ICVG: PRACTICAL CONSTRUCTIVE VOLUME GEOMETRY FOR INDIRECT VISUALIZATION

Mark Laprairie, Howard J. Hamilton, and Andrew Geiger

Department of Computer Science, University of Regina, Regina, Canada

ABSTRACT

The task of creating detailed three dimensional virtual worlds for interactive entertainment software can be simplified by using Constructive Solid Geometry (CSG) techniques. CSG allows artists to combine primitive shapes, visualized through polygons, into complex and believable scenery. Constructive Volume Geometry (CVG) is a super-set of CSG that operates on volumetric data, which consists of values recorded at constant intervals in three dimensions of space. To allow volumetric data to be integrated into existing frameworks, indirect visualization is performed by constructing and visualizing polygon meshes corresponding to the implicit surfaces in the volumetric data. The Indirect CVG (ICVG) algebra, which provides constructive volume geometry operators appropriate to volumetric data that will be indirectly visualized is introduced. ICVG includes operations analogous to the union, difference, and intersection operators in the standard CVG algebra, as well as new operations. Additionally, a series of volumetric primitives well suited to indirect visualization is defined.

KEYWORDS

Constructive solid geometry, constructive volumetric geometry, surface construction, isosurface, virtual volume sampling function

1. INTRODUCTION

Volumetric data consists of values recorded at constant intervals in three dimensions of space. In mathematics, volumetric data can be treated as a 3D scalar field. In practise, these data sets are stored in three dimensional array structures. Such data are utilized in a variety of tasks, including scientific visualization, medical imaging, and interactive entertainment. A software system that takes volumetric data, processes them, and produces visualization is called a *voxel system*. One method for visualizing volumetric data is to first perform *surface construction* and then visualize the resulting surface [1]. Surface construction, also called *isosurface extraction* or *surface reconstruction*, extracts information about surfaces that are implicitly present in the volumetric data [2] and uses this information to build geometric meshes corresponding to the surfaces. Visualizing volumetric data through surface construction is referred to as *indirect visualization*.

In computer graphics, *Constructive Solid Geometry* (CSG) is a modeling technique that facilitates the rapid construction of geometric data [3]. Complex scenes can be constructed by applying a series of Boolean operations, such as *union*, *intersection*, and *difference*, to small sets of 3D objects. *Constructive Volume Geometry* (CVG) is an extended version of the CSG technique that operates on volumetric data instead of conceptually solid objects [4]. The naive approach to implementing CVG leads to visual artifacts when indirect visualization is performed on the resulting volumetric data.

This paper focuses on techniques for performing CVG that reduce the number of visual artifacts when indirect visualization is performed. The discussion is restricted to volumetric data where all values are non-negative integers. This restricted domain is appropriate for indirect visualization techniques because they primarily work with density data. The Indirect CVG (ICVG) algebra, which provides constructive volume geometry operators appropriate to volumetric data that will be indirectly visualized, is introduced. ICVG includes operations analogous to the union,

DOI : 10.5121/ijcga.2017.7201

intersection, and difference operators in the standard CVG algebra, as well as several new operations. Additionally, a series of volumetric primitives well suited to indirect visualization is defined.

The remainder of this paper is organized as follows. Section 2 describes background material on CSG, volumetric visualization, CVG, and voxelization. Section 3 discusses practical issues related to developing an indirect CVG system, as well as the changes made to the CVG algebra to create the ICVG algebra. Section 4 presents experimental results. Section 5 concludes and summarizes the findings. Possible future extensions to the approach are also described.

2. BACKGROUND

This section surveys relevant previous work. Section 2.1 presents the main concepts of CSG and Section 2.2 describes Boolean operations for CSG. Section 2.3 discusses techniques for visualizing volumetric data. Section 2.4 describes the CVG algebraic framework. Finally, Section 2.5 describes methods of voxelization.

2.1. Constructive Solid Geometry

As mentioned previously, CSG is a modeling technique that allows for the rapid construction of geometric data [3]. The technique's effectiveness depends on the idea of combining well-defined, relatively simple 3D objects called *primitives* through the use of Boolean operators. Since well-defined objects are used as the basis for construction of a 3D geometric object, here called *geometry*, the resulting compound object retains the same well-defined nature as the primitives from which it was made.

The conceptual simplicity of working with a few, relatively simple, primitives enable users to create models quickly, as is required in the interactive entertainment industry. Having only a few primitive objects, an artist can easily conceptualize the construction of a scene through its parts. This conceptualization is similar to how a cartoon artist first defines the main geometric form of a character and then fills in details.

CSG primitives are easily combined into more complex and interesting shapes called *compound objects*. In Figure 1, three common primitives are shown: a sphere, a cube, and a cylinder. These primitives are easily defined mathematically, and thus the mathematical properties of any resulting compound object can be readily calculated.

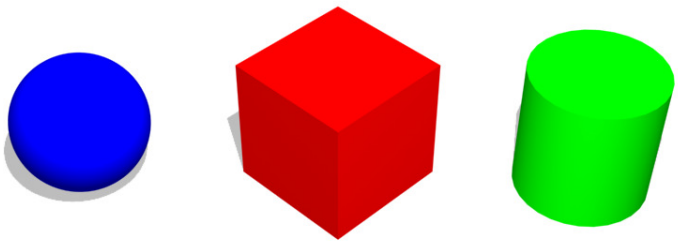


Figure 1. Constructive Solid Geometry primitives

The well-defined nature of CSG is suited to engineering tasks [5]. Using this method of construction allows artists and engineers to produce geometry that is guaranteed to be solid [3]. For example, if an engineer wishes to design an engine compartment that is water tight in a Computer Aided Design (CAD) environment, CSG can help. The engineer can start with a solid cube, and then use other primitives to carve cavities within it. If no action is performed that pierces the surface of the cube, then the resulting cavity is guaranteed to be fully enclosed and thus water tight.

In interactive entertainment, CSG has gained popularity as a method for modelling indoor environments. Valve Software's *Hammer* editor implements CSG [6]. *Hammer* has been used successfully in commercial computer games, including *Half-Life 2*, *Portal*, *Team Fortress 2,* and *Counter-Strike Source*. The editor provides wedge, torus, spike, sphere, cylinder, block, and arch primitives, which can be combined using CSG operations into more complex shapes.

2.2. Boolean Operators for CSG

The operators used in CSG have traditionally been called *Boolean operators* [3]. Boolean operators allow CSG to describe novel shapes through a series of relations between two objects. The most common operators are: *union*, *intersection,* and *difference*.

2.2.1. Union (∪)

The union of primitives A and B produces a new piece of geometry that encompasses the volume of both A and B. Figure 2 shows A ∪ B, the union of a cube A and a sphere B, for a particular combination of locations and sizes of these primitives. Union is symmetric, i.e., $A \cup B = B \cup A$.

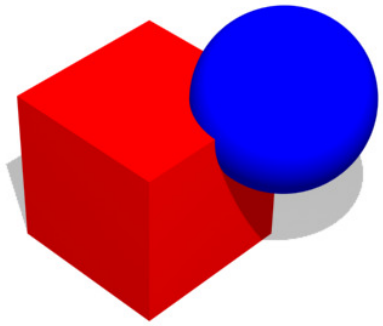


Figure 2. Union

2.2.2. Intersection (∩)

The intersection of primitives A and B results in a new piece of geometry that contains only the volume where A and B overlap. Figure *3* shows A ∩ B, the intersection of a cube A and a sphere B; the removed parts of the cube and sphere are shown with transparent shapes. Intersection is also symmetric, i.e., $A \cap B = B \cap A$.

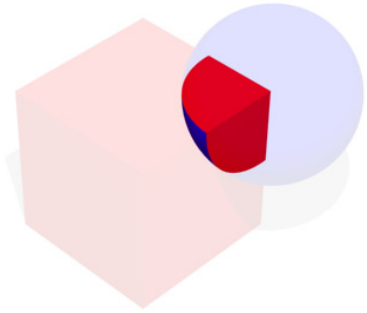


Figure 3. Intersection

2.2.3. Difference (−)

The difference of primitives A and B results in a new piece of geometry that encompasses the volume of A, less the shared volumes of A and B. In practice, this operator is often referred to as

the *carve* operator, because it allows artists to carve into a piece of geometry [7]. Figure 4 shows A − B, the difference of a cube A and a sphere B; the location of the sphere is shown with a transparent shape. Difference is asymmetric, i.e., in general $\boldsymbol{A - B \neq B - A}$.

