The Scalability of X3D4 PointProperties: Benchmarks on WWW Performance

Yanshen Sun

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Application

Nicholas F. Polys, Chair

Doug A. Bowman

Peter Sforza

Aug 14, 2020

Blacksburg, Virginia

Keywords: Point Cloud, WebGL, X3DOM, x3d

Copyright 2020, Yanshen Sun

The Scalability of X3D4 PointProperties: Benchmarks on WWW Performance

Yanshen Sun

(ABSTRACT)

With the development of remote sensing devices, it becomes more and more convenient for individual researchers to acquire high-resolution point cloud data by themselves. There have been plenty of online tools for the researchers to exhibit their work. However, the drawback of existing tools is that they are not flexible enough for the users to create 3D scenes of a mixture of point-based and triangle-based models. X3DOM is a WebGL-based library built on Extensible 3D (X3D) standard, which enables users to create 3D scenes with only a little computer graphics knowledge. Before X3D 4.0 Specification, little attention has been paid to point cloud rendering in X3DOM. PointProperties, an appearance node newly added in X3D 4.0, provides point size attenuation and texture-color mixing effects to point geometries. In this work, we propose an X3DOM implementation of PointProperties. This implementation fulfills not only the features specified in X3D 4.0 documentation, but other shading effects comparable to the effects of triangle-based geometries in X3DOM, as well as other state-of-the-art point cloud visualization tools. We also evaluate the performances of some of these effects. The result shows that a general laptop is able to handle most of the examined conditions in real-time.

The Scalability of X3D4 PointProperties: Benchmarks on WWW Performance

Yanshen Sun

(GENERAL AUDIENCE ABSTRACT)

With the development of remote sensing devices, it becomes more and more convenient for individual researchers to acquire high-resolution point cloud data by themselves. There have been plenty of online tools for researchers to exhibit their work. However, the drawback of existing tools is that they are not flexible enough for the users to create 3D scenes of a mixture of point-based and triangle-based models. X3DOM is a WebGL-based library built on Extensible 3D (X3D) standard, which enables users to create 3D scenes with only a little computer graphics knowledge. Most of the 3D Scenes can be created with several lines of HTML and JavaScript code. Before X3D 4.0 Specification, little attention has been paid to point cloud rendering in X3DOM. PointProperties, an appearance node newly added in X3D 4.0, provides point size attenuation and texture-color mixing effects to point geometries. It applies to all point-based geometries in X3DOM and distinguishes point cloud from naive particles. In this work, we propose an X3DOM implementation of PointProperties. This implementation fulfills not only the features specified in X3D 4.0 documentation but other shading effects to produce appearance comparable with triangle-based geometries in X3DOM, as well as other state-of-the-art point cloud visualization tools. We also evaluate the performances of some of these effects. The result shows that a general laptop can handle most of the examined conditions in real-time.

Acknowledgments

First of all, I want to thank my advisor Dr. Nicholas Polys for guiding me on the research and supporting me in submitting my first conference paper in my whole life. I want to thank my committee members, Dr. Peter Sforza and Dr. Doug A. Bowman, for supporting me with the thesis. I also want to extend my appreciation to my colleagues in the Office of Analytics & Institutional Effectiveness for funding me during my master's period. Thank Dr. Jianger Yu for helping me with academic writing. Thank Mr. Daniel Hung for helping me practice my defense presentation. Finally, I want to thank my parents, Dr. Hong Wang and Mr. Zhifeng Sun, for supporting me in pursuing my master's degree overseas.

Contents

List of Figures vii

List of Tables ix

1 Introduction 1

1.1 Motivation 2

1.2 Overview 4

1.3 Problem Statement, Hypothesis, and Approach 6

2 Review of Literature 9

2.1 Computer Graphics 10

2.1.1 History of Computer Graphics 10

2.1.2 Rendering Pipeline and GLSL 12

2.1.3 X3D and X3DOM 20

2.2 Point Clouds Rendering Techniques 25

2.2.1 Performance Optimization for Large Point Clouds 25

2.2.2 Spatial Data Structure and Point Cloud Compression 28

2.2.3 Point Splatting 32

2.2.4 Lighting and Shadowing 35

2.3 Previous Works in Point Cloud Rendering 42

3 Method and Experiment **45**

3.1 Introduction of X3D Elements 46

3.1.1 Lighting and Shadow 47

3.1.2 Appearance Node 49

3.1.3 BinaryGeometry Node 50

3.1.4 Animation 53

3.1.5 X3DOM Runtime and Interaction 55

3.2 Node design 58

3.3 Experiment 63

3.3.1 Data and Device 63

3.3.2 Shader Implementation and Parameter Evaluation 64

4 Result and Discussion **73**

5 Summary and Future Work **90**

Bibliography **93**

List of Figures

1.1 Effects of point size attenuation in point cloud rendering 5

2.1 Fixed-Function Rendering Pipeline 15

2.2 Programmable Rendering Pipeline 18

2.3 Organizational Structure of X3D 23

2.4 X3DOM Project Architecture 24

2.5 Data flow between CPU and GPU 27

3.1 State panel of X3DOM scenes 57

3.2 Factors to be considered in PointProperties Implementation 60

3.3 Screen space coordinate mapping to local screen space coordinate system of a point 66

3.4 Effects of light effect in point cloud rendering 68

3.5 Effects of PointProperties on PointSet 69

4.1 Line Plots of the Relationship between the Medians of Number of Point and FPS. 75

4.2 Line Plots of the Relationship between the Medians of Scale and FPS. . . . 76

4.3 Line Plots of the Relationship between the Medians of Texture Size and FPS. 77

4.4 Pair-wise ANOVA Comparison of Medians of Texture Size and Light, Grouping by Number of Point and Scale. 84

4.5 Pair-wise ANOVA Comparison of Medians of Scale and Texture Size, Grouping by Number of Point and Light. 85

4.6 Pair-wise ANOVA Comparison of Medians of Scale and Light, Grouping by Number of Point and Texture Size. 89

List of Tables

3.1 Official definition of attributes in PointProperties 62

3.2 Parameters Examined in Performance Evaluation 72

4.1 OLS Regression Summary 80

4.2 Multi-way ANOVA Summary Table 81

Chapter 1

Introduction

1.1 Motivation

”Point cloud” is a term used to refer to a set of points with 3D coordinates and sometimes with color and normal information per point. There are primarily two methods to generate a point cloud. Firstly, point cloud can be generated by mathematical models or triangle-based geometries, which can be considered as an even, discrete sampling of contiguous surfaces. Secondly, it can be derived from 3D radar scans or photos with overlapping coverage of the real world. An example of 3D point cloud radar scanner is Light Detection and Ranging (LiDAR). The laser light beam’s high energy pulse guarantees little diffusion with distance and a precise location measurement within a very short time. LiDAR data has been used to model and identify geospatial characteristics of natural landscapes since the 1990s (Wehr et al., [81]). While carried by a drone or other aircraft, the LiDAR sensor can acquire extensive area data within a reasonable time.

As a data form that contains extremely dense sampling points, point clouds are considered as an ideal resource for elevation, vegetation, and hydrology model extraction (https://www.usgs.gov/core-science-systems/ngp/3dep). There are two characteristics that makes point cloud more and more popular in today’s geospatial researches: high precision and high accessibility. Public point cloud data providers (e.g., USGS) guarantee a three feet horizontal resolution and a centimeter-level vertical resolution. To acquire and analyze point cloud data with higher precision, researchers can even acquire millimeter-level resolution data by themselves with terrestrial or drone scanners.

In Visualization and Interface Designing, Kerren et al. [43] introduced three aspects to evaluate a visualization system: generalizability, precision, and realism. Generalizability is the ability of a system to solve a type of problem or serve different people with similar objectives. Precision measures the difference between the measured values and the actual

values. Realism is how the experiment environment is similar to the real situation to be faced. In this case, point clouds can be considered an effective way to visualize 3D data, as it provides highly accurate and detailed data of the real world. While occlusion and and discontinuous of points make the performance and perception of large point clouds a challenge. Therefore, proper methods are required for visualizing point cloud data.

Most current point cloud analysis tools (PDAL, PCL, CloudCompare, LASTools) provide statistical analysis functionalities while paying little attention to refining their point cloud visualization algorithms. Aside from statistical methods, visual validation is also an important, and even more intuitive method. A well-designed visualization method helps researchers better understand the connections among points and the relations between the raw point cloud data and the analysis results.

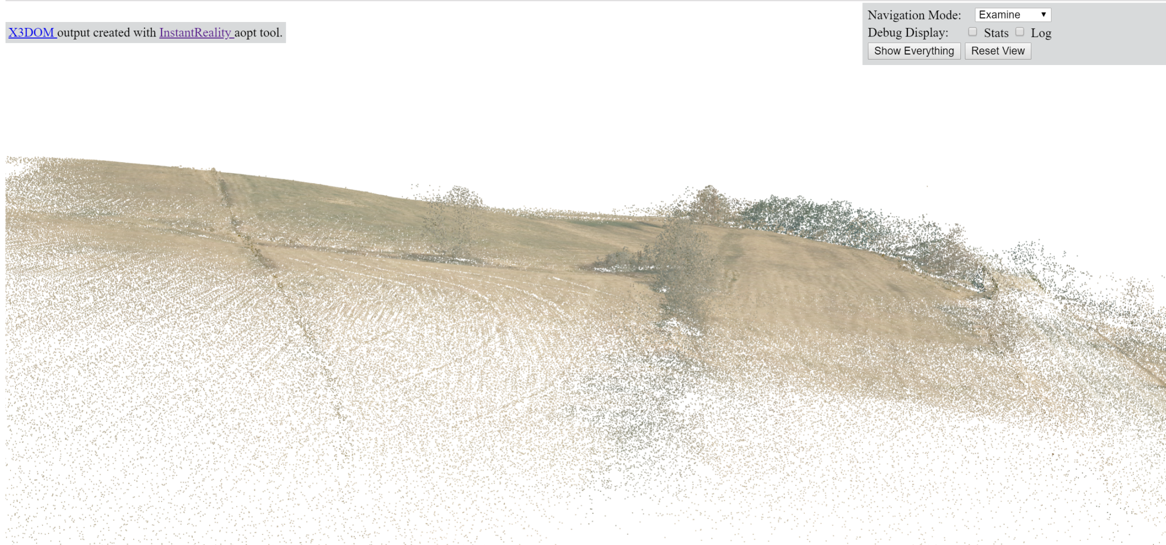
Currently, most of the point cloud analysis and visualization tools are desktop Applications (Cloudcompare[1], Meshlab[20], Paraview[2]). This type of tool is usually faster than online applications, as they save time transmitting a large amount of data through the internet. In the condition that the researchers intend to share their data's imagery to the public, online point cloud visualization tools are required to support client-side uploading and user-defined appearance. Some online point cloud viewers, like Potree[52], support data uploading. The functionality and appearance, on the other hand, are relatively limited. Therefore, it is crucial to have a tool that enables the user to define their own 3D scenes while requiring little computer graphics knowledge. Most of the well-known web graphics libraries, such as VTK[68] and three.js[76], are designed to develop web games. However, these libraries require Computer Graphics (CG) background. Therefore, an online point cloud visualization tool is needed to balance flexibility and learnability. That is, the tool should allow the users to create their own scenes with different types of geometries and appearances, while at the same time, it should be easy to learn for researchers with little CG knowledge.

1.2 Overview

With web services development, XML and HTML are becoming more and more popular formats in web information transmission. People usually have been exposed to HTML and XML code more or less, even if they are not professional programmers. Extensible 3D (X3D) is a suite of standards including an abstract scene graph content model with several equivalent encodings and language APIs. As a high-level, platform-independent declarative model, the X3D scene graph can be rendered with OpenGL, WebGL, DirectX, and POV-RAY. X3D encapsulates 3D graphic elements (geometry, material, transformation, etc.) with animations and behaviors, so that 3D scenes can be built only with a few lines of XML or VRML instead of redundant code for creating buffers and complex matrix calculations. Tools based on X3D (e.g. X3DOM, X_ITE, Xj3D) have been developed and employed for constructing 3D games and CAD applications. In the year of 2009, X3D is claimed to be the official 3D scene declaration format in the specification of HTML5. X3D content can be now be loaded with HTML anchor links and rendered in Web browsers without any plugins.

With the advent of HTML5, virtual vector graphics are supported by browsers without the help of plugins, i.e., graphics can be drawn directly through HTML coding. The X3DOM is a WebGL implementation that translates X3D content written XML/VRML/HTML into HTML contents and vice versa. X3DOM enables users to create 3D scenes and animations through HTML, which is much easier to learn than JavaScript libraries. Therefore, with the support of proper algorithms, X3DOM will become an ideal point cloud visualization tool, as discussed above. As one of the most successful X3D implementation projects, X3DOM performs well on processing primitive geometries and triangulating meshes. However, it is not fully developed for rendering and manipulating point cloud visualization yet.

Currently, X3DOM supports two basic point-based geometry nodes. *PointSet* and *ParticleSet*.



(a) PointSet with default point size (size=1)

(b) PointSet with point size attenuation

Figure 1.1: Effects of point size attenuation in point cloud rendering

There are also compressed point-based geometries nodes in X3DOM, while these nodes are just compressed versions of *PointSet* or *ParticleSet*. *PointSet* draws 1×1 point with colors. However, it is difficult to identify points and spatial relations among points due to the small and unified point sizes (Figure 1.1a). *ParticleSet* can draw point sprites if per-point size data are provided. However, lighting, shadowing, and picking are not applied to *ParticleSet*, limiting the richness of the appearances of points. The X3D community has paid more attention to point-based geometries in the most recent version of X3D. In X3D 4.0 Specification, an appearance node called "*PointProperties*" is proposed to display point-based geometries on the screen better. This Specification provides basic point size attenuation and texture-point color mixture rules. While lighting, shadowing, and normal interpolation are also important in a general point cloud visualization tool.

1.3 Problem Statement, Hypothesis, and Approach

This research intends to make the appearance and rendering speed of point cloud in X3DOM competitive with state-of-the-art point cloud visualization tools. We explored the algorithms used in state-of-the-art online point cloud visualization tools (e.g., hierarchical structure, lighting, shadowing, and point splatting) and methods that are claimed to be useful in point cloud rendering. Algorithms currently employed by X3DOM are also examined. These algorithms are popular in mesh-based rendering, while not commonly used in point cloud visualization. Therefore, problems arise: (1) how to fuse techniques of state-of-the-art point-based rendering with those used in X3DOM, and (2) can our implementation of point cloud rendering method be executed in real-time in a web-browser of an ordinary personal computer/laptop? Our hypothesis are: (1) *PointProperties* can be implemented by combining the techniques of X3DOM and state-of-the-art point cloud visualization tools, and (2) the

functions of *PointProperties* can be executed in real-time in a web-browser environment of an ordinary personal computer/laptop.

The contributions of this research are listed below. First of all, we proposed a benchmark of how to develop an appearance node in X3DOM. Then, we implement an appearance node called "*PointProperties*" for X3DOM. Our implementation includes point size attenuation, texture and points color mixing, normal estimation, and interaction with other nodes (e.g., material nodes, texture nodes, geometry nodes, and light nodes). Most state-of-the-art tools employ the same set of algorithms to deal with point cloud visualization problems, such as spacing-based point size attenuation, sphere/Gaussian/Parabolic normal interpolation function, and eye-dome lighting model. To be consistent with X3D 4.0 Specification and the other appearance nodes, our implementation of *PointProperties* adopts distance-based point size attenuation and Blinn-Phong illumination model. With this shader, X3DOM can provide "a better visualization mode" for point cloud data, becoming a practicable visualization tool for point cloud researchers. More specifically, "a better visualization mode" refers to a mode the spacial relationships among points are emphasized. More color and illumination effects are available to achieve different visual effects based on the users' demand. Finally, a performance evaluation is required, as our implementation employs a set of methods for point cloud visualization that is different from the other tools. The experiment is conducted with varying parameters (defined in *PointProperties*) to better understand the relationships and ranges of performance per parameter in X3DOM. The experiment provides guidance for *PointProperties* parameter ranges and trade-offs.

The other chapters are organized as follows. Chapter 2 introduces the techniques used in existing point cloud rendering web applications, the X3DOM project, and techniques used in our implementation. Chapter 3 describes the method of integrating our "*PointProperties*" node to X3DOM, as well as how the performance of our implementation is examined. Chapter

4 discusses the result of performance evaluation and future work is extended in Chapter 5.

Chapter 2

Review of Literature

During the past few decades, plenty of methods have been developed to draw point-based geometries on the screen. Some of them focus on enhancing the spacial relations among point. Others pursue high frame rates while rending large amount of data points. In this chapter, Section 2.1 introduces the mechanism a Computer Graphics Unit (GPU) rendering 3D scenes and the tools used to manipulate the rendering process. Then, techniques used for online point cloud rendering are presented in Section 2.2. Finally, Section 2.3 introduces techniques that are used in several state-of-the-art web point cloud visualization tools.

2.1 Computer Graphics

To interpret the relationship between the rendering speed and properties of a 3D scene, researchers should acknowledge how GPU processing 3D data. Section 2.1.1 presents a brief history of computer graphics and how CG techniques are applied to web 3D rendering. Section 2.1.2 describes the strategies of data exchanging among CPU, GPU, and the memories, as well as rendering pipelines of GPU. Finally, the architectures of the tools used in this research, X3D and X3DOM, are introduced. This picking method is discussed in Section 2.3.

2.1.1 History of Computer Graphics

A modern graphics processing unit (GPU) is defined as an electronic circuit that specifically designed to rapidly generate images from 3D and drawing the images on the screen. The procedure of generating images from 2D or 3D models is also called ”render” in the field of computer graphics. Although devices have been produced to accelerate graphical display since the 1950s, not until 1998, modern GPU – standalone hardware device that accelerates

both 2D and 3D model rendering – has been invented. Initially, GPUs are only responsible for rasterization and texture mapping. In the year of 1999, NVIDIA promoted ”the world's first GPU”, GeForce 256. GeForce 256 is the first consumer-grade GPU and one of the most popular second-generation GPU. Aside from rasterization and texture mapping, the second generation GPU supports 3D Object Transformation and Lighting (T&L) as well as cube map. In 2001, GPUs started to provide vertex programmability, which made them the third generation GPU. The fourth generation of GPU provides fragment programmability in 2003. Afterward, more and more work-load are moved from CPU to GPU. Meanwhile, GPU opens up more and more sections in the GPU rendering pipeline for programming. Current GPUs also fuse more and more rendering functionalities, such as real-time ray tracing, artificial intelligence, and programmable shading.

With the development of hardware performance, more customization is expected for the software part of the GPU. Nowadays, Khronos OpenGL (Open Graphics Library), Microsoft DirectX, NVIDIA NVAPI are the world's most popular programming interface (API) for rendering 2D and 3D vector graphics. While DirectX can only run on Windows operating system and NVAPI only works for NVITIA's GPU, OpenGL is a cross-language, cross-platform API, and thus widely adopted by web applications and embedded systems. The first version of OpenGL was first published in 1992. In the year of 2004, OpenGL 2.0 was published to support the programmable pipeline. However, it hadn't become popular until ATI (Another GPU company aside from Nvidia, acquired by AMD in 2006) produced a number of OpenGL extensions.

Nowadays, most of the web3D applications are developed based on WebGL, which wraps ”OpenGL for Embedded Systems” (OpenGL ES/GLES) with a set of JavaScript interfaces. All mainstream web browsers currently support WebGL without any plugins. To manipulate client-side GPU, web browsers translate WebGL and OpenGL ES calls into platform-specific

APIs of the corresponding operating system through graphic engines, e.g., Almost Native Graphics Layer Engine (ANGLE).

2.1.2 Rendering Pipeline and GLSL

GPU rendering pipeline is a standard workflow for GPU to turn 2D and 3D models into pixels to be displayed on the screen. Nowadays, three major GPU programming languages (DirectX, NVAPI, OpenGL) employ different rendering pipelines and different shader languages. X3DOM is implemented in Javascript on top of the WebGL renderer (a WWW-compatible subset of OpenGL).

Fixed-Function Rendering Pipeline

Before OpenGL 2.0, OpenGL only provides APIs for users to specify configuration arguments for the rendering process. This kind of pipeline is called fixed-function pipeline. Fixed-function pipeline of OpenGL consists of six major stages: ”vertex specification”, ”transform and lighting”, primitive assembly”, ”rasterization”, ”texture environment”, and ”per-sample operation” [84, 85].

