University of California
Santa Barbara

Stylized 3D Scene Synthesis in Virtual Reality

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Media Arts and Technology

by

Han-Wei Kung

December 2019

The Dissertation of Han-Wei Kung is approved.

Professor Laurel Beckman

Professor JoAnn Kuchera-Morin

Professor Curtis Roads, Committee Chair

December 2019

Stylized 3D Scene Synthesis in Virtual Reality

Acknowledgements

I would like to express my gratitude to my advisor, Curtis Roads, for his willingness to guide me through my Ph.D. career. This dissertation would not have happened without his tender guidance and kind help. I also want to thank Laurel Beckman, Jo Ann Kuchera-Morin, and Tobias Hollerer, for their professional advice and insightful feedback.

My summer internship at DreamWorks Animation Studios in Los Angeles was one of the best times during my Ph.D., for which I must thank Feng Xie, for the fascinating and enlightening experience, and more importantly, her invaluable friendship.

I would also like to acknowledge my Pinscreen colleagues, Hao Li, Frances Chen, Jens Fursund, Cosimo Wei, Jaewoo Seo, Koki Nagano, Aviral Agarwal, Lok Kuang, Lain Goldwhite, Aaron Hong, Kyle San, Huiwen Luo, Zejian Wang, Qingguo Xu, and Liwen Hu, for creating a positive and comfortable work environment. A special thanks to Hao Li for his willingness to take a chance on me and invite me into the talented and diligent team. Thank you for your generous support and angelic patience, especially since it took years to complete this dissertation.

Many thanks to my parents for their unconditional love over the years. They have put so many things into giving me the opportunity. I also want to thank my sisters and brothers-in-law, who are always excited about my work. It is encouraging to have someone who always shows great enthusiasm about my work, especially when I was struggling and feeling helpless.

Finally, I want to thank all my friends. Without them, this long journey would have been much more difficult and less enjoyable.

Curriculum Vitæ
Han-Wei Kung

Education

2019	Ph.D. in Media Arts and Technology, University of California, Santa Barbara.
2014	M.S. in Visualization, Texas A&M University.
2012	M.Eng. in Computer Science, Cornell University.
2007	B.S. in Computer Science, National Chiao Tung University.

Publications

Kung, Han-Wei. “Into the Vitality: Responsive Modulation in Graphics.” In The application track, posters and demos of EuroVR: Proceedings of the 16th Annual EuroVR Conference - 2019, pp. 90-93. VTT Technical Research Centre of Finland. VTT Technology, No. 357 https://doi.org/10.32040/2242-122X.2019.T357

Zhou, Yi, Liwen Hu, Jun Xing, Weikai Chen, **Han-Wei Kung**, Xin Tong, and Hao Li. “Hairnet: Single-view hair reconstruction using convolutional neural networks.” In Proceedings of the European Conference on Computer Vision (ECCV), pp. 235-251. 2018.

Peraza Hernandez, Edwin A., Darren J. Hartl, Richard J. Malak, Ergun Akleman, Ozgur Gonen, and **Han-Wei Kung**. “Design tools for patterned self-folding reconfigurable structures based on programmable active laminates.” Journal of Mechanisms and Robotics 8, no. 3 (2016).

Hernandez, Edwin Alexander Peraza, Shiyu Hu, **Han Wei Kung**, Darren Hartl, and Ergun Akleman. “Towards building smart self-folding structures.” Computers & Graphics 37, no. 6 (2013): 730-742.

Abstract

Stylized 3D Scene Synthesis in Virtual Reality

by

Han-Wei Kung

Many forms of life in the natural world have the extraordinary capacity to sense their environments, to learn, and to remember, just as humans do, even though they are vastly different from us. In this dissertation, I presented novel techniques developed to exhibit an interactive abstract virtual reality experience that invites viewers to see the natural world from a different perspective. I developed the vertex displacement and color turbulence approaches to showcase organisms. The organisms can also modulate their shapes according to the volumes and frequencies of sound. Furthermore, the experience displays turbulent flow on the organisms surface to demonstrate the concept of energy flow, or vitality, among all organisms in the natural world. Another novel feature is that viewers can interact with the surface colors through ray casting from a handheld controller.

Contents

Chapter 1

Introduction

As our knowledge of other living beings unfolds, it has been revealed that they have intelligence and abilities that are vastly different from our own. For example, mosses live in the interstices between rocks and logs, and they have thrived with limited resources for millions of years. Whales and other marine species have evolved to depend on hearing as their primary sense to adapt to the perpetually dark world of the deep ocean. The life energy of plants and animals inspired the "re-visualization" of the natural world in virtual reality (VR) conducted in this study. Illustrating the flow of energy or vitality in the inhabitants or organisms of the virtual world was attempted, while also enabling them to modulate their shapes based on sound.

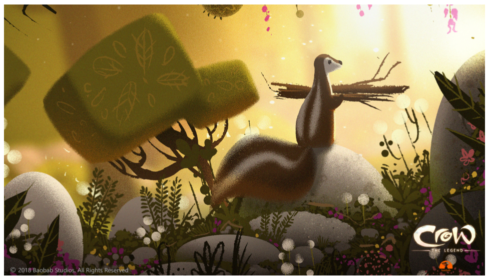
1.1 VR as an Aesthetic Medium

VR is a technology that presents an immersive computer-simulated virtual environment with which people can explore and interact, just as they would in reality. Common sense seems to assume that VR in its ultimate implementation must require photorealistic graphics to resemble actual reality. While it is true that hyper-realistic graphics are a good goal for VR, it does not necessarily follow that they should be the only goal. Indeed, it is highly likely that non-photorealistic, stylized, or abstract graphics can also look fantastic.

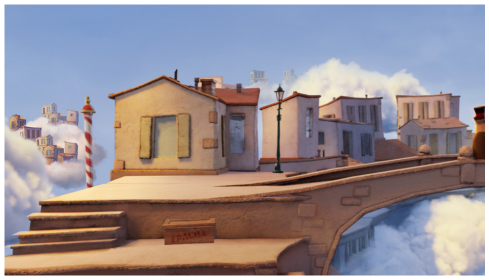
The idea of creating a stylized VR piece has led to several projects. For example, the illustrative VR narrative from Oculus Story Studio, *Dear Angelica* (Figure 1.1a), was made using Quill which essentially allows the artist to paint brushstrokes directly in VR. Watching *Dear Angelica* is like entering a cartoon world where illustrations constantly construct themselves and swirl around the viewer from scene to scene. Similarly, Google's Spotlight Stories project *Pearl* is an animated short film that looks hand-drawn and has a painterly quality (Figure 1.1b). The Google Spotlight Stories group has also created a number of stylized narrative short films in VR, each of which supports the specific director's vision; they all look quite different from each other. Furthermore, Baobab Studios and Penrose Studios have produced VR animation using stylized characters and motion as Disney and Pixar did for their animated feature films (Figure 1.1c and 1.1d).

(a) *Dear Angelica*, 2016

(b) *Pearl*, 2016



(c) *Crow: The Legend*, 2017



(d) *Allumette*, 2016

Figure 1.1: Examples of animated VR films with non-photorealistic graphics: Oculus Story Studio's *Dear Angelica*, Google's *Pearl*, Baobab Studios' *Crow: The Legend*, and Penrose Studios' *Allumette*.

1.2 Research Statement

Although the projects described above have impressive results, the style of each project cannot be readily applied to a different 3D scene. Therefore, the aim of this study was to develop an approach that can transform an arbitrary 3D scene into a stylized one in VR to convey narrative mood and emotion.

In this dissertation, two novel techniques for procedurally transforming an arbitrary 3D scene into an abstract virtual world are presented. The first, called the vertex displacement technique, transforms 3D models into organic forms that can be further modulated by sound. The other one is the color turbulence technique which creates animated textures that resemble fluid motion for audiences to interact with while immersed in VR.

These two techniques can also be combined to create a variety of organic 3D shapes with flow animation in their textures. An interactive narrative 3D environment is presented to demonstrate the results of applying these techniques in VR. It includes multiple different animated objects with a natural look. The audience can interact with them by making sounds or using VR handheld controllers.

1.3 Contributions and Novelty

The contributions of this study include the following:

- A real-time vertex displacement technique that deforms a 3D object in an organic way and makes it reactive to audio input.
- A real-time color turbulence technique that creates the appearance of a flowing texture and can invite audience interaction.
- A demo that integrates these two techniques into VR and showcases moving organisms that change shape or color when touched or when sound is played.

In addition, the following features of the presented techniques are novel:

- Using audio to deform 3D organic shapes in VR through shaders.
- Displaying interactive flow animation on 3D objects in VR through shader programs to explain a concept of life energy flow.
- Inviting viewers to interact with textures with VR handheld controllers.
- The above processes can be combined and applied to an arbitrary 3D object.

1.4 Organization

The organization of this dissertation is as follows:

- Chapter 2, Related Work. This chapter contains an overview of related work which covers two wide fields: non-photorealistic rendering and VR.

- Chapter 3, Vertex Displacement with Perlin Noise. This chapter offers an explanation of why Perlin noise was used to create the shapes for the organisms and details the vertex displacement technique using Perlin noise, which agitates the form and structure of 3D meshes.

- Chapter 4, Audio Responsive Modulation in Graphics. This chapter offers a discussion of the frequency and time domain representations of audio signals. Then, how the frequency domain representation can be useful for creating audio reactive materials is described.

- Chapter 5, Color Turbulence with Curl Noise. This chapter contains a discussion of why curl noise was used to visualize the flow of energy and introduces the color turbulence technique using curl noise, which creates swirling and flowing animation across the surfaces of 3D objects. Moreover, demos are provided to show how the technique allows objects to become interactive in VR.

- Chapter 6, An Assembly of the Two Techniques. This chapter presents multiple demos that show different ways that the vertex displacement technique and color turbulence technique could be applied. First, it presents various experimental results of combing both of the techniques and applying them to different models. In addition, an immersive virtual environment to show how the techniques can give an existing 3D scene a more lifelike feel to serve a narrative function is presented in this chapter.

- Chapter 7, Evaluation. This chapter provides an analysis of the technical and aesthetic evaluation results. First, the techniques described in the dissertation are evaluated by measuring their performance in VR. Furthermore, a comparative examination of the results generated by different types of noise functions is covered in this chapter.
- Chapter 8, Conclusions. The dissertation is concluded by summarizing the cultural meaning, limitations, and future research of the work conducted in this study.

Chapter 2

Related Work

The approaches in this study borrow from and build on two different fields of research and applications: non-photorealistic rendering and VR. Herein, the relevant background is divided into two parts. The first is an introduction to non-photorealistic rendering methods, while the second is a review of some VR history, properties, expectations, and projects.

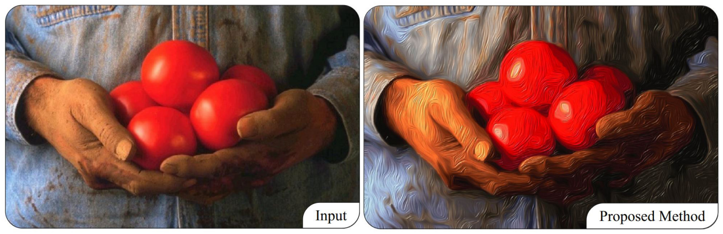
2.1 Non-photorealistic Rendering

Transforming concrete or realistic objects into an abstract or unrealistic style has been an endeavor in computer graphics for nearly two decades. Even though work to mimic the visual appearance of real-world works of art has produced various examples of stylization, it remains a subject of current interest because of the complex and rich styles of visual expression.

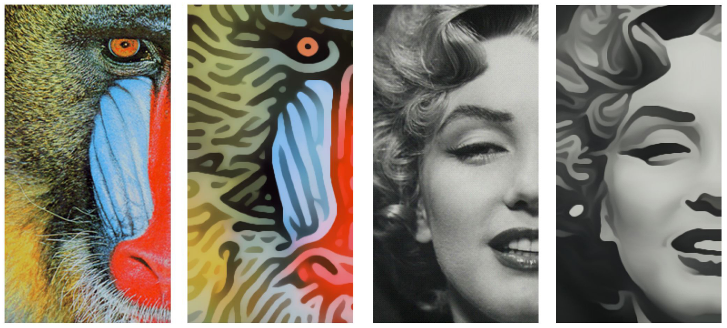
The methods used in non-photorealistic rendering can be separated into two groups: methods that render 2D input images or video into stylized images [1] and methods that render 3D models into stylized representations. Two publications provide surveys on image and video stylization algorithms [2, 3]. Here I refer to [4] and divide these algorithms into the filtering-based and example-based techniques.

- Filtering-based techniques. Filtering-based techniques synthesize stylized images by combining various brush strokes with different sizes, colors, textures, and orientations, using image processing filters that facilitate edge detection, thresholding, blurring, and so on. The state of the art can create pleasing results that resemble real-world artwork, but the visual range of what they can produce is constrained to the limited expressive properties of the predefined strokes or filters. Several interactive filtering-based rendering systems exist to support real-time image and video stylization using a set of brush strokes [5] and image processing filters [6, 7, 8, 9].
- Example-based techniques. Example-based techniques create a stylized version of a target image by transferring color and texture from an arbitrary style exemplar image to the target image. In other words, as long as a style exemplar image is given, the techniques can synthesize a stylized version of a target image. Therefore, the techniques alleviate the disadvantage of the limited visual range of filtering-based techniques. Although many of these techniques suffer from time-consuming computation and are not suitable for real-time applications, it has recently been shown that example-based techniques can be performed in real time due to the advances of deep neural networks [10, 11].

(a) Stroke-based methods [5]

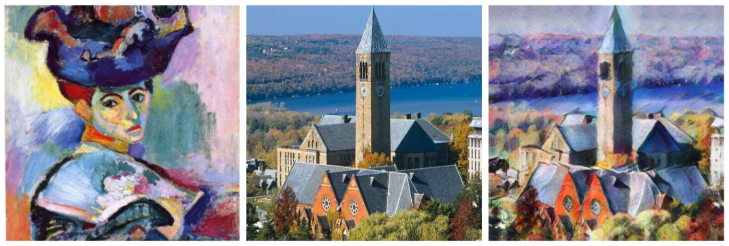


(b) Oil paint filtering [9]



(c) Coherence-enhancing filtering [8]

Figure 2.1: Artistic image stylization using stroke-based methods and image filtering.



(a) Example style transferring [10]

Figure 2.2: Artistic image stylization using example-based methods.

The methods that render 3D models into stylized representations can generally be classified to fall into the following categories [12]:

- Silhouettes and creases: Silhouettes, or contours, are lines that connect back-facing to front-facing regions of an object's surface. Creases are lines that represent discontinuous or sharp regions of the surface (Figure 2.3). A number of researchers have introduced schemes for real-time silhouette rendering of 3D scenes [13, 14, 15, 16, 17, 18, 19, 20].

- Hatching: Hatching is a line drawing technique that draws a set of lines onto an object's surface to convey its tone and form. Compared to drawing silhouettes and creases, hatching is a more sophisticated line drawing method used by artists. Several systems have addressed real-time hatching in 3D scenes [16, 13, 15, 21, 22, 23, 24].

- Stippling: Similar to hatching, stippling is a drawing technique that involves a collection of small dots drawn onto an object's surface to suggest its tone and shape. Many approaches have been proposed to digitally simulate stippling effects [25], and some of them can work in real time for interactive 3D applications [26, 27, 28, 29].

- Illustrative shading methods: Illustrative shading methods are used to shade a surface in a seemly painterly style or natural media manner. Unlike photorealistic shading, which often focuses on simulating real-world light propagation, illustrative shading does not necessarily need to perform such simulation. A substantial amount of research has been devoted to this field. For instance, Gooch proposed some classical techniques that attempt to create images similar to technical illustrations [14, 30]. Another popular area of illustrative shading is toon or cell shading, which captures the feel of cartoons and can be implemented on graphics hardware to provide interactive feedback [31]. Moreover, several methods have been proposed to simulate the appearance of real-world artistic media, such as charcoal [32, 33], ink [34], and watercolor [35, 36]. In addition, several user interface frameworks exist to facilitate artists in creating and controlling different shading styles with real-time feedback [37, 38, 39, 40].

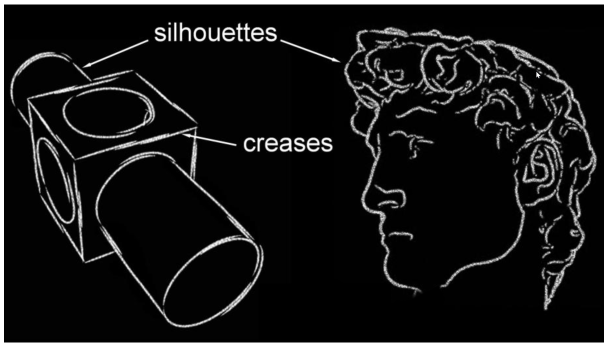
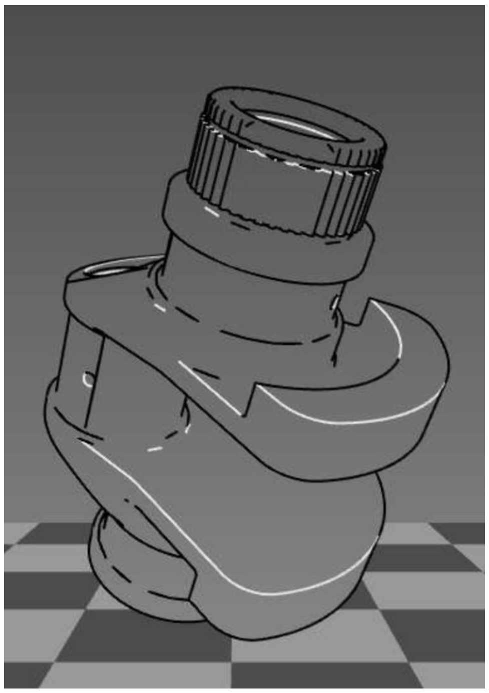
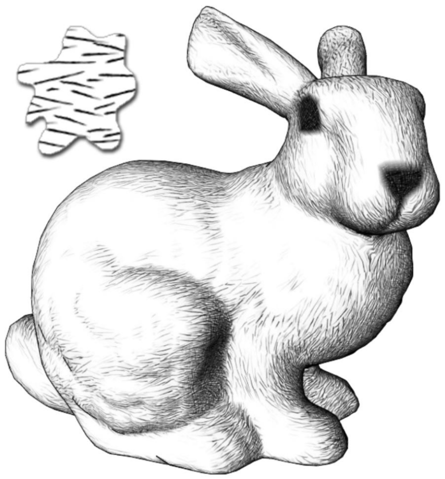


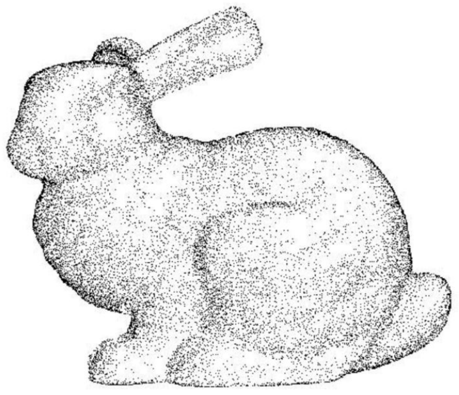
Figure 2.3: For polygonal meshes, silhouettes consist of all edges that separate front-facing from back-facing polygons. For a smooth surface, silhouettes include the loci of those surface points with a surface normal perpendicular to the view vector [41]. Creases consist of all edges that are shared by two neighboring faces, and the angle between the two faces is greater than a certain threshold value. [19]



(a) Silhouettes (black) and creases (white) [14]

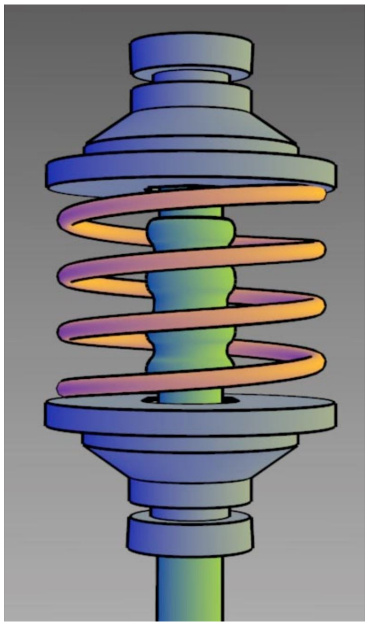


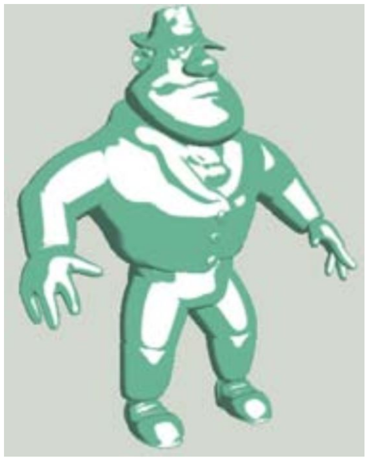
(b) Hatching [24]

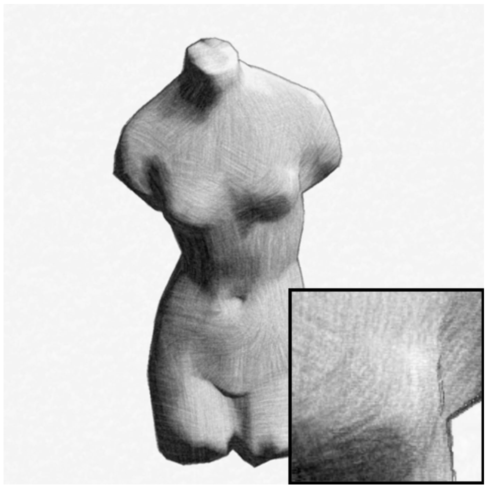


(c) Stippling [27]

Figure 2.4: Models are rendered with silhouettes and creases, hatching lines, or dots.


(a) Technical illustration [30]


(b) Toon shading [31]


(c) Charcoal rendering [33]

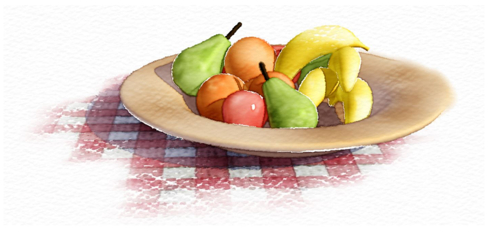

(d) Watercolor rendering [36]

Figure 2.5: Models with different shading methods that mimic artistic styles or natural media.

2.2 VR

In this section, I give an overview of some of the milestones in the history of VR. Then, I review the literature that gives a theoretical perspective of VR and discuss a few VR audio experiences and abstract or non-photorealistic VR projects.

2.2.1 VR's Moment

In 1962, cinematographer Morton Heilig developed the Sensorama [42] (Figure 2.6a), which was one of the earliest known examples of immersive, multi-sensory technology. It featured stereo speakers, a stereoscopic 3D display, fans, smell generators, and a vibrating chair.

Then, in 1965, Dr. Ivan Sutherland proposed the "ultimate display" concept that summarized the three core characteristics of VR [43].

1. It is a virtual world that is viewed through a head-mounted display and appears realistic through 3D sound and tactile feedback.

2. It simulates reality in real time.

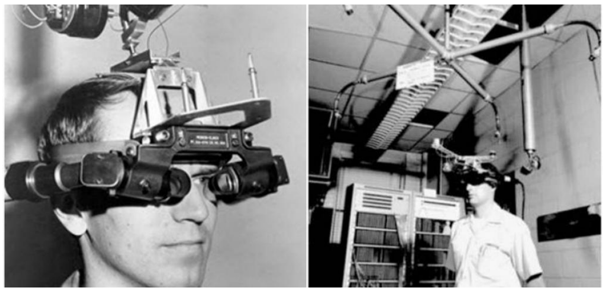
3. It allows users to interact with the objects in the virtual world just as they would in the real world.

Later, in 1968, Sutherland invented a head-mounted 3D display system, which was referred to as the Sword of Damocles [44] (Figure 2.6b). The system is considered to be the world's first VR system with a head-mounted display.

Even after all of this development in VR, a term to describe the field still did not exist. This all changed in 1987 when Jaron Lanier, founder of the visual programming lab, coined the term "virtual reality." The research area now had a name. In 1993, Sega announced the Sega VR headset for the Sega Genesis console (Figure 2.6c). However, the

Sega VR headset remained only a prototype and was never released to the general public because it could induce motion sickness and severe headaches in users [45, 46]. In 1995, Sega's chief competitor, Nintendo, released the Virtual Boy, but the device also brought discomfort after extended play [47] (Figure 2.6d). In 2016, Oculus released its Oculus Rift headset (Figure 2.6e), which was able to track the user's orientation and delivered a high-quality consumer-level VR experience. Soon, HTC also released its high-quality VR headset—HTC VIVE—which can track the orientation and position of the user so that the user can move in 3D space (Figure 2.6f). By the year 2020, hundreds of technology companies were developing VR hardware, content, and services [48].

(a) Sensorama, 1962



(b) Sword of Damocles, 1968

(c) Sega VR, 1993

(d) Nintendo Virtual Boy, 1995

(e) Oculus Rift, 2016

(f) HTC Vive, 2016

Figure 2.6: Examples of VR technical developments.

2.2.2 Properties and expectations of VR

In Murray's influential book *Hamlet on the Holodeck* [49], she explains that a computer-generated environment provides vivid and powerful forms of immersive narratives that are shaped by the properties native to computers: their procedural, participatory, spatial, and encyclopedic aspects. The procedural property means that any digital artifact is potentially procedural. That is, it is made of executable procedures. The participatory property suggests that the digital environment may invite human interaction through technical and sensory means. Furthermore, a digital environment is spatial because it is possible to navigate through it as a 3D virtual space. Finally, a digital environment is encyclopedic because it can store a high capacity of multimedia information.

Murray also explains three pleasures that define the aesthetics of digital media: immersion, agency, and transformation. The experience of being immersed in a virtual world is a relaxing and pleasurable activity in itself. Agency is the satisfying power to take action and perceive the effects, whether in real life or in a digital environment. Transformation refers to the computers' ability to create and simulate an environment to role-play.

In her book *Narrative as Virtual Reality* [50], Ryan claims that the Holodeck from the TV series *Star Trek* is the ultimate form of VR. She describes the ideal VR with the following expectations:

- Active embodiment. VR can track the participant's motion and direction of gaze and thus invite the participant to move within the virtual space with his or her actual body.
- The spatiality of the display. VR is a complete surround environment. It offers not only a 360-degree panoramic view but also depth perception, motion parallax, and spatialized audio. In this way, the participant feels the virtual environment is

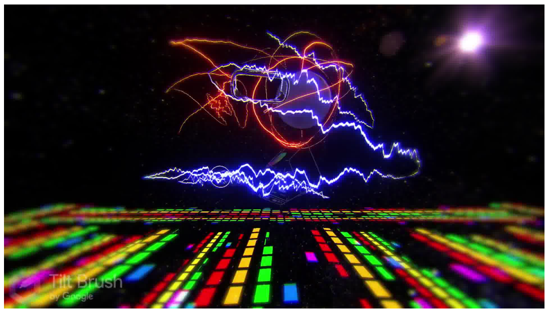
three-dimensional, instead of flat like a 2D screen.

- The transparency of the medium. Although the virtual world is the simulation result of computer programs, the computer that runs these programs is invisible to the participant.
- The dream of natural language. VR affords natural gestures and movements so that the participants can naturally interact with the objects and inhabitants in the virtual environment just as they would in the real world.
- Alternative embodiment and role-playing. VR is a first-person medium for the participant to become a character in the virtual world. The participant is able to take meaningful action and see the results.
- Simulation as narrative. The participant creates a story by engaging in the virtual world with its various affordances and themes.
- VR as a form of art. Being transported to an elaborately simulated place and enacting a role in the place is pleasurable and relaxing.