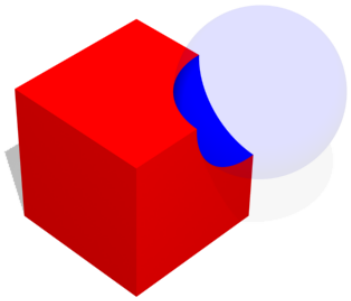


Figure 4. Difference

2.3. Volumetric Visualization

Volumetric data is comprised of an array of values, each representing a *voxel.* A voxel is a volumetric primitive that occupies a single unit of three-dimensional space. A voxel can have various associated values, such as density or temperature. *Volumetric density data* is comprised of an array of values, each of which stores a single value for density. In this array representation for volumetric data, the position of a voxel in three-dimensional space is inferred from its location in the array.

To visualize volumetric data, two broad approaches are used. In the *direct approach,* each value is directly visualized using a process called *volume rendering* [8]. Volume rendering works directly with the data to produce visualization through a variety of techniques, including ray casting [9], splatting [10], and shear warp [8]. In the *indirect approach* (or *polygon approach*), a three-dimensional *polygon mesh* representing the surface is constructed and then visualized. Density values at or above an arbitrary *surface threshold* (or *extraction threshold*) value are said to describe parts of solid objects. Similarly, values below this surface threshold are said to be outside the solid objects. The polygon mesh typically consists of a list of vertices and edges, which can be displayed in a three-dimensional rasterization system such as OpenGL or DirectX. A polygon mesh built from triangles can be readily and efficiently visualized and manipulated with modern graphics processors. This paper focuses on CVG techniques and primitives well-suited to the indirect visualization of volumetric density data.

The best-known method for surface construction from volumetric data is the *marching cubes* algorithm [11, 12]. The algorithm processes eight neighbouring locations in the voxel data to form an imaginary cube called a *cell*. The algorithm then iterates through all possible cells in the voxel data. For each cell, the algorithm decides whether or not it partitions the object from the outside world. In other words, it detects cells that form the boundary of the object. If a cell does so, then it is a part of the surface, and the algorithm constructs a small number of triangles that form the portion of the surface corresponding to the cell. The collection of all such triangles forms the polygon mesh.

To construct the triangles corresponding to a cell, the algorithm utilizes a look up table of polygon shapes corresponding to all possible partitions. Each cell is examined to determine which of the eight sampling points are within the surface. There are a total of 256 (2^8) possible configurations for each cell. The total number of configurations can be reduced by considering reflections and symmetrical rotations of each configuration. Figure 5 illustrates the fifteen unique cell configurations shown in the original marching cubes paper [11]. The cube configurations are designed in such a way that, when placed together, they produce a visibly solid surface.

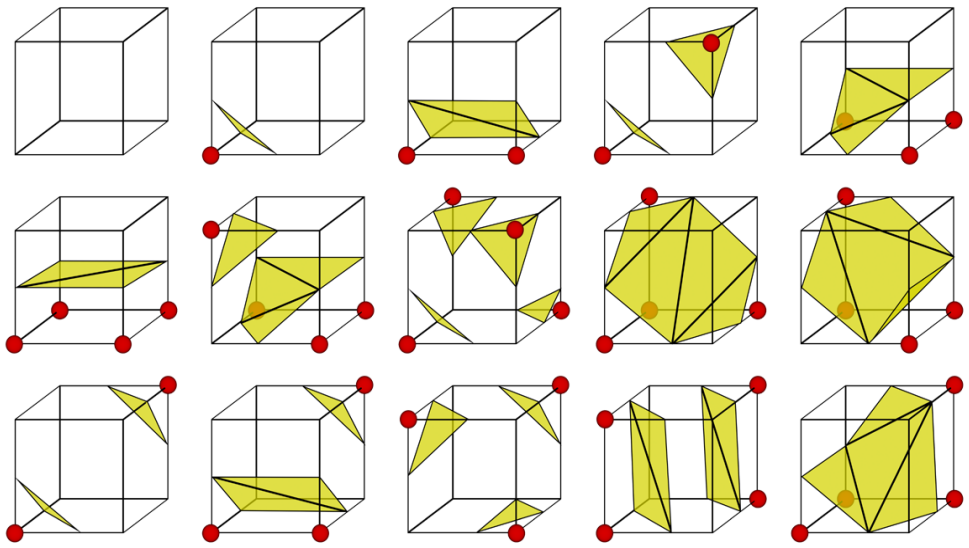


Figure 5. Unique cell configurations for the marching cubes algorithm [11]

To create more accurate surfaces, the vertices of the polygon mesh are manipulated. Each vertex of the polygon mesh is positioned on its cell's edge such that it is linearly interpolated between the two voxel values connected to that edge. The final position of the vertex on a cell's edge corresponds to the surface threshold position as determined by interpolation. This step modifies the visibly solid surface, produced by converting the volume data into the corresponding cube configurations, to better fit the surface implicit in the volume data.

Voxel systems for indirect visualization are able to use external colouring information through *texture mapping*. Texture mapping applies surface detail supplied by an external raster image file onto the constructed surface. This mapping can be done through a variety of methods, often using a method called *triplaner mapping* [11]. To allow for multiple mappings of different textures, a material identifier parameter can be added to each voxel to specify which raster file to use when performing the mapping. Williams proposed the use of discrete material identifiers for the voxel values to allow for multiple such mappings instead of colouring information [13]. The value of the identifier tells which material should be placed on a given portion of surface. The use of a texture map makes the number of texture elements (texels) independent of the resolution of the volumetric data. This independence allows for high quality colouring of visualizations of volumetric data without the overhead of storing per-voxel colouring information.

2.4. Constructive Volume Geometry

Chen and Tucker devised the *Constructive Volume Geometry* (*CVG*) framework for describing algebraic operations on volumetric data [4, 14]. The primary deficiency of CSG is its inability to describe the interior of the objects it is defining. CVG rectifies this deficiency by working with objects defined by mathematical scalar fields, which are treated as analogues to volumetric data. The CVG algebra provided by Chen and Tucker was intended to encourage further discussion rather than serve as a final standard [4].

In their algebra, Chen and Tucker considered voxels with four distinct parameters: opacity (O) (here analogous to density) and three colour fields representing red (R), green (G) and blue (B). Chen and Tucker defined versions of these operators suited for a voxel system where the value for each voxel is composed of four components, opacity, red, green, and blue. Thus, colour information is assumed to be embedded in the volumetric data.

Chen and Tucker define a *spatial object* as a tuple $\mathbf{o} = (O, A_1 \ldots A_k)$ of scalar fields defined in $\mathbb{R}^3$, including an *opacity* field $O : \mathbb{R}^3 \rightarrow [0, 1]$ specifying the "visibility" of every point p in $\mathbb{R}^3$ and possibly other *attribute* fields $A_1 \ldots A_k : \mathbb{R}^3 \rightarrow \mathbb{R}, k > 0$ [4]. They define the traditional CSG operations of union, intersection, and difference for objects $\mathbf{o}_1$ and $\mathbf{o}_2$ as functions over their component opacity fields O_1 and O_2 as follows:

$$\textit{union}: \quad \cup(\mathbf{o}_1, \mathbf{o}_2) = MAX(O_1, O_2) \tag{1}$$

$$\textit{intersection}: \quad \cap(\mathbf{o}_1, \mathbf{o}_2) = MIN(O_1, O_2) \tag{2}$$

$$\textit{difference}: \quad -(\mathbf{o}_1, \mathbf{o}_2) = (O_1 - O_2) \tag{3}$$

where *MAX* and *MIN* are as specified in the Appendix. Chen and Tucker showed that CVG is a superset of CSG [14]. They defined additional operators for CVG other than those available in CSG. These operators included cap and trim, which are as follows:

$$\textit{cap}: \quad cap(\mathbf{o}_1, \mathbf{o}_2) = (CAP(O_1 - O_2), CAP(R_1 - R_2), CAP(G_1 - G_2), CAP(B_1 - B_2)) \tag{4}$$

$$\textit{trim}: \quad trim(\mathbf{o}_1, \mathbf{o}_2) = (TRIM(O_1 - O_2), TRIM(R_1 - R_2), TRIM(G_1 - G_2), TRIM(B_1 - B_2)) \tag{5}$$

where *CAP* and *TRIM* are as specified in the Appendix.

