EECS 494: Terrain Generation

Mt. Rainier, created by Manel Giuli with Terragen

Outline (1 of 2)

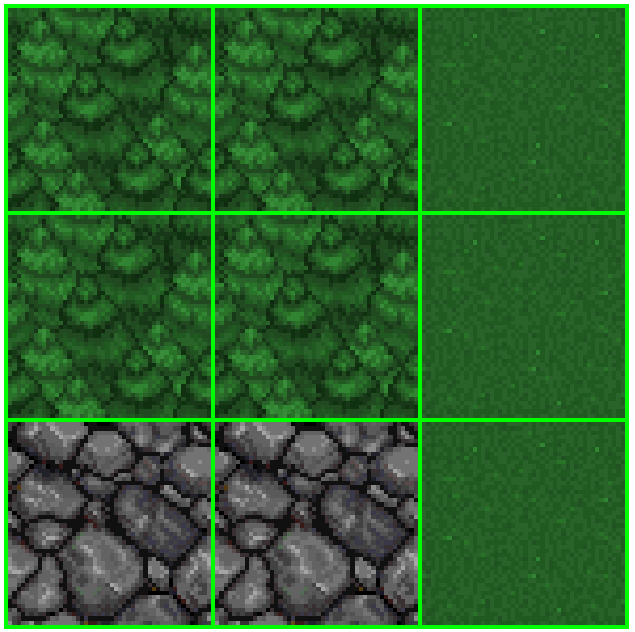
- Possible data representations
  - Tiles
  - Heightmaps
  - Terrain Meshes
  - Mini case-study: Chips
- Autogeneration
  - Fault Formation
  - Midpoint Displacement
  - Particle Deposition
  - Other methods

Outline (2 of 2)

- Tools
  - L3DT
  - Terragen
- Optimization methods
  - Delaunay Triangulation
  - ROAM and LOD
  - Catmull-Clark Subdivision Algorithm

2D Representation: Tiles - 1

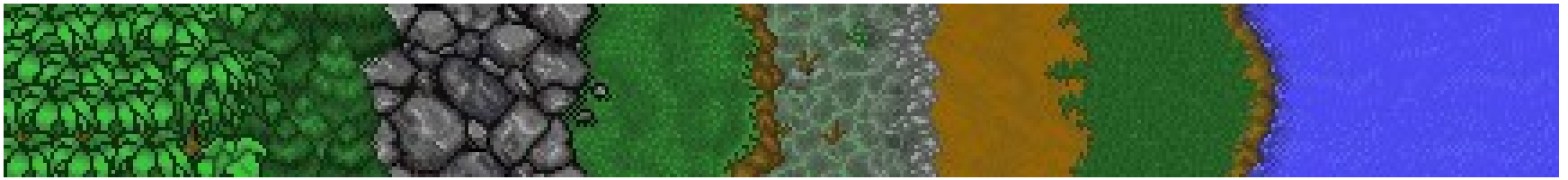
- Pretty simple; you have a set of tiles and a regularly-spaced grid. Each grid in the square is occupied by a tile.
  - Square grid
  - 45º-view square grid (same as the square grid)
  - Hexagonal “grid”



2D Representation: Tiles - 2

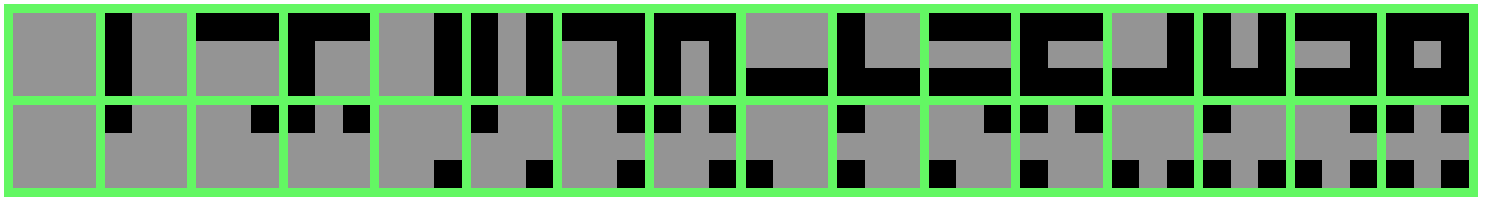
- The tile-based approach is simple, but it potentially suffers from ugliness because the transitions between tiles is much too harsh.
- Many games employ a technique described by David Michael (http://www.gamedev.net/reference/articles/article934.asp) of Samu Games (http://www.samugames.com/): blend transitions between tiles.
- Rather than have lots and lots of tiles to implement transitions, Michael implemented a precendence-and-transparency approach.

2D Representation: Tiles - 3



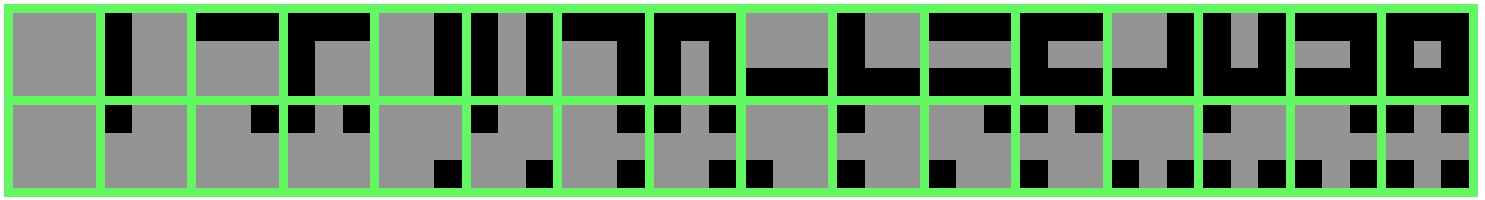
- Here we have the eight tiles for Michael's game's map. Terrains are arranged so that there is a strict precedence.
- As Michael notes, “Precedence does not reflect the relative elevations of the terrain but is instead based on which terrains looks best when overlapping other terrains.”

2D Representation: Tiles - 4

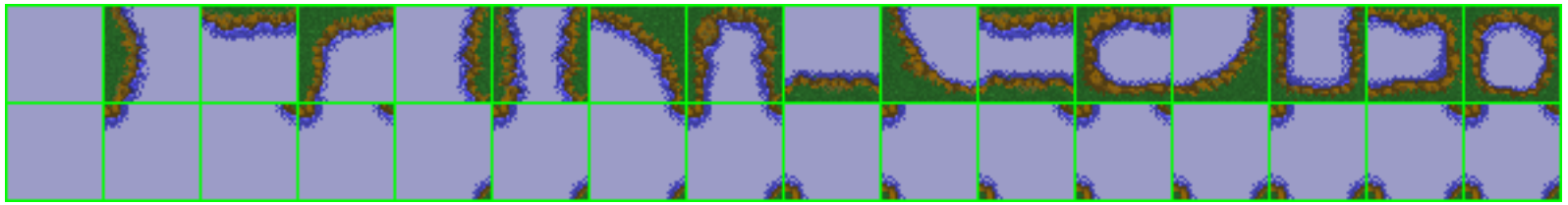


- 16 possible edge transitions, and 16 possible corner transitions.
- We can combine the edge transitions with the corner transitions to create all 256 (???) possible transitions.
- Above are the transition templates.
  - Top are edge transitions
  - Bottom are corner transitions
  - Black areas are transparent; grey are opaque

2D Representation: Tiles - 5



- An artist uses these transitions to create the 32 (30) terrain transitions – for each tile type.
- Many more extra tiles, but not exponentially more.

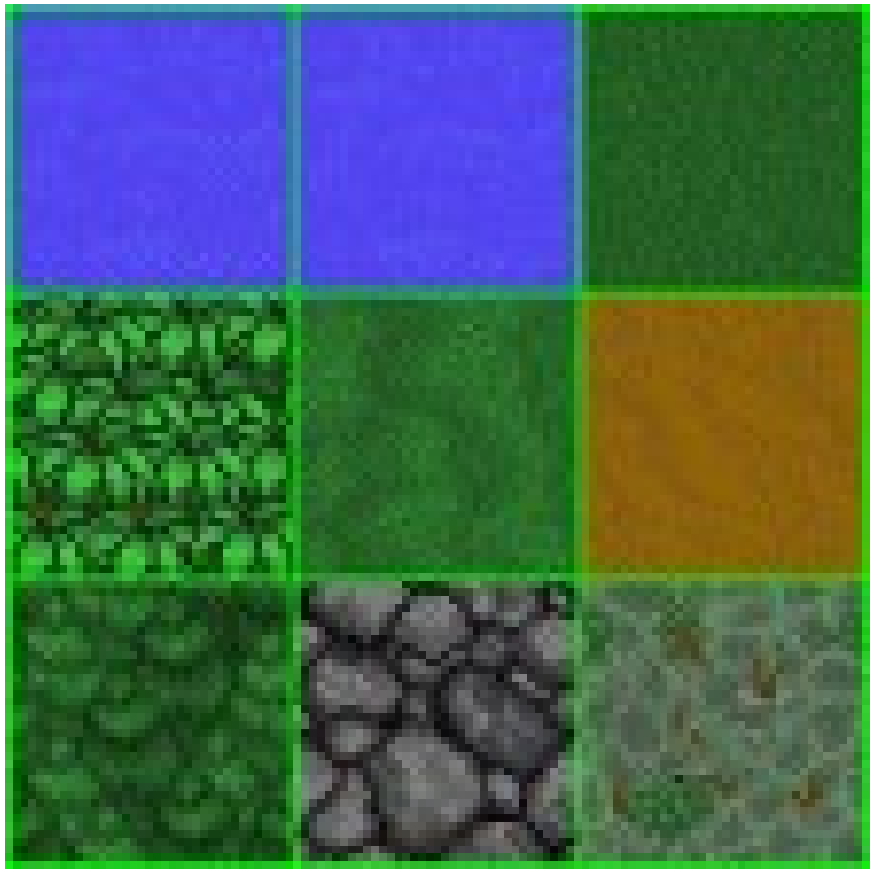


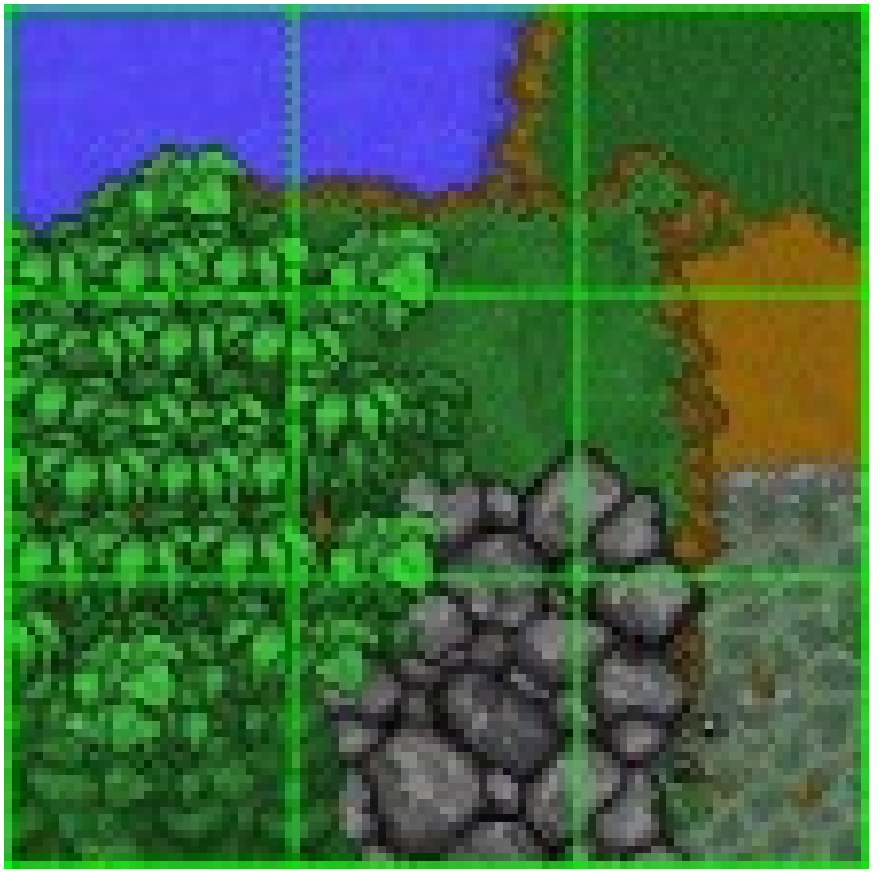
2D Representation: Tiles - 6

- Drawing is a two-step process. For each cell in your grid:
  - Draw the base terrain
  - Then draw any transition overlays, in reverse order of precedence.