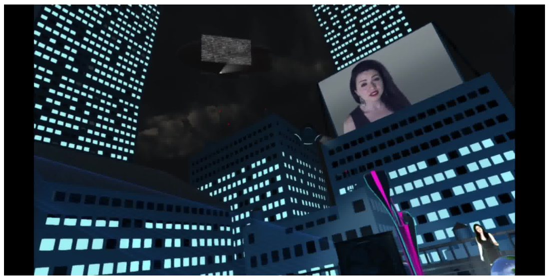
2.2.3 VR Audio Experiences

Audio-driven graphics have long been an option in motion graphics. Plenty of audio applications for VR are also available to give users immersive interaction via audio and visuals. Tilt Brush, a VR painting application from Google, introduces a collection of audio-reactive brushes in various forms, such as dots, neon pulses, and chromatic waves (Figure 2.7a). When the user plays Tilt Brush with audio in the background, the digital paint comes to life with strokes vibrating to the beat of the audio. Amplify VR, is a virtual platform that enables bands and musicians to stage performances in immersive environments (Figure 2.7b). By analyzing the rhythm and color of a music video, it translates the music video into an immersive landscape where audiences can participate.

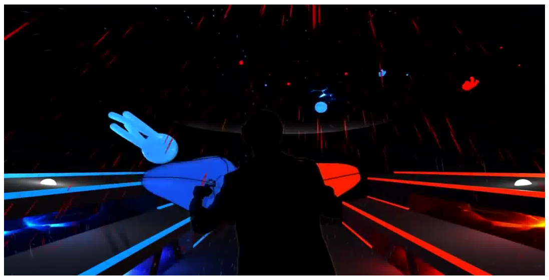
Audioshield is a rhythm game that asks the player to use virtual shields to block blobs that fly toward the player from various directions in VR (Figure 2.7c). The blobs represent crucial notes in a piece of music, so the game gives players a physical workout while they play songs. Similarly, *Beat Saber* is another rhythm game that involves slashing moving blocks with lightsabers to the beat of a song (Figure 2.7d).



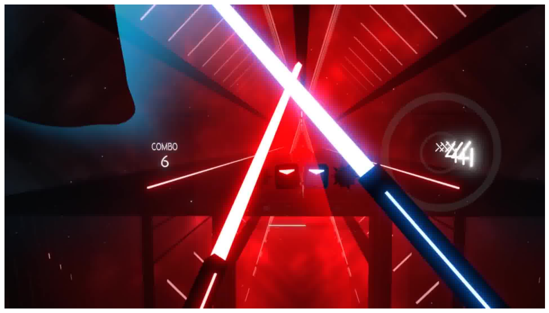
(a) Tilt Brush: Audio Reactive Brushes, 2016



(b) Amplify VR, 2016



(c) *Audioshield*, 2016



(d) *Beat Saber*, 2018

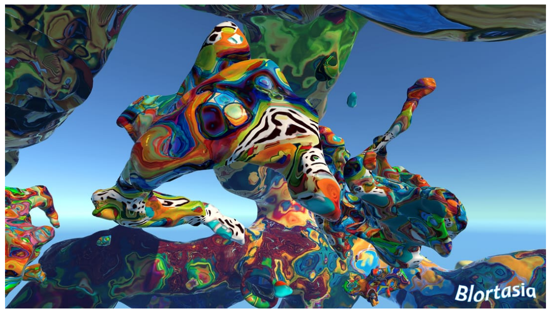
Figure 2.7: Examples of VR audio applications. (a) With Tilt Brush's various audio reactive brush strokes, users can create complex three-dimensional paintings that vibrate to an audio beat. (b) Amplify VR is a platform for artists to showcase their music in immersive environments in which the audience is an active participant. (c) *Audioshield* is a VR rhythm game that has the player blocking incoming streaks of light in time with two shields. (d) Much like *Audioshield*, *Beat Saber* is another VR rhythm game that asks the player to slice musical blocks using dual lightsabers.

2.2.4 Abstract or Non-photorealistic VR Pieces

Some well-known pieces of art have been transformed into 3D scenes in VR. For instance, Jing Yan's *Reincarnation* is a VR art experience adapted from Yves Tanguy's paintings [51]. It immerses the viewer in the fantastic beauty of the surrealist artworks and invites the viewer to participate in the unconscious environment. The viewer can interact with abstract and biomorphic creatures and join simulated flocking species.

Unlike Jing Yan's *Reincarnation*, Kevin Mack's *Blortasia* does not recreate famous paintings in VR [52]. Instead, the work is an abstract and mysterious 3D world where viewers can navigate through evolving shapes and textures. (Figure 2.8b). Similarly, Putnam's *Mutator VR: Vortex* combines organic graphics and sounds to deliver integrated sensory experiences in VR [53] (Figure 2.8c).

(a) *Reincarnation*, 2019 [51]



(b) *Blortasia*, 2017 [52]

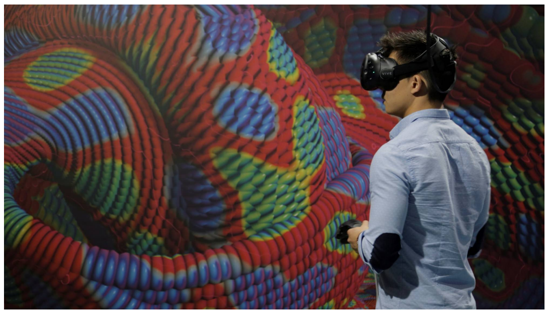
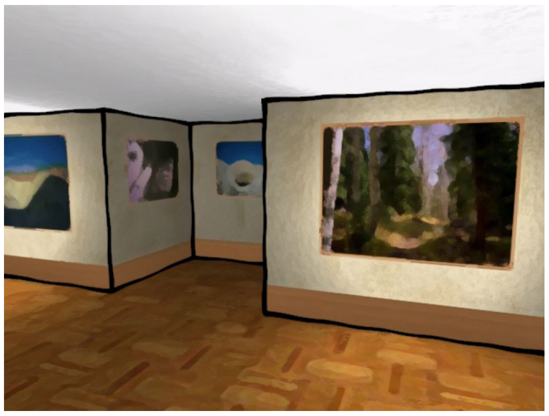

(c) *Mutator VR: Vortex*, 2017 [53]

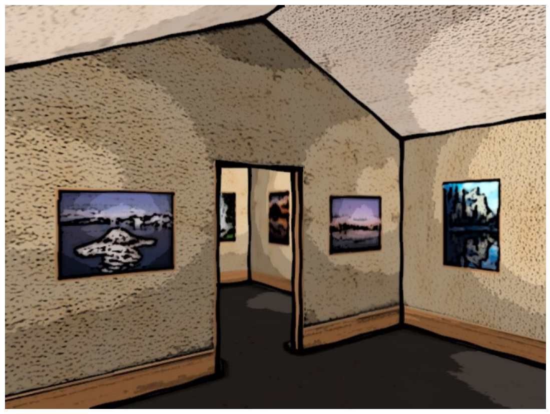
Figure 2.8: Several installations give the viewer an immersive interaction with abstract forms in VR.

These works were developed mainly by following the production pipeline in three stages: modeling, animation, and rendering [54]. The modeling stage turns 2D concepts into 3D model representations of characters, props, and environments. The animation stage animates characters and simulates physics dynamics, such as cloth and fluid. The rendering stage creates lighting and rendering effects, such as global illumination and volumetric fog.

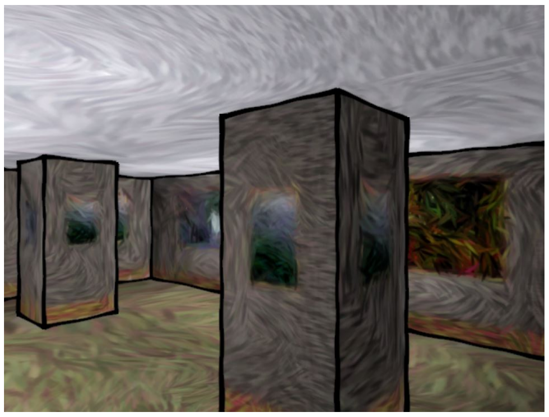
Each of these works presents one particular rendering style that was designed specifically for its own scenes. On the contrary, Klein presented a system that allows a virtual environment to be rendered in a wide variety of non-photorealistic styles [55] (Figure 2.9). It maps non-photorealistic textures that are processed with stroke-oriented filters in advance onto 3D models and then renders creases, silhouettes, and non-photorealistic textures in real time.



(a) Drybrush



(b) Pastel



(c) Van Gogh

Figure 2.9: Klein provides a method that allows a variety of styles for one virtual environment [55].

Chapter 3

Vertex Displacement with Perlin Noise

As noted in my research statement from Chapter 1, I wanted to create a work that has organic looking and lifelike behaviors. To create organic shapes, one must generate a bit of randomness that resembles the patterns found in nature. However, a purely random number generator produces numbers that are unrelated to each other and do not necessarily mimic natural shapes. In nature, most things are not purely random. For example, clouds and terrain seem to have elements of randomness, but there are many complex interactions among the many tiny particles that compose these natural patterns. There is an algorithm known as Perlin noise (developed by Ken Perlin) that can create more natural results. The shapes generated by Perlin noise have a more organic appearance because the algorithm generates a smooth sequence of pseudo-random numbers.

In this chapter, I describe how I used Perlin noise to displace the vertices of a 3D mesh to create a more natural-looking shape. Then, I will present the results generated using my approach.

3.1 Perlin Noise

In 1985, Ken Perlin wrote a SIGGRAPH paper on procedural texture generation using novel noise functions [56]. Then, in 2001, he invented "simplex noise [57]," which improved the classic noise algorithm. These noise algorithms are now known as Perlin noise. In this dissertation, I describe the classic Perlin noise only. For further details on how to generate Perlin noise, in addition to Ken Perlin's original papers, several helpful online resources are available [58, 59, 60, 61, 62].

Perlin noise can scale to any number of dimensions. I explain classic 1D Perlin noise only because it is easier to understand and implement than its other forms. To generate Perlin noise in one dimension, first, choose a random number between -1 and 1 at every integer point x = 0, 1, 2, 3, with a hash function. Each random number represents the slope of a line at that point. Then, if a given input point X is at an integer coordinate, the Perlin noise function simply returns zero. Otherwise, if X is somewhere between two integer coordinates, we interpolate a smooth function between the values where those two adjacent sloped lines cross x = X.

This interpolation is not linear with distance because that would result in a sequence of numbers that do not transition smoothly into each other. Mathematically speaking, the derivative of the noise function would not be continuous at the integer points. Instead, we interpolate a smooth function that has zero derivatives at both its endpoints so that the rate of change is always zero at both ends.

Originally, Ken Perlin used a cubic Hermite function where its first order derivative was zero at its endpoints. However, he later suggested a fifth-degree polynomial because it also has zero second order derivatives at its endpoints, which makes the noise function have continuous second order derivatives everywhere. Such property is good for some of the common computer graphics tasks, such as surface displacement and specular highlights rendering.

3.2 Implementation and Results

Because Perlin noise generates coherent noise over a space, displacing the vertices of a 3D mesh using it results in a smooth appearance and an organic feeling. In this study, this technique was implemented through vertex shaders in the Unity game engine.

Once we move the vertices, we have to recalculate the vertex normals to get correct shading across the surface (Figure 3.1). We can calculate the new normals by calculating the cross-product between the tangent and the bitangent at the new vertex position. For more details on the implementation, the reader is referred to the vertex displacement shader code listed in Appendix A.

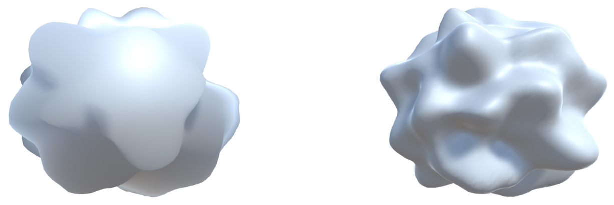
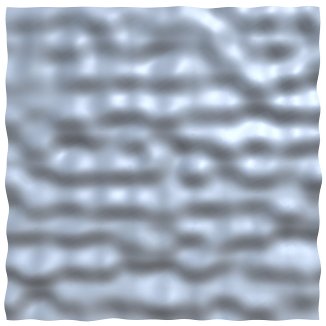
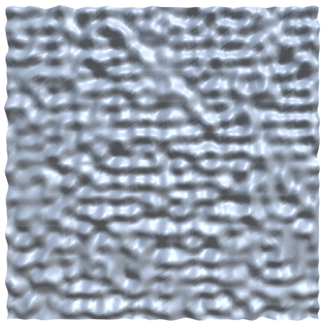


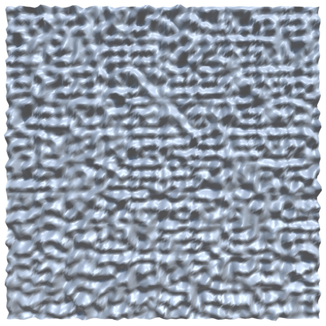
Figure 3.1: After the object's vertices are moved, the normals also need to be recalculated to get correct shading across the surface of the object. The object on the left side does not have recalculated normals. On the contrary, the one on the right side does, and the shadow correctly matches up with its shape.

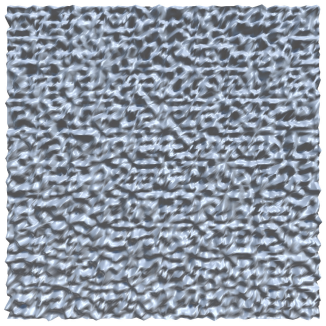
Figures 3.2, 3.3, 3.4, 3.5, and 3.6 show the results of applying the vertex displacement method to different models with varied frequency values. The vertex displacement shader has two parameters: scale and frequency. In this example, for each model, the vertex displacement shader's scale variable is held at a constant value of 0.25, while its frequency variable is changed from 0 to 5, with a constant step size of 1. Similarly, Figures 3.7, 3.8, and 3.9 show the results of applying the vertex displacement method to different creature models with different frequency values.

(a) Frequency 0.0, scale 0.25


(b) Frequency 1.0, scale 0.25


(c) Frequency 2.0, scale 0.25


(d) Frequency 3.0, scale 0.25


(e) Frequency 4.0, scale 0.25

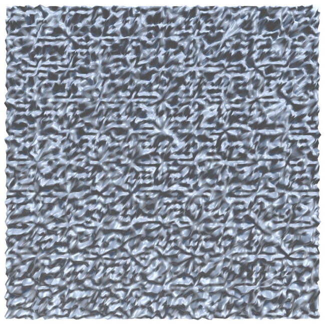
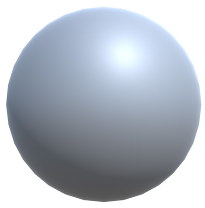
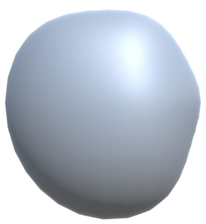
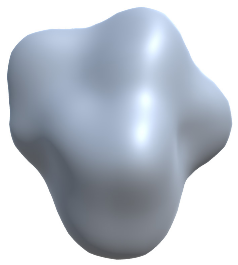
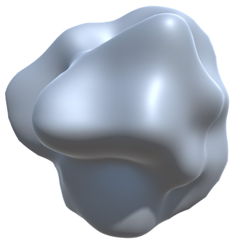

(f) Frequency 5.0, scale 0.25

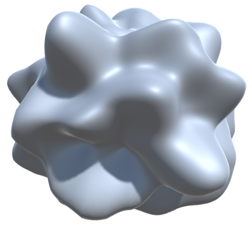
Figure 3.2: Results of applying the vertex displacement method to a plane with a fixed scale value and different frequency values.


(a) Frequency 0.0, scale 0.25


(b) Frequency 1.0, scale 0.25


(c) Frequency 2.0, scale 0.25


(d) Frequency 3.0, scale 0.25


(e) Frequency 4.0, scale 0.25

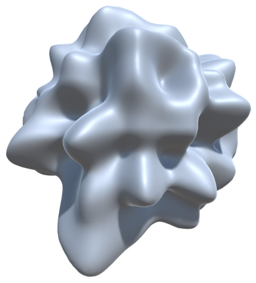
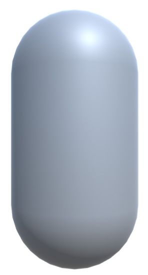
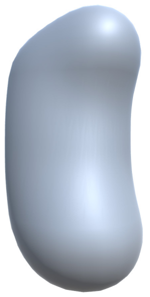
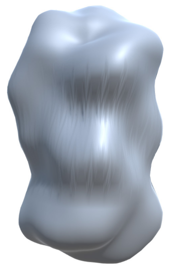
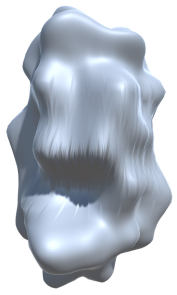

(f) Frequency 5.0, scale 0.25

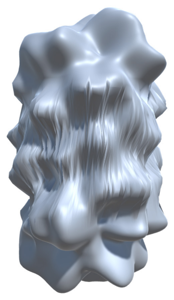
Figure 3.3: Results of applying the vertex displacement method to a sphere with a fixed scale value and different frequency values.


(a) Frequency 0.0, scale 0.25


(b) Frequency 1.0, scale 0.25


(c) Frequency 2.0, scale 0.25


(d) Frequency 3.0, scale 0.25


(e) Frequency 4.0, scale 0.25

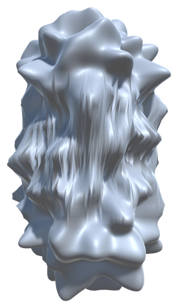
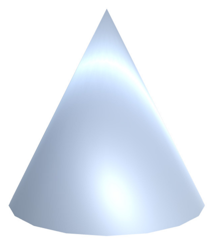
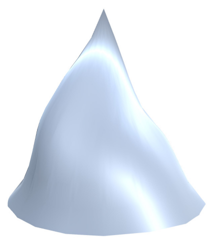
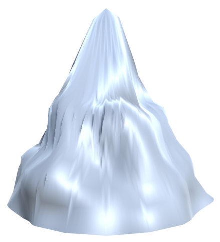
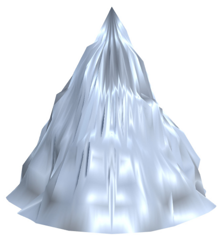

(f) Frequency 5.0, scale 0.25

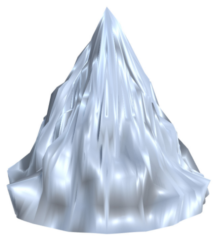
Figure 3.4: Results of applying the vertex displacement method to a capsule with a fixed scale value and different frequency values.


(a) Frequency 0.0, scale 0.25


(b) Frequency 1.0, scale 0.25


(c) Frequency 2.0, scale 0.25


(d) Frequency 3.0, scale 0.25


(e) Frequency 4.0, scale 0.25

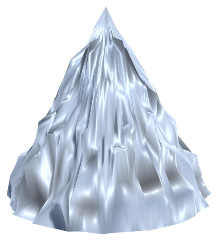
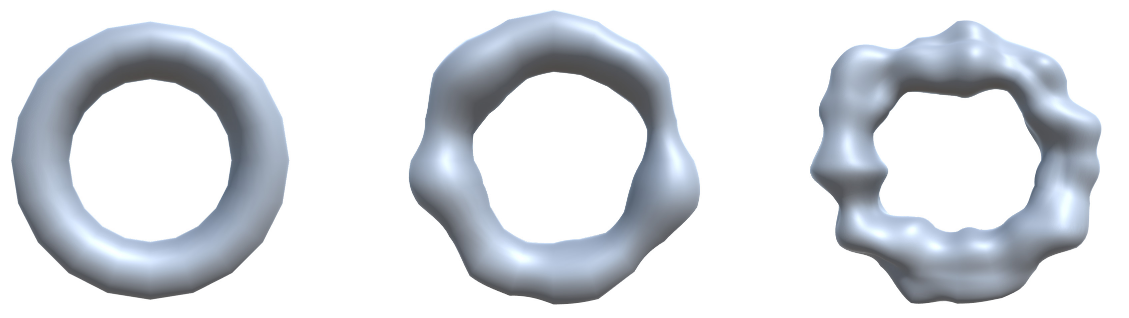

(f) Frequency 5.0, scale 0.25

Figure 3.5: Results of applying the vertex displacement method to a cone with a fixed scale value and different frequency values.


(a) Frequency 0.0, scale 0.25 (b) Frequency 1.0, scale 0.25 (c) Frequency 2.0, scale 0.25

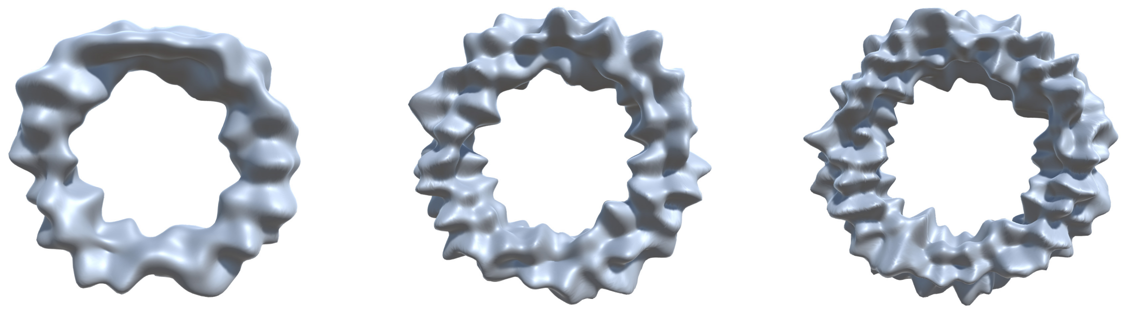
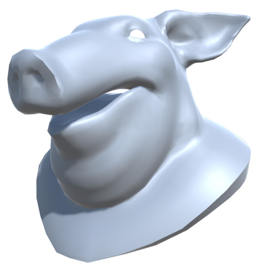
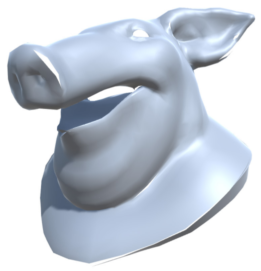
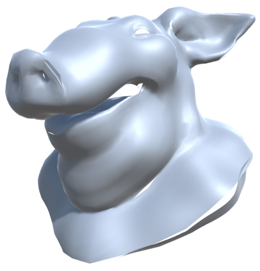
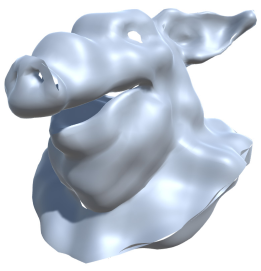

(d) Frequency 3.0, scale 0.25 (e) Frequency 4.0, scale 0.25 (f) Frequency 5.0, scale 0.25

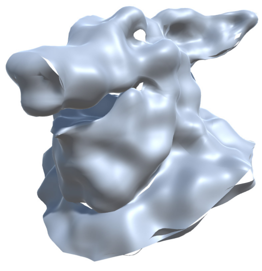
Figure 3.6: Results of applying the vertex displacement method to a torus with a fixed scale value and different frequency values.


(a) Scale 0.1, frequency 0.0


(b) Scale 0.1, frequency 1.0


(c) Scale 0.1, frequency 2.0


(d) Scale 0.1, frequency 3.0


(e) Scale 0.1, frequency 4.0

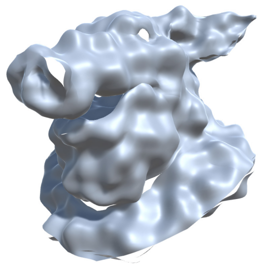
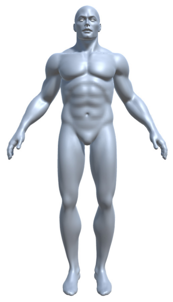
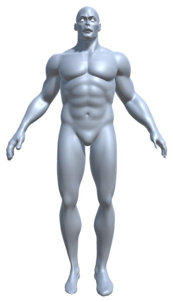
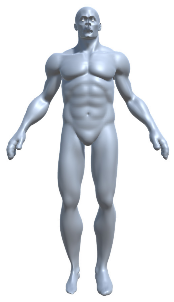
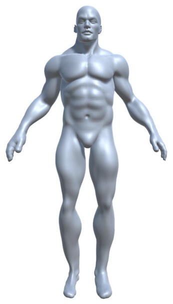

(f) Scale 0.1, frequency 5.0

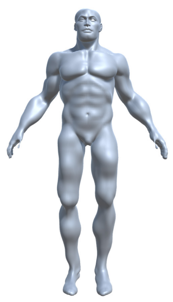
Figure 3.7: Results of applying the vertex displacement method to a pig's head with a fixed scale value and different frequency values.


(a) Scale 0.025, frequency 0.0


(b) Scale 0.025, frequency 2.0


(c) Scale 0.025, frequency 4.0


(d) Scale 0.025, frequency 6.0


(e) Scale 0.025, frequency 8.0

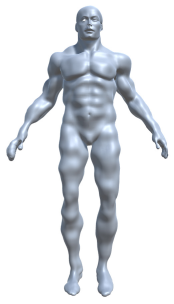
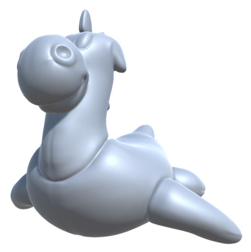
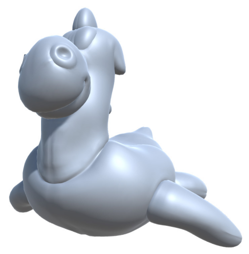
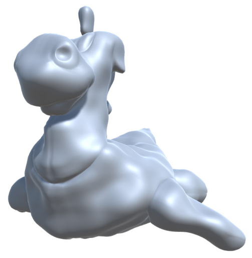
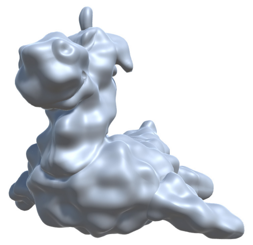

(f) Scale 0.025, frequency 10.0

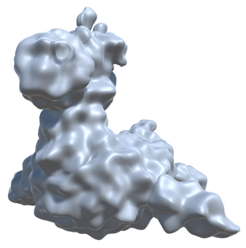
Figure 3.8: Results of applying the vertex displacement method to a human body with a fixed scale value and different frequency values.


(a) Scale 0.075, frequency 0.0


(b) Scale 0.075, frequency 2.0


(c) Scale 0.075, frequency 4.0


(d) Scale 0.075, frequency 6.0


(e) Scale 0.075, frequency 8.0

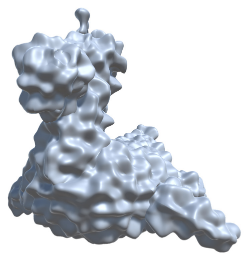

(f) Scale 0.075, frequency 10.0

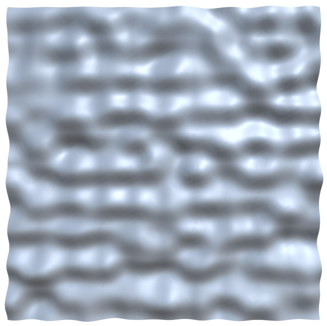
Figure 3.9: Results of applying the vertex displacement method to a toy with a fixed scale value and different frequency values.

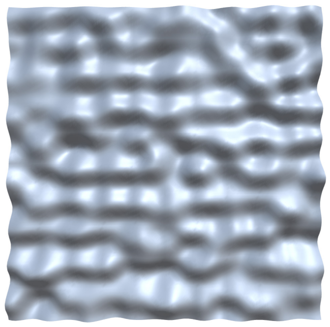
In addition, Figures 3.10, 3.11, 3.12, 3.13, and 3.14 show the results of applying the vertex displacement method to different models with varied scale values. For each model, the shader's frequency variable is held at a constant value of 6, while its scale variable is changed from 0.0 to 0.5, with a constant step size of 0.1. Likewise, Figures 3.15, 3.16, and 3.17 show the results of applying the vertex displacement method to different creature models with different scale values.

