Rukai Zhao

Project 2: Face mesh operation

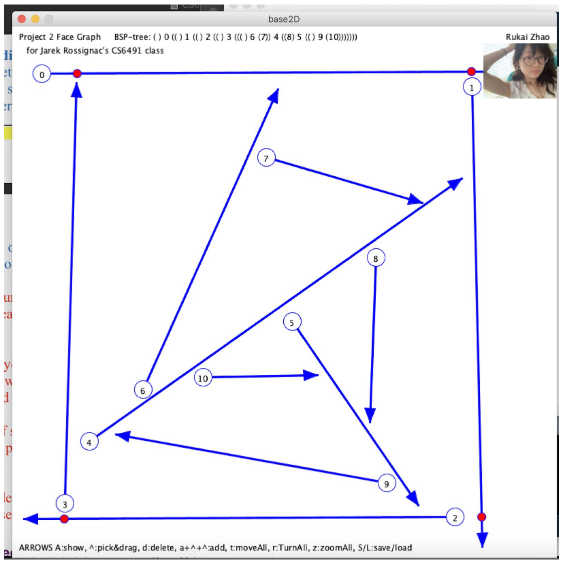
(In this project, I implemented all the features in the 2D version of the face mesh.)

Objectives:

In this project, we need to construct a BSP tree for all input 2D arrows to determine whether the newly added arrow is on the left side or right side of the current added arrow on the plane. After finishing BSP tree, we need to use the data structure discussed during lecture to construct a face mesh. After that, we need to implement the containment tree.

For the PMC(Point-Membership classification),we find which part the point belongs to on the face graph. We have an origin, a normal, left child, right child for each node. When adding point into the tree, we check whether the point is inside which part of the face, in that face we find whether the point is on the left or on the right.

Input:



```
Project 2 Face Graph     BSP-tree: ( ) 0 (( ) 1 (( ) 2 (( ) 3 ((( ) 6 (7)) 4 ((8) 5 (( ) 9 (10)))))))     Rukai Zhao
  for Jarek Rossignac's CS6491 class
ARROWS A:show, ^:pick&drag, d:delete, a+^+^:add, t:moveAll, r:TurnAll, z:zoomAll, S/L:save/load
```

When running the code without pressing any keys, it will show all the 2D input arrows on the screen. Delete the arrows using ‘d’ keys and adding arrows using ‘a’ keys. When pressing ‘f’ key, it will start computing the face-mesh. Pressing ‘e’, you can see the red border line on the screen. Pressing ‘b’, you can see all the yellow good faces. Pressing ‘n’ to perform the next operator and ‘s’ to perform the swing operator.

Computing BSP-tree:

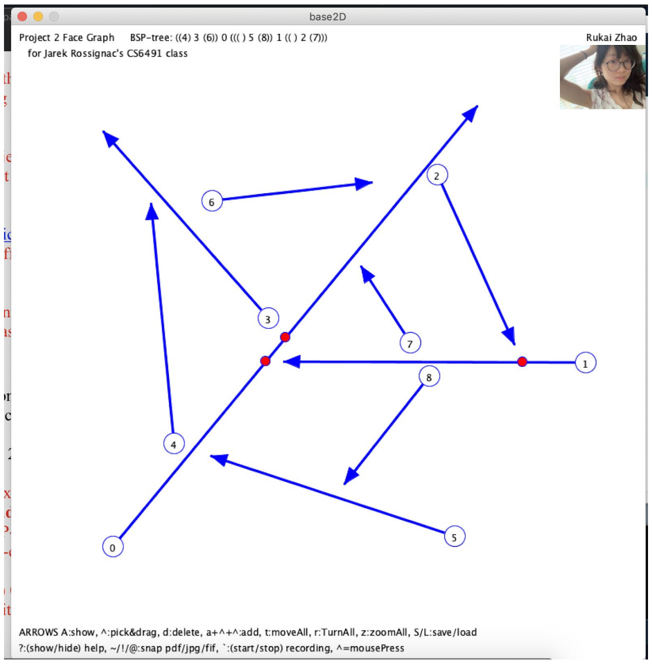
Binary Space Partition (BSP) Trees are binary trees which partition the space into two half spaces. Arrows partition the space into two half spaces. The left side of the space is defined as bad or “water” and the right side of the space is defined as good or “land”. By using BSP tree, we are able to know which part of the half space partitioned by the input arrow is good(land) face and which part is bad(water).