”Vertex specification” stage is to pass vertex data from CPU to GPU in a form of Vertex Array Objects (VAO) and Vertex Buffer Objects (VBO), where VAO defines what are included in the vertex data and VBO stores the exact data.

The ”transform and lighting” stage prepares geometrical attributes for light effect calculations and conversion from the object’s local coordinate systems to the screen space coordinate system. During this process, the coordinates and normals of an object are first transformed from the object’s local space to the world space, so that they are in a unified coordinate system. The second step is to project the objects to the view space. The view space is

based on a frustum in the world space. The frustum is defined by camera position (tip of the frustum), camera orientation (axis of the frustum), as well as the width, height, and positions of the near plane and the far plane (two bases of the frustum). The x and y axes are on the near plane, parallel to the height and width of the plane. The z-axis is the axis of the frustum. Before projected to the screen space, the frustum is first re-scaled to (-1, -1, -1) to (1, 1, 1). The normalized frustum is called Canonical view volume, (CVV). Objects outside CVV are clipped out to improve rendering performance (Viewing-Frustum Culling). Thus, the space defined by CVV is also called clipping space. The final step is to project CVV to screen space. The near plane is mapped to the viewport, and collision is calculated along the frustum's z-axis.

The "primitive assembly" stage turns vertices into triangle-based primitive and executes triangle level clipping (Viewport Culling, Back-Face Culling (Optional), and Occlusing Culling (Optional)). Viewport Culling is to clip out primitives outside or partially outside the viewport. Back-Face Culling discards faces that are back on the camera. Occlusion Culling eliminates primitives occluded by the others.

"Rasterization" stage is to mapping geometric primitives to a grid that to be mapped to screen pixels. The cells of this grid are called fragments. Line drawing algorithm and area filling algorithm are executed in this stage.

"Texture environment" is to combine textures with fragment color, lighting, and fog effects. Texture data are loaded outside the rendering pipeline. After the fragment data are prepared, texture data are sampled and mapped to fragments. Then, texture color is merged with fragment color, lighting, and fog effects.

After acquiring the per-fragment data, the final stable is to map fragments to the screen pixels. "per-sample operation" include a series of screen-level test and fragment-to-pixel

mapping rules. Pixel ownership test is to exam if a screen pixel should be rendered. Scissor Test is to cut off fragments mapped to pixels outside a portion of the screen. Alpha test is to discard fragments with alpha values outside the determined range. Stencil Test is usually used to limit the area of rendering or accelerate depth test. Depth test is to eliminate fragments that are occluded. Generally, if all the objects are opaque, only the fragments closest to the viewpoint are kept. For transparent fragments, if not occluded by opaque fragments, pixel color is a mixture of the color of all the fragments that are visible from the viewpoint. This process is called color blending. The contribution of a fragment is usually defined in the alpha channel of RGBA color mode. Aside from the commonly used functions listed above, there are also functions such as Dither, Logical Operation, and Write Mask. The final generated pixel data are stored to a framebuffer and drawn into the screen.

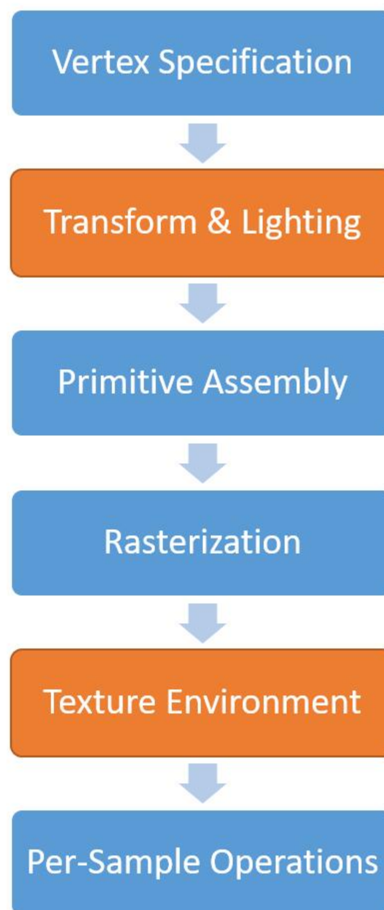


Figure 2.1: Fixed-Function Rendering Pipeline

Programmable Rendering Pipeline

As the demand for high performance and realism visual effects grows, several stages in the fixed-function pipeline are opened to user-provided programming for further improvement of performance. These programmable parts of the pipeline are also called ”shaders”, which means to produce an image with a mixture of light, texture, and color. At first, there are only two programmable shaders. The first is ”vertex shader”, which is used to replace the ”transform and lighting” stage of the fixed-function pipeline. The vertex shader enables a user-defined calculation of geometry features of the vertices. The second shader is called ”fragment shader” (also known as ”pixel shader”). This shader is used to define ”texture environment” and ”alpha test” in the fixed-function pipeline. Afterward, the ”primitive assembly” stage was divided into tessellation, geometry shader, vertex post-processing, and primitive assembly. In OpenGL, Geometry shader was first opened to user-defined programming, and then tessellation was partially exposed to shader programming. Geometry shader was added to the pipeline in OpenGL 3.0 (2009). It is a programmable stage that defines the amount and attribute of primitives generated from each primitive in the former stage. The tessellation stage splits surfaces into primitives on the GPU to amplify the details. In OpenGL 4.0 (2010), tessellation was subdivided into three parts, of which the first and the third parts are programmable. The Tessellation Control Shader (TCS) defines how patches with different Level of Detail (LoD) joint with each other. Then, the patches are subdivided by tessellation primitive generator (not programmable), according to the output of TCS. The Tessellation Evaluation Shader (TES) calculates vertex values for each vertex generated from subdivided patches. WebGL does not yet support geometry shader, TCS, and TES. [87]

Programmable shaders enables user-defined algorithms in the most heave-loaded sections in a rendering pipeline. Lighting, texture mapping, and some other calculations are no longer

necessary parts in a rendering pipeline, which saves resources and time used for rendering.

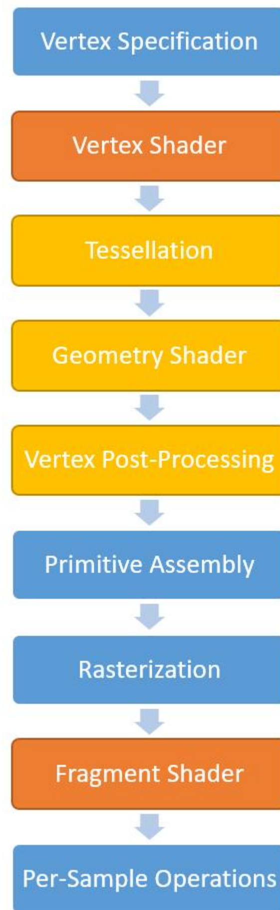


Figure 2.2: Programmable Rendering Pipeline

OpenGL Shading Language (GLSL)

The most popular real-time shading languages right now are OpenGL - GLSL (OpenGL Shading Language), NVidia - Cg (C for Graphic), and Direct3D - HLSL (High Level Shading Language). In this paper, only GLSL for WebGL is discussed. In WebGL, shading language is used to program vertex shader and fragment shader. Shaders are functions that to be executed per input element [87]. Most of the computations executed in shaders are based on matrices and vectors. Therefore, aside from general variable types, shader languages also provide data types of vector and matrix from 2D to 4D, as well as fast matrix computations. Except for data type, three keywords are used to define the usage of data in GLSL 3.x [86]: attribute, uniform, and varying. ”Attribute” refers to per-vertex data pulled out from vertex buffer object. In vertex shader, attributes are coordinates, normals, and colors of vertices. There is no need to define ”attribute” for fragment shaders, as they acquire all data provided by the former stage of the pipeline. ”Uniform” indicates variables stay the same for all vertices in a draw call. Transformation matrices, texture samplers, and other user-provided global parameters are usually assigned as ”uniform”. ”Varying” is used to mark attributes to be passed from vertex shader to fragment shader. Corresponding attributes need to be of the same name in both vertex shader and fragment shader.

In WebGL shader, a series of binary vertex data (attribute) and some global variables (uniform) are fed to the rendering pipeline. The vertices are then processed by vertex shader individually. Among the vertices’ attributes, the geometrical ones are transformed from model space to clipping space and then passed to the next stage as needed. Colors and texture coordinates, if any, are directly sent to the next stage. Fragment shader is not a necessary stage. If fragment shader is not defined, fragment color is automatically interpolated according to vertex color. Otherwise, texture mapping and lighting and fog effects can be defined in this stage. Alpha test and blending can also be implemented by modifying alpha

values of fragment color.

2.1.3 X3D and X3DOM

Extensible 3D (X3D) Graphics is a royalty-free open standard proposed and maintained by Web3D Consortium [17]. It provides standards defining a browsable and interactive 3D scenes on the Web, enabling people to create 3D scenes with only a little programming. Instead of complex buffer operations, in X3D, the user feeds data and configuration arguments to the specification. X3D inherits from Virtual Reality Modeling Language (VRML) and now supports XML encoding .x3d, Classic VRML encoding .x3dv, VRML97 encoding .wrl and JavaScript Object Notation (JSON). X3D files can be viewed by WebGL-enabled browser (or browser with Flash 11 plugin) with InstantReality plugin installed. Some desktop 3D browsers such as ParaView and MeshLab supports X3D formats. Web3D Consortium also provides its X3D viewers, such as InstantReality, FreeWRL/FreeX3D, H3D/H3DViewer, Xj3D, and X_ITE. As a 3D graphics application architecture, X3D has been bonded to programming languages, such as JavaScript and Java, for users to extend it easily.

As the first generation of API for programmable rendering pipeline was proposed in 2004, the architecture of X3D was designed based on the fixed-function rendering pipeline (Section 2.1.2). A 3D scene usually contains several elements: 3D and 2D objects, motions of the objects, environmental factors, and interactions between users and objects. Based on these elements, the components of X3D can be divided into seven types: primitive components, navigation components, animation components, interactivity components, environment components, status components, and sound components (Figure 2.3). Primitive components define the 3D and 2D objects in a 3D scene. In an X3D scene, an object consists of 2 parts: geometry and appearance. Geometry can be single-mesh nodes (point, line, and polygon),

geometric primitives (box, cone, cylinder, and sphere) and more complex combination of meshes such as extrusion, point set, particle system, elevation grid, and nonuniform rational Bézier spline (NURBS) parametric surfaces. Appearance defines how the geometry looks like with material color and texture. Shader nodes are added as child nodes of appearance node in the third version of X3D. The combination of geometry and appearance is called ”shape” in the node definitions of X3D. Shape nodes can be organized with higher-level nodes called ”group node”. Navigation components are a set of nodes that defines the motion of viewpoint as well as frustum attributes. Environment components include nodes with global effects on the 3D scene, such as lighting, background color, fog, and sound, and environmental sensor. Environmental sensors handle environment-object interactions, such as visibility, proximity, and collision. Animation components define how a shape or a group moves over time and how the action interpolated between critical points. Interactivity components determine how objects in the scene react to user actions such as mouse-clicking, mouse-dragging, and key-pressing. Developers can also specify customized event listeners by inserting ECMAScript nodes. To reduce resource load during rendering, X3D employs a directed acyclic scene graph to define the location relationships and appearances of the 3D world. All the nodes can be reused with the combination of attribute ”DEF” and ”USE” to avoid re-loading the same data sources. Currently, X3D provides different profiles to support various functionalities in the field of gaming and AR/VR, engineering and scientific visualization, CAD and architecture, Geo-spatial, Human animation, 3D printing, 3D Scanning, medical visualization, training and simulation, multimedia, entertainment, education, and more.

X3DOM is a DOM-based integration framework for declarative 3D graphics in HTML5. The architecture of X3DOM can be divided into three parts [40]: the user agent, the connector, and the X3D runtime. The user agent defines how DOM tags and their attributes map to 3D elements. The connector is a JavaScript-layer that translate DOM elements into X3D decla-

rations by handling the syntax differences between HTML and XML/XHTML, supporting standard JavaScript and jQuery functions of insert/removal/select elements and attribute changes, and reacting to standard HTML events (onClick, onKeyPress, onMouseMove, etc.). The last but most important part of the architecture of X3DOM is X3D runtime. X3DOM supports different types of X3D runtime backends, while in this project, a WebGL implemented backend is employed.

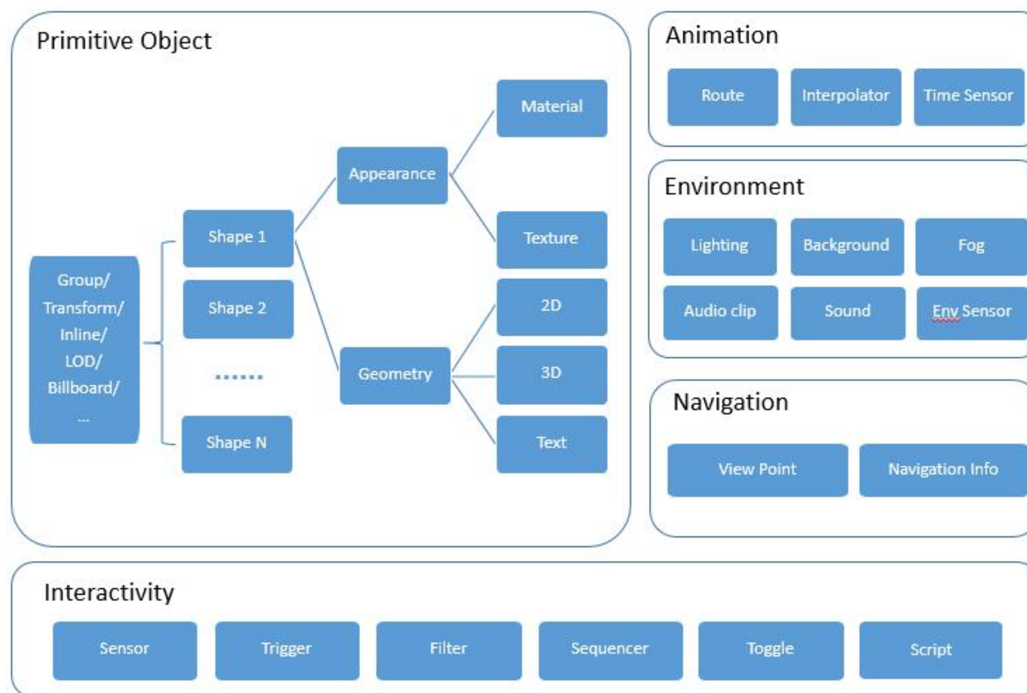


Figure 2.3: Organizational Structure of X3D

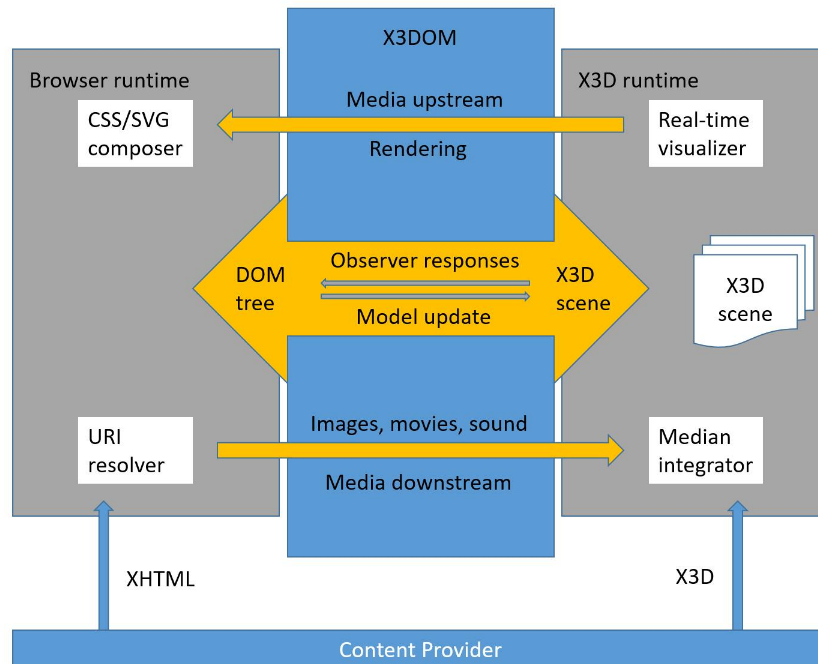


Figure 2.4: X3DOM Project Architecture

2.2 Point Clouds Rendering Techniques

Generally, a point cloud visualization tool contains at least four functionalities. The first is to create hierarchical structures for multi-LOD (Level of Detail) visualization. Using hierarchical structures enables the tool to render a larger amount of points. The second function is to vary the size of points, also known as ”point splatting” techniques. Most of the point splatting methods aim to fill the gaps among points, while the size of points can also indicate other attributes of points (e.g., the distance to the camera). Besides, Lighting and shadowing effects are also used to enhance the visual effects of spacial differences among points. Finally, point picking should also be implemented for users to interact with individual points. GPU picking is usually considered outstanding in both precision and performance, and thus employed by almost all web 3D scene development tools. Asides of the four functions mentioned above, some visualizations tools also leverage optimization algorithms to improve the performance.

2.2.1 Performance Optimization for Large Point Clouds

Performance optimization aims to render more frames within a second while maintaining the rendering quality per-frame. Most people will not notice the lack of fluidity when the FPS (frame per second) is larger than 8, while 10 FPS is generally considered the minimum standard of frame rate. 30 FPS is the recommended standard of non-interactive entertainment (movies); 60 FPS is more beneficial for interactive entertainment (video games), as it takes less time reacting to player inputs. In VR games, 90+ FPS is required, and 120+ FPS is recommended, to synchronize image changes with head movement. Rendering quality generally means the number of details a frame presents, which is usually affected by geometrical resolution, spectral resolution, aliasing, blending, and lighting/shadow effects.

During the past three decades, computer hardware such as CPU and GPU have been developed for faster and higher-quality graphic rendering. CPU is designed to process data in a single/several process mode while GPUs are designed to work parallelly. Generally, CPU owns 1 to 16 cores with quick-access to the memory of several gigabytes. A modern GPU usually contains a group of computing units and a large number of cores within each compute unit, enabling GPU to process thousands of vertices/fragments/texels at the same time. On the other hand, the directly accessible memory is rather small (less than 1GB). In a rendering process, the data are first pulled from the disk to RAM. These data are then processed by the CPU and divided into batches. The batches are bound to GPU memory one by one through API. Different types of data are assigned to different GPU objects with some rules (e.g., geometry to vertex buffer object (VBO), texture to texture object, and transform matrices to uniform parameters of shaders). Through these mapping rules, data are transported from RAM to the Video Random-Access Memory (VRAM) of GPU. Finally, GPU makes use of these data and produce FrameBuffer Object (FBO) as the output of the whole process (Figure 2.5). Among all the data interchange procedures, the slowest part is the interchange between RAM and VRAM. Therefore, in the aspect of programming, three stages can be improved while rendering large point could: data pre-processing in CPU, data exchange between CPU and GPU, and rendering in GPU.

There are two principles in optimization to increase FPS. The first is to reduce the overall data in the workflow, while the second is to make use of both CPU and GPU resources sufficiently. On the CPU side, the number of vertices can be reduced by sub-sampling or denoising; a batch should contain thousands of vertices and one material object, to save the time CPU sending ”interchanging” data request and fully leverage the parallel structure of GPU. In terms of data transition between RAM and VRAM, compression algorithms reduce the size of the data stream and thus shorten the time in data transition; rendering

batches sharing the same data together reduces possible data replacement in VRAM. On the GPU side, hierarchical structures help to filter out vertices representing unnecessary detail data. Techniques like view-frustum culling and back-face culling eliminate fragments to be rendered to the framebuffer. Unnecessary multi-pass rendering for lighting and shadow should be avoided. Data (coordinates, colors, normals) should be stored in a large array in an interleaved manner to ensure the fillrate of GPU cores and reduce the time searching for different data of the same node.

Among the optimization methods listed above, most of them can be easily implemented, except for the compression and hierarchical structure algorithms. Therefore, this chapter introduces previous work of compression and hierarchical spatial data structure and blending and lighting methods working well specifically for point cloud rendering. To better depict how the algorithms work together, benchmarks of several state-of-the-art point cloud rendering tools are explored.

