Minkowski Operators for Voxel Based Sculpting

Saurabh U. Sethia*
S. Manohar

Department of Computer Science & Automation,
Indian Institute of Science, Bangalore, India - 560 012.

23 April 1997

1 Introduction

A sculpting package can be seen as a 3D analog of a drawing package. In analogy to a drawing package it provides facilities to create and edit 3D objects. A sculpting/drawing package provides the following two things to a user. A set of primitive shapes and a set of operators that operate on existing shapes to generate new shapes.

Real world sculpting provides a good guiding line for defining operators. Hence we often consider one of the operands to be clay on which we are working and the other to be a tool which is rigid. A natural tool for sculpting is a material remover (Cutting tool) which removes material from wherever it moves. Complementery to this is a material adder (Pasting tool) which leaves material.

*Author for correspondence. Current address: Department of Computer Science, State University of New York at Stony Brook, Stony Brook, New York 11794, USA. Email: saurabh@ams.sunysb.edu
Another useful operator to provide is Smoothing. This is like low pass filtering. It allows the user to smoothen sharp features of her sculpture.

In the next section we make the observation that these tools can be implemented in terms of Minkowski operators. Minkowski operators can be implemented using various data structures. Implementation using Boundary representation is explored in [1]. But it restricts the topology of the two shapes to simple polyhedrons. Other more sophisticated boundary representations like parametric curves also suffer from the same drawback. Although many nice operations can be performed on parametrically represented objects, changing topology is difficult. But in a sculpting package we cannot afford to put any restrictions on topology. Hence we need a data structure which deals with arbitrary topology with uniform ease. Any voxel-based data structure serves this purpose. Voxel based data structures are inherently topology insensitive. Objects represented in the form of voxels are dealt with uniform ease irrespective of whether they are simple or have a high genus. Due to its simplicity 3D voxel arrays seem to be the most promising. Galyean and Hughes discuss a sculpting package in [2] that works with a voxel grid. It provides useful tools like material remover/adder, sandpaper and heat gun. It also has the ability to switch between two resolutions. However it operates only in very small grids, due to space and speed limitations.

As we see in section 3, implementation of Minkowski operators is trivial if we use 3D voxel arrays to represent our clay. The only drawback being
that the size of a voxel array grows cubically with resolution. If we need very high resolution and have insufficient main memory, we can use an octree whose space requirement growth is quadratic w.r.t. resolution. In section 4 we give algorithms to implement Minkowski operators on octrees that are asymptotically as good as or better than those for voxel arrays. Though constants are of course higher for octrees.

2 Minkowski Operators

2.1 Minkowski Addition

Minkowski addition of two sets $A$ and $B$ in $\mathbb{R}^d$ is defined as the union of sets obtained by positioning one of them, say $B$, at every point of the other, say $A$. i.e. the set of points obtained by vectorially adding each point in $A$ with each point in $B$.

Mathematically, if $A_p$ denotes translate of a set $A$ by the vector $p$, i.e., $A_p = A \oplus \{p\}$, then,

$$A \oplus B = B \oplus A = \bigcup_{a \in A} B_a = \bigcup_{b \in B} A_b$$

(1)

which is same as,

$$A \oplus B = \{ a + b : a \in A, b \in B \}$$

(2)

where $\oplus$ stands for Minkowski addition.
2.2 Minkowski Decomposition

If $A$ and $B$ are two sets in $\mathbb{R}^d$ space, then

$$A \odot B = \bigcap_{b \in B} A_{-b} = \bigcap_{b \in B} A_b$$

(3)

where $\odot$ stands for Minkowski decomposition operation. We call $A \odot B$ as the decomposition of $A$ by $B$. (As per our previous notation, "$A_{-b}$" means the translate of $A$ by the vector $-b$, i.e., $A \odot -b$.

The set $\hat{B} = -b : b \in B$ is generally known as the symmetrical set of $B$ with respect to the origin.

**Lemma 1** The Minkowski decomposition of set $A$ by set $B$ is equal to the complement of the Minkowski addition of the complement of $A$ with the symmetrical set of $B$. i.e.

$$A \odot B = \overline{A \oplus \hat{B}}$$

(4)

**Proof:** From definition,

$$A \odot B = \bigcap_{b \in B} A_b$$

$$\Rightarrow \overline{A \odot B} = \bigcap_{b \in B} \overline{A_b}$$

$$= \bigcup_{b \in B} \overline{A_b}$$

$$= \overline{\bigcup_{b \in B} A_b}$$

$$= \overline{A \oplus \hat{B}}$$

$$\Rightarrow A \odot B = \overline{A \oplus \hat{B}}$$

Q.E.D.
With this result we can now concentrate on the Minkowski addition only.

