Testing a Binary Space Partitioning Algorithm with Metamorphic Testing

Fei-Ching Kuo*, Shuang Liu and T. Y. Chen
Centre for Software Analysis and Testing
Swinburne University of Technology
Hawthorn, Victoria 3122, Australia

ABSTRACT

Surface visibility problem is a well known problem in computer graphics. The problem is to decide the visibility of an object surface to a viewer in a space. *Binary Space Partitioning tree* (BSP tree) has been proposed to enable efficient determination of surface visibility. The BSP tree has also been used to solve many problems, such as collision detection, ray tracing and solid modelling. As a consequence of the extensive applications of the BSP tree, it is important to ensure the correctness of its implementation. In this paper, we report our experience of testing software that deals with the surface visibility using the BSP tree. We demonstrate how we analysed the software behaviour, selected test cases to reveal a real-life software failure, and debugged the program to fix the fault. This study provides insight into testing of other BSP-tree related algorithms.

Keywords: Software testing, test oracle, metamorphic testing, binary space partitioning, surface visibility problem

1. INTRODUCTION

Surface visibility problem is a well known problem in computer graphics. The problem is to decide the visibility of an object surface to a viewer in a space. Having two surfaces p_i and p_j in front, the viewer may be able to see an entire p_i, but not p_j. This occurs when p_i is sitting between the viewer and p_j, and partially blocks the viewer's vision of p_j. To the viewer, p_i is the *"front" surface* while p_j is the *"back" surface* and these surfaces partially *overlap* with each other. Deciding visibility of surfaces (p_i and p_j) requires determination of each surface's position (front or back) with respect to the viewer.

Binary Space Partitioning (BSP) is a technique to decompose the space using a set of planes [6]. It produces a binary tree called "BSP tree". In many cases, surfaces do not change their position in the space, but the viewer does. Therefore, Fuchs et al. [6] proposed the BSP technique to partition the space using the plane of surfaces, organize these surfaces into a BSP tree, and let the viewer move along the tree to determine the position of each surface (front or back). The proposed algorithm is named *BSP tree visible surface* (abbreviated as "*BSP-treeVS*") algorithm.

The BSP tree has also been used to solve many problems, such as collision detection [12], ray tracing [8, 14], and solid modelling [12]. Owing to the extensive applications of the BSP tree, it is important to ensure the correctness of its implementation.

In this paper, we report our experience of testing the BSP-treeVS software. We found that testing the BSP-treeVS software suffers from a well known testing problem, called the "oracle problem". The oracle problem occurs when it is infeasible to obtain a mechanism for verifying the software output correctness (known as the "test oracle") for a test. Without the test oracle, a test's outcome ("pass" or "failure") cannot be concluded, and thus the test becomes useless. Our study mainly targets at the oracle problem of testing the BSP-treeVS software. It would provide insight into testing of other BSP-tree related software (such as software using the BSP tree for collision detection).

The rest of the paper is organized as follows: Section 2 gives some background information about the BSP tree and BSP-treeVS algorithm as well as our testing subject. Section 3 shows the technique that we used to tackle the oracle problem, and how we derived the necessary software properties for applying the chosen technique. Section 4 discusses our test settings and findings. Section 5 concludes the paper and presents our future work.

2. PRELIMINARY

2.1 BSP tree

BSP tree is a binary tree. Its structure is in the form $T = \{N, A, r\}$, where N is a set of *nodes*, A is a set of *parent-and-child relation* between two nodes in N, r is a special node in N, called "*root*" that has no parent but at most 2 children. Any nodes other than r must have one and only one parent, and at most 2 children. Nodes that share the same parent are the "sibling nodes". A node without any children is called a "*leaf*". It is possible that T only has r without any other nodes. In this case, r is also the *leaf*, and A is an empty set.

T can be decomposed into *subtrees*. For example, the *top node* of T (that is, r) splits the tree into two subtrees T_L and T_R, where the left child of r (denoted as r_L) and right child of r (denoted as r_R) are the top node of T_L and T_R,

*Corresponding author: dkuo@groupwise.swin.edu.au

SAC'11 March 21-25, 2011, TaiChung, Taiwan.

respectively. Similarly, T_L and T_R can also be decomposed into subtrees ($T_{L.L}$, $T_{L.R}$, $T_{R.L}$ and $T_{R.R}$). Subtrees that share the same top node are called the "sibling trees".

As the binary tree is used to store data in a certain order, the tree can be traversed in various ways to find or sort data.

2.2 BSP tree visible surface algorithm

The inputs to the BSP-treeVS algorithm include a set of surfaces (P) and the viewer's coordinate in the space. In a 3-dimensional space, each surface p_i has at least 3 vertices as well as a plane whose equation $f_i(X, Y, Z)$ is $a_iX + b_iY + c_iZ + d_i = 0$. Every vertex of p_i (denoted as $p_i.vertex_j$) lies in the plane, as shown in Figure 1.

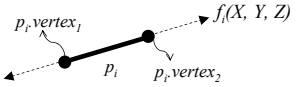


Figure 1: A surface and its plane and vertices

The following outlines the two major steps of the algorithm.

1. Construction of a BSP tree
   Select one surface p_i from P according to certain rules (to be explained later), and use the plane of p_i to partition the space S into two disjoint subspaces (S_L and S_R), where S_L locates on the positive side of p_i (denoted as p_i+) and S_R on the negative side of p_i (denoted as p_i-). Such partitioning will also divide the remaining elements of P into two disjoint sets P_L and P_R, located in S_L and S_R, respectively. Store p_i as the root node r of a BSP tree (T).
   Follow the same rules of selecting p_i to select one element in P_L to decompose S_L, and one element in P_R to decompose S_R. The element selected from P_L will form the top node of the subtree T_L, while that from P_R will form the top node of the subtree T_R. Repeat the same process to decompose the subspace of S_L and that of S_R, and other subspaces generated afterward, until no further decomposition is possible. The resultant tree is the BSP tree.