(a) Scale 0.0, frequency 1.0

(b) Scale 0.1, frequency 1.0

(c) Scale 0.2, frequency 1.0


(d) Scale 0.3, frequency 1.0


(e) Scale 0.4, frequency 1.0

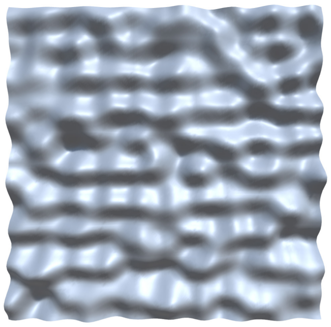
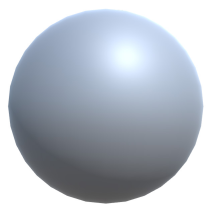
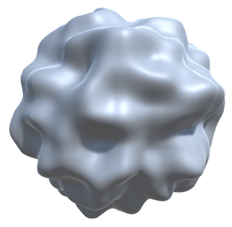
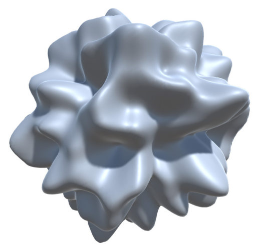
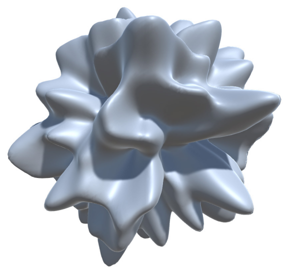

(f) Scale 0.5, frequency 1.0

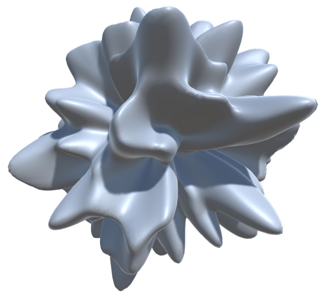
Figure 3.10: Results of applying the vertex displacement method to a plane with a fixed frequency value and different scale values.


(a) Scale 0.0, frequency 6.0


(b) Scale 0.1, frequency 6.0


(c) Scale 0.2, frequency 6.0


(d) Scale 0.3, frequency 6.0


(e) Scale 0.4, frequency 6.0

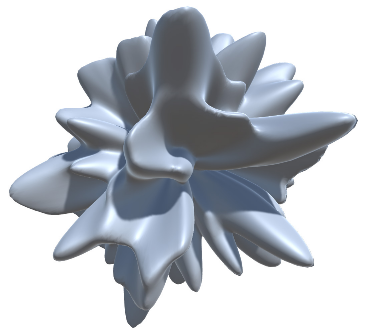
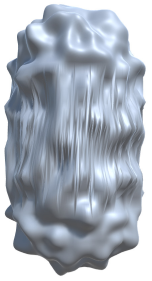
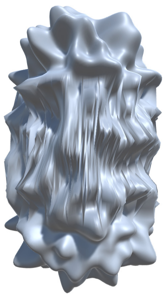
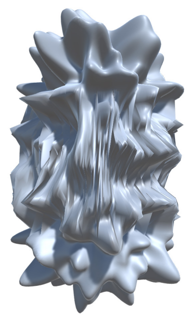

(f) Scale 0.5, frequency 6.0

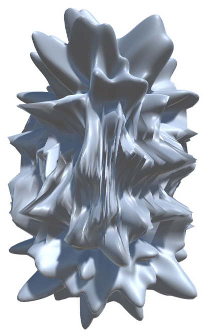
Figure 3.11: Results of applying the vertex displacement method to a sphere with a fixed frequency value and different scale values.

(a) Scale 0.0, frequency 6.0


(b) Scale 0.1, frequency 6.0


(c) Scale 0.2, frequency 6.0


(d) Scale 0.3, frequency 6.0


(e) Scale 0.4, frequency 6.0

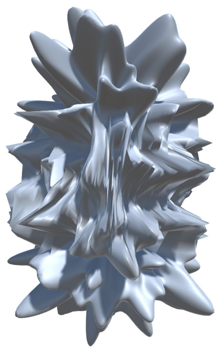
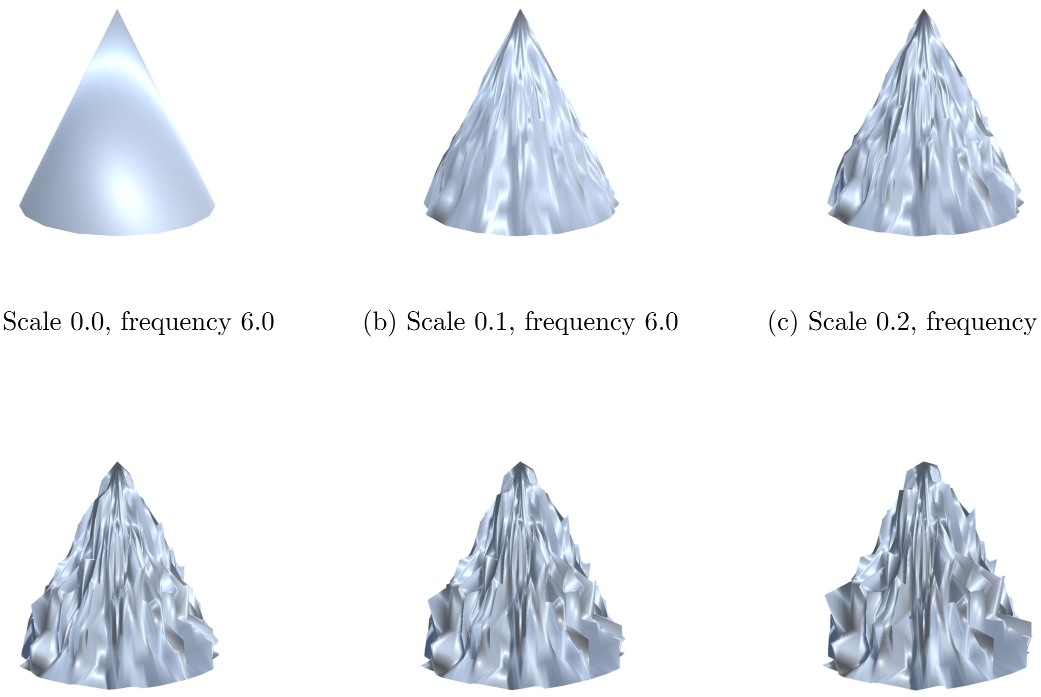

(f) Scale 0.5, frequency 6.0

Figure 3.12: Results of applying the vertex displacement method to a capsule with a fixed frequency value and different scale values.



(a) Scale 0.0, frequency 6.0

(b) Scale 0.1, frequency 6.0

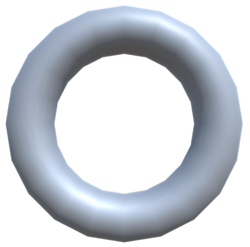
(c) Scale 0.2, frequency 6.0

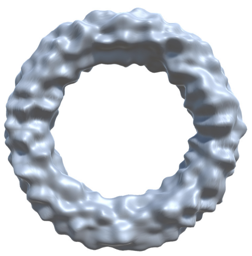
(d) Scale 0.3, frequency 6.0

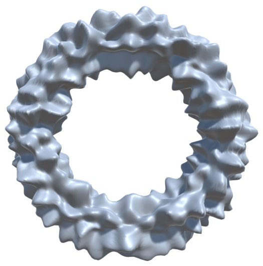
(e) Scale 0.4, frequency 6.0

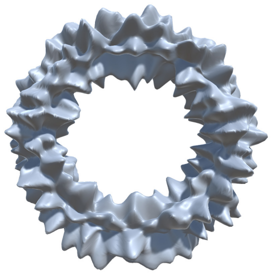
(f) Scale 0.5, frequency 6.0

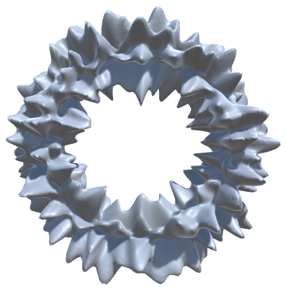
Figure 3.13: Results of applying the vertex displacement method to a cone with a fixed frequency value and different scale values.


(a) Scale 0.0, frequency 6.0


(b) Scale 0.1, frequency 6.0


(c) Scale 0.2, frequency 6.0


(d) Scale 0.3, frequency 6.0


(e) Scale 0.4, frequency 6.0

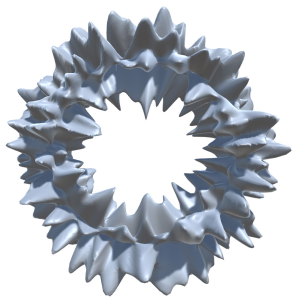
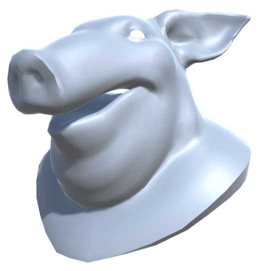
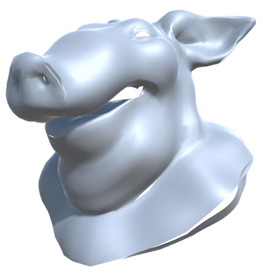
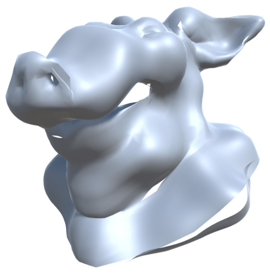
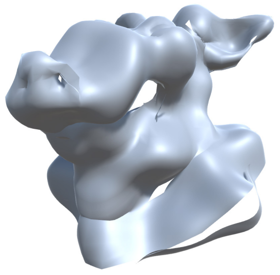

(f) Scale 0.5, frequency 6.0

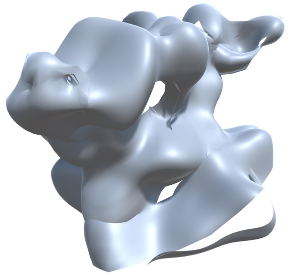
Figure 3.14: Results of applying the vertex displacement method to a torus with a fixed frequency value and different scale values.


(a) Scale 0.0, frequency 2.0


(b) Scale 0.1, frequency 2.0


(c) Scale 0.2, frequency 2.0


(d) Scale 0.3, frequency 2.0


(e) Scale 0.4, frequency 2.0

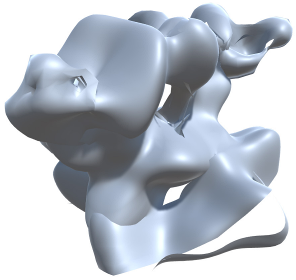
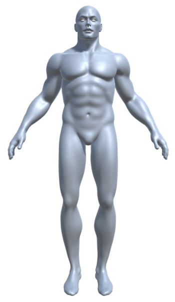
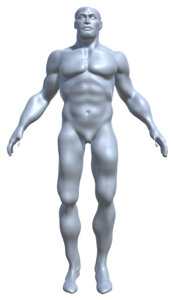
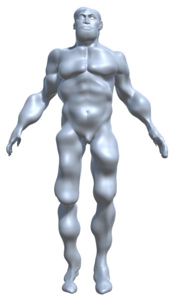
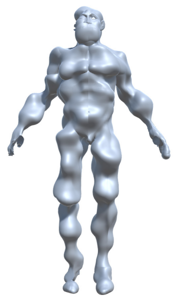

(f) Scale 0.5, frequency 2.0

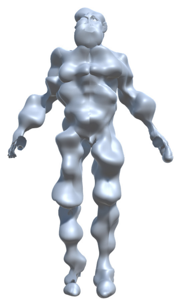
Figure 3.15: Results of applying the vertex displacement method to a pig's head with a fixed frequency value and different scale values.


(a) Scale 0.0, frequency 8.0


(b) Scale 0.02, frequency 8.0


(c) Scale 0.04, frequency 8.0


(d) Scale 0.06, frequency 8.0


(e) Scale 0.08, frequency 8.0

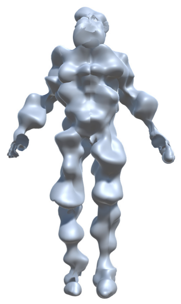
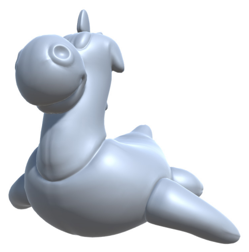
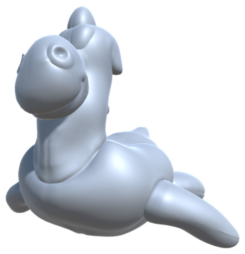
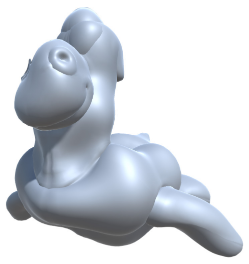
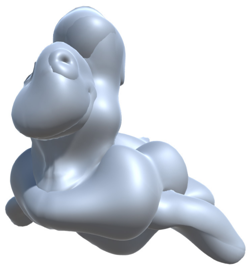

(f) Scale 0.1, frequency 8.0

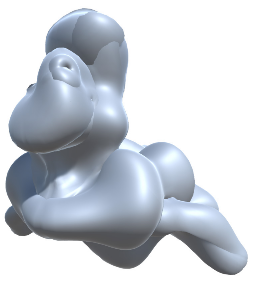
Figure 3.16: Results of applying the vertex displacement method to a human body with a fixed frequency value and different scale values.


(a) Scale 0.0, frequency 2.0


(b) Scale 0.1, frequency 2.0


(c) Scale 0.2, frequency 2.0


(d) Scale 0.3, frequency 2.0


(e) Scale 0.4, frequency 2.0

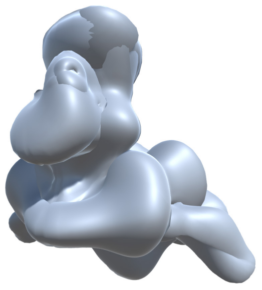

(f) Scale 0.5, frequency 2.0

Figure 3.17: Results of applying the vertex displacement method to a toy with a fixed frequency value and different scale values.

Chapter 4

Audio Responsive Modulation in Graphics

Organisms in the virtual world can modify their structure and appearance. Thus, their shapes can be modulated based on sound input from music or microphones through shader programs. In this sense, the organisms "respond to" sound. This chapter covers the details of how the organisms were made to be sound-reactive.

4.1 Fourier Transform

The first step to have audio modulate the shapes through a shader is to interpret the audio with a meaningful representation that can provide useful information for the shader to process. One common way to represent audio is by showing how loud the audio is over time (this representation is called the time domain, as shown in Figure 4.1). However, this does not help us understand what we actually hear. Sound consists of different frequencies of sine waves, and so instead of representing audio in the time domain, we can alternatively represent it in the frequency domain by using the Fourier transform; this is a mathematical tool that converts a time-dependent signal into a frequency-dependent signal, thereby revealing information about the frequencies of the sine waves that make up the original signal. As a result, instead of examining audio signal loudness and time, we determine their loudness and frequency.

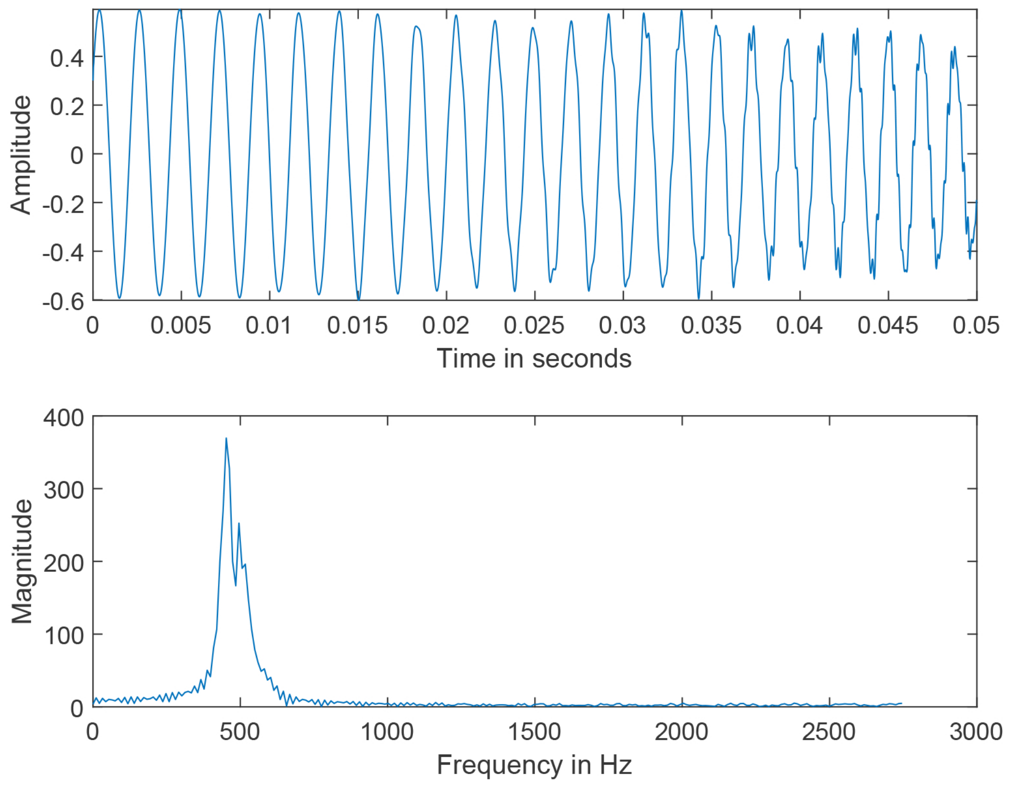
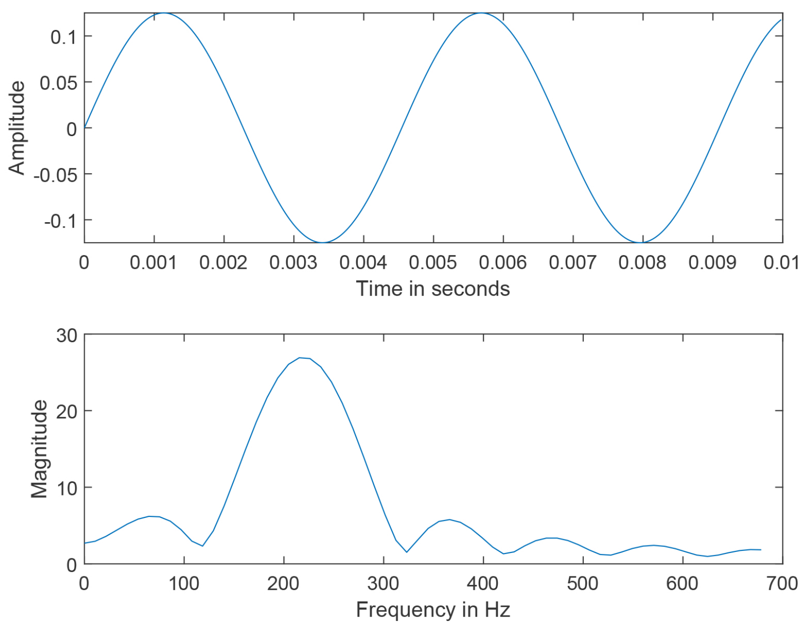


Figure 4.1: Audio wave visualization in time and frequency domains.

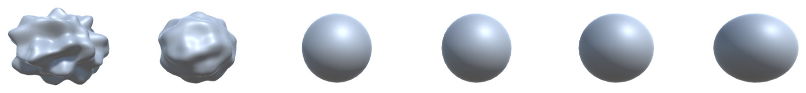
4.2 Audio-driven Interactions

The spectrum of frequencies of the sine waves that compose the audio signal can be useful in driving the vertex displacement shader. Specifically, given an incoming audio stream, an array of magnitude values were calculated across the frequency spectrum using the Fourier transform for the implementation in this study. Each organism was then programmed to respond to a specific frequency range. The shader program attached to the organism displaces the vertices of the organisms 3D mesh according to the magnitude in that frequency range.

As an illustration, Figure 4.2 to 4.7 show different frequencies of sound that modulate six 3D spherical meshes. Each mesh's vertex displacement shader is programmed to react to different frequency ranges. Specifically, the six frequency ranges are 87 Hz to 258 Hz, 259 Hz to 602 Hz, 603 Hz to 1290 Hz, 1291 Hz to 2666 Hz, 2667 Hz to 5418 Hz, and 5419 Hz to 10922 Hz.

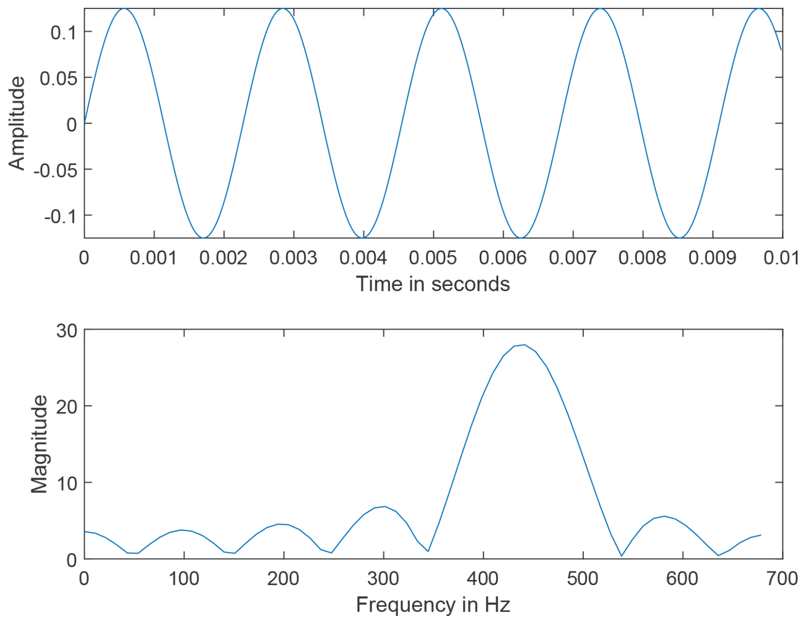


(a) 220 Hz audio wave visualization in time and frequency domains

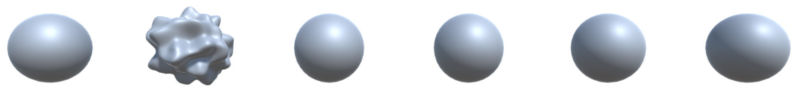


(b) 220 Hz audio modulates the shape of the sphere that has a vertex shader reacting to frequencies from 87 Hz to 258 Hz.

Figure 4.2: 220 Hz audio wave visualization and the modulation response of the vertex displacement shader.

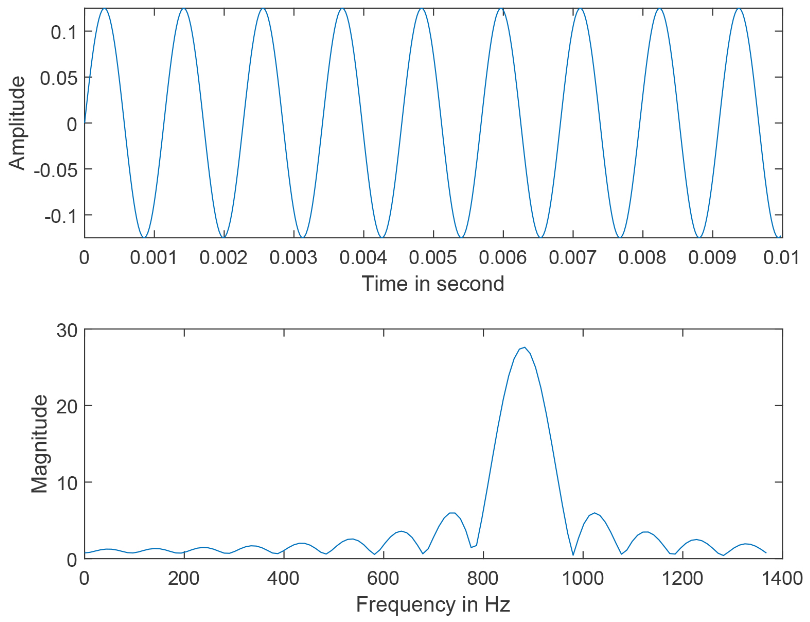


(a) 440 Hz audio wave visualization in time and frequency domains

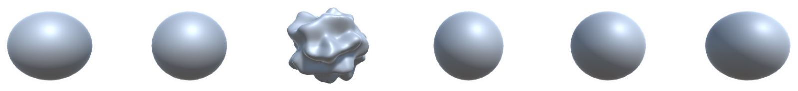


(b) 440 Hz audio modulates the shape of the sphere that has a vertex shader reacting to frequencies from 259 Hz to 602 Hz.

Figure 4.3: 440 Hz audio wave visualization and the modulation response of the vertex displacement shader.

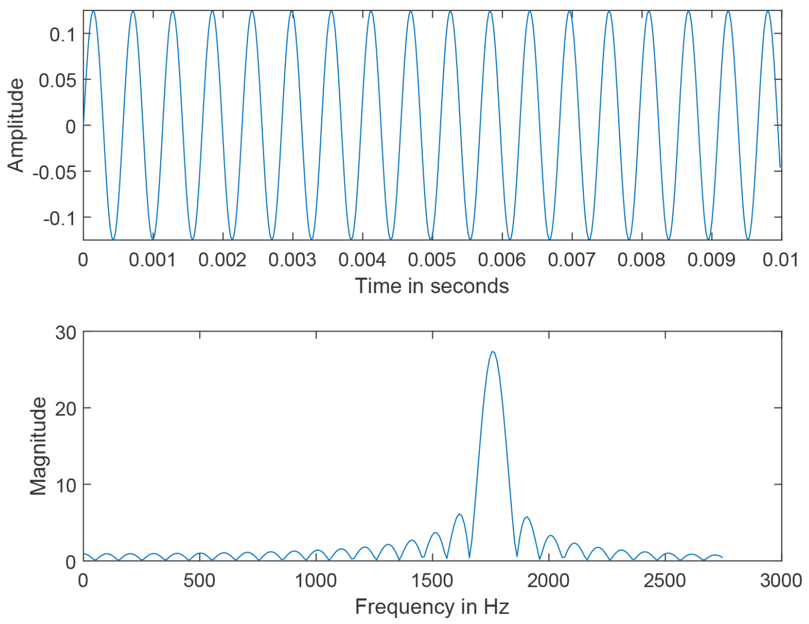


(a) 880 Hz audio wave visualization in time and frequency domains

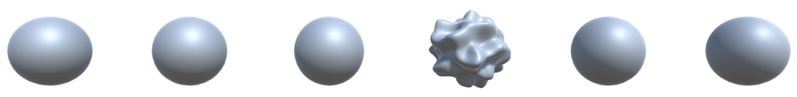


(b) 880 Hz audio modulates the shape of the sphere that has a vertex shader reacting to frequencies from 603 Hz to 1290 Hz.

Figure 4.4: 880 Hz audio wave visualization and the modulation response of the vertex displacement shader.