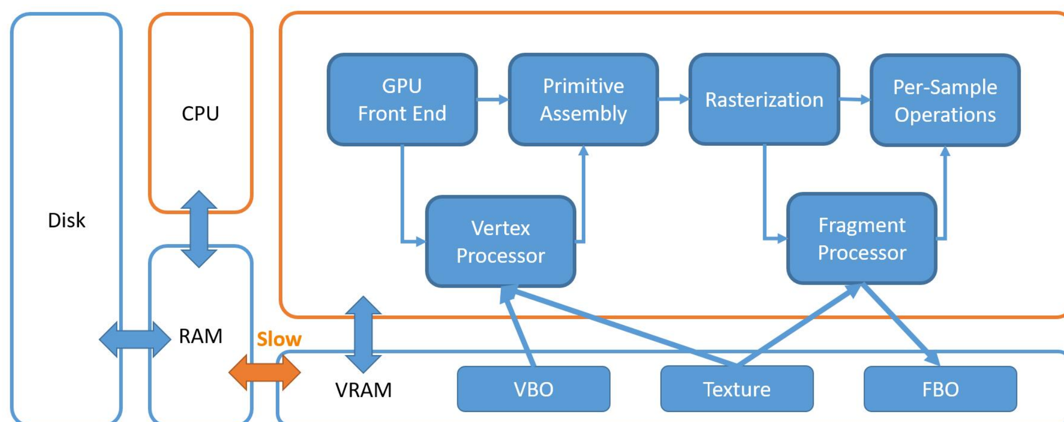


Figure 2.5: Data flow between CPU and GPU

2.2.2 Spatial Data Structure and Point Cloud Compression

In an ideal situation, the fastest way of rendering a scene is to load the entire 3D scene to the GPU. Considering the size of CPU and GPU memory of a current personal computer (PC) and the number of points in a point cloud, it is usually impossible to load and render the whole point cloud at once on a high-performance PC, let alone ordinary PC and mobile devices. To reduce the time used in data loading and storage, one can (1) reduce the size of the data, (2) only load the most important part of the data. Point cloud Compression is an element-level technique that aims to limit the bits used for data storage. It reduces the size of the point cloud file by compressing the number of bits required for storing each point. Therefore, compression reduces the time spent on data transportation among PC hardware and thought the internet. The second method is to organize the point cloud in a hierarchical structure. This method can quickly locate points with specific spatial properties, and thus enabling rapid loading and rendering. Hierarchical structures can also reduce the number of data points per frame. Besides, constructing hierarchical indices for the data, algorithms of node searching, exchanging, and predicting should also be developed to render objects with different LOD efficiently.

Point Cloud Compression

Point cloud compression, also called point cloud encoding, is a process that converts a point cloud of other formats into a format requiring less memory to store. Typically, point cloud compression includes three parts[71]: geometry compression, attribute compression, and dynamic motion-compensated compression. Geometry compression is to compress coordinate data of the point cloud. Attribute compression focuses on compression of color, normal, texture coordinate, and other attributes. Dynamic motion-compensated compression dy-

namically updates the compressed point cloud structure according to the insertion/deletion/edition operations of the original point cloud. Most of these compression methods can be either lossy or lossless, depending on how the parameters are set.

Some compression algorithms compress point clouds by saving data in memory-efficient data structures such as [35, 55, 67]. For this type of compression methods, the mainstream idea is to divide the bounding box of the point cloud into voxels and record these voxels instead of the original XYZ. Generally, the voxels are organized with an Octree or kd-tree. Schnabel[67] first proposes to use the Octree structure for point cloud compression. Huang et al. [39] developed a progressive point cloud compression based on Octree. Later, Octree is used in dynamic point cloud encoding in [53] and [42]. Aside from octree, quadtree can also be used by considering the point cloud as a serial of images.[74] The images can then be compressed and transmitted as videos.

Due to the sparsity of the voxel representation of point clouds, some algorithms compress point clouds by constructing a graph. Cohen et al. compact points with block based prediction and shape adaptive Discrete Cosine Transform (DCT) in [21] and [22]. The approach encodes dynamic point clouds with video encoding techniques. These techniques are similar to methods used in intra frame prediction, motion estimation, and compensation.

Since critical points are distributed irregularly across the images, Tian et al. [77] used graph transform to represent the critical point trajectories. This method makes the encoding more efficient than traditional DCT based transformation, making it easier for energy compacting. Thanous et al. [75] proposed to compress color, texture, and motions by extending video compression techniques from 2D to 3D.

Besides, there are also algorithms compressing similar subsets of point cloud by clustering[88, 89]. This type pf algorithm focus on compressing color attributes separately from geometry

and mapping the color back to the points during decoding.

Two common point cloud compression tools are PCL [60] and Draco [16]. PCL employs Kammerl's method [42]. It compresses point cloud by converting raw data to voxels and encoding the voxels with an Octree. Points within each voxel are transformed into the voxel's local coordinate system. The compression ratio can be adjusted by changing the voxel precision. The attributes are compressed the same way as geometry. For example, point colors are encoded by recording the mean color value of the voxels and the divergences of each point. For dynamic motion-compensated compression, Kammerl proposes "double-buffer Octree" to store the changes of nodes in a second buffer and update the first buffer during the next compression.

Draco employs and improves the kd-tree model proposed by Devillers[26]. Draco's model bisects the point cloud in the middle of each axis that keeps the point cloud as clustered as possible. The subsets of the point cloud are then further split until the size of the node is 1. With this binary tree structure, a node in the tree can be described by its segment coordinates and the number of points of its child node with fewer points.

Hierarchical Structure

Similar to compression, tree structures are also used to organize the hierarchical data structure of point clouds. Generally, hierarchical structures for 3D object space can be divided into two types: space-partitioning structure and data-partitioning structure. Octree and kd-tree are the most famous space-partitioning structure for point clouds, while Bounding volume hierarchies (BVHs) is the representative of data-partitioning structures. For point cloud rendering, spatial-partitioning structures can be extended to BVHs as there is no connectivity information among points. Aside from octree and kd-tree, R-tree, B-tree, and

clustering algorithms can also be used as point-grouping strategies for BVHs[34]. Surfels[56] and QSplat[48] are the first ones proposed hierarchical point-based rendering. The method of Surfels is to construct an Octree structure by storing sampled points at each level. QSplat represents points in non-leaf nodes with a large sphere defined by the size and means values of attributes of the points within the node. Both of Surfels and QSplat choose breadth-first search (BFS) for tree serialization. Dachsbacher et al. [31] was the first one moving hierarchical data structure construction to GPU. With the help of instance rendering technique, point clouds can be assigned to different chunks while stored in the same vector buffer. Gobbetti and Marton [24] implements the Level of Detail (LOD) approach by splitting the point cloud into chunks. In the LOD structure, Low-level nodes contain uniformly sub-sampled points on the surface within their coverage. Therefore the approach ensures that each point of the original data set is assigned to exactly one binary tree node.

Scheiblauer and Wimmer[65] proposed a method called modifiable nested octree (MNO). The modifiable nested octree structure stores sub-samples of the original point cloud in each node. The original MNO subsamples the point cloud through an inscribed three-dimensional grid with 128^3 cells. Initially, points are added to the root node one by one until the number of points in the root node exceeds a certain threshold. The excess points are accumulated in a temporary buffer until there are enough points for a new child node. This method avoids empty nodes and thus saves the memory. However, it does not guarantee a certain minimum distance between points, which leads to an uneven distribution of points in low LOD views.

He and Liang[36] create a hierarchical index and simplify the point cloud by combining QSplat techniques with curvature simplification. The approach first constructs a hierarchical structure through hierarchical clustering, which assigns each point to a leaf node of a kd-tree. Then ”large points” are generated bottom-up from the leaf nodes to the root node. The color, position, radius, and normal of a ”large point” is the averaged or normalized sum of

its children. The point cloud is then further simplified by discarding leaf nodes with a small deviation of normal angles. To deal with dynamic changes of the point cloud, He and Liang keep the parent nodes of rendered nodes in a link-list, and traversal the link-list instead of the entire hierarchical structure. Discher et al. .[27] built a multi-level point-cloud renderer with a kd-tree for virtual reality usage. They also programmed for view-frustum culling, occlusion culling by themselves to improve the performance. For example, the points are drawn in a front-to-end order to avoid the unnecessary drawing of occluded fragments.

All LOD techniques mentioned above are called ”discrete LOD” which means that space is divided into cubes and points in each cube are loaded or unloaded as a whole. Schütz et al. [70] propose the concept of ”continuous LOD (CLOD)” which targets to avoid sudden change of point density during rendering. The algorithm repeatedly creates a reduced low-resolution version of the full point cloud at runtime. The sub-sample spacing is a function of the 3D distance between a specific point and the camera. As the update is only done every few frames and there are no calculations for hierarchical traversal, CLOD's performance is compatible with the other state-of-the-art LOD algorithms.

2.2.3 Point Splatting

Splatting is a technique that developed to enhance the visual quality of point-based rendering. The idea of splatting is to represent points in point clouds with a square/elliptic disk/sphere. The quality of a splatting method can be evaluated from three perspectives: the smoothness of the surface formed by splats, the size splats, the contiguity of adjacent splat colors. By properly expanding the coverage and blending each point's color, the points joints with each other form smooth and continuous surfaces. In the era of the fixed-function rendering pipeline, algorithms can only be written and execute on CPU, which limits the operations

in the screen space. In 1985, Levoy and Whitted proposed rendering points as dots instead of triangles onto the screen[49]. However, rendering based on points were mainly used for volume rendering at that time, because of the high cost of CPU anti-aliasing and the discontinuity of surface generated from single-pixel points. Not until 2000, after anti-aliasing was implemented on GPU and programmable shaders was added to the GPU rendering pipeline, point-based rendering has become a mainstream rendering method.

As GPU is designed for triangle-based rendering acceleration, researchers need to develop specific methods for different stages of point-based rendering. Generally, there are six stages to be defined to create a fully functional point-based rendering system: pre-processing, hierarchical structure construction, filing and loading strategy, node attributes calculation, visibility culling, and blending. Pre-processing consists of methods used for retrieving point properties, such as resampling the point data and calculating normals. Hierarchical structure construction means to create spacial indices for the point data to establish LOD rendering. Filing and loading define the strategies for data caching and hierarchical structure traversal. Node attributes calculation is to infer attributes of each hierarchical node from the points. Visibility culling is to eliminate fragments that are not to be rendered on the screen. Blending is to draw the splats on the screen with proper color blending strategies.

Surfel[56] and QSplat[48] are the first ones proposed benchmarks for point-based rendering. The pre-processing stage of QSplat is to acquire normal and spacing of each point from the original data (if mesh provided) or calculated from each point and its k-nearest neighbors(kNN). QSplat creates a hierarchical sphere structure with median-cut kd-tree (k=2) in top-down order. The attributes of each node are then calculated by traversing the kd-tree bottom-up. For each node, the color is the mean color of its child nodes; the normal cone is the collection of the normals of the points within it; the size is the maximum spacing this node to its neighbors plus the sum of the sizes of its child nodes. For visibility culling,

nodes are kept if they are entirely lying inside the view frustum. Faces are eliminated if a node's normal cone faces entirely away from the viewer. During rendering, the system stops loading child nodes of a node if (1) the number of loaded points exceeds a threshold, (2) the node does not pass culling tests, or (3) the node itself is a leaf node. For blending, QSplat employs a two-pass rendering to blend points within a depth range while maintaining the correct occlusion. Three types of representations of points are examined: non-antialiased OpenGL point, opaque circle, and fuzzy spot with an alpha that decays radially in a Gaussian manner. Pfister et al. [56] propose a rendering method based on the concept of surfel. Similar to pixel, voxel, and texel, surfel represents an oriented unit square (voxel face) with texture in a voxel. The method casts rays from three axis-aligned orthographic viewing directions to get sub-samples of the surface.

The pre-processing stage leverages the three surface samples of the same block to generate a tangent disk to represent the surface within the block. The block's size is assigned to be the size of the surfel; the color is rendered to the texture space with an Elliptical Weighted Average (EWA) filter applied. The hierarchical structure is an Octree built bottom-up based on the surfels. Finally, before rendering to the screen, holes in the screen space are filled with a radially symmetric Gauss filter. Afterward, Surface Splatting[92] is proposed to improve the point sampling and blending techniques of surfel with an anisotropic anti-aliasing EWA filter.

To improve the per-splat rendering quality, researchers Botsch et al. [12] proposed a GPU-friendly three-pass blending points algorithm. DepthBuffer is first rendered to eliminate occluded fragments. Then the attributes of overlapping fragments are blended by modifying their alpha values. Finally, the blended colors are normalized and delivered to the framebuffer. To deal with point cloud without normals, Scheiblauer et al. [64] suggest to rendering uses screen-aligned circles instead of oriented ellipse disks. Phong splatting[13]

modifies fragment normals to render each point as a hemisphere instead of a planar disk. Zwicker et al. [93] propose a clip-lines method to preserve sharp edges. Sigg et al. [73] render splats as quadric surfaces projected on the tangent planes of the splats. This method is later combined with surface splatting by Weyrich et al[82]. He and Liang[36] store the Gaussian filter in texture during pre-processing and thus achieve view-independent splatting and blending. To reduce the blurriness of splat blending while filling the hole pixels, Schütz and Wimmer[69] proposed a method to quickly render point cloud by combining the re-projected points from the last frame and a set of randomly generated points. Sibbing et al. [72] proposed several methods to interpolate color to holes in the rendering images. Bui et al. [18] improve Sibbing's method with deep learning techniques.

There are also methods proposed based on surface reconstruction techniques or object-space resampling. Alexa et al. [4, 5] thin the point cloud by generating a moving least squares (MLS) surface from the original point cloud and then replace the points with their projection on the MLS surface. This approach enables fast down-sampling and up-sampling of the point cloud data and thus forms smooth and continuous surfaces with the assistance of QSplat technique[48]. This kind of method works better with photorealistic illumination models by leveraging the local surfaces.

2.2.4 Lighting and Shadowing

In point cloud rendering, lighting and shadowing are important in representing and even enhancing the spacial relationship among points. Illumination models, also known as shading model or lightning model, is used to simulate the visual effects light reflected from different surfaces. There are three main characters in an illumination model: the light source, the surface, and the observer. Light source can be divided into three types by source: point light,

parallel light, and spotlight. Directional light emits rays parallel to a given 3D vector. Point light evenly emits rays from a 3D point to all the directions. Spotlight shades light from a single point to a direction within a solid angle. The position, electromagnetic spectrum, and shape of light sources determine the lightning effect.

Reflective surfaces usually generate either diffuse reflection or specular reflection. Diffuse reflection is caused by coarse surfaces, which tend to reflect light equivalent to all directions. The idea surface that generates only diffuse reflection is called Lambertian reflectance. Smooth surfaces tend to reflect most of the incident rays in one direction, which is known as specular reflections. The reflectance position and properties are important parameters for illumination model computation. Besides, in more complex models, the positions of nearby surfaces are also crucial for shadow and multiple reflectance calculations.

To simulate the real-world illumination, the light environment in a 3D scene usually combines several illumination models. The most simple illumination model is called ambient light, which is to imitate the non-directional and constant light generated by multiple reflections among walls and other objects. The reflected intensity of ambient light can be calculated by:

$$I_{amb} = k_d I_a$$

I_a is the intensity of ambient light; $k_d(0ltk_dgt1)$ is the ambient reflectivity of materials. The equation of diffusion reflection of Lambertian reflectance was proposed by Gouraud[33] in 1971.

$$I_{diff} = k_d I_l (N \cdot L)$$

Where I_l is the intensity of a point light; k_d is the surface diffuse reflectivity; N is the normal of a point on the surface; L is the unit vector from the point to the light source. For specular reflection, the intensity of reflected light can be derived from the specular reflectivity, the incident light intensity, and a function called the specular term. The value of this function is calculated from the view vector, the reflectance vector, and several parameters with empirical values.

$$I = I_{amb} + I_{diff} + k_s I_s R_s$$

Phong[57] uses a shininess index to indicate the radius and gradient of reflectance intensity. Blinn[11] propose a modified Phong model in 1977. Blinn-Phong model replaces the view vector (V) and reflection vector (R) with normal vector and half vector of the incident ray (L) and the view vector (V), to achieve faster performance and softer light effect. In 1981, Cook and Torrance developed a specular light model[23], which considers surface material. Cook-Torrance model considers rough surfaces composed of plenty of micro-facets that are perfect specular reflectors. The specular term of this model is calculated from the proportion of the reflected light intensity F (Fresnel Reflect Term), distribution of the micro surfaces D (Normal Distribution Factor), and intensity decay caused by facets shadowing G (Geometric Shadowing Term).

$$f_r(v, l) = \frac{D(H, a)G(V, L, a)F(V, H, f0)}{4(N \cdot V)(N \cdot L)}$$

In Equation 2.2.4, a is the surface roughness, while f0 represents the specular reflectance at normal incidence.

Photorealistic Lighting

The concept of Photorealistic rendering (PR) was proposed in late 1980s. The principle is to add more details and flaws to geometry, texture, and lighting effects to make Scenes looks as real as possible. For light effects, photorealism means to simulate indirect and anisotropic light effects. Models simulates indirect illuminations are called ”Global Illumination”. Illumination models above are called ”local illumination”, as only direct interaction between light sources and surfaces are considered. Global illumination talks more about interactions among surfaces, such as shadows, reflections, refractions, and light inter-reflections A equation for global illumination model was proposed by Kajia[33] in 1986.

$$L_o(x, w_o) = L_e(x, w_o) + \int_{\Omega} f_r((x, w_i, w_o)L_i(x, w_i)(n \cdot w_i)dw_i$$

Where x is a specific point on a surface; w_o is the reflection direction at x; w_i is the incident direction at x. The ”output” (reflected) light intensity $L_o(x, w_o)$ equals to the ”output” emission energy (equals to 0 if the surface is not self-luminous) plus the incident energy, which is determined by the incident light intensity, the possibility light reflected towards w_o, and the incident angle. As ambient light and diffuse light are view-independent, they can be rendered to texture before the other light effects rendered.

$$L_o(x, w_o) = L_e(x, w_o) + L_{diff} + \int_{\Omega} f_{rs}((x, w_i, w_o)L_i(x, w_i)(n \cdot w_i)dw_i$$

$f_r((x, w_i, w_o)$ is used to determine the reflectance pattern of anisotropic surfaces, which can be estimated by BRDF (Bidirectional Reflectance Distribution Function). This model is a function of 4 real variables (w_i and w_o each has two degree of freedoms) that defines the percentage the light reflected to each direction at a point of an opaque surface. Parameters

of BDRFs are acquired by taking millions of reflectance samples, which is expensive. Therefore, most of the modern game engines provides tabulated BRDFs parameters, so that the developers can render anisotropic light effects in real-time.

For incident light intensity estimation, the values can be estimated from functions like the ones listed above or from sampling light transport trajectory. Comparing to the light model functions, the sampling model is usually more realistic but more resource-consuming. Radiosity[37] samples the energy which leaves a surface and strikes another surface. Photon mapping[41] scatters sample photon from the light sources to the scene and records energy changing every time a surface struck. The most widely used category of methods is called ray-tracing. Ray-tracing takes samples by shooting rays from the eye to each pixel and collect illumination information. NVIDIA OptiX and Microsoft DirectX have already incorporated GPU-accelerated ray-tracing in their API. The original ray-tracing model was proposed for mirror effect rendering[83], while then extend to indirect specular light simulation. The model first emits rays from the camera to each pixel. The reflected rays are traced for several times (usually less than 3) and finally trace the ray to the light and calculate the accumulated result. As the original ray-tracing model only works for specular reflection, path tracing was proposed for both specular and diffuse light tracing. The naive path tracing emits several rays to each pixel at random. Naive path tracing is expensive as more than 1000 samples per pixel are required to ensure most of the pixels are filled. Bidirectional path tracing[37] reduces the sampling size by tracing rays originate from both the eye and the light sources. Metropolis light transport further improves the performance by estimating the direction of a reflected ray from the corresponding incident ray.

Non-Photorealistic Lighting

Unlike Photorealistic Rendering, Non-Photorealistic Rendering (NPR) aims to apply artistic effects on the 3D scenes instead of simulating reality. Therefore, an NPR scene attaches more importance to feature enhancement and abstraction than detailed visualization.