2D Representation: Tiles - 7

- Good results. Here's a before and after:

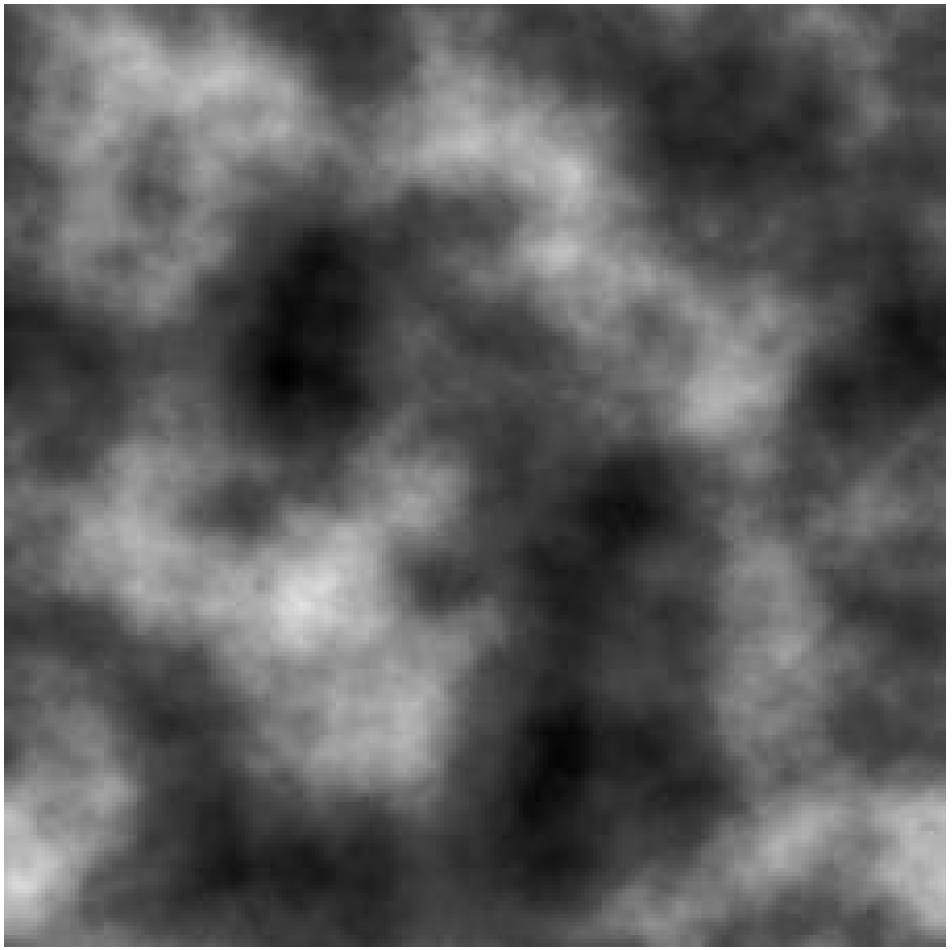


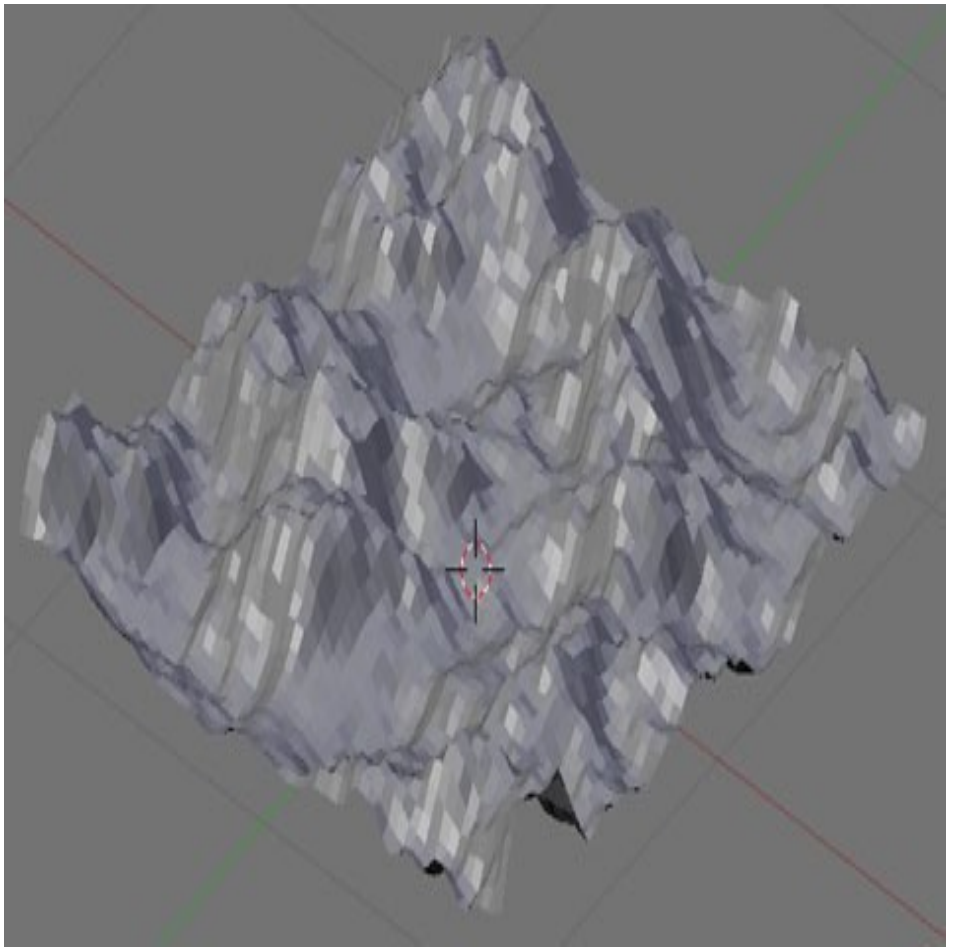


Heightmaps - 1

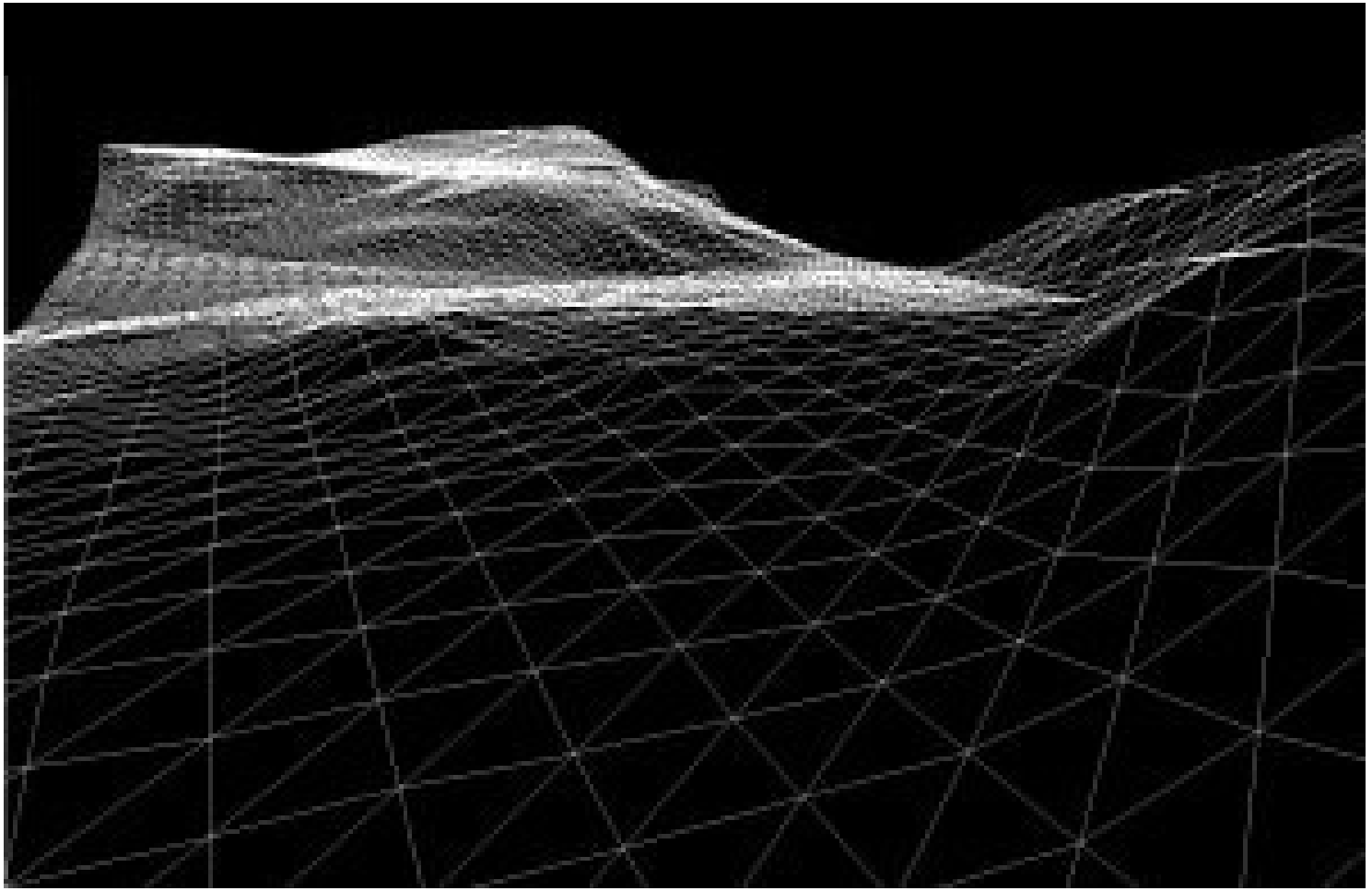
- Essentially a 2D array of pixels
- Normally in the range 0-255
- Black pixels are the lowest; light pixels the highest
- Generally, we use a *clouds* filter in Photoshop, *plasma* in the Gimp, Perlin Noise or some such technique to create a noisy (but smooth) image
- We translate each point in a grid by a factor of the corresponding pixel in the image

Heightmaps - 2



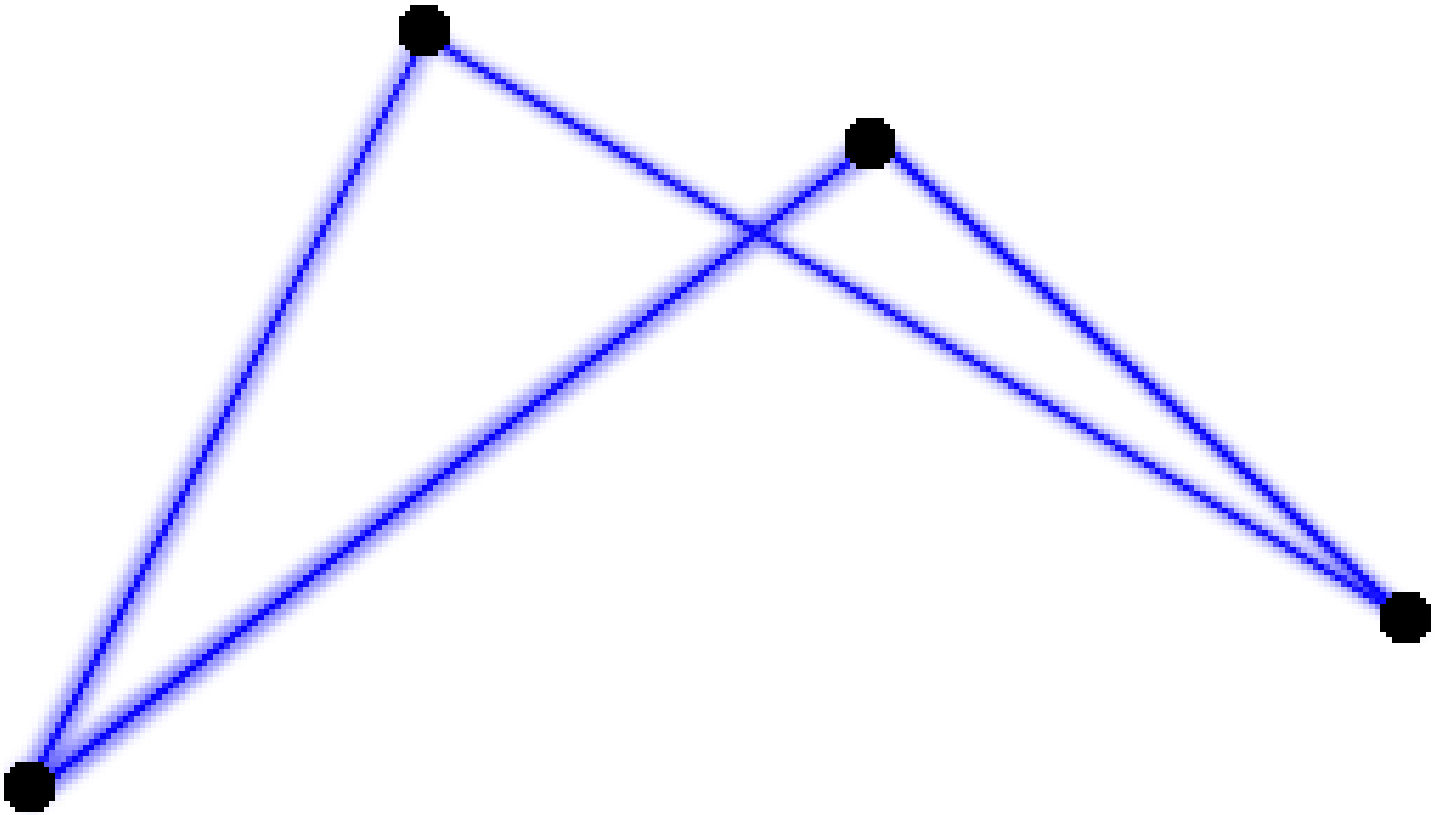


Heightmaps, regularly-spaced grids and triangle-based terrains



Can you spot the potential problem? (hint: think of jagged terrains)

