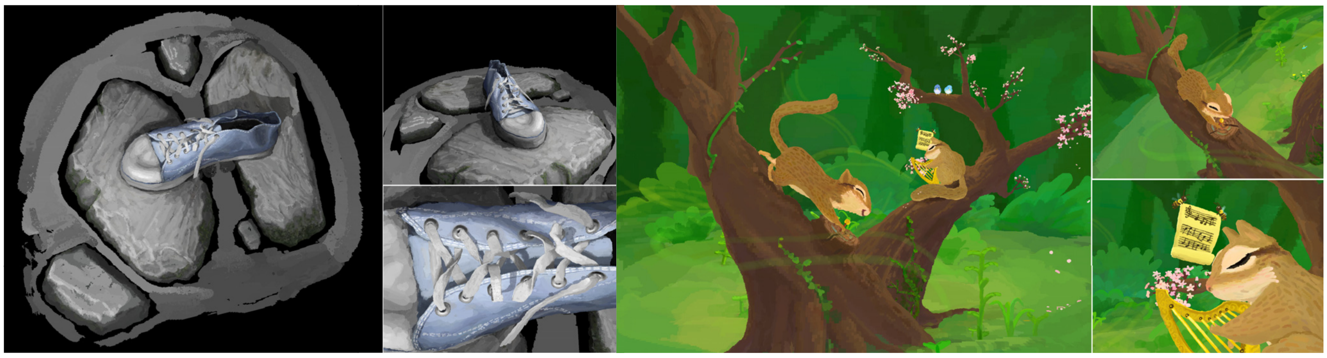
Pacific Graphics 2018
H. Fu, A. Ghosh, and J. Kopf
(Guest Editors)

Volume 37 (2018), Number 7

Dynamic Deep Octree for High-resolution Volumetric Painting in Virtual Reality

Yeojin Kim[1] Byungmoon Kim[2] Young J. Kim[1]

[1]Ewha Womans University, South Korea
[2]Adobe Systems Incorporated, USA



(a) A Sneaker (b) Spring Concert

Figure 1: *Our dynamic tree allows interactive volumetric painting on a very large canvas (40km^3) with high details (down to 0.3mm^3). Volumetric painting with ray casting supports transparency (green breeze in 1(b)) with a very high depth complexity. Unlike geometry-based painting, recoloring and color mixing operations are easily extended from 2D painting. Artists can paint for several hours in virtual reality, due to the hard real-time constraint for frame rates and low-latency update of dynamic octree.*

Abstract

With virtual reality, digital painting on 2D canvas is now being extended to 3D space. In this paper, we generalize the 2D pixel canvas to a 3D voxel canvas to allow artists to synthesize volumetric color fields. We develop a deep and 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 GPU with low latency without compromising the frame rates of the rendering. Our octree is balanced and uses a novel 3-neighbor connectivity for format simplicity and efficient storage, while allowing constant neighbor access time in ray casting. To further reduce the GPU-side 3-neighbor computations, we precompute a culling mask in CPU and upload it to GPU. Finally, we analyze the numerical error-propagation in ray casting through high resolution octree and present a theoretical error bound.

CCS Concepts
•***Computing methodologies*** $\rightarrow$ ***Virtual reality; Volumetric models;*** *Rendering;*

1. Introduction

With virtual reality (VR), digital painting on 2D surfaces is being extended into 3D volumes. VR-based 3D painting applications have recently surged and are now widely accepted as a new art form by artists. 3D painting is distinguished from traditional 3D modeling/rendering. While 3D modeling/rendering typically assumes a team of artists performing a sequence of works such as modeling, lighting, rendering, and composition, a 3D painting system allows an artist to finish an 3D painting within her/his own budget [Kat17]. Also, a 3D painting system enables an artist, without physical constraints, to rapidly sketch a large scene and to directly manipulate

© 2018 The Author(s)
Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd. Published by John Wiley & Sons Ltd.

the color fields of the scene in 3D space rather than draw it indirectly by adjusting materials and lighting.

State-of-the-art VR painting systems such as Tilt brush and Quill [Goo15, Ocu16] are *geometric painting* systems that emit 2D surface geometries as artists make brush strokes (Fig. 2(b)). These systems have their intrinsic boundaries. Color mixing between strokes is not supported since overlaps between stroke geometries are infinitely thin, and hard and expensive to compute robustly. The simple recoloring of part of a stroke would be complicated since a texture map needs to be invoked such that updated colors can be stored into the texture map. In addition, semi-transparent volume depiction would be limited since efficient handling of a large number of stroke geometries in high-depth complexity is nontrivial.

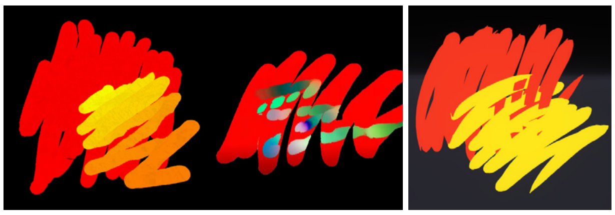

(a) Ours (b) Tilt Brush

Figure 2: *Our volume painting system (a) allows trivial color mixing and recoloring operations, which are very difficult to be implemented in geometry-based painting systems. A state-of art system (b) simply does not allow color interactions between strokes.*

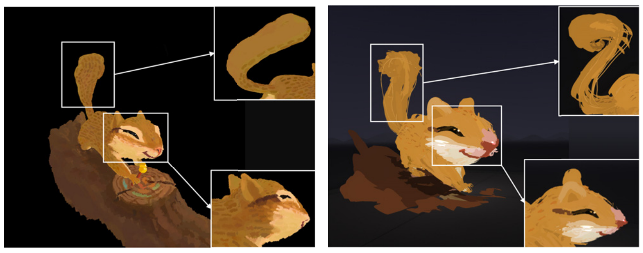

(a) Ours (b) Tilt Brush

Figure 3: *In geometry-based painting (b), over-painting strokes should be very carefully planned so that the new stroke is slightly above previous strokes to avoid occlusion and z-fighting. However, in (a), voxels can be simply added, erased, refined, coarsened, or recolored with various color mixing modes.*

In this paper, we explore a new route for VR painting in 3D space: *volumetric painting*. Volumetric strokes applied to a 3D grid are amenable to depicting both solid and non-solid shapes (see green, translucent breeze in Fig. 1). Volumetric painting also naturally supports color mixing. Moreover, artists 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 Fig.3(a), which is a very typical practice in 2D painting. Finally, the volumetric painting system naturally handles semi-transparent strokes and renders the resulting semi-transparent volume using ray-casting.

On a 3D canvas, unlike a 2D planar 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. This requires a 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; e.g. a virtual canvas with a painting space of up to 40 Km^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 (see Fig. 1).

We believe that the volumetric painting system supporting very large canvases with fine details using a highly adaptive tree is a promising new direction as a new VR art style, and perhaps even new fine art may be developed over time. In this paper, anticipating to pave the new way for 3D painting art styles that can be persistent in the future, we advance technology towards this overarching goal.

After extensive experiments and conversations with artists, we have concluded that the following elements are required for an interactive volumetric painting system in VR.

- *Dynamic tree update*: artists will continuously modify the underlying tree; we therefore need to update the tree dynamically.
- *Constant frame rates*: artists spend several hours painting in VR, and consideration must therefore be given to mitigating the possibility of VR-sickness. 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 an artist applies a stroke, the tree should be modified immediately and rendered back to the artist. Therefore, we require low latency for stroke display.
- *Low memory consumption*: when a very large canvas is used, we found artists 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.

