이화여자대학교 대학원
2018학년도
박사학위 청구논문

High-Resolution 3D Painting in Virtual Reality with Volumetric Brush Models

컴 퓨 터 공 학 과
Yeojin Kim
2019

High-Resolution 3D Painting in Virtual Reality with Volumetric Brush Models

이 논문을 박사학위 논문으로 제출함

2019 년 7 월

이화여자대학교 대학원

컴 퓨 터 공 학 과 Yeojin Kim

Yeojin Kim 의 박사학위 논문을 인준함

지도교수 김 영 준 ____

심사위원 이 상 호 ____

민 동 보 ____

오 유 란 ____

김 병 문 ____

김 영 준 ____

이화여자대학교 대학원

Table of Contents

List of Figures

List of Tables

Abstract

Nowadays 2D digital painting is in the transition to 3D with virtual reality. Virtual reality (VR)-based 3D painting applications have recently surged and are now widely accepted as a new art form by artists. However, state-of-the-art VR painting systems are all surface painting systems that emit 2D surface geometries as users make brush strokes. A user has difficulty with coloring, recoloring, mixing the color, or painting semi-transparent color, while these are natural activities in the real painting. In a broader perspective in 3D digital art, various 3D art tools have been researched over decades, but 3D art tools do not focus on such painting aspects and cannot fully support diverse styles for individuals. Meanwhile, modern 3D art tools allow users to create various 3D digital art, from voxel dot designs to sophisticated 3D models. From an artistic point of view, remarkably sophisticated art in 3D is made technically possible. However, established 3D digital art tools require professional skills with several steps in the workflow and a large team of high experts, or limit the free expression due to shape representation, scale, or interfaces.

In this dissertation, as a new medium for 3D painting, a high-resolution volumetric painting system in VR is introduced to extend the 2D pixel canvas to a 3D voxel canvas. We develop a dynamic octree-based painting and rendering system using both CPU and GPU to take advantage of the characteristics of both processors—CPU for octree modeling and GPU for volume rendering. On the CPU-side, we dynamically adjust an octree and incrementally update the octree to a GPU. To allow constant neighbor access time in ray casting, our octree uses novel 3-neighbor connectivity for format simplicity and efficient storage. We further reduce the GPU-side 3-neighbor computations by precomputing a culling mask in a CPU and uploading it to a GPU. To verify the performance of our update strategy, we conduct experiments for

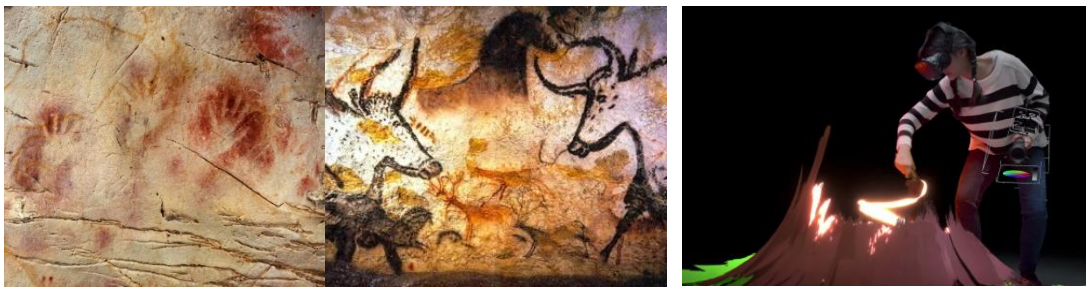
compromising low latency and the high frame rates of the rendering. We also analyze the problem of our update strategy and suggest methods to reduce artifacts in immediate visual feedback. In rendering, we introduce a cell-local coordinate system as a solution for the numerical error problem in ray casting through high-resolution octree. We analyze the numerical error propagation with the cell-local coordinate system in ray casting, present a theoretical error bound, and prove our theory by experiments. To accelerate rendering, we revise the CPU-based quadtree/octree interpolation for the GPU and design foveated rendering based on a quadtree.

We also address the problem of generalizing 2D brushes that manipulate a large number of pixels to 3D brushes that manipulate a large number of voxels in the aspect of a 3D painting. Toward this goal, we first generalize 2D brush tools commonly found in 2D digital painting system to 3D, including voxel blur, voxel smudge, and voxel dodge/burn, working for high-resolution octrees. We also propose volumetric brush tools designed to solve the problems specific in volumetric painting. For example, voxel resolution control tools are introduced to adjust painting details and manage memory consumption while reducing repetitive painting tasks. “Voxel melt” refines voxels or smooths out resolution variations while retaining the painting details using resolution diffusion. Easy-to-use voxel merge filters, such as room, voxel mosaic, and voxel merging based on iso-values, save memory by coarsening invisible or unnoticed voxels. Finally, we propose hybrid brush models, by combining surface and volumetric paintings to address the problems of depth perception and stoke neatening in volumetric painting.

I. Introduction

A. Background

From prehistoric times to the present, humans have painted on 2D canvas. Around 40,000 years ago, homo sapiens created the oldest painting in El Castillo [Fig. I-1 (a)] and a Neanderthal man engraved rocks in Gorham's cave around the same time. Afterward, pioneers in art continued to find new art forms over many centuries. Many interesting attempts, such as Realism, Surrealism, Impressionism, or Cubism have been made on 2D canvas. Today, 2D painting is transitioning to 3D with virtual reality (VR). Recently, VR-based 3D painting [Fig. I-1 (b)] applications have surged and are now widely accepted by artists as a new art form. As generating 3D art is vital in computer graphics, such as product design, animation, and game design, the importance of VR painting is expected to grow.

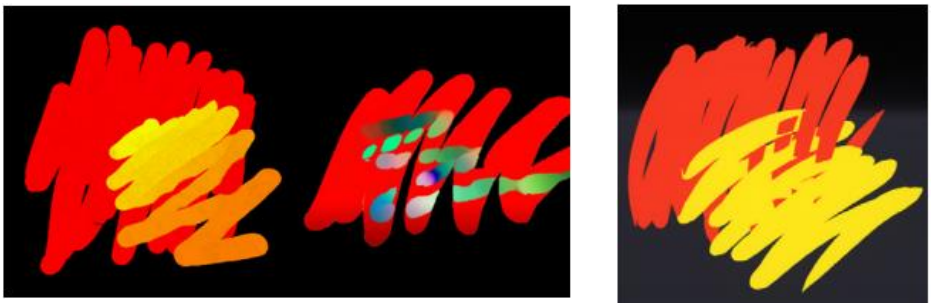


(a) Cave Art of Homo Sapiens (B.C. 38800~17000) (b) Tilt Brush [33] (2015)

Figure I-1. A transition from a 2D canvas to a 3D canvas

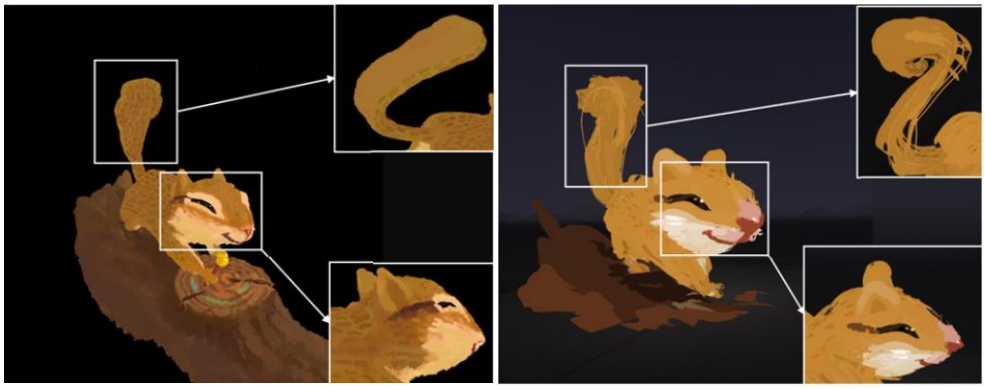
State-of-the-art 3D painting systems in VR, such as Tilt Brush [33] and Quill [66], are surface painting systems that emit 2D surface geometries as users make brush strokes [Fig. I-2 (b)].

However, these systems have intrinsic boundaries. Color mixing between strokes is not supported as overlaps between stroke geometries are infinitely thin and expensive to robustly compute. The simple recoloring of part of a stroke is complicated because a texture map must be invoked, such that updated colors may be stored into the texture map. In addition, semi-transparent volume depiction is limited because efficient handling of many stroke geometries in high-depth complexity is nontrivial.



(a) Volumetric painting [91] (b) Surface painting [33]

Figure I-2. Comparison of mixed strokes between two different painting systems



(a) Volumetric painting (Ours [91])

(b) Surface painting (Tilt Brush [33])

Figure I-3. Comparison of paintings between two different painting systems

Thus, the first aim of this dissertation is to explore a new route for VR painting in 3D space; namely, *volumetric painting*. Volumetric strokes applied to a 3D grid are amenable to depicting both solid and non-solid shapes. Volumetric painting also naturally supports color mixing [Fig. I-2 (a)]. Moreover, users can repeatedly apply strokes to the same area until they are satisfied with the mixed color pattern without considering occlusion and z-fighting, as demonstrated in Figure 1-3 (a), which is a very typical practice in surface painting. Finally, a volumetric painting system naturally handles semi-transparent strokes.

In a broader perspective of 3D digital art, various 3D art tools have been extensively researched over decades. Modern 3D art tools allow users to create various 3D digital art, from voxel dot designs to sophisticated 3D models. From an artistic point of view, remarkably realistic art in 3D is only technically possible by a large team of high experts [24]. Established 3D digital art tools require professional skills with several steps in the workflow, or limit the free expression due to shape representation, scale, or user interface.

In 3D modeling, modeling artists undergo intensive tool training for sculpting, texture mapping, and parametric controlling for material properties/rendering. Such tools are complex and difficult for 2D digital artists and traditional artists, who mostly create artwork with a paintbrush. Painting on models is also difficult for the following reasons: (1) creating an underlying model is a prerequisite for following texture mapping/authoring, (2) the discordance between model space and texture space requires parameterization, and (3) painting on models cannot fully exploit 3D space. For these reasons, intuitive interfaces for modeling, such as sketch-based modeling [50], [75] or painting with implicit 3D canvas [44] have been studied.

Figure I-4. Sketch-based modeling [75]

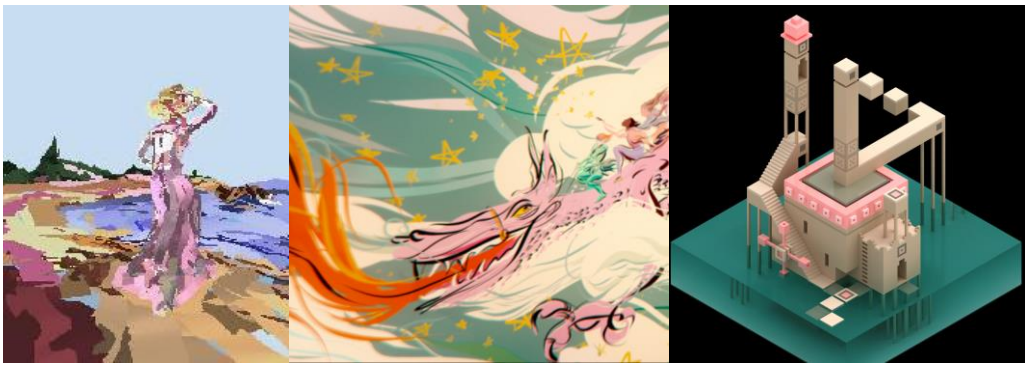
Figure I-5. Implicit 3D canvas [44]

Sketch-based modeling introduces a simple way of creating 3D models; however, inherently poses modeling errors because of ambiguous transitions from 2D to 3D. To avoid ambiguity, a user requires extra touch-up on 3D models, another drawing in various viewpoints [1], [73], or create a limited range of shapes based on primitives [6], [92]. In a model painting step, OverCoat [44] supports off-model painting for users to blend color like digital painting on 2D canvas. However, even with these easy-to-use interfaces, users must make back-and-forth steps between model creation and painting.

One of the 3D painting methods that generate meshes [25], [26], [33], [54], [66], [71], called surface painting, is aimed for 3D digital art rather than product design. Surface painting can both express thin curves or volumetric hull with brush interfaces, yet, it does not provide stroke

composition, pigment deposition with a color mix, or partial modifications that are natural attributes of painting. In addition, users sometimes use several types of software for painting and rendering to achieve the desired style.

Another 3D painting method is voxel editing [28], [61], where users control voxel properties with high-quality ray tracing/path tracing. Current voxel painting systems support various sizes of paintings, however, a CSG-like interface overloads users to refine their artwork. For example, a moderate-sized 3D painting, such as Figure VII-1, has over ten million voxels. Such a heavy workload makes 3D art remain a very specialized domain.



(a) Keefe et al. [25] (b) Quill [66] (c) MagicaVoxel [28]

Figure I-6. Examples of surface painting (left and middle) and voxel painting (right)

The second aim of this dissertation is to study volumetric paintbrush models that allow a novice user to create 3D artwork in a 3D digital painting metaphor. When a novice user is inspired to create 3D art, she or he can immediately begin through 3D brush stroking without any knowledge of 3D modeling. Like classic 2D painting, a user can simultaneously shape and color in the 3D space. A user does not need to consider a limited painting area on 3D space or the ambiguity of strokes.

B. Research Objectives and Contributions

The main objective of the dissertation is to study volumetric painting as a new 3D digital painting. For this research goal, our key questions are:

- How can we build a volumetric painting system that extends 2D digital painting, especially supporting various scale painting in 3D space?
- What are the minimum performance and memory requirements for interactivity and affordability in such a system? How can we achieve interactivity and affordability?
- How can we extend 2D painting models (e.g., 2D brush models) in a volumetric painting system?
- What specific brush models are newly required for volumetric painting?

Unlike a 2D planar canvas, on a 3D canvas, perspective can be compromised: distant objects in the background can be painted at a relatively large size compared to the foreground objects. For example, distant mountains can be painted at their actual sizes with large voxels. However, this requires very large canvas support. For a large canvas, we use an array of octrees of high-depth (e.g., level 24 or higher). Using an octree, we can maintain a very large canvas with a painting space of up to $40Km^3$ and with very fine details of tiny voxels painted at a size of $0.3mm^3$ with respect to the typical room-scale VR setup.

With a prototype of volumetric painting system based on an array of 24-level octrees, we conduct extensive experiments and conversations with users. Based on our prior experiments, we have concluded that the following elements are required for an interactive and affordable volumetric painting system in VR:

- ***Dynamic tree update.*** Users will continuously modify the underlying tree. Therefore, we need to update the tree dynamically.
- ***Constant frame rates.*** Users spend several hours painting in VR, and therefore, consideration must be given to mitigate the possibility of VR-sickness [18], [40], [80]. One source of such sickness is hitching or stuttering in the rendering frame rates, which should not be compromised; the frame rates should stay constant.
- ***Low latency stroke display***. When a user applies a stroke, the tree should be modified immediately and rendered back to the user. Therefore, we require low latency for stroke display.
- ***Low memory consumption***. When a very large canvas is used, we found users tend to paint a very large world and add details in multiple locations. Thus, a low memory requirement is beneficial for maintaining a large canvas.

Based on a volumetric painting system, we design various volumetric brush models which extend common brush models in 2D digital painting. Beyond a simple extension from previous digital painting techniques, our brush models address the following fundamental challenges in volumetric painting. First, manipulating properties of each voxel rapidly increases user workload as the painting complexity increases. For fine quality variously-sized 3D artwork, a painting interface is essential. Second, memory consumption affects time and the quality of paintings. The size of paintings gradually increases over time and eventually reaches the memory limit. For a user who has no engineering knowledge, yet, does not want to give up the scale or the details of a painting, control tools for the voxel resolution are required. Last, the performance of rendering and volume authoring always should be balanced for usability.

In summary, our contributions are as follows:

For a volumetric painting system,

- Interactive adjustment of a large octree in a CPU for painting.
- Strategies to perform adaptive painting strokes and distributed grid adjustment over multiple time steps.
- Incremental, low latency octree update to the GPU without adverse impact on the already GPU-intensive volume rendering.
- New and simple octree neighbor connectivity with only three connections per cell for fast traversal to neighboring cells.
- Numerical error propagation analysis during ray traversal on a high-depth octree.
- Novel quadtree-based foveated rendering for acceleration.

For volumetric brush models:

- Interactive and intuitive volumetric brush models that manipulate a large number of voxels with adaptive grids.
- The generalization of common and frequently used 2D brush models, including paint, recolor, color mix, blur, smudge, dodge/burn, etc.
- Novel volume-specific brush models, such as voxel resolution controls for memory management, a voxel split brush based on diffusion and easy-to-use voxel merging filters.
- A shift to a hybrid 3D painting system with surface painting and volumetric painting and hybrid brush models that address the problem of depth perception and stroke neatening in volumetric painting.

C. Organization

The remainder of this dissertation is organized as follows. In Chapter II, we survey 3D painting, the elements for volumetric painting (i.e. octrees and rendering based octrees), and brush models for volumetric painting. In Chapter III, we outline tasks in a volumetric painting system for clarity. In Chapter IV, we describe underlying octree representation and its update strategy. We discuss how we dynamically update our octree on a GPU at interactive rates and weaken visual artifacts from delayed updates. In Chapter V, we develop an accurate ray casting based on the 24-level octrees to avoid ray drifting and distortion. We analyze numerical error propagation with our ray casting method and discuss experimental results of accuracy. Based on the interpolation of adaptive grids, we introduce a rendering acceleration technique, a quadtree-based foveated rendering. We briefly describe the interface of our system and introduce volumetric brush models, which are the extended brush models from 2D brush models and the volume-specific brush models in Chapter VI. We present and discuss our painting results in Chapter VII, and finally, we conclude in Chapter VIII.

II. Related Work

In this chapter, we first explore a 3D painting system, including 3D modeling, surface painting, voxel editing, and volumetric painting. We then survey the relevant octree works, especially focusing on representation, updating, and authoring techniques, as well as GPU-based ray casting for high-resolution volumetric painting and rendering. Last, we study previous volumetric brush models for voxel painting.

A. 3D Painting System

1. 3D Modeling and Texture Authoring

3D painting on surfaces [53], [76] has been researched well to achieve high quality in 3D product designs today. The core idea of 3D painting on surfaces is to separate image space and 3D object space and bridge the two spaces through parameterization. By using high-resolution color-field space, a user can paint detailed color on given models. Various color representation, for example, multiple uniform 2D/3D textures [3], [58], adaptive 2D/3D textures [4], [19], [23], [41], [85], a texture atlas [11], [74], point samples [7], [65], per face texture, mesh color, etc., have been used for 3D painting on surfaces (Fig. II-1). While all these works restrict expressible space to the surface of the model, OverCoat [44] proposed isosurface and cross-level painting tools which relax the restriction on the 3D painting space. Similar to Kalnins et al. [78], OverCoat supports non-photorealistic painting tools on 3D models to pursue the freedom of expression in 3D space. OverCoat exploits the signed distance field to embed 2D paint strokes to 3D space and renders strokes by projecting them to camera space (Fig. II-2 left).

(a) Octree Texture [85] (b) Point Samples [7]

(c) Texture Atlas [11] (d) Per-face Texture [11]

Figure II-1. Various coloring methods for painting on 3D models

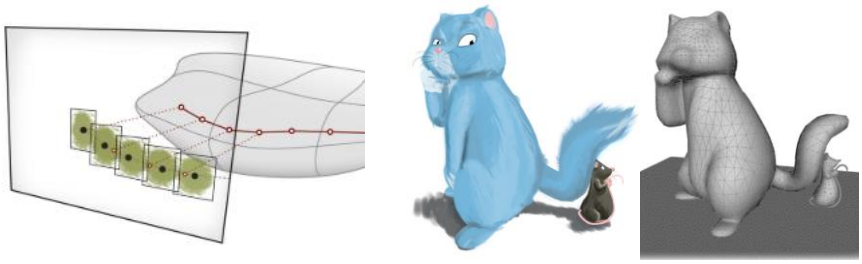


Figure II-2. Painting on isosurfaces of the underlying 3D model [44]

Regardless of painting algorithms, sculpting and painting are integrated into software, but two steps are separate and dependent for each step in the 3D modeling. Sculpting precedes painting for equipping vertex positions, textures, or distance field to mapping color in 3D space (Fig. II-2 right). For the majority of users, sculpting with fine control is a nontrivial task and requires professional training. In our study, we develop and exploit an octree-based volumetric painting system that color representation and model space coincide in high-resolution. We do not assume any underlying models and their surface representation, such as polygonal meshes or NURBs;

therefore, a user can paint 3D shapes and colors in the same way as pixel painting without space restriction. Shape and color modification lie in one space, so back-and-forth steps between sculpting and painting are unnecessary.

2. Surface Painting

Besides a 3D model painting, surface painting is a 3D painting that generates thin, connected open surface with color. This kind of shape representation is expensive to express in voxel editing and volumetric painting because voxels are refined to very small sizes and consume a lot of memory (Fig. I-6 left). Surface painting can depict a volumetric stroke by generating a shell along with a stroke [25], [26], [33], [54], [66]. Some of the semi-transparent volumetric effects like fog also can be expressed based on 2D textures [33], [66]. Currently, only a few surface painting systems [29], [66] support partial stroke modification and stroke merging similar to 3D modeling. This is because identifying the intersected meshes and modifying their properties are complex tasks. Rosales et al. [29] studied stroke merging that connects triangle strips in surface painting to form a full 3D model (Fig. II-3); however, a user cannot merge sparse (Fig. II-3 left) or randomly oriented strokes (Fig. II-3 right). For this reason, such functionalities are devolved onto other painting/rendering/modeling tools. A user needs to export their artwork from one software to another and learn how to use this software to achieve the desired art styles. For a novice user, z-fighting [Fig. I-2 (b)] and occlusion make users repeatedly draw strokes on similar locations.

