The Visual Computer
https://doi.org/10.1007/s00371-023-02976-x

ORIGINAL ARTICLE

LVDIF: a framework for real-time interaction with large volume data

Jialin Wang[1,2] · Nan Xiang[1] · Navjot Kukreja[2] · Lingyun Yu[1] · Hai-Ning Liang[1]

Accepted: 9 June 2023

Abstract
The interest in real-time volume graphics has grown rapidly in the last few years, driven by the increasing demands from both academia and industry. GPU-based volume rendering has been used in a wide variety of fields, including scientific visualization, visual effects, and video games. Similarly, real-time volume editing has been used to build terrain and create visual effects during game development; it has even become an integral part of gameplay in various video games (e.g., Minecraft). Nowadays, as the size of volume data increases, processing large volume data in real time is inevitable in many modern application scenarios. However, manipulation and editing of large volume data are associated with various challenges, such as dramatically increasing memory usage and computational burden. In this paper, we present a framework for interactive manipulation and editing of large volume data to address these challenges. A robust and efficient method for large signed distance function (SDF) volume generation is presented and incorporated into the framework. Also, a complete implementation with specialized GPU optimization is introduced to demonstrate its usefulness and effectiveness—it is included in the framework as well. The framework can be an easy-to-use middleware or a plugin that is able to integrate into game engines for the development of various types of applications (e.g., video games). It can also contribute to the research looking at large volume data from a user-centered perspective (e.g., for human–computer interaction researchers).

Keywords Volume graphics · Large volume data · Volume editing · Volume manipulation · Framework

1 Introduction

Manipulation and editing of models derived from volume data in real-time have been an important topic over the past few decades due to their widespread applications in multimedia. Various technologies and methods have been developed. Marching Cubes is an algorithm for creating polygonal representations from volumetric data and is widely applied in the surface construction of 3D medical data [23]. More recently, this algorithm has also been used in procedural terrain generation and gameplay within video games. As an important volumetric representation, signed distance volume can likewise be used to create high-quality animation with volume rendering methods such as ray marching [15]. In general, rendering is a process of generating a 2D image from a 2D or 3D model but does not deal with interactive operations. To support interactions, other processes are also needed (e.g., collision detection). For example, despite raycasting-based volume rendering being able to achieve good performance on large volume data, raycasting cannot generate general collision meshes that benefit a wide range of interaction scenarios (e.g., game physics that need to prevent objects from colliding with each other). Also, most interaction scenarios involving volumetric 3D models and mesh generation use small volume data (typically, less than 1024^3) due to several drawbacks caused by large volume data, especially the dramatically increased memory usage and computational burden [5–7,9,11,16,28,35]. In this era of information explosion, the size of raw data has become very large, which has sparked the desire for the interactive manipulation and editing with larger volume data (interactive volume for short). Meanwhile, the large computational cost and memory load caused by large volume data need an effective approach to cope with, thereby ensuring efficient interaction. A framework for processing large volume data in real-time is expected to allow creating and editing large interactive volume models with ease, such as subdividing volumetric 3D models with richer details and higher precision. It will be tremendously beneficial to a

✉ Hai-Ning Liang
haining.Liang@xjtlu.edu.cn

1 Department of Computing, School of Advanced Technology, Xi'an Jiaotong-Liverpool University, Suzhou, China

2 Department of Computer Science, University of Liverpool, Liverpool, UK

number of applications in the fields of game development, medical simulation, and human–computer interaction (HCI) research.

Interactive volume models have become an important element of gameplay in various video games. One example is Minecraft, a famous volume-based game released in 2011. Its creative gameplay is built upon an editable voxel world and has attracted millions of players from all over the world. Besides, the feature of interactive volume gives Minecraft enormous potential in both areas of education and research. Another typical case is Tilt Brush, a line rendering-based 3D painting application for virtual reality (VR). It allows artists to draw 3D paintings in an immersive VR world [4]. However, a line rendering-based canvas limits users' artistic creation (e.g., the lack of color blending), which can be enhanced by introducing a volume-based rendering pipeline into the application. Although there are some professional 3D modeling applications with 3D sculpting support such as Zbrush, 3D Coat, and Adobe Medium, editing with volumetric 3D models requires more than 3D sculpting capabilities. Efficient operations with volume data require a series of optimizations on volume rendering and data structure (e.g., voxel quantization and parallel processing), which can be simplified by using a set of optimization tools usually presenting as a framework (a paradigm for creating solutions) or a software library (a suite of data and code for developing applications). For instance, volumetric applications used for film production can use OpenVDB (an open-source software library for working with sparse volumetric data) to simplify the implementation of volume rendering-based visual effects [25]. However, to our knowledge, there is no such framework available for facilitating the development of applications that need to implement large interactive volume models. Moreover, to make these editing features easily accessible, it is helpful to integrate them into an existing game development platform like Unity.[1]

Mesh generation from volume data (e.g., marching cubes used for volume rendering via geometric primitives) often cannot achieve satisfactory performance on large volume data objects or models. There are several obstacles when implementing mesh generation on large volume data sets. For instance, increasing generated mesh number and volume data size can cause more memory usage and require larger computational power which is difficult to satisfy in many applications. Even array indexing of a large volume set would lead to performance issues in some cases. As an example, the .NET framework prior to v.4.5 (which is still being used in Unity as the default version at the time of the writing of this paper) does not support an array larger than 2GB, whereas a 1024^3 32-bit floating-point volume requires 4GB. A 1024^3 color volume with 32-bit RGBA color would also require 4GB. The size of large RGBA volume can be compacted using data structure based on quantization. As described later in this paper, this is one of the optimizations in our framework. On the other hand, high frame rate and resolution are important to ensure a good user experience and performance in virtual environments [36,37]. Our framework aims to provide a series of optimization methods that allow developers to access efficient operations with large interactive volume data in virtual environments with high frame rates and resolutions.

In this paper, we introduce a framework that focuses on allowing the manipulation and editing of large volume data models. In short, the main contributions of this work are threefold:

1. A robust and efficient approach for the sign calculation of large SDF volume generation from triangle mesh (Sect. 3.2);
2. A GPU-based strategy for optimization of interactive 1024^3 RGBA SDF volume (Sect. 3.3);
3. A lightweight framework that integrates the first two contributions to support cross-device input for editing of large volume models (Sect. 3.4).

Our framework, while lightweight, contains all the foundational blocks needed for large volume data interaction, including volume generation, rendering, editing, and storage. It can benefit both artists looking to create artistic expressions with high-level of detail and HCI and visualization researchers interested in exploring interaction techniques for large volume data sets.

2 Related work

2.1 Volume data and volume rendering

Volume data are commonly comprised of volumetric pixels (voxels). A voxel represents a specific grid value in a regularly spaced, three-dimensional grid. The data type of a voxel can be a binary, an integer, or a floating-point number. A binary voxel uses 1 and 0 to represent the presence of a voxel [27]. It has the smallest size in memory due to its binary format. However, binary voxels contain the least amount of details. An integer voxel is widely used in Minecraft-like environments, though it contains limited states or types for the voxel. One of the typical floating-point voxels is a signed distance field (SDF), which can be used to represent shapes by signed floating-point voxels in a volume grid. Floating-point voxels enable storing more details with higher memory usage. Regarding the data structure of the voxel, vector and scalar voxels are the two common structures used for volume data. A scalar voxel contains only one scalar such as

[1] https://unity.com/.

computed tomography (CT) and magnetic resonance imaging (MRI) volume. In contrast, a vector voxel contains more than one scalar value such as RGBA color.

Volume rendering is an essential technique in drawing volume data. In some prior work, volume rendering with raycasting typically has acceptable real-time performance since it only calculates the values along the ray without concerning reflection or refraction. Ray marching is a typical raycasting approach that can use SDF in rendering volume data. It has been used in making visual effects and special rendering styles in video games (e.g., Claybook); it is even capable of creating real-time animations (e.g., I. Quilez's animation [15]). However, implementing the volume rendering algorithm and various SDF primitives with high performance requires strong programming skills. It can be especially time consuming and challenging for game designers and digital artists who are not very experienced programmers. A lightweight interaction framework may be handier for artists and novices to manipulate and edit large volume data because it lowers the need for programming experience.

