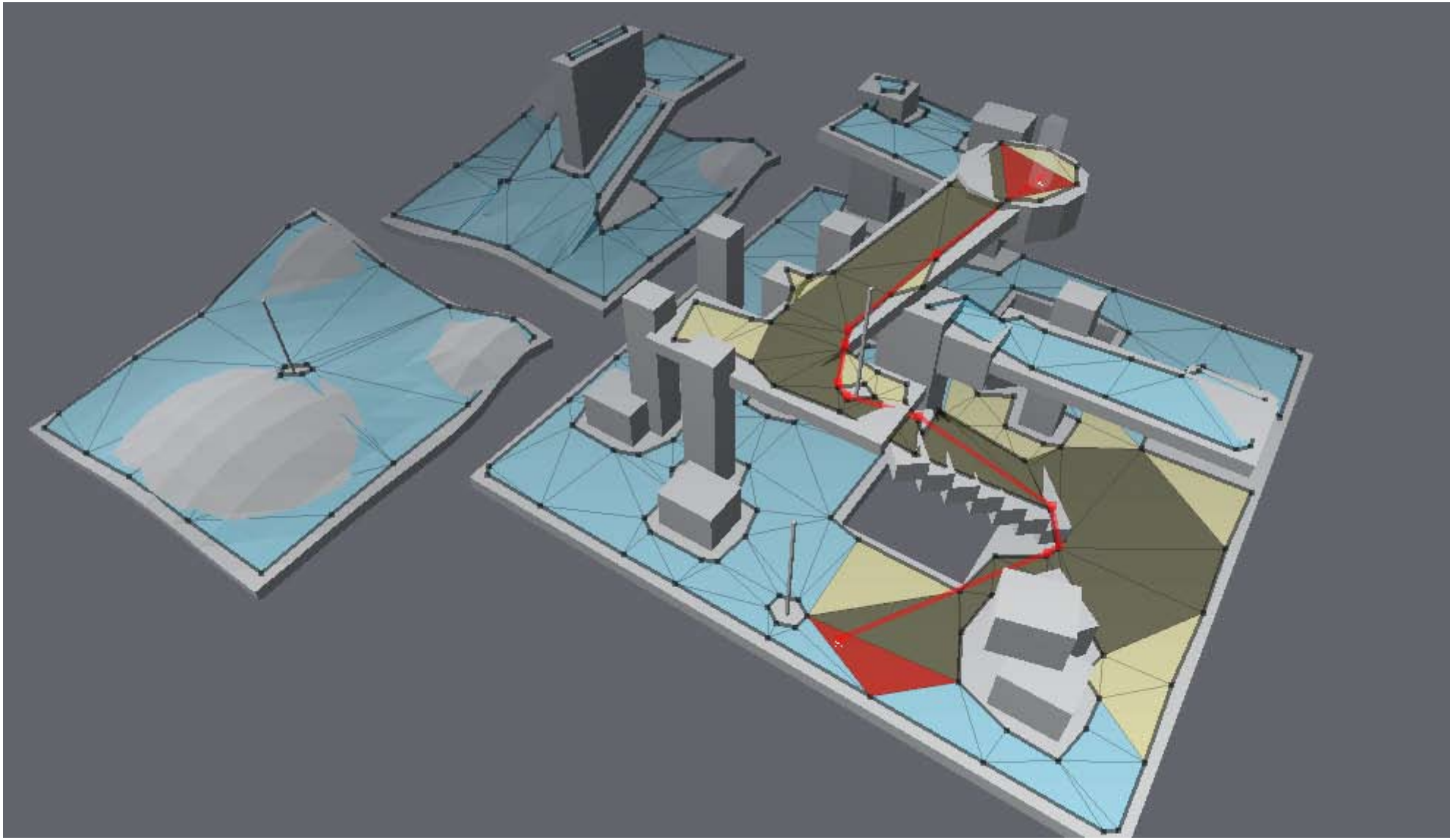
Automatic Navmesh Generation via Watershed Partitioning



Mikko Mononen 2009

Goal

- Our aim is to generate navmesh from a triangle soup

- We use rasterization to create unified representation of the input data and to extract the walkable areas

- The big problem if this method is how to turn the voxels back to polygons

- There are many 2D algorithms, which are well studied and would solve out problem, but our data is 2,5D, it contains overhangs
    - Bitmap to vector conversion is well studied and implemented field
    - All existing robust triangulation algorithms require 2D data

- And it needs to be fast to have quick turnaround times for level designers!

- Inspiration came from:
  Crowds In A Polygon Soup: Next-Gen Path Planning, David Miles

Goal

- IDEA: partition the voxel area into simpler areas

- First implementation used “depth peeling”
    - worked, but sometimes created weird holes in on overlapping areas

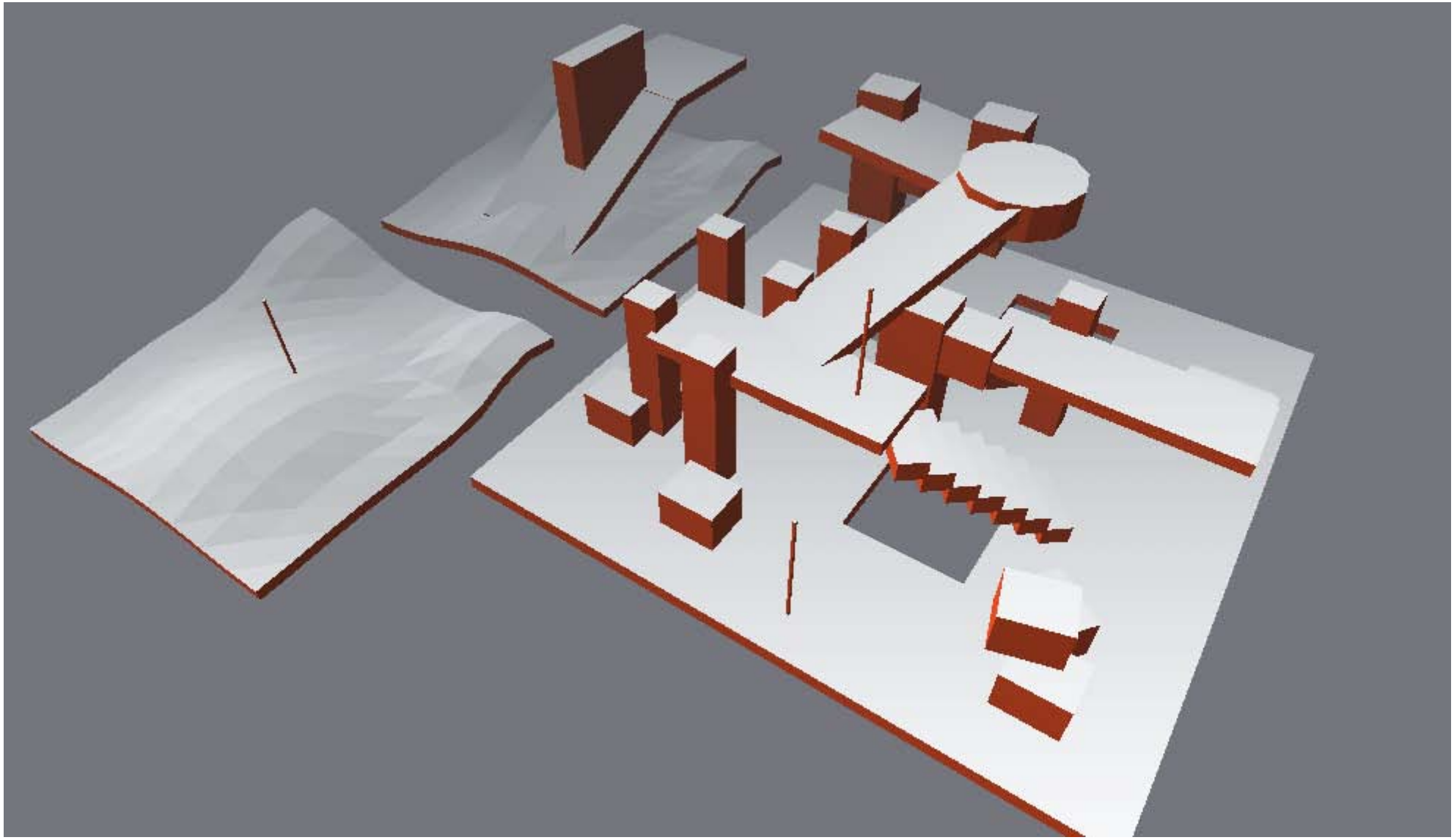
- Second attempt was to try to find an algorithm which could partition a bitmap into approximately convex shapes
    - could not find one
    - but the research sparked an idea:
        - we do not necessarily need a convex area, just simpler area
        - i.e. monotone convex, taxi-cab convex, “simple polygon”

- Found watershed partitioning from automatic portal generation literacy
    - badly behaving and well studied algorithm

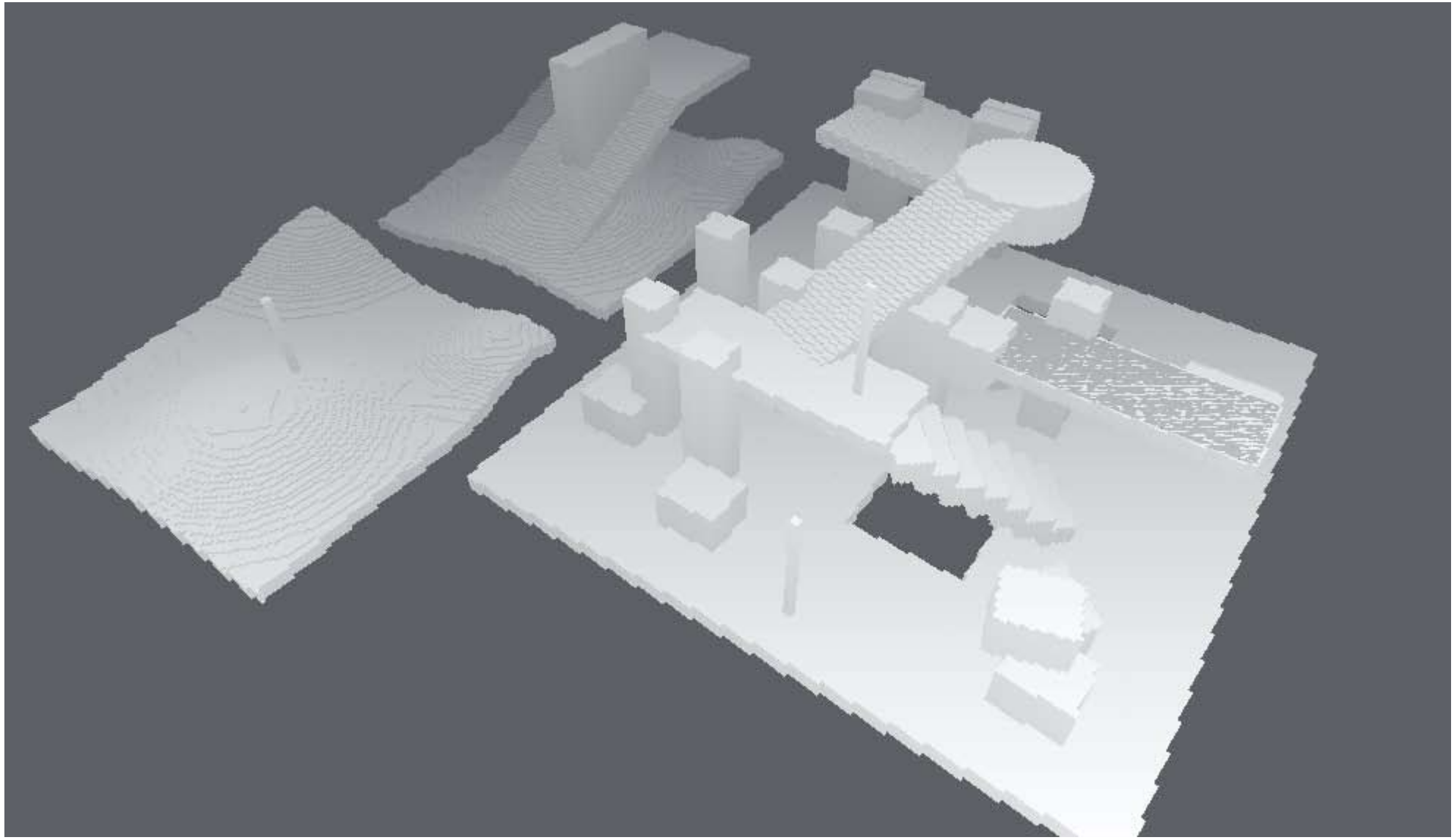
- Watershed paritioning has nice property that the partitions are “simple”
    - that is, no overlap and no holes, all we need for robust triangulation!