Edge enhancement is a type of method rendering 3D models in styles that stress the sharp geometrical features and discontinuities. Saito and Takahashi[61] propose a method extracting discontinuity and contour maps by applying 3*3 filters to the scene's geometrical features. Afterward, research has been proposed to depict scenes with a set of lines, such as suggestive contour[25], abstracted shading[47], demarcating curves[45], and Laplacian lines[90].

Another type of NPR rendering method is to depict shape with shading. Gooch et al. [32] reproduced lighting effects based on ambient occlusion by mapping a cold-warm color scale to the degree of exposure to ambient light. Eye-Dome Lighting (EDL) was first proposed by Christian Boucheny in his Ph.D. thesis[14] to improve depth perception in the visualization of massive 3D datasets. Similar to Crytek Screen-Space Ambient Occlusion (SSAO)[7], for a pixel p, a neighbor pixel will reduce the lighting at p if it is closer to the viewer than p.

The third type of method is to improve color contrast based on surface curvature. Mean-curvature shading technique[44] maps a bright-dark scale to surface concavities and convexities, respectively. Toler-Franklin et al. [78] make the mean-curvature method multi-scale and thus improve the performance. Vergne et al. [79] leverage radiance scaling to enhance details based on both surface curvature and material characteristics. Zhao et al. [91] smoothen the contrast changing pattern through gradient enhancement. Cignoni et al. [19] exaggerate the high-frequency component of the surface by modifying the normals. Afterward, normals are used to enhance or simplify curvature features for different shading models[3].

Shadow

The primary tasks for shadow rendering are (1) acquiring the correct coverage of shadows and (2) making the shadows seem realistic. There are several methods to achieve the first objective, such as shadow mapping, shadow volume, and silhouette based techniques, such as smoothies and penumbra wedges. As the precision of silhouette detection is unstable, shadow mapping and shadow volume turn out to be the most popular methods nowadays. Shadow volume can generate smooth shadows without filters, and thus widely used in mobile terminals. While shadow mapping can be combined with different types of filters to generate more photorealistic shadows. In shadow mapping, shadowing effects are calculated in two passes for each light source.[29] The first pass renders the depth map from the viewpoint of the light source to a texture called a shadow map. The second pass is to acquire the scene of the actual viewpoint. A fragment is shadowed if its distance to the light source is greater than the corresponding depth map element. Shadow volume leverages stencil buffer to mask out unshadowed fragments. This method first finds the volume sheltered from the light source, and then emits rays from the viewpoint to the shadow volume and check if the rays hit any fragments inside the volume.

Penumbra indicates the shadow effects caused by partially blocked light. The major objective of shadow filters is to fade shadow effects smoothly. PCF (Percentage Closer Filtering), VSM (Variance Shadow Maps), CSM (Convolution Shadow Maps), ESM (Exponential Shadow Maps) generates fixed-size penumbra. PCSS (Percentage Closer Soft Shadows) is used to generate variable-size penumbra. PCF sample the shadow test results of several pixels around the projected point, and then determines the shadow intensity by the number of pixels passing the test. VSM[46] considers the depth values within the filter kernel as a depth distribution. Depth values greater than a threshold are extracted, which are then used for shadow intensity calculation. ESM[62] makes the shadow test by checking the value of $exp(k*z)*exp(-k*d)$

instead of $d - z$. CSM considers the light intensity a 1D Fourier expansion of a function of light-fragment distance and the corresponding depth value.[6] PCSS[30] is a method to determine the shadowed area and needs to be combined with a filter to smooth the edge of shadows. For each projection point, PCSS samples and values of the shadow map within a searching area and get the average depth of them. Assuming a blocker is placed at the average depth, the penumbra width can be estimated with a similar triangles method. Finally, the penumbra will be blurred by PCF/VSM/CSM/ESM.

2.3 Previous Works in Point Cloud Rendering

Researching state-of-the-art point cloud visualization tools helps in designing the architectures and functionalities of our own tools. Richter and Döllner[58] have developed an out-of-core point cloud visualization platform that enables real-time visualization of point cloud with 5 billion points. The point cloud data is split and organized in an Octree file structure during pre-processing. For each non-leaf node, a similar principle of QSplat is employed for visualization, which means that bounding spheres of nodes are rendered rather than points from the original point cloud. According to the authors, the pre-processing took 14 hours for 73GB point cloud data to build the hierarchical file structure. At the beginning, only several high-level nodes are loaded to the memory. As the user surveys the point cloud, nodes are added or removed according to the view frustum's variation. To provide fine detail in the visualization, the authors set 60 pixels as the projected point spacing (PPS) threshold. If a node's point is larger than the threshold on the screen and the frame rendering rate is acceptable, child nodes of this node will be loaded to the memory. With these two strategies, the system ensures that less than 4 million points rendered in a frame and thus guarantees the rendering speed of each frame.

Discher et al. [28] have produced a system that provides services in point cloud storage, processing, and rendering. In this application, Level of Detail (LoD) and subset selection are implemented with kd-tree. By rendering point id into frame buffer objects or 2D textures, the indices of points can be passed from GPU to CPU by mouse clicking events. Thus the functionality of real-time picking and single point modification is achieved. After the execution of the shaders, some post-processing effects are added to the generated images to assist users visually identifying the structure of the point cloud, such as Screen Space Ambient Occlusion[54] and Eye-Dome Lighting[15]. Both thick-client and thin-client are provided in this application, where thick-client receives all the data in one request while thin-client gets rendering result images every time the viewport updated. The application is claimed to be able to produce an interactive frame rate (>30fps) in thick-client and 60fps in thin-client with a general personal computer.

Potree employs and improves the work of Scheiblauer and Wimmer[65] for hierarchical structure construction and traversal. In the modified MNO of Potree, Passion-disk sub-sampling and Dart-throwing strategy are leveraged to solve the sample distance checking problem. A point is added to a node only if its distance to other points inside that node and adjacent nodes is larger than the spacing. However, this distance checking strategy makes the point cloud not modifiable. During traversal, view-frustum culling was first applied. Then, the nodes are sorted by their projected size on the screen and visited from nodes with large sizes to those with small sizes. The projected size is a function of the field of view, the distance to the center of the node, the node's bounding sphere radius, and the height of the screen. This process continues until (1) no node to be visited or (2) meeting the point budget (threshold of the number of points) or (3) the point's screen projection size is below a threshold.

For point splatting, Potree employs the three-pass rendering algorithm proposed by Scheiblauer et al. [64]. The three passes are of the same definition of the algorithm of Botsch et al. [12]

discussed in Section 2.2.3. The first pass is depth pass, which is to write the nearest fragments' depths into corresponding framebuffer cells. A parabolic function is used to calculate the per-fragment depth-offset. The second pass is called the attribute pass. Each framebuffer cell's attributes are calculated by a weighted sum of the attributes of fragments within the blend depth. The last pass, normalized pass, is to rescale the attribute values to a meaningful range. The point size is the spacing of a node's deepest visible child[63].

Potree employs Eye-Dome Lighting (EDL)[15] as its illumination model. In Potree's modified EDL, the decrement of a pixel's luminance is a function of logarithmic depth difference between the pixel and its top, bottom, left, and right neighbors. The further the pixel behind its neighbors, the darker it is.

Chapter 3

Method and Experiment

3.1 Introduction of X3D Elements

All X3D scene graph nodes (in XML or HTML5, elements) should be contained by a $< x3d >$ tag. In the X3DOM runtime (HTML5, Javascript, WebGL) for the X3D standard, the CSS features of this tag can be manipulated in the same way as those of $< div >$ tags. Within each $< x3d >$, a $< scene >$ tag is used to contain elements of a 3D space. In this project, all five types of elements (Figure 2.4) are used. For environment nodes, $< background >$ node is used to set background color. For lighting, one or more of $< DirectionalLight >$, $< PointLight >$, and $< SpotLight >$ can be chosen as light source. Besides, headlight is stuck with the camera and thus can be turned on/off within $< NavigationInfo >$ tag. Except for headlight, $< NavigationInfo >$ also defines the speed, visibility, and travel mode while exploring the scene. Another navigation node is $< ViewPoint >$. $< ViewPoint >$ defines the position, orientation, and view frustum properties of the camera. Multiple cameras are allowed in a scene. The users can switch between viewpoints by setting the attribute ”bind” of each viewpoint. Primitive nodes contain group nodes and shape nodes. Group nodes are used for organizing multiple shape nodes so that they can share some properties, such as bounding boxes and transform matrices. Shape nodes contain a geometry node and an appearance node. Colors, materials, textures, and shaders can all be child nodes of appearance nodes. Animation nodes usually cooperate with each other. For example, $< TimeSensor >$ is used to set up time slots of the keyframes of the animation, while $< PositionInterpolator >$ and $< OrientationInterpolator >$ define spacial variations of each keyframe. $< route >$ is to set the connections between Sensors and Interpolators. To persist the experiment results, JavaScript and HTML elements are used to control animations and store the FPS. X3DOM runtime provides access to information of the scene, such as frame rate, drawcall, and the number of vertices. Attribute values of nodes can be acquired and modified by *setAttribute*() and *getAttribute*() methods. In the following subsections,

details are discussed about (1) X3DOM lighting models, (2) BinaryGeometry node, (3) animation nodes, (4) X3DOM runtime, and (5) our implementation of $PointProperties$ shader.

3.1.1 Lighting and Shadow

X3D provides four types of light nodes: directional light, point light, spotlight, and headlight. Directional light illuminates along with rays parallel to a given 3D vector. Point light evenly emits rays from a 3D point to all the directions. Spotlight shades light from a single point to a direction within a solid angle. Headlight is a light sticking with the camera. Among these lights, headlight doesn't have its own settings. Therefore, only the first three light types are discussed in this section. Light parameters available for shader include "On", "Type", "Location", "Direction", "Color", "Attenuation", "Radius", "Intensity", "AmbientIntensity", "BeamWidth", "CutOffAngle", and "ShadowIntensity", where "Radius" is for point light and "BeamWidth" and "CutOffAngle" are for spotlight only. All these parameters can be acquired by $light + index + _ + parameter$ within shaders. For example, if there are both directional light and spotlight, the intensity of the directional light can be acquired through $uniformfloatlight0_intensity$ within shaders.

By default, the illumination model is a Blinn-Phong model. For more realistic lighting effects, a Physiologically Based Rendering (PBR) model, Cook-Torrance specular BRDF (Equation 2.2.4), is employed. The Normal distribution function (D) is estimated by the function proposed by Walter et al. [80] The Geometric Shadowing (G) is calculated from the equation of Heitz et al. [38]. The Fresnel Term (F) employs the function of [66]. More specifically, the method of Vergne et al. [79] is used for enhancing the appearance of subtle geometrical differences. For $< CommonSurfaceShader >$, with materials provided,

X3DOM employs BRDF for PBR rendering and Blinn/Phong/Lanbert/Constant models for general light rendering, according to the material provided.

Photorealistic shadows can be controlled by several parameters in X3DOM: ”shadowMapSize”, ”shadowFilterSize”, ”shadowCascades”, ”shadowIntensity”, ”shadowOffset”, ”shadowSplitFactor”, and ”shadowSplitOffset”. Shadow is calculated with shadow mapping method.[29] ”shadowMapSize” is used to define the shadow map’s sampling rate. ”shadowOffset” corrects the accuracy loss of depth map. To improve rendering efficiency, the shadow map is split into several sections by depth. The closer sections are of higher rendering priorities. ”shadowIntensity” decades by the distance from the light source to the fragment. Variance Shadow Map (VSM) and Exponential Shadow Map (ESM) are implemented in X3DOM.

The light source used in this research is a directional light source. The definition of the node is listed in List 3.1.

Listing 3.1: Directional light settings

```
1 <x3d>
2     <Scene>
3         ......
4           <DirectionalLight direction='-0.5 1 0.5' on ="TRUE" intensity
              ='1.0' shadowIntensity='0.5' ambientIntensity="0.5"> </
              DirectionalLight>
5         ......
6     </Scene>
7 </x3d>
```

3.1.2 Appearance Node

The X3D Appearance Node is used to describe the essential properties (such as Material and Texture) that determine a Shape's visual effects. The supported child nodes now include BlendMode node, ColorMaskMode node, DepthMode node, Material node, Shader node, Texture node, and Texture Transform node. The depth mode contains the parameters used for the depth test. The BlendMode controls the blending and alpha test. Pixels can be drawn using a function that blends the incoming (source) RGBA values with the RGBA values that are already in the frame buffer (the destination values). ColorMaskMode node defines whether R/G/B/A is rendered. Material nodes define the color attributes to be used for calculating light effects. The color attributes include ambient intensity, diffuseColor, emissiveColor, specularColor, shiness, and transparency. Texture node supports 2D textures (ImageTexture, ImageTextureAtlas, and MovieTexture), 3D textures (CubeMapTexturing/X3DEnvironmentTextureNode and Texturing3D/X3DTexture3DNode), generated textures (PixelTexture, RenderedTexture, and Shaders/SurfaceShaderTexture), and multiple textures (MultiTexture). Texture Transform nodes are used to specify transformations of texture coordinates. Shader nodes currently include CommonSurfaceShader and ComposedShader. CommonSurfaceShader enables users to specify physical materials (ambient map, diffuse map, specular map, shininess map, transparency map, and normal map) by uploading or generating texture images. ComposedShader node supports two types of shader parts (vertex shader and fragment shader) and different GPU programming languages. The shader will replace the default shader of the corresponding geometry.

Aside from the *PointProperties* node, a version using online X3DOM resources is also developed. This version leverages the $< ComposedShader >$ node and the shader generation functions provided by the X3DOM library. The generated shader can be merged with user-customized shader through JavaScript programming, which can process shader coding as

strings and assign the merged result to the innerHTML of the $< ShaderPart >$ nodes. Texture and parameters of $PointProperties$ can be shared with the shaders through child nodes of $< ComposedShader >$ called $< field >$. The structure of our $< Appearance >$ node can be found in List 3.2.

3.1.3 BinaryGeometry Node

Generally, an HTML document larger than several megabytes can lead to an unpleasant delay while being loaded. Thus, large 3D datasets are needed to be stored and transported through an external data format so that they can be loaded asynchronously on the client-side. Optimization in data size and encoding/decoding time can both contribute to shortening the loading time. X3DOM supports external nodes $< Inline >$, $< ImageGeometry >$, $< BinaryGeometry >$[10], $< PopGeometry >$[51], $< BufferGeometry >$, and $< ExternalGeometry >$. $< Inline >$ is an entire subtree of the scene graph encoded in XML/VRML. $< ImageGeometry >$ encodes vertex data (index, coordinate, color, normal, tangent, bitangent, etc.) as images and makes use of WebGL inherent $< img >$/$< video >$ compression and streaming techniques. The streamed images can be dumped to VBOs through the vertex texture fetch feature of OpenGL ES 2.0/WebGL.$< BinaryGeometry >$ sends TypedArray through an XHRResponse, which can be directly mapped to VBOs. $< PopGeometry >$ is a multi-LOD compressed 3D data format, where "POP" is the abbreviation of "Progressively Ordered Primitive". The method first generates a tree structure by computing axis-aligned bounding boxes, each of which contains a vertex at the center of itself. The vertices are then sorted by their detail levels, and thus the order data transferred conforms to the order of detail levels. $< BufferGeometry >$ is developed to load binary data like GL Transmission Format (glTF)[59], which transmits data as several separate JSON files through XmlHTTPRequest. $< ExternalGeometry >$ is a node used to contain X3D data of Shape Resource Container

(SRC) format[50]. SRC is a format similar to glTF, but additionally supports progressive transmission.

Avalon Optimizer (AOPT) is a command-line tool that is part of the Instant Reality software installation, developed to optimize and compress large 3D data, especially X3D and VRML geometries. AOPT provides functions for model optimization and large dataset partitioning. Model optimization includes removing duplicate materials and unused attributes, merging meshes with the same material, flattening scene graph, and improving batching of draw calls. Large data partitioning includes generating loose KD tree or octree index and vertex cache optimization. In this research, AOPT is used for subsampling and transforming the point cloud data in .ply to a binary geometry format.

Listing 3.2: Binary geometry node example

```
1 <x3d>
2     <Scene>
3       ......
4         <Shape>
5             <Appearance>
6                 <imagetexture url="circle_texture.png"></imagetexture>
7                 <ComposedShader>
8                   <field name="tex" type="SFInt32" value="0"></field>
9                   <field id="minPointField" name='minPoint' type='
                      SFFloat' value='1.0'></field>
10                  <field id="maxPointField" name='maxPoint' type='
                      SFFloat' value='10.0'></field>
11                  <field id="scaleField" name='scale' type='SFFloat'
                      value='10.0'></field>
12                  <field id="attField" name='att' type='MFVec3f' value
```

```
                        ='[1.0, 0.0, 0.0]'></field>
13                  <ShaderPart id='vertShader' type='VERTEX' style="
                        display:none;">
14                  ......
15                  </ShaderPart>
16                  <ShaderPart id='fragShader' type='FRAGMENT' style="
                        display:none;">
17                  ......
18                  </ShaderPart>
19              </Appearance>
20              <binaryGeometry id="bg50" DEF='BG_50'
21                              vertexCount='1146222'
22                              primType='"POINTS"'
23                              position='247.080001831 126.869995117
                                    -19.8899993896'
24                              size='563.619995117 505.299987793
                                    84.7600021362'
25                              coord='aopt50BG_0_coordBinary.bin+8'
26                              color='aopt50BG_0_colorBinary.bin+4'
27                              coordType='Int16' colorType='Uint8'>
28              </binaryGeometry>
29          </Shape>
30          ......
31      </Scene>
32  </x3d>
```

3.1.4 Animation

Animations in X3D can be set up with follower nodes plus route nodes or time-sensor nodes plus interpolator nodes plus route nodes. Both follower node and interpolator node can create animations on the values of different attributes, such as color, position, orientation, normal, and scale. Follower nodes include Chaser nodes and Damper nodes. Chaser nodes transmit to a destination value whenever the destination value is updated. Damper nodes also perform transitions based on the update of destination values. The difference between Chaser and Damper is that the transition speed of damper is calculated based on the speed of the former transition and the value of "order" field, "tau" field, and "tolerance" field.

Time-sensor nodes control the start/pause/stop/loop/duration of animations. The $fraction_changed$ field of time-sensor node output a floating-point value in the closed interval $[0, 1]$. The output value is set to be 0 before $startTime$, and then gradually increases to 1 within $cycleInterval$ seconds. The field $set_fraction$ of the interpolator node is used to synchronize the attribute changes to time lapse, denoted by $fraction_changed$ filed of the timeSensor node. Interpolator nodes interpolate among a list of key values to produce a smooth $value_changed$ output event, which will gradually update the target field of a specific node. The key field lists the key time slot rescaled to $[0, 1]$, while $keyValue$ denotes differences between the value of each time slot to the initial value. Finally, route nodes are used to connect $fraction_changed$, $set_fraction$, and $value_changed$ and field values of nodes that practices the animation. In our research, an animation is set by timeSensor, interpolator, and route node to make the camera traverse through the point cloud. The animation is triggered by a button-clicking event and ended while the first cycle is finished. The code is listed in List 3.3.

Listing 3.3: Animation applied to the view point

```
1 <x3d>
```

```
2     <Scene>
3         ......
4         <transform DEF="vp0Trans" id="vp0Trans">
5           <Viewpoint DEF="vp0" id='vp0' position="387.5 357.91 20.45"
6                      orientation="0.36959 0.57356 0.73105 2.35696"
                          description="perspective"></Viewpoint>
7         </transform>
8         ......
9       <!-- Animation -->
10      <timeSensor DEF="time" id="time" cycleInterval="10" loop="false"
          enabled="false"></timeSensor>
11      <PositionInterpolator DEF="trans" id="trans" key="0 1" keyValue="
          0 0 0  -352.4 -454.98 -9.74"></PositionInterpolator>
12      <OrientationInterpolator DEF="rotate" id="rotate" key="0 1"
          keyValue="0 0 1 0  0 0 1 0"></OrientationInterpolator>
13
14      <Route id="rt1" fromNode="time" fromField ="fraction_changed"
          toNode="trans" toField="set_fraction"></Route>
15      <Route id="rt2" fromNode="trans" fromField ="value_changed"
          toNode="vp0Trans" toField="translation"></Route>
16      <Route id="rr1" fromNode="time" fromField ="fraction_changed"
          toNode="rotate" toField="set_fraction"></Route>
17      <Route id="rr2" fromNode="rotate" fromField ="value_changed"
          toNode="vp0Trans" toField="rotation"></Route>
18      ......
19    </Scene>
20 </x3d>
```

3.1.5 X3DOM Runtime and Interaction

X3DOM runtime is an API that provides live access to system through ECMAScript. In this application runtime proxy object attached to each $< x3d >$ element, and thus can be used to check the current status of any x3d objects. For example, $x3dom.runtime.ready()$ indicates if a node is entirely loaded. $x3dom.getViewingRay()$ points out the direction of the camera. X3DOM Runtime can also be used to acquire state information listed in the state panel of X3DOM (Figure 3.1). In this research, $runtime.exitFrame$ is used to determine if the frame is fully rendered, which is the correct time to record FPS (see List 3.4).