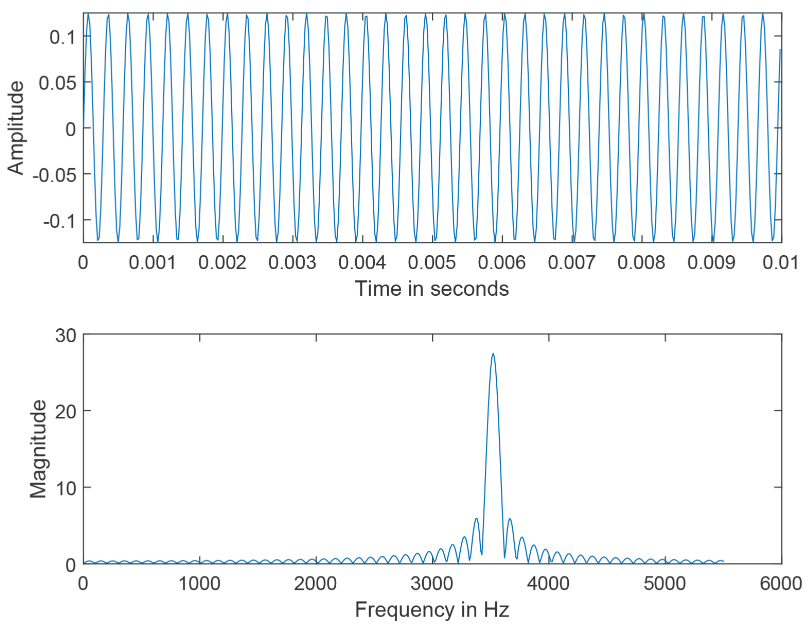


(a) 1760 Hz audio wave visualization in time and frequency domains

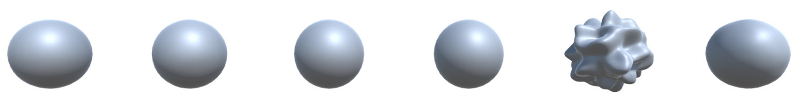


(b) 1760 Hz audio modulates the shape of the sphere that has a vertex shader reacting to frequencies from 1291 Hz to 2666 Hz.

Figure 4.5: 1760 Hz audio wave visualization and the modulation response of the vertex displacement shader.

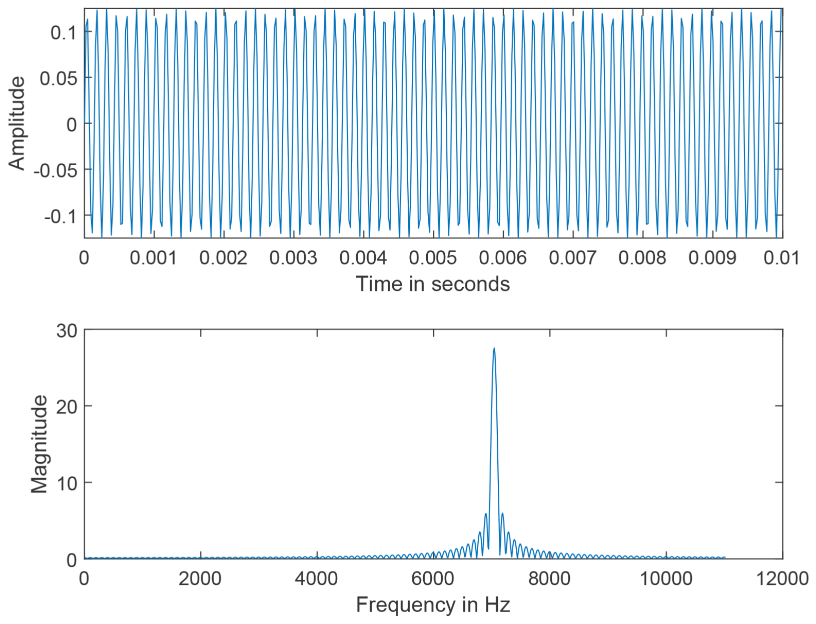


(a) 3520 Hz audio wave visualization in time and frequency domains

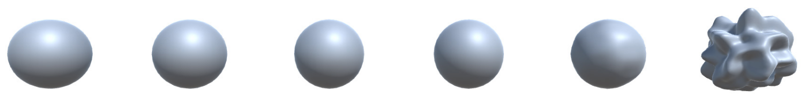


(b) 3520 Hz audio modulates the shape of the sphere that has a vertex shader reacting to frequencies from 2667 Hz to 5418 Hz.

Figure 4.6: 3520 Hz audio wave visualization and the modulation response of the vertex displacement shader.



(a) 7040 Hz audio wave visualization in time and frequency domains



(b) 7040 Hz audio modulates the shape of the sphere that has a vertex shader reacting to frequencies from 5419 Hz to 10922 Hz.

Figure 4.7: 7040 Hz audio wave visualization and the modulation response of the vertex displacement shader.

Chapter 5

Color Turbulence with Curl Noise

The flow of life energy in the natural world was represented by applying flow animation to the 3D models textures. Therefore, I developed a color turbulence technique that uses curl noise to define a variety of naturally fluid-like movements to distort and animate textures. In this chapter, I describe how the curl noise can support the color turbulence approach and present the results generated using the approach. Finally, I present some virtual environments where I applied the approach along with additional interactivity to the scenes.

5.1 Curl Noise

The ideal candidate method for flow animation was determined to be curl noise (created by Robert Bridson [63]). A curl is a mathematical operator that measures the "rotation" in a vector field. Its input is a vector field and its output is a divergence-free vector field, which means that there are neither sources nor sinks in the flow. Applying the curl operator to Perlin noise as an input generates curl noise containing varying vectors which can be used to control the direction of the flow. Therefore, shader programs that use curl noise as a flow map to distort and animate textures were developed.

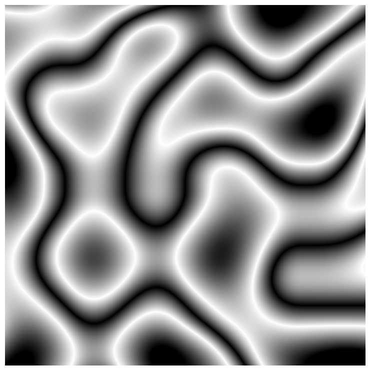
5.2 Implementation and Results

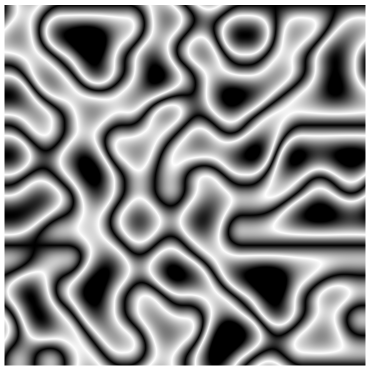
With curl noise, the flow animation can be realized through a combination of two shaders: the flow shader and the blend shader. In this section, I give the details of implementing these two shaders. The code of the flow shader and blend shader is listed in Appendix B.1 and B.2, respectively.

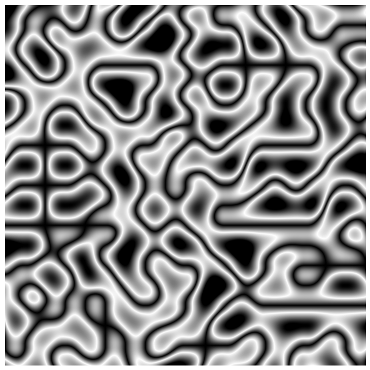
First, we need to create Perlin noise before creating curl noise because curl noise is generated by applying the curl operator to Perlin noise as an input. In the dissertation, I wrote C# scripts in the Unity game engine to create Perlin noise. Furthermore, a few parameters can affect the appearance of Perlin noise, and they are frequency, octave, lacunarity, and persistence. Consider the six types of Perlin noise in Figure 5.1 and 5.2. They were created with the same lacunarity and persistence but different octave and frequency values. Consequently, applying the curl operator to the Perlin noise generated the six corresponding types of curl noise, as also shown in Figure 5.1 and 5.2. The curl noise contains 2D vectors, with each vector's U component in the R channel and the V component in the G channel.

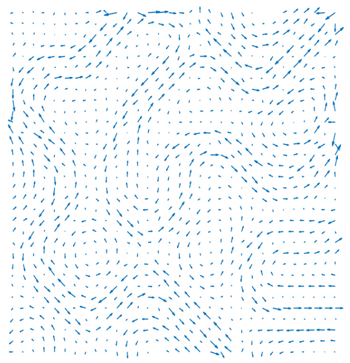
Now that I had curl noise, which stores flow vectors, I developed the flow shader and added the curl noise as a property to the shader to distort and animate an input texture. However, the input texture gets distorted more and more as time progresses. As a result, each color of the texture cannot retain its hue and dulls the other colors. To prevent the colors from becoming too mingled, the flow shader has to eventually reset the texture. A simple way to do this is to use the fractional part of the time for the animation. That is, time progresses from 0 to 1 and then returns to 0. Once time resets to 0, the input texture resets to its initial state and the animation starts over.

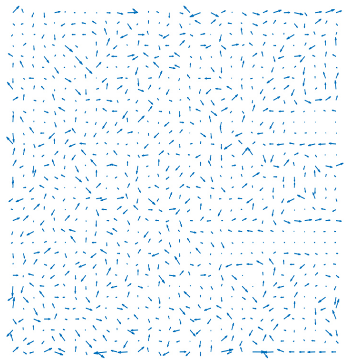
At this point, although we have flow animation, it resets every second with noticeable discontinuity. To make the entire flow seamlessly loop, we can first create two identical flow animations but offset one of them. Then, we can create a seemingly continuous flow by making one animation fade away into the other repeatedly. Note that we want the flow animations to sporadically dissolve into each other to smooth out the transition. Therefore, I added an additional Perlin noise to the flow shader to indicate when one flow animation should disappear. Finally, I developed the blend shader, which alternately displays the two flow animation.


(a) Perlin noise (octave 1, frequency 3, lacunarity 2, persistence 0.5)


(b) Perlin noise (octave 1, frequency 6, lacunarity 2, persistence 0.5)


(c) Perlin noise (octave 1, frequency 9, lacunarity 2, persistence 0.5)


(d) Curl noise generated by applying the curl operator to (a)


(e) Curl noise generated by applying the curl operator to (b)

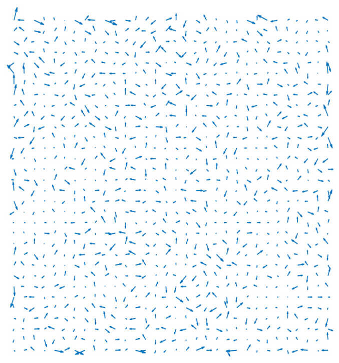
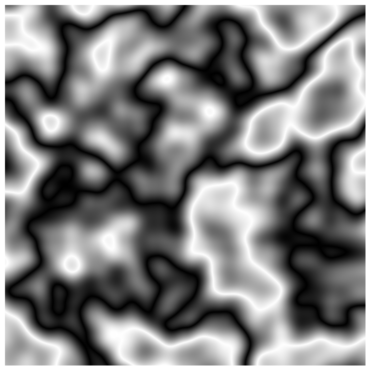
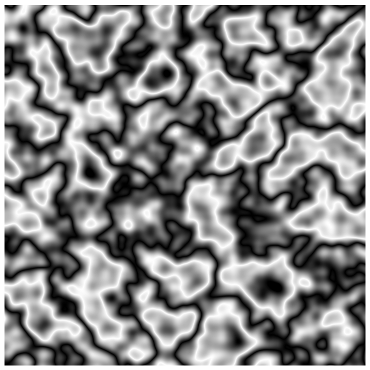
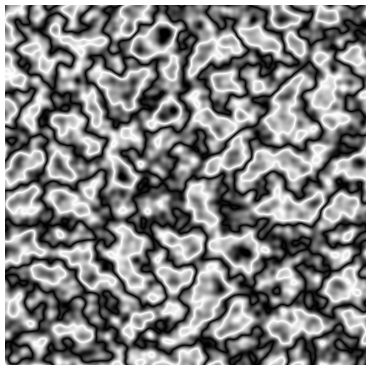
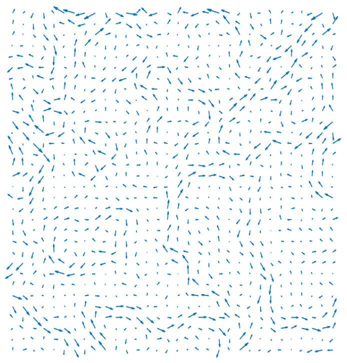

(f) Curl noise generated by applying the curl operator to (c)

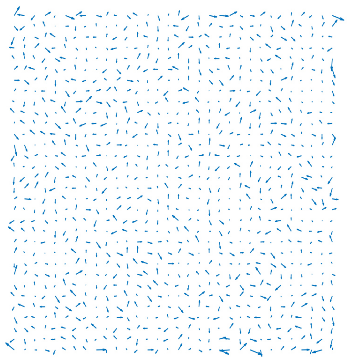
Figure 5.1: Flow maps and their corresponding Perlin noise maps that are created with the same octaves, lacunarity, and persistence but different frequencies.


(a) Perlin noise (octave 3, frequency 3, lacunarity 2, persistence 0.5)


(b) Perlin noise (octave 3, frequency 6, lacunarity 2, persistence 0.5)


(c) Perlin noise (octave 3, frequency 9, lacunarity 2, persistence 0.5)


(d) Curl noise generated by applying the curl operator to (a)


(e) Curl noise generated by applying the curl operator to (b)

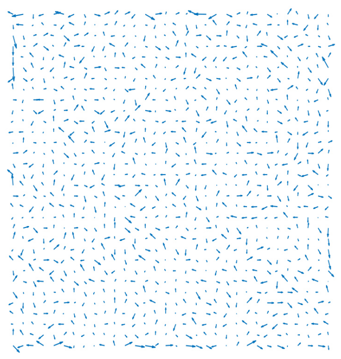
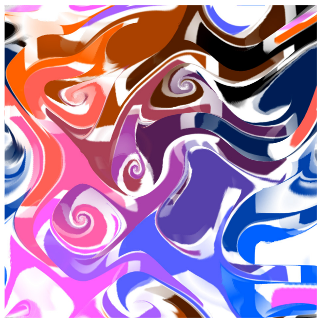
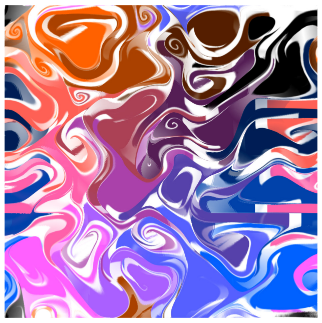
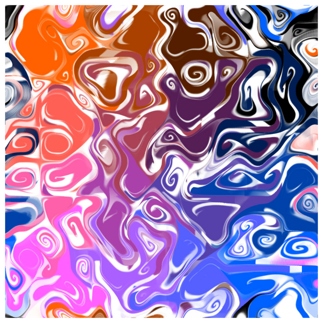

(f) Curl noise generated by applying the curl operator to (c)

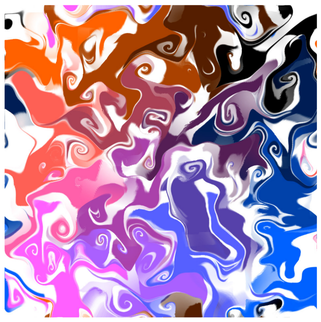
Figure 5.2: Flow maps and their corresponding Perlin noise maps that are created with the same octaves, lacunarity, and persistence but different frequencies.

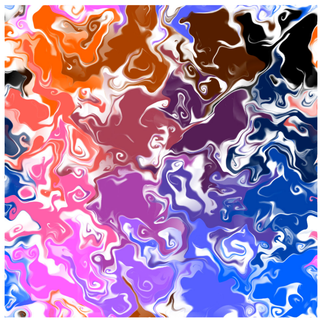
Figures 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, and 5.10 show the results of applying the color turbulence approach to different models with various flow maps.


(a) Octave 1, frequency 3, lacunarity 2, persistence 0.5


(b) Octave 1, frequency 6, lacunarity 2, persistence 0.5


(c) Octave 1, frequency 9, lacunarity 2, persistence 0.5


(d) Octave 3, frequency 3, lacunarity 2, persistence 0.5


(e) Octave 3, frequency 6, lacunarity 2, persistence 0.5

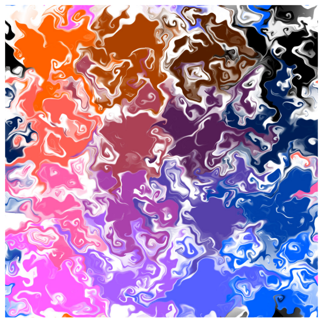
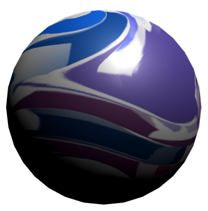
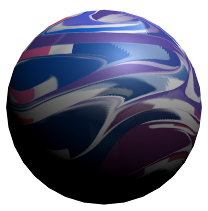
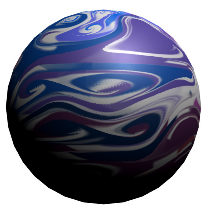
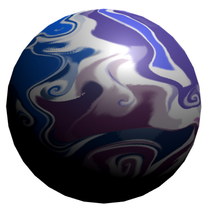

(f) Octave 3, frequency 9, lacunarity 2, persistence 0.5

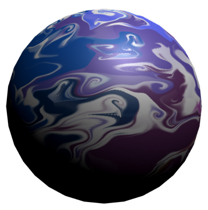
Figure 5.3: Results of applying the color turbulence method to a plane with different flow maps.


(a) Octave 1, frequency 3, lacunarity 2, persistence 0.5


(b) Octave 1, frequency 6, lacunarity 2, persistence 0.5


(c) Octave 1, frequency 9, lacunarity 2, persistence 0.5


(d) Octave 3, frequency 3, lacunarity 2, persistence 0.5


(e) Octave 3, frequency 6, lacunarity 2, persistence 0.5

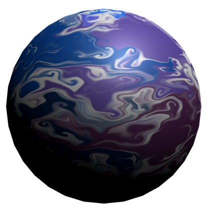
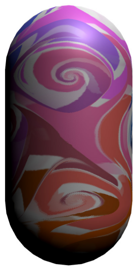
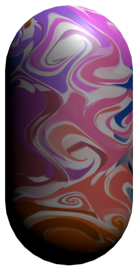
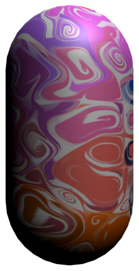
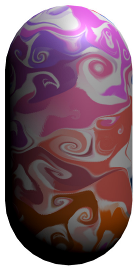

(f) Octave 3, frequency 9, lacunarity 2, persistence 0.5

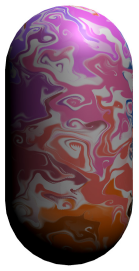
Figure 5.4: Results of applying the color turbulence method to a sphere with different flow maps.


(a) Octave 1, frequency 3, lacunarity 2, persistence 0.5


(b) Octave 1, frequency 6, lacunarity 2, persistence 0.5


(c) Octave 1, frequency 9, lacunarity 2, persistence 0.5


(d) Octave 3, frequency 3, lacunarity 2, persistence 0.5


(e) Octave 3, frequency 6, lacunarity 2, persistence 0.5

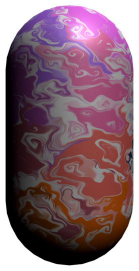
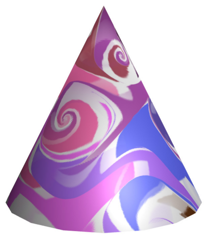
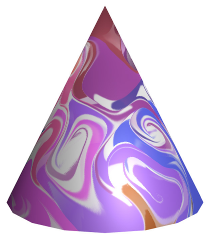
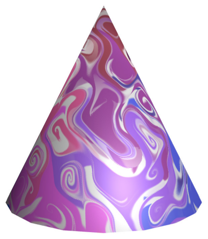
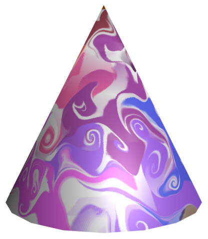

(f) Octave 3, frequency 9, lacunarity 2, persistence 0.5

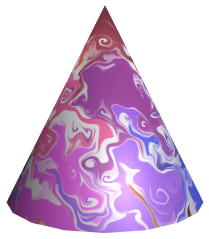
Figure 5.5: Results of applying the color turbulence method to a capsule with different flow maps.


(a) Octave 1, frequency 3, lacunarity 2, persistence 0.5


(b) Octave 1, frequency 6, lacunarity 2, persistence 0.5


(c) Octave 1, frequency 9, lacunarity 2, persistence 0.5


(d) Octave 3, frequency 3, lacunarity 2, persistence 0.5


(e) Octave 3, frequency 6, lacunarity 2, persistence 0.5

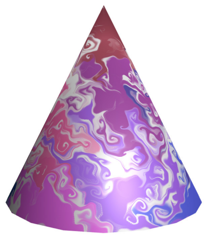
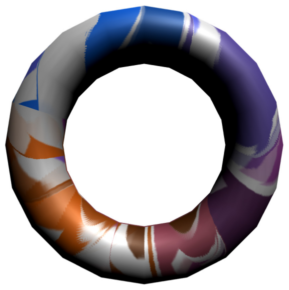
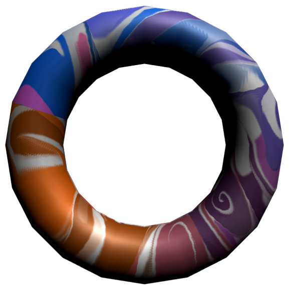
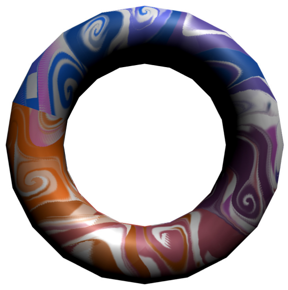
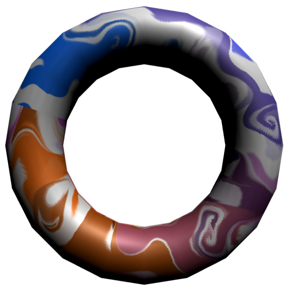

(f) Octave 3, frequency 9, lacunarity 2, persistence 0.5

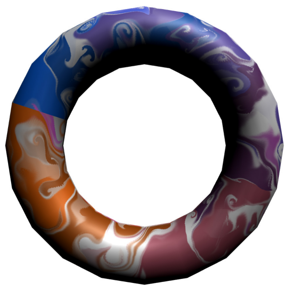
Figure 5.6: Results of applying the color turbulence method to a cone with different flow maps.


(a) Octave 1, frequency 3, lacunarity 2, persistence 0.5


(b) Octave 1, frequency 6, lacunarity 2, persistence 0.5


(c) Octave 1, frequency 9, lacunarity 2, persistence 0.5


(d) Octave 3, frequency 3, lacunarity 2, persistence 0.5


(e) Octave 3, frequency 6, lacunarity 2, persistence 0.5

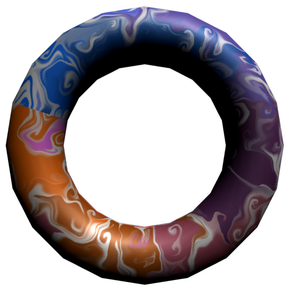
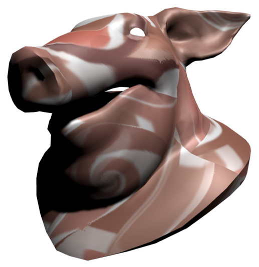
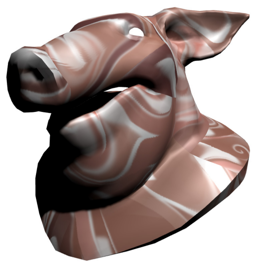
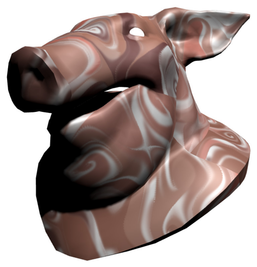
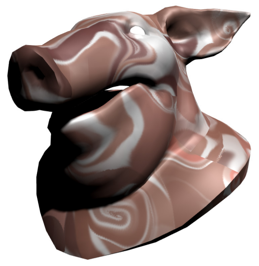

(f) Octave 3, frequency 9, lacunarity 2, persistence 0.5

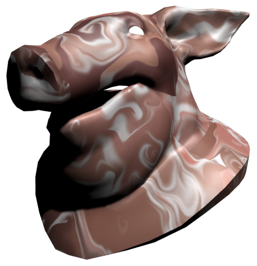
Figure 5.7: Results of applying the color turbulence method to a torus with different flow maps.


(a) Octave 1, frequency 3, lacunarity 2, persistence 0.5


(b) Octave 1, frequency 6, lacunarity 2, persistence 0.5


(c) Octave 1, frequency 9, lacunarity 2, persistence 0.5


(d) Octave 3, frequency 3, lacunarity 2, persistence 0.5


(e) Octave 3, frequency 6, lacunarity 2, persistence 0.5

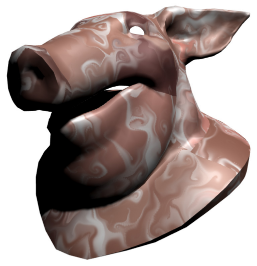
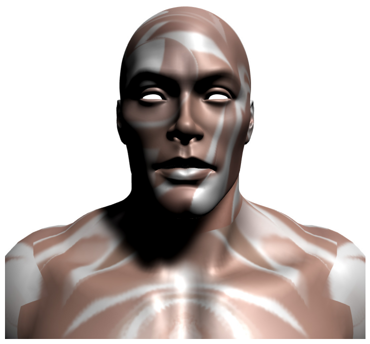
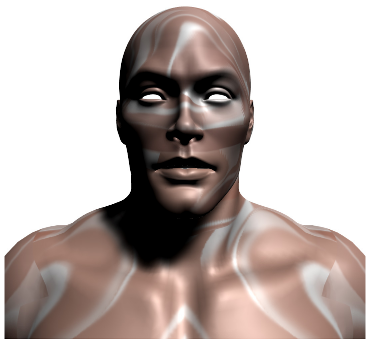
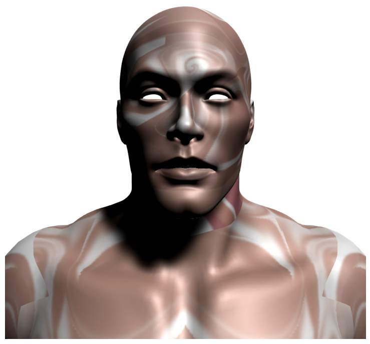
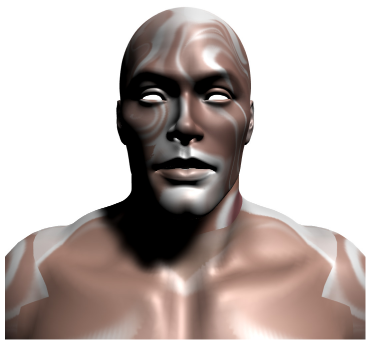

(f) Octave 3, frequency 9, lacunarity 2, persistence 0.5

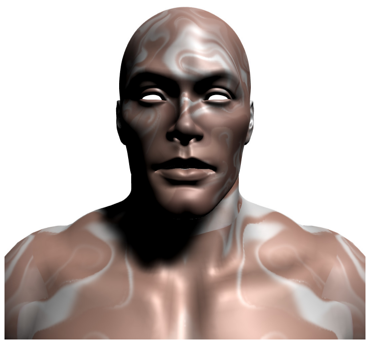
Figure 5.8: Results of applying the color turbulence method to a pig's head with different flow maps.