To realize such requirements, the use of GPU is inevitable. However, using GPU alone would be lesser efficient than using CPU and GPU together for the following reasons. First, a CPU can handle complex tree adjustment and color blending operations in a more versatile manner than a GPU. Second, a CPU has an independent separate memory pool that can perform low-latency incremental updates to the tree with uninterrupted rendering in a GPU. In this way, we utilize one octree in a CPU for painting, and the other octree in a GPU for rendering. Overall, we propose incremental update strategies from the CPU to the GPU that satisfy all the aforementioned requirements. Finally, we solve the numerical issues in traversing a high-depth octree. In summary, our contributions are as follows:

- 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 3D volumetric field painting effects such as color pick-up and color mix with adaptive grids.

2. Related Works

In this section, we survey relevant octree works in the literature including representation, update, and authoring techniques as well as GPU-based ray-casting methods.

2.1. Adaptive Spatial Grids

2.1.1. Octree representation

Octree grids have been applied to various problems such as distance field generation [FPRJ00, KBSS01], texturing [DGPR02, BD02, LHN05], modeling [JLSW02, KH13, HWB*13], simulation [LGF04, Mus13, Hoe16], model reconstruction [ZGHG11, CBI13], and visualization [BNS01, GMG08, CNLE09, JMH15] 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 of using a high-depth octree is the traversal time from a root cell to a leaf cell. To reduce this traversal time, a more shallow tree has been used [LSK*06, LK10, Mus13, Hoe16]. [LSK*06] 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 [LSK*06]), 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.

In order to render large-scale scene, full or out-of-core [GMG08, CNLE09, Mus13, HBJP12, RTW13] updates to GPU have been made for rendering applications. Recently, studies on a directed acyclic graph with scalar fields [DKB*16, DSKA18] 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.

2.1.2. Dynamic octree update

Besides one-time construction or full reconstruction [DGPR02, BD02, LHN05, CNLE09], octree can also be incrementally adjusted. Real-time dynamic octree adjustment in GPU has been applied [LSK*06, CNS*11]. In [CNS*11], 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 is stored on a CPU and the subtree data is streamed through CPU-GPU data transfer in a view-dependent manner [GMG08]. In order to retain connectivity information with subtrees, the indices of all eight children are stored. Recently, [Hoe16] supports dynamic topological updates on GPU, however, it only supports insertion for now.

2.1.3. Octree texture authoring

Glift has been adopted for an adaptive texture authoring application [LSK*06, KLS*05]. Given a model represented in an adaptive grid, the authors found voxels in a brush stroke by projecting the model and the brush stroke onto a 2D screen-space and then update the color of voxels. If a higher resolution is required, they reallocated a tile populated with interpolated color. This approach limits volumetric painting; non-surface voxels are not considered and cannot be colored. In fact, the problem of octree texture authoring [KLS*05, DGPR02, BD02] significantly differs from our problem. We do not need an underlying model to paint as our aim is to author a general volumetric field including transparency. In some sense, our work models various structures and objects with clear and blurry boundaries while simultaneously coloring them with volumetric brush stroking.

2.2. GPU-Based Ray Casting on Adaptive Grids

Ray casting has been extensively researched for several decades [EHK*06, HLSR09]. Since a ray traversal on 3D adaptive grids is expensive, acceleration techniques such as neighbor precomputation, early ray termination, and empty space skipping [KW03] are often used. In order to reduce the cost of finding neighbors, the ROPE algorithm [HBv98] was developed for a k-d tree and neighbor linking was proposed [Sam89, MB90] 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 axial directions instead of pointing to neighbor cells directly [PGSS07]. An octree, even when 2-to-1 balanced, needs a maximum of 24 neighbors per cell. [GMG08] has reduced the number of neighbors down to six per cell by pointing the parents of neighbors. In our paper, 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 dynamic updating of an octree on-the-fly on a GPU as the tree connectivity changes.

A sparse voxel octree (SVO) [CNLE09, LK10] 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 dynamic update. Particular objects in [CNS*11] 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 levels, called OpenVDB [Mus13]. Recently, OpenVDB structure was implemented in GPU [Hoe16], which enables efficient neighbor access using ghost voxels and GPU-based ray casting. [Mus13, Hoe16] address dynamic scene rendering problem that does 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 $128K^3$ [DKB*16], 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 view points, which makes rendering unstable in a VR environment. [Ize13] used a padding factor in order not to miss a cell due to such precision. In this paper, we study a more principled approach - i.e. we propose to analyze the propagation of numerical error and ensure that ray computation does not increase numerical error regardless of the ray length.

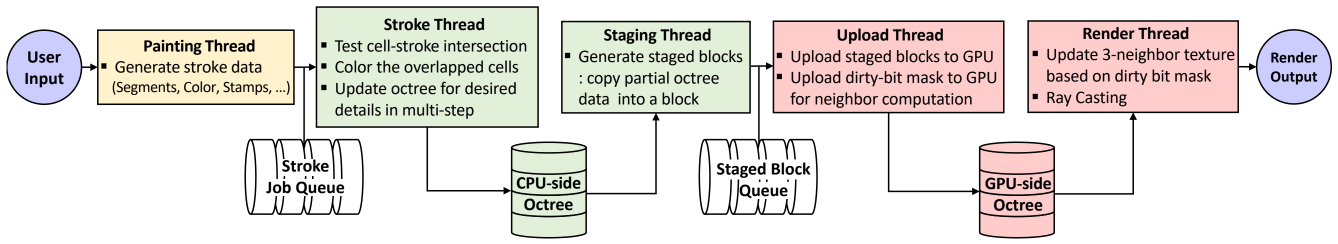
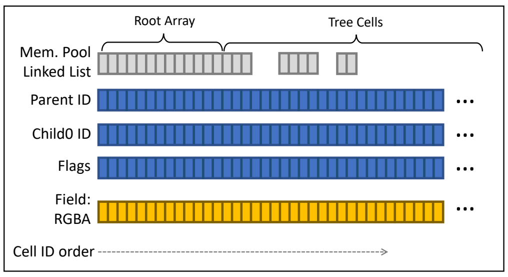

User Input
Painting Thread
Generate stroke data (Segments, Color, Stamps, ...)
Stroke Thread
Test cell-stroke intersection
Color the overlapped cells
Update octree for desired details in multi-step
Stroke Job Queue
CPU-side Octree
Staging Thread
Generate staged blocks : copy partial octree data into a block
Staged Block Queue
Upload Thread
Upload staged blocks to GPU
Upload dirty-bit mask to GPU for neighbor computation
GPU-side Octree
Render Thread
Update 3-neighbor texture based on dirty bit mask
Ray Casting
Render Output

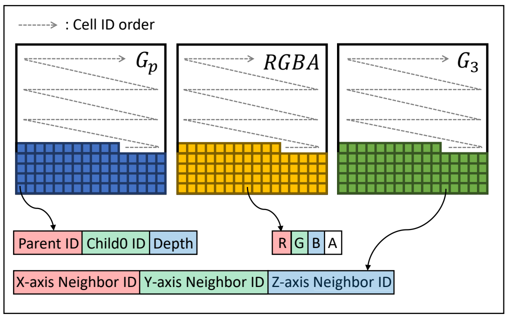
Figure 4: *A sequence of CPU-side threads from user input to rendering output. After applying brush strokes, the octree is dynamically adjusted and incrementally uploaded to GPU for rendering. We enable uninterrupted painting by using the stroke job queue, and enable uninterrupted rendering by staging blocks. We achieved very small latency from painting to rendering, measured at under 20 milliseconds on average.*

3. System Primer

In section 4, we describe both CPU and GPU memory layouts of our system. In general, the CPU-side octree is constructed of linear pools (parent/child indices and fields, as shown in Fig. 5 (a)) that are packed into textures in the GPU. The CPU has temporary pools (for painting) that do not exist in the GPU. For an efficient ray traversal, the GPU has a neighborhood connectivity pool, $\mathcal{G}_3$ texture, that CPU does not have. In section 5, we design adjustment and update strategies that meet the requirements outlined in section 1. We subsequently slice a painting task into smaller tasks by using one-level adjustment.Then we stage the changed blocks, compute the update masks for $\mathcal{G}_3$, upload them to the GPU, and validate the results.