### 2.3 Minkowski operators for Sculpting tools

Cutting and pasting tools can be implemented using Minkowski sum of the tool with its trajectory. The sum gives the volume swept by the tool. Finding the union (difference) of this volume with (from) the clay gives us the pasting (cutting) tool. We approximate the trajectory by a polyline. Hence the task reduces to finding the Minkowski sum of the tool with a straight line segment.

Smoothing can be implemented using the standard morphological operations known as opening and closing. These in turn can be expressed in terms of Minkowski operators as follows.

\[
\text{opening} = (X \ominus B) \oplus B
\]  \hspace{1cm} (5)

\[
\text{closing} = (X \oplus B) \ominus B
\]  \hspace{1cm} (6)

where \(X\) is the object and \(B\) is a small sphere at origin.

To implement smoothing we need to apply both opening and closing to the object. The radius of the sphere controls the level of smoothing done.
3 Sculpting Operators on Uniform Voxel Grids

3.1 Definition

A Uniform Voxel Grid representation has a finite universe which is usually cubical in shape. The universe is uniformly subdivided into cubical volume elements called voxels, which are arranged in the form of a $N \times N \times N$ uniform grid aligned with the coordinate axes. This is exactly analogous to how images are represented by an array of pixels. Voxels are the basic building blocks in this representation and in case of binary objects each voxel is either filled or unfilled which is represented by 1 and 0 respectively. Filled voxels are often called Black and the unfilled ones as White. Antialiased rendering needs Gray voxels as well. In which case they are represented as numbers between 0 and 1. The voxel grid can represent objects in its universe only. Information is lost if an object moves out of its universe.

3.2 Minkowski addition of two objects

The most trivial algorithm is to find Minkowski addition of each filled voxel in one object with each filled voxel in the other and take the union. The Minkowski sum of two voxels is a cube of twice the side-length of a voxel, located at a point which is the vector sum of the locations of the two voxels. This cube will occupy exactly eight voxels in the result grid. The number of voxels in an object is $O(N^3)$ hence this algorithm takes $O(N^6)$ time. However note that the constant
is very small. We only need to find vector addition of two points which involves three additions and set the value of eight voxels.

We can improve the asymptotic response with a little increase in the constants if we use the following lemma.

**Lemma 2** We can find the Minkowski sum of two objects $A$ and $B$ by putting one object on the boundary voxels of the other and vice versa and take the union.

**Proof**: Let for any $a \in A$ and $b \in B$, $s = a + b$. If at least one of $a$ and $b$ is a boundary voxel, $s$ is covered by the given method. Else consider, the right neighbour of $a$ and left neighbour of $b$. The vector sum of the two is $s$. Hence if at least one of them is a boundary voxel, $s$ is covered by the given method. If not we repeat the process till we reach a boundary voxel.

Q.E.D.

Now since the boundaries of objects grow as $O(N^3)$ we have an $O(N^5)$ algorithm. The only overhead is to check whether a voxel is a boundary voxel or not.

The above algorithms do not assume any thing about the input objects. However, for sculpting one of the objects is usually a tool which is very small when compared to the object being sculpted. In such cases we may like to represent the tool as a list of filled voxels. This will lead to significant reduction in the constant as we don't need to look at the empty voxels.
3.3 Minkowski sum of an object with a line segment

This can be done using a 3D line drawing algorithm analogous to Brashenham's 2D line drawing algorithm[3]. For each filled voxel in the given object which represents the boundary of the object simply draw the given line segment starting from that voxel. For drawing a 3D line consider w.l.o.g. that the line belongs to the first octant and that $z$ is the fastest increasing coordinate as we move along the line. Now project the line on $z$-$y$ and $z$-$z$ planes. Draw the two projections. The values of $y$ and $z$ for each $z$ gives the voxel to be filled for drawing the 3D line. The algorithm first needs to find the boundary voxels. This is $O(N^3)$ time. Then for each boundary voxel it needs to draw the 3D line. The number of boundary voxels is $O(N^2)$ and the line drawing is $O(N)$. Hence this step also needs $O(N^3)$ time. Thus the total running time is $O(N^3)$.