Overview

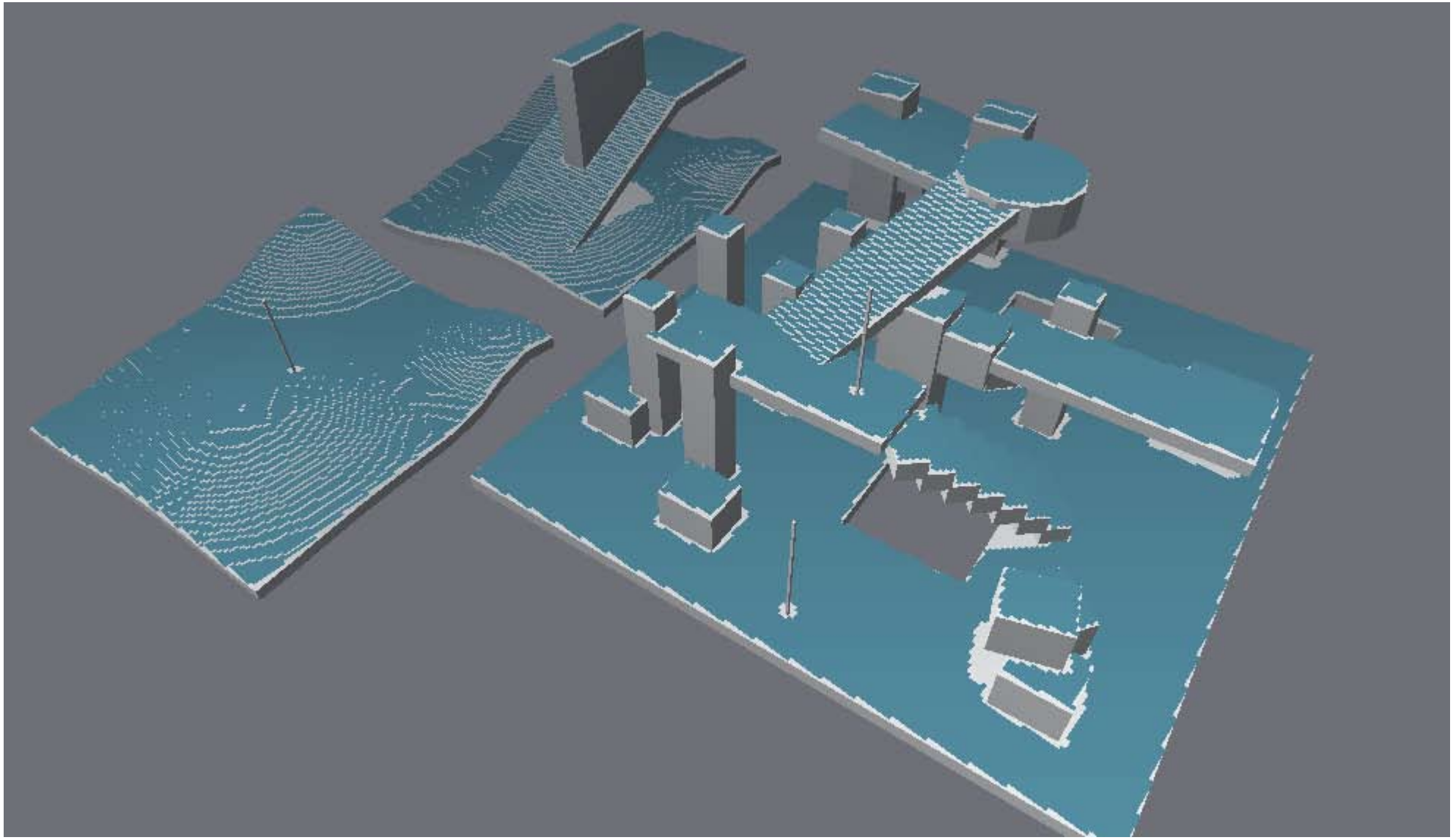
Automatic navmesh generation in 5 simple steps :)



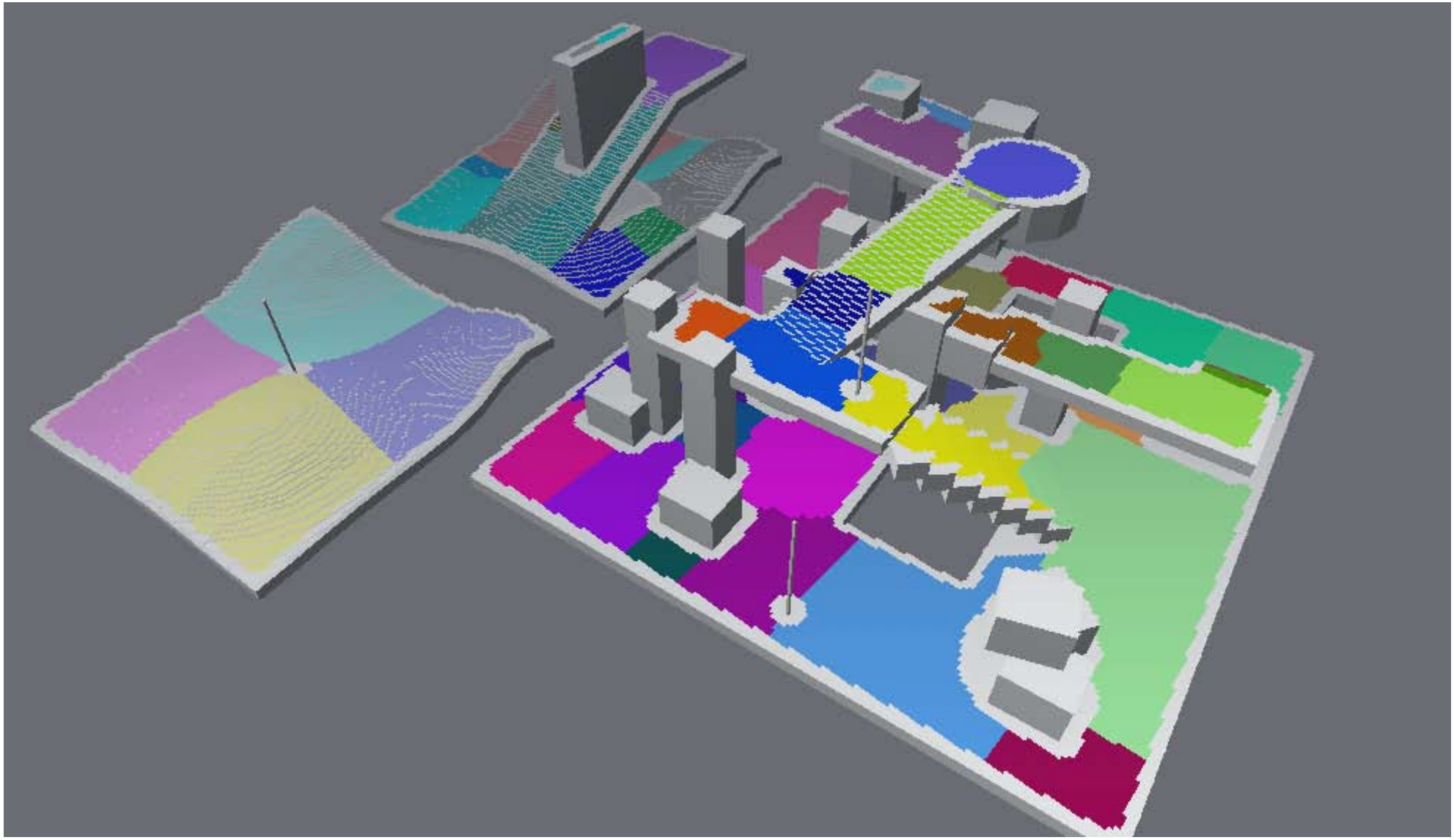
Input: Arbitrary polygon soup with triangles marked as walkable.



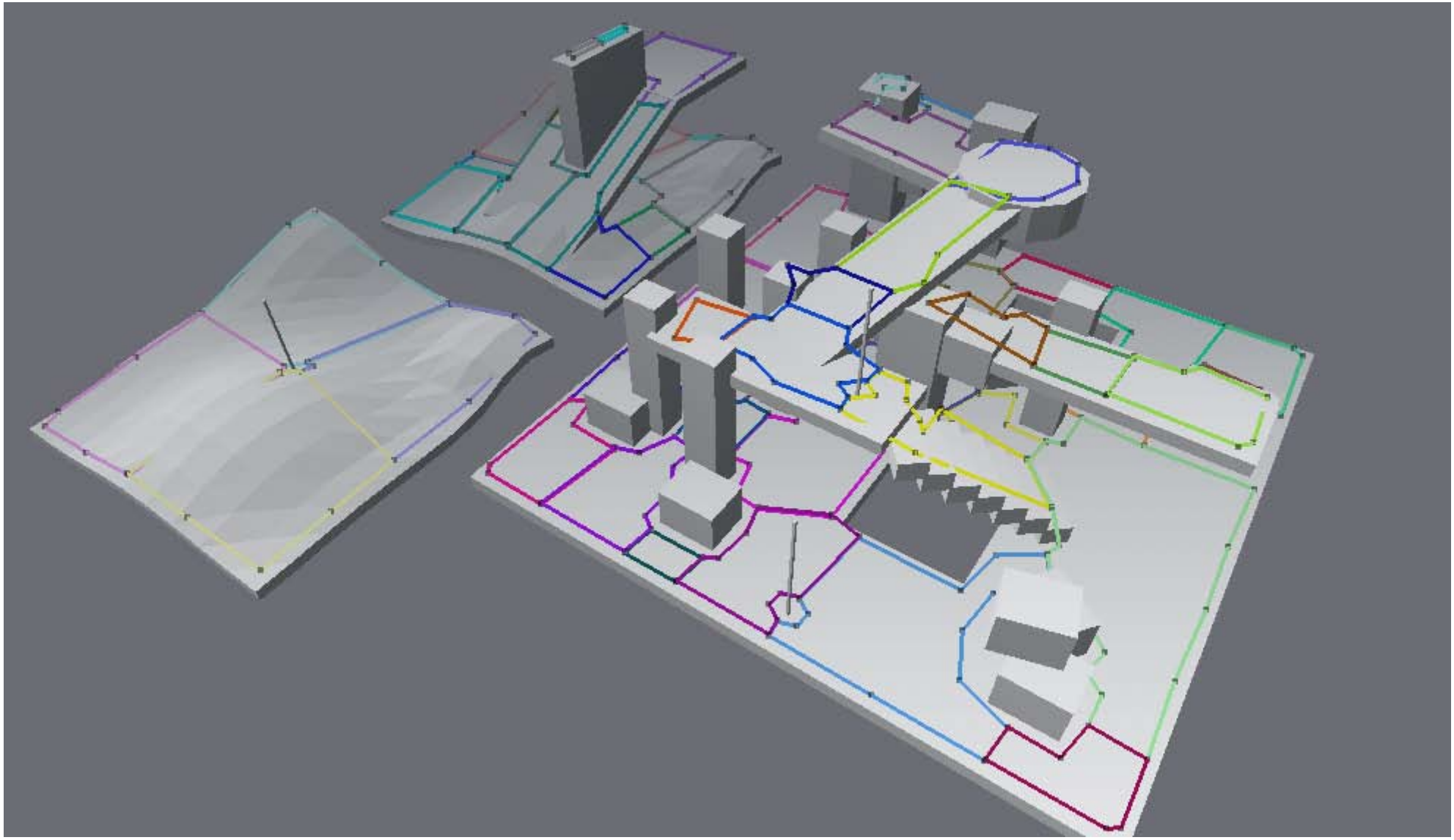
Step 1: Voxelize the polygons.



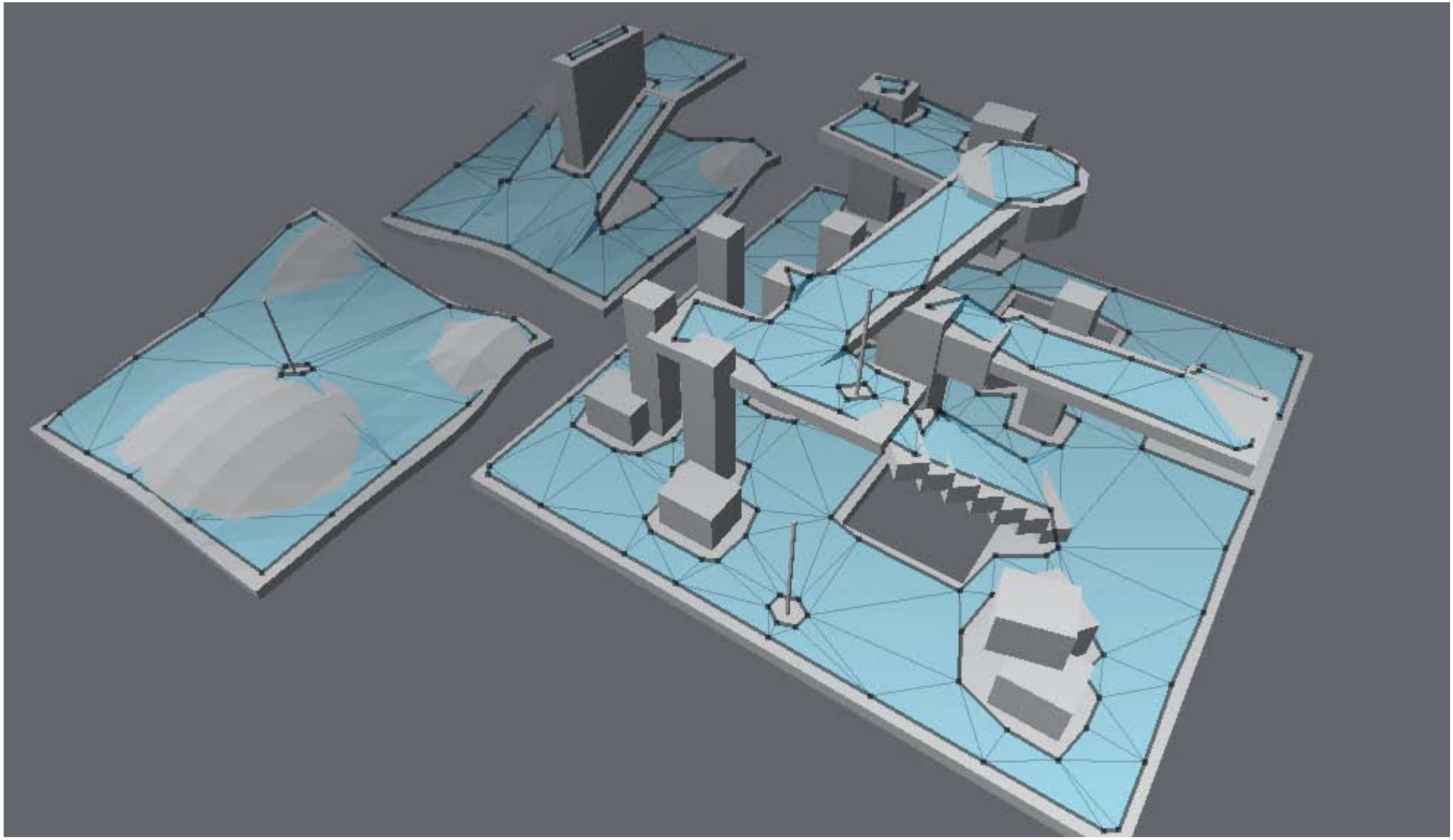
Step 2: **Build navigable space from solid voxels.**



Step 3: **Build watershed partitioning and filter out unwanted regions.**



Step 4: Trace and simplify region contours.



