

## John Romero Programming Proverbs

- 6. “As soon as you see a bug, you fix it. Do not continue on. If you don’t fix your bugs your new code will be built on a buggy codebase and ensure an unstable foundation.”
- John Romero, “The Early Days of Id Software - John Romero @ WeAreDevelopers Conference 2017”

# Visportals and BSP trees

- consider this map
  - the walls are in red

chisel/maps/spiral.txt

```
define 1 room 1
define 2 room 2
define 3 room 3
define 4 room 4
define 0 monster monster_demon_imp
define H monster monster_demon_hellknight
define S worldspawn
```

# Visportals and BSP trees



chisel/maps/spiral.txt

```
#####
#1      #2      #
#      .      #
#      .      #
#      .      #
#      #      #
#.....#####.....###
#3      #4      #
#      #  O O O  #
#      S      #
#      #  O O O  #
#      #      #
#####
```

## Visportals and BSP trees

- we can see that we are spawned in room 3
  - the imps are in room 4
- let us build this map with chisel

```
$ cd
$ cd Sandpit/chisel/python
$ ./developer-txt2map ../maps/spiral.txt
txt2pen: pass
Total rooms = 4
Total cuboids = 769
Total cuboids expanded (optimised) = 0
Total entities used = 89 entities unused = 4007
Total brushes used = 769
pen2map: pass
```

## Visportals and BSP trees

- now compile the map within dhewm3
  - and play it!
  
- notice when you are spawned at the beginning of the game
  - turn and if you face room 4
  - watch the fps drop !
  - why is this the case?

## Visportals and BSP trees

- now we can implement visportals for open doors
  - compile the map and play the game
- notice after the player is spawned you can face room 4
  - and what happens?
  - why?

# Algorithm for splitting an area into a BSP tree

```
PROCEDURE makeTree (polyList: polygon) : tree;  
VAR  
    root: polygon ;  
    backList, frontList: polygonP ;  
    p, backPart, frontPart: polygon ;  
BEGIN  
    IF polyList = NIL  
    THEN  
        RETURN NIL  
    ELSE
```

## Algorithm for splitting an area into a BSP tree

```

root := selectAndRemovePolygon (polyList) ;
backList := NIL ;
frontList := NIL ;
FOR p in polyList DO
    IF p is in front of root
    THEN
        frontList := addToList (frontList, p)
    ELSIF p is behind root
    THEN
        backList := addToList (backList, p)
    ELSE
        (* polygon, p, must be split as it spans across, root.  *)
        splitPolygon (p, root, frontPart, backPart) ;
        frontList := addToList (frontList, frontPart) ;
        backList := addToList (backList, backPart) ;
    END
END ;
RETURN combineTree (makeTree (frontList),
                    root,
                    makeTree (backList))
END
END makeTree;

```



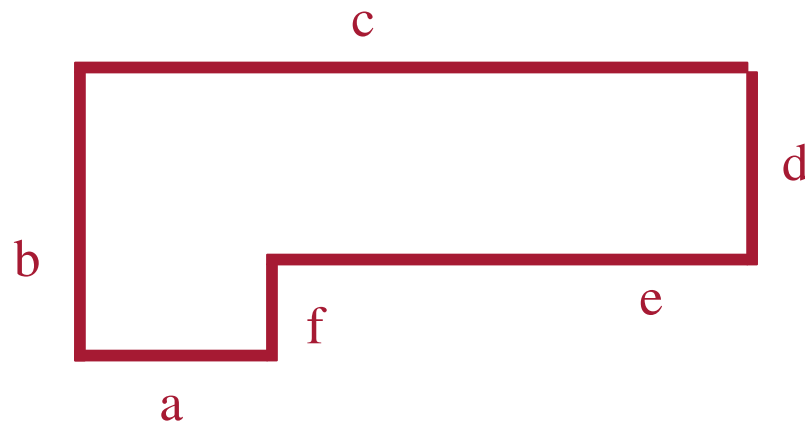
## BSP trees

- consider the following walls in a map



# BSP trees


- we will label each line



## BSP trees

- we initialise our `polyList` to the labeled lines
- and the tree to empty
  - the tree will be a binary tree, built by a number of 3 item lists
    - in the list, element 0, is the left, which represents behind,
    - middle, element 1, is the node label
    - right, element 2, means forward
    - forward means right for a vertical line
    - forward means above a horizontal line
  - the algorithm will create a binary tree of ordered lines
  - the algorithm keeps recursing until each node is a label (rather than a list)
  - and each leaf contains a label (rather than a list)