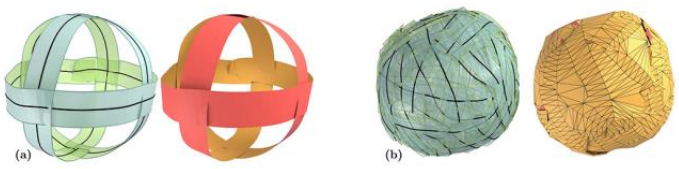
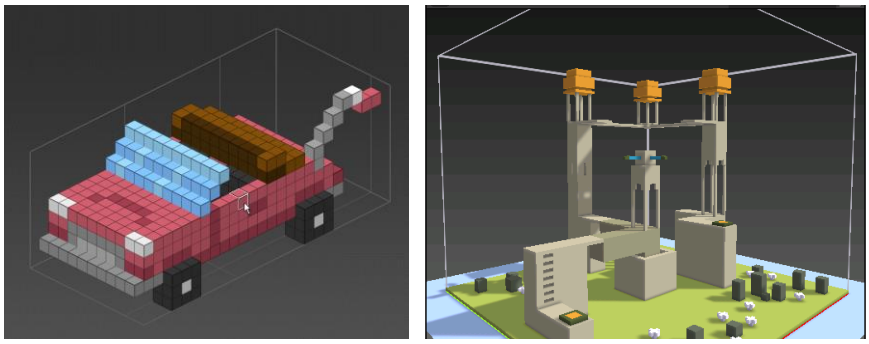


Figure II-3. Nontrivial stroke merging problems in surface painting [29]

3. Voxel Editing and Volumetric Painting

Instead of painting on a separate color representation or triangle stripes generation, some research and software of 3D painting employ voxels. When using voxels, a user can begin with a blank canvas and partially modify their artwork by painting, recoloring, and erasing voxels. Semi-transparent, refractive, or reflective expression is also possible. Compared to other 3D painting methods, it is easy to design topologically complex objects with voxels. Voxel editing is a painting method to create 3D voxel art by dotting on a 3D space [28], [32], [52], [57], [61]. Some voxel editors [28], [32], [61] provide a flexible creation environment to change artwork styles with a user-defined shader. Essentially, voxel editors are based on 3D uniform textures (Fig. II-4), however, a large-scale artwork can be supported by combining a voxel engine with sparse voxel octrees [48], [79], [86].

Voxel editors support a palette tool to manipulate the color of many voxels and uniform volumetric brushes [28], [32], [61]. Yet, they do not support a color mix and pigment deposition and have CSG-like tools, such as voxel dotting, carving tools, or snapping tools. In volumetric painting, cloud painting [16], [77] has been studied for modeling clouds with scattering (Fig. II-5). Using cloud painting, a user can generate complex cloud models with intuitive painting interfaces with no knowledge of 3D modeling, volumetric effect, and lighting. The aim of cloud painting is to generate specific objects in 3D space and general voxel painting is beyond their scope. To the best of our knowledge, our work is the first volumetric painting system that allows high-resolution painting in 3D based on octrees and extends 2D pixel painting.



(a) Qubicle [57] (b) MagicaVoxel [28]

Figure II-4. Uniform 3D textures in voxel editing applications

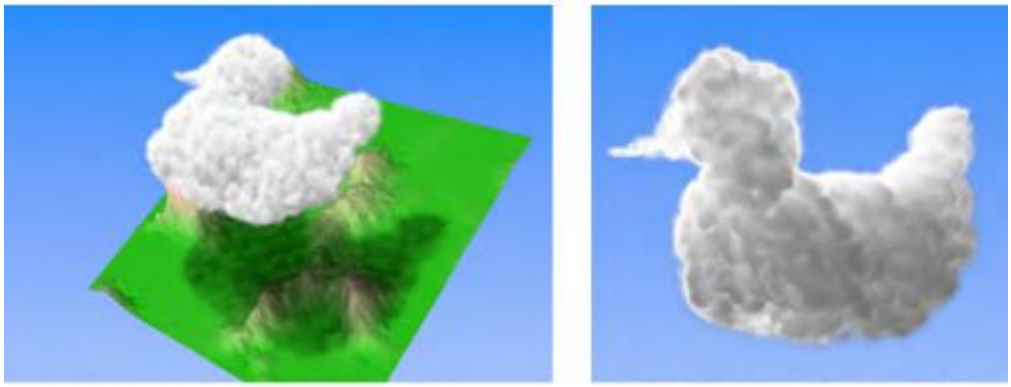


Figure II-5. Cloud painting [16]

B. 3D Adaptive Spatial Grids

1. Octree Representation

Octree grids have been applied to various problems, such as distance field generation [51], [84], texturing [19], [23], [85], modeling [5], [60], [88], simulation [30], [48], [79], model reconstruction [38], [49], and visualization [12], [13], [27], [36], to name a few. Without predetermined, fixed topological configurations, an octree is ideal for painting on a large canvas, as the tree can be refined at any location at a desirable depth. However, one concern when using a high-depth octree is the traversal time from a root cell to a leaf cell. To reduce this traversal time, a shallower tree has been used [4], [48], [79], [86]. Lefohn et al. [4] used multi-level page tables and brick-border voxels to achieve O(1) memory access, even for the look-ups from a root cell. While accessing octree cells from a root at a constant time is the key feature for coordinate-based look-ups, (e.g., texture fetching problem [4]), octree cells are still accessed locally in many applications, such as starting from the leaves, moving to the children of the non-root parent nodes, or accessing neighbors. To render a large-scale scene, full or out-of-core [13], [27], [31], [48], [56], updates to GPU have been made for rendering applications. Recently, studies on a directed acyclic graph with scalar fields [8], [22] have achieved rendering scenes in high-resolution ($32K^3 \sim 128K^3$), which is compressed on GPU memory using geometry redundancy. For dynamic updates, a directed acyclic graph needs real-time compression techniques, otherwise, reconstruction takes several minutes for updates.

2. Dynamic Octree Update

In addition to one-time construction or full reconstruction [13], [19], [23] [85], octree can also be incrementally adjusted. Here, real-time dynamic octree adjustment in GPU has been applied [4], [14]. In Crassin et al. [14], a scene is classified to static and dynamic parts and stored as separate memories in GPU. When objects move, the entire dynamic part is updated. Note that in our volumetric painting application, dynamic and static parts cannot be separated. In another study, an octree was stored on a CPU and the subtree data was streamed through CPU-GPU data transfer in a view-dependent manner [27]. To retain connectivity information with subtrees, the indices of all eight children are stored. Recently, Hoetzlein [79] supports dynamic topological updates on GPU; however, it only supports insertions for now.

C. GPU-Based Ray Casting on Adaptive Grids

Ray casting has been extensively researched for several decades [46], [55]. Since a ray traversal on 3D adaptive grids is expensive, acceleration techniques, such as neighbor precomputation, early ray termination, and empty space skipping [42] are often used. To reduce the cost of finding neighbors, the ROPE algorithm [89] was developed for a k-d tree, and neighbor linking was proposed [34], [39] for an octree. Since a k-d tree has a varying number of neighbors per cell, six ropes were linked from a cell to bounding boxes along with axial directions rather than pointing to neighbor cells directly [70]. An octree, even when 2-to-1 balanced, needs a maximum of 24 neighbors per cell. Gobbetti et al. [27] have reduced the number of neighbors down to six per cell by pointing to the parents of neighbors. In this dissertation, we use only three neighbors per cell, computed on a GPU with the primal octree represented by only two indices: a parent and the first child. Our precomputed neighbors enable stackless ray casting and the dynamic updating of an octree on-the-fly on a GPU as the tree connectivity changes.

A sparse voxel octree (SVO) [13], [86] showed both high-quality rendering and efficient ray traversal benefits of shallow tree topology and bricks. These works address the static scene rendering problem that does not require a dynamic update. Particular objects in Crassin et al. [14] can be updated dynamically while rendering. In this work, rendering with dynamic updates, which are not limited to specific objects, was not the target problem. SVOs extend to a tree with resolution configurable at each level, called OpenVDB [48]. Recently, OpenVDB structure was implemented in GPU [79], which enables efficient neighbor access using ghost voxels and GPU-based ray casting. OpenVDB [48] and GVDB [79] address rendering of dynamic scenes that do not have hard real-time constraints while updating structure simultaneously.

To the best of our knowledge, octrees as deep as 24 have not been used for ray casting. The deepest 3D adaptive structure we found in the literature was $128K^3$ [8], which is equivalent to the octree depth of 17 [whereas, our canvas is equivalent to $(4 \times 2^{24})^3$]. Moreover, the ray angle drift error has not been identified as an important challenge due to the limited size of a 3D scene. Spacing between floating points can cause sudden movement of the ray origin with continuously changing viewpoints, which makes rendering unstable in a VR environment. Ize et al. [87] used a padding factor to not miss a cell due to such precision. In this dissertation, we study a more principled approach; we propose to analyze the propagation of numerical error and ensure that ray computation does not increase the numerical error regardless of the ray length.

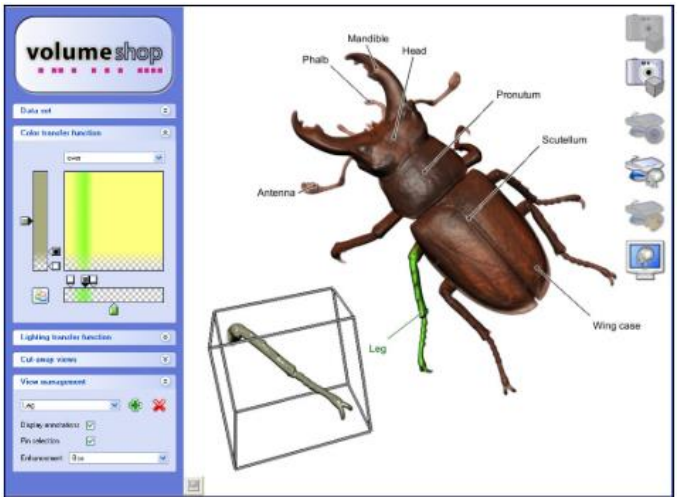
D. Volumetric Brush Models

Since prehistoric times, paint-brushing is one of the common tools to show one's creative activity and is employed in a variety of fields. For clarity, we here define a volumetric brush model as a brush interface that (1) authors elements for volume rendering at any locations in 3D, (2) has a full volume brush stamp, and (3) updates elements along the swept volume of a stroke.

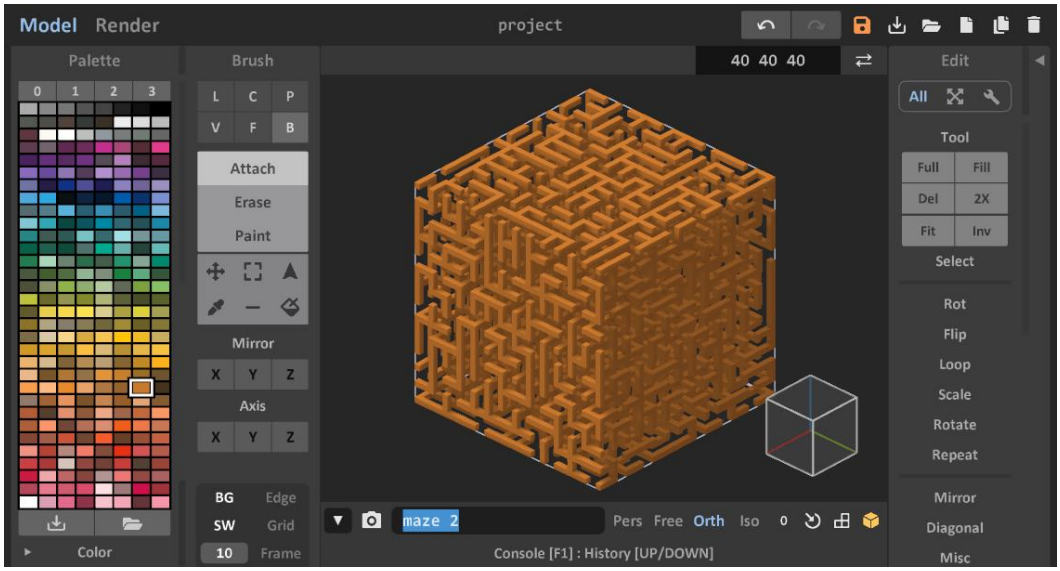
In medical imaging, a volume editing interface has been researched in the context of assisting visual perception. Such an interface aims to emphasize the region of interest or to reveal the important features in a volume data set by erasing voxels. VolumeShop [82] introduces an interactive brush model to select the region of interest in a given volume for segmentation. A set of all the intersected voxels with a brush can be transformed, rendered indifferent materials, or recolored through a color transfer function [Fig. II-6 (a)]. Bürger et al. [45] introduced a brush operating at a higher resolution than that of VolumeShop. This brush model directly interacts with 3D textures on the GPU; therefore, a user can paint either voxels or an iso-surface similar to sculpting/painting in 3D modeling. Both studies adopt a non-photorealistic recoloring or erasing, however, 3D painting is beyond their scope. Voxel editing interfaces have also developed volumetric brushes in 3D voxel dotting art [28], [32], [61]. They have a CSG-like interface (Fig.II-6), yet, these works provide a volumetric brush model functioning as a brush with max deposition. All these works assume the underlying volume is a uniform grid with limited resolution, therefore, brush models or filters for voxel resolution or 3D painting are not their concern.

Several basic volumetric brush models are presented in volumetric painting. Wei [16] and Brucks [77] presented cloud brush models based on uniform 3D textures. Although cloud brush models do not pursue a user's own style, it is an effective way of painting clouds; painting illumination and shade is a difficult task even with a single light in 3D and is much harder with light-particle interactions. Among cloud brush models, one interesting brush model is a velocity painting brush to animate clouds [77]. While cloud brush models promote specific purposes, our work provides interactive brush models for general purpose painting based on octrees.

To the best of our knowledge, volumetric brush models have not been studied in the context of 3D non-photorealistic painting that generalizes 2D pixel painting. We believe that this is the first work that identifies important problems in digital painting when transitioning from 2D to 3D and extends 2D brushes to 3D brushes.



(a) Transfer function (VolumeShop [82])



(b) A palette and a CSG-like interface (MagicaVoxel [28])

Figure II-6. Interfaces in the voxel editing applications

III. System Overview

Painting and rendering are considered simultaneous tasks in a painting system. For our interactive painting system, we build a CPU-side octree for painting and a GPU-side octree for rendering. The benefits of maintaining two copies of the same octree are to exploit the characteristics of the hardware and make them hazard-free. A CPU-side octree is primarily for painting operations, including color blending and octree adjustment (Sec. IV.D.1). A GPU-side octree is for rendering operations, including volume ray casting and rendering acceleration (Sec. V.A). Both octrees are synchronized by uploading the CPU-side octree to the GPU in a streamlined fashion (Sec. IV.D.2 ~ IV.D.4); therefore, a user can immediately see what he or she draws in a 3D space.

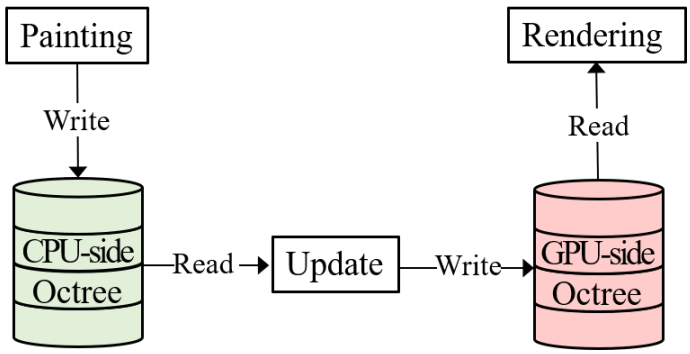


Figure III-1. Simultaneous read/write access to the octrees

However, simultaneous read/write operations can interrupt other tasks even though two octrees independently handle painting and rendering (Fig. III-1). Therefore, in our system, a stroke drawing consists of several subtasks in parallel: a stroke processing, the CPU-side octree adjustment, the GPU-side octree update, and the octree-based volume rendering (Fig. III-2). When a user starts to draw a stroke, the stroke properties, such as color, the stamp, hardness,

and segments are pushed to a stroke job queue in the painting thread. Next, we perform a stroke-cell intersection test and split/merge the intersecting cells based on stroke properties in the stroke thread. In the staging thread, we duplicate a sequential memory block containing colored, split, or merged cells into a temporary buffer called a *staged block* (Fig. IV-6). We also queue staged blocks and later pop them when overwriting staged blocks to the GPU in the upload thread. Last, we render output based on the octree-based ray casting in the rendering thread.

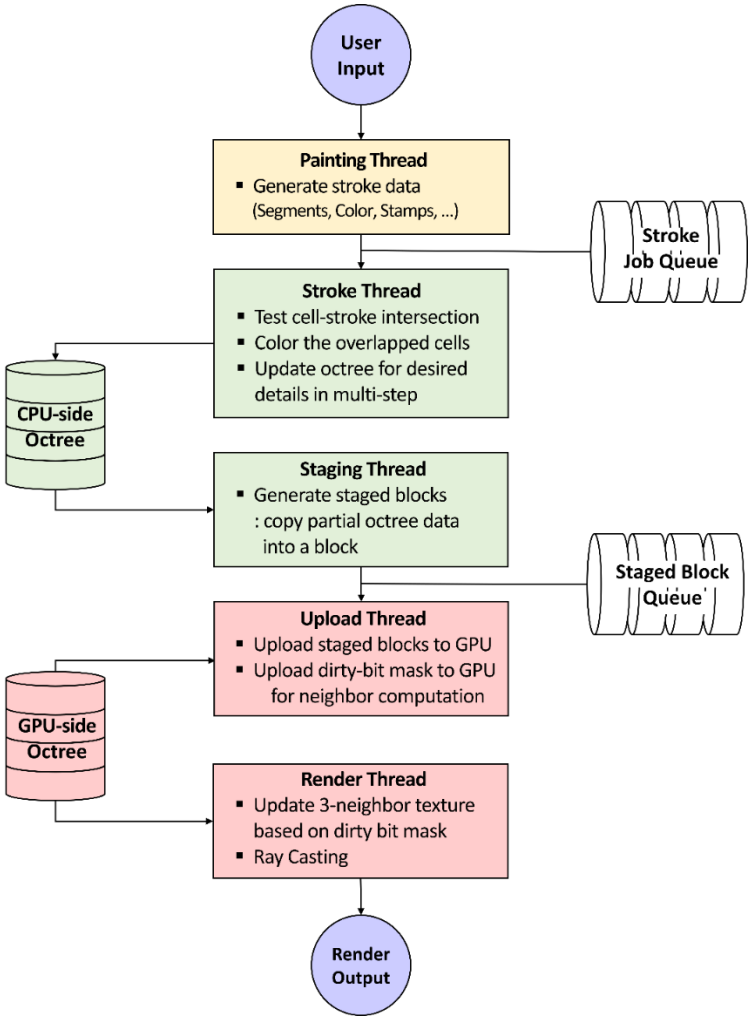


Figure III-2. A flow of CPU-side threads

IV. Dynamic Octree Representation

In our system, a 3D array of octrees is used to support deep levels and efficient octree traversal, which lead to high resolution and dynamic volumetric field authoring. Since each element of a 3D array corresponds to the root cell of a single octree, we call this array a *root array*. The maximum depth of each element in a root array is currently up to 24 levels, which sufficiently supports fine details in the painting. A root array can reduce the tree depth in octree traversal compared to a single octree by assuming that the level of root cells is greater than zero. For example, the highest resolution of a 64^3 root array with 14-level octrees is equal to that of a 20-level single octree. The maximum depth in octree traversal is 14 in the former and 20 in the latter. A root array also provides several other benefits, including trivial parallelization for each element. However, these advantages can rapidly diminish with highly adaptive details in a large canvas. Therefore, we depend on a relatively coarse root array (i.e., a 4^3 array of 24-level octrees, which is equal to a single 26-level octree). In the room-scale VR environment, this resolution sustains a volumetric space from 0.3mm^3 to 40km^3.

A. Dynamic Octree Representation on a CPU

Our dynamic octrees are based on a 2:1 balanced octree [10]. In a 2:1 balanced octree, the depth between every cell and their neighbors is less than or equal to one. One cell is refined or coarsened to eight children for simplicity, as painting can occur anywhere in the 3D space. For each cell, we use the unique index, *I*, of the cell rather than pointers, similar to Gobbetti et al. [27].

As illustrated in Figure IV-1, our CPU-side octree consists of many linear memory pools that are accessed through *I*. Parent and child pools are essential and define the structure of our octree. In a parent pool and a child pool, the indices of the parent and the first child for each cell are stored. We can obtain the indices of the remaining seven children by consecutively numbering from the first child. Any volumetric properties (e.g., color, distance field, refractive indices, or temporary variables) can be dynamically updated in separate field pools. Field pools can be supplemented if additional volumetric properties are required. If several volumetric properties are frequently accessed together, they can be grouped and stored in a single memory pool to increase the cache hit. For dynamic octree adjustment, we develop memory management based on a linked list. Per generated or freed eight cells, the amount of an allocation or deallocation unit is fixed. When freeing eight cells, we check the front of the memory pool to ascertain whether it is populated from the front. The memory pool containing a root array cannot be deallocated. The depth of the cells and bit flags are separately stored for octree adjustment.