Step 5: **Triangulate the region polygons and build triangle connectivity.**

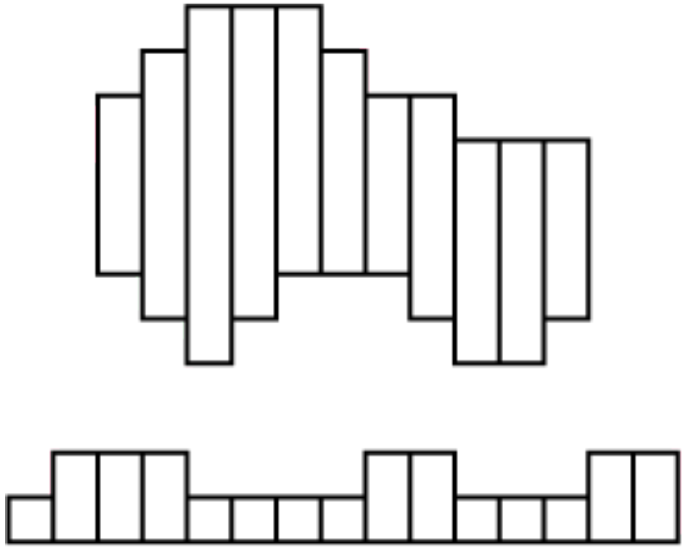
Step 1: Voxelize the polygons

- Many viable algorithms and voxel representations
    - sampling using physics
    - rasterization
    - octrees
    - one bit per voxel

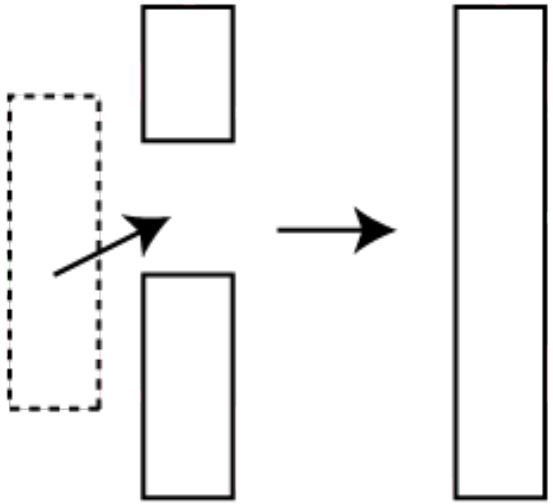
- We choose to use RLE encoded voxels
    - simple to implement
    - fast to rasterize
    - compact representation
    - provides quick access to neighbours

- Conservative rasterization
    - done by clipping the input triangles into cell boundaries
    - min/max height extracted from the resulting polygon

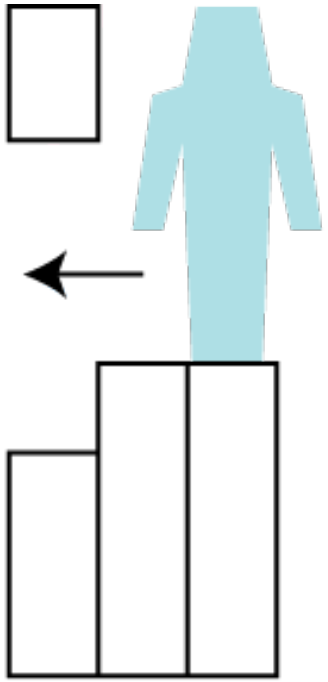
Step 1: Voxelize the polygons



Stored as heightfield of overlapping spans



Easy to construct



Really fast neighbour and reachability tests

Step 2: Build navigable space from solid voxels

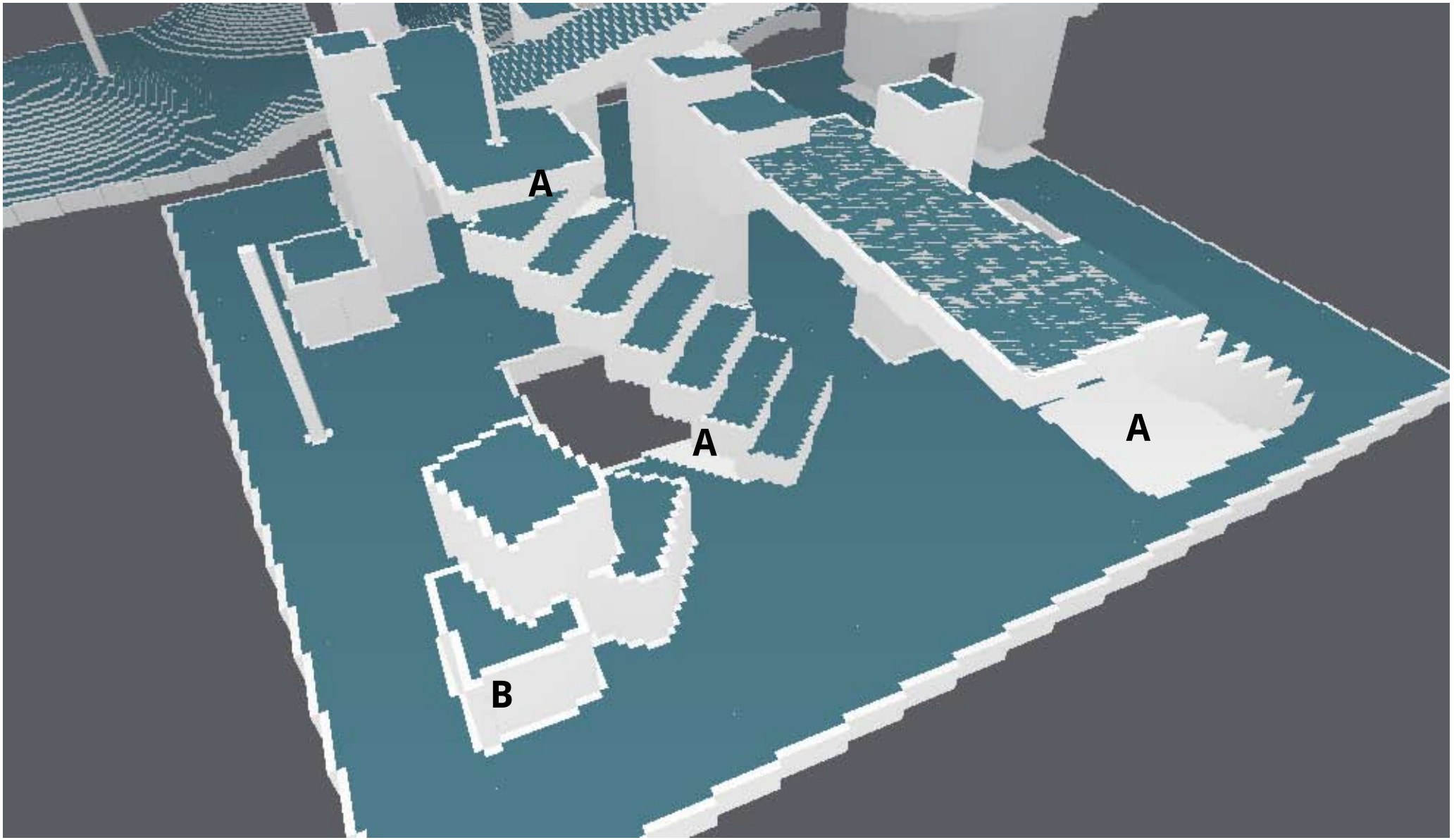
- Filter out voxels where the character cannot stand
    - too steep slopes (use triangle flags)
    - too low places
    - exchange the representation from rigid space to free space

- Need to compensate for the conservative rasterization on ledges
    - erosion by on ledge cells 1 is good approximation

- We take care of the agent radius later

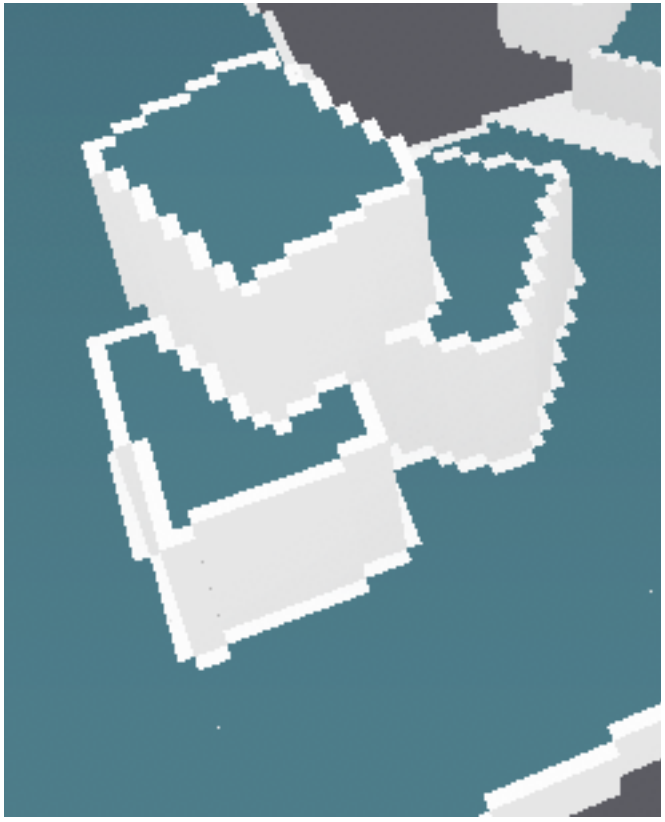
- Change to more compact data representation
    - rasterization done with linked lists
        - the voxelization is treated as static from this point on
    - further process done on index/size cells

- The navigable space could be already used as pathfinding data

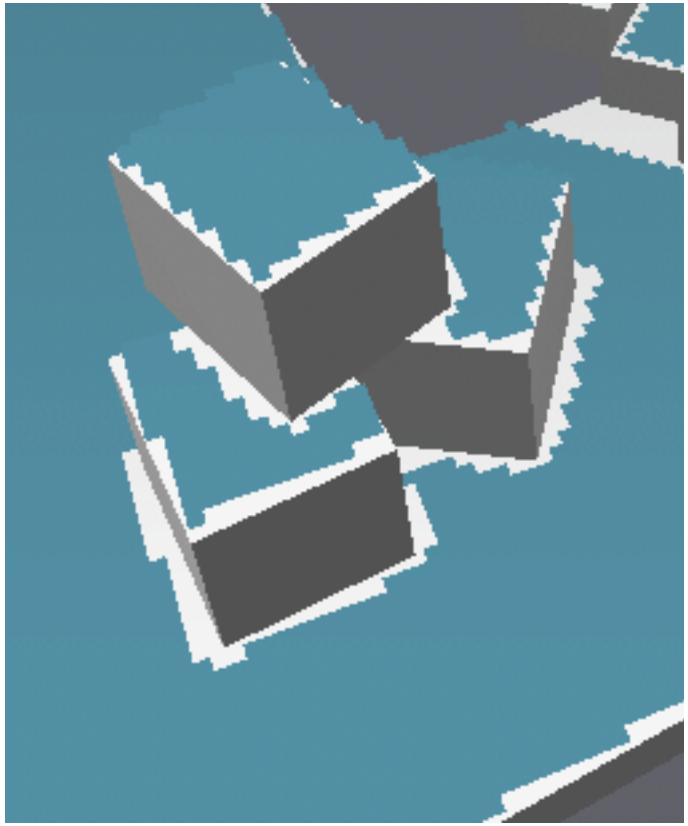


A) Too low cells filtered

B) Erosion of the edges



Voxelization with walk-able cells marked

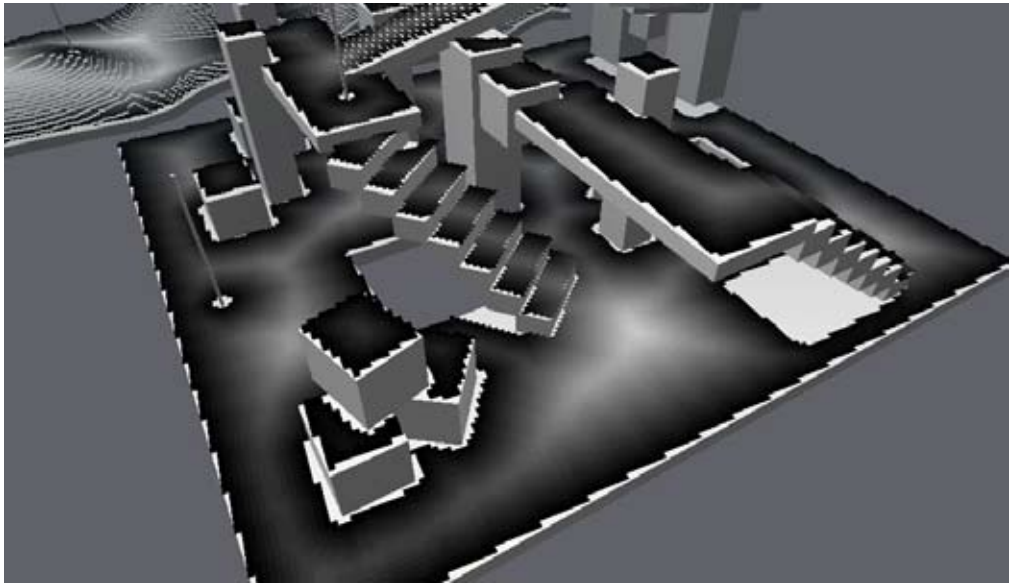


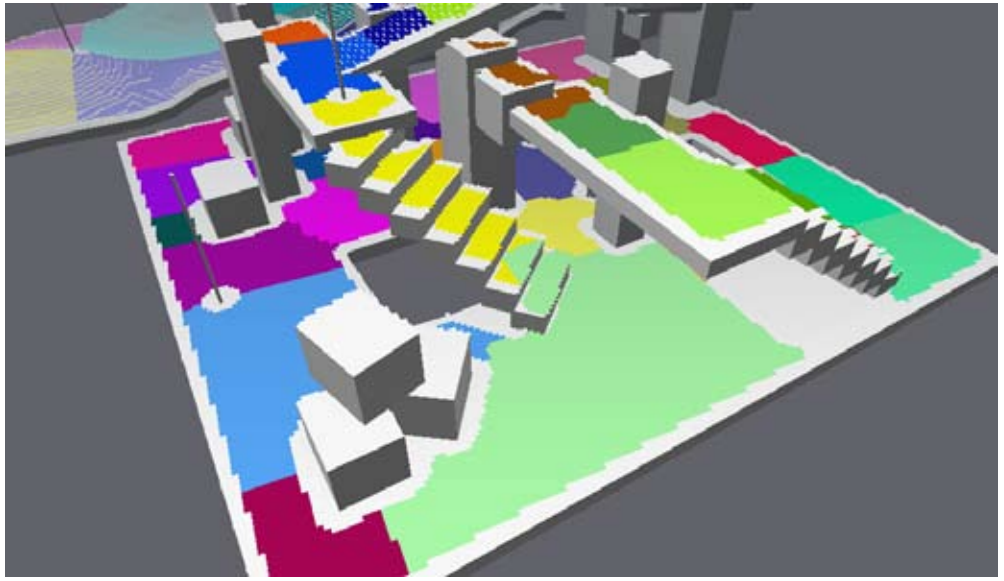
Walkable cells overlayed on top of input geometry

