VoxSculpt: An Open-Source Voxel Library for Tomographic Volume Sculpting in Virtual Reality

Lucas Siqueira Rodrigues
Cluster of Excellence "Matters of Activity"
Humboldt-Universität zu Berlin
Berlin, Germany
[0000-0001-7675-136X]

Felix Riehm
Angewandte Informatik
Hochschule für Technik und Wirtschaft Berlin
Berlin, Germany
felix.riehm@student.htw-berlin.de

Stefan Zachow
Visual and Data-Centric Computing
Zuse Institute Berlin
Berlin, Germany
[0000−0001−7964−3049]

Johann Habakuk Israel
Forschungsgruppe CENTIS
Hochschule für Technik und Wirtschaft Berlin
Berlin, Germany
[0000-0002-8513-6892]

***Abstract*—Manual processing of tomographic data volumes, such as interactive image segmentation in medicine or paleontology, is considered a time-consuming and cumbersome endeavor. Immersive volume sculpting stands as a potential solution to improve its efficiency and intuitiveness. However, current open-source software solutions do not yield the required performance and functionalities. We address this issue by contributing a novel open-source game engine voxel library that supports real-time immersive volume sculpting. Our design leverages GPU instancing, parallel computing, and a chunk-based data structure to optimize collision detection and rendering. We have implemented features that enable fast voxel interaction and improve precision. Our benchmark evaluation indicates that our implementation offers a significant improvement over the state-of-the-art and can render and modify millions of visible voxels while maintaining stable performance for real-time interaction in virtual reality.**

Keywords—voxel library, volume sculpting, virtual reality

I. Introduction

Tomographic imaging techniques, such as CT, MRI, and ultrasound, are important in a variety of fields including medicine, paleontology, and engineering [1]. Advances in this area have enabled the non-destructive exploration of internal structures, providing valuable information for practitioners in different fields. Material separation and labeling of relevant structures of interest in processes such as image segmentation are necessary to provide an understanding of such data. Despite recent developments in the automated segmentation of tomographic data volumes, manual processes are necessary for complex scenarios that require human expertise [2]. Interactive image segmentation of tomographic data, including volume editing, is cumbersome as current solutions utilize slice-based 2D tools for interacting with 3D data [3]. Volume sculpting poses as a potential interaction metaphor that could improve the efficiency of interactive image segmentation. Virtual reality would be an evident medium for volume sculpting as it enables *direct manipulation* in the 3D space and may support better understanding and decision-making in such tomographic imaging operations [4][5].

Since tomographic imaging practitioners engage in workflows that are unique to their fields, custom tools must be designed to cater to their specific requirements. Game engines are widely regarded as the most practical and cost-effective instrument to enable researchers in designing custom solutions [6]. However, current game engine voxel libraries do not yield the performance and functionalities necessary to enable volume sculpting in virtual reality. The present study aims to address this gap by answering the following research question: "Can a novel open-source game engine voxel library enable volume sculpting in virtual reality?". In the following section, we describe the state-of-the-art and its limitations to the proposed application. Next, we describe the design of a novel open-source game engine voxel library that supports real-time immersive volume sculpting. Finally, we evaluate our design through benchmark comparison against the state-of-the-art and discuss the impact of our contribution.

II. Background

Relevant literature ranges at different points of significance to our research question's criteria. Most contributions focus on displaying tomographic data in VR scenes, but their interactivity is limited to scene visualization and transformation [7]. Faludi et al. present a method for the direct visual and haptic rendering of volumetric medical data sets in virtual reality [8]. King et al. created an application that renders CT scans through the communication between 3D Slicer and *Unity* [9]. Wheeler et al. employed OpenGL context sharing to display VTK objects in a *Unity* scene [10]. Other works enable volume editing in other specific contexts but are not open-source libraries. Rizzi implemented volume deformation within an immersive surgical simulation [11]. Reddivari and Smith created a visualization tool for MRI images that allows for volume editing [12]. Zhang et al. incorporated haptic feedback into their surgery simulator by using *Unity* and *Nvidia Flex* [13]. Other works implement direct volume rendering, which is used widely in medical visualization tools [14] [15] [16]. Escobar-Castillejos et al. describe an architecture to build a visuo-haptic application as a *Unity* plugin [17]. In terms of open-source solutions, Williams introduces *Cubiquity*, a voxel engine that shares many similarities with the popular *Minecraft* game [18]. *Cubiquity* provides classes to build and manage voxel data, including tomographic image slices. Although platform-independent, *Cubiquity* includes a *Unity* plugin integration.

Amongst the works listed above, *Cubiquity* stands as the closest available alternative to the aforementioned criteria, but literature indicates that the library has performance limitations when supporting real-time voxel interaction in virtual reality. Duncan et al. integrated *Cubiquity* into their intervention and "faced many challenges in the display and manipulation" of a scaled-down "MRI with 256 voxels per dimension" [26]. Similarly, Chheang et al. reported that "low FPS happens during the cutting simulation" as they tracked "collision synchronization noticeably lagging behind the modifications of cutting" [27]. Similar performance issues were described by Rodrigues et al. [19]. Thus, we contribute *VoxSculpt* as a novel voxel library aimed to advance the state-of-the-art.

III. Methodology

In this section, we discuss the design and implementation of a custom voxel library as a proposed solution for virtual reality volume sculpting in the *Unity* game engine. In the following subsections, we describe our library's data structure, volume modifiers, ray casting, physic colliders, and rendering modules. Under *Data Structure*, we present the library's voxel storage, state management, chunk organization, and I/O operations leveraging *Unity Jobs.* Next, we describe the implementation of four *Volume Modifiers*: cutting, restoring, filtering, and undoing. Under *Ray Casting*, we describe how our *VoxSculpt* enables fast volume editing by casting a fixed-size ray to acquire voxels within the probe's reach. Then, *Physics Colliders* are presented as a solution for higher sculpting precision through the detection of collisions between volume voxels and mesh colliders. Finally, *Rendering* describes the use of GPU instancing for optimal performance. *Fig. 1* summarizes our library's design. Subsequently, this section details our implementation.