2. Determination of front and back nodes
   Traverse the entire tree starting from the root node. For each visited node $n \in N$, check the viewer's position with respect to n (either on the positive side or negative side of n, that is, $n+$ or $n-$). Then, determine which subtree of n is *opposite* to the viewer and which subtree is on the same side as the viewer. The subtree's nodes on the *same side* are denoted as frontN(n), while those on the opposite side are denoted as backN(n). The currently visited node n is sitting between frontN(n) and backN(n).
   The tree is traversed such that the back nodes are printed first, followed by the currently visited node, and then the front nodes (called the *back-to-front order* in this paper). Such an output order allows that the front surfaces override the back surfaces if there are any overlaps between these surfaces.

Note that every time a node is selected, the plane of the node is used to partition the associated space into two. Such space partitioning may cut an input surface into "sections" residing in two separate subspaces. Suppose that there are two surfaces p_i and p_j in the space. As shown in Figure 2, if p_i is selected as a tree node to partition the space, its plane could cut p_j into two sections separately residing in two disjoint subspaces (one section in p_i+, and the other one in p_i-). Each of these sections will form a tree node of the associated subtree. Therefore, the size of the BSP tree (that is, number of tree nodes) may be bigger than the size of P (that is, number of input surfaces). The more a surface is cut into sections during the partitioning, the larger the BSP tree will be produced. The tree size is critical to the efficiency of the BSP-treeVS algorithm [5]. Since it may take exponential time to construct the smallest tree, heuristics are normally used to guide the space partitioning.

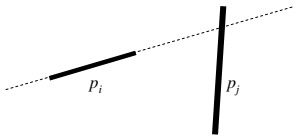


Figure 2: Cutting another surface into sections

2.3 Testing subject

In this project, one program was selected for testing from Section III.5 of a book called "Graphics Gems V" [3]. We will call this program "BSP-treeVS" software hereafter. This software consists of seven modules, where bspTree.c and bspPartition.c are the modules handling the key logic and functions inside the BSP-treeVS algorithm. The viewer's coordinate (0, 5, 10) was hard-coded in the software.

The software is provided in the book with a sample input file "foo.dat". In foo.dat, there are six *input surfaces* {p_1, p_2, p_3, p_4, p_5, p_6}, each of which consists of 4 vertices shown in Table 1. The plane's equation of each surface is given in Table 2. Figure 3 depicts the six surfaces in the space.

We are going to use the input foo.dat to illustrate how software implements the 2 major steps of the BSP-treeVS algorithm in Section 2.2.

Surface	Vertices
p_1	(2.25, 0, -2), (4.25, 0, -2), (4.25, 7, -2), (2.25, 7, -2)
p_2	(4.25, 0, -2), (4.25, 0, -4), (4.25, 7, -4), (4.25, 7, -2)
p_3	(4.25, 0, -4), (2.25, 0, -4), (2.25, 7, -4), (4.25, 7, -4)
p_4	(2.25, 0, -4), (2.25, 0, -2), (2.25, 7, -2), (2.25, 7, -4)
p_5	(2.25, 7, -2), (4.25, 7, -2), (4.25, 7, -4), (2.25, 7, -4)
p_6	(-12, 0, 9), (9, 0, 9), (9, 0, -9), (-12, 0, -9)

Table 1: Vertices of six surfaces in foo.dat

Surface	Equation of the associated plane
p_1	$f_1(X, Y, Z) : 0X + 0Y + 1Z + 2 = 0$
p_2	$f_2(X, Y, Z) : 1X + 0Y + 0Z - 4.25 = 0$
p_3	$f_3(X, Y, Z) : 0X + 0Y - 1Z - 4 = 0$
p_4	$f_4(X, Y, Z) : -1X + 0Y + 0Z + 2.25 = 0$
p_5	$f_5(X, Y, Z) : -0X + 1Y + 0Z - 7 = 0$
p_6	$f_6(X, Y, Z) : -0X + 1Y + 0Z - 0 = 0$

Table 2: Plane's equation for each surface in foo.dat

To minimize the tree size, the software partitions the space by selecting a surface, whose plane would cut the fewest other surfaces. For example, instead of selecting p_i, the software selects p_j to partition the space to avoid cutting other surfaces (such as p_i) into sections, as shown in Figure 4. If there are more than one input surfaces whose plane cuts the fewest other surfaces, the software will select the *first listed* surface in the foo.dat as a node to partition the space.

Consider the surfaces in foo.dat. Planes of p_1, p_2, p_3 and p_4 cut one surface p_6, while planes of p_5 and p_6 cut no surfaces. p_5 is the first listed surface in the input file having the fewest cut surfaces, so it will be selected as the root node and its plane is used to partition the space into two subspaces. The position of the remaining input surfaces will then be evaluated against the plane's equation of r. After the evaluation, these surfaces are grouped into two disjoint sets P_L and P_R to partition the space S_L and S_R, respectively. Take p_6 as an example. $f_5(X, Y, Z)$ is evaluated to -7 for all vertices of p_6, hence p_6 is positioned in p_5-. Based on this, p_6 is in the set P_R and will become some node(s) in T_R. The above process is repeated to select nodes until no further decomposition is possible.