- Need to compensate for the conservative rasterization on ledges
    - erosion by on ledge cells 1 is good approximation

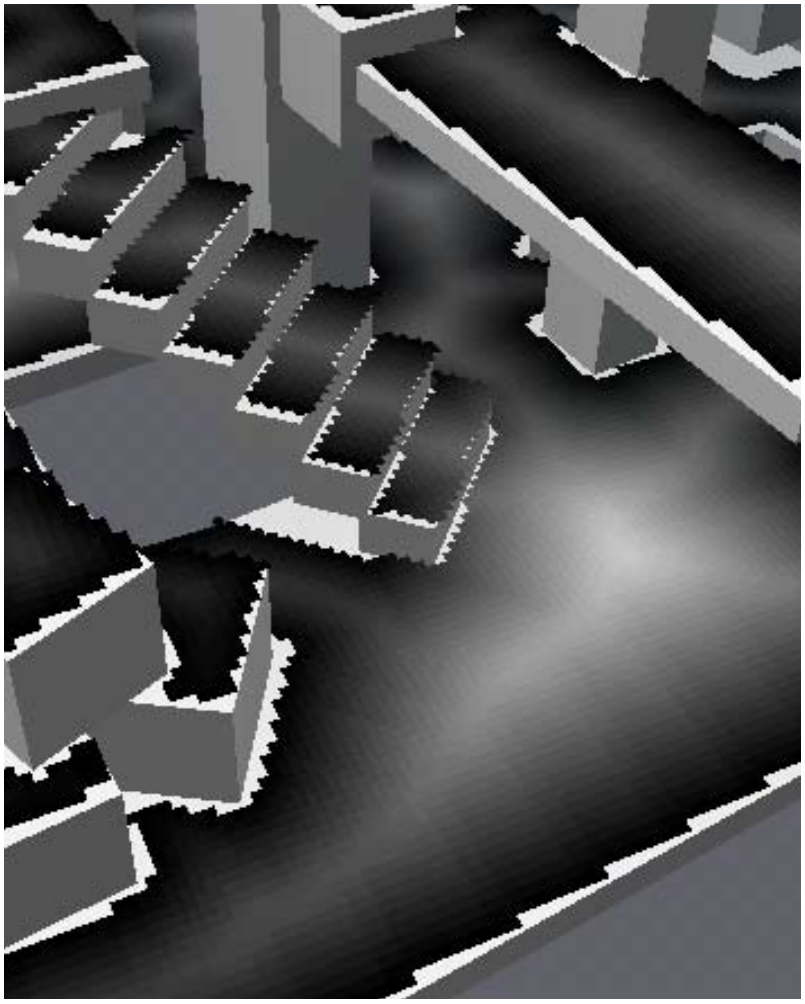
Step 3: Build watershed partitioning and filter out unwanted regions

- Watershed algorithm in nutshell:
    1) Build distance transform of the input areas
    2) Start from the highest distance one slice at a time:
        a) Find any new catchment basins and fill them with a new ID
        b) Expand existing regions

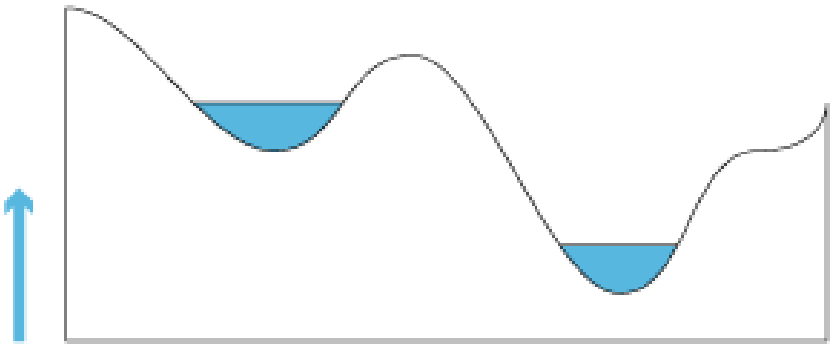




Step 3: Build watershed partitioning and filter out unwanted regions

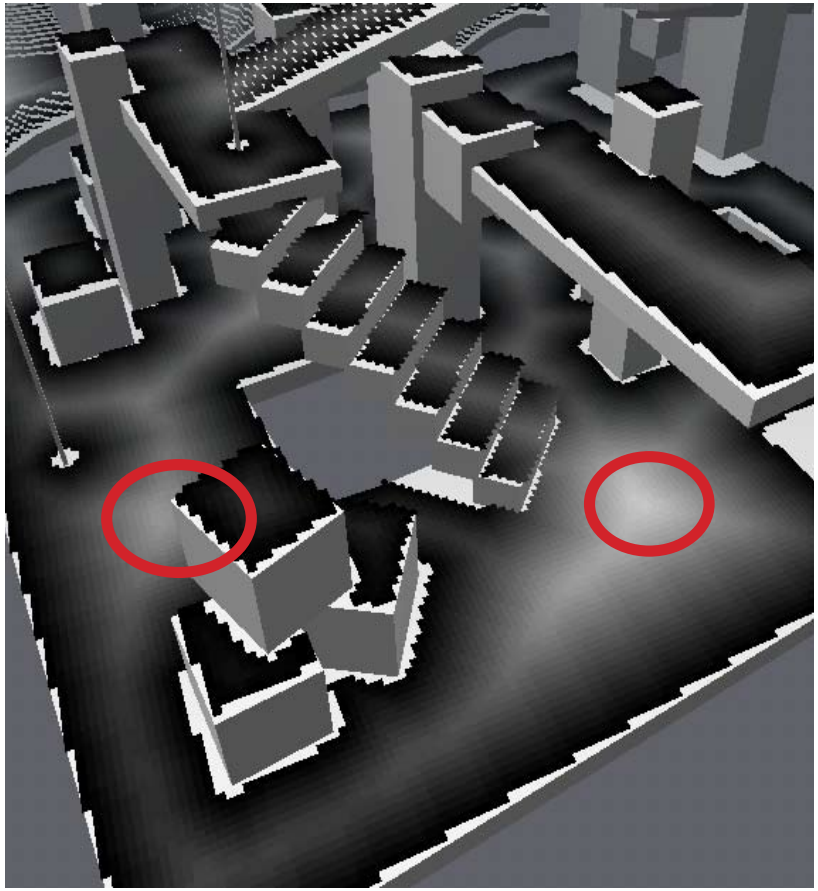


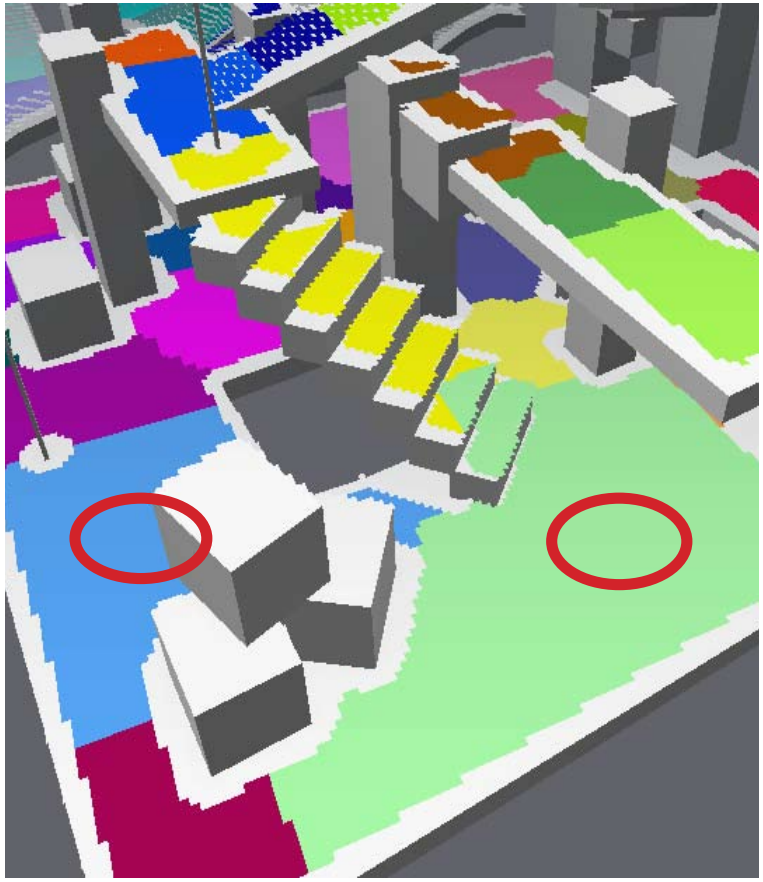
White area represents lower region.



As the water level is raised, new catchment basins are found.

Step 3: Build watershed partitioning and filter out unwanted regions

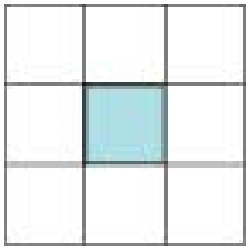




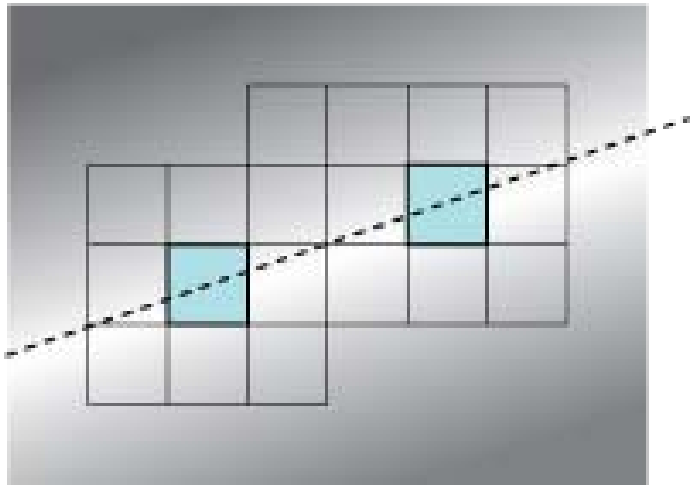
The catchment basins become the centers of the regions.

Step 3: Build watershed partitioning and filter out unwanted regions

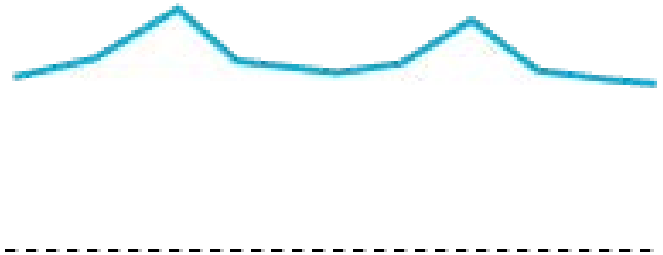
- The algorithm is prone to noise and aliasing in the distance field
    - blurring the distance field helps quite a bit



Search neighbourhood



Problem case



Problem case cutout

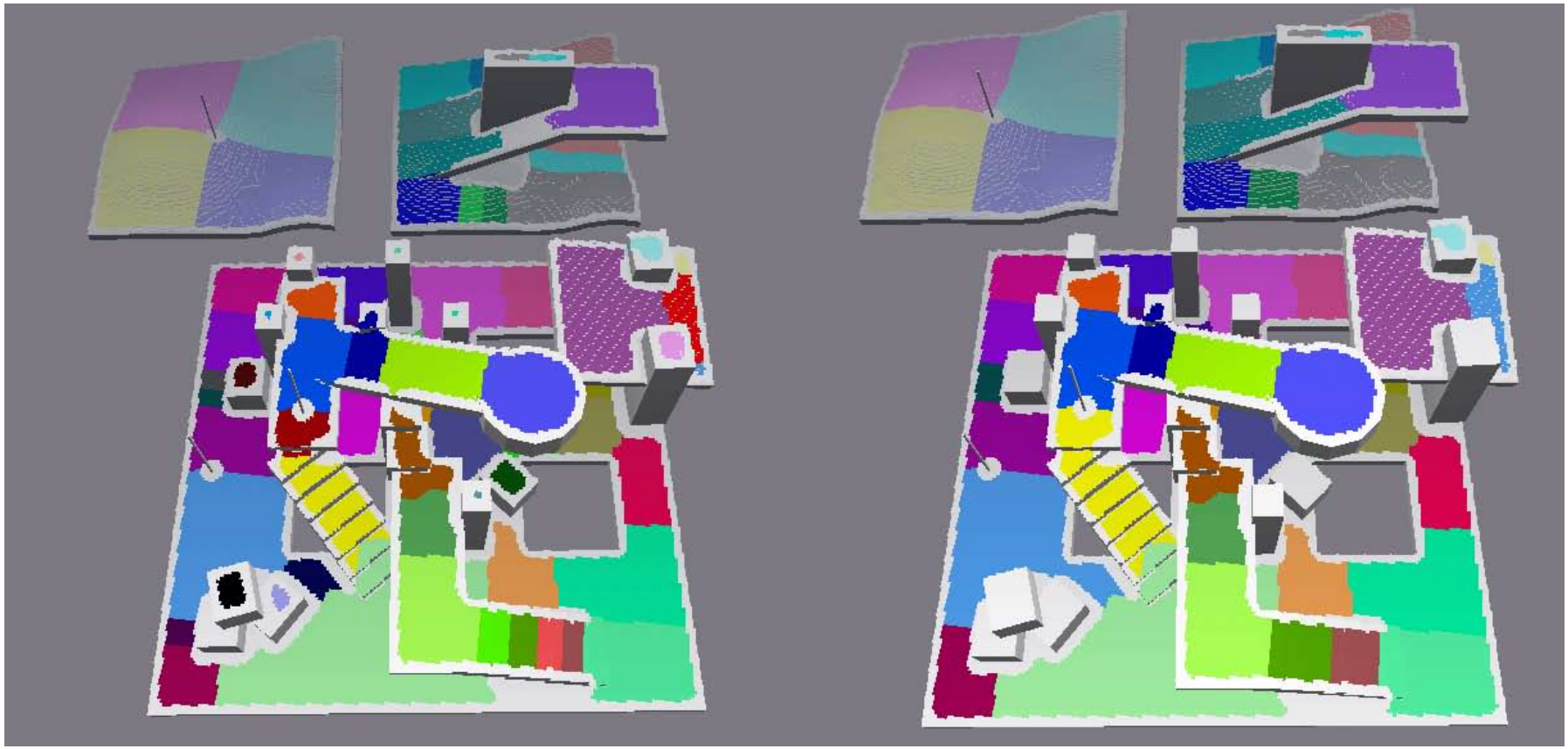
Distance Field

Blurred Distance Field

The example scene is rotated by 15 degress to amplify the problem.