Listing 3.4: Acquire FPS from X3DOM runtime API

```
1        x3dom.runtime.ready = function () {
2            ......
3            const x3dElement = document.getElementById('x3dElement');
                // get <x3d> node
4            const runtime = x3dElement.runtime;
5
6            runtime.exitFrame = function () {
7                fpsDiv.innerHTML = runtime.fps.toFixed(2);
8                drawcalls.innerHTML = runtime.states.infos['#DRAWS: '].
                    toFixed(2);
9            }
10           ......
11       }
```

Interactions to 3D objects can be achieved through JavaScript event listeners and X3DOM sensor nodes (CylinderSensor, PlaneSensor, SphereSensor, and TouchSensor). CylinderSensor turning mouse dragging or key pressing events into a rotation along a specific axis.

PlaneSensor handles transforming events, while SphereSensor enables rotations in all the directions. TouchSensor node is implemented with a single-pass render-buffer approach[74]. This approach renders one or more properties (e.g., vertex ID, fragment color, depth, and normals) of a small region around the mouse pointer (2 by 2 pixels by default) to the depth buffer, and then acquire the 4-pixel values with $gl.readPixels()$ method. The cross product of the 4-pixel values is then calculated to produce the output result.

FPS	2.98
ANIM	0.00
TRAVERSE	0.38
SORT	0.00
SHADOW	0.50
RENDER	3.64
DRAW	3.64
PICKING	339.25
#NODES:	8
#SHAPES:	1
#DRAWS:	1
#POINTS:	1,146,222
#TRIS:	382,074
#ACTIVE	0
#TOTAL	3
#LOADED	3
#FAILED	0

Figure 3.1: State panel of X3DOM scenes

3.2 Node design

PointProperties node is defined as an X3DAppearanceChildNode. To *PointProperties* to the X3DOM schema, the node should be consistent with both the architecture and existing functionalities of the X3DOM project. There are three steps to register the *PointProperties* node to the project. The first step is to create a new node type called "*PointProperties*", which defines the corresponding DOM node. The new node should at least contain fields described in Table 3.1. Other fields may also be added to implement additional functionalities. The new node type should also be registered in the appearance node as a field, so that < *PointProperties* > node can be used as a child node of < *Appearance* > node. The second part is to set up the connections between the *PointProperties* field values and the shader through WebGL. Finally, GLSL codes relevant to *PointProperties* node are added to the shaders.

Current point-based geometry nodes in X3DOM are only used to represent particles, while they can also be used to represent 3D objects with point splatting techniques. Therefore, except for the X3D 4.0 specified functionalities, the shader of *PointProperties* should also be able to (1) handle internal and external geometry types, (2) properly blend with the effects of lighting and shadowing, and (3) react to GPU-based interaction events such as picking. Aside of PointSet node, X3DOM also supports external nodes < *ImageGeometry* >, < *BinaryGeometry* >[10], < *PopGeometry* >[51], < *BufferGeometry* >, and < *ExternalGeometry* >[50]. Attributes of different types of geometry are mapped to different attribute and uniform parameters of the shader, and thus need to be deal with separately. For example, the bounding box, index, and texture size of BinaryGeometry are defined by variables of names starting with "bg", while corresponding attributes are named as "IG_xxx" for ImageGeometry. Attributes of PopGeometry and geometry with ClipPlane also are mapped to different shader parameters.

Proper fragment-wise normal and occlusion calculation models are required to leverage X3DOM's lighting models (Blinn-Phong[11] and Cook-Torrance specular BRDF[23]) with different materials and types of light sources. Besides, multiple textures and shader switching should also be supported to deal with shadowing and picking (Screen-Space Ambient Occlusion (SSAO)[7], shadow mapping[29]), and GPU picking[9]).

To fully reuse the shader codes mentioned above, there are two ways to merge them to *PointProperties* shader code. The first method is to add the pieces of *PointProperties* related GLSL code to the dynamically generated shader of X3DOM, just like what is done by ParticleSet node. This dynamic shader is the default shader that applied to all types of geometries, whose content is generated according to the parameters and properties provided. The second method is to create a shader node for *PointProperties*, and then merge the GLSL code with shader code pieces (e.g. geometry loading and lighting effects) generated by X3DOM pre-defined shader parts. This research prefers the second method based on the following reasons. The major reason is that there might be many more lines of GLSL code than ParticleSet. For example, instead of just a few lines defined point sizes, normal estimation is also an important part of lighting effects. For point cloud data without normal provided, the fragment normals can be either estimated from k-nearest neighbors (kNN) or a function of the fragment coordinates relative to corresponding vertex coordinate, just like Potree[52]. Both of the normal estimation methods would introduce a pack of codes irrelevant to most of the other shader effects.

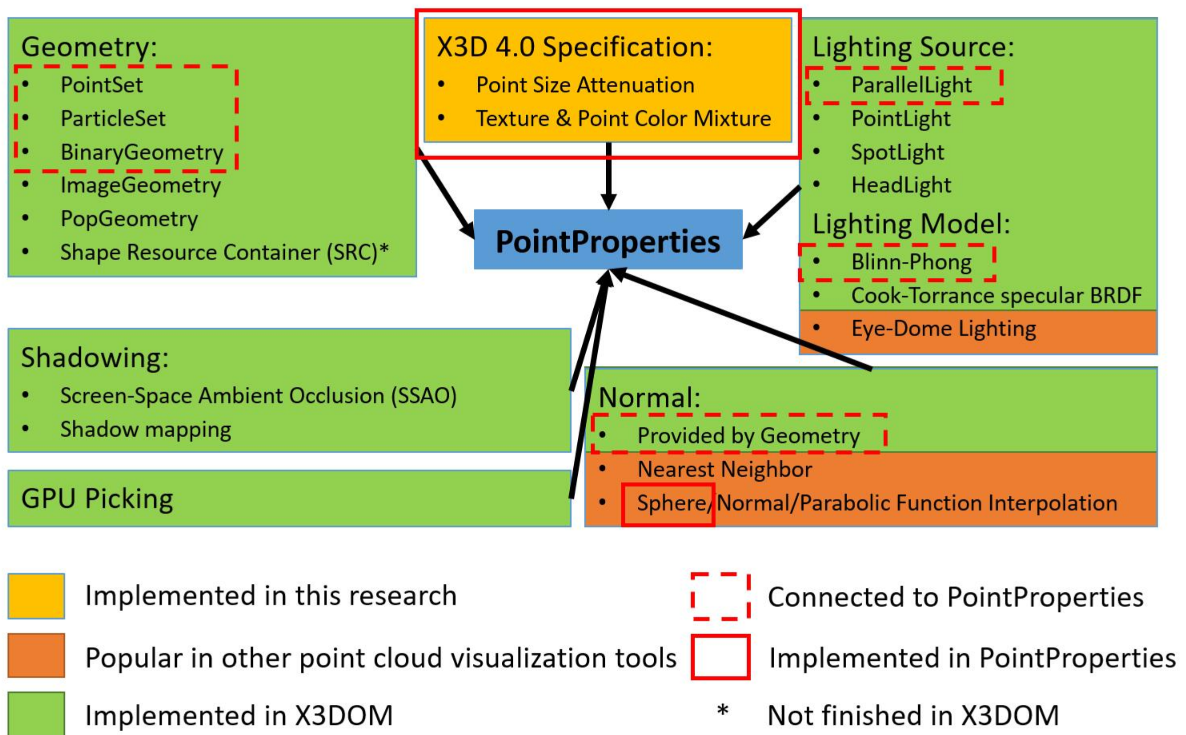


Figure 3.2: Factors to be considered in PointProperties Implementation

The work in the aspect of implementation is summarized in Figure 3.2. First of all, a class called "*PointProperties*" is created. This class works as a transmitter that feeds node attribute values to shaders. Then, a shader class is created for "*PointProperties*". The shaders in the shader class are dynamically generated through JavaScript to use different input argument sets and code segments while dealing with different geometry, lighting, and appearance nodes. Finally, the functions listed in X3D 4.0 Specification are implemented. Details of the attributes and equations are shown in Table 3.1. According to the official document of "*PointProperties*" (https://www.web3d.org/specifications/X3dSchemaDocumentation4.0/x3d-4.0_PointProperties.html), there are 6 attributes in this shader, 5 of which may affect the appearance of points. The actual point size can be calculated by Equation 3.2. In the equation, a, b, and c are the three values given by *pointSizeAttenuation*. d is the distance between the view-point and the point. For $colorMode$, $TEXTURE_AND_POINT_COLOR$ means calculating the color of each fragment of a point by multiplying the color of the texture and the color assigned to the point in the PointSet object.

$$pointSize = min(max(\frac{pointSizeScaleFactor}{a + b * d + c * d^2}, pointSizeMinValue), pointSizeMaxValue)$$

Table 3.1: Official definition of attributes in PointProperties

Name	Type	Mode	Default	Domain
pointSizeScaleFactor	SFFloat	[in,out]	1	$[1, \infty)$
pointSizeMinValue	SFFloat	[in,out]	1	$[0, \infty)$
pointSizeMaxValue	SFFloat	[in,out]	1	$[0, \infty)$
pointSizeAttenuation	MFFloat	[in,out]	1 0 0	$[1, \infty)$
colorMode	SFString	[in,out]	"TEXTURE_AND _POINT_COLOR"	["POINT_COLOR"| "TEXTURE_COLOR"| "TEXTURE_AND _POINT_COLOR"]
metadata	SFNode	[in,out]	NULL	[X3DMetadataObject]

3.3 Experiment

In this experiment, the major purpose is to estimate how different attributes and shading effects influence the scene's rendering speed. We created a 3D scene with the X3DOM nodes that mentioned in Section 3.1. We also provide control panels with sliders and buttons for parameter modification and status monitoring. The evaluated parameters include different *PointProperties* parameter values (point size scale, min/max point size, point size attenuation), texture sizes, number of points, and whether to use lighting effects. To force the scene to be updated, the attributes are examined against FPS values during travel through the point cloud. A simple backend is developed with Python and Flask to record the experiment data into files for further analysis.

3.3.1 Data and Device

Our point cloud data covers a segment of the Catawba Creek in Virginia with its surrounding area, which area is mostly covered by farmland with a section of Catawba Valley Drive and some woods by roads or the creek. The data were collected with a LiDAR mapping system equipped on a VAPOR 35 Unmanned Aerial Vehicle (UAV). The LiDAR mapping system includes a multi-echo LiDAR sensor, GNSS RTK+PPK receiver, bi-frequency L1/L2, calibrated IMU, embedded computer, two integrated batteries, power and data cables, and data pre-processing software to produce geo-referenced point clouds. YellowScan employs near-infrared (NIR) with a wavelength of 905nm for ranging. Different scans were finely aligned with the GPS measurements in QGIS and then offered to us.

Virginia Tech's Catawba Sustainability Center operates from this property, which experiments, demonstrates, and educates about small-farm management and diversification, including crops, livestock, and water management. Researchers in the Stream Lab have flown

drones over this property once a season over the last four years. Wether from photogrammetry or Lidar, the datasets are large: dense and covering a large extent.

The experiment is run on a 4-year-old Apple laptop with a 64-bit operating system installed. The CPU is Intel(R) Core(TM) i7-5557U (3.10GHz) with 16.0GB RAM. The GPU is Intel(R) Iris(TM) Graphics 6100 with 1.00GB RAM. The screen resolution is $2560 \times 1600 \times 59 hertz$.

3.3.2 Shader Implementation and Parameter Evaluation

Shader Implementation

In vertex shader, the vertex position, size, color, and eye vector are calculated to be passed to fragment shader. The vertex data acquired from binary geometry is different from the vertex data of general geometry nodes, as the data are compressed by normalizing and rescaling float to integer. Therefore, the values need to be scale back to the original values with the codes in List 3.5.

Listing 3.5: Binary geometry data processing

```
1    vec3 vertPosition = bgCenter + bgSize * position.xyz /
        bgPrecisionMax;
2    gl_Position = modelViewProjectionMatrix * vec4(vertPosition, 1.0);
3    col = color / bgPrecisionColMax;
```

The ”w” component of the view-space position is set to be the same as ”z” component by the second line of List 3.5. Therefore, the depth value equals to gl_Position.w. The attenuated point sizes can be calculated by List 3.6.

Listing 3.6: Point Size Calculation

```
1    float depth = gl_Position.w;
2    gl_PointSize = clamp(scale*1.0/(att.x+att.y*depth+att.z*depth*depth
         ), minPoint, maxPoint);
```

In fragment shader, each point has its own local screen space coordinate system, where $(0.5, 0.5)$ is at the vertex position and $(gl_PointSize/2, gl_PointSize/2)$ is scaled to $(1, 1)$. In our implementation, the point splat is supposed to reflect light like a half-sphere facing the camera. The fragment normals can be calculated by considering each fragment as a point on a unit sphere centered at the corresponding vertex. The range of the local screen space coordinate system is first scaled from [0, 1] to [-1, 1]. The x, y coordinates in the local screen space are the x, y coordinates in the sphere space. The z coordinate of each fragment can then be calculated by $\sqrt{1 - (x^2 + y^2)}$. The exact code for fragment calculation is in List 3.7.

Listing 3.7: Fragment Normal Calculation

```
1    vec4 color = vec4(col, 1.0);
2    vec2 cxy = gl_PointCoord * 2.0 - 1.0;
3    float r = dot(cxy, cxy);
4    float z = sqrt(1.0 - cxy.x*cxy.x - cxy.y*cxy.y);
5    vec3 N = normalize(vec3(cxy, z));
```

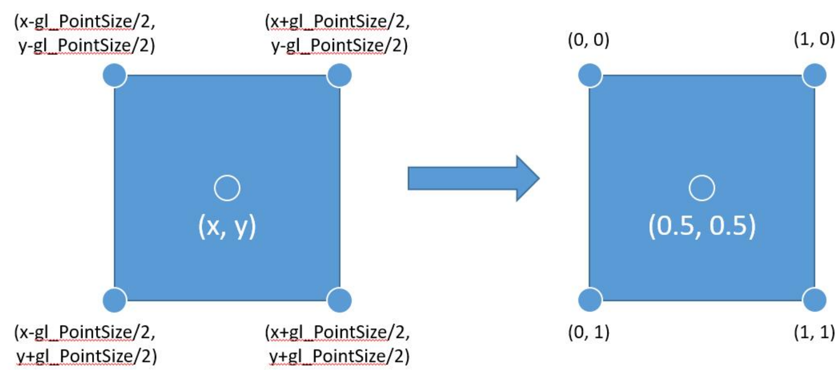
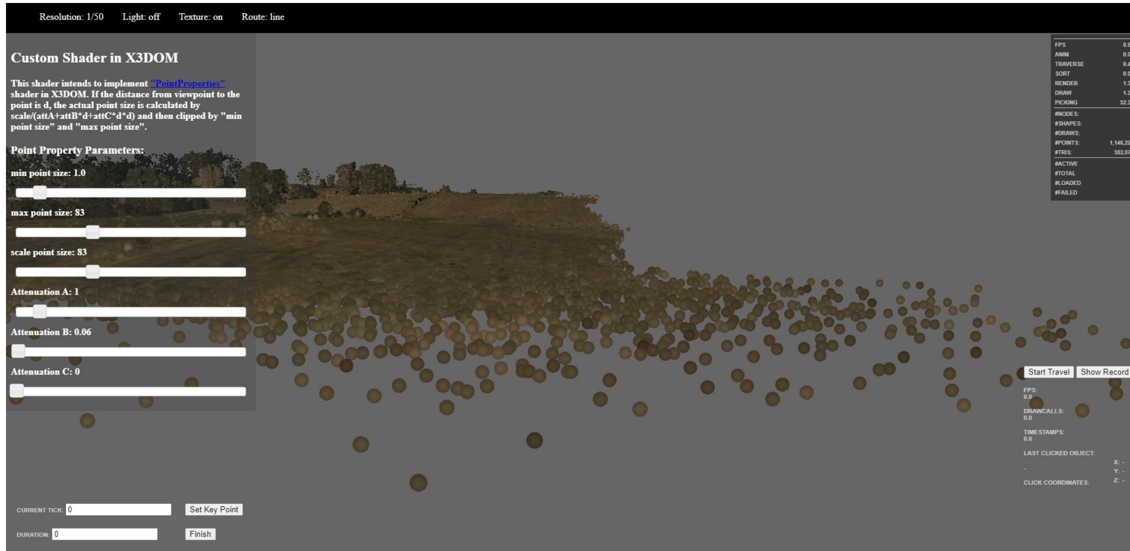


Figure 3.3: Screen space coordinate mapping to local screen space coordinate system of a point

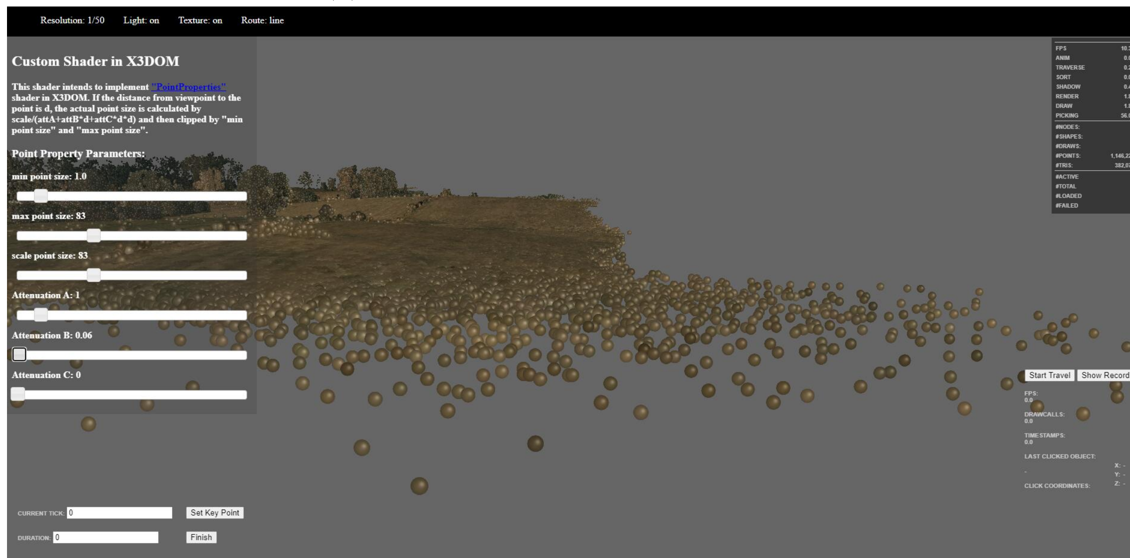
If a texture is used, the texture color should be mixed with the point color. According to the X3D specification, the mixed color should be acquired by adding up the texture color and the point color. However, the addition usually leads to excessive white color. Therefore, in our implementation, the mixed color equals to texture color times point color. Point splats are rendered as rectangles by default. To generate circular splat, fragments further than 0.5 to (0.5, 0.5) should be discarded. Besides, if the color contains alpha channel, fragments with alpha value larger than 0.5 should also be discarded, as suggested by the X3DOM shader codes.

Finally, the light effects are added to the final fragment color. $x3dom.shader.light()$ is employed to perform the most basic Blinn-Phong lighting model. This function first appends the attributes of all the light sources. Then, it adds a function called lighting() to the shader. Accumulation of lighting effects can be achieved by repeatedly calling lighting() with different light attributes. The comparison of points with and without lighting effects is shown in Figure 3.4.

Aside from binary geometry, the shader is also examined on $< PointSet >$ geometry nodes.