We can reduce the constant by a factor of two by only considering those voxels that have boundary faces in the direction of the directed line segment.

4 Sculpting Operators on Octrees

4.1 Definition

The octree is a compact hierarchical data structure that represents objects lying in its universe as a union of disjoint cubes of fixed location and sizes that are powers of two. Objects lying outside the universe cannot be represented. Information is lost if an object moves or extends out of the universe as a result
of some operation.

Consider a binary universe consisting of an array of $N \times N \times N$ cubic voxels, where $N = 2^n$. An octree is a tree whose each internal node has exactly eight children. Each node in the tree corresponds to a cubic space in the universe. The root node corresponds to the entire universe covered by the array of voxels. If the cubic space represented by a node is homogeneous, we ascribe its colour (Black for filled, White for empty) to the node. Otherwise, we subdivide the cubic space into eight octants. The node is coloured Gray indicating partial filling and the eight octants, which in turn are represented by octrees, are made its children. Nodes that do not have any children are called leaf nodes. While others are called internal nodes.

We define the level of a node as the logarithm of the side-length of the cube it represents. Thus the level of root node is $n$ and that of a node representing a single voxel is zero.

### 4.2 Minkowski Addition of two octrees

The algorithm is based on the following properties of Minkowski addition. We present this trivial algorithm in some detail to simplify the presentation of algorithm 2.

**Property 1** \( (A \cup B) \oplus C = (A \oplus C) \cup (B \oplus C) \)

**Proof:** Immediately follows from the definition.
The following property immediately follows from property 1.

**Property 2** If \( A = \bigcup_{i=1}^{n} A_i \) and \( B = \bigcup_{i=1}^{m} B_i \), then \( A \oplus B = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} (A_i \oplus B_j) \)

This property implies that to find the Minkowski sum of two octrees we can find the Minkowski sum of each black leaf in one with each leaf in the other and take the union.

**Algorithm 1** Minkowski sum of two octrees \( A \) and \( B \):

**Step 1:** Find Minkowski sum of each black leaf of \( A \) with each black leaf of \( B \).

This will give us a set of non-disjoint black cubes.

**Step 2:** Find the octree corresponding to the union of cubes found in step 1.

**To implement step 1:** Minkowski sum of two black leaves \( X \) and \( Y \) is a cube whose side-length is equal to the sum of the side-lengths of \( X \) and \( Y \) and whose location is the vector sum of the locations of \( X \) and \( Y \). The location of a cube can be considered as its center or one of its corner. It doesn’t matter how we define it as long as we are consistent.

**To implement step 2:** Starting from an empty octree keep inserting the cubes as they are found in step 1, into the result octree.

We thus find the union of cubes found in step 1 incrementally starting from a White octree. Inserting is equivalent to finding the union of a cube with the octree of the union of cubes found so far.

**Inserting:** Finding the union of a cube \( C \) and an octree is equivalent to finding
the leaves of the octree occupied by the cube. To do this we start from root and compare the cube with the cubes corresponding to the nodes of the octree. If $C$ completely encloses the node, i.e. if the node lies inside $C$ then we colour the node Black. If $C$ doesn’t intersect the node that is the node lies outside $C$ then we do nothing. Else there is an ambiguity because $C$ intersects the node partially. So we repeat the process with all the children of the node. In case the node is a leaf we split it into eight children of the same colour. Since we are finding union, we need not do any thing for Black leaves.

IF the root is a Black leaf THEN
    Return.

Compare the cube, say $C$, to insert with the root of the octree, say $R$.

The following cases may arise:--

1. IF $R$ lies inside $C$ THEN
    Make $R$ Black.

2. ELSE IF $R$ lies outside $C$ THEN
    Do nothing.

3. ELSE \ partial intersection
   3.1 Split $R$ if it is White
   3.2 insert $C$ in the children of $R$.
   3.3 Check if all children of $R$ are Black in which case remove them and make the parent Black.
**Compare:** While comparing the given node $R$ with the given cube $C$ we look for three cases. (1) $R$ lies inside $C$, (2) $R$ lies outside $C$ (3) $R$ partially intersects $C$.

**Inside:** If $R$ is bigger than $C$, it cannot lie inside. Else we check if the top face of $R$ is above the top face of $C$. In which case $R$ cannot be inside $C$. Similarly we do checks for all six faces. If all six checks pass then $R$ is inside $C$.

**IF $R$ is bigger than $C$, THEN**

Return FALSE.

