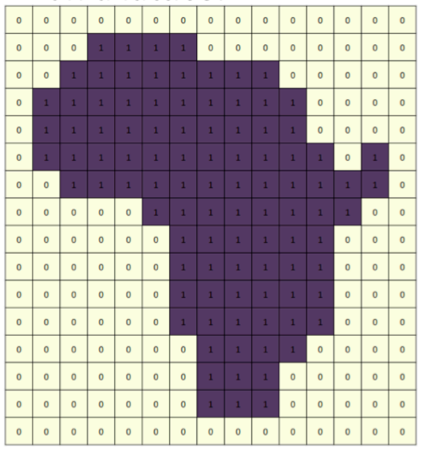
Coloring Outside the Lines

(an attempt to describe tool path generation without putting everybody to sleep)

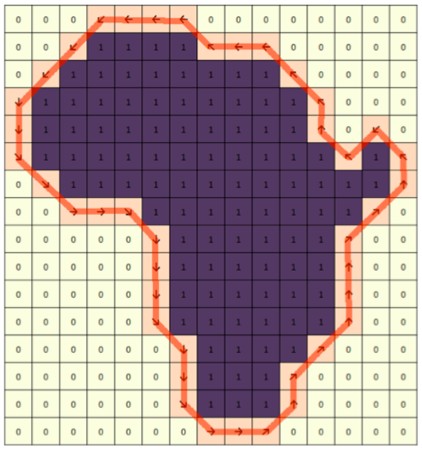
MIT MAS.S62 Spring 2012
How To Make Something That Makes (Almost) Anything
Steve Leibman: sleibman@alum.mit.edu

The Problem

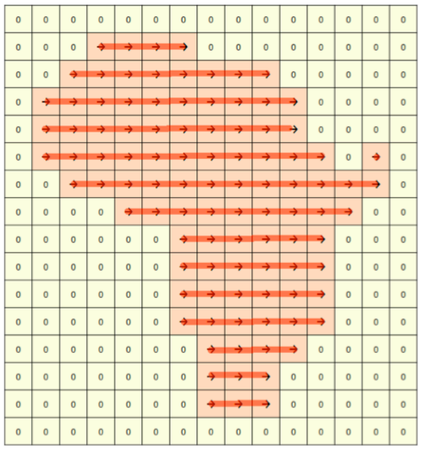
Given a digital description of geometry on a lattice:

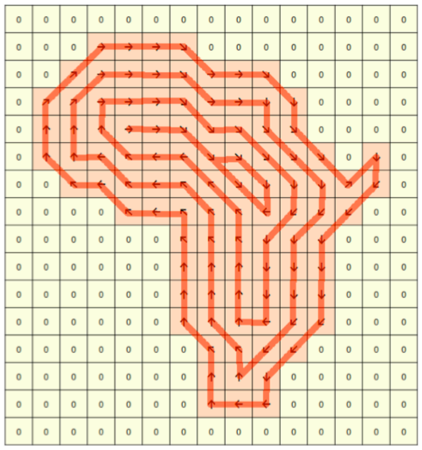


Find the best tool path to carve it:



Or fill it:





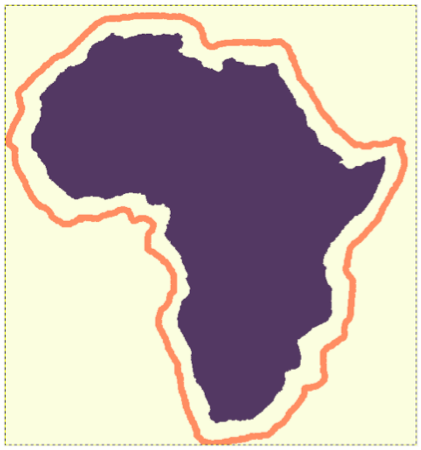
Motivation

This is the core of the problem we need to solve in order to use all of our manufacturing machines – both additive and subtractive.

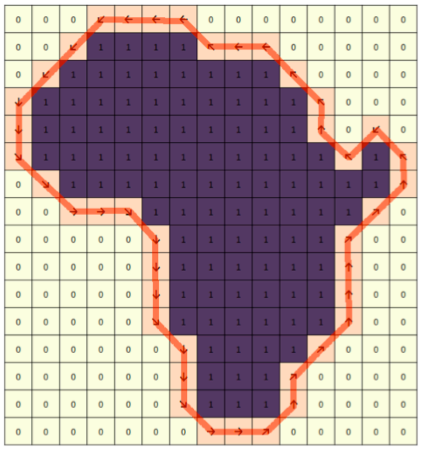
Not Covered
(because you really don't need it)

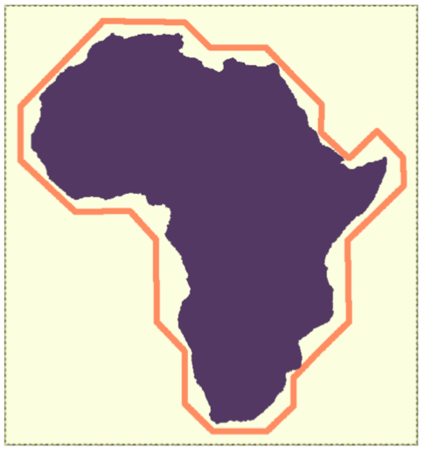
Given a vector / floating point description:

Find the best tool path to carve it:



Discrete and continuous versions are identical *for a given resolution* (which is what matters in our world)