Although the performance of volume rendering can be improved by raycasting techniques (e.g., ray marching), as the most popular representation of model data, the mesh data is still necessary for the rest of the processes, such as procedural terrain and collision detection, which are essential to a 3D interactive environment. Marching cubes and dual contouring are two 3D surface construction algorithms that have been widely used in terrain editors for both game development and real-time gameplay in video games (e.g., No Man's Sky). As stated, our proposed framework focuses on direct manipulation and editing of volume data and real-time mesh generation to support more interaction and model creation scenarios.

Binary volume can only produce Minecraft-like rough (non-smooth) surfaces using marching cubes. The issue can be solved by dual methods. For instance, surface nets is a typical dual method that generates smooth surfaces from binary volume data when running multiple iterations [8]. Although binary volume data tend to be less memory intensive, it does not contain enough information for more complex interactive volume. Moreover, technological progress is usually chasing growth in demand. The optimization of 3D surface construction with Marching cubes on larger SDF volume deserves further exploration because it is able to recover a much more detailed structure of surfaces from a large-scale interactive volume in a high-efficient way.

Octree and sparse voxel octree (SVO) are typical two structures that can benefit volume rendering and mesh generation from volume data [2,21], as well as the chunk system, a commonly used optimization method for a large interactive mesh [24]. In addition, empty space skipping (ESS) is considered an essential optimization method for efficient volume rendering and mesh generation from volume data [31]. It optimizes the rendering process by ignoring the invisible components of the volume. For manipulation and editing of volume data models and to improve performance, ESS can also be employed by skipping the inactive components of the volume. Although both SVO and chunk system can achieve ESS optimization, the chunk system is more suitable than SVO for large interactive volume, especially in processing mesh data such as the mesh collider. The mesh collider has better performance in smaller equal-size chunks rather than SVO chunks which may produce a large mesh collider with poor collision detection performance. A mesh collider is helpful for interactions on the mesh generated from the volume.

2.2 Mesh generation from volume data

Marching cubes is one of the most popular 3D surface construction algorithms because of its simple design that allows easy implementation in different programming languages [26]. One of its disadvantages is that the reconstructed shapes or models lack sharp features. The other two algorithms, dual marching cubes and dual contouring, can recover sharp features [18,29]. Although sharp feature reconstruction is the main advantage of dual marching cubes and dual contouring, as pointed out earlier, marching cubes can lead to better performance due to its simple design and implementation. Moreover, increasing the size of the volume data can also allow enhanced sharp features because further details are then available (e.g., more complex shapes). Therefore, it is worth designing a framework for interactive operations with large volume data because of the additional benefits it brings to various applications.

2.3 Optimization of mesh to SDF

Mesh to SDF is a composite process, which consists of different stages with individually optimized algorithms including optimization of distance field calculation and optimization of sign calculation [17]. The calculation of SDF can be optimized by using a ray map, a ray-based data structure that preserves geometric meaning while reducing the amount of work to be done for ray tests [20]. Liu et al. [22] propose a multi-BVH structure and octree-based optimization of SDF calculation. We use a simple octree-based overlapping chunk system to keep the framework simple (see Sect. 3.1). However, SDF calculation can produce wrong signs for mesh with discontinuous normal vectors at the edges and vertices according to Andreas Bærentzen [1]'s work. Our framework concentrates more on the robustness of the sign of SDF since wrong signs can affect the rendering effect. Therefore, we aim to solve this problem by designing a series of filters.

In general, the calculation of signs for SDF can be regarded as a problem of point in polygon (PIP) [12]. A recent algorithm to solve the PIP problem for triangle mesh is generalized winding numbers (GWN) [14]. Although it produces robust results for SDF volume generation, its calculation process is extremely time-consuming and less efficient. GWN cannot be optimized by parallel computing (e.g., an overlapping chunk system in Sect. 3.1) since GWN requires iterating the entire mesh for each voxel. In other words, GWN is inefficient for large SDF generation which needs to iterate the entire mesh for a large number of voxels. Therefore, we design a series of sign voxel filters (SVFs) to correct wrong signs to generate high-quality SDF in a much more efficient way (see Sect. 3.2).

2.4 Interactive rendering, manipulation, and editing of large volume data

Interactive rendering, manipulation, and editing of volume data can be used for various applications including surgical simulation, interactive medical image segmentation, and terrain editing [3,10,34,38]. The issue we are focusing on is a common problem, which is performance optimization with large volume data. Data compression is one of the main optimization methods in interactive volume rendering since physical memory may not be sufficient to meet the demands of large volume data operations in many cases [30]. For example, a 1024^3 SDF volume (with the data type of 32-bit floating-point) needs 4 GB memory which was difficult to have in the early computers. However, nowadays, a standard high-end computer can have 24 GB RAM but could still face challenges dealing with large volume data in real time. Moreover, marching cubes, one of the most popular 3D surface construction algorithms, is more practical for volume objects with small data size in real-time simulation [6,11,28] (see Table 1). Therefore, our framework is designed for efficient operations with large interactive volume data. In our framework, instead of focusing on improving compression algorithms like some prior work, we are proposing a novel optimization strategy to address challenges for large interactive volume, which is lightweight, easy to implement, and can be integrated into common development platforms such as Unity. It can be useful for HCI researchers and interaction designers in creating and testing new interaction techniques. In addition, artists can also benefit because a volume version of Google's Tilt Brush in VR could be possible. It can also be used to generate procedural and editable volume worlds to create new types of video games. To verify the validity and applicability of the framework, we integrated it into the popular game engine Unity. Moreover, our modular design with low coupling of the framework makes it easier to be used in other game engines, third-party systems, and applications. The source code is available for download (https://github.com/Chaosikaros/LVDIF) so that other researchers can use it or conduct further development with it.

Table 1 Summary of papers that use one of the most popular 3D surface construction algorithms, marching cubes for interactive volume manipulation (RTR: real-time rendering; RTS: real-time simulation; the latter are highlighted for readability)

Year	Max volume size	Scenario
2005 [9]	$256 \times 256 \times 225$	RTR
2006 [16]	$256 \times 256 \times 256$	RTR
2008 [7]	$255 \times 255 \times 255$	RTR
2009 [11]	$128 \times 128 \times 128$	RTS
2010 [28]	$256 \times 256 \times 256$	RTS
2010 [35]	$277 \times 244 \times 222$	RTR
2013 [5]	$512 \times 512 \times 512$	RTR
2015 [6]	$97 \times 4 \times 55$	RTS

3 LVDIF: large volume interaction framework

3.1 Mesh to SDF via octree-based optimization

Algorithm 1 states a straightforward method to calculate an SDF volume from a triangle mesh. It assumes that the mesh has normal vectors that are pointing outward (some irregular or broken mesh may have normal vectors that are pointing inside). This algorithm can handle non-watertight models by using normal vectors to calculate the sign of distance values (see Fig. 1c–d). The time complexity of Algorithm 1 is $\mathcal{O}(n^2)$ for a large SDF volume or a large mesh. As such, the program would encounter performance issues when generating a very large SDF volume with a large mesh. To reduce the time complexity, we use an octree of overlapping chunks (calculated from the bounding box of the input mesh), which contains mesh chunks of the entire mesh in the nodes of the octree (see Fig. 1e). Each overlapping chunk contains all the triangles that overlap with the corresponding nodes of the bounding box, which is located in an octree structure consisting of overlapping bounding boxes. Each overlapping bounding box is a scaled original bounding box node in the octree. The default scale factor is set to 2 in our following experiments. In other words, the overlapping bounding box is an octree of the scaled original bounding box. We use overlapping chunks to handle the triangles that overlap between adjacent chunks. The basic idea is to find the closest non-empty chunk (a chunk that contains triangles) in the octree and then apply Algorithm 1 to this chunk. It is an application of the ESS optimization method that can reduce the time complexity for a large mesh. The entire volume is also split into smaller volume chunks to avoid performance issues when