The opacity class provided by Chen and Tucker was restricted to the range [0, 1]. Opacity values from 0 to 1 are well-suited to visualization. Based on the work of Chen and Tucker, Johnson and Tucker subsequently developed a formal approach to specifying a spatial object as a data type [15, 16]. They gave a useful algebra of continuous functions that has operations derived from operations on space and data, and is equipped with an appropriate topology. To the best of our knowledge, no previous CVG framework optimized for indirect visualization or using material identifiers has been described in the literature.

2.5. Voxelization

Other related research concerns *voxelization* (also known as *3D scan conversion*), which is the process of taking a geometric representation of a continuous 3D object and converting it to a voxel representation that approximates the continuous object. A naive approach would either test every voxel exhaustively for an intersection with the object or else perform recursive subdivision of the object. Unfortunately, the voxelizations generated by this approach are often too course and include more voxels than necessary [17]. A more refined approach was devised by Kaufman et al. [17]. They define two voxels to be 6-adjacent if they share a face, 18-adjacent if they share an edge or a face, and 26-adjacent if they share a vertex, an edge, or a face. Based on these definitions, they define notions of N-separating and covering and then propose that an effective voxelization should provide a minimal cover of the object. A more recent approach, called *topological voxelization*, is based on the topological properties of the objects [18]. Every voxel is given an associated geometric intersection target and the voxel is marked as solid if any input primitive intersects this target. Laine proved that selecting appropriate intersection targets results in voxelizations with desirable connectivity and separability properties. The various voxelization approaches were compared by Nourian and Zlatanova [19].

Related research addressed the challenge of performing fast voxelization using a Graphics Processing Unit (GPU). Schwarz and Seidel introduced a conservative surface voxelization technique, which marks all voxels as solid if they are partially or fully overlapped by a mesh's triangles [20]. The technique uses a triangle/box overlap test that can be adapted to yield a 6-separating surface voxelization with two important properties: it is thinner than the naive voxelization and it is still connected and gap-free. Recently, Baert et al. have devised an approach suited to objects represented as very large 3D meshes [21]. Their approach allows the input triangle mesh, the intermediate 3D voxel grid, and the voxelized output to all be larger than available memory.

Although the previous voxelization approaches are fast and effective, none of them address the problem of creating visual artefacts that is overcome by the method proposed in this paper.

3. CONSTRUCTIVE VOLUME GEOMETRY FOR INDIRECT VISUALIZATION

This section discusses the ICVG approach to implementing a voxel system that performs CVG for indirect visualization of volumetric density data. CVG was implemented for indirect visualization in a voxel system named *Isovox*. *Isovox* manipulates volumetric data and produces polygon meshes for visualization. Surface colours for the meshes are obtained by texture mapping from a texture external to the volumetric data. Section 3.1 presents assumptions for this work. Section 3.2 describes techniques for developing visually correct volumetric primitives for indirect visualization. Section 3.3 defines the ICVG algebra designed for indirect volumetric visualization.

3.1. Assumptions

All values in the volumetric data are assumed to be discrete and non-negative integers in the range [L, H], where L is the lowest possible value and H is the highest possible value. Throughout this paper, H, L, and S are used as values rather than variables. Such values are well-suited for volumetric density data, which are always non-negative. Floating point data would need to be converted to integers. For example, suppose a value is represented as a single byte of data. The use of a byte data type, as opposed to an unsigned integer, reduces memory requirements. For one byte voxels, the L is 0, and the H is 255. Following the 2011 release of the commercial volume editing tool *3D-Coat* [22], the surface threshold (S) is defined to be the midpoint between L and H, as shown in Equation 6.

$$S = \left\lfloor \frac{H-L}{2} \right\rfloor \tag{6}$$

For example, $S = \left\lfloor \frac{255-0}{2} \right\rfloor = 127$.

Throughout this paper, a right-handed coordinate system is used with right, up, and out corresponding to the positive X, Y, and Z axis. When performing indirect visualization, a voxel is visualized if its associated value in the volumetric data is greater than or equal to S.

3.2. ICVG Primitives

When defining primitives for indirect visualization of volumetric data, several factors need to be considered. The straightforward approach to specifying primitives may produce undesirable artifacts when indirect visualization is performed. Figure 6 illustrates the desired appearance of four common volumetric primitives, namely the cuboid, the ellipsoid, the cylinder and the cone. Within this section, Section 3.2.1 defines virtual volume sampling functions; Section 3.2.2 introduces clipping and padding; and Sections 3.2.3 through 3.2.6 describe the cuboid, ellipsoid, cylinder, and cone primitives, respectively.

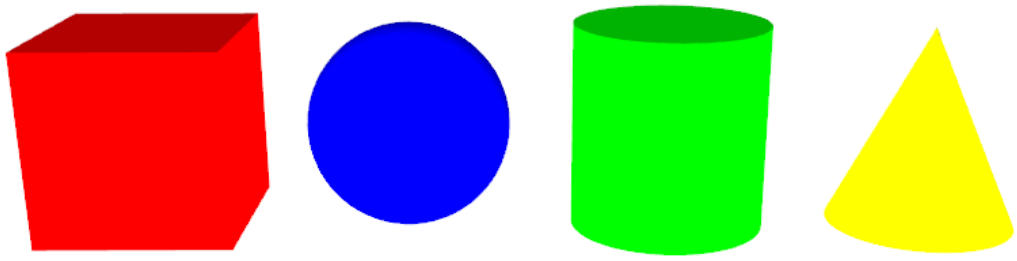


Figure 6. Primitives constructed from volumetric data sets

3.2.1. Virtual Volume Sampling Functions

A *virtual volume sampling function* is now defined. Let $\mathbb{N}$ represent the non-negative integers. Let $\vec{B} = (B_x, B_y, B_z) \in \mathbb{N}^3$ represent the upper bounds on a 3D space, called the *extension space* (also known as the *volume*), with origin (0, 0, 0). Thus, the width, height, and depth of the extension space correspond to B_x, B_y and B_z, respectively. Let $\vec{V} = (V_x, V_y, V_z) \in \mathbb{N}^3$, such that $V_x < B_x, V_y < B_y, V_z < B_z$, represent a position in the extension space. Thus, the positions in the extension space range from (0, 0, 0) to $(B_x - 1, B_y - 1, B_z - 1)$. One voxel is located at each position. A virtual volume sampling function, $f: \vec{V} \longrightarrow [\mathrm{L}, \mathrm{H}]$, specifies the density value for any position (i.e., voxel) in the extension space. The function f takes a position, called the *sampling position*, in an arbitrarily large extension space as input and returns as output a value in the range [L, H]. This output value represents a density.

Virtual volume sampling functions are resolution independent, meaning that a virtual volume can be sampled at any level of granularity by changing the resolution of the extension space. To fully reconstruct the virtual volume at a specified resolution, the virtual volume function is sampled at every point in an extension space at that resolution.

3.2.2. Clipping and Padding

As previously mentioned, the marching cubes algorithm operates on eight data points at a time. Thus, data points along the edge of the extension space cannot be processed in the same fashion as the interior data points. Some implementations of the marching cubes algorithm handle this difficulty by pretending the extension space is surrounded by voxels with L or H values. Suppose H values are used. If the surface of the object extends to any edge of the extension space, then holes will appear where the surface touches the edge. This artifact is referred to as *clipping*.