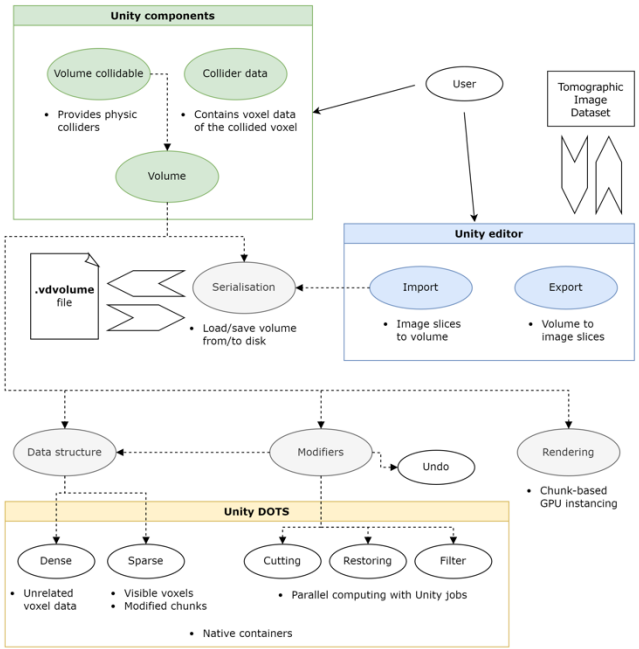

Fig. 1. *VoxSculpt's* implementation concept from a user's perspective.

A. Data structure

In *VoxSculpt's* data structure, voxels are densely stored in a 3D array for permanent storage. Each voxel stores its index space position, 8-bit grayscale color, and state: *Visible*, *Solid*, or *Removed*. State changes do not remove voxels from the data structure. The voxel grid has been designed to be permanently dense in order to support a restoring tool that requires collision detection on removed voxels. An octree implementation was considered, but this approach yielded low performance in *Cubiquity* as octree regeneration upon volume updates and the full-depth traversal are costly operations, especially for dense grids. Meanwhile, the chunk-divided array delivers direct access to volume voxels. In our implementation, chunks are represented by sparse storage [20], composed of multiple supporting hash maps storing visible voxels inside of visible chunks (see *Fig. 2*). More information about the supporting data structures is available in the source code annotations.

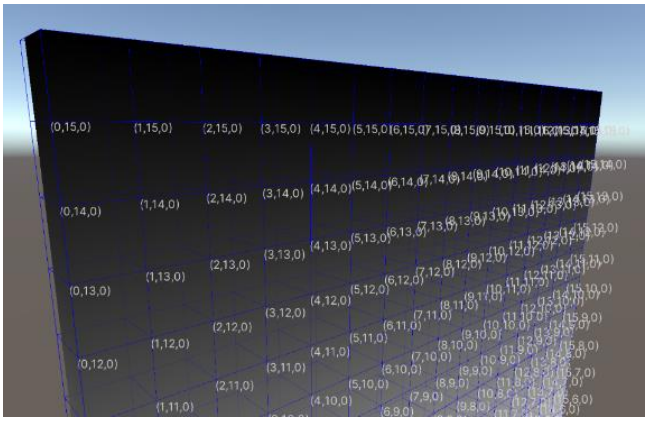

Fig. 2. A 256x256x12 voxel grid divided into 16x16 visible chunks. White labels indicate the index space position (x,y,z) inside the volume.

Chunks enclose voxels inside their cubic shape. When an imported volume is not fully dividable by chunks, our application evens out incomplete chunks by adding empty voxels. Resizing only has an impact on updating the compute buffer described in the *Rendering* subsection. These buffers are updated when modifying the voxel grid and are mapped to the volume's chunks. When scaling down a chunk, the application must iterate over smaller compute buffers, which leads to numerous computationally-expensive I/O operations to the GPU. Oversizing a chunk reduces the number of compute buffers. However, when an operation modifies a chunk, more voxels must be sent to the GPU, which increases I/O operation time.

All data structures used in the dense and sparse storage leverage *Unity.Collections*, which integrates native containers in the *Unity Entities* package. Dense storage implements a 1D *NativeArray*, whereas sparse storage uses a *NativeMultiHashMap* and several regular *NativeHashMaps*. The entire application solely employs 1D arrays. 3D positions from 1D indices, and vice-versa, are calculated at runtime. Starting with (x:0, y:0, z:0), columns are filled first to match *Unity's* positive x-direction, followed by rows, matching the positive y-direction, followed by depth in the positive z-direction. *Unity Jobs'* native collections provide fast memory allocation outside of the managed C# runtime, which prevents garbage collector memory reallocation. Through the allocation

of a single memory block, the data is also cache-coherent, which may result in fewer cache misses during cutting or restoring operations. From a future perspective, 1D arrays are also needed to parallelize the application on a GPU, since frameworks like *Nvidia CUDA* and *OpenCL* demand 1D array input. Native containers only work with non-reference types to enforce safety. Therefore, voxel data is stored inside of a struct. In general, the application avoids creating new C# objects, uses object pooling, and pre-allocates memory whenever possible.

B. Volume Modifiers

VoxSculpt provides four ways to modify a volume: *cutting*, *restoring*, *filtering*, and *undoing*.

1) Cutting and restoring

We define *cutting* as removing a voxel, whereas *restoring* means reestablishing a removed voxel. These operations require the management of voxel states (dense storage) and visible voxels list (sparse storage). *Fig. 3* illustrates the process of *cutting* and *restoring* a single voxel.