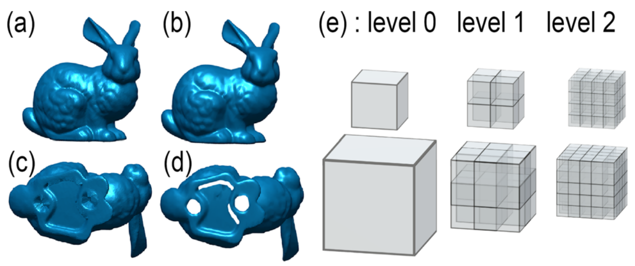


Fig. 1 Comparison of an original Stanford bunny mesh and a mesh of 1024^3 SDF volume generated by marching cubes. **a**, **b** front views of SDF mesh and Stanford bunny mesh; **c**, **d** bottom views of SDF mesh and Stanford bunny mesh. **e** Normal (first row) and overlapping (second row) octree data structures. Each cube has up to eight children and one parent box

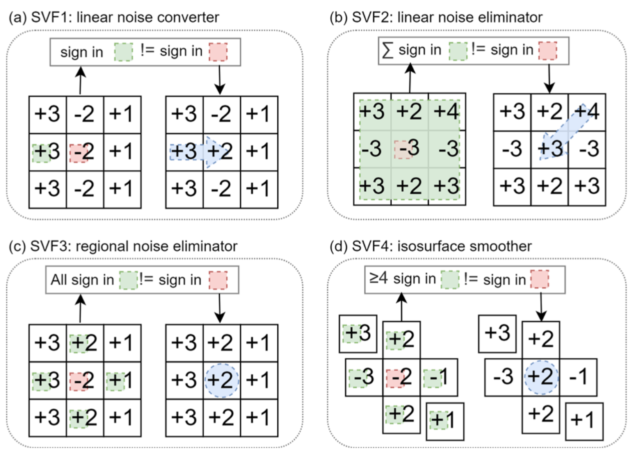

(a) SVF1: linear noise converter
sign in != sign in
(b) SVF2: linear noise eliminator
∑ sign in != sign in
(c) SVF3: regional noise eliminator
All sign in != sign in
(d) SVF4: isosurface smoother
≥4 sign in != sign in

Fig. 2 Example of 4 sign voxel filters for correcting wrong sign. **a**–**d** are SVF1 to SVF4 with their roles during the SVF processing. The red and green squares mean the objective sign and input sign of the filters. The blue arrows indicate a sign overwriting operation from the start voxel to the end voxel. The blue circle shows an inverse operation of the objective sign

dealing with a large volume. The use of overlapping chunks can guarantee the proper functioning of SDF generation for large volume with a large mesh. For example, the CUDA program of the optimized algorithm can generate 1024^3 SDF volume from the Stanford bunny mesh (30,338 triangles) by using 16^3 overlapping chunks. The entire running time is 17 min on a single NVIDIA RTX3090 GPU.

ALGORITHM 1: Triangle mesh to SDF volume algorithm

```
for $Voxel$ in $VolumeGrid$ do
    $Point_A$ = TransfromToBboxSpace($Voxel$,$MeshBbox$);
    $Distance_{Min}$ = Max floating-point number;
    for $Triangle$ in $Mesh$ do
        $Point_B$ = ClosestPointOnTriangle($Triangle$);
        $Vector_{Temp}$ = $Point_B$ - $Point_A$;
        $Distance_V$ = Length($Vector_{Temp}$);
        if ($Distance_V < Distance_{Min}$) then
            $Distance_{Min} = Distance_V$;
            $Vector_X = Vector_{Temp}$;
            $Index_{Tri}$ = IndexOfTriangle($Triangle$);
        end
    end
    $Normal_X = GetNormalByTriIndex(Index_{Tri})$;
    if DotProduct($Normal_X$, $Vector_X$) > 0 then
        $Distance_{Min}$ = -$Distance_{Min}$;
    end
end
```

3.2 Sign voxel filters for robust mesh to SDF

3.2.1 Design of sign voxel filter 1

Algorithm 1 can produce wrong signs for the mesh (e.g., Stanford dragon) with discontinuous normal vectors (e.g., reverse normal vectors, see Figs. 3a and 4a. The meshes are converted by using marching cubes method.) at the edges and vertices [1]. The wrong sign produced by Algorithm 1 for discontinuous normal vectors is a regular noise with common features. Our SVFs consist of 4 different filters (from SVF1 to SVF4) that target different noise features during the whole SVF process. We use the Stanford Dragon (201,031 triangles), Stanford Lucy (99,970 triangles), CyberWare Horse (96,966 triangles), and Stanford Armadillo (345,944 triangles) as examples to explain and demonstrate SVFs in Figs. 3, 4, and 5 (the time is measured by the code running on a single NVIDIA RTX3090 GPU). The designs of SVFs need to satisfy the common features of wrong signs in both inside and outside of the SDF. Otherwise, they may generate a hollow volume rather than a solid one. Most initial wrong signs generated by Algorithm 1 (see Fig. 3a) have a common feature: most wrong signs are far away from isosurface0 (isosurface with the value 0). Therefore, we isolate sign distances with an absolute value bigger than a threshold (that is bigger than isosurface0, which is 2 in our case, after remapping the distance values to the grid space) before applying SVFs (except SVF4). A small threshold will affect some correct signs, while a big one will likely miss some wrong signs. This can also protect the near zero isosurface with fewer wrong signs from being over-corrected by SVFs. SVF1 runs on a 2D plane along one axis ($y-z$ plane along the x axis in default). The algorithm of SVF1 is to overwrite the objective sign whose previous sign in the previous plane along the same axis is different from the objective one. Figure 2a shows the process of SVF1. -2 was overwritten by $+2$ because the previous sign is positive in $+3$ (the previous sign that is different from the current one). SVF1 is simple but important since it can convert all wrong signs to more regular linear noise (see Fig. 3b).

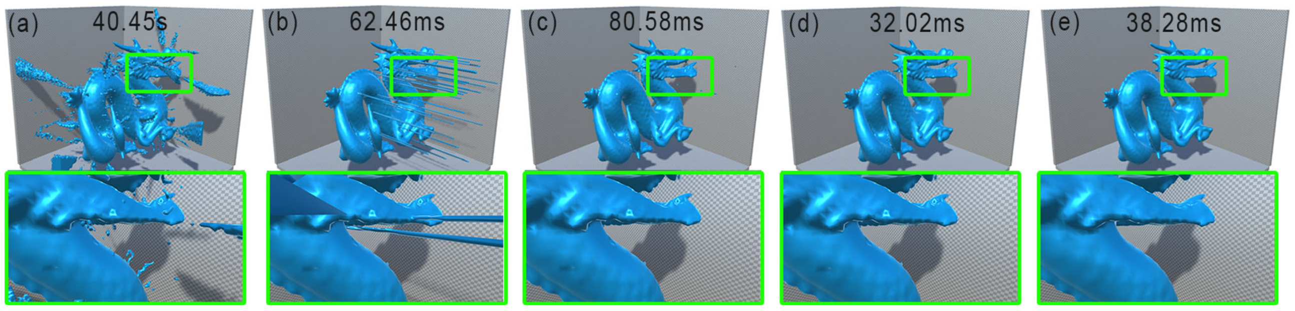


Fig. 3 Comparison and ablation study of SVFs. The meshes are converted from 128^3 SDF volumes generated by Algorithm 1 and different SVFs. The edge size of the grid is equal to the edge size of the volume. **a** Algorithm 1 without SVFs; **b**–**e** SVF1, SVF2, SVF3, and SVF4. The time cost shown in **a** indicates the running time (in s) of Algorithm 1. The time in **b**–**e** indicates the running time (in milliseconds) of each SVF on the SDF volume from **a**