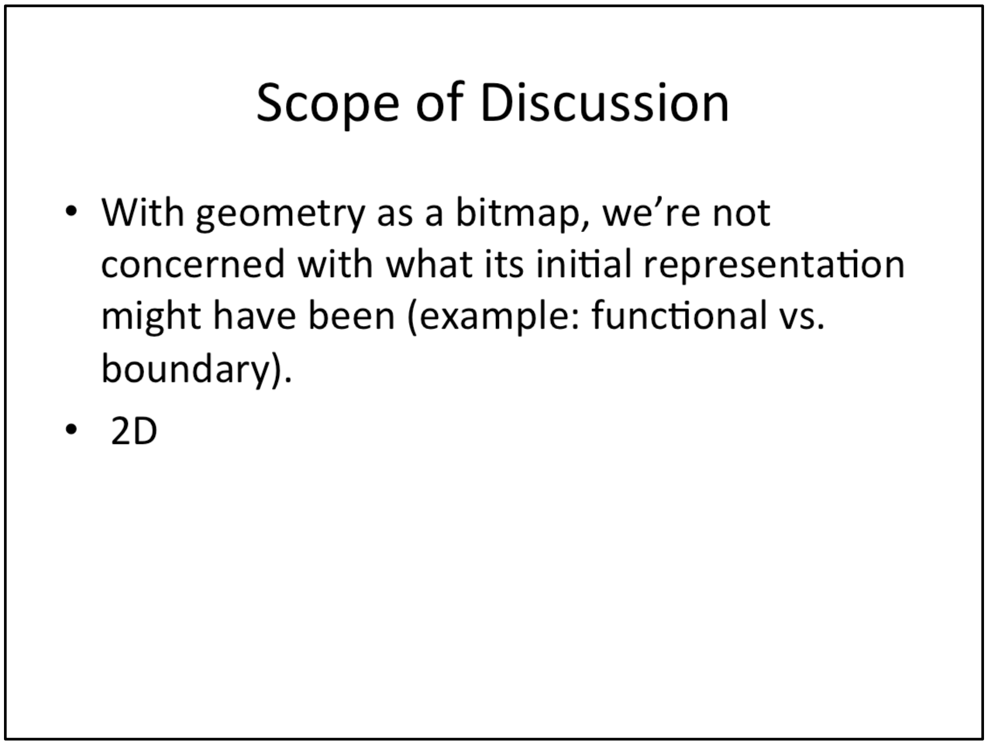




It is, of course, still possible to do all of this with floating point representations.

In this discussion we restrict ourselves to the case in which the geometry is represented on a lattice:

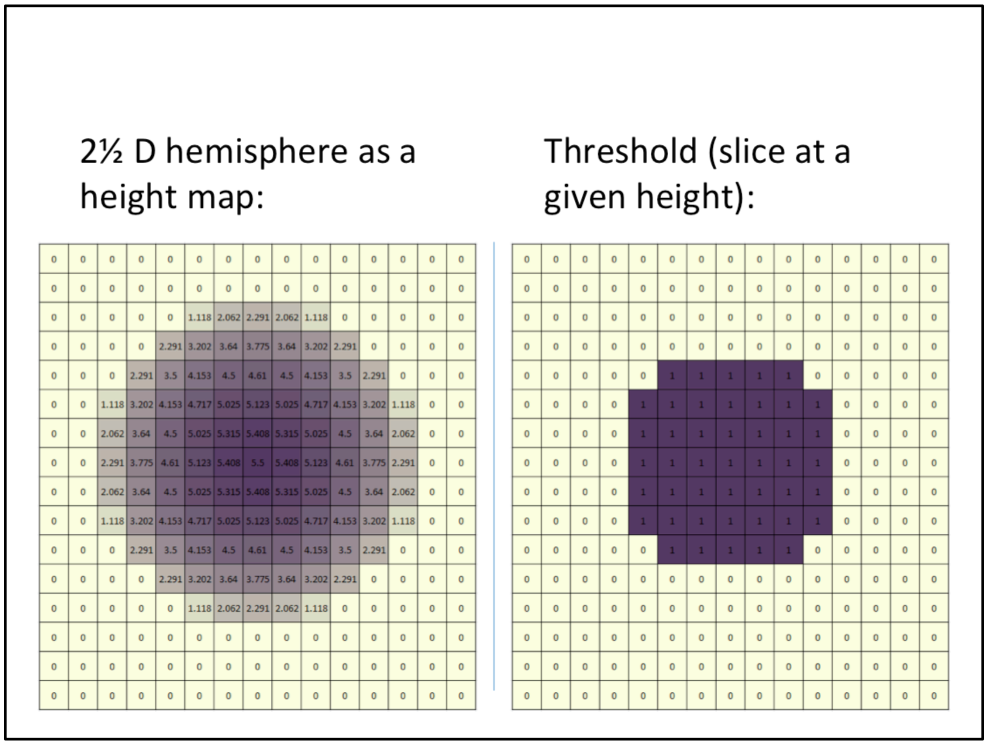
1. Machines are discrete anyway
2. Discrete algorithms are clean, simple, linear time, bulletproof.



Scope of Discussion

- With geometry as a bitmap, we're not concerned with what its initial representation might have been (example: functional vs. boundary).
- 2D

With geometry as a bitmap, we're not concerned with what its initial representation might have been (example: functional vs. boundary).
2D. Most of our practical 3D tool paths are a sequence of 2D problems.



2½ D hemisphere as a height map:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1.118	2.062	2.291	2.062	1.118	0	0	0	0	0
0	0	0	0	2.291	3.202	3.64	3.775	3.64	3.202	2.291	0	0	0	0
0	0	0	2.291	3.5	4.153	4.5	4.61	4.5	4.153	3.5	2.291	0	0	0
0	0	1.118	3.202	4.153	4.717	5.025	5.123	5.025	4.717	4.153	3.202	1.118	0	0
0	0	2.062	3.64	4.5	5.025	5.315	5.408	5.315	5.025	4.5	3.64	2.062	0	0
0	0	2.291	3.775	4.61	5.123	5.408	5.5	5.408	5.123	4.61	3.775	2.291	0	0
0	0	2.062	3.64	4.5	5.025	5.315	5.408	5.315	5.025	4.5	3.64	2.062	0	0
0	0	1.118	3.202	4.153	4.717	5.025	5.123	5.025	4.717	4.153	3.202	1.118	0	0
0	0	0	2.291	3.5	4.153	4.5	4.61	4.5	4.153	3.5	2.291	0	0	0
0	0	0	0	2.291	3.202	3.64	3.775	3.64	3.202	2.291	0	0	0	0
0	0	0	0	0	1.118	2.062	2.291	2.062	1.118	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