B. Dynamic Octree Representation on a GPU

Our octree mapping from a CPU pool to a GPU texture is one-to-one correspondence in memory space. On the GPU-side, we adopt 2D textures and map the linear index, *I*, to two indices $(\mathrm{t_i}, \mathrm{t_j})$ because of the limited resolution of a 1D texture. Where a dimension of 2D texture is $\mathrm{W} \times \mathrm{H}$, the mapping formula between the two octrees is as follows:

$$\begin{aligned} \mathrm{t_i} &= \left\lfloor \frac{I}{W} \right\rfloor \\ \mathrm{t_j} &= I \bmod W \end{aligned} \qquad (4.1)$$

Since contemporary GPUs provide more than 16,000 texels per dimension, we can supply more than 265M cells with 2D textures. As illustrated in Figure IV-2, parent, child, and the depth are stored to an octree structure, G_p, in a 32-bit integer format. We store RGBA color as a 16-bit float texture. Another 32-bit integer texture, G_3 texture, is a 3-neighbor texture for the indices of neighbors in XYZ direction. Note that G_3 is not used on the CPU. Along with the dynamic update, we construct a G_3 of each cell in parallel from the G_p texture (See Sec. IV.C). Although the G_3 computation is fast on the GPU, it hinders the frame rate in an interactive painting system. In Chapter IV.D.3, we adopt a tactic to reduce the amount of G_3 computation with a lightweight mask calculation and to deliver the mask from the CPU to the GPU.

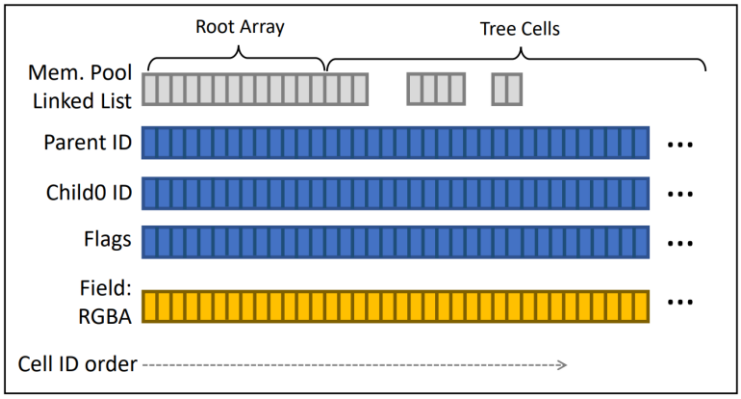


Figure IV-1. A dynamic octree on the CPU

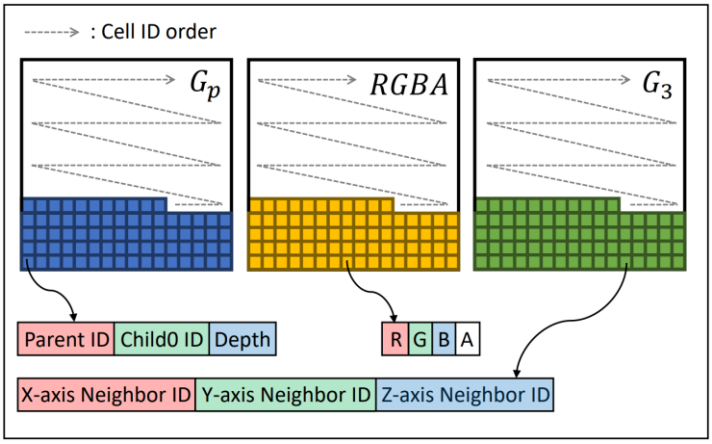


Figure IV-2. A dynamic octree on the GPU

C. GPU-Based 3-Neighbor Computation

Only with the indices of a parent and the first child, rendering performance decreases while the maximum tree depth increases. For example, an octree traversal path from a leaf and its neighbor is 48 in a worst-case scenario based on G_p. Because the octree traversal is a sequence of conditional branches, this long and divergent branching can slow down ray casting on the GPU [59]. For this reason, topologies for directly accessing neighbors (i.e., neighbor linking for octrees [34], [39] or ROPEs for KD-trees [89]) have been researched to accelerate the rendering. As illustrated in Figure IV-3, two cases of the neighbors for given cells, *C* and *D*, the parent-level neighbor, C_0, and the child-level neighbors in D_1 can exist in a 2-to-1 balanced octree except for same-level neighbors (C_1~C_3, D_0, D_2, D_3). Here we illustrate the example as quadtrees with one different-level neighbor for simplicity, but 6 to 24 neighbors can exist in an octree.

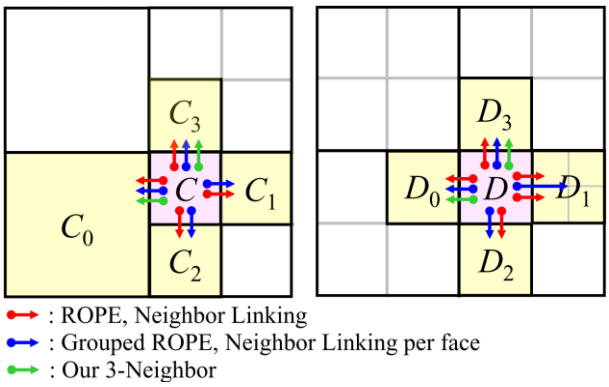


Figure IV-3. The minimum number of neighbor connectivity for a cell

Neighbor linking [34], [39] and ROPE [89] essentially link all the neighbors, which are across the faces of *C* and *D* (the blue connections). To connect the same-level or parent-level neighbors [27] or a group of neighbors per face [70], we can reduce the number of neighbors to six.

Considering that the number of cells in the volumetric painting of moderate size is possibly greater than millions, the total number of neighbors can be over 6 million. This is a huge memory consumption and causes performance degradation due to neighbor computation. Therefore, we decrease the number of neighbors to three.

When considering the parent-level neighbor, $C_{0,}$ and the child-level neighbors in D_1, every cell has six neighbors because only one neighbor exists across each face in +/-XYZ directions. As addressed in Chapter IV.A, eight children have consecutive indices. Thus, three of these neighbors share the same parent and can be immediately obtained from its associated cell index. For example, C_1 and C_2 share the same parent with *C* and can be accessed by adding an offset to the index of C ($\mathrm{C}_1 = C + 1,\ \mathrm{C}_2 = C - 2$). It is the same for D_1 and D_2 with *D*. The other three neighbors may have different parents (C_0, C_3, D_0 and D_3 in Fig. IV-3), and computing such neighbors can be long tree traversals, as mentioned earlier in this section. Therefore, we precompute the indices of the three neighbors, called *3-neighbor topology* G_3.

Using the union of G_3 and G_p, we can quickly discover all the 6 to 24 neighbors, since in $G_p \cup G_3$, the distance between two cells sharing a face is 2 in G_p, or 1 or 2 in G_3. Therefore, finding neighbors in our scheme incurs a low cost, while finding a group of cells [27], [70] can have a distance between neighbors greater than 2 in G_p. In Figure IV-3, the child cells of D_1 that are in contact with *D* can be found easily as $CHILD(D_1)$ and $CHILD(D_1) + 2$, where $CHILD(\cdot)$ denotes the first child. When a ray traverses to D_1 in ray casting, we can quickly identify $CHILD(D_1)$ or $CHILD(D_1) + 2$ using the ray parameter (See Chapter V.A.1). Using G_3, we simplified variable numbers of immediate neighbors to a constant 3. We also reduced the memory required for a fast traversal from the maximum 24 neighbors to three neighbors, while still allowing small constant time access to all 24 neighbors.

D. Dynamic Octree Update

Avoiding sickness, nausea, and postural instability has significance in VR [18], [40], [80] because painting can last several hours. Among factors on those symptoms, a little delay in synchronization between the rendered scene and the head motion is a minimum requirement. However, simply copying an octree to a texture (i.e., 134M cells, which is equivalent to 2.15GB memory) will take 222 milliseconds even with the full bandwidths of a CPU, a PCIe, and a GPU. In this section, we address how we incrementally update the octree on a CPU and dynamically update the octree on the GPU rather than updating the entire octree.

1. Incremental Octree Adjustment on a CPU

Even though we effectively ignore cells that do not intersect with brush stamps, painting an octree is still expensive. A large brush stroke can be applied near a highly-refined region, or a small brush stroke can be applied near a coarsened region. In other words, we may coarsen 24-level leaves to a root-level cell or refine a root-level cell to 24-level leaves in the worst-case scenario on the CPU. To address a sharp change in the depth of cells, we develop a multi-step octree adjustment strategy. Again, we illustrate the example of incremental tree adjustment using a quadtree for simplicity, as shown in Figure IV-4. For a given stroke from a user (a capsule in Frame t_0), we read stroke data from the stroke job queue in a CPU thread separate from rendering. We first paint on the CPU tree without tree adjustment and update on the GPU at Frame t_{1a}. The next step is the tree adjustment stage. We mark cells that should be refined or coarsened based on the intersection status of the cells (outside, boundary, inside at Frame t_{1b} in Fig. IV-4). Outside and inside cells are candidates to be coarsened, and boundary cells are candidates to be refined. All refining candidates are refined, but coarsening candidates are

only coarsened when a coarsening candidate is not a refining candidate and is a parent cell of a leaf cell to satisfy the 2-to-1 balance [10]. After marking the cells, we perform one-level refinement or coarsening per frame. The result of the one-level octree adjustment of Frame t_1 is illustrated in Frame t_2. After one-level tree adjustment, we reflect these changes to the GPU. We repeat this process until no cell needs to be refined or coarsened (Frame t_1 to t_n, where n is the desired depth for a given stroke). In the next section, we propose algorithms for dynamic low latency updating to the GPU while keeping the frame rate constant for VR.

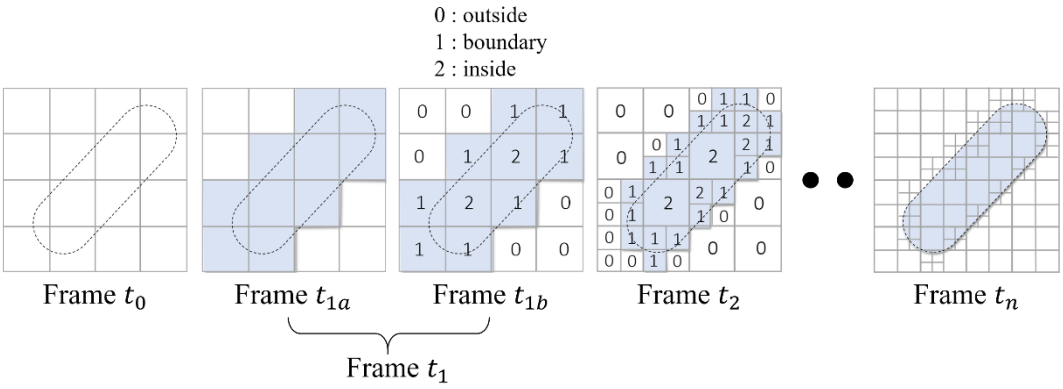


Figure IV-4. Incremental octree adjustment on the CPU

2. Block-Based Staging and the GPU-Side Octree Update

In our system, up to approximately 1,000 cells per stroke would require updates as the stroke diameter is set to be approximately 10 cells for a stroke length of one. Since uploading 1,000 times to the GPU would be prohibitively slow, we use large blocks to reduce the upload counts. Since our CPU-side memory manager maintains a free pool in the last-in-first-out (LIFO) manner, texture memory G_p tends to be filled from bottom to top in the texture space. For example, in Figure IV-5, colored cells at Frame t_2 are generated later than colored cells at

Frame t_1; therefore, they stay relatively right in a CPU-side octree and top in a GPU-side octree. Based on this spatial coherence in 3D space, the CPU, and GPU, we use a *block*, which refers to linear pitched packing that divides texture horizontally. The width of the texture of each block is equal to 16,384 with a relatively smaller height, shown as translucent boxes on octree textures in Figure IV-5.

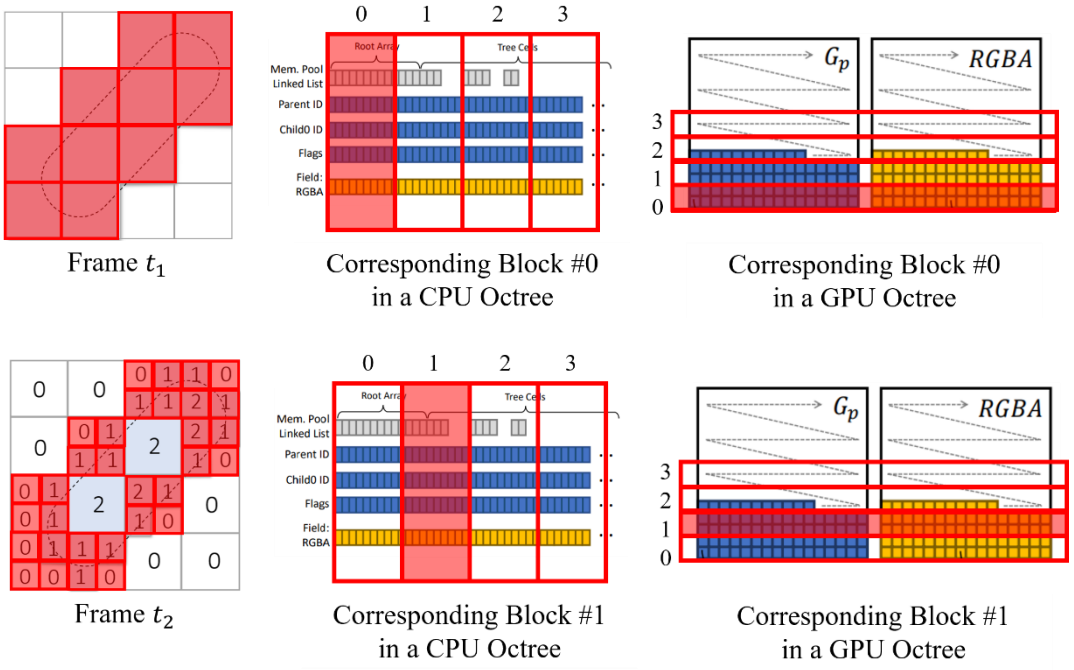


Figure IV-5. The blocks for updating color at Frame t_1 and t_2

If we directly upload updated blocks to the GPU, the whole CPU-side tree would be locked and the painting thread would stall (See Fig. III-1). Rendering would also stall while overwriting the whole GPU-side octree. To avoid this painting and rendering interruption, we improved the simple block-based update for the CPU-side hazard control (Fig. IV-6). Note that we are basing this on the idea that brushing a stroke in a 3D space is local not only in the space but also in the

memory (red cells in the 3D space and the octree on a CPU). We divide the octree into blocks shown as sky blue and orange boxes in the CPU-side octree. Among the blocks, we find a block (orange box) containing updated cells (dotted ellipse). We then copy the block to a staging buffer that serves as an update queue (a box with an orange line). A block copied to a staging buffer is called a staged block. We collect the staged blocks in a separate thread using only small atomic sections during tree adjustments (see Chapter IV.D.4 for discussion on hazards) and queue them in the LIFO queues with upload-to-GPU tasks. Along with staged blocks, we upload a neighbor computation mask (a box with an orange line) that we discuss in the next section. Note that G_3 is not uploaded but is rather computed on the GPU. Finally, we simply upload one block per rendering frame and compute the 3-neighbor based on a neighbor computation mask.

To verify that the algorithms we developed in this section satisfy the hard frame rate and latency requirements, we develop custom benchmark tests. To reproduce painting practice in the real use case, we first load a pre-painted scene and play a pre-recorded set of painting strokes. We fix the head-mounted display (HMD) position to make rendering time nearly constant. Dominant variables are thus controlled variables: updating parameters, such as the number of blocks per frame, block sizes, and the granularity of neighbor computation mask for selective G_3 rendering. The only uncontrolled variables are the CPU thread allocations, the GPU command dispatches, and other minor random system interventions. Our painting stroke sequences are sufficiently long to minimize the impacts of these variables, shown as minor fluctuations in Figures IV-8 and IV-9.

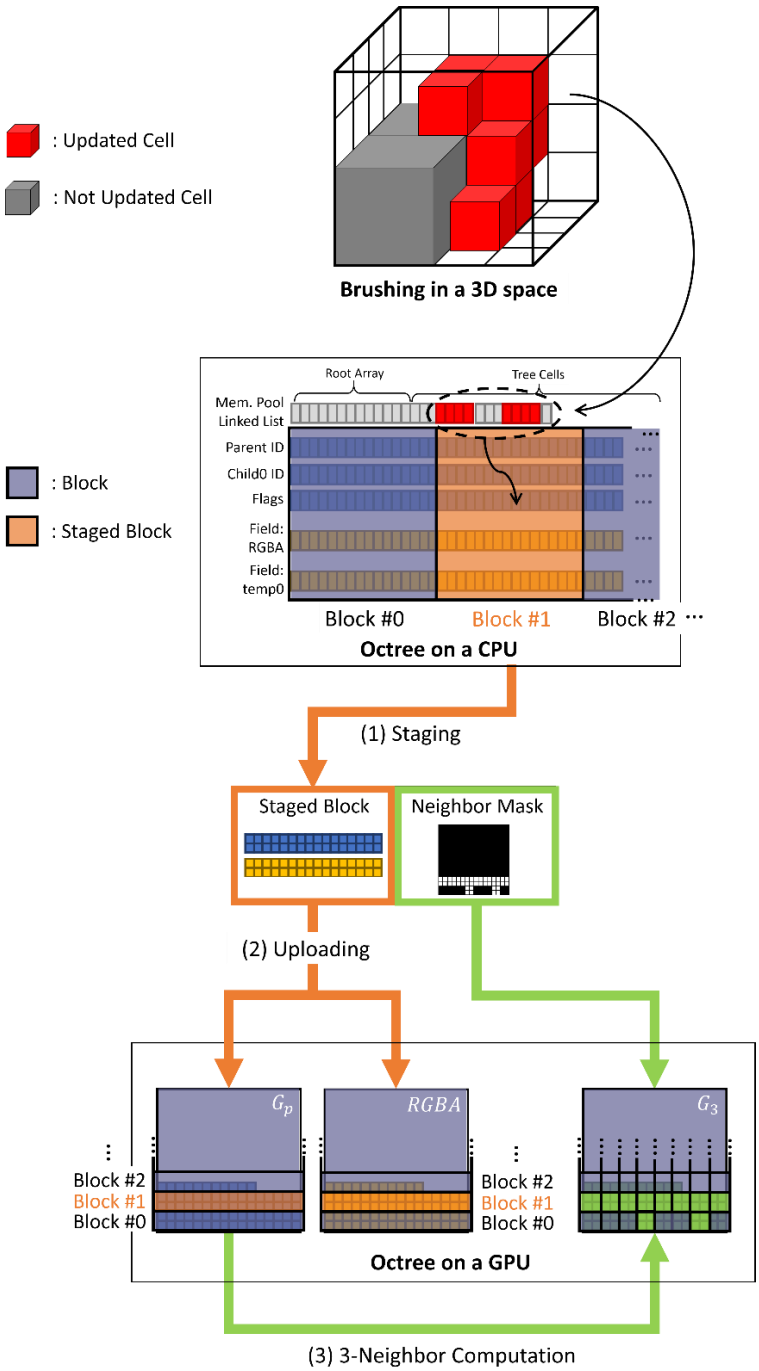


Figure IV-6. Block-based update with a neighbor mask

On the CPU-side, Table IV-1 shows the average staging time and the maximum number of updated cells per second. Staging larger blocks tends to increase the maximum number of updated cells per second and reduces the number of updates. Although larger blocks can update more cells per second, latency tends to increase, and it would be inefficient to upload a large block when only a small number of cells have changed. On the other hand, staging smaller blocks decreases the staging time and provides frequent updates, although bandwidth utilization may be lower. For example, when the size of the block is 16,384×4 (i.e., the number of cells per block is equal to 65K), blocks can be updated 77 times per second, and the maximum number of uploaded cells per second is five million cells. The average bandwidth of the block size under two million cells is less than 5ms, thus, it is sufficient to stage blocks at the refresh rate of 60 or higher, corresponding to a rather long drawing sequence. While a block-based upload improves the frame rate from five frames per second (FPS) up to 22 FPS, this rate is still not acceptable. The next bottleneck is a neighbor connectivity update, which will be discussed in the next section.

Table IV-1. Average staging time, the maximum number of staged blocks and staged cells that can be processed per second

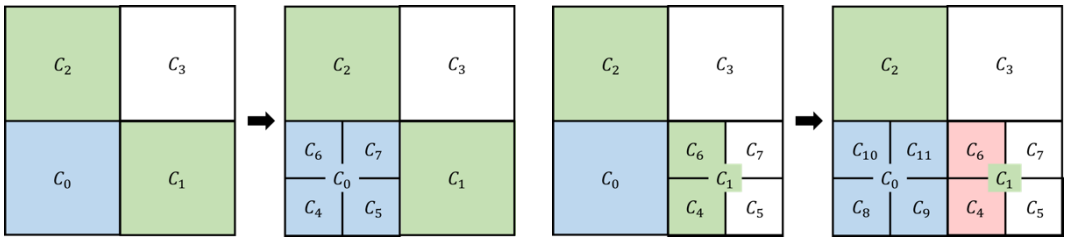
Block Size (cells/block)	Staging Time (ms)	# of Staged Blocks (blocks/sec)	# of Staged Cells (cells/sec)
No Block	585.50	1	36.3M
65K	12.91	77	5.0M
131K	23.36	42	5.5M
262K	37.32	26	6.8M
524K	66.82	14	7.3M
1M	129.76	7	7.3M
2M	248.52	4	8.4M

3. Neighbor Update with Neighbor Computation Mask

Even though a neighbor computation on the GPU corresponds to a simple computation of the graph G_3, the resolution of the texture can be large (16384×8192), which affects the interactivity (See the no mask case in Fig. IV-9). Therefore, we develop a simple and efficient method to dramatically reduce this rendering cost. Once a cell is created or deleted, not only the cell but also its neighbors should be updated in G_3. Since neighbors may not be inside a block that contains the cell, updating G_3 within the block would not be sufficient. One solution would be to have an additional list of expanded blocks for the neighbor update. Since this will increase complexity in the system, we instead propose a simpler approach. We compute a very small mask in the CPU that contains dirty bits, indicating which cells need to recompute their neighbors due to the tree topology change. While staging the cells on the CPU, we upload this small mask to the GPU and perform a neighbor computation only on the cells in the marked area.