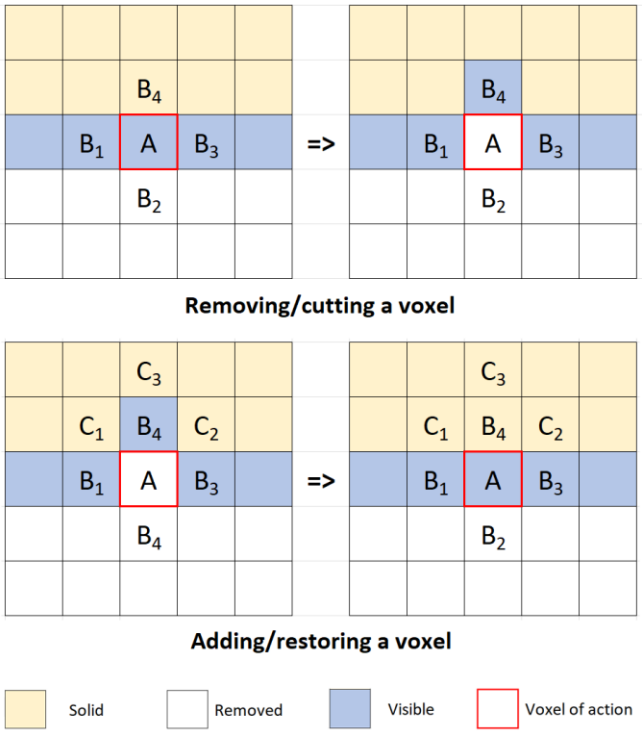

Fig. 3. Illustration of the process when removing or adding a voxel.

In this scenario, *Voxel A* is only removed under the *Visible* state. Otherwise, neighbor voxels (*B1* to *B4*) are checked for the *Solid* state, in which case that voxel becomes visible. The ruleset for adding a voxel is larger to prevent visible voxel artifacts when re-adding voxels. In *Fig. 3*, the bottom-left grid illustrates the process of making *Voxel B4* solid when re-adding *Voxel A*, which would create unnecessary voxel rendering. Therefore, when re-adding *Voxel A*, the neighbors *C1* to *C3* of a neighbor *Voxel B* are additionally checked. If *B4* is *visible*, and if *C1* to *C3* are all *visible* or *solid*, *Voxel B4* becomes *solid*. If one of the neighbors of *Voxel A* (*B1* to *B4*) is *removed*, *Voxel A* becomes *visible*. If *Voxel A* is already *visible*, *Voxel A* becomes *solid* when all neighbors are *solid* or *visible*.

2) Parallel filtering

Filtering enables users to batch-remove voxels that match a specified intensity value range. *Unity Jobs* was leveraged to accelerate through fast thread creation. Worker threads are equally distributed on available CPU to minimize context switching. Threads are reused to avoid recreation, and *Unity Burst's* compiler optimizes each job system for performance. The filtering process is divided into three phases as demonstrated in *Fig. 4*.

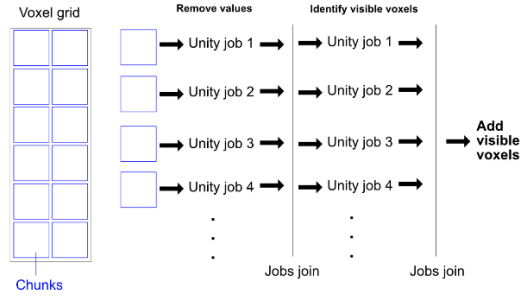

Fig. 4. Process when filtering a value or a value range.

Unity Jobs operates in individual chunks. In the first phase, values are removed from a chunk, modifying the volume's dense storage. In the second phase, voxels that should be made visible after removal are identified. These voxels are added to the volume in the last phase. Threads merge between each phase, except for the first two phases, which are executed in parallel.

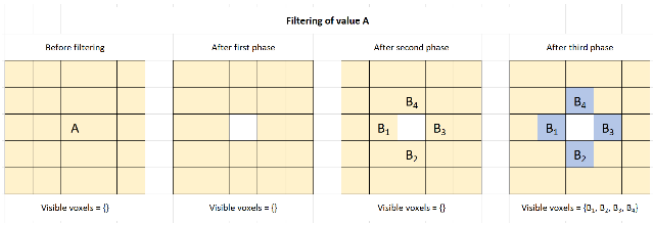

Fig. 5. Data structure at each phase when filtering value A.

Fig. 5 demonstrates the effects of four phases on the data structure. On the leftmost grid, *Value A* stands as a voxel that has been marked for filtering. Next, this voxel is removed from dense storage, leaving a blank value that would violate previously described rules. To address this issue, the next phase identifies voxels that should be made visible (*B1-B4*). The state of these voxels is then set to *visible* in the final phase. In this last stage, these voxels are also added to sparse storage, which holds *visible* voxels that should be rendered.

This process has been divided into discrete phases as parallel work is not possible on sparse storage data structures. Indeed, parallel access to sparse storage hash maps would create race conditions and result in *undefined* behavior. Thread-safe access is achievable, but it would be accomplished by semaphores which are blocking calling threads, leading to performance decreases. Therefore, phase three exists because *visible* voxels cannot be identified in the second phase due to chunk-edge problems (see *Fig. 6*).

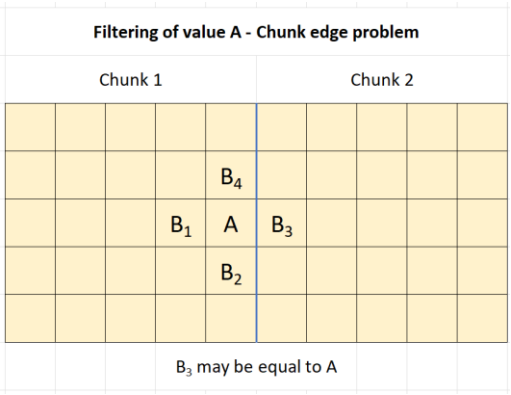

Fig. 6: Chunk-edge problem when filtering a value A.

Fig. 6 shows a problem that can occur when trying to find visible voxels in the second phase. If a job *J1* works on *Chunk 1* while another job *J2* is working simultaneously on *Chunk 2*, *J1* cannot determine if *B3* is a *visible* voxel because *B3* might be removed from job *J2*. Therefore, phase three depends on phase one. Phase three uses a custom data structure for saving identified *visible* voxels because all jobs must write in parallel on the same data structure.