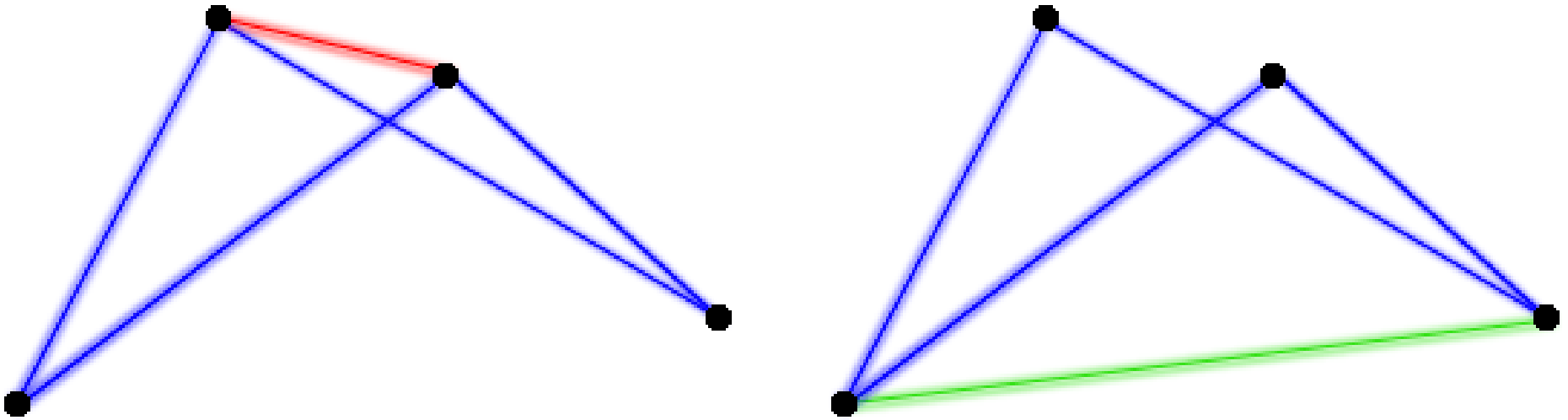
Heightmaps, regularly-spaced grids and triangle-based terrains



How should we triangulate this quad?

Heightmaps, regularly-spaced grids and triangle-based terrains

Two possibilities:

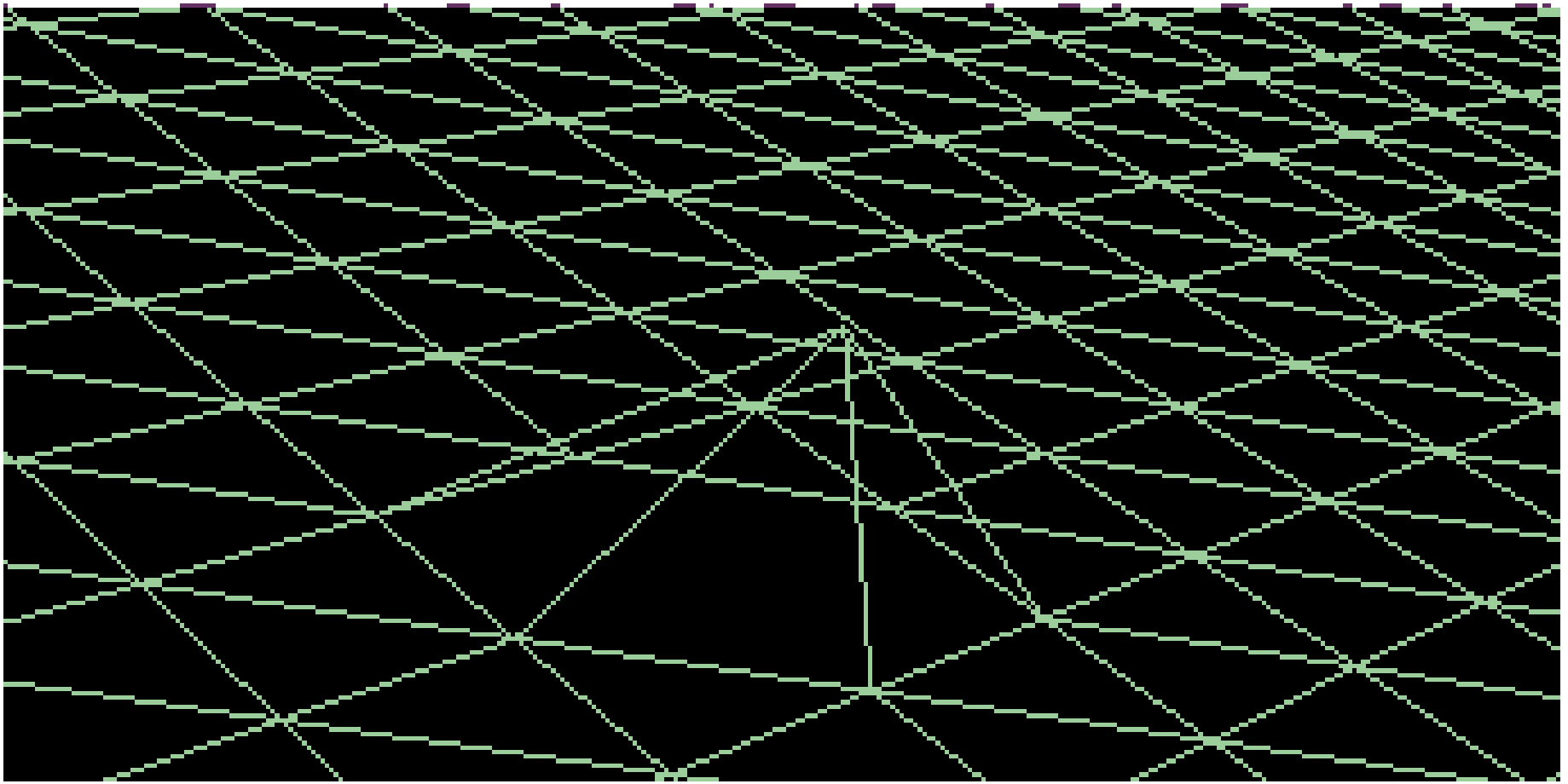


There are a couple of solutions you could try:

- Allow your terrain editor to switch diagonals for a given quad

- Constrain all quads to be planar

Heightmaps, regularly-spaced grids and triangle-based terrains

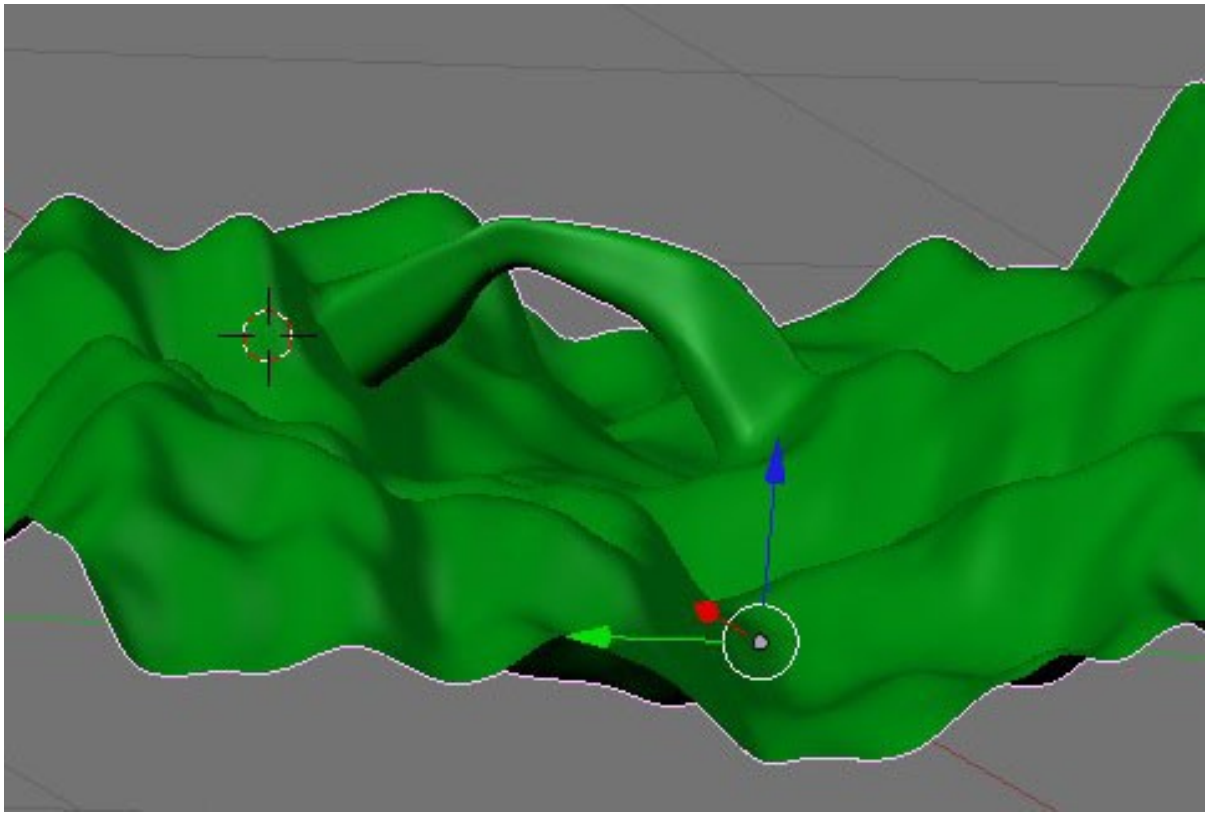


Heightmap limitations

- What kind of terrain formations can we *not* represent with a simple heightmap?

Heightmap limitations

- What kind of terrain formations can we *not* represent with a simple heightmap?



Also overhangs, “Devil's Tower”-type landforms, etc.

