Scene Management Using BSP Trees

Skipping chapters

- **3D Graphics**
  - Types of 3D Rendering
  - Don't Forget Your Math
  - 3D Basics
  - 3D Math
  - Polygons
  - 3D Transforms
  - A Simple 3D Pipeline
  - Camera Movement
  - Solid Objects and Back-Face Removal
  - Scan-Converting Polygons
  - 3D Clipping
  - Final Rendering Pipeline
- **Texture Mapping and Lighting**
  - Perspective-Correct Texture Mapping Basics
  - A Simple Texture-Mapper
  - Optimizing Texture Mapping
  - Simple Lighting
  - Implementing Texture Lighting
  - Advanced Lighting Using a Shade Map
  - Additional Concepts

Skipping chapters (2)

- **3D Objects**
  - Hidden Surface Removal
  - 3D Animation
  - Polygon Groups
  - Loading Polygon Groups from an OBJ File
  - Game Objects
  - Managing Game Objects
  - Putting It All Together
  - Future Enhancements

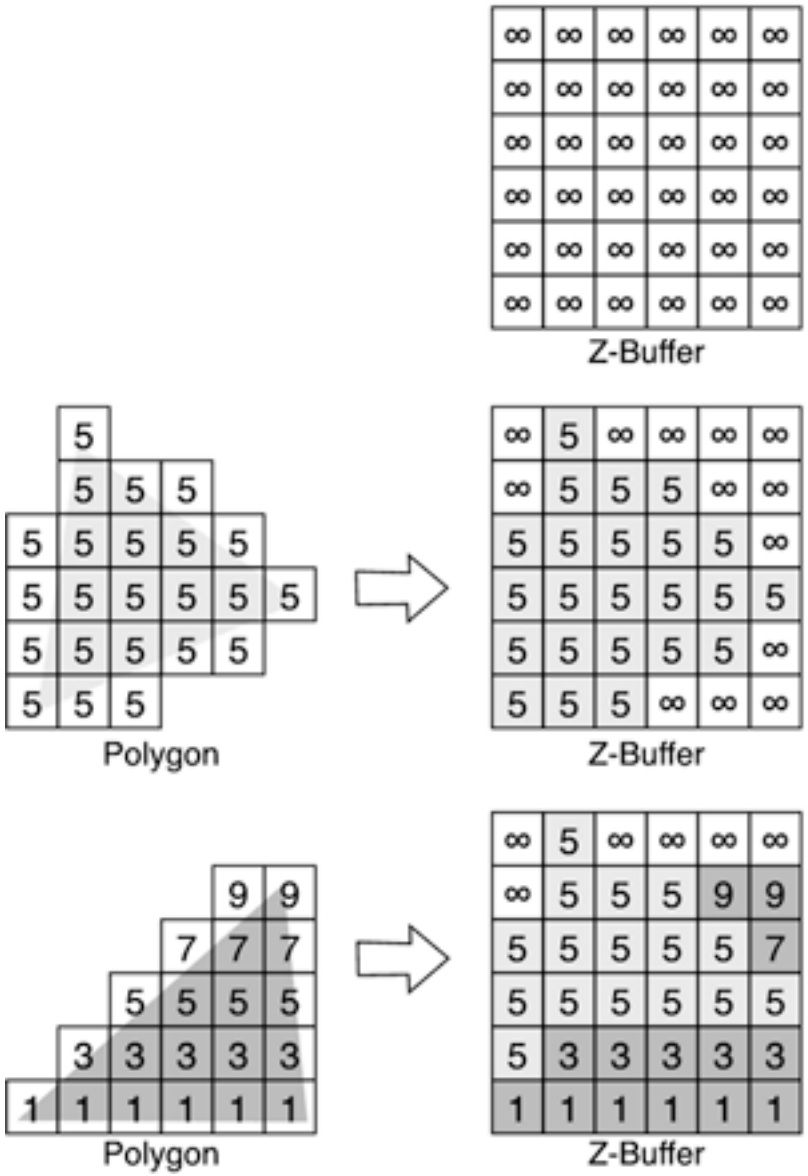
Content

1. BSP Tree Intro
2. Binary Tree Basics
3. The One-Dimensional BSP Tree
4. The Two-Dimensional BSP Tree
5. Implementing a 2D BSP Tree
6. Drawing Polygons Front to Back
7. First BSP Example
8. Drawing Objects in the Scene
9. Loading Maps from a File
10. Putting It All Together
11. Enhancements

1. BSP Tree Introduction

- What you really want is an alternative to the z-buffer with the following goals:
  - To potentially manage a large number of polygons
  - To quickly decide which polygons are visible from any location in the scene
  - To draw only the visible polygons (and to not draw parts of polygons that aren't visible)
  - To draw every pixel only once (no overdraw)

Excursus: z-Buffer



- z-buffering: the depth of each pixel in the polygon is recorded in a buffer the same size as the view window.
- A pixel is drawn only if the depth of that pixel is closer than the depth currently in that location in the z-buffer.
- Results in a pixel-perfect 3D scene, no matter what order the polygons are drawn.

BSP Tree Introduction (2)

- At first, those seem like some difficult goals to accomplish. Of course, the technique you use to solve these goals really depends on the type of game you have (e.g. buildings, open spaces)

- You can organize polygons in many different ways to make it easy to decide which polygons are visible. Besides BSP trees, some other types of polygon-organization techniques are octrees and scene graphs. BSP trees enable you to draw all polygons from front-to-back order from any location in the scene.

- By extension, using BSP trees solves all of the previous goals. However, the BSP tree isn't perfect. Here are a couple drawbacks:
  - The world must be static (no moving walls here). However, 3D game objects can move around, so you can use 3D game objects for things such as monsters and doors.
  - It's computationally expensive to build the tree. However, the ideal solution is to build the tree beforehand, loading the built tree at startup time.

BSP Tree Introduction (3)

- Games such as Doom were the first to implement BSP trees. Doom used a 2D BSP tree with variable-height floors and ceilings, and the rendering engine had a restriction that a player could not look up and down in the game.

- Even though we demonstrate a 2D BSP tree in this chapter, a 3D BSP tree, used in many modern games, is not that much more difficult to master. A 2D BSP tree just makes the examples and equations a bit easier to follow, and other topics such as collision detection and path finding are easier to describe.

- The drawback to using a 2D BSP tree is that you can make a world with only vertical walls and horizontal floors and ceilings. However, you still can make some cool worlds with this limitation.

2. Binary Tree Basics

- A BSP tree is a binary tree, so first let's run through some basic binary tree terminology and concepts.

- A binary tree is a data structure that contains nodes in a hierarchical structure. Binary trees have the following properties:

  - Each node has, at most, two children, often called the left child and the right child.

  - The nodes have data associated with them.

  - A node with no children is called a leaf.

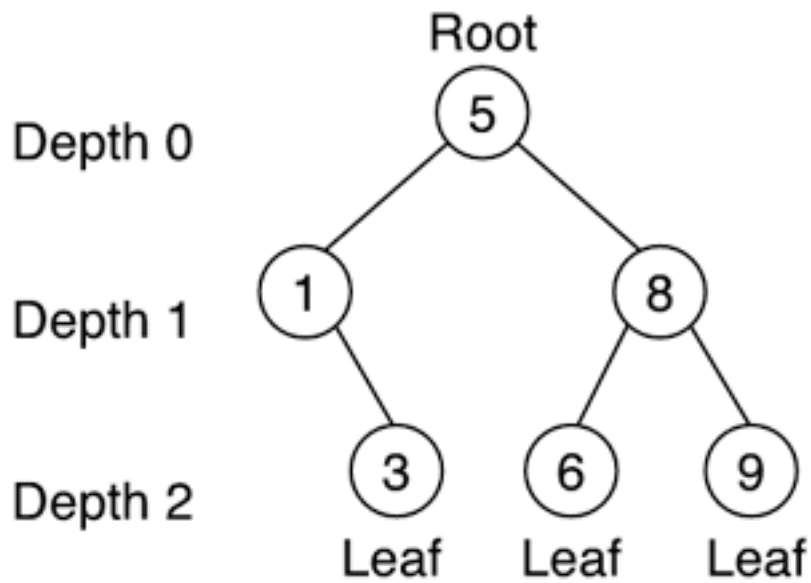
  - The node with no parent is called the root node.

  - A sorted binary tree, often called a binary search tree, contains nodes sorted by their data.

Binary Tree Basics (2)

- Sorted binary tree in example. It contains a list of numbers—1, 3, 5, 6, 8, and 9. For every node, all values less than that node's value are on the left side of that node, and all values greater than that node's value are on the right side of that node.

- For example, because the root node has the value of 5, every number that is less than 5 is on the left side of the tree, and every number that is greater than 5 is on the right side of the tree.



Binary tree with three leaves and a depth of 2.

Binary Tree Basics (3)

- Now we'll implement a simple binary tree and later use the same concepts to make a BSP tree. We'll use integers as the binary tree's data, just like in the example. Let's start with a node:

```
public class Node {
    Node left;
    Node right;
    int value;

    public Node(int value) {
        this.value = value;
    }
}
```

- The Node class has an integer value and references to the left and right children (which are initially null, so it has no children). Remember, a node can be a root, a leaf, or just a plain node. Creating a new Node is easy—here, you create a root node:

```
Node root = new Node(5);
```

Binary Tree Basics (4)

- Next, you write a method, **insertSorted()**, to insert a new value into a sorted tree.

```
public void insertSorted(Node node, int value) {
    if (value < node.value) {
       if (node.left != null) {
           insertSorted(node.left, value);
       }
       else {
           node.left = new Node(value);
       }
    }
    else if (value > node.value) {
       if (node.right != null) {
           insertSorted(node.right, value);
       }
       else {
           node.right = new Node(value);
       }
    }
    else {
        // do nothing - value already in tree
    }
}
```

- To insert the value 8 into the tree, call the following:

```
insertSorted(root, 8);
```

Binary Tree Basics (5)

- **insertSorted(node.left, value)** just inserts the value into the node's left subtree.