The data structure for identified *visible* voxels storage is conceptionally a 2D array, although the actual implementation utilizes a 1D array. Each column stores 1D indices of identified visible voxels and is mapped to a chunk. Thus, all jobs can simultaneously write to the array. To accelerate adding and iterating processes, the lengths of the stored identified *visible* voxels are also stored. Each column is pre-allocated with the number of voxels inside a chunk. The last phase adds these voxels to the *visible* voxels list.

3) *Parallel cutting and restoring*

For coarse but fast *cutting* and *restoring*, a parallel alternative is implemented. This method currently only works with cubes, but a rectangular cuboid or other primitives could be added. This method divides cubes into layers to feed the jobs with separated data (see *Fig. 7*).

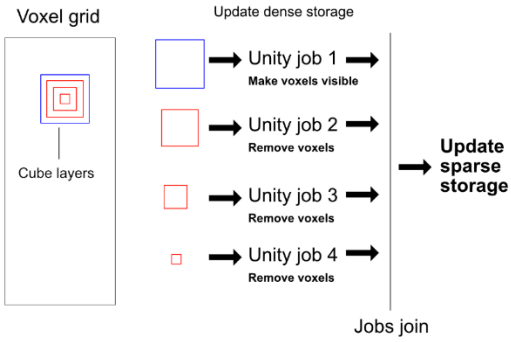

Fig. 7. The parallel cutting process. Blue: Hull layer. Red: Inner layers.

When voxels in the inner layers are removed, the hull layer job adds *visible* voxels to the volume. The process works similarly to the filtering process. When restoring, the inner layer jobs add voxels by making them *solid*, while the hull layer job makes the voxels *visible*.

4) *Undoing*

The *undo* functionality allows users to reverse manual voxel operations. *Removed* or *restored* voxels associated with their respective actions are added as commands in a ring buffer having a size of *262.144* (2^{18}), a limit that can be programmatically adjusted. When users call the *undo* function, recent commands are reversed, beginning with the last command added to the ring buffer and continuing with the most recently added operations until the buffer is emptied.

C. *Ray Casting*

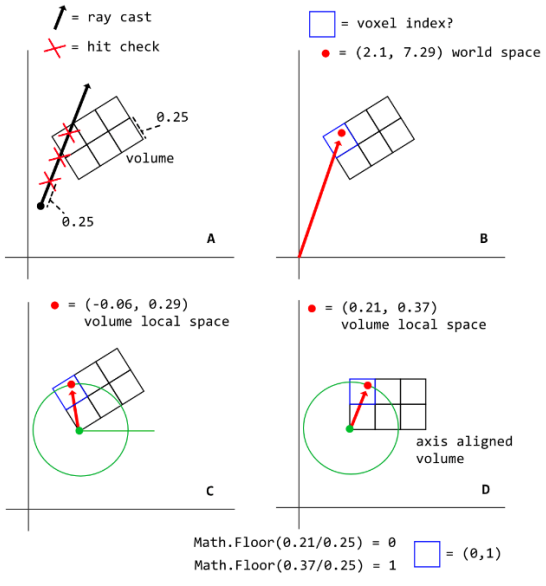

Fig. 8. The process of ray casting.

Fig. 8 illustrates our ray casting implementation: As shown in *Segment A*, a fixed-size step checks if a point is inside a voxel. To check if a point in world space (*B*) is inside a voxel, the script attempts to retrieve the voxel by a voxel index coordinate. To get the voxel index, the world space point is transformed to volume local space (*C*). The volume has its origin in the lower-left corner, which is aligned with the origin of the local coordinate system. After this step, the inverse rotation of the volume is applied to point (*D*). This mimics a situation where a volume would be in an axis-aligned pose. Next, the components of the index coordinate are calculated by dividing each component of the rotated point with the scale of a voxel, followed by applying the *Math.Floor()* function to that result.

D. *Physic Colliders*

When sculpting a volume, users can interact with the volume using *ray casting* or voxel *physic colliders*. Ray casting suffers from inaccuracy as numerous ray casts in multiple directions would be required to fully sample *visible* voxels, and its algorithm would need to delete voxels that enclose the probe, which would only be possible for primitive shapes. For a more accurate and robust solution, we have implemented *physics colliders* to enable the possibility of detecting contact between any mesh collider and volume voxels (see *Fig. 9*). Higher sculpting accuracy can only be achieved by supporting *physics*

colliders. To use this feature, users only need to add a *VolumeCollidable* script to any *Game Object*. This action transforms the entity into a probe that can listen to *Unity's* native *Collision* events and access collided voxel data in its *ColliderData* component.

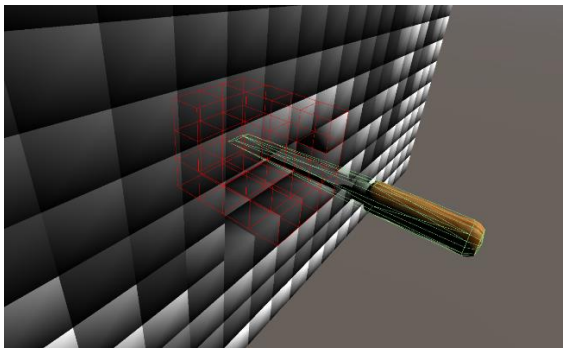

Fig. 9. An example of a custom mesh collider (a chisel) cutting a volume.

A local approach was chosen as adding individual 3D objects (i.e., *Unity game objects*) and colliders for millions of voxels would heavily impact performance. To prevent the creation of an unnecessary number of colliders, *VolumeCollidable* continuously checks for voxels inside the collider's axis-aligned bounding box (*AABB*). Checking is performed by stepping with the size of a voxel over the space of the AABB. If a voxel is detected, the *BoxCollider* component's game object is aligned with the voxel mesh. Game objects of all possible colliders are pre-allocated to create an object pool. For example, for an *AABB* that can enclose a maximum of *16* voxels, 16 game objects are allocated along with their respective box colliders. When aligning a collider, the script uses a game object from this object pool. Only box colliders that are aligned with volume voxels are enabled while the remaining *Box Colliders* are deactivated (see *Fig. 10*).