Heightmap limitations

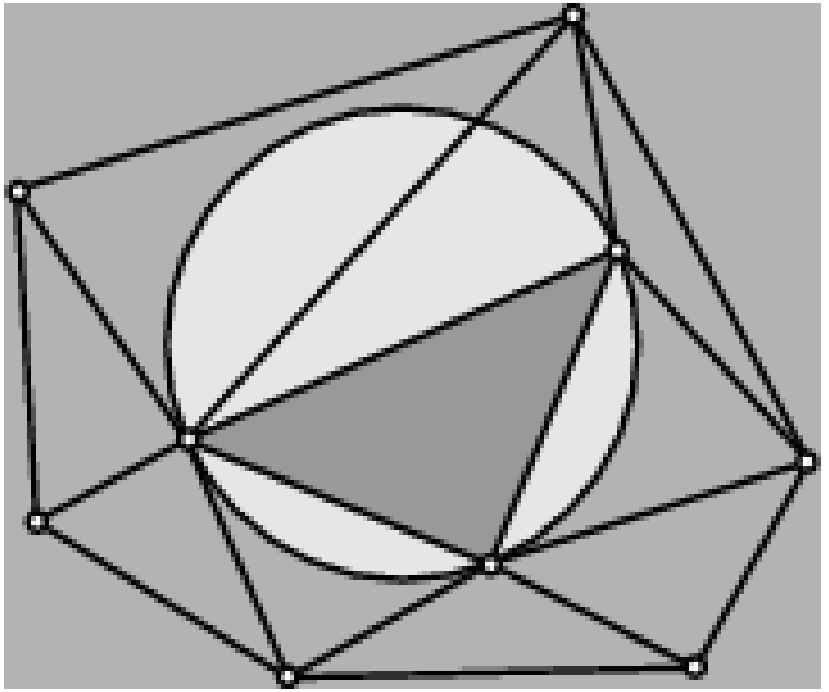
- There are a few ways we can “solve” this problem:
  - Only use heightmaps for the initial generation; then, sculpt terrain as we see fit, and be sure not to do anything with the terrain that requires 2D topology
  - Treat these types of formations as non-terrain objects, which we place on the terrain grid AFTER formation
  - Apply “modifiers” to the terrain, which non-destructively transform the geometry, late in the terrain pipeline
  - Have a layer of, say, three heightmaps – the bottom one of which is our base, the middle one a kind of “ceiling” heightmap, and the top one another base. Intersections of the heightmaps are the boundaries at which it changes overhang to underhang, and vice versa.
    - This won't work for bridge-under-a-bridge, though

Delaunay Triangulation

- We want to figure out a way to reduce the number of vertices and triangles in our terrain mesh. We also want to reduce the number of very thin *"sliver"* triangles in our mesh; probably we can render the same scene with fewer, larger triangles.
- This is where Delaunay Triangulation becomes useful.
- Intuitively, Delaunay Triangulation takes a set of points and triangulates them.
- http://www.cs.cornell.edu/Info/People/chew/Delaunay.html

Delaunay Triangulation

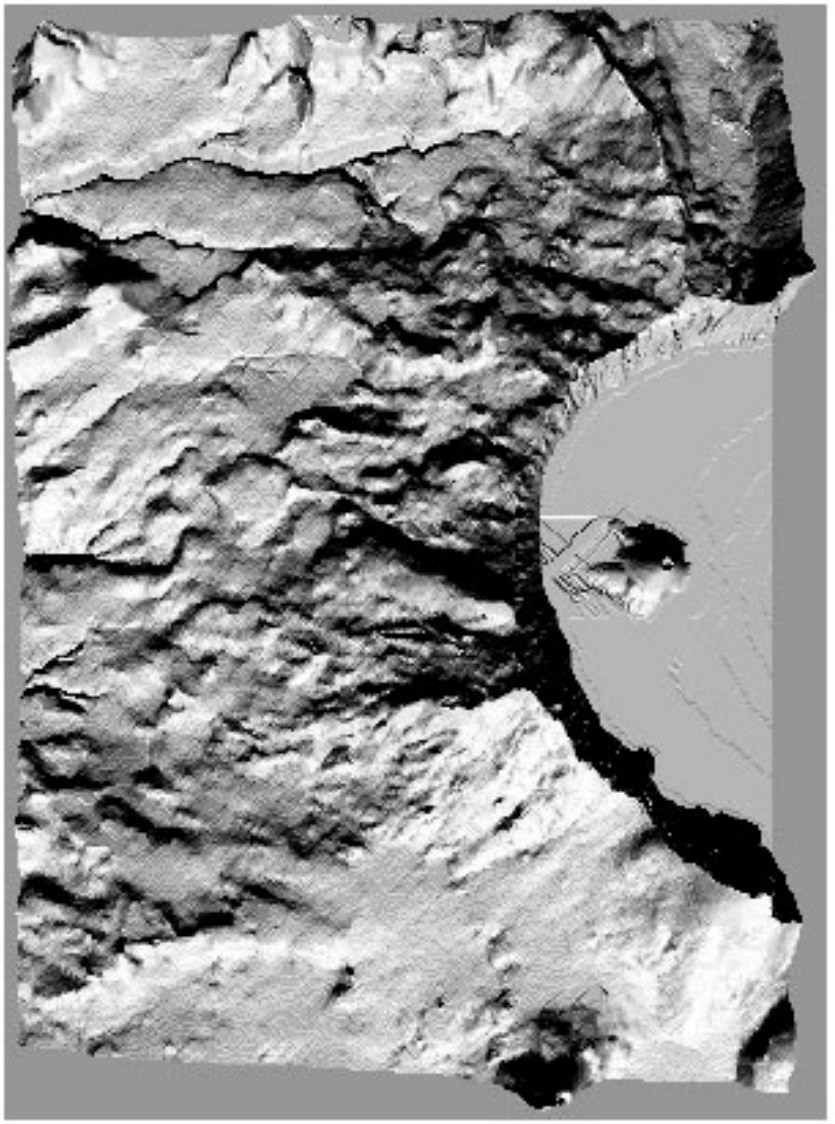
- The Delaunay Triangulation, in a nutshell, is one in which every edge has satisfied the Delaunay property. That is to say, the circumcircle that contains the vertices of the edge and the other one of the first triangle does not contain the opposing vertex (and vice versa).
- Delaunay triangulations are unique.

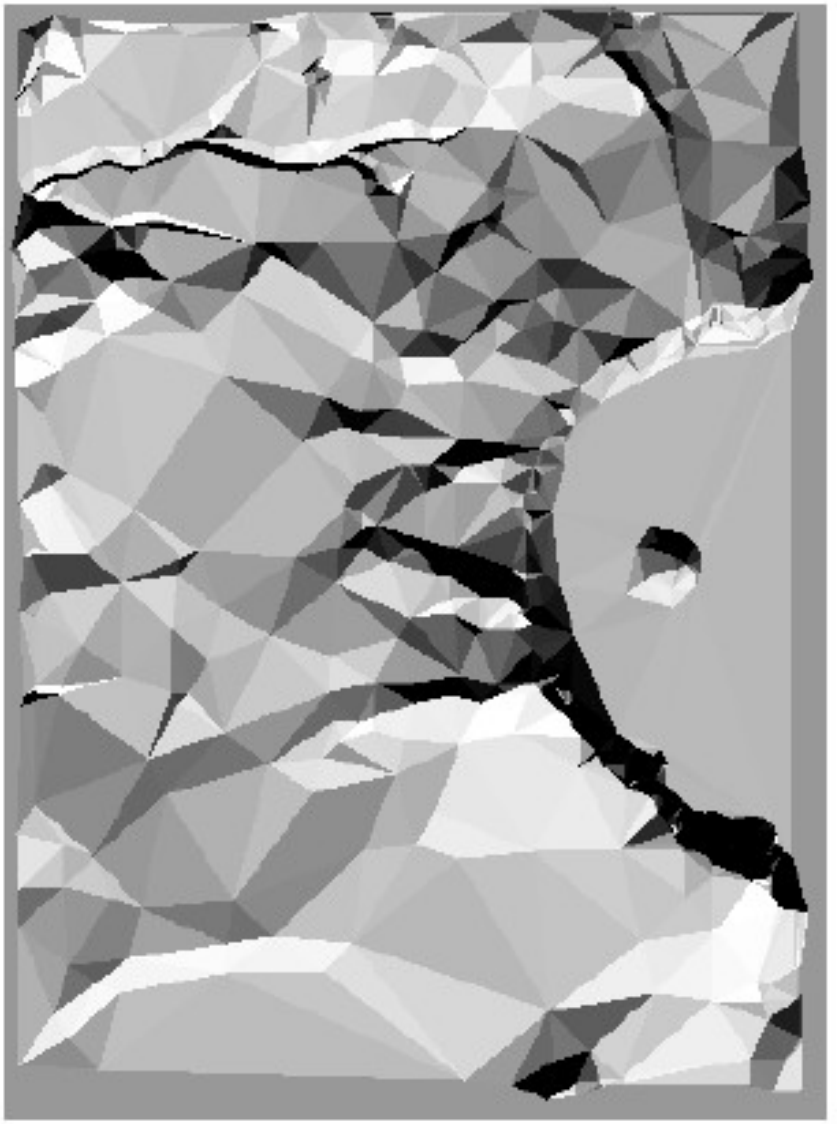


Delaunay Triangulation

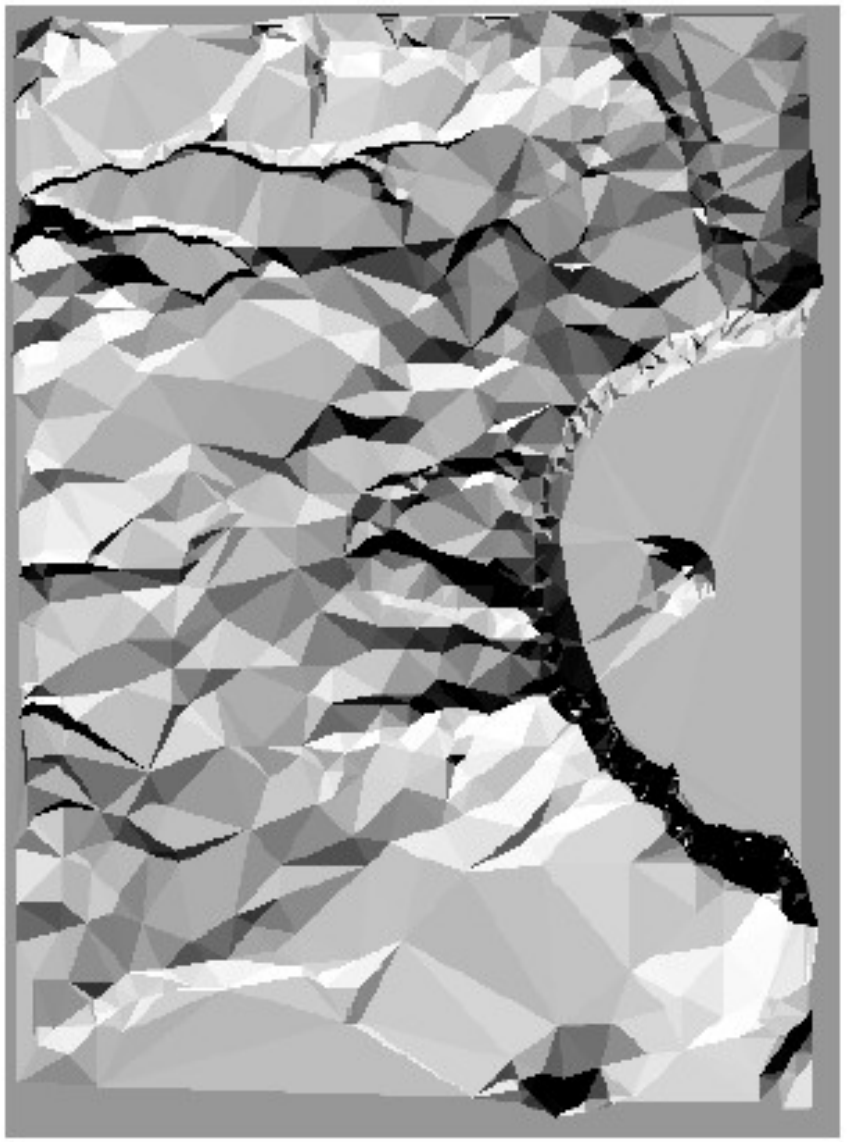
- Michael Garland and Paul S. Heckbert described an algorithm that takes as input a heightmap and creates a Delaunay triangulation for that heightmap. It is an iterative refinement algorithm; we stop when the error is "*small enough.*"
- Greedy algorithm
- http://www.bowdoin.edu/~ltoma/teaching/cs350/spring04/Handouts/scape.pdf