Step 3: Build watershed partitioning and filter out unwanted regions

- We further apply a filtering pass to the newly creation regions
  - to remove small unconnected regions
  - to merge small regions together

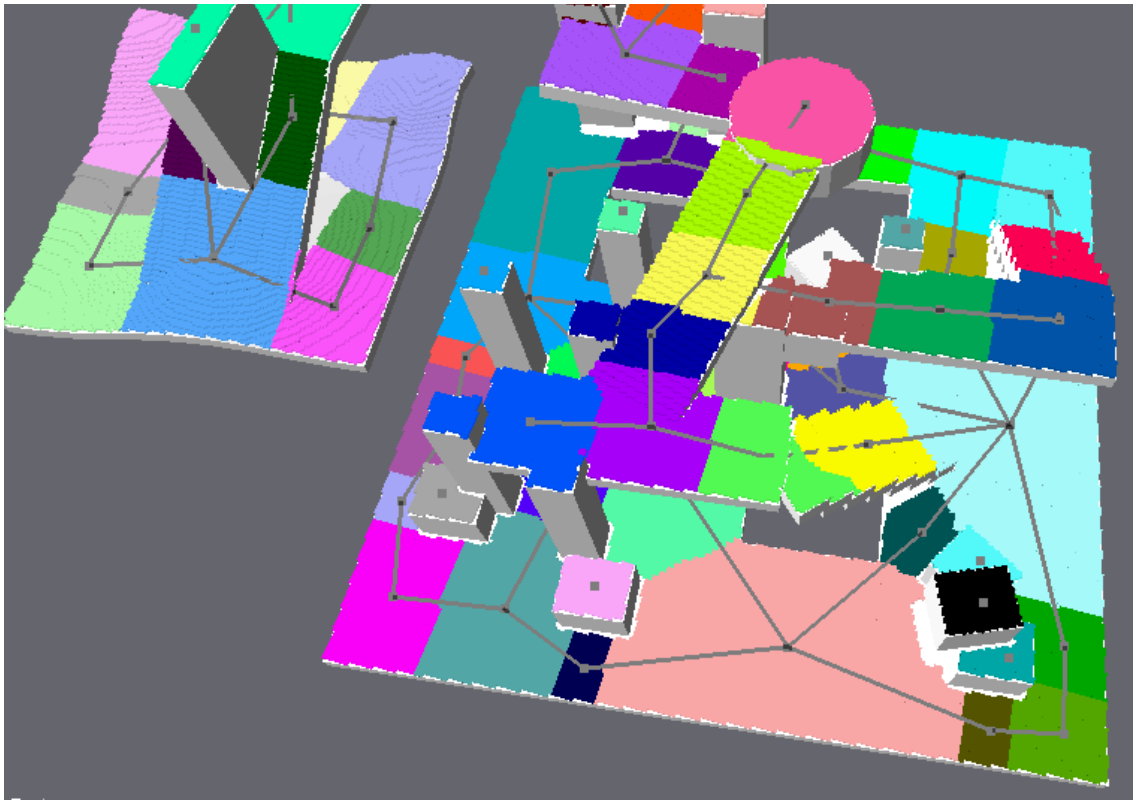


Blurred Distance Field

Filtered Regions

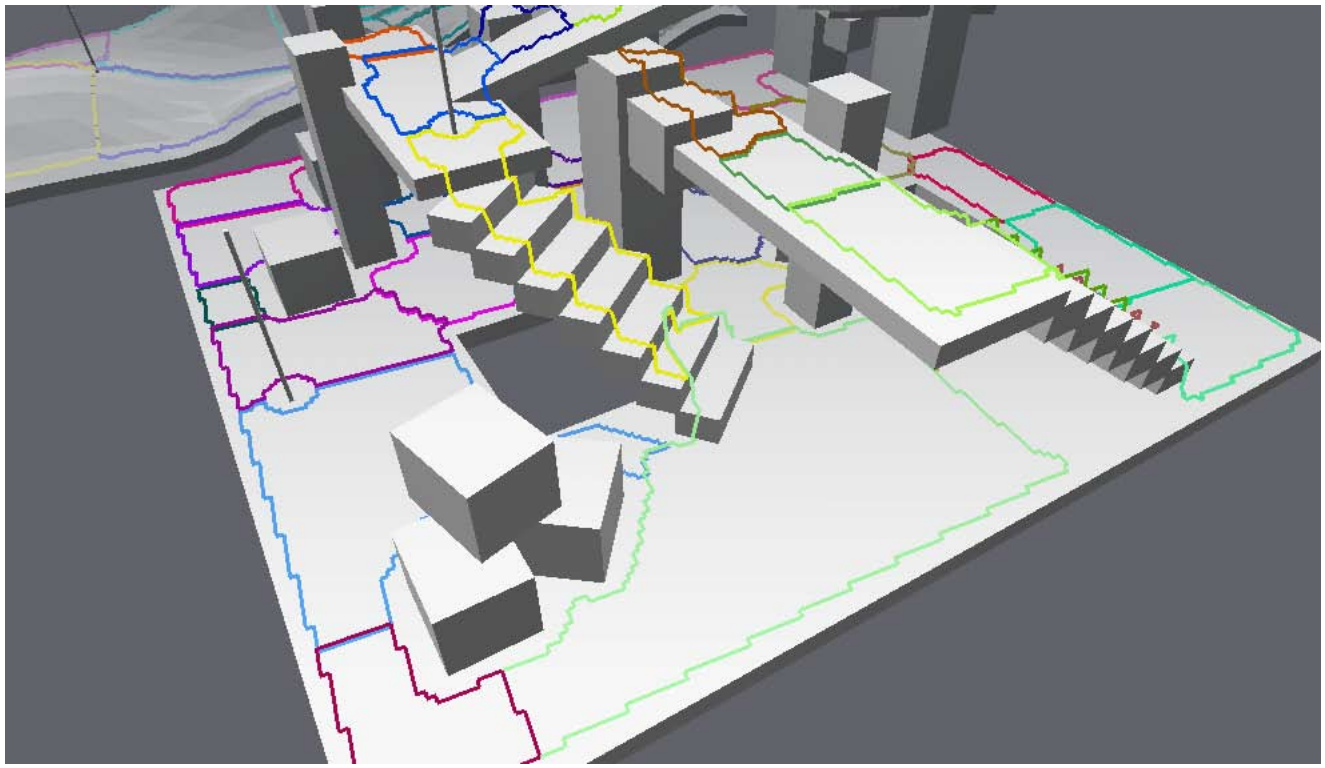
Step 3: Build watershed partitioning and filter out unwanted regions

- If desired, the we can erode the walkable space during the process so that the regions include Minkowski sum of the agent radius
- The result is set of nonoverlapping simple regions
- The resulting partitioning could be used as abstraction for cell level pathfinding (HPA*) or as basis for generating waypoint graphs



Step 4: Trace and simplify region contours.

- Find contours
    1) Find starting point to start tracing (region edge cell)
    2) Trace around the boundaries of the regions
        - at each step store
            - the cell corner points which will form the polygon
            - the neighbour region ID

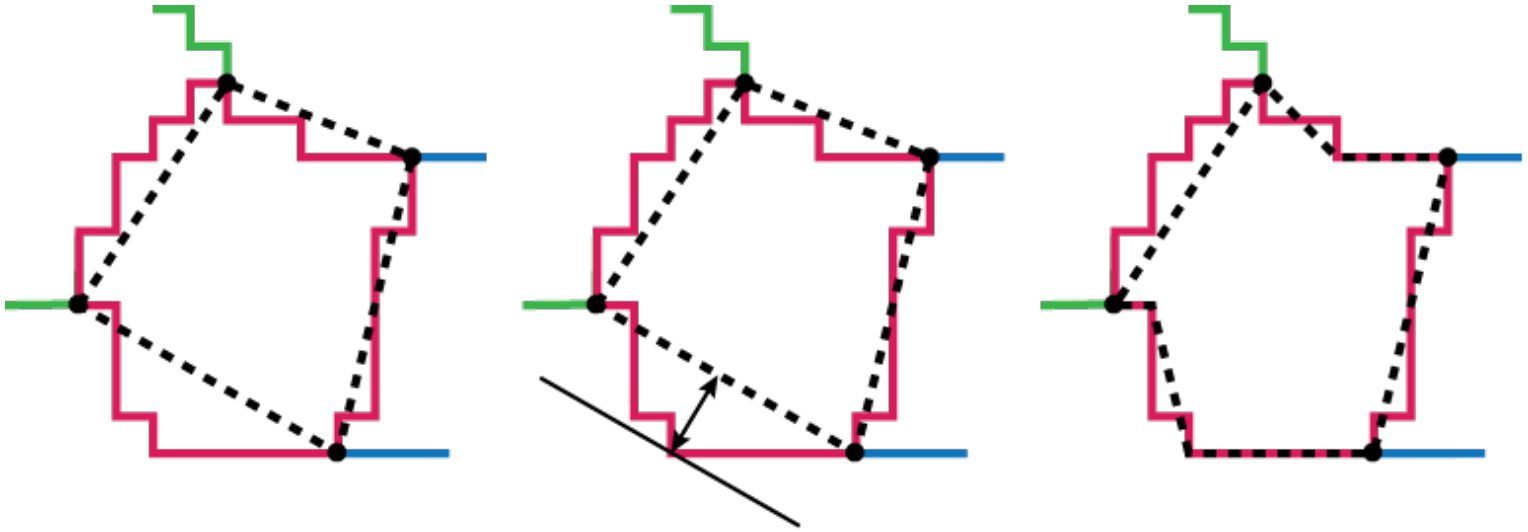


Step 4: Trace and simplify region contours.

- Simplify contours using Ramer-Douglas-Peucker algorithm
    - find initial segments
        - lock vertices which are between two different regions
        - if the region is not connected, lock most two extrema vertices
    - iterate through all simplified segments
        - walk through the raw points between two end points
        - if the simplified segment is between two regions, skip it
        - if distance to the simplified segment is larger than max error
            - subdivide the segment at maximum error vertex
        - too long edges are subdivided too

- The result is set of simple polygons
    - the initial vertices allow us later to find common edges between the polygons

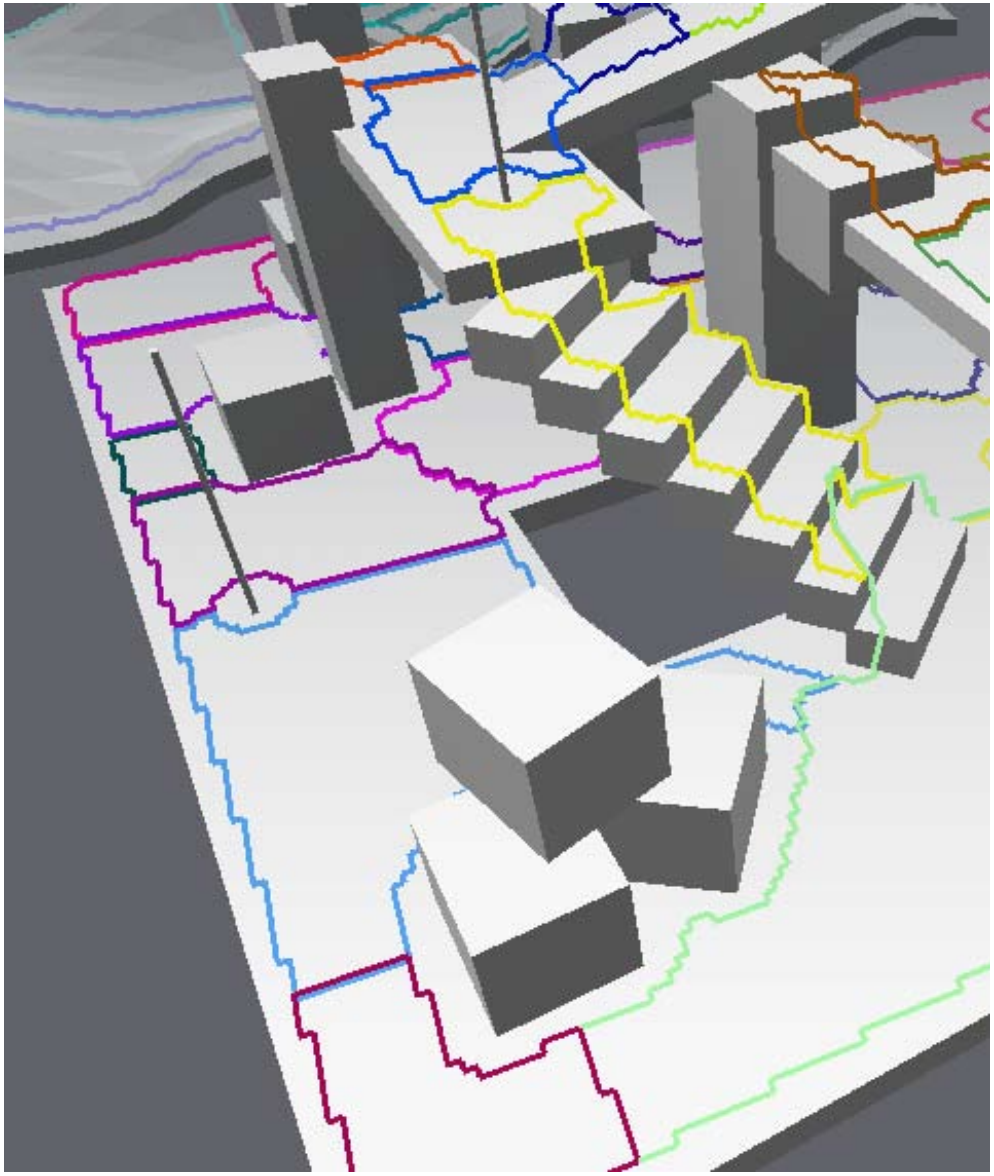
Step 4: Trace and simplify region contours.