When computing the BSP tree, I have a class named Node, which stores the index of this node, and its left child node and right child node. When building the bsp tree, I have a function called Node build_bsp(ArrayList<Integer> input). Each time in the function, I split the input arrows into two parts, the left and the right, for each I recursively passing to build_bsp. It will reach the leaf if the input array size is zero.

Pseudocode:

```
Node build_bsp(ArrayList<Integer> input){
        if(input size is empty)
                Return null
        Else{
          Root = input[0]
          Array left
          Array right
         For each remaining elements in input:{
                Vec root_line =  V(P(P.G[2*root]),P(P.G[2*root+1]));
                vec next_line = V(P(P.G[2*root]),P.G[2*input.get(i)]);
                 if(det(root_line,next_line)>0)
                         right.add(input.get(i));
                 else if(det(root_line,next_line)<0)
                        left.add(input.get(i));}

         node.index = root;
         node.left = build_bsp(left);
         node.right = build_bsp(right);
         Return node;
}
}
```


base2D

Project 2 Face Graph BSP-tree: ((4) 3 (6)) 0 ((( ) 5 (8)) 1 (( ) 2 (7)))

for Jarek Rossignac's CS6491 class

Rukai Zhao

ARROWS A:show, ^:pick&drag, d:delete, a+^+^:add, t:moveAll, r:TurnAll, z:zoomAll, S/L:save/load

?:(show/hide) help, ~/!/@:snap pdf/jpg/fif, `:(start/stop) recording, ^=mousePress

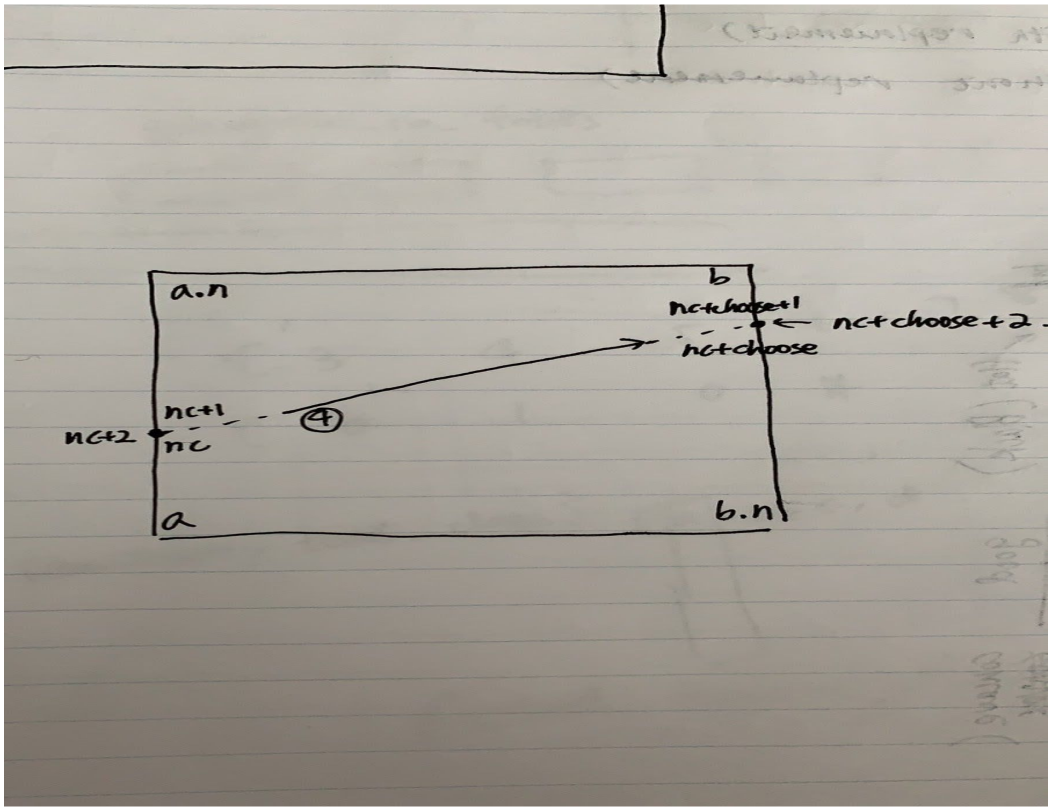
In this situation, the bsp tree is displaced on the head of the screen.

