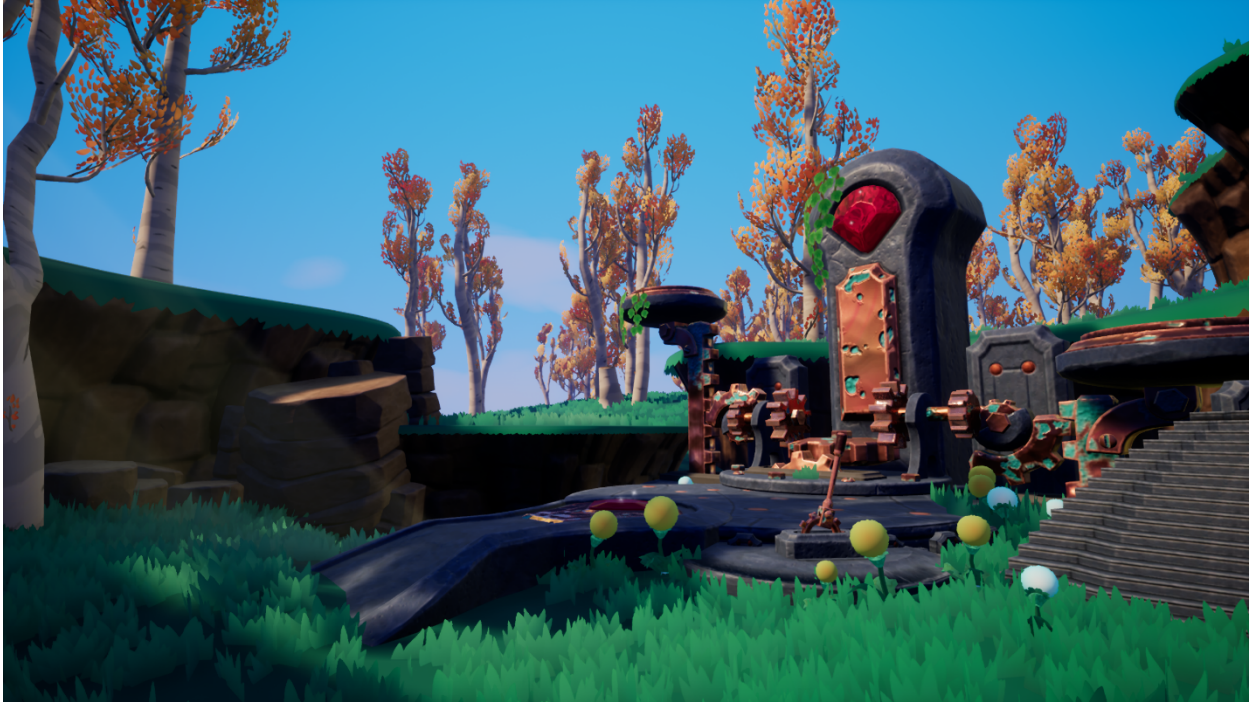


Figure 2.3. Visual Example of how RGB values exist on an XYZ axis. FallOut Software. <<http://www.falloutsoftware.com/tutorials/gl/normal-map.html>>

---

<sup>3</sup> (Squirle Art. 2017)



*Figure 2.4. Arcane Scale. Zoey Schlemper. Digital Multimedia. 2018.*

This technique was used for many of the assets, including almost all the rocks, and hero assets such as the Arcane Scale (figure 2.4). Time management was crucial for this project due to the limited manpower compared to the volume of assets required. Occasionally, high-poly models and normal maps were not necessary to make if some assets were not going to receive as much attention by users, such as the grass blades.

There were some instances where I had to hand-modify the normals of a mesh in order to achieve the best visual results. This is most noticeable on the trees and grass models. Much vegetation in videogames is made of “cards,” which are essentially 2D planes with pictures of leaves and stems on them. These cards are intersected in order to create volume for minimal cost. However, these intersecting cards tend to react to light in a jarring, simplistic way (fig 2.5, left half of image). I tilted the normal of the

planes up and softened the edge transitions so that the grass would softly transition from light to dark. I also kept them one-sided and had two planes facing opposite directions in order to form the illusion of a single double-sided card. This is because the normals are tilted up but not perfectly perpendicular to the ground, causing a lighting inversion on the card's backside (fig 2.6).<sup>4</sup>

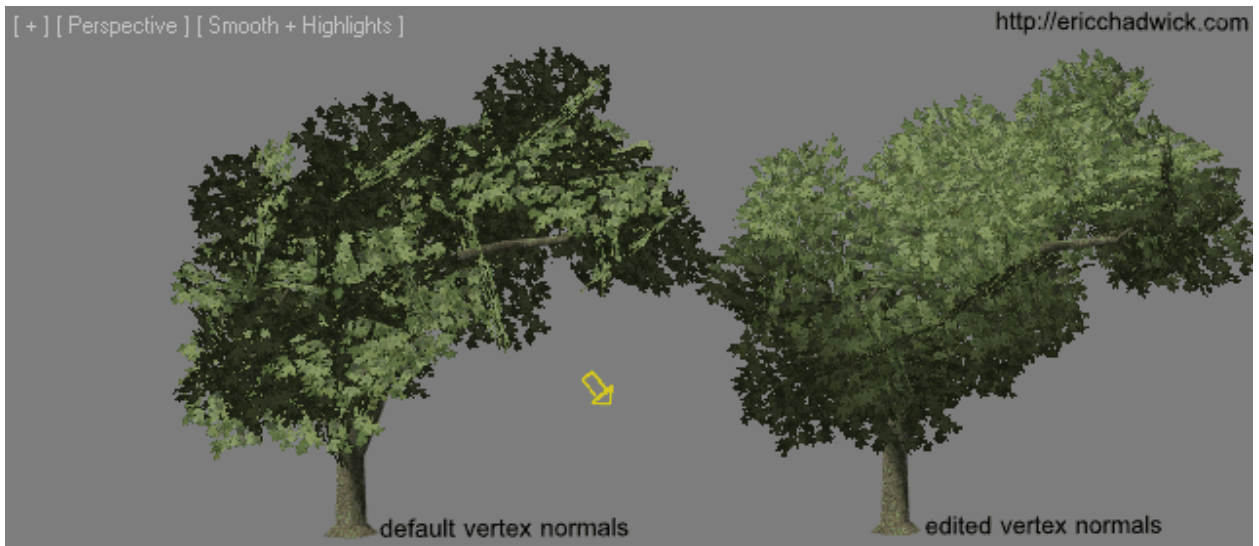
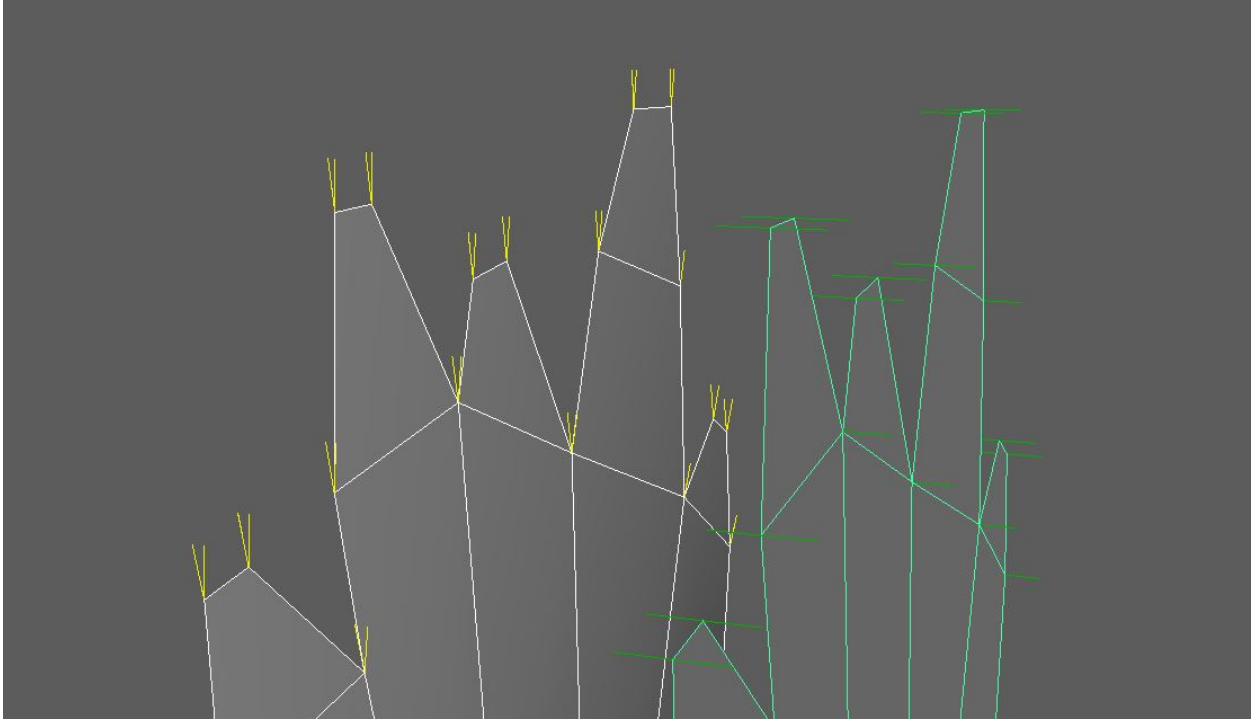


Figure 2.5. *Hard Vertex Normals (Left) and Smooth Vertex Normals (right) on Trees.* Eric Chadwick. *Digital Multimedia.* < <http://wiki.polycount.com/wiki/Foliage> >

---

<sup>4</sup> (Authors 2018)



*Figure 2.6. Modified Normals (left) and Regular Normals (right) for grass cards. Zoey Schlemper. Digital 3D. 2018.*

## **Texturing Workflow**

I found myself intrigued by procedural texturing, and attempted to apply it to stylized environment work. Procedural textures can achieve startling levels of realism, and allow artists to plow through otherwise mind-numbing levels of detail necessary for that realism by using automatic tiling, noise manipulation and other generators (fig 2.7). In terms of stylized artwork, procedural texturing is useful for saving time, achieving perfect tiles, and rapid prototyping. The downsides of procedural texturing include a lack of control in fine detail, and a general lifelessness of color and texture in terms of the stylized and hand-painted genre. In some cases, achieving the desired effect in a



procedural texture may take more time than doing it hand-painted, neutralizing a key benefit of the procedural workflow.

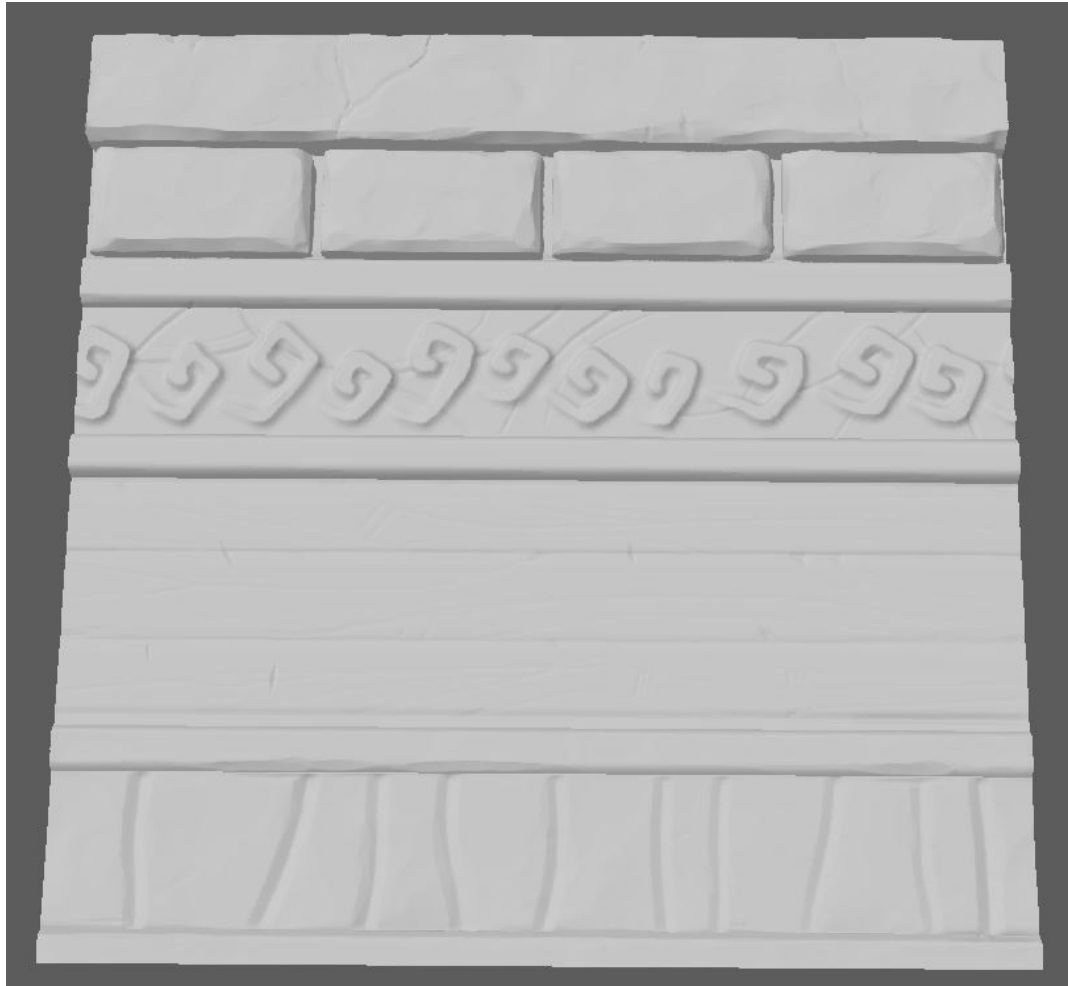


*Figure 2.7. Sloppy Brick Wall Material Study. Joshua Lynch. Procedural Shader Network.*

In this project, procedural textures worked best for less important materials that aren't meant to draw a lot of attention. For example, the cast iron on the archway in Windmill Valley was procedural. However, the wood, brick, and decorative trim were sculpted by hand (fig 2.8). Combining procedural and hand-made textures in the stylized genre required me to make artistic choices that allow the two production techniques to have consistent results. For example, I sometimes took the albedo from a procedural texture into Photoshop in order to add hand-painted details. When creating textures manually, I made sure to copy the general process of color build up used in the



procedural textures, such as working in grayscale for the albedo and then applying a color ramp, before proceeding to hand-painted detail. Some textures are purely one or the other, while others use a combination of both to achieve the desired effect.



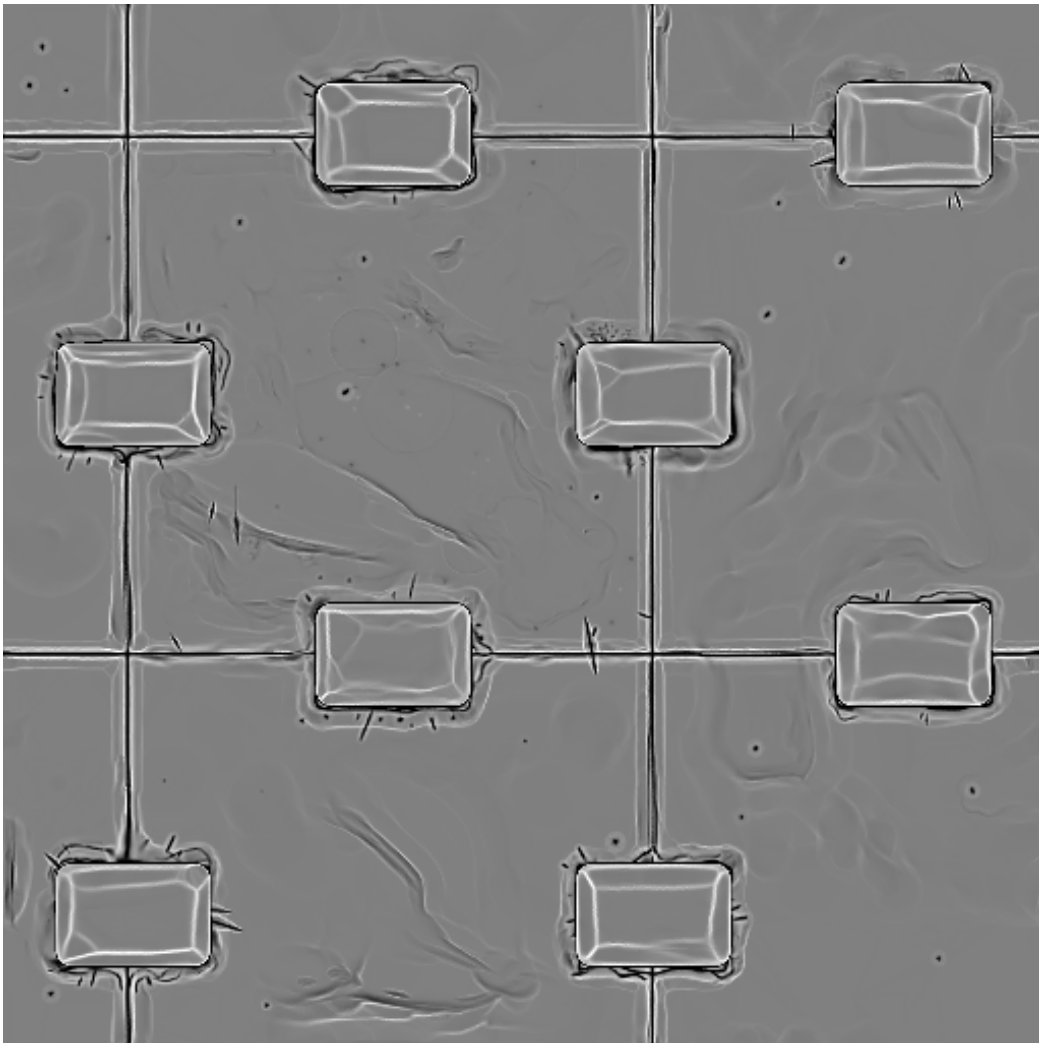
*Figure 2.8. Trim Sheet Sculpt. Zoey Schlemper. Digital Sculpture. 2017.*

5

---

<sup>5</sup> This is an early version of the sculpted trim sheet that appears in the project, and appears here solely for the purpose of describing a technique rather than an outcome.

Procedural workflows require an understanding of how the textures will be implemented at various levels of the pipeline. For example, a stylized copper shader I made relies on a curvature map<sup>6</sup> (fig 2.9) in order to generate some detail. The white areas of the map were used to add edge wear. In figure 2.9.A we can see a curvature map being used to add some detail to the edges of the gear.



*Figure 2.9. Curvature Map Render. Zoey Schlemper. Digital Multimedia. 2017.*

---

<sup>6</sup> A Curvature map shows the change in angle from one polygon to another as black and white.



Figure 2.9.A. Large Copper Gear. Digital Multimedia. Zoey Schlemper. 2018.

As an environment artist, most of my work will need to have as much mileage and flexibility as possible. In order to widen the use-case of all textures and meshes, I separated each procedural material into two parts: The base material (like copper) and detail layers (dirt, scratches, oxidization) that add variety. We can see these details working in figure 2.9.A. I can then combine these different layers in Substance Painter, using mesh-specific maps such as curvature or thickness in order to build complex and mesh-unique materials from a few simple textures. This “texture library” approach was used to impressive effect in the game *The Order: 1886* (fig 2.11).

These techniques allowed me to consider nature in a pragmatic way. Just as the earth is created from many layers of dirt, rock and minerals, my environment surface is made from similar layers: dirt, cobblestone, stone, grass, and snow. Although a user

may not be aware of this technique, the parallelism of natural phenomenon and artistic approach is worth noting, especially since a key part of this project was to create a conduit of nature. By copying her approach to terraforming, I hope that my system can show some form of reverence.



Figure 2.11. Sample Textures from *The Order: 1886*. Ready at Dawn Studio. Procedural Shader Network. 2015.

## Shader Development in Unreal Engine 4.16

Creating shaders for Unreal Engine required me to become heavily involved in visual scripting interfaces. I picked up many skills from established studios, benevolent students, and scholarly professionals. Rather than reiterate their work here, I will

describe broader interpretations and principles of the design process for this project. If you are curious about explicit details, I discuss some minutiae of these techniques in other documentation, and point towards specific research there.

The visual effects were inspired heavily by RiME (fig 1.4) and Breath of the Wild (fig 1.5). I used some of the exact same techniques as the creators of RiME, but applied them in a different way. All of these shaders were produced in Visual Scripting Environments with no use of text coding.