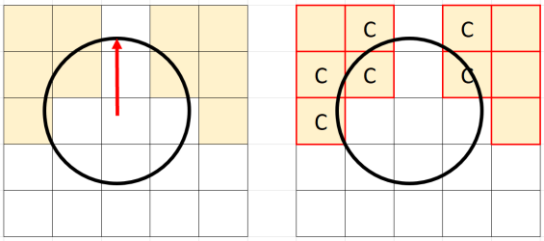

Fig. 10. Collision model comparison. Left: Ray casting method. Right: Collidable method. While ray casting does not detect solid (yellow) voxels, the collidable method detects all voxels colliding with the sphere mesh. Red voxel borders show enabled colliders, and "C" voxels receive collisions.

Unity game objects are never deactivated to prevent performance declines. When aligning a collider, a *ColliderData* component is added to its game object to hold voxel data. *Fig. 11* illustrates this process within a *Unity* scene.

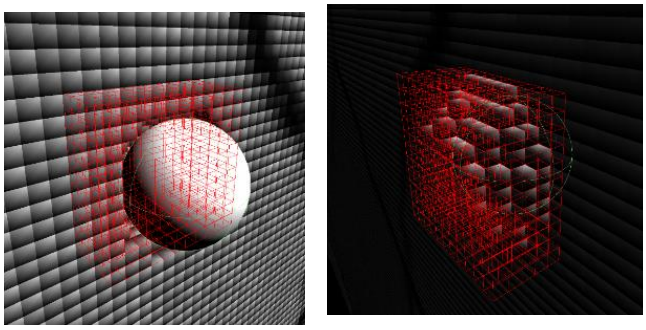

Fig. 11. Visualization of the enabled colliders when cutting the volume with a "VolumeCollidable"-object. Left: The volume viewed from the outside. Right: The same volume as seen from the inside.

E. Rendering

VoxSculpt limits rendering to visible voxels that are stored in sparse storage. Moreover, *GPU instancing* is leveraged to further improve voxel rendering performance (see *Fig. 12).*

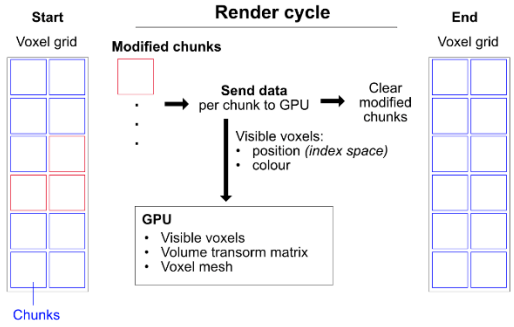

Fig. 12. *VoxSculpt*'s rendering cycle process.

Instancing reduces the amount of data sent to the GPU, which also reduces state switches inside the graphic API. All relevant rendering data is stored on the GPU, which includes voxel position in index space, voxel color, voxel meshes, and the transform matrix for transforming the vertices to the correct world space coordinate. The GPU then draws visible voxels using a shader.

Upon user-initiated volume modifications, the GPU only receives voxel data from chunks that were modified in the last frame in order to maintain stability in the cutting-rendering process. Once the GPU receives this data, modified chunks are reset to their unmodified state. Upon volume initialization, chunks containing visible voxels are marked as modified.

In *VoxSculpt*, the shader not only renders voxels but also translates them to their correct positions. *GPU instancing* optimizes this process through parallel execution, given that the GPU receives the volume transform matrix whenever the volume's *Unity* transform object changes. This matrix not only transforms the voxel mesh's origin to the correct world position, but it also repositions the mesh vertices relatively to that point in the correct coordinate. This is achieved by overwriting the last column in that matrix with the translation vector of the transformed world position of the voxel origin [21]. Without this step, only the world position of the voxel origin would have the same rotation as the volume, but not the voxel mesh when

rotating the volume. *VoxSculpt* fully supports volume scaling, rotation, and translation.

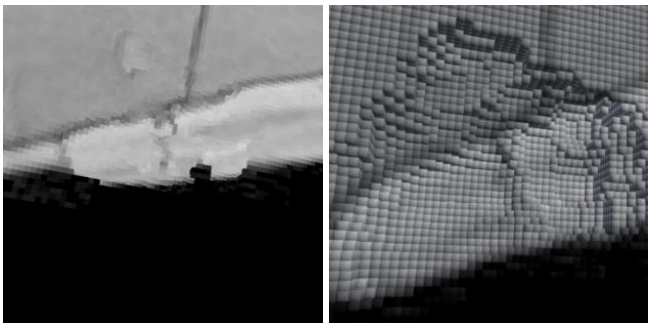

Fig. 13. Left: Voxels rendered with only their color value. Right: Voxels rendered with Phong shading

Our shader applies *Phong shading* to individual voxels to improve visual salience between neighboring voxels. *Fig. 13* compares voxel rendering before (left) and after (right) applying *Phong shading*. Although this effect improves voxel differentiation, one could argue that it distorts original color values that may be of importance. Light intensity and color in the *Unity* scene have an impact on appearance. This shading also introduced screen door and Moiré effects when viewing the image from a certain distance. Activating *8x MSAA* reduced these effects and yielded a cleaner image.

F. Source Code and User Documentation

All relevant code has been evaluated with unit tests and fully commented. We have created user documentation and a *Getting Started* guide with code examples using available *prefabs*, *scenes*, and example *volume data*. The repository can be accessed at: *https://github.com/lsrodri/VoxSculpt*

IV. Results

We present the results of benchmark tests comparing *VoxSculpt* against *Cubiquity*. This evaluation assesses these libraries' performance for different voxel operations and indicates their performance limitations. Benchmark results were extracted from a development build and analyzed with *Unity's* editor profiler in an isolated testing environment. For both voxel libraries, we compared the best benchmark results out of 10 runs. The application build was performed under the following settings: Resolution: *1440x1600*; Quality setting: *high*; *MSAA*: disabled; Default contact offset: *0.0001*; Fixed physic time step: *0.01111*; VSync: *90 fps*. *The physic time step* was set to match the frame rate, which was limited to *90 fps*. The system used for the benchmarks had the following configuration: Intel Core i5-10400 with six cores, each having *2.9* to *4.3 GHz*; *16 GB DDR4-2666 RAM*; *GeForce RTX 3060* with *12GB GDDR6*. According to the *Unity* profiler, memory usage after starting the application and initializing the volume was *1.84 GB*.