Computing Face-graph:

For implementing Face_graph, I implemented a class called class mesh. Inside the Mesh class I have:

```
Class Mesh{
int maxnv = 100*2*2*2*2*2*2*2*2;
pt[] G = new pt [maxnv]; //geometry(vertices) position
int[] V =new int [maxnv]; //corner to vertex indices
int[] F = new int[maxnv];//corner to face indices
int[] N = new int[maxnv];//next operation table
int[] S = new int[maxnv];//swing operation table
boolean[] I = new boolean[maxnv];//storing whether the corner is good or bad
int nf = 0;   //number of face
int nc = 0; //number of corners
int nv = 0;  // number of vertices

ArrayList <Integer> start = new ArrayList<Integer> (); //storing the good face starting corner
//which will be used later on for border loop calculations
}
```

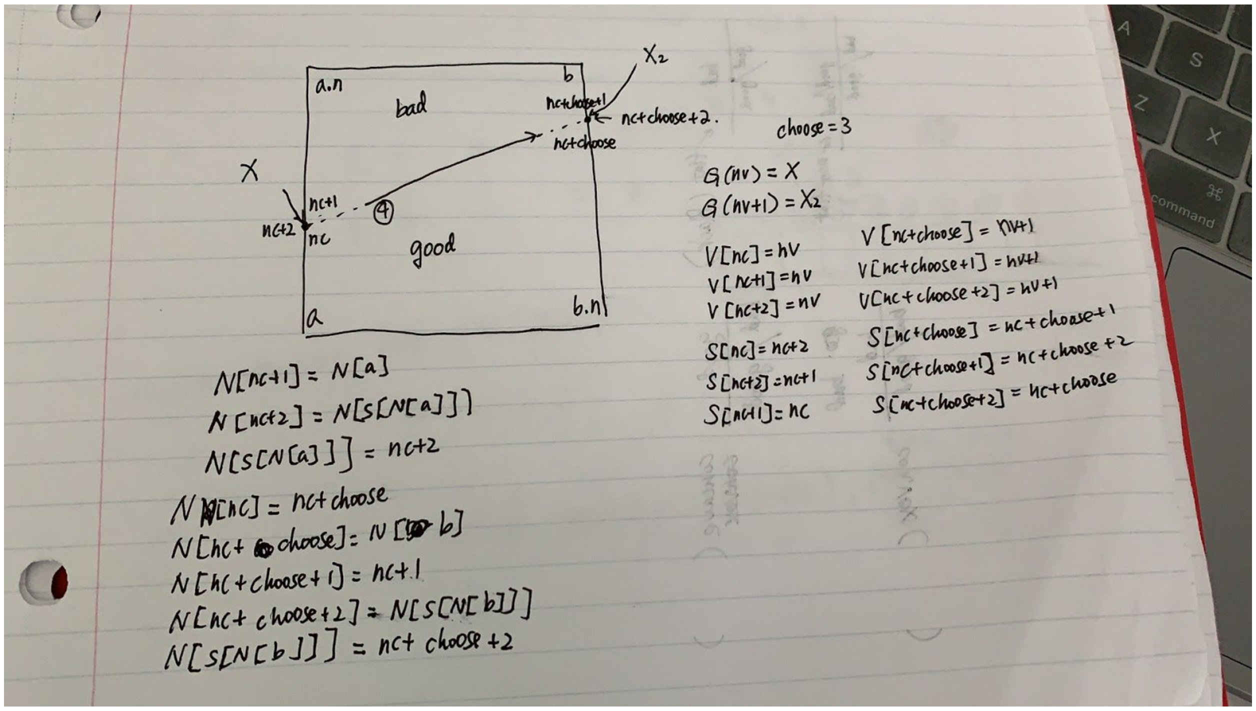


Suppose we have a face and we want to insert arrow 4 into the face, I have written a function called boolean split which to check whether the input arrow intersect with