A fast 3D seed fill algorithm avoiding redundant seed searching by 2 ½ D linked lists

Ming-Dar Tsai, Yee-De Yeh, Shyan-Bin Jou

Department of Information and Computer Engineering, Chung Yuan Christian University, Chung-Li, Taiwan 32023, R.O.C.

E-mail:{tsai@, jou,yeh@earth.}ice.cycu.edu.tw

This paper presents a 3D seed and flood algorithm that uses a 2 ½ D linked list to avoid the redundant seed searches that appear in the current 3D seed and fill algorithms. The linked lists records a set of shadows cast by the filled spans. We maintain the linked lists and compare the extent of the current span with the extents in neighboring spans to minimize references to the voxels. The count for filled voxels is corrected and verified by the comparision with the Oikarinen's 3D algorithm. Verification of our algorithm and the Feng's algorithm by artificial and measured volumes demonstrates the effectiveness of our method in eliminating redundant seed searching and thus improving the efficiency.

Key words: Seed and fill, Seed searching, Volume graphics, Computer Graphics

Contact to Ming-Dar Tsai associate 03-456-3171*4718

1 Introduction

A seed is an atom for generating other components with the same property. It may be a specified closed contour from which other parallel contours can be matched to form 3D closed surfaces (Liu and Ma 1999) or for calculating different region between two slices (Zhou and Toga 2000). However, a seed is generally considered as a point among a 2D (pixel) or 3D (voxel) regularly partitioned area or volume and acts as an interior pixel (or voxel) to fill all of the pixels or voxels inside a closed boundary.

Many applications utilize the seed and fill principle. For example, Oikarinen et al (1998) used 2.5D seed filling in a view lattice to avoid processing empty space to accelerate their volume rendering. They also combined the seed filling algorithm with a template algorithm to optimize stepping in resample volumes and thus to fast render maximum intensity projections in time-variant medical data (Oikarinen and Jyrkinen 1998). Levoy (1990) used seed filling instead of scene-sensitive octrees or bounding-boxes in accelerating ray-tracing to leap over the empty space. Suárez et al (1996), introduced a complementary region concept that dynamically located all unknown region complementary to some region by surface filling. They also demonstrated an application to robot path planning. Yu and Wang (1999) used 3×3 window masking to detect edges during filling an image. Tsai et al (2001) manipulated (such as cut, join etc) volumetric objects by seed and fill instead of using memory-consuming voxel extension or manipulation-inconvenient linked lists. They also demonstrated an application of such seed and fill to surgical simulations (Tsai et al 2001a).

Straightforward seed and fill algorithms choose new seeds from unfilled neighbors (edge-sharing pixels or face-sharing voxels). From these new seeds, fills are implemented using a recursive function. The recursive call can have a simple program but inevitably consumes huge amounts of memory especially in 3D volumes and thus spend much time. Fish-

kin and Basky (1985) improved the efficiency by assigning shorter voxel spans with higher priorities. Albert et al (1995) used a queue list to store new neighboring seeds and iteratively implemented new fill from a seed in the list until the list is empty. Albert proposed the idea of filling consecutive voxels along some direction instead of filling one neighboring voxel to consume less memory and improve efficiency.

Feng and Soon (1998) also filled consecutive voxels (called a span of voxels) along some (say *x*-) direction and stacked 4 new seeds along the *+y*, *-y*, *+z* and *-z* directions neighboring the *x* span of the filled voxels. Fig. 1 shows the filling sequence, in which the voxel fD is the initial seed.

The Feng's algorithm has the merit of reducing the stack size, however still suffers problems of abundantly stacking seeds and searching seeds in filled spans that absolutely contains no new seeds. The reasons of bring redundant seed stacking and visiting on filled spans are introduced in Section 2.1. Section 2.2 discusses possible solutions for solving the problems and thus makes clear our strategy. Section 2.3 introduces the proposed algorithm for solving the problems.