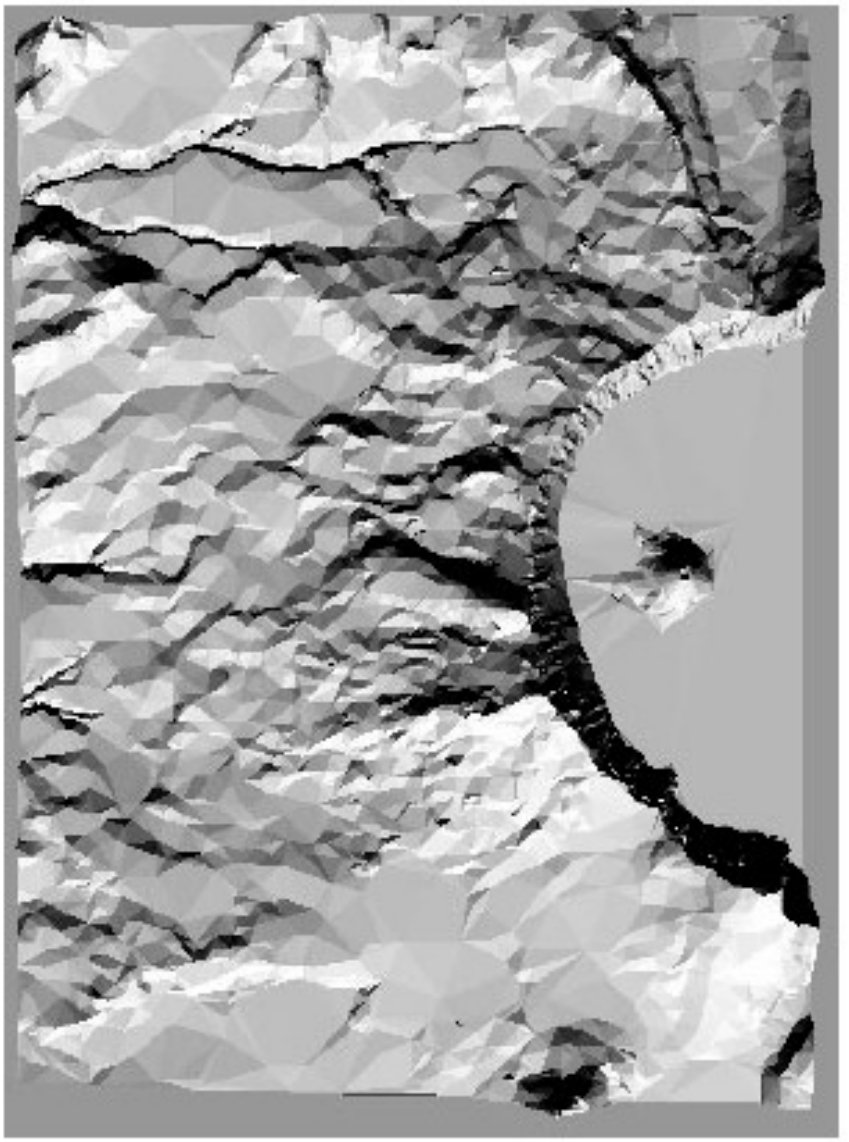
Delaunay Triangulation



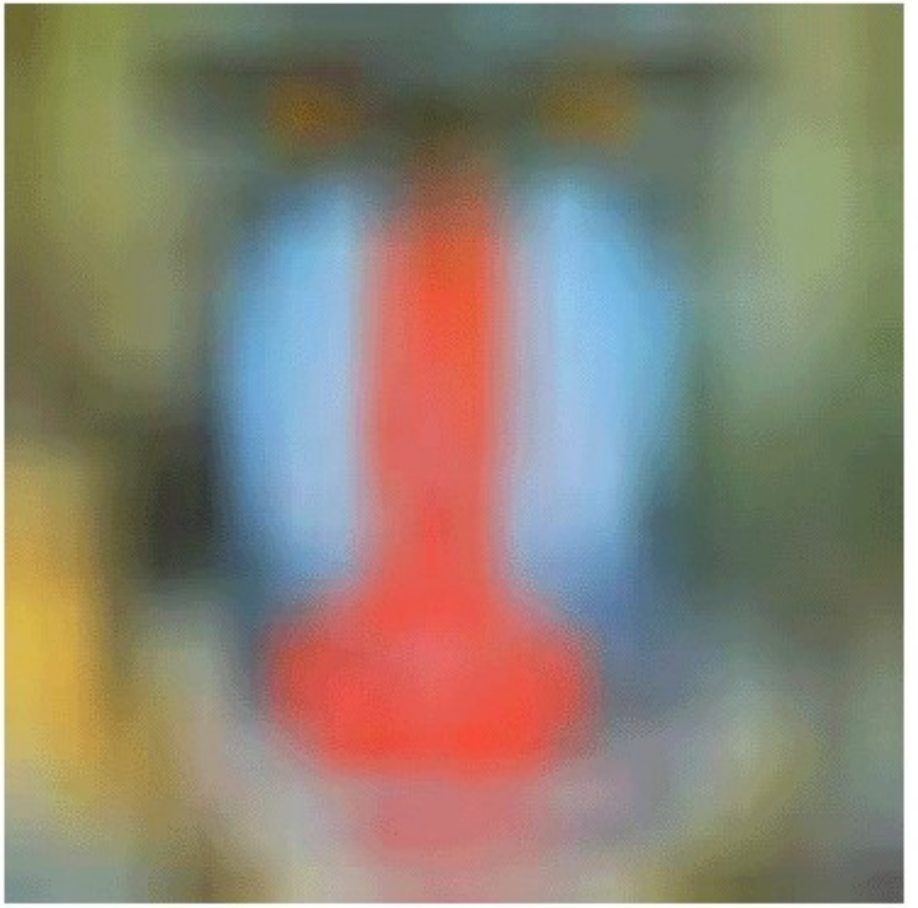


Delaunay Triangulation

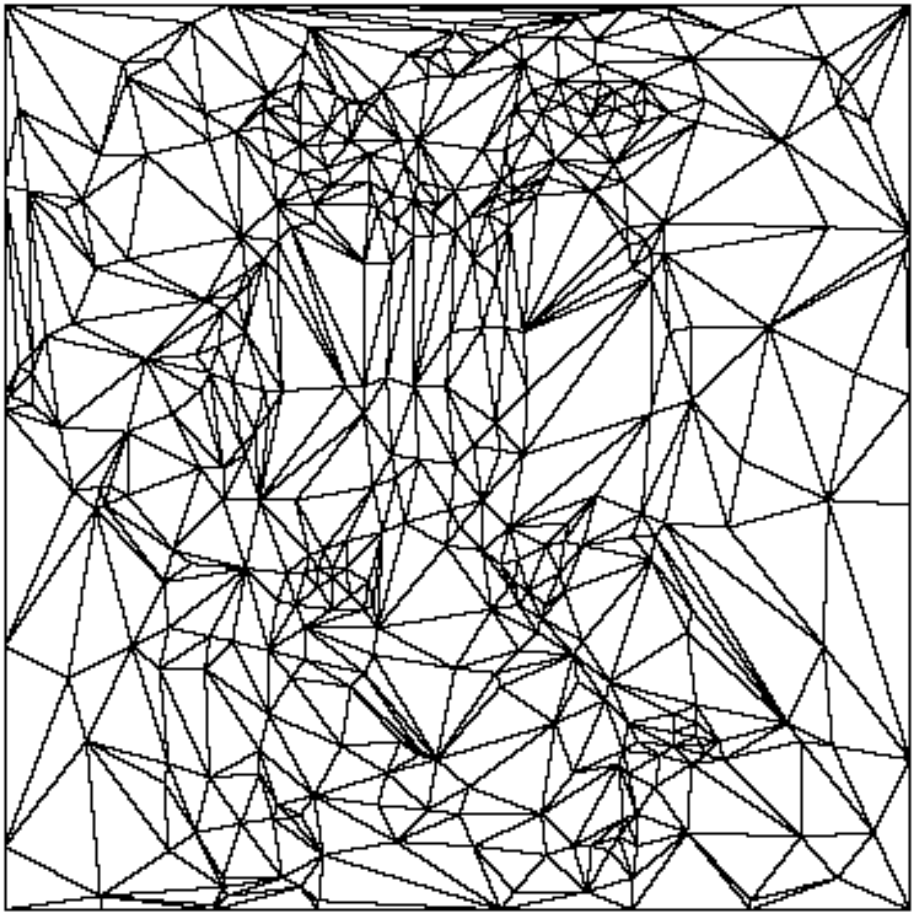




Delaunay Triangulation



Delaunay Triangulation



“Chips”

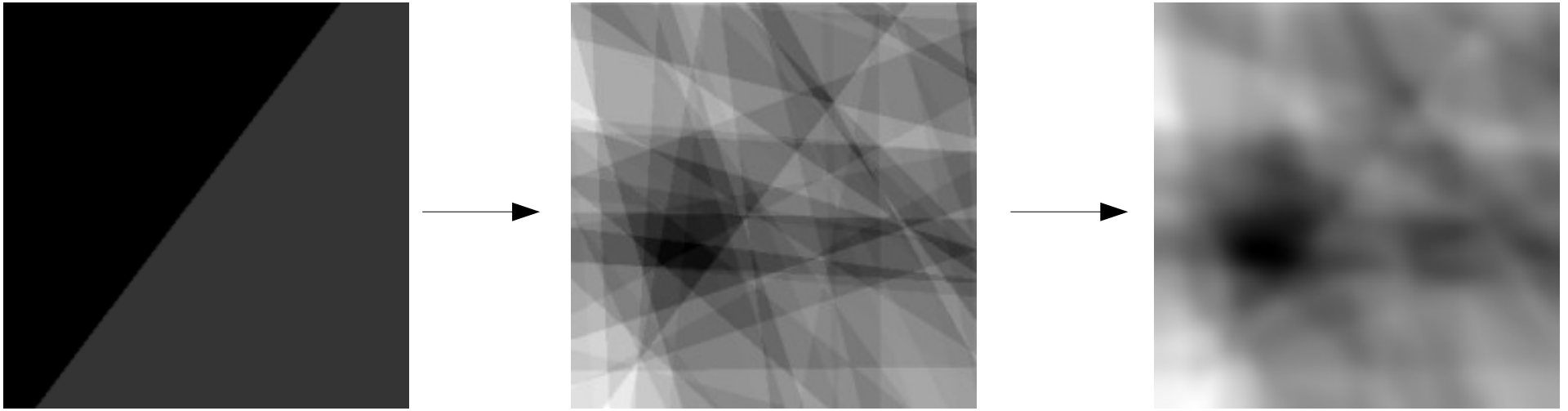
- Has many of the benefits of both tile-assembly and heightfield-like surfacing
- Create a set of “*chips*” with a heightmap terrain tool. These chips don't have to be square; they can be long, or L-shaped, for example.
- Assemble the chips in another tool. Where chips meet, add points (a la Delaunay, or *Contrained* Delaunay), and average heights (or better yet, smooth across an area).

Terrain Autogeneration

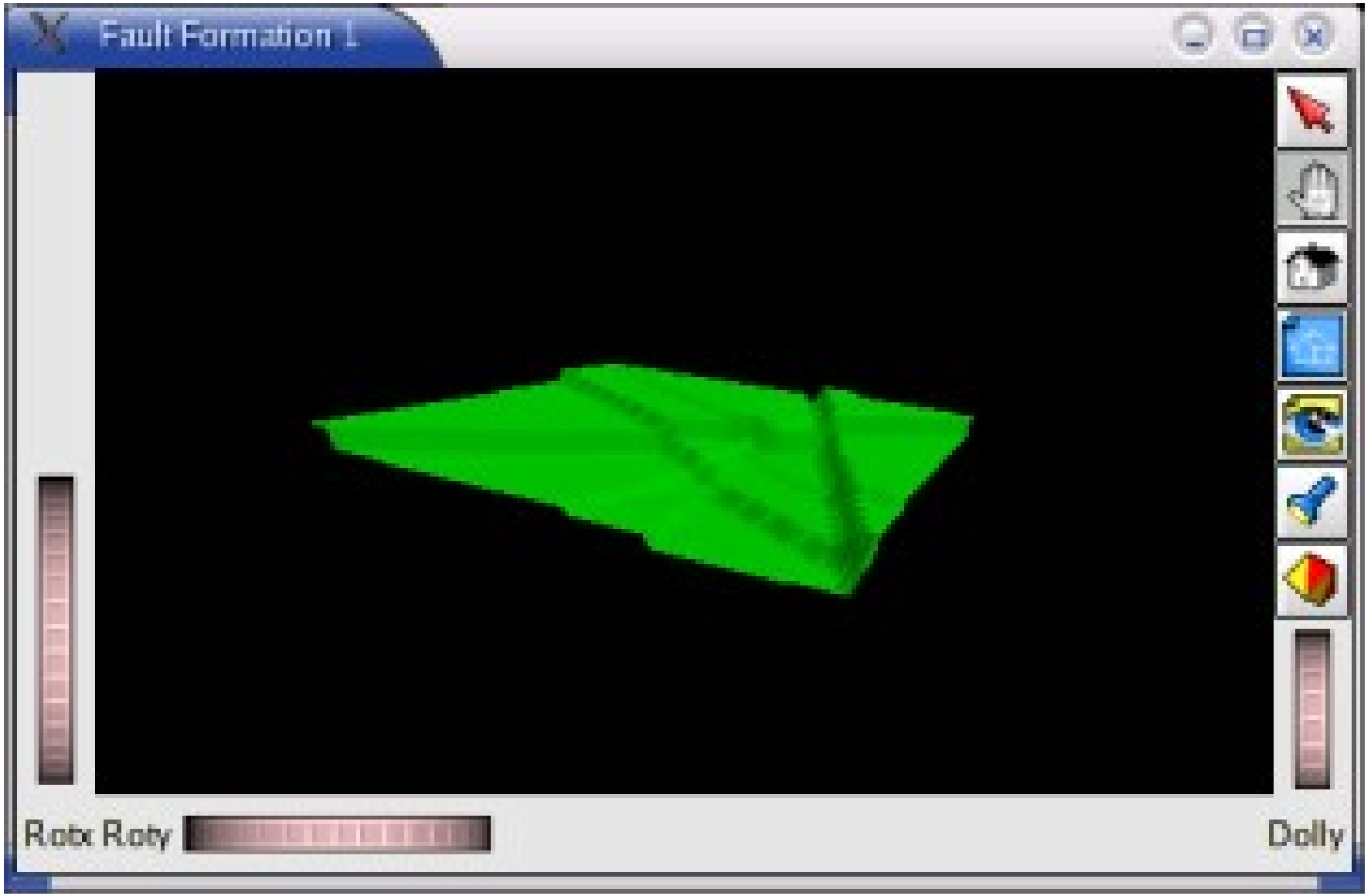
- It's easier, and more realistic in many cases, to get the computer to create the landscape for us
- These are often referred to as *“fractal” terrain generation*
- These techniques are especially useful for VERY BIG maps and dynamic maps. For big maps, we might not have enough resources to pay artists to create big worlds for us. For dynamic maps – well, they're created at runtime!
- Especially for static maps, we can always tweak the map after autogeneration with a 3D tool

