3D Region Representation Based on Run-lengths: Operations and Efficiency

Xinquan Shen* and Michael Spann
School of Electronic and Electrical Engineering
The University of Birmingham, Birmingham B15 2TT, UK

Abstract

This paper discusses a volumetric representation of 3D regions achieved by the generalisation of run-length coding for 2D binary images. The data structure for describing the space occupation of the region is investigated with a performance analysis of the memory expense. A set of basic region operations are implemented, followed by a quantitative analysis on the computational cost of these operations. With these basic operations, the representation scheme is useful to applications such as computer vision, computer graphics and robotics.

Computer Vision **Region/Object Representation**
Volumetrical Representation **Region Operations**
Run-length Representation

1 Introduction

3D regions are important entities in applications of computer vision and image processing. A 3D region can be an abstraction of an object to be identified in high

*Author to whom correspondence should be addressed. E-mail: shenx@eee.bham.ac.uk; Fax: 44 121 4144291.

level processing[1], or an entity with certain attributes (e.g. homogeneous in grey level) in low level processing[2]. Operations applied to 3D regions are various, and, roughly, can be categorised into 3 classes: algebraic operations (e.g. computing union/intersection of two regions, checking whether a voxel is inside the region, etc[3]); geometric operations (e.g. computing surface properties such as the normal and curvature, principal axes, etc[4]); and topological operations (e.g. computing the Euler number). Algebraic operations are typically required by applications of low level processing such as segmentation[5, 6], while geometrical and topological operations are normally involved in applications of high level processing such as object recognition[7, 8]. A representation scheme associated with a 3D region thus consists of two parts: (1) a data structure describing the region; and (2) a set of region operations based on that data structure.

Techniques for describing the region are often devised with regard to the operations applied to the region. 3D regions often have a large volumetric size. Simply using a set of voxels to describe the physical entity of a 3D region is expensive in memory requirements and thus not efficient for applications. A concise data structure which describes the region using less memory is obviously important. However, an efficient data structure does not necessarily allow for fast implementation of some particular region operation. Also, a data structure suitable for one class of region operations may not suitable for another class of region operations.

A 3D region can be characterised by either its appearance or its occupation in space, leading to two categories of methods for describing the region: surface representations and volumetric representations[9, 10]. A surface representation characterises a 3D region by describing its surface structure. Techniques for describing surfaces are normally cheap in memory requirements (e.g. a B-spline surface[11, 12]) and enable efficient implementation of many geometrical and topological region operations. Computations such as surface normal and curvature are often straightforward. Internal parts of the region are not explicitly expressed. Thus, algebraic operations such as the union and intersection of regions are not trivial. Successively accessing each voxel in the region is difficult, due to the complexity involved in checking the belongingness of a voxel to the region. Surface representation schemes are generally used in visualisation and high level applications of computer vision such as object modelling, tracking and recognition[13, 14, 7, 8]. On the other hand, volumetric representations characterise a 3D region by describing its space occupation. Each voxel in the region is recorded, allowing for efficient access to each voxel in the region. Most region operations can be implemented efficiently. For example, the computation of region volume, centroid, and inertia axes can be easily implemented. However, techniques for describing region occupation normally require more memory than surface representations. The representation scheme can be used in both low level and high level applications of computer vision, although some operations such as the computation of surface properties may be expensive.

The octree[3] has been generally used as a data structure for describing the space occupation of a 3D object/region in the image. The idea of an octree representation follows directly from the concept of a quadtree[15] used for representing 2D images. The object is assumed to be in a $2^k \times 2^k \times 2^k$ ($k > 0$ is an integer) image block. The block is recursively partitioned into 8 octants, until all voxels in the octant have the same label. Each node in the octree corresponds to an octant, and has at most 8 children. The space requirement of an octree is dependent on the object configuration (shape and size) and position in the image space. In the worst case, when the recursive partition has to be processed down to the single voxel level, the octree has $\sum_{i=0}^{k} 8^i$ nodes. The octree representation is often useful in applications such as object rendering, navigation and computer animation[16, 17, 18, 19]. However, it is not economical for describing small regions (relative to the whole image block). In applications such as image segmentation[5, 20, 21], many small regions are first created and then updated. Using an octree for each individual region in this situation is not efficient.

A more flexible way is to record all the rasterised segments within the region along the scan lines[22, 23]. Martin and Aggarwal[22] use a set of rasterised segments (called *volume segments*) to volumetrically describe a 3D object. The idea can be treated as the generalisation of the run-length coding for 2D binary images[24].

Each volume segment is a run describing a sequence of non-intermittent object voxels in a row parallel to a coordinate axis, and thus is described by positions of its start and end voxels. The space occupation of the object is thus determined by a set of non-intersected runs. Unlike the octree, the run-length representation does not impose any restriction on image size. The memory expense of the representation only depends on the configuration of the region in the space, no matter what the position of the region in the image space is.

In this paper, we examine the run-length representation for volumetrically describing 3D regions. The data structure for recording runs to describe the space occupation of the region is investigated. We show that, in the worst case, the run-length representation has a better performance on the memory expense than the octree representation. A set of basic region operations are implemented by directly accessing the data structure describing the region. The computational cost of these operations is also analysed.

The paper is organised as follows: Section 2 describes the data structure for describing the space occupation of a 3D region using run-lengths. An analysis of the space requirement of this structure is presented, followed by a discussion on some characteristics of the run-length representation. Section 3 gives algorithms for a set of operations applied to 3D regions described by run-lengths. The computational cost of these algorithms is analysed in Section 4. Section 5 presents an application of the proposed representation scheme. Finally, Section 6 is for conclusions.

2 Volumetric description using run-lengths

2.1 Data structure

For 2D images, the run-length representation(24) has been frequently used for encoding binary images. The image is typically described by a list of contiguous horizontal runs of foreground pixels. For each run, the position of its start pixel and the position of its end pixel (or, alternatively, its length) is recorded. Obviously, the spatial configuration of a 2D region can be efficiently described by the runs (see

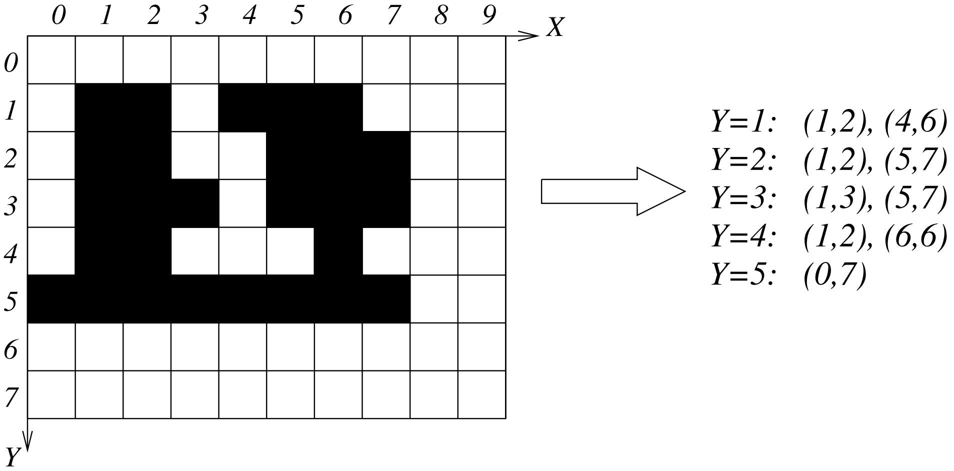


Figure 1: Run-length description of a 2D region (black area)

Fig. 1 for an example).