(a) Dynamic Deep Octree on the CPU



(b) Dynamic Deep Octree on the GPU

Figure 5: *Two octrees with different flavors. (a) A CPU-side octree for dynamic adjustment and color blending, pick up, erase, and recolor operations. (b) A GPU-side octree for rendering. Each side has only one copy of the octree.*

Interactive volumetric field painting is composed of several parallel tasks: processing strokes, adjusting the octree, uploading the octree to the GPU, and rendering the octree. These are implemented in multiple threads as illustrated in Fig. 4. In the *painting thread*, the segments, color, and the stamp of strokes are queued. In the *stroke thread*, we conduct a stroke-cell intersection test, and refine or coarsen the intersecting cells. The octree memory is divided into a uniformly-sized blocks. In the *staging thread*, we copy a sequence of cells including newly refined or coarsened cells into a separate memory, which we call staged block. These staged blocks are pushed to the staged blocks queue. In the *upload thread,* we consume this staged queue by uploading staged blocks to the GPU. Finally, the *rendering thread* renders the octree using ray casting.

In order to support concurrent tasks such as painting and rendering, we adopt a strategy to construct a single octree in the CPU for painting and another octree in the GPU for rendering. The advantage of using octrees both on CPU and GPU is that it enables different-flavored buffers (Fig. 5), tree adjustment, and color blending for the CPU and volume ray casting for the GPU, which make painting and rendering tasks hazard-free. The remaining challenge is to upload a tree from the CPU to the GPU at the right time with an appropriate strategy.

4. Octree Representation and Memory Layouts

4.1. Root Array and Tree Depth

In order to author highly-detailed and dynamic volumetric fields, we use an array of octrees. We store the roots of the octree as a 3D array, which we call a *root array*. Each root can be refined up to 24 times, which is the maximum depth in our current implementation. The painting details appear to be sufficiently fine at this level. While the root array helps to reduce the tree depth and offers several advantages such as trivial parallelization, in a very large canvas with highly adaptive tree, the advantage appears to diminish. Therefore, we resort to a relatively coarse, 4^3 array of octree roots, each of which can be refined to a maximum depth. Effectively, this is equivalent to a single 26-deep octree root. This resolution can span a volumetric space of from $0.3mm^3$ to $40Km^3$ with respect to the room-scale VR setup.

4.2. Dynamic Deep Octree Representation in a CPU

Based on a previous study [KTHS15], we construct a 2:1 balanced octree on the CPU-side, where the difference in the depth of the two neighboring cells is less than or equal to one. Since we are authoring volumetric fields that can be defined at any location in the 3D space and for the sake of simplicity, eight children are created when a cell is refined. Similar to [GMG08], we do not use pointers, but instead use the index I that uniquely identifies each cell. Our octree is made up of multiple linear memory pools indexed by I. Painting properties such as color, density, and temporary variables are stored as separate field pools. More field pools can be added, even dynamically if needed, such as alpha values. We group properties that are likely to be accessed together and then store them in a single memory pool. To allow dynamic refinement and coarsening, we implement linked-list based memory management. The size of an allocation unit is fixed as we allocate or free eight cells. When a unit is freed, we return to the beginning of the pool to ensure that the pool is populated from the beginning. The pool begins with a uniform root array, i.e., 4^3 array, that cannot be freed. We have another separate flag pool for the depth of cells and other bit-field flags for tree adjustment.

4.3. Tree Graph $\mathcal{G}_p$ and Novel Neighbor Graph $\mathcal{G}_3$

Our octree is defined only by parents and children pools that store two indices of the parent and the first child, since indices of the remaining seven children are consecutively numbered and hence do not need to be stored. Let this primal tree graph be $\mathcal{G}_p$. Since we do not use a dual tree such as used in [LSK*06, LK10], each cell in $\mathcal{G}_p$ is represented only by two 32-bit indices: the parent and the first child. Note that ray casting using only $\mathcal{G}_p$ may have poor performance as the depth of the tree increases; e.g. if the maximum depth is 24, the traversal path from $\mathcal{G}_p$ to a neighbor can be as long as 48 in the worst case. Therefore, an immediate neighbor topology such as neighbor linking for octrees [Sam89, MB90], or ROPEs for KD-trees [HBv98], is useful to accelerate the ray traversal. ROPE connects cells that share a face, resulting in the blue connections from cells C and D in Fig. 6. Note that D has five neighbors. In a 2-to-1 balanced octree, since four neighbors can exist on cell's each faces, the number of neighbors can be six to 24.

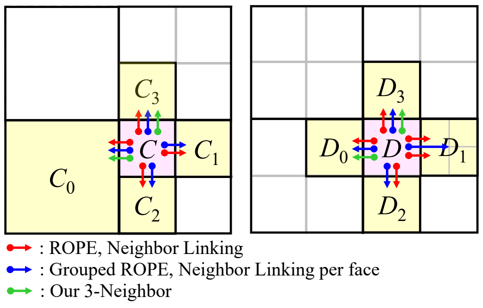


Figure 6: *Neighbors of C (left), and D (right) in the quadtree. C, C_1, and C_2 share the same parent, and hence computing C_1 and C_2 from C is easy. On the other hand, tree distances between C and C_0, C_3, and between D and D_0, D_3 can be very large. Therefore, we precompute C_0, C_3, D_0, and D_3 and stored them in a separate texture.*

By grouping cells per face with ROPE [PGSS07], or linking the same depth or the smaller-depth neighboring cells per face [GMG08], the number of neighbors is reduced to 6. In our work, we further reduce the number of neighbors to three as follows. First, let's consider the same depth cell (e.g. D_1 in Fig. 6) or the smaller-depth neighboring cell (C_0). Since each face has only one such neighbor, every cell has six neighbors. We then store only three of these six neighbors. Since eight children have consecutive indices, three of these neighbors share the same parent (e.g. C_1, C_2 of C and D_0, D_2 of D) and can be obtained immediately from its associated cell index (e.g., $C_1 = C + 1$). The other three neighbors may have different parents (e.g. C_0, C_3, D_1, D_3), and because computing such neighbors requires long tree traversals, they are precomputed. We refer to this neighbor structure as the 3-neighbor topology, denoted by $\mathcal{G}_3$. The graph $\mathcal{G}_3$ is a collection of the three neighbors for each cell.

Using the union of $\mathcal{G}_3$ and $\mathcal{G}_p$, we can quickly discover all the six to 24 neighbors, since in $\mathcal{G}_p \cup \mathcal{G}_3$, the distance between two cells sharing a face is 2 in $\mathcal{G}_p$, or 1 or 2 in $\mathcal{G}_3$. Therefore, finding neighbors in our scheme incurs a low cost, while finding a group of cells, as in previous works [PGSS07, GMG08], can have the distance between neighbors greater than 2 in $\mathcal{G}_p$. In Fig. 6, the leaves of D_1 that are in contact with D can be found easily as $\mathcal{C}(D_1)$ and $\mathcal{C}(D_1)$+2, where $\mathcal{C}(\cdot)$ denotes the first child. In the case where a ray proceeds to D_1, we can quickly identify $\mathcal{C}(D_1)$ or $\mathcal{C}(D_1)+2$ using the ray parameters. Thus, $\mathcal{G}_3$ is represented by three indices per cell. We store $\mathcal{G}_3$ as a separate neighbor pool. Using $\mathcal{G}_3$, we simplified variable numbers of immediate neighbors to a constant 3. We also reduced the memory required for a fast traversal from 24 (maximum) neighbors to 3 neighbors, while still allowing small constant time access to all of the 24 neighbors.