Oikarinen (1998) proposed a 2D filling algorithm with the idea of never visiting twice the filled voxels. The Oikarinen's algorithm avoids recursions by designating neighboring pixels of the current pixel as child pixels and examines them in sequence (Fig. 2). Once an unfilled pixel is found, it is filled and then taken as the current pixel or "rollback" to the parent pixel when all of its child pixels have been filled. This process is completed when the initial seed pixel is encountered. The algorithm uses no stack and thus has the merit of constant memory, however suffers the time-consuming demerit of repetitive rollback for filled pixels. We rewrote the algorithm as a 3D version (Appendix A) to prove that our proposed algorithm has also the merit of never visiting twice the filled voxels. Section 3 presents some experimental results and comparisons among the Feng's, Oikarinen's and proposed algo-

rithm. Conclusions are made in Section 4.

2 Efficient Seed and Fills with Linked Lists Recording Filled Spans

2.1 Redundant seed stacking for unfilled spans and redundant seed searching for filled span

The Feng's seed and fill algorithm has two problems. First, multiple seeds inside the same span may be redundantly stacked and results over-counting filled voxels. An example (Fig. 3) shows how the redundant seeds are generated in the Feng's algorithm. Voxel S (3,2,2) is the initial seed. Then, the voxels in-between Voxels (2,2,2) and (5,2,2) are filled. Voxels (4,3,2) and (4,1,2) are news seeds in the *y*/-*y* -axes, Voxels (2,2,3) and (4,2,3) are seeds in the *z*-axis, and Voxel (4,2,1) is the seed in the -*z*-axis (Labeled with 1,2,3,4,5 respectively). The five seeds are stacked and the last seed (4,2,1) is popped. Consequently, voxels inside the span ((2,2,1)~(4,2,1)) are filled. Voxels (4,3,1) and (4,1,1) become seeds in the *y*/-*y* –axes respectively and are stacked. Again, voxels inside the span ((2,1,1)~(5,1,1)) are filled, and only one new seed appears that is the same as seed 2 (**bold** entries) in the stack.

Fig. 3 also illustrates the other problem: searching seeds in filled spans. For example, the voxels between the span ((2,2,2)~(4,2,2)) are checked again for a new seed when the seed (4,2,1) is being processed, however the voxels of (2,2,2)~(5,2,2) are already filled when the initial seed is processed.

2.2 Possible solutions preventing redundant seed stacking and seed searching

A strategy that checks if a voxel in a span including the candidate seed has already been a seed in the stack can solve the problem of redundant seed stacking. Although intuitive, such examination is time-consuming without any auxiliary structure, because voxels between the candidate seed and every seed in the stack must be check. Therefore, our strategy for preventing re-counting filled voxels allows the generation of redundant seeds but modifies the Feng's algorithm as to check of voxel status in the loop. That means when the seed is popped out it is checked to know whether already filled. If it is, the span will not be filled again and the count on the total filled voxels will not be performed.

A possible strategy that improves the problems of searching for seeds can be filling neighbors without stacking the seeds and visiting voxels only once (no rollback). As Fig.1 shows, (fD is the initial seed) if neighboring spans are directly filled without finding new seeds in advance, span D is filled, followed by span E and span F. The process then stops because no seed has been stacked and produces the wrong result. To solve this problem, the leftmost filled voxel can be chosen as a new seed after filling the entire span. However, such an algorithm cannot also continue filling. For example, dE (leftmost) is chosen as a seed after the span is filled. dF, the neighbor in the *y* direction, is a boundary voxel. Therefore, neither voxels are filled nor new seeds are found, and span F will then not be filled. There might have some tricks to handle such case, but the algorithm would become tedious and time-consuming.

Therefore, our strategy records the filling process to prevent redundant seed searching for filled spans. We use one node to record the extent (leftmost and rightmost) of a span and then a linked list of nodes to record all spans of the same row. The extents of neighboring

spans of a neighboring row and the extent of the current span are compared before searching seeds along the row. The comparison result tells whether the spans along the neighboring row were already filled and determines if a seed search is implemented or not.

2.3 2 1/2 D linked lists reducing repeated seed searches and fill counts

Fig. 4 shows the linked lists of the 2D array and the data structure in which **listhead[y][z]** are the pointers pointing to the linked lists. Each pointer points to a set of filled spans along the *x*-direction. A node stores the extent of a span. The nodes of the same row are added to the same linked list. In the above example, a linked list pointed by pointer (2,3) contains two nodes, while the remaining lists all have one node.