- Notice that this method doesn't allow duplicates into the tree: If a value is already in the tree, it's not inserted again.

- Finally, you make a method to print every value in the order they are in the tree. This is called an in-order traversal because, you guessed it, the values are traversed in order.

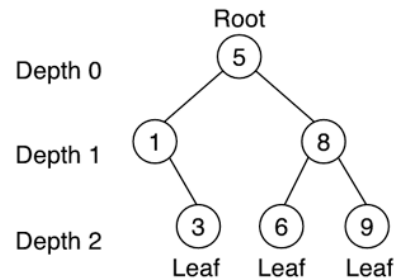
```
public void printInOrder(Node node) {
    if (node != null) {
        printInOrder(node.left);
        System.out.println(node.value);
        printInOrder(node.right);
    }
}
```

3. The One-Dimensional BSP Tree

- Eventually, this chapter will focus on 2D BSP trees, but understanding how a 2D or 3D BSP tree works can be difficult at first. So, we first explain a one-dimensional BSP tree. This example might seem trivial, but it will give you the idea of how BSP trees work.

- In short, a BSP tree's node divides a world into halves—or, in other words, it is a binary space partition. Using a BSP tree allows you to sort polygons front to back from the camera's location.

- Now imagine the player is standing in a simple world with a row of houses, as shown in the example. Note that the houses are sorted by their numbers. Also, the houses have the same numbers as in the example in the example tree.



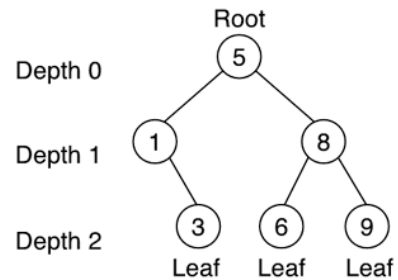
The player is standing at position 7 in a row of houses.

The One-Dimensional BSP Tree (2)

- Now assume that each house is a node partition. In this one-dimensional example, each house partitions the world into two spaces: the left space (everything to the left of the house) and the right space (everything to the right of the house).

- Building the tree follows the same process you used to build a sorted binary tree in the previous section. The resulting binary tree in this example could look just like the one in the figure on the right bottom.

- Now let's say you want to draw these houses in a 3D world. You want to draw the houses front to back from the camera's location, to eliminate overdraw and avoid the speed shortcomings of a z-buffer. So, in this example, you want to draw the houses closer to the player before drawing the farther ones.



The player is standing at position 7 in a row of houses.

The One-Dimensional BSP Tree (3)

- Looking at the figure, the order to draw the houses in front to back order is broken down into two simple rules:
  - Draw houses to the left of the camera in reverse order: (6, 5, 3, 1).
  - Draw houses to the right of the camera in order: (8, 9).
- From a 3D perspective, the houses on the left side will never overlap the houses on the right side. Therefore, it doesn't matter whether the order in which you draw the houses is first the left side and then the right (6, 5, 3, 1, 8, 9) or whether the two sides are mixed in some variation (6, 8, 5, 9, 3, 1). It matters only if each side is drawn in order. So, this technique works no mater which way the player is facing.

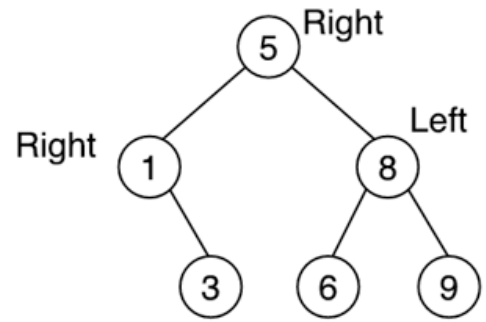
The One-Dimensional BSP Tree (4)

- You'll implement these two rules with a variation of the printInOrder() method you created in the previous section, creating a printFrontToBack() method.

```
public void printFrontToBack(Node node, int camera)
{
    if (node == null) return;
    if (camera < node.value) {
        // print in order
        printFrontToBack(node.left, camera);
        System.out.println(node.value);
        printFrontToBack(node.right, camera);
    }
    else if (camera > node.value) {
        // print in reverse order
        printFrontToBack(node.right, camera);
        System.out.println(node.value);
        printFrontToBack(node.left, camera);
    }
    else {
        // order doesn't matter
        printFrontToBack(node.left, camera);
        printFrontToBack(node.right, camera);
    }
}
```

The One-Dimensional BSP Tree (5)

- Let's walk through this method by looking at this example. With the binary tree calling *printFrontToBack()* with camera location 7 follow these steps:



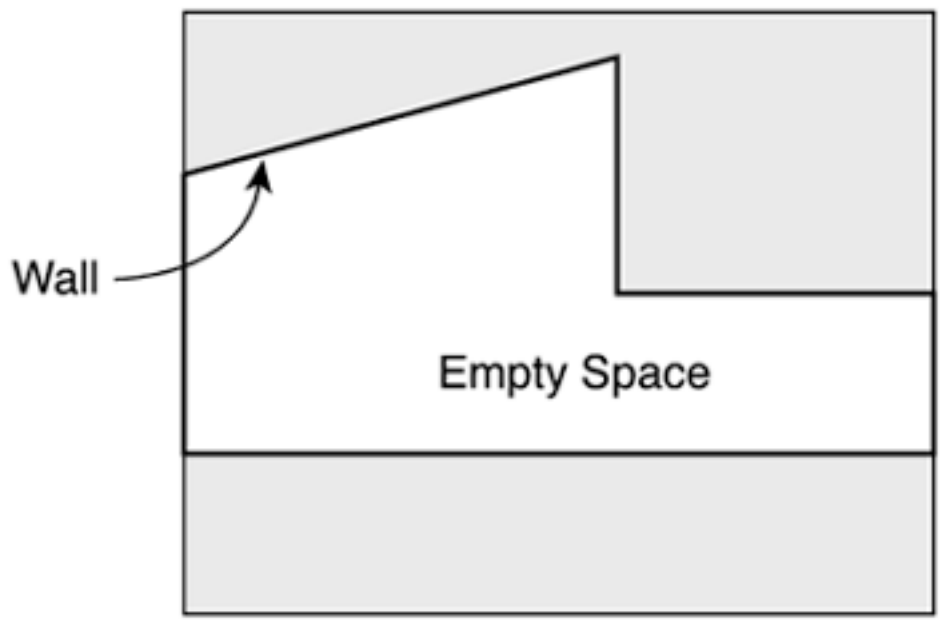
1. At the root node, 7 is greater than 5, so traverse the right node.
2. At node 8, 7 is less than 8, so traverse the left node.
3. Node 6 is a leaf, so print 6 and return to the parent.
4. Print 8 and traverse its right node.
5. Node 9 is a leaf, so print 9 and return to the parent.
6. You're finished traversing node 8, so return to the parent.
7. Print 5 and traverse its left node.
8. At node 1, 7 is greater than 1, so traverse the right node.
9. Node 3 is a leaf, so print 3 and return to the parent.
10. Node 1 does not have a left child, so print 1 and return.
11. You're finished traversing the root.

- So, calling *printFrontToBack()* with camera location 7 prints the order (6, 8, 9, 5, 3, 1). Everything on the left is printed in front-to-back order (6, 5, 3, 1), and everything on the right it printed in front-to-back order (8, 9), which is just what we wanted.

4. The Two-Dimensional BSP Tree

- To start, we'll go through an example of building and traversing a 2D BSP tree; afterward, we'll create an implementation.

- Take a look at the example 2D floor plan in this figure. We'll build a BSP tree based on this floor plan.



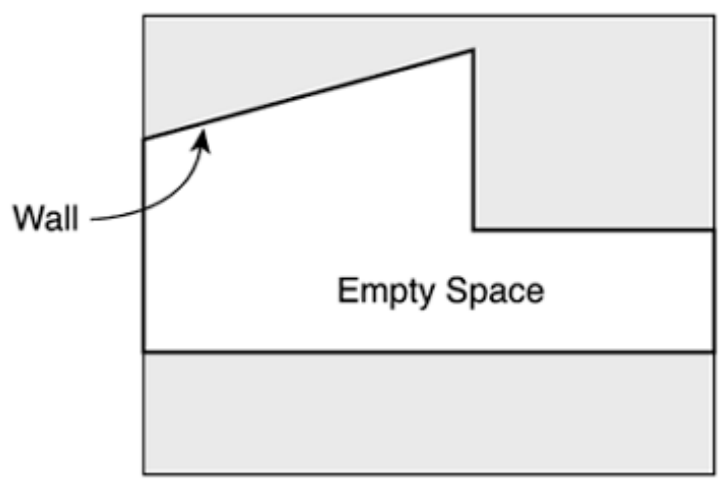
The 2D floor plan shows a simple room with four walls.

The Two-Dimensional BSP Tree (2)

- In the one-dimensional example, each house was a partition, so partitions were points. In 2D, partitions are lines, and in 3D the partitions are planes.