A. General Performance Compared to Cubiquity

To compare *VoxSculpt's* and *Cubiquity's* general performance, we integrated both libraries into the benchmarking command line tool to automatize the benchmark process. Table I summarizes how *VoxSculpt* outperforms *Cubiquity* in all 4 operations.

TABLE I: Performance Comparison of VoxSculpt and Cubiquity

Operation/Library	***Cubiquity***	***VoxSculpt***
Initialize (seconds)	39.139	2.275
Delete (seconds)	0.283	0.123
Add (seconds)	0.295	0.145
Modify (seconds)	0.295	0.015

B. Sculpting Performance Compared to Cubiquity

To compare actual sculpting performance, we applied the exact setup and implementation used in *Cubiquity*. Wherever it was necessary. we replaced volume and method calls with their *VoxSculpt* equivalents.

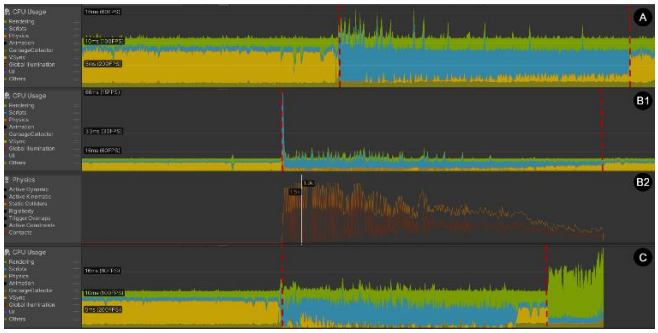

Fig. 14: Cutting performance of the ray cast (A), the collider (B1) with collider count (B2), and the parallel methods (C). Red lines mark the start and end of each benchmark.

1) Ray cast method

We compared cutting in *VoxSculpt* and *Cubiquity* by setting the probe size in both systems to *0.06m*, which removes *4,139* voxels upon collision. *VoxSculpt* delivers a steady *90 FPS* performance, as *Graph A* of *Fig. 14* demonstrates. Moreover, the graph indicates that a larger probe would be feasible under this criterium as the application mostly waits for *VSync*. On the other hand, *Fig. 15* shows that *Cubiquity* cannot deliver stable framerates.

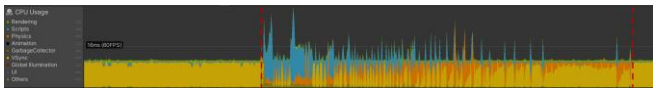

Fig. 15. Cubiquity performance using a 0.06m probe.

2) Collider method

Since the *collider method* has been designed for operations that require higher precision at the expense of lower speed, the probe was rescaled to *0.025m* in both libraries. Under this configuration, *cutting* removes *504* voxels per contact. *Graph B1* in *Fig. 14* shows that the application provides a steady *90 FPS* experience. *Graph B2* shows the number of colliders in the scene at each moment. The scene contained a maximum of about *1,300* enabled colliders and *811* contact points. The apparent jitter is due to the physic time at initialization, which did not occur during collision. The spike at the start of *B1* comes from initializing the object pool of *collidable* objects. Since changing the size of the *collidable* object also results in a different number

of possible colliders, this must be done when changing the size of the collidable or volume.

3) Parallel cutting

Parallel cutting uses ray casting for collision detection. We scaled a cubic probe to *0.1m*, which removes *35,937* voxels per collision. Despite this higher voxel count, this method still provided a solid *90 FPS* experience as shown in *Graph C* of *Fig. 14*. Meanwhile *Cubiquity* performs at *50 FPS* under similar conditions. The graph also shows that there are no performance spikes, except for the period where the profiler was set in the background to stop the recording in *Unity's* main window, which is irrelevant to this measure.

C. Performance of Undo, Filtering, and Volume Initialization

We replicated the setup and implementation of the previous benchmark test to gather data about the performance of the *undo*, *filtering*, and *initialization* operations.

1) Undo

As described earlier, the *undo* operation reestablishes all voxels in the buffer. In this test, the voxel buffer size for the undo operation varied. The *undo* operation was executed after *cutting* the volume with a *0.05m* probe through the *ray cast method*. Results are presented in Table I.

TABLE II: PERFORMANCE RESULTS OF THE UNDO OPERATION

Voxel buffer size	131,072	262,144	524,288	1,048,576
Time (seconds)	0.05	0.10	0.19	0.37

2) Filtering

We executed two *filtering* operations utilizing different value ranges. We reinitialized the volume before each operation. The results are summarized in Table III.

TABLE III. PERFORMANCE RESULTS OF THE FILTERING OPERATION

Value Range	0-42	0-255
Time (seconds)	1.13	1.24

3) Initialization

The *initialization* test includes rendering and reading our proprietary volume file. Volume *initialization* was performed at the *6:59 second mark.*

D. Performance Limitations

This section details circumstances under which *VoxSculpt* is unable to render a voxel volume at *90 FPS*. The setup and implementation from previous sections were used for all *cutting* performance tests in this section.

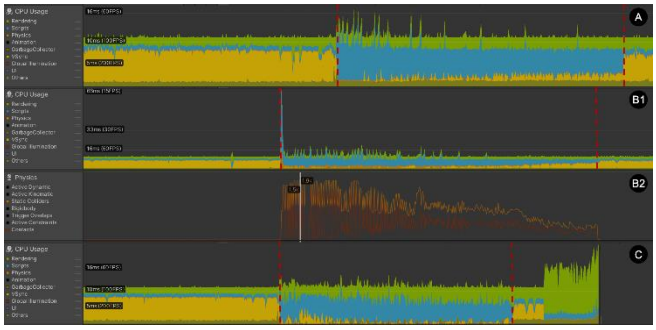

Fig. 16. Performance limitations of the ray cast, collider, and parallel cutting methods. Red lines mark the start and end of each benchmark.