A neighbor computation mask is allocated per a block and accumulates dirty bits when a cell has topological changes. Every cell with its own topological change requires updating in G_3. If a cell is allocated, all three neighbors should be newly computed. If a cell is deallocated, all three neighbors should be cleaned. However, not all the neighbors of a cell need updates on G_3. For the neighbors of a cell, changes in G_3 can be routed to the four cases in Figure IV-7, where: (a) the case that a cell C_0 is refined with the same-level neighbor C_1, (b) the case that a cell C_0 is refined with the child-level neighbors C_4 and C_6, (c) the case that a cell C_7 is refined with the parent-level neighbor C_1, and (d) the case that a cell C_7 is refined with the same-level neighbors C_{10}. We assume that $C_0 \sim C_3$ have different parents. In case (a), neighbors (C_1 and C_2) of a cell to be refined (C_0) do not need to update its G_3 because the same-level neighbor

of C_1 and C_2 is still C_0 after a cell refinement. This is the same in (b), however, the neighbors of child-level cells (C_4 and C_6) in C_1 change from C_0 to C_9 and C_{11}. In coarsening cases, neighbors (C_1 and C_2) of a cell to be coarsened (C_7) have no changes in G_3, because C_7 is the same-level neighbor after coarsening. However, if a cell is coarsened with the same-level neighbor as in (d), the neighbor (C_{10}) of C_7 has the neighbor C_0, not C_7. In summary, from the point of view of neighbors of a cell to be refined or coarsened, (1) if a cell is the parent level and will be refined, or (2) if a cell is the same level and will be coarsened, we mark in a neighbor computation mask to recompute G_3 for a neighbor of a cell.



(a) Refinement w/ the same-level neighbor (b) Refinement w/ the child-level neighbor

(c) Coarsening w/ the parent-level neighbor (d) Coarsening w/ the same-level neighbor

Figure IV-7. The cases of the G_3 update after adjusting a cell

We then tested various mask sizes. If the size of the neighbor computation mask is over 64 by 32 (which takes only 256 bytes), the overall frame rate stays around 90 FPS, which is a lower bound of frame rate, as shown in Figure IV-9. We also further tested the degree to which the

size of the blocks affects the frame rates with lightweight neighbor precomputation cost. Figure IV-8 shows that the frame rates remain stable when the number of cells per block is under 524K.

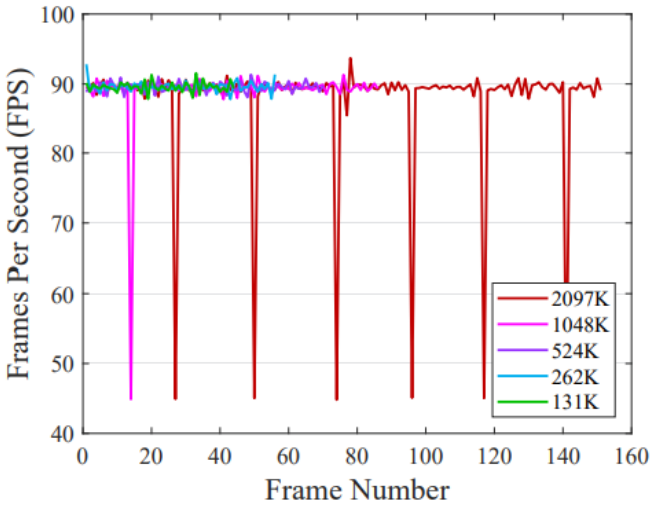


Figure IV-8. The performance results with staged blocks of different sizes

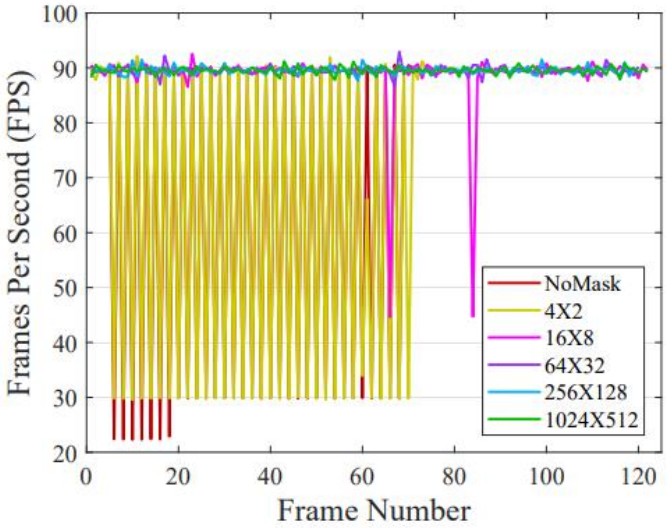


Figure IV-9. The performance results with neighbor computation masks of different sizes

4. Immediate Visual Feedback

To minimize wait state between painting and staging threads, we divide the painting task into the finer grained jobs in the earlier section. Here, we review all steps from painting to rendering for potential hazards in updating: when a stroke is applied to the CPU-side octree, we incrementally adjust the octree by one level in the stroke thread (Figures III-2 and IV-4). This ensures that the CPU-side octree is valid without orphan cells (i.e., cells that are allocated but not linked to their parents) or balancing violations. In the next step, staging blocks may depend on each other due to octree connectivity. For example, if the parent cell exists in block 0 and the child cell exists in block 1, the parent has the index of its first child, but the children do not exist in G_p texture. Hence, dependent blocks may be grouped, uploaded to GPU-side staging buffers, and then copied together in the GPU at the beginning of rendering. However, in our experiment, the GPU-side staging buffer always yielded greater latency. More importantly, the dependency chain between staging buffers can be very large, resulting in much higher latency for accurate rendering results.

Therefore, we experiment with strategies to ignore dependency between staging blocks. As a result, users can see the strokes at a lower latency at the cost of temporal GPU-side violations. First, 2:1 balancing can be violated to 3:1 balancing. This can cause minor temporal visual artifacts in the raycaster that assumes 2:1 balancing. Second, since parent-child referencing is valid only inside a block, some inter-block parent and child indices may be invalid. This causes visible artifacts. Thus, we designed two strategies to reduce these artifacts: aggressive staging and handling invalid parent/child. To experiment with the strategies, we developed a stroke replay system to render frames with and without corruptions. We then computed the number of rendered frames that had mismatching pixels.

By just ignoring inter-staging buffer dependency, corruption occurs in 33.3% of the total frames [Fig. IV-11 (a)]. The corruption lasts up to 450ms. Aggressive staging is to use smaller atomic sections that can temporally break the CPU-side 2:1 balancing to 3:1. However, since aggressive staging does not lock the CPU-side octree with whole one-level octree adjustment, aggressive staging guarantees stable and fast staging time (Fig. IV-10) and the corruption goes away faster [Fig. IV-11 (b)]. The corruption rate falls to 21.7%. Handling invalid parent/child is simply to check whether parent and child mutually point to each other, and if fails, to ignore visiting such cells. This way, the corruption rate goes further down to 3.62% [Fig. IV-11 (c)].

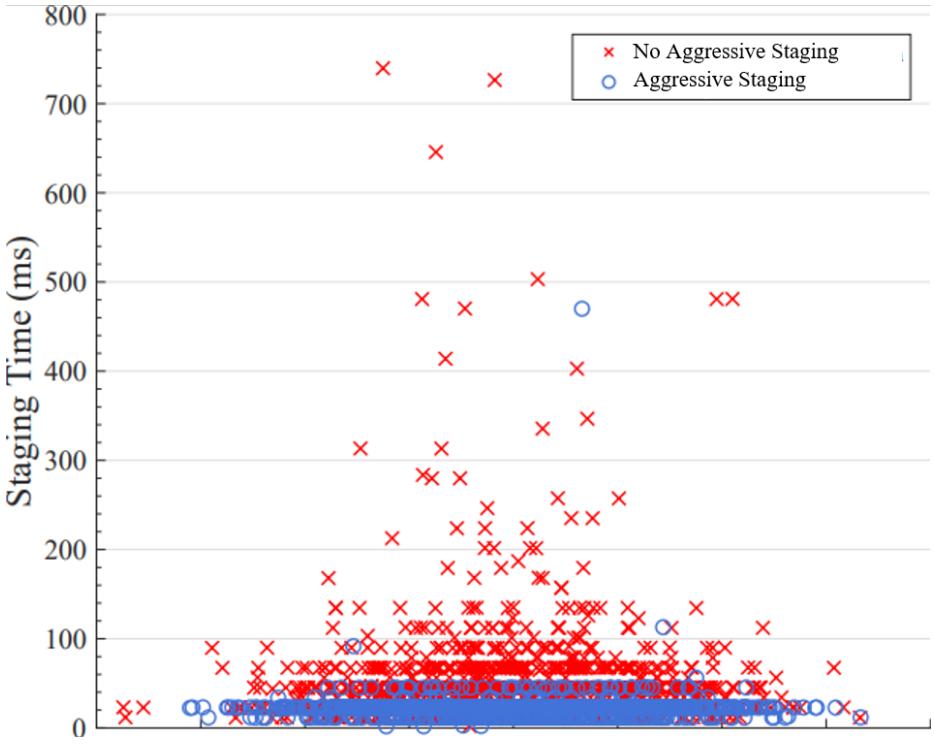
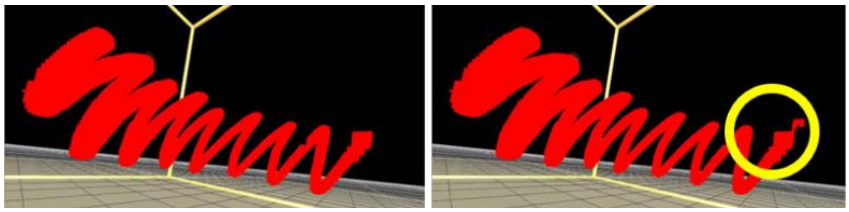


Figure IV-10. The CPU-side staging time with/without aggressive staging

(a) Naively ignoring staging block dependencies



(b) Aggressive staging

(c) Aggressive staging and preventing access to invalid parent/child

Figure IV-11. The frames with corruptions for the same stroke

V. Ray Casting and Color Interpolation on the GPU-side Octree

Rendering 3D volumetric fields poses challenges in accuracy and performance. One viable solution for accuracy involves extracting voxel faces and rendering them through a raster graphics pipeline similar to Minecraft [61]. However, as the number of grids in the non-uniform size increases, the extracted vertex positions, particularly far from the origin, may not be accurate due to numerical error. More significantly, geometric extraction requires a substantial amount of computational time as the number of voxels grows. In this chapter, we explore an accurate approach of ray casting through an octree volumetric field. The proposed cell-local coordinate system is accurate and enables faster rendering with 3-neighbor access (Chapter IV.C); however, we further explore an interpolation-based acceleration technique for stable frame rates in VR.

A. A Cell-Local Coordinate System

Figure V-1. Example of 24-level painting with color-encoded depth

When editing fine detail, users should be able to zoom in to observe the cells that have the highest depth (i.e., 24, as in Fig. V-I). However, the size of these tree cells may be even smaller than the single precision floating point granularity except near the origin. For example, in a VR environment, naively using the floating point for the eye position in a world coordinate will force the head positions to jump toward nearby floating points, and more significantly, the eye distance will be erratic. This leads to extreme discomfort. Therefore, we carefully maintain canvas-to-VR, VR-to-HMD, and HMD-to-eye coordinate transformations to avoid loss of the eye position precision.

Based on the ideas that (1) floating point numbers are dense near the zero, and (2) the world coordinate system is only required for extracting volumetric properties of a given cell, not related to the rendering result, we propose computing the ray starting point in a cell-local coordinate frame and retain the positioning error of the starting point sufficiently small.

$$\mathrm{P_L} = (id, nx, ny, nz) \quad (-0.5 \leq nx, ny, nz \leq 0.5) \qquad (5.1)$$

Our cell-local coordinate system represents position $\mathrm{P_L}$ as the combination of indices of cells and a barycentric coordinate system that has the origin at the cell center with a size of one, as in Equation 5.1. The range of coordinates inside the cell is [-0.5, 0.5]. Consequently, the finest resolution inside a cell is 0.5×2^{-23} in single precision floating point regardless of the size of the cell. If a ray starts from a leaf cell whose depth is 24, its resolution inside the leaf becomes extremely high. In addition, using the world coordinate system, we first compute the index of a cell by traversing based on the position. This can be a long traversal from the root to a leaf in the worst-case. We then obtain volumetric properties of the cell using the index of the cell. On the other hand, our cell-local coordinate system holds the index of the cell and does not need to

traverse the octree from the top to the bottom. We can directly access volumetric properties using the index of the cell in a cell-local coordinate system.

1. Ray Traversal with the Cell-Local Coordinate System

As illustrated in Figure V-2, given a ray and its direction *d*, and a cell-entry point p_i of the ray into the i^{th} cell, we compute the cell traversal distance t_i and the cell-exit point $p'_{i,}$ as well as a neighboring cell containing p'_i. Since our volumetric canvas covers a large space and the cell sizes vary by a large magnitude, using a global coordinate system to calculate p'_i and the ray traversal length *t* may be inaccurate. In contrast, the cell-local coordinate system can produce accurate results regardless of the zoom level. We represent p_i and p'_i with respect to the frame, whose origin is located at the cell center and the size is normalized to one. Using the intersecting face which contains p'_i and the neighbor texture described in Chapter IV.C, we choose the neighbor cell [the $(i+1)^{th}$ cell] to visit and set p'_i to p'_{i+1}. This process is repeated until the ray terminates after accumulating full opacity or exits the canvas.

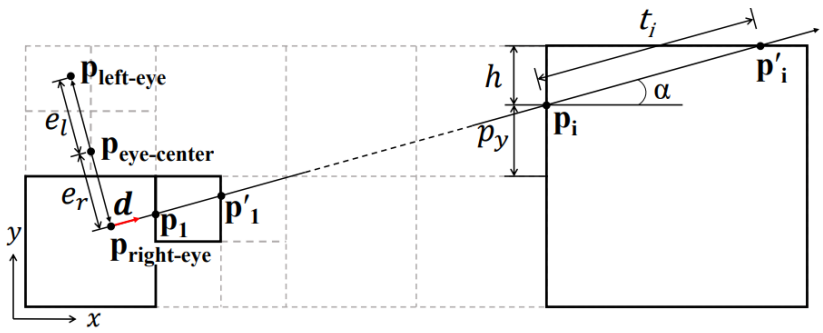


Figure V-2. A 2D illustration of ray casting on adaptive grids

2. Accuracy Analysis

As illustrated in Figure V-2, in the i^{th} cell, if the ray hits the top surface, the ray traversal length t_i is computed as $t_i = (0.5 - p_y)/d_y$, where p_y is the y coordinate of p_i, and d_y is the y component of *d*. The error in t_i will be proportional to t_i. When taking the machine epsilon $\varepsilon = 2^{-23}$ for single precision, x and y for arbitrarily accurate real numbers, and $f(x)$ for a floating-point representation of *x*, $f(x+y) = (x+y)(1+\varepsilon_+)$ with some $\varepsilon_+ \leq \varepsilon$. Similarly, the error in t_i is computed as:

$$f(t_i) = f\left(\frac{f(0.5 - p_y)}{d_y}\right) = \frac{(0.5 - p_y)(1+\varepsilon_1)}{d_y(1+\varepsilon_2)} \qquad (5.2)$$
$$= t_i(1 + \varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2) = t_i(1 + 2\varepsilon_t), \quad \varepsilon_1, \varepsilon_2 \leq \varepsilon$$

Note that ignoring $\varepsilon_1\varepsilon_2$, we have $\varepsilon_t \leq \varepsilon$. We then compute:

$$f(p_i') = f(p_i + f(t_i)) = f(p_i + t_i(1 + 2\varepsilon_t))$$
$$= (p_i + t_i(1 + 2\varepsilon_t))(1 + \varepsilon_3)) \qquad (5.3)$$
$$= (p_i + t_i)(1 + 3\varepsilon_p)$$

for some $\varepsilon_p \leq \varepsilon$, ignoring $\varepsilon_3\varepsilon_t$. Next, we transform the coordinates of $f(p_i')$ to the neighboring $(i+1)^{th}$ cell where the ray continues. This point is computed as:

$$p_{i+1} = f(f(f(p_i')s) + c) = ((p_i + t_i)s + c)(1 + 5\varepsilon_i) \qquad (5.4)$$

for some $\varepsilon_i \leq \varepsilon$, where scale *s* and shift *c* depend on depth and location. Therefore, $f(\widetilde{p_{i+1}}) = \widetilde{p_{i+1}} + 1(1 + 5\varepsilon)$, where $\widetilde{p_{i+1,}}$ is the exact value computed from p_i. Thus, the numerical

error added during the traversal point is proportional to the coordinate values, the number of floating-point operations and ε.

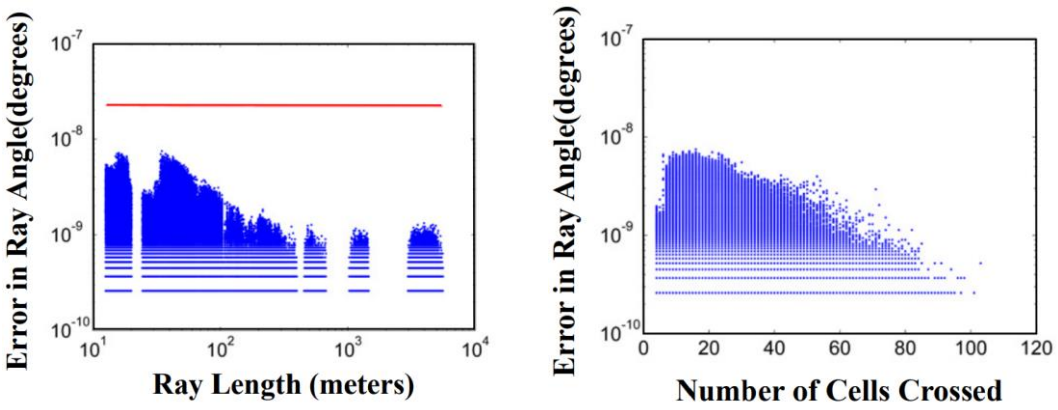
Since we are using a cell coordinate system, each coordinate of p_i is in [-0.5,0.5]. Therefore, the error is always bounded by 2.5ε. In a global coordinate system, the error is bounded by $2.5\varepsilon w_i$, where w_i is the size of the i^{th} cell along the ray. Let e_0 be the error in the eye location (i.e., the error introduced to compute $p_{right-eye}$ in the cell-local coordinate frame). Starting from this initial error e_0, traversing *n* cells results in total error:

$$\begin{aligned} & e_0 + \sum_{i=0}^{n} 2.5\varepsilon w_i = e_0 + 2.5\varepsilon \sum_{i=0}^{n} w_i \\ & \leq e_0 + 2.5\sqrt{3}\varepsilon L \\ & < e_0 + 5\varepsilon L, \end{aligned} \tag{5.5}$$

where L is the ray length. Note that $\sum_{n}^{i=0} w_i \leq \sqrt{3}L$. Thus, by using the cell coordinate system for a ray/voxel traversal, we have shown that the error is proportional to L. Moreover, the error bound in angle $sin^{-1}\left(\frac{e_0}{L} + 5\varepsilon\right)$ does not increase as a function of L, and consequently, the ray does not deviate from the pixel center by more than a small fixed angle regardless the length of the ray L.

(a) Test Scene (b) Errors in Ray Angle (c) # of Cells Crossed

Figure V-3. A ray traversal using cell-local coordinate system



(a) Ray length VS Ray angle Error (b) Crossed cells VS Ray Angle error

Figure V-4. The error in ray angle (degrees)

To verify our analysis, we performed an experiment, as demonstrated in Figure V-3. The maximum level of the octree in our test scene reaches 24. In (b), we visualize the error angle between the final ray location and the initial ray direction, magnified by 10^9 for display in gray. We also visualize the number of cells crossed in gray [(c) in Fig. V-3]. Figure V-4 is the result of our test scene. We show that the screen space ray-deviation from the pixel center, formulated as the ray angle error, does not accumulate during ray traversal. In fact, the error is indeed very small and is relatively larger in the nearby pixels (the maximum ray angle error is 7.5×10^{-9} degrees) due to the initial position error (the position error is 2.5×10^{-7} in L_2 norm). It then slightly decays as L increases. The red line in (a) is the worst-case error bound. In (b), as the number of cells crossed by the ray increases, the stochastic decay increases. The experimental result implies that we can perform the ray casting even on mobile GPUs with only half-precision floats ($\varepsilon = \frac{1}{1024}$) using fragment shaders. We compare the results using the world coordinate and local coordinate in Figure V-5.