4.4. Dynamic Deep Octree Representation in a GPU

Mapping the octree pools in a CPU to textures in a GPU is straightforward. Since textures have a resolution-limit in each dimension, we cannot use a 1D texture. We must use 2D or 3D textures and map a linear index I to two or three indices. Since modern GPUs support up to 16 thousand texels per dimension, 2D textures can support up to $256M$ cells. We pack $\mathcal{G}_p$ (parent, child) and the depth into a texture. RGBA color is stored as another texture.

$\mathcal{G}_3$ (3-neighbor) is another integer texture with three channels. Note that we do not use $\mathcal{G}_3$ on the CPU. The $\mathcal{G}_3$ texture is built from the $\mathcal{G}_p$ texture upon updating. While computing $\mathcal{G}_3$ is fast, it is still slow enough to hamper the frame rate. In section 5, we develop a strategy to reject most of this GPU-side computation with a small amount of mask computation and transfer it from the CPU to the GPU.

5. Dynamic Octree Update With Low-Latency and Consistent Frame Rates

The importance of avoiding nausea, sickness, and postural instability [Reg95, HVP02, PWP06] is escalated in VR painting lasting several hours. Among factors on those symptoms, little delay in sync between the rendered scene and the head motion is an minimum requirement that cannot be compromised. However, simply copying an octree to a texture will take 222ms even with full bandwidths of a CPU, a PCIe, and a GPU. Therefore, instead of updating the

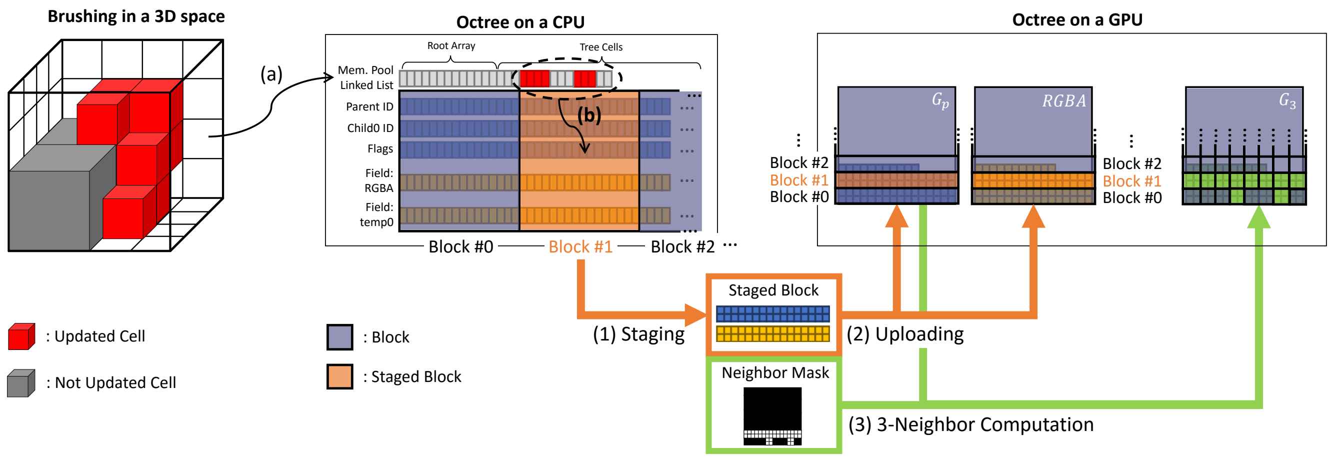


Figure 7: *Staged updates with a neighbor computation mask. (a) Brushing a stroke in a 3D space is local not only in the space but also in the memory (red cells in the 3D space and the octree on a CPU). Therefore, we divide the octree into blocks shown as sky blue and orange boxes. (b) Among blocks, we find a block (orange box) containing updated cells (dotted ellipse). In the middle of the painting, (1) the block on the CPU is staged with a neighbor computation mask and (2) uploaded to the GPU. (3) Note that $\mathcal{G}_3$ is not uploaded, but is rather computed on the GPU.*

entire tree (for example, 134M cells which is equivalent to 2.15GB memory in Fig. 1), we incrementally and dynamically update the underlying octree.

5.1. Incremental Tree Adjustment

Although we can effectively reject cells that do not intersect with brush stamps, painting an octree can still be expensive. A broad brush stroke can be applied near a highly-refine region or a fine brush stroke can be applied near a coarsened region. To address a sharp change in the depth of cells, we develop a multi-step strategy. In a CPU-thread separate from rendering, we first paint on the CPU tree without tree adjustment and update on the GPU. The next step is a tree-adjustment stage where we mark cells that should be refined or coarsened and perform one-level refinement or coarsening per frame. 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. In the next section, we propose algorithms for dynamic low-latency updating to the GPU while keeping the frame rate constant for VR.

5.2. Block-based Update using Staging

Since the stroke diameter is set to be approximately 10 cells, for a stroke length of one, about up to 1,000 cells per stroke would require updates. Since uploading a 1,000 times to the GPU would also 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 $\mathcal{G}_p$ tends to be filled from bottom to top in the texture space. Therefore, we use a *block*, which refers to linear pitched packing that divides texture horizontally, where 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 Fig. 7.

If we directly upload updated blocks to the GPU, the whole CPU-based tree would be locked and the painting thread would stall. To avoid this painting interruption, we first copy the block to a staging buffer, designed for CPU-side hazard control, that serves as an update queue. 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 section 5.4 for discussion on hazards), and queue them in the LIFO queues with upload-to-GPU tasks. We then simply stage and upload one block per rendering frame.

To verify that the algorithms we develop 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 $\mathcal{G}_3$ 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 Fig. 8.

On CPU side, table 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

Table 1: *Average staging time (second column), the maximum number of staged blocks (third column), and staged cells (last column) 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

the maximum number of uploaded cells per second is five million cells. The average bandwidth of block size under two million cells is less than 5ms, therefore this 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 5 frames per second (FPS) up to 22 FPS, this rate is still not acceptable. The next bottleneck is neighbor connectivity update, which will be discussed in the next section.

5.3. Neighbor Computation Mask

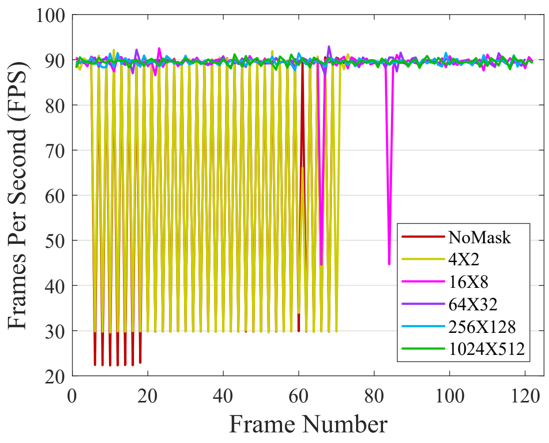
Even though a neighbor computation on the GPU corresponds to a simple computation of the graph $\mathcal{G}_3$, the resolution of the texture can be large (16384×8192) which affects the interactivity (See the no mask case in Fig. 8(a)). Therefore, we develop a simple and efficient method to dramatically reduce this rendering cost.

Once a cell is created or deleted, not only the cell but also its neighbors should be updated in $\mathcal{G}_3$. Since neighbors may not be inside a block that contains the cell, updating $\mathcal{G}_3$ within the block would not be sufficient. One solution would be to have an additional list of expanded blocks for 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.