the edge contained corner a and corner a.n and with the edge contained corner b ad corner b.n. I have also written a function that calculates the intersection point.

Pseudocode:(find the intersection point when arrow AB intersect with the edge contained corner a and corner a.n, similar for the edge contained corner b ad corner b.n)

```
Distance = infinity
Best_a = none
For all the corner a in the face graph:
    If split(A,B, corner a):
        x=Intersection(Line(a,b),line(G[v[a]],G[v[n[a]]]
        If dot(AB,AX)<distance
            Best_a = a
```

After finding the best_a a corner and the intersections, I did the following things for constructing the array:



choose = 3

G(nv) = X
G(nv+1) = X_2

V[nc] = nv
V[nc+1] = nv
V[nc+2] = nv

V[nc+choose] = nv+1
V[nc+choose+1] = nv+1
V[nc+choose+2] = nv+1

S[nc] = nc+2
S[nc+2] = nc+1
S[nc+1] = nc

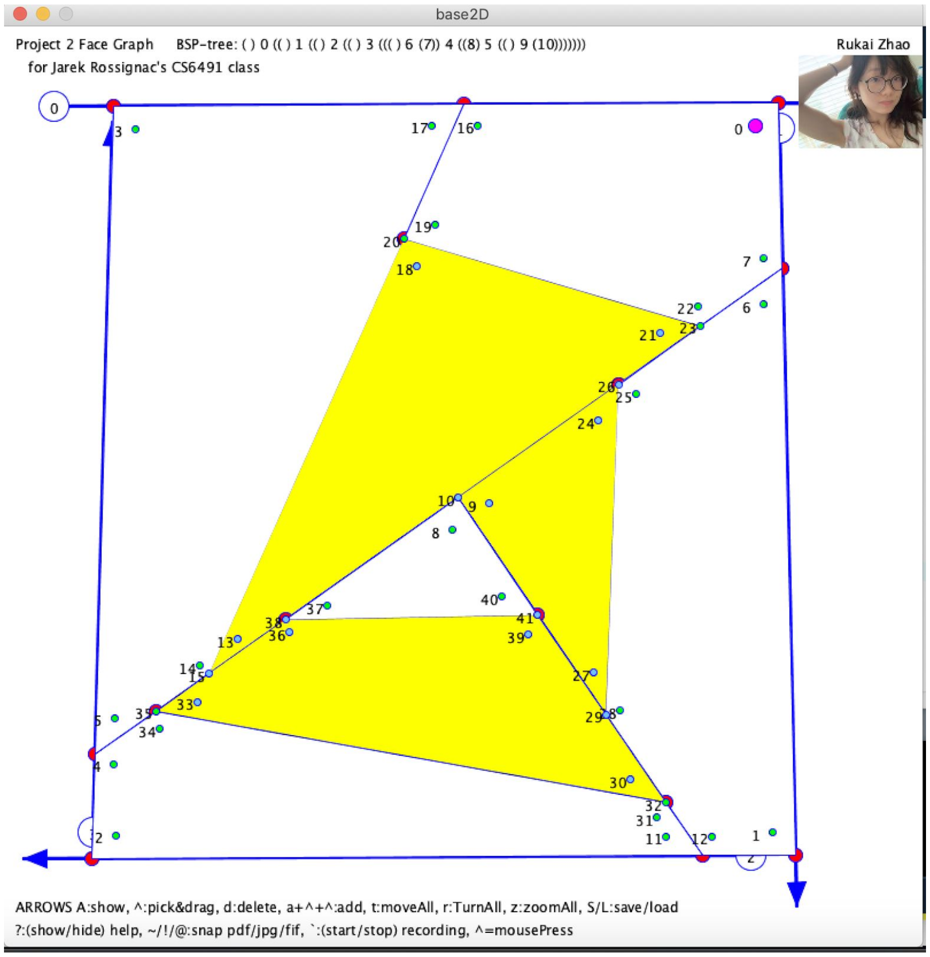
S[nc+choose] = nc+choose+1
S[nc+choose+1] = nc+choose+2
S[nc+choose+2] = nc+choose