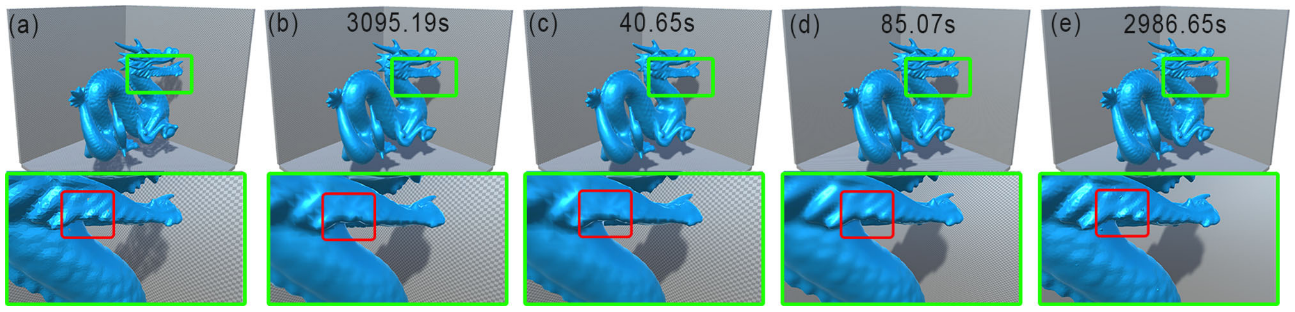


Fig. 4 Comparison of GWN algorithm and our proposed method on generating different sizes of SDF volumes. **a** The original Stanford dragon mesh; **b** is 128^3 SDF volumes generated by the GWN algorithm. **c**–**e** are 128^3, 256^3, and 1024^3 SDF volume generated by Algorithm 1 with all 4 SVFs; The time in **b**–**e** indicates the running time (in s) of each algorithm. The red rectangle highlights a region that contains 3 triangles with a reverse normal vector

3.2.2 Design of sign voxel filter 2

The linear noise is caused by wrong signs close to the threshold, which can be further corrected by SVF2. The first step of SVF2 is to compare the sign of the summation in 3×3 matrix (detection matrix), including the objective sign with itself (all 9 voxels). If the sign of the sum of signs is different from the objective sign, SVF2 will overwrite the objective sign when it finds an adjacent voxel whose absolute value is greater than that of the objective voxel. As shown in Fig. 2b, the sum of signs in the 3×3 matrix is $+3$ with a sign that is different with objective voxel -3. And $+4$ is one voxel in the matrix whose absolute value $|+4|$ is bigger than $|-3|$. Therefore, -3 was overwritten by $+4$. SVF2 tends to find a wrong sign by applying a local area checking process. If SVF2 detects a wrong sign, it will find a flow of the correct sign to fix the wrong one, since sign distances are directional and the border between wrong and correct signs has a feature of inverse sign flow. SVF2 can work alone without other SVFs. Although SVF2 is efficient for regular wrong sign features generated by SVF1, in the pilot study, we found it inefficient for large SDF since it requires: (1) scanning at all 3 axes and (2) increasing sizes of detection matrix and sign flow matrix along with the increased SDF volume size.

3.2.3 Design of sign voxel filter 3

SVF2 cannot handle the linear noise that is close to each other when using 3×3 detection and sign flow matrix. It will leave some small isolated areas of (regional) wrong signs (see Fig. 3c) that can be corrected by SVF3. Although a larger size of detection and sign flow matrix can also solve this problem, it can cause efficiency issues for the whole SVFs process as pointed out earlier. Therefore, we choose to use SVF3 instead of a larger matrix size. SVF3 is introduced to find wrong signs by comparing the objective sign with the signs of 4 neighbor voxels in a same plane. If the 4 neighbor voxels have the same signs, which are different from the objective voxel's sign (the feature of small isolated areas with wrong signs), SVF3 will do an inverse operation for the objective sign (see Fig. 2c).

3.2.4 Design of sign voxel filter 4

SVF1, SVF2, and SVF3 need to be looped for each plane until there are no objective wrong signs detected. SVF4 is the only filter that needs to be executed once. SVF4 can generate a smooth isosurface0 (see Fig. 3e). It targets all remaining wrong signs (for instance, wrong signs missed by other SVFs and wrong signs close to isosurface0) without the need for a threshold (see Fig. 3d). It calculates the number of all 6 neighbor voxels (4 from the same plane, and 2 from the prior and next plane) that have different signs with objective sign in the center. If the number is bigger or equal to 4, SVF4 will do an inverse operation on the objective sign (see Fig. 2d).

3.2.5 Results of sign voxel filters

Finally, our proposed SVFs can produce a smooth and high-quality SDF volume, which is more efficient compared with GWN (GWN shown in Fig. 4b and ours shown in Fig. 4c). It works well for larger SDF volumes (see 256^3 and 1024^3 SDF volume shown in Fig. 4d, e with the SDF generation time of 84.23 and 2929.81 s and the SVFs processing time of 0.84 and 56.84 s). Although GWN-based SDF generation is also robust, it uses almost the same time (3095.19 s) but only generates a 128^3 SDF volume. Theoretically, it may cost about 63 h to generate a 1024^3 SDF volume, whereas our method is about 76 times faster than GWN in this case. Moreover, the SVFs combined with Algorithm 1 can also generate robust SDF volume for various meshes (see Fig. 5). Therefore, our SVFs-based method is efficient and robust for large-scale SDF generation.

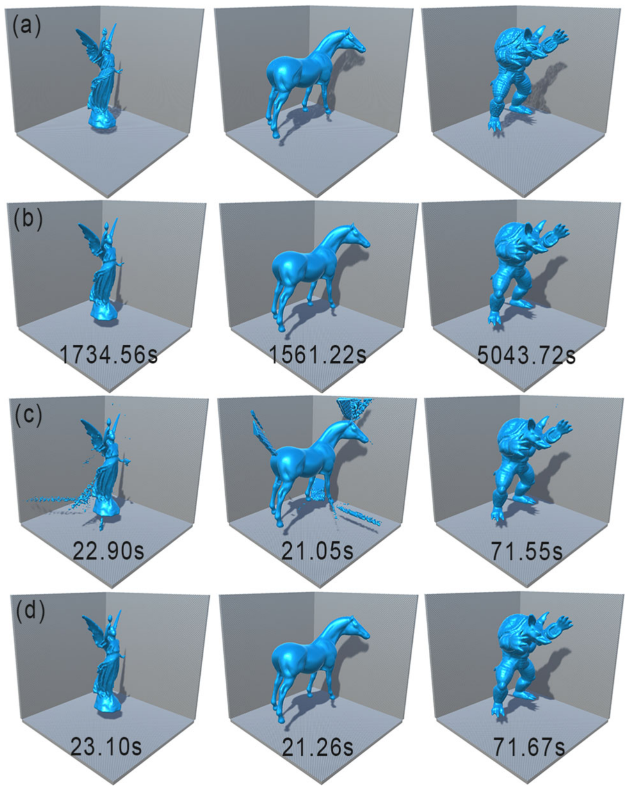


Fig. 5 Comparison of mesh (generated by marching cubes) and time cost of 128^3 SDF volumes generated by different algorithms. From left to right: Stanford Lucy (99,970 triangles), CyberWare Horse (96,966 triangles), and Stanford Armadillo (345,944 triangles). **a** Original mesh; **b–d** are SDF volume generated by the GWN algorithm, Algorithm 1 without SVFs, and Algorithm 1 with SVFs. The time indicates the running time (in s)

3.3 GPU-based mesh generation for large volume data

3.3.1 Design of the rendering pipeline

To take advantage of multiple GPUs, we use CUDA to implement GPU-based mesh generation for large interactive volumes. The marching cubes procedure in the framework can run in other threads on other GPUs to release the GPU memory usage and computational work from the main thread. The main thread and main GPU can focus on mesh rendering, simulation, and interactions. The framework can be easily implemented using OpenCL or Compute Shader. The rendering module contains the steps of the GPU-based mesh generation with mesh collider and vertex colors to support real-time interactive operations. The marching cubes kernel uses the same addressing method as the marching cubes example project from official CUDA examples. Updating the entire mesh from a large SDF volume can be too slow for meaningful smooth interactions. There are three bottlenecks: (1) the generation of the large mesh on the marching cubes kernel; (2) data transfer between a large GPU buffer and CPU buffer; and (3) the generation of a large mesh collider. Therefore, it is necessary to use mesh chunks and volume chunks as two ESS optimizations to achieve large volume. But it is not the case that using more mesh chunks is better since each mesh chunk needs one draw call. The number of draw calls increases with the number of chunks, which can then lower the rendering performance. When the size of chunks increases (which results in the increased number of voxels in one chunk), there is a negative impact on the efficiency of the mesh generation caused by inactive parts of the volume data, since mesh generation from volume data does not typically require frequent updates of the entire mesh (e.g., a subtraction operation of a 3^3 sphere primitive on a 256^3 chunk needs to update the entire 256^3 chunk). Therefore, there is a trade-off between the size of mesh chunks and rendering performance. In our experiments, the rendering pipeline has the best performance (without updating the mesh chunks: 97.71 FPS or frames per second) with a chunk size 64^3 on a (1024^3) SDF volume. The marching cubes kernel can run in 20.20 FPS with a chunk size 64^3 on a (1024^3) SDF volume