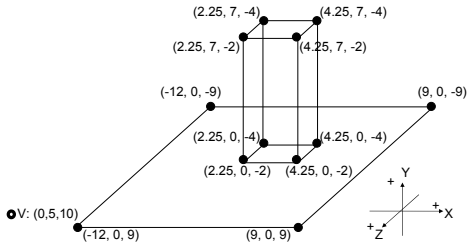


Figure 3: Surfaces for the foo.dat sample input

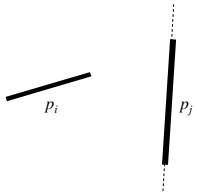


Figure 4: Avoidance of cutting other surfaces into sections

During the tree construction, no input surfaces in foo.dat happen to be cut, thus the BSP tree is comprised of 6 nodes. Table 3 lists the cut surfaces associated with each node selection. Figure 5(a) shows the BSP tree, where each node is positioned. In this figure, the parent-and-child relation marked by '+' denotes that the child node is located on the positive side of the parent node, while that marked by '-' denotes that the child is on the negative side of the parent.

Table 4 shows how the viewer's position with respect to each node is determined. Figure 5(b) includes the viewer's position in the BSP tree shown in Figure 5(a). Based on the viewer's position, we can identify the front nodes and back nodes for each visited node. For example, the viewer V=$(0, 5, 10)$ is located on the negative side of the root node (that is, p_5-), so frontN(p_5) and backN(p_5) are associated with the subtree in p_5- and p_5+, respectively. In this case, frontN(p_5) contains {p_6, p_1, p_2, p_3, p_4} but backN(p_5) is empty. Figure 5(c) shows which subtree contains the front/back nodes with respect to each visited node.

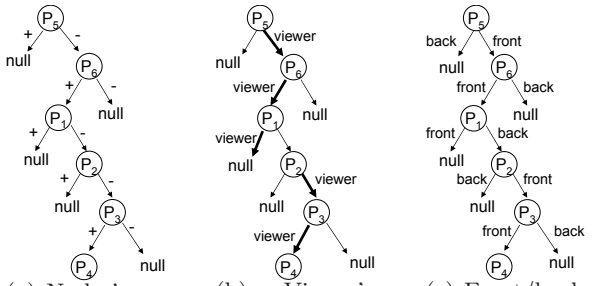
The BSP-treeVS software is designed to traverse the BSP tree such that the back nodes are printed first, followed by the currently visited node, and then the front nodes. Take the tree in Figure 5(c) as an example. First, the software visits the root node p_5. Since p_5 does not have the back nodes but front nodes, p_5 is printed first. The software then moves to the top node of the subtree that contains the frontN(p_5). Neither does this top node (that is, p_6) have the back nodes, so p_6 is printed after p_5. On the other hand, p_1 has some back nodes (that is, p_2, p_3 and p_4) but no front nodes, so these back nodes will be printed after p_6. After printing the back nodes, p_1 will be printed. By applying this traversal rule to the entire tree, the software will print the nodes in the following order: $p_5 \rightarrow p_6 \rightarrow p_2 \rightarrow p_3 \rightarrow p_4 \rightarrow p_1$.

i =	1	2	3	4	5	6	
Plane of p_1	cut p6	cut p6	cut nil	select p_1 as the top node of $T_{R.L}$			
Plane of p_2	cut p6	cut p6	cut nil	cut nil	select p_2 as the top node of $T_{R.L.R}$		
Plane of p_3	cut p6	cut p6	cut nil	cut nil	cut nil	select p_3 as the top node of $T_{R.L.R.R}$	
Plane of p_4	cut p6	cut p6	cut nil	cut nil	cut nil	cut nil	select p_4 as the top node of $T_{R.L.R.R.L}$
Plane of p_5	cut nil	select p_5 as the root node					
Plane of p_6	cut nil	cut nil	select p_6 as the top node of T_R				

Table 3: Cut surfaces for the i^{th} node selection

Node	Outcome of plane equation of each node	Viewer's position against each node
p_1	$f_1(0, 5, 10) = 10 + 2 > 0$	p_1+
p_2	$f_2(0, 5, 10) = -4.25 < 0$	p_2-
p_3	$f_3(0, 5, 10) = -1 \times 10 - 4 < 0$	p_3-
p_4	$f_4(0, 5, 10) = 2.25 > 0$	p_4+
p_5	$f_5(0, 5, 10) = 1 \times 5 - 7 < 0$	p_5-
p_6	$f_6(0, 5, 10) = 1 \times 5 - 0 > 0$	p_6+

Table 4: Evaluating each node's plane equation using the viewer's coordinate (0, 5, 10)



(a) Nodes' position in the BSP tree

(b) Viewer's position in respect of each visited node

(c) Front/back subtrees in respect of each visited node

Figure 5: BSP-treeVS software execution by foo.dat

The above scenario is not yet complex. We only deal with 6 input surfaces parallel to the X-Y, X-Z or Y-Z plane. None of these surfaces are cut into sections during the space partitioning. Given such a simple scenario, it is already difficult to know the expected output. When we have more input surfaces of more complex orientation and more cutting of surfaces, it becomes more challenging to know the expected output for a single test (known as the oracle problem).

3. TACKLING THE ORACLE PROBLEM

Metamorphic testing (MT) was proposed to provide an alternative test oracle for verifying the software output correctness [2]. Instead of checking output correctness for one test execution, MT checks outputs of multiple executions against some necessary software properties (a form of multiple-inputs-and-outputs relations, known as metamorphic relations (MRs)). MT has been used to test various types of software, which has the oracle problem [7, 9, 11, 10, 13, 15].