To address the clipping issue, it would seem sufficient to assign L values to voxels outside of the extension space, but doing so may make CVG operations on some objects less efficient. The constraints of an application may ensure that some portions of a constructed surface can never be viewed. If there are constraints of this fashion, then time will be wasted performing surface construction and visualization of these unseen boundaries. Suppose that mountainous terrain is created for a first person shooter game but that the player can never see the mountains from below the surface. Therefore any visualization of the bottom of these mountains is wasted.

(a) View from above (b) View from below

Figure 7. Hollow island surface displayed by *Isovox*

A more flexible solution to this problem is to provide additional voxel values called *padding* voxels surrounding the extension space of interest. In the case of the cuboid primitives, these

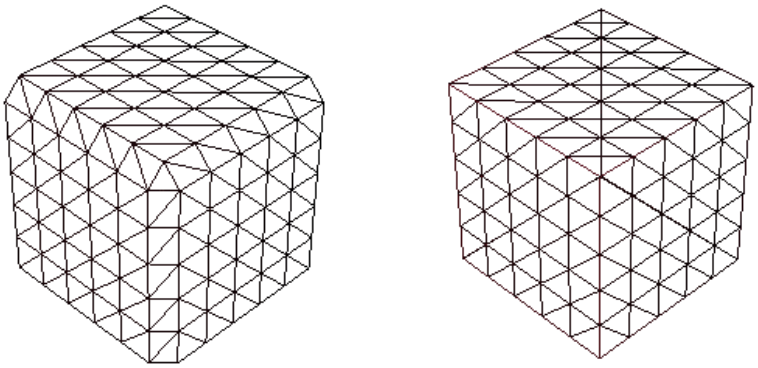
padding voxels are set to L, to ensure the creation of the expected surface mesh. In the mountainous terrain example, no such padding voxels would be placed at the bottom of the extension space, resulting in a visually correct, hollow mesh. If one desired the bottom portion of the mesh to be filled in with polygons, a set of padding L values could be added. Figure 7 (a) shows a visually correct unpadded terrain example and Figure 7 (b) shows the hollow underside view of the same surface.

3.2.3. Cuboid

The cuboid volumetric primitive is used for representing rectangular solids. The construction of a cuboid primitive for CVG may seem trivial, but two problems need to be addressed when designing such a primitive for indirect visualization: clipping and beveling.

The naive approach to defining a cuboid primitive is to simply fill a volume uniformly with H values; this approach suffers from clipping. The clipping problem can be avoided by padding the outside of the volume, where needed, with L values, as described in Section 3.2.2.

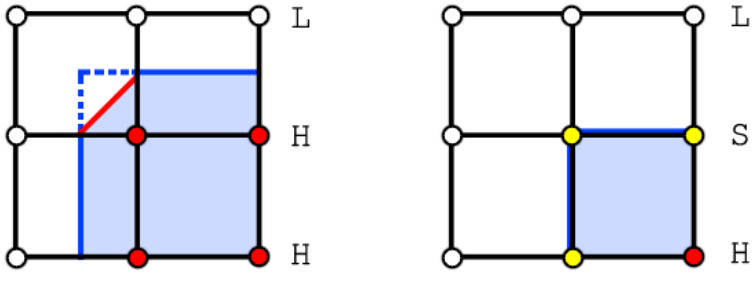
The second problem is beveling. When a cuboid primitive is visualized, it should have sharp corners. However, the naive definition of the cuboid primitive allows beveling artifacts to appear on the edges of the visualized surface.
Figure *8* (a) shows how a beveling artifact can have an undesirable smoothing effect on the cuboid primitive, shown here with a checkerboard texture to facilitate seeing the edges. Observe how the edges of the cuboid have a smooth, rounded appearance.



(a) With beveling artifact (b) Without beveling artifact

Figure 8. Cuboid primitive with and without beveling artifact

This problem can be easily understood by considering the two-dimensional case rather than the three-dimensional one. Figure 9 (a) illustrates the desired appearance of a square primitive with a dotted line, while the solid line shows how the actual beveled surface is constructed.



(a) Corner with beveling artifact (b) Corner with artifact removed

Figure 9. Two-dimensional view of beveling artifact

The beveling of the corner of the square primitive is an artifact of the marching cubes algorithm. When the corner cell is sampled, two points out of eight are within the boundary of the volume, corresponding to the third case in Figure 5. The look up table selects a diagonal corner piece to be placed in the cell. The vertices are positioned on their related cell edges using linear interpolation. With the naive cuboid definition, the interpolated surface needs to be between H and L density, which forces the vertices of the constructed triangles to be placed on the midpoints of the edges.

In ICVG, beveling is addressed by adding a thin layer (one voxel wide) of surface S value voxels surrounding the interior H values of the cuboid. Figure 9 (b) shows how adding these surface value voxels around a square can remove the beveling artifact. Although the look up function chooses a diagonal corner piece to be placed in the cell, as before, when the location of the vertices are interpolated between L and S, they are placed touching the S side. Figure 8(b) shows a cuboid primitive with the beveling artifact removed, resulting in sharp corners, as desired.
The ICVG cuboid sampling function f_{cu} is defined as follows:

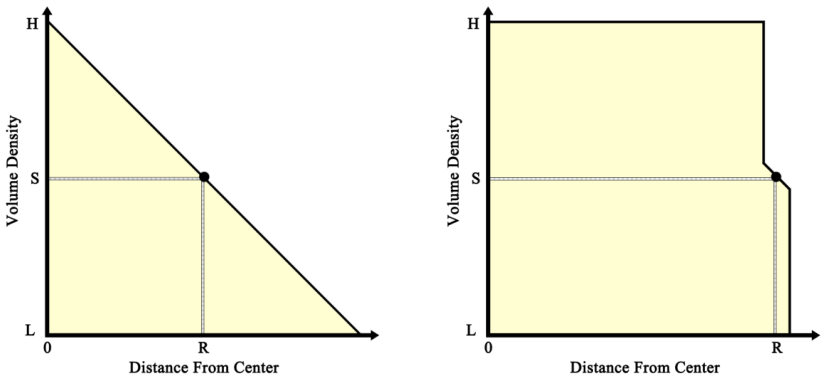
$$f_{cu}(\vec{V}) = \begin{cases} \text{L if } (V_x = 0) \vee (V_x = B_x - 1) \vee (V_y = 0) \vee (V_y = B_y - 1) \\ \qquad \vee (V_z = 0) \vee (V_z = B_z - 1), \\ \text{S otherwise if } (V_x = 1) \vee (V_x = B_x - 2) \vee (V_y = 1) \\ \qquad \vee (V_y = B_y - 2) \vee (V_z = 1) \vee (V_z = B_z - 2), \\ \text{H otherwise} \end{cases} \tag{7}$$

B_x, B_y, B_z are restricted to values greater than 2. Given a position $\vec{V}$ bounded within the volume dimensions $\vec{B}$, the cuboid sampling function f_{cu} returns the voxel value at that position, which is L, S, or H. If the position is touching the bounding volume, then it is assigned L. Positions that are one voxel away from the edges of the bounding volume are assigned S. All other positions are assigned H. This method of defining the cuboid sampling function overcomes the clipping and beveling problems described above.

An undesirable side effect of having the cuboid primitive contain three possible values instead of two (i.e., it contains L, H, and S values instead of only H and L values) is that the potential for compression is reduced. Generally, having less homogenous volumetric data reduces the expected compression ratio.

3.2.4. Ellipsoid

The ellipsoid volumetric primitive is used for representing spherical and other rounded shapes. The naive approach to the ellipsoid primitive would be to linearly interpolate values from H to L radically from the midpoint of the volume. Figure 10 (a) shows the density function along some axis where the ellipsoid has radius *R* that results from this approach. This approach produces a visually correct surface mesh, but it suffers from three flaws. First, much of the volume will be filled with wasted space, i.e., values outside of the surface mesh that do not affect construction. This wasted space means that the ellipsoid surface is relatively small compared to the size of the volume. Secondly, these invisible values may have undesired effects on CVG operations. Finally, the volumetric data resulting from this approach will be inefficient to compress because the values inside the surface and outside the surface, which do not affect surface construction, vary widely. Figure 10 (b) shows the density function corresponding to the ICVG virtual volume sampling function, which addresses the problems with the naive approach. Only a small amount of extension space is allocated past the radius *R*, and values well below this threshold are assigned values of H.