(a) Octave 1, frequency 3, lacunarity 2, persistence 0.5


(b) Octave 1, frequency 6, lacunarity 2, persistence 0.5


(c) Octave 1, frequency 9, lacunarity 2, persistence 0.5


(d) Octave 3, frequency 3, lacunarity 2, persistence 0.5


(e) Octave 3, frequency 6, lacunarity 2, persistence 0.5

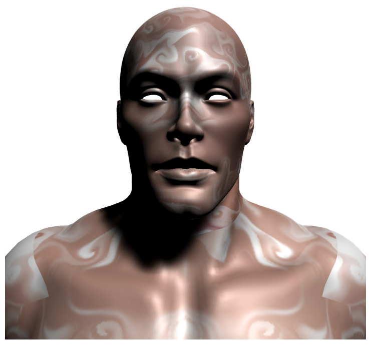
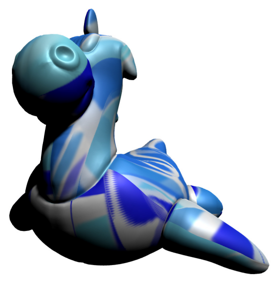
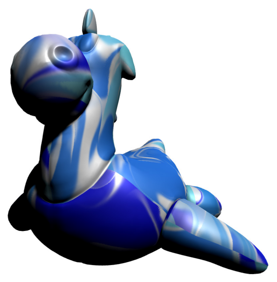

(f) Octave 3, frequency 9, lacunarity 2, persistence 0.5

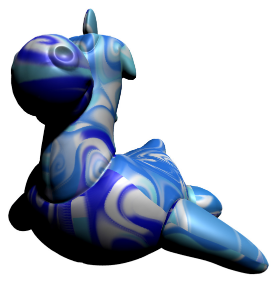
Figure 5.9: Results of applying the color turbulence method to a human body with different flow maps.



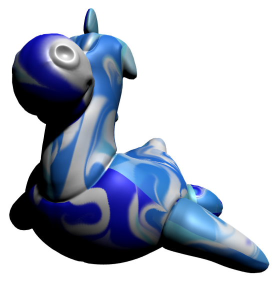
(a) Octave 1, frequency 3, lacunarity 2, persistence 0.5



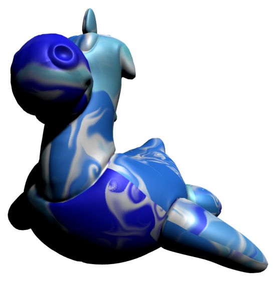
(b) Octave 1, frequency 6, lacunarity 2, persistence 0.5



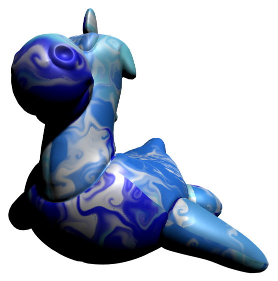
(c) Octave 1, frequency 9, lacunarity 2, persistence 0.5



(d) Octave 3, frequency 3, lacunarity 2, persistence 0.5



(e) Octave 3, frequency 6, lacunarity 2, persistence 0.5

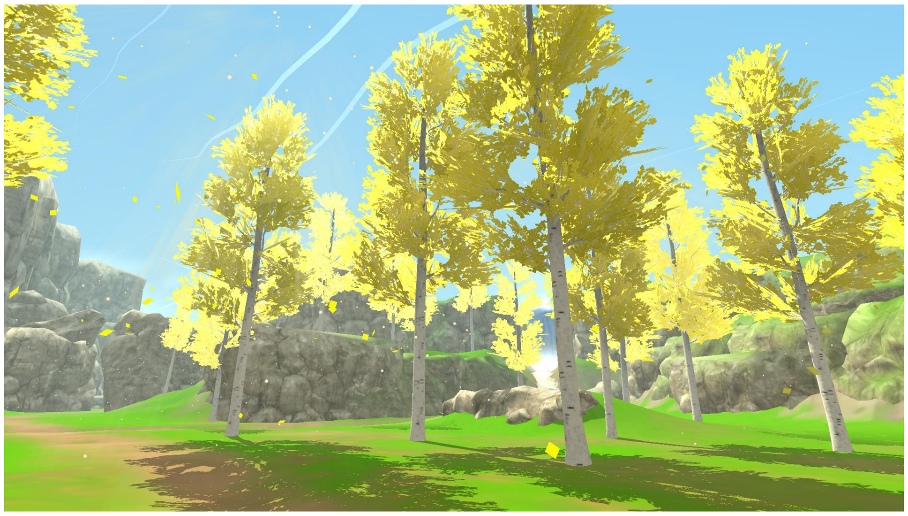


(f) Octave 3, frequency 9, lacunarity 2, persistence 0.5

Figure 5.10: Results of applying the color turbulence method to a toy with different flow maps.

5.3 VR Integration and Interactivity

In this section, I present some virtual environments where a given 3D scene was procedurally transformed into an expressive virtual world with animated textures that resemble fluid motions by using the color turbulence technique (Figures 5.11, 5.12, 5.13, and 5.14). In addition, the shader programs described in the previous section allow viewers to interact with the textures by injecting colors into them (Figure 5.15). The original 3D scene is from the Fantasy Adventure Environment asset by Staggart Creations [64], and all the 3D meshes, lighting, and particles are included in the asset.


(a) A given scene in VR

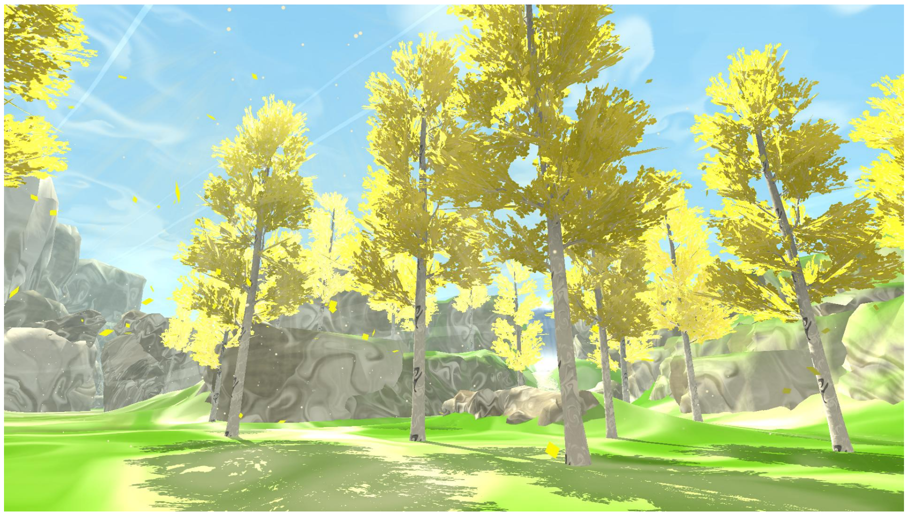
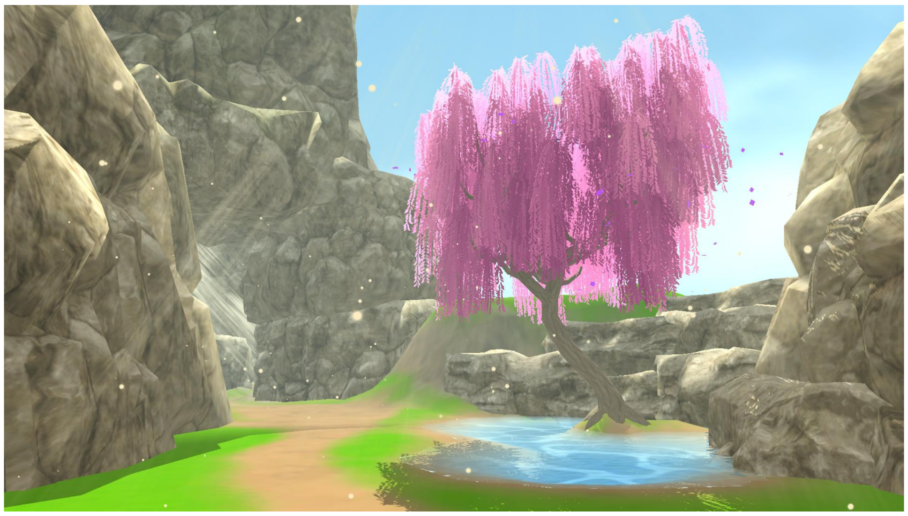
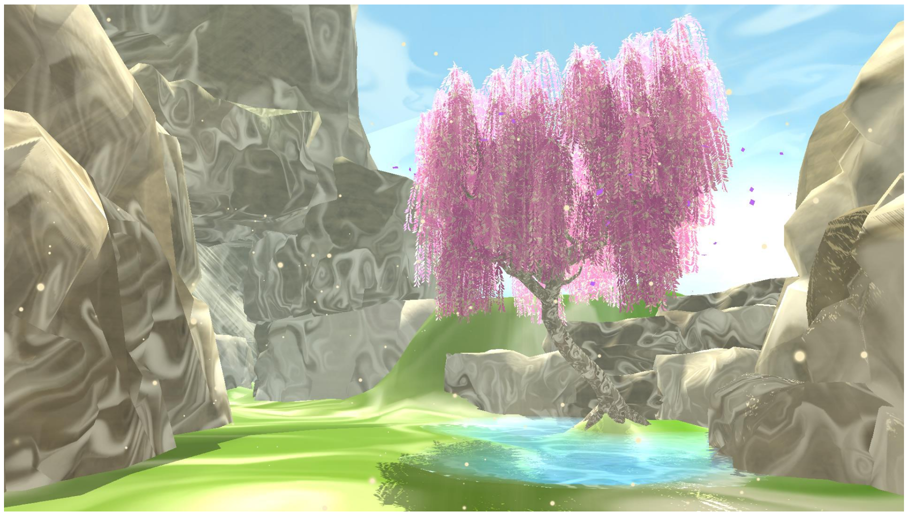

(b) The resulting scene after applying the color turbulence method

Figure 5.11: Screenshots showing a scene in VR before and after being assigned the color turbulence method. The method paints moving and swirling patterns onto the barks, rocks, ground, and sky.

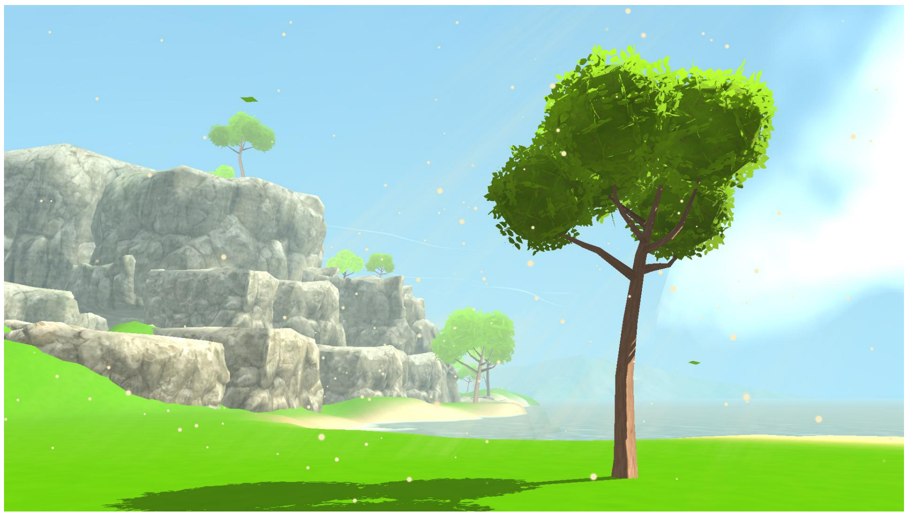


(a) A given 3D scene

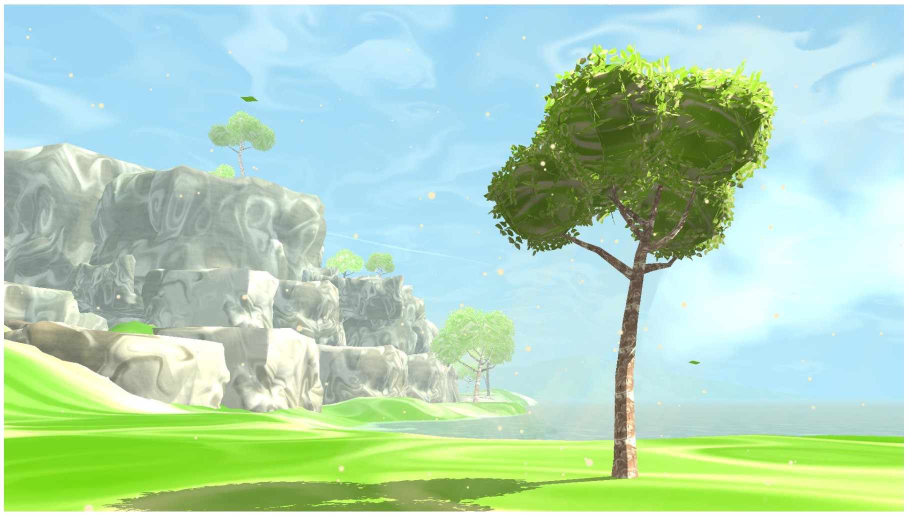


(b) The resulting scene after applying the color turbulence method

Figure 5.12: Screenshots showing a scene in VR before and after being assigned the color turbulence method. The method paints moving and swirling patterns onto the leaves, barks, rocks, ground, and sky.

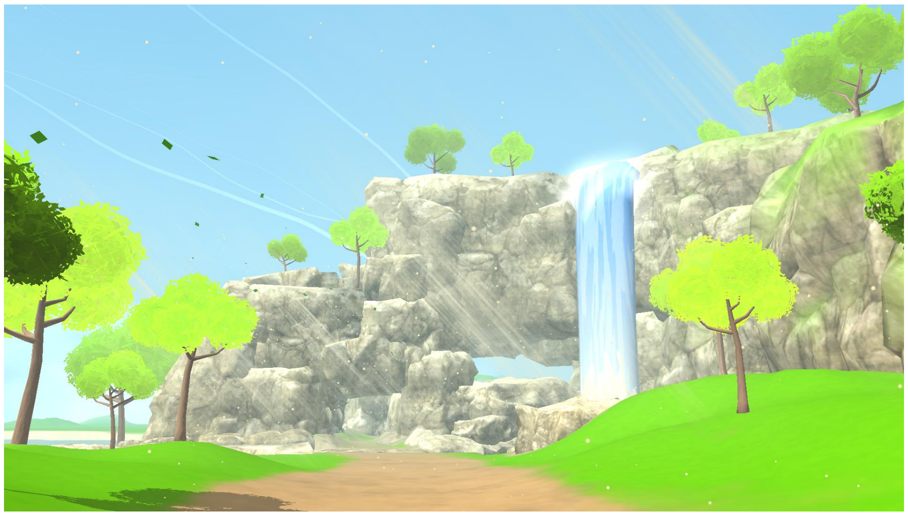


(a) A given 3D scene

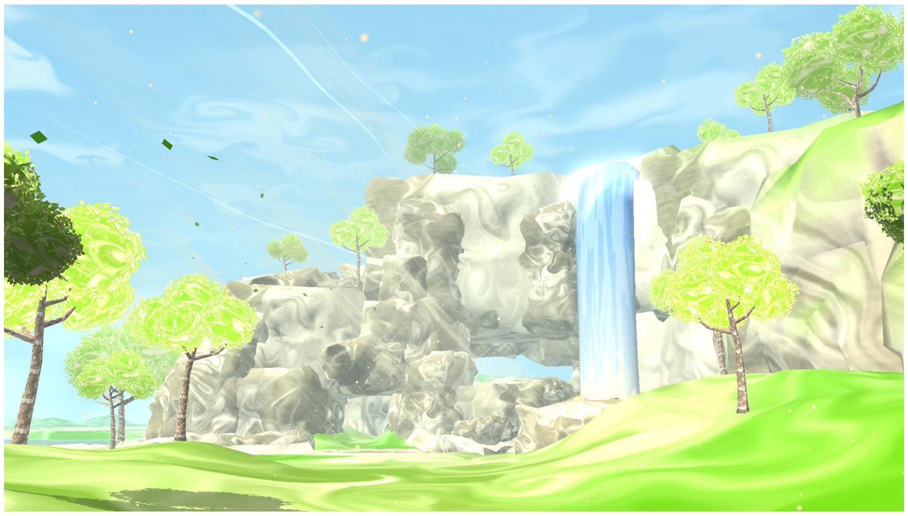


(b) The resulting scene after applying the color turbulence method

Figure 5.13: Screenshots showing a scene in VR before and after being assigned the color turbulence method. The method paints moving and swirling patterns onto the leaves, barks, rocks, ground, and sky.

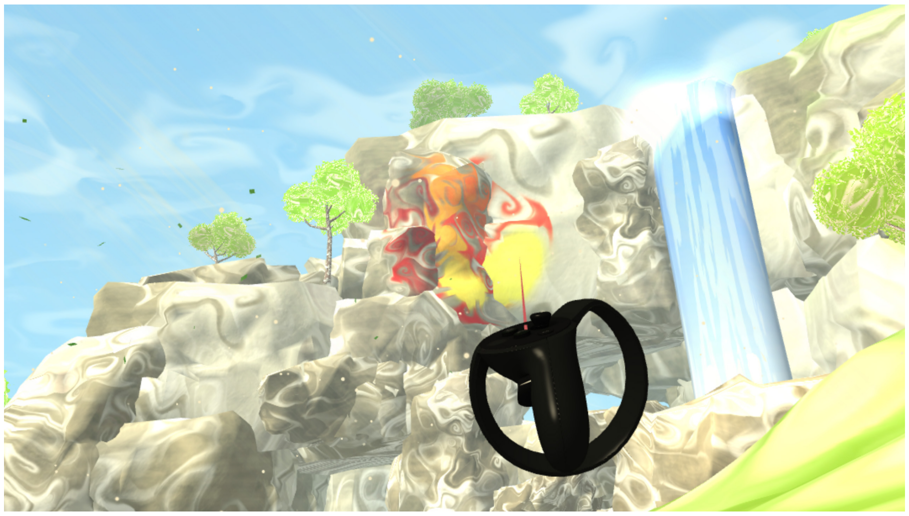


(a) A given 3D scene

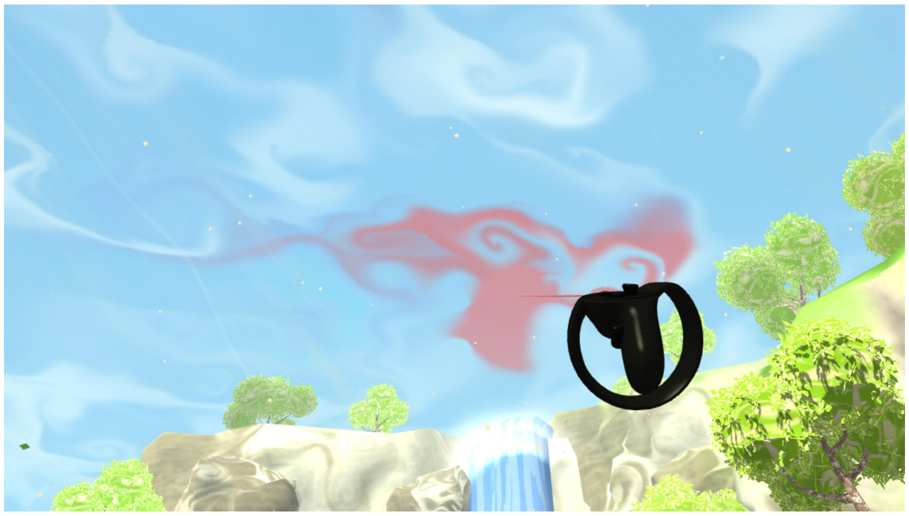


(b) The resulting scene after applying the color turbulence method

Figure 5.14: Screenshots showing a scene in VR before and after being assigned the color turbulence method. The method paints moving and swirling patterns onto the leaves, barks, rocks, ground, and sky.



(a)



(b)

Figure 5.15: Screenshots showing that an attendee interacts with the swirling texture of different objects in VR.

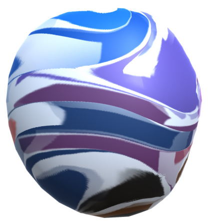
Chapter 6

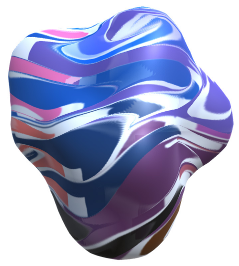
An Assembly of the Two Techniques

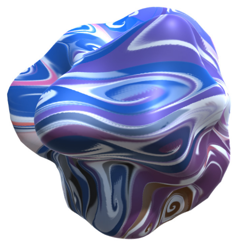
In previous chapters, images and examples are presented to show the results of individual technique. This chapter shows the results of combining both the vertex displacement technique and the color turbulence technique, beginning with applying both techniques to various 3D meshes. Then, a VR system that the two techniques were integrated into is available to demonstrate the results of these techniques with a number of 3D models in a fully immersive environment.

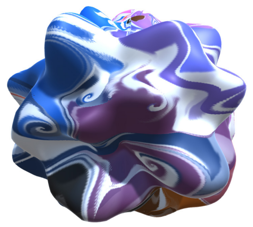
6.1 Combination of Vertex Displacement and Color Turbulence

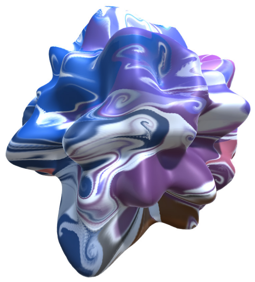
Combining the vertex displacement technique and the color turbulence technique can create an enormous variations in organic 3D models with turbulence in their textures. Figures 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, and 6.7 show the results of combining both techniques and assigning them to different models.


(a) Frequency 1.0, scale 0.25, flow map settings: octave 1, frequency 3, lacunarity 2, persistence 0.5


(b) Frequency 2.0, scale 0.25; flow map settings: octave 1, frequency 6, lacunarity 2, persistence 0.5


(c) Frequency 3.0, scale 0.25; flow map settings: octave 1, frequency 9, lacunarity 2, persistence 0.5


(d) Frequency 4.0, scale 0.25; flow map settings: octave 3, frequency 3, lacunarity 2, persistence 0.5


(e) Frequency 5.0, scale 0.25; flow map settings: octave 3, frequency 6, lacunarity 2, persistence 0.5

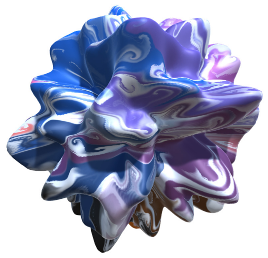
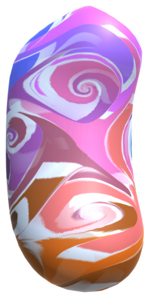
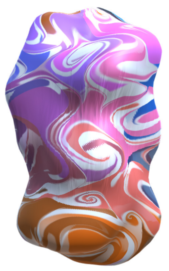
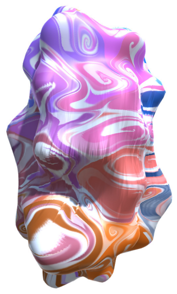
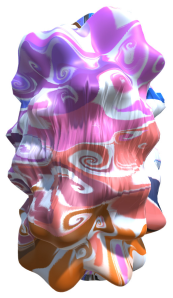

(f) Frequency 6.0, scale 0.25; flow map settings: octave 3, frequency 9, lacunarity 2, persistence 0.5

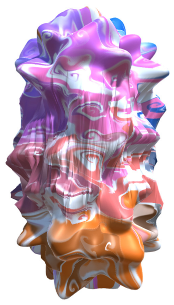
Figure 6.1: Results of applying the vertex displacement method with different frequencies and different flow maps to a sphere.


(a) Frequency 1.0, scale 0.25; flow map settings: octave 1, frequency 3, lacunarity 2, persistence 0.5


(b) Frequency 2.0, scale 0.25; flow map settings: octave 1, frequency 6, lacunarity 2, persistence 0.5


(c) Frequency 3.0, scale 0.25; flow map settings: octave 1, frequency 9, lacunarity 2, persistence 0.5


(d) Frequency 4.0, scale 0.25; flow map settings: octave 3, frequency 3, lacunarity 2, persistence 0.5


(e) Frequency 5.0, scale 0.25; flow map settings: octave 3, frequency 6, lacunarity 2, persistence 0.5

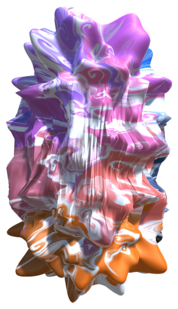
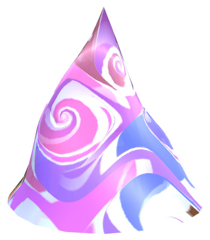
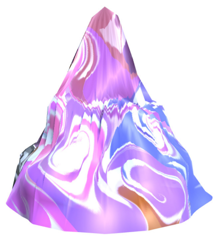
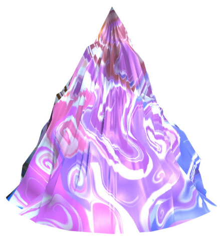
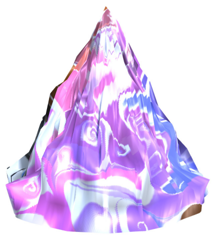

(f) Frequency 6.0, scale 0.25; flow map settings: octave 3, frequency 9, lacunarity 2, persistence 0.5

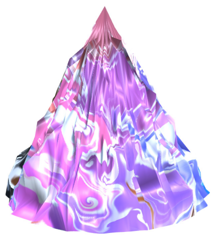
Figure 6.2: Results of applying the vertex displacement method with different frequencies and different flow maps to a capsule.


(a) Frequency 1.0, scale 0.25; flow map settings: octave 1, frequency 3, lacunarity 2, persistence 0.5


(b) Frequency 2.0, scale 0.25; flow map settings: octave 1, frequency 6, lacunarity 2, persistence 0.5


(c) Frequency 3.0, scale 0.25; flow map settings: octave 1, frequency 9, lacunarity 2, persistence 0.5


(d) Frequency 4.0, scale 0.25; flow map settings: octave 3, frequency 3, lacunarity 2, persistence 0.5


(e) Frequency 5.0, scale 0.25; flow map settings: octave 3, frequency 6, lacunarity 2, persistence 0.5

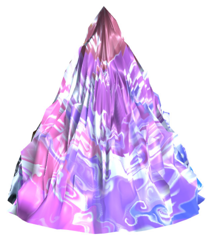
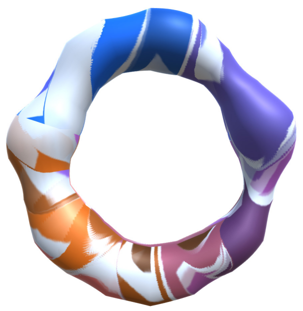
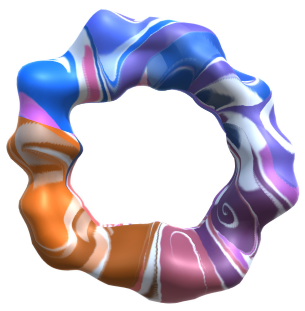
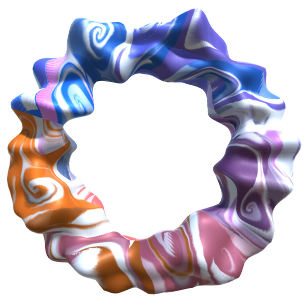
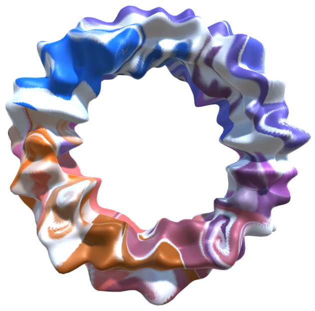

(f) Frequency 6.0, scale 0.25; flow map settings: octave 3, frequency 9, lacunarity 2, persistence 0.5

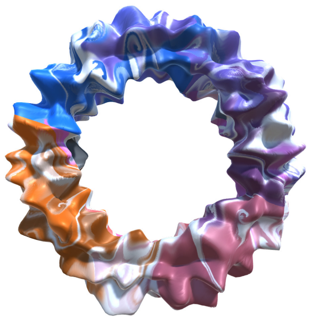
Figure 6.3: Results of applying the vertex displacement method with different frequencies and different flow maps to a cone.