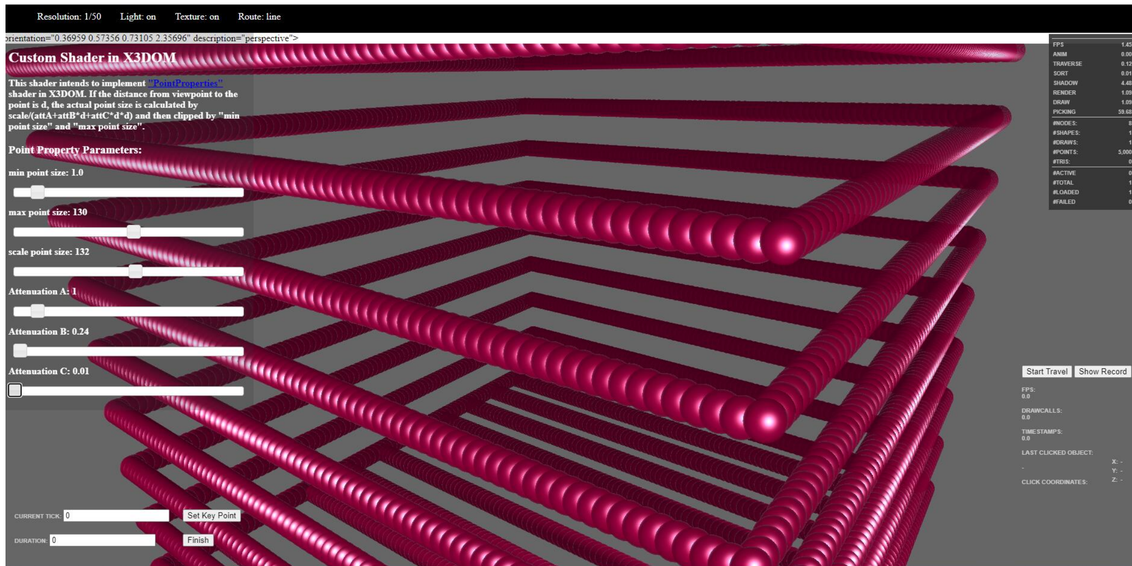


(a) Point cloud without lighting effects

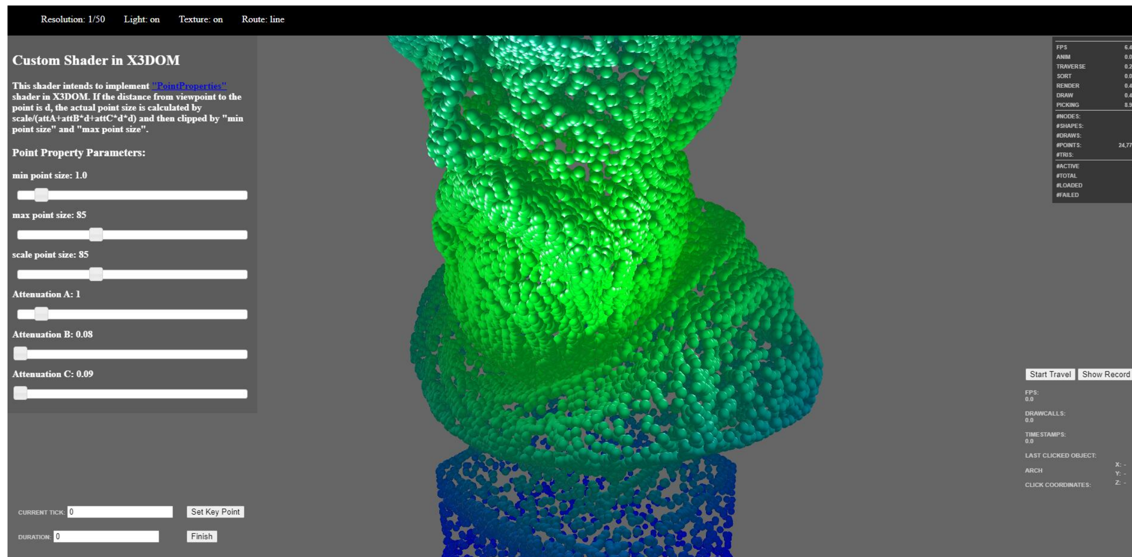


(b) Point cloud with lighting effects

Figure 3.4: Effects of light effect in point cloud rendering



(a) PointProperties on Point5000.x3d (https://www.web3d.org/x3d/content/examples/ConformanceNist/Geometry/PointSet/points5000.x3d)



(b) PointProperties on ArchimedesPointProperties.x3d (https://www.web3d.org/x3d/content/examples/Basic/Points/ArchimedesPointProperties.x3d)

Figure 3.5: Effects of PointProperties on PointSet

Parameter Evaluation

X3DOM is optimized to refresh the frame only if the field of view is changed to conserve GPU resources. Therefore, to evaluate how the values of $PointProperties$ attributes affect FPS, an animation is created to make the camera travel through the point cloud within 10 seconds. During the experiment results, it is found that the variation of trajectories of the camera does not influence the FPS measurements. The track is thus set to be a diagonal line of the bounding box of the point cloud. For each 10-second animation, the FPS value is recorded once per second. For each set of parameters, 100 rounds of the animation are performed. Therefore, 1000 samples of FPS for each set of parameters are recorded.

Four primary factors affect the frame rendering speed: number of vertices, number of fragments, the time complexity of per-vertex/per-fragment operation, and texture size. The number of vertices can be represented by the number of points in the point cloud. Point sizes control the amount of fragments. The most time-consuming part of $PointProperties$ shader is to calculate per-fragment normals and light effects. A large texture image decreases mainly FPS by occupying too much GPU memory and thus leading to more frequent data exchange between CPU and GPU. Texture mapping can also affect FPS if the interpolation method is ”bilinear”. However, X3DOM employs the ”nearest neighbor” approach by default, which is considered to be much faster and thus has little influence on performance.

Our experiment is performed only one of the four variables, each of which represents a factor mentioned above. Considering that there are too many factors to be evaluated, for parameters related to point size, only point size scale is selected, as it controls the range of point size. Point size attenuation, minimum point size, and maximum point size are set to be fixed: $minPointSize = 1.0$, $maxPointSize = pointSizeScale$, $attenuation = [1.0, 0.0, 0.01]$. According to Potree[52]'s implementation, 50 is a reasonable maximum threshold of point

size. Therefore, point size scale ranges from 1.0 to 60.0 are tested, while extremely large point size (>= 40.0) leads to a GPU memory overflow for more than 10 million points. 1.0, 10.0, 20.0, and 30.0 are thus used for factor evaluation.

The other three factors are the number of points, texture size, and whether using lights. The experiment is performed on 4 binary geometries with different number of points (1146222, 2292445, 5731114, and 11462229). Each binary geometry consists of 2 .bin files, one of which contains the coordinate information, while the other contains color information. The raw texture of our experiment is a grayscale circular of 96×96 pixels. To estimate the limitation of our device's texture processing ability, we keep upsampling our texture until the GPU memory usage reaches 100%. Finally, 7 different texture sizes are examined (no texture and texture of 96×96, 192×192, 480×480, 960×960, 2400×2400, and 4800×4800 pixels). Discarding pixels also causes significant FPS changes, i.e., rendering the default rectangle point is faster than rendering points of other shapes. Therefore, circle shape is employed for all conditions to eliminate the FPS reduction caused by discarding fragments with alpha values less than 0.5. For simplification, the lighting effect is only measured on a directional light source with the Blinn-Phong model. In the experiment, "0" indicates that the light is off, while "1" means the light is on. A list of examined values of each parameter is shown in Table 3.2.

Table 3.2: Parameters Examined in Performance Evaluation

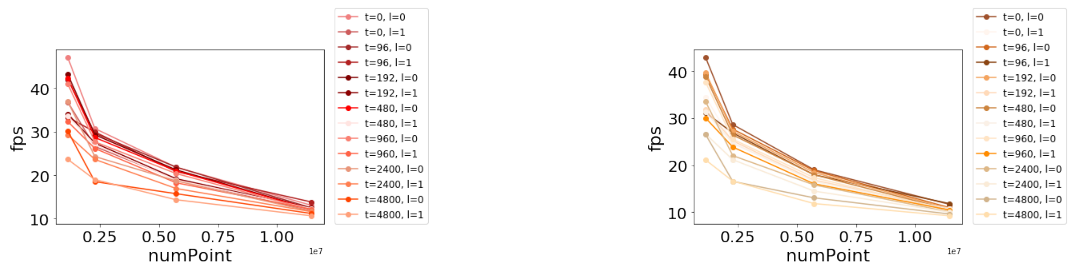
Name	Value
numPoint	1146222, 2292445, 5731114, 11462229
scale	1.0, 10.0, 20.0, 30.0
texSize (pixel/edge)	96, 192, 480, 960, 2400, 4800
light	0.0 (off), 1.0 (on)
minPointSize	1.0
maxPointSize	=scale
Attenuation	[1.0, 0.0, 0.01]

Chapter 4

Result and Discussion

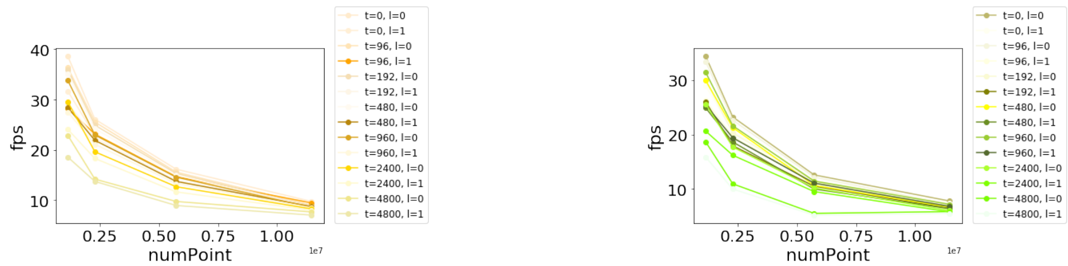
This chapter explores how FPS is affected by point size, number of points, texture size, and light effect. Basic statistical results are first presented. Then, multi-way Analysis of variance (ANOVA) is applied to normalized number of point, point size scale, texture size, and light effect on/off. The result is interpreted and explained based on the rendering workflow of GPU. Finally, pair-wise t-tests are performed on factors whose interaction effects do not pass the test of ANOVA, to further explore the characteristics of the factors.

The experiment result is evaluated with Python's "statsmodels" package. Figure 4.1, 4.2, and 4.3 show the changes of medians of FPS by individual factors. In the legend, $texSize$ means texture size; $numPoint$ is "number of point"; $scale$ means "point size scale"; $light$ indicates "light effect". According to Figure 4.1, our device is able to achieve the minimum frame rate standard ($>= 10FPS$) for less than 10 million points with scale no more than 20.0 of any texture size and light conditions. Figure 4.2 indicates that the scale should be no more than 10.0 to ensure 10 million points rendered with $FPS >= 10$. Figure 4.3 shows that the effect of texture size on FPS is not as obvious as the other two factors. The bumps in Figure 4.3 indicates that texture size is not related to FPS when it is small (less than 1000×1000 pixels). The slight effect caused by the texture size could not even suppress the effect of the instability of the performance of our GPU. Besides, texture with less than 1 million pixels generally causes little effect on rendering speed. Therefore, for our device, or devices of similar hardware settings, to guarantee an FPS over 30, it is preferred that the number of points to be rendered is less than 2 million, scale less than 20.0, and texture size less than 1-million pixels. The users should take strategies, such as hierarchical structures and simpler illumination models, to render more or larger points.



(a) Relationship between Number of Point and FPS (by Texture Size and Light, Scale = 1.0)

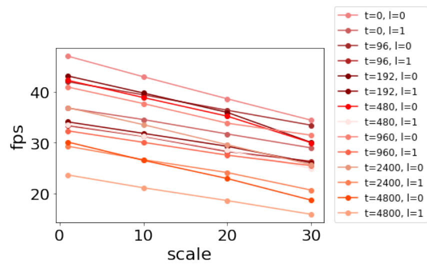
(b) Relationship between Number of Point and FPS (by Texture Size and Light, Scale = 10.0)



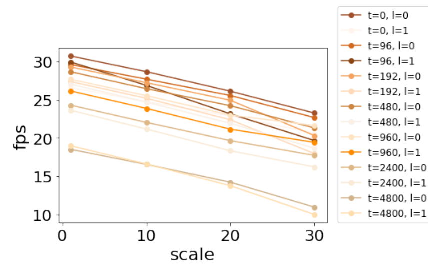
(c) Relationship between Number of Point and FPS (by Texture Size and Light, Scale = 20.0)

(d) Relationship between Number of Point and FPS (by Texture Size and Light, Scale = 30.0)

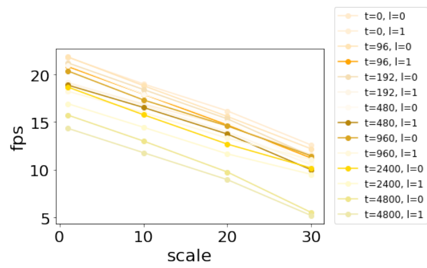
Figure 4.1: Line Plots of the Relationship between the Medians of Number of Point and FPS.



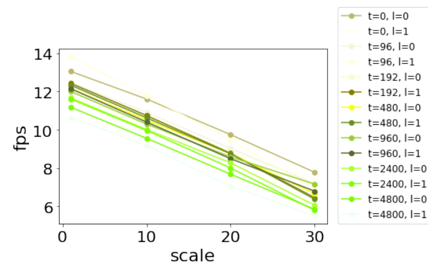
(a) Relationship between Scale and FPS (by Texture Size and Light, Number of Point = 1146222)



(b) Relationship between Scale and FPS (by Texture Size and Light, Number of Point = 2292445)

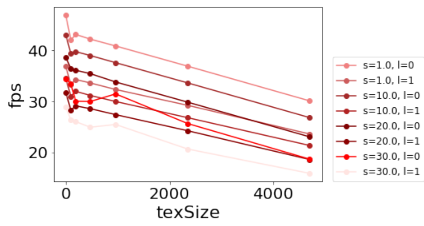


(c) Relationship between Scale and FPS (by Texture Size and Light, Number of Point = 5731114)

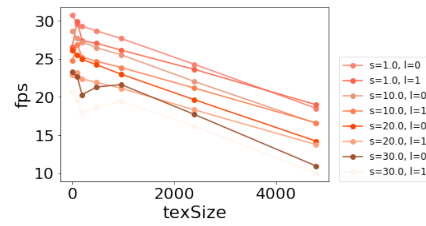


(d) Relationship between Scale and FPS (by Texture Size and Light, Number of Point = 11462229)

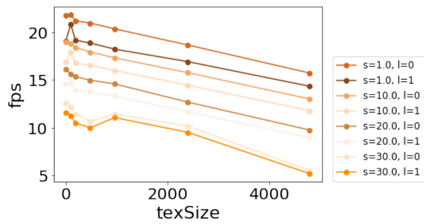
Figure 4.2: Line Plots of the Relationship between the Medians of Scale and FPS.



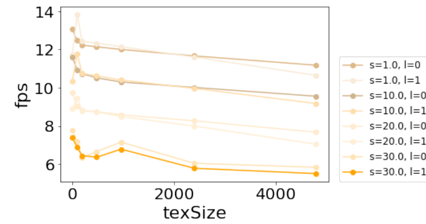
(a) Relationship between Texture Size and FPS (by Scale and Light, Number of Point = 1146222)



(b) Relationship between Texture Size and FPS (by Scale and Light, Number of Point = 2292445)



(c) Relationship between Texture Size and FPS (by Scale and Light, Number of Point = 5731114)



(d) Relationship between Texture Size and FPS (by Scale and Light, Number of Point = 11462229)

Figure 4.3: Line Plots of the Relationship between the Medians of Texture Size and FPS.

The multi-way ANOVA is performed to evaluate interactions among different attributes. Generally, ANOVA is used for testing the significance of differences among categories of sampled data. The P-value of a pair of categories is less than 0.05 if the difference between these two sample sets is unlikely to have occurred by chance[8]. Multi-way ANOVA allows the comparison among combinations of categories of multiple variables, whose nature is to compare multi-variable linear regression models built on different parts of the sampled data. In this research, the four attributes' values are sampled from continuous domains instead of discrete categories. All of the data are first scaled to $[0, 1]$ with the "min-max normalization" method. Then, to maintain the numerical meaning of the data, a linear model is built with Ordinary Least Squares (OLS). This linear model is then analyzed by the anova_ml (ANOVA linear model) tool. The anova_ml tool computes the differences between the linear model with a specific term and the linear model of a lower order than the term. The P-value of the comparison is high if the term doesn't improve the model significantly.

Ordinary least squares (OLS) model is summarized in Table 4.1. The linear model is evaluated from four perspectives. The first perspective is the descriptivity of the model. That is, the model should be close enough to most of the data points, so that it can represent the distribution of those points. R-squared and adjusted R-squared are the most common indices of the "descriptivity". Both of the R-squared and adjusted R-squared values of our OLS model are both 0.853, which indicates that the linear model is reliable. F-statistic (85161.737) and corresponding P-value (0.0000) verify the reliability of the model. The second perspective is the sensitivity of the dependent variable to an independent variable, which is represented by "condition number". A common cause of a high condition number value is multi-collinearity, which means that several independent variables are not "independent" enough, and thus produce similar effects on the dependent variable. The ideal condition number is less than 30, while our condition number of 59.8 is good enough. The third perspective is homoscedastic-

ity. Homoscedasticity, generally measured by Durbin-Watson index, indicates whether the variance grows along a particular direction. In this model, Durbin-Watson is 0.647, which is close to its ideal range ($[1, 2]$). The last perspective is the distribution of errors. Ideally, the errors should normally distributed along the linear model, while a non-normal distribution may indicate there is a better non-linear model than the current linear model. Omnibus, Prob(Omnibus), Jarque-Bera (JB), and Prob(JB) are all indicators of the distribution of errors. Omnibus and JB (102.386 and 104.882) are much higher than the ideal value (0.0) in our model. Prob(Omnibus) and Prob(JB) are almost 0.0 in our case, which means that the linear model may not be the best fit. This is understandable, as (1) number of point shows a inverse proportional relationship to FPS and (2) small texture size (less than 1000×1000 pixels) does not affect FPS much. Pair-wise t-tests are also performed on individual factors. The result shows that most of the factors always cause significant different in FPS distributions ($reject_rate_{numPoint} = 1.00$, $reject_rate_{scale} = 0.99$, $reject_rate_{light} = 0.98$), except for texture size ($reject_rate_{texSize} = 0.90$. The statistical results are consist of our observation of Figure 4.3.

Table 4.1: OLS Regression Summary

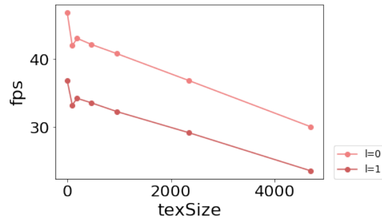
Name	Value
R-squared	0.853
Adjusted R-squared	0.853
F-statistic	85161.737
p(F-statistic)	0.0000
Condition No.	59.8
Durbin-Watson	0.647
Omnibus	102.386
Prob(Omnibus)	0.0000
Jarque-Bera(JB)	104.882
Prob(JB)	0.0000

Table 4.2: Multi-way ANOVA Summary Table

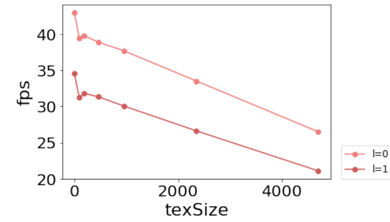
	$\text{sum}_s q$	df	F	PR(>F)
numPoint	4828.309711	1.0	965425.624661	0.000000e+00
scale	676.921340	1.0	135351.136698	0.000000e+00
texSize	513.689443	1.0	102712.746569	0.000000e+00
light	108.330062	1.0	21660.710203	0.000000e+00
numPoint:scale	21.643334	1.0	4327.607394	0.000000e+00
numPoint:texSize	155.915772	1.0	31175.523299	0.000000e+00
numPoint:light	97.916984	1.0	19578.604355	0.000000e+00
scale:texSize	0.245096	1.0	49.007106	2.557597e-12
scale:light	0.270518	1.0	54.090297	1.921434e-13
texSize:light	3.506025	1.0	701.033330	3.111288e-154
numPoint:scale:texSize	0.028479	1.0	5.694379	1.702017e-02
numPoint:scale:light	2.694283	1.0	538.724645	4.965394e-119
numPoint:texSize:light	3.313142	1.0	662.466280	7.146492e-146
scale:texSize:light	0.015219	1.0	3.043052	8.108473e-02
numPoint:scale:texSize:light	0.006629	1.0	1.325399	2.496267e-01
Residual	1103.795122	220705.0	NaN	NaN

Table 4.2 shows the significance of main effects (effects of individual variables) and their interaction effects. Considering effects of $PR(> F) < 0.05$ as significant effects, all the main effects, two-factor interaction effects, and most of the three-factor interaction effects are significant. Interaction effect of scale, texSize and light is not significant. The four-factor interaction effect is not significant either.