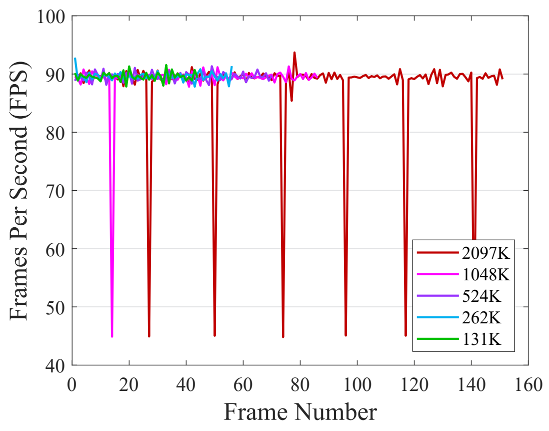
We tested various mask sizes. If the size of the neighbor computation mask is over 64 by 32 (which takes only 256 bytes), the overall frame rate stays around 90 FPS, which is a lower bound of frame rate, as shown in Fig. 8(a). We also further tested the degree to which the size of blocks affects the frame rates with lightweight neighbor precomputation cost. Fig. 8(b) shows that the frame rates stay stable when the number of cells per block is under 524K.

5.4. Immediate Visual Feedback

We write and read the tree simultaneously in painting and staging threads. To minimize waits between painting and staging threads, we divide the painting task into finer grained jobs. We adjust the tree by one level in the stroke thread (Fig. 4). This ensures that the CPU-side tree is valid without orphan cells or balancing violations.



(a) Performance results with different sizes of neighbor computation mask (excluding effect of the block size). No neighbor computation mask (no mask) or a low resolution mask (the mask size of 4x2) does not show the stable frame rate because of heavy neighbor computation. When the resolution of a neighbor computation mask increases, the frame rate becomes more stable.

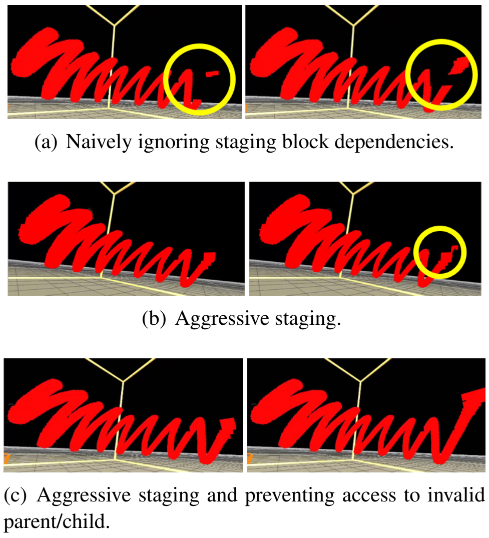


(b) Performance results with staged blocks of different sizes (excluding the effect of neighbor precomputation). While the neighbor computation is reduced enough, the size of staged blocks still affects the frame rate.

Figure 8: *Performance tests with two controlled variables, the size of block and granularity of neighbor computation mask. In order to observe the effects of each independent variable, (a) we fixed the block size to contain 262K cells and (b) we fix the neighbor computation mask to 64 by 32 and update the total 57K cells with a single stroke.*

Staging blocks may depend on each other and hence dependent blocks may be grouped, uploaded to GPU-side staging buffers, and then copied together in GPU at the beginning of rendering. However, in our experiment, 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. Therefore, we experiment strategies to ignore dependency between staging blocks. As a result, artists can see the strokes at 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 our ray caster that assumes 2:1 balancing. Second, since parent-child referencing is valid only inside a block, some of inter-block parent and child indices can be invalid. This causes visible artifacts. We now design two strategies that reduce these artifacts: aggressive staging and handling invalid parent/child. For the purpose of this experiment, we develop a stroke replay system to render frames with and without corruptions. We then compute the number of rendered frames that have mismatching pixels.

By just ignoring inter-staging buffer dependency, corruption occurs in 33.3% of the total frames. The corruption lasts up to 450ms. Aggressive staging is to use smaller atomic sections that can even temporally break the CPU-side 2:1 balancing to 3:1. However since staging happens earlier, the corruption goes away faster. 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.9).


(a) Naively ignoring staging block dependencies.

(b) Aggressive staging.

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

Figure 9: *Frames with corruptions (yellow circle). (a) Artifacts occur rather frequently (33.3% of frames have corrupted pixels), and corruptions remain for 0.45s. (b) 21.7% frames have corruptions. (c) Only 3.62% of frames have corrupted pixels. Corruption disappears within 0.03s.*

6. Accurate Volume Rendering with High-depth Octree

Rendering 3D volumetric fields poses another challenge. One viable solution involves extracting voxel faces and rendering them through raster graphics pipeline using, for example, OpenGL similarly to Minecraft [MOJ17]. 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. Consequently, we explore an alternative approach of ray casting through octree volumetric field.

6.1. Accurate Ray Starting Point

When editing fine detail, users should be able to zoom in to observe the cells that have the highest depth (e.g. 24). However, the size of

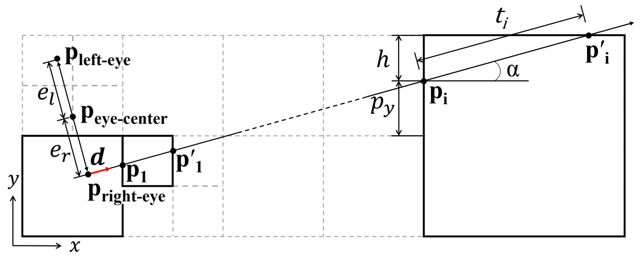


Figure 10: *A 2D illustration of ray casting on adaptive grids. A ray is fired from the right eye of the user at* $\mathbf{p}_{right\text{-}eye}$ *along* $\mathbf{d}$*, and the first cell-entry and exit points* $\mathbf{p}_1$ *and* $\mathbf{p}\prime_1$*, respectively, are calculated for the ray. This process continues for other cells until the ray is terminated.*

these tree cells can 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 towards nearby floating point values, 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. We propose computing the ray starting point in a cell-local coordinate frame, and keep the positioning error of the starting point sufficiently small for the above reasons. Our cell-local coordinate system is a barycentric coordinate system that has the origin at the cell center with a size of one. 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 of depth 24, its resolution inside the leaf becomes extremely high.

6.2. Accurate Ray Traversal

As illustrated in Fig. 10, given a ray and its direction $\mathbf{d}$, and a cell-entry point $\mathbf{p}_i$ of the ray into the i^{th} cell, we compute the cell-traversal distance t_i and the cell-exit point $\mathbf{p}_i\prime$ as well as a neighboring cell containing $\mathbf{p}_i\prime$. Since our volumetric canvas covers a large space and the cell sizes vary by a large magnitude, using a global coordinate system to calculate $\mathbf{p}_i\prime$ and the ray traversal length t can be inaccurate. In contrast, the cell-local coordinate system can produce accurate results regardless of the zoom level. We represent $\mathbf{p_i}$ and $\mathbf{p}\prime_i$ 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 $\mathbf{p}_i\prime$ and the neighbor texture described in section 4.3, we choose the neighbor cell (the $i+1^{\text{th}}$ cell) to visit, and set $\mathbf{p}_i\prime$ to $\mathbf{p}_{i+1}\prime$. This process is repeated until the ray terminates after accumulating full opacity or exits the canvas.