A 3D image can be considered to be a stack of 2D slices (images) parallel to a coordinate plane in the image coordinate system. The 3D region in the image can then be characterised by describing its space occupation in each slice respectively. The intersection of a 3D region with a slice is a set of 2D regions within the slice[†]. For each intersection, we can use a run-length representation. Martin and Aggarwal[(22)] devised a data structure (Fig. 2) for recording runs describing the region.

A Z-coordinate uniquely determines a slice parallel to the X-Y coordinate plane (called a *X-Y slice*). All the X-Y slices intersecting the region are then recorded with their corresponding Z coordinates flexibly maintained (in the order of their values) in a list (called a *Z-list*). Within each X-Y slice indexed by a Z coordinate, all the rows which are parallel to the X axis (called *X-rows*) and containing the voxels in the region are recorded with their corresponding Y coordinates also ordered into a linked list (called *Y-list*). For each X-row indexed by a Z coordinate and a Y coordinate, voxels in the region are recorded in terms of runs with a singly linked list (called *X-list*) in which each node represents a run by storing X coordinates (x' and

[†]Depending on the shape of the 3D region, the intersection may be a null set.

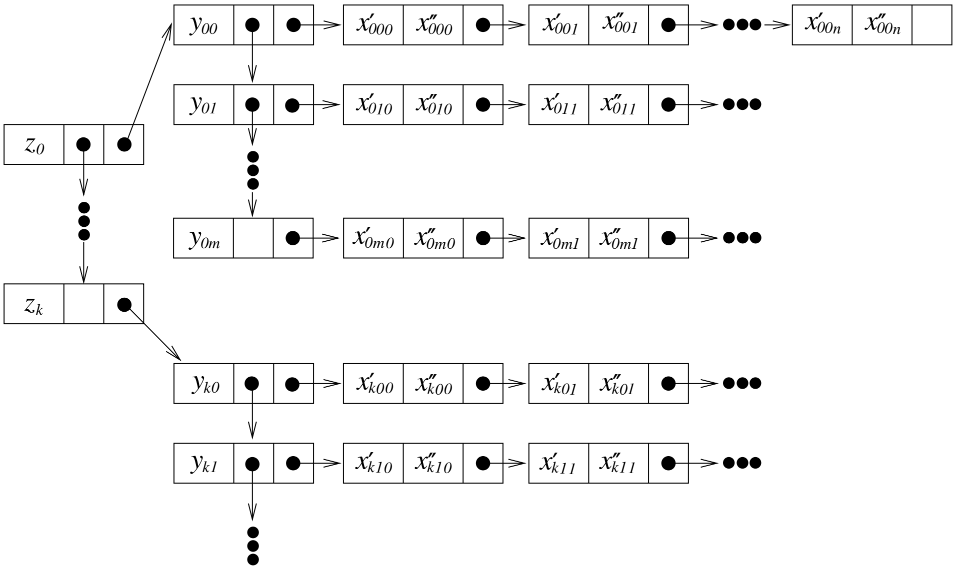


Figure 2: A volumetric description of 3D regions.

x'') of start and end voxels of the run. Nodes in the X-list are ordered by values of X coordinate.

2.2 Memory requirements

For each node in the description structure shown in Fig 2, we need 3 memory units‡. When the region is a cube measuring 2^k units on each side and within a single octant in an image extending 2^{k+1} units in each direction (k is a non-negative integer), the run-length description leads to 2^{2k} runs of length 2^k. Noting that we need 2^k and 2^{2k} nodes to record the Z and Y coordinates of the runs, the cost of the representation in terms of memory space is $3(2^k + 2 \cdot 2^{2k})$. In contrast, the octree representing this cube has only 9 nodes, achieving its best performance on the memory expense. However, when the cube is translated one unit, the octree expands greatly, while the number of nodes in the run-length description remains the same[22]. In general, we have the following theorem regarding the memory expense for the run-length

‡For the convenience of the discussion, we treat each data/pointer field as one unit.

description of 3D regions.

Theorem 2.1 *The memory expense for describing a 3D region in a* $N_x \times N_y \times N_z$ *image using the volumetric representation shown in Fig. 2 is bounded by* $\frac{3}{2}N_xN_yN_z + \frac{9}{2}N_xN_y + 3N_z$.

Proof

For any given region, it intersects with, at most, N_z X-Y slices in the image. Within each X-Y slice, there are, at most, N_y rows containing the voxels in the region. For each row in an X-Y slice, voxels in the region can be recorded by no more than $\left\lceil \frac{N_x}{2} \right\rceil$ runs ($\lceil \cdot \rceil$ means to take the nearest integer greater than or equal to). Thus, the memory space required for describing the region is no more than

$$\begin{aligned} & 3\left(\left\lceil \frac{N_x}{2} \right\rceil N_y + N_y\right) N_z + 3N_z \\ & \le 3\frac{N_x+1}{2}N_yN_z + 3N_yN_z + 3N_z \\ & = \frac{3}{2}N_xN_yN_z + \frac{9}{2}N_yN_z + 3N_z \end{aligned} \tag{1}$$

Q.E.D.

The worst-case memory usage occurs for the region consisting of chessboards, i.e. the intersection of the region with each X-Y slice interchangedly occupies the voxels in the slice (Fig. 3). In this situation, each run contains only one voxel.

It can be shown that, when the region in a $2^k \times 2^k \times 2^k$ image has the configuration shown in Fig. 3, the octree representing the region has $\sum_{i=0}^{k} 8^i$ nodes(3). This is the worst case for the octree representation. Since each node in the octree requires at least 8 memory units, the memory expense for this case is $\sum_{i=1}^{k+1} 8^i$. Substituting $N_x = 2^k$, $N_y = 2^k$, and $N_z = 2^k$, into (1), we have

$$\begin{aligned} & \frac{3}{2}N_xN_yN_z + \frac{9}{2}N_yN_z + 3N_z \\ & = \frac{3}{2}8^k + \frac{9}{2}4^k + 3 \cdot 2^k \\ & < \sum_{i=1}^{k+1} 8^i, \quad \text{when} \quad k > 0 \end{aligned} \tag{2}$$