N[nc+1] = N[a]
N[nc+2] = N[S[N[a]]]
N[S[N[a]]] = nc+2

N[nc] = nc+choose
N[nc+choose] = N[b]
N[nc+choose+1] = nc+1
N[nc+choose+2] = N[S[N[b]]]
N[S[N[b]]] = nc+choose+2

For computing F[c] for each corner, I loop through corner start with nc using next operator to set F[c] = F[a] and loop through corner start with nc+choose+1 using next operator to set F[c] = current face + 1. Setting F[nc+2] and F[nc+choose+2] to the face containing these to corners.

For computing I[c] for each corner, since corner nc is always in the good plane and nc+choose+1 is always in the bad face, I used next operator to denote all the corners in the good face following corner nc to be I[c]=true and used next operator to denote all the corners in the bad face following corner nc+1 to be I[c] = false. For corner nc+2, I looped through the face contained nc+2 to denote nc+2, same for nc+choose+2. For each loop I colored the good face yellow and the bad face white:


base2D

Project 2 Face Graph BSP-tree: ( ) 0 (( ) 1 (( ) 2 (( ) 3 ((( ) 6 (7)) 4 ((8) 5 (( ) 9 (10)))))) Rukai Zhao

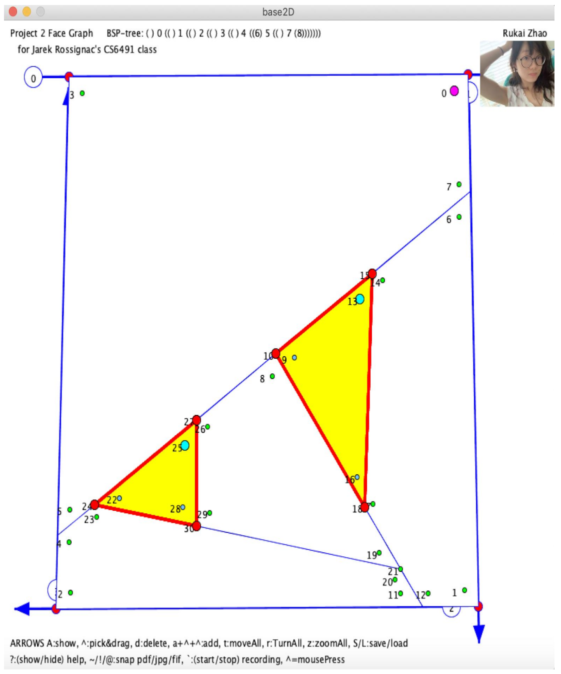
for Jarek Rossignac's CS6491 class

ARROWS A:show, ^:pick&drag, d:delete, a+^+^:add, t:moveAll, r:TurnAll, z:zoomAll, S/L:save/load

?:(show/hide) help, ~/!/@:snap pdf/jpg/fif, `:(start/stop) recording, ^=mousePress

The small green corner denoted that this corner is on the bad face and the small blue corner denoted that this corner is on the good face. The 180 degree corner is drawn on the intersection points. The big magenta corner on the top right is where the corner operation start. For example, you can press ‘n’, the magenta will move on to the next corner and press ‘s’, the magenta will move on to the swing corner. Since I denote that the corner on the very outer loop to be only two or three, pressing swing on a vertex only have one corner will stay on the same corner.