Simon Schreibt goes into detail about how Tequila Softworks (the creators of RiME) produce water, fire, and vertex painting techniques in a stylized manner. Their technique for fire revolves around creating color masks by deforming a simple gradient (fig 2.12) with panning UV values. With meticulous balancing between amount of deformation and the type of gradient this is applied to, it is possible to create mesmerizing and smooth flames (fig 2.13). <sup>7</sup>

---

<sup>7</sup> (Schreibt 2017)



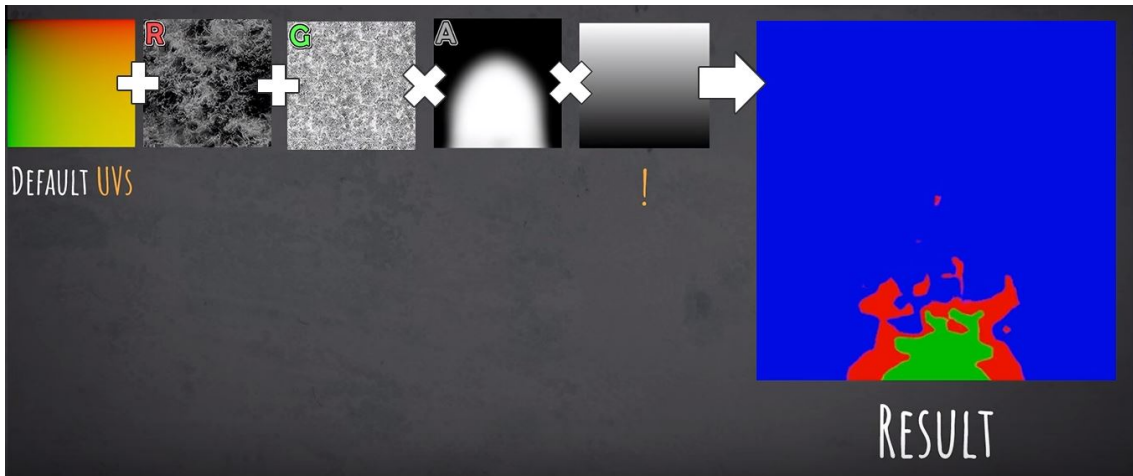


Figure 2.12. RiME Flame VFX Breakdown. Simon Schreibt. Presentation Slide. 2017.

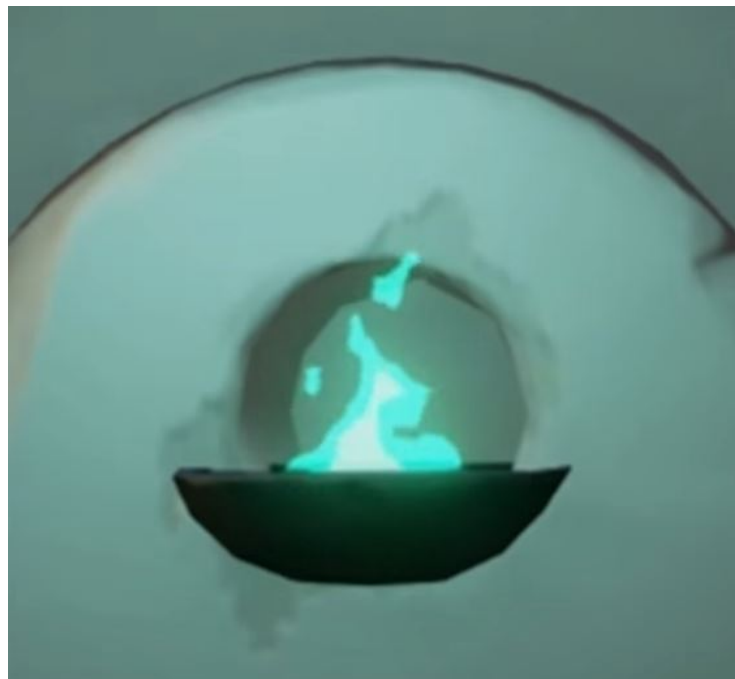


Figure 2.13. RiME Flame VFX. Tequila Softworks. 2017.

While developing my procedural cloud shader, I discovered this technique and adapted it to the clouds. Before that, I was using a rather primitive multiply mask (fig 2.14), which created problems with value banding and a lack of control of value placement, resulting in very lumpy, flat clouds that often became pixelated. The use of

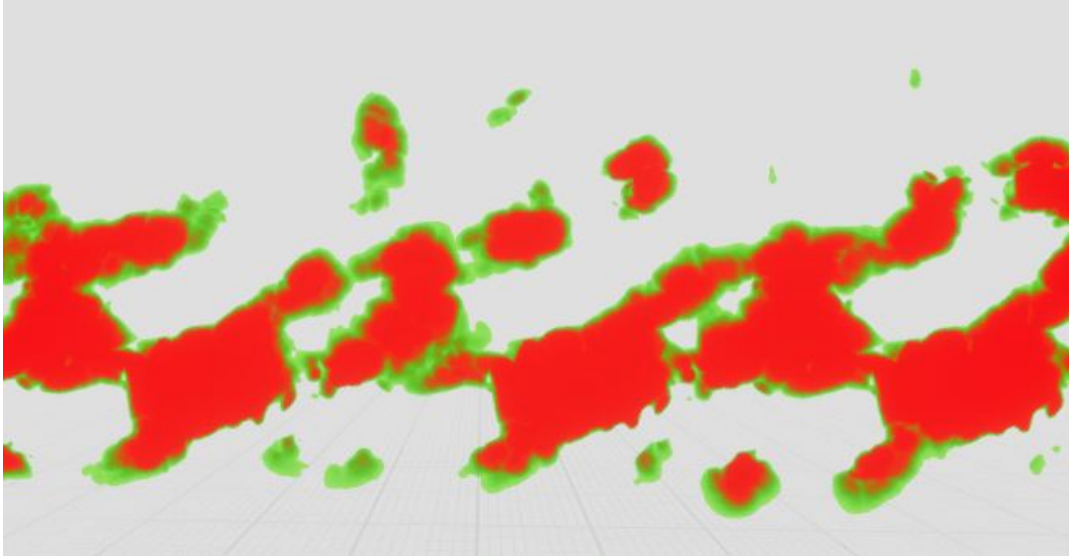
UV deformation allowed me to take this basic set up and use it to shape a higher resolution gradient, solving all of the previously stated problems (fig 2.15). For a full explanation of the cloud shader, please refer to *Appendix A: 2D Procedural Sky Shader: Using Unreal Engine 4.16 Material Editor*.<sup>8</sup>



*Figure 2.14. Procedural Cloud Shader Black and White Texture (BETA). Zoey Schlemper. Procedural Shader Network. 2016.*

---

<sup>8</sup> (Schlemper, 2D Procedural Sky Shader: Using Unreal Engine 4.16 Material Editor 2017)



*Figure 2.15. Procedural Cloud Shader Gradient Mask UV Distortion. Zoey Schlemper. Procedural Shader Network. 2017.*

Other important principles of shader development included using consistent value ranges for variables that allowed them to be both easy to understand and universally applicable in many different mathematical situations. For example, a universal “cloud amount” value controls the intensity of sunlight, the amount of clouds that appear and how much a gel/cookie blocks the sunlight to name a few. The cloud amount needs a value between  $-0.3$  and  $0$ , whereas the sunlight intensity needed a value between  $1-7$ . The original value is actually just  $0-1$ , and it is remapped to the range required by each unique shader. This helps reduce confusion and allows you to infinitely expand the complexity of the system without worry—it just works.

Another key area was using parameter collections effectively.<sup>9</sup> In Unreal Engine, it is possible to change the material of something while the game is running. Changing a balloon from red to blue and from shiny to dull is an example of this. However, if you

---

<sup>9</sup> (Epic Games 2018)

wanted to change all of your objects in a scene from one color to another, you would have to specifically code the action of changing each unique shaders color. Unreal has a special feature called Parameter Collection that turns this into one action. If you use a Parameter from a Collection in a material, you only have to change that parameter in one place and that will be propagated throughout the other shaders that use the same parameter.

Context-sensitive shaders are also important to this project. For example, the ground materials of each world change based on how vertical they are. If some polygons pass a certain angle threshold, the texture will change from grass to rock, creating the illusion of a cliff or mountain. This effect is mostly the result of a dot-product from a never-changing vector (in this case pointing straight down the vertical) compared to the vector from each polygons surface normal on an object. This same principle was used to mask off the snow texture when it appears, as snow doesn't stick to vertical surfaces.

My shaders are made up of chunks, inspired by object oriented programming, called material functions.<sup>10</sup> These are small networks of shaders that often have some inputs and outputs. They let you copy large sections of code across materials very quickly. Some of them include a snow material transition, a wetness effect, and a wind vertex animator. After the unique features of the shader are put together, these chunks are added to the final result, effectively seating them in the weather system (figure 2.16). In this figure, the nodes highlighted in blue contain something similar to the code in figure 2.17.

---

<sup>10</sup> (Epic Games 2018)

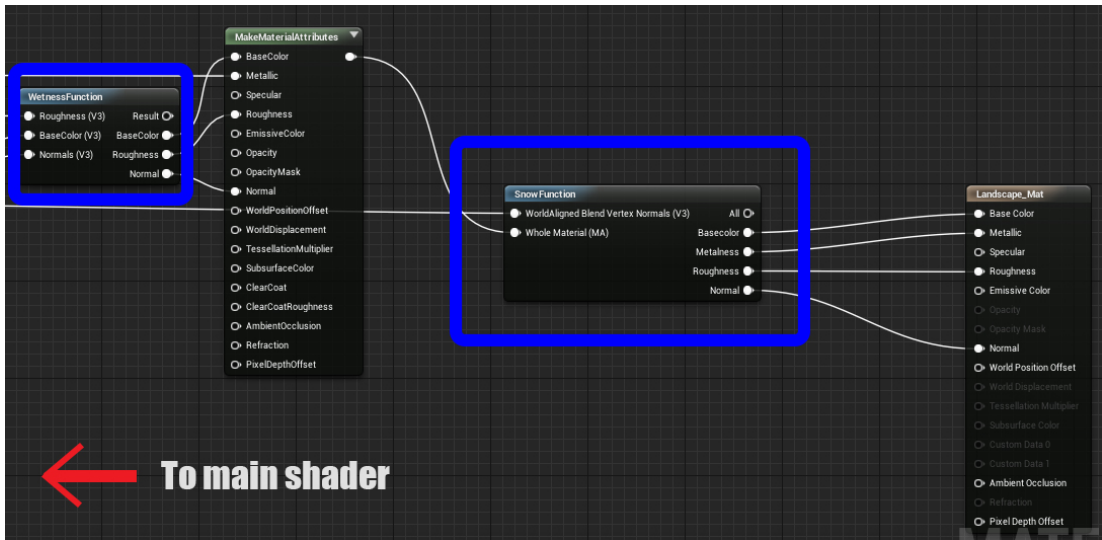


Figure 2.16. Snippet of A Shader Graph Detailing the Material Functions. Unreal Engine Material Blueprint. Zoey Schlemper. 2018.

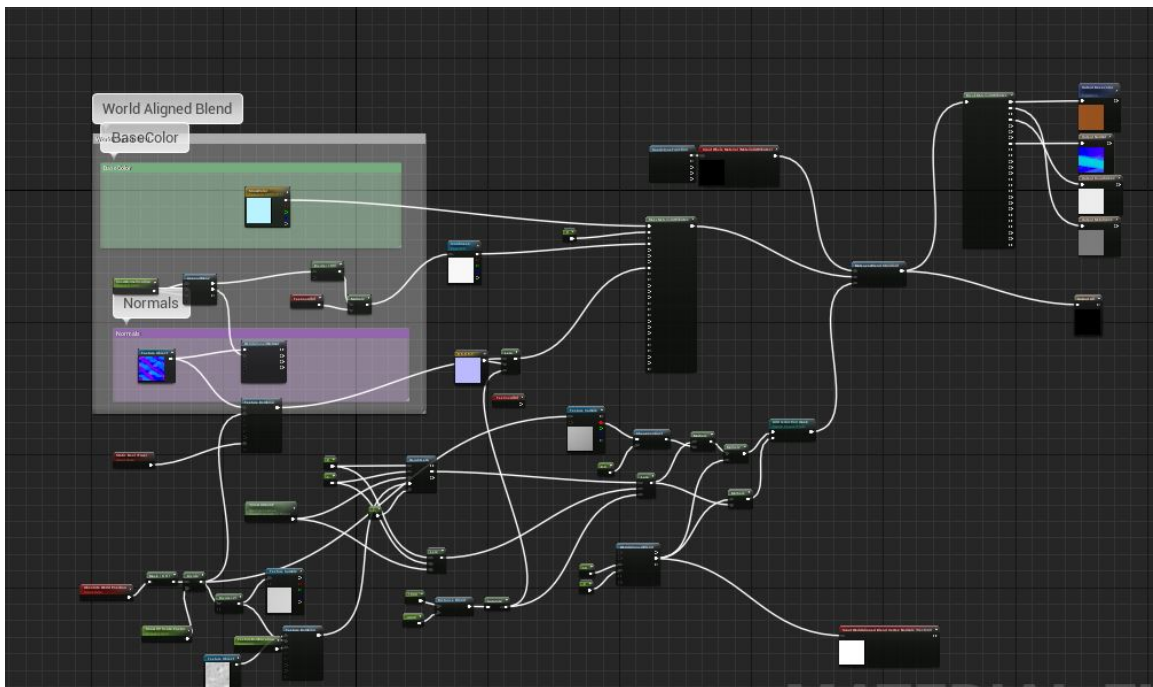


Figure 2.17. Inside the Snow Material Function. Unreal Engine Material Blueprint. Zoey Schlemper. 2018.

There are a number of connections between my approach to shader construction and the end-goal of creating a dynamic weather system. The first is how I utilized the same principles of shape reduction as discussed in chapter 1. It is interesting



that an approach to fire and water can also be applied to clouds/vapor, arguably a combination of the two (really its more like water and temperature/pressure, but I digress). The second is in the emphasis on inter-connection of shader elements, and how they react without human intervention (after being set up the first time). This is paradoxical, because of the immense effort needed to reproduce the phenomenon that can't help but happen in nature. As we have seen, parameter collections are the best emulation Unreal Engine has of the sheer scale of natural physical reactions. Every flat surface collects snow by merely existing in the real world. In contrast, a computer needs to test every vertex that has a shader tailored to display snow before adjusting the shader. It requires immense overhead cost in terms of shader creation and computing power, but the end result is something that we can take for granted, and that we fully expect from past experience. This inter-connection, although wildly different in terms of how it comes to be, is one of the most important elements of a dynamic weather system that strives to pay homage to the real thing.

### **UX Design Process for Menu Systems**

The tutelage of Jennifer Ash was priceless when developing the menu system, as I had little prior knowledge of UX design. The user experience design was fueled by two imperatives: minimalism and control. The information architecture separates the program into two main areas: the World and the Weather (in other words, the stage and the actor). The idea is to keep the controls close at hand, but to make them invisible otherwise. I go into detail about the User Experience design in chapter four.

## Chapter 2.5: Anatomy of the Weather System

This weather system consists of a few different parts that rely on each other in different ways. The vast majority of the weather system is contained in a single blueprint, which is like a folder that contains multiple files and instructions for the computer to use. In this section, I will detail the structural hierarchy and inventory the individual components that make up the system.

First, I will list the weather system assets in their entirety:

- Weather System Blueprint:
  - Sky Shaders:
    - Cloud shader
    - Overcast shader
    - Sky and sun shader
  - Sky Meshes:
    - Sky dome
    - Overcast dome
    - Cloud ring 1
    - Cloud ring 2
  - Directional Light (Dynamic)
    - Cloud shadow effect light function
  - Particle Systems:
    - Rain
    - Snow
    - Dust cloud / low fog
    - Spirit motes
  - Color Curves:
    - Cloud color based on time of day
    - Sun color based on time of day
    - Sky color based on time of day
- Shader Functions:
  - Snow on flat surfaces
  - Wetness / rain on flat surfaces
  - Raindrip screenspace post process effect
  - Wind animation with compass direction
  - Heatwave distortion post process effect

- Parameter Collections:
  - Weather intensity collection (Wind amount, rain amount, snow amount)
- User Interface Blueprints:
  - Heads Up Display
  - Main Menu System

The general relationship is as follows. These assets interact with each other and update via code inside the blueprint. Shader functions cannot exist within the blueprint, because they need to be applied to the 3D objects in the scene that the blueprint interacts with. The parameter collection allows the computer to update all those shader functions at once by adjusting one variable as opposed to adjusting it at each location that variable is used. The user interface (UI) blueprints communicate with the weather system blueprints by both giving and receiving data. The UI adjusts its sliders and options based off the current value of key variables from the weather system, and then sends new data for those same variables when a user interacts with the UI.

## Chapter Three: Coding Key Features of the Weather System

### The XML parser: How Weather Data is collected from the Internet

One unexpected area of difficulty I ran into was coding the weather system. I needed to develop a series of tools and functionality that did the following:

- Download a weather website's content as a string
- Parse the string for a specific set of words and variables
- Compare this parsed list to an exhaustive list and return hits
- Adjust the shader networks and particle systems in order to replicate the weather and time of day based on this data.

During my research for the best way to get weather data, I discovered the National Oceanic Association of America's (NOAA) weather feed<sup>11</sup>, which featured two formats, including XML. This is a standardized format for data that can be simplified to :

```
<DataTag1> Your_data_here </DataTag1>
```

Or: 

```
<FlavorOfIcecream> Mint_ChocolateChip </FlavorOfIcecream>
```

This standardized presentation of weather information was exactly what I needed to make my project work!

---

<sup>11</sup> (National Oceanic and Atmospheric Administration 2018)

The free plug-in *VaRest*<sup>12</sup> solved the problem of requesting the URL and downloading the XML for me. From there, I developed a parsing function that would look for the weather data I needed. I gave it a list, such as “temperature, type of weather, time of day, windspeed, location” and it would find those data tags and copy the contents into an array.

In order to use what I downloaded, I needed a method of translating words into correct numbers for shader manipulation. For example, if the XML parser returned “Heavy Rain” I needed to translate that into “rain = on” and “rain intensity = 2.” The NOAA graciously included an exhaustive list of all possible weather types the XML may include. I then reduced the list to its primary vocabulary. For example, the weather types “heavy rain” and “heavy snow” can be reduced to 3 unique words: “heavy,” “rain,” and “snow.” When I finished the reduction, I had about 20 possible words that may be returned.

I then classified the words based on their impact on the weather. In the previous example, “heavy” is an intensity modifier, whereas “rain” and “snow” are precipitation types. For the computer, heavy means multiply the amount of precipitation by 2, and rain or snow indicates which particle system to turn on. In order to make this translation, I used the “Dictionary” variable type, which makes a 1 to 1 connection from one variable type to another (i.e. I can say that “Heavy” = 2 and “Light” = .5 ).

---

<sup>12</sup> (Alyamkin 2016)



This was especially helpful for 2-dimensional data, such as wind direction. For example, when the parser returns North-East wind direction it can be mapped directly to the vector "1,-1" where "0,-1" means North and "-1,0" means West. You may be wondering why North, traditionally upwards pointing when drawn on paper, is negative on the Y axis. This is because UV coordinates start with (0,0) in the top left corner and end with (1,1) in the bottom right corner. Please consider figure 3.1 to better understand.

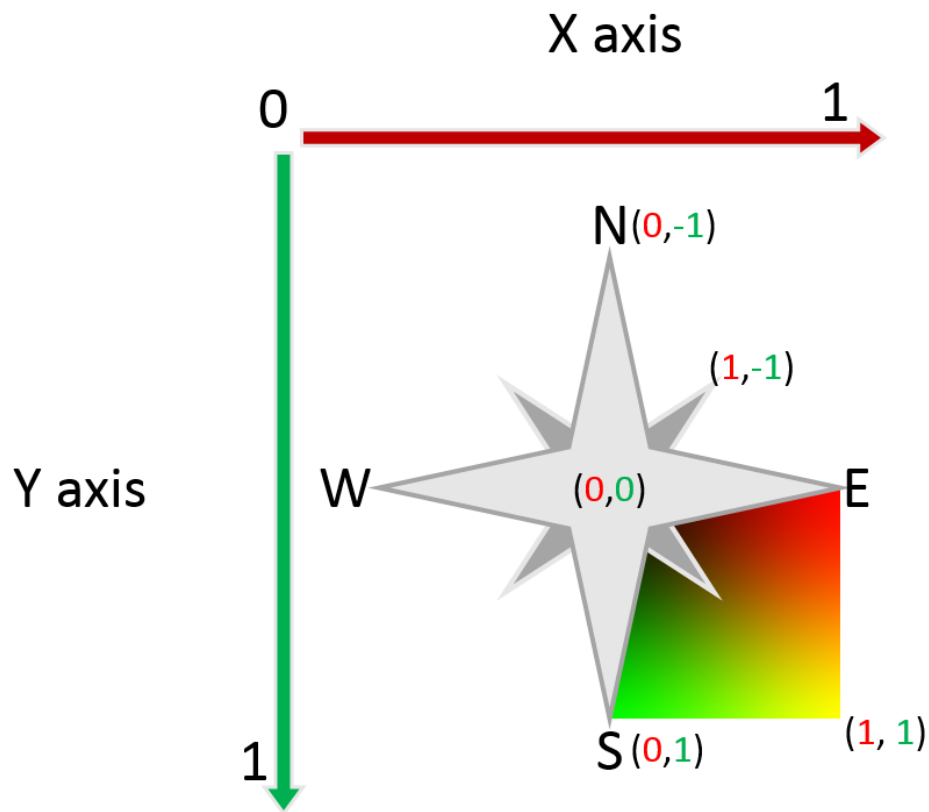


Figure 3.1 Mapping Compass Direction to Vectors in UV Space. Provided by Author. 2018.