(a) World Coordinate (b) Cell-local Coordinate

Figure V-5. The comparison on distortion with two coordinate systems

B. Rendering Acceleration Based on Color Interpolation

In ray casting, a ray traverses 3D space while sampling color from the octree. Major cost factors for ray casting are (1) the number of rays per pixel, (2) the numbers of samples per ray, and (3) the computational complexity of sampling color. In an earlier section, we introduced techniques to reduce the computational complexity of sampling color by directly moving from a cell to its neighbor cell along the ray (Chapters IV.C and V.A.1). We further simply reduce the number of samples per ray by skipping empty cells. If the sampled cell is empty, we traverse toward the root of the octree to find the largest empty cell and resume the ray traversal; otherwise, we traverse to the leaf and move to the next sample position. We can also reduce the number of rays per pixel; however, discontinuity of color increases due to missing pixel color in screen space. In the following section, we first describe the color interpolation of a quadtree and an octree, which generates continuous color vectors. We then propose quadtree-based foveated rendering, which reduces the number of rays per pixel with color interpolation of a quadtree.

1. GPU-Based Quadtree/Octree Interpolation

As we construct a 2:1 balanced octree on the GPU, we also transplant quadtree/octree interpolation from the CPU to the GPU based on Kim et al. [10]. A 2:1 balanced quadtree/octree generates the limited number of local cell connectivity types called *stencils* [10]. Without rotational and reflection symmetry, a quadtree has 16 stencils (Fig. V-6), and an octree has 255 stencils. In other words, we have 16 different cases in the quadtree interpolation and 255 different cases in the octree interpolation.

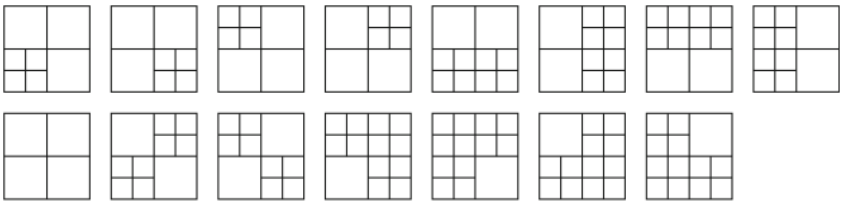


Figure V-6. 16 stencils in a 2:1 balanced quadtree

While interpolating samples in a quadtree/octree, samples to be interpolated (vertices of red dotted boxes and green dotted boxes in Fig. V-7) and their interpolation weights are different for each stencil. We compactly store local connectivity of the samples and interpolation weights in a 16-bit float texture T_w with RG channels and the indices of lookup for T_w in a 16-bit integer texture T_i. Based on G_p, G_3, T_w, and T_i, we rapidly interpolate random color samples in a quadtree on the GPU (Fig. V-7 bottom).

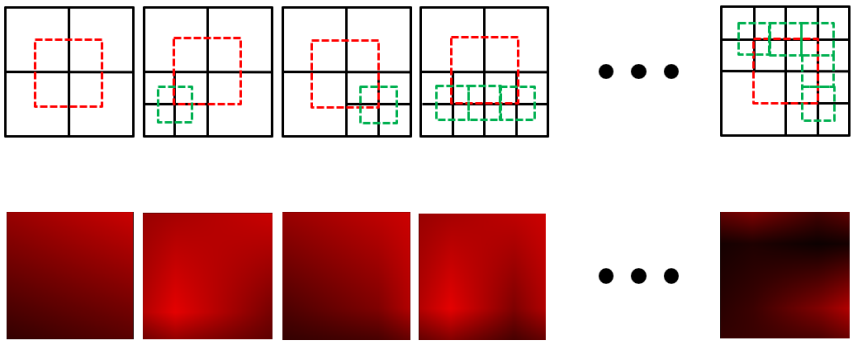


Figure V-7. Stencils in 2D (top) [10] and their quadtree interpolation on the GPU (bottom)

Octree interpolation has only one higher dimension of quadtree interpolation; however, it degrades the performance on the GPU, especially during ray casting. The reason is that 255 interpolation cases aggravate the divergent branching for the SIMD architecture of the GPU [59]. Accessing all the cells to be interpolated and reading the interpolation weights are also not

inexpensive operations. Multiple texture reads degrade the GPU performance even with the help of caching. In the worst-case, octree interpolation on empty cells with eight children significantly lowers the performance, while the resulting color is simply transparent.

Therefore, we accelerate octree interpolation in the following ways. First, we reduced the number of texture-read operations by using the more compact interpolation weight table that prevents multiple access on the same cell. Furthermore, to avoid repeated unnecessary computation, we precompute the stencil type and a flag for the existence of interpolated color per cell on the GPU. These properties are not changed until the octree changes its topology or field and can avoid the aforementioned worst-case. Last, we store T_w and T_i as uniform buffer objects [47] that have smaller storage than texture objects, yet, provide faster read operation. After acceleration, we obtain the result of the octree interpolation 255 stencils with random colors, as illustrated in Figure V-8.

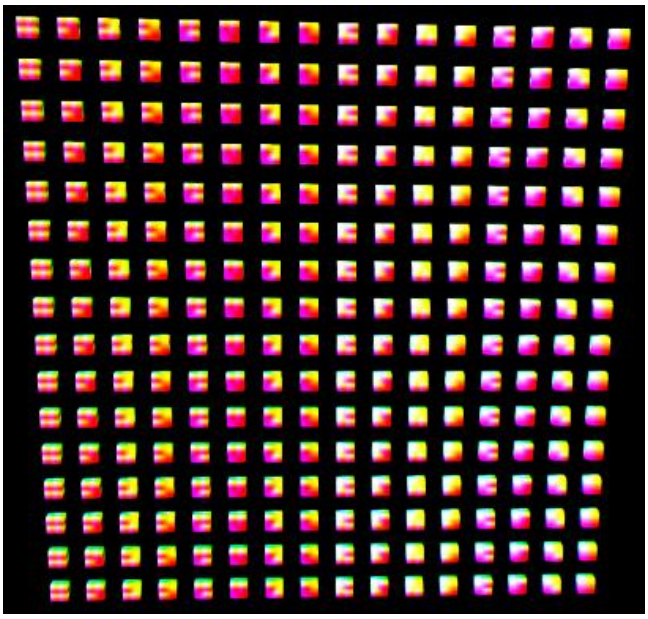


Figure V-8. The octree interpolation for 255 stencils [10] on the GPU

2. Foveated Rendering Based on a Quadtree

Unlike common 2D screens, the position of the eyes is fixed with respect to the screen in a VR environment. Since the view angles of the eyes have limits, the range of visible area in the screen is limited in VR, and the rendering cost can be reduced by ignoring the invisible area. By obtaining further physiological information from the eyes, more cost reduction is also possible. With the availability of an affordable gaze tracker, we can obtain a gaze point, at which a user looks in VR. Human eyes perceive sharply near the gaze point and vaguely far from the gaze point. This degree of perception forms layers from the gaze point, including the foveal region, the peripheral region, etc. By observing the physiological nature of the eyes, we reduce rendering computation in the screen space by rendering the focused area at a high resolution and the rest at a low resolution, a technique known as foveated rendering. Notice the number of rays (red dots) in the full resolution and in the screen space quadtree for the same area in Figure V-9. Foveated rendering greatly reduces the rendering cost, while providing the same rendering quality to users.

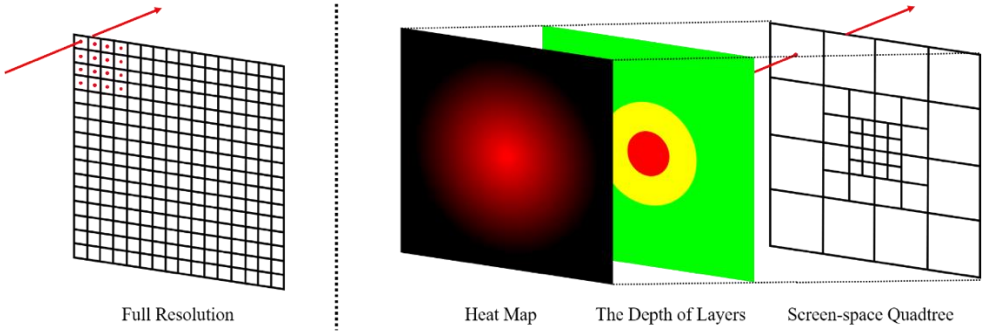


Figure V-9. The ray casting in full resolution and quadtree-based foveated rendering

The core idea of our foveated rendering method is to generate several layers of a rendered screen and composite the layers for the final screen, as with Guenter et al. [9] (Fig. V-9). The main difference, however, is that we represent the screen space using a quadtree (the rightmost figure in Fig.V-9) and blend layers using quadtree interpolation. Based on the gaze point and the area ratio of each layer, we first create a *heat map*, which captures the normalized degree of perception for a human subject. Note that we can add multiply-focused points in a heat map and freely add additional regions of interest. Next, we determine the size of each layer and assign the depth to the layers based on the heat map (the depth of layers in Fig. V-9). We then refine or coarsen the screen space quadtree and perform ray casting in various resolutions based on the depth of layers from low to high-resolution, as shown in Figure V-10. Finally, for the final composite scene, we interpolate the rendering results with various resolutions by traversing the quadtree using screen space quadtree interpolation, as in Figure V-10.

By adjusting the size of the layers and the maximum depth of the quadtree (i.e., the number of layers), we can find a balance between the rendering performance and quality. In our experiment, our quadtree-based foveated rendering is on average 2-4 times faster than full resolution rendering, without noticeable visual difference inside the VR environment (Fig. V-11).

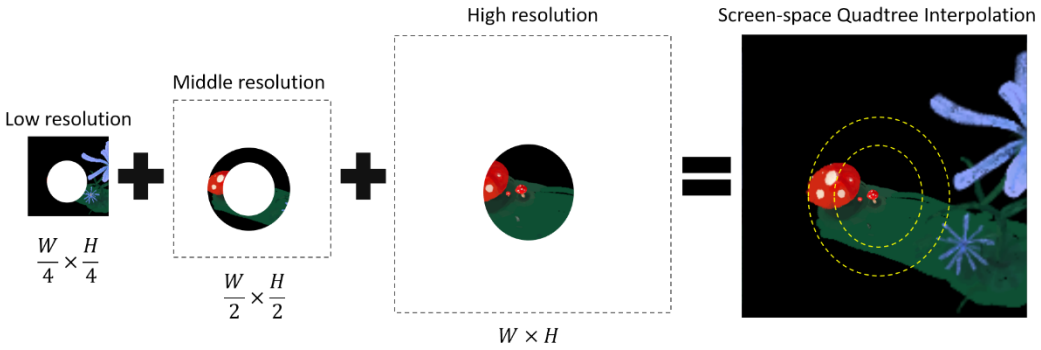


Figure V-10. The rendering and blending layers in different resolutions

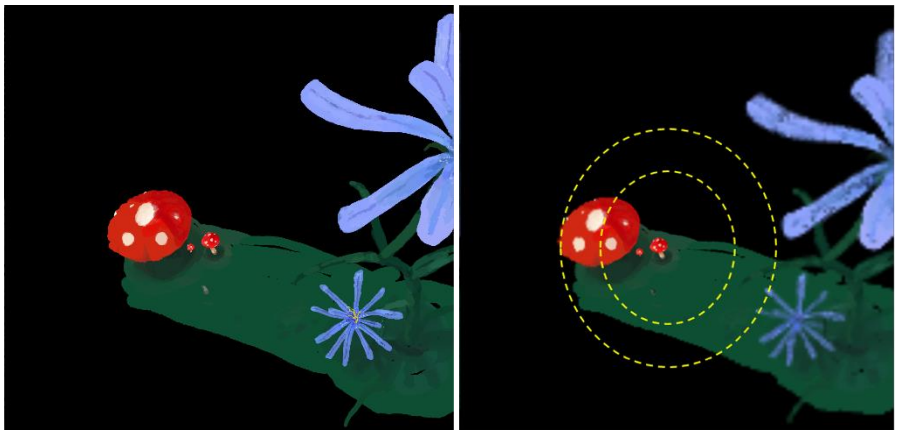


Figure V-11. The full resolution rendering and foveated rendering of the paintings

VI. Painting Interfaces and Volumetric Brush Models

In 2D painting, brushes allow users to paint a large number of pixels. Along with the development of 2D digital painting, a large number of brushing tools have been developed. Most of these tools can be naturally generalized to 3D in our painting framework, and they can be immediately understood by 2D users who can paint a very large number of voxels. In this chapter, we introduce a few examples of such 2D brushes. We also describe volume-specific brush models and filters, which can provide a resolution control, support other shape representations, enhance depth perception, and neaten strokes.

A. Painting Interfaces

Our volume painting system supports conventional painting tools for users, such as color mixing, erasing, recoloring, and color blending. As a choice for painting user interfaces, we use off-the-shelf VR controllers, such as HTC Vive controllers [35]. When the controller is triggered, we record the location of the controller and sub-sample the location at a lower frequency (about 5*Hz*) to paint. Left-handed and right-handed horizontal touch-swipe changes the painting transparency and the brush radius for each. The brush radius is scaled by the zoom level. Thus, to have a larger brush, the user must zoom out. Trigger pressure also affects the brush radius.

We implemented novel 3D color picking interfaces that allow the user to select a color using a left-handed controller while drawing (Figures VI-1 and VI-2). Note that existing color pickers often require a user to use both of their hands, and do not allow changing color during stroking. Our first color picker displays an RGB color cube, and the user can move the controller inside

the cube to change the brush color. As illustrated in Figure VI-1, using only one hand, a user can choose a yellowish color (left), rotate or translate the cube (middle), and change color while applying strokes at the same time (right).

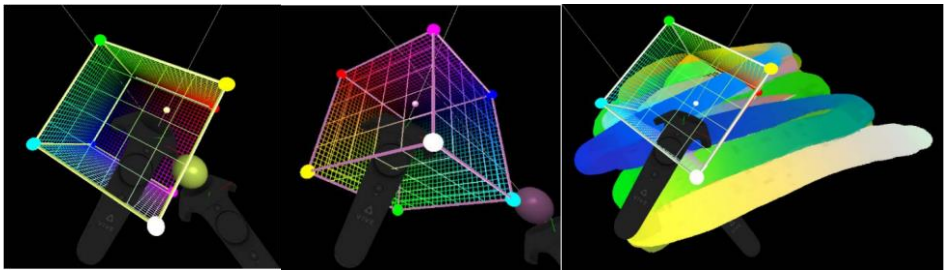


Figure VI-1. A one-handed color picker (RGB color cube)

Since the user may have difficulty finding the proper color and its brightness level, our advanced color picker model is shaped as a cylinder in the HSV color space (Fig. VI-2). A user can choose the color projected on a disk, which is a horizontal cross section of the cylinder, and select the brightness of color by moving up or down inside the cylinder. If a user wants to choose a color from painting, a user may also directly sample the color from the painting by pointing at the desired color and pushing a grab button on the right-hand HMD controller.

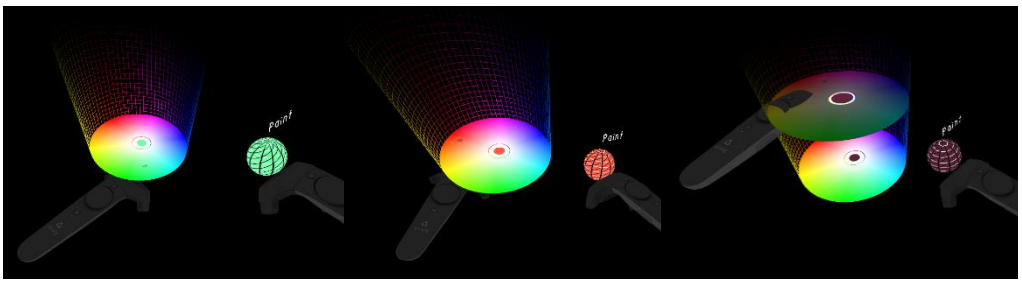


Figure VI-2. A one-handed color picker: HSV color cylinder

In addition to color picking, we provide users with several shortcuts for painting modes and options. A user can quickly choose their paint mode, such as paint, erase, recolor, or color mix by clicking the left/right side of a touchpad on the right-hand controller. However, as the number of painting modes increase, this approach takes time to choose the proper paint mode. Therefore, our system provides a quick menu for paint mode and options using the left-hand controller (Fig. IV-3). For navigating the canvas, painters use both hand controllers in such a way that they grab some points in the canvas and move them around.

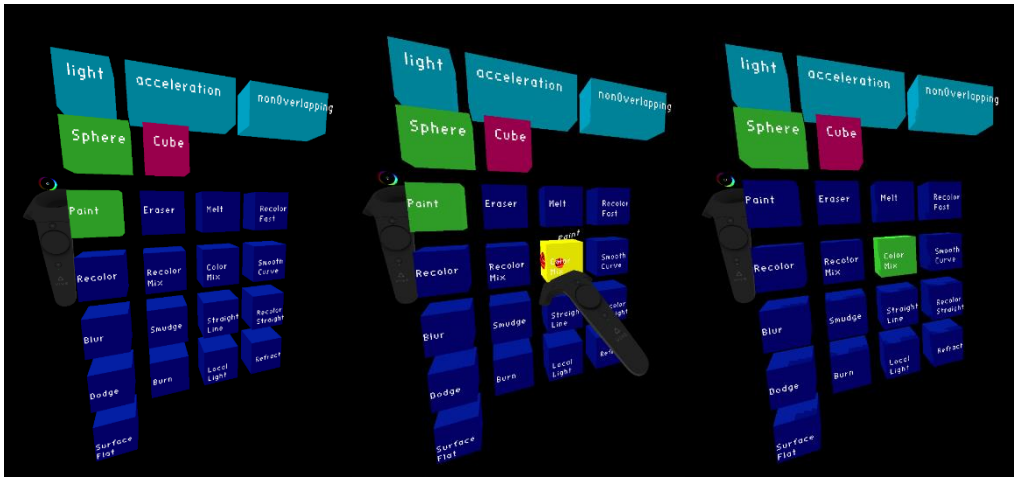


Figure VI-3. A quick menu for paint modes and options

B. Basic Volumetric Brush Models

In 2D painting, users quickly apply brush strokes and react to immediate feedback. Modern CPUs can handle reasonably large brush sizes of up to a few hundred pixels at interactive rates, however, for a larger brush, interactivity begins to diminish. This can be quite restrictive to users because the user must tolerate the delay or switch to a smaller brush to fill in the large area. In 3D painting, this delay can be felt even at a much smaller brush scale. One solution is to use adaptive brush strokes, where we refine the grid only up to a resolution that is sufficient to represent the brush details. For a smaller brush, we further refine the adaptive grid, and for a larger brush, we stop at a certain resolution, which is roughly ten voxels corresponding to the brush radius. The sky in Figure VII-7 was painted with a very large brush (spanning about one kilometer in size), the effective radius of which in the finest resolution is millions of voxels.

In digital painting, pigment deposition from the brush and mixing with the canvas color is a separate topic and is beyond the scope of this dissertation. Fortunately, color flow, deposition, and mixing methods developed in 2D digital painting applications are directly applicable to our 3D volumetric painting. Here, we implement the popular per stroke maximum blending mode (i.e., "paint" mode in Tab. VI-1) found in painting applications [2] to extend 2D digital painting to 3D (Fig. VI-4). We denote t_r as a certain point in run time and t_e as the time when the input stroke ends. By exploiting a temporary RGBA buffer on the CPU-side, we update the color of the cells that intersect with the input stroke in a blank canvas (i.e., temporary RGBA buffer) and then merge color to the painted color (i.e., RGBA pool). This prevents the repeated accumulation of alpha at the cells that interest with more than two segments in the input stroke (black spheres in Fig. VI-5). Note that many other alternatives are available, such as physically-

based pigment mixing using the Kubelka-Munk mode [90], RYB mixing [94], or advanced RGB space color mixing [43].

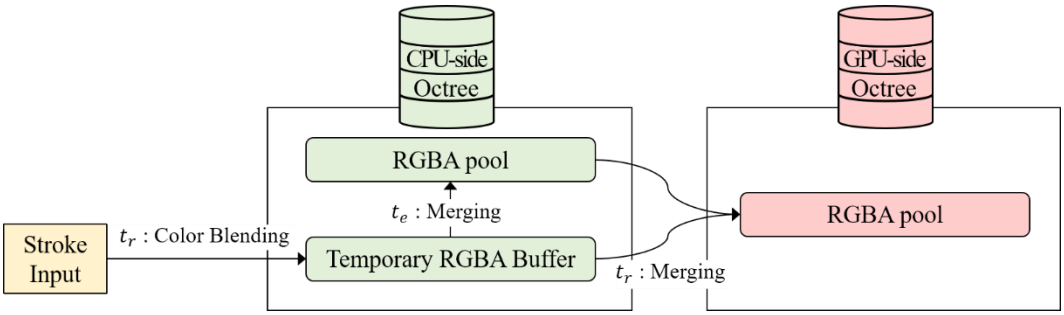


Figure VI-4. A flow of pigment deposition with color buffers

2D brush stamping methods (another orthogonal topic) can also be directly applied to 3D painting. We currently support multiple stamp shapes: a sphere, a cylinder, a box, a cone, and procedural Perlin noise. For spherical stamping, our system supports the sweeping tool, resulting in tapered capsules. We place these tapered capsules to connect the two consecutive samples of a stroke path (the swept stroke algorithm in Diverdi [83], Fig. IV-5).