6.3. Analysis

As illustrated in Fig. 10, 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 $\mathbf{p}_i$, and d_y is the y component of $\mathbf{d}$. The error in t_i will be proportional to t_i. Take the machine epsilon $\varepsilon = 2^{-23}$ for single precision. Let x, y be arbitrarily accurate real numbers, and $f(x)$ be a floating point representation of x. Then

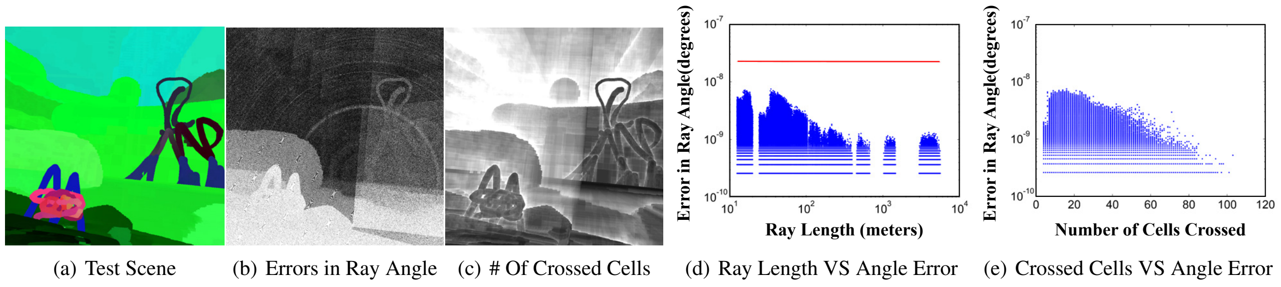


Figure 11: *A ray traversal using local coordinate system. The maximum level of the octree reaches to 24. (a) Test Scene, (b) Error angle between the final ray location and the initial ray direction, magnified by 10^9 for display in gray. (c) The number of cells crossed in gray. (d) and (e) show that the ray-angle error does not grow as the ray traverses. In (d), the initial error e_0/L is dominant when L is small. In (e), as the number of cells crossed by the ray increases, the stochastic decay increases. The red line in (d) is the worst case error bound.*

$f(x+y) = (x+y)(1+\varepsilon_+)$, with some $\varepsilon_+ \le \varepsilon$. Similarly, the error in t_i is computed as

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

Note that ignoring $\varepsilon_1\varepsilon_2$, we have $\varepsilon_t \le \varepsilon$. We then compute $f(\mathbf{p}_i\prime) = f(\mathbf{p}_i + f(t_i)) = f(\mathbf{p}_i + t_i(1+2\varepsilon_t)) = (\mathbf{p}_i + t_i(1+2\varepsilon_t))(1+\varepsilon_3)) = (\mathbf{p}_i + t_i)(1+3\varepsilon_p)$, for some $\varepsilon_p \le \varepsilon$, ignoring $\varepsilon_3\varepsilon_t$. Next, we transform the coordinates of $f(\mathbf{p}_i\prime)$ to the neighboring $i+1^{\text{th}}$ cell where the ray continues. This point is computed as $\mathbf{p}_{i+1} = f(f(f(\mathbf{p}_i\prime)s)+c) = ((\mathbf{p}_i+t_i)s+c)(1+5\varepsilon_i)$ for some $\varepsilon_i \le \varepsilon$, where scale s and shift c depends on the depth and location. Therefore, $f(\mathbf{p}_{\tilde{i+1}}) = \tilde{\mathbf{p}}_{i+1}(1+5\varepsilon)$, where $\tilde{\mathbf{p}}_{i+1}$ is the exact value computed from $\mathbf{p}_i$. Thus, the numerical error added during the traversal point is proportional to the coordinate values, the number of floating point operations 5, and ε.

Since we are using a cell coordinate system, each coordinate of $\mathbf{p}_i$ 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 $\mathbf{p}_{\text{right-eye}}$ in the cell-local coordinate frame. Starting from this initial error e_0, traversing n cells results in total error $e_0 + \sum_{i=0}^{n} 2.5\varepsilon w_i = e_0 + 2.5\varepsilon\sum_{i=0}^{n} w_i \le e_0 + 2.5\sqrt{3}\varepsilon L < e_0 + 5\varepsilon L$, where L is the ray length. Note that $\sum_{i=0}^{n} w_i \le \sqrt{3}L$. 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 $\sin^{-1}(e_0/L+5\varepsilon)$ 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 of the length of the ray L. See Fig. 11(d).

To verify our analysis, we performed an experiment, as demonstrated in Fig. 11. We show that the screen-space ray-deviation from 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), and then slightly decays as L increases. This experimental result implies that we can perform the ray casting even on mobile GPUs with only half-precision floats ($\varepsilon = 1/1024$) using fragment shaders. We compare the results using the world coordinate and local coordinate in Fig. 12.

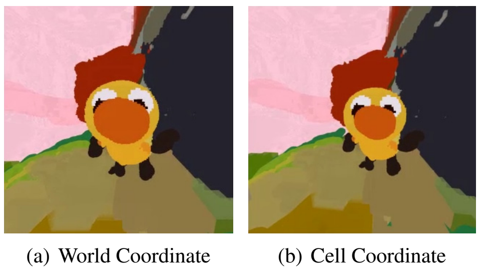


Figure 12: *When a world coordinate system was used, the ray drifts while traversing cells even with regular 32-bit floats. This results in a large angle error (a). We show that this problem is solved by using cell-local coordinates (b).*

7. Extending 2D Digital Painting to 3D Volumetric Painting

In 2D painting, artists apply brush strokes fast and react to immediate feedback. Modern CPUs can handle reasonably large brush sizes of up to a few hundred pixels at interactive rates, but for a larger brush, interactivity starts to diminish. This can be quite restrictive to artists since the artist 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. A solution is to use adaptive brush strokes, where we refine the grid only up to a resolution 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 Fig. 1 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 paper. Fortunately, color flow, deposition, and mixing methods developed in 2D digital painting applications are directly applicable to our 3D volumetric painting. In this study, we implement the popular per-stroke maximum blending mode found in painting applications [Inc16] to extend 2D digital painting to 3D. Note that many other alternatives are available, such as physically-based pigment mixing using Kubelka-Munk mode [BWL04], RYB mixing [CKIW15], or advanced RGB-space color mixing [LDC*14].

2D brush stamping methods (another orthogonal topic) can also be directly applied to 3D painting. We currently support multiple stamp shapes: sphere, cylinder, box, 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 [DiV13]).

Another common 2D painting practice is color pick-up. We implement a simple color pick-up method 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 Fig. 2 (a), surface-based painting systems such as Tilt Brush do not have color mixing between strokes. In contrast, color pick-up automatically generates spatially-varying colors, and this is a popular method that artists use to generate color gradation in painting as shown in Fig. 2 (b).

8. Results and Discussion

We use C++ programming language with Visual Studio 2015, OpenGL 4.3, OpenVR, and the Grizzly library [KTHS15] under Windows 10 OS to implement our system. Our hardware setup includes Nvidia GTX Titan Xp GPU and Intel Core i7-4790 CPU with 16GB RAM for rendering computation and HTC Vive for interfacing immersive and personal VR environments.

8.1. Volumetric Painting Results

We invited digital-painting artists to produce volumetric paintings. Island (Fig. 13) represents a good example as an extension of 2D painting to 3D. The artist painted the scene with three distinct locations, with a large sky background. Semi-transparent objects such as cloud and smoke and detailed objects such as wires and lamb also show the benefits of our system. The artist uses recolor mode and color mix mode for adding rough shades on buildings and on the island. Similar to Island, the artist explores a much larger painting space in Flying Dragons (Fig. 15). Rougher and more colorful shades on the dragon's body-surface were used, but fine details on the eyes, teeth, and horns were maintained.

Our system shows development possibility to a serious 3D digital painting tool as well. Each images of A Sneaker (Fig. 1(a)) seems like the results of 2D digital painting, except that the painting can be appreciated from any view point. Another example with various size of objects is Fig. 14. In Music (Fig. 1(b)), the artist not only did a digital painting, but also developed a story of full 3D scene with retouched chipmunks, tiny bees with a detailed score, and translucent, green spring haze. From this aspect, we anticipate that many 2D digital artists can extend their existing techniques to 3D.

The artists interestingly 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 remark is very interesting, as it appears to be the result of being able to paint in very large 3D canvas. We believe future explorations with artists will reveal how perspectives can be painted on 3D canvases. For example, recoloring remote mountains from a foreground location would be an interesting approach.