## One-Way Data Manipulation in Unreal Engine 4

At this point, the coding only supported one website that was hard-coded. I then had to develop a user-friendly system for choosing one website from a list of hundreds (this would be which city and state you want to pull your weather data from). The number of weather stations was well over 500, and there was no way I was going to write down each weather station code one by one. I had to automate this process for both my sanity and to save precious time. There were a few approaches that came to mind directly: Use the string parsing module in Unreal Engine in order to find the correct codes at run-time; create a separate tool that would compile the list directly from the NOAA sites.<sup>13</sup>

After some research and deliberation with my technical advisor, it became clear that the only viable option was to create a python script. This is because Unreal Engine lacks any reasonable method for manipulating a spreadsheet at runtime. Most of its spreadsheet capabilities are designed for user analytics and data pointers. This means Unreal can record statistics from gameplay and exports it in a spreadsheet, or it can use keywords in spreadsheets in order to associate two things, like an image of a sword with the skill it represents. So although I could import the spreadsheet of all 50,000 airport codes into Unreal, I could not manipulate it any further.

---

<sup>13</sup> (Python for Beginners 2017)

## Structure of the Python Weather Station Code Compiler

At its most basic functionality, the compiler uses one “for” loop nested in another “for” loop. In English, this means the compiler does “Job B” for each “Job A.” Because the NOAA website is for US territories only, “Job A” is to download the web-page that lists all the codes for a single state. So it needs to do Job A about 50 times (plus Cuba and others). Once it downloads a state’s web-page, it then runs Job B, which is to start at the beginning of the webpage and use HTML headers as landmarks in order to pinpoint the Weather Station Code, write it to a text file and insert a Delimiter<sup>14</sup>. It then finds the name of the location (i.e. Seattle-Tacoma Airport, Seattle, WA) corresponding to that Code and writes it to a different text file. It does Job B until it can’t find any more codes. Then, it moves on to the next state and starts Job A again. In the end, two string variables are returned: the list of codes, and the list of locations those codes correspond to. These lists are in the exact same order.<sup>15</sup> This script only needs to run once for the entire program, assuming it runs correctly.

Once these variables were brought into Unreal Engine, I designed a combo box system that auto populates based on user input. To start, a user can only select a state. This is a hard-coded array, and the combo box never changes. Once they pick a state, the program searches the master list of location names and populates the second combo box with only ones that match that state. A user selects the location they want to get weather data from, and Unreal Engine uses the index of that location in order to

---

<sup>14</sup> A delimiter is any characters you use to denote a separation of list elements. So the delimiter in “Apples,+Oranges,+Bananas” is “,+”

<sup>15</sup> (Python for Beginners 2017)

locate the correct weather code. This is why it was so important that they be in the same order.

Finally, the weather code is used to download the weather data by inserting the code into the base URL used by the NOAA database.

This Url is: [http://w1.weather.gov/xml/current\\_obs/ABCD.xml](http://w1.weather.gov/xml/current_obs/ABCD.xml)

Where “ABCD” is the weather code desired. The program replaces ABCD with what the user selected, and collects the weather data using the XML parser described earlier.

### **Can the System adapt to Additional Databases?**

Although this system has been designed specifically around the XML format and the unique URL’s of the NOAA website, it is possible for this system to be expanded to additional databases that include European, Asian, or any other weather stations you can find, with only a little extra work. XML is an extremely common format that is used for all sorts of applications, and the likelihood of finding other weather databases that use it is high. The python compiler would definitely need to be tweaked to fit the set up of other databases, but these changes are small compared to the other work that has gone into the system in its entirety.

## **Chapter Four: User Experience Design**

The menu system (figure 4.1) was created by testing it periodically on users and adjusting course based on their feedback. The largest impact from the user testing

included vocabulary for buttons, and information architecture design. I found that clearly expressing the function of a button to any person without fail is extremely difficult, and that creating proper user flows requires a lot of iteration.

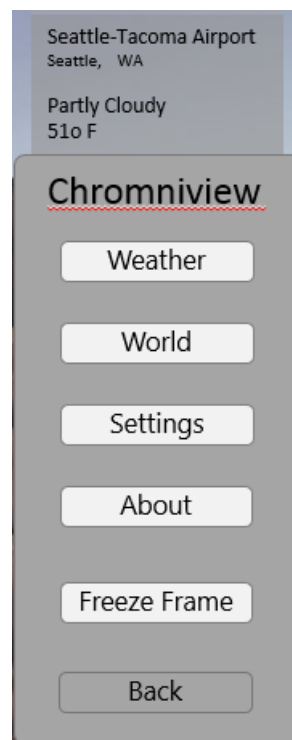
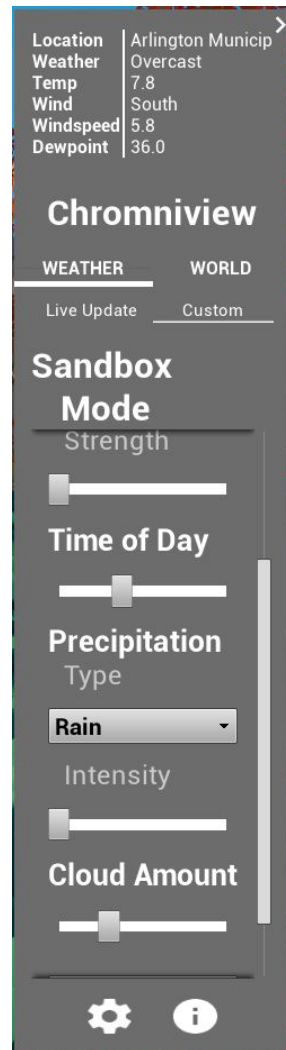


Figure 4.1. (Left) Chromnview Menu Design. Zoey Schlemper. User Interface. 2018.

Figure 4.2. (Right) Chromnview Menu Design (Early Draft). Zoey Schlemper. User Interface. 2018.

After a round of A/B Testing, it was discovered that 5/5 users preferred Tab-based menu system<sup>16</sup> (fig 4.1) over a basic buttons layout (fig 4.2). In the Tabs system,

<sup>16</sup> (Schlemper, Chromnview Prototype: User Testing 2018)



all menus are 2 or less actions away at any time. This brevity allows a user to remember what they are doing and to perform their actions quickly. Second, the ever-present tabs indicate the informational hierarchy at all times, giving the user a clear understanding of how the system works.

This system also helps to enact the thesis statement by reducing user focus on the UI. While a necessary part of the experience, the UI is only a tool to experience nature via the dynamic weather system. The better designed this menu is, the less it matters to the user, ensuring attention is paid to the artistic statement.

### **Designing Based on Established Methods**

In terms of Chromnview, the menu design is a liaison between the message and the user. It is not something with which I ever intended to experiment, create an artistic statement, or otherwise influence the trajectory of the thesis statement in a direct way. Instead, it is intended to bridge the gap between the weather system and the user. It helps to create a meaningful experience by giving the user control, but it isn't meant to add any "flavor," if I am to continue the metaphor from Chapter One. For these reasons, directly referencing established methods of UX design were my primary method for creating the menu system. I pulled heavily from the *Material* design philosophy researched by Google.<sup>17</sup>

Much of *Material's* design focuses on consistency, clarity, and minimalism. It is no surprise that the style fit into Chromnview's tenants of minimization and ease of

---

<sup>17</sup> (Google 2018)

use. The functionality of *Material* is typified by “Components,” which are essentially UX-centric Archetypes: The Button, The Expansion Panel, and Dialogs are just a few of these. These Components are organized through strict Layout guidelines. Margin and spacing amount, relativity of text size, and placement on the screen have been given consistent roles for relaying information to a user. This reduces the opportunity to be creative, but it also enforces the importance of a positive and well-designed user experience.

The UX design process was very fluid. It seemed like I was constantly changing direction, and “going back to the drawing board,” as they say. I have been told that this is the nature of UX design. If my process can be boiled down to a few steps, they were: brainstorming the desired experience, enumerating needed functionality, paper prototype (fig 4.3), user testing, revision of paper prototype, user testing, digital prototype (fig 4.4), user testing, A/B testing of two menus (digital prototypes), implementation into Unreal Engine, user testing, revision of system, user testing. During the A/B testing is when I discovered the tab-based menu system that is in the final version. This required a hard pivot and reworking front-end systems, but was required based on user testing results as mentioned earlier.



Figure 4.3. Chromnview UX Paper Prototype. Zoey Schlemper. 2018.



Figure 4.4. Chromnview UX Digital Prototype. Zoey Schlemper. Powerpoint Presentation. 2018.

One of the most important lessons I learned while working on UX design was something writers often talk about: killing your babies. Of course they mean metaphorically, and of course they are referring to passages written that one has an

irrational attraction to. Having never been fond enough of my own writing, the words were lost on me. What pain is there in erasing a few sentences?

After spending a month and a half designing and coding the menu system, I had become very attached to it without realizing it. But user testing revealed that there was a better option at hand, and I'd have to pursue it if I were worth my weight. It pained me greatly to erase almost all the front-end work on the project. I was very fortunate to have enough time to handle this setback without having to sacrifice a different area of my academics. Like mentioned before, the project benefited from it.

The UI system has the least amount of polish amongst the various systems that are in play in Chromnview, and this further enhances the artistic statement. To polish or refine the UI is to make it a spectacle, and this is not something that bodes well to maintaining weather and precipitation at center stage. By ensuring the UX was as unintrusive and minimizable as possible, and by spending a minimal amount of time on the refinement of the graphic design of it, I have hopefully kept these buttons in the absolute back of a user's mind.

## Chapter 4.5: Weather Effects Compilation

*Figure 4.5 All images were created and provided by the author. Digital Multimedia. 2017-2018.*

There are a total of 3 environments in Chromnview, featuring a multitude of viewpoints. The first ever made, depicted in example 1, is an abandoned fountain in the desert. While researching the weather and planning my approach, I wanted to make sure that I explored biomes and environments that are stereotypically known for certain weather types. In this way, I started with a hot desert, which has been contrived as being completely devoid of water in all its forms.

The second environment, shown in example 5, and featured in the concept paintings from 1.2.1-4, is a guarded mountain pass. In this environment, I was exploring a possible concept of a wind serpent that enjoyed the mountains for their wind-tunnel like valleys. In this world, the locals might've developed a religion around such a majestic beast. I also wanted to avoid cultural misappropriation, and attempted to combine some design sensibilities from both Eastern and Western cultures. In the end, I didn't have enough time to model the intricate designs that went with the wind-serpent religion, but I was able to begin combining Western Feudalism with Eastern Shintoism, in the turrets and spirit golems respectively.

Finally, the ancient arcane scale included as the 2<sup>nd</sup> in example 3, came about through a simple desire to create a mechanical prop. From that simple prompt, I began designing a pseudo steam-punk gear system that uses ancient magicks and integrates actual gameplay elements, like staircases and levers. This environment may indeed exist

in the same world as the guarded mountain pass, but the styles vary enough that they may not.



## Examples of Weather Changes

### *Example 1: Sunny to Overcast*

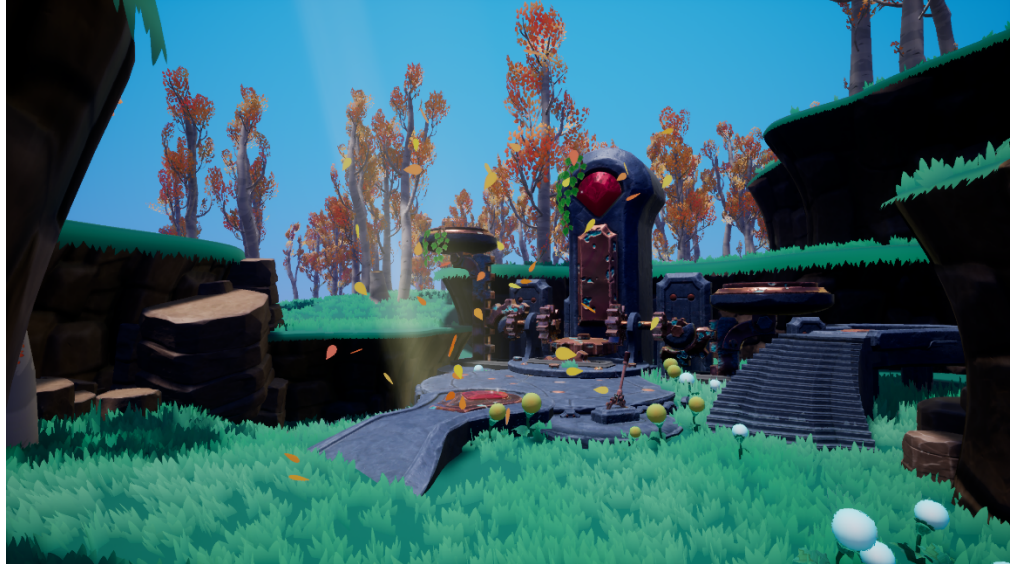


*Example 2: Dawn to Mid-Day*



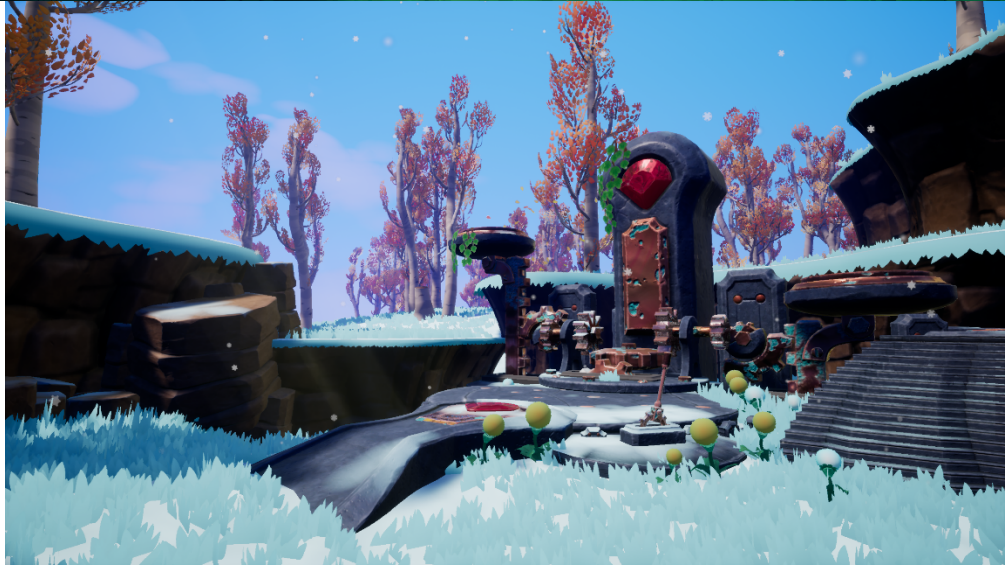
*Example 3.1 and 3.2: Mid-Day to Dusk*



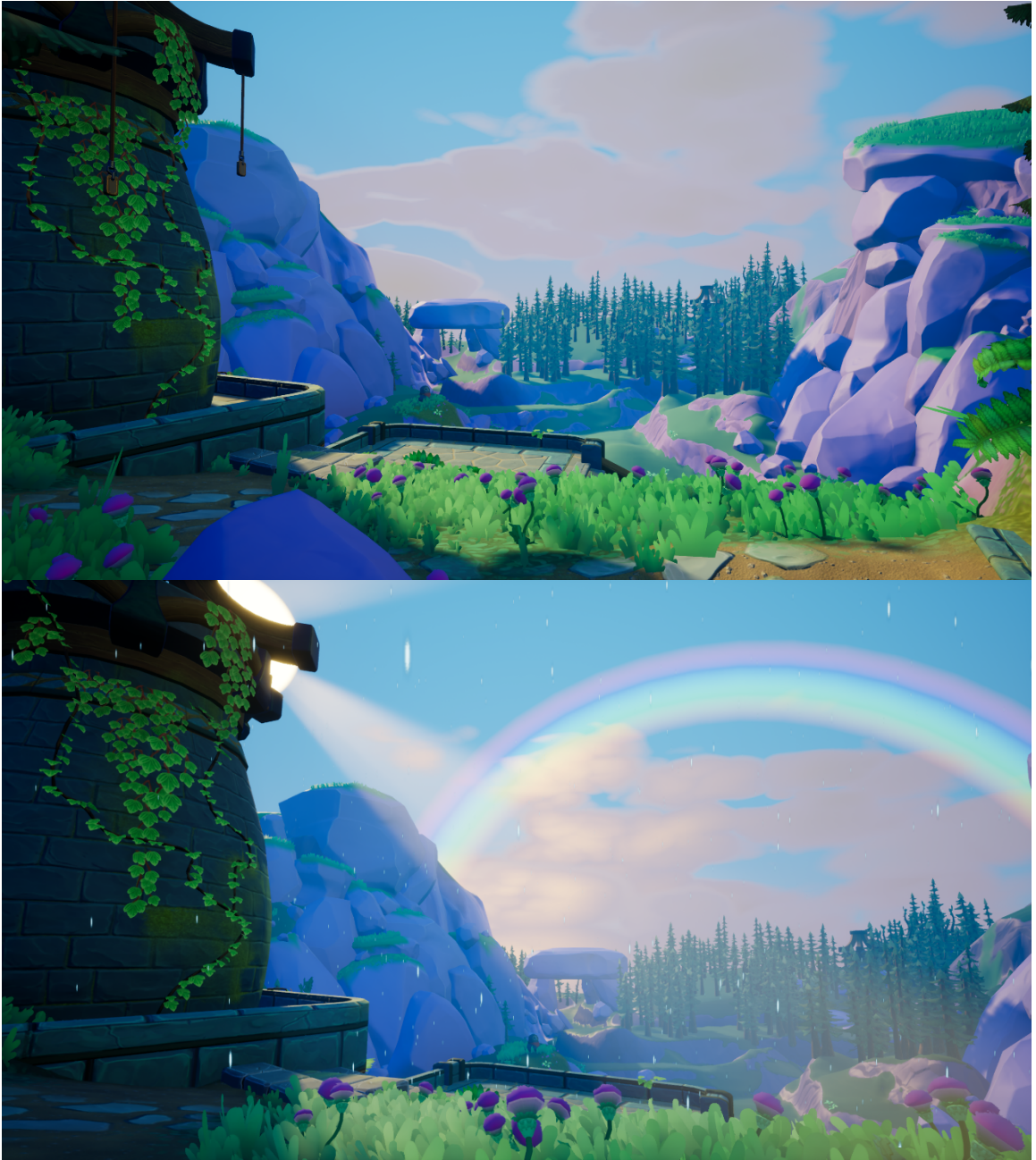




*Example 4: Clear to Snowy*



*Example 5: Clear to Rainy*





Example 6: Day to Night



## Chapter Five: Conclusion and Looking Forward

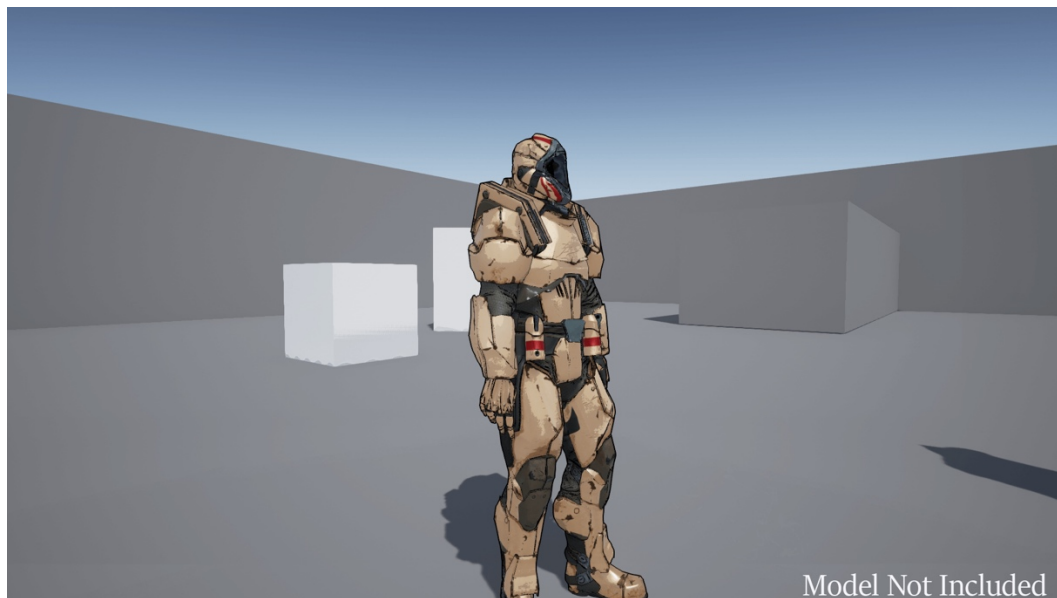
The artistic vision of the thesis was heavily influenced by two dimensional representations of reality and possible methods for translating its inherently graceful simplifications of volume. The choice of color, and the type of stylization in shape language were selected in order to create an homage to nature. The primary goal of artistic direction was to emphasize the beauty of nature, and to bend the human will to serve her needs. This creates an inverted influence pattern by creating an artificial reflection of natural consequence, giving humanity the opportunity to decide whether it is worth caring about or not.