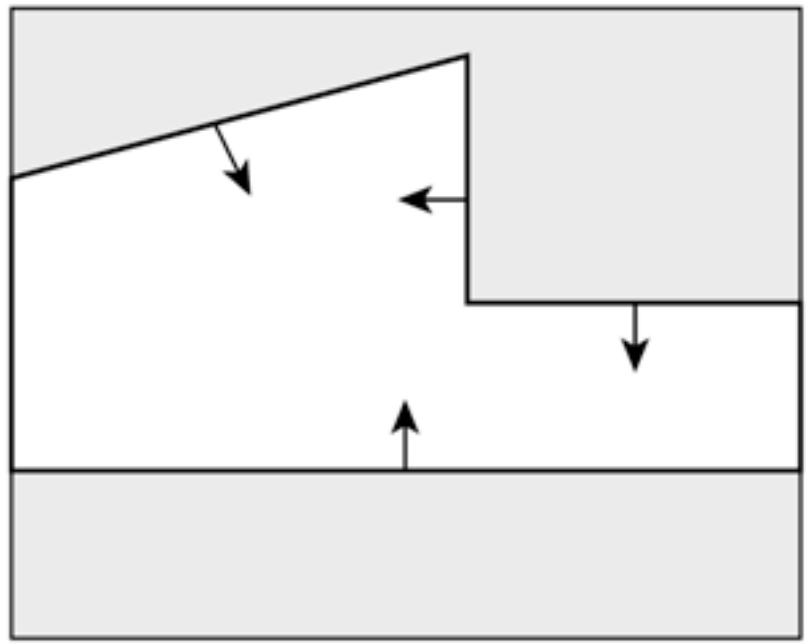
- Lines easily partition a 2D area into two halves, but to actually build a BSP tree, you need to know what's to the "left" of the line and what's to the "right."

- Actually, the terms left and right are a bit inaccurate in this case. These terms work fine for a one-dimensional world, but in 2D, lines can be oriented in any way. A vertical line divides the world into "left" and "right," but a horizontal line divides the world into "north" and "south."



The Two-Dimensional BSP Tree (3)

- Instead, we use the terms front and back. Line segments have a front and back just like polygons do, with the normal pointing in the direction of the front side, as shown in the figure. The front side is the visible side. This means you can use walls as the partitions.

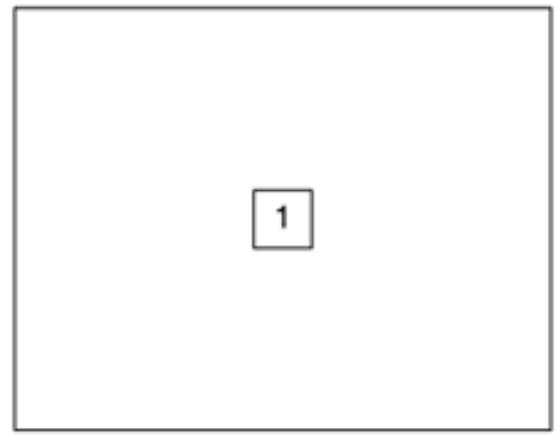


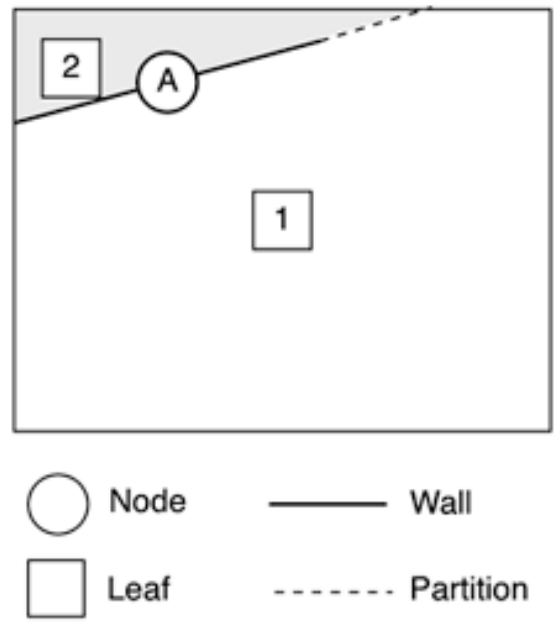
Line segments representing walls have a front and a back

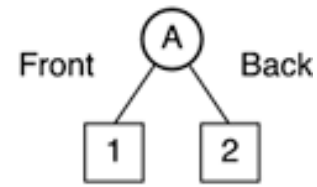
The Two-Dimensional BSP Tree (4)

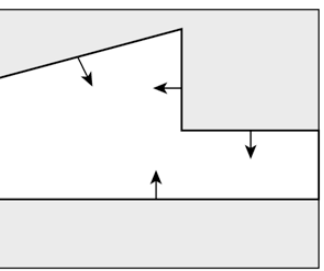
- As an example, you'll build the floor plan from our figure by adding one wall at a time. The order in which you add walls will be arbitrary.

- In the lower figure, the first partition, wall A, is added to the tree. Notice that the partition formed by wall A extends beyond wall A, shown as a dotted line.



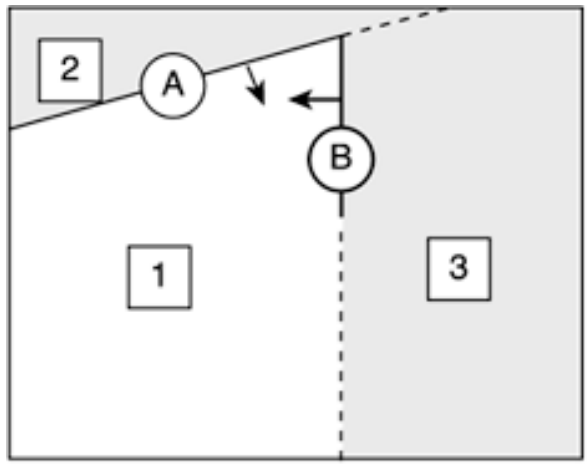


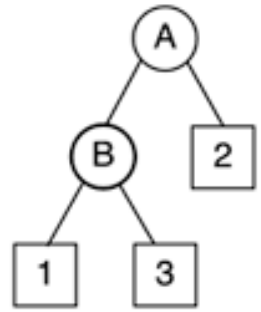




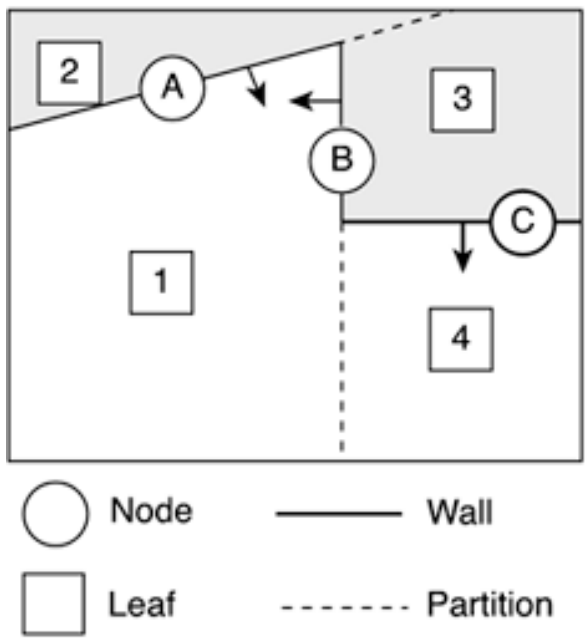
The Two-Dimensional BSP Tree (5)

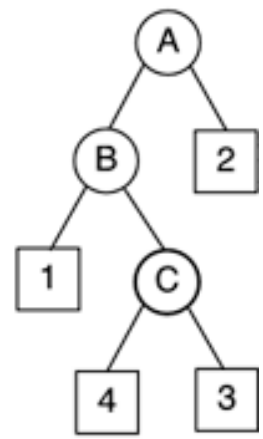
- The empty spaces will always be convex because splitting a convex shape along a line always creates two convex shapes. So, the floor and ceiling polygons will be convex as well.

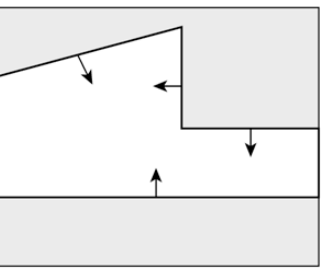




- Next, in the lower figure, walls B and C are added to the tree. Wall B is in front of A, so it is added to the front node of A. This splits leaf 1 into two leaves (1 and 3). Wall C is in front of A and in back of B, so it is added to the back node of B and splits leaf 3 into two leaves (3 and 4).