Border edges:



base2D

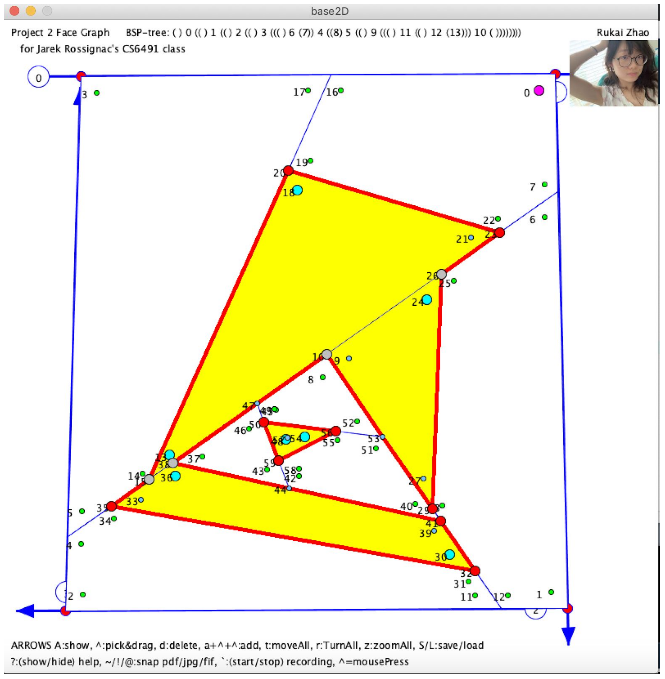
Project 2 Face Graph BSP-tree: ( ) 0 (( ) 1 (( ) 2 (( ) 3 (( ) 4 ((6) 5 (( ) 7 (8)))))))

Rukai Zhao

for Jarek Rossignac's CS6491 class

ARROWS A:show, ^:pick&drag, d:delete, a+^+^:add, t:moveAll, r:TurnAll, z:zoomAll, S/L:save/load

?:(show/hide) help, ~/!/@:snap pdf/jpg/fif, `:(start/stop) recording, ^=mousePress



base2D

Project 2 Face Graph BSP-tree: ( ) 0 (( ) 1 (( ) 2 (( ) 3 ((( ) 6 (7)) 4 ((8) 5 (( ) 9 ((( ) 11 (( ) 12 (13))) 10 ( ))))))))