In creating this application, the artist posits that nature creates meaning by influencing humanity, and we benefit from this only as much as we understand the connection between nature's power and our personal lives. The application and all its simulations mean nothing except for what they inform the user about the real world. The meaning of nature, arbitrary when experienced directly and in-person, is suddenly made conditional on the grounds that a person finds a reason to care about this reflection.

I utilized all sorts of techniques, from cutting-edge programs like Substance Designer for creating procedural textures to modified versions of established UV distortion techniques in order to follow the art direction. I made sure to create shaders and gameplay logic that allows the art to respond and emulate the many states of

nature. I used these techniques to add weathering to man-made objects and to increase the sense of how weather influences static objects.

The user interface also followed the art direction, keeping nature the center of attention. The menus encourage exploration of the system while also letting nature control it for you, achieving a meeting between the will of humanity and the chaos of nature, utilizing modern technology as the liaison.

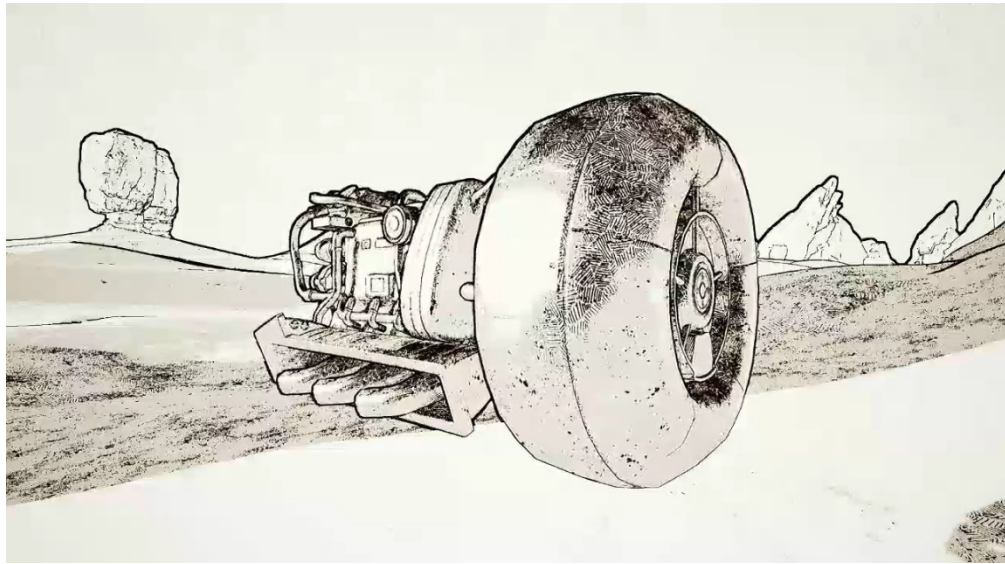


*Figure 5.1 Celshading And Outline Material. PipeRift. Digital. 2016.*

However, the lack of post-processing effects such as an outline shader (fig 5.1) and shadow manipulation (5.2) leave room for a more refined emulation of 2D art. These types of effects were popularized in the west by games like *Borderlands* (fig 5.3), and eastern examples of the effect include recent entries in the *Guilty Gear* franchise (fig 5.4) and *Dragonball* franchise (5.5).



*Figure 5.2. Shadow Material Shader. Tom Looman. Digital. 2017.*



*Figure 5.3. Borderlands 3 Tech Demo. Gearbox Software. Digital. 2017.*

The implications of this project are many. First, it is an adaptable, fully procedural weather system, library of material functions, asset creation workflow, and supporting custom-made Python script that results in a full technical environment art suite for stylized 3D environments with miniscule fiscal budget. It could be picked up by



a large variety of studios and adapted to their workflow thanks to its compartmentalized design. Second, any weather recording system in the real world that converts its data to XML can be utilized by this system with a little adaptation. This allows for scalability as the technology and resources of a studio change or increase.



Figure 5.4. Guilty Gear XRD. Arc System Works. Digital. 2016.



*Figure 5.5. DragonBall Xenoverse 2. Bandai Namco. Digital. 2017.*

Finally, the project has only had one full iteration by a single artist, and it could be expanded in various ways. Some of these include: Geometry offset for snow system; Full wind system for physics based objects; 3D emulation of light in cloud shader; and many more. Rainfall could fill objects with water, allowing a system for puzzles based on water levels to be implemented. The snow could freeze these puddles and turn them into platforms, further increasing gameplay opportunities. This project shows a proof of concept for creating a procedural weather system that can be driven by real time data in order to generate narrative and other effects.

### **Possible Next Steps**

There are a number of weaknesses in the body of work known as “Chromnview.” They range from artistic direction to scope and subject matter. These



critiques are opinions by the author about his own work, and one might agree or disagree with them. They are necessary, for an artist's work is never finished. They are not discouraging, but stern and confident as they offer new avenues of exploration. In pursuing these clashing ideas one might awaken new paradigms for judging value, and eventually break ground for the bright-eyed hopefulness that urges one to create.

The first critique is on artistic direction. The stylized design sense can be considered half-baked. The high concept of the piece aimed at a more loyal interpretation of 2 dimensional animation, including visible outlines and a cel-shaded look that utilized cross hatching patterns. Scope and efficiency are the primary reasons the style has not pursued these tenants. Rather than spend time building static shaders like an outliner, the artist aimed to build the weather system and refine it as much as possible. Furthermore, celshaded and outliner techniques force a forward rendering process, while Unreal Engine uses deferred rendering by default. This would cause noticeable costs to memory and rendering budgets. Since the final concept of this work is to be easily accessible, efficiency is key. In the end, there was not enough time to flesh out the style further or to ensure that the overhead of those style-specific shaders could be minimized properly.

The second critique is on the medium used for creating the project. Unreal Engine was initially chosen due to its emphasis on visual scripting environments and high quality and simple lighting and rendering. These were great boons during the development of the project, and arguably were necessary for completing the weather

system. The projects I have coded in the conventional text editor environment pale in comparison to the complexity of the Unreal Blueprints in this project, suggesting that I could not have coded this effectively in a text editor.

However, the use of such a robust engine brought with it so much opportunity to create all sorts of things, such as particle systems, complex shaders, multiple levels, character controls, and user interface elements, to name a few. The wide variety of objects the artist created meant less time refining the individual elements. Had the entire project been constructed in Wallpaper engine using only shader code and a few 3D objects, there would've been far more time to refine individual pieces of the project. Not only that, but the artist would've been exposed to lower level code libraries, perhaps offering more educational value in terms of technical art.

The implications of this include a weaker final portfolio, because employers often look for quality over quantity. The artist became distracted with the sheer amount of options for adding to the scope of the project in an engine like Unreal that the project suffered for its added complexity. On the other hand, perhaps the use of Unreal Engine resulted in a stronger portfolio, because the visual scripting environments lowered the bar of entry into game and art programming, allowing the artist to explore more complex code structures faster than a text editing environment.

Had the artist created primarily in Wallpaper Engine, he might've made a small diorama and had to learn complex libraries for things like shaders, particle systems, and JSON requests in a Javascript-based game engine. This certainly would've been much

more difficult than working in Unreal Engine. This would've shifted the project farther away from "Environment Artist" and closer to "Graphics Programmer" and "Technical Artist."

One feature of this executable that was pitched during the inception of the project was to display the entire experience on the desktop of an average computer, behind the icons. This helped to align the project with the initial goal of the thesis to enshrine nature, and to create a void for her to fill. The importance of this feature to the final delivery of the project has been low since the initial pitch, as can be seen in figures 5.6 and 5.7, which show the most important slides. While this feature was not pursued, it seems to garner interest and spark the imaginations of many who hear of it. The idea is worth hanging onto. Stills from the mock-up (also from the initial pitch) can be seen in figure 5.8.

While there are other critiques to be made about the project, these are the ones that clung to the author's mind most persistently. In reading this, I hope you learned something, and I hope you make something great.

Thank you!

---

## Thesis Statement

Create a narrative experience using a 3D environment and UI overlay that reacts to weather, time, and notification information from internet sources in real-time.



---

*Figure 5.6. Thesis Statement Slide from official thesis pitch. Slideshow. Zoey Schlemper. April 2017.*

## Narrative through Extension of Real Life

The natural elements and environmental effects are the actors.

It is the endless unfolding of nature.

A void for real life to fill.

---

*Figure 5.7. Key points concerning the enshrinment of nature set out by the Artist. Slideshow. Zoey Schlemper. April 2017.*



Mock Up



Mock Up



Mock Up

Figure 5.8. Mock Up of Desktop Application. Photoshop Animation. Zoey Schlemper. 2017.

## References

Alyamkin, Vladimir. 2016. *Va-Rest: Github Repository*. Accessed 2017.

<https://github.com/ufna/VaRest>.

Authors, Multiple Unnamed. 2018. *Foliage*. March 8. Accessed 2018.

<http://wiki.polycount.com/wiki/Foliage>.

Epic Games. 2018. *Material Functions*. Accessed 2018.

<https://docs.unrealengine.com/en-us/Engine/Rendering/Materials/Functions>.

—. 2018. *Material Parameter Collections*. Accessed 2018.

<https://docs.unrealengine.com/en->

[us/Engine/Rendering/Materials/ParameterCollections](https://docs.unrealengine.com/en-us/Engine/Rendering/Materials/ParameterCollections).

Google. 2018. *Material Design*. <https://material.io/guidelines/#introduction-goals>.

National Oceanic and Atmospheric Administration. 2018. *National Weather Service: XML*

*Feeds of Current Weather Conditions*. Accessed 2018.

[http://w1.weather.gov/xml/current\\_obs/seek.php?state=wa&Find=Find](http://w1.weather.gov/xml/current_obs/seek.php?state=wa&Find=Find).

Python for Beginners. 2017. *Reading and Writing Files in Python*. Accessed 2018.

<http://www.pythonforbeginners.com/files/reading-and-writing-files-in-python>.



—. 2017. *String Manipulation*. Accessed 2018.

<http://www.pythonforbeginners.com/basics/string-manipulation-in-python>.

Schlemper, Zoey Norman. 2017. "2D Procedural Sky Shader: Using Unreal Engine 4.16 Material Editor." Game Art Documentation.

Schlemper, Zoey Norman. 2018. "Chromniview Prototype: User Testing." User Experience Test, Redmond.

Schreibt, Simon. 2017. *Stylized VFX in RiME*. Accessed 2017.

<https://simonschreibt.de/gat/stylized-vfx-in-rime/>.

Squirle Art. 2017. *Normal Map Generation*. ". February 26. Accessed 2017.

<https://squircleart.github.io/shading/normal-map-generation.html>.

## Appendix A

*Additional Resources Written by the Author*



# 2D Procedural Sky Shader

---

*Using Unreal Engine 4.16 Material Editor*

*By Zoey Schlemper*

Professors Matt Brunner and Mark Henne | Digipen Institute of Technology | CG 598 |  
December 3, 2017

## Contents

Introduction.....	83
Prerequisites.....	83
Software Information .....	84
Resources for Learning the Techniques Mentioned in the Tutorials.....	85
Creating Cloud Textures in Substance Designer and Exporting to Unreal.....	85
Expose Parameters in Substance Designer.....	89
Exporting a Substance for Unreal .....	95
The Substance Plug In for Unreal Engine.....	96
Installation .....	96
The Plug In Explained.....	97
The Parameter Window.....	99
Essential Nodes in Unreal Material Editor .....	100
Add and Subtract: the Value Shifters.....	100
Multiply: The Mask Node.....	101
Clamp: Keeping values under control.....	102
Simple, Yet Powerful.....	104
The Core Node Network .....	104
Section one: The Sun .....	105
Section Two: The Clouds.....	106
On your Own.....	108
Learning Resources.....	108
UV Warping A Red and Green Texture Map.....	109
Packing Textures to RGB in Photoshop.....	110

## Introduction

**Please read the manual before watching the videos. To acquire the videos, please contact the author.**

This Tutorial Video Series Describes the creation process of making a sky shader. Rather than doing a node-by-node follow along, I elected to walk through the finished materials and focus on describing why the nodes work. I do some step by step nodes for the more difficult sections, but for the most part I assume you will be able to make your own version of each node I talk about.

I employed a wide variety of techniques to create this shader. The resources where I learned the skills not taught in the Video are included in this manual.

## Prerequisites

Before watching the videos, make sure you understand the following. If you don't know how to do it, then refer to the page indicated.

- **Basic use of Unreal Editor 4.16+ Material Blueprint Editor** (Essential Nodes in Unreal Material Editor, Pg. 17)
- **Using UV Warping on RGB Packed Textures** (UV Warping a Red and Green Texture Map Pg. 26)
- **Creating Packed RGB Textures in Photoshop** (pg. 27)
- **Substance Designer Skills in order to create simple cloud textures** (Creating Clouds in Substance Designer, Pg. 3)

## **Software Information**

Unreal Editor 4.16.3

Substance Designer 6

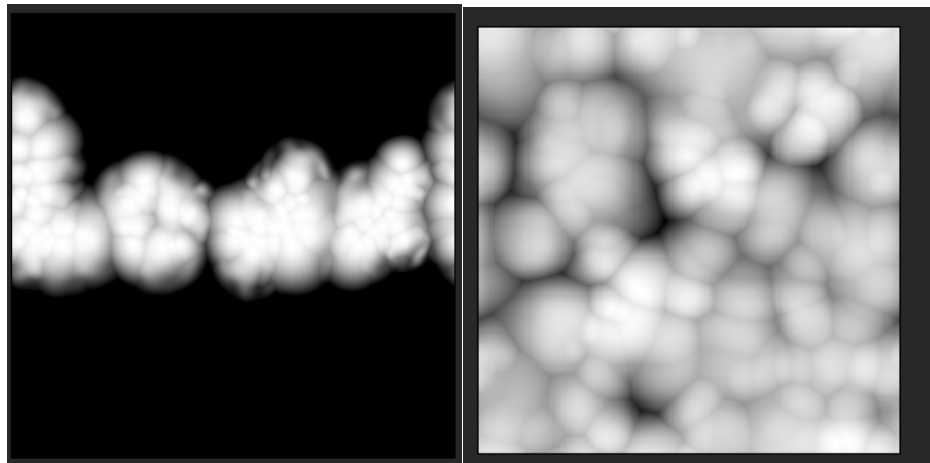
## Resources and How-To's

Some segments of the shader graphs use more complex techniques that I learned from Professionals. Rather than waste your time with my sub-par descriptions of how these work, I will direct you to the official sources of this information.

Other skills that I did have confidence in are described in full here.

### Creating Cloud Textures in Substance Designer and Exporting to Unreal

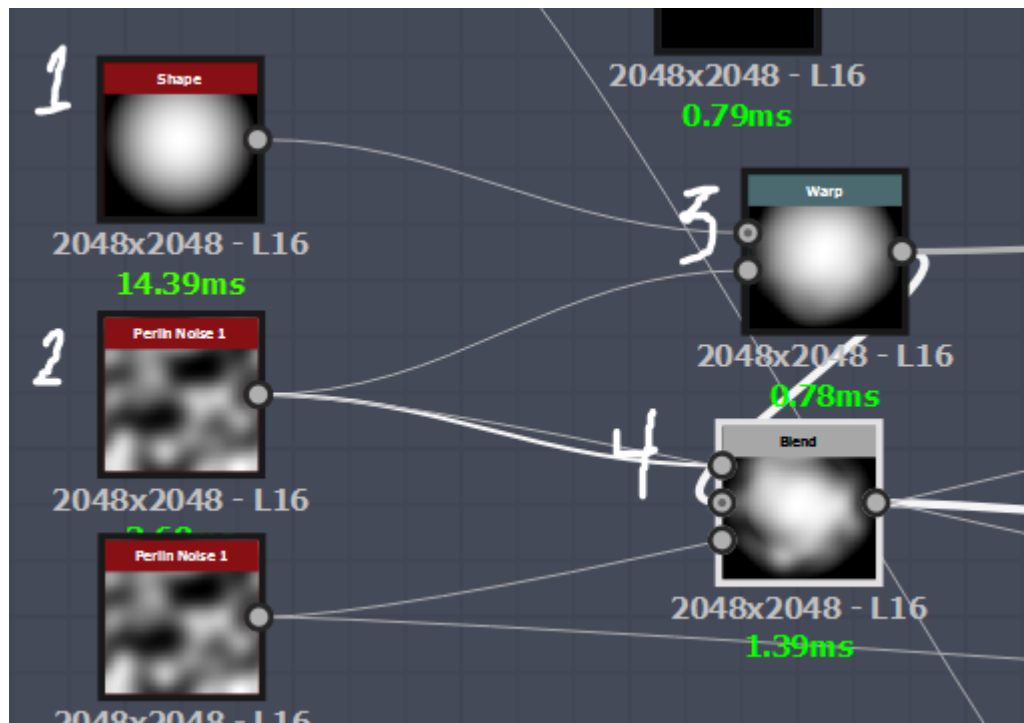
My design for the clouds utilizes textures with two different roles: A *Stencil* type, which creates the silhouette of the clouds, and a *Fill* type that mixes and creates variation within those bounds. They are pictured in **Figure 1.2**





**Figure 1.2** The “Stencil” (Left) and the “Fill” (Right). Notice that the stencil is in the top half of its image space. This is very important, since we are mapping to an entire sphere, and the horizon ends at that middle point.

Before we get into Unreal Engine at all, we need to have some basic cloud textures to work with. Looking at my reference images, I decide to make my clouds out of large puffball shapes. See **Figure 1.3**



**Figure 1.3** Snippet of Cloud Substance

1. Starting with a Paraboloid **Shape** node, I use...
2. **Perlin Noise** to...
3. **Warp** the shape...

4. And **Blend** both together to add more lumps.

After that, I use the **Tile Sampler**, see **Figure 1.4**.

This was the base for all my cloud textures. They can definitely use some fine tuning in order to get more personality, but they are sufficient.

From here, I apply a mask to the **Tile Sampler** for my stencil shape. I used a **Waveform** node **blended** with a **Transformed Rectangle Shape** as a mask in the **Tile Sampler**. See

**Figure 1.5**.

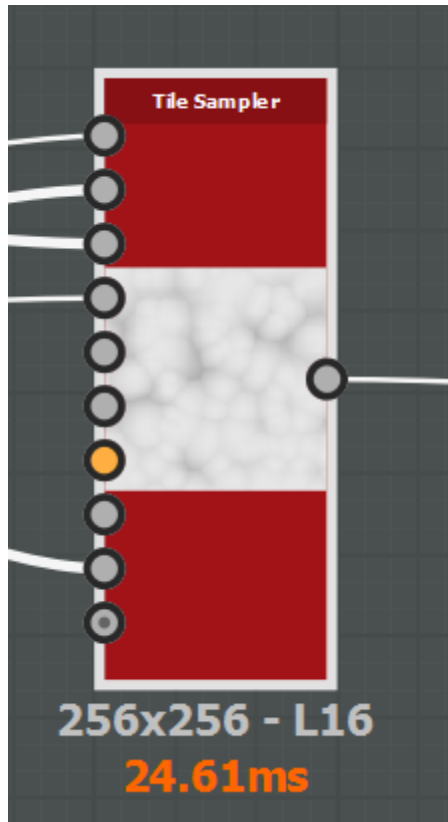
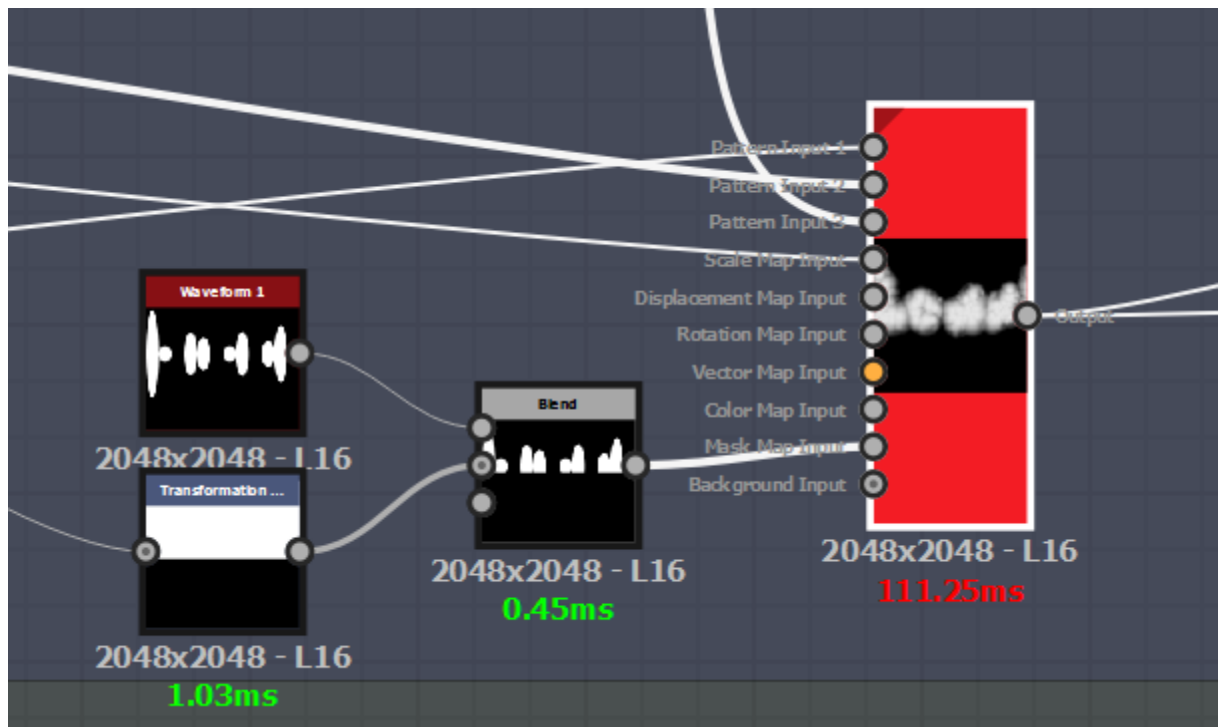


Figure 1.4 The Tile Sampler.