Table 2 Rendering performance and marching cubes kernel performance on the SDF volume (1024^3 and 512^3, group by the horizontal line) of the Stanford bunny under different conditions

Volume size	Chunk size	C1 FPS	C2 FPS	C3 FPS	C4 FPS
1024^3	256^3	125.56	1.26	84.77	96.76
1024^3	128^3	123.32	13.89	83.16	110.63
1024^3	64^3	115.50	20.20	97.71	114.70
1024^3	32^3	44.30	4.79	42.96	43.22
512^3	256^3	393.64	1.56	245.43	267.81
512^3	128^3	393.54	16.52	229.50	333.86
512^3	64^3	369.91	52.08	260.45	343.34
512^3	32^3	227.56	32.67	205.39	212.19

C1 Without updating, *C2* updating one chunk, *C3* individual thread, *C4* individual GPU, *FPS* frames per second (measured in unity). 1024^3 and 512^3 SDF volume of Stanford bunny contain 7,339,856 and 1,822,144 triangles

using one NVIDIA RTX3090 in a 3840 × 2160 resolution (see Table 2). In our framework, the marching cubes kernel is in an individual thread (see the FPS of individual thread (C3) from Table 2) to avoid FPS drop (see the FPS of updating one chunk (C2) from Table 2) in main thread caused by blocked marching cubes kernel. It is an important optimization to ensure a stable FPS in the main thread to allow smooth input sampling in the input module. If the marching cubes kernel is run in an individual GPU, the rendering FPS will also be increased due to the individual GPU releases the GPU memory usage and computational work from the main GPU (see the FPS of individual GPU (C4) from Table 2). Such a multiple GPU architecture is also one of the advantages of our framework.

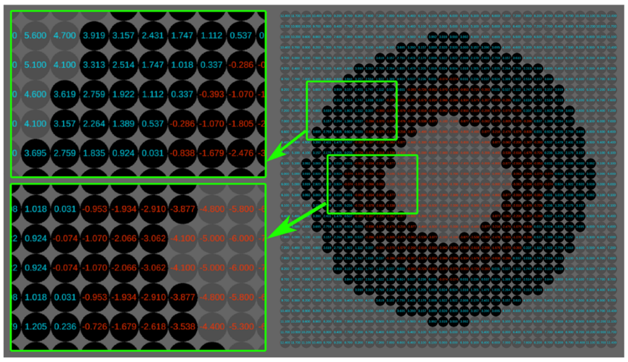


Fig. 6 A slice from a 32^3 SDF volume of a sphere. The red text means negative SDF values. The blue text means positive SDF values. The black cell means Group A: floating-point data with 3 decimal places in $\{-4.001, \ldots, 4.001\}$. The gray cell means Group B: the rest of 65,535 floating-point data in $\{-2880.6, \ldots, -4.1\} \cup \{4.1, \ldots, 2880.6\}$

3.3.2 Design of the voxel data structure

To ensure good performance of random read/write operations, the entire volume data are loaded into GPU memory instead of using a compressed I/O stream. The generated volume of this framework contains both an SDF value (distance value) and an RGAB color value in each vector voxel. We use a 2D unsigned short integer (ushort2) dictionary to reduce memory usage. The vector voxel is in a ushort2 format (2D vector in X and Y). X represents the key of the SDF value in the SDF dictionary, while Y represents the key of the RGBA color value in the color dictionary. The use of volume data and 3D surface construction for interaction can use only an isovalue of 0 for the isosurface. In this case, most voxels with an absolute value of SDF greater than 0 have little contribution to a collision-based interactive volume. Therefore, we use an SDF dictionary (a floating-point dictionary) to decide the decimal places for each possible SDF value. The SDF value in the SDF volume can be split into two groups: *close* (Group A) and *far* (Group B) from the isovalue (with 0 as default). The floating-point number in Group A has 3 decimal places which are enough for calculating smooth normal vectors. Two or fewer decimal places will likely produce incorrect normal vectors. Four or more decimal places will cause a narrower range of the SDF dictionary, which then cannot be used for large volume ($\geq 1024^3$) data). The range of −4.001 to 4.001 is adjustable. Group B contains the rest positive and negative floating-point numbers with 1 decimal place in the rest of the 65,535 pairs ($\{-2880.6, \ldots, -4.1\} \cup \{4.1, \ldots, 2880.6\}$) in the SDF dictionary (see Fig. 6).

Although one ushort can only represent 65,535 integers, it is enough for volume with an edge $(E) \times \sqrt{3}$ (space diagonal, the biggest possible value in a 3D SDF volume) $<$ 2880.6. For instance, the biggest volume is 1663^3 for a Group A of $\{-4.001, \ldots, 4.001\}$ ($2880.6 \div \sqrt{3} \approx 1663.12$). The original SDF value needs to be remapped from the original bounding box space to the SDF volume space (that is $\{-E\sqrt{3}, \ldots, E\sqrt{3}\}$). The color dictionary can store 65,535 indexes of different colors. Such voxel structure can change the storage of SDF value and RGBA color to 2 ushorts. The storage size can be reduced to 4 bytes (ushort2) instead of 8 bytes (32-bit floating-point data + 32-bit RGBA color). A

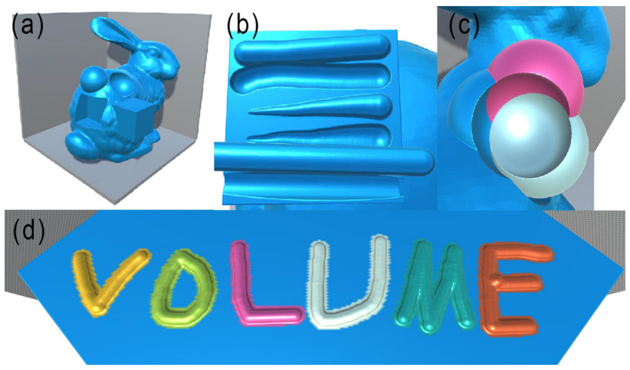


Fig. 7 Interactive editing examples using a mouse-based brush on a 1024^3 RGBA SDF volume of the Stanford Bunny. **a** Basic union and subtraction operations of the 200^3 cube and sphere primitives; **b** brush input with a radius of 15 for painting and erasing using a mouse: (from top to bottom): basic brush, dynamic brush, and drill brush; **c** subtraction on a union of three 200^3 sphere primitives with different colors; and **d** a painting brush example in a radius of 3 on a 200^3 cube primitive

1024^3 RGBA SDF volume in SDF and color dictionary format needs 4 GB of GPU memory. If the isovalue is fixed and color is not needed, it is possible to use one 8-bit unsigned char (uchar) and a smaller uchar SDF dictionary with a smaller Group A of SDF volume for the voxel to save more space since only Group A of SDF volume is necessary in this case (similar to SVO volume).

3.4 Input module

3.4.1 Design of the volume input system and color rendering

We use the sphere and cube SDF volumes as the two volume brush primitives. The function for the SDF primitives and basic union and subtraction operations are adapted from Quilez's work.[2] The brushing method is modified from a common method for 2D images. The raw input of the brush is a set of 3D sampling points from the input device. Smoothing is done via interpolation on the points with a fixed interval.