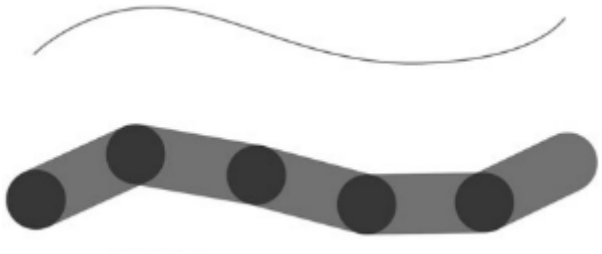


Figure VI-5. An input stroke (top) and its swept stroke (bottom) [83]

Based on Figure IV-4, we provide basic brush models, including paint, erase, recolor, and color mix. We implement a simple color mix method, another common 2D painting practice, as follows. At each sampling point, a brush can pick up color from the canvas and blend it with

the current brush color. As shown in Figure I-2 (b), surface painting systems like Tilt Brush [33] do not have color mixing between strokes. In contrast, color mix automatically generates spatially-varying colors, and this is a popular method used to generate color gradation in a painting, as shown in Figure I-2 (a).

"Voxel dodge" and "voxel burn", which can increase or decrease the brightness of voxels, work like recoloring. For the voxels intersected with a brush stamp, we pick up their original color, update the color by increasing or decreasing the value in HSV color space, and overwrite the original color with the updated one. An interesting point observed during our experiments is that users frequently employ voxel dodge and voxel burn to depict highlight, shadow, and shade, while both brushes were originally introduced to emphasize photos in 2D.

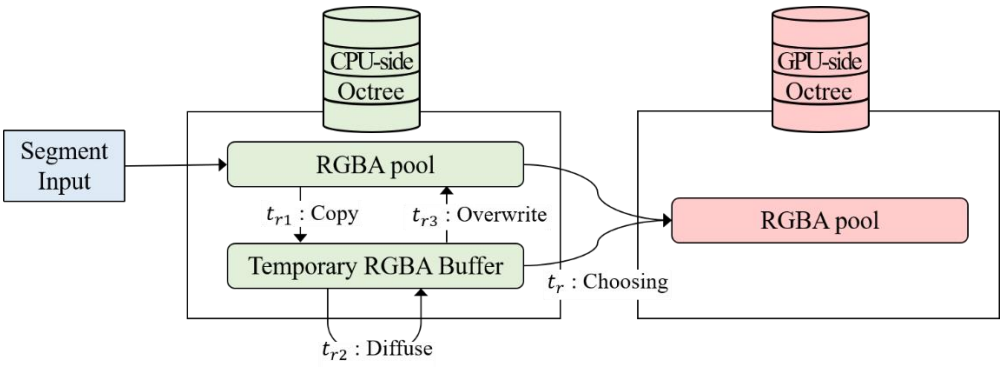


Figure VI-6. A flow of diffusion with color buffers

Using the same color buffers shown in Figure VI-4, we implement other basic brush models based on diffusion (Fig. VI-6), such as voxel blur or voxel smudge. Note that copy at the time t_{r1}, diffuse at the t_{r2}, and overwrite at the time t_{r3} transpire in order. The voxel blur and smudge brushes spread color using tree adjustment on octrees. The RGBA pool on the CPU-side preserves painted color as input for the diffusion step. The temporary RGBA buffer stores

diffused color of cells. At each diffusion step, the output color in the temporary RGBA buffer is overwritten to the RGBA pool. When users draw a stroke with a voxel blur brush (Fig. VI-7), the weighted one-step diffusion is applied to each cell with their nearest six neighbors, as shown in Equation 6.1. The diffused RGB C_d and alpha A_d of a cell can be obtained from the averaged RGB/alpha value of the cell (C_p, A_p) and six neighbors (C_n, A_n) along +/- XYZ direction, where D is the diffusion coefficient.

$$A_d = A_p + D \times \Sigma_{n=1}^{6}(A_n - A_p)$$
$$C_d = \frac{C_p + D \times \Sigma_{n=1}^{6}(C_n * A_n - C_p * A_p)}{A_d} \qquad (6.1)$$

If we allow only 3-neighbor diffusion in the direction of a stroke, we obtain voxel smudge (Fig. IV-8). For rendering, our system decides between the color in the RGBA pool and color in the temporary RGBA buffer (choosing at t_r in Fig. VI-6). In the middle of drawing a stroke, we render color in the temporary RGBA buffer if the diffused alpha is not zero; otherwise, we render color in the RGBA pool.

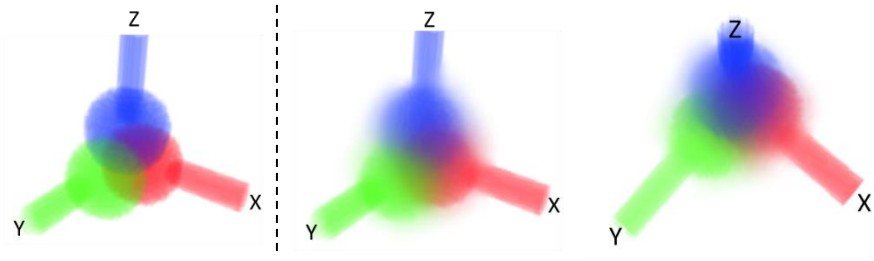


Figure VI-7. The volume strokes before/after applying voxel blur

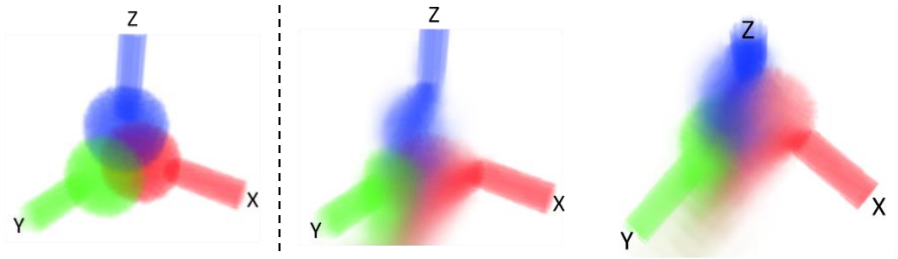


Figure VI-8. The volume strokes before/after applying voxel smudge

Table VI-1. Color blending operations and color merging methods of basic brush models

Brush Mode	Color Blending Operation		Color Merging	
	Per Segment	Per Stroke	Per Segment	Per Stroke
Paint	max	weighted average		O
Eraser	clear	clear		O
Recolor	weighted average	average		O
Color mix	max	weighted average		O
Voxel dodge	weighted average	average		O
Voxel burn	weighted average	average		O
Voxel blur	diffusion	weighted average	O	
Voxel smudge	diffusion	weighted average	O	

C. Volume-specific Brush Models and Filters

1. Voxel Resolution Control

In our system, the brush size determines tree depth so that users do not have to choose the size of an individual voxel (Chapter VI.B). To change the voxel size in painted strokes, users typically adjust the zoom setting to decrease or increase the brush size and then paint over strokes. Although this is a natural workflow of controlling voxel size, more advanced resolution control tools are desired to refine/coarsen voxels or control resolution variations over a large area. We introduce such tools for 3D painting, including voxel melt, voxel merge, mosaic, and view-dependent adjustment. We adopt various strategies for increasing/decreasing the size of voxels based on user behavior. A user develops their artwork from a rough sketch to a detailed painting and locally retouches their painting to complete the painting; in this case, only a small number of voxels is required to increase resolution. On the other hand, a user may decrease the resolution of many voxels at once when the voxel memory reaches its limit. By observing user behavior, we introduce novel brush models for generating finer voxels, as in Chapter VI.C.1.a, and filters for merging voxels in Chapter VI.C.1.b.

a. Voxel-Refining Brush Model

Voxel melt is a voxel-refining brush that smooths paintings by splitting large voxels into smaller voxels and coloring split voxels. Like voxel blur and voxel smudge, voxel melt also exploits diffusing color. One difference is that voxel melt composes the final color from the original color based on the *alpha validity test*, which decides the existence of color. For the alpha validity test, we preserve the alpha channel of all the cells intersected with the voxel melt brush

before color diffusion t_s, in the temporary alpha buffer (Fig. VI-9). We then diffuse the color using the RGBA pool and the temporary RGBA buffer as with voxel blur and voxel smudge. Next, we perform the alpha validity test that compares the original alpha and the diffused alpha reading from the RGBA pool and the temporary alpha buffer. If the diffused alpha of the cells is less than the user-specified threshold of the original alpha, we erase the color of the cell. If not, we restore the alpha from the temporary RGBA buffer after finishing a segment (t_e).

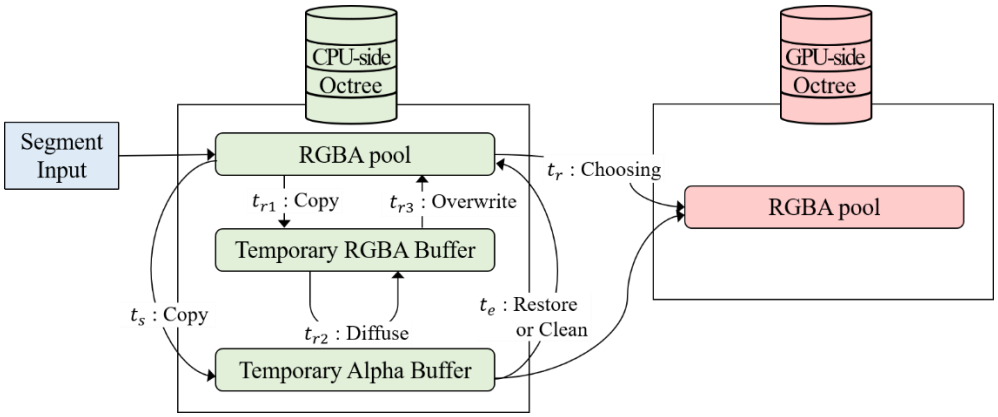


Figure VI-9. A flow of diffusion and alpha test with color buffers

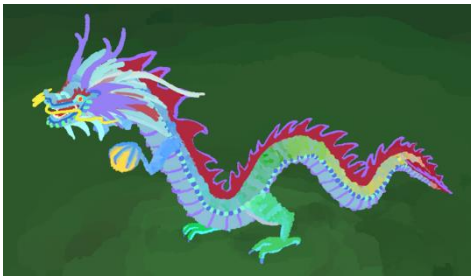
Simultaneously, our system determines whether the cells may be adjusted by computing the desired size of the cell in a view-dependent manner. In the tree adjustment step (Fig. VI-9), we merge the cells if all the siblings of an invalid cell have no color. In case of valid cells, we split the voxels, which are bigger than the desired size of the voxels. After generating children of valid cells, we repeat the color diffusion and alpha validity test without further tree adjustment. In Figure VI-10, we show the example case when a voxel melt brush is desired. A user can smooth the boxy head of an octopus by applying the voxel melt brush. In a similar way, we can design a swollen brush model based on the maximum blending for each diffusion step and color validity test with inverse conditions.

Figure VI-10. Before/after applying voxel melt on the octopus head

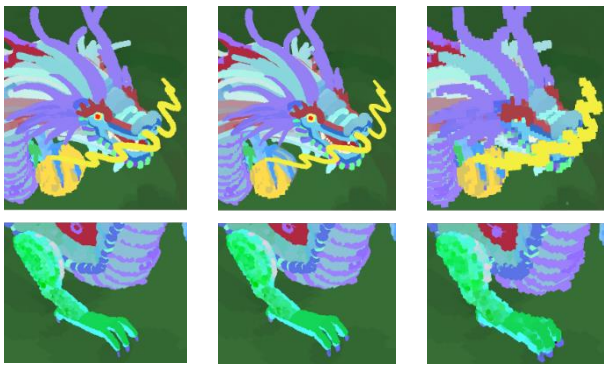
b. Voxel Merging Filters

One solution for decreasing voxel resolution is to regulate the maximum depth of cells in the octree from the beginning. Alternatively, we provide small user-defined areas called *rooms* for an intended appreciation view of the painting (yellow boxes in Fig. VI-11 left). The detailed painting inside the rooms corresponds to foreground objects, and the remainder of the scene serves as a background. When a user defines a room with scale, space, and location, the maximum voxel resolution of entire cells varies based on the distance or visibility from the rooms. Multiple rooms can be defined as well by computing the maximum voxel resolution from the closest room. By defining rooms, users can roughly make a memory plan for their painting and navigate quickly in a large 3D painting. In addition, rooms support a fast startup of 3D painting, thanks to lightweight tree adjustment. A user already hints where they make painting details, therefore, the difference between the size of pre-refined voxels in the rooms and the desired voxels to be painted is small.

Figure VI-11. Example of a painting with two rooms © 2019 Jisu Kim



A painting example © 2018 Jini Kwon



(a) Level=24 Memory=100%

(b) Level=20 Memory=28.95%

(c) Level=18 Memory=14.67%

Figure VI-12. An application of voxel mosaic to a painting example

Another filter used to restrict the maximum depth of the cell during painting is called *voxel mosaic*. Voxel mosaic merges voxels in a painting (Fig. VI-12 top) with a preview at different viewpoints (Fig. VI-12 bottom). Thanks to octree mipmapping, the expected results of voxel mosaic are interactively rendered by limiting the maximum depth in tree traversal on the GPU. Once a user specifies the smallest size (or the maximum depth) of voxels in a region of interest, as well as a blending mode (the maximum alpha/the average alpha blending), voxels smaller than the user-specified size are merged with the blending mode. Voxel mosaic saves multiple and expensive topological changes in the octree when voxel merge is expensive and irreversible. As illustrated in Figure VI-12, a user can also make sure that the intended details of the painting do not disappear after applying the voxel mosaic. While preserving a similar appearance, a user can reduce 71.05% of memory consumption within minutes [Fig. VI-12 (a) and (b)].

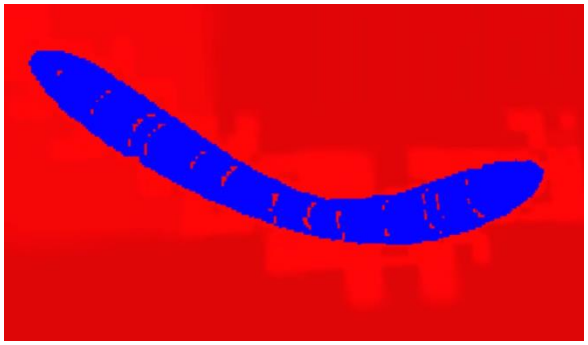


Figure VI-13. The color-encoded cross section of a volumetric stroke

However, simply restricting the depth is not enough to merge voxels efficiently in some cases. In Figure VI-13, we visualize the cross section of a volumetric stroke, coloring the area with opaque color as blue and the area without color as red. Like red voxels in the middle of the blue voxels, the swept stroke algorithm (Fig. VI-5) may produce refined voxels inside a stroke. Some cells can be inside one segment but outside another. While adjusting the octree, the intersection

status of these cells is overwritten and the cells are refined as outside. Or while elaborating artwork, small and opaque voxels become an inner part of the painting, or residual voxels remain after being erased. These types of voxels are undetectable to the naked eye or voxel merging filters above and may gradually increase over time. We address the problem by filtering voxels with a user-specified iso-value, such as 3D erosion or dilation (Fig. VI-14). Based on the distance field generated from the painting, we classify voxels whether they are inside (blue in Fig. VI-14) or outside (red in Fig. VI-14) of the painting with respect to the iso-values. If the outside voxels have colorless siblings or the inside voxels have siblings with the same color, such voxels are merged. With different iso-values, users can decide how much detail they preserve in the painting (yellow circles in Fig. VI-15) and save memory.

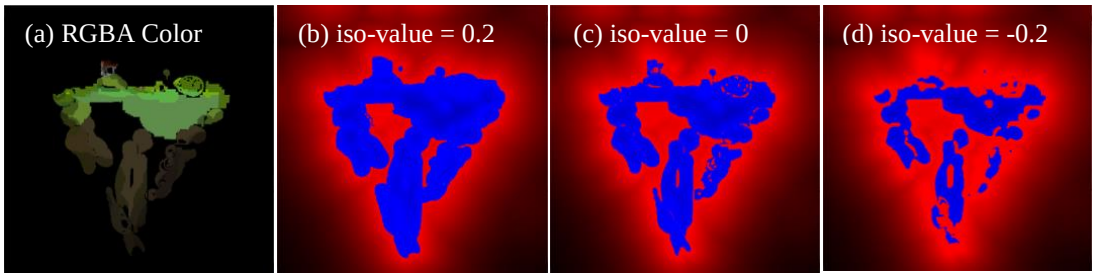


Figure VI-14. The cross section of “Floating Island” and its color-encoded distance field with different iso-values

(a) iso-value = 0.2
Memory = 25.15%

(b) iso-value = 0
Memory = 55.26%

(c) iso-value = 0.2
Memory = 80.11%

Figure VI-15. Results and memory ratio after merging voxels with different iso-values

2. Hybrid 3D Painting and Brush Models

In 3D painting, each painting method has its own advantage or disadvantage over the creation process, painting material, shape support, and user-customizable resource control. If different choices on painting methods are available to users, they cannot only express various styles but also manage the achievability of their design plan. For example, depending on memory budget or desired painting quality, a user may choose to either paint various materials on a volumetric canvas or paint a volumetric hulls stroke with texture maps. Another example is that users can provide an interesting view to their spectators with a semi-transparent volumetric brush (e.g., looking outward from inside smoke) or want to restrict viewpoints, in some cases, to save memory (e.g., looking outward from inside a rock). For these reasons, we provide a hybrid 3D painting that combines both surface and volumetric paintings for general 3D painting. With a hybrid system, a user can paint both volumetric and surface strokes in one scene (Fig. VI-16).

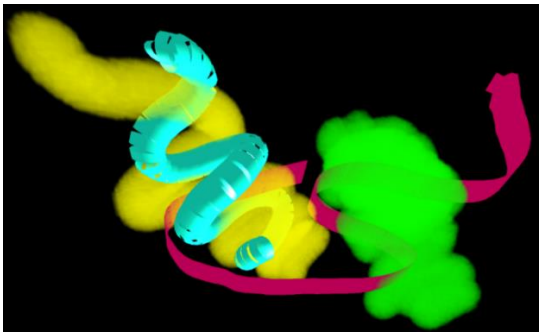


Figure VI-16. Example of strokes using hybrid 3D painting

Painting and rendering engines are independent for two different methods in hybrid 3D painting. One application of hybrid 3D painting is to use a surface painting engine for fast sketching and use volumetric painting for color painting (Fig. VI-17). Thanks to inexpensive mesh

generations and shading in the surface painting engine, a user may quickly sketch with depth perception. During the painting phase, a user can paint color in the 3D space without occlusion or z-fighting. Since rendering engines are separate, a user can exploit two painting engines like two different layers.

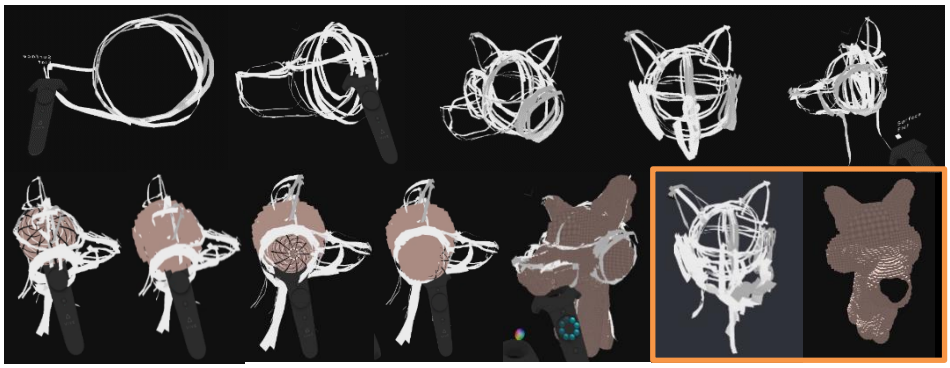


Figure VI-17. Fast sketching with surface painting
and color painting with volumetric painting

To take advantage of the interaction between the two painting methods, we invent a *smooth volume brush*, which postpones the creation of a volume stroke like a 3D printing pen (Fig. VI-18). In volumetric painting, a user experiences difficulty in drawing a straight line or a smooth curve, as the volumetric brush can easily create squiggles due to imprecise hand control and unstable controller tracking (Fig. VI-20 left and Fig. VI-20 left). Moreover, the position change in volume segments during painting may lower the performance due to frequent modification on octrees. We address this problem by using the smooth volume brush.

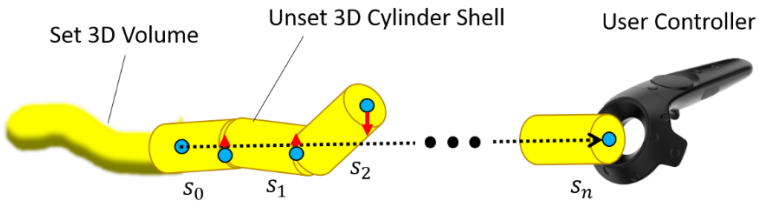


Figure VI-18. A modeling of a smooth volume brush

A smooth volume brush first generates 3D cylindrical shell segments from the user input to visualize the positions of segments using a surface painting engine (unset 3D cylinder shell in Fig. VI-18). In this stage, a user may modify the position of the 3D shell segments by meticulously moving the input controller. 3D shell segments are linearly interpolated by translation between the starting point of the first segment and the end point of the last segment. Similar to the solidification progress, the ratio of the delayed movement to the interpolated position is inversely proportional to the elapsed time from generation. That is, the older segments move shorter than the new ones. After a user-specified period passes, all the 3D shell segments are completely solidified into 3D volume segments using the volume painting engine (set 3D volume in Fig. VI-18). Thus, a user can control the solidification progress by specifying the scale of the movement and the time to complete solidification.