Consider testing a program that computes the cosine function. By executing the cosine function with the input 59.3, suppose that the software returns 0.5105. It is not obvious whether this test output is correct. A property of the cosine function "$\cos(x) = -\cos(180 - x)$" can be used as a MR. By executing the function again with another input $(180 - 59.3) = 120.7$, if the second output cos(120.7) is -0.5105, then the test satisfies the defined MR. If the two test results do not comply with the MR, a software "failure" is detected, and debugging is required to fix the fault. The first test using cos(59.3) is called the *source test*, and the second using cos(120.7) is called the *follow-up test*.

As shown in the above example, a MR defines the condition of the follow-up test input, and the relation between the source test output and follow test output. Sometimes, a MR may impose some constraints on its source test input, and hence only test inputs that satisfy the constraints are considered "valid" for applying the MR.

To define MRs, we need to know what the software inputs and outputs are. The inputs to the BSP-treeVS software are a list of input surfaces P and the outputs are a list of tree nodes printed in the back-to-front order. From the output, we can see the relative position of nodes (front or back) with respect to the viewer. As explained before, input surfaces may be cut into sections during the space partitioning. Each section forms an individual node in the BSP tree. There is no one-to-one mapping between the surfaces in P and nodes in the tree T. However, given a node in the output, we can easily find the input surface where the node comes from. We will apply the above knowledge to derive MRs for testing the BSP-treeVS software.

In the following subsections, we will describe various properties of the algorithm and related MRs. Hereafter, we will use $V{=}(v_x, v_y, v_z)$ to denote the viewer's coordinate.

3.1 First property

As long as the BSP tree structure is not changed and the coordinate of the viewer remains the same, the output of the software will remain unchanged.

If we uniformly scale every input surface p_i of P with a scaling factor k to obtain a corresponding surface p', the relative position amongst all resultant surfaces will not change, and hence the tree structure will not change. Equation 1 shows how to obtain each vertex of each resultant surface.

$$p'.vertex_j = (p.vertex_j - V) \times k + V \quad (1)$$

This equation is equivalent to the following equation:

$$\begin{pmatrix} p'.vertex_j.x \\ p'.vertex_j.y \\ p'.vertex_j.z \end{pmatrix} = \begin{pmatrix} p.vertex_j.x - v_x \\ p.vertex_j.y - v_y \\ p.vertex_j.z - v_z \end{pmatrix} \times k + \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$$

Given a surface p in Figure 6, Equation 1 will create another surface p' in Figure 7.

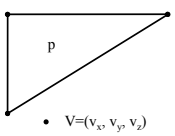


Figure 6: One surface p

We therefore can have the first MR as follows:
MR1: Given any list of surfaces as the source test input (P), the follow-up test input P' is generated by uniformly scaling all the surfaces of P with the scaling factor $k = 2$. The source and follow-up tests are expected to produce the same output, provided that the viewer's coordinate is unchanged (as a reminder, the viewer's coordinate is hard-coded in the BSP-treeVS software, so this setting will definitely remain the same throughout our testing.).

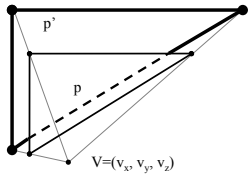


Figure 7: Scaling p to p'

3.2 Second property

As shown in Section 2.3, if no input surfaces were cut during the space decomposition, each surface will only associate with one node in the tree. As a consequence, we can predict the software output based on each surface's position in the space. How can we do so if the two surfaces have a clear front and back relation (such as p' and p in Figure 7, where p' is the back surface and p is the front to the viewer) but they are cut by the plane of the other surface?

Consider a case, where the space consists of p_1, p_2, p_3 and other input surfaces. After space partitioning and tree construction, p_3 is not cut and selected as a node before p_1 and p_2, and both p_1 and p_2 are only cut by the plane of p_3 (as shown in Figure 8).

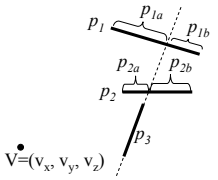


Figure 8: Viewer's position in the space after cutting p_1 and p_2 by the plane of p_3

In this case, p_3 will be the top node of two subtrees, where the sections of p_1 and p_2 separately reside. As previously mentioned, the space contains other input surfaces that are undisclosed in Figure 8, hence there could be other nodes sitting between p_3, p_{1a}, p_{2a}, p_{1b} and p_{2b}. Despite that we do not know what other nodes exist in the tree, the sections p_{1a}, p_{1b}, p_{2a} and p_{2b} could only be selected as a tree node in the following four orders:

1. p_{1a} is selected before p_{2a}, while p_{1b} is selected before p_{2b} as a node
2. p_{2a} is selected before p_{1a}, while p_{2b} is selected before p_{1b} as a node
3. p_{1a} is selected before p_{2a}, while p_{2b} is selected before p_{1b} as a node
4. p_{2a} is selected before p_{1a}, while p_{1b} is selected before p_{2b} as a node

These orders of node selection will produce four types of the tree structures under p_3, as shown in Figure 9, where dashed

lines are used to mean possible occurrence of other nodes in the tree. Given Figure 9 and the viewer's coordinate in Figure 8, we can determine the front and back nodes with respect to each visited node, as shown in Figure 10. Regardless of which type of the tree the software will finally produce, the software's traversal rule will always print p_{1b} before p_{2b}, followed by p_3, p_{1a} and p_{2a}.