8.2. Qualitative Comparisons of Using Dynamic Deep Octree for Volume Painting

For volumetric painting, we need to 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 [Mus13, LSK*06] are therefore 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 priory requirement is hard for artists 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, but not a key requirement in existing dynamic tree update techniques. In simulation problems [Hoe16, SABS14], trees are updated globally since they are often adjusted based on velocity, smoke, proximity to liquid surface, or details on the liquid surface. Also, rendering is not required and the frame rate is less demanding than the VR painting applications. In another study [LSK*06, CNS*11], heavy updating was allowed in a GPU-only dynamic tree, where the CPU cannot be involved in painting. Because Octomap [HWB*13] does not require immediate visual feedback, a low latency visualization method has not been studied, and thus Octomap cannot be used for volume painting.

8.3. Limitations

While painting a large stroke over a detailed complex area, a large number of octree cell should be deleted. In this case, although the frame rate is still constant, the delay can be quite large. An indicator would notify that a heavy-weight operation is taking place. Our system can also eventually suffer from out-of-memory. Although the memory size available in modern GPU is increasing over time, artists can indeed consume all the GPU memory. This can be greatly relieved in future automatically by using topology-cleaning operations. In addition, a memory plan-ahead interface would be required. Digital artists usually create multiple layers. Therefore multiple layer support that compromises memory and performance would be required. Finally, since artists spend long time wearing VR headset, the weight of headset is currently the dominating discomfort factor. One way of reducing such discomfort is to move some tasks, such as recoloring, from VR to conventional 2D monitor.

9. Conclusions

We proposed a *volumetric VR 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.

To achieve this goal, we used octree with high depth and perform ray casting to render the volume. We used a CPU to implement dynamic tree adjustment, and proposed low latency update methods that keep the rendering frame rate at highly interactive rates. We showed that small staged blocks and neighbor computation masks maintain the system performance while the latency and artifacts are well suppressed. To reduce memory footprint, we showed that three neighbors per cell are sufficient for efficient neighbor access in an octree. Finally, we provided a ray-traversal error bound using posterior error analysis and verified the bound with experiments.

In future, we intend to explore various volume-specific painting models, user interfaces, and combining these with geometry-based paintings to shift to a more complete 3D volume painting system. Error analysis suggests that ray-casting can be performed even in half-float precision of $\varepsilon = 1/1024$. Therefore, we plan to examine the performance and accuracy implications of using half precision for ray casting and tracing applications.

Acknowledgements

We appreciate all artists who evaluate our system, especially Daichi Ito, Jini Kwon, Yun-hyeong Kim, and Jaehyun Kim for their paintings in this paper. This project was supported in part by the NRF in Korea (2017R1A2B3012701) and Adobe gift funds.

References

[BD02] BENSON D., DAVIS J.: Octree textures. *ACM Transactions on Graphics (TOG) 21*, 3 (July 2002), 785–790. 3

[BNS01] BOADA I., NAVAZO I., SCOPIGNO R.: Multiresolution volume visualization with a texture-based octree. *The Visual Computer 17*, 3 (May 2001), 185–197. 3

[BWL04] BAXTER W. V., WENDT J., LIN M. C.: IMPaSTo: A realistic, interactive model for paint. In *Proceedings of the International Symposium on Non-Photorealistic Animation and Rendering (NPAR)* (June 2004), Spencer S. N., (Ed.), pp. 45–56. 10

[CBI13] CHEN J., BAUTEMBACH D., IZADI S.: Scalable real-time volumetric surface reconstruction. *ACM Transactions on Graphics (TOG) 32*, 4 (July 2013), 113:1–113:16. 3

[CKIW15] CHEN Z., KIM B., ITO D., WANG H.: Wetbrush: Gpu-based 3d painting simulation at the bristle level. *ACM Transactions on Graphics (TOG) 34*, 6 (Oct. 2015), 200:1–200:11. 10

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)* (Feb. 2009). 3

[CNS*11] CRASSIN C., NEYRET F., SAINZ M., GREEN S., EISEMANN E.: Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum (Proceedings of Pacific Graphics 2011) 30*, 7 (Sep. 2011). 3, 10

[DGPR02] DEBRY D. G., GIBBS J., PETTY D. D., ROBINS N.: Painting and rendering textures on unparameterized models. *ACM Transactions on Graphics (TOG) 21*, 3 (July 2002), 763–768. 3

[DiV13] DIVERDI S.: *A Brush Stroke Synthesis Toolbox*. Springer London, London, 2013, pp. 23–44. 10

[DKB*16] DADO B., KOL T. R., BAUSZAT P., THIERY J.-M., EISEMANN E.: Geometry and attribute compression for voxel scenes. *Computer Graphics Forum 35*, 2 (2016), 397–407. 3

[DSKA18] DOLONIUS D., SINTORN E., KÄMPE V., ASSARSSON U.: Compressing color data for voxelized surface geometry. *IEEE Transactions on Visualization and Computer Graphics (TVCG)* (2018). 3

[EHK*06] ENGEL K., HADWIGER M., KNISS J., REZK-SALAMA C., WEISKOPF D.: *Real-time volume graphics*. CRC Press, 2006. 3

[FPRJ00] FRISKEN S. F., PERRY R. N., ROCKWOOD A. P., JONES T. R.: Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (2000), SIGGRAPH '00, ACM Press, pp. 249–254. 3

[GMG08] GOBBETTI E., MARTON F., GUITIÁN J. A. I.: A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer 24*, 7-9 (2008), 797–806. 3, 5

[Goo15] GOOGLE: Tilt brush by google. https://www.tiltbrush.com/, 2015. 2

[HBJP12] HADWIGER M., BEYER J., JEONG W., PFISTER H.: Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach. *IEEE Transactions on Visualization and Computer Graphics 18*, 12 (Dec 2012). 3

[HBv98] HAVRAN V., BITTNER J., ŽÁRA J.: Ray tracing with rope trees. In *14th Spring Conference on Computer Graphics* (1998), pp. 130–140. 3, 5

[HLSR09] HADWIGER M., LJUNG P., SALAMA C. R., ROPINSKI T.: Advanced illumination techniques for gpu-based volume raycasting. In *ACM SIGGRAPH 2009 Courses* (2009), SIGGRAPH '09, ACM, pp. 2:1–2:166. 3

[Hoe16] HOETZLEIN R. K.: Gvdb: Raytracing sparse voxel database structures on the gpu. In *Proceedings of High Performance Graphics* (2016), HPG '16, Eurographics Association, pp. 109–117. 3, 10

[HVP02] HAKKINEN J., VUORI T., PAAKKA M.: Postural stability and sickness symptoms after hmd use. In *IEEE International Conference on Systems, Man and Cybernetics* (2002), vol. 1, pp. 147–152. 5

[HWB*13] HORNUNG A., WURM K. M., BENNEWITZ M., STACHNISS C., BURGARD W.: Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots 34*, 3 (2013), 189–206. 3, 10

[Inc16] INCORPORATED A. S.: Adobe photoshop user guide, 2016. URL: http://www.photoshop.com/. 9

[Ize13] IZE T.: Robust bvh ray traversal-revised. *Journal of Computer Graphics Techniques (JCGT) 2*, 2 (2013), 12–27. 3

[JLSW02] JU T., LOSASSO F., SCHAEFER S., WARREN J.: Dual contouring of hermite data. *ACM Transactions on Graphics (TOG) 21*, 3 (2002), 339–346. 3

[JMH15] JOHANNA B., MARKUS H., HANSPETER P.: State-of-the-art gpu-based large-scale volume visualization. *Computer Graphics Forum 34*, 8 (2015), 13–37. 3

[Kat17] KATAOKA D.: Art and virtual reality, new tools, new horizons. Silicon Valley VR Expo., 2017. 1