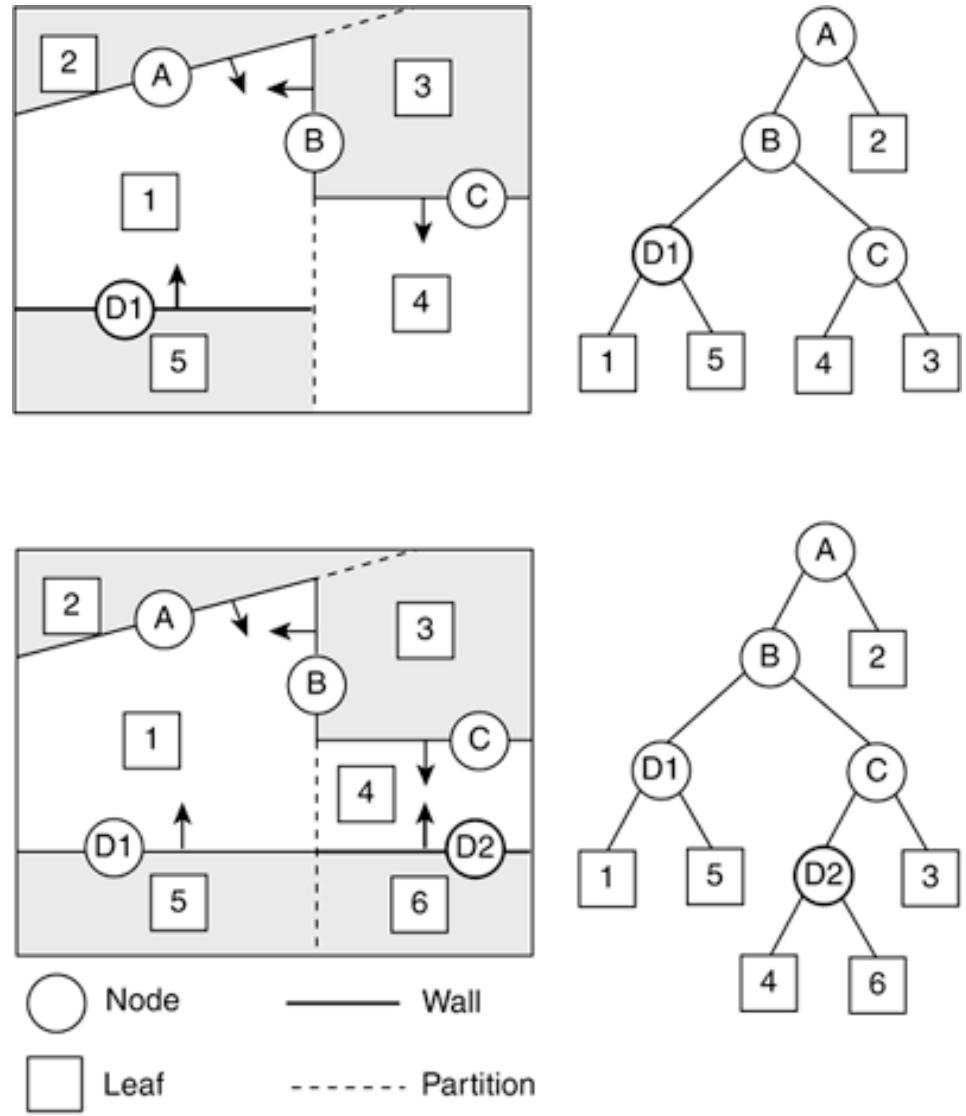


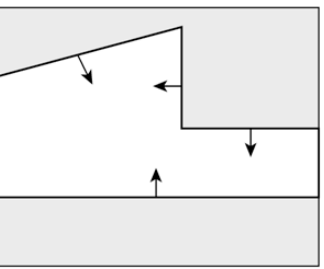




The Two-Dimensional BSP Tree (6)

- The BSP building continues in Figure 10.8. There's just one more wall to add: wall D. But wall D actually spans the partition formed by wall B. Walls can't exist in two different locations in the tree, so wall D is split into two walls: D1 and D2.

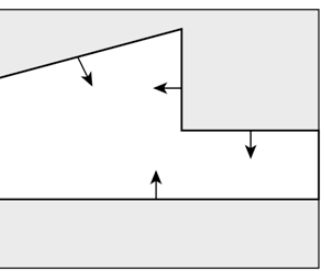


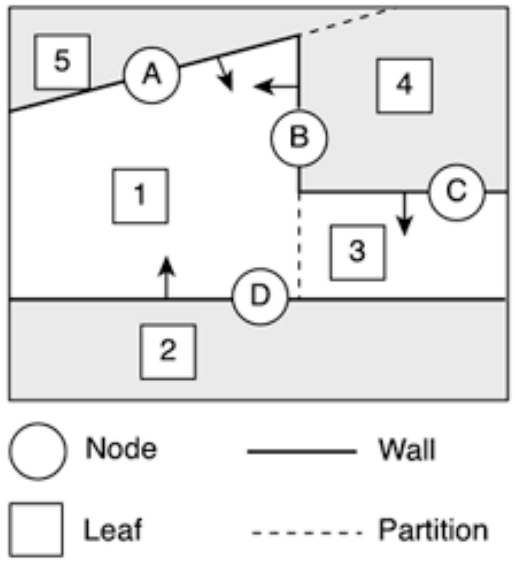


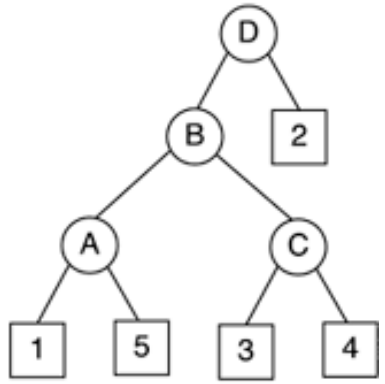
The Two-Dimensional BSP Tree (7)

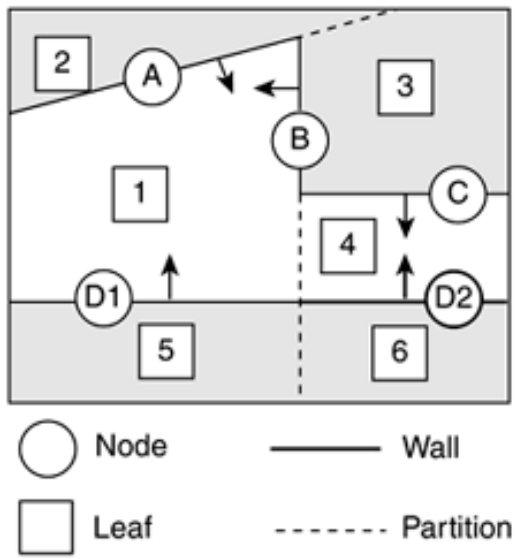
- Going through the process of building the tree, you probably noticed that the tree would be different if the walls were added in a different order.

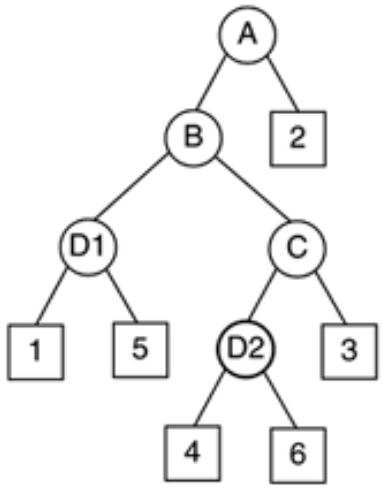
- For example, if the walls were added in the order D, B, C, A, you would get the tree in the figure on the top. This tree didn't require any walls to be split, so there are fewer nodes. Also, the tree has a shorter depth. However, the tree isn't well balancec —all the walls are in the front of the first node.











The Two-Dimensional BSP Tree (8)

- When you have enough polygons, shorter, balanced trees will make for faster searches. So, the order in which you add walls to the tree is important.

- An algorithm to decide the order in which you choose partitions should keep two things in mind:

  - Minimize splits. Fewer splits mean fewer polygons and less tree traversal.

  - Keep the tree as balanced as possible. Shorter, balanced trees means less tree traversal for unbalanced trees.

- Sometimes these goals are mutually exclusive: Keeping the tree balanced can create more splits, and minimizing splits can create an unbalanced tree. Finding the right algorithm can be complicating. We present a couple of ideas here.

The Two-Dimensional BSP Tree (9)

- First, you could just minimize the number of splits, ignoring the balance of the tree. Here, every time you choose a partition, you would choose the one resulting in the minimum number of splits. For example, you wouldn't choose wall B before wall D because wall B splits wall D.

- Second, you could just try to keep the tree as balanced as possible, ignoring the number of splits. Here you would always choose a partition that keeps the same number of walls (plus or minus one) on either side of the partition. For example, you could choose wall B as the first partition because two partitions would be on either side of it (even though it splits wall D).

- Or, you could combine these two ideas. For a set of partitions that keep the tree relatively well balanced (with a certain degree), choose the partition with the minimum number of splits.

The Two-Dimensional BSP Tree (10)

- **BSP Tree-Traversing Example**

- Traversing the BSP tree is similar to how you traversed the one-dimensional BSP tree. Two rules apply:

  - Draw polygons in front of the camera in order (front node, current node, back node).

  - Draw polygons in back of the camera in reverse order (back node, current node, front node).

The Two-Dimensional BSP Tree (11)

- Take a look at the example in this figure. Imagine the camera is at the location marked with a star. This traversal algorithm traverses the nodes in this order: 4, D2, 6, C, 3, B, 1, D1, 5, A, 2. The polygons are traversed front to back, which is just what you want.