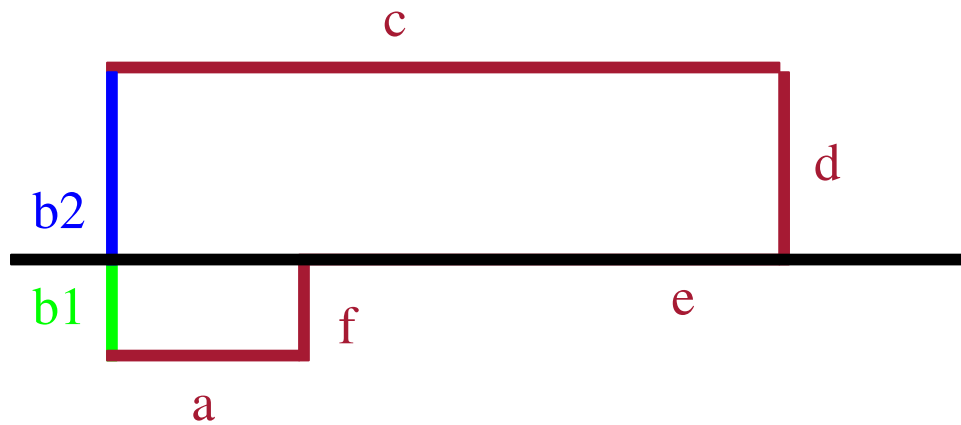
# BSP trees



```
polyList = [a, b, c, d, e, f]  
tree     = []
```

## BSP trees

- we will choose an existing line to perform the split (marked in black)

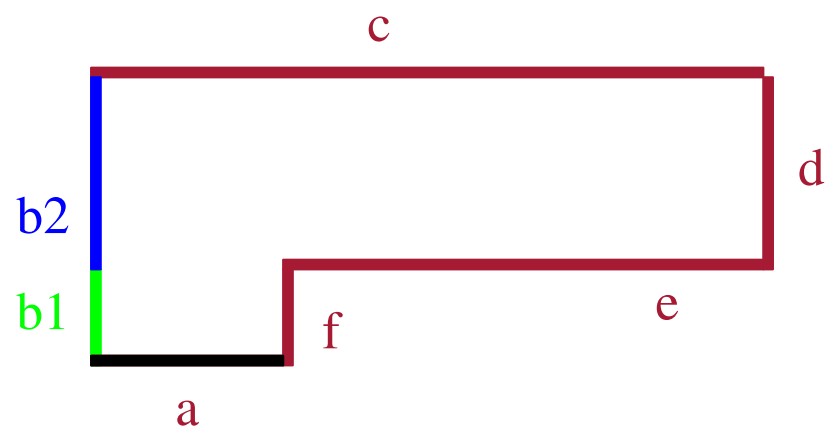


## BSP trees


```
polyList = [a, b1, b2, c, d, e, f]
tree = [makeTree ([b1, a, f]), # left node, below
        e, # pivot node
        makeTree ([b2, c, d])] # right node, above
```

■ we will call `makeTree ([b1, a, f])`

**makeTree ([b1, a, f])**



## **makeTree ([b1, a, f])**

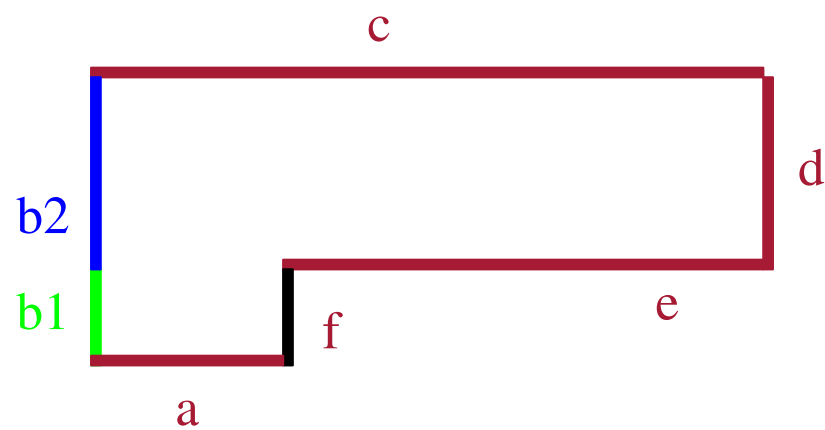


```
polyList = [b1, a, f]

tree = [combineTree ([],    # left, below
                    a,
                    makeTree ([b1, f])] # right, above
```



**makeTree ([b1, f])**



## makeTree ([b1, f])

```
tree = [b1, f, []]
```

therefore the caller to makeTree now creates the tree as:

```
tree = [combineTree ([], # left, below  
                    a, # pivot node  
                    makeTree ([b1, f]))] # right, above  
tree = [a, b1, f]
```