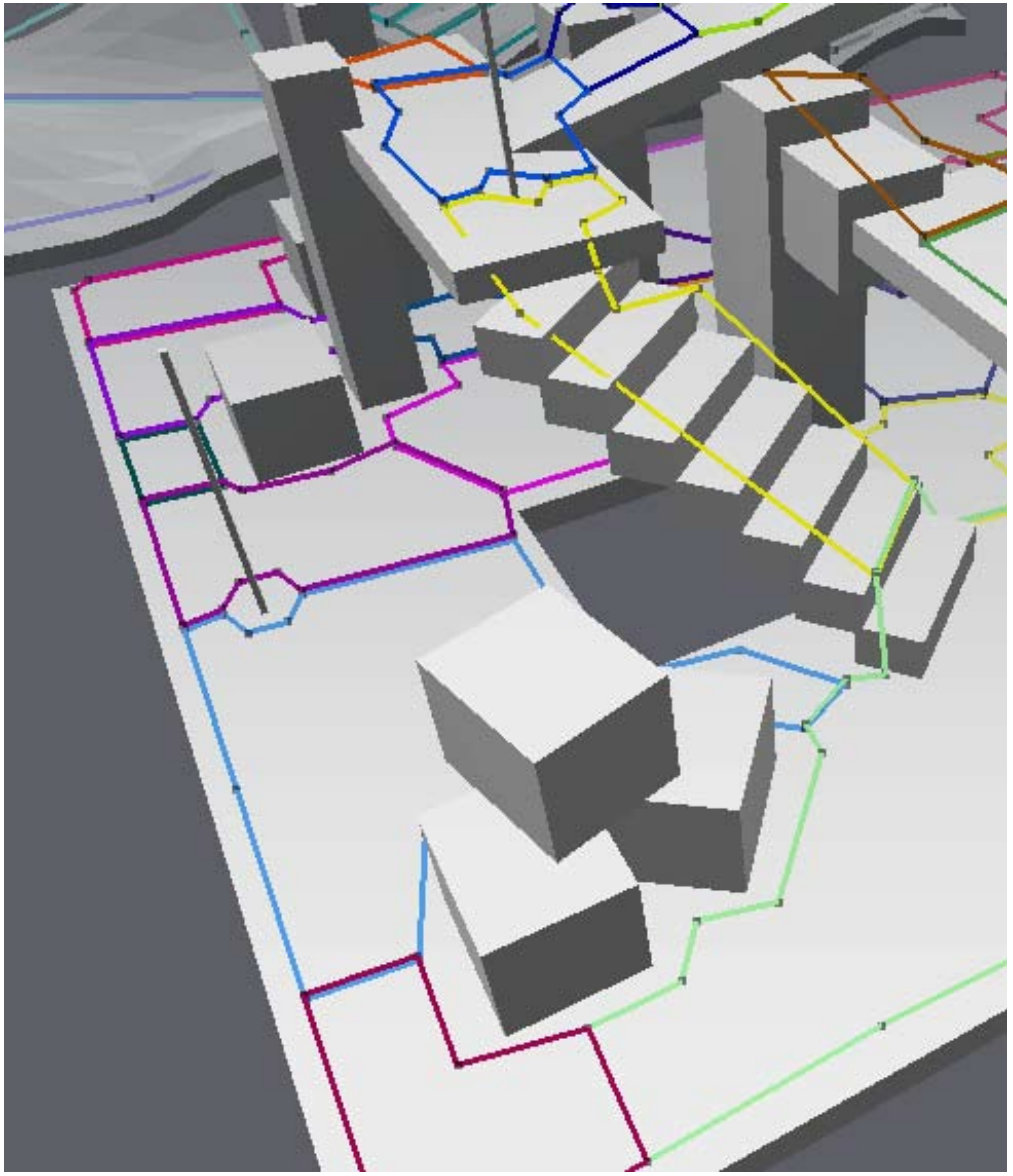
Initial vertices at region edges

Find vertex with maximum error, and subdivide

Iterate untill certain error criteria is met



Traced Contours



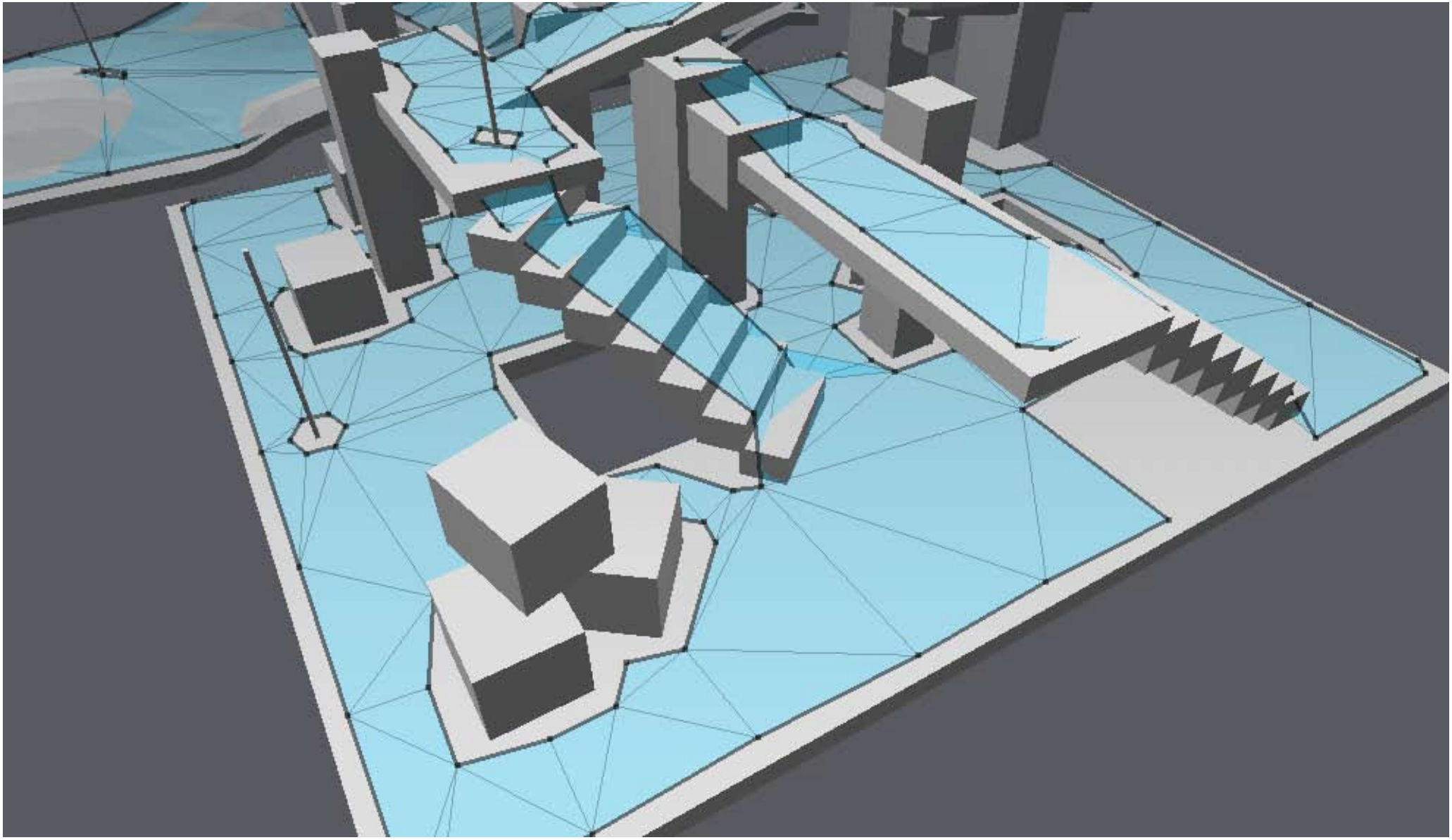
Simplified Contours

Step 5: Triangulate the region polygons and build triangle connectivity

- Now that we have set of simple polygons we can use any simple triangulation algorithm
- The polygons have generally only few vertices, so even n^2 algorithms will suffice
    - we use modified algorithm from Computational Geometry in C
    - instead of clipping the first possible ear clip the ear that has shortest diagonal
        - this generates more optimal triangulation
        - pretty close to Minimum Weight Triangulation

- The final step is to combine the triangles and find edge connectivity
    - we use hash lookup to find identical vertices
    - we use code from “Building an Edge List for an Arbitrary Mesh”

- Could go further and combine triangles to create convex polygons



References

Conservative Voxelization
http://www.cad.zju.edu.cn/~chenwei/research/CGI2007_zhang.pdf

Real-time Voxelization for Complex Polygonal Models
http://www.mpi-inf.mpg.de/~dong/download/PG04.pdf

Single-pass GPU Solid Voxelization and Applications
http://artis.imag.fr/Publications/2008/ED08a/solidvoxelizationAuthorVersion.pdf

GPU Gems 2: Flow Simulation with Complex Boundaries
http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter47.html

GPU Gems 3:
http://developer.download.nvidia.com/books/gpu_gems_3/samples/gems3_ch30.pdf

Way-Finder: guided tours through complex walkthrough models
http://www.lsi.upc.edu/~moving/papers/WayFinder.pdf

Volumetric Cell-and-Portal Generation
http://artis.inrialpes.fr/Publications/2003/HDS03/

Skeleton Extraction of 3D Objects with Visible Repulsive Force
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.13.1699&rep=rep1&type=pdf

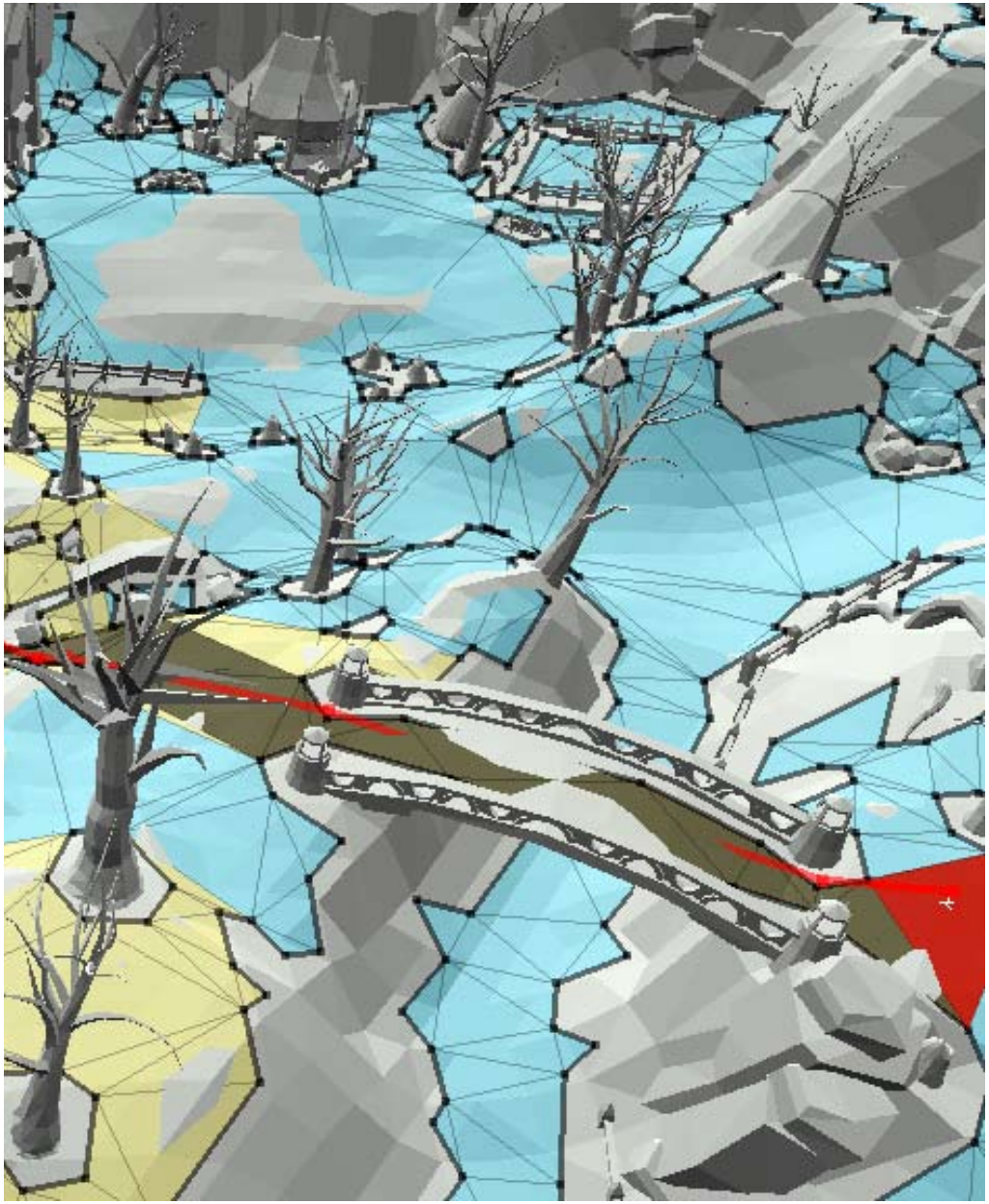
Automated Static and Dynamic Obstacle Avoidance in Arbitrary 3D Polygonal Worlds
http://www.intechweb.org/book.php?isbn=978-953-7619-01-5&content=mostdownloaded&sid=1

A Navigation Graph for Real-time Crowd Animation on Multilayered and Uneven Terrain
vrlab.epfl.ch/~jpettre/publis/05vcrowdpettre.pdf

Pedestrian Reactive Navigation for Crowd Simulation: a Predictive Approach
http://www.irisa.fr/bunraku/GENS/jpettre/

Real World Examples

Models from
AI Game Programming Wisdom 4,
Automatic Generation of Navigation Meshes,
Ratcliff, JW, February 2008