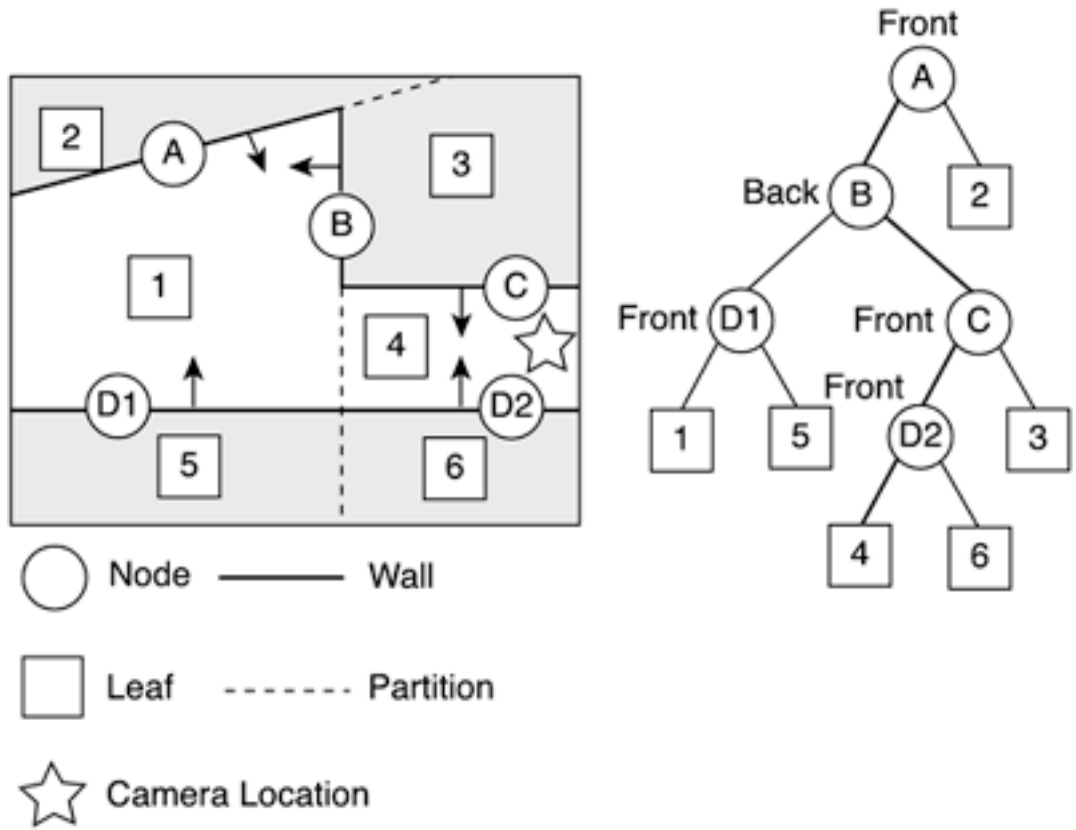
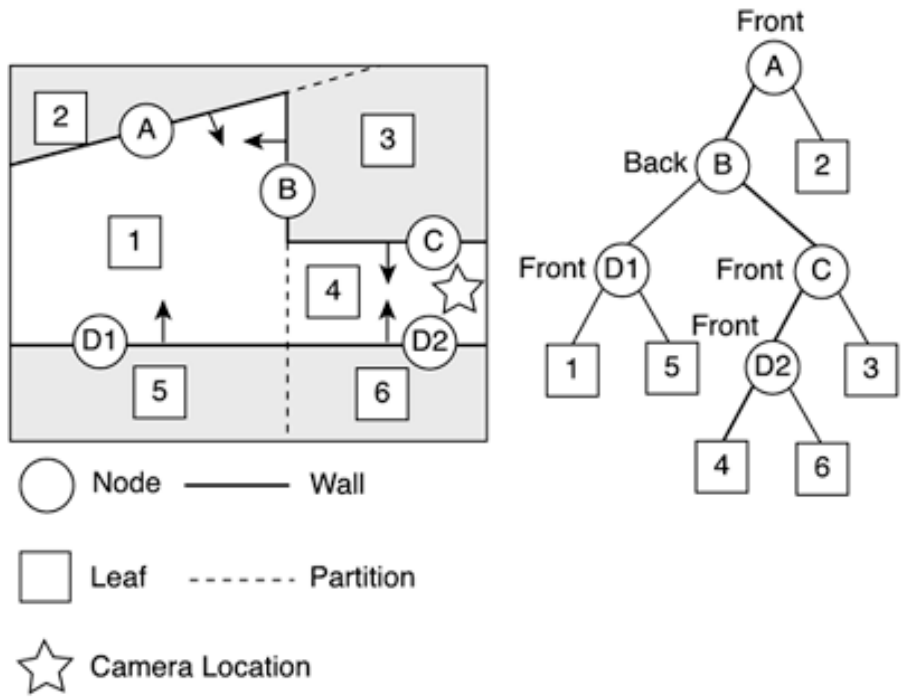


Figure 10.10. Traversing the tree from the star's location.

The Two-Dimensional BSP Tree (12)

- Also with this algorithm, the first leaf traversed is the leaf that the camera is in. Because horizontal polygons can have different heights, if you know the leaf the player is in, you can determine at what height the player should be located. For example, if the player is 100 units tall and the player is in a leaf with a floor at y=200, the player's camera will be y=300.

- Now with the idea of building and traversing the polygons in a BSP tree, we move on a step deeper and write the code for a 2D BSP tree renderer.



5. Implementing a 2D BSP Tree

- To start, you'll create the simple **BSPTree** class shown in this listing. This class defines the data structure of the tree as nodes and leaves.

- The BSPTree class contains the inner class Node, which is a node in the tree. The node contains references to front and back nodes, a partition, and a list of polygons.

- The partition is a BSPLine, which we discuss in the next.

```
public class BSPTree {

    /**
        A Node of the tree. All children of the node are either
        to the front or back of the node's partition.
    */
    public static class Node {
        public Node front;
        public Node back;
        public BSPLine partition;
        public List polygons;
    }


    /**
        A Leaf of the tree. A leaf has no partition or front or
        back nodes.
    */
    public static class Leaf extends Node {
        public float floorHeight;
        public float ceilHeight;
        public Rectangle bounds;
        public boolean isBack;
    }

    private Node root;

    /**
        Creates a new BSPTree with the specified root node.
    */
    public BSPTree(Node root) {
       this.root = root;
    }


    /**
        Gets the root node of this tree.
    */
    public Node getRoot() {
        return root;
    }
}
```

Implementing a 2D BSP Tree (2)

- For nodes that aren't leaves, the list of polygons is a list of all the wall polygons collinear with the partition. For leaves (open spaces), this list contains floor and ceiling polygons.

- The inner class Leaf is a subclass of Node. It contains the height of the floor and ceiling. It also contains the rectangular bounds of the leaf.

```
public class BSPTree {

    /**
        A Node of the tree. All children of the node are either
        to the front or back of the node's partition.
    */
    public static class Node {
        public Node front;
        public Node back;
        public BSPLine partition;
        public List polygons;
    }


    /**
        A Leaf of the tree. A leaf has no partition or front or
        back nodes.
    */
    public static class Leaf extends Node {
        public float floorHeight;
        public float ceilHeight;
        public Rectangle bounds;
        public boolean isBack;
    }

    private Node root;

    /**
        Creates a new BSPTree with the specified root node.
    */
    public BSPTree(Node root) {
       this.root = root;
    }


    /**
        Gets the root node of this tree.
    */
    public Node getRoot() {
        return root;
    }
}
```

Implementing a 2D BSP Tree (3)

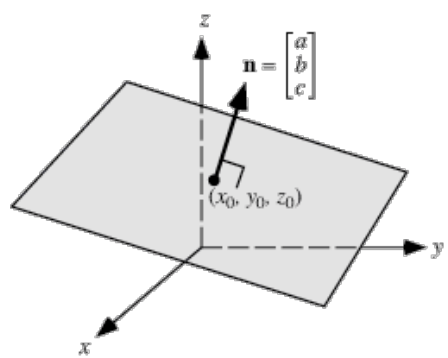
- **The BSP Line**

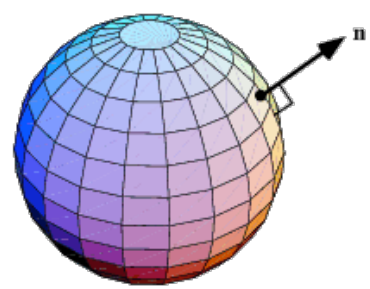
- The line you'll use for partitions (and a few other uses) is the BSPLine.

- Subclass of Line2D.Float. The Line2D.Float class is part of the java.awt.geom package that has floating-point fields x_1, y_1, x_2, and y_2 that define a line segment.

- In BSPLine, you also include the values for the line's normal, or the direction perpendicular to the line.

```
package com.brackeen.javagamebook.bsp2D;

import java.awt.geom.*;
import com.brackeen.javagamebook.math3D.*;

public class BSPLine extends Line2D.Float {

    /**
        X coordinate of the line normal.
    */
    public float nx;

    /**
        Y coordinate of the line normal.
    */
    public float ny;
}
```





Implementing a 2D BSP Tree (4)

- **Determining the Side of a Point Relative to a Line**

- One thing you need to determine is whether polygons are in front of a line, in back of a line, or spanning a line. You do this by polygon clipping one point at a time. For each point in a polygon, you determine whether the point is in front of or in back of the line.

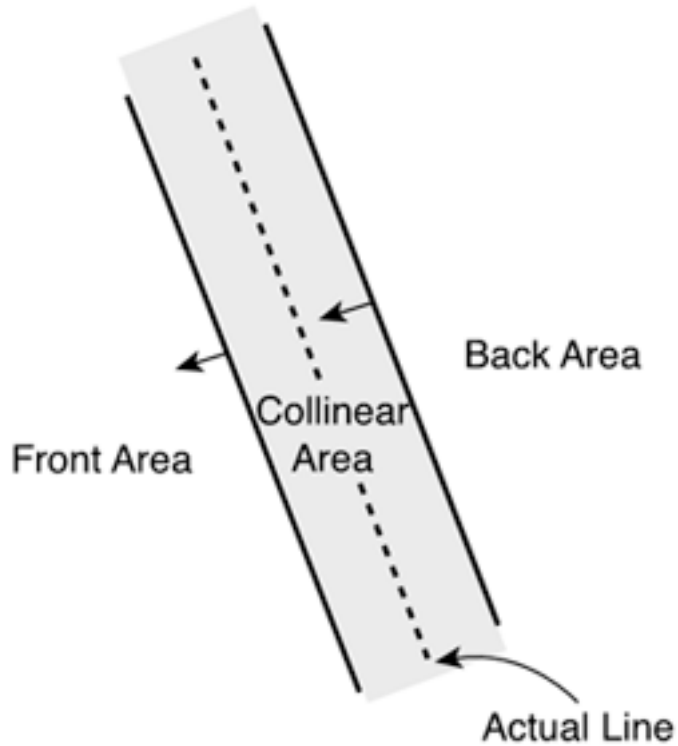
- You also need to know whether a point is actually on a line (in other words, is collinear with a line). For example, a polygon might have some points collinear with a line and other points in front. In this case, the polygon would be considered in front of the line, even though parts of it are collinear.

- Finding whether a point is in front or back of a line is similar to how you determined whether a point was in front or in back of a polygon, only in 2D: Just find the dot product between the vector to the point and the normal of the line. If this value is less than 0, the point is in back; if it is greater than 0, the point is in front; and if it is equal to 0, the point is collinear.

Implementing a 2D BSP Tree (5)

- One problem is that, due to floating-point inaccuracy, it is pretty rare for a point to actually be collinear. This can cause problems if, say, a collinear point is calculated as being in back of the line while other points in a polygon are in front. In this case, the polygon would be considered to be spanning the line, which is inaccurate.