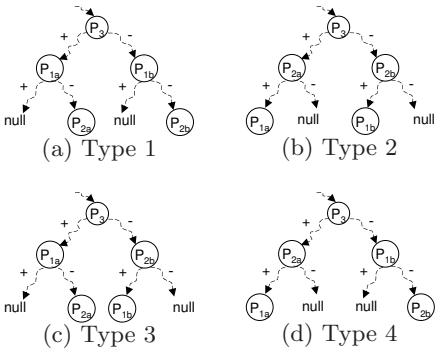


Figure 9: Possible tree structures under p_3

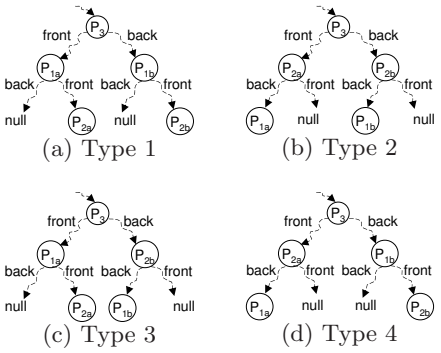


Figure 10: Front and back subtrees in respect of each visited node

The above analysis tells us that if the surfaces p' and p in Figure 7 are the surfaces p_1 and p_2 studied here (so p_2 fully blocks the viewer's vision of p_1), then regardless of whether p_1 and p_2 would be cut into sections (note: when a surface is not cut, the surface is the whole section itself), at least one section of p_1 will be printed out before all sections of p_2. We therefore have the second MR as follows:

MR2: Randomly select one surface p_q from the source test input P, such that $f_q(v_x, v_y, v_z) \neq 0$ (so the viewer is not a point in p_q). Construct the follow-up test input P' by appending one additional surface p' to P, where p' is constructed from p_q using Equation 1 and $k = 1.5$. The follow-up test is expected to output at least one section of p' before all sections of p_q.

3.3 Third property

The software output depends on the front and back relations between nodes (describing which nodes are the front/back nodes for a visited node). As long as the back and front relations remain the same, the output will be unchanged.

Swapping the position of all sibling nodes (as Figure 11(a)) and at the same time, totally reversing the viewer's position in the tree (as Figure 11(b)) will not change the front and back relations between nodes (as shown in Figure 11(c)), and consequently have no effect on the output. Such swapping and reversing can be achieved by projecting all input surfaces to the other side of the viewer. Equation 2 shows how to obtain each vertex of each resultant surface.

$$p'.vertex_j = -p.vertex_j + 2V \tag{2}$$

Given a surface p in Figure 6, Equation 2 will create another surface p' in Figure 12.

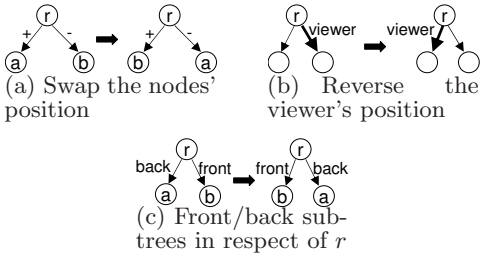


Figure 11: Effect of swapping the nodes and reversing the viewer's position in the tree

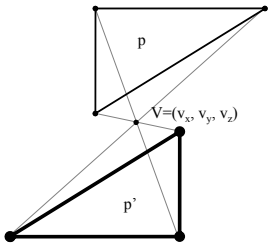


Figure 12: Projecting p to the other side of V

We therefore can have the third MR as follows:

MR3: Given a source test input P, where no input surfaces contain V as a point, construct the follow-up test input P' by projecting each surface p of P to the other side of V. The source and follow-up tests are expected to produce the same output.

3.4 Fourth property

The left subtree and right subtree of the root node r are processed separately with the same traversal rule (that is, back-to-front order), so as long as the viewer's coordinate and root node both remain the same, changing the structure of one subtree will only have a regional impact on the output (that is, only the nodes related to the changed subtree will be affected).

Consider appending one surface p_7 to foo.dat, such that p_7 is behind p_5 in Figure 3 and not cut by the plane of p_5. Such a change will still keep p_5 as the root node, and add p_7 to the backN(p_5) in Figure 5(c). Consequently, the software will print p_7 before p_5, and retain the same sequence of nodes after p_5. The initial foo.dat can be considered as the source test input P, while the modified foo.dat can be considered as the follow-up test input P' for the fourth property.

We are going to discuss the fourth and fifth MRs used in this study. These two MRs require the source test input P to be constructed as follows:

1. $P = \{p_1\} \bigcup P_I \bigcup P_{II}$
2. The surface p_1 is the first listed surface in P and all its vertices have the x-coordinate $= t$, such that $t \neq v_x$ (so p_1 is in parallel to the Y-Z plane, and the viewer is not a point in p_1)
3. For each surface in P_I, all its vertices have the x-coordinate $< t$

4. For each surface in P_{II}, all its vertices have the x-coordinate $> t$

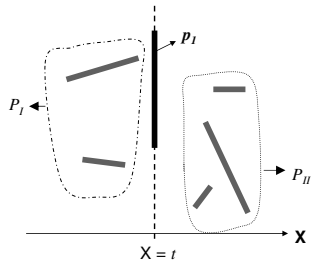
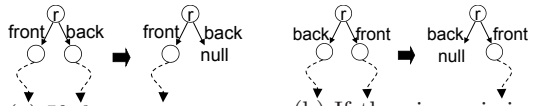


Figure 13: Source test input P for MR4 and MR5

Figure 13 depicts how the elements of P are placed in the space with respect to X = t. Since p_1 is the first listed surface that cuts the fewest other surfaces, p_1 will be the root node, and P_I and P_{II} will separately form two distinct subtrees. If the viewer is on the same side as P_I with respect to p_1, then the subtree related to P_I will form frontN(p_1), and that related to P_{II} will form backN(p_1).