**Figure 1.5** The mask used for the “Stencil” version of the Tile Sampler.

I don’t go into detail of how exactly I made these files because I want you to experiment and try it out for yourself! Have fun in Substance Designer!

### Exposing Parameters for Use in Unreal Engine 4

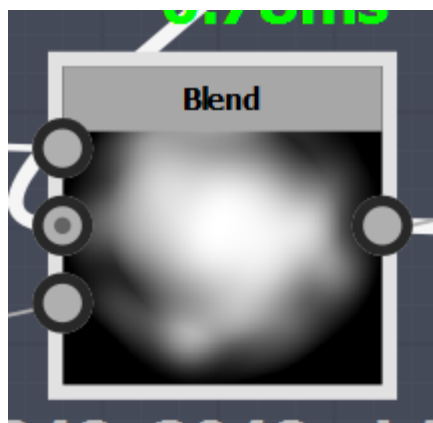
Now you should have at least 1 “stencil” type and 1 “fill” type cloud texture. I could easily export these bitmaps and use them in Unreal Engine, but I would have to re-open Substance Designer and re-export new images if I wanted to make any change, no matter how small.. Instead, I am going to expose some parameters. This way, I save time and simplify my workflow.

### Expose Parameters in Substance Designer

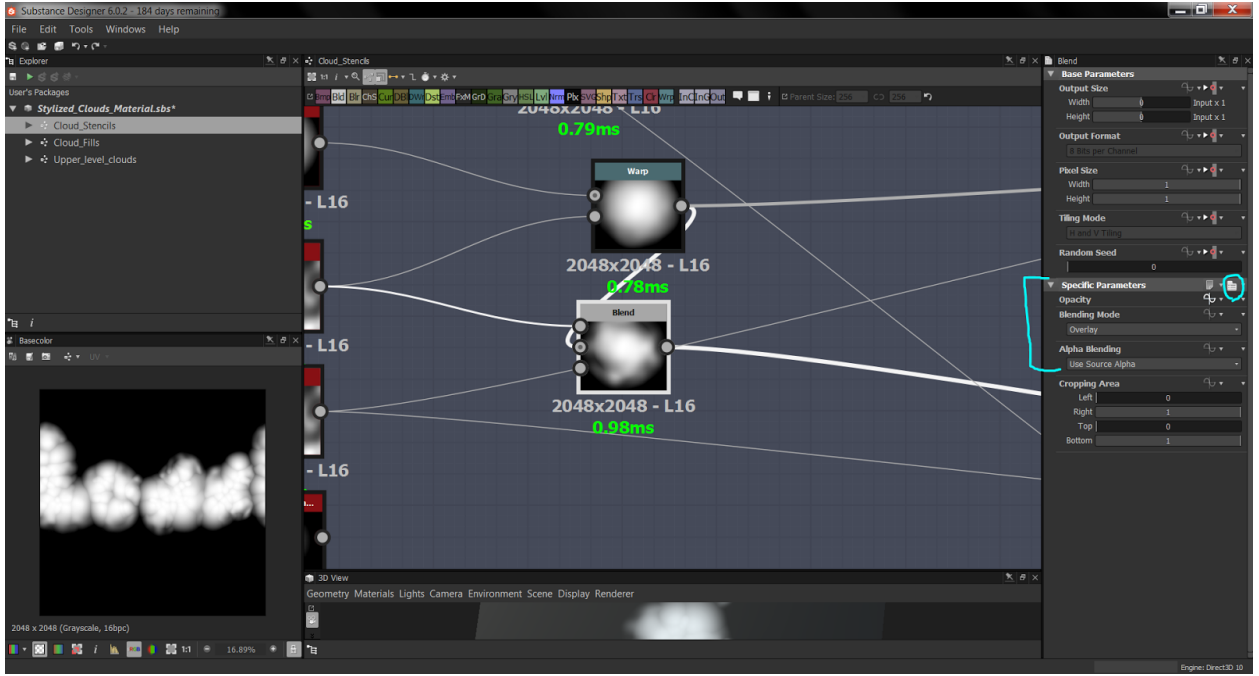
In your Substance Graph, pick a parameter that gives you some type of control over your texture output. In this example, I am going to expose the Opacity Slider of my

**Blend Node from Figure 1.3.**

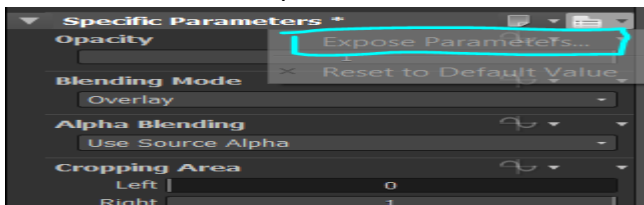
1. Select the node :



2. Navigate to the Specific Parameters section, usually on the right side of the UI. Click the button in the top right of that section, circled in cyan.



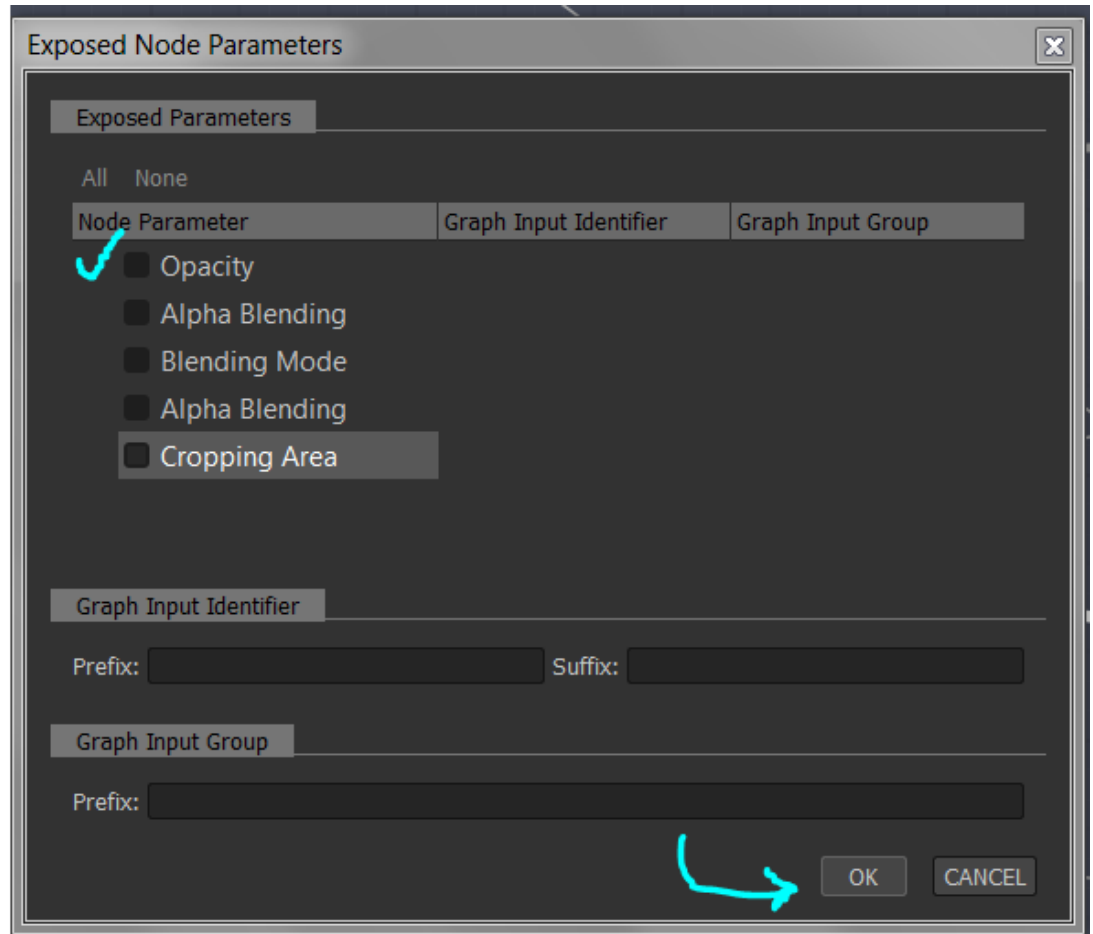
3. Then click "Expose Parameters"



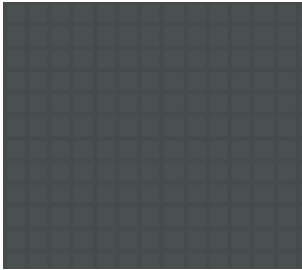




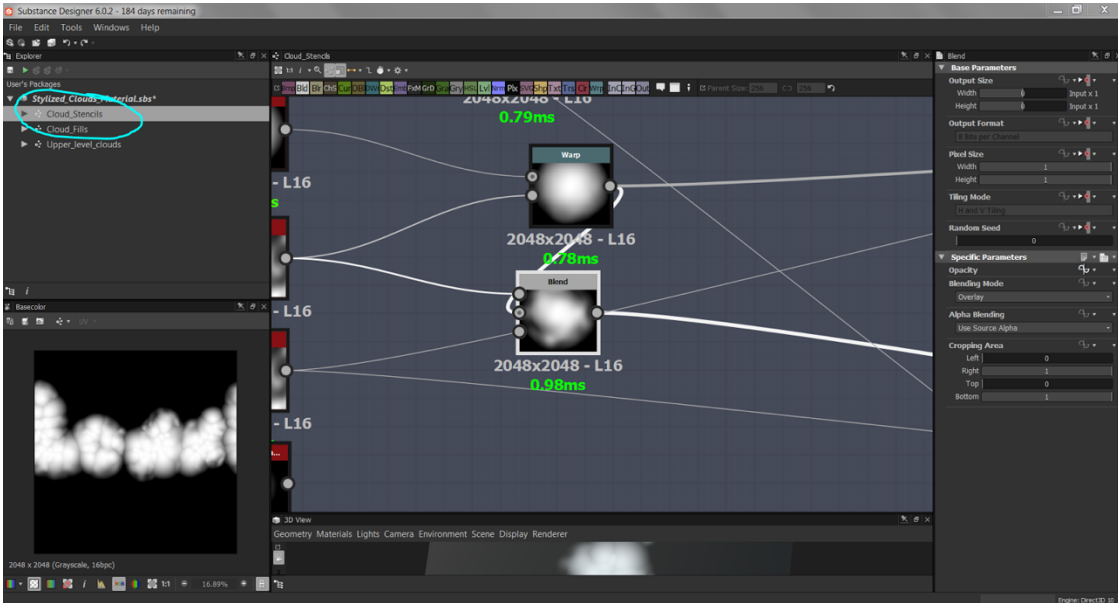
4. This window will appear. You now are able to toggle the parameters you'd like to expose. Click "OK."



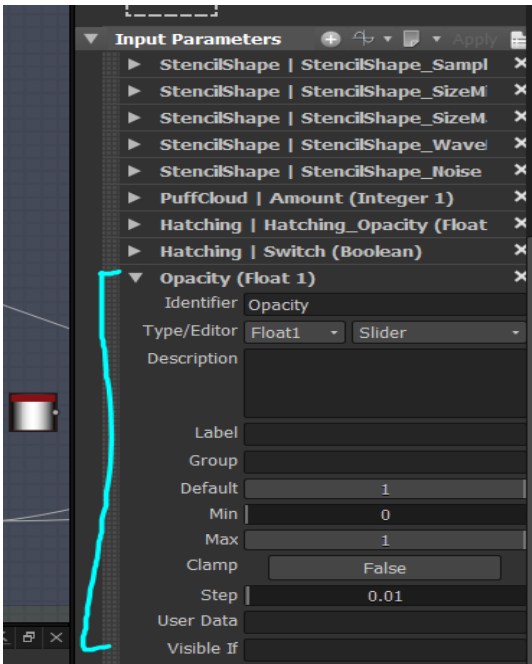
5. Double click on the background of your substance graph OR your graph's name



OR



6. Then, in the rightmost pane, scroll down to "Input Parameters." Here it is on the top level of our Graph!

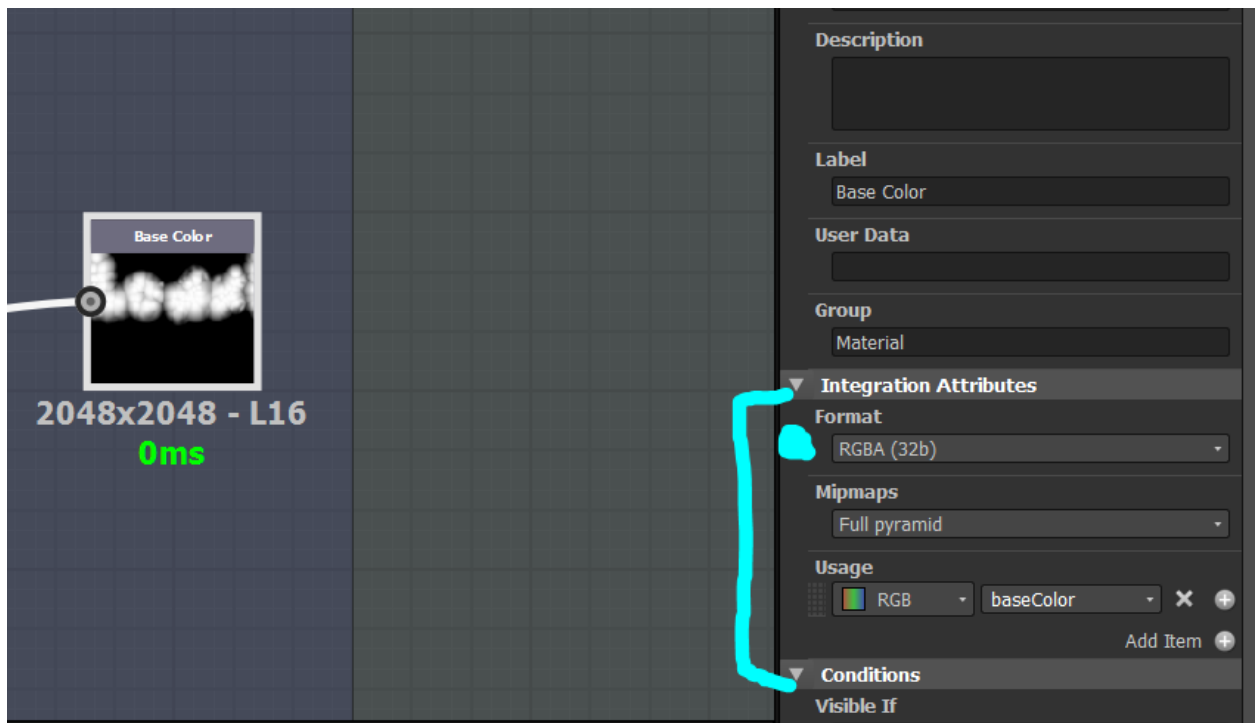


We are almost ready to export. Before we do, we want to make sure we have the right texture format selected.

It is common knowledge that you will use an image like a .jpeg , .tga, or .png when making textures for games. However, how you save that file and how it is converted by the engine will change the quality of it, sometimes drastically. For this texture, we are using primarily Grayscale images. When exporting from an authoring software like Substance Designer, we want one of the following:

- A 32 bit image (RGB)
- An 8 bit single color image (grayscale)

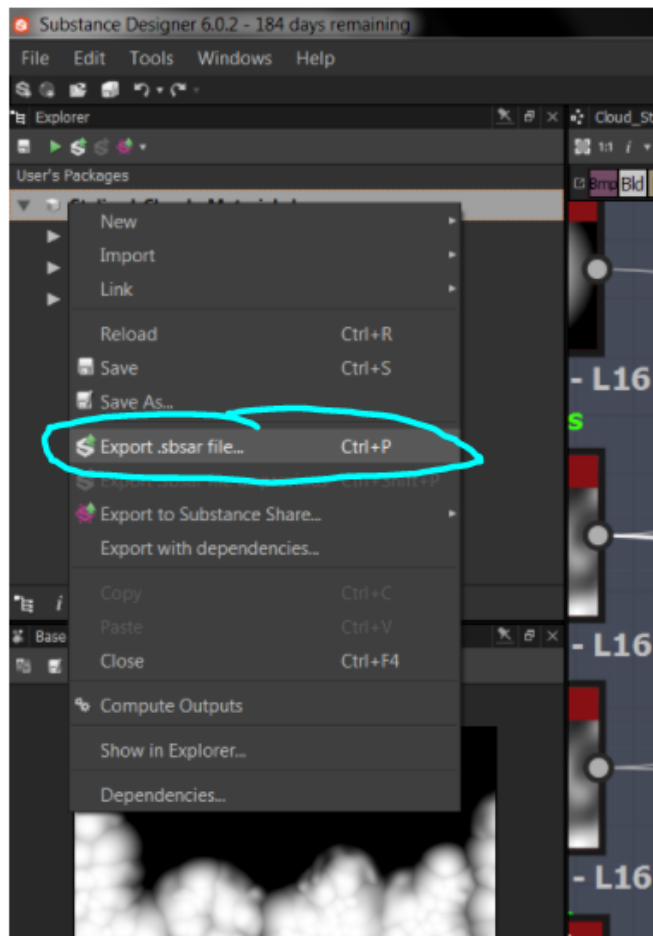
Refer to **Figure 2.0** for how to do this.



**Figure 2.0** When we have our output node selected in Substance Designer, the **Integration Attributes** indicated here shows important options. **Format** lets us select either 8 or 32 bit from a variety of options. For this texture, I used 32 bit.

### Exporting a Substance for Unreal

This is the same export process as prepping for Substance Painter. Once you have all your parameters exposed and the image format selected, simply right click the graph name and select “export SBSAR file” as seen in **Figure 2.1**



**Figure 2.1** The final step for exporting a Substance Designer File for Unreal.

Do the same process for your Cloud “fill” texture, and let’s move into Unreal Engine

4.16!