The extent comparison between the current span and the spans recorded in the linked list of a neighboring row is carried out by node's leftmost and rightmost data. This comparison is classified into 5 cases as shown in Fig.5. Ideally, the spans recorded in the linked list are sorted meaning that the node with the extent of smaller (x) coordinate should be pointed out earlier than the one with the larger coordinate. The first case means that no spans in the linked list overlap with the current span. Seed searching along the row is necessary. This case has four subcases: no spans, the extent of the current span is smaller than the extents of all spans in the linked list, the current extent is larger than all extents and the current extent is in-between two extents of the nodes in the linked list. The second subcase is obtained after processing the first node, while the third subcase is obtained after all nodes were processed. The second case means that the current extent is completely overlapped with some extent in the linked list. Seed searching is therefore not needed. The third case means that the current extent partially overlaps with some extent of a neighboring span. Therefore, seed searching in the overlapped part is not necessary, while a search must be implemented in the remaining

part. The forth case is also a partial overlapped case; however the overlapped part becomes the part with the smaller coordinate. This case must continue to process the consecutive nodes in the linked list instead of directly searching a seed in the remaining part. The fifth case has three parts: one overlapped in-between the remaining two parts. Seed searching should be implemented in the part with the smallest coordinate and consecutive nodes should be processed in the part with the largest coordinate.

However, to obtain a sorted linked list means that the new node must be inserted not just attached to the linked list. In this study, we take the simple way that simply attaches the new node to the linked list without sorting and does not process consecutive nodes if the current span partially overlaps with a neighboring span (as the forth and fifth cases shown in Fig. 5). Such method may be more efficient if extents of spans are not large.

The pseudo code and flowchart of the proposed algorithm are illustrated in Appendix B and C, respectively. Bold and Italic font statements in the pseudo code are the additions to the Feng's algorithm. The former is used to prevent recounting filled voxels. The latter is used to eliminate the redundant seed searching for filled spans. If no voxels filled (the span has already been filled), the seed searching is not necessary. Otherwise, a node with filled extent (xleft, xright) is added to the linked list. Then a seed searching function is implemented. The parameters *y*, *z* represents the index to 2D pointer array, and the xleft (leftmost) and xright (rightmost) indicate the searching extent. The returned value shows the case that the comparison result belongs to.

3 Experiment results

We use six criteria: the number of filled voxels, number of span-referencing, number of

references to voxels, maximum amount of memory (by stack or linked list), number of spans and execution time to compare the proposed, Feng's and Oikarinen's algorithms.

Example I, as shown in Fig. 6, is a teapot consisted of 256×256×32 voxels (by 3D Studio MAX). Example II, as shown in Fig. 7, is a flowerpot with smaller amount of spans in a row than example I. Example III has different span direction with Example II. Example IV (128×128×32) is a scaled-down of Example II. Example V, as shown in Fig. 8, is a medical measured volume (256×256 resolutions in 45 slices).

Table 1 illustrates the implementation results under a Pentium II-400 with 128M RAM. The results show that our algorithm is accurate on the calculation of the number of filled voxels (the same as the Oikarinen's algorithm) meaning that our statements in preventing multiple counts on filled voxels.

The voxel referencing for seed searching has 47.1~49.0 % reduction by the proposed algorithm. The ideal reduction should be about 50% if all consecutive nodes of a linked list are processed in the forth and fifth cases (shown in Fig.5). The simple way has nearly the same reduction with the ideal way meaning that processing one overlapped span is enough to eliminate redundant seed-searching.

A disadvantage of our method is the memory consumption required for nodes (span) and pointers. The latter is proportional to the product of y and z ranges of the volume. The former depends on the complexity of objects in the volume. One span contains two integers and one pointer, and has the same size as a seed in the stack. A complex object contains more spans than a simple one under the same size (with the same voxels) that its spans have a small number of voxels. For example, the medical object shown in Example V has averagely about 19 voxels in a span. The simple object shown in Fig. 1 has averagely 112 voxels in a span. Besides the linked lists, our method must have a stack same size as that in the Feng's algorithm. Meanwhile, the Oikarinen's algorithm does not consume any extra memory for the

seed information.