Figure 7a shows the results of the basic union and subtraction operations with 200^3 cube and sphere primitives on a 1024^3 RGBA SDF volume of the Stanford Bunny. We also designed two more brushes: (1) a dynamic brush with different brush point sizes, and (2) a drill brush for gesture-based input such as using hand motions or with VR controllers (see Fig. 7b). We apply tetrahedral interpolation in the color rendering [19]. We take 8 voxel colors and volume space coordinates from 8 cube vertices in a cube space in marching cubes as the voxel color space. The color is calculated inside the fragment shader by applying tetrahedral interpolation to

[2] https://iquilezles.org/www/articles/distfunctions/distfunctions.htm.

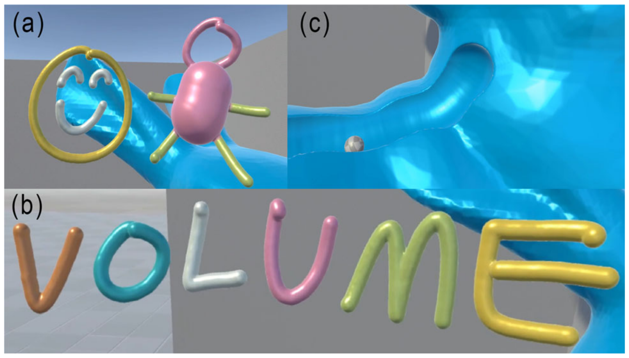


Fig. 8 Interactive manipulation and editing of a 512^3 RGBA SDF volume of Stanford bunny in a VR 3D scenario. **a** A drawing example with different brush radii (Chunk size: 64^3); **b** a writing example with a brush radius of 3 (Chunk size: 64^3); **c** interaction with a sphere that has a rigid body (Chunk size: 16^3)

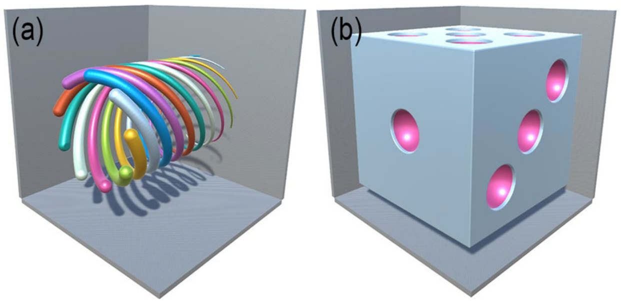


Fig. 9 2 256^3 volume models generated by procedural modeling. **a** colorful 3D spiral lines; **b** a dice

the pixel volume space coordinate, which can be calculated by 3D triangle barycentric coordinate from the 3 uninterpolated triangle vertices and interpolated barycentric coordinate from the vertex shader. The updating of mesh chunks is optimized by using a bounding box octree of the input brush set. The rendering pipeline only updates the mesh chunks that are overlapping with bounding box nodes in the octree.

3.4.2 Applications based on input module

VR controllers can be used in 3D painting scenarios and applications such as Google's Tilt Brush [33]. However, Tilt Brush uses a line renderer that contains no actual 3D voxels, it can be combined with volume graphics to support more abundant interaction scenarios like constructive solid geometry (CSG) used in solid modeling. Our framework can be used to develop 3D painting applications with the generated mesh with colors in each voxel and triangle (see Fig. 8 (a) to (c)). We have used a 512^3 RGBA SDF volume to introduce collision detection in VR for two reasons. First, a 512^3 RGBA SDF volume can run above 90 FPS (frames per second) in VR without updating the chunks. Second, interaction

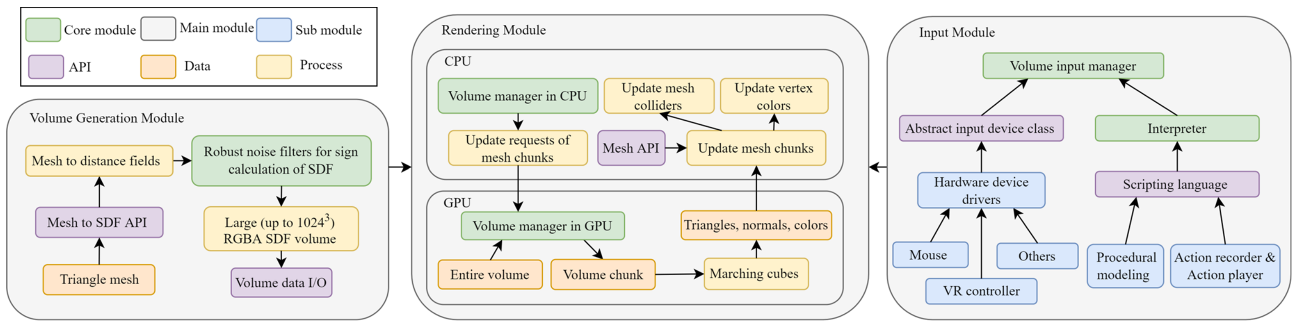


Fig. 10 A schematic diagram of our framework for real-time manipulation and editing of large volume data

with a rigid body requires convex mesh colliders, which can produce a better shape in a 16^3 chunk size since convex mesh colliders reduced the mesh to 255 triangles. Moreover, a 512^3 volume has better performance than a 1024^3 volume with a chunk size of 16^3 because of the lower number of chunks and draw calls. Although non-convex mesh colliders can handle raycast-based interactions (e.g., brush input using the mouse), it has poor performance for collision detection with colliders. Therefore, a 512^3 volume with a chunk size of 16^3 is one of the ideal settings for collision detection on a mesh collider generated from a large volume in VR. However, convex mesh colliders cannot handle a small brush radius such as 3. Although a non-convex mesh collider can resolve this issue, it does not have a good performance on a large mesh. One possible solution is to implement mesh simplification for facilitating the generation of mesh colliders.

The integration of other input devices is similar to the mouse and the VR controller. Users only need to write a class to implement a derived class based on an abstract class (for input device integration) provided by our framework. It is convenient to integrate a new input device as long as the device drivers from manufacturers are available. For example, the tracking positions from six degrees of freedom (6 dof) hand gestures (e.g., Meta Quest hand tracking) [32], and haptic devices (e.g., Phantom Omni) can be set as the input in the derived input device classes. Users can use additional input devices with tracking ability in our framework by implementing new input device classes which are derived from the abstract class. Such cross-device input support is one of the important features of our framework.

The volume input manager in our framework is equivalent to CSG. Therefore, its input actions can be easily reconstructed by scripting. We provide such scripting language consisting of a set of application programming interface (API) and an interpreter that can convert the script to input actions to the volume input manager. Users can also build 3D models by writing the scripting language (see Fig. 9). It is also possible to use it for other procedural modeling processes like run-time terrain deformation. We also implement an action recorder and its player in our framework based on the scripting language and interpreter. The recorded input actions can be used to reconstruct a CSG-based volume model. Artists can use the recorded input actions to spread their volume artworks instead of using the volume file itself. The size of the input actions file is very small compared to the size of the volume files. It could be an alternative volume media for such CSG-based volume models.

3.5 Discussion

Figure 10 shows the three main modules in our framework. The algorithm and optimization details have been presented hereinbefore, among which Sects. 3.1 and 3.2 introduce a robust and efficient method of the generation of large SDF volume, which is an essential component of our framework. It allows users to convert usual triangle mesh to high quality large volume data in an efficient way compared to prior work. Our SVF filters do not require a full iteration of all triangles but GWN does. It means SVF filters can save much time for SDF volume generation from large triangle meshes. Moreover, our method is GPU-friendly and can benefit further from the powerful massive parallel GPU computation capabilities compared to the GWN algorithm. Section 3.3 introduces a GPU-based mesh generation pipeline for large volume data with a novel SDF dictionary method to save GPU memory. Section 3.4 introduces the function of the input module in our framework. We also discuss several challenges resulting from the application of the usual chunk system when our framework is integrated into a game engine. The benefits of convenient interaction components (e.g., mesh collider and input system) brought by game engines come with some limitations (e.g., FPS drops), which need a trade-off and further improvement. 3D sculpting is also one of the main applications of interactive volume. However, a sculpting software or general modeling software is mostly designed for single-input devices such as the mouse and VR controller, which do not provide native support for cross-device input. Therefore, cross-device input support is one of the main features of our framework (see Table 3).