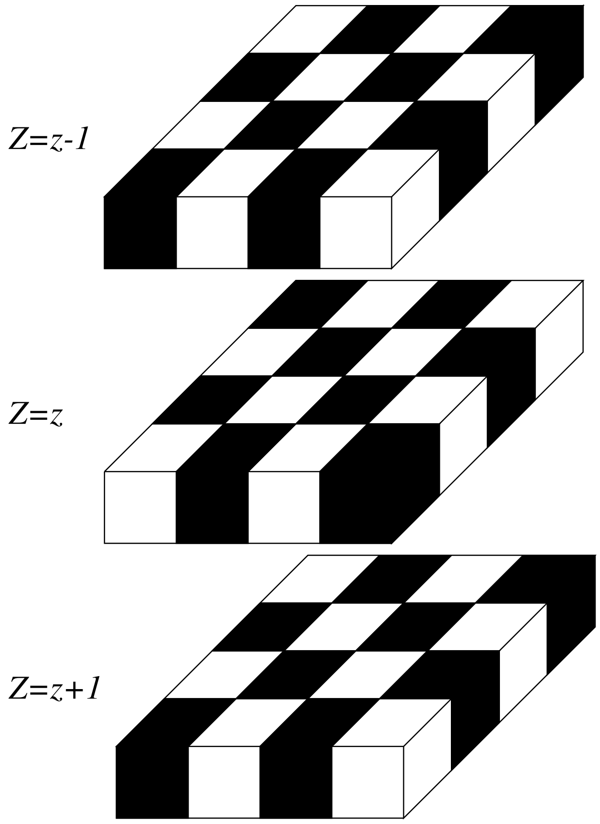


Figure 3: A 3D region consisting of chessboards leads to the worst-case memory requirement for the run-length representation.

This indicates that, in the worst case, the run-length representation has a better performance on the memory expense than the octree representation.

2.3 Discussion

The run-length representation of a 3D region describes the space occupation of the region by decomposing the region into a set of rasterised runs in which runs in the the same raster line are neither intersected nor adjacent, and there exists a one-to-one map between a voxel in a run and a voxel in the region. As shown in Fig. 2, each run can be uniquely described by coordinates of its start and end voxels. Thus, unlike the octree representation, the run-length representation only depends on the coordinates determining the runs and applies no restriction on the image dimension. Since all runs in the representation are parallel to a coordinate axis (in this case the X axis) of the image coordinate system, the translation of a region leads to the same configuration of the description structure except that the data field of nodes stores new coordinate values describing runs corresponding to the translated region. Thus, the run-length representation supports a fast computation on the translation of the region. Also, the memory expense of the representation is invariant under the translation operations. In the traditional octree representation, translation of the region may leads to a significant change in the tree structure and, therefore, a different requirement on memory space.

Given a region in the image, its corresponding run-length representation can be generated by scanning the image along each raster line (here is X-row). The computation only needs to visit each voxel in the image once. In contrast, the transformation from a region in the image to an octree representation requires to repeatedly visit voxels in the image in order to decide whether voxels in an octant have the same label.

Like the octree, the run-length representation is based on the image coordinate system. It is therefore not surprising that the representation is sensitive to the motion (translation and/or rotation)[§] of the region within the image space. A region undergoing motion will, in general, lead to a different structure of the representation. However, as we pointed out previously, translation of the region does not change the configuration of the description structure shown in Fig. 2 except the coordinate values stored in the nodes. Straightforwardly, the translation of a region described by run-lengths can be implemented by simply updating the coordinate values in the description structure. The computation requires to visit each node in the description structure at most once[¶]. Computing the rotation of a region directly based on the run-length representation of the region is not trivial. Meagher[(16)], Ahuja and

[§] Here, we refer to the rigid motion

[¶] In the situation that, for example, there is only a translation along the Z coordinate axis, the computation only needs to visit the nodes with Z coordinate values.

Nash[18], and Weng and Ahuja[19] discussed translation and rotation operations on octrees. The computation of the octree representation of a translated region is significantly more expensive than that of the run-length representation.

One disadvantage of the run-length representation is perhaps that there is no hierarchical nature inherent in the representation. The structure shown in Fig. 2 is unable to represent the region at several different levels of details and, therefore, to support multi-scale/resolution processing[25]. The octree, on the contrary, offers a better flexibility in the hierarchical representation of regions. Each level of the tree can be taken as a representation of the region in a resolution corresponding to the current level and can be directly used as an input to a multi-resolution algorithm.

3 Region operations

In this section, we discuss the implementation of a set of operations applied to 3D regions volumetrically described by the run-length representation shown in Fig. 2. These region operations can be categorised as algebraic operations and are frequently used in many applications. Performance of these operations are analysed in the next section.

In the following discussion, a node in the X-list (see Section 2.1) indexed by $Y = v$ and $Z = w$ is denoted as $(u', u''; v, w)$, where, u' and u'' are X coordinates in the image space. The node $(u', u''; v, w)$ then uniquely specifies a run from the start position (u', v, w) to the end position (u'', v, w) in the image. Thus, we do not distinguish a node in an X-list and the corresponding run the node describes when the meaning in the context is clear. The notation $(u', u''; v, w)$ is used to denote both a physical run in the image and a node in the X-list indexed by $Y = v$ and $Z = w$.

3.1 Belonging

Given a voxel $h(x, y, z)$, we often want to check whether this voxel is inside a region. This is can be easily implemented by, first, searching z and y from the Z-list and the Y-list indexed by z to find a corresponding X-list, and if found, then checking whether there is a run recorded in the list containing the voxel. Obviously, a voxel at the position (x, y, z) is within a run $(u', u''; v, w)$ if:

$$\begin{aligned} &y = v \\ &z = w \\ &u' \leq x \leq u'' \end{aligned} \tag{3}$$

Let $a(p, q)$ be the X-list indexed by $Y = p$ and $Z = q$. The following algorithm returns a logic "true", if the voxel $h(x, y, z)$ is in the region.

Algorithm 1 *Belonging(x,y,z)*

flag $\Leftarrow$ **false**
if $\exists a(y, z)$
  /* *Need to check nodes in the* $a(y, z)$ */
  if $\exists(x'_i, x''_i; y, z) \in a(y, z)$, satisfying $x'_i \leq x \leq x''_i$
    flag $\Leftarrow$ **true**   /* $h(x, y, z)$ *is in the region* */
return flag

3.2 Add and delete

Adding a voxel $h(x, y, z)$ to the region involves either creating a new run or expanding an existing run. If h is an immediate preceding/succeeding voxel of a run corresponding to the ith node, $r_i(x'_i, x''_i; y, z)$, in the X-list $a(y, z)$ in which nodes are organised in an ascending order of the X coordinate, h can be included in the region by simply updating x'_i or x''_i. The resulting r_i may further need to be merged with its neighbour (r_{i-1} or r_{i+1}), if their corresponding runs become adjacent. Otherwise, a new node $(x, x; y, z)$ must be created and inserted into the X-list $a(y, z)$. The algorithm is as follows:

Algorithm 2 *Add(x,y,z)*

if $\nexists a(y,z)$
  / There is no run in the X-row indexed by y and z */*
  create an X-list $a(y,z)$; insert $(x,x;y,z)$ into $a(y,z)$
  return
if $\exists\ r_i(x'_i,x''_i;y,z)\in a(y,z)$, satisfying $x'_i = x+1$ or $x''_i + 1 = x$
  if $x'_i = x+1$ */* Immediate preceding voxel */*
    $x'_i \Leftarrow x'_i - 1$
    if $\exists\ r_{i-1}(x'_{i-1},x''_{i-1};y,z)$, such that $x''_{i-1}+1 = x'_i$
      / Merge with the previous run */*
      $x''_{i-1} \Leftarrow x''_i$; delete r_i
  else   */* Immediate succeeding voxel */*
    $x''_i \Leftarrow x''_i + 1$
    if $\exists\ r_{i+1}(x'_{i+1},x''_{i+1};y,z)$, such that $x''_i + 1 = x'_{i+1}$
      $x''_i \Leftarrow x''_{i+1}$; delete r_{i+1}
else
  insert $(x,x;y,z)$ into the $a(y,z)$

It should be noted that each X-list, Y-list, and Z-list are automatically maintained in an ascending order of the X coordinate, Y coordinate, and Z coordinate respectively during the operation. To decide whether the voxel h is an immediate preceding/succeeding voxel of a run in the region, we have to search through the X-list $a(y,z)$. As $a(y,z)$ is currently in order, the decision can only be made when we pass through a node $r_k(x'_k,x''_k;y,z)$ such that:

$$\begin{aligned} &x''_i \le x, \quad i \le k \\ &x \le x'_{k+1} \end{aligned} \tag{4}$$

This is the right position for inserting a node $(x,x+m;y,z)$ ($m \ge 0$), in an ascending order of the X coordinate, to the current list $a(y,z)$. The cases for the Y-list and Z-list are analogous.

Deleting a voxel $h(x,y,z)$ from the region involves either updating an existing run or splitting an existing run into two new runs. If h is a start/end voxel of a run determined by the node $r_i(x'_i,x''_i;y,z)$, it can be excluded from the region by simply updating x'_i or x''_i. If h is an internal voxel of the run, it is deleted by splitting the r_i into two nodes. Similar to the "Add" operation, we need to search through the X-list $a(y,z)$ in order to decide whether h is an internal voxel or not. The algorithm is as follows:

Algorithm 3 *Delete(x,y,z)*

if $\nexists a(y,z)$
  return /* *The voxel $h(x,y,z)$ is not in the region* */
if $\exists\ r_i(x'_i, x''_i; y, z) \in a(y,z)$, satisfying $x'_i \leq x$ and $x \leq x''_i$
  if $x = x'_i$ or $x = x''_i$
    /* *$h(x,y,z)$ is the start/end voxel of the current run* */
    if $x = x'_i$ /* *Start voxel* */
      $x'_i \Leftarrow x'_i + 1$
    else /* *End voxel* */
      $x''_i \Leftarrow x''_i - 1$
    if $x'_i > x''_i$, delete r_i /* *r_i corresponds to an empty run* */
  else /* *$h(x,y,z)$ is an internal voxel of the current run* */
    /* *Split r_i into two nodes* */
    $s_1 \Leftarrow (x_i, x-1; y, z)$; $s_2 \Leftarrow (x+1, x''_i; y, z)$
    delete r_i; insert s_1 and s_2 into the $a(y,z)$

3.3 Boundary voxels

In some applications, voxels on the boundary of the region are often more important than internal voxels. Given a voxel $h(x,y,z)$, we want to know whether h is on the boundary of the region. Obviously, h will be a boundary voxel if it is a start/end voxel of a run. However, if h is an internal voxel of a run, we need further to check whether its neighbours are all in the region. This involves examining runs in X-rows neighbouring the X-row indexed by the y and z.

Let $a(y,z)$ be the X-list at the yth X-row in the zth X-Y slice. Assuming that the region is 26-connected[‖], the voxel $h(x,y,z)$ is a boundary voxel if and only if it is a start/end voxel of a run, or it is an internal voxel of a run and satisfies one of the following conditions for an i and j, $i,j \in \{-1,0,1\}$ and $|i| + |j| \neq 0$:

1. $a(y+i, z+j)$ does not exist.
2. $\exists r_k(x'_k, x''_k; y+i, z+j)$, such that $x''_k \leq x$ and r_{k+1} does not exist.
3. $\exists r_k(x'_k, x''_k; y+i, z+j)$, such that $x \leq x'_k$ and r_{k-1} does not exist.
4. $\exists r_k(x'_k, x''_k; y+i, z+j), r_{k+1}(x'_{k+1}, x''_{k+1}; y+i, z+j)$, such that $x''_k \leq x \leq x'_{k+1}$.

[‖]6-connected regions can be similarly treated.

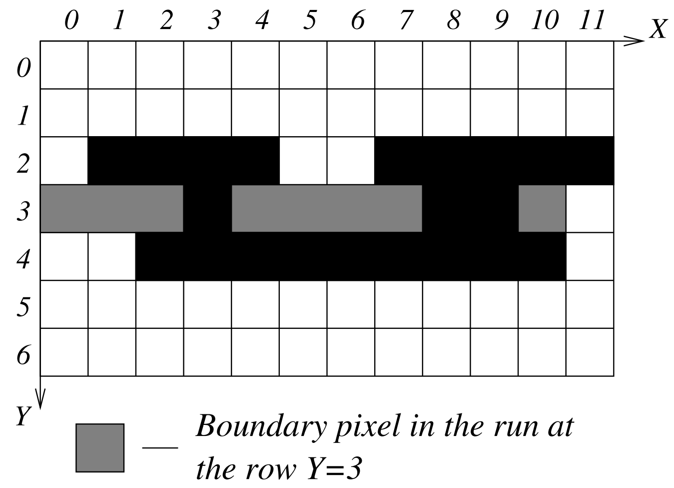


Figure 4: An illustration of boundary intervals in a 8-connected 2D region. The run at the row $Y = 3$ contains 3 boundary intervals (0,2;3), (4,7;3), and (10,10;3).

The algorithm for the boundary testing of a voxel is thus straightforward and is not presented. This implementation also has a side product which indicates whether the given voxel belongs to the region.

In computer graphics and virtual reality[26, 27], applications often require access to all the voxels on the boundary in order to render the surface of the region. Thus, given a region, we need to find all the boundary voxels. Testing boundary conditions for each voxel in a run is not efficient. Instead, we detect boundary intervals in each run. A boundary interval in a run is a maximum sub-run in which every voxel is on the boundary of the region (see Fig. 4 for an illustration in 2D case[**]).