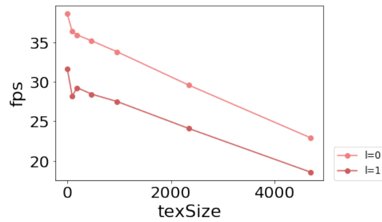
The weak interaction effects among scale, texture size, and light demonstrates that there are no non-linear relationship among these three factors (Term $scale * texSize * light$ can be decomposed to a sum of several lower order terms). To explore the meaning of this result, pair-wise ANOVA are performed on these three factors ($texSize * scale$, $texSize * light$, and $texSize * light$) grouping by number of points. The pair-wise comparisons and $PR(> F)$ values are shown in Figure 4.4, 4.5, and 4.6. It was found that the interaction effects between each pair of these three factors tends to decrease as the number of point grows. For example, for determined number of point and light condition, the interaction effect tends to be weak between texSize and scale when the number of point is large. The reason behind is the overwhelming effect the number of point and the nonlinear decrements of FPS, i.e., there are no much room for the other factors to change FPS. The statistical results are consist of our observation of Figure 4.1, 4.2, and 4.3.



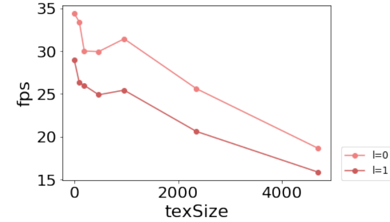
(a) Texture Size vs. Light (Number of Point = 1146222, Scale = 1.0, PR(>F) = 0.000)



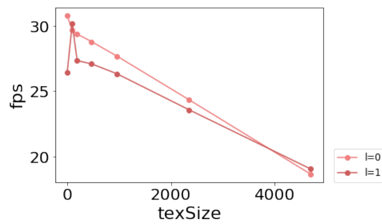
(b) Texture Size vs. Light (Number of Point = 1146222, Scale = 10.0, PR(>F) = 0.000)



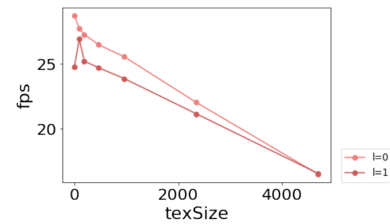
(c) Texture Size vs. Light (Number of Point = 1146222, Scale = 20.0, PR(>F) = 0.000)



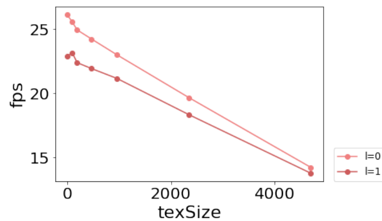
(d) Texture Size vs. Light (Number of Point = 1146222, Scale = 30.0, PR(>F) = 0.000)



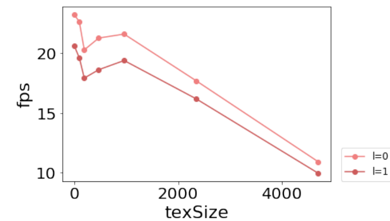
(e) Texture Size vs. Light (Number of Point = 2292445, Scale = 1.0, PR(>F) = 0.000)



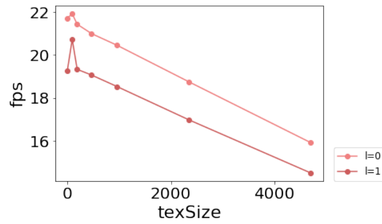
(f) Texture Size vs. Light (Number of Point = 2292445, Scale = 10.0, PR(>F) = 0.000)



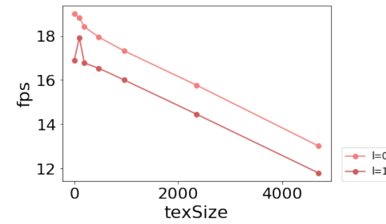
(g) Texture Size vs. Light (Number of Point = 2292445, Scale = 20.0, PR(>F) = 0.000)



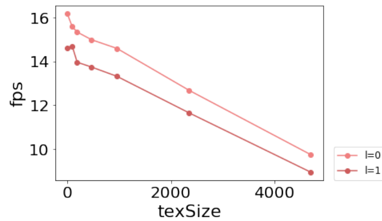
(h) Texture Size vs. Light (Number of Point = 2292445, Scale = 30.0, PR(>F) = 0.000)



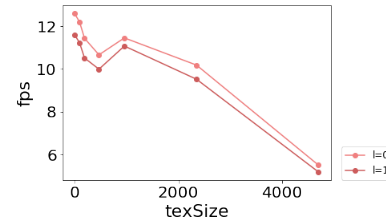
(i) Texture Size vs. Light (Number of Point = 5731114, Scale = 1.0, PR(>F) = 0.000)



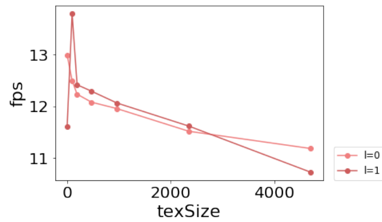
(j) Texture Size vs. Light (Number of Point = 5731114, Scale = 10.0, PR(>F) = 0.000)



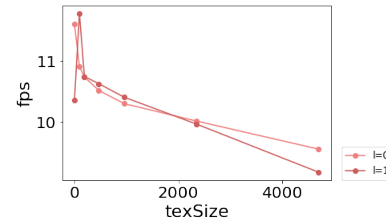
(k) Texture Size vs. Light (Number of Point = 5731114, Scale = 20.0, PR(>F) = 0.000)



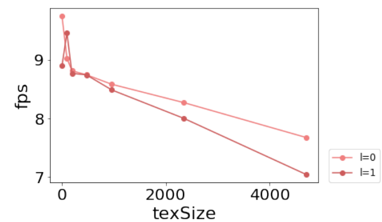
(l) Texture Size vs. Light (Number of Point = 5731114, Scale = 30.0, PR(>F) = 0.000)



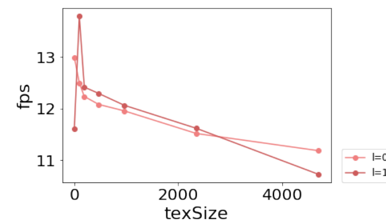
(m) Texture Size vs. Light (Number of Point = 11462229, Scale = 1.0, PR(>F) = 0.000)



(n) Texture Size vs. Light (Number of Point = 11462229, Scale = 10.0, PR(>F) = 0.000)

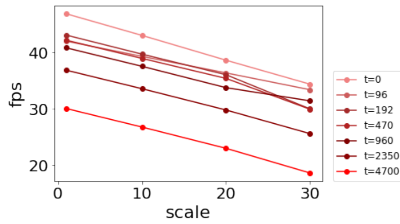


(o) Texture Size vs. Light (Number of Point = 11462229, Scale = 20.0, PR(>F) = 0.000)

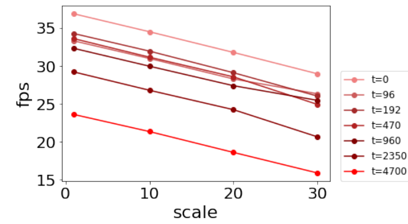


(p) Texture Size vs. Light (Number of Point = 11462229, Scale = 30.0, PR(>F) = 0.000)

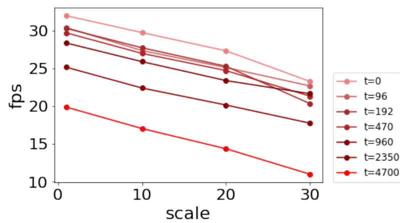
Figure 4.4: Pair-wise ANOVA Comparison of Medians of Texture Size and Light, Grouping by Number of Point and Scale.



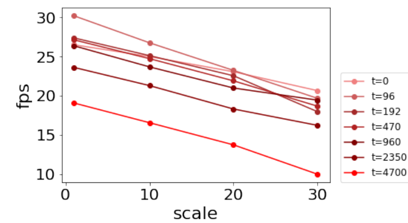
(a) Scale vs. Texture Size (Number of Point = 1146222, Light = 0.0, PR(>F) = 0.000)



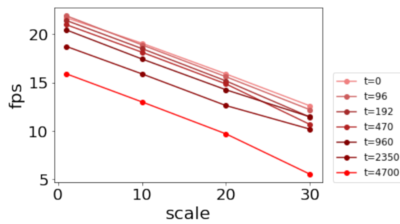
(b) Scale vs. Texture Size (Number of Point = 1146222, Light = 1.0, PR(>F) = 0.000)



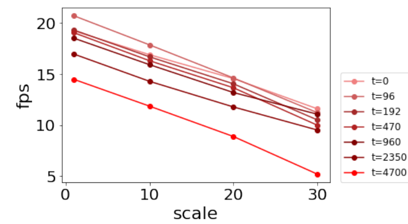
(c) Scale vs. Texture Size (Number of Point = 52292445, Light = 0.0, PR(>F) = 0.000)



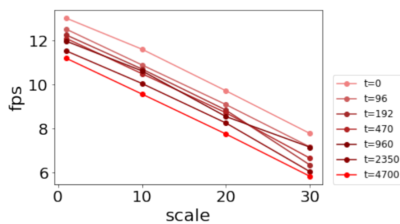
(d) Scale vs. Texture Size (Number of Point = 2292445, Light = 1.0, PR(>F) = 0.510)



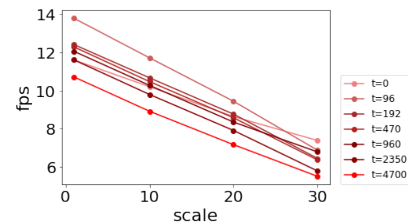
(e) Scale vs. Texture Size (Number of Point = 5731114, Light = 0.0, PR(>F) = 0.059)



(f) Scale vs. Texture Size (Number of Point = 5731114, Light = 1.0, PR(>F) = 0.084)

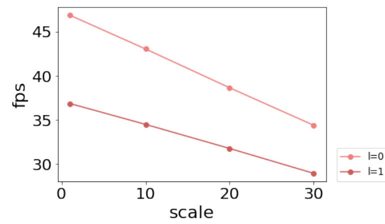


(g) Scale vs. Texture Size (Number of Point = 11462229, Light = 0.0, PR(>F) = 0.000)

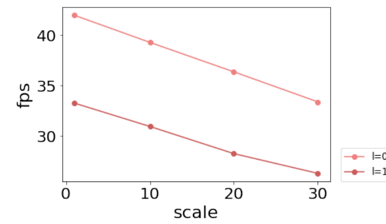


(h) Scale vs. Texture Size (Number of Point = 11462229, Light = 1.0, PR(>F) = 0.105)

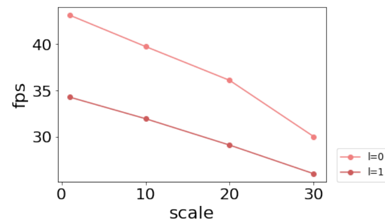
Figure 4.5: Pair-wise ANOVA Comparison of Medians of Scale and Texture Size, Grouping by Number of Point and Light.



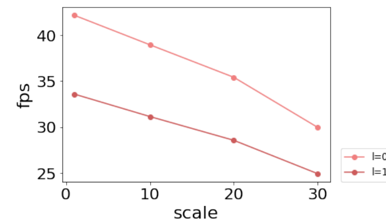
(a) Scale vs. Light (Number of Point = 1146222, texSize = 0.0, PR(>F) = 0.000)



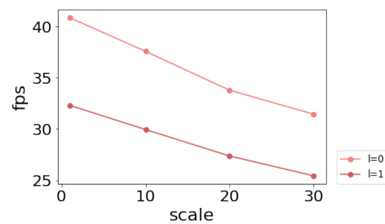
(b) Scale vs. Light (Number of Point = 1146222, texSize = 96.0, PR(>F) = 0.000)



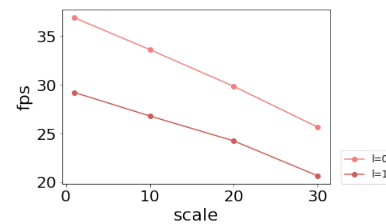
(c) Scale vs. Light (Number of Point = 1146222, texSize = 192.0, PR(>F) = 0.000)



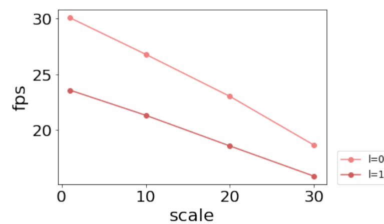
(d) Scale vs. Light (Number of Point = 1146222, texSize = 480.0, PR(>F) = 0.000)



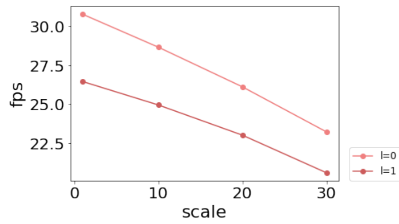
(e) Scale vs. Light (Number of Point = 1146222, texSize = 960.0, PR(>F) = 0.000)



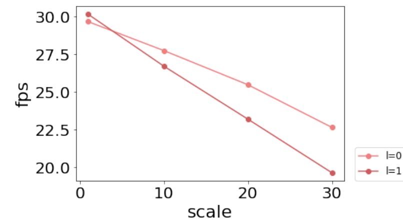
(f) Scale vs. Light (Number of Point = 1146222, texSize = 2400.0, PR(>F) = 0.000)



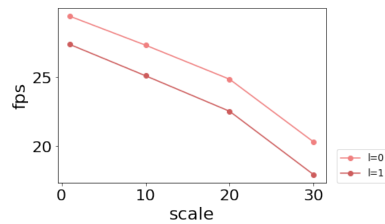
(g) Scale vs. Light (Number of Point = 1146222, texSize = 4800.0, PR(>F) = 0.000)



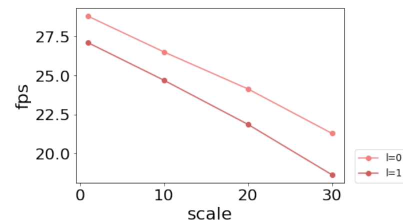
(h) Scale vs. Light (Number of Point = 2292445, texSize = 0.0, PR(>F) = 0.000)



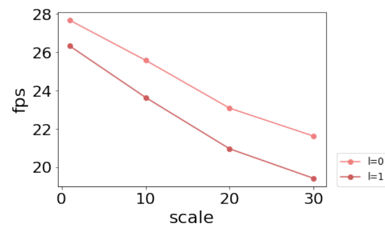
(i) Scale vs. Light (Number of Point = 2292445, texSize = 96.0, PR(>F) = 0.000)



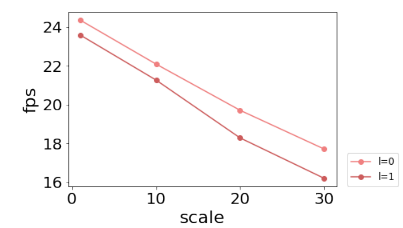
(j) Scale vs. Light (Number of Point = 2292445, texSize = 192.0, PR(>F) = 0.000)



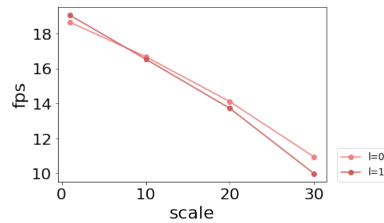
(k) Scale vs. Light (Number of Point = 2292445, texSize = 480.0, PR(>F) = 0.000)



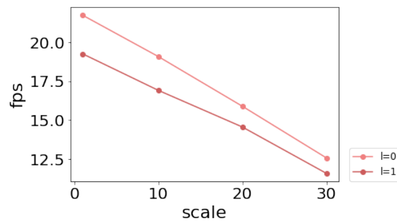
(l) Scale vs. Light (Number of Point = 2292445, texSize = 960.0, PR(>F) = 0.000)



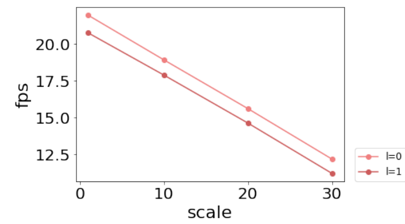
(m) Scale vs. Light (Number of Point = 2292445, texSize = 2400.0, PR(>F) = 0.000)



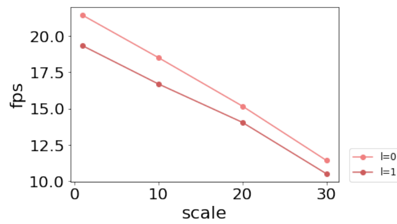
(n) Scale vs. Light (Number of Point = 2292445, texSize = 4800.0, PR(>F) = 0.000)



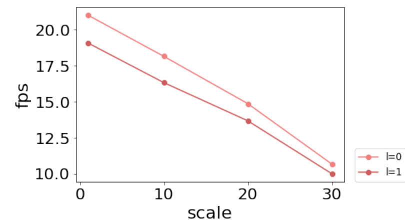
(o) Scale vs. Light (Number of Point = 5731114, texSize = 0.0, PR(>F) = 0.000)



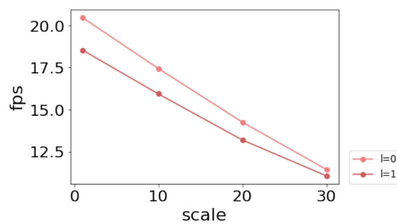
(p) Scale vs. Light (Number of Point = 5731114, texSize = 96.0, PR(>F) = 0.074)



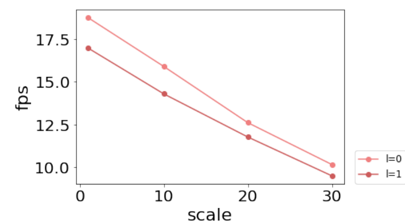
(q) Scale vs. Light (Number of Point = 5731114, texSize = 192.0, PR(>F) = 0.000)



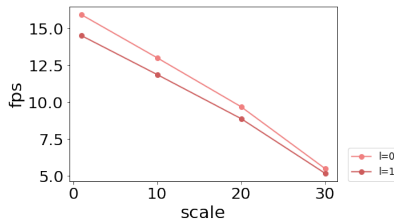
(r) Scale vs. Light (Number of Point = 5731114, texSize = 480.0, PR(>F) = 0.000)



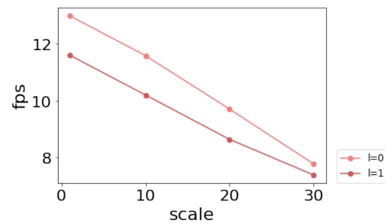
(s) Scale vs. Light (Number of Point = 5731114, texSize = 960.0, PR(>F) = 0.000)



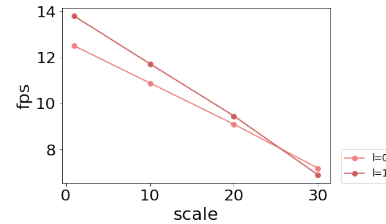
(t) Scale vs. Light (Number of Point = 5731114, texSize = 2400.0, PR(>F) = 0.000)



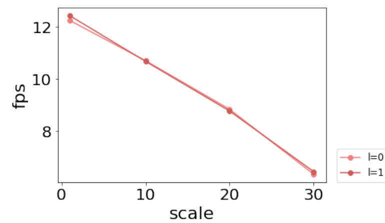
(u) Scale vs. Light (Number of Point = 5731114, texSize = 4800.0, PR(>F) = 0.000)



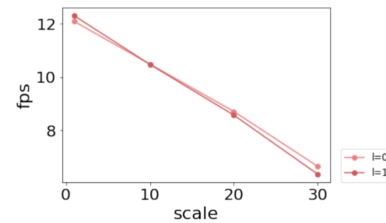
(v) Scale vs. Light (Number of Point = 11462229, texSize = 0.0, PR(>F) = 0.000)



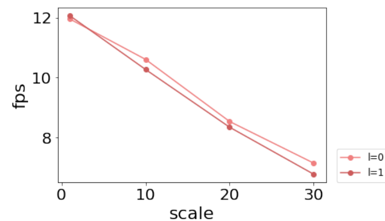
(w) Scale vs. Light (Number of Point = 11462229, texSize = 96.0, PR(>F) = 0.000)



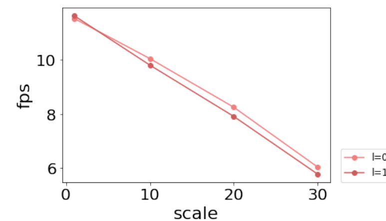
(x) Scale vs. Light (Number of Point = 11462229, texSize = 192.0, PR(>F) = 0.398)



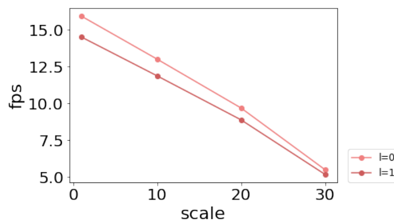
(y) Scale vs. Light (Number of Point = 11462229, texSize = 480.0, PR(>F) = 0.152)



(z) Scale vs. Light (Number of Point = 11462229, texSize = 960.0, PR(>F) = 0.002)



() Scale vs. Light (Number of Point = 11462229, texSize = 2400.0, PR(>F) = 0.023)



() Scale vs. Light (Number of Point = 11462229, texSize = 4800.0, PR(>F) = 0.000)

Figure 4.6: Pair-wise ANOVA Comparison of Medians of Scale and Light, Grouping by Number of Point and Texture Size.