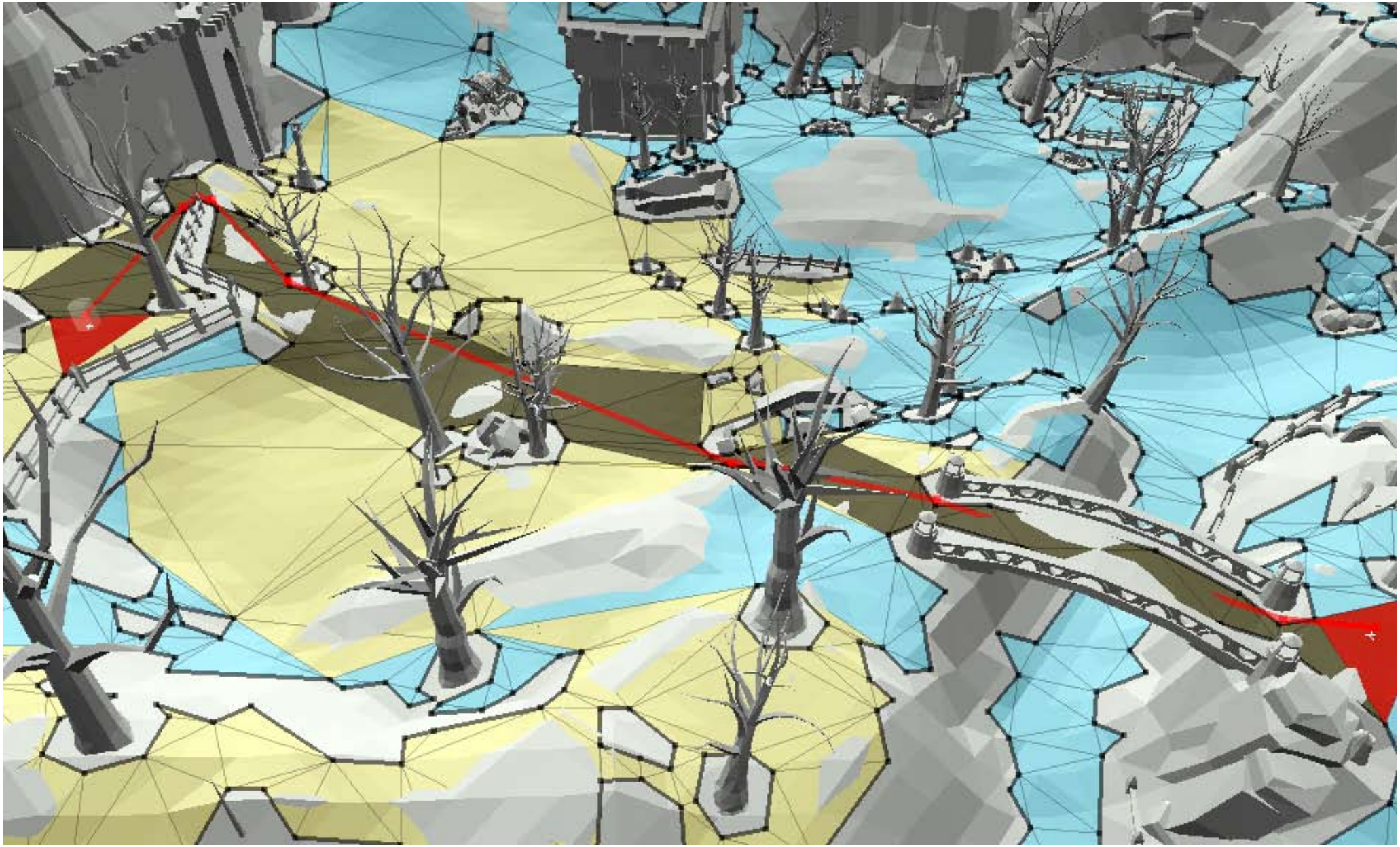


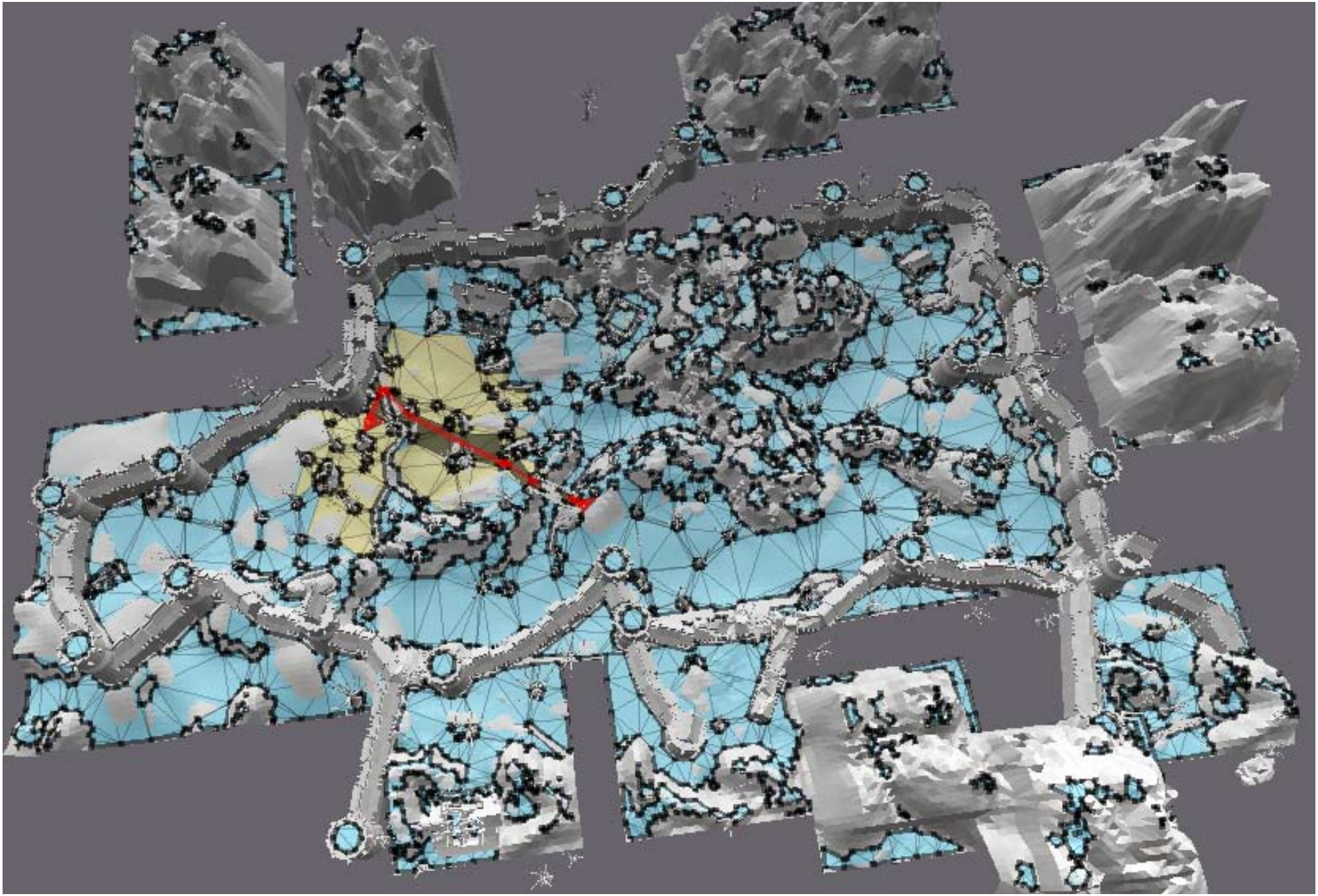
lowlands.obj

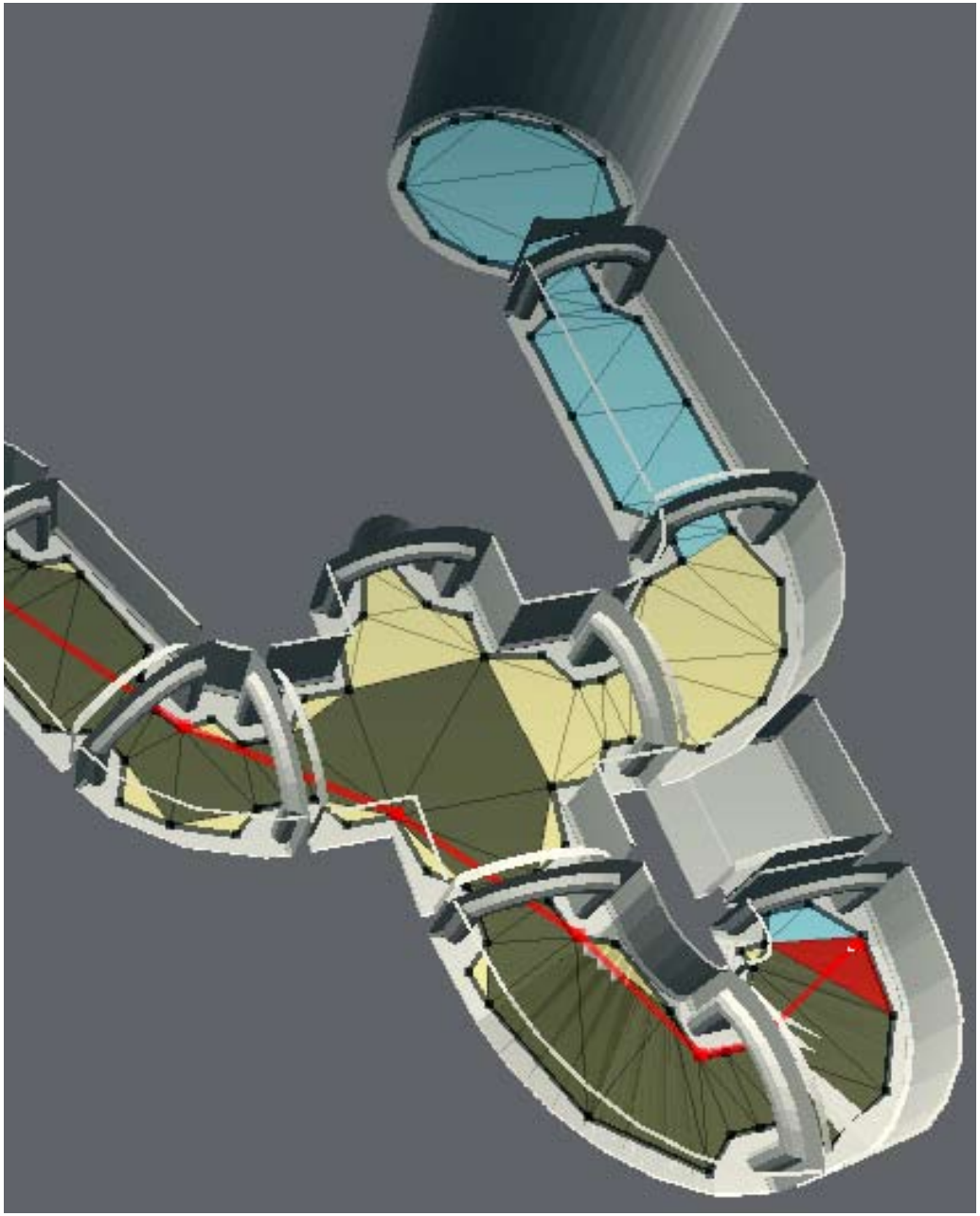
- Input: 185886 verts, 340249 tris
- Voxels: 1287 x 1533
- Navmesh: 5193 verts, 5142 tris
- Navmesh memory: 110.8 kB

Build time: 9414.4 ms

- Rasterize: 1332.958 ms
- Filter border: 117.895 ms
- Filter walkable: 19.402 ms
- Mark reachable: 97.607 ms
- Build compact: 182.808 ms
- Build distance field: 3503.382 ms
- Build regions: 4041.232 ms
- Create contours: 88.155 ms
- Triangulate contours: 8.043 ms