Boundary intervals in a run can be determined by examining all the runs spatially neighbouring the current one. A boundary interval found from a run recorded in an X-list $a(y, z)$ can be uniquely described by a run $(x_s, x_e; y, z)$. To find all the boundary intervals from the runs recorded by an X-list $a(y, z)$, we simply scan through the $a(y, z)$ and $a(y + i, z + j)$ ($i, j \in \{-1, 0, 1\}$ and $|i| + |j| \neq 0$), and

[**] We use $(x', x''; y)$ to denote a run in a 2D region.

determine the x_s and x_e for each boundary interval. Let $r_k(x'_k, x''_k; y, z)$ be the currently visited node in the $a(y, z)$. Initialising both x_s and x_e to be the x'_k, access to each $a(y+i, z+j)$ is then stepped forward until a node $s_{l_{ij}}(u'_{l_{ij}}, u''_{l_{ij}}; y+i, z+j)$ satisfying the condition $u''_{l_{ij}} > x_e + 1$ is reached. A part of the boundary interval can be determined by updating the x_e as follows:

$$x_e = \begin{cases} x''_k & \text{if } \nexists s_{l_{ij}} \text{ such that } u''_{l_{ij}} > x_e + 1 \\ \min\left\{x''_k, \max_{i,j}\{x'_k, u'_{l_{ij}}\}\right\} & \text{otherwise} \end{cases} \tag{5}$$

An X coordinate x_n used as a candidate of the x_s for the next boundary interval needs to be maintained. Using the initial value x''_k, x_n can be determined as follows:

$$x_n = \begin{cases} x_e + 1 & \text{if } x_n \leq x_e \\ \min_{i,j}\{x''_k, u''_{l_{ij}}\} & \text{otherwise} \end{cases} \tag{6}$$

The x_e and x_n need to be repeatedly updated until a complete boundary interval is found when $x_e = x''_k$ or $x_n > x_e + 1$. The access to an $a(y+i, z+j)$ is stepped to the next node $s_{l_{ij}+1}(u'_{l_{ij}+1}, u''_{l_{ij}+1}; y+i, z+j)$ as long as $u''_{l_{ij}} \leq x_e + 1$. After a boundary interval is determined, the x_s and x_e are set to the x_n for finding the next boundary interval. The access to the $a(y, z)$ is stepped to the next node $r_{k+1}(x'_{k+1}, x''_{k+1}; y, z)$ if $x_s > x''_k$. Consequently, all the boundary intervals in runs recorded by an X-list $a(y, z)$ can be detected by a one pass scan simultaneously through the $a(y, z)$ and $a(y+i, z+j)$. Fig. 5 illustrates an example in the 2D case for finding boundary intervals in an X-row.

The following algorithm detects all the boundary voxels in a region R, and puts them into a set S.

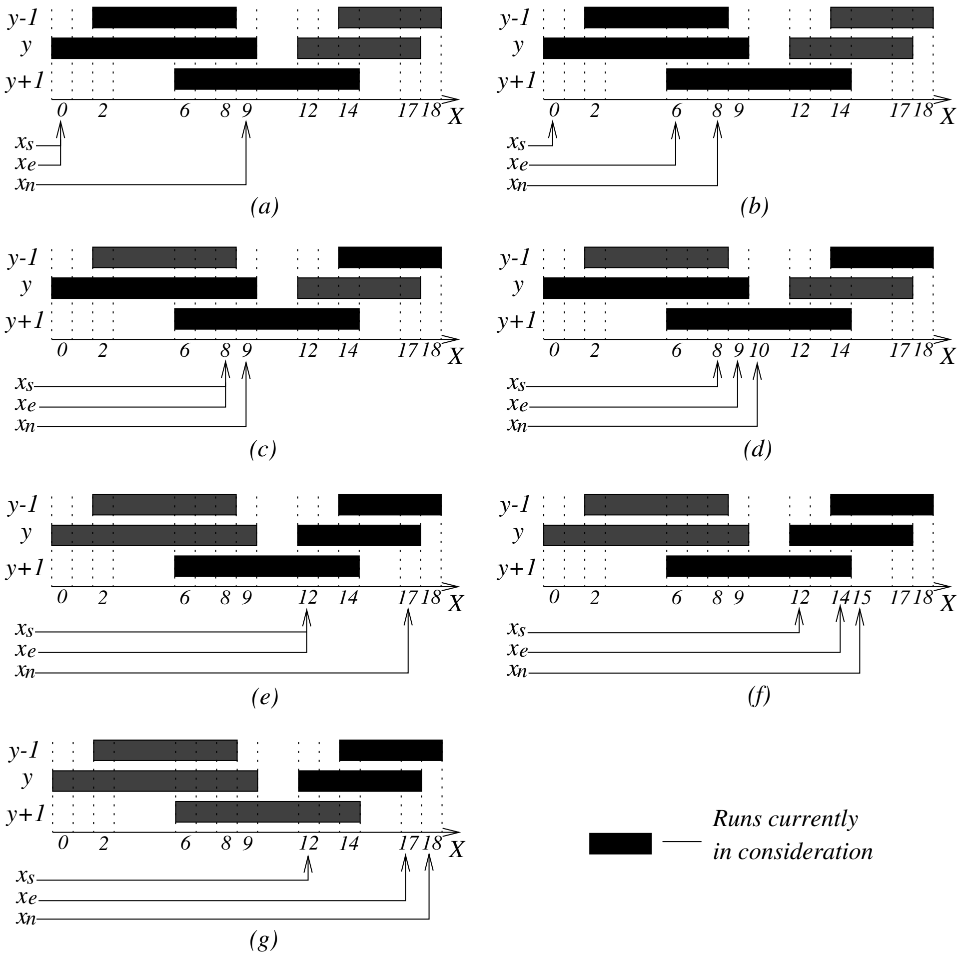


Figure 5: Detecting boundary intervals in the row $Y = y$ in a 2D region. (a) Initialise x_s, x_e, and x_n, and step to the corresponding runs; (b) Update x_e and x_n. $x_n > x_e{+}1$, $(0, 6; y)$ is thus a boundary interval; (c) Update x_s, x_e, and x_n for finding the next boundary interval. Step to the next run in the y–1 row. (d) Update x_e and x_n. x_e is at the end position of the current run in the y row. $(8, 9; y)$ is thus a boundary interval; (e) Update x_s, x_e, and x_n for finding the next boundary interval, and meanwhile step to the next run in the y row; (f) Update x_e and x_n. $x_n = x_e + 1$, a part of the boundary interval has been found. The next run in the $y+1$ row does not exist; (g) Continue to update x_e and x_n. x_e is at the end position of the current run in the y row. $(12, 17; y)$ is thus a boundary interval.