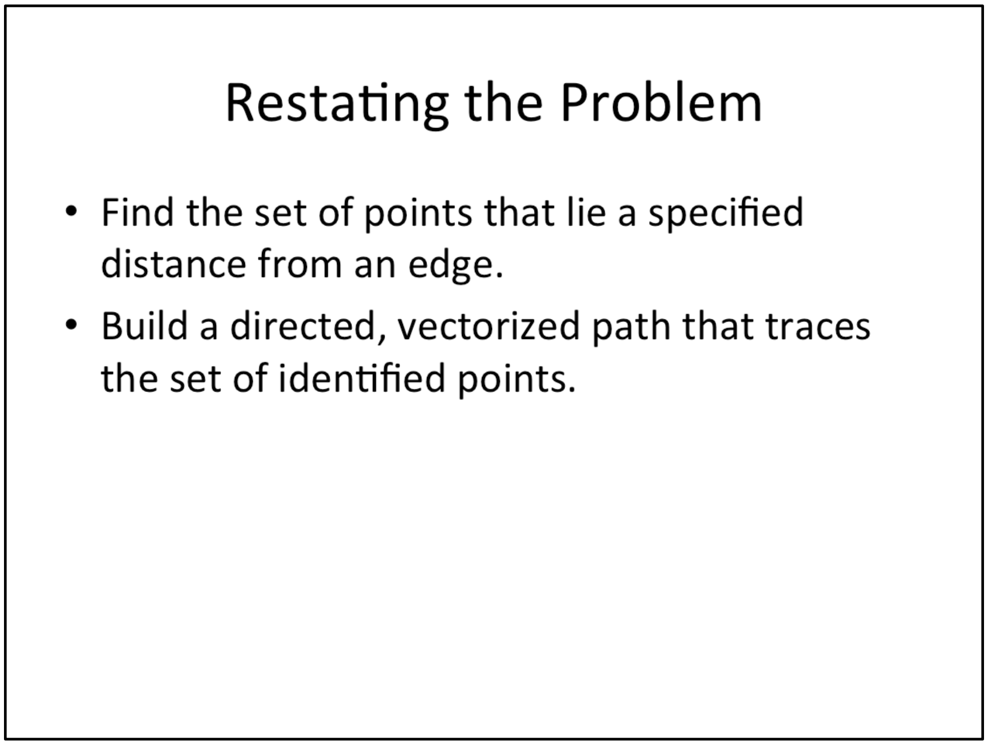
Threshold (slice at a given height):

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	0	0	0	0	0
0	0	0	0	1	1	1	1	1	1	1	0	0	0	0
0	0	0	0	1	1	1	1	1	1	1	0	0	0	0
0	0	0	0	1	1	1	1	1	1	1	0	0	0	0
0	0	0	0	1	1	1	1	1	1	1	0	0	0	0
0	0	0	0	1	1	1	1	1	1	1	0	0	0	0
0	0	0	0	0	1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

In this example, consider a 2-1/2 D hemisphere created on a vertical axis milling machine.
We might do rough cuts along horizontal plane slices, and then use a ball-end mill to do finish cuts along vertical plane slices.
In both cases, we only have to solve a series of 2D problems.

Note that with current technology for processes like stereolithography and fused-deposition modeling, the nature of the device forces us to treat it as a sequence of parallel 2D problems anyway.

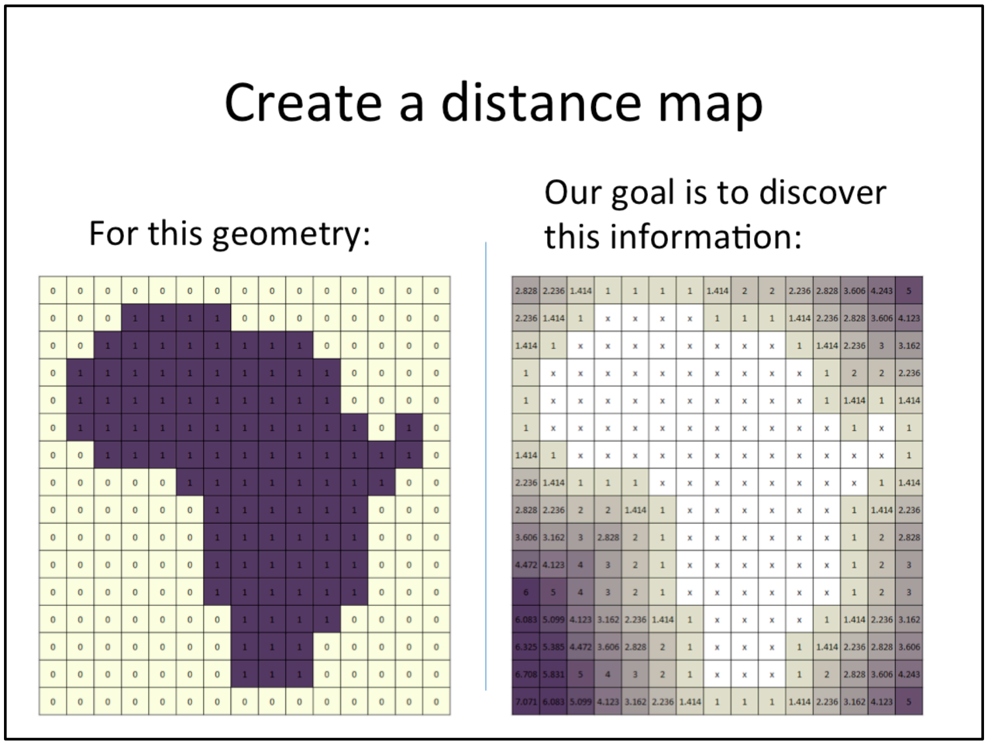
The picture on the right shows a bitmap of binary values. This represents a horizontal plane sliced through the hemisphere object at a given height. This was obtained by applying the threshold operation “Z>4.0” to the values in the picture on the left.



Restating the Problem

- Find the set of points that lie a specified distance from an edge.
- Build a directed, vectorized path that traces the set of identified points.

Note that the set of points we're interested in happens to be the "level set", a.k.a. "isoline" solution to a distance function.