However, the Oikarinen's algorithm has the longest execution time without the help of seeds to start new fills. In the Feng's algorithm, the execution time relates to number of the referencing voxels (for seed searching) and the complexity of the object. The execution increases not just linearly as the number of referencing voxels increases (comparing to Example II and IV). Therefore, no matter natural or artificial objects, our method can save more than half execution time even when only near half referencing voxels are reduced. For example, if the fifth case is not implemented the voxel referencing for searching seeds becomes 29.8~41.7 % reduction and the execution time increases. Example I has the most referencing voxels reduced by the fifth case (about 38% among the five cases). Therefore, the execution time increases two folds without the fifth case (0.17 to 0.07 second).

4 Conclusions

We discussed the seed filling problems and proposed algorithms to prevent re-counting filled voxels and searching for seeds in neighboring filled span. The recounting prevention is easy to achieve by checking whether the popped seed has already filled. However, the redundant seed searching for filled spans has to be completed by the help of an auxiliary structure, a 2 1/2 D linked lists to record all filled spans. All examples demonstrated the efficiency can be improved using our method.

For minimizing the extent of seed searching in the neighboring spans, we classified 5 cases of comparisons in which some cases can iterate comparing the extent of the current filling span with the extents of neighboring spans. However, we took a simple fashion in which iteration and linked lists are not sorted. Whether the iteration or the sorted linked lists

bring more efficiency or not can be furthermore studied. Meanwhile our experiments show the position of the initial seed and the implementation orders of the five cases have little or no effects in all criteria.

The sorted linked lists may have other benefits: manipulating objects (Shareef and Yagel 1995) etc. The span information stored in unsorted or sorted linked lists can also be used to refilling objects without searching and stacking seeds again. These potential benefits will be outlined in our future works.

Acknowledgement: This work was supported by National Science Council (NSC), Taiwan/ROC; grant numbers NSC-90-2213-E-033-040.

References

1. Albert TA, Slaaf DW (1995) A rapid regional filling technique for complex binary images. Comput. & Graph 19(4):541-549
2. Freund J (1997) Accelerated volume rendering using homogeneous encoding. Proceedings of Visualization '97, 191-196
3. Feng L, Soon SH (1998) An Effective 3D seed fill algorithm. Comput. & Graph 22(5):641-644
4. Fishkin KP, Barsky BA (1985) An Analysis and Algorithm for Filling Propagation. Graphics Interface Proceedings, pp 203-212.
5. González E, Suárez A, Moreno C, Artigue F (1996) Complementary regions: a surface

filling algorithm. IEEE international Conference on Robotics and Automation 1:909-914

6. Hojjatoleslami SA, Kittler J (1998) Region growing: a new approach. IEEE Trans on Image Processing 7(7):1079-1084

7. Levoy M (1990) Efficient ray tracing of volume data. ACM Trans on Graph 9(3):245-261

8. Liu S, Ma W (1999) Seed-growing segmentation of 3-D surface from CT-contour data. Computer-Aided Design 31(8):517-536

9. Oikarinen J (1998) Using 2- and 21/2 dimensional seed filling in view lattice to accelerate volumetric rendering. Comput. & Graph 22(6):745-757

10. Oikarinen JT, Jyrkinen LJ (1998) Maximum intensity projection by 3-dimensional seed filling in view lattice. Computer Networks and ISDN systems (30):2003-2014.

11. Suárez A, González E, Cabo JC, Y. Rollot Y, Manuel B, Moreno C, Artigue F (1995) An autonomous vehicle for surface filling. Proceedings of the Intelligent Vehicles '95 Symposium, pp 364-369

12. Shneiderman B (1992) Tree visualization with tree-maps: 2-d space-filling approach. ACM Trans on Graph 11(1):92-99

13. Shareef N, Yagel R. (1995) Rapid previewing via volume-based solid modeling. ACM Solid Modeling'95, pp 285-291

14. Tsai MD, Jou SB, Hsieh MS (2001) Accurate surface voxelization for manipulating volumetric surfaces and solids with application in simulating musculoskeletal surgery. The Ninth Pacific Conference on Computer Graphics and Applications, IEEE CS Press, pp 234-243

15. Tsai MD, Hsieh MS, Jou SB (2001a) Virtual reality orthopedic surgery simulator. Com-

put. Biol. Med. 31(3): 333-351.