Algorithm 4 *Boundary(R, S)*

for each $a(y,z)$ in R
  $k \Leftarrow 0$; $r \Leftarrow (x'_k, x''_k; y, z) \in a(y,z)$ /* *Get the first node in* $a(y,z)$ */
  /* *Get the access to nodes in* $a(y+i, z+j)$ */
  for each $a(y+i, z+j)$, $-1 \leq i, j \leq 1$ and $|i| + |j| \neq 0$
    $l_{ij} \Leftarrow 0$; $s_{ij} \Leftarrow (u'_{l_{ij}}, u''_{l_{ij}}; y+i, z+j) \in a(y+i, z+j)$
  /* *Find all boundary intervals in runs recorded in the* $a(y,z)$ */
  /* *Use* x_s *and* x_e *to record the current boundary interval* */
  $x_s \Leftarrow x'_k$; $x_e \Leftarrow x_s$
  while r exists
    $x_n \Leftarrow x''_k$ /* *Use* x_n *to record the next candidate for* x_s */
    /* *Find all boundary intervals in the run* $(x'_k, x''_k; y, z)$ */
    for each s_{ij}
      /* *Get access to the corresponding runs in* $a(y+i, z+j)$ */
      while s_{ij} exists and $u''_{l_{ij}} \leq x_e + 1$
        $l_{ij} \Leftarrow l_{ij} + 1$; $s_{ij} \Leftarrow (u'_{l_{ij}}, u''_{l_{ij}}; y+i, z+j)$
      if s_{ij} does not exists
        /* *Run to the end of a* $a(y+i, z+j)$. $(x_s, x''_k; y, z)$ *and the* */
        /* *remaining runs recorded in* $a(y,z)$ *are all boundary intervals* */
        end_flag $\Leftarrow$ **true**
        break /* *Exit from for-loop. Needn't to check other* $a(y+i, z+j)$ */
      if $u'_{l_{ij}} \geq x_s$, $x_e \Leftarrow \min\left\{x''_k, \max\{u'_{l_{ij}}, x_e\}\right\}$ /* *Update* x_e */
      if $x_n \leq x_e$, $x_n \Leftarrow x_e + 1$, **else** $x_n \Leftarrow \min\{u''_{l_{ij}}, x_n\}$ /* *Update* x_n */
    if end_flag = **true**
      add voxels in $(x_s, x''_k; y, z)$ to S
      add voxels in the remaining runs recorded in $a(y,z)$ to S
      break /* *Exit from while-loop. Check a new* $a(y,z)$ */
    if $x_n > x_e + 1$ or $x_e = x''_k$
      /* *Found a complete boundary interval* */
      add voxels in $(x_s, x_e; y, z)$ to S
      $x_s \Leftarrow x_n$; $x_e \Leftarrow x_s$
    if $x_s > x''_k$
      /* x_s *has passed the current run* r. *Check the next node in* $a(y,z)$ */
      $k \Leftarrow k + 1$; $r \Leftarrow (x'_k, x''_k; y, z)$
      $x_s \Leftarrow x'_k$; $x_e \Leftarrow x_s$

3.4 Relational operations

Two regions in the image could be spatially intersecting, adjacent, or separate. The spatial relationship between two regions can be determined by examining the spatial relationship between the runs in two respective regions. Let $r(x', x''; y, z)$ and

$s(u', u''; v, w)$ be two runs in regions R_1 and R_2 respectively. The conditions associated with the relationship between R_1 and R_2 can be summarised as follows:

1. R_1 and R_2 are intersecting, if and only if there exists a r and s satisfying one of the following conditions:

$$y = v, z = w, x' \leq u' \leq x''$$
$$y = v, z = w, u' \leq x' \leq u''$$

2. R_1 and R_2 are adjacent, if and only if they are not intersecting and there exists a r and s satisfying one of the following conditions for $i, j \in \{-1, 0, 1\}$ and $|i| + |j| \neq 0$:

$$y = v, z = w, x' = u'' + 1$$
$$y = v, z = w, u' = x'' + 1$$
$$y = v + i, z = w + j, x' \leq u' \leq x'' + 1$$
$$y = v + i, z = w + j, u' \leq x' \leq u'' + 1$$

3. R_1 and R_2 are separate, if and only if they are neither intersecting nor adjacent.

The algorithm can be implemented by successively scanning through X-lists in R_1 and testing the above conditions by simultaneously scanning through relevant X-lists in R_2. Obviously, the algorithm will terminate as soon as an intersection of two runs in the respective regions is detected. However, the decision for the adjacent and separate relation can only be made after examining each run in R_1. Details of the algorithm are not presented.

On some occasions, voxels on the adjacent borders between two regions are needed. Given two regions R_1 and R_2, adjacent voxels in R_1 can be computed by scanning through each run in R_1. For each run in R_1, sub-runs on the adjacent border can be detected by testing the adjacent condition using the relevant runs in R_2 and determining the adjacent voxels in a similar way for finding boundary intervals.

3.5 Union and intersection

Given two regions R_1 and R_2, their union, R, can be computed by copying R_2 to R, and then merging all the runs in R_1 into R. Let $a(y,z)$ and $b(y,z)$ be two corresponding X-lists in R and R_1 respectively. Merging $b(y,z)$ into $a(y,z)$ can be implemented by simultaneously stepping through the $b(y,z)$ and $a(y,z)$ respectively and updating the list $a(y,z)$ according to the spatial relationship between the corresponding runs. For a node $s \in b(y,z)$ whose corresponding run is not overlapping or adjacent to any run recorded in the $a(y,z)$, it is simply inserted into the list $a(y,z)$. Otherwise, the first node $r \in a(y,z)$ whose corresponding run is overlapping or adjacent to the run determined by s is updated. Nodes succeeding r are deleted from the $a(y,z)$ if their corresponding runs become overlapping or adjacent to the run determined by the updated r. The following algorithm computes the union of R_1 and R_2, and puts the result into R.

Algorithm 5 *Union(R_1, R_2, R)*

copy R_2 to R
for each $b(y,z) \in R_1$
  if $\nexists a(y,z) \in R$
    copy $b(y,z)$ to R;
    continue */* Skip the following to continue the for-loop */*
  / Merge $b(y,z)$ into $a(y,z)$ */*
  $k \Leftarrow 0$, $r \Leftarrow (x'_k, x''_k; y, z) \in a(y,z)$; $l \Leftarrow 0$; $s \Leftarrow (u'_l, u''_l; y, z) \in b(y,z)$
  while s exists
    if $u''_l + 1 < x'_k$ */* $(u'_l, u''_l; y, z)$ is on the left of $(x'_k, x''_k; y, z)$ */*
      insert s to $a(y,z)$
      $l \Leftarrow l+1$; $s \Leftarrow (u'_l, u''_l; y, z)$ */* Check next node in $b(y,z)$ */*
    else if $u'_l > x''_k + 1$ */* $(u'_l, u''_l; y, z)$ is on the right of $(x'_k, x''_k; y, z)$ */*
      $k \Leftarrow k+1$; $r \Leftarrow (x'_k, x''_k; y, z)$ */* Check next node in $a(y,z)$ */*
    else */* $(u'_l, u''_l; y, z)$ is overlapping/adjacent to $(x'_k, x''_k; y, z)$ */*
      if $u'_l < x'_k$, $x'_k \Leftarrow u'_l$ */* Overlapping/adjacent to the left side */*
      if $u''_l > x''_k$, $x''_k \Leftarrow u''_l$ */* Overlapping/adjacent to the right side */*
      $l \Leftarrow l+1$; $s \Leftarrow (u'_l, u''_l; y, z)$ */* Check next node in $b(y,z)$ */*
    / Delete successors of r if their corresponding runs are */*
    / overlapping/adjacent to the run determined by the current r */*
    while $x'_{k+1} \leq x''_k + 1$
      if $x''_{k+1} \geq x''_k$, $x''_k \Leftarrow x''_{k+1}$
      delete $(x'_{k+1}, x''_{k+1}; y, z)$ */* The next one becomes $k+1$th node */*
    if r does not exist */* Come to the end of $a(y,z)$ */*
      append $(u'_{l+i}, u''_{l+i}; y, z)$, $i = 0, 1, \ldots$, to $a(y,z)$

The intersection of two regions can be computed by examining two corresponding X-lists in respective regions and detecting the intersected runs. The intersected intervals form runs in the resulting region. The algorithm is not presented.