(a) Frequency 1.0, scale 0.25; flow map settings: octave 1, frequency 3, lacunarity 2, persistence 0.5


(b) Frequency 2.0, scale 0.25; flow map settings: octave 1, frequency 6, lacunarity 2, persistence 0.5


(c) Frequency 3.0, scale 0.25; flow map settings: octave 1, frequency 9, lacunarity 2, persistence 0.5


(d) Frequency 4.0, scale 0.25; flow map settings: octave 3, frequency 3, lacunarity 2, persistence 0.5


(e) Frequency 5.0, scale 0.25; flow map settings: octave 3, frequency 6, lacunarity 2, persistence 0.5

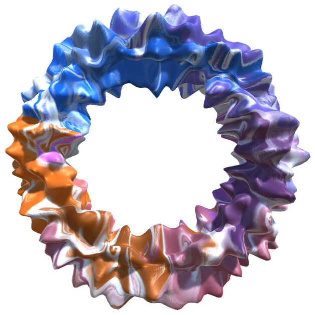
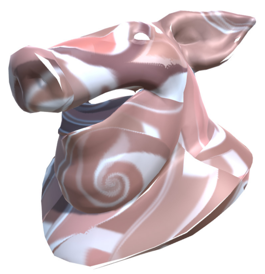
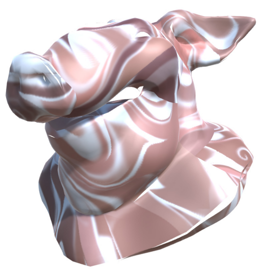
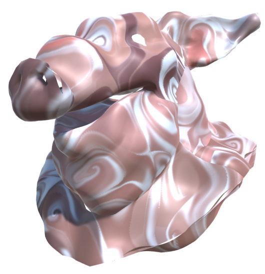
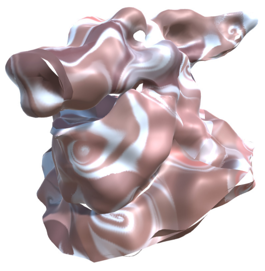

(f) Frequency 6.0, scale 0.25; flow map settings: octave 3, frequency 9, lacunarity 2, persistence 0.5

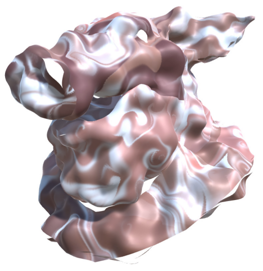
Figure 6.4: Results of applying the vertex displacement method with different frequencies and different flow maps to a torus.


(a) Frequency 1.0, scale 0.1; flow map settings: octave 1, frequency 3, lacunarity 2, persistence 0.5


(b) Frequency 2.0, scale 0.1; flow map settings: octave 1, frequency 6, lacunarity 2, persistence 0.5


(c) Frequency 3.0, scale 0.1; flow map settings: octave 1, frequency 9, lacunarity 2, persistence 0.5


(d) Frequency 4.0, scale 0.1; flow map settings: octave 3, frequency 3, lacunarity 2, persistence 0.5


(e) Frequency 5.0, scale 0.1; flow map settings: octave 3, frequency 6, lacunarity 2, persistence 0.5

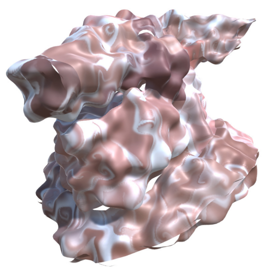
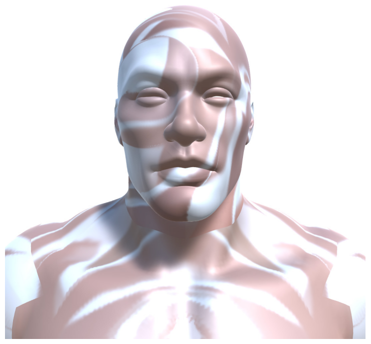
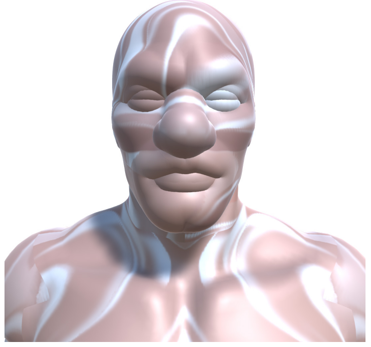
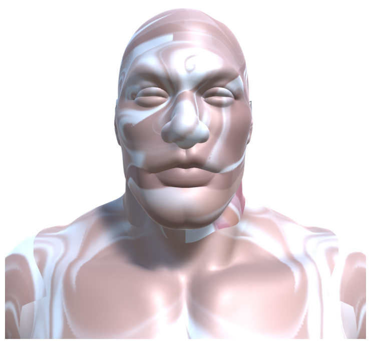
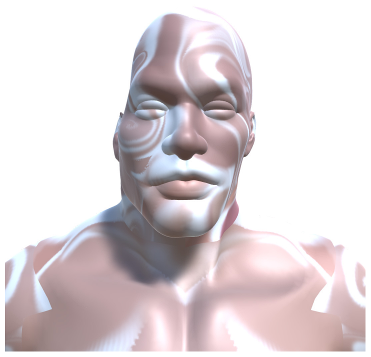

(f) Frequency 6.0, scale 0.1; flow map settings: octave 3, frequency 9, lacunarity 2, persistence 0.5

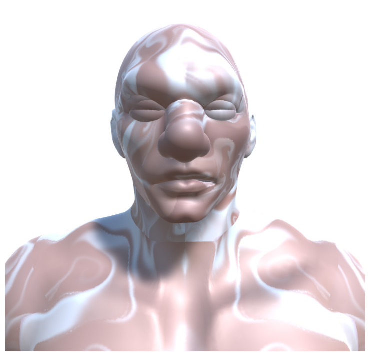
Figure 6.5: Results of applying the vertex displacement method with different frequencies and different flow maps to a pig's head.


(a) Frequency 1.0, scale 0.25; flow map settings: octave 1, frequency 3, lacunarity 2, persistence 0.5


(b) Frequency 2.6, scale 0.25; flow map settings: octave 1, frequency 6, lacunarity 2, persistence 0.5


(c) Frequency 3.0, scale 0.25; flow map settings: octave 1, frequency 9, lacunarity 2, persistence 0.5


(d) Frequency 4.8, scale 0.25; flow map settings: octave 3, frequency 3, lacunarity 2, persistence 0.5


(e) Frequency 5.0, scale 0.25; flow map settings: octave 3, frequency 6, lacunarity 2, persistence 0.5

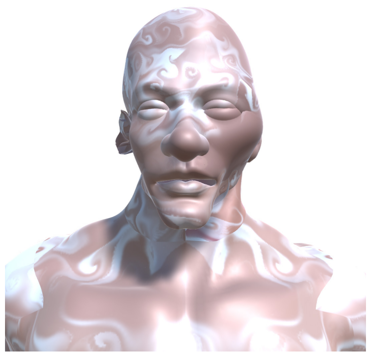
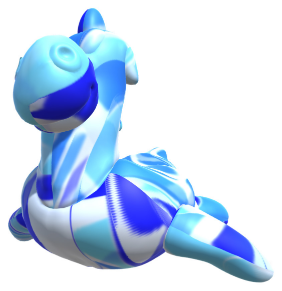
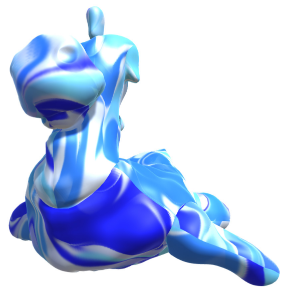

(f) Frequency 9.0, scale 0.25; flow map settings: octave 3, frequency 9, lacunarity 2, persistence 0.5

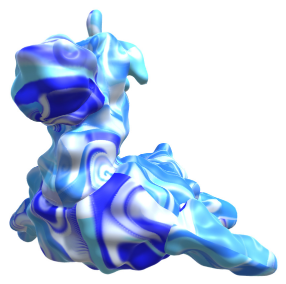
Figure 6.6: Results of applying the vertex displacement method with different frequencies and different flow maps to a human body.



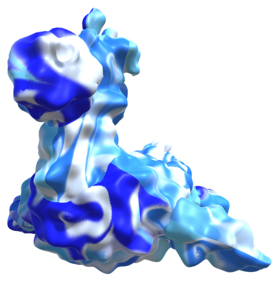
(a) Frequency 2.0, scale 0.25; flow map settings: octave 1, frequency 3, lacunarity 2, persistence 0.5



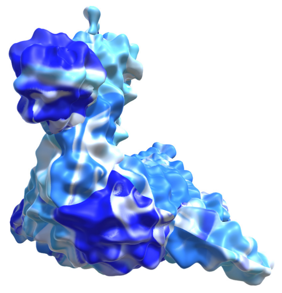
(b) Frequency 4.0, scale 0.25; flow map settings: octave 1, frequency 6, lacunarity 2, persistence 0.5



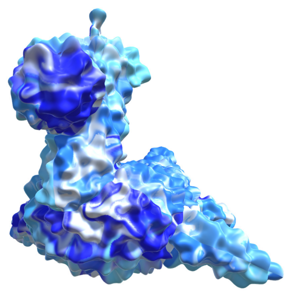
(c) Frequency 6.0, scale 0.25; flow map settings: octave 1, frequency 9, lacunarity 2, persistence 0.5



(d) Frequency 8.0, scale 0.25; flow map settings: octave 3, frequency 3, lacunarity 2, persistence 0.5



(e) Frequency 10.0, scale 0.25; flow map settings: octave 3, frequency 6, lacunarity 2, persistence 0.5



(f) Frequency 12.0, scale 0.25; flow map settings: octave 3, frequency 9, lacunarity 2, persistence 0.5

Figure 6.7: Results of applying the vertex displacement method with different frequencies and different flow maps to a toy.

6.2 Narrative Demo

In this section, an example to demonstrate how the approach could serve as a narrative in VR was constructed (Figure 6.8). The example is an interactive abstract virtual environment that seeks to imagine the natural world as alive in a way that is different from that perceived by human beings. For instance, turbulent flow exists in all kinds of virtual objects (inhabitants, rocks, and land), thereby representing the vitality or energy flow of the natural world. Furthermore, all agents have the capacity to sense acoustic signals and deform their shapes according to the volumes and frequencies of the sounds (Figure 6.9). Participants can also interact with the virtual objects which respond by altering their colors (Figure 6.10). To summarize, the aim of the example is to awaken the participants' imagination and raise their awareness of the responsive natural world around them through animated shapes, textures, and motions.

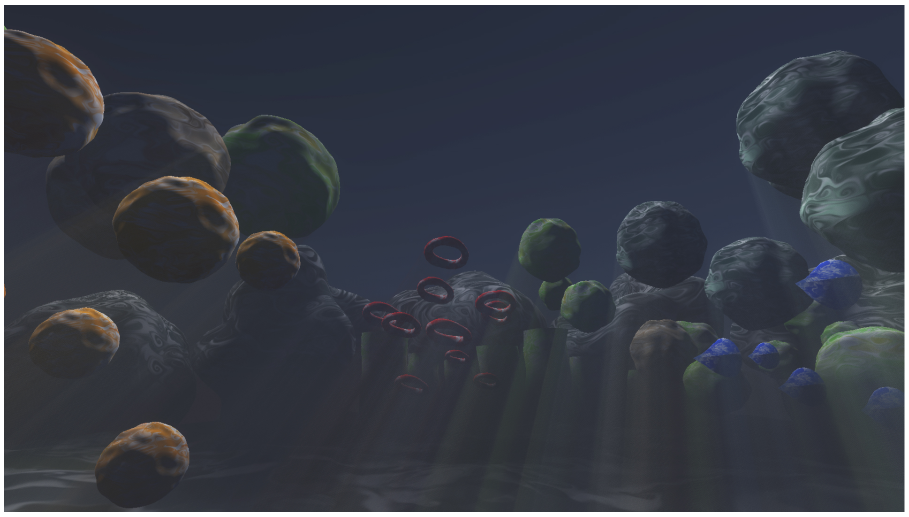
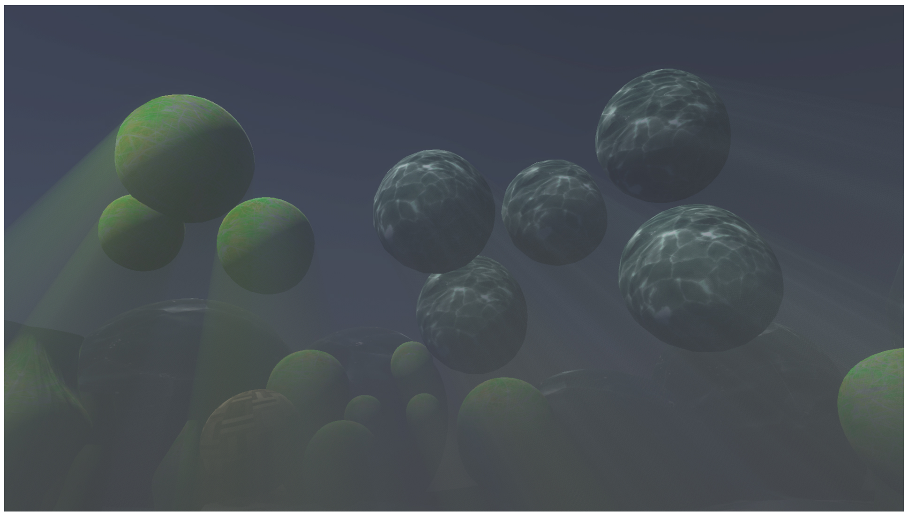
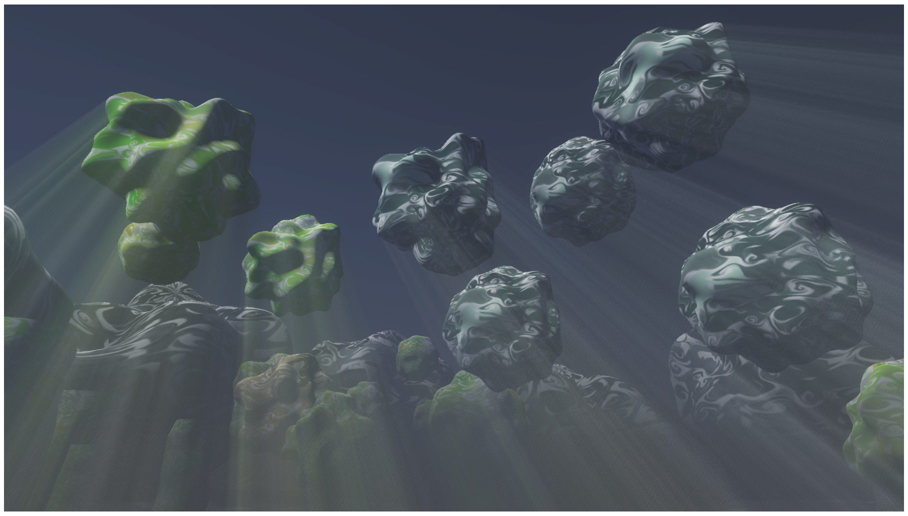


Figure 6.8: A screenshot showing a scene from the narrative demo.

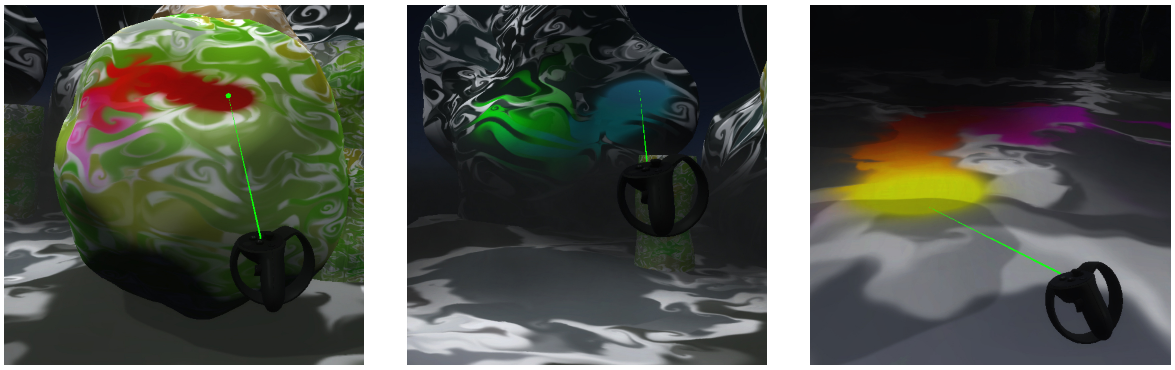


(a)



(b)

Figure 6.9: Screenshots showing that the organisms modulate their shapes by sound. Their shapes change as the volume of the sound goes from gentle (top) to loud (bottom).



(a) (b) (c)

Figure 6.10: Screenshots showing that an attendee interacts with the animated textures of different objects.

Chapter 7

Evaluation

This chapter offers technical and aesthetic evaluations of vertex displacement and color turbulence techniques. Herein, the chapter is divided into two sections. The first section reports on the performance of each technique. The performance is evaluated by measuring the frame rate in VR. The performance of the color turbulence method is affected by several factors: the resolution of textures, the amount of time interval to animate, and the number of 3D models in the scene. In contrast, the performance of the vertex displacement method is affected by the number of 3D models.

The second section contains the aesthetic evaluation. The techniques presented in the paper are based on Perlin noise and curl noise generation algorithms. Since many different types of noise functions exist, the results generated by other types of noise functions were compared according to qualitative metrics of smoothness and form to understand the ways in which these various noise functions differ.

7.1 Technical Evaluation

Measuring the frame rate is a common way to analyze the performance and quality of the experience in VR. Maintaining a frame rate similar to the refresh rate of the display used in the VR headset is essential for a quality VR experience. If the frame rate drops below the headset's refresh rate, it can degrade the experience to the point of motion discomfort for the user [65, 66].

Since only a few parameters vary the visual quality and performance of VR experience, I performed experiments with each parameter with different values and measured the frame rate under each value. Moreover, to see how stable the frame rate is during the rendering process for each parameter value, I measured the frame rate five times and calculated its maximum, minimum, and average values.

Firstly, for the color turbulence approach, I used the following parameters:

- Resolution of render texture: Render textures are textures that can be updated at runtime. In my case, the animation of turbulence is rendered into render textures. In general, the higher the texture's resolution is, the less pixelated the rendered image looks and the greater amount of computational effort is involved.
- Resolution of initial texture. Initial textures define the color map for the material as the animation of turbulence begins.
- Resolution of flow texture. Flow textures contain 2D vectors that indicate the flow directions across the surface of the material.
- Resolution of noise texture. My approach achieves seamless looping by blending two color maps. Noise textures control when one map should transit the other.
- Amount of time suspended for the system to calculate the animation's next state.

This is the time interval for which animation will pause until the time interval ends and then resume where it left off on the following frame.

- Number of 3D models in the scene. Typically, 3D models are constructed with many little polygons. A virtual environment with a high poly count delivers more detailed graphics but could reduce the performance.

Second, the limiting factor for using the vertex displacement approach in terms of performance is the number of 3D models in the scene.

The virtual environment that I used for the experiments was a plain 3D scene with one default direction light from the Unity game engine. All the frame rates were measured on a 2.20 GHz Intel Xeon desktop with a 64 GB RAM and a dual NVIDIA GeForce GTX 1080 Ti graphics card. Furthermore, since I used an Oculus Rift VR headset, the recommended frame rate was 90 hz [67].

Figures 7.1 to 7.6 display the frame rates versus each parameter for the color turbulence approach, while Tables 7.1 to 7.6 show their corresponding statistics logged during runtime. The frame rate begins to drop when the resolution of texture or the number of 3D models increases, and this is shown in Figures 7.1 to 7.4 and Tables 7.1 to 7.4. In addition, examining the frame rate results in Figures 7.6 and Table 7.6, we see that the frame rate is more likely to meet the recommended 90 frames per second when more suspended time is allowed.

For the vertex displacement approach, the application can meet a consistent 90 frames per second, even though the number of 3D models increases, as shown in Figure 7.7 and Table 7.7. In contrast, the performance of the color turbulence approach is more prone to drop in frame rate due to an increasing number of 3D models.

Table 7.1: Frame rate results for different resolutions of render texture for the color turbulence method (see results in Figure 7.1).

	64	128	256	512	1024	2048
Maximum frame rate	91.15	92.21	95.03	95.09	96.95	90.11
Average frame rate	86.94	86.47	93.43	93.62	95.21	49.86
Minimum frame rate	82.31	82.71	91.88	92.11	94.28	22.54

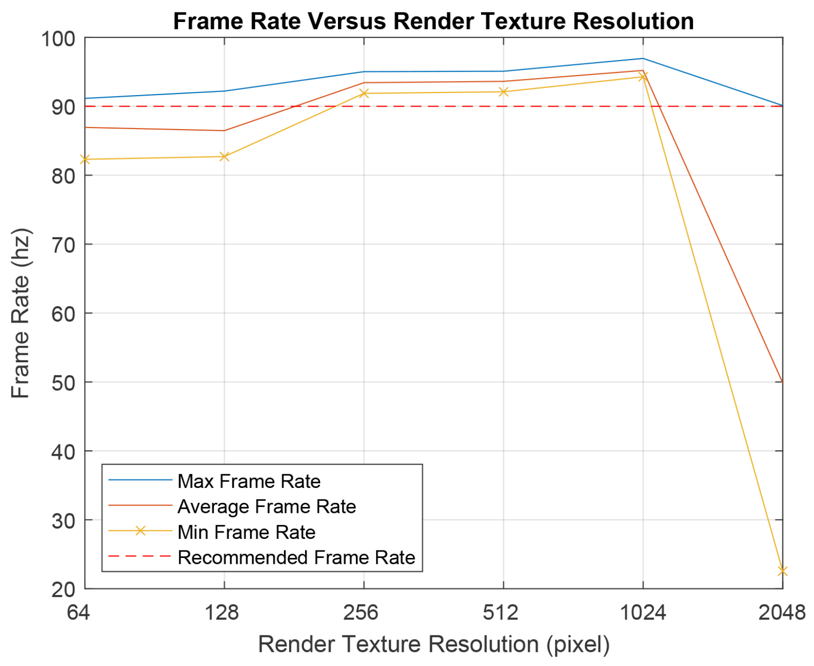


Figure 7.1: The frame rate versus the resolution of render texture for the color turbulence method. Table 7.1 shows the quantitative descriptions. The test environment contains 4,800 triangles. The test was performed five times to calculate the maximum, minimum, and average frame rate values.

Table 7.2: Frame rate results for different resolutions of initial texture for the color turbulence method (see results in Figure 7.2).

	64	128	256	512	1024	2048
Maximum frame rate	90.01	90.2	90.14	96.18	95.93	47.55
Average frame rate	89.91	89.95	90.06	94.61	93.44	46.93
Minimum frame rate	89.81	89.83	90.01	93.7	90.69	46.31

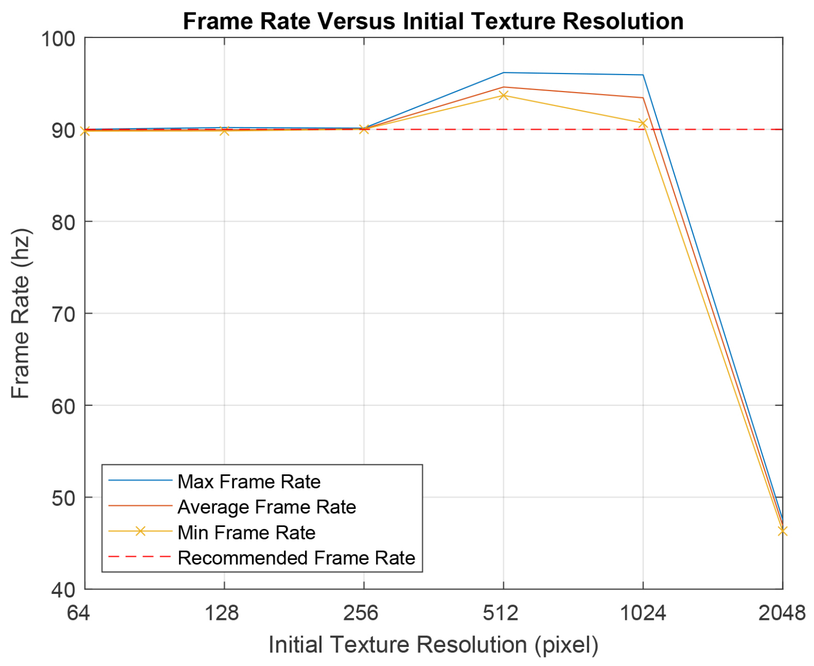


Figure 7.2: The frame rate versus the resolution of initial texture for the color turbulence method. Table 7.2 shows the quantitative descriptions. The test environment contains 4,800 triangles. The test was performed five times to calculate the maximum, minimum, and average frame rate values.

Table 7.3: Frame rate results for different resolutions of initial texture for the color turbulence method (see results in Figure 7.3).

	64	128	256	512	1024	2048
Maximum frame rate	93.37	94.57	94.57	90.74	81.27	51.54
Average frame rate	90.85	93.75	92.56	63.34	64.01	46.45
Minimum frame rate	90.15	92.87	90.24	45.04	45.02	45.03

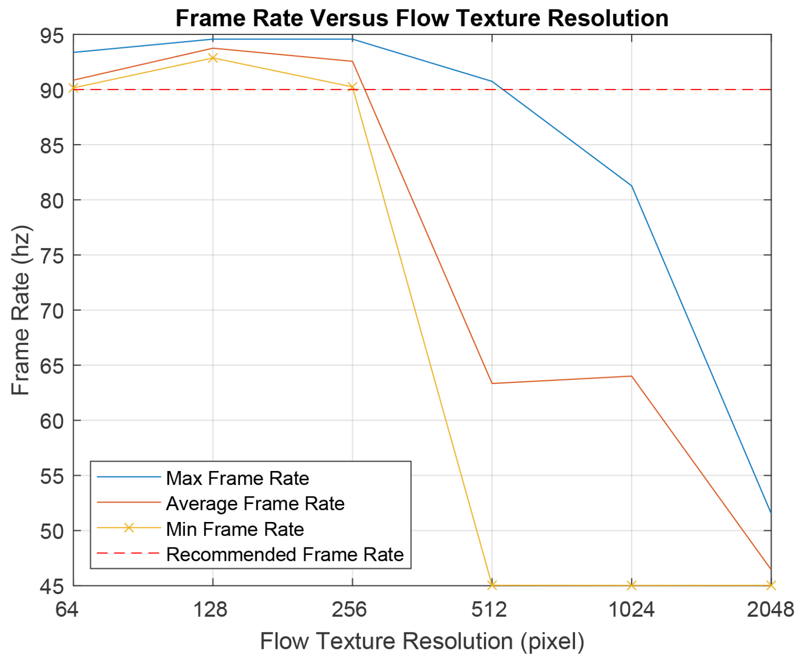


Figure 7.3: The frame rate versus the resolution of flow texture for the color turbulence method. Table 7.3 shows the quantitative descriptions. The test environment contains 4,800 triangles. The test was performed five times to calculate the maximum, minimum, and average frame rate values.

Table 7.4: Frame rate results for different resolutions of noise texture for the color turbulence method (see results in Figure 7.4).