Running the software with the input P will output the tree nodes in the order: backN(p_1) $\rightarrow$ p_1 $\rightarrow$ frontN(p_1). Given a node n in backN(p_1) (any output node before p_1), we can easily deduce the viewer's position in the tree, as well as the set $P_{front} \subset P$ that constitutes the frontN(p_1), and the set $P_{back} \subset P$ that constitutes the backN(p_1). For example, if the vertices of n have the x-coordinate $> t$, we know that the viewer's x-coordinate (that is, v_x) will be $< t$, $P_{front} = P_I$ and $P_{back} = P_{\text{II}}$.

MR4: Use the above P as the source test input to run the software. Construct the follow-up test input (P') by having p_1 as its first listed surface, followed by P_{front}. This setting is to prune the backN(p_1) in the BSP tree of the follow-up test, as shown in Figure 14. Consequently, the follow-up test output will exclude backN(p_1), but retain the same output sequence for p_1 and frontN(p_1).



(a) If the viewer is in the left subtree of r

(b) If the viewer is in the right subtree of r

Figure 14: Prune the backN(r) in the BSP tree of the follow-up test

MR5: Use the above P as the source test input to run the software. Construct the follow-up test input P' by appending another surface p' to P, where p' is on the same side as P_{back} with respect to p_1 (so p' or its sections will be part of backN(p_1)). This setting is to increase the size of backN(p_1) in the BSP tree of the follow-up test. The source and follow-up tests are expected to print the same nodes in the same order after p_1 (because the change made in P' should not affect the frontN(p_1)), whereas they may print different nodes in different orders before p_1.

4. TEST SETTINGS AND FINDINGS

Testing of the BSP-treeVS software suffers from the oracle problem. We applied 5 MRs defined in Section 3 to verify the test output correctness of the software.

For each MR, 1000 "valid" source test inputs were randomly generated from the following input domain:

- Each test input has at least 2, but at most 7 surfaces.
- Each surface has three vertices, and all these vertices are within the domain (X, Y, Z) where $-50 <$ X < 50, $-50 <$ Y < 50 and $-50 <$ Z < 50.

Each of these source test inputs was then used by the corresponding MR to generate a follow-up test input. Consequently, for each MR, 1000 pairs of inputs (source and follow-up) were used to test the software.

For each pair of source and follow-up tests, we classified their results into three categories: crash, failure and pass. The crash refers to the situation where software terminated abnormally without completing test execution in either source, follow-up or both tests. The failure refers to the situation where the software completed both source and follow-up test executions, but their results violate the MR. The pass refers to the situation where both source and follow-up tests were completely executed, and their results complied the MR. For each MR and each mutant, the sum of the number of crashes, failures and passes will be 1000.

When we tested the BSP-treeVS software with the above settings, no failures were detected, but 25 crashes. All these crashes were detected by the follow-up test inputs of MR1. The crash and failure results for each MR on the BSP-treeVS software are summarized in the 3rd row of Table 5. After a careful examination of the code, we found that the crash was caused by a fault inside the software. The fault was related to the incorrect implementation of the BSP tree construction (first step in the BSP-treeVS algorithm, as described in Section 2.2). After locating this fault, we debugged the program. Table 7 summarizes the corrections that we made in the software. To see whether we indeed corrected the fault in the original program, we repeated the previous tests on the amended version of BSP-treeVS software, and found that the amended version passed all the tests (no crashes and no failures, as shown in the 4th row of Table 5). Our testing experience shows that metamorphic testing is good in generating effective follow-up test cases to detect a real life software fault. It enables us to debug the software and improve its correctness.

To further evaluate the effectiveness of our MRs in failure detection, we randomly seeded one fault into the amended version of BSP-treeVS software (either bspPartition.c or bspTree.c module) to create 10 faulty versions. This is a well-known technique in software testing, called "mutation analysis" that allows effectiveness measurement of a testing technique [1, 4]. Each faulty version is called "mutant". The more mutants a testing technique can kill, the more effective the technique would be. Table 8 summarizes the faults that we seeded into the amended version of BSP-treeVS software.

After that, we repeated all tests on 10 mutants. Table 6 summarizes the number of crashes and failures detected by our MRs. For each mutant, if the number of crashes or that of failures being detected is greater than 0, it means that the mutant has been killed. In all our tests, no crashes were detected. With respect to the failure detection, we found that MR3 had the highest effectiveness in detecting failures, followed by MR2, MR4, MR5 and MR1. Furthermore, MR2 killed the largest number of mutants, followed by MR1, MR3, MR4 and MR5. The study shows that different faults are sensitive to different MRs. The MRs defined in this study can collectively kill all the mutants.

	MR1		MR2		MR3		MR4		MR5	
	F	**C**	**F**	**C**	**F**	**C**	**F**	**C**	**F**	**C**
Original	0	25	0	0	0	0	0	0	0	0
Amended	0	0	0	0	0	0	0	0	0	0

Table 5: Test results for the original and amended software version (F denotes "failure" while C denotes "crash")

	MR1		MR2		MR3		MR4		MR5	
	F	**C**	**F**	**C**	**F**	**C**	**F**	**C**	**F**	**C**
$\mu 1$	0	0	497	0	905	0	0	0	0	0
$\mu 2$	8	0	2	0	2	0	0	0	0	0
$\mu 3$	7	0	3	0	3	0	0	0	0	0
$\mu 4$	5	0	3	0	0	0	456	0	346	0
$\mu 5$	534	0	2	0	744	0	0	0	0	0
$\mu 6$	0	0	170	0	990	0	0	0	0	0
$\mu 7$	4	0	0	0	0	0	0	0	0	0
$\mu 8$	1	0	3	0	0	0	0	0	0	0
$\mu 9$	182	0	4	0	429	0	14	0	0	0
$\mu 10$	1	0	491	0	905	0	408	0	496	0
total	742	0	1175	0	3978	0	878	0	842	0

Table 6: Test results for the 10 mutants (μ) (F denotes "failure" while C denotes "crash")

5. CONCLUSION

Surface visibility problem is a well known problem in computer graphics. *Binary Space Partitioning tree* (BSP tree) has been proposed to enable efficient determination of surface visibility. It has also been used to solve many problems, such as collision detection, ray tracing and solid modelling. Due to the extensive applications of the BSP tree, it is important to ensure the correctness of its implementation. In this paper, we report our experience of testing the BSP-treeVS software.