(a) Naive density function (b) ICVG density function

Figure 10. Density functions for ellipsoid primitives

Let $\vec{R} = (R_x, R_y, R_z) \in \mathbb{N}^3$, where $1 \leq R_x \leq \left\lfloor \frac{B_x}{2} \right\rfloor$, $1 \leq R_y \leq \left\lfloor \frac{B_y}{2} \right\rfloor$, and $1 \leq R_z \leq \left\lfloor \frac{B_z}{2} \right\rfloor$ represent the desired radius in the X, Y, and Z axes. Let, $\vec{T} = (T_x, T_y, T_z) \in \mathbb{N}^3$, such that $\vec{T} = (\vec{V} - \vec{R})$, represent the position in the volume translated so that (0, 0, 0) corresponds as nearly as possible to the center of the ellipsoid. Additionally the $EllipsoidBody: \vec{P} \rightarrow (0, \infty)$ function is defined as $EllipsoidBody(\vec{P}) = \left(\frac{P_x^2}{R_x^2} + \frac{P_y^2}{R_y^2} + \frac{P_z^2}{R_z^2}\right)$, which corresponds to the well-known ellipsoid function. The ellipsoid sampling function f_{el} is defined as follows:

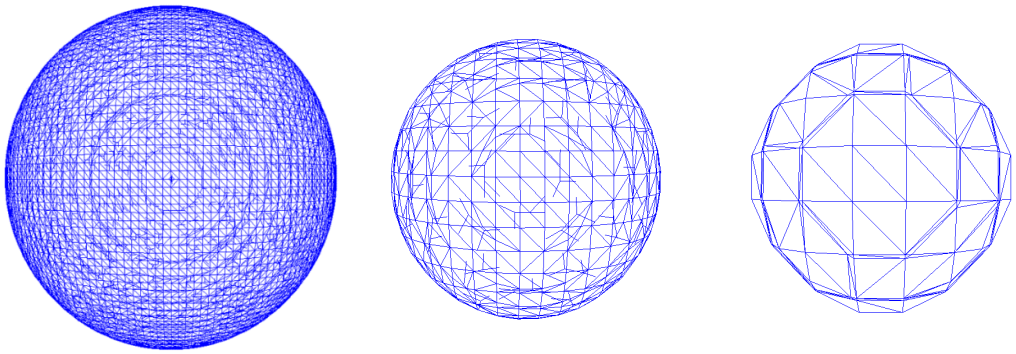
$$f_{el}(\vec{V}) = \begin{cases} \text{H} & \text{if } EllipsoidBody(\vec{T} + \hat{T}) < 1, \\ \text{L} & \text{if } EllipsoidBody(\vec{T} - \hat{T}) > 1, \\ \left\lfloor \frac{\text{S}}{EllipsoidBody(\vec{T})} \right\rfloor & \text{otherwise} \end{cases} \tag{8}$$

where $\hat{T} = \vec{T} / \|\vec{T}\|$ represents the unit vector in the same direction as $\vec{T}$. Function f_{el} produces volumetric data that can be visualized as an ellipsoid. This function reduces the number of unique values in the volumetric data by identifying voxels that will not influence surface construction because they are definitely inside or definitely outside the ellipsoid. These voxels are assigned H values if they are inside the ellipsoid and L values if they are outside. Voxels near the surface of the ellipsoid are assigned the value of one divided by the ellipsoid body function, scaled by S. By calculating the near surface values in this fashion, the polygon mesh produced from the data will appear to have a relatively smooth surface. Nonetheless the volumetric data will be highly compressible due to the reduced number of unique values.

An ellipsoid primitive constructed with the ellipsoid sampling function f_{el} is centered around the center point $\vec{R}$ of the extension space. Function f_{el} can be used to create a variety of ellipsoids by altering the separate components of R. In cases where an ellipsoid is defined in a volume such that $R_x = R_y = R_z$, the resulting primitive is a sphere.

The visual quality of an ellipsoid primitive is limited by the dimensions of the volume $\vec{V}$ for which it is defined, as well as the range of possible values for each voxel. For smaller values of $\vec{V}$, the primitive is less smooth in appearance. When the range of values for the voxel is insufficient, a distinctive *banding* artifact appears on the constructed polygon mesh. Figure 11 (a) shows the banding artifact on an ellipsoid visualized using a volumetric ellipsoid primitive with $\vec{V} = (32,32,32)$ and a voxel range of 256. This artifact can be reduced in severity by increasing the number of possible values for each voxel. Figure 11 (c) shows the roughness artifact on an ellipsoid with $\vec{V} = (8,8,8)$ and the same voxel range. Figure 11 (b) shows no

distinctive artifacts on an ellipsoid with $\vec{V} = (16,16,16)$, which is large enough to minimize the roughness artifact, but small enough to not require additional voxel range.



(a) $\vec{V} = (32,32,32)$ (b) $\vec{V} = (16,16,16)$ (c) $\vec{V} = (8,8,8)$

Figure 11. Ellipsoid primitives with varying $\vec{V}$

3.2.5. Cylinder

The cylinder sampling function f_{cy} for a volume $\vec{V}$ is defined as follows :

$$f_{cy}(\vec{V}) = \begin{cases} \text{L} & \text{if } \left(EllipseBody\left(\vec{T}+\hat{T}\right) > 1\right) \vee \left(V_y = 0\right) \\ & \quad \vee \left(V_y = B_y - 1\right), \\ \text{S} & \text{if } \left(V_y = 1 \vee V_y = B_y - 2\right) \\ & \quad \wedge \left(EllipseBody\left(\vec{T}+\hat{T}\right) < 1\right), \\ \text{H} & \text{if } EllipseBody\left(\vec{T}+\hat{T}\right) < 1, \\ \left\lfloor \frac{\text{S}}{EllipseBody(\vec{T})} \right\rfloor & \text{otherwise} \end{cases} \tag{9}$$

where $EllipseBody: \vec{P} \to (0,\infty)$ is a function such that $EllipseBody\left(\vec{P}\right) = \left(\frac{P_x^2}{R_x^2} + \frac{P_z^2}{R_z^2}\right)$, which represents the well-known two-dimensional ellipse function. As can be seen from the above definition, the cylinder primitive is related to the ellipsoid primitive. The base of the cylinder is a two-dimensional ellipse, defined on the X/Z plane. Surface detail is calculated in the same fashion as for the ellipsoid primitive, with the ellipsoid body function replaced by a two-dimensional ellipse body function. The ellipse base is used at each level of the cylinder, except for the ends, which must be treated differently in order to cap the primitive. The sampling position height V_y is used as a parameter to determine if the position is vertically within the cylinder, i.e., between 0 and $B_y - 1$. At the vertical ends of the cylinder, S values are placed to form the surface, followed by L values for padding.

3.2.6. Cone

The cone volumetric primitive is used for representing cones and other conical shapes. A conical shape is defined in terms of its height and the radiuses of its elliptical base. The sampling height h is defined as $\left(1 - \frac{V_y}{B_y}\right)$, which linearly interpolates from 0 to 1 based on the position in the extension space, relative to the height of the extension space. More precisely, the value of h linearly interpolates from 1 to 0 as the position height value V_y varies from the bottom to the top of the extension space. Let $ConeBody: \vec{P} \to (0,\infty)$, where $ConeBody\left(\vec{P}\right) = \left(\frac{P_x^2}{(R_x h)^2} + \frac{P_z^2}{(R_z h)^2}\right)$,

represent the ellipse function, modified by the height-dependent h variable. The cone sampling function f_{co} for a volume $\vec{V}$ is defined as follows:

$$f_{co}(\vec{V}) = \begin{cases} \text{L} & \text{if } (V_y = 0) \vee (ConeBody(\vec{T} - \hat{T}) > 1, \\ \text{S} & \text{otherwise if } (V_y = 1) \wedge (ConeBody(\vec{T} + \hat{T}) < 1), \\ \text{H} & \text{otherwise if } ConeBody(\vec{T} + \hat{T}) < 1, \\ \left\lfloor \frac{\text{S}}{ConeBody(\vec{T})} \right\rfloor & \text{otherwise} \end{cases} \quad (10)$$

As with the cylinder, the sampling height h is used to determine if a position is vertically outside of the primitive, but it is also used as a parameter for altering the desired radiuses of the cone. Specifically, h reduces the ellipse radiuses as the positions progress up the cone. L and S padding are applied at the bottom of the cone.