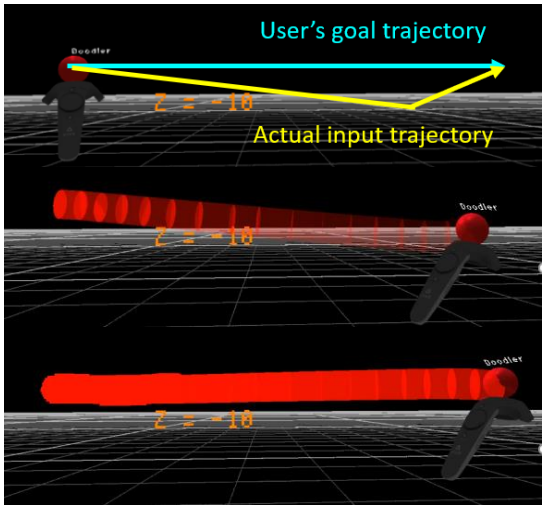


Figure VI-19. The actual use of a smooth volume brush

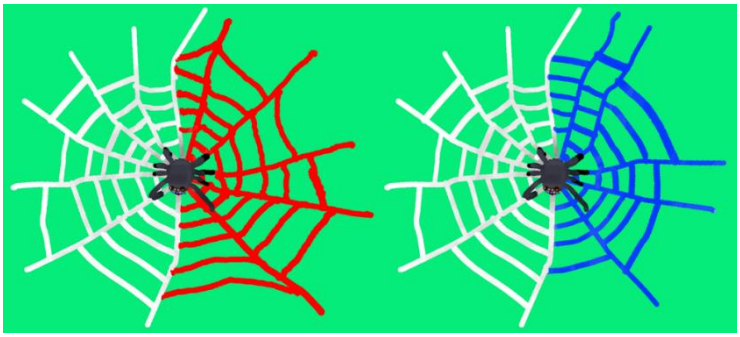


Figure VI-20. The comparison on paintings without/with a smooth volume brush

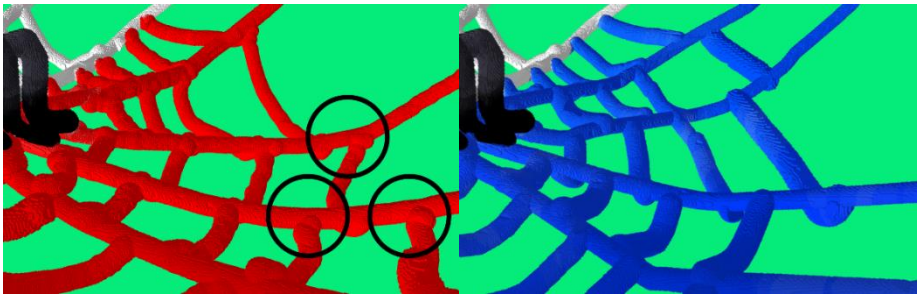


Figure VI-21. The zoomed comparison on paintings without/with a smooth volume brush

As illustrated in Figure VI-20 right and Figure VI-21 right, the smooth volume brush model aids quick painting with straight/curved lines for users. A smooth volume brush model is simple enough for novice users in that it provides a "squeezing paint" interface and an automatic stroke revision with simple 3D geometric shells. This reduces squiggles due to imprecise hand control and unstable controller tracking (circle in Fig. VI-21 left) and increases the painting quality (Fig. VI-21 right). In 3D painting, stroke alignment (e.g., two crossed lines meeting at a point or two parallel lines) is more difficult than in 2D painting due to the lack of depth perception. With a smooth volume brush, users would rather align their strokes using a preview of the 3D cylinder shell than repeatedly draw lines at similar locations multiple times. As a result, users can easily match radial lines and spiral lines of the spider web at the intersection points in Figure VI-21.

VII. Results and Discussion

We use C++ programming language with Visual Studio 2017, OpenGL 4.3, OpenVR, and the Grizzly library [10] under Windows 10 OS to implement our system. Our hardware setup includes Nvidia GTX Titan Xp GPU and an Intel Core i7-4790 CPU with 16GB RAM for rendering computation and HTC Vive for interfacing immersive and personal VR environments.

A. Painting Results

We invited digital painting users and novice users to produce volumetric paintings. The "Island" (Fig. VII-3) represents a good example of an extension of 2D painting to 3D. The user painted the scene with three distinct locations and a large sky background. Semi-transparent objects, such as clouds and smoke, and other detailed objects like wires and lamb, also show the benefits of our system. The user uses a recolor mode and a color mix mode for adding rough shades on buildings and on the island. Similar to Island, the user explores a much larger painting space in "Flying Dragons" (Fig. VII-7). Rougher and more colorful shades on the dragon's body-surface were used, however, fine details on the eyes, teeth, and horns were maintained.

Our system also shows development possibilities for serious 3D digital painting tools. Each image of "A Sneaker" (Fig. VII-1) seems like the results of a 2D digital painting, except that the painting can be appreciated from any viewpoint. Another example with various sizes of objects is Figure 14. In "Spring Concert" (Fig. VII-2), the user not only produced a digital painting, but also developed a story with a full 3D scene, including retouched chipmunks, tiny bees with a detailed score, and a translucent spring haze. From this aspect, we anticipate that many 2D digital users may extend their existing techniques to 3D. As with Spring Concert, the

user creates a funny story with a twist exploiting 3D space in “A Detective Dog and A Turtle Killer” (Fig. VII-4).

Novice users who do not have any experience with 3D modeling can rapidly create their 3D artwork using our volumetric brush models (Table VII-1). Novice users have difficulty finding proper colors to express brightness or shadow because they typically do not have training experience that strengthens their color sensibility. Using voxel dodge/voxel burn brushes, novice users can easily paint highlights, shading, or shadows without picking colors in “An Eye” (Fig. VII-10). “A Spider” (Fig.VII-11) only took a total of 15 minutes to finish using the smooth volume brush without requiring heavy training on the tools. Painting times vary according to an individual’s painting style, the fineness and scale of the artwork, and interval times for changes in the painting plan. However, most of the users update $29K - 287K$ of voxels per minute.

Table VII-1. Memory usage, the number of voxels, and painting time of artwork

Painting Name	Memory (GB)	# Voxels	Painting Time	# Voxels/ min.
Spring Concert	1.96	130,746,223	56 h.	38,913
A Turtle Killer	1.60	71,297,055	40 h.	29,707
A Sneaker	1.36	90,959,591	24 h.	63,166
Island	1.28	85,510,671	-	-
Floating Island	1.08	36,266,863	9 h.	67,160
Flying Dragons*	0.808	52,557,063	11 h.	79,632
Snow Mountain*	0.796	51,834,039	3 h.	287,967
Nature	0.375	24,433,799	-	-
An Eye*	0.217	9,452,719	40 min.	236,318
Cupcake Monsters*	0.162	7,068,215	1 h. 30 min.	78,536
A Spider*	0.089	3,895,383	15 min.	259,692

* Novice users’ painting

Interestingly, the users remarked that they needed to change the way they had previously perceived painting and that they needed to step away from the familiarity of perspectives on the 2D canvas. The second part of the remark is very interesting, as it appears to be the result of being able to paint on a very large 3D canvas. We believe future explorations with users will reveal how perspectives can be painted on 3D canvases. For example, recoloring remote mountains from a foreground location would be an interesting approach.

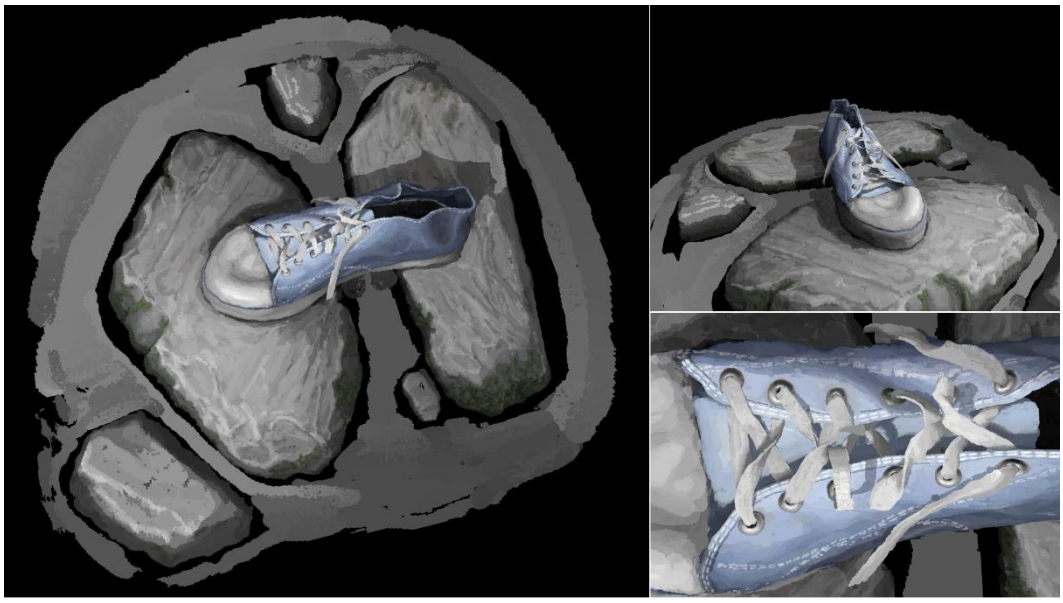


Figure VII-1. "A Sneaker" from various viewpoints © 2018 Daichi Ito

Figure VII-2. "Spring Concert" from various viewpoints © 2018 Jini Kwon

Figure VII-3. "Island" from various viewpoints © 2018 Daichi Ito

Figure VII-4. "A Turtle Killer" from various viewpoints © 2019 Jini Kwon

Figure VII-5. "Nature" from various viewpoints © 2018 Jini Kwon

Figure VII-6. "Floating Island" from various viewpoints © 2017 Jaehyun Kim

Figure VII-7. "Flying Dragons" from various viewpoints © 2017 Yunhyeong Kim

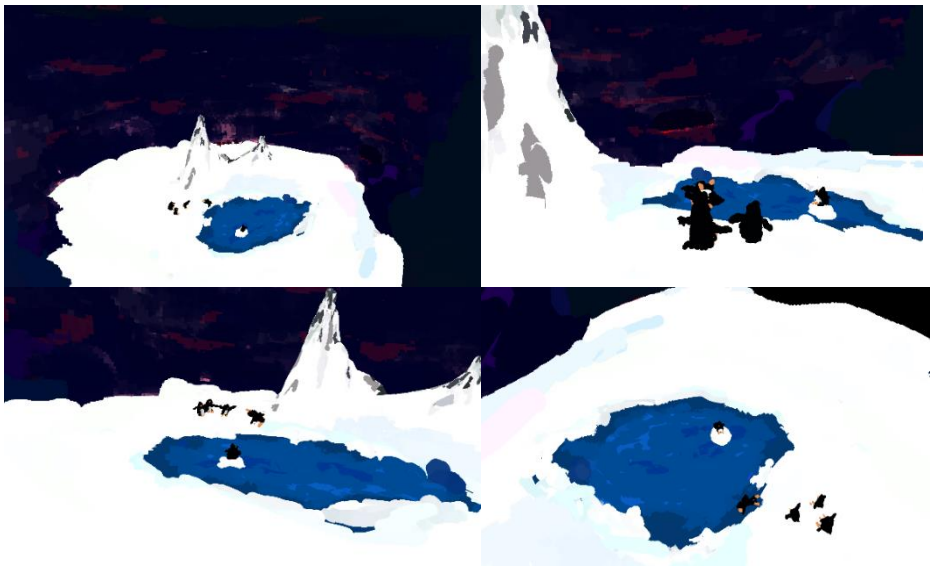


Figure VII-8. "Snow Mountain" from various viewpoints © 2017 Daeun Song

Figure VII-9. "Cupcake Monsters" from various viewpoints © 2019 Daeun Song

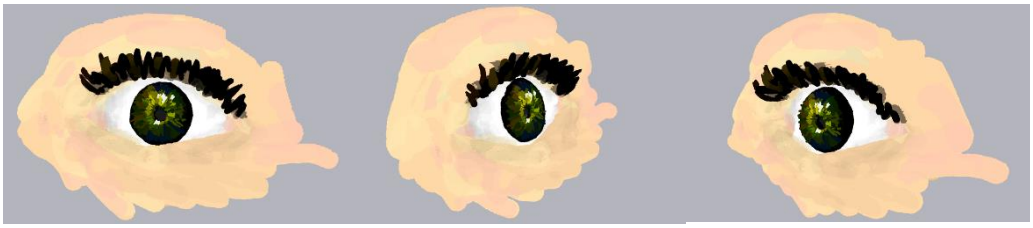


Figure VII-10. "An Eye" from various viewpoints © 2019 Yunhyeong Kim

Figure VII-11. "A Spider" from various viewpoints © 2019 Yunhyeong Kim

B. Qualitative Comparisons of Using Dynamic Octree

For volumetric painting, we must locate parent cells that contain a brush, and from these parents (not from roots) we then refine, coarsen, or compute blending. For rendering, we visit cells from a child to its neighbor using our novel memory-efficient neighbor representation and dynamic and incremental tree update strategy. Thus, volumetric painting application does not require traversal from a root, and shallow tree benefits [4], [48] are minimal. In addition, shallow N-trees would require selecting the depth and the tile size N (per level or cell) at an early stage. This priority requirement is difficult for users to understand and modify at a later stage. We claim that the simple octree, which has uniform adaptivity and painting quality in the canvas regardless of zoom levels, is a more viable approach to volume painting.

Incremental dynamic update, while maintaining high rendering frame rates, is essential for volume painting, yet, not a key requirement in existing dynamic tree update techniques. In simulation problems [79], [81], trees are updated globally as they are often adjusted based on velocity, smoke, proximity to the liquid surface, or details on the liquid surface. Moreover, rendering is not required, and the frame rate is less demanding than the VR painting applications. In other studies [4], [14], heavy updating was allowed in a GPU-only dynamic tree, where the CPU cannot be involved in painting. Because Octomap [5] does not require immediate visual feedback, a low latency visualization method has not been studied, thus, Octomap cannot be used for volume painting.

C. Limitations and Future Work

While painting a large stroke over a detailed complex area, a large number of octree cells should be deleted. In this case, while the frame rate is still constant, the delay can be quite large. An indicator would notify the users of a heavy-weight operation. Our system may also eventually suffer from out-of-memory. Although the memory size available in modern GPUs is increasing over time, users can indeed consume all the GPU memory. This can be greatly relieved by using the topology cleaning operations introduced in Chapter VI.C.1.b., however, more memory plan-ahead interfaces would be required. Because users spend a long time wearing a VR headset, the weight of the headset is currently a dominating discomfort factor. One way to reduce such discomfort is to move some tasks, such as recoloring, from the VR to a conventional 2D monitor.

Error analysis suggests that ray casting can be performed in the half-float precision of $\varepsilon = \frac{1}{1024}$. Therefore, we plan to examine the performance and accuracy implications of using half-precision for ray casting and tracing applications. In color blending, the distance of the ray inside the cell and the alpha value are used to compute the opacity of a color. Currently, color blending formulas with two independent variables show sudden changes in opacity near 0 and 1. By analyzing the independent variable patterns, in the future, we can model more accurate color blending formulas at a high resolution.

In digital painting, having multiple layers is indispensable to avoid an unintended touch to finished parts and apply various effects per layer. The support for multiple volumetric layers will be a challenging topic for future research. Currently, a user must begin painting on an empty canvas. Constructing 3D adaptive volumes from uniform 3D volumes, voxelization with polygonal models [15], [21], [62], [63], [72], or 3D reconstruction and voxelization with

multiple 2D images [64], [69] can reduce user workload and support a fast start-up of 3D painting. Editing functionality, like undo/redo, copy/paste, select, and history look-up, is also essential for users to reduce work time. For undo/redo operations, one possible solution is to first provide the rendering results of the editing tools and update octree after users confirm the operation. For example, if we store texture blocks before/after applying strokes, we can show the rendering results of an undo/redo without changing the octree on the CPU. For selection tools, choosing voxels based on various criteria in addition to color expands user's options [47], [80]. In this case, we can support volume segmentation by storing each segmented volume to different layers or allocating a new scalar field pool for labels of segmentation.

Our system is equipped with several basic brush models: volume-specific brush models, filters, and brush models in hybrid 3D painting. To increase the quality of painting and support various styles, more volumetric brush models and filters should be further explored. Typically, many brushes and filters in 2D painting are easily extended to 3D without significant changes in modeling designs. However, some tools can be computationally expensive in 3D painting if we directly reuse their 2D modeling designs, due to heavy tree adjustment. Therefore, in the future work, we would like to study lightweight 3D brushes and filters for such tools. Similar to proposed approaches in Chapter IV and Chapter VI, we can adopt incremental or 'rendering first/update later' strategies for lightweight 3D tools; for example, a color-fill operation with 3D contouring exploits distance fields to decide the interior and exterior of the painting, and incrementally update the color of voxels near the cursor. As with updating color, various physical properties, such as gravity, viscosity, or heat, locally simulated brushes and advanced brush models for the complex design of physical properties [67] might result in interesting painting styles. More diverse volumetric brush models, such as stylized brush models, including watercolor [17] and Impasto [90], and sculpting brush models, including 3D warp brush [93] and procedural brushes [68], will be interesting ways to create various styles of 3D digital art.

Various styles of rendering [20], including isosurface rendering based on our GPU-side octree interpolation, will be part of our future work as well.

Finally, we discussed a volumetric painting system and its brush models in the scope of a new medium for 3D painting. Other than 3D painting, many research areas (e.g. modeling [5], [60], [88], film and game design [19], [23], [85], simulation [30], [48], [79], robot navigation [5], scientific visualization [12], [13], [27], [36], 3D printing [79], and medical imaging [27], to name a few) employ 3D volumes for representing data. Our system provides both authoring and visualizing tools for 3D volumes, therefore applying volumetric painting system to those areas would be interesting.

VIII. Conclusions

In this dissertation, we proposed the first volumetric painting system that can paint volumetric strokes, mix colors, recolor existing strokes, erase, and depict semi-transparency at a very large scale and with high details in VR. To achieve this goal, we used octree with high-depth and performed ray casting to render the volume. We used the CPU to implement dynamic tree adjustment and proposed low latency update methods that keep the rendering frame rate highly interactive. We showed that small staged blocks and neighbor computation masks maintain the system performance, while the latency and artifacts were well suppressed. To reduce the memory footprint, we showed that three neighbors per cell are sufficient for efficient neighbor access in an octree. For rendering, we provided a ray traversal error bound using posterior error analysis and verified the bound with experiments. To accelerate rendering, we extended the CPU-side quadtree/octree interpolation to the GPU and devised quadtree-based foveated rendering.

For 3D painting for non-expert individuals with various style support, we extended common 2D brushes to 3D, such as paint, erase, recolor, color mix, blur, smudge, dodge, and burn. We also proposed volume-specific brushes and filters, which reduces repetitive painting works: voxel melt, room, voxel mosaic, and voxel merge based on iso-values to control the resolution of many voxels and user-driven memory management. We proposed other volume-specific brush models based on the hybrid 3D painting system, which is a combination of surface painting and volumetric painting. Hybrid brush models solve problems of depth perception [37] and stroke neatening in volumetric painting.

Bibliography

[1] A. Andre, S. Saito, “Single-view Sketch Based Modeling,” In *Proceedings of the Eighth Eurographics Symposium on Sketch-Based Interfaces and Modeling (SBIM'11)*, pp. 133–140, 2011.

[2] Adobe Systems Incorporated, “Adobe photoshop user guide,” http://www.photoshop.com/, 2016.

[3] A. D. Gregory, S. A. Ehmann, M. C. Lin, “intouch: Interactive multi-resolution modeling and 3d painting with a haptic interface,” *Proceedings IEEE Virtual Reality 2000*, pp. 45–52, 2000.

[4] A. E. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, J. D. Owens, “Glift: Generic, efficient, random-access GPU data structures,” *ACM Transactions on Graphics (TOG)*, vol. 25, no. 1, pp. 60–99, 2006.

[5] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, W. Burgard, “Octomap: An efficient probabilistic 3d mapping framework based on octrees,” *Autonomous Robots,* vol. 34, no. 3, pp. 189–206, 2013.

[6] A. Shtof, A. Agathos, Y. Gingold, A. Shamir, D. Cohen-Or, “Geosemantic Snapping for Sketch-Based Modeling,” *Computer Graphics Forum*, vol. 32, no. 2pt2 , pp. 245–253, 2013.

[7] B. Adams, M. Wicke, P. Dutré, M. H. Gross, M. Pauly, M. Teschner, “Interactive 3D Painting on Point-Sampled Objects,” *Symposium on Point Based Graphics(SPBG)*, pp. 57–66, 2004.

[8] B. Dado, T. R. Kol, P. Bauszat, J.-M. Thiery, E. Eisemann, “Geometry and attribute compression for voxel scenes,” *Computer Graphics Forum*, vol. 35, no. 2, pp. 397–407, 2016.

[9] B. Guenter, M. Finch, S. Drucker, D. Tan, J. Snyder, “Foveated 3D graphics,” *ACM Transactions on Graphics (TOG)*, vol. 31, no. 6, pp. 164, 2012.