16. Udupa JK, Odhner D (1991) Fast visualization, manipulation and analysis of binary volumetric objects. IEEE Comput Graph & Appl 11(6):53-63

17. Yu YW, Wang JH(1999) Image segmentation based on region growing and edge detection. IEEE International Conference on Systems, Man, and Cybernetics, IEEE SMC '99 Conference Proceedings, pp 798-803

18. Zhou Y, Toga AW (2000) Turning unorganized points into contours. The Eighth Pacific Conference on Computer Graphics and Applications, IEEE CS Press, pp 243-252

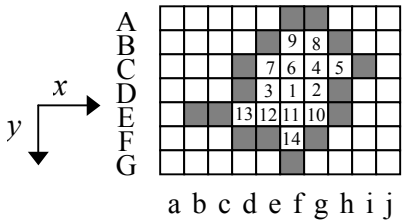


Fig. 1 An example of Feng's algorithm shown in 2D for clarity. *x* and -*x* -direction are filling directions. 1~14 represents the filling sequence.

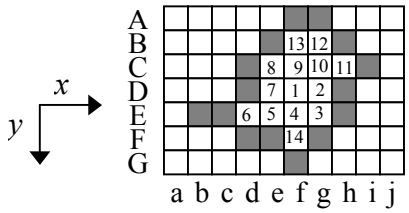
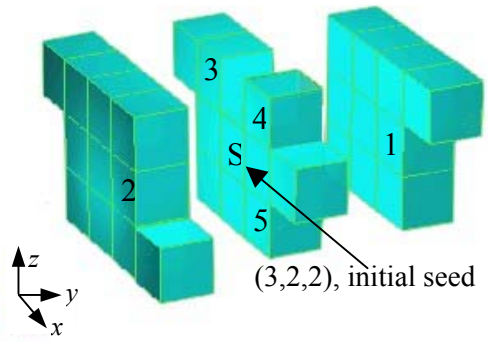
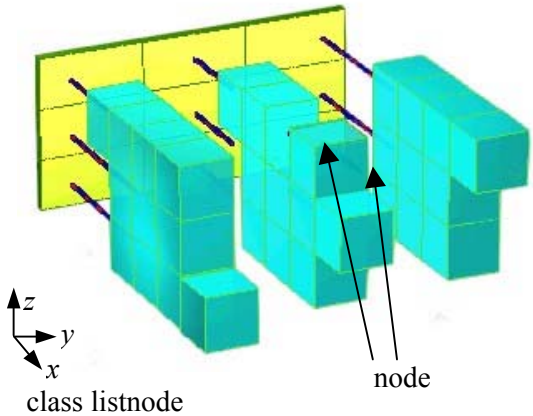


Fig. 2 An example of Oikarinen's algorithm. *x*, -*x*, *y* and -*y* -direction are filling orders. 1~14 represents the filling sequence.



		(4,1,1)	**(4,1,2)**
	(4,2,1)	(4,3,1)	(4,3,1)
	(4,2,3)	(4,2,3)	(4,2,3)
	(2,2,3)	(2,2,3)	(2,2,3)
	(4,1,2)	(4,1,2)	**(4,1,2)**
(3,2,2)	(4,3,2)	(4,3,2)	(4,3,2)

Fig.3. An example shows occurrence of redundant seeds in Feng's algorithm. Each cubic is a voxel inside a closed boundary.




```
class listnode
    { public:
     int xleft,xright;    /extent of a span
    listnode *next;
    }*listhead[y][z];
```