Create a distance map

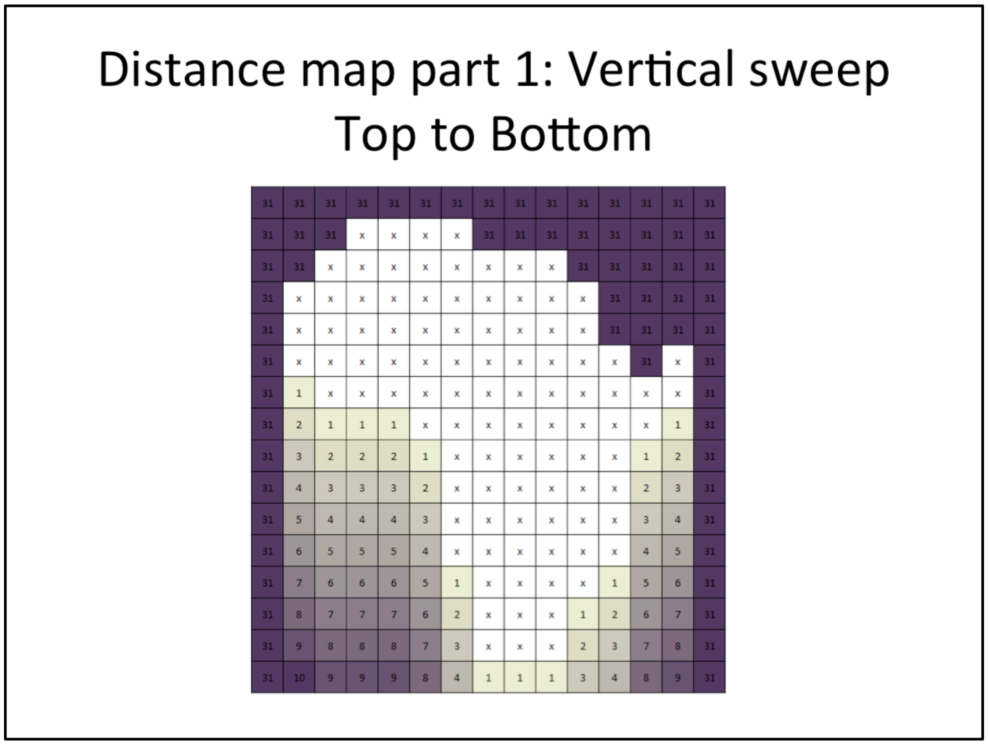
For this geometry:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1	1	0	0	0	0	0
0	1	1	1	1	1	1	1	1	1	1	0	0	0	0
0	1	1	1	1	1	1	1	1	1	1	0	0	0	0
0	1	1	1	1	1	1	1	1	1	1	1	0	1	0
0	0	1	1	1	1	1	1	1	1	1	1	1	1	0
0	0	0	0	0	1	1	1	1	1	1	1	1	0	0
0	0	0	0	0	0	1	1	1	1	1	1	0	0	0
0	0	0	0	0	0	1	1	1	1	1	1	0	0	0
0	0	0	0	0	0	1	1	1	1	1	1	0	0	0
0	0	0	0	0	0	1	1	1	1	1	1	0	0	0
0	0	0	0	0	0	0	1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	1	1	1	0	0	0	0	0
0	0	0	0	0	0	0	1	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Our goal is to discover this information:

2.828	2.236	1.414	1	1	1	1	1.414	2	2	2.236	2.828	3.606	4.243	5
2.236	1.414	1	x	x	x	x	1	1	1	1.414	2.236	2.828	3.606	4.123
1.414	1	x	x	x	x	x	x	x	x	1	1.414	2.236	3	3.162
1	x	x	x	x	x	x	x	x	x	x	1	2	2	2.236
1	x	x	x	x	x	x	x	x	x	x	1	1.414	1	1.414
1	x	x	x	x	x	x	x	x	x	x	x	1	x	1
1.414	1	x	x	x	x	x	x	x	x	x	x	x	x	1
2.236	1.414	1	1	1	x	x	x	x	x	x	x	x	1	1.414
2.828	2.236	2	2	1.414	1	x	x	x	x	x	x	1	1.414	2.236
3.606	3.162	3	2.828	2	1	x	x	x	x	x	x	1	2	2.828
4.472	4.123	4	3	2	1	x	x	x	x	x	x	1	2	3
6	5	4	3	2	1	x	x	x	x	x	x	1	2	3
6.083	5.099	4.123	3.162	2.236	1.414	1	x	x	x	x	1	1.414	2.236	3.162
6.325	5.385	4.472	3.606	2.828	2	1	x	x	x	1	1.414	2.236	2.828	3.606
6.708	5.831	5	4	3	2	1	x	x	x	1	2	2.828	3.606	4.243
7.071	6.083	5.099	4.123	3.162	2.236	1.414	1	1	1	1.414	2.236	3.162	4.123	5

The Euclidean Distance Transform (EDT) gives us the Euclidean distance from any given point in the scene to the nearest edge. We can then trace out contours in linear time which will define appropriate toolpaths. The distance transform is at the core of this entire approach and is what makes it possible to create arbitrary nested toolpaths in linear time. Note that if we only want one cut-out toolpath at a known distance from the edge, it may be possible to omit the distance transform, but in that case additional contours would cost additional iterative runs.

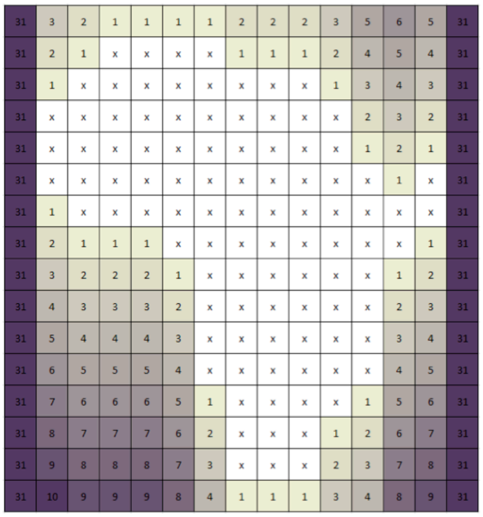


Distance map part 1: Vertical sweep Top to Bottom

31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
31	31	31	x	x	x	x	31	31	31	31	31	31	31	31
31	31	x	x	x	x	x	x	x	x	31	31	31	31	31
31	x	x	x	x	x	x	x	x	x	x	31	31	31	31
31	x	x	x	x	x	x	x	x	x	x	31	31	31	31
31	x	x	x	x	x	x	x	x	x	x	x	31	x	31
31	1	x	x	x	x	x	x	x	x	x	x	x	x	31
31	2	1	1	1	x	x	x	x	x	x	x	x	1	31
31	3	2	2	2	1	x	x	x	x	x	x	1	2	31
31	4	3	3	3	2	x	x	x	x	x	x	2	3	31
31	5	4	4	4	3	x	x	x	x	x	x	3	4	31
31	6	5	5	5	4	x	x	x	x	x	x	4	5	31
31	7	6	6	6	5	1	x	x	x	x	1	5	6	31
31	8	7	7	7	6	2	x	x	x	1	2	6	7	31
31	9	8	8	8	7	3	x	x	x	2	3	7	8	31
31	10	9	9	9	8	4	1	1	1	3	4	8	9	31

In the first of four steps for computing the distance transform, we walk down each column from top to bottom. If we have not yet encountered an edge, we assign a value of infinity, or another number guaranteed to be larger than any possible path length in the scene. We take Meijster's practical suggestion of defaulting to a value of image height + image width, since all distance values must be less than this. Notice even this default is overly conservative, since there is an even tighter bound for Euclidean distance – Meijster's paper demonstrates his algorithm for Manhattan distance measures as well as Euclidean distance, which explains his choice.
After encountering each edge in the sweep, cells outside the solid geometry can be assigned distances which grow incrementally larger from the most recently encountered edge.