- To fix this, you must sometimes treat lines as being "thick," as shown in the figure:



Implementing a 2D BSP Tree (6)

- If you assume that the normal to the line is a unit normal (has a length of 1), we can easily "shift" the line forward and backward, and determine the side the point is on for both of these lines. If you want a 1-unit-wide line, shift the line by half of the length of the normal. This is shown in the **getSideThick()** method (in BSPLine,.java).

```
public static final int BACK = -1;
public static final int COLLINEAR = 0;
public static final int FRONT = 1;
public static final int SPANNING = 2;

...

/**
    Normalizes the normal of this line (make the normal's
    length 1).
*/
public void normalize() {
    float length = (float)Math.sqrt(nx * nx + ny * ny);
    nx/=length;
    ny/=length;
}


/**
    Gets the side of this line the specified point is on.
    Because of floating point inaccuracy, a collinear line
    will be rare. For this to work correctly, the normal of
    this line must be normalized, either by setting this line
    to a polygon or by calling normalize().
    Returns either FRONT, BACK, or COLLINEAR.
*/
public int getSideThin(float x, float y) {
    // dot product between vector to the point and the normal
    float side = (x - x1)*nx + (y - y1)*ny;
    return (side < 0)?BACK:(side > 0)?FRONT:COLLINEAR;
}
```

```
/**
    Gets the side of this line the specified point is on.
    This method treats the line as 1-unit thick, so points
    within this 1-unit border are considered collinear.
    For this to work correctly, the normal of this line
    must be normalized, either by setting this line to a
    polygon or by calling normalize().
    Returns either FRONT, BACK, or COLLINEAR.
*/
public int getSideThick(float x, float y) {
    int frontSide = getSideThin(x-nx/2, y-ny/2);
    if (frontSide == FRONT) {
        return FRONT;
    }
    else if (frontSide == BACK) {
        int backSide = getSideThin(x+nx/2, y+ny/2);
        if (backSide == BACK) {
            return BACK;
        }
    }
    return COLLINEAR;
}
```

Implementing a 2D BSP Tree (7)

- These methods are also in the BSPLine class. Both of these methods use "thick" lines. A polygon or line is considered to be SPANNING only if one point is front while another is in back. Later, you'll split spanning polygons in two so that one polygon is in front of the line and one is in back.

```
/**
    Gets the side of this line that the specified line segment
    is on. Returns either FRONT, BACK, COLLINEAR, or SPANNING.
*/
public int getSide(Line2D.Float segment) {
    if (this == segment) {
        return COLLINEAR;
    }
int p1Side = getSideThick(segment.x1, segment.y1);
    int p2Side = getSideThick(segment.x2, segment.y2);
    if (p1Side == p2Side) {
        return p1Side;
    }
    else if (p1Side == COLLINEAR) {
        return p2Side;
    }
    else if (p2Side == COLLINEAR) {
        return p1Side;
    }
    else {
        return SPANNING;
    }
}
```

```
/**
    Gets the side of this line that the specified polygon
    is on. Returns either FRONT, BACK, COLLINEAR, or SPANNING.
*/
public int getSide(BSPPolygon poly) {
    boolean onFront = false;
    boolean onBack = false;

    // check every point
    for (int i=0; i<poly.getNumVertices(); i++) {
        Vector3D v = poly.getVertex(i);
        int side = getSideThick(v.x, v.z);
        if (side == BSPLine.FRONT) {
            onFront = true;
        }
        else if (side == BSPLine.BACK) {
            onBack = true;
        }
    }

    // classify the polygon
    if (onFront && onBack) {
        return BSPLine.SPANNING;
    }
    else if (onFront) {
        return BSPLine.FRONT;
    }
    else if (onBack) {
        return BSPLine.BACK;
    }
    else {
        return BSPLine.COLLINEAR;
    }
}
```

Implementing a 2D BSP Tree (8)

- **Traversing a BSP Tree**

- You need two ways to traverse a BSP tree, just like you did for the 1D BSP tree example: in-order and front-to-back order. Sometimes you just need to traverse every polygon in the tree (such as during the building process), so you need something like an in-order traversal. When you're drawing, you need a front-to-back traversal.

- You'll create a **BSPTreeTraverser** class that actually performs traversals. First, though, you'll create a listener interface so that the BSPTreeTraverser can notify a listener when the polygons in a node are traversed. This **BSPTreeTraverseListener** interface, shown in the next listing, typically is used for a polygon renderer and when working with all polygons in the tree.

Implementing a 2D BSP Tree (9)

```
package com.brackeen.javagamebook.bsp2D;

/**
    A BSPTreeTraverseListener is an interface for a
    BSPTreeTraverser to signal visited polygons.
*/
public interface BSPTreeTraverseListener {

    /**
        Visits a BSP polygon. Called by a BSPTreeTraverer.
        If this method returns true, the BSPTreeTraverer will
        stop the current traversal. Otherwise, the BSPTreeTraverer
        will continue if there are polygons in the tree that
        have not yet been traversed.
    */
    public boolean visitPolygon(BSPPolygon poly,
        boolean isBackLeaf);

}
```

- The BSPTreeTraverseListener interface provides a **visitPolygon()** method that is called whenever a polygon is visited. If the implementation of this method returns true, the traversal stops. This way, you can stop a traversal at any time. When drawing, you don't need to continue traversing the tree after the screen is filled, so you can stop the traversal at that point. Of course, this method provides a polygon as a parameter, but in reality, you're traversing nodes.

Implementing a 2D BSP Tree (10)

```
private boolean traversing;
private BSPTreeTraverseListener listener;
private GameObjectManager objectManager;

...

/**
    Visits a node in the tree. The BSPTreeTraverseListener's
    visitPolygon() method is called for every polygon in
    the node.
*/
private void visitNode(BSPTree.Node node) {
    if (!traversing || node.polygons == null) {
        return;
    }

    boolean isBack = false;
    if (node instanceof BSPTree.Leaf) {
        BSPTree.Leaf leaf = (BSPTree.Leaf)node;
        isBack = leaf.isBack;
        // mark the bounds of this leaf as visible in
        // the game object manager.
        if (objectManager != null && leaf.bounds != null) {
            objectManager.markVisible(leaf.bounds);
        }
    }

    // visit every polygon
    for (int i=0; traversing && i<node.polygons.size(); i++) {
        BSPPolygon poly = (BSPPolygon)node.polygons.get(i);
        traversing = listener.visitPolygon(poly, isBack);
    }
}
```

- This method just visits every polygon in a node. Also, if there is a GameObjectManager, this method notifies the manager that the objects within the leaf's bounds are visible.

Implementing a 2D BSP Tree (11)

- **In-Order Traversal**

```
/**
    Traverses a tree in-order.
*/
public void traverse(BSPTree tree) {
    traversing = true;
    traverseInOrder(tree.getRoot());
}


/**
    Traverses a node in-order.
*/
private void traverseInOrder(BSPTree.Node node) {
    if (traversing && node != null) {
        traverseInOrder(node.front);
        visitNode(node);
        traverseInOrder(node.back);
    }
}
```

- There's nothing really new in these methods, except that the traversal stops if the traversing Boolean is set to false.

- As usual, you're using recursion to traverse the tree. If you're wondering about recursion speed, don't worry about it in this case. With binary trees traversal in Java, you won't see any significant speed improvement if you don't use recursion.

Implementing a 2D BSP Tree (12)

- **Building a Tree**

- Building a tree is a recursive process just like traversing the tree. The basic idea is to take a node with a list of polygons, choose a partition, build the front node using the polygons in front of the partition, and then build the back node using the polygons in back of the partition.

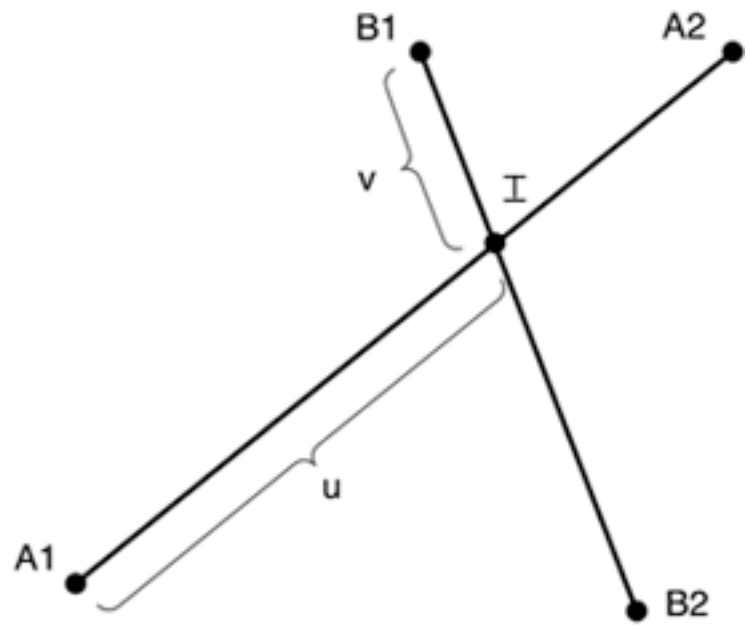
`Source: BSPTreeBuilder.java`

Implementing a 2D BSP Tree (13)

- **Finding the Intersection of Two Lines**

- When you clip a polygon to a line, you actually ignore the y component of the polygon to make it a 2D clip. You need to find the intersection of an edge of the polygon with a line. In other words, you need to find the intersection of two lines.

- The idea behind finding the intersection is shown in the figure.



$$I = A1 + u\,(A2 - A1)$$
$$I = B1 + v\,(B2 - B1)$$

Implementing a 2D BSP Tree (14)

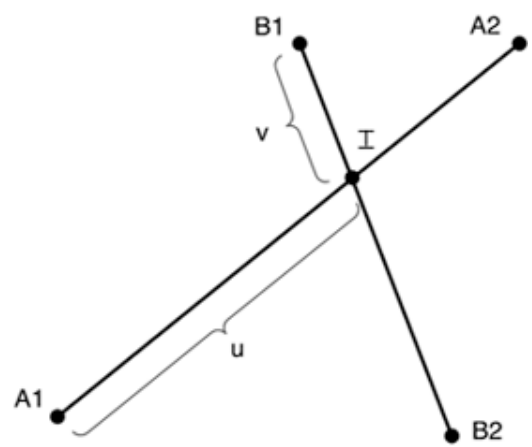
- The point I is the intersection between the two lines. The value u is a fraction of the intersection within line A, so if the two line segments intersect, u will be between 0.0 and 1.0. The same thing applies for value v in line B.