## The Substance Plug In for Unreal Engine

### Installation

Install the free Substance Plug In from the Unreal Market Place (Figure 4)

To enable it in your project, see Figure 4.1

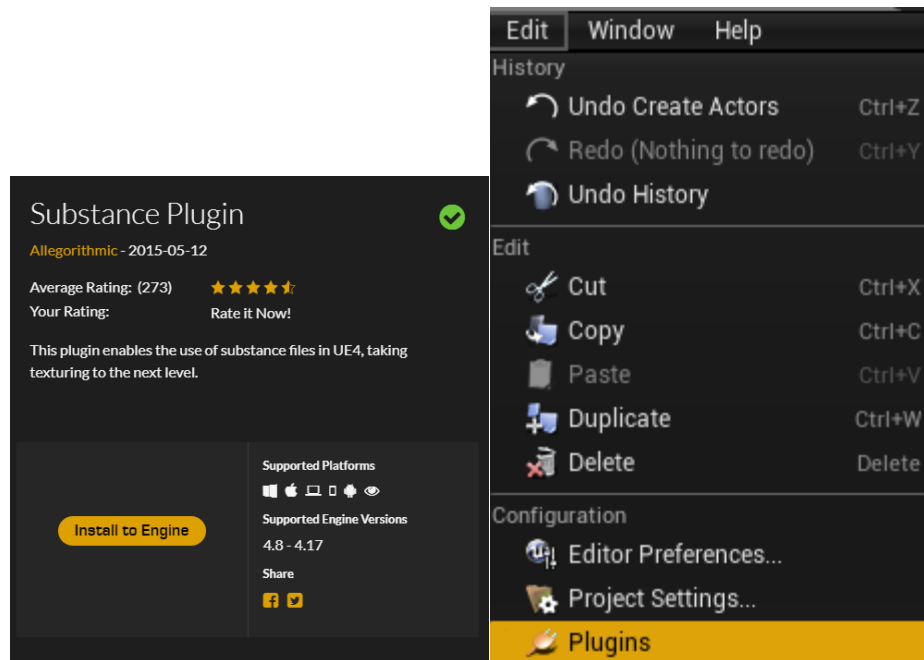
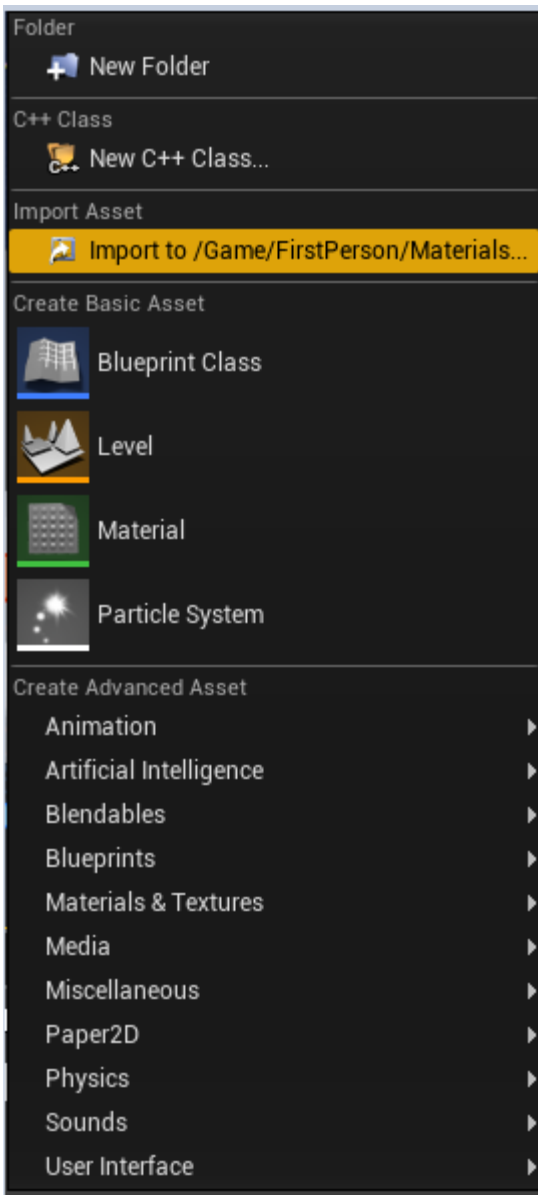


Figure 4

Figure 4.1

## The Plug In Explained

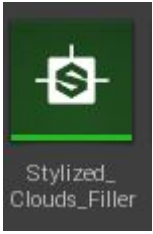
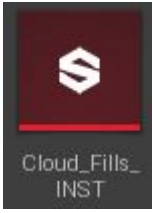

Before we work on the new material, let's import our substance files. Use the same process as we did for the mat, but instead of a new mat, select "import asset." **Figure 4.2**



**Figure 4.2** Import Asset Option Location

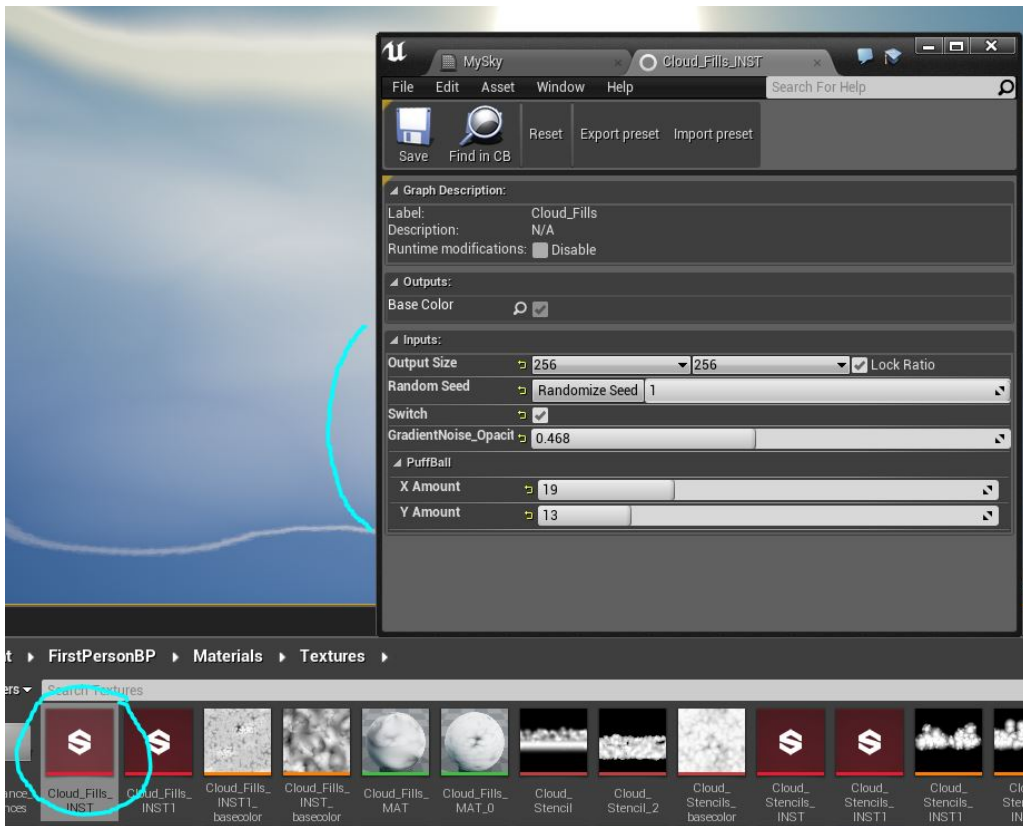


Once you import a .SBSAR file, you will be given 3 new files and a material at the locations specified during the import process. Here is a table explaining them:

Icon	Name	Description	How to Use It
 Stylized_Clouds_Filler	The Instance Factory	The file that generates new Instances of your substance graphs as Unreal assets.	To make a new instance: Rightclick the Icon > Create Graph Instance > <i>Graph_Name</i>
 Cloud_Fills_INST	Substance Material Instance	The File that contains your parameters	Double Click this to Open Parameter Window <b>see Figure 3.3</b>
 Cloud_Fills_INST_basecolor	Texture Asset	An image file output from your texture graph. Normal, roughness, etc. maps also might exist.	These are standard Unreal assets and also auto assigned in the corresponding material that was created.

- One .SBSAR file can contain multiple substance graphs, which can each have their own instance created by the Instance Factory. In this way, you can have a “ROCKS” factory with a graph for Marble, Granite, and Limestone inside. Add more graphs to your Project file in Substance Designer in order to take advantage of this!

## The Parameter Window



**Figure 4.3** The Substance Material Instance Parameter Window

This window allows you access to any parameters you exposed while in Substance Designer as well as these defaults:

- Output map Toggles
- Output Texture Size
- Global Random Seed

Once both of our graphs have had instances created, dive into the Material Editor window for your Sky Mat!

## Essential Nodes in Unreal Material Editor

In this section, I describe the basic math nodes used in almost every Unreal Material. I also list out all the nodes I used, and offer resources for learning them on your own. Unfortunately, I don't have the skill or time to effectively teach each node and its specific importance in the Sky shader graph. There are people who have already done it faster and better than me, and I direct you to them!

### Add and Subtract: the Value Shifters

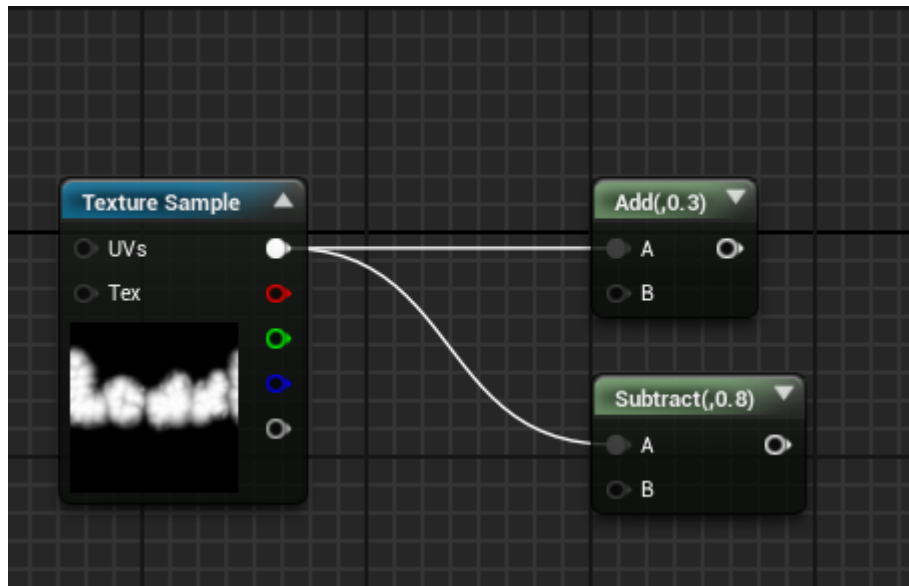
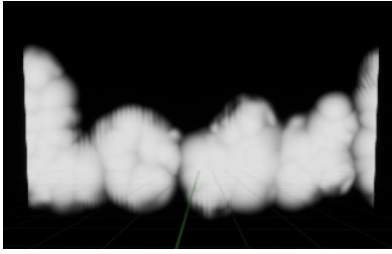
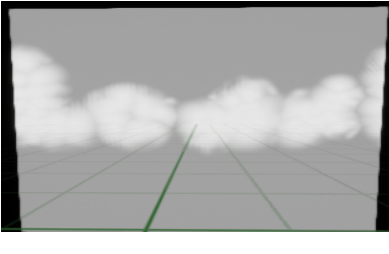
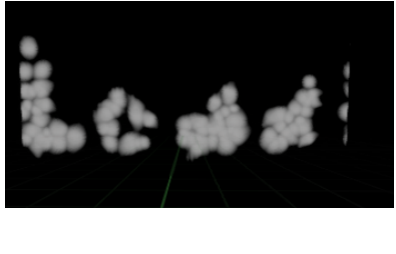


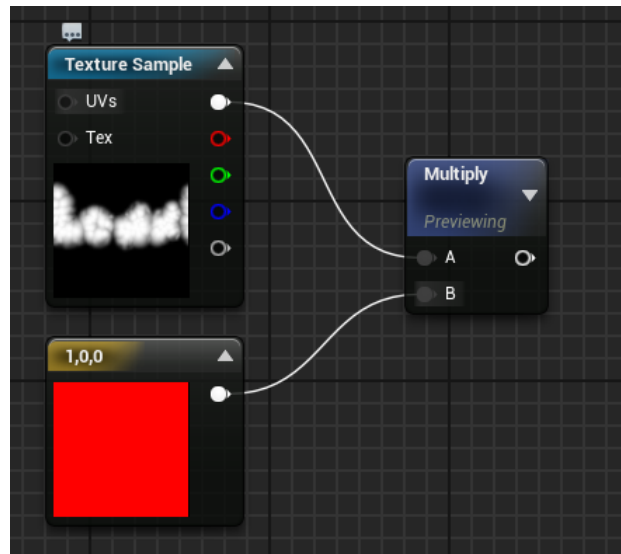
Figure 4.5

The **Add** and **Subtract** nodes take an input and shift the value of it closer to black (zero) or to white (one). If a value exceeds one, it will begin to be emissive. **Figure 4.5.1** shows us exactly how this works.

		
<b>Default Texture</b>	<b>Add Node</b>	<b>Subtract Node</b>
	All values in the image have been increased, or made closer to 1. The interior of the clouds have exceeded 1 slightly, and become mildly emissive.	All values in the image have been decreased, or brought closer to black. The original blacks have gone negative, and this will influence other operations done on those areas accordingly.

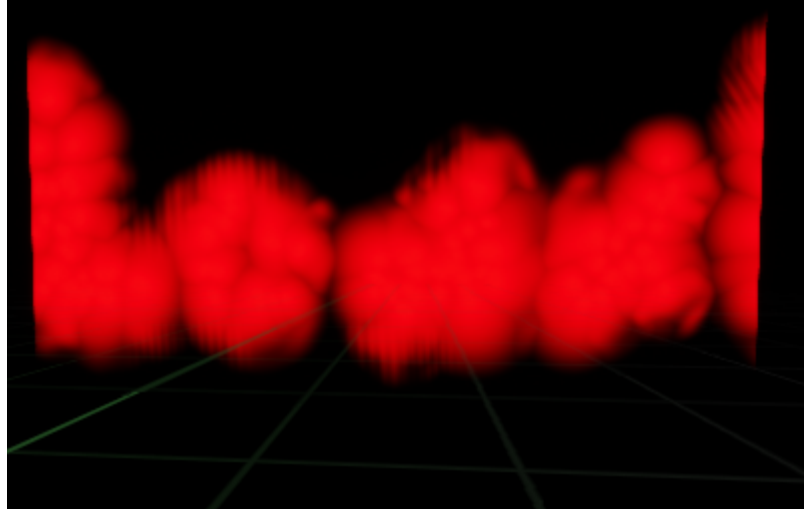
**Figure 4.5.1**

### Multiply: The Mask Node



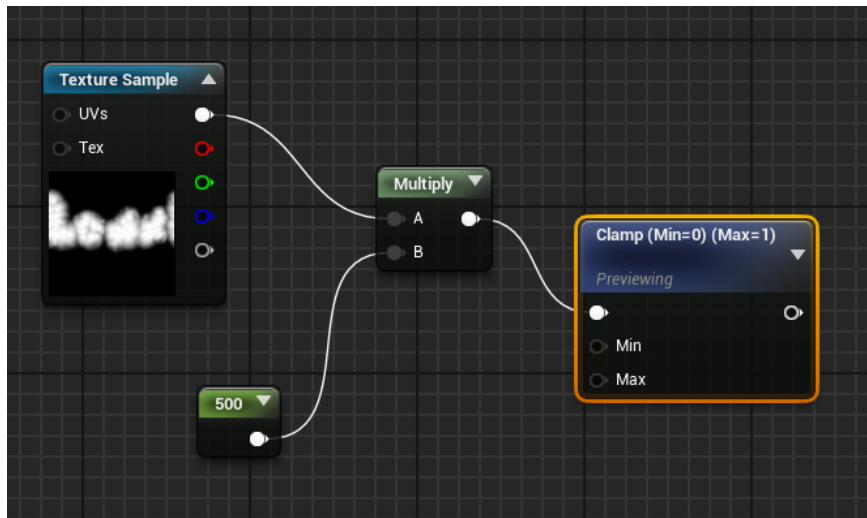
**Figure 4.6**

Multiply is useful for many things, but the masking effect is most powerful. Any black areas in an image that are multiplied disappear. This also means any areas multiplied by a number less than one have their brightness reduced, and multiplying by a number larger than one creates an emissive effect. Here is what the node setup in figure 4.6 produces:



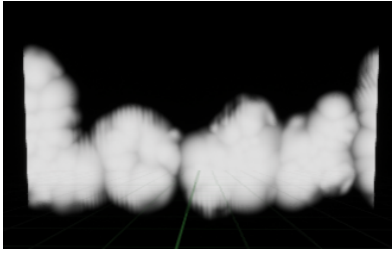
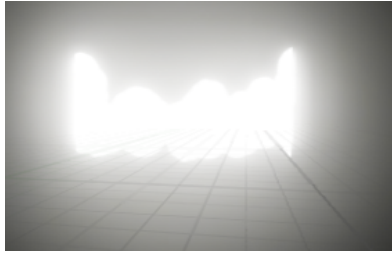
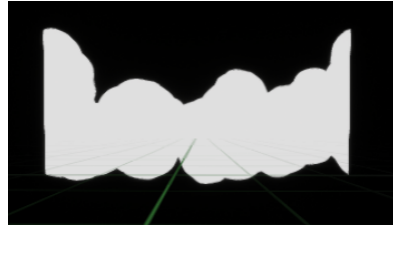
**Figure 4.6.1** The result of Figure 3.6

### Clamp: Keeping values under control



**Figure 4.7**

Occasionally you will end up with large positive or negative values for a certain area of your shader . But then for further calculations you need them to be between 0 and 1, like if you need to use it as a mask (**Figure 4.7.1**). Clamp does this for you by cutting off any value higher or lower than specified. This node does NOT re-map your values to a 0-1 space.

		
<b>Default Texture</b>	<b>Multiplied by 500</b>	<b>And then Clamped</b>
Notice the grays around the edges of the clouds.	All areas $> 0$ have become excessively emissive.	The grays multiplied by 500 were cut off at 1. The values “broke the ceiling” of 1 and have only been cut off to that height.

**Figure 4.7.1** The results of Figure 4.7

Thanks to this calculation, I now have the silhouette of my clouds, which could be useful as a visibility mask for the sun!



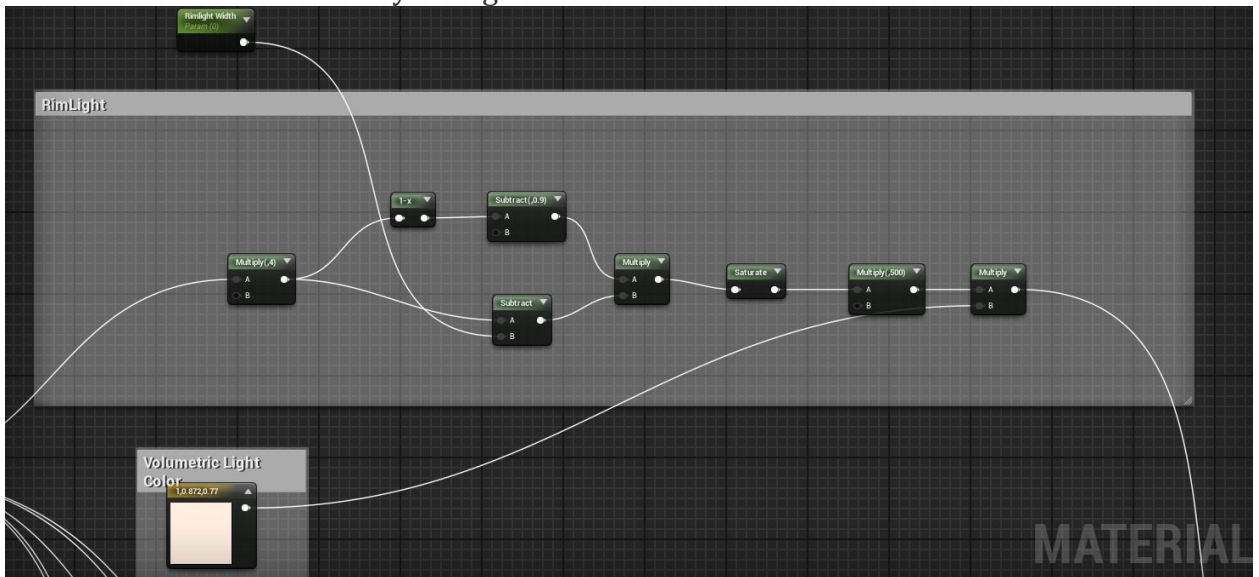
**Figure 3.7.2**

- The **Saturate** node has the exact same functionality as a default **Clamp** node, but it is often a *free calculation* on modern Graphics cards!



## Simple, Yet Powerful

These nodes, although simple, are surprisingly important when manipulating your textures and values during material creation. I used dozens of them in my sky material. They are also inexpensive on your processing resources. As an example, look at this calculation for my rimlight:



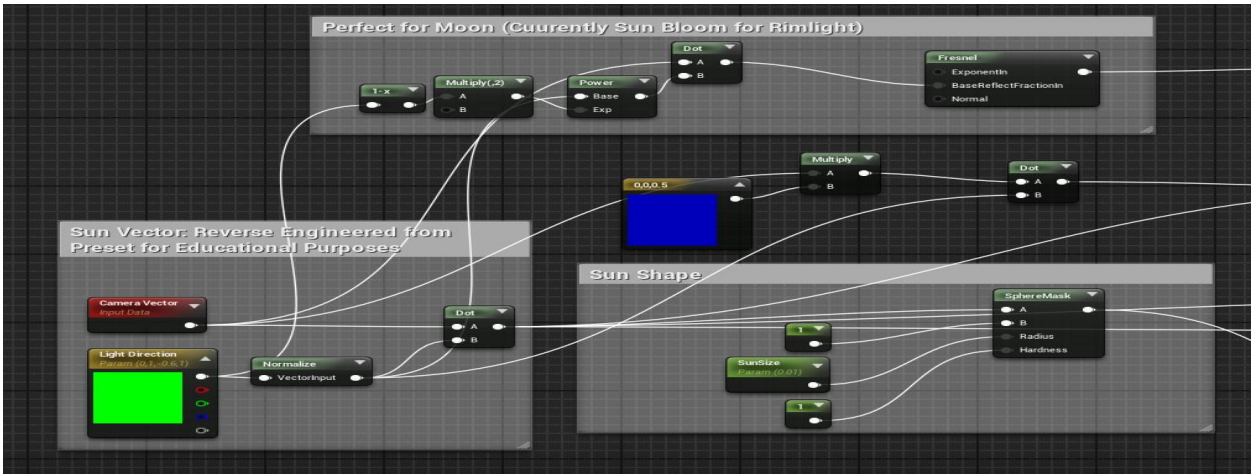
**Figure 4.8**

Of the 10 nodes depicted, 7 of them are one of those basic nodes, 2 are either a single number or a color, and only *one* node is a more complex node called **1-X**.

## The Core Node Network

The backbone of my material is made up of two networks. The first network **Figure 5** is a direct translation of Unreal's default sky shader. The second is inspired by the same, but features a unique solution to generating stylized cloud cover **Figure 5.5**.