[10] B. Kim, P. Tsiotras, J. Hong, O. Song, “Interpolation and parallel adjustment of center-sampled trees with new balancing constraints,” *The Visual Computer*, vol. 31, no. 10, pp. 1351–1363, 2015.

[11] B. Lévy, S. Petitjean, N. Ray, J. Maillot, “Least squares conformal maps for automatic texture atlas generation,” *ACM transactions on graphics (TOG)*, vol. 21, pp. 362–371, 2002.

[12] B. Johanna, H. Markus, P. Hanspeter, “State-of-the-art GPU-based large-scale volume visualization,” *Computer Graphics Forum*, vol. 34, no. 8, pp. 13–37, 2015.

[13] C. Crassin, F. Neyret, S. Lefebvre, E. Eisemann, “Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering,” *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, 2009.

[14] C. Crassin, F. Neyret, M. Sainz, S. Green, E. Eisemann, “Interactive indirect illumination using voxel cone tracing,” *Computer Graphics Forum (Proceedings of Pacific Graphics 2011),* vol. 30, no. 7, 2011.

[15] C. Crassin, S. Green , “Octree-based sparse voxelization using the GPU hardware rasterizer,” *OpenGL Insights*, pp. 303-318, 2012.

[16] Chen Wei, “Volumetric cloud generation using a Chinese brush calligraphy style,” Ph.D. Dissertation, *University of Cape Town*, 2014.

[17] C. J. Curtis, S. E. Anderson, J. E. Seims, K. W. Fleischer, D. H. Salesin, “Computer-generated watercolor,” *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 421–430, 1997.

[18] C. Regan, “An investigation into nausea and other side-effects of head-coupled immersive virtual reality” *Virtual Reality*, vol. 1, no. 1, pp. 17–31, 1995.

[19] D. Benson, J. Davis, “Octree textures,” *ACM Transactions on Graphics (TOG)*, vol. 21, no. 3, pp. 785–790, 2002.

[20] D. Coeurjolly, P. Gueth, J. Lachaud, “Regularization of voxel art,” *SIGGRAPH Talk 2018*, 2018.

[21] D. Cohen-Or, A. Kaufman, “Fundamentals of surface voxelization,” *Graphical models and image processing*, vol. 57, no. 6, pp. 453-461, 1995.

[22] D. Dolonius, E. Sintorn, V. Kämpe, U. Assarsson, “Compressing color data for voxelized surface geometry,” *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 2018.

[23] D. G. Debry, J. Gibbs, D. D. Petty, N. Robins, “Painting and rendering textures on unparameterized models”, *ACM Transactions on Graphics (TOG)*, vol. 21, no. 3, pp. 763–768, 2002.

[24] D. Kataoka, “Art and virtual reality, new tools, new horizons,” *Silicon Valley VR Expo.*, 2017.

[25] D. F. Keefe, D. A. Feliz, T. Moscovich, D. H. Laidlaw, J. J. LaViola, Jr. “CavePainting: A Fully Immersive 3D Artistic Medium and Interactive Experience,” *Proceedings of the 2001 Symposium on Interactive 3D Graphics (I3D)*, pp. 85–93, 2001.

[26] D. Keefe, R. Zeleznik, D. Laidlaw, “Drawing on Air: Input Techniques for Controlled 3D Line Illustration,” *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, vol. 13, no. 5, pp. 1067–1081, 2007.

[27] E. Gobbetti, F. Marton, J. A. I. Guitián, “A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric

datasets," *The Visual Computer,* vol. 24, no. 7-9, pp. 797–806, 2008.

[28] Ephtracy, "MagicaVoxel," https://ephtracy.github.io, 2013.

[29] E. Rosales, J. Rodriguez, A. Sheffer, "SurfaceBrush: From Virtual Reality Drawings to Manifold Surfaces," arXiv e-prints, Article arXiv:1904.12297, 2019.

[30] F. Losasso, F. Gibou, R. Fedkiw, "Simulating water and smoke with an octree data structure," *ACM Transactions on Graphics (TOG)*, vol. 23, no. 3, pp. 457–462, 2004.

[31] F. Reichl, M. Treib, R. Westermann, "Visualization of big SPH simulations via compressed octree grids," *2013 IEEE International Conference on Big Data,* pp. 71–78, 2013.

[32] Guillaumechereau, "Goxel" http://guillaumechereau.github.io/goxel/, 2015.

[33] Google, "Tilt brush by google," https://www.tiltbrush.com/, 2015.

[34] H. Samet, "Implementing ray tracing with octrees and neighbor finding," *Computers & Graphics*, vol. 13, no. 4, pp. 445–460, 1989.

[35] HTC Corporation, "HTC Vive," https://www.vive.com/, 2011.

[36] I. Boada, I. Navazo, R. Scopigno, "Multiresolution volume visualization with a texture-based octree," *The Visual Computer*, vol. 17, no. 3, pp. 185–197, 2001.

[37] I. P. Howard, "Depth perception," *Stevens' handbook of experimental psychology*, 2002.

[38] J. Chen, D. Bautembach, S. Izadi, "Scalable real-time volumetric surface reconstruction," *ACM Transactions on Graphics (TOG)*, vol. 32, no. 4, 2013.

[39] J. D. Macdonald, K. S. Booth, "Heuristics for ray tracing using space subdivision," *The Visual Computer,* vol. 6, no. 3, pp. 153–166, 1990.

[40] J. Hakkinen, T. Vuori, M. Paakka, "Postural stability and sickness symptoms

after HMD use," *IEEE International Conference on Systems*, Man and Cybernetics, vol. 1, pp. 147–152. 2002.

[41] J. Kniss, A. Lefohn, R. Strzodka, S. Sengupta, J. D. Owens, "Octree textures on graphics hardware," *ACM SIGGRAPH 2005 Sketches*, 2005.

[42] J. Kruger, R. Westermann, "Acceleration techniques for GPU-based volume rendering," *Proceedings of the 14th IEEE Visualization 2003 (VIS)*, IEEE Computer Society, p. 38, 2003.

[43] J. Lu, S. Diverdi, W. Chen, C. Barnes, A. Finkelstein, "RealPigment: Paint compositing by example," *Proceedings of the 12th International Symposium on Non-photorealistic Animation and Rendering (NPAR)*, 2014.

[44] J. Schmid, M. S. Senn, M. Gross, R. W. Sumner, "OverCoat: An Implicit Canvas for 3D Painting," *ACM SIGGRAPH 2011 Papers*, Article no. 28, 2011.

[45] K. Bürger, J. Krüger, R. Westermann, "Direct volume editing," *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, vol. 14, no. 6, pp. 1388–1395, 2008.

[46] K. Engel, M. Hadwiger, J. Kniss, C. Rezk-salama, D. Weiskopf, "Real-time volume graphics", *CRC Press*, 2006.

[47] Khronos Group, "Uniform Buffer Object- OpenGL," https://www.khronos.org/opengl/wiki/Uniform_Buffer_Object, 2017.

[48] K. Museth, "VDB: High-resolution sparse volumes with dynamic topology," *ACM Transactions on Graphics (TOG)*, vol. 32, no. 3, 2013.

[49] K. Zhou, M. Gong, X. Huang, B. Guo "Data-parallel octrees for surface reconstruction," *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, vol. 17, no. 5, pp. 669–681, 2011.

[50] L. Olsen, F. F. Samavati, M. C. Sousa, J. A. Jorge, "Sketch-based modeling:

A survey," *Computers & Graphics*, vol. 33, no. 1, pp. 85–103, 2009.

[51] L. P. Kobbelt, M. Botsch, U. Schwanecke, H.-P. Seidel, "Feature sensitive surface extraction from volume data," *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 57–66, 2001.

[52] L. Siemon, "VoxelShop," https://blackflux.com/, 2013.

[53] M. Agrawala, A. C. Beers, and M. Levoy, "3D painting on scanned surfaces," *Proceedings of the 1995 symposium on Interactive 3D graphics (I3D)*, pp. 145–ff, 1995.

[54] M. F. Deering, "HoloSketch: A Virtual Reality Sketching/Animation Tool," *ACM Transations Computer-Human Interaction*, vol. 2, no. 3, 1995.

[55] M. Hadwiger, P. Ljung, C. R. Salama, T. Ropinski, "Advanced illumination techniques for GPU-based volume raycasting," *ACM SIGGRAPH 2009 Courses*, pp. 2:1–2:166, 2009.

[56] M. Hadwiger, J. Beyer, W. Jeong, H. Pfister, "Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach," *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, vol. 18, no. 12, 2012.

[57] Minddesk Software GMBH., "Qubicle," https://www.minddesk.com/, 2005.

[58] M. Foskey, M. A. Otaduy, M. C. Lin, "ArtNova: touch-enabled 3D model design," *Proceedings IEEE Virtual Reality 2002*, pp. 119–126, 2002.

[59] M. Harris, I. Buck , "GPU flow control idioms," *GPU gems*, vol. 2, pp. 547-555, 2005.

[60] M. Kazhdan, H. Hoppe, "Screened poisson surface reconstruction," *ACM Transactions on Graphics (TOG)*, vol. 32, no. 3, 2013.

[61] Mojang, "Official site|minecraft," https://minecraft.net/en-us/?ref=m, 2009.

[62] M. Pätzold, A. Kolb, “Grid-free out-of-core voxelization to sparse voxel octrees on GPU,” *Proceedings of the 7th conference on high-performance graphics (HPG),* pp. 95-103, 2015.

[63] M. Schwarz, H. P. Seidel , “Fast parallel surface and solid voxelization on GPUs,” *ACM Transactions on Graphics (TOG)*, vol. 29, no. 6, pp. 179, 2010.

[64] M. Zollhöfer, P. Stotko, A. Görlitz, C. Theobalt, M. Nießner, R. Klein, A. Kolb, “State of the Art on 3D Reconstruction with RGB-D Cameras.” *Computer Graphics Forum*, vol. 37, no. 2, pp. 625-652, 2018.

[65] M. Zwicker, M. Pauly, O. Knoll, M. Gross, “Pointshop 3D:An Interactive System for Point-based Surface Editing,” *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 322–329, 2002.

[66] Oculus, “Quill by story studio,” https://storystudio.oculus.com/en-us/, 2016.

[67] O. Klehm, I. Ihrke, H. Seidel, E. Eisemann, “Property and lighting manipulations for static volume stylization using a painting metaphor,” *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, vol. 20, no. 7, pp. 983–995, 2014.

[68] S. Longay, A. Runions, F. Boudon, P. Prusinkiewicz, “Treesketch: interactive procedural modeling of trees on a tablet,” *Proceedings of the international symposium on sketch-based interfaces and modeling*, pp. 107–120, 2012.

[69] S. M. Seitz, B. Curless, J. Diebel, D. Scharstein, R. Szeliski, “A comparison and evaluation of multi-view stereo reconstruction algorithms,” *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 519–528, 2006.

[70] S. Popov, J. Günther, H.-P. Seidel, P. Slusallek, “Stackless kd-tree traversal for high performance GPU ray tracing,” *Computer Graphics Forum*, vol. 26, no. 3, pp. 415–424, 2007.

[71] S. Schkolne, M. Pruett, P. Schröder, “Surface Drawing: Creating Organic 3D Shapes with the Hand and Tangible Tools,” *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pp. 261–268, 2001.

[72] S. W. Wang, A. E. Kaufman, “Volume sampled voxelization of geometric primitives,” *Proceedings of the 4th conference on Visualization'93,* pp. 78-84, 1993.

[73] S. Tsang, R. Balakrishnan, K. Singh, A. Ranjan, “A Suggestive Interface for Image Guided 3D Sketching,” *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pp. 591–598, 2004.

[74] T. Igarashi, D. Cosgrove, “Adaptive unwrapping for interactive texture painting,” *Proceedings of the 2001 symposium on Interactive 3D graphics*, pp. 209–216, 2001.

[75] T. Igarashi, S. Matsuoka, H. Tanaka, “Teddy: a sketching interface for 3D freeform design,” *ACM SIGGRAPH 2007 courses*, 2007.

[76] P. Hanrahan, P. Haeberli, “Direct WYSIWYG Painting and Texturing on 3D Shapes,” *ACM SIGGRAPH Computer Graphics*, vol. 24, no. 4, pp. 215–223, 1990.

[77] R. Brucks, “*Content-Driven Multipass Rendering in UE4*,” https://youtu.be/QGIKrD7uHu8 GDC, 2017.

[78] R. D. Kalnins, L. Markosian, B. J. Meier, M. A. Kowalski, J. C. Lee, P. L. Davidson, M. Webb, J. F. Hughes, A. Finkelstein, “WYSIWYG NPR: Drawing strokes directly on 3D models,” *ACM Transactions on Graphics*

(TOG), vol. 21, pp. 755–762, 2002.

[79] R. K. Hoetzlein, "GVDB: Raytracing sparse voxel database structures on the GPU," *Proceedings of High Performance Graphics (HPG)*, pp. 109–117, 2016.

[80] R. Patterson, M. D. Winterbottom, B. J. Pierce, "Perceptual issues in the use of head-mounted visual displays," *Human factors*, vol. 48, no. 3, pp. 555–573, 2006.

[81] R. Setaluri, M. Aanjaneya, S. Bauer, E. Sifakis, "SPGrid: A sparse paged grid structure applied to adaptive smoke simulation," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 6, pp.205, 2014.

[82] S. Bruckner, M. E. Groller, "Volumeshop: An interactive system for direct volume illustration," *IEEE Visualization (VIS)*, 2005

[83] S. Diverdi, "A brush stroke synthesis toolbox," *Springer London*, pp. 23–44, 2013.

[84] S. F. Frisken, R. N. Perry, A. P. Rockwood, T. R. Jones, "Adaptively sampled distance fields: A general representation of shape for computer graphics," *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 249–254. 2000.

[85] S. Lefebvre, S. Hornus, F. Neyret, "Octree textures on the GPU," *GPU Gems 2*, Pharr M., (Ed.). Addison-Wesley, pp. 595–613, 2005.

[86] S. Laine, T. Karras, "Efficient sparse voxel octrees," *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 55–63, 2010.

[87] T. Ize, "Robust BVH ray traversal-revised," *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 2, pp. 12–27, 2013.

[88] T. Ju, F. Losasso, S. Schaefer, J. Warren, "Dual contouring of hermite data,"

ACM Transactions on Graphics (TOG), vol. 21, no. 3, pp. 339–346, 2002.

[89] V. Havran, J. Bittner, J. Žára, "Ray tracing with rope trees," *14th Spring Conference on Computer Graphics*, pp. 130–140, 1998.

[90] W. V. Baxter, J. Wendt, M. C. Lin, "IMPaSTo: A realistic, interactive model for paint," In *Proceedings of the International Symposium on Non-Photorealistic Animation and Rendering (NPAR)*, pp. 45–56. 10, 2004.

[91] Y. Kim, B. Kim, Y. J. Kim, "Dynamic Deep Octree for High-resolution Volumetric Painting in Virtual Reality," *Computer Graphics Forum*, vol. 37, no. 7, pp. 179-190, 2018.

[92] Y. Gingold, T. Igarashi, D. Zorin, "Structured Annotations for 2D-to-3D Modeling," *ACM SIGGRAPH Asia 2009 Papers (SIGGRAPH Asia)*, No. 148, p. 9, 2009.

[93] Y. J. Kil, P. Renzulli, O. Kreylos, B. Hamann, G. Monno, O. G Staadt, "3D warp brush modeling," *Computers & Graphics*, vol. 30, no. 4, pp. 610–618, 2006.

[94] Z. Chen, B. Kim, D. Ito, H. Wang, "Wetbrush: GPU-based 3D painting simulation at the bristle level," *ACM Transactions on Graphics (TOG)*, vol. 34, no. 6, 2015.

국문초록

가상현실 기반 고해상도 3차원 페인팅과 볼륨 브러시 모델 연구

김여진

컴퓨터공학과

이화여자대학교 대학원

최근 들어 가상현실(VR) 기술이 발전함에 따라 2차원 디지털 페인팅이 3차원 공간으로 확장되고 있다. VR기반의 3차원 페인팅 시스템들이 잇따라 등장하면서, 아티스트들이 점차 VR 페인팅을 하나의 예술 장르로 받아들이고 있는 추세이다. 그러나 현재의 VR 페인팅 시스템은 모두 표면기반 페인팅 시스템으로, 사용자가 3차원 공간에 그린 경로를 따라 3차원 볼륨이 아닌 2차원 평면 기하만을 생성한다. 표면기반 페인팅 시스템에서는 그려진 스트로크들은 서로 합치기가 힘들며, 사용자가 실제 페인팅을 하듯이 색상을 칠하거나, 재색칠 하거나, 색상을 섞거나, 혹은 불투명한 색상을 칠하는 것이 매우 어렵다. 뿐만 아니라 3차원 디지털 아트라는 거시적인 관점에서도 이 문제를 조명해보자면, 오랫동안 다양한 종류의 3차원 저작 도구들이 연구되어 왔음에도 불구하고 3차원 공간 자체에 디지털 페인팅을 할 수 있는 프로그램은 부재한 상황이다. 물론 기존의 3차원 저작 도구를 이용하면 복셀 도트 디자인, 복잡한 3차원 모델 디자인 등과 같은 특정한 스타일의 3차원 디지털 아트 작품을 만들 수 있다. 특히 3차원 모델링 분야에서, 사용자가 저작 도구를 이용하여 정교하고 사실적인 3차원 아트 작품도 만드는 것도 가능하다. 하지만 기존 3차원 저작 도구들은 여러 단계의 워크플로우를 거치는 전문적인 기술을 요구하거나, 팀 단위의 저작 활동을 필요로 하거나, 혹은 모양, 표현, 규모, 사용자 인터페이스 측면에서의 한계점 때문에 3차원 디지털 아트의 확장성을 떨어뜨린다.

본 학위 논문에서는 새로운 3차원 디지털 아트의 한 장르로써, 2차원 디지털 페인팅을 3차원 디지털 페인팅으로 확장하는 가상현실 기반의 고해상도 볼륨 페인팅 시스템을 제안한다. 제안하는 시스템은 동적 팔진 트리 기반의 페인팅 및 렌더링 시스템으로, 각 프로세서의 하드웨어적 특성을 반영하여 CPU 기반 팔진 트리는 팔진 트리 모델링을, GPU 기반 팔진 트리에서는 볼륨 렌더링을 위해 사용한다. 입력한 스트로크에 대해 CPU 상에서 팔진 트리는 동적으로 노드를 생성/삭제하며, CPU 상 팔진 트리 구조의 변화를 GPU 상 팔진 트리 구조에 점진적으로 업데이트 한다. GPU 상 팔진 트리에서 광선투사 시 상수시간의 이웃 노드 접근을 보장하기위해, 형식적으로 간결하면서도 효율적으로 메모리를 사용하는 3-이웃 연결성을 새롭게 제시한다. 나아가 GPU 상에서의 3-이웃 계산량을 줄일 수 있는 컬링 마스크를 CPU상에서 계산하여 GPU쪽으로 업로드하는 기법에 대해서도 기술한다. 이 과정에서 렌더링 프레임 률과 저 지연 업데이트를 절충하는 업데이트 성능을 실험적으로 검증하며, 업데이트가 지연될 시 일어날 수 있는 잘못된 시각적 피드백을 줄이는 기법을 제시한다. 렌더링 측면에서는, 고해상도 팔진 트리에서 광선 투사시 발생하는 수치 오차 문제를 해결하기 위해 셀 국부 좌표계 기법을 제시하였다. 또한 광선 투사 시 수치 오차가 전파되는 과정을 분석하여 제시한 기법이 이론적인 오차범위에서 정밀함을 보였으며, 실험적으로도 이를 입증하였다. 추가적으로 렌더링 속도를 가속화하기 위하여 CPU기반의 사진 트리/팔진 트리의 보간을 GPU기반의 보간으로 확장하고, 사진 트리 기반의 포비티드 렌더링(Foveated rendering)기법을 기술하였다.

또한 본 논문에서는 3차원 디지털 페인팅 측면에서, 많은 복셀을 동시에 다루는 3차원 페인트 브러시 모델을 제안한다. 우선, 2차원 디지털 페인팅에서 주로 사용되는 칠하기, 지우기, 재색칠, 색 섞기, 블러, 스머지 등과 같은 2차원 브러시 모델을 고해상도 팔진트리에서 사용할 수 있는 브러시 모델로 재구성하였다. 두 번째로, 3차원 디지털 페인팅에서 발생하는 고유한 문제들을 다루는 볼륨 특이적 브러시 모델을 제안하였다. 3차원 볼륨 페인팅에서 자주 일어나는 작업 중 하나인 생성된 복셀의 해상도를 조절하는 반복작업을 줄이기 위해, 페인팅 디테일을 조정하면서도 메모리 관리를 할 수 있도록 돕는 복셀 해상도 조절 도구들을 모델링한다. 복셀 해상도를 증가시

키는 모델로는, 색상과 복셀 해상도의 확산을 기반으로 하는 복셀 멜트 브러시를 모델링한다. 복셀 멜트는 페인팅의 디테일을 유지하면서도, 급격한 복셀 해상도 변화를 완만하게 한다. 또한, 사용자가 복잡한 공학지식이 없어도 사용할 수 있는 룸, 복셀 모자이크, 등위값 기반 복셀 머지와 같은 복셀 해상도를 낮추는 필터들을 모델링한다. 마지막으로, 서피스 기반의 페인팅 엔진과 볼륨 기반의 페인팅 엔진 결합을 처음으로 시도하여, 볼륨 페인팅에서의 깊이 감각 문제를 해결하고 불안정한 입력에 대해 직선/곡선을 그릴 수 있도록 하는 하이브리드 브러시 모델들을 제안한다.