- The equations are just the basic equations of a line, but you can use these formulas to get the intersection. If you solve for u, you get this:

```
numerator = (B2y-B1y)(A2x-A1x)-(B2x-B1x)(A2y-A1y)
denominator = (B2y-B1y)(A2x-A1x)-(B2x-B1x)(A2y-A1y)
u = numerator/denominator
```

- Then, with u, you can find the intersection point:

```
x = A1x+u(A2x-A1x)
y = A1y+u(A2y-A1y)
```



I = A1 + u (A2 - A1)
I = B1 + v (B2 - B1)

Implementing a 2D BSP Tree (15)

- Intersection Methods of BSPLine.java

```
/**
    Returns the fraction of intersection along this line.
    Returns a value from 0 to 1 if the segments intersect.
    For example, a return value of 0 means the intersection
    occurs at point (x1, y1), 1 means the intersection
    occurs at point (x2, y2), and .5 means the intersection
    occurs halfway between the two endpoints of this line.
    Returns -1 if the lines are parallel.
*/
public float getIntersection(Line2D.Float line) {
    // The intersection point I, of two vectors, A1->A2 and
    // B1->B2, is:
    // I = A1 + u * (A2 - A1)
    // I = B1 + v * (B2 - B1)
    //
    // Solving for u gives us the following formula.
    // u is returned.
    float denominator = (line.y2 - line.y1) * (x2 - x1) -
        (line.x2 - line.x1) * (y2 - y1);

    // check if the two lines are parallel
    if (denominator == 0) {
        return -1;
    }

    float numerator = (line.x2 - line.x1) * (y1 - line.y1) -
         (line.y2 - line.y1) * (x1 - line.x1);

    return numerator / denominator;
}


/**
    Returns the intersection point of this line with the
    specified line.
*/
public Point2D.Float getIntersectionPoint(Line2D.Float line) {
    return getIntersectionPoint(line, null);
}
```

```
/**
    Returns the intersection of this line with the specified
    line. If intersection is null, a new point is created.
*/
public Point2D.Float getIntersectionPoint(Line2D.Float line,
    Point2D.Float intersection)
{
    if (intersection == null) {
        intersection = new Point2D.Float();
    }
    float fraction = getIntersection(line);
    intersection.setLocation(
        x1 + fraction * (x2 - x1),
        y1 + fraction * (y2 - y1));
    return intersection;
}
```

Implementing a 2D BSP Tree (16)

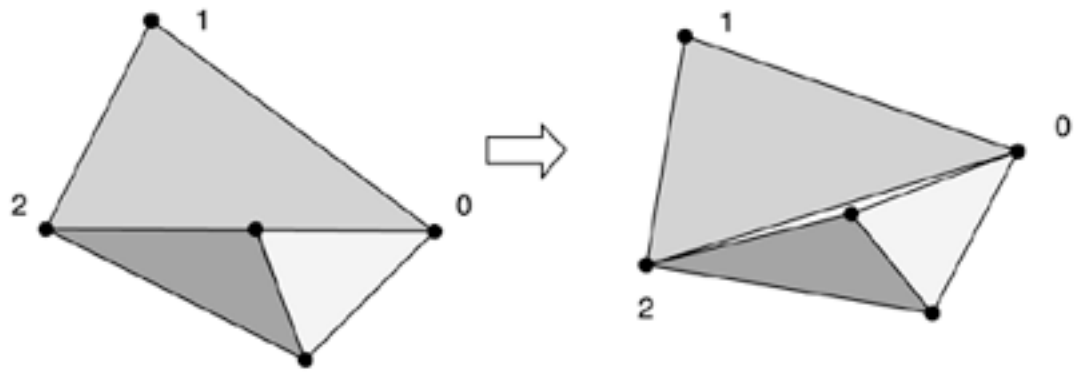
- **Clipping Polygons by a Line, Clip Methods of BSPBuilder.java**

```
/** Clips away the part of the polygon that lies in front
    of the specified line. The returned polygon is the part
    of the polygon in back of the line. Returns null if the
    line does not split the polygon. The original
    polygon is untouched.
*/
protected BSPPolygon clipFront(BSPPolygon poly, BSPLine line)
{
    return clip(poly, line, BSPLine.FRONT);
}

/** Clips away the part of the polygon that lies in back
    of the specified line. The returned polygon is the part
    of the polygon in front of the line. Returns null if the
    line does not split the polygon. The original
    polygon is untouched.
*/
protected BSPPolygon clipBack(BSPPolygon poly, BSPLine line) {
    return clip(poly, line, BSPLine.BACK);
}

/**
    Clips a BSPPolygon so that the part of the polygon on the
    specified side (either BSPLine.FRONT or BSPLine.BACK)
    is removed, and returns the clipped polygon. Returns null
    if the line does not split the polygon. The original
    polygon is untouched.
*/
protected BSPPolygon clip(BSPPolygon poly, BSPLine line,
    int clipSide)
{
    ArrayList vertices = new ArrayList();
    BSPLine polyEdge = new BSPLine();

    // add vertices that aren't on the clip side
    Point2D.Float intersection = new Point2D.Float();
    for (int i=0; i<poly.getNumVertices(); i++) {
        int next = (i+1) % poly.getNumVertices();
        Vector3D v1 = poly.getVertex(i);
        Vector3D v2 = poly.getVertex(next);
        int side1 = line.getSideThin(v1.x, v1.z);
        int side2 = line.getSideThin(v2.x, v2.z);
        if (side1 != clipSide) {
            vertices.add(v1);
        }
        if ((side1== BSPLine.FRONT && side2 == BSPLine.BACK) ||
            (side2== BSPLine.FRONT && side1 == BSPLine.BACK))
        {
            // ensure v1.z < v2.z
            if (v1.z > v2.z) {
                Vector3D temp = v1;
                v1 = v2;
                v2 = temp;
            }
            polyEdge.setLine(v1.x, v1.z, v2.x, v2.z);
            float f = polyEdge.getIntersection(line);
            Vector3D tPoint = new Vector3D(
                v1.x + f * (v2.x - v1.x),
                v1.y + f * (v2.y - v1.y),
                v1.z + f * (v2.z - v1.z));
            vertices.add(tPoint);
            // remove any created t-junctions
            removeTJunctions(v1, v2, tPoint);
        }
    }
    // Remove adjacent equal vertices. (A->A) becomes (A)
    for (int i=0; i<vertices.size(); i++) {
        Vector3D v = (Vector3D)vertices.get(i);
        Vector3D next = (Vector3D)vertices.get(
            (i+1) % vertices.size());
        if (v.equals(next)) {
            vertices.remove(i);
            i--;
        }
    }
    if (vertices.size() < 3) {
        return null;
    }

    // make the polygon
    Vector3D[] array = new Vector3D[vertices.size()];
    vertices.toArray(array);
    return poly.clone(array);
}
```

Implementing a 2D BSP Tree (17)

- **Removing T-Junction Gaps**

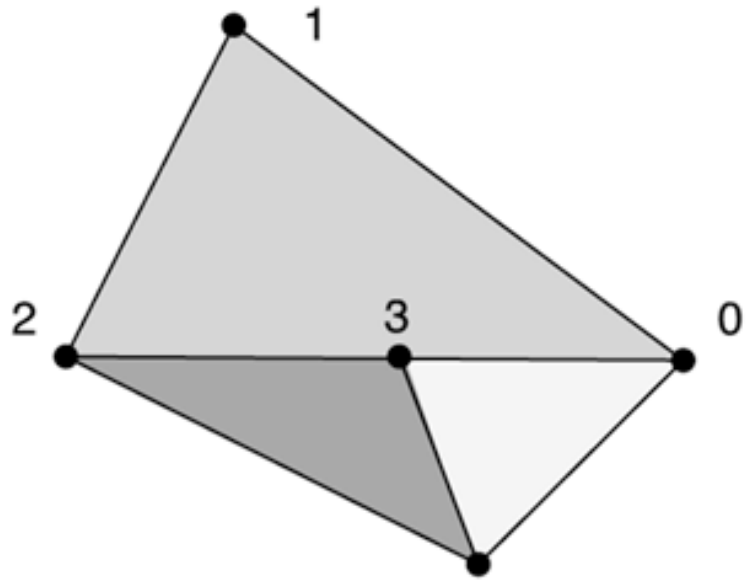
- Example for a T-junction gap. Due to floating-point inaccuracy, T-junctions can sometimes lead to gaps between polygons, sometimes only when viewed from certain angles. In this example, the large polygon has only three vertices. When the three polygons are transformed, a gap shows up between them. Most of the time, this gap isn't visible. However, occasionally, the gap appears as a missing pixel or two, or even a speckled line of missing pixels.



Implementing a 2D BSP Tree (18)

- To eliminate the gap between the T-junctions, you can add another vertex to the spanning polygon at the point of intersection. This is shown in this figure:

- To eliminate T-junction gaps, the large polygon has a fourth vertex it shares with the other two polygons.



- So, to eliminate T-junction gaps, the code inserts vertices into polygon edges whenever a polygon split occurs. This way, building the BSP tree never creates new T-junctions when it splits polygons.

Implementing a 2D BSP Tree (19)

- So, to eliminate T-junction gaps, the code inserts vertices into polygon edges whenever a polygon split occurs. This way, building the BSP tree never creates new T-junctions when it splits polygons.