	64	128	256	512	1024	2048
Maximum frame rate	96.52	95.6	93.29	83.28	53.7	47.64
Average frame rate	94.41	94.30	91.56	80.16	51.55	46.85
Minimum frame rate	93.49	93.33	90.08	79.12	47.65	46.06

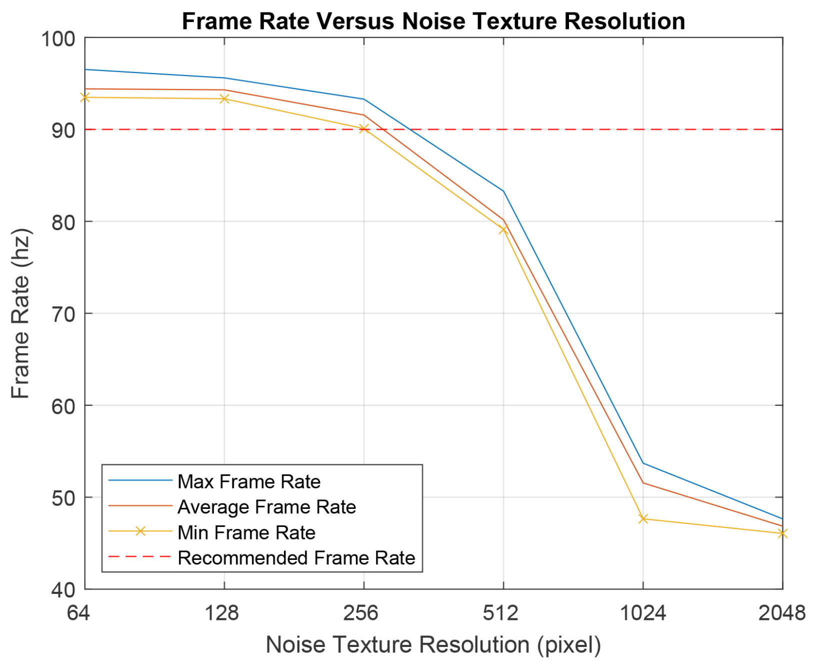


Figure 7.4: The frame rate versus the resolution of noise texture for the color turbulence method. Table 7.4 shows the quantitative descriptions. The test environment contains 4,800 triangles. The test was performed five times to calculate the maximum, minimum, and average frame rate values.

Table 7.5: Frame rate results for different numbers of 3D models for the color turbulence method (see results in Figure 7.5).

	15	16	17	18	19	20	21	22	23	24
Maximum frame rate	95.67	95.83	95.31	96.09	96.88	95.74	89.22	62.32	48.8	48.42
Average frame rate	92.57	93.49	93.54	93.57	94.98	94.79	83.23	52.16	46.84	46.21
Minimum frame rate	90.76	90.11	91.81	91.21	93.21	94.15	79.33	46.02	45.71	44.07

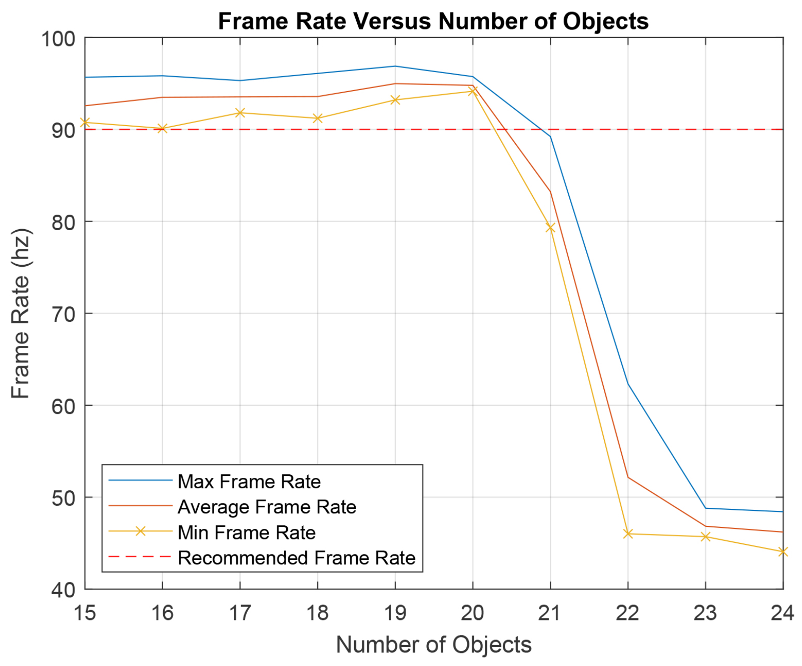


Figure 7.5: The frame rate versus the number of 3D models for the color turbulence method. Table 7.5 shows the quantitative descriptions. Every model contains 200 triangles. The test was performed five times to calculate the maximum, minimum, and average frame rate values.

Table 7.6: Frame rate results for different amount of suspended time for the color turbulence method (see results in Figure 7.6).

	64	128	256	512	1024	2048
Maximum frame rate	32.26	33.76	83.54	85.63	94.41	93.28
Average frame rate	23.95	25.93	50.84	61.10	90.73	90.51
Minimum frame rate	20.65	20.68	24.34	34.11	89.7	89.64

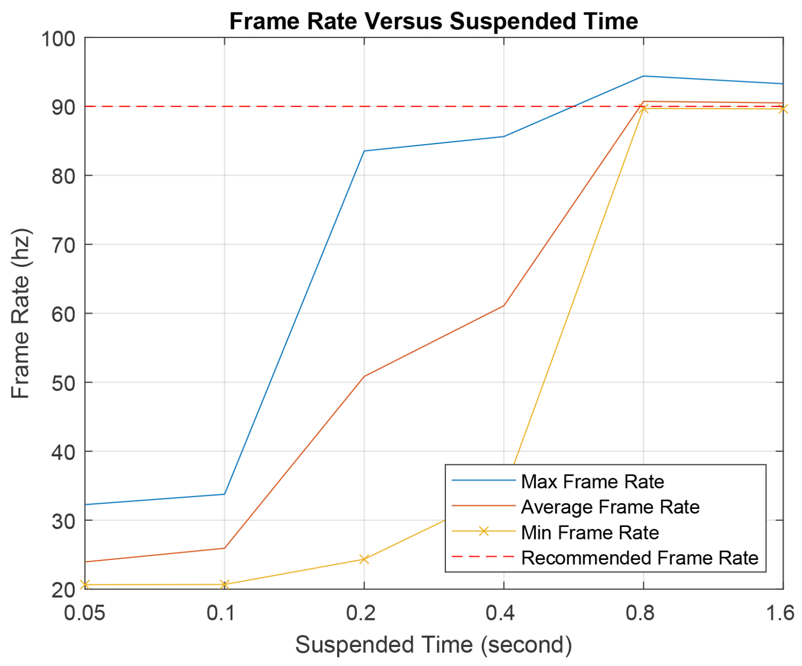


Figure 7.6: The frame rate versus the suspended time for the color turbulence method. Table 7.6 shows the quantitative descriptions. The test environment contains 4,800 triangles. The test was performed five times to calculate the maximum, minimum, and average frame rate values.

Table 7.7: Frame rate results for different numbers of 3D models for the vertex displacement approach (see results in Figure 7.7).

	15	16	17	18	19	20	21	22	23	24
Maximum frame rate	90.1	90.08	90.04	90.02	90.02	90.06	90.11	90.13	90.14	90.03
Average frame rate	89.9	89.99	89.86	89.82	89.88	89.9	89.96	89.91	89.93	89.92
Minimum frame rate	90.01	90.02	89.97	89.95	89.97	89.97	90.06	89.99	90.00	89.98

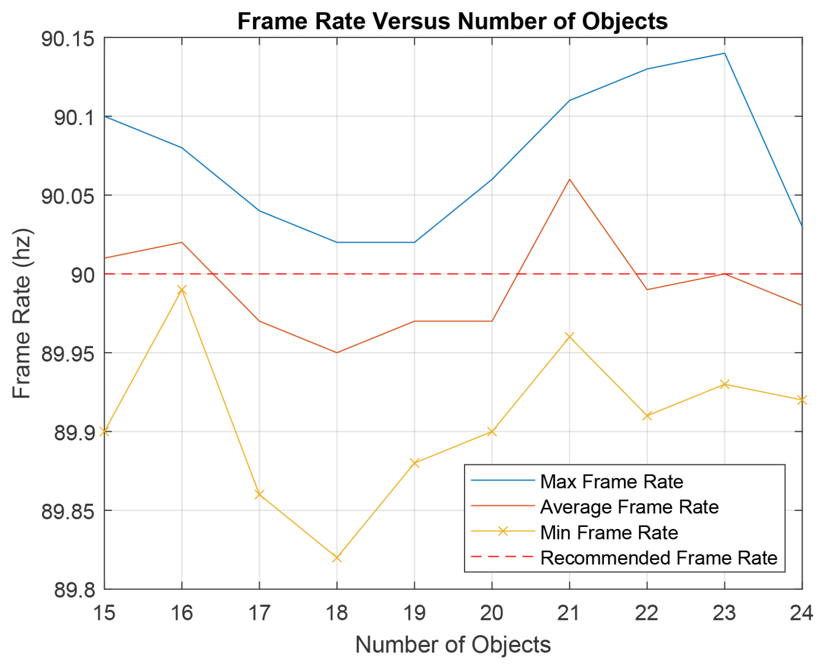
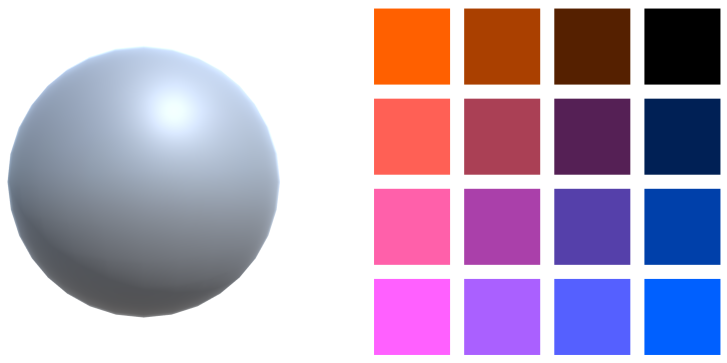


Figure 7.7: The frame rate versus the number of 3D models for the vertex displacement approach. Table 7.7 shows the quantitative descriptions. Every model contains 200 triangles. The test was performed five times to calculate the maximum, minimum, and average frame rate values.

7.2 Aesthetic Evaluation

Different noise functions create different geometric and texture distortion results. Therefore, I set up a set of subjective analytic axes to indicate some of the ways in which the noise functions contrast with one another. The noise function types covered include white (random) noise, value noise, Perlin noise, and Worley noise. A comparison between different results along the axes of form and smoothness was chosen because these attributes potentially convey personality and narrative emotion. A 3D sphere model and a 2D checkered pattern were used as the input to the geometric and texture distortion methods, respectively (Figure 7.8).



(a) (b)

Figure 7.8: A sphere and a checkered pattern were used as the input for aesthetic evaluation.

7.2.1 Form

Curves, circles, and round shapes may be described as having a warm and cheerful personality because we often find similar forms in babies and puppies. In contrast, angles can be used to portray a devious, conniving, or evil character because diagonal or asymmetrical lines can convey something that does not belong or is unstable [68, 69]. Figure 7.9 shows a frosted texture created via white noise. The frosted pattern is not juxtaposed with other texture distortion results because it does not strongly convey a sense of form or smoothness. In Figure 7.10 and 7.11, we can see that white noise creates shapes that are covered with spines, and Worley noise generates prickly shapes and a polygon pattern. In contrast, value noise and Perlin noise create geometries consisting of mostly round shapes and circular or spiral patterns.

7.2.2 Smoothness

How smooth is the geometric shape or texture pattern that a noise function can create? Figure 7.10 shows that there are no smooth transitions between vertex positions of the shapes distorted by white noise. Furthermore, Figure 7.10 and 7.11 show that some regions of the shapes or textures distorted by Worley noise contain smooth transitions and some regions contain sharp changes. Finally, value noise and Perlin noise generate the smoothest shapes and textures. The difference between value noise and Perlin noise is that since value noise uses values that are interpolated, it may create a flat surface or a less distorted texture. On the contrary, the results from using Perlin noise have a higher amount of variance because interpolation is not carried out between values but is calculated between tangents instead.

From the examination described above, I recommend using white noise or Worley noise for a hostile or aggressive character, and Perlin noise or value noise for a friendly or unthreatening character.

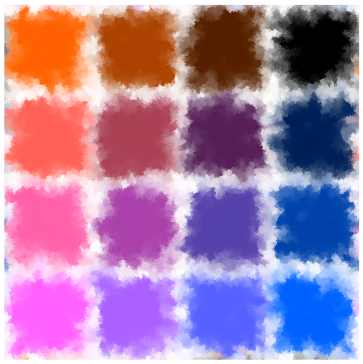


Figure 7.9: A frosted texture created via white noise

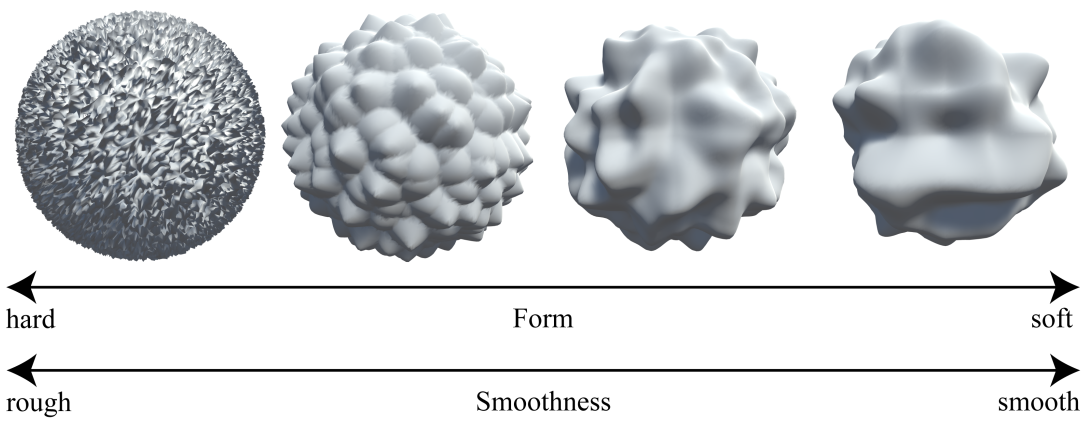


Figure 7.10: A comparative examination of different geometric distortion results created via white noise, Worley noise, Perlin noise, and value noise (left to right) along the axes of form and smoothness.

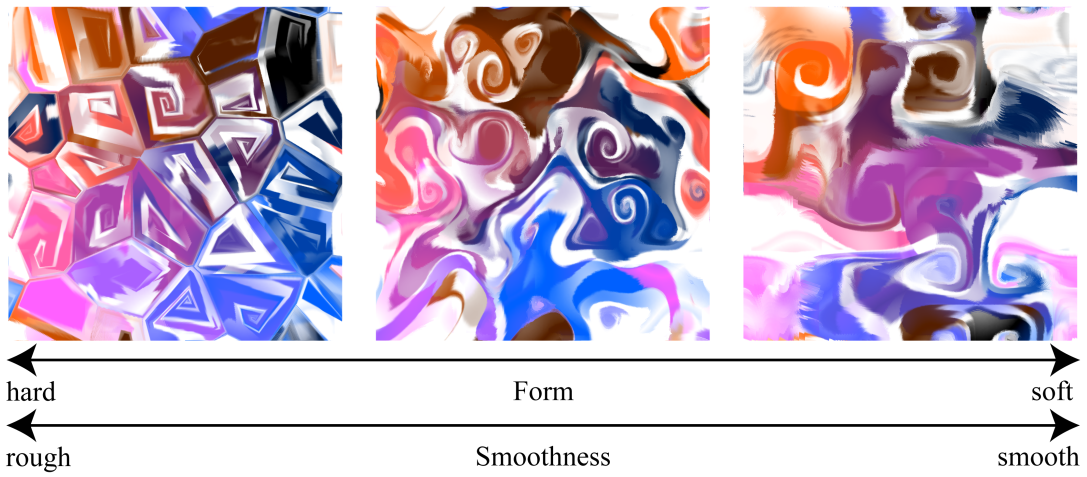


Figure 7.11: A comparative examination of different texture distortion results created via Worley noise, Perlin noise, and value noise (left to right) along the axes of form and smoothness.

Chapter 8

Conclusions

In this dissertation, techniques to give 3D models an organic and flowing appearance based on a combination of two noise algorithms are presented. With these techniques, models can also become audio-reactive and interact with the audience in VR. These techniques can be applied not only to VR but also across other areas of computer graphics, with benefits for computer animation, video games, and interactive user interfaces.

The final chapter of this dissertation covers the cultural meaning, limitations, and future work of the presented techniques.

8.1 Cultural Meaning

Essentially, this work is about our bond with the natural world. Our generation, being insulated by technology and synthetic materials from nature itself, might have left the natural world behind. However, at a very deep level, we still have a longing to link with the natural world in which we evolved and where we can truly find peace. The fantasy world of the work appears to be the product of technology, but in fact, it is the result of my urge to retain part of the natural world in the synthetic virtual world.

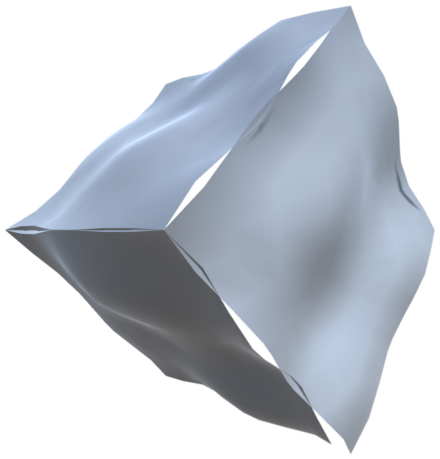
8.2 Limitations

In the course of implementing the methods that support this dissertation, I have come to identify a few limitations of my methods. For the vertex displacement method, the resulting shape remains smooth or continuous if the input shape is also smooth or continuous. An example of a smooth shape is a sphere or a torus. On the contrary, the resulting shape can be broken if the input shape is not smooth, as shown in Figure 8.1. A shape is not smooth if its surface has some sharp corners. For instance, cubes and tetrahedrons are not smooth shapes.

The color turbulence method also has its limitations. First, the method works only on the models that are UV mapped. UV mapping is the process of generating a flat representation of a 3D geometry. A UV map is the 2D representation of a 3D geometric surface. Each coordinate on the UV map corresponds with the vertex of the 3D object. Therefore, UV maps are vital for my method to work because they provide the link between a surface mesh and how the turbulent flow texture gets applied onto that surface.

The other limitation of the color turbulence method is that the user interaction will become absurd if the model's texture is tiled to create a repeating pattern. In the case of tiled texture, since the UV map also tiles itself, the shader is unable to inject colors at the exact position of the point on the surface of the model that the user touched. Instead, colors are injected at each tile across the surface of the model at the same time, as show in Figure 8.2.

Finally, both methods have this aesthetic limitation: their results can end up being unimpressive if people overuse these them. Like any special effects, the visuals will be obsolete once audiences see them. Whether it matters to viewers or not will still depend on story and character. Therefore, the methods should be carefully used to support the narrative in an innovative way.


(a)

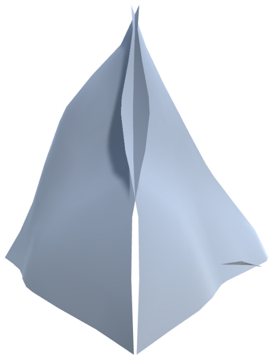
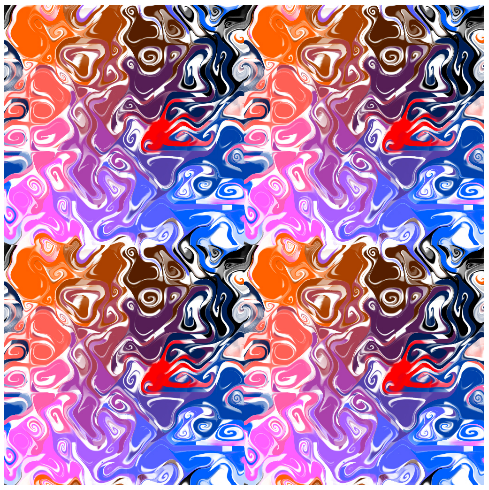

(b)

Figure 8.1: Applying the vertex displacement method to a 3D model can result in a broken shape if the input shape is not smooth.


(a)

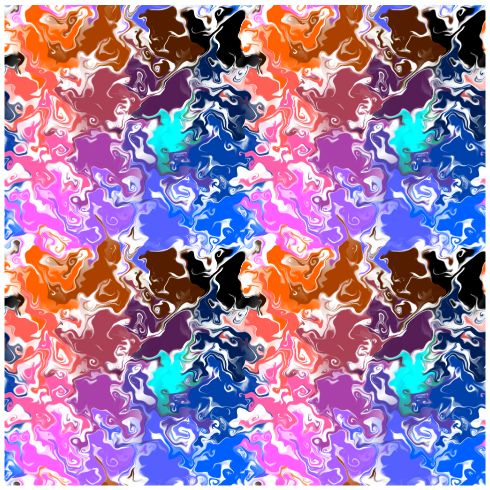

(b)

Figure 8.2: One of the limitations of the color turbulence technique is that the user interaction does not work for the 3D models that have repeating texture.

8.3 Future Work

While the limitations described in the preceding section are overcome, several promising directions remain for future research with user interaction. In the framework described in the dissertation, the player can use only one handheld controller to interact with the objects in VR. Such user interaction does not feel as natural as the lifelike interaction, where many people can manipulate the objects using both of their hands. Allowing for two handheld controllers not only provides the player a more natural interaction but also makes the interaction more interesting. Consider that the player can paint on the objects using two different colors and observe how one pigment moves and reacts to another. Other issue related to user interaction is that collaborative interactivity should be addressed. In other words, multiple players should be allowed to participate at the same time. This function brings the entire audience into a shared environment just as they could experience in cinema or at concerts.

Another direction for future research is testing my approach on smartphone VR headsets and standalone VR headsets. Smartphone VR headsets, such as Google Cardboard, Google Daydream, and Samsung Gear VR, use compatible mobile devices for their VR content. Standalone VR headsets, such as Oculus Go and Oculus Quest, are similar to smartphone VR headsets but with one extra benefit: they do not require a smartphone to operate. Although smartphone VR headsets and standalone VR headsets do not have the graphical capabilities of tethered VR headsets, which are connected to a PC to push its visuals, they provide portable VR experiences and are less expensive than tethered headsets. The techniques described in this dissertation can be performed on a tethered VR headset that is powered by a high-end PC with modern graphics cards. However, the performance on mobile devices or standalone VR headsets was not investigated. If the techniques can be smoothly run on smartphone VR headsets and standalone VR headsets, casual VR users who do not own high-end PCs can still experience the results in a simpler and more affordable way.

Appendix A

Cg Implementation of Vertex Displacement Method

Appendix A.1 lists the Cg code that manipulates the vertex positions of a 3D object's surface using Perlin noise. The code also updates the shadow to match the object's new shape and recalculates normals based on the new shape. The Perlin noise function that the code uses is provided by Keijiro Takahashi [70] and is included in Appendix A.2.

A.1 Vertex Displacement Shader

```
Shader "Custom/LitVertexDisplacement" {
  Properties {
    _Tess ("Tessellation", Range(1, 8)) = 4
    _Color ("Color", Color) = (1,1,1,1)
    _MainTex ("Albedo (RGB)", 2D) = "white" {}
    _Glossiness ("Smoothness", Range(0,1)) = 0.5
    _Metallic ("Metallic", Range(0,1)) = 0.0
    _NoiseScale ("Noise Scale", float) = 0.5
    _NoiseFrequency ("Noise Frequency", float) = 0.5
    _NoiseOffset ("Noise Offset", Vector) = (0, 0, 0)
    [Toggle] _EnableEmission ("Enable Emission?", int) = 0
    [HDR] _EmissionColor("Emission Color", Color) = (1,1,1)
    _EmissionMap("Emission", 2D) = "white" {}
  }

  SubShader {
    Tags { "RenderType"="Opaque" }

    CGPROGRAM
    // Physically based Standard lighting model, and enable
    // shadows on all light types
    #pragma surface surf Standard fullforwardshadows
    #pragma tessellate:tess vertex:vert addshadow

    // Use shader model 3.0 target, to get nicer looking lighting
    #pragma target 4.6

    // You may try different noise methods, such as
    // #include "noiseSimplex.cginc"
    #include "ClassicNoise3D.hlsl"

    sampler2D _MainTex, _EmissionMap;
```

```
struct Input {
  float2 uv_MainTex;
};

float _Tess;
half _Glossiness;
half _Metallic;
fixed4 _Color, _EmissionColor;
int _EnableEmission;

float _NoiseScale, _NoiseFrequency;
float3 _NoiseOffset;

float4 tess()
{
  return _Tess;
}

float3 getNewVertPosition(float3 p, float3 n)
{
  float noise = cnoise(float3(p.x + _NoiseOffset.x,
    p.y + _NoiseOffset.y,
    p.z + _NoiseOffset.z) * _NoiseFrequency);
  // float noise = snoise(float3(p.x + _Time.y,
  //    p.y + _Time.y,
  //    p.z + _Time.y) * _NoiseFrequency);
  // remap 'noise' from its default range [-1, 1] to a target
  // range of [0, 1]
  // noise = ((noise + 1) / 2);
  p += _NoiseScale * noise * n;

  return p;
}

void vert(inout appdata_full v)
{
  float3 position = getNewVertPosition(v.vertex.xyz,
    v.normal);

  // calculate the bitangent (sometimes called binormal) from
  // the cross product of the normal and the tangent
  float3 bitangent = cross(v.normal, v.tangent.xyz);

  // how far we want to offset our vert position to calculate
  // the new normal
  float vertOffset = 0.01;

  float3 positionAndTangent = getNewVertPosition(
    v.vertex.xyz + v.tangent.xyz * vertOffset, v.normal);
  float3 positionAndBitangent = getNewVertPosition(
    v.vertex.xyz + bitangent * vertOffset, v.normal);

  // now we can create new tangents and bitangents based on
  // the deformed positions
  float3 newTangent = positionAndTangent - position;
  float3 newBitangent = positionAndBitangent - position;

  // recalculate the normal based on the new tangent &
  // bitangent
  v.normal = normalize(cross(newTangent, newBitangent));

  v.vertex.xyz = position;
}

void surf (Input IN, inout SurfaceOutputStandard o) {
  // Albedo comes from a texture tinted by color
  fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
  o.Albedo = c.rgb;
  // Metallic and smoothness come from slider variables
  o.Metallic = _Metallic;
  o.Smoothness = _Glossiness;
  o.Alpha = c.a;
  o.Emission = _EnableEmission *
    tex2D (_EmissionMap, IN.uv_MainTex) * _EmissionColor;
  }
  ENDCG
}
FallBack "Diffuse"
}
```