The BSP-treeVS software suffers from the test oracle problem. We tackled this problem using the metamorphic testing technique. We identified 5 metamorphic relations for the BSP-treeVS algorithm. We detected a real fault in the BSP-treeVS software and fixed it. We also found that our metamorphic relations can collectively killed all the 10 mutants generated in this study.

The BSP tree has been used in many different problems. Our study would provide insight to testing of other BSP tree related programs. For example, some of our ways to generate follow-up test cases may be applicable to test these programs; and by some modification in our MR output relations, a new set of metamorphic relations may be generated. It is interesting to study how reusable our metamorphic relations are in testing of these programs. Many computer graphics programs also face the oracle problem. Our future work is to investigate other computer graphics software and identify useful metamorphic relations for these computer graphics programs.

6. ACKNOWLEDGMENT

This project is supported by the Australian Research Council Discovery Project (ARC DP0984760).

7. REFERENCES

[1] T. A. Budd. Mutation analysis: Ideas, examples, problems and prospects. In *Computer Program Testing (Proceedings of the Summer School on Cornpurer Program Testing)*.

[2] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, 1998.

[3] N. Chin. A walk through BSP trees. In A. W. Paeth., editor, *Sectio III.5 of Graphics Gems V*, pages 121–138. 1995.

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: help for the practicing programmer. *IEEE Computer*, 11(4):31–41, 1978.

[5] A. Dumitrescu, J. S. B. Mitchell, and M. Sharir. Binary space partitions for axis-parallel segments, rectangles, and hyperrectangles. *Discrete and Computational Geometry*, 31(2):207–227, 2004.

[6] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structure. In *Proceedings of the Seventh Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1980)*, pages 124–133, 1980.

[7] A. Gotlieb and B. Botella. Automated metamorphic testing. In *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC'03)*, pages 34–40, 2003.

[8] T. Ize, I. Wald, and S. Parker. Ray tracing with the bsp tree. In *IEEE Symposium on Interactive Ray Tracing (RT'08)*, pages 159–166, 2008.

[9] F.-C. Kuo, Z. Zhou, J. Ma, and G. Zhang. Metamorphic testing of decision support systems: a case study. accepted to appear in *IET Software*, 2010.

[10] C. Murphy, K. Shen, and G. Kaiser. Automatic system testing of programs without test oracles. In *Proceedings of the eighteenth international symposium on Software testing and analysis (ISSTA'09)*, pages 189–200, 2009.

[11] C. Murphy, K. Shen, and G. Kaiser. Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles. In *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation (ICST'09)*, pages 436–445, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[12] B. Naylor, J. Amanatides, and W. Thibault. Merging bsp trees yields polyhedral set operations. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques (SIGGRAPH'90)*, pages 115–124, 1990.

[13] S. Segura, R. M. Hierons, D. Benavides, and A. Ruiz-Cortes. Automated test data generation on the analyses of feature models: A metamorphic testing approach. accepted to appear in *Proceedings of the 3rd IEEE International Conference on Software Testing, Verification and Validation (ICST'10)*.

[14] K. Sung and P. Shirley. Ray tracing with the BSP tree. In D. Kirk, editor, *Graphics Gems III*, pages 271–274. Boston : Harcourt Brace Jovanovich, 1992.

[15] Z. Q. Zhou, S. J. Zhang, M. Hagenbuchner, T. H. Tse, F.-C. Kuo, and T. Y. Chen. Automated functional testing of online search services. accepted to appear in *Software Testing, Verification and Reliability*, 2010.