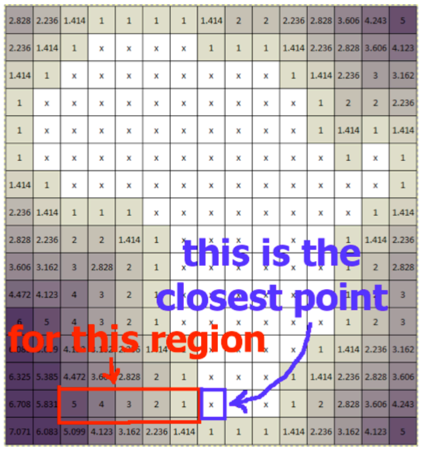
Distance map part 2: Vertical sweep Bottom to Top

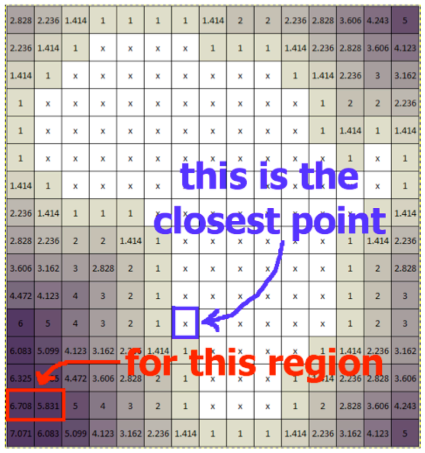


31	3	2	1	1	1	1	2	2	2	3	5	6	5	31
31	2	1	x	x	x	x	1	1	1	2	4	5	4	31
31	1	x	x	x	x	x	x	x	x	1	3	4	3	31
31	x	x	x	x	x	x	x	x	x	x	2	3	2	31
31	x	x	x	x	x	x	x	x	x	x	1	2	1	31
31	x	x	x	x	x	x	x	x	x	x	x	1	x	31
31	1	x	x	x	x	x	x	x	x	x	x	x	x	31
31	2	1	1	1	x	x	x	x	x	x	x	x	1	31
31	3	2	2	2	1	x	x	x	x	x	x	1	2	31
31	4	3	3	3	2	x	x	x	x	x	x	2	3	31
31	5	4	4	4	3	x	x	x	x	x	x	3	4	31
31	6	5	5	5	4	x	x	x	x	x	x	4	5	31
31	7	6	6	6	5	1	x	x	x	x	1	5	6	31
31	8	7	7	7	6	2	x	x	x	1	2	6	7	31
31	9	8	8	8	7	3	x	x	x	2	3	7	8	31
31	10	9	9	9	8	4	1	1	1	3	4	8	9	31

In step 2 of 4 for computing the distance transform, we walk each column from bottom to top, repeating the same procedure as was applied in the other direction. This gives us a complete view of how far any given pixel is from an edge in a vertical direction.

Distance map part 3: Left to Right Scan to Find “Regions of Influence”



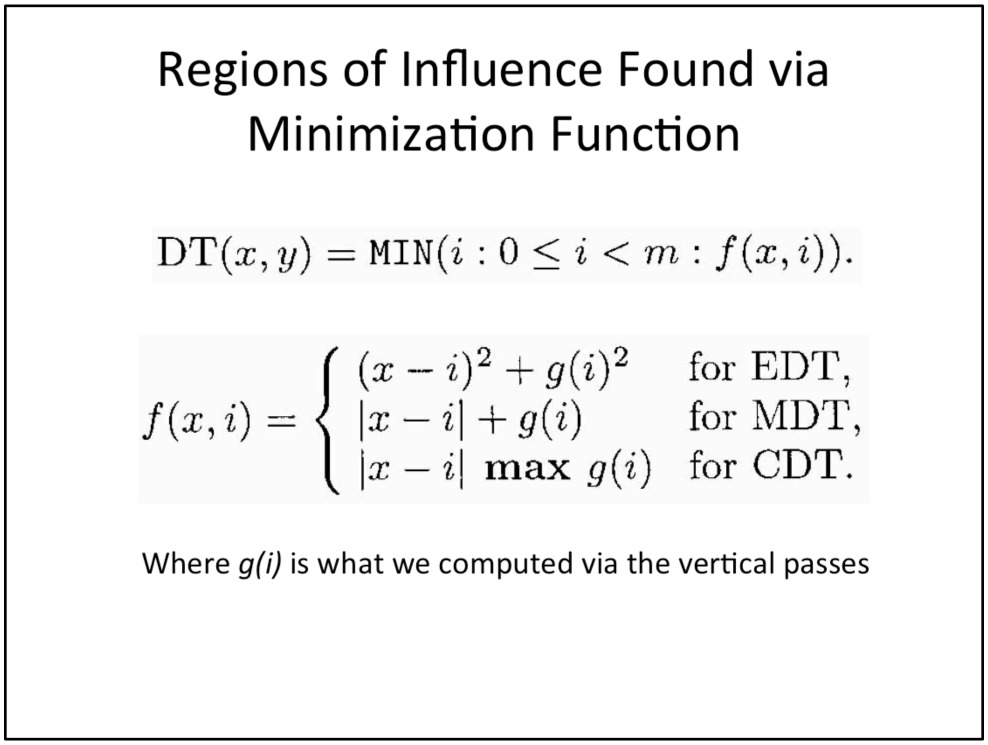


Prove this to yourself by counting boxes and applying the Pythagorean theorem.

The third step in computing the distance transform is both the most expensive and most complicated of the four. In this step we sweep along the rows from left to right, with the goal of segregating regions into groups within which all members have their minimum-distance-to-edge defined by the same point in the solid geometry.

See Meijster’s paper for a slightly different physical interpretation and a different explanation of the same process.

To accomplish this, it is necessary to do a search within a radius of each point. The search radius is initially bounded by the value that was assigned in the cell by the vertical sweep steps, and then the result of the search for the previous point helps to bound it more tightly.