[KBSS01] KOBBELT L. P., BOTSCH M., SCHWANECKE U., SEIDEL H.-P.: Feature sensitive surface extraction from volume data. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2001), SIGGRAPH '01, ACM, pp. 57–66. 3

[KH13] KAZHDAN M., HOPPE H.: Screened poisson surface reconstruction. *ACM Transactions on Graphics (TOG) 32*, 3 (2013), 29. 3

[KLS*05] KNISS J., LEFOHN A., STRZODKA R., SENGUPTA S., OWENS J. D.: Octree textures on graphics hardware. In *ACM SIGGRAPH 2005 Sketches* (2005), SIGGRAPH '05, ACM. 3

[KTHS15] KIM B., TSIOTRAS P., HONG J., SONG O.: Interpolation and parallel adjustment of center-sampled trees with new balancing constraints. *The Visual Computer 31*, 10 (2015), 1351–1363. 4, 10

[KW03] KRUGER J., WESTERMANN R.: Acceleration techniques for gpu-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (2003), IEEE Computer Society, p. 38. 3

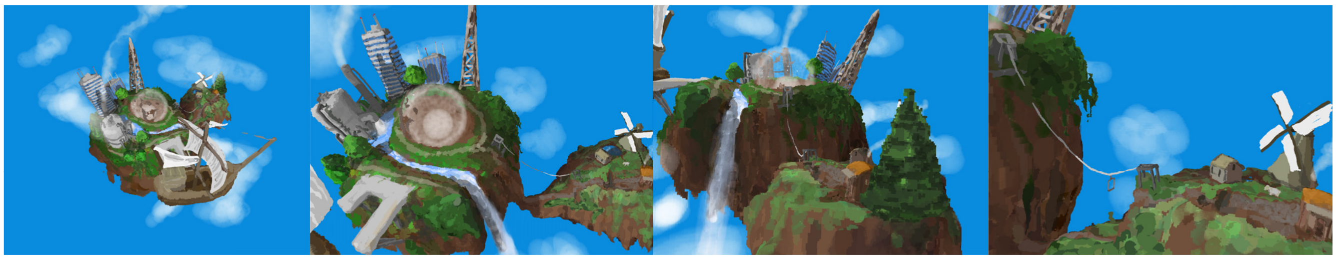


Figure 13: *"Island" from different view points.*

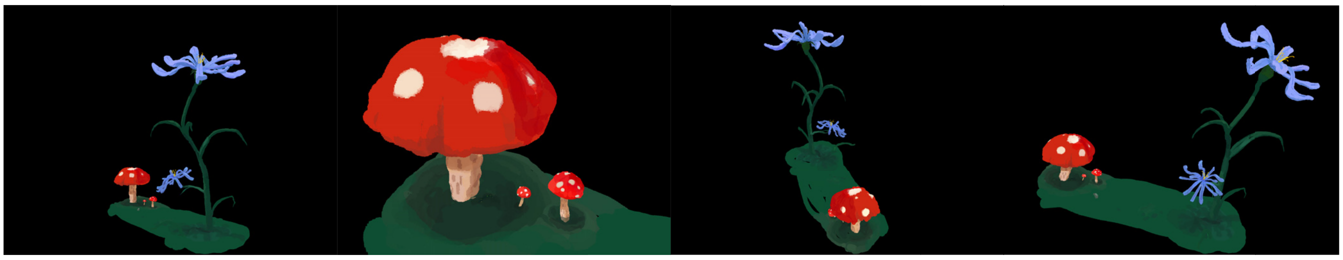


Figure 14: *"Nature" from different view points.*

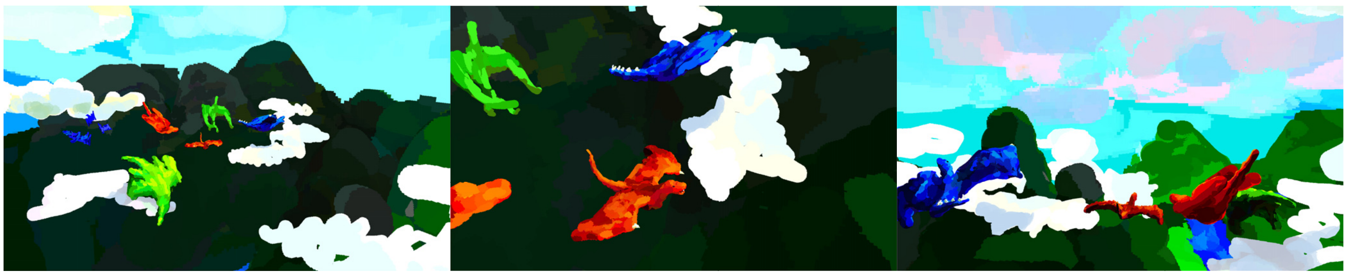


Figure 15: *"Flying dragons" from different view points.*

[LDC*14] Lu J., DiVerdi S., Chen W., Barnes C., Finkelstein A.: RealPigment: Paint compositing by example. *NPAR 2014, Proceedings of the 12th International Symposium on Non-photorealistic Animation and Rendering* (Jun 2014). 10

[LGF04] Losasso F., Gibou F., Fedkiw R.: Simulating water and smoke with an octree data structure. *ACM Transactions on Graphics (TOG) 23*, 3 (Aug. 2004), 457–462. 3

[LHN05] Lefebvre S., Hornus S., Neyret F.: Octree textures on the gpu. In *GPU Gems 2*, Pharr M., (Ed.). Addison-Wesley, 2005, pp. 595–613. 3

[LK10] Laine S., Karras T.: Efficient sparse voxel octrees. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2010), I3D '10, ACM, pp. 55–63. 3, 5

[LSK*06] Lefohn A. E., Sengupta S., Kniss J., Strzodka R., Owens J. D.: Glift: Generic, efficient, random-access gpu data structures. *ACM Transactions on Graphics (TOG) 25*, 1 (2006), 60–99. 3, 5, 10

[MB90] MacDonald J. D., Booth K. S.: Heuristics for ray tracing using space subdivision. *The Visual Computer 6*, 3 (1990), 153–166. 3, 5

[MOJ17] Mojang: Official site | minecraft. https://minecraft.net/en-us/?ref=m, 2009-2017. 8

[Mus13] Museth K.: Vdb: High-resolution sparse volumes with dynamic topology. *ACM Transactions on Graphics (TOG) 32*, 3 (2013), 27. 3, 10

[Ocu16] Oculus: Quill by story studio. https://storystudio.oculus.com/en-us/, 2016. 2

[PGSS07] Popov S., Günther J., Seidel H.-P., Slusallek P.: Stackless kd-tree traversal for high performance gpu ray tracing. *Computer Graphics Forum 26*, 3 (2007), 415–424. 3, 5

[PWP06] Patterson R., Winterbottom M. D., Pierce B. J.: Perceptual issues in the use of head-mounted visual displays. *Human factors 48*, 3 (2006), 555–573. 5

[Reg95] Regan C.: An investigation into nausea and other side-effects of head-coupled immersive virtual reality. *Virtual Reality 1*, 1 (1995), 17–31. 5

[RTW13] Reichl F., Treib M., Westermann R.: Visualization of big sph simulations via compressed octree grids. In *2013 IEEE International Conference on Big Data* (2013), pp. 71–78. 3

[SABS14] Setaluri R., Aanjaneya M., Bauer S., Sifakis E.: Spgrid: A sparse paged grid structure applied to adaptive smoke simulation. *ACM Transactions on Graphics (TOG) 33*, 6 (2014), 205. 10

[Sam89] Samet H.: Implementing ray tracing with octrees and neighbor finding. *Computers & Graphics 13*, 4 (1989), 445–460. 3, 5

[ZGHG11] Zhou K., Gong M., Huang X., Guo B.: Data-parallel octrees for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics 17*, 5 (2011), 669–681. 3