Table 3 Comparison of different software or frameworks (our framework (ours), (a) professional 3D modeling software (e.g., 3D Coat[a]); (b) Tilt Brush[b], and (c) other interactive volume frameworks for game engines (e.g., MudBun[c]) for 3D modeling or 3D painting

Features	Ours	(a)	(b)	(c)
Cross-device input support	Y	N	N	N
Large volume ($\geq 1024^3$) support	Y	Y	N	N
Game integration	Y	N	Y	Y
CUDA integration	Y	Y	N	N

[a] https://3dcoat.com/
[b] https://www.tiltbrush.com/
[c] https://longbunnylabs.com/mudbun/

Our motivation is that there is no software or framework that provides both proper generation of large volume data and sufficient editing functions for large volume data, which supports different platforms (e.g., VR and desktop), various input devices, and different interaction scenarios. We believe our framework can fill this gap. With our framework, artists can create artistic expressions with a high level of detail by using creative input methods. Also, HCI and visualization researchers can use it to explore interaction techniques for large volume data with various input devices and on different platforms. Moreover, since our framework is open-sourced, we plan to collect feedback from researchers and other users who use it in various scenarios to continue its improvement and development.

4 Limitations and future work

The experimental results demonstrated our proposed approaches can efficiently deal with large volume generation, interaction, and visualization. Here, we present the challenges ahead to further develop and improve this framework in the future. A potential issue is that the SVF4 is likely to over-correct the signs of voxels when processing small size of volumes (e.g., one tooth is missing in the 128^3 volume in Fig. 4c). However, in our experiments, such over-correction is not a problem when the volume size is large enough (e.g., the missing tooth in 128^3 does not occur in the 256^3 and 1024^3 volume in Fig. 4c–e). In the future, we plan to improve the SVF4 for better performance on the small SDF volume.

We did not optimize the GPU pipeline of marching cubes implementation in order to keep the framework simple. Instead of improving the efficiency of the marching cubes pipeline, we assign an individual thread to it to avoid FPS drop (see Table 2). However, a more efficient marching cubes pipeline will increase the efficiency of user input. Therefore, we plan to optimize the marching cubes pipeline in the future.

In addition, although the volume data can be exported and imported as a binary file in our framework, we have not explored improving the storage efficiency of the large volume data. For instance, a 1024^3 RGBA SDF volume (in ushort2 format without further compression) can produce a 4 GB file. For a 4 GB RGBA SDF volume of the Stanford bunny, it can be compressed to 1.12 GB using LZMA, an optimized version of the LZ77 compression algorithm.[3] However, we have not used compression methods on the volume structure except for the ushort2 dictionary. An efficient volume compression method could be added to the framework. Similarly, along with improving storage efficiency, future work can also explore optimization mechanisms to increase memory efficiency. The generation of a 1024^3 RGBA SDF volume requires about 5 GB of GPU memory. This means that it is difficult to generate a larger RGBA SDF volume, such as 2048^3, which may require more than 40 GB of GPU memory. As such, it will be useful to have a more efficient mechanism that can lower memory requirements to make interacting with larger volume more accessible.

As described earlier (see Sect. 1), we did not implement raycasting for the volume rendering pipeline to allow richer collision-based interactions. Gesture-based interactions without collision detection or the generation of a mesh and mesh colliders can provide better performance for raycasting. We plan to implement raycasting for efficient interactions that require no collision detection. In addition, the parallel computation of the framework is implemented in CUDA and integrated with Unity to take advantage of user input and multiple GPUs. Compute Shader, while not as mature as CUDA for parallel computing, can be an alternative to open the framework to a wider range of GPUs. This approach and others can be explored in the future. We will also try to fine-tune existing large language models (LLMs) to create a new model that is specific to the use case of our scripting language via approaches like low-rank adaptation (LoRA), a training method that accelerates the training of large models while consuming less memory [13].

5 Conclusion

This paper introduced a framework for real-time interaction with large volume data. The framework contains a series of basic blocks that have been fully implemented in this work, including a robust and efficient approach for the sign calculation of large SDF volume data generation, a GPU optimization strategy for enabling interactive operations with the 1024^3 RGBA SDF volume, and several brush editing tools. Examples with 2D and 3D input modalities have shown positive results. The framework can be easily adapted to support projects that require large interactive volume applications. It can also benefit interactive operations with large volume

[3] https://www.7-zip.org/7z.html.

data that require a high level of visual details. The code is available on GitHub[4] to allow other researchers to use it and conduct further research with it.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s00371-023-02976-x.

Acknowledgements We thank the reviewers for their valuable time and insightful comments that helped improve our paper. This research was partly funded by Xi'an Jiaotong-Liverpool University Special Key Fund (#KSF-A-03), the National Natural Science Foundation of China (#62272396), and XJTLU Research Development Fund (#RDF-21-02-065, #RDF-19-02-11).

Declarations

Conflict of interest There are no conflicts of interest to report. Other relate data and materials can be available upon reasonable request to the corresponding author.