1) Ray cast method

Graph A in *Fig. 14* shows the ray cast method's *cutting* performance limitation when setting the probe scale to *0.087m*, which removes *11,459* voxels per cut. The limiting factor of this method is the iterative removal of voxels.

2) Collider method

Graph B1 in *Fig. 14* displays benchmark results for *cutting* with the *collider method*. The probe's scale was set to *0.03*, which removes *792* voxels per collision. *Graph B2* shows that about *1,900* enabled colliders were used at most. The number of colliders is the limiting factor of this method, which depends on the volume and the *collidable* object scales. An excessive number of colliders would cause the *collidable* script to iterate over too many possible colliders. Updating the colliders' corresponding game objects is a particularly demanding task. With fewer enabled colliders to update, performance improves even when the script must iterate over the entire *AABB*. As shown in the profiler graph, performance increases over time as the cutting object incurs in smaller steps and does not have to create colliders where voxels are already removed. The calculation of *Unity's* contact points modestly improves over time.

3) Parallel cutting

Graph C in *Fig. 14* presents benchmark results for *parallel cutting*. The cubic probe's scale was set to *0.15*, removing *132,651* per cut. Here, the limiting factor is the number of cores available for executing the cube layers mentioned in the *3) Parallel cutting and restoring* subsection. Thread execution time increases along with the number of voxels. Rendering eventually limits the parallel cutting operation.

4) Rendering

Another limiting factor is the number of visible voxels that the application can render. Our parallel cutting benchmark, using a cubic probe of *0.2m* shows that the application can handle *2,476,484* visible voxels. At the end of the operation, rendering is constantly slightly below *90 FPS* as *Unity's* rendering thread must wait for its main thread and vice versa.

TABLE IV. RENDERING LIMITATION OF 2,476,484 VISIBLE VOXELS AFTER VOLUME INITIALIZATION AND AFTER APPLYING A FILTER OPERATION WITH A VALUE RANGE OF 0-42.

	After initialization	**After filter operation (0-42)**
Visible chunks	2,344	4,197
Visible voxels	654,936	1,225,995

Table IV summarizes visible voxel variation after volume initialization and the application of a filter operation of a *0-42* value range. The *512x348x176 voxel* volume used for these benchmarks stores *31,358,976* voxels, which are divided into *8,712* chunks, represented by a *33x22x12* chunk grid.

TABLE V. APPROXIMATE CUTTING PERFORMANCE LIMITS OF CUBIQUITY AND VOXSCULPT UNDER A STABLE OF 90 FPS

Library	***Cubiquity***	***VoxSculpt***	***VoxSculpt***	***VoxSculpt***
Method	Ray cast	Collider	Ray cast	Parallel
Cutting voxel limit	4,139	792	11,459	132,651
Object scale	0.06	0.03	0.087	0.15

The most relevant data from Table V demonstrates that *VoxSculpt* delivers improved volume sculpting performance compared to *Cubiquity*. Using the *Ray Casting method*, *VoxSculpt* can delete about 2.7 times more voxels than *Cubiquity* while still maintaining a stable *90 FPS* experience.

V. DISCUSSION

Benchmark comparison results demonstrate that *VoxSculpt* outperforms *Cubiquity* in measured operations. Sculpting operations are faster because they only access dense storage and write new color values to the 3D array without the creation or removal of *visible* voxels. The same rationale applies to the volume initialization since this method primarily accesses the 3D array, followed by making the volume's surfacing voxels *visible*, which translates to adding voxels to the *visible* voxel list. In general, performance is heavily affected by calculations that ensure that only *visible* voxels are shown.

Cubiquity still outperforms VoxSculpt's rendering in the *parallel cutting* operation *as Graph C* in *Fig. 14* demonstrates. This issue might be attributed to a different rendering schema, as *Cubiquity* only renders visible voxel faces, and it presumably does not render occluded voxels. *VoxSculpt* renders cubes with *12* triangles and renders voxels that are occluded by other voxels, which generates a significant calculation overhead and is compensated by *GPU Instancing*.

VoxSculpt has limitations regarding the maximum size of the tomographic dataset volumes that can be rendered and sculpted. This is especially true for the *collider method*, necessary for precise *cutting*. *Table 5* summarizes the number of voxels that can be cut while still maintaining a stable *90FPS* performance.

VoxSculpt's rendering capacity limitation could potentially be addressed through chunk *occlusion culling* [22] through the *flood fill algorithm* [23]. Another approach to improve rendering would be to implement *greedy meshing* [24] to reduce the number of rendered triangles for each voxel.

Another limitation is the maximum number of supported colliders. A pragmatic approach to increase this limit would be to schedule the collider updating algorithm in parallel using *Unity DOTS* to replace the current *game objects* with cache-coherent entities. *AABB testing* could also be divided into chunks and assigned to *Unity Jobs*, which has the potential issue of being limited to primitive shape colliders.

Ray casting accuracy is another candidate for improvement. This issue is mostly unnoticeable under sufficiently large volume resolution and camera distance, but a closer distance could show that the *ray casting* method misses contact points while nearing corners. Better accuracy could be achieved through an algorithm based on the *Digital Differential Analyzer* as it is a fast and robust choice for a tiled object like a voxel grid [25].

VI. CONCLUSION

In this article, we have presented *VoxSculpt*, a novel open-source voxel library for tomographic volume sculpting in virtual reality. Our library is built on top of a popular game engine and leverages *GPU instancing*, *parallel computing*, and a native *chunk-based data structure* to optimize rendering performance. Our benchmark evaluations indicate that *VoxSculpt* offers a significant improvement over the state-of-the-art and can render and modify millions of visible voxels for stable real-time interaction in virtual reality. We have also implemented a range of new features such as *filtering*, *undoing*, *restoring*, and improved sculpting precision.

By enabling real-time immersive volume sculpting, *VoxSculpt* has the potential to improve the efficiency and intuitiveness of manual processing of tomographic data volumes, such as interactive image segmentation. As we publish *VoxSculpt* as a freely-available resource, we anticipate that this voxel library might assist researchers in creating custom solutions for tomographic imaging practitioners.