With the above cone sampling function, it is important to consider where in the volume the primitive's center point on the X/Z plane is to be located. If this center point is located in the true middle of volume, the tip of the resulting cone shape may contain artifacts. **Error! Reference source not found.** shows the *broken tip* artifact, where the topmost portion of the primitive contains undesirable surface deformation. This artifact is a result of the logical center being defined between voxels in the extension space. If the center of the cone is forced to be on a voxel, this artifact will not occur.

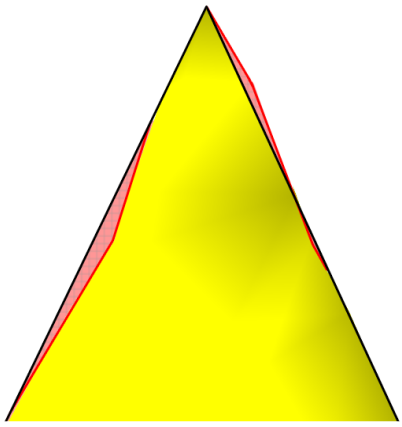


Figure 12. Cone primitive with the broken tip artifact

3.3. The ICVG Algebra

Chen and Tucker's definition of CVG operators was purposely general, and allowed for real and negative values. The modified algebra, called ICVG, concentrates on the use of primitives and operations optimized for indirect visualization. Consequently, the algebra is limited to a restricted range non-negative integer values. The ICVG union, intersection, trim, and cap operations are the same as those of CVG for opacity, except that the input and output values are restricted to discrete positive volumes:

union: $\cup (D_1, D_2) = MAX(D_1, D_2)$ (11)

intersection: $\cap (D_1, D_2) = MIN(D_1, D_2)$ (12)

trim: $trim(D_1, D_2) = TRIM(D_1, D_2)$ (13)

cap: $cap(D_1, D_2) = CAP(D_1, D_2)$ (14)

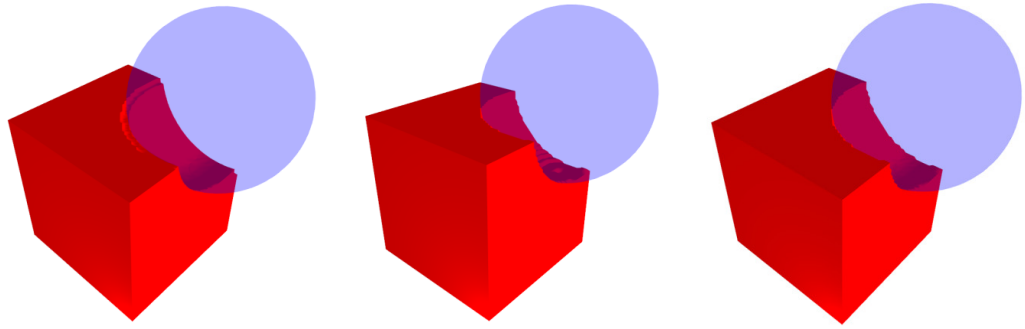
The difference operator is defined differently in ICVG than in CVG to ensure that the values in the resulting volume are always non-negative. The following three possible operators were considered:

$$\textit{differenceA}: \quad -(D_1, D_2) = MAX(D_1 - D_2, \mathrm{L}) \tag{15}$$

$$\textit{differenceB}: \quad -(D_1, D_2) = \begin{cases} MAX(D_1 - D_2, \mathrm{L}) & \text{if } D_2 \geq \mathrm{S}, \\ D_1 & \text{otherwise} \end{cases} \tag{16}$$

$$\textit{difference}: \quad -(D_1, D_2) = \begin{cases} MAX(D_1 - D_2, \mathrm{S}) & \text{if } (D_1 \geq \mathrm{S}) \wedge (D_2 \geq \mathrm{S}), \\ MAX(D_1 - D_2, \mathrm{L}) & \text{otherwise} \end{cases} \tag{17}$$

The *differenceA* operator represents the range-restricted version of the difference operator described by Chen and Tucker [4]. Figure 13 (a) shows that the *differenceA* operator produces undesirable artifacts on the surface of the resulting object when applied with a cuboid for D_1 and an ellipsoid for D_2. Observe that the cavity created in D_1 is smooth, but every edge of this cavity has an apparent beveling artifact. Values in the ellipsoid primitive that are below S are not visualized, but they affect the cuboid primitive when the *differenceA* operator is applied. These problems are typical of those encountered when combining any two primitives with *differenceA*. The *differenceB* operator is defined similarly to *differenceA* but ignores the values that are below S, i.e., values corresponding to parts of the ellipsoid primitive that are outside the surface of the ellipsoid. Figure 13 (b) shows how ignoring these invisible values in the ellipsoid primitive results in a jagged surface on the carved areas of the cuboid. This artifact appears because the ellipsoid primitive owes part of its smoothness to these invisible values. As shown in Figure 13 (c), the *difference* operator combines the desirable features of the *differenceA* and the *differenceB* operators. In cases where the surface of the source object is visible and the surface of the operand object is not, this operator performs a subtraction operation that is bounded between S and H. Otherwise, it acts in the same manner as *differenceA*. The *difference* operator does not suffer from the beveling artifact that occurs with *differenceA*, and it does not produce the jagged interior surface that occurs with *difference B*.



(a) *differenceA* (b) *differenceB* (c) *difference*

Figure 13. Three possible difference operators

Additionally, ICVG defines the *add* and *modulate* operators as follows:

$$\textit{add}: \quad +(D_1, D_2) = MIN(D_1 + D_2, \mathrm{H}) \tag{18}$$

$$\textit{module}: \quad *(D_1, D_2) = MIN(D_1 * D_2, \mathrm{H}) \tag{19}$$

These two operators may produce compound objects with surfaces not found in the source primitives because when two densities that are lower than the surface threshold are combined the result may be a density that is higher than the surface threshold. Thus these two operations have less predictable behavior than the previously defined operations.

Two possible divide operators, called *divideA* and *divide*, were also considered. The naive *divideA* operator is defined as follows:

$$\textit{divideA}: \qquad \div(D_1, D_2) = \begin{cases} D_1 \div D_2 & \text{if } D_2 \geq 0, \\ \text{H} & \text{otherwise} \end{cases} \tag{20}$$

and the more useful *divide* operator is defined as follows:

$$\textit{divide}: \qquad \div(D_1, D_2, S_L) = \begin{cases} D_1 \div (D_2 * S_L) & \text{if } D_2 \geq 0, \\ \text{H} & \text{otherwise} \end{cases} \tag{21}$$

where S_L is a real number in the range 0 to 1.

Since an object's surface is the middle of the valid range of possible voxel values, the *divideA* operator produces a compound object with no visible surface wherever $D_2 \geq 2$. The *divide* operator is similar to *divideA*, but it multiplies the divisor by a real-valued scalar during each operation, providing additional control. The *divide* operator does not require D_2 to be specified as a value in the real domain in order to provide small adjustments. The *divide* operator can be implemented using floating point operations during calculations, allowing for a greater range of results, but only integers are used as input and output. Thus, only integers need to be stored.