Terrain Autogeneration – Fault Formation

- First, all cells of the heightmap to 0.
- Draw a line across the cell. Raise all cells on one side of the line by a certain amount.
- Continue like this, drawing lines and raising one side – but with each iteration, we decrease the amount by which we raise
- If we want a smoother terrain, no problem – just apply a smooth (e.g. blur) filter

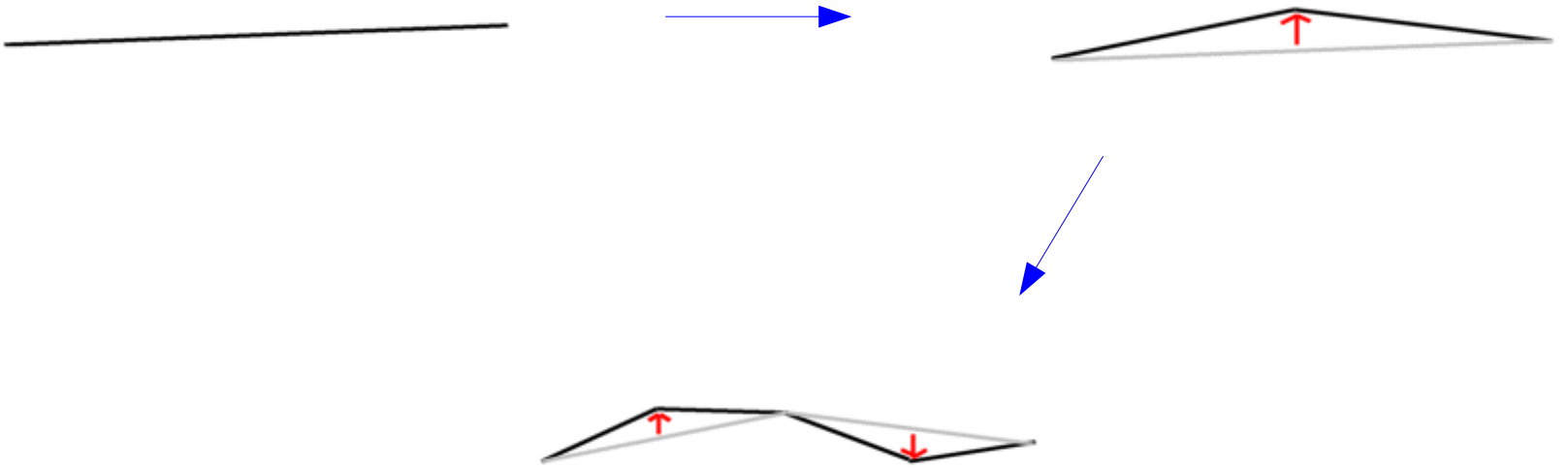


Terrain Autogeneration – Fault Formation



Terrain Autogen. – Midpoint Displacement

- For the 2D case: take a line segment and find its midpoint.
- Insert a vertex in that midpoint, and translate it up or down by some random amount.
- Recursively apply the same procedure on the two new (connected) line segments

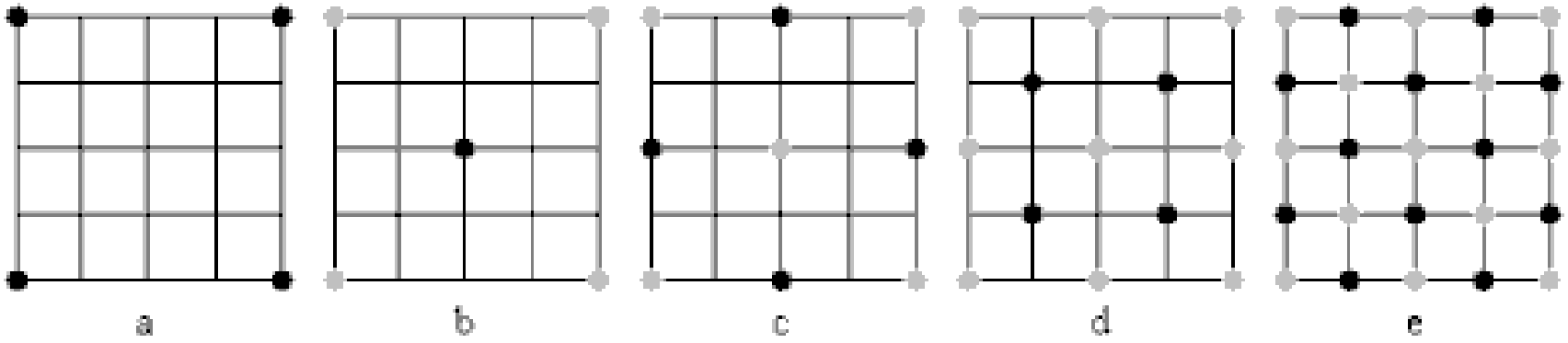


Terrain Autogen. – Midpoint Displacement

- Continue until you reach a “sufficient level of detail”
- Useful for 2D autogeneration

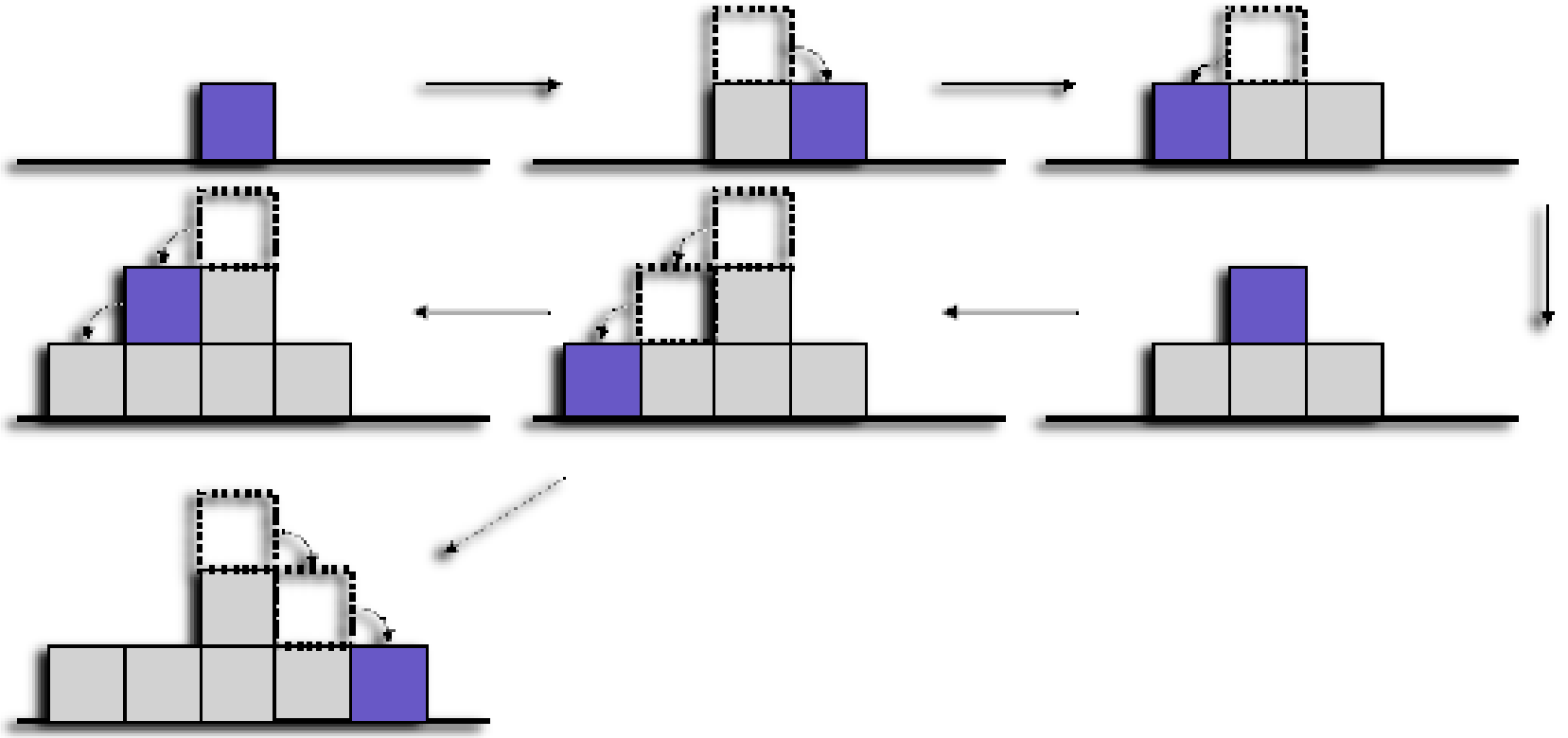
Terrain Autogen. – Midpoint Displacement

- 3D is the same idea; we just operate and recurse on square fragments of the map.
- The *Diamond-square algorithm*