4 Analysis

In this section, we give a performance analysis for the region operations discussed in the previous section. For the convenience of the discussion, the image size is assumed to be $N \times N \times N$. The performance is measured by the number of basic operations required and expressed in relation to the image size and/or the number of runs in the region. In the following discussion, we use M to denote the total number of runs in the region and L to denote the smaller value of M and N, i.e. $L = \min\{M, N\}$.

To determine whether the representation of a region contains an X-list $a(y, z)$, we need to check through the Z-list and the Y-list indexed by the z (see Section 2.1). The length of each of these lists is at most L. Thus, detecting the existence of the $a(y, z)$ requires no more than $2L$ operations.

Given a voxel $h(x, y, z)$, operations *Belonging*, *Add*, and *Delete* all involve processing for finding a run satisfying some conditions applied to the run and the voxel coordinate. If the representation of the region contains the X-list $a(y, z)$, nodes in this list are then successively visited to test the conditions. In the worst case, the last node in the list must be visited before the decision for any further processing can be made. Since there are at most $\min\{\left\lceil \frac{N}{2} \right\rceil, M\}$ node in an X-list, scanning through an X-list requires no more than L operations. Incorporating the time required for detecting the existence of the $a(y, z)$, the performance of these operations is thus bound by $O(3L)$.

The operation for determining whether a given voxel $h(x, y, z)$ is on the boundary of the region involves detecting the existence of X-lists $a(y, z)$ and $a(y+i, z+j)$ ($i, j \in \{-1, 0, 1\}$ and $|i| + |j| \neq 0$), and then scanning through the nodes in the $a(y, z)$ to check the boundary conditions described in Section 3.3. In the worst case, the algorithm scans through the $a(y, z)$ and all 8 $a(y + i, z + j)$ and reaches a decision only after the last node in each of these lists has been visited. Thus, the performance of the operation is bound by

$$8 \cdot 2L + 8L = 24L$$

To analyse the performance of *Boundary(R,S)* and relational operations, we first consider the computational cost related to accessing all the runs in the region. Let M_1 and M_2 be the number of runs in regions R_1 and R_2 respectively. We have the following theorem regarding the cost for accessing runs in the region.

Theorem 4.1 *Assume that successively visiting each node once in a X-list $a(y, z)$ in R_1 also requires access to each node once in X-lists $b(y+i, z+j)$, $i, j \in \{-1, 0, 1\}$, in R_2. The number of operations, C, required for successively accessing all the runs in R_1 is no more than $3M_1 + 18M_2^2$.*

Proof
Let $A(y, z)$ and $B(v, w)$ be the number of nodes in the X-lists $a(y, z) \in R_1$ and $b(v, w) \in R_2$ respectively. Thus, successively visiting each node once in $a(y, z)$ and $b(v, w)$ requires $A(y, z)$ and $B(v, w)$ operations respectively. According to the assumption, for each X-list $a(y, z)$, we need also to scan the 9 corresponding X-lists $b(y + i, z + j)$, $i, j \in \{-1, 0, 1\}$. Therefore, nodes in each $b(v, w)$ have to be repeatedly visited at most 9 times in order to scan through all the X-lists in R_1. Noting that scanning through the Z-list and all Y-lists in R_1 requires no more than $2M_1$ operations, and that detecting the existence of a $b(v, w)$ requires no more than $2M_2$ operations, we have

$$\begin{aligned} C &\leq 2M_1 + \sum_{y,z} A(y, z) + 2M_2 \cdot \sum_{v,w} 9B(v, w) \\ &= 3M_1 + 18M_2^2 \end{aligned} \tag{7}$$

Q.E.D.

Based on this theorem, if scanning through each X-list $a(y, z)$ in R_1 also requires scanning through 8 X-lists $a(y + i, z + j)$, $i, j \in \{-1, 0, 1\}$ and $|i| + |j| \neq 0$, the number of operations, C', required for accessing all the runs in R_1 then satisfies the following condition:

$$C' \leq 3M_1 + 16M_1^2 \tag{8}$$

It is also clear that for a region with M runs, accessing each run once requires no more than $3M$ operations.

The operation *Boundary(R,S)* detects all the voxels on the boundary of the region. Boundary voxels within each run recorded in a X-list $a(y, z)$ are determined by also examining the nodes in at most 8 X-lists $a(y + i, z + j)$, $i, j \in \{-1, 0, 1\}$ and $|i| + |j| \neq 0$. According to the above discussion, the computational cost of the operation in the worst case is $O(3M_1 + 16M_1^2)$.

The relation between two regions is determined based on the positional relation of runs recorded in $a(y, z)$ and $b(y+i, z+j), i, j \in \{-1, 0, 1\}$. For the adjacent and separate regions, the decision can be made only after all the runs in R_1 have been checked. According to the above theorem, the computational cost in the worst case is $O(3M_1 + 18M_2^2)$.

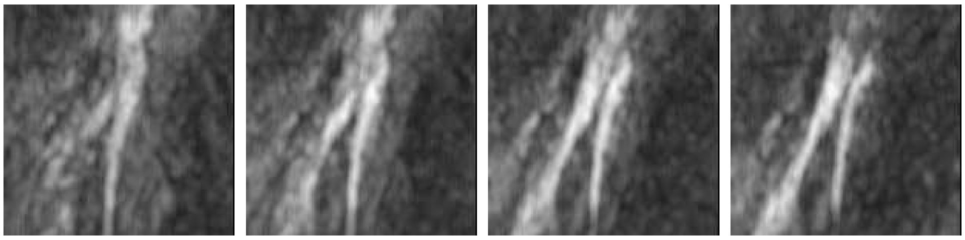
The *Union(R_1, R_2, R)* operation first spends at most $3M_2$ operations to copy the region R_2 to R. For each X-list $b(y, z)$ in R_1, a corresponding X-list $a(y, z)$ in R is scanned. Nodes in $b(y, z)$ are merged into R by either copying the $b(y, z)$ to R or updating the already existing $a(y, z)$. All nodes in $b(y, z)$ and $a(y, z)$ are at most visited once. Since each X-list, Y-list and Z-list are maintained in an ascending order of X, Y and Z coordinate respectively, the operation just simultaneously steps through each node in R and R_1 respectively. The computational cost of the *Union(R_1, R_2, R)* is thus bound by

$$3M_2 + 3M_1 + 3M_2 = 3M_1 + 6M_2$$

The performance for the intersection of two regions can be similarly analysed.