- T-Junction Removal Methods of BSPBuilder.java

```
/**
    Remove any T-Junctions from the current tree along the
    line specified by (v1, v2). Find all polygons with this
    edge and insert the T-intersection point between them.
*/
protected void removeTJunctions(final Vector3D v1,
    final Vector3D v2, final Vector3D tPoint)
{
    BSPTreeTraverser traverser = new BSPTreeTraverser(
        new BSPTreeTraverseListener() {
            public boolean visitPolygon(BSPPolygon poly,
                boolean isBackLeaf)
            {
                removeTJunctions(poly, v1, v2, tPoint);
                return true;
            }
        }
    );
    traverser.traverse(currentTree);
}
```

```
/**
    Remove any T-Junctions from the specified polygon. The
    T-intersection point is inserted between the points
    v1 and v2 if there are no other points between them.
*/
protected void removeTJunctions(BSPPolygon poly,
    Vector3D v1, Vector3D v2, Vector3D tPoint)
{
    for (int i=0; i<poly.getNumVertices(); i++) {
        int next = (i+1) % poly.getNumVertices();
        Vector3D p1 = poly.getVertex(i);
        Vector3D p2 = poly.getVertex(next);
        if ((p1.equals(v1) && p2.equals(v2)) ||
            (p1.equals(v2) && p2.equals(v1)))
        {
            poly.insertVertex(next, tPoint);
            return;
        }
    }
}
```

Implementing a 2D BSP Tree (20)

- This is the brute-force approach to finding matching T-junctions: Just check every edge of every polygon in the tree. Admittedly, this could be accomplished faster by just searching for polygons that share an edge with the line of the T-junction, but this works as well.

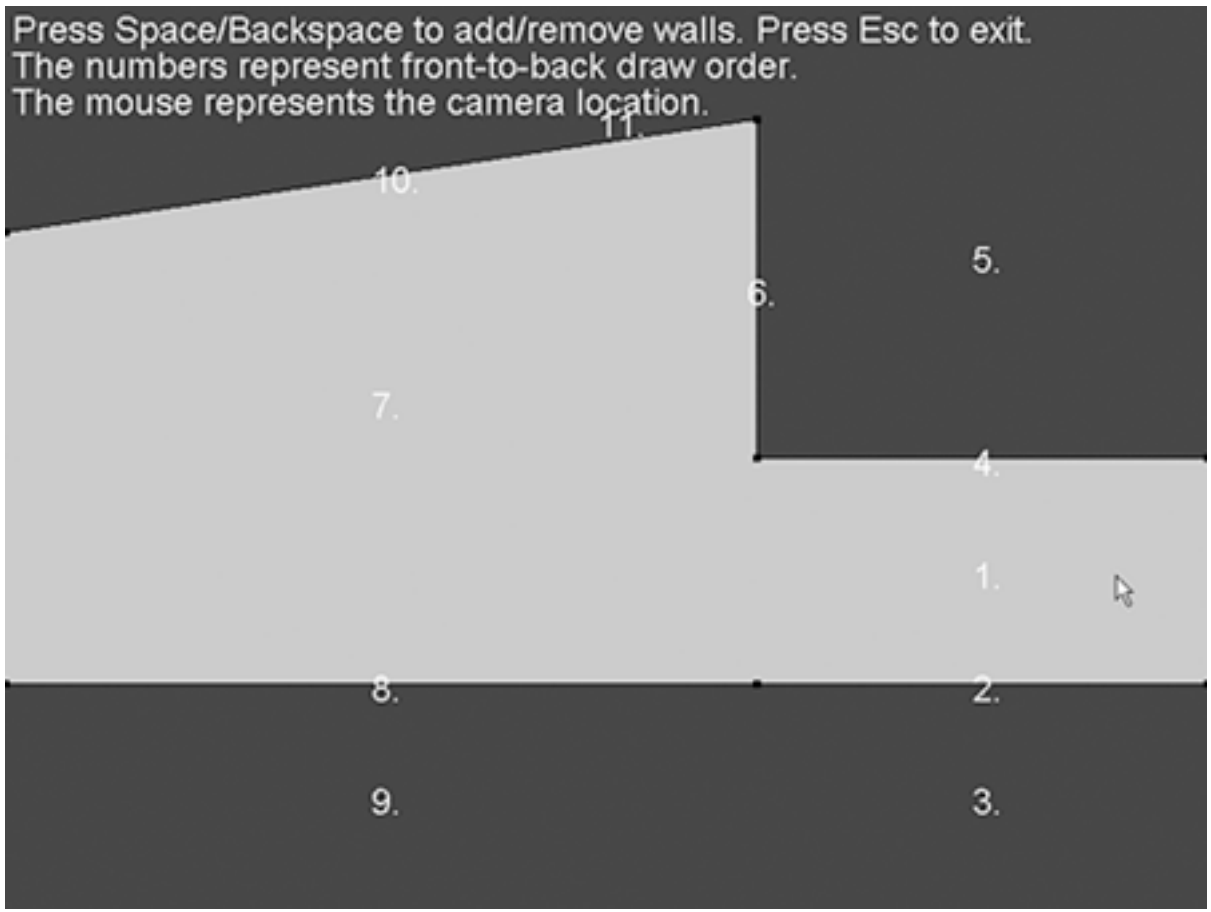
- Notice that the BSP tree building removes only T-junctions that are created from clipped polygons. In other words, it assumes that polygons passed to the BSP builder don't have any T-junctions.

Implementing a 2D BSP Tree (21)

```
public void createPolygons() {
    // The floor polygon
    BSPPolygon floor = new BSPPolygon(new Vector3D[] {
        new Vector3D(0,0,0), new Vector3D(0,0,600),
        new Vector3D(800,0,600), new Vector3D(800,0,0)
        }, BSPPolygon.TYPE_FLOOR);
    polygons.add(floor);

    // vertices defined from left to right as the viewer
    // looks at the wall
    BSPPolygon wallA = createPolygon(
        new BSPLine(0, 150, 500, 75), 0, 300);
    BSPPolygon wallB = createPolygon(
        new BSPLine(500, 75, 500, 300), 0, 300);
    BSPPolygon wallC = createPolygon(
        new BSPLine(500, 300, 800, 300), 0, 300);
    BSPPolygon wallD = createPolygon(
        new BSPLine(800, 450, 0, 450), 0, 300);
    polygons.add(wallA);
    polygons.add(wallB);
    polygons.add(wallC);
    polygons.add(wallD);
}

public BSPPolygon createPolygon(BSPLine line, float bottom,
    float top)
{
    return new BSPPolygon(new Vector3D[] {
        new Vector3D(line.x1, bottom, line.y1),
        new Vector3D(line.x2, bottom, line.y2),
        new Vector3D(line.x2, top, line.y2),
        new Vector3D(line.x1, top, line.y1)
        }, BSPPolygon.TYPE_WALL);
}

public void buildTree() {
    BSPTreeBuilder builder = new BSPTreeBuilder();
    bspTree = builder.build(polygons.subList(0, numWalls+1));
}
```

- **Testing the BSP Tree**
- **BSPPolygons** are created to match the previous floor plan example. One giant floor that covers the entire area is created, which is unrealistic in the real-world (you wouldn't want non-visible floors that are behind walls), but this works well as an example.
- The rest of BSPTest2D draws the polygons from a bird's-eye perspective.

Implementing a 2D BSP Tree (22)

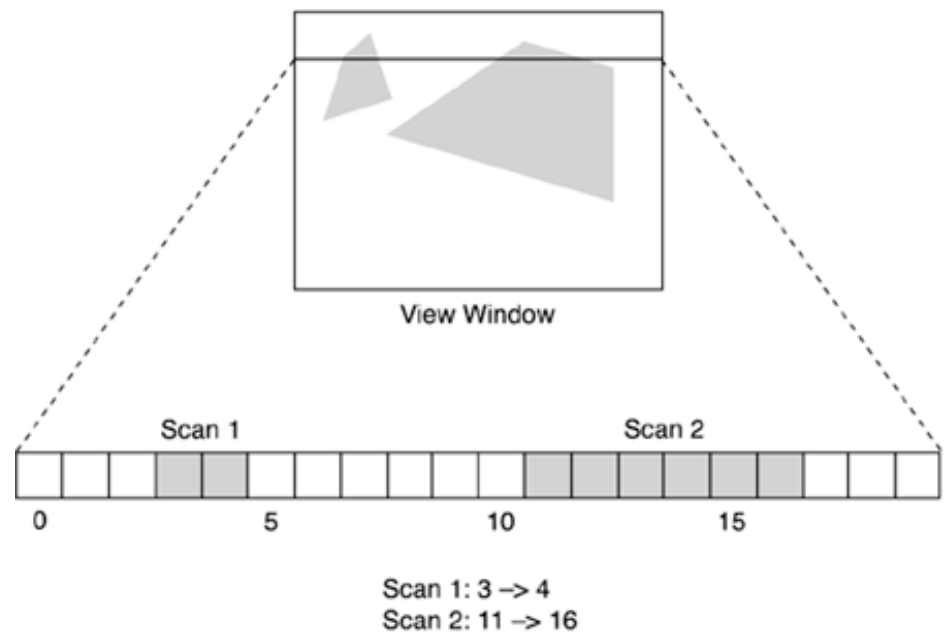


- The mouse represents the camera's location, and the numbers represent the order in which the polygons are drawn from the camera location.

Source: BSPTest2D.java

Drawing Polygons Front to Back

- Now that you have a BSP tree to traverse polygons from front to back, to draw polygons front to back you need to do two things:
  - Avoid drawing over pixels that are already on the screen.
  - Easily check whether the view is filled (whether every pixel is drawn).
- Just check and set the z-buffer—but...
- Converting polygons to horizontal scans: easily keep track of which scans in the view window have already been drawn.
  To do this, keep a list of scans
  for each horizontal row of pixels in
  the view window.



Drawing Polygons Front to Back (2)

- New polygon, after scan-converting you add that scan to the list of scans in the view window.
- An example of this is in this figure. The polygon scan is shortened so that it is not drawn over what is already in the view. Also, the list of scans in the view window for that row is merged to become one scan.