Terrain Autogen. – Particle Deposition

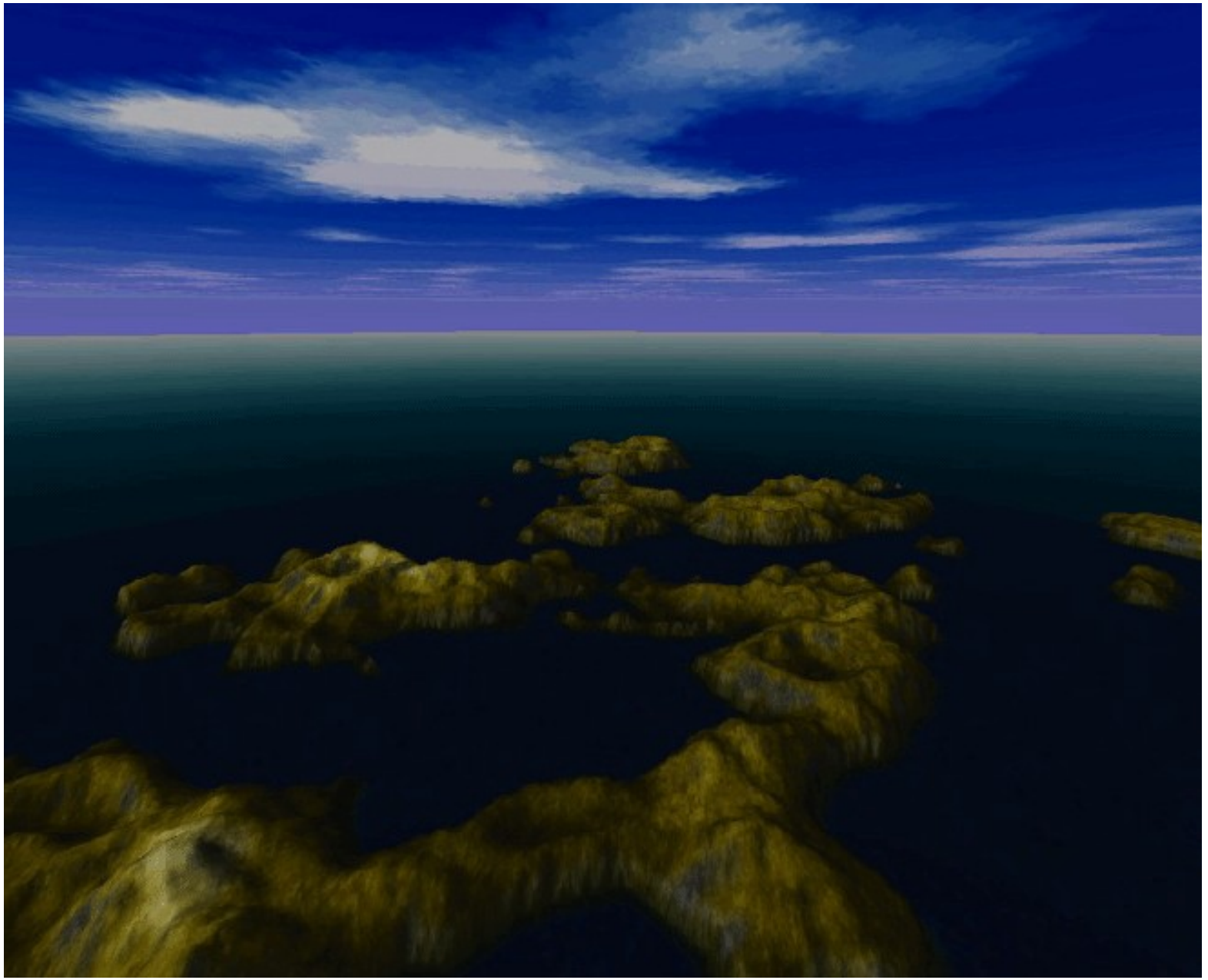
- The idea is that we choose a point on the map and *deposit* particles (that is, add to the heightmap) over and over again.
- Particles will “fall” to their neighbor – *which* neighbor is randomly determined - if they can.



Terrain Autogen. – Particle Deposition

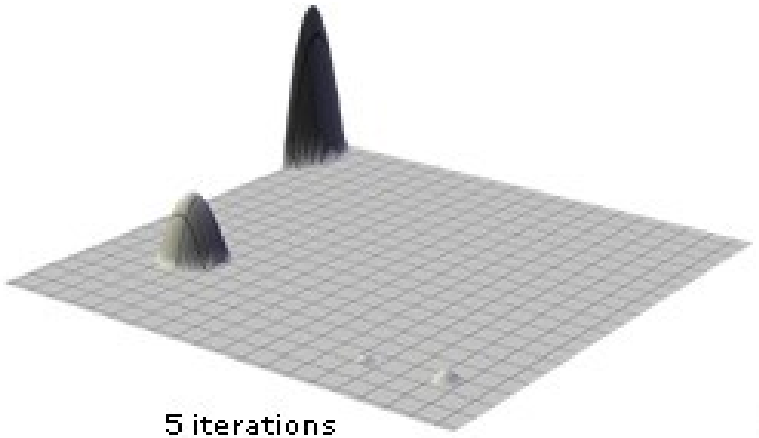
- We can occasionally move the point of deposition so as to not create one big island or mountain
- We can also create calderas by treating deposits past a certain threshold as *subtractions* in height rather than *additions*. These subtractions are applied at the end (effectively *inverting* those areas)

Terrain Autogen. – Particle Deposition

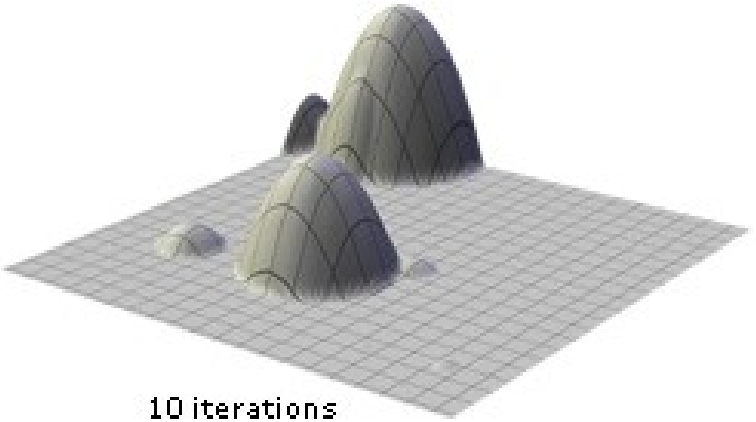


Terrain Autogeneration - *Hills*

http://www.robot-frog.com/3d/hills/hill.html



5 iterations



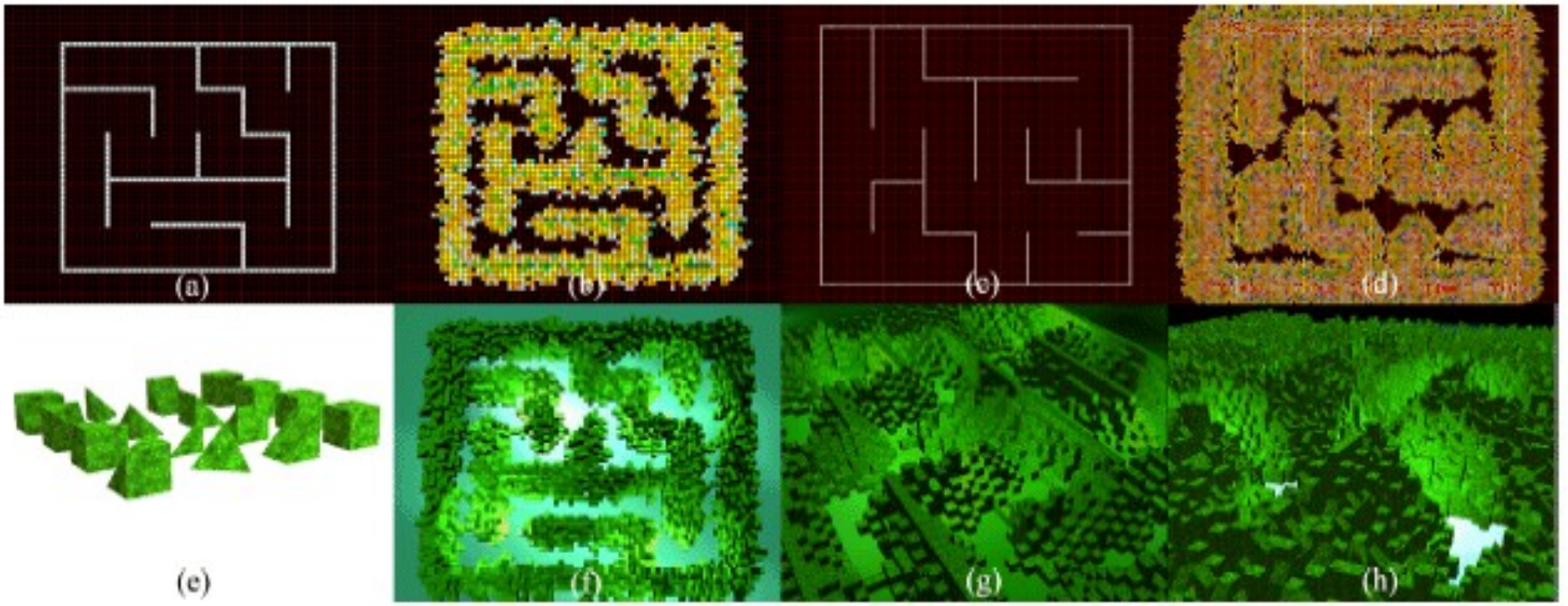
10 iterations

50 iterations

200 iterations

Terrain Autogeneration – *Cellular Automata*

http://www.cs.ubc.ca/~van/GI2005/Posters/Wijaya_Callele_GI2005_Poster_Abstract.pdf



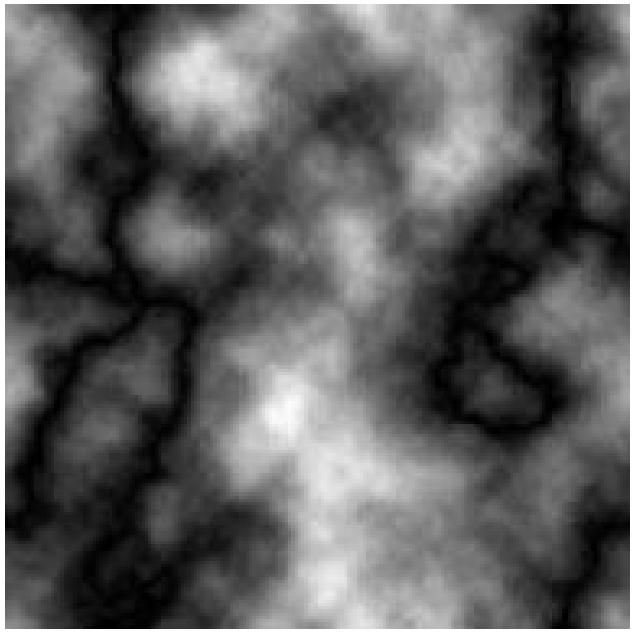
Terrain Autogeneration – Which is best?

- Which method is best?
- Well, there is no “best.” Different techniques produce different results.
- Several techniques in concert (that is to say, in additive sequence) can be used to get the best of all worlds

Terrain Autogeneration – Autotexturing

- We can get automatically texture terrain based on the features of the terrain. One simple example might be:
  - Region1(Snow) : 256-192
  - Region2(Rock) : 192-128
  - Region3(Grass): 128-64
  - Region4(Sand) : 64-0
- At the boundaries, we blend between features
- As we'll see, programs like L3DT do this – and we can configure which textures correlate to which heights

Terrain Autogeneration – Autotexturing



L3DT

- Very sophisticated autogenerator
- Very large maps possible
- Start with a “design map” - which you can alter – that serve as *seeds* for the creation of terrain features
- Lots of realistic terrain-creation techniques; e.g. several types of erosion (channeling erosion, thermal erosion, etc.); water table (flooding) creation; normal creation; attribute creation; texture creation... and more
- You can export all of these generated tables – e.g. as a BMP
- http://www.bundysoft.com/L3DT/