5 Applications

The proposed representation scheme has been implemented as an independent C++ class, and has been successfully used in our recent work on the segmentation of 3D



Slice at $Z = 3$ Slice at $Z = 6$ Slice at $Z = 9$ Slice at $Z = 12$

Figure 6: A $150 \times 150 \times 15$ 3D seismic image. Only 4 X-Y slices are shown.

gray level images(6). The segmentation is achieved by applying a bottom-up hierarchical analysis to progressively agglomerate small regions into larger homogeneous regions. Region operations such as the union of two regions, examining spatial relations between two regions, and computing adjacent borders between two regions are heavily used. The efficient representation scheme plays an important role in the efficiency (both on computation and memory) of the segmentation algorithm. An operation for transforming the run-length representation of a 3D region into a geometric surface model for visualisation of the region has also been implemented, by generating a mesh from boundary voxels of the region.

Fig. 6 shows 4 X-Y slices of a 3D ($150 \times 150 \times 15$) seismic image. The image contains two channels (with bright gray levels) whose 3D structures are most important to geologists. Fig. 7 shows the extracted channels obtained from the segmentation result. The channel regions are rendered by transforming the corresponding run-length representations into quad-mesh models.

6 Conclusion

We have discussed a representation scheme for volumetrically describing 3D regions. The space occupation of the region is described using run-lengths which is a generalisation of the run-length coding for 2D binary images. The memory expense of the data structure for describing the region has been analysed and com-

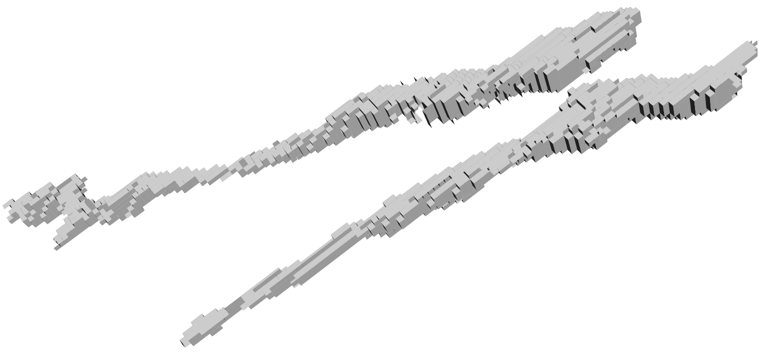


Figure 7: Two channel regions extracted from the image shown in Fig. 6.

pared with that of the octree representation. We have shown that, in the worst case, the run-length representation has a better performance than the octree representation. Some characteristics of the run-length representation has also been discussed in comparison with the octree representation. A set of region operations which are basic to many applications have been implemented by directly accessing the data structure describing the region. The representation scheme has been tested in our work on the segmentation of 3D gray level images. With these basic region operations, the representation scheme is useful to applications such as computer vision, computer graphics and robotics.

References

1. D. H. Ballard and C. M. Brown. *Computer Vision*. Prentice-Hall, 1982.

2. A. Rosenfeld and A. C. Kak. *Digital Picture Processing*. Academic Press, 2nd edition, 1982.

3. C. L. Jackins and S. L. Tanimoto. Oct-trees and Their Use in Representing Three-dimensional Objects. *Computer Graphics and Image Processing*, CGIP-14(3):249–270, November 1980.

4. A. Pentland and S. Sclaroff. Closed-form Solution for Physically Based Shape Modelling and Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-13(7):715–729, July 1991.

5. M. Spann and E. P. Ong. A Least-Median-of-Squares Based Region Growing Algorithm for Greylevel and Motion Segmentation, 1997. Submitted to *Computer Vision and Image Understanding*.

6. X. Shen, M. Spann, and P. Nacken. Segmentation of 2D and 3D Images through a Hierarchical Clustering Based on Region Modelling, 1997. Submitted to *Pattern Recognition*.

7. P. J. Besl and R. C. Jain. Three-dimensional Object Recognition. *ACM Computing Surveys*, 17(1):75–145, March 1985.

8. R. T. Chin and C. R. Dyer. Model-based Recognition in Robot Vision. *ACM Computing Surveys*, 18(1):67–108, March 1986.

9. P. J. Besl. Geometric Modeling and Computer Vision. *Proceedings of IEEE*, 76(8):936–958, August 1988.

10. D. C. Hogg. Shape in Machine Vision. *Image and Vision Computing*, 11(6):309–316, 1993.

11. X. Shen and D. Hogg. Generic 3-D Shape Model: Acquisition and Applications. In V Hlavac and R Sara, editors, *6th International Conference on Computer Analysis of Images and Patterns*, pages 98–105. Springer-Verlag, 1995.

12. X. Shen and M. Spann. 3D Shape Modelling Using A Multi-scale Surface Model. In *IEEE International Conference on Image Processing*, 1997. Accepted.

13. D. Terzopoulos, A. Witkin, and M. Kass. Symmetry-seeking Models and 3D Object Reconstruction. *International Journal of Computer Vision*, 1:211–221, 1987.

14. P. Lindsey and A. Blake. Real-time Tracking of Surfaces with Structured Light. In E. Hancock, editor, *British Machine Vision Conference 1994*, volume II, pages 619–628. BMVA Press, 1994.

15. H. Samet. Region Representation: Quadtrees from Boundary Codes. *The Communication of ACM*, 23(3):163–170, March 1980.

16. D. J. Meagher. Geometric Modelling Using Octree Modelling. *Computer Graphics and Image Processing*, 19(2):129–147, 1982.

17. N. Ahuja, R. T. Chien, R. Yen, and N. Bridwell. Interference Detection and Collision Avoidance Among Three Dimensional Objects. In *1st National Conference on Artificial Intelligence*, pages 44–48, 1980.

18. N. Ahuja and C. Nash. Octree Representation of Motion Objects. *Computer Vision, Graphics and Image Processing*, 26:207–216, 1984.

19. J. Weng and N. Ahuja. Octree of Objects in Arbitrary Motion: Representation and Efficiency. *Computer Vision, Graphics and Image Processing*, 39:167–185, 1987.

20. P. Nacken. Image Segmentation by Connectivity Preserving Relinking in Hierarchical Graph Structures. *Pattern Recognition*, 28(6):907–920, 1995.

21. A. Montanvert, P. Meer, and A. Rosenfeld. Hierarchical Image Analysis Using Irregular Tessellations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(4):307–316, April 1991.

22. W. N. Martin and J. K. Aggarwal. Volumetric Descriptions of Objects from Multiple Views. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(2):150–158, March 1983.

23. C. Montani. Region Representation: Parallel Connected Stripes. *Computer Vision, Graphics and Image Processing*, 28:139–165, 1984.

24. R. M. Haralick and L. G. Shapiro. *Computer and Robot Vision*. Addison-Wesley Publishing Company, 1992.

25. A. Rosenfeld. *Multiresolution Image Processing and Analysis*. Springer Verlag, Berlin, 1984.

26. J. D. Foley. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, 1982.

27. H. Rheingold. *Virtual Reality*. Summit Books, 1991.