ACKNOWLEDGMENT

The author acknowledges the support of the Cluster of Excellence »Matters of Activity. Image Space Material« funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2025 – 390648296.

REFERENCES

[1] J. K. Iglehart, "The new era of medical imaging—progress and pitfalls," New England Journal of Medicine, vol. 354.26, pp. 2822-2828, 2006.

[2] H. Ramadan, C. Lachqar, and H. Tairi, "A survey of recent interactive image segmentation methods," Comp. Visual Media, vol. 6, no. 4, pp. 355–384, Dec. 2020.

[3] Chowdhary, C. L., & Acharjya, D. P., "Segmentation and feature extraction in medical imaging: a systematic review.," Procedia Computer Science, vol. 167, pp. 26-36, 2020.

[4] E. Hutchins, l. Hollan and D. A. Norman, "Direct manipulation interfaces," Human–computer interaction, vol. 1(4), pp. 311-338, 1985.

[5] Meyer, Tom, and Al Globus. "Direct manipulation of isosurfaces and cutting planes in virtual environments." Department of Computer Science, Brown University, 1993.

[6] H. Marin-Vega, G. Alor-Hernández and G. Zatarain-Cabad, "A brief review of game engines for educational and serious games development," Language Learning and Literacy: Breakthroughs in Research and Practice, pp. 447-469, 2020.

[7] S. G. Izard, J. A. J. Mendez, P. R. Palomera and F. J. Garcia-Penalvo, "Applications of virtual and augmented reality in biomedical imaging," Journal of medical systems, vol. 43, no. 4, pp. 1-5, 2019.

[8] B. Faludi, E. I. Zoller, N. Gerig, A. Zam, G. Rauter and P. C. Cattin, "Direct visual and haptic volume rendering of medical data sets for an immersive exploration in virtual reality," in International Conference on Medical Image Computing and Computer-Assisted Intervention, Springer, 2019, pp. 29-37.

[9] F. King, J. Jayender, S. K. Bhagavatula, P. B. Shyn, S. Pieper, T. Kapur, A. Lasso and G. Fichtinger, "An immersive virtual reality environment for diagnostic imaging," Journal of Medical Robotics Research, vol. 1, no. 1, 2016.

[10] G. Wheeler, S. Deng, N. Toussaint, K. Pushparajah, J. A. Schnabel, J. M. Simpson and A. Gomez, "Virtual interaction and visualisation of 3D medical imaging data with VTK and Unity," Healthcare technology letters, vol. 5, no. 5, pp. 148-153, 2018.

[11] S. Rizzi, "Volume-based graphics and haptics rendering algorithms for immersive surgical simulation," University of Illinois at Chicago, 2013.

[12] Reddivari and J. Smith, "VRvisu++: A Tool for Virtual Reality-Based Visualization of MRI Images," in 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC), IEEE, 2020, pp. 1129-1130.

[13] J. Zhang, Y. Lyu, Y. Wang, Y. Nie, X. Yang, J. Zhang and J. Chang, "Development of laparoscopic cholecystectomy simulator based on Unity game engine," in Proceedings of the 15th ACM SIGGRAPH European Conference on Visual Media Production, 2018, pp. 1-9.

[14] P. Kalshetti, P. Rahangdale, D. Jangra, M. Bundele and C. Chattopadhyay, "Antara: An interactive 3D volume rendering and visualization framework," arXiv preprint arXiv:1812.04233, 2018.

[15] S. You, L. Hong, M. Wan, K. Junyaprasert, A. Kaufman, S. Muraki, Y. Zhou, M. Wax and Z. Liang, "Interactive volume rendering for virtual colonoscopy," in Proceedings. Visualization'97 (Cat. No. 97CB36155), IEEE, 1997, pp. 433-436.

[16] M. B. Hoffensetz and C. N. Daugbjerg, "Volume visualization of medical scans in virtual reality," 2018.

[17] D. Escobar-Castillejos, J. Noguez, R. A. Cardenas-Ovando, L. Neri, A. Gonzalez-Nucamendi and V. Robledo-Rella, "Using Game Engines for Visuo-Haptic Learning Simulations," Applied Sciences, vol. 10, no. 13, 2020.

[18] Williams, D. Cubiquity Voxel Engine [Source code]. https://github.com/DavidWilliams81/cubiquity [Accessed 09 12 2021].

[19] L. S. Rodrigues, J. Nyakatura, S. Zachow and J. H. Israel, "An Immersive Virtual Paleontology Application," Haptics: Science, Technology, Applications. EuroHaptics, pp. 478-481, 2022.

[20] S. Laine and T. Karras, "Efficient sparse voxel octrees–analysis, extensions, and implementation," NVIDIA Corporation, 2010.

[21] J. de Vries, "LearnOpenGL - Transformations," [Online]. Available: https://learnopengl.com/Getting-started/Transformations. [Accessed 21 02 2022].

[22] Coorg, S., & Teller, S. Real-time occlusion culling for models with large occluders. In Proceedings of the 1997 symposium on Interactive 3D graphics (pp. 83-ff), 1997.

[23] M. Barthet, A. Liapis and N. Georgios, "Open-ended evolution for Minecraft building generation," IEEE Transactions on Games, 2022.

[24] C. Farhat and M. Lesoinne, "Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics," International Journal for Numerical Methods in Engineering, vol. 36.5, pp. 745-764.

[25] Hu, H., Liu, M., Zhong, J., Deng, X., Cao, Y., & Fang, P. A Case Study of the 3D Water Vapor Tomography Model Based on a Fast Voxel Traversal Algorithm for Ray Tracing. Remote Sensing, 13(12), 2422, 2021.

[26] Duncan, D., Newman, B., Saslow, A., Wanserski, E., Ard, T., Essex, R., & Toga, A. VRAIN: Virtual reality assisted intervention for neuroimaging. 2017 IEEE Virtual Reality (VR), 467–468, 2017.

[27] Chheang, V., Saalfeld, P., Huber, T., Huettl, F., Kneist, W., Preim, B., & Hansen, C. Collaborative Virtual Reality for Laparoscopic Liver Surgery Training. 2019 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR), 1–17, 2019.