L3DT

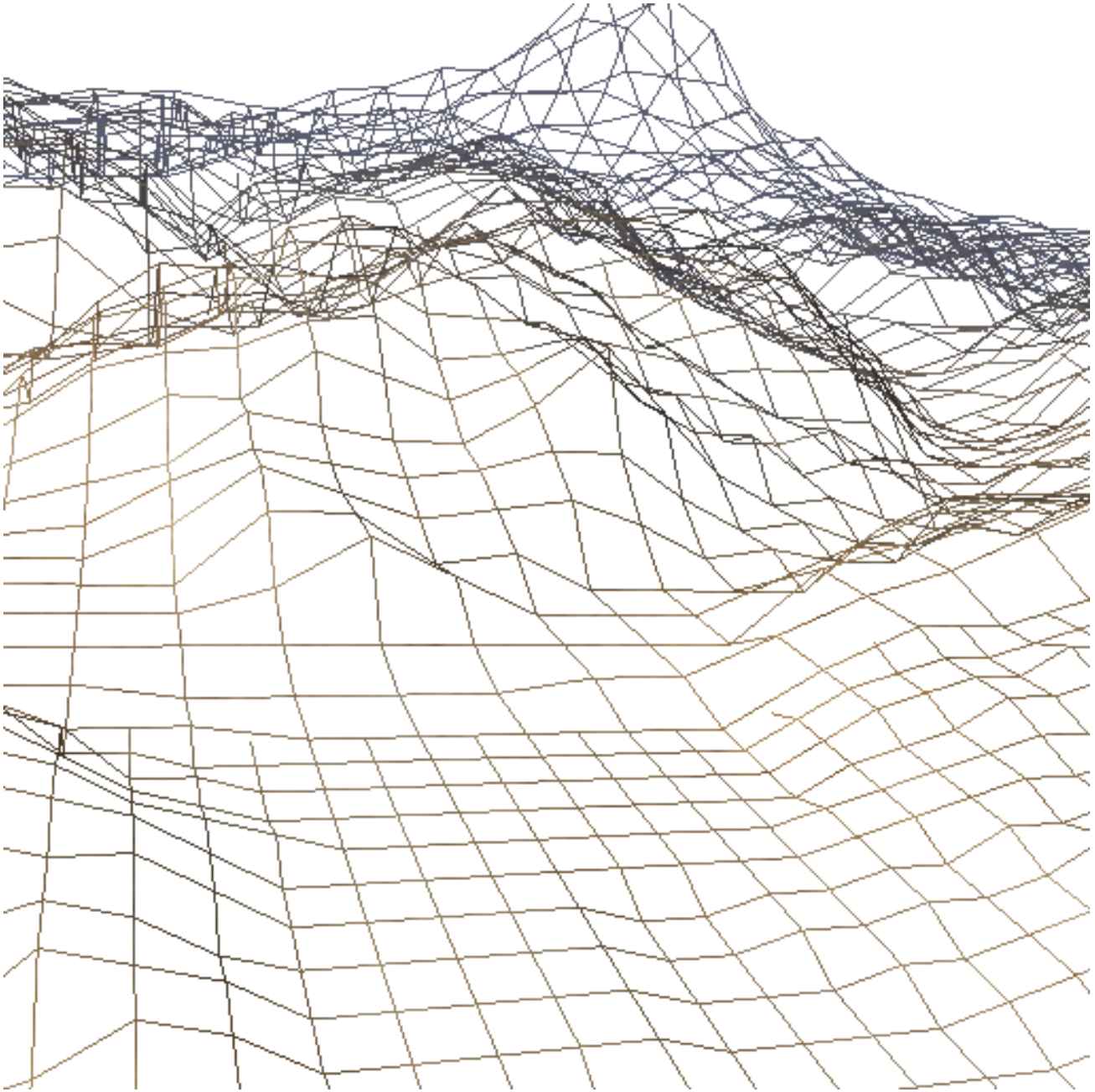
Terragen

- Less sophisticated generation than L3DT, for example
- But, in-tool editing is supported
- As is a 3D renderer
- http://www.planetside.co.uk/terragen/

ROAM

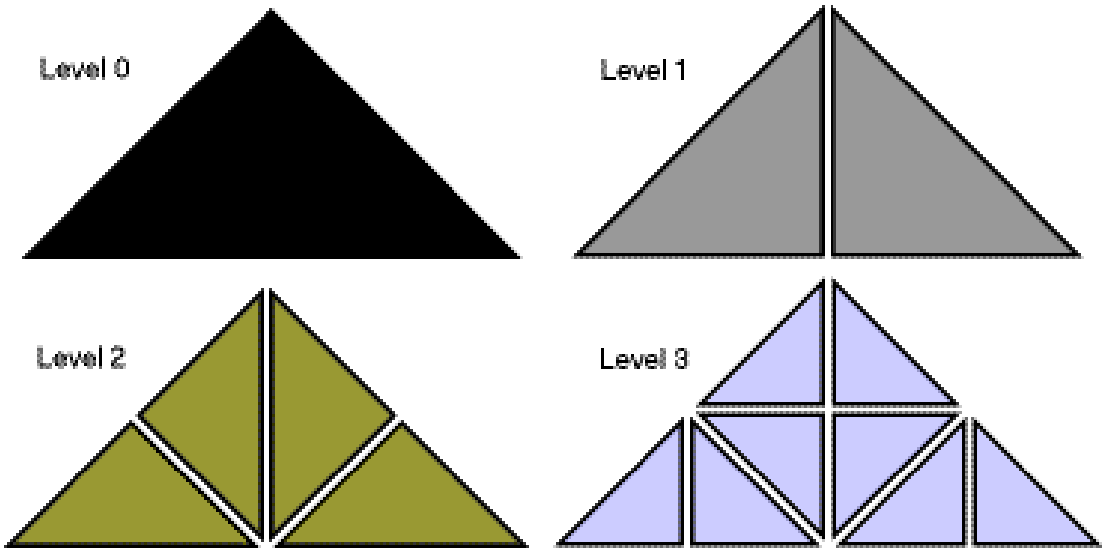
- *Real-time Optimally Adapting Meshes*
- This is a way of reducing the number of triangles that we send to the rendering pipeline.
- This algorithm belongs to a class of algorithms called *Level of Detail* (LOD) algorithms
- LOD algorithms in a nutshell: render small, detailed triangles CLOSE to the camera. Render large, rough triangles far away from the camera.
- All triangles – once transformed to projection space - should have about the same area

LOD



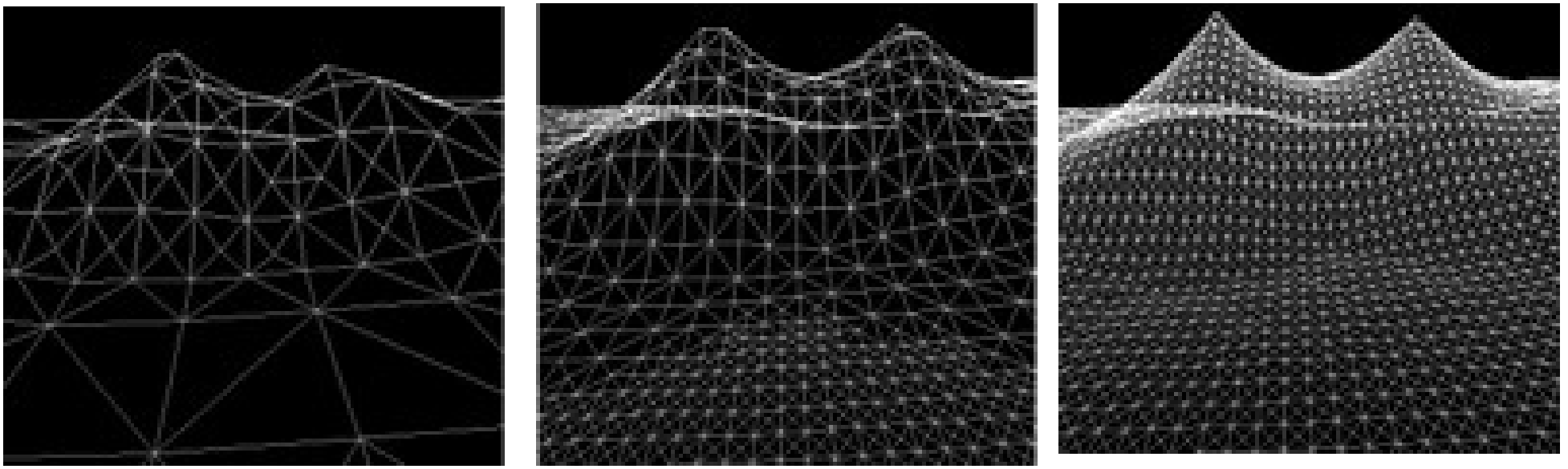
ROAM

- ROAM recursively partitions the landscape into pairs of isosceles right triangles.



- The closer the camera is, the deeper we render.

ROAM



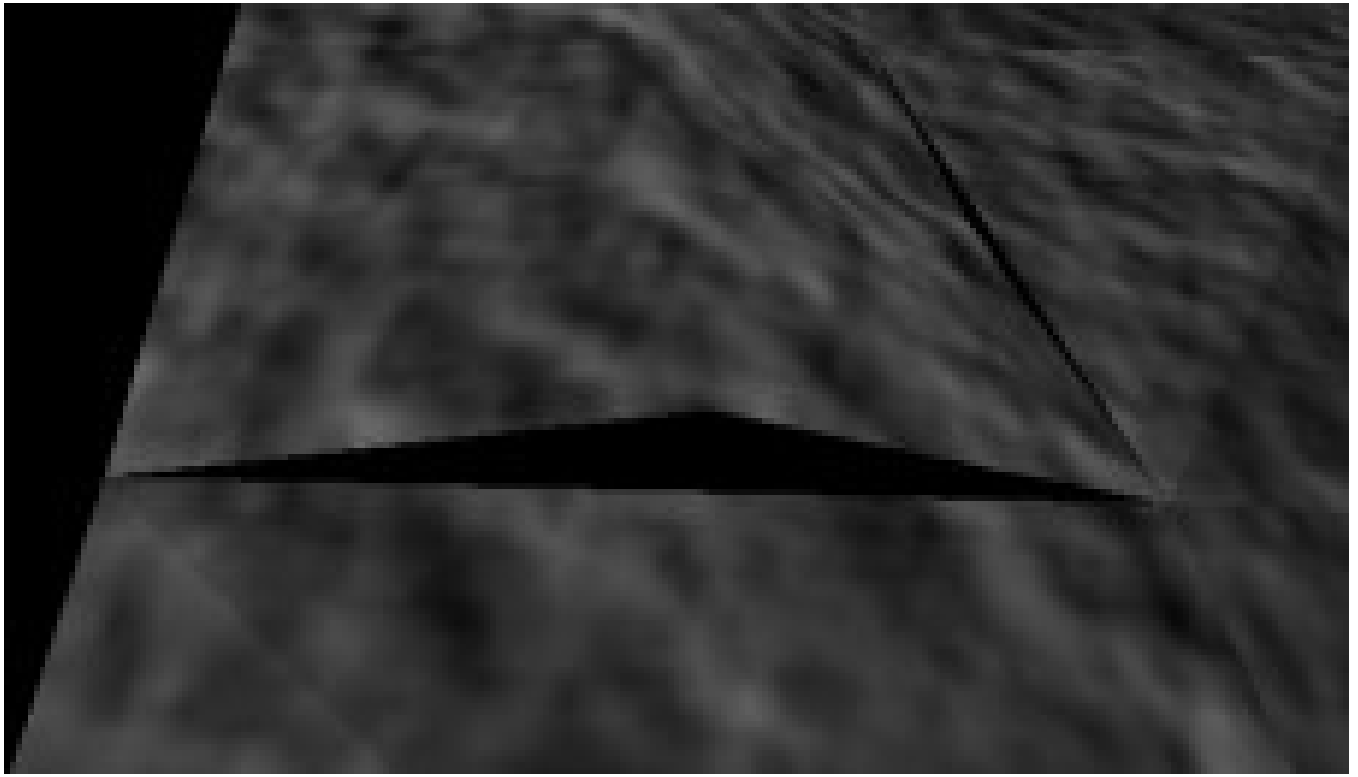
ROAM sample: http://www.gamasutra.com/features/20000403/turner_01.htm

ROAM

- Notice in the sample: we're sending more triangles to the pipeline than we apparently need; e.g. triangles way outside of the viewing volume. Why?

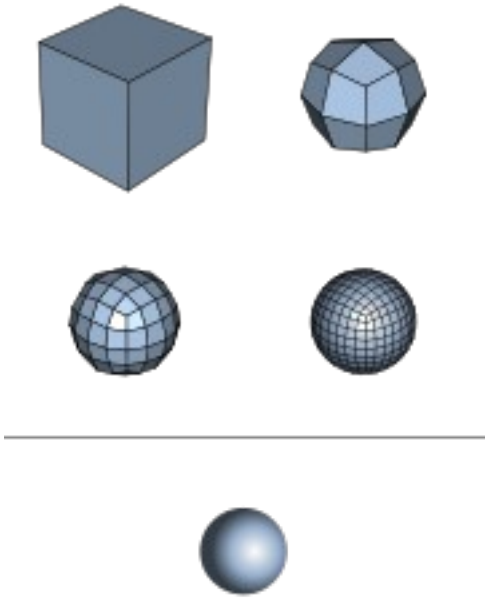
ROAM

- Notice in the sample: we're sending more triangles to the pipeline than we apparently need; e.g. triangles way outside of the viewing volume. Why?
- To avoid *cracking*.



Catmull-Clark Subdivision Algorithm

- Developed by E. Catmull (of Pixar) and J. Clark
- Non-destructive, geometry recursive subdivision and smoothing algorithm
- Iterate until it looks “good enough”
- http://symbolcraft.com/graphics/subdivision/



Catmull-Clark Subdivision Algorithm

- Let the *original vertices* denote the input vertices for our algorithm (or iteration)
- First, calculate the center of each face. These are called the *face points*.
- Create *new edge points* for each edge. Each new edge point position is the average of three vertices: the position of the midpoint of the original edge, and the position of the face point for the two adjacent faces.
- Connect face points to edge points with new edges
- The positions of the *original vertices* are then recalculated as the weighted average of the original vertices, the midpoints of the edges that shared the old vertex, and the face points surrounding the old vertex.

Catmull-Clark Subdivision Algorithm