## Section one: The Sun

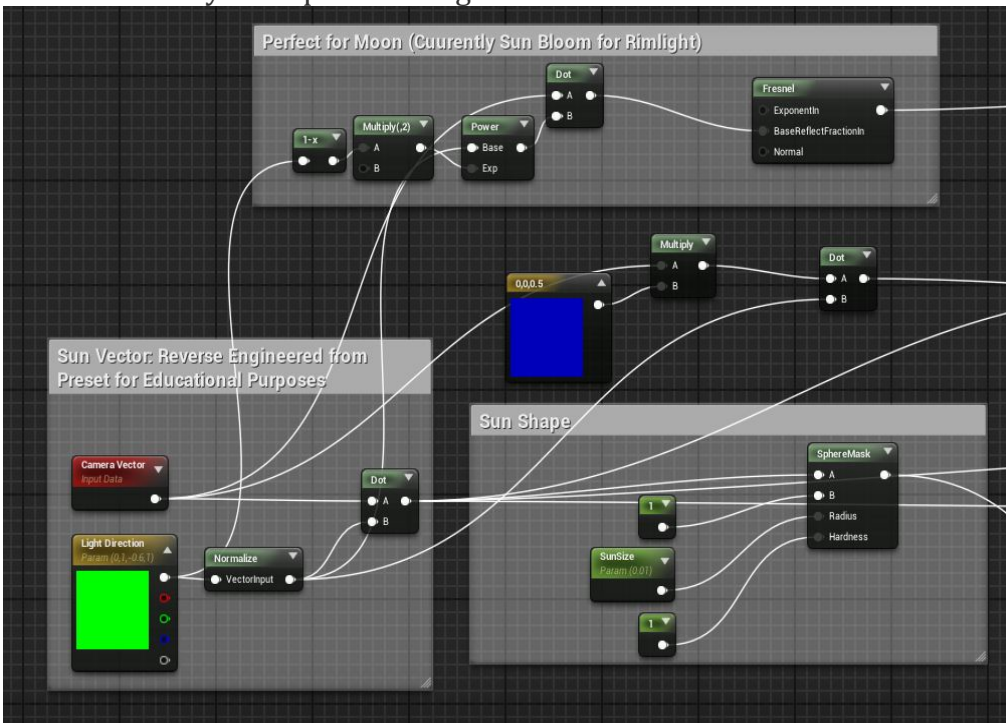


**Figure 5** The Calculations for placing a sun in the sky.

Epic Games uses the **Dot Product** node in a clever way to create a sunspot that can be controlled by a vector. **Figure 5.1**

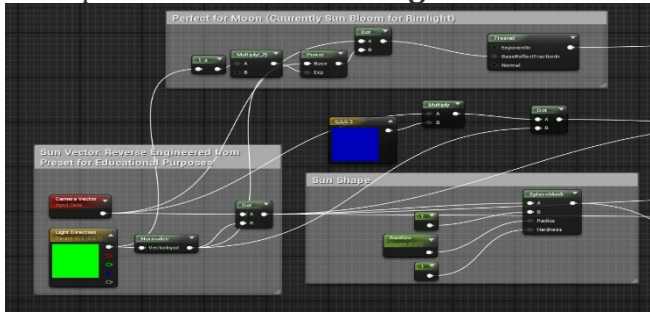
To start, they take the light direction, represented as a **Float 3**, (the three values indicate X, Y, Z, direction), and they **normalize** it. This takes the direction the light is pointing and scales it to a standardized ratio.

After that, they compare this direction against the active camera's vector with a **Dot Product**. Essentially, this causes a gradient from 0 to 1 across the dome, with 1 being at the very point where our "light vector" is coming from. In other words, the light from the sun emanates from a single point, and its strongest when we look directly at its point of origin!

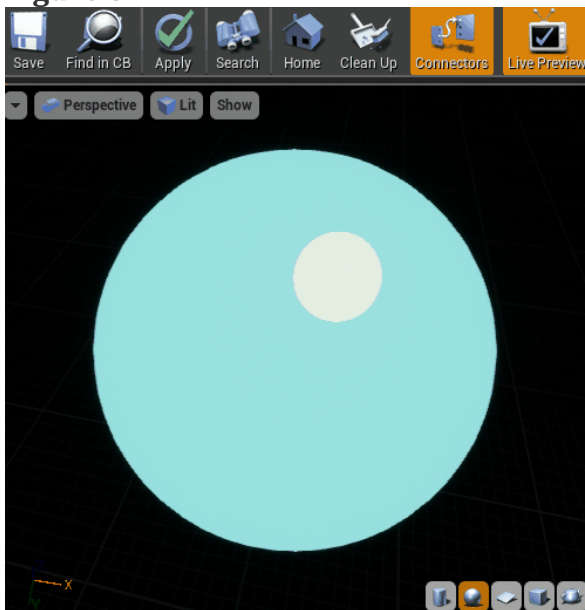


**Figure 5.1**

Currently we are creating a gradient across the entire mesh, and we need to reduce the size. We isolate the exact point of 1 in our gradient by creating a **Spheremask** node at our Dot Product's value of 1 **Figure 4.2**. This puts a small circular mask over the gradient, emanating from our sun location calculation. Example of this in action in **Figure 4.3**.



**Figure 5.2**



**Figure 5.3**

## Section Two: The Clouds

As mentioned at the beginning of this tutorial, my clouds use a stencil texture to control macro shape and fill textures to produce interior variation. The result produces large, billowing forms like **Figure 4.4**. The Network that drives these shapes is depicted in **Figure 4.5**.



Figure 5.4

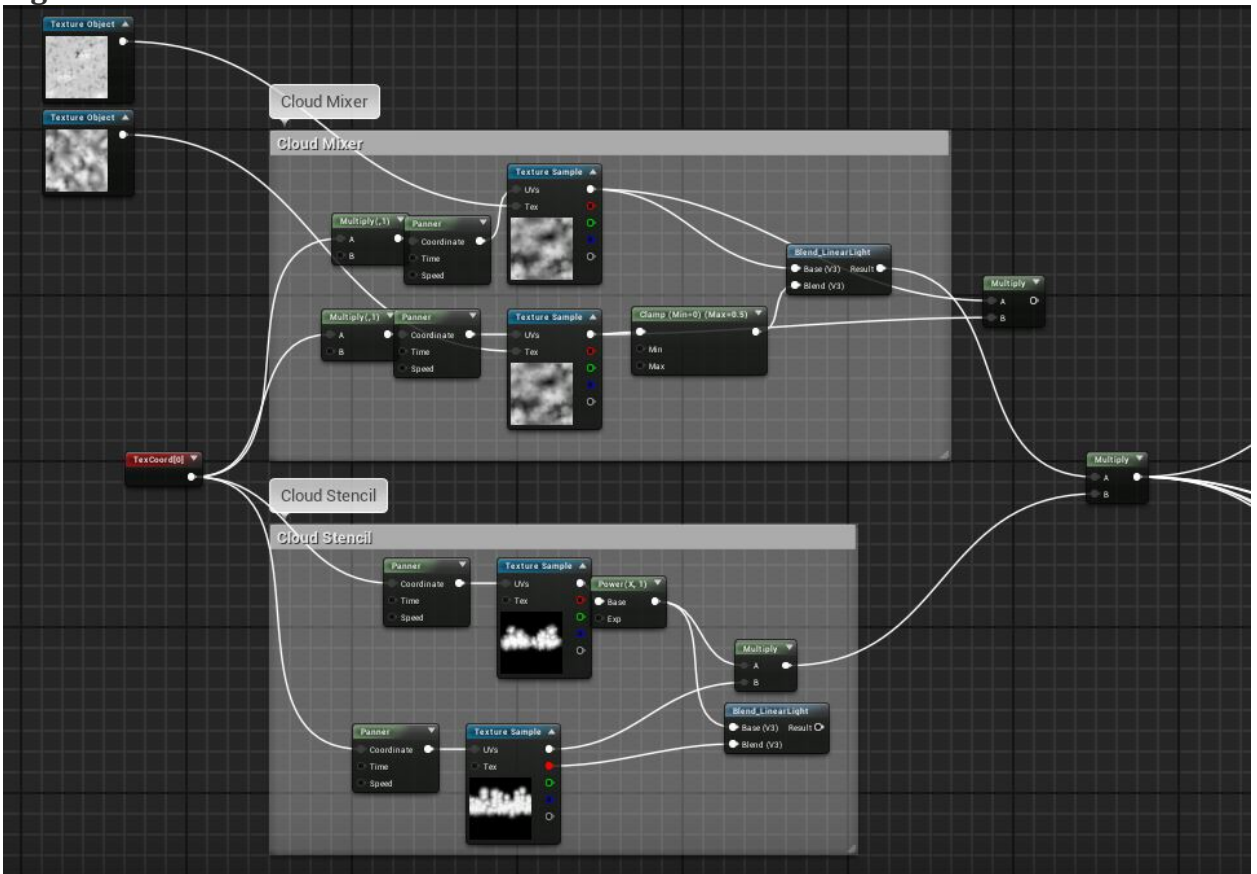


Figure 5.5 The Cloud Mixer

I use a standard **Texture Coordinate** node hooked up to a **Panner** node with very low values. This setup goes into each individual texture's UV input. There are 2 variations for stencil and fill each, resulting in 4 total textures that pan. These textures have minor edits done to them before being multiplied within their

respective types, producing subtle cloud “roiling.” Finally, the 2 moving textures are multiplied together, resulting in a very controlled but dynamic band of puffy clouds.

All of the effort we put into these textures in Substance Designer is going to pay off here:

- The 32 bit format allows the grayscale to be a full 8 bits, resulting in minimal banding.
- The inclusion of parameters allows us to either influence the texture instances dynamically during run time, or for us to make final edits and randomization before each run time, saving hours of boring menus.
- Substance Designer does auto tiling of textures for us.
- We learn two different versions of node-based shading.

### On your Own

From here, you have the power to create rimlight, volumetric light, and other interactions between your clouds and sun. Most of my calculations for these things revolve around the main nodes I described in detail, and the Linear Interpolate node. Have fun coming up with your own solutions to all of those lighting problems!

### Learning Resources

Looking for a website I mentioned ? They are all here:

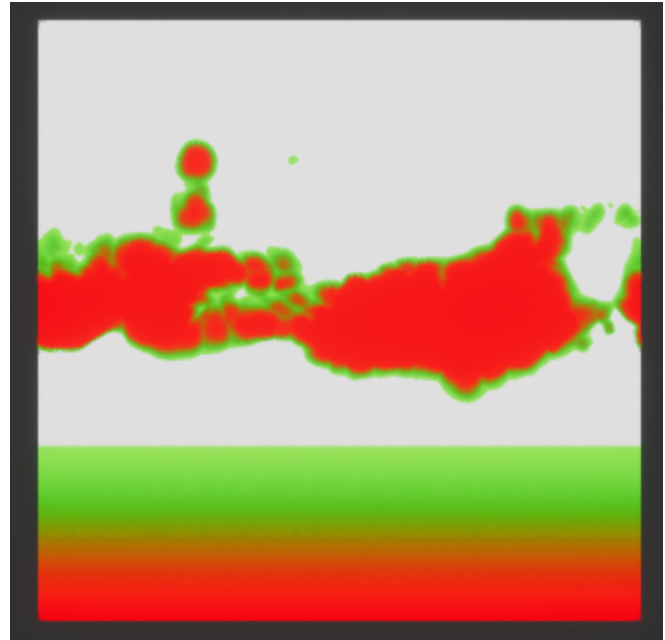
Title	Link	Description
<b>WTF IS</b> by Matthew Wadstein	<a href="https://goo.gl/A9oX8m">https://goo.gl/A9oX8m</a>	In depth explanation of specific nodes and concepts in Unreal Engine, with examples and short runtimes.
<b>Material Walkthroughs</b> with Dean Ashford	<a href="https://goo.gl/5b7nwJ">https://goo.gl/5b7nwJ</a>	A great way to expose yourself to the many different nodes and see how they work. Each project has you producing a really nice shader that you can feel proud of.
<b>Getting Started with Substance Designer</b> by Allegorithmic	<a href="https://goo.gl/2Pduy3">https://goo.gl/2Pduy3</a>	Very similar to Dean Ashfords material, but done by an official Allegorithmic rep. Explains many confusing and key concepts with grace and precision.

## UV Warping A Red and Green Texture Map

Originally used by the VFX artists at Tequila Works for creating the game **RIME**. They use this technique to create flames, and I adapted it to make my clouds (figure 3.1 & 3.2).

<https://simonschreibt.de/gat/stylized-vfx-in-rime/>

**UV Warping starts at 3:40**



Note: In order to use this effectively, make your base texture using the “Packing Textures to RGB in Photoshop” technique.

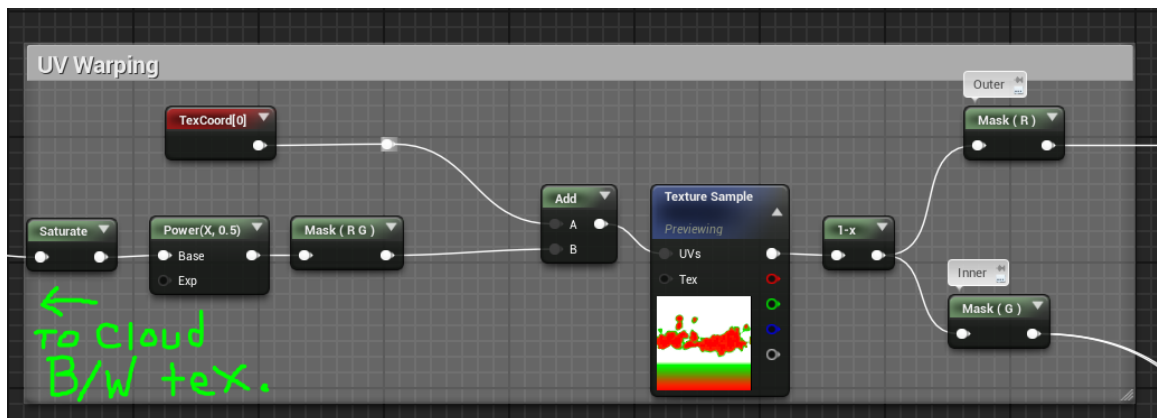


Figure 3.1 (upper right) The base red and green texture with the uv warping applied.

Figure 3.2 (lower) The shading network that produces this effect. The saturate node on the far left is connected to the panning black and white cloud textures.



## Packing Textures to RGB in Photoshop

It is possible to pack 3 separate black and white textures to a single image by having one image each in the Red, Green, and Blue color channels. Keeping these channels 100% pure can be tough, but luckily 3D Artist Xavier Coelho-Kostolny has made a super simple and great guide showing how to do this in Photoshop.

<https://80.lv/articles/easy-way-to-pack-textures-into-rgb-channels/>

# Grass Wind Vertex Animation with True Compass Direction Control.

---

By Zoey Schlemper

5/18/2018

## **Compatible with Unreal 4.16 and higher**

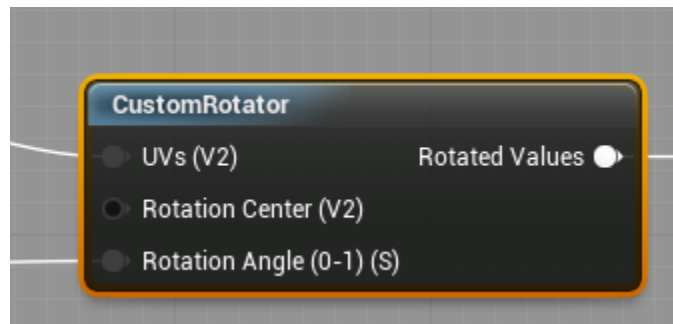
Hello! Today we are going to be making a grass wind effect that adds a new layer of complexity to the beautiful grass wind shader created by Epic Games' Jess Hider.

<https://jesshiderue4.wordpress.com/materials/stylized-wind-blown-grass/>

Once you have a wind animation material (I recommend making it a material function) that is similar or exactly like hers, you are ready for this project! From there, I am going to break down how we can add a control for changing the direction of the panning movement based on North-South and East-West directions. Her solution includes controlling the XY direction of the panning, but it doesn't properly rotate the world offset texture, causing a break in the illusion of "wind walls". The solution I am going to present rotates the panning texture so that it consistently produces the same wind effect. We will walk through the math step-by-step in order to develop a deeper understanding of how to code materials!

## The Main Problem to Solve: Translating Vector Direction into Degree Rotations

If we imagine looking straight down at the landscape of our grass, we can see that the panning node needs to change its forward movement on the XY plane, where moving in the Positive Y direction is North and moving in the Positive X is East. We can achieve this change in direction using the Custom Rotator Node (Fig 1). The “Rotation Angle” is 0-360 degrees remapped to 0-1. So that means .25 = 90 degree rotation clockwise, or moving North to point East (Fig2).



*Figure 2. Custom Rotator Node*

So if we input a compass direction as a normalized vector direction, and then translate that into an angle between 0 and 360, then remap that to 0-1 space, we will achieve the result we want! Now, the trouble is how to get there.

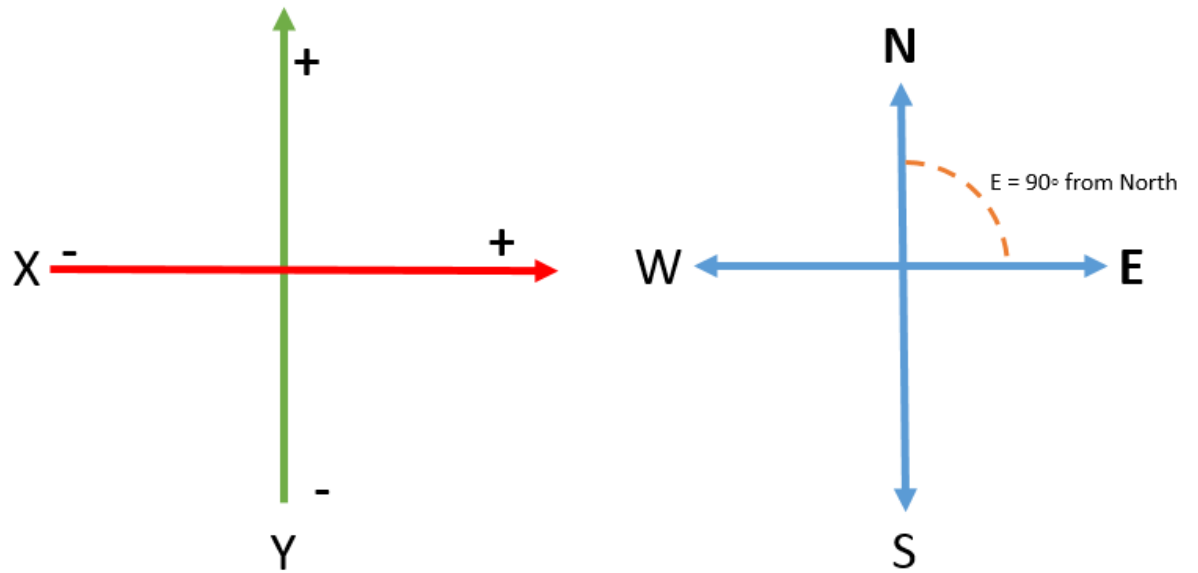
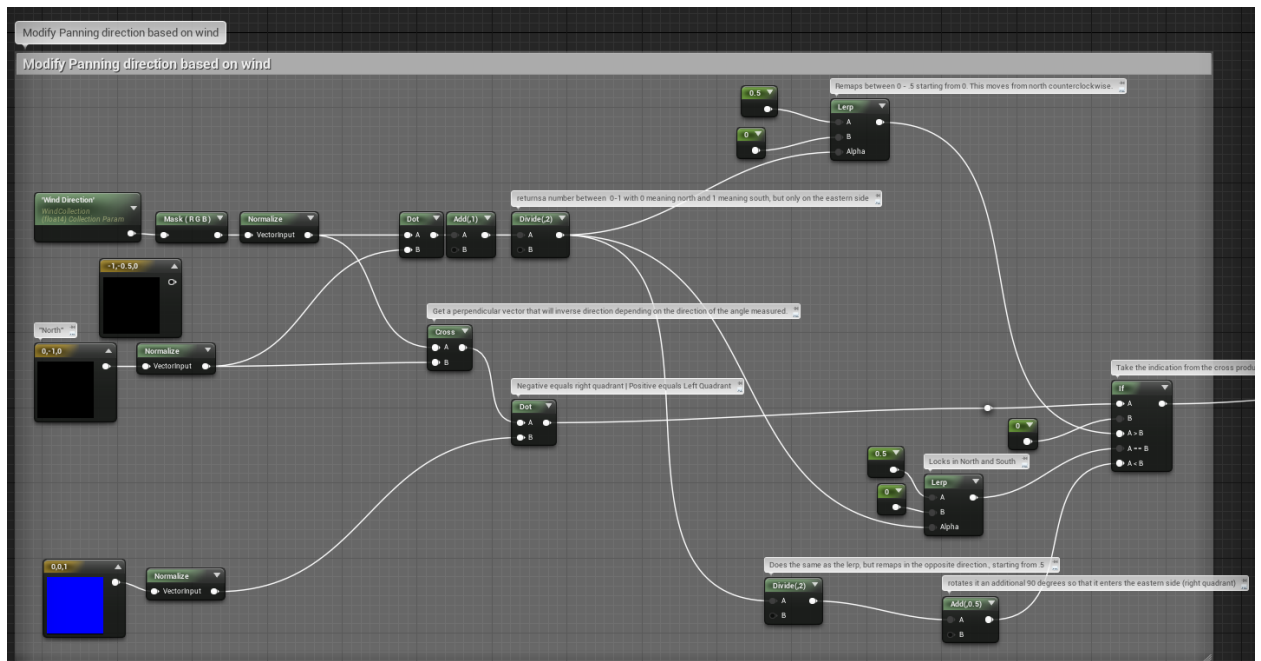


Figure 3. Comparing Compass Direction to 2-Vector Direction

## The Whole Graph



Here is the entire graph we will be constructing. Each step I will zoom in and describe what is happening and why.