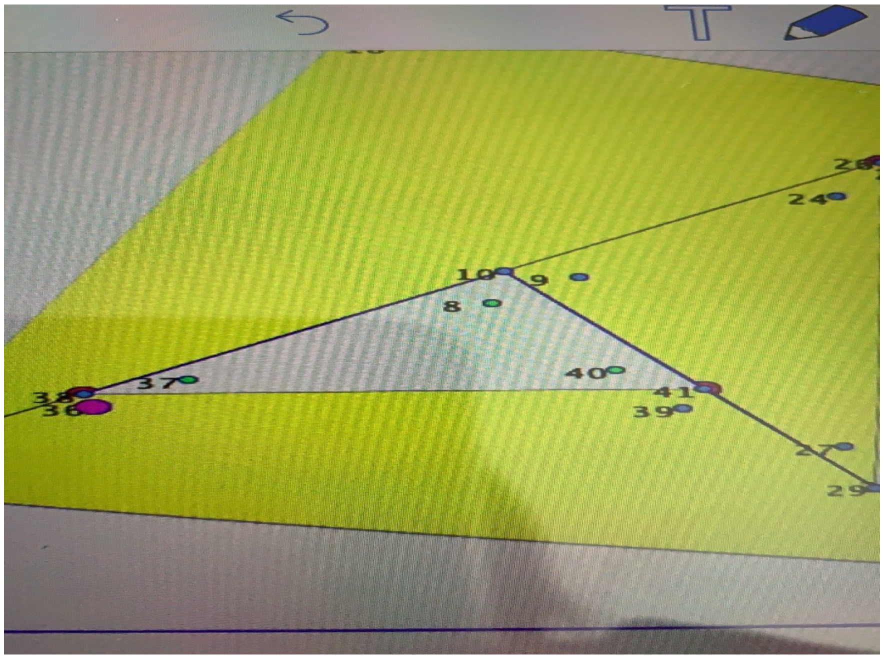
Rukai Zhao

for Jarek Rossignac's CS6491 class

ARROWS A:show, ^:pick&drag, d:delete, a+^+^:add, t:moveAll, r:TurnAll, z:zoomAll, S/L:save/load

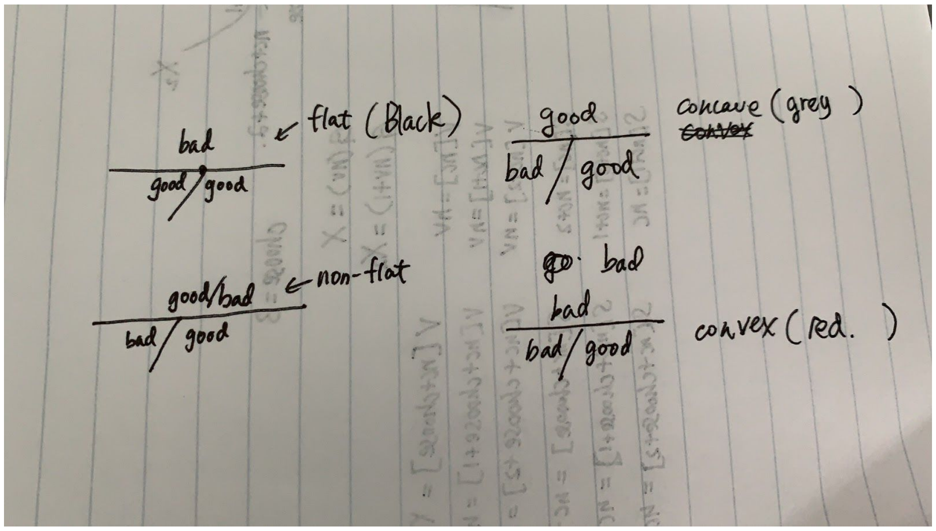
?:(show/hide) help, ~/!/@:snap pdf/jpg/fif, `:(start/stop) recording, ^=mousePress

Since for each good face I have kept a starting point denoted the start corner of the border loop, then I check for whether the corner and next corner are on the border, if yes, I will draw the red line, if not I will swing swing to the corner and draw the line with that corner and its next corner.



Consider the corner 39, it is not on the border for line(39,29), it is 41 is on the border since line(41,9) is a border edge. In this case, 39 has to swing to 41 to continue the border loop.

From the two images above, the red vertex is the convex non-flat vertex and the grey vertex is the concave non-flat vertex.



I used the above cases for computing the flat (colored black) vertex and concave non-flat vertex (colored grey) and convex non-flat vertex(colored red).

Containment tree:
Starting with all the border loops we have defined in the previous section, there are several loops maybe repetitive. Storing all the loops in an ArrayList border loop, removing repetition if two loops have the exact same vertices. Then we have to define which loop are inside which loop or they are parallel.
For each random vertex, we first find which loop it is inside, for example if we have already computed three loops, then for each loop computes whether it is inside that loop. That's how I found which loop the vertex is inside. To check whether a loop is inside a loop, I loop through all the vertices in one loop and check if the vertices on that loop all contained inside the outer loop. If so, check the inner loop with the rest of the loop. If not, that means these two loops are not contained within each other.