FOR EACH coordinate

**IF difference between constant**

coordinates of positive direction

faces of $C$ and $R$ is negative THEN

Return FALSE.

IF difference between constant

coordinates of negative direction

faces of $C$ and $R$ is positive THEN

Return FALSE.

Return TRUE.

**Outside:** We check if the top face of $R$ is above the bottom face of $C$. In which case $R$ has to be outside $C$. Similarly we do checks for all six faces. If any of the six checks pass then $R$ is outside $C$.

FOR EACH coordinate

**IF difference between positive direc-**
tion face of $R$ and negative direction

face of $C$ is non-positive THEN

Return TRUE.

IF difference between negative direc-
tion face of $R$ and positive direction

face of $C$ is non-negative THEN

Return TRUE.

Return FALSE.

4.3 Minkowski addition of an octree with a line segment

We assume that one of the ends of the line segment is at the origin. Minkowski addition with a general line segment will just need a translation of the result octree. To find the sum, we again make use of Property 1. We find the Minkowski sum of each black leaf with the line segment and take the union. Weng and Ahuja [4] use similar technique for translating and rotating octrees.

Algorithm 2 Minkowski sum of an octree $A$ and a line segment $L$:

Step 1: Find the Minkowski sum of each black leaf of $A$ with $L$.

Step 2: Find the octree corresponding to the union of the result from step 1.

To implement step 1:
1. Find the directions of the coordinate axes along which \( L \) is moving away from origin. For example if \( L \) lies in the first octant the directions will be \(+z, +y, +z\). Corresponding to each direction there is a face of the leaf.

2. Sweep each face considered in step 1 along \( L \) to get a parallelopiped. The union of all of these and the leaf itself gives the space swept by the leaf when it is moved along \( L \). This is nothing but the Minkowski sum of the leaf with \( L \).

**To implement step 2:** Starting from the given octree keep inserting parallelopipeds as they are found in step 1, into the result octree.

**Inserting:** Same as in Algorithm 1 except that it inserts parallelopipeds instead of cubes and uses the following compare.

**Compare:** While comparing the given node \( R \) with the given parallelopiped \( P \) we look for three cases. (i) \( R \) lies inside \( P \), (ii) \( R \) lies outside \( P \) and (iii) \( R \) partially intersects \( P \).

Now w.l.o.g. consider that \( P \) is generated by sweeping the \(+\)ve x-direction face, \( F_{+x} \), of a cube along \( L \). Let \( P_{xy} \) be the projection of \( P \) on the xy-plane. \( P_{xy} \) will be a parallelogram obtained by sweeping the projection of \( F_{+x} \) along the projection of \( L \) on the xy-plane. Similarly we can have \( P_{xz} \). Now if we sweep \( P_{xy} \) along the z-axis we get an infinite cylinder, \( C_{xy} \), with \( P_{xy} \) as its cross-section. Similarly we have \( C_{xz} \). A little reflection would show that \( P = C_{xy} \cap C_{xz} \).
Hence if $R$ is outside either of $C_{xy}$ or $C_{xz}$ then it is outside $P$. And if $R$ is inside both $C_{xy}$ and $C_{xz}$ then it is inside $P$. To compare $R$ with $C_{xy}$ or $C_{xz}$ is equivalent to comparing their projections. Hence we have only two 2D problems to solve.

**2D Compare:** Let $p$ be the parallelogram and $l$ be the 2D line segment. Let $l_x, l_y$ be the components of $l$. We assume w.l.o.g. that $p$ has been swept from a line segment parallel to $y$-axis. We further assume that $l_x$ is $+$ve. The case when $l_x$ is negative is analogous.

**STEP A:** Find the intersection of the projections of $p$ and $R$ on $z$-axis. Let this segment be $AB$ (with $A$ having the lower $x$-coordinate). The intersection of $p$ and $R$ will lie between $A_x$ and $B_x$. Find the parts of $p$ and $R$ that lie between these $x$-coordinates. Let them be $p'$ and $R'$ respectively. In general, $R'$ will now be a rectangle.

**STEP B:** Now consider the boundary segments of $p'$ and $R'$ along line $z = A_x$. Let the line segments be $R_s$ and $p_s$. These two line segments can have three different relative positions. (a) $R_s$ lies above $p_s$ (b) $R_s$ lies below $p_s$ (c) They overlap.

Find which one is the case. Repeat the same for the boundary segments along line $z = B_x$.