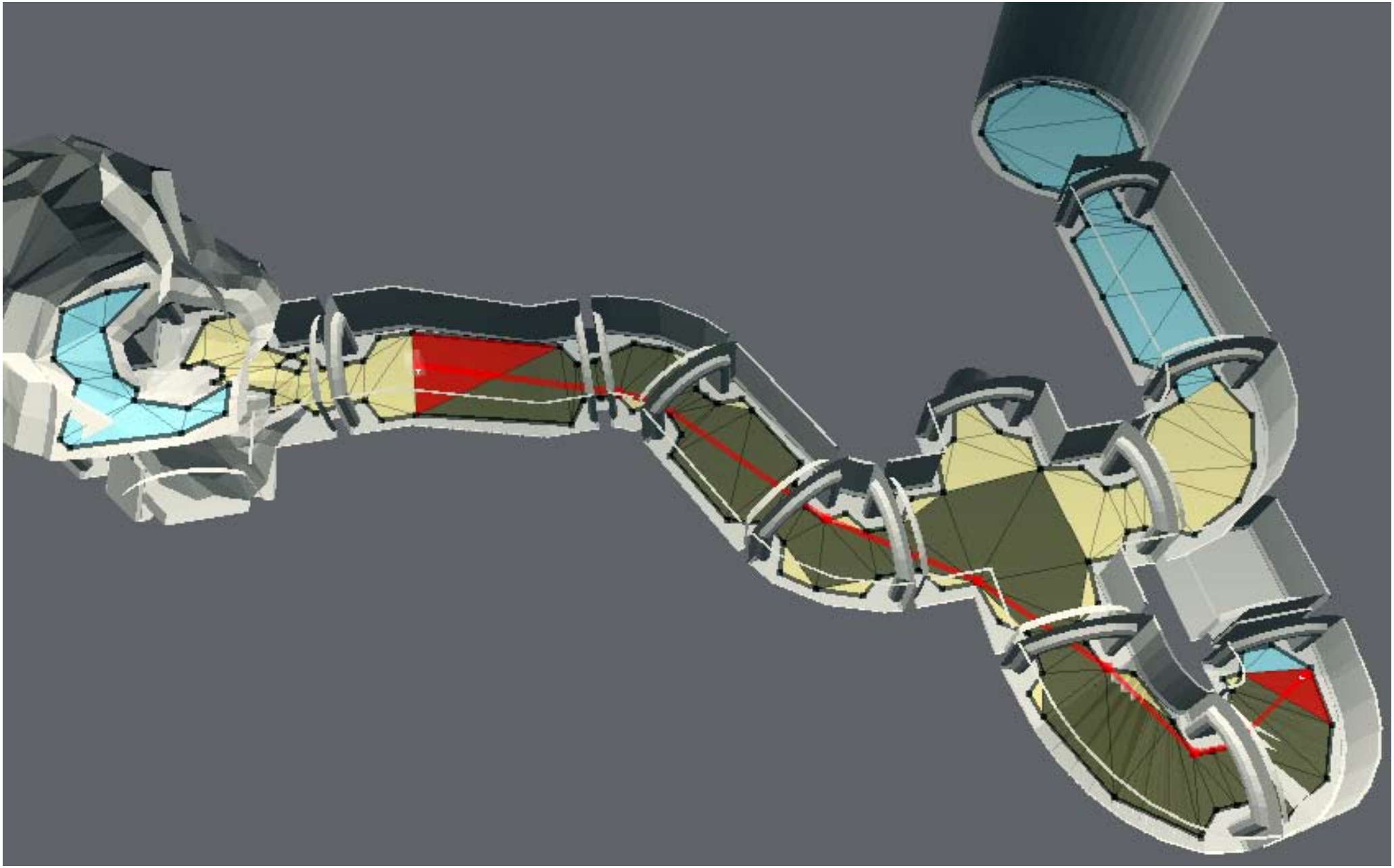


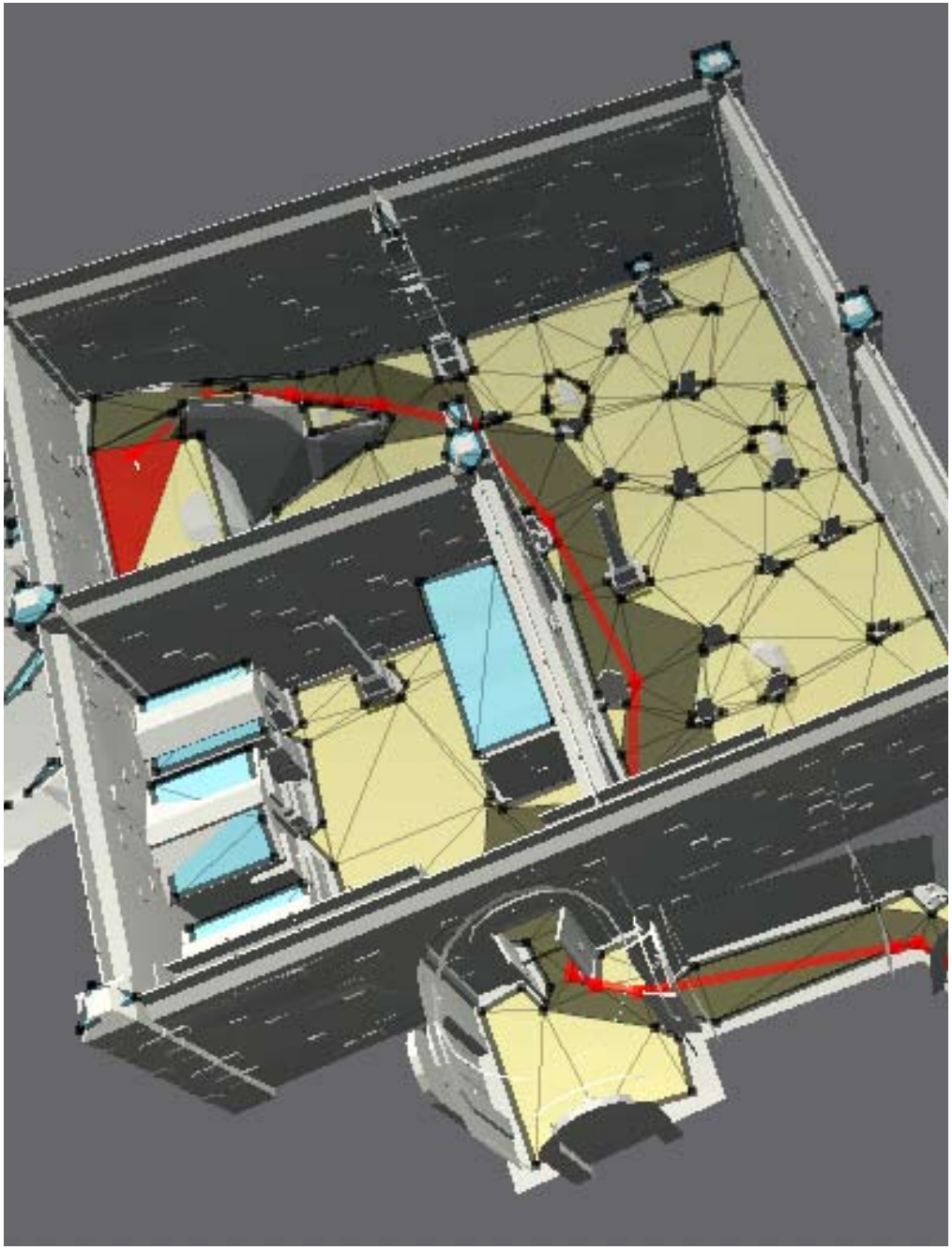
dungeon.obj

- input: 5101 verts, 10133 tris
- voxels: 248 x 330
- navmesh: 171 verts, 165 tris
- navmesh memory: 3.6 kB

Build time: 164.6 ms

- Rasterize: 47.244 ms
- Filter border: 3.154 ms
- Filter walkable: 0.474 ms
- Mark reachable: 2.685 ms
- Build compact: 5.449 ms
- Build distance field: 47.069 ms
- Build regions: 54.436 ms
- Create contours: 2.581 ms
- Triangulate contours: 0.395 ms



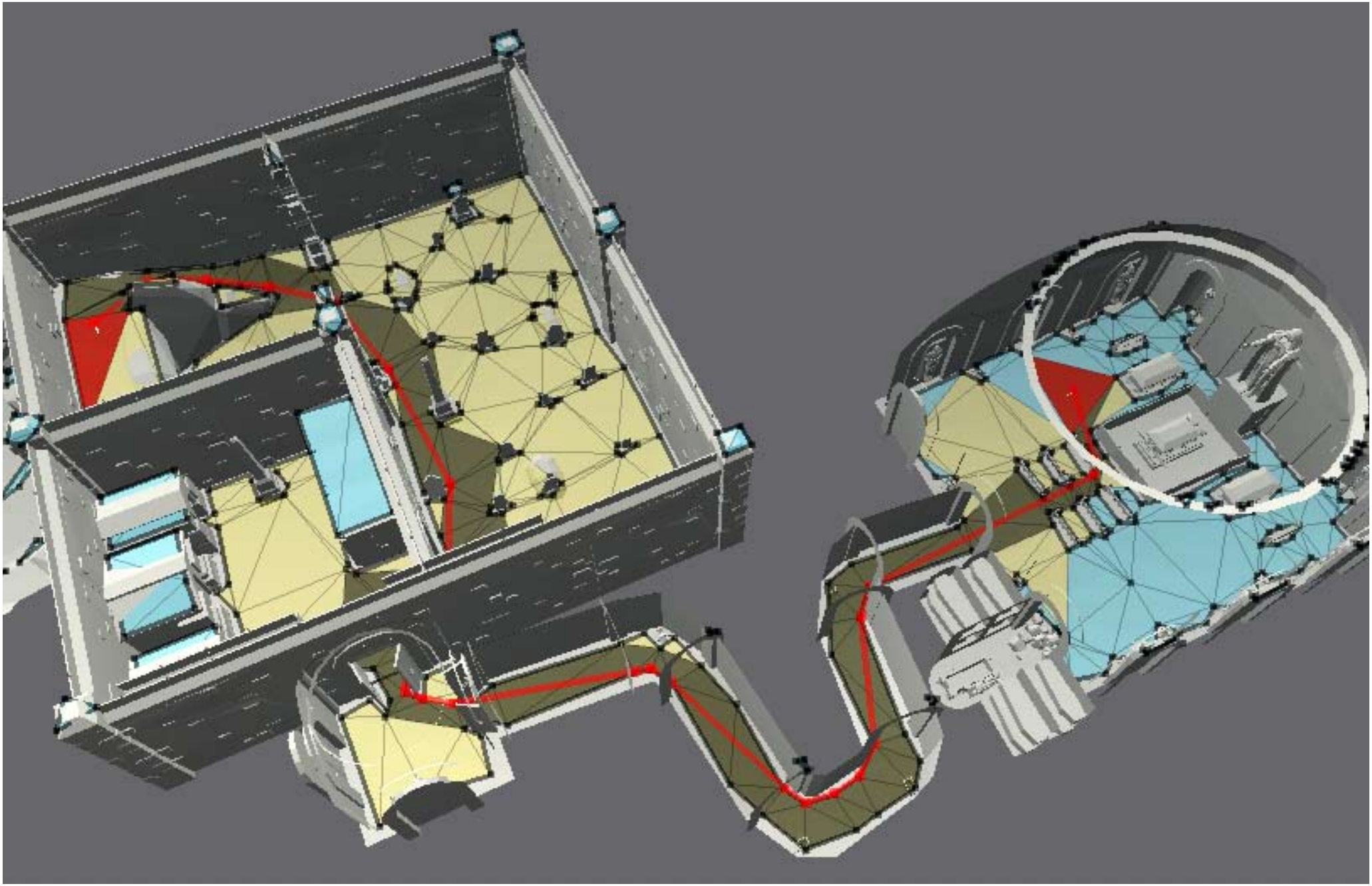


graveyard.obj

- input: 34719 verts, 66737 tris
- voxels: 216 x 390
- navmesh: 673 verts, 625 tris
- navmesh memory: 13.8 kB

Build time: 194.5 ms

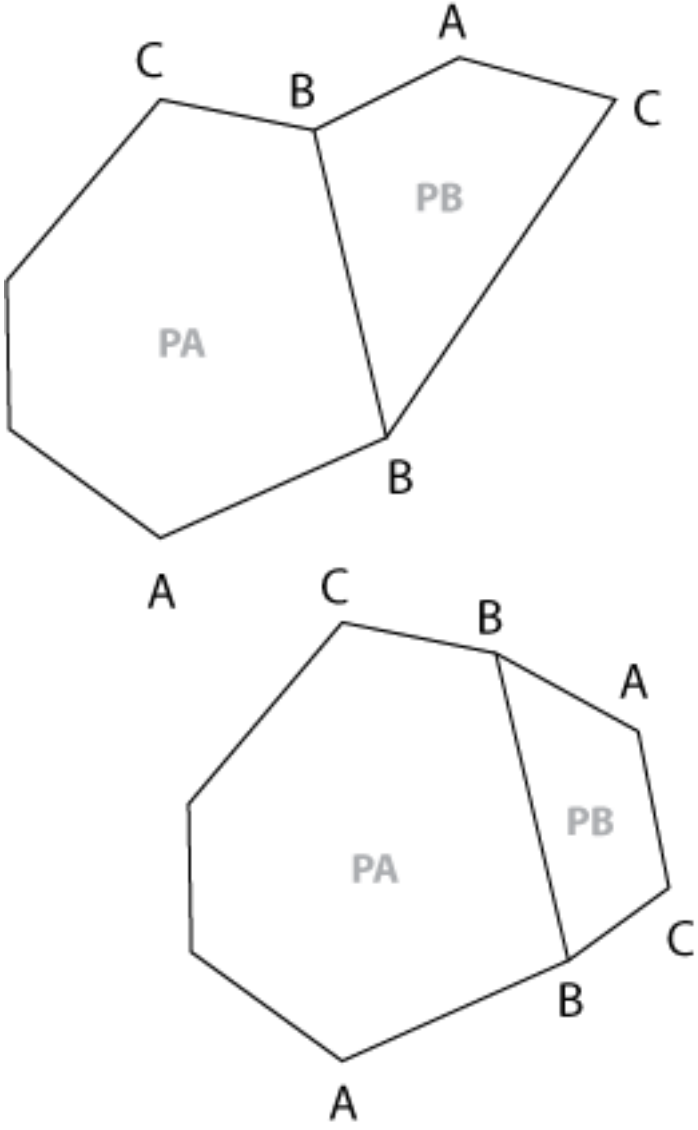
- Rasterize: 91.335 ms
- Filter border: 5.588 ms
- Filter walkable: 0.675 ms
- Mark reachable: 2.832 ms
- Build compact: 5.860 ms
- Build distance field: 36.290 ms
- Build regions: 45.837 ms
- Create contours: 4.008 ms
- Triangulate contours: 0.581 ms



Triangles vs. Convex polygons

- Used simple greedy algorithm
  - executed per region after triangulation
  - no connectivity info
  - complexity probably O(n^3)
  - 4x slower than just triangulation
    - lowlands: 9ms vs. 35ms
    - 0.35% of the total build time

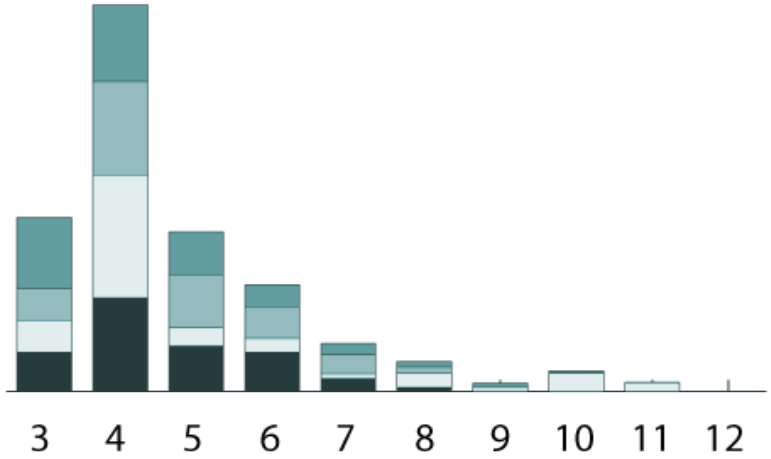
- Algorithm
  - Until cannot simplify anymore
    - find two polygons
      - which can be merged
      - which share the longest edge
    - merge polygons and repeat



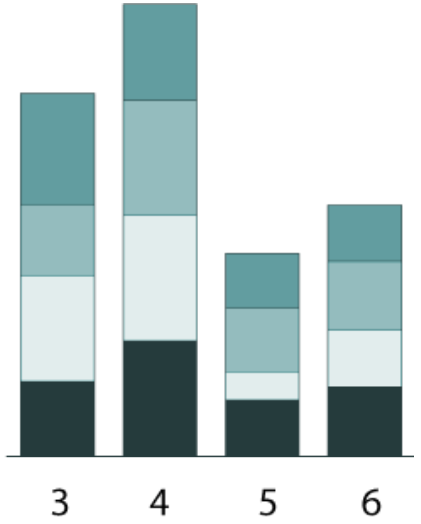
Convex if, Area(A,B,C) <= 0

Triangles vs. Convex polygons

- On average the number of polygons is 40% number of triangles
- Mostly polygons with 3, 4, 5 or 6 vertices.



Max 12



Max 6

	Max 6	Max 12
nav_test.obj	42.0%	38.4%
dungeon.obj	47.3%	31.5%
graveyard.obj	41.5%	36.9%
lowlands.obj	45.9%	40.3%