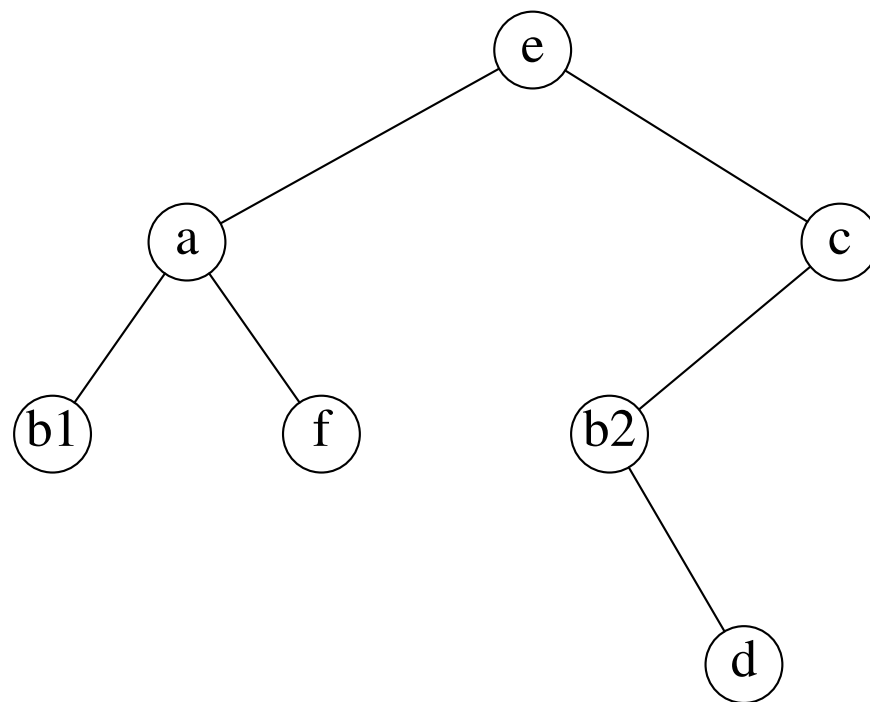
## makeTree ([b1, f])

- and the caller to this makeTree

- ```
tree = [makeTree ([a, b1, f]), # left, below
        e,
        makeTree ([b2, c, d])] # right, above
```

- ```
tree = [[a, b1, f], # left, below
        e,
        makeTree ([b2, c, d])] # right, above
```

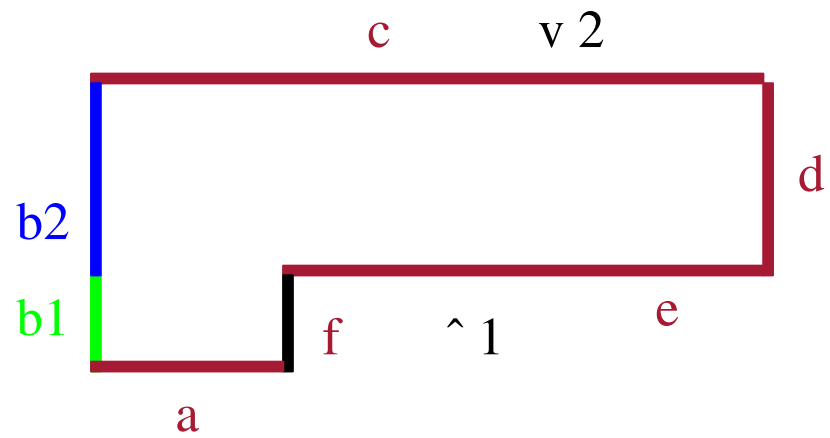
# BSP tree



## Display BSP Tree algorithm

```
PROCEDURE displayTree (tree)
BEGIN
  IF tree # NIL
  THEN
    IF Viewer is in front of tree->line
    THEN
      (* display back child, root, front child. *)
      displayTree (tree->left) ; (* back. *)
      displayLine (tree->line) ;
      displayTree (tree->right) (* front. *)
    ELSE
      (* display front child, root and back child. *)
      displayTree (tree->right) ; (* front. *)
      displayLine (tree->line) ;
      displayTree (tree->left) (* back. *)
    END
  END
END displayTree ;
```

# Display BSP Tree algorithm



## Line (polygon) display order using the bsp tree and display algorithm

- given position 1 (facing upwards)
  - c, b2, d, e, b1, a, f
  
- given position 2 (facing downwards)
  - b1, a, f, e, b2, d, c