In addition to the binary operators just discussed, ICVG also includes scalar and unary operations. A scalar operation is performed by applying the same arithmetic operation to every voxel in volumetric data. An example of a scalar operation is *addScalar*, which adds a single number to a data set in order to uniformly increase, or decrease, the densities. A unary operator, such as *inverse*, requires a single operand. The inverse operation creates an extension space where solidness and emptiness are reversed. Taking the inverse of an object D_1 is equivalent to applying the *difference* operator to a completely solid extension space and D_1. The *inverse* operator can be implemented by computing the difference between H and each voxel. The *inverse* operator is defined as follows:

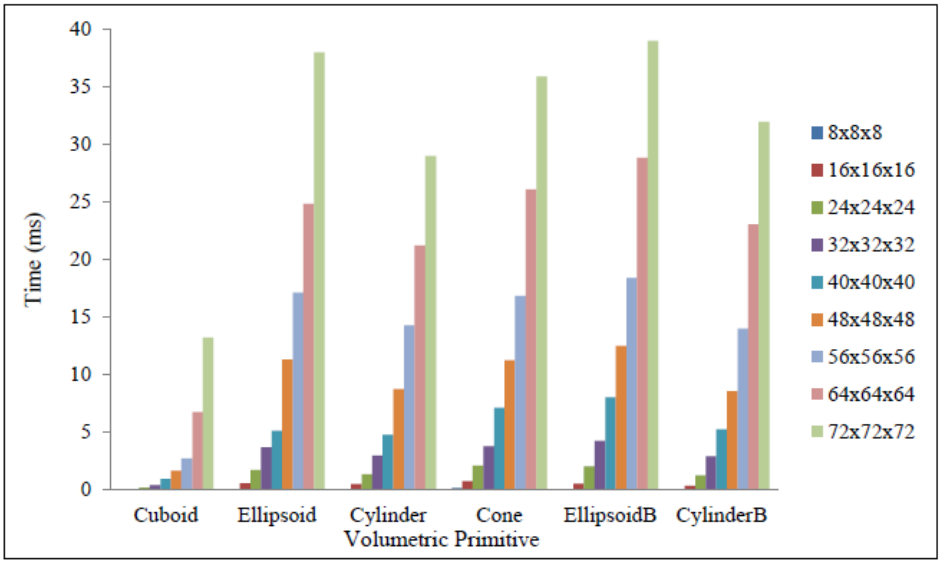
$$\textit{inverse}: \qquad I(D_1) = (\text{H} - D_1) \tag{22}$$

4. EXPERIMENTAL RESULTS

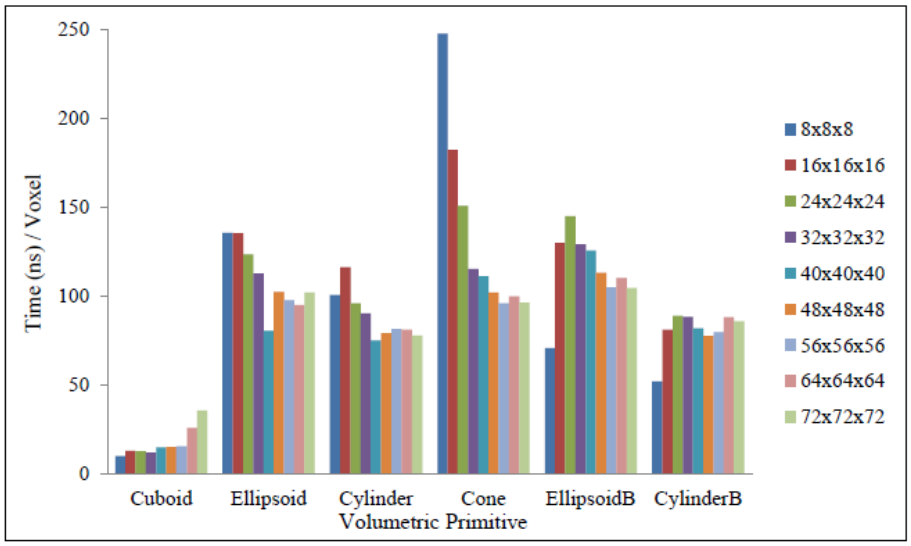
All sampling functions described in Section 3.2 were implemented in both CPU and GPU versions in a common test bed, with the only differences being those that are essential to invoke CPU versus GPU operations. Experiments were run on a Dell Alienware Aurora computer with an Intel Core i7-3820 CPU and an NVIDIA GeForce GTX 560 GPU. Every result reported is the average (mean) of the results for 100 separate runs, each of which involved generating 100 identical volumetric solids.

The four graphs shown in Figure 14 and Figure 15 depict the performance of volume generation for four primitives (cuboid, ellipsoid, cylinder, and cone) using the sampling functions defined in ICVG. Figure 14 (a) and Figure 15 (a) show the execution times in milliseconds on the CPU and GPU, respectively. For example, the first set of eight lines in Figure 14 (a) shows the execution time on the CPU while generating cuboids in various sizes of extension spaces. The "EllipsoidB" and "CylinderB" results are for alternative implementations that are not restricted to aligning precisely with the grid. Figure 14 (b) and Figure 15 (b) represent the same timing data, but with the time (in nanoseconds) per voxel in the extension space. For example, for generating a cuboid on the CPU in an 8 x 8 x 8 extension space, the time of 0.005 milliseconds is divided by 512 (8^3) voxels, giving 9.7 nanoseconds per voxel.

The results in Figure 14 (a) show that, for each primitive, the execution time on the CPU increases as the primitive is generated in larger extension spaces. Figure 14 (b) shows that the execution time per voxel was roughly constant, except for the cone. When generating a cone, a fixed overhead occurred, which increased the cost per voxel for small sizes of extension spaces.



(a) Time per primitive

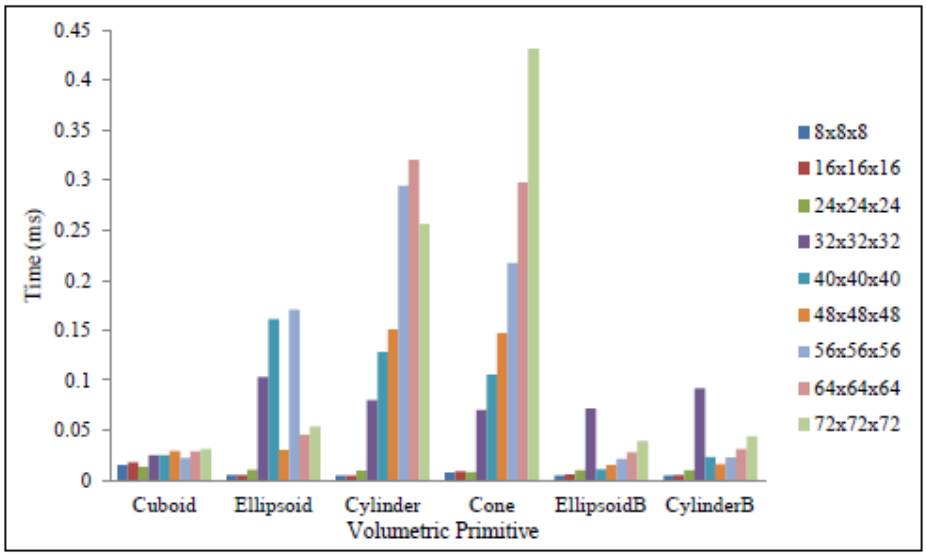


(b) Time per voxel

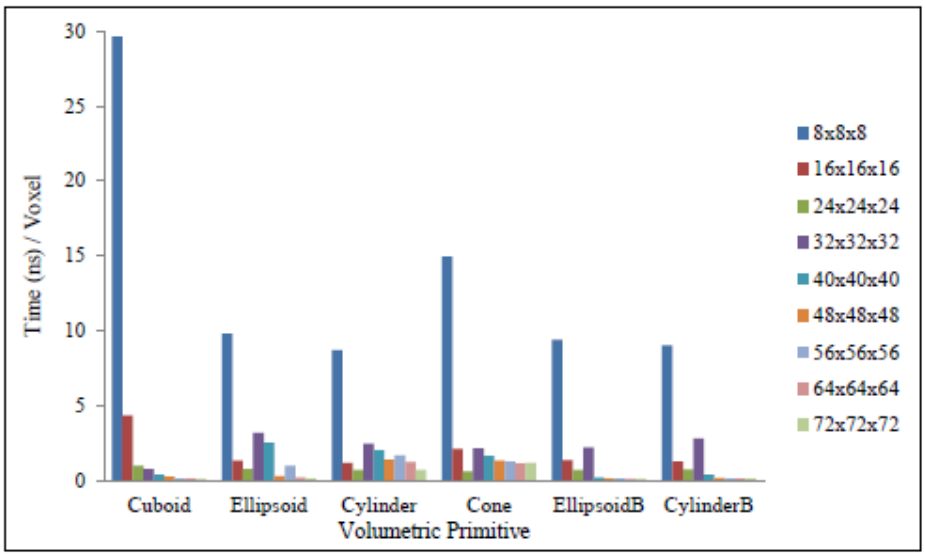
Figure 14. CPU execution times for generating primitives with varying sizes of extension spaces