Chapter 5

Summary and Future Work

In this research, we implemented an appearance child node called "*PointProperties*", which is capable of attenuating point size by distance, mixing texture color with point color, handling different geometry types, and responding to light sources. We also examined the rendering speed of binary geometries with "*PointProperties*". With our implementation, the X3DOM users are able to amplify the appearance of their point-based geometries, and point cloud researchers are able to exhibit their point cloud with X3DOM on their websites.

The experiment on the parameters' impact on performance shows that the major factors that affect the Frames-Per-Second (FPS) are: (1) the number of points, (2) point size and light, and (3) texture size ranking by their influences on the rendering speed. These results provide a reference for the rendering limitation of large point clouds on an ordinary laptop. As the experiment is performed on research data of a laboratory at Virginia Tech, the experiment results could be especially useful for VT researchers in the field of agriculture, geography, and civil engineering.

There are also limitations to this research. With time limitation, we were not able to implement techniques used in state-of-the-art point cloud visualization tools and compare them with the current implementation of X3DOM. Besides, the experiment is only performed on a single device. Considering that different devices are usually different from each other (e.g., hardware performance, the volume of memories, and data exchange strategies), our experiment results are not representative enough. To be more persuasive, we should examine the parameters against performance on more devices.

The future work can be divided into 3 stages. The first stage is to refine the current *PointProperties* implementations. Currently, our *PointProperties* node support only Blinn-phong lighting model, and sphere-like fragment normals. The shader code should be extended to involve different X3DOM lighting models, normal estimation methods.

The second stage is to support multi-pass shaders, such as multiple light sources, shadows, and GPU picking. Different from ParticleSet node, which does not support any lighting, shadowing, and picking effects, *PointProperties* should enable points seems like 3D objects with volumes, while maintaining the high performance of point-based rendering at the same. Besides, current shader does not update depth buffer after normal estimation. This may lead to a wrong impression of spacial relationships between closeup points, as there are no partial culling of point splats. The solution to the culling problem requires an additional pass to render the new fragment depth to the depth buffer.

The third stage is to develop shaders that work specifically to point-based rendering. Although there are already several online point cloud visualization tools like Potree, the high customizability and easy-to-learn features of X3DOM makes it worth to develop a point-based rendering toolset for the X3DOM architecture. Besides, innovative point splatting methods are worth to be explored. For example, instead of point spacing and distance to the camera, the size of points can also be defined by the incident direction and foot print of the laser scan.

Bibliography

[1] Cloudcompare. URL http://www.cloudcompare.org/.

[2] James Ahrens, Berk Geveci, and Charles Law. *ParaView: An End-User Tool for Large Data Visualization.* Elsevier, 2005.

[3] Riyad Al-Rousan, Mohd Shahrizal Sunar, and Hoshang Kolivand. Geometry-based shading for shape depiction enhancement. *Multimedia Tools and Applications*, 77(5): 5737–5766, 2018.

[4] et al. Alexa, Marc. Computing and rendering point set surfaces. *Visualization and Computer Graphics IEEE Transactions on 9.1*, pages 3–15, 2003.

[5] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva. Point set surfaces. In *In Proceedings Visualization, 2001. VIS'01*, pages 21–29. IEEE, 2001.

[6] Thomas Annen, Tom Mertens, Philippe Bekaert, Hans-Peter Seidel, and Jan Kautz. Convolution shadow maps. *Rendering Techniques*, 18:51–60, 2007.

[7] Louis Bavoil and Miguel Sainz. Screen space ambient occlusion. *NVIDIA developer information: http://developers. nvidia. com*, 6, 2008.

[8] R Bedre. Anova using python, Oct 2018. URL https://reneshbedre.github.io/blog/anova.html.

[9] J. Behr, Y. Jung, T. Drevensek, and A. Aderhold. Dynamic and interactive aspects of x3dom. In *In Proceedings of the 16th International Conference on 3D Web Technology*, pages 81–87, 2011.

[10] J Behr, Y Jung, T Franke, and T. Sturm. Using images and explicit binary container for efficient and incremental delivery of declarative 3d scenes on the web. In *In Proceedings of the 17th international conference on 3D web technology*, pages 17–25, 2012.

[11] James F Blinn. Models of light reflection for computer synthesized pictures. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198, 1977.

[12] Mario Botsch, Alexander Hornung, Matthias Zwicker, and Leif Kobbelt. High-quality surface splatting on today's gpus. In *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005.*, pages 17–141. IEEE, 2005.

[13] Kobbelt L Botsch M., Spernat M. Phong splatting. In *In Proceedings of the First Eurographics Conference on Point-Based Graphics 2004, SPBG '04*, pages 25–32, 2004.

[14] Christian Boucheny. *Interactive Scientific Visualization of Large Datasets: Towards a Perceptive-Based Approach*. Thesis, 2009.

[15] Christian Boucheny. *Interactive Scientific Visualization of Large Datasets: Towards a Perceptive-Based Approach*. Thesis, 2009.

[16] Jamieson Brettle and Frank Galligan. Introducing draco: compression for 3d graphics, 2017.

[17] D. Brutzman and L. Daly. *X3D: extensible 3D graphics for Web authors*. Elsevier, 2010.

[18] G. Bui, Le. T., B. Morago, and Y. Duan. Point-based rendering enhancement via deep learning. *The Visual Computer*, 34(6-8):829–841, 2018.

[19] Paolo Cignoni, Roberto Scopigno, and Marco Tarini. A simple normal enhancement technique for interactive non-photorealistic renderings. *Computers & Graphics*, 29(1): 125–133, 2005.

[20] Paolo Cignoni, Massimiliano Corsini, and Guido Ranzuglia. Meshlab: an open-source 3d mesh processing system. 2008(73), 2008. URL http://dblp.uni-trier.de/db/journals/ercim/ercim2008.html#CignoniCR08.

[21] Robert A Cohen, Dong Tian, and Anthony Vetro. Point cloud attribute compression using 3-d intra prediction and shape-adaptive transforms. In *2016 Data Compression Conference (DCC)*, pages 141–150. IEEE, 2016.

[22] Robert A Cohen, Dong Tian, and Anthony Vetro. Attribute compression for sparse point clouds using graph transforms. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 1374–1378. IEEE, 2016.

[23] Robert L Cook and Kenneth E Torrance. A reflectance model for computer graphics. *ACM Siggraph Computer Graphics*, 15(3):307–316, 1981.

[24] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. *ACM Transactions on Graphics (TOG)*, 22(3):657–662, 2003.

[25] Doug DeCarlo, Adam Finkelstein, and Szymon Rusinkiewicz. Interactive rendering of suggestive contours with temporal coherence. In *Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering*, pages 15–145, 2004.

[26] O. Devillers and P. . Gandoin. Geometric compression for interactive transmission. In *Proceedings Visualization 2000. VIS 2000 (Cat. No.00CH37145)*, pages 319–326, 2000.

[27] Sören Discher, Leon Masopust, and Sebastian Schulz. A point-based and image-based multi-pass rendering technique for visualizing massive 3d point clouds in vr environments. 2018.

[28] & Döllner J. Discher S, Richter R. A scalable webgl-based approach for visualizing mas-

sive 3d point clouds using semantics-dependent rendering techniques. In *In Proceedings of the 23rd International ACM Conference on 3D Web Technology*, pages 1–9, 2018.

[29] N. Eicke, Y. Jung, and A. Kuijper. Robust real-time shadows for dynamic 3d scenes on the web. In *International Conference on Human-Computer Interaction*, pages 574–578. Springer, 2014.

[30] Randima Fernando. Percentage-closer soft shadows. In *ACM SIGGRAPH 2005 Sketches*, pages 35–es. 2005.

[31] Enrico Gobbetti and Fabio Marton. Layered point clouds: a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Computers & Graphics*, 28(6):815–826, 2004.

[32] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 447–452, 1998.

[33] H Gouraud. *Computer Shading of Curved Surfaces*. PhD thesis, Ph. D. dissertations, University of Utah. Also in: IEEE Transaction on Computers, 1971.

[34] Markus Gross and Hanspeter Pfister. *Point-based graphics*. Elsevier, 2011.

[35] Stefan Gumhold, Zachi Kami, Martin Isenburg, and Hans-Peter Seidel. Predictive point-cloud compression. In *ACM SIGGRAPH 2005 Sketches*, pages 137–es. 2005.

[36] & Liang X. He Z. A multiresolution object space point-based rendering approach for mobile devices. In *In Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 7–13, 2007.

[37] Paul S Heckbert. Adaptive radiosity textures for bidirectional ray tracing. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 145–154, 1990.

[38] Eric Heitz. Understanding the masking-shadowing function in microfacet-based brdfs. 2014.

[39] Yan Huang, Jingliang Peng, C-C Jay Kuo, and M Gopi. A generic scheme for progressive point cloud coding. *IEEE Transactions on Visualization and Computer Graphics*, 14 (2):440–453, 2008.

[40] J. Jankowski, S. Ressler, K. Sons, Y. Jung, J. Behr, and P. Slusallek. Declarative integration of interactive 3d graphics into the world-wide web: Principles, current approaches, and research agenda. In *In Proceedings of the 18th International Conference on 3D Web Technology*, pages 39–45, 2013.

[41] Henrik Wann Jensen. *Realistic image synthesis using photon mapping.* AK Peters/CRC Press, 2001.

[42] Julius Kammerl, Nico Blodow, Radu Bogdan Rusu, Suat Gedikli, Michael Beetz, and Eckehard Steinbach. Real-time compression of point cloud streams. In *2012 IEEE International Conference on Robotics and Automation*, pages 778–785. IEEE, 2012.

[43] A. Kerren, J. Stasko, J. D. Fekete, and C. (Eds.). North. Information visualization: Human-centered issues and perspectives. *Springer*, 4950, 2008.

[44] Gordon Kindlmann, Ross Whitaker, Tolga Tasdizen, and Torsten Moller. Curvature-based transfer functions for direct volume rendering: Methods and applications. In *IEEE Visualization, 2003. VIS 2003.*, pages 513–520. IEEE, 2003.

[45] Michael Kolomenkin, Ilan Shimshoni, and Ayellet Tal. Demarcating curves for shape illustration. In *ACM SIGGRAPH Asia 2008 papers*, pages 1–9. 2008.

[46] Andrew Lauritzen and Michael McCool. Layered variance shadow maps. In *Graphics Interface*, volume 8. Citeseer, 2008.

[47] Yunjin Lee, Lee Markosian, Seungyong Lee, and John F Hughes. Line drawings via abstracted shading. *ACM Transactions on Graphics (TOG)*, 26(3):18–es, 2007.

[48] M. LEVOY and S. RUSINKIEWICZ. Qsplat: A multiresolution point rendering system for large meshes. In *ACM SIGGRAPH*, page 343–352, 2000.

[49] M. LEVOY and T. WHITTED. The use of points as a display primitive. Report, University of North Carolina at Chapel Hill, 1985.

[50] M. Limper, M. Thöner, J. Behr, and D. W. Fellner. Src-a streamable format for generalized web-based 3d data transmission. In *In Proceedings of the 19th International ACM Conference on 3D Web Technologies*, pages 35–43, 2014.

[51] Max Limper, Yvonne Jung, Johannes Behr, and Marc Alexa. The pop buffer: Rapid progressive clustering by geometry quantization. In *Computer Graphics Forum*, volume 32, pages 197–206. Wiley Online Library, 2013.

[52] Schütz M. *Potree: Rendering large point clouds in web browsers.* Thesis, 2016.

[53] Rufael Mekuria, Kees Blom, and Pablo Cesar. Design, implementation, and evaluation of a point cloud codec for tele-immersive video. *IEEE Transactions on Circuits and Systems for Video Technology*, 27(4):828–842, 2016.

[54] Martin Mittring. Finding next gen: Cryengine 2. In *In ACM SIGGRAPH 2007 courses*, page 97–121, 2007.

[55] Vicente Morell, Sergio Orts, Miguel Cazorla, and Jose Garcia-Rodriguez. Geometric 3d point cloud compression. *Pattern Recognition Letters*, 50:55–62, 2014.

[56] M. PFISTER, H.and ZWICKER, J. V. BAAR, and GROSS M. Surfels: Surface elements as rendering primitives. In *ACM SIGGRAPH*, page 335–342, 2000.

[57] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[58] & Döllner J. Richter R. Out-of-core real-time visualization of massive 3d point clouds. In *In Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, pages 121–128, 2010.

[59] Fabrice Robinet, Rémi Arnaud, Tony Parisi, and Patrick Cozzi. gltf: Designing an open-standard runtime asset format. *GPU Pro*, 5:375–392, 2014.

[60] Radu Bogdan Rusu and Steve Cousins. 3d is here: Point cloud library (pcl). In *2011 IEEE international conference on robotics and automation*, pages 1–4. IEEE, 2011.

[61] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 197–206, 1990.

[62] Marco Salvi. Rendering filtered shadows with exponential shadow maps. *ShaderX*, 6: 257–274, 2008.

[63] Claus Scheiblauer. *Interactions with gigantic point clouds*. PhD thesis, 2014.

[64] Claus Scheiblauer and Michael Pregesbauer. Consolidated visualization of enormous 3d scan point clouds with scanopy. *Proc. CHNT*, pages 242–247, 2011.

[65] Claus Scheiblauer and Michael Wimmer. Out-of-core selection and editing of huge point clouds. *Computers & Graphics*, 35(2):342–351, 2011.

[66] Christophe Schlick. An inexpensive brdf model for physically-based rendering. In *Computer graphics forum*, volume 13, pages 233–246. Wiley Online Library, 1994.

[67] Ruwen Schnabel and Reinhard Klein. Octree-based point-cloud compression. *Spbg*, 6: 111–120, 2006.

[68] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit–An Object-Oriented Approach To 3D Graphics*. Kitware, Inc., fourth edition, 2006.

[69] & Wimmer M. Schuetz M. Progressive real-time rendering of unprocessed point clouds. In *In ACM SIGGRAPH 2018 Posters*, pages 1–2, 2018.

[70] Markus Schütz, Katharina Krösl, and Michael Wimmer. Real-time continuous level of detail rendering of point clouds. In *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pages 103–110. IEEE, 2019.

[71] Yiting Shao, Qi Zhang, Ge Li, Zhu Li, and Li Li. Hybrid point cloud attribute compression using slice-based layered structure and block-based intra prediction. In *Proceedings of the 26th ACM international conference on Multimedia*, pages 1199–1207, 2018.

[72] Dominik Sibbing, Torsten Sattler, Bastian Leibe, and Leif Kobbelt. Sift-realistic rendering. In *2013 International Conference on 3D Vision-3DV 2013*, pages 56–63. IEEE, 2013.

[73] C. Sigg, T. Weyrich, M. Botsch, and M. Gross. Gpu-based ray-casting of quadratic surfaces. In *Proceedings of the 3rd Eurographics / IEEE VGTC conference on Point-Based Graphics, 2006, SPBG '06*, pages 59–65, 2006.

[74] C. Stein, M. Limper, and A. Kuijper. Spatial data structures for accelerated 3d visibility computation to enable large model visualization on the web. In *In Proceedings of the 19th International ACM Conference on 3D Web Technologies*, pages 53–61, 2014.

[75] Dorina Thanou, Philip A Chou, and Pascal Frossard. Graph-based compression of dynamic 3d point cloud sequences. *IEEE Transactions on Image Processing*, 25(4): 1765–1778, 2016.

[76] three.js. three.js / editor, 2015. URL http://threejs.org/editor/.

[77] Dong Tian, Huifang Sun, and Anthony Vetro. Graph transformation for keypoint trajectory coding. In *2016 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 445–449. IEEE, 2016.

[78] Corey Toler-Franklin, Adam Finkelstein, and Szymon Rusinkiewicz. Illustration of complex real-world objects using images with normals. In *Proceedings of the 5th international symposium on Non-photorealistic animation and rendering*, pages 111–119, 2007.

[79] R. Vergne, R. Pacanowski, P. Barla, X. Granier, and C. Schlick. Radiance scaling for versatile surface enhancement. In *In Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 143–150, 2010.

[80] Bruce Walter, Stephen R Marschner, Hongsong Li, and Kenneth E Torrance. Microfacet models for refraction through rough surfaces. *Rendering techniques*, 2007:18th, 2007.

[81] & Lohr U. Wehr A. Airborne laser scanning—an introduction and overview. *ISPRS Journal of photogrammetry and remote sensing*, 54(2-3):68–82, 1999.

[82] T. Weyrich, S. Heinzle, T. Aila, D. B. Fasnacht, S. Oetiker, M. Botsch, C. Flaig, S. Mall, K. Rohrer, N. Felber, H. Kaeslin, and M. Gross. A hardware architecture for surface splatting. *ACM Transactions on Graphics*, 2007.

[83] T. J. Whitted. An improved illumination model for shaded display. *CACM*, 23(6): 343–349, 1980.

[84] OpenGL Wiki. Fixed function pipeline — opengl wiki, 2015. URL http://www.khronos.org/opengl/wiki_opengl/index.php?title=Fixed_Function_Pipeline&oldid=12121.

[85] OpenGL Wiki. Rendering pipeline overview — opengl wiki, 2019. URL http://www.khronos.org/opengl/wiki_opengl/index.php?title=Rendering_Pipeline_Overview&oldid=14511.

[86] OpenGL Wiki. History of opengl — opengl wiki, 2019. URL http://www.khronos.org/opengl/wiki_opengl/index.php?title=Core_Language_(GLSL)&oldid=14594.

[87] OpenGL Wiki. History of opengl — opengl wiki, 2020. URL http://www.khronos.org/opengl/wiki_opengl/index.php?title=History_of_OpenGL&oldid=14676. [Online; accessed 25-August-2020].

[88] Ke Zhang, Wenjie Zhu, and Yiling Xu. Hierarchical segmentation based point cloud attribute compression. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3131–3135. IEEE, 2018.

[89] Ke Zhang, Wenjie Zhu, Yiling Xu, and Ning Liu. Point cloud attribute compression via clustering and intra prediction. In *2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pages 1–5. IEEE, 2018.

[90] Long Zhang, Ying He, Xuexiang Xie, and Wei Chen. Laplacian lines for real-time shape illustration. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 129–136, 2009.

[91] Fukai Zhao and Xinguo Liu. 3d gradient enhancement. *The Visual Computer*, 30(1): 113–126, 2014.

[92] M. Zwicker, H. Pfister, Baar. J. van, and Gross. M. Surface splatting. In *In Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '01*, pages 371–378, 2001.

[93] M. Zwicker, Räsänen. J., M. Botsch, C. Dachsbacher, and M. Pauly. Perspective accurate splatting. In *Proceedings of Graphics Interface, 2004, GI '04*, pages 247–254, 2004.