Fig. 4. Use of *y***z* linked lists storing filled spans

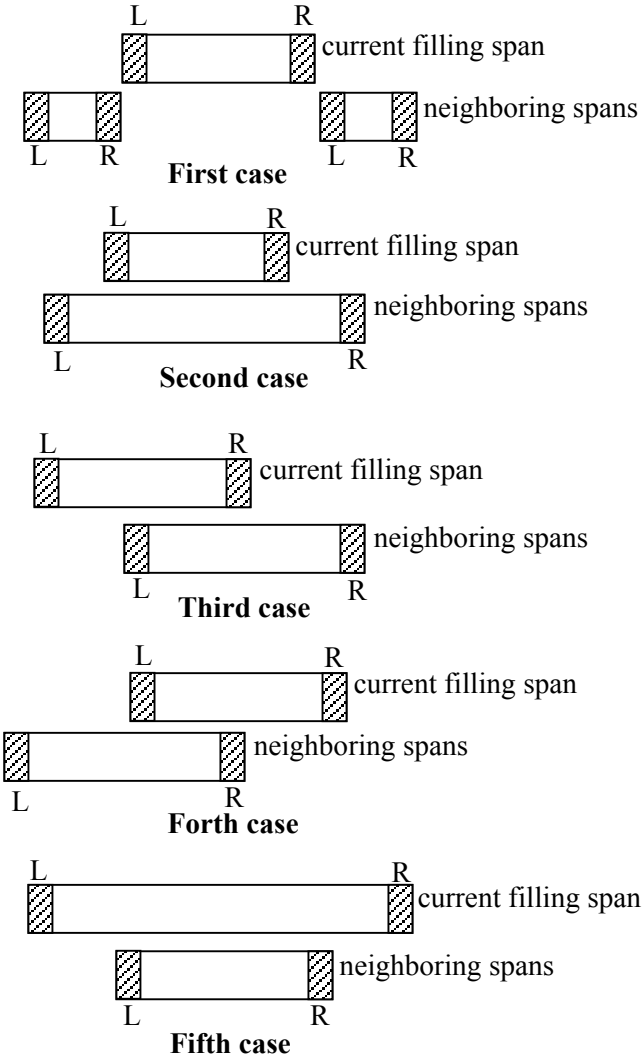


Fig.5 Span comparison
(L:Leftmost R:rightmost)

Fig. 6 A teapot (example I)

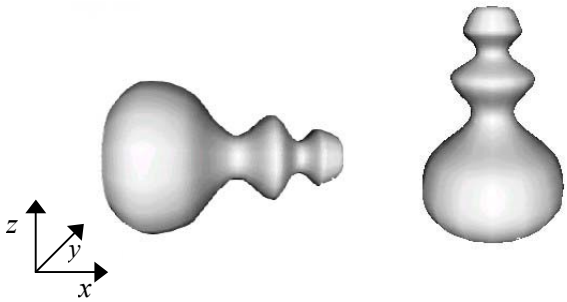


Fig. 7 A flowerpot (example II, III and IV).

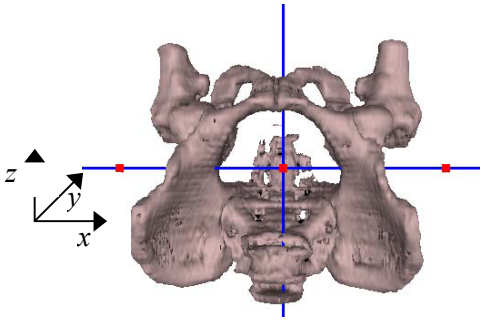


Fig.8 A rendered pelvis (example V).

Table 1 A comparison of Feng's, proposed and Oikarinen's algorithm for examples I-V.