Regions of Influence Found via Minimization Function

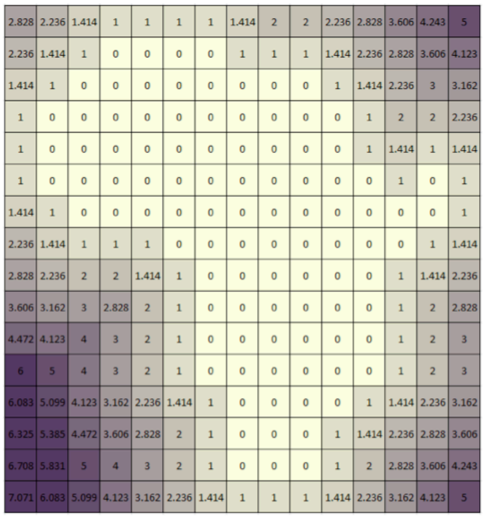
$$\mathrm{DT}(x,y) = \mathtt{MIN}(i : 0 \le i < m : f(x,i)).$$

$$f(x,i) = \begin{cases} (x-i)^2 + g(i)^2 & \text{for EDT,} \\ |x-i| + g(i) & \text{for MDT,} \\ |x-i| \ \mathbf{max} \ g(i) & \text{for CDT.} \end{cases}$$

Where *g(i)* is what we computed via the vertical passes

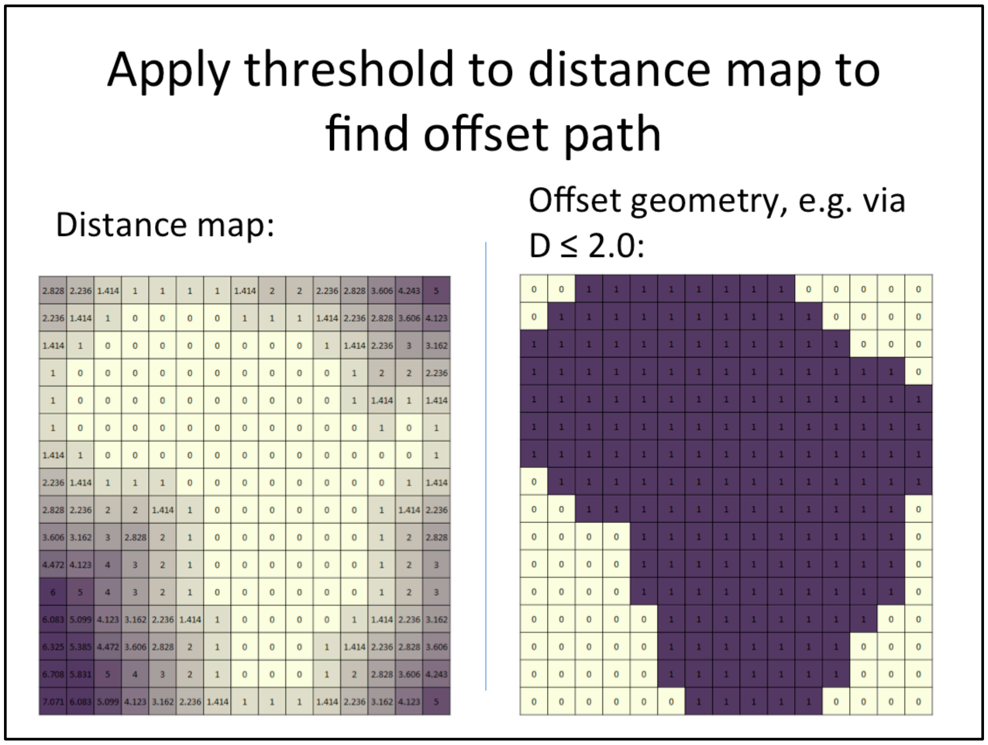
See

Distance map part 4: Right to Left for computation of distance transform



2.828	2.236	1.414	1	1	1	1	1.414	2	2	2.236	2.828	3.606	4.243	5
2.236	1.414	1	0	0	0	0	1	1	1	1.414	2.236	2.828	3.606	4.123
1.414	1	0	0	0	0	0	0	0	0	1	1.414	2.236	3	3.162
1	0	0	0	0	0	0	0	0	0	0	1	2	2	2.236
1	0	0	0	0	0	0	0	0	0	0	1	1.414	1	1.414
1	0	0	0	0	0	0	0	0	0	0	0	1	0	1
1.414	1	0	0	0	0	0	0	0	0	0	0	0	0	1
2.236	1.414	1	1	1	0	0	0	0	0	0	0	0	1	1.414
2.828	2.236	2	2	1.414	1	0	0	0	0	0	0	1	1.414	2.236
3.606	3.162	3	2.828	2	1	0	0	0	0	0	0	1	2	2.828
4.472	4.123	4	3	2	1	0	0	0	0	0	0	1	2	3
6	5	4	3	2	1	0	0	0	0	0	0	1	2	3
6.083	5.099	4.123	3.162	2.236	1.414	1	0	0	0	0	1	1.414	2.236	3.162
6.325	5.385	4.472	3.606	2.828	2	1	0	0	0	1	1.414	2.236	2.828	3.606
6.708	5.831	5	4	3	2	1	0	0	0	1	2	2.828	3.606	4.243
7.071	6.083	5.099	4.123	3.162	2.236	1.414	1	1	1	1.414	2.236	3.162	4.123	5

In the fourth and final step of computing the distance transform, we sweep back across the rows from right to left. At this stage the nearest edge point has been identified for every location in space, so computing the actual distance is a trivial operation.



Apply threshold to distance map to find offset path

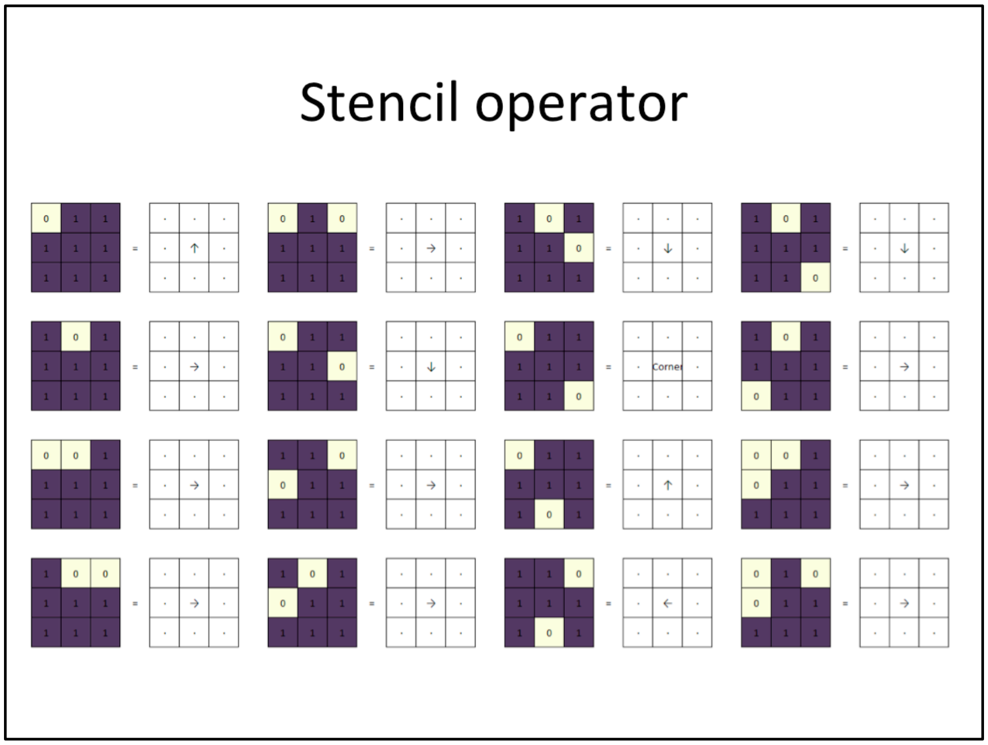
Distance map:

2.828	2.236	1.414	1	1	1	1	1.414	2	2	2.236	2.828	3.606	4.243	5
2.236	1.414	1	0	0	0	0	1	1	1	1.414	2.236	2.828	3.606	4.123
1.414	1	0	0	0	0	0	0	0	0	1	1.414	2.236	3	3.162
1	0	0	0	0	0	0	0	0	0	0	1	2	2	2.236
1	0	0	0	0	0	0	0	0	0	0	1	1.414	1	1.414
1	0	0	0	0	0	0	0	0	0	0	0	1	0	1
1.414	1	0	0	0	0	0	0	0	0	0	0	0	0	1
2.236	1.414	1	1	1	0	0	0	0	0	0	0	0	1	1.414
2.828	2.236	2	2	1.414	1	0	0	0	0	0	0	1	1.414	2.236
3.606	3.162	3	2.828	2	1	0	0	0	0	0	0	1	2	2.828
4.472	4.123	4	3	2	1	0	0	0	0	0	0	1	2	3
6	5	4	3	2	1	0	0	0	0	0	0	1	2	3
6.083	5.099	4.123	3.162	2.236	1.414	1	0	0	0	0	1	1.414	2.236	3.162
6.325	5.385	4.472	3.606	2.828	2	1	0	0	0	1	1.414	2.236	2.828	3.606
6.708	5.831	5	4	3	2	1	0	0	0	1	2	2.828	3.606	4.243
7.071	6.083	5.099	4.123	3.162	2.236	1.414	1	1	1	1.414	2.236	3.162	4.123	5

Offset geometry, e.g. via D ≤ 2.0:

0	0	1	1	1	1	1	1	1	1	0	0	0	0	0
0	1	1	1	1	1	1	1	1	1	1	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	1	1	1	1	1	1	1	1	1	1	1	1	0
0	0	0	0	1	1	1	1	1	1	1	1	1	1	0
0	0	0	0	1	1	1	1	1	1	1	1	1	1	0
0	0	0	0	1	1	1	1	1	1	1	1	1	1	0
0	0	0	0	0	1	1	1	1	1	1	1	1	0	0
0	0	0	0	0	1	1	1	1	1	1	1	0	0	0
0	0	0	0	0	1	1	1	1	1	1	1	0	0	0
0	0	0	0	0	0	1	1	1	1	1	0	0	0	0

Once we have computed the distance transform, it is possible to create a mask that demarcates a surface whose boundaries lie at any desired distance from the edge of the solid geometry. This is easily done by applying a threshold operation. For example, to generate the image on the right, we have True values in all cells whose distance values are less than or equal to 2.0, and False values elsewhere.



Stencil operator

In order to define tool directions along the boundary that we identified in the previous step, we apply a stencil operator to all pixels in the scene.

It defines a function on the 3x3 neighborhood surrounding a given pixel, and as output, generates a direction of travel for that pixel. A 3x3 neighborhood has 9 elements, and thus 2^9 possible states. For implementation, the possible states have been enumerated. In practice, there are far fewer than 2^9 unique cases, because we can take advantage of rotation. Note that we cannot take advantage of reflection to pare the number down further because the rules have handedness in their properties in order to ensure a consistent direction of travel.

This is represented in the "fab.c" source file, in the function:

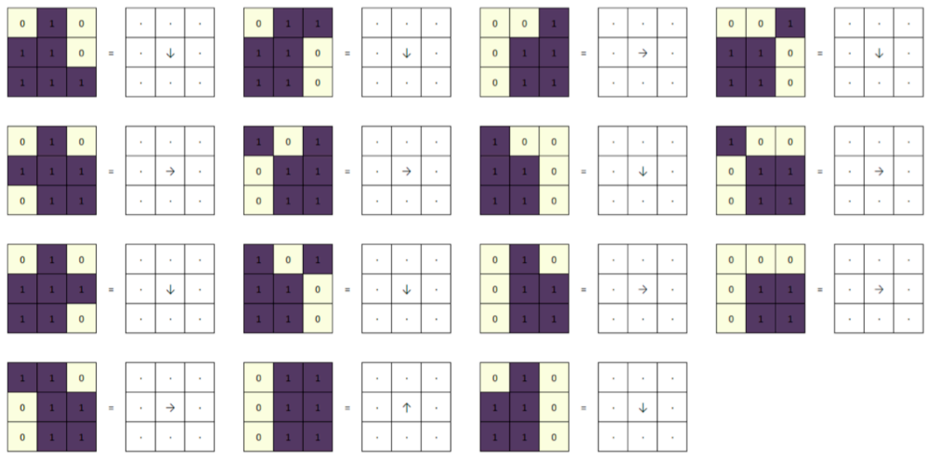
```
void fab_directions(struct fab_vars *v)
```

State is represented as a 32-bit integer, where the nine positions in a 3x3 neighborhood are each allocated 2 bits of state information.