Timing results for generating the primitives on the GPU were also obtained. The version of DirectX that was used for our implementation did not provide the ability to query the time that it took for a dispatch call (which in this case causes a single primitive to be generated on the GPU). To counter this absence, the results of the dispatch calls were immediately pulled back into CPU memory after the dispatch call was made and the timer was stopped when the transfer completed. If a request is made to bring GPU data back into CPU memory, then all queued operations (such as the dispatched generation procedures) on that data are performed as soon as possible and the data afterwards is transferred back to the CPU. To obtain timing information for only the GPU

calculations, the fixed time cost of transferring the data from the GPU was subtracted, which otherwise would have distorted the results for small extension spaces. However, this introduced some inaccuracy for these small extension spaces. Thus, overall the timing mechanism for our experiments on the GPU was less accurate than the one for those on the CPU.



(a) Time per primitive



(b) Time per voxel

Figure 15. GPU execution times for generating primitives with varying sizes of extension spaces

The results in Figure 15 (a) show that execution was much faster on the GPU than the CPU, e.g., 0.43 ms instead of 36 ms for the largest cone. For the cuboid, the execution time on the GPU was roughly constant as the cuboid is generated in larger extension spaces. For the other primitives, small linear increases in total execution time on the GPU were observed as the size of the extension space increased, possibly because of timing inaccuracies, as described above. Given the timing inaccuracies, the execution time per voxel was roughly constant for all primitives (see Figure 15(b)).

Overall, acceptable performance was obtained with our implementations of ICVG on both the CPU and GPU. The time required to generate any 72 x 72 x 72 shape was less than 40 ms on the CPU and less than 0.45 ms on the GPU.

5. CONCLUSIONS AND FUTURE WORK

The ICVG algebra was described and descriptions of primitives for volumetric data presented through indirect visualization were provided. The concept of a volume sampling function was introduced, and examples were provided for cuboids, ellipsoids, cylinders, and cones. Potential banding, beveling, and clipping artifacts were identified and methods for avoiding them were provided. Three alternative definitions for the difference operator were described, and the visual results for indirect visualization were compared. As well, the *divideA*, *divide*, *modulate* and *add* CVG operations were defined. The *inverse* unary operation was also specified.

In future work, experiments could be conducted on alternative methods for reducing the remaining artifacts due to the difference operation. Alternative values of S should be investigated to determine whether the visual impact of the artifact can be reduced. Modified versions of the marching cubes algorithms could be investigated to see if their changes to the resulting meshes affect the presence of the identified artifacts. Additional primitive objects could be defined, including torus, pyramid, capsule, and prism objects [23].

REFERENCES

[1] L. Kobbelt, M. Botsch, U. Schwanecke, and H. Seidel, "Feature Sensitive Surface Extraction from Volume Data," in *28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'01)*, Los Angeles, 2001.

[2] R. Geiss, "Generating Complex Procedural Terrains Using the GPU," in *GPU Gems*, Boston, Pearson Education, 2008, pp. 5-37.

[3] A. Requicha and H. Voelcker, "Constructive Solid Geometry," Production Automation Project, University of Rochester, Rochester, 1977.

[4] M. Chen and J. Tucker, "Constructive Volume Geometry," *Computer Graphics Forum,* vol. 19, no. 4, pp. 281-293, 2000.

[5] M. Schifko, B. Jüttler, and B. Kornberger, "Industrial Application of Exact Boolean Operations for Meshes," in *26th Spring Conference on Computer Graphics (SCCG'10)*, Budmerice, Slovakia, 2010.

[6] *Hammer Editor,* Valve [computer software], 2017.

[7] N. Zhang, H. Qu, and A. Kaufman, "CSG Operations on Point Models with Implicit Connectivity," in *Computer Graphics International Conference (CGI'05)*, Stony Brook, NY, 2005.

[8] P. Lacroute and M. Levoy, "Fast Volume Rendering Using a Shear-warp Factorization of the Viewing Transformation," in *21st Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'94)*, Orlando, 1994.

[9] B. Kainz, M. Grabner, A. Bornik, S. Hauswiesner, J. Muehl, and D. Schmalstieg, "Ray Casting of Multiple Volumetric Datasets with Polyhedral Boundaries on Manycore GPUs," *ACM Transactions on Graphics,* vol. 28, no. 5, pp. 1-9, 2009.

[10] D. Laur and P. Hanrahan, "Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering," in *18th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'91)*, Las Vegas, 1991.

[11] W. Lorensen and H. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," in *14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'87)*, Anaheim, 1987.

[12] T. Newman and H. Yi, "A Survey of the Marching Cubes Algorithm," *Computer & Graphics,* vol. 30, no. 5, pp. 854-879, 2006.

[13] D. Williams, "Volumetric Representation of Virtual Environments," in *Game Engine Gems*, Sudbury, Jones and Bartlett, 2010, pp. 39-60.

[14] M. Chen and J. Tucker, "Constructive Volume Geometry," Department of Computer Science, University of Wales, Swansea, 1998.

[15] K. Johnson and J. V. Tucker, "The Data Type of Spatial Objects," *Formal Aspects of Computing,* vol. 25, no. 2, pp. 189-218, 2013.

[16] K. H. A. Johnson, *The Data Type of Spatial Objects with Applications to Constructive Volume Geometry,* Doctoral thesis, University of York, UK, 2007.

[17] A. Kaufman, D. Cohen, and R. Yagel, "Volume Graphics," *IEEE Computer,* vol. 26, no. 7, pp. 51-64, 1993.

[18] S. Laine, "A Topological Approach to Voxelization," *Computer Graphics Forum,* vol. 32, no. 4, pp. 77-86, 2013.

[19] P. Nourian Ghadi Kolaee and S. Zlatanova, "Voxels and Voxel Algorithms; Two Methods of Rasterization, Raster Specific Operations," 31 12 2015. [Online]. Available: https://repository.tudelft.nl/islandora/object/uuid:7954d601-5e37-47bc-b73e-1df435efea6d?collection=research. [Accessed 2 10 2017].

[20] M. Schwarz and H.-P. Seidel, "Fast Parallel Surface and Solid Voxelization on GPUs," in *3rd ACM SIGGRAPH Conference and Exhibition on Computer Graphics and Interactive Techniques in Asia (SIGGRAPH Asia 2010)*, Seoul, South Korea, 2010.

[21] J. Baert, A. Lagae, and P. Dutré, "Out-of-Core Construction of Sparse Voxel Octrees," *Computer Graphics Forum,* vol. 33, no. 6, pp. 220-227, 2014.

[22] *3D Coat,* Pilgway [computer software], 2011.

[23] *3D Studio Max,* Autodesk / Discreet [computer software], 2017.

APPENDIX

The following functions are defined on scalars in the range of [L, H]:

$$MAX(X_1, X_2) = \begin{cases} X_1 \text{ if } X_1 > X_2, \\ X_2 \text{ otherwise} \end{cases} \quad \text{(A1)}$$

$$MIN(X_1, X_2) = \begin{cases} X_1 \text{ if } X_1 < X_2, \\ X_2 \text{ otherwise} \end{cases} \quad \text{(A2)}$$

$$TRIM(X_1, X_2) = \begin{cases} X_1 \text{ if } X_1 \geq X_2, \\ X_2 \text{ otherwise} \end{cases} \quad \text{(A3)}$$

$$CAP(X_1, X_2) = \begin{cases} X_1 \text{ if } X_1 \leq X_2, \\ X_2 \text{ otherwise} \end{cases} \quad \text{(A4)}$$