A.2 Classic Noise Functions

```
// Noise Shader Library for Unity
// https://github.com/keijiro/NoiseShader
//
// Original work (webgl-noise)
// Copyright (C) 2011 Stefan Gustavson
// Translation and modification was made by Keijiro Takahashi.
//
// This shader is based on the webgl-noise GLSL shader. For
// further details of the original shader, please see the
// following description from the original source code.
//


//
// GLSL textureless classic 3D noise "cnoise",
// with an RSL-style periodic variant "pnoise".
// Author:  Stefan Gustavson (stefan.gustavson@liu.se)
// Version: 2011-10-11
//
```

```
// Many thanks to Ian McEwan of Ashima Arts for the
// ideas for permutation and gradient selection.
//
// Copyright (c) 2011 Stefan Gustavson. All rights reserved.
// Distributed under the MIT license. See LICENSE file.
// https://github.com/ashima/webgl-noise
//

float3 mod(float3 x, float3 y)
{
  return x - y * floor(x / y);
}

float3 mod289(float3 x)
{
  return x - floor(x / 289.0) * 289.0;
}

float4 mod289(float4 x)
{
  return x - floor(x / 289.0) * 289.0;
}

float4 permute(float4 x)
{
  return mod289(((x*34.0)+1.0)*x);
}

float4 taylorInvSqrt(float4 r)
{
  return (float4)1.79284291400159 - r * 0.85373472095314;
}

float3 fade(float3 t) {
  return t*t*t*(t*(t*6.0-15.0)+10.0);
}

// Classic Perlin noise
float cnoise(float3 P)
{
  float3 Pi0 = floor(P); // Integer part for indexing
  float3 Pi1 = Pi0 + (float3)1.0; // Integer part + 1
  Pi0 = mod289(Pi0);
  Pi1 = mod289(Pi1);
  float3 Pf0 = frac(P); // Fractional part for interpolation
  float3 Pf1 = Pf0 - (float3)1.0; // Fractional part - 1.0
  float4 ix = float4(Pi0.x, Pi1.x, Pi0.x, Pi1.x);
  float4 iy = float4(Pi0.y, Pi0.y, Pi1.y, Pi1.y);
  float4 iz0 = (float4)Pi0.z;
  float4 iz1 = (float4)Pi1.z;
```

```
float4 ixy = permute(permute(ix) + iy);
float4 ixy0 = permute(ixy + iz0);
float4 ixy1 = permute(ixy + iz1);

float4 gx0 = ixy0 / 7.0;
float4 gy0 = frac(floor(gx0) / 7.0) - 0.5;
gx0 = frac(gx0);
float4 gz0 = (float4)0.5 - abs(gx0) - abs(gy0);
float4 sz0 = step(gz0, (float4)0.0);
gx0 -= sz0 * (step((float4)0.0, gx0) - 0.5);
gy0 -= sz0 * (step((float4)0.0, gy0) - 0.5);

float4 gx1 = ixy1 / 7.0;
float4 gy1 = frac(floor(gx1) / 7.0) - 0.5;
gx1 = frac(gx1);
float4 gz1 = (float4)0.5 - abs(gx1) - abs(gy1);
float4 sz1 = step(gz1, (float4)0.0);
gx1 -= sz1 * (step((float4)0.0, gx1) - 0.5);
gy1 -= sz1 * (step((float4)0.0, gy1) - 0.5);

float3 g000 = float3(gx0.x,gy0.x,gz0.x);
float3 g100 = float3(gx0.y,gy0.y,gz0.y);
float3 g010 = float3(gx0.z,gy0.z,gz0.z);
float3 g110 = float3(gx0.w,gy0.w,gz0.w);
```

```
float3 g001 = float3(gx1.x,gy1.x,gz1.x);
float3 g101 = float3(gx1.y,gy1.y,gz1.y);
float3 g011 = float3(gx1.z,gy1.z,gz1.z);
float3 g111 = float3(gx1.w,gy1.w,gz1.w);

float4 norm0 = taylorInvSqrt(float4(dot(g000, g000),
                                    dot(g010, g010),
                                    dot(g100, g100),
                                    dot(g110, g110)));
g000 *= norm0.x;
g010 *= norm0.y;
g100 *= norm0.z;
g110 *= norm0.w;

float4 norm1 = taylorInvSqrt(float4(dot(g001, g001),
                                    dot(g011, g011),
                                    dot(g101, g101),
                                    dot(g111, g111)));
g001 *= norm1.x;
g011 *= norm1.y;
g101 *= norm1.z;
g111 *= norm1.w;

float n000 = dot(g000, Pf0);
float n100 = dot(g100, float3(Pf1.x, Pf0.y, Pf0.z));
```

```
  float n010 = dot(g010, float3(Pf0.x, Pf1.y, Pf0.z));
  float n110 = dot(g110, float3(Pf1.x, Pf1.y, Pf0.z));
  float n001 = dot(g001, float3(Pf0.x, Pf0.y, Pf1.z));
  float n101 = dot(g101, float3(Pf1.x, Pf0.y, Pf1.z));
  float n011 = dot(g011, float3(Pf0.x, Pf1.y, Pf1.z));
  float n111 = dot(g111, Pf1);

  float3 fade_xyz = fade(Pf0);
  float4 n_z = lerp(float4(n000, n100, n010, n110),
                    float4(n001, n101, n011, n111), fade_xyz.z);
  float2 n_yz = lerp(n_z.xy, n_z.zw, fade_xyz.y);
  float n_xyz = lerp(n_yz.x, n_yz.y, fade_xyz.x);
  return 2.2 * n_xyz;
}

// Classic Perlin noise, periodic variant
float pnoise(float3 P, float3 rep)
{
  // Integer part, modulo period
  float3 Pi0 = mod(floor(P), rep);
  // Integer part + 1, mod period
  float3 Pi1 = mod(Pi0 + (float3)1.0, rep);
  Pi0 = mod289(Pi0);
  Pi1 = mod289(Pi1);
  float3 Pf0 = frac(P); // Fractional part for interpolation
```

```
float3 Pf1 = Pf0 - (float3)1.0; // Fractional part - 1.0
float4 ix = float4(Pi0.x, Pi1.x, Pi0.x, Pi1.x);
float4 iy = float4(Pi0.y, Pi0.y, Pi1.y, Pi1.y);
float4 iz0 = (float4)Pi0.z;
float4 iz1 = (float4)Pi1.z;

float4 ixy = permute(permute(ix) + iy);
float4 ixy0 = permute(ixy + iz0);
float4 ixy1 = permute(ixy + iz1);

float4 gx0 = ixy0 / 7.0;
float4 gy0 = frac(floor(gx0) / 7.0) - 0.5;
gx0 = frac(gx0);
float4 gz0 = (float4)0.5 - abs(gx0) - abs(gy0);
float4 sz0 = step(gz0, (float4)0.0);
gx0 -= sz0 * (step((float4)0.0, gx0) - 0.5);
gy0 -= sz0 * (step((float4)0.0, gy0) - 0.5);

float4 gx1 = ixy1 / 7.0;
float4 gy1 = frac(floor(gx1) / 7.0) - 0.5;
gx1 = frac(gx1);
float4 gz1 = (float4)0.5 - abs(gx1) - abs(gy1);
float4 sz1 = step(gz1, (float4)0.0);
gx1 -= sz1 * (step((float4)0.0, gx1) - 0.5);
gy1 -= sz1 * (step((float4)0.0, gy1) - 0.5);
```

```
float3 g000 = float3(gx0.x,gy0.x,gz0.x);
float3 g100 = float3(gx0.y,gy0.y,gz0.y);
float3 g010 = float3(gx0.z,gy0.z,gz0.z);
float3 g110 = float3(gx0.w,gy0.w,gz0.w);
float3 g001 = float3(gx1.x,gy1.x,gz1.x);
float3 g101 = float3(gx1.y,gy1.y,gz1.y);
float3 g011 = float3(gx1.z,gy1.z,gz1.z);
float3 g111 = float3(gx1.w,gy1.w,gz1.w);

float4 norm0 = taylorInvSqrt(float4(dot(g000, g000),
                                    dot(g010, g010),
                                    dot(g100, g100),
                                    dot(g110, g110)));
g000 *= norm0.x;
g010 *= norm0.y;
g100 *= norm0.z;
g110 *= norm0.w;
float4 norm1 = taylorInvSqrt(float4(dot(g001, g001),
                                    dot(g011, g011),
                                    dot(g101, g101),
                                    dot(g111, g111)));
g001 *= norm1.x;
g011 *= norm1.y;
g101 *= norm1.z;
```

```
  g111 *= norm1.w;

  float n000 = dot(g000, Pf0);
  float n100 = dot(g100, float3(Pf1.x, Pf0.y, Pf0.z));
  float n010 = dot(g010, float3(Pf0.x, Pf1.y, Pf0.z));
  float n110 = dot(g110, float3(Pf1.x, Pf1.y, Pf0.z));
  float n001 = dot(g001, float3(Pf0.x, Pf0.y, Pf1.z));
  float n101 = dot(g101, float3(Pf1.x, Pf0.y, Pf1.z));
  float n011 = dot(g011, float3(Pf0.x, Pf1.y, Pf1.z));
  float n111 = dot(g111, Pf1);

  float3 fade_xyz = fade(Pf0);
  float4 n_z = lerp(float4(n000, n100, n010, n110),
                    float4(n001, n101, n011, n111), fade_xyz.z);
  float2 n_yz = lerp(n_z.xy, n_z.zw, fade_xyz.y);
  float n_xyz = lerp(n_yz.x, n_yz.y, fade_xyz.x);
  return 2.2 * n_xyz;
}
```

Appendix B

Cg Implementation of Color Turbulence Method

The color turbulence method consists of two programs—the flow shader and the blend shader. The flow shader generates flow animation in the texture of an object via a flow map, which contains 2D vectors. The blend shader morphs back and forth between two flow animations that have slightly different time offsets so that the overall flow looks continuous.

B.1 Flow Shader

```
Shader "Unlit/Flow"
{
  Properties
  {
    _MainTex ("Animated Texture (RGB)", 2D) = "white" {}
    _OriginalTex ("Original Texture (RGB)", 2D) = "white" {}
    _FlowMap ("Flow (RG)", 2D) = "black" {}
    _NoiseMap ("Noise (A)", 2D) = "white" {}
    _PhaseOffset ("Phase Offset", Float) = 0.0
    _Speed ("Speed", Float) = 1.0
    _FlowStrength ("Flow Strength", Float) = 1.0
  }
  SubShader
  {
    Tags { "RenderType"="Opaque" }
    LOD 100

    Pass
    {
      CGPROGRAM
      #pragma vertex vert
      #pragma fragment frag

      #include "UnityCG.cginc"

      struct appdata
      {
        float4 vertex : POSITION;
        float2 uv : TEXCOORD0;
      };
```

```
struct v2f
{
  float2 uv : TEXCOORD0;
  float4 vertex : SV_POSITION;
};

sampler2D _MainTex, _OriginalTex, _FlowMap, _NoiseMap;
float _PhaseOffset, _Speed, _FlowStrength;

float4 _MainTex_ST;
float4 _FlowMap_TexelSize;

float2 _ForceOrigin;
float _ForceExponent;


v2f vert (appdata v)
{
  v2f o;
  o.vertex = UnityObjectToClipPos(v.vertex);
  o.uv = TRANSFORM_TEX(v.uv, _MainTex);
  return o;
}

#define IF(a, b, c) lerp(b, c, step((fixed) (a), 0));
```

```
fixed4 frag (v2f i) : SV_Target
{
  // Time parameters
  float time = _Time.y * _Speed;

  float2 flow = normalize(tex2D(_FlowMap, i.uv).rg * 2 - 1)
     * _FlowMap_TexelSize.xy;
  flow *= _FlowStrength;
  float2 uv = i.uv;
  uv.x -= flow.x;
  uv.y += flow.y;

  float noise = tex2D(_NoiseMap, i.uv).a;

  float progress = frac(time + noise + _PhaseOffset);
  fixed3 colAnimated = tex2D(_MainTex, uv).rgb;
  fixed3 colOriginal = tex2D(_OriginalTex, uv).rgb;
  fixed3 col = lerp(colAnimated,
    colOriginal,
    smoothstep(0.9, 0.95, progress));

  float fade = 1 - abs(1 - 2 * progress);

  // Dye (injection col)
```

```
        float3 dye = saturate(
          sin(time * float3(2.72, 5.12, 4.98)) + 0.5);

        // Blend dye with the col from the buffer.
        float2 pos = (i.uv - 0.5);
        float amp = exp(
          -_ForceExponent * distance(_ForceOrigin, pos));
        col = lerp(col, dye, saturate(amp * 100));

        return fixed4(col, fade);
      }

      ENDCG
    }
  }
}
```

B.2 Blend Shader

```
Shader "Unlit/Blend"
{
  Properties
  {
    _MainTex ("Transparency (A)", 2D) = "white" {}
    _TexA ("Texture A (RGB, A noise)", 2D) = "white" {}
    _TexB ("Texture B (RGB, A noise)", 2D) = "white" {}
```

```
}
SubShader
{
  Tags { "Queue"="Transparent" "RenderType"="Transparent" }
  LOD 100

  ZWrite Off
  Blend SrcAlpha OneMinusSrcAlpha

  Pass
  {
    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag

    #include "UnityCG.cginc"

    struct appdata
    {
      float4 vertex : POSITION;
      float2 uv : TEXCOORD0;
    };

    struct v2f
    {
```

```
    float2 uv : TEXCOORD0;
    float4 vertex : SV_POSITION;
};

sampler2D _MainTex, _TexA, _TexB;
float4 _MainTex_ST;



v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);
    return o;
}

fixed4 frag (v2f i) : SV_Target
{
    // sample the texture
    fixed4 texA = tex2D(_TexA, i.uv) * tex2D(_TexA, i.uv).a;
    fixed4 texB = tex2D(_TexB, i.uv) * tex2D(_TexB, i.uv).a;
    fixed4 col = texA + texB;
    col.a = tex2D(_MainTex, i.uv).a;

    return col;
```

```
            }
            ENDCG
        }
    }
}
```

Appendix C

Research Timeline

In this appendix, I provide a graphical timeline of research to indicate significant steps and milestones along the way. This illustrates the long-term effort and work involved in this dissertation.

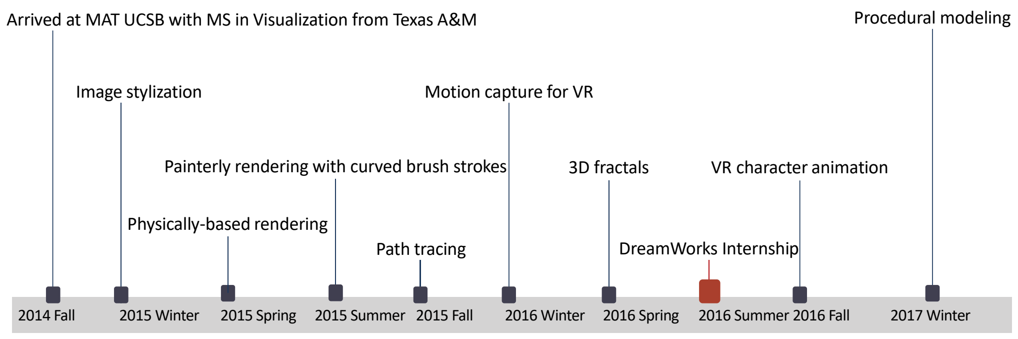


Figure C.1: 2014-2017 Research timeline

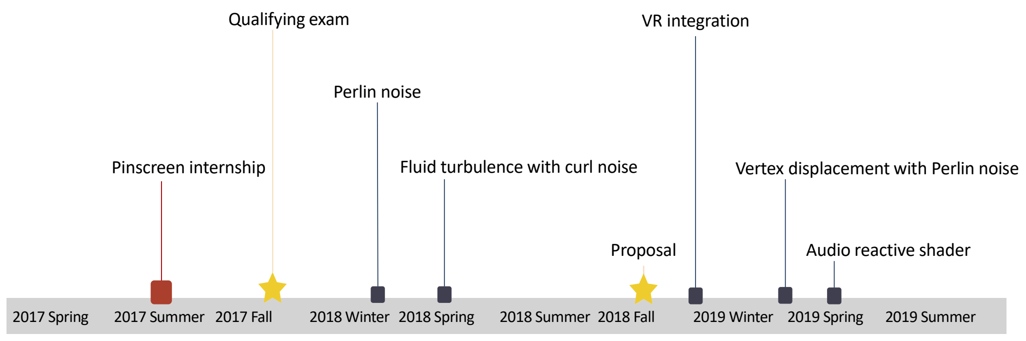


Figure C.2: 2017-2019 Research timeline

Bibliography

[1] P. Rosin and J. Collomosse, *Image and video-based artistic stylisation*, vol. 42. Springer Science & Business Media, 2012.

[2] J. E. Kyprianidis, J. Collomosse, T. Wang, and T. Isenberg, *State of the" art: A taxonomy of artistic stylization techniques for images and video*, *IEEE transactions on visualization and computer graphics* **19** (2012), no. 5 866–885.

[3] M. P. Kumar, B. Poornima, H. Nagendraswamy, and C. Manjunath, *A comprehensive survey on non-photorealistic rendering and benchmark developments for image abstraction and stylization*, *Iran Journal of Computer Science* (2019) 1–35.

[4] J. Fišer, O. Jamriška, D. Simons, E. Shechtman, J. Lu, P. Asente, M. Lukáč, and D. Sỳkora, *Example-based synthesis of stylized facial animations*, *ACM Transactions on Graphics (TOG)* **36** (2017), no. 4 155.

[5] J. Lu, P. V. Sander, and A. Finkelstein, *Interactive painterly stylization of images, videos and 3d animations*, in *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pp. 127–134, ACM, 2010.

[6] H. Winnemöller, S. C. Olsen, and B. Gooch, *Real-time video abstraction*, in *ACM Transactions On Graphics (TOG)*, vol. 25, pp. 1221–1226, ACM, 2006.

[7] N. Redmond and J. Dingliana, *Adaptive abstraction of 3d scenes in realtime*, .

[8] J. E. Kyprianidis and H. Kang, *Image and video abstraction by coherence-enhancing filtering*, in *Computer Graphics Forum*, vol. 30, pp. 593–602, Wiley Online Library, 2011.

[9] A. Semmo, D. Limberger, J. E. Kyprianidis, and J. Döllner, *Image stylization by interactive oil paint filtering*, *Computers & Graphics* **55** (2016) 157–171.

[10] X. Huang and S. Belongie, *Arbitrary style transfer in real-time with adaptive instance normalization*, in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1501–1510, 2017.

[11] D. Sỳkora, O. Jamriška, O. Texler, J. Fišer, M. Lukáč, J. Lu, and E. Shechtman, *Styleblit: Fast example-based stylization with local guidance*, in *Computer Graphics Forum*, vol. 38, pp. 83–91, Wiley Online Library, 2019.

[12] K. Lawonn, I. Viola, B. Preim, and T. Isenberg, *A survey of surface-based illustrative rendering for visualization*, in *Computer Graphics Forum*, vol. 37, pp. 205–234, Wiley Online Library, 2018.

[13] G. Elber, *Interactive line art rendering of freeform surfaces*, in *Computer Graphics Forum*, vol. 18, pp. 1–12, Wiley Online Library, 1999.

[14] B. Gooch, P.-P. J. Sloan, A. Gooch, P. Shirley, and R. Riesenfeld, *Interactive technical illustration*, *SI3D* **99** (1999) 31–38.

[15] A. Hertzmann and D. Zorin, *Illustrating smooth surfaces*, in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 517–526, ACM Press/Addison-Wesley Publishing Co., 2000.

[16] L. Markosian, M. A. Kowalski, D. Goldstein, S. J. Trychin, J. F. Hughes, and L. D. Bourdev, *Real-time nonphotorealistic rendering*, in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 415–420, ACM Press/Addison-Wesley Publishing Co., 1997.

[17] J. Northrup and L. Markosian, *Artistic silhouettes: A hybrid approach*, in *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pp. 31–37, ACM, 2000.

[18] R. Raskar and M. Cohen, *Image precision silhouette edges*, in *Proceedings of the 1999 symposium on Interactive 3D graphics*, pp. 135–140, ACM, 1999.

[19] S. Rusinkiewicz, F. Cole, D. DeCarlo, and A. Finkelstein, *Line drawings from 3d models*, in *ACM SIGGRAPH 2008 classes*, p. 39, ACM, 2008.

[20] P. Bénard and A. Hertzmann, *Line drawings from 3d models*, *arXiv preprint arXiv:1810.01175* (2018).

[21] A. Lake, C. Marshall, M. Harris, and M. Blackstein, *Stylized rendering techniques for scalable real-time 3d animation*, in *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pp. 13–20, ACM, 2000.

[22] M. P. Salisbury, M. T. Wong, J. F. Hughes, and D. H. Salesin, *Orientable textures for image-based pen-and-ink illustration*, in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 401–406, ACM Press/Addison-Wesley Publishing Co., 1997.

[23] B. Freudenberg, M. Masuch, and T. Strothotte, *Walk-through illustrations: Frame-coherent pen-and-ink style in a game engine*, in *Computer Graphics Forum*, vol. 20, pp. 184–192, Wiley Online Library, 2001.

[24] E. Praun, H. Hoppe, M. Webb, and A. Finkelstein, *Real-time hatching*, in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, p. 581, ACM, 2001.

[25] D. Martín, G. Arroyo, A. Rodríguez, and T. Isenberg, *A survey of digital stippling*, *Computers & Graphics* **67** (2017) 24–44.

[26] A. Lu, C. J. Morris, D. S. Ebert, P. Rheingans, and C. Hansen, *Non-photorealistic volume rendering using stippling techniques*, in *Proceedings of the conference on Visualization'02*, pp. 211–218, IEEE Computer Society, 2002.

[27] A. Lu, C. J. Morris, J. Taylor, D. S. Ebert, C. Hansen, P. Rheingans, and M. Hartner, *Illustrative interactive stipple rendering*, *IEEE Transactions on Visualization and Computer Graphics* **9** (2003), no. 2 127–138.

[28] J. Krüger and R. Westermann, *Efficient stipple rendering*, *Proceedings of IADIS Computer Graphics and Visualization* (2007) 19–26.

[29] R. Schmidt, T. Isenberg, P. Jepp, K. Singh, and B. Wyvill, *Sketching, scaffolding, and inking: a visual history for interactive 3d modeling*, in *Proceedings of the 5th international symposium on Non-photorealistic animation and rendering*, pp. 23–32, ACM, 2007.

[30] A. Gooch, B. Gooch, P. Shirley, and E. Cohen, *A non-photorealistic lighting model for automatic technical illustration.*, in *SIGGRAPH*, vol. 98, pp. 447–452, 1998.

[31] P. Barla, J. Thollot, and L. Markosian, *X-toon: an extended toon shader*, in *Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, pp. 127–132, ACM, 2006.

[32] A. Majumder and M. Gopi, *Hardware accelerated real time charcoal rendering*, in *Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, pp. 59–66, ACM, 2002.

[33] H. Lee, S. Kwon, and S. Lee, *Real-time pencil rendering*, in *Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, pp. 37–45, ACM, 2006.

[34] M. Yuan, X. Yang, S. Xiao, and Z. Ren, *Gpu-based rendering and animation for chinese painting cartoon*, in *Proceedings of Graphics Interface 2007*, pp. 57–61, ACM, 2007.

[35] S. E. Montesdeoca, H. S. Seah, H.-M. Rall, and D. Benvenuti, *Art-directed watercolor stylization of 3d animations in real-time*, *Computers & Graphics* **65** (2017) 60–72.

[36] S. E. Montesdeoca, H. S. Seah, P. Bénard, R. Vergne, J. Thollot, H.-M. Rall, and D. Benvenuti, *Edge-and substrate-based effects for watercolor stylization*, in *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering*, p. 2, ACM, 2017.

[37] R. D. Kalnins, L. Markosian, B. J. Meier, M. A. Kowalski, J. C. Lee, P. L. Davidson, M. Webb, J. F. Hughes, and A. Finkelstein, *Wysiwyg npr: Drawing strokes directly on 3d models*, in *ACM Transactions on Graphics (TOG)*, vol. 21, pp. 755–762, ACM, 2002.

[38] J. Schmid, M. S. Senn, M. Gross, and R. W. Sumner, *Overcoat: an implicit canvas for 3d painting*, in *ACM Transactions on Graphics (TOG)*, vol. 30, p. 28, ACM, 2011.

[39] M. Gerl and T. Isenberg, *Interactive example-based hatching*, *Computers & Graphics* **37** (2013), no. 1-2 65–80.

[40] S. E. Montesdeoca, H. S. Seah, A. Semmo, P. Bénard, R. Vergne, J. Thollot, and D. Benvenuti, *Mnpr: a framework for real-time expressive non-photorealistic rendering of 3d computer graphics*, in *Proceedings of the Joint Symposium on Computational Aesthetics and Sketch-Based Interfaces and Modeling and Non-Photorealistic Animation and Rendering*, p. 9, ACM, 2018.

[41] A. Hertzmann, *Introduction to 3d non-photorealistic rendering: Silhouettes and outlines*, *Non-Photorealistic Rendering. SIGGRAPH* **99** (1999), no. 1.

[42] M. L. Heilig, *Sensorama simulator*, Aug. 28, 1962. US Patent 3,050,870.

[43] I. E. Sutherland, *The ultimate display*, *Multimedia: From Wagner to virtual reality* (1965) 506–508.

[44] I. E. Sutherland, *A head-mounted three dimensional display*, in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pp. 757–764, ACM, 1968.

[45] L. J. Hettinger and G. E. Riccio, *Visually induced motion sickness in virtual environments*, *Presence: Teleoperators & Virtual Environments* **1** (1992), no. 3 306–310.

[46] L. Rebenitsch, *Managing cybersickness in virtual reality*, *XRDS: Crossroads, The ACM Magazine for Students* **22** (2015), no. 1 46–51.

[47] S. Boyer, *A virtual failure: Evaluating the success of nintendo's virtual boy*, *The Velvet Light Trap* (2009), no. 64 23–33.

[48] "For ar/vr 2.0 to live, ar/vr 1.0 must die." `https://www.digi-capital.com/news/2019/01/for-ar-vr-2-0-to-live-ar-vr-1-0-must-die`.

[49] J. H. Murray and J. H. Murray, *Hamlet on the holodeck: The future of narrative in cyberspace.* MIT press, 2017.

[50] M.-L. Ryan, *Narrative as virtual reality*, *Immersion and Interactivity in Literature* (2001).

[51] M. Chen and J. Yan, *Reincarnation: virtual reality recreation of yves tanguy's world*, in *ACM SIGGRAPH 2019 Virtual, Augmented, and Mixed Reality*, p. 13, ACM, 2019.

[52] K. Mack, *Blortasia: a virtual reality art experience*, in *ACM SIGGRAPH 2017 VR Village*, p. 2, ACM, 2017.

[53] L. Putnam, W. Latham, and S. Todd, *Flow fields and agents for immersive interaction in mutator vr: Vortex*, *Presence: Teleoperators and Virtual Environments* **26** (2017), no. 2 138–156.

[54] J. Cantor and P. Valencia, *Inspired 3D Short Film Production (Inspired).* Course Technology Press, 2004.

[55] A. W. Klein, W. Li, M. M. Kazhdan, W. T. Corrêa, A. Finkelstein, and T. A. Funkhouser, *Non-photorealistic virtual environments*, in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 527–534, ACM Press/Addison-Wesley Publishing Co., 2000.

[56] K. Perlin, *An image synthesizer*, *ACM Siggraph Computer Graphics* **19** (1985), no. 3 287–296.

[57] K. Perlin, *Improving noise*, in *ACM transactions on graphics (TOG)*, vol. 21, pp. 681–682, ACM, 2002.

[58] "Perlin noise / fuzzy notepad." `https://eev.ee/blog/2016/05/29/perlin-noise`.

[59] "Pretty pictures with perlin noise fields ryan marcus." `https://rmarcus.info/blog/2018/03/04/perlin-noise.html`.

[60] "Simplex noise demystified." `http://weber.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf`.

[61] "Perlin noise: Part 2 (perlin noise)." https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/perlin-noise-part-2.

[62] "Understanding perlin noise." https://flafla2.github.io/2014/08/09/perlinnoise.html.

[63] R. Bridson, J. Houriham, and M. Nordenstam, *Curl-noise for procedural fluid flow*, in *ACM Transactions on Graphics (ToG)*, vol. 26, p. 46, ACM, 2007.

[64] "Fantasy adventure environment." https://assetstore.unity.com/packages/3d/environments/fantasy/fantasy-adventure-environment-70354.

[65] J. J. LaViola Jr, *A discussion of cybersickness in virtual environments*, *ACM Sigchi Bulletin* **32** (2000), no. 1 47–56.

[66] S. Davis, K. Nesbitt, and E. Nalivaiko, *A systematic review of cybersickness*, in *Proceedings of the 2014 Conference on Interactive Entertainment*, pp. 1–9, ACM, 2014.

[67] "Unity manual: Vr overview." https://docs.unity3d.com/Manual/VROverview.html.

[68] T. Bancroft, *Creating characters with personality.* Watson-Guptill, 2016.

[69] C. Solarski, *Drawing basics and video game art: classic to cutting-edge art techniques for winning video game design.* Watson-Guptill, 2012.

[70] "Github - keijiro/noiseshader: Noise shader library for unity." https://github.com/keijiro/NoiseShader.