**For Outside:** $R$ lies outside $p$ iff EITHER (a) is true at both $z = A_x$ and $z = B_x$ OR (b) is true at both $z = A_x$ and $z = B_x$. 

15
For Inside: $R$ lies inside $p$ iff $R_x$ is a subset of $p_x$ at both $x = A_x$ and $x = B_x$.

The end-points of $p_x$ can be calculated using the slope of $L$. But that would involve a floating point operation. To avoid that we rasterize $L$ before starting the algorithm and lookup for the $y$-coordinate corresponding to the $A_x$ and $B_x$.

4.4 Analysis

Both algorithm involve insertion of cubes or parallelopipeds into an octree. To find the average case complexities of these algorithms, we need to find the average case complexity of the insertion step. We use the following theorem known as the quadtree complexity theorem[5][6] to do this.

*The number of nodes in a quadtree of an object is proportional to the sum of the maximum resolution required to describe the object and the perimeter of the object.*

The 3D generalization of this theorem is:

*The number of nodes in an octree of an object is proportional to the sum of the maximum resolution required to describe the object and the surface area of the object.*

This however, is a worst case result and in all but the most pathological cases the perimeter of the object is much higher than the resolution. Hence for
all practical purposes we can assume that the size of an octree is linear in the surface area of the object it represents. The average complexities increase by a factor of $\log N$ if we do not make this assumption.

In calculating the average case complexities, we make the assumption that the probability of a node belonging to level $l$ of the source octree is proportional to the maximum possible number of nodes at level $l$.

The maximum possible number of nodes at level $l$ is equal to $8^{n-l}$.

The maximum possible number of nodes, $T$, in the octree can be easily found to be equal to $(8^{n+1} - 1)/7$.

Hence the probability of a node belonging to level $l$, say $p(l)$, is given by,

$$p(l) = \frac{8^{n-l}}{T}$$

Let $b, b_1, b_2$ etc. denote the number of black leaves in source octrees.

4.4.1 Analysis of Algorithm 1

In this algorithm we insert cubes generated by finding minkowski sum of each leaf in one octree with each leaf in the other. Let $l_1, l_2$ be the level of the leaves of the two octree. The surface area of the resulting cube will be proportional to $(2^{l_1} + 2^{l_2})^2$. The probability of this combination is the product of individual probabilities. Hence the expected total number of nodes generated by the algorithm is proportional to,
\[ \sum_{l_2=0}^{n} \sum_{l_1=0}^{n} (2^{l_1} + 2^{l_2})^2 p(l_1) p(l_2) \]  

(7)

This comes out to be \( O(1) \). Hence the complexity of the algorithm is \( O(b_1 b_2) = O(N^4) \).

### 4.4.2 Analysis of Algorithm 2

In this algorithm we insert parallelopipeds generated by sweeping the cube faces of the black leaves in the source octree in the direction of the given directed line segment. The surface area of the parallelopipeds are obviously proportional to \((2^l)^2 + 2 \cdot 2^l \cdot L\), where \( l \) is the level of the leaf and \( L \) is the length of the given line segment. Hence the expected total number of nodes generated by the algorithm is proportional to,

\[ \sum_{l=0}^{n} ((2^l)^2 + 2 \cdot 2^l \cdot L) p(l) \]  

(8)

This comes out to be \( O(L) \). Hence the complexity of the algorithm is \( O(b_1 L) = O(N^3) \), since \( L = O(N) \).

### 5 Summary and results

In this paper we highlighted the importance of Minkowski operators for Interactive Sculpting and gave algorithms for their implementation using voxel based data structures. If our world is a cube of size \( N \times N \times N \) then we can find
Minkowski sum of two objects in $O(N^5)$ time using voxel arrays and $O(N^4)$ time using octrees. We can find the Minkowski sum of an object with a line segment in $O(N^3)$ time for both voxel arrays and octrees. The algorithms described in this paper form the core of ongoing projects in voxel-based modeling, and machining simulations. A software system for interactive virtual machining, that has voxel-based sculpting at its core, is currently in the advanced stages of completion and will be described in a forthcoming paper. Figure 1 shows two voxel models created using this system and demonstrates the potential of the algorithms presented in this paper.

Acknowledgements

We thank Prof. Vijay Chandru for pointing out Minkowski operations as possible sculpting tools and N. Mahesh for creating the models in Figure 1 using his interactive virtual machining system.

References