Fixing errors in bspPartition.c		
LineNo	Original statement	Corrections made to fix the error
20	static SIGN whichSideIsFaceWRTplane(FACE *face, const PLANE *plane);	static SIGN whichSideIsFaceWRTplane(FACE *face, const FACE * chosen);//const PLANE *plane);
30-32	void BSPpartitionFaceListWithPlane(const PLANE *plane,FACE **faceList, FACE **faceNeg, FACE **facePos, FACE **faceSameDir, FACE **faceOppDir)	void BSPpartitionFaceListWithPlane(const FACE *chosen,FACE **faceList, FACE **faceNeg, FACE **facePos, FACE **faceSameDir, FACE **faceOppDir)
47	v1= findNextIntersection(ftrav->vhead,plane, &ixx1, &iyy1, &izz1, &signV1);	v1= findNextIntersection(ftrav->vhead, &(chosen->plane), &ixx1, &iyy1, &izz1, &signV1);
52	v2= findNextIntersection(v1->vnext, plane, &ixx2, &iyy2, &izz2, &signV2);	v2= findNextIntersection(v1->vnext, &(chosen->plane), &ixx2, &iyy2, &izz2, &signV2);
90	side= whichSideIsFaceWRTplane(ftrav,plane);	side= whichSideIsFaceWRTplane(ftrav,chosen);
97-99	if (IS_EQ(ftrav->plane.aa,plane->aa) && IS_EQ(ftrav->plane.bb,plane->bb) && IS_EQ(ftrav->plane.cc,plane->cc))	if (IS_EQ(ftrav->plane.aa,chosen->plane.aa) && IS_EQ(ftrav->plane.bb,chosen->plane.bb) && IS_EQ(ftrav->plane.cc,chosen->plane.cc))
223	static SIGN whichSideIsFaceWRTplane(FACE *face, const PLANE *plane)	static SIGN whichSideIsFaceWRTplane(FACE *face, const FACE * chosen)//const PLANE *plane)
230	empty line	add the following code to the line: if(face->plane.aa == chosen->plane.aa && face->plane.bb == chosen->plane.bb && face->plane.cc == chosen->plane.cc && face->plane.dd == chosen->plane.dd) return (ZERO);
232-233	value= (plane->aa*vtrav->xx) + (plane->bb*vtrav->yy) + (plane->cc*vtrav->zz) + plane->dd;	value= (chosen->plane.aa*vtrav->xx) + (chosen->plane.bb*vtrav->yy) + (chosen->plane.cc*vtrav->zz) + chosen->plane.dd;
252-253	value= (plane->aa*vtrav->xx) + (plane->bb*vtrav->yy) + (plane->cc*vtrav->zz) + plane->dd;	value= (chosen->plane.aa*vtrav->xx) + (chosen->plane.bb*vtrav->yy) + //Diana change (chosen->plane.cc*vtrav->zz) + chosen->plane.dd;//Diana change
Fixing errors in bspTree.c		
LineNo	Original statement	Corrections made to fix the error
7	static void BSPchoosePlane(FACE *faceList,PLANE *plane);	static void BSPchoosePlane(FACE *faceList,FACE * chosen);//PLANE *plane);
21	BSPNODE *newBspNode; PLANE plane;	BSPNODE *newBspNode; //PLANE plane;
23	empty line	add the following code to the line: FACE chosen;
25	BSPchoosePlane(*faceList,&plane);	BSPchoosePlane(*faceList,&chosen);
26-27	BSPpartitionFaceListWithPlane(&plane,faceList, &faceNegList, &facePosList, &sameDirList, &oppDirList);	BSPpartitionFaceListWithPlane(&chosen,faceList, &faceNegList, &facePosList, &sameDirList, &oppDirList);
100	static void BSPchoosePlane(FACE *faceList,PLANE *plane)	static void BSPchoosePlane(FACE *faceList,FACE * chosen)//PLANE *plane)
121	*plane= chosenRoot->plane;	chosen->plane= chosenRoot->plane;
Fixing errors in bsp.h		
LineNo	Original statement	Corrections made to fix the error
76-78	void BSPpartitionFaceListWithPlane(const PLANE *plane,FACE **faceList, FACE **faceNeg, FACE **facePos, FACE **faceSameDir, FACE **faceOppDir);	void BSPpartitionFaceListWithPlane(const FACE *chosen,FACE **faceList, FACE **faceNeg, FACE **facePos, FACE **faceSameDir, FACE **faceOppDir);

Table 7: Fixing errors in the original BSP-treeVS software

ID	Class	LineNo	Original Statement	Seeded Fault
μ1	bspTree.c	58	BSPtraverseTreeAndRender(bspNode−>node−>**negativeSide**,position);	BSPtraverseTreeAndRender(bspNode−>node−>**positiveSide**,position);
μ2	bspTree.c	109	for (rootrav= faceList, ii= 0; rootrav **!=** NULL_FACE && ii< MAX_CANDIDATES; rootrav= rootrav−>fnext, ii++)	for (rootrav= faceList, ii= 0; rootrav **==** NULL_FACE && ii< MAX_CANDIDATES; rootrav= rootrav−>fnext, ii++)
μ3	bspTree.c	115	if (doesFaceStraddlePlane(ftrav, &rootrav−>plane)) count**++**;	if (doesFaceStraddlePlane(ftrav, &rootrav−>plane)) count**−−**;
μ4	bspTree.c	109	for (rootrav= **faceList**, ii= 0; rootrav != NULL_FACE && ii< MAX_CANDIDATES;rootrav= rootrav−>fnext, ii++)	for (rootrav= **faceList−>fnext**, ii= 0; rootrav != NULL_FACE && ii< MAX_CANDIDATES; rootrav= rootrav−>fnext, ii++)
μ5	bspTree.c	137	float value= plane−>aa*vtrav−>xx **+** plane−>bb*vtrav−>yy + plane−>cc*vtrav−>zz + plane−>dd;	float value= plane−>aa*vtrav−>xx ***** plane−>bb*vtrav−>yy + plane−>cc*vtrav−>zz + plane−>dd;
μ6	bspTree.c	69	**BSPtraverseTreeAndRender(bspNode−> node−> negativeSide,position)**;	**MISSING**
μ7	bspPartition.c	189	if (vtemp == v2)	if (vtemp **!=** v2)
μ8	bspPartition.c	341	return((sign1 == **-1**) ? NEGATIVE : POSITIVE);	return((sign1 == **1**) ? NEGATIVE : POSITIVE);
μ9	bspPartition.c	292	temp1= (aa*x1) + (bb*y1) + (cc*z1) + **dd**;	temp1= (aa*x1) + (bb*y1) + (cc*z1) + **++dd**;
μ10	bspPartition.c	92	PREPEND_FACE(ftrav,***faceNeg**);	PREPEND_FACE(ftrav,***facePos**);

Table 8: Fault in each mutant (denoted as μ)