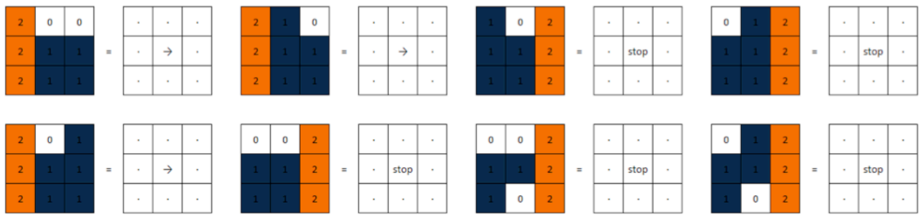
For some of the possible edge shapes, it is very easy to see how the resulting direction was identified. The criteria are as follows:

- If we travel one pixel in the resulting direction, we must land in a pixel that is still filled with solid geometry.
- We should travel in the direction that results in clockwise travel around the edge of the geometry. Note that as long as the rules are consistent, we can always

Stencil operator



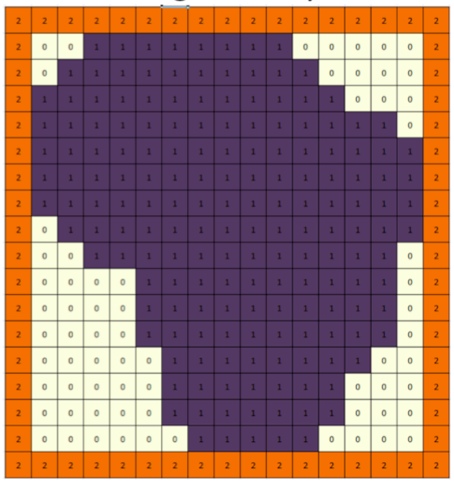
Stencil operator



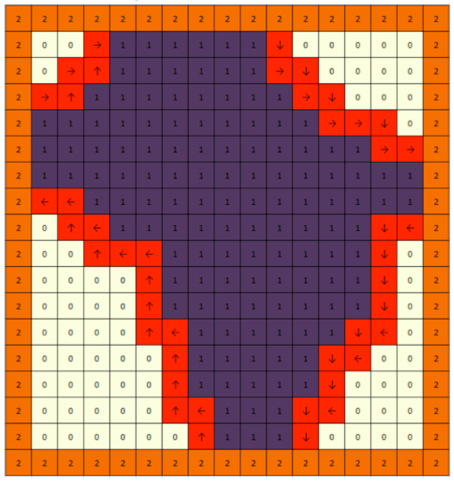
🟧 2 = edge of scene

Apply stencil operator to find offset edges and direction on edge

Offset geometry:

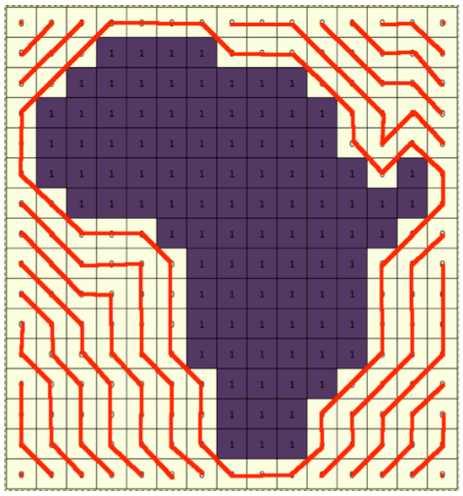


Basic path information:

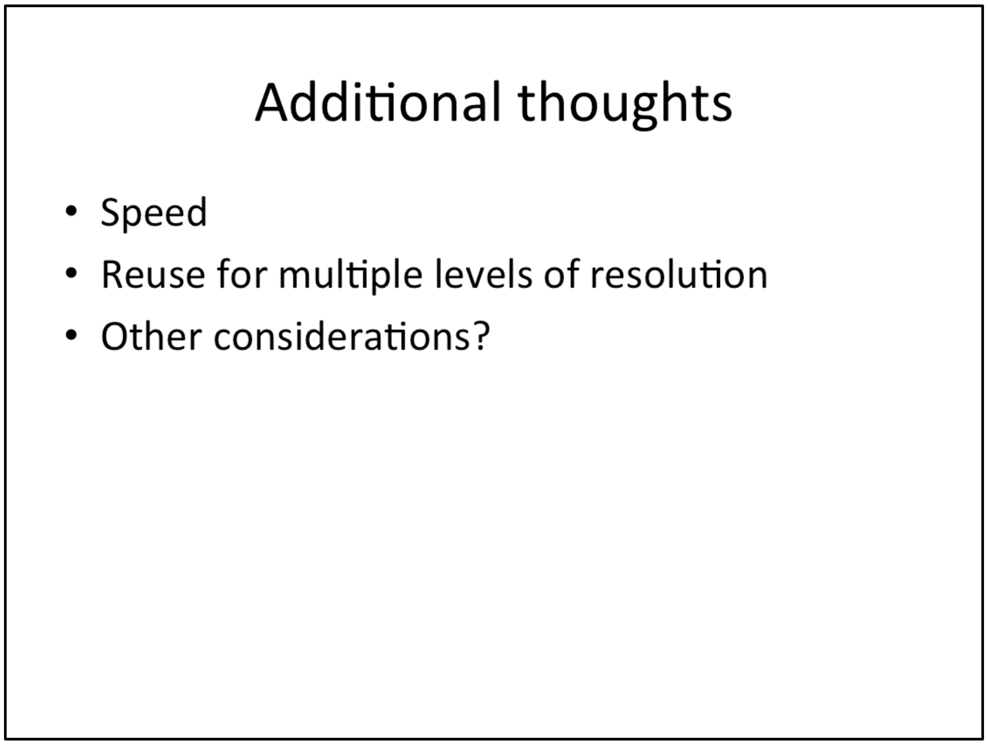


Refer to the stencil operator function definitions on the previous pages to see how each directional arrow was obtained from that recipe.

Linear pass to identify each closed path; Follow and vectorize each path



For nested paths, note that we don't have to repeat the work done to obtain the distance transform.



Additional thoughts

- Speed
- Reuse for multiple levels of resolution
- Other considerations?

Speed:

The distance transform was at the heart of this approach, and with the algorithm used, it is linear time, and parallelizable – processing for each column can be done independently of the other columns, and processing of each row can be done independently of the other rows, but the work on all columns must be complete before we begin on the rows.

The speed of this approach grows linearly with the number of lattice elements in the grid, which in turn is defined by the chosen resolution. This is pretty good – good enough for all our practical applications at the moment. We can theoretically improve the speed if we decrease the number of lattice elements, e.g. by reducing the resolution in the areas that don't have edges or tool paths. For example, if we built an octree with low resolution on the interior of a part (for subtractive machining) or on the exterior of a part (for additive manufacturing).

There may also be a way to devise an algorithm that scales approximately with the edge length instead of scaling with the number of discrete elements in the entire space, though this would require incorporating additional knowledge (e.g. knowledge of how many parts/edges exist in the scene) because otherwise there has to be at least one step that searches the entire space to make sure that all edges have been found.

References

Fab modules:
http://kokompe.cba.mit.edu/dist/downloads.html

Meijster; “A General Algorithm for Computing Distance Transforms in Linear Time”

http://fab.cba.mit.edu/classes/S62.12/docs/Meijster_distance.pdf