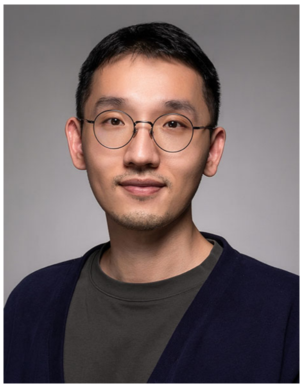
References

1. Andreas Bærentzen, J.: Robust generation of signed distance fields from triangle meshes. Volume Graphics 2005 Eurographics/IEEE VGTC Workshop Proceedings—4th International Workshop on Volume Graphics (IMM), pp. 167–175 (2005). https://doi.org/10.1109/vg.2005.194111
2. Boada, I., Navazo, I., Scopigno, R.: Multiresolution volume visualization with a texture-based octree. Vis. Comput. **17**(3), 185–197 (2001)
3. Bürger, K., Krüger, J., Westermann, R.: Direct volume editing. IEEE Trans. Vis. Comput. Graph. **14**(6), 1388–1395 (2008). https://doi.org/10.1109/TVCG.2008.120
4. Chittenden, T.: Tilt Brush painting: chronotopic adventures in a physical-virtual threshold. J. Contemp. Paint. **4**(2), 381–403 (2018). https://doi.org/10.1386/jcp.4.2.381_1
5. Cirne, M.V.M., Pedrini, H.: Marching cubes technique for volumetric visualization accelerated with graphics processing units. J. Braz. Comput. Soc. **19**(3), 223–233 (2013). https://doi.org/10.1007/s13173-012-0097-z
6. Cristie, V., Berger, M., Bus, P., Kumar, A., Klein, B.: CityHeat, pp. 1–4 (2015). https://doi.org/10.1145/2818517.2818527
7. Dyken, C., Ziegler, G., Theobalt, C., Seidel, H.P.: High-speed marching cubes using histopyramids. Comput. Graph. Forum **27**(8), 2028–2039 (2008). https://doi.org/10.1111/j.1467-8659.2008.01182.x
8. Gibson, S.F.: Constrained elastic surface nets: Generating smooth surfaces from binary segmented data. In: International Conference on Medical Image Computing and Computer-Assisted Intervention, pp. 888–898 (1998)
9. Goetz, F., Junklewitz, T., Domik, G.: Real-Time Marching Cubes on the Vertex Shader. Eurographics **2005**, 1–4 (2005)
10. Guthe, S., Wand, M., Gonser, J., Straßer, W.: Interactive rendering of large volume data sets. Proc. IEEE Vis. Confer. **D**, 53–60 (2002). https://doi.org/10.1109/visual.2002.1183757
11. Hoetzlein, R., Höllerer, T.: Interactive water streams with sphere scan conversion. Proceedings of I3D 2009: The 2009 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games **1**(212), 107–114 (2009). https://doi.org/10.1145/1507149.1507166
12. Hormann, K., Agathos, A.: The point in polygon problem for arbitrary polygons. Comput. Geom. **20**(3), 131–144 (2001). https://doi.org/10.1016/s0925-7721(01)00012-8
13. Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W.: LoRA: low-rank adaptation of large language models, pp. 1–26 (2021)
14. Jacobson, A., Kavan, L., Sorkine-Hornung, O.: Robust inside-outside segmentation using generalized winding numbers. ACM Trans. Graph. (2013). https://doi.org/10.1145/2461912.2461916
15. Jeremias, P., Quilez, I.: Shadertoy: live coding for reactive shaders. In: ACM SIGGRAPH 2013 Computer Animation Festival, p. 1 (2013)
16. Johansson, G., Carr, H.: Accelerating marching cubes with graphics hardware. In: Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research, pp. 39–es (2006)
17. Jones, M.W., Bærentzen, J.A., Sramek, M.: 3D distance fields: a survey of techniques and applications. IEEE Trans. Vis. Comput. Graph. **12**(4), 518–599 (2006). https://doi.org/10.1109/TVCG.2006.56
18. Ju, T., Losasso, F., Schaefer, S., Warren, J.: Dual contouring of Hermite data. In: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques, pp. 339–346 (2002)
19. Kasson, J.M., Plouffe, W., Nin, S.I.: Tetrahedral interpolation technique for color space conversion. In: Device-Independent Color Imaging and Imaging Systems Integration, vol. 1909, pp. 127–138 (1993)
20. Krayer, B., Müller, S.: Generating signed distance fields on the GPU with ray maps. Vis. Comput. **35**(6–8), 961–971 (2019). https://doi.org/10.1007/s00371-019-01683-w
21. Laine, S., Karras, T.: Efficient sparse voxel octrees. IEEE Trans. Vis. Comput. Graph. **17**(8), 1048–1059 (2010)
22. Liu, F., Kim, Y.J.: Exact and adaptive signed distance fields computation for rigid and deformable models on GPUS. IEEE Trans. Vis. Comput. Graph. **20**(5), 714–725 (2014). https://doi.org/10.1109/TVCG.2013.268
23. Lorensen, W.E., Cline, H.E.: Marching cubes: a high resolution 3D surface construction algorithm. In: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987, vol. 21, no. 4, pp. 163–169 (1987). https://doi.org/10.1145/37401.37422
24. Mawhorter, P., Mateas, M.: Procedural level generation using occupancy-regulated extension. In: Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games, pp. 351–358 (2010)
25. Museth, K., Lait, J., Johanson, J., Budsberg, J., Henderson, R., Alden, M., Cucka, P., Hill, D., Pearce, A.: OpenVDB: an open-source data structure and toolkit for high-resolution volumes. In: ACM SIGGRAPH 2013 Courses, p. 1 (2013)
26. Newman, T.S., Yi, H.: A survey of the marching cubes algorithm. Comput. Graph. **30**(5), 854–879 (2006)
27. Nooruddin, F.S., Turk, G.: Simplification and repair of polygonal models using volumetric techniques. IEEE Trans. Vis. Comput. Graph. **9**(2), 191–205 (2003)
28. Qin, J., Chui, Y.P., Pang, W.M., Choi, K.S., Heng, P.A.: Learning blood management in orthopedic surgery through gameplay. IEEE Comput. Graph. Appl. **30**(2), 45–57 (2010). https://doi.org/10.1109/MCG.2009.83
29. Schaefer, S., Warren, J.: Dual marching cubes: primal contouring of dual grids. Comput. Graph. Forum **24**(2), 195–201 (2005). https://doi.org/10.1111/j.1467-8659.2005.00843.x
30. Schneider, J., Westermann, R.: Compression domain volume rendering. In: IEEE Visualization, 2003. VIS 2003, pp. 293–300 (2003). https://doi.org/10.1109/VISUAL.2003.1250385

[4] https://github.com/Chaosikaros/LVDIF.

31. Sherbondy, A., Houston, M., Napel, S.: Fast volume segmentation with simultaneous visualization using programmable graphics hardware. In: IEEE Visualization, 2003. VIS 2003, pp. 171–176 (2003)
32. Shi, R., Zhang, J., Yue, Y., Yu, L., Liang, H.N.: Exploration of bare-hand mid-air pointing selection techniques for dense virtual reality environments. In: Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems, CHI EA '23. Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3544549.3585615
33. Shi, R., Zhu, N., Liang, H.N., Zhao, S.: Exploring head-based mode-switching in virtual reality. In: 2021 IEEE International Symposium on Mixed and Augmented Reality (ISMAR), pp. 118–127 (2021). https://doi.org/10.1109/ISMAR52148.2021.00026
34. Treib, M., Reichl, F., Auer, S., Westermann, R.: Interactive editing of GigaSample terrain fields. Comput. Graph. Forum **31**(2), 383–392 (2012). https://doi.org/10.1111/j.1467-8659.2012.03017.x
35. Tzevanidis, K., Zabulis, X., Sarmis, T., Koutlemanis, P., Kyriazis, N., Argyros, A.: From multiple views to textured 3d meshes: a GPU-powered approach. In: Proceedings of the 11th European Conference on Trends and Topics in Computer Vision-Volume Part II, pp. 384–397 (2010)
36. Wang, J., Shi, R., Xiao, Z., Qin, X., Liang, H.N.: Effect of render resolution on gameplay experience, performance, and simulator sickness in virtual reality games. Proc. ACM Comput. Graph. Interact. Tech. (2022). https://doi.org/10.1145/3522610
37. Wang, J., Shi, R., Zheng, W., Xie, W., Kao, D., Liang, H.N.: Effect of frame rate on user experience, performance, and simulator sickness in virtual reality. IEEE Trans. Vis. Comput. Graph. **29**(05), 2478–2488 (2023). https://doi.org/10.1109/TVCG.2023.3247057
38. Wu, J., Wang, D., Wang, C.C., Zhang, Y.: Toward stable and realistic haptic interaction for tooth preparation simulation. J. Comput. Inf. Sci. Eng. **10**(2), 1–9 (2010). https://doi.org/10.1115/1.3402759

Jialin Wang is currently a Ph.D student at Xi'an Jiaotong-Liverpool University, Suzhou, China. His re-search interests focus on virtual reality, robotics, and computer graphics.



Nan Xiang is a Lecturer at the Department of Computing, Xi'an Jiaotong-Liverpool University, Suzhou, China. He received his B.S degree (2012) in software engineering from Nanchang University (China), MA (2017) in animation from Communication University of China, and Ph.D (2021) in computer animation from National Centre for Computer Animation, Bournemouth University (UK). His current research interests include 3D reconstruction, computer animation, virtual surgery, and XR technologies.

Navjot Kukreja is a Lecturer (Assistant Professor) in the Department of Computer Science at the University of Liverpool. His main area of interest is in parallel and distributed computing, especially high-performance computing. He looks at this from an angle of Domain-specific languages, "Can we describe the problem at a high level and then use code generation and just-in-time compilation to do the performance optimizations automatically?". He has worked on PDE solvers, specifically inverse problems based on PDEs—around building images of the earth. His more recent work is in the direction of integrating statistical methods including deep learning into such a toolchain.

Lingyun Yu is an Associate Professor at the Department of Computing, Xi'an Jiaotong-Liverpool University, Suzhou, China. She obtained the PhD degree on Scientific Visualization and Interaction Techniques from the University of Groningen in 2013. After that, she worked as a Lecturer and Associate Researcher at Hangzhou Dianzi University from 2014 to 2017, and a Postdoctoral Research Fellow at the University of Groningen and the University Medical Center Groningen from 2016 to 2019. Her research focuses on interactive visualization, immersive visualization, human–computer interaction and virtual/augmented reality.

Hai-Ning Liang is Professor of Computing and inaugural Head of the Department of Computing at Xi'an Jiaotong-Liverpool University (XJTLU), Suzhou, China. He is also Deputy Director of the Suzhou Key Laboratory of Intelligent Virtual Engineering and the XJTLU Virtual Engineering Center. He completed his PhD in Computer Science from Western University, Canada. Prior to joining XJTLU, he was with the University of Queensland in Australia and the University of Manitoba in Canada. He does research in human–computer interaction, focusing on virtual/augmented reality and gaming technologies.