First, we need a point of reference, similar to how a compass has a faceplate with the cardinal directions painted on it. Since we are assuming the un-rotated direction of our Panner node is North, we will start with a  $(0, -1, 0)$  vector (fig 3). We need to include a 0 in the Z-channel for a proper 360 degree rotation that will be explained later.



*Figure 4. Making True North for our calculations.*

Now we bring in the Wind Direction variable. I am using a Parameter collection because I want to make sure wind direction can easily affect any shader, such as this grass, other foliage, water, and particle systems. I mask out the Alpha channel on it

because it is a 4 vector, and normalize both it and my True North. Then I take the dot product of the two, which will return a number between -1 and 1. We add 1 to this result and divide it by two so that it re-maps that range between 0 and 1. (Fig 4)

Now, 0 means our wind direction is north and 1 means our wind direction is south, but it doesn't differentiate between East and West yet.

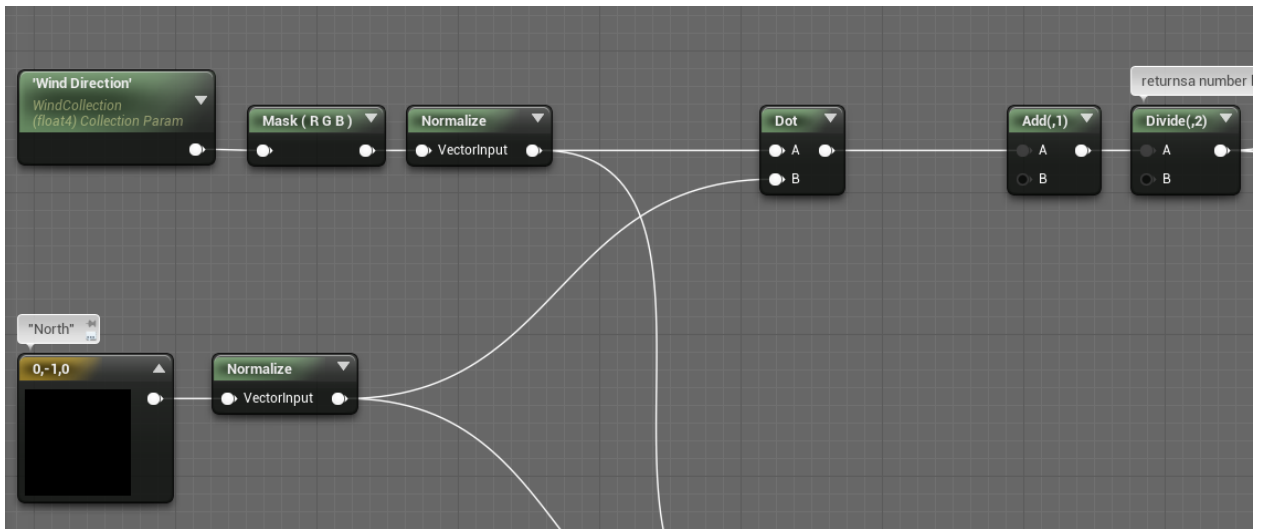
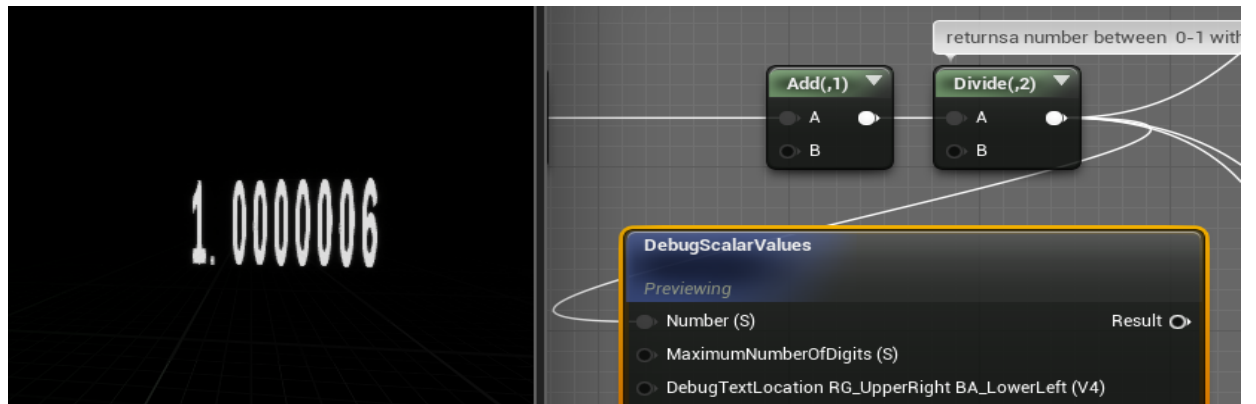


Figure 5. Differentiating between North and South on a scale from 0 – 1

**Quick Tip!** Use the “Debug” nodes in order to see what numbers you are getting when solving some math in-editor. It saves time to see the numbers instead of relying on Grayscale values or your final result.





Now we need to find a way to tell if the dot product is moving across the Eastern half or the Western half. This is where the Z-channel in our True North vector comes in. If we do a cross product of True North with our wind direction, it returns the vector perpendicular to both. Since we are comparing two vectors that exist only on the XY plane, the perpendicular result will always be a positive or negative Z vector. It returns a positive result while Wind Direction is anywhere on the Eastern Half (+X, +-Y) and a negative result while on the Western Half (-X, +-Y). Now we have a value that directly correlates with East and West. Next, we need to use these two values to produce the correct amount of rotation. (Fig 5)

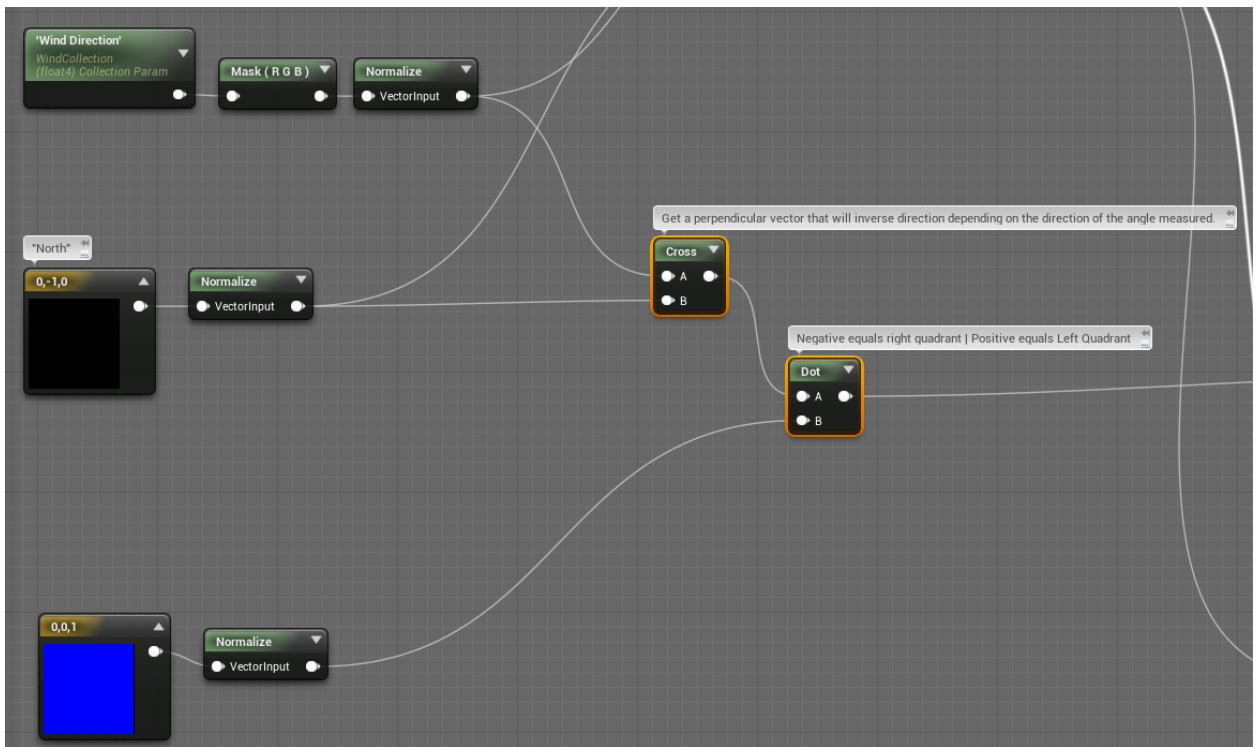
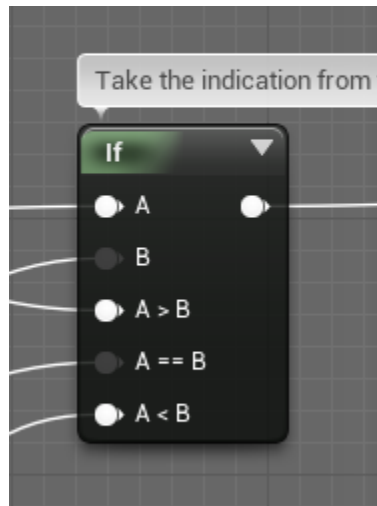


Figure 6. Using Cross Product and Z Vector to differentiate between West and East.

Now we are going to use an IF node in order to get the correct rotation values from our Wind Direction and True North dot product. The IF node (fig 6) takes in two values: A and B. Then it passes through 1 of 3 results of your choice depending on if  $A > B$ ,  $A < B$ , or  $A == B$ .



*Figure 7. The IF node.*

Remember that the dot product between a positive Z vector and our cross product of True North and Wind Direction results in either a positive or negative? If we make that value A and input 0 for B, then we have essentially created a switch that lets us pass in specific calculations for reaching either the Eastern or Western halves. (Fig 7)

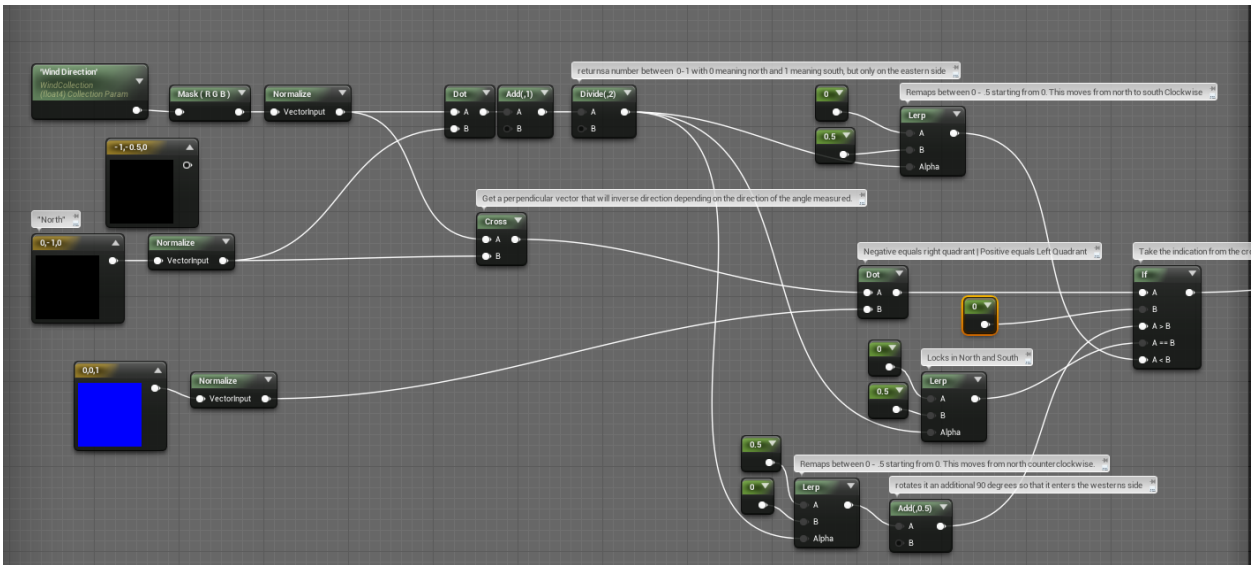


Figure 8. Connecting The IF node Properly.

When  $A > 0$ , we know we are in the eastern half. Thus, we take our 0 to 1 amount of change from north to south and lerp from 0 to 0.5, because that corresponds to a 0 to 180 degree clockwise rotation from North to South in the Custom Rotator node.

When  $A < 0$ , we know that the direction is in the Western half, or the Custom Rotator range of 0.5 to 1. We do another lerp, this time adding 0.5 afterwards because we are in the Western half, which when a full rotation clockwise from North = 1, is between .5 and 1. Our Wind Direction and True North dot product only considers how far away from North we are, so we have to inverse the relationship between 0 and .5 because now we are rotating from a starting point of South. Thus, when Wind direction is pointing South West, the dot product returns a value of about 0.75, but we really need it to return 0.125. When we add the 0.5 to that amount, we get .625, or a 225 degree clockwise rotation from North. I hope that makes sense, it is tough to explain.

Essentially we are inverting the relationship of the 0 =North and 1= South number so that it can properly increase when starting from South.

To explore this idea, try using the same lerp in both  $A > B$  and  $A < B$ . Your compass directions on the Western half will act reversed!

Finally, we use the exact same lerp as our Eastern half ( $A > B$ ) for  $A == B$ . This is because when the cross product of our Z vector and Wind Direction/True North dot product returns 0, it means that the dot product value is either exactly 0 (Wind direction vector is the exact same direction as True North) or 1 (Wind Direction is exactly the opposite of True North). In our  $A > 0$  lerp, that corresponds to 0 rotation for North, or 0.5 rotation for South. We are almost there!

Finally, we can take the result of our IF node and pump it into the Rotation Angle of our Custom Rotator. Now, when you change the Wind Direction Parameter in your Collection, the grass wind effect will properly rotate! But wait... the displacement keeps pushing the grass in the same direction. So even though our Wind Walls are moving North, the grass is animated like its being pushed South. (Fig 8)

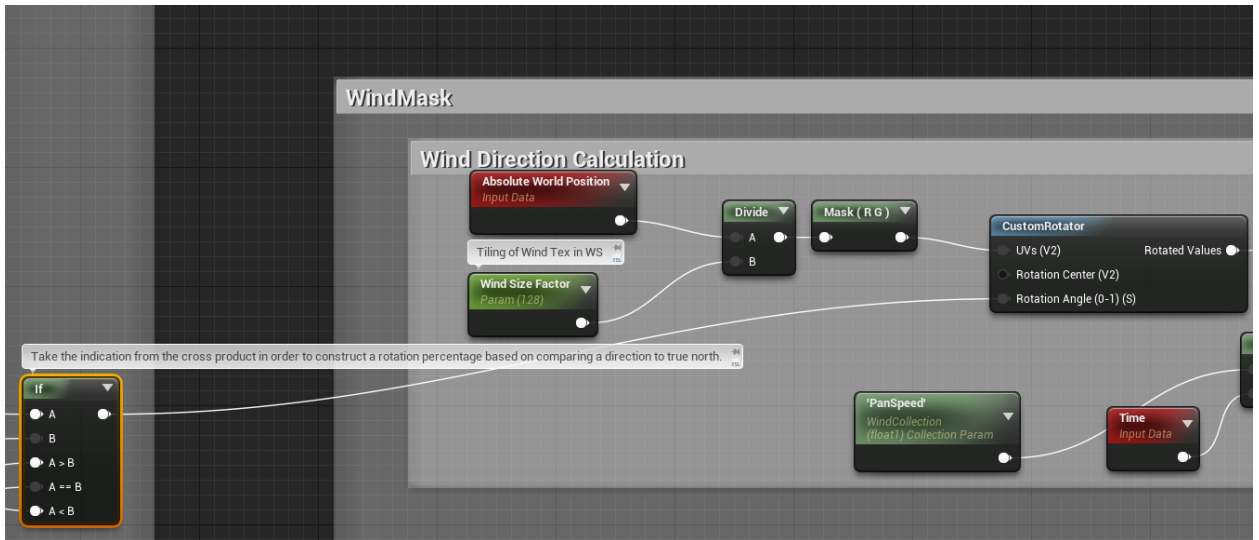


Figure 9. Connecting to the Custom Rotator

The solution is in our Wind Direction Vector Parameter. It includes the proper plus and minus in X and Y relative to our panning direction, we just need to inverse the signs and multiply it directly before outputting to World Position Offset. I added a scalar parameter for Wind Strength in order to get the offset just right. And then I mask to Red and Green to ensure no Z offset values are passed through. (Fig 9)

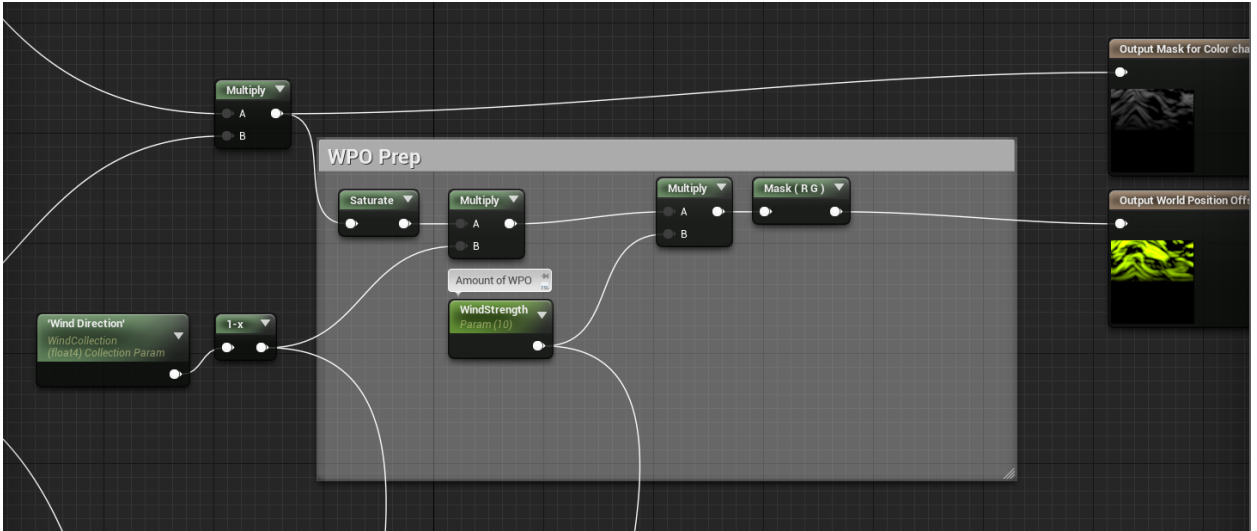


Figure 10. Aligning WPO positive/negative values with the panning direction.

Last but not least, we have to change the way our panning node moves. Change the panning speed to only positive Y speed. This keeps the texture moving along its wind walls and lets the rotator do all the work involving direction. (Fig 10)



Figure 11. Panner Settings.

I use the time node to change the speed of the panning, but you could also apply some variables to the Speed Y here for a similar effect.

That's it! You are done! If you have found better solutions while working through this or gotten creative with this concept, please let me know, I would love to see the awesome things you are making!



## Glossary

### **Banding**

The resulting visible artifact of excessive compression or deterioration of a range of values.

### **Curvature Map**

A texture that encodes relative surface angle change in a single float value.

### **Delimiter**

A syntax device in coding that separates individual entries in variable types like sets, lists, and strings. In `Cow+_Dog+_Cheese`, the delimiter is `"+_."`

### **Forward rendering**

A traditional 3D rendering pipeline which renders lighting information per-object. It is a linear process that becomes exponentially less effective as the number or size of dynamic lights increases.

### **Index**

The numeric position of an element in a data type that values position, such as an array or list. In the array `[apple, banana, orange]`, the index of `"apple"` is `"0"` and the index of `"orange"` is `"2."`

**Information Architecture**

The navigational structure of data in a program. The avenues through which data can be changed or passed with or without human interaction.

**Polygon mesh**

A 3D model made of vertices rendered using polygon primitives

**Polygon primitive**

The rendered result of the area defined by connecting 3 or more vertices.

**Post processing**

Effects applied to each rendered frame of an image as the last step in a rendering pipeline, for example tone mapping, some approximations to motion blur, and blooms.

**Ramp**

Maps individual color values to certain values on a grayscale gradient.

**Screen space**

The coordinate space of the resulting 2D image after a frame of 3D is rendered. The space is often normalized to a value from 0 to 1 for the X and Y axis.

## **String**

A data type in most coding languages that contains any number of alpha numeric characters. Cannot be processed as a number, even if a string contains only numbers (You must type-cast to an integer or float variable type).

## **Normal (or surface normal)**

In shading calculations, the “facing direction” of a given vertex, typically compared with the light and view vectors to compute the resulting visible colour.

## **UV coordinates**

Coordinates in a 2D space assigned to vertices. These coordinates reference a specific image. This image can then be rendered to the surface of the polygon primitives that those vertices construct.