Example	algorithms	number of filled voxels	number of span-referencing	number of references to voxels	maximum memory(by stack or linked list)	maxmum number of stack or number of spans	execution time(sec)
	Feng's	381223	*	5993344	28608	2384	0.16
Teapot	Proposed	377979	25568	3128512	40572	3381	0.07
(256×256×32)				**4205688**			**0.17**
	Oikarinen's	377979	*	*	*	*	0.39
	Feng's	229932	*	3556400	23544	1962	0.11
Flowerpot	Proposed	226225	28523	1815064	47256	3938	0.05
(horizontal)				**2072340**			**0.11**
(256×256×32)	Oikarinen's	226225	*	*	*	*	0.22
	Feng's	227898	*	3545760	26412	2201	0.11
Flowerpot	Proposed	224876	19135	1821704	38760	3230	0.05
(vertical)				**2239952**			**0.05**
(256×256×32)	Oikarinen's	224876	*	*	*	*	0.22
	Feng's	55626	*	831296	11832	986	0.00
Flowerpot	Proposed	53863	13483	431164	22668	1889	0.00
(horizontal)				**477188**			**0.00**
(128×128×32)	Oikarinen's	53863	*	*	*	*	0.05
	Feng's	155077	*	2320768	48204	4017	0.11
Pelvis	Proposed	150267	44058	1227624	61788	5149	0.05
				1480008			**0.08**
(256×256×54)	Oikarinen's	150267	*	*	*	*	0.17

Bold entries show the number of referencing voxels or execution time if the fifth case is omitted. The execution is too quick in Example IV to measure correctly.

Appendix A

Pseudo code of 3D filling algorithm without stack

```
struct DIRECTION
{ int x; int y; int z;} direct[6]={{1,0,0},{-1,0,0}, {0,1,0},{0,-1,0},{0,0,1},{0,0,-1}};
        // define the filling order: +x,-x,+y,-y,+z,-z
make direction of initial seed to -1;
void 3D_fill_Nostack(int i ,j, k)
{ while(I<6) //examine 6 face-neighboring voxels
    {i+=direct[I].x;
    j+=direct[I].y;
    k+=direct[I].z;
    if(current voxel is not boundary nor filled)
        {fill the voxel;
        get the direction from I; // record the direction for rollback
        I=0; //to get first neighboring for next voxel
        }
    else
        I++; // increment I to get the next voxel to be filled
    }
record the directions to I
if(I<0)     return; // initial seed is encountered
move back to previous voxel
I++;
}
```

Appendix B

Pseudo code of the proposed and Feng's 3D seed fill algorithm

```
create an empty stack;
push seed voxel to stack;
while(stack is not empty)
{
      pop a seed to x,y,z
      span_filled_flag =0;
      if(voxel[x][y][z is not filled)
      {     fill voxel[x][y][z];
            numfilled++;
            span_filled_flag =1;
      }
      fill this span.
      length=xright-xleft+1;
      if (span_filled_flag!=0)
      {     add_list(y,z,xleft,xright);
            y++;
            if (check_list(y,z,xleft,xright)==1)    //checkseed in full interval;
            else if (check_list(y,z,xleft,xright)==2) // no need for checkseed;
            else if (check_list(y,z,xleft,xright)==3)    //checkseed in left interval;
            else if (check_list(y,z,xleft,xright)==4 )//checkseed in right interval
            else if (check_list(y,z,xleft,xright)==5) // checkseed in full interval
            // do checkseed() in –y/z/-z neighboring interval
      }
}
void add_list(int y,int z,int xleft,int xright)
{
      head=pointer pointing head of linked-list [y][z];
      while(head ->next!=NULL)
            head=head->next;
      create a node with xleft, xright
      head->next=t;
}
check_list(int y,int z,int xleft,int xright)
{     t=pointer pointing to linked list[y][z];
      while(t !=NULL)
      {       if (t->xleft<=xleft && t->xright>=xright) {return(2);} //a node cover all interval
            else if (t->xleft>xleft    && t->xleft<=xright&& t->xright>=xright){ return(3);} //a
node cover right interval
            else if (t->xleft<=xleft && t->xright>xleft && t->xright<xright){ return(4);} //a
node cover left interval
            else if (t->xleft>xleft&&t->xright<xright) {return(5); } //a node inside the interval
            t=t->next;
      }
return(1);}
}
checkseed(xleft,,xright)
{     x=xleft;
      while(x<=xright)
      {
      if(voxels are non-boundary, unfilled to boundary or non-boundary, unfilled to filled)
            add a seed to stack
      x++;
      }
      check the last voxel
}
```

Appendix C
Flowchart of the proposed algorithm