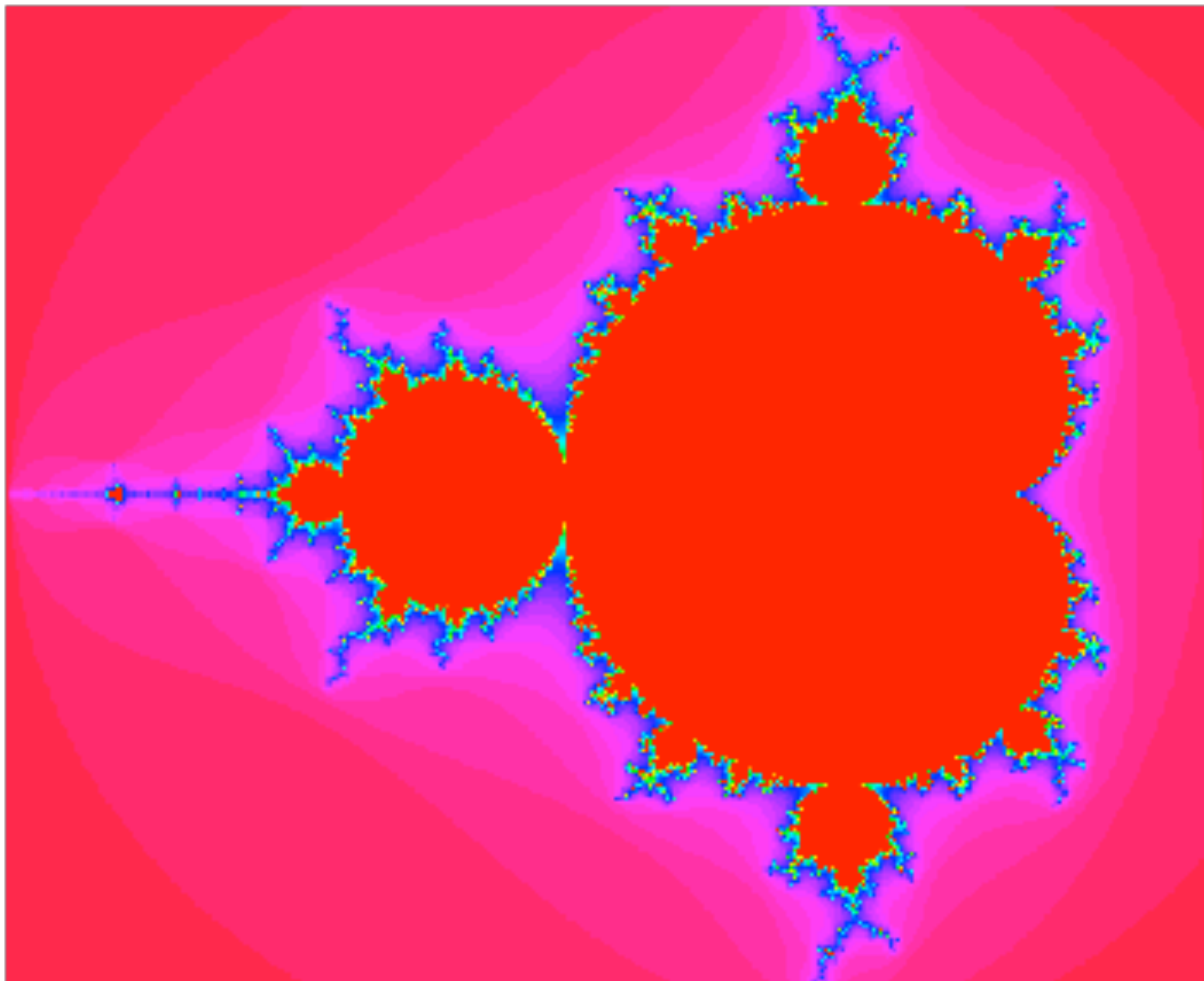


Lecture 5: Landscapes, Noise, Fractals, Agents

Procedural Content Generation, Autumn 2013

Noor Shaker, Julian Togelius and Mark Nelson





Maps and Landscapes: where and why?

- In a strategy game, to move units on
- In a flight simulator, to fly over
- In most first person games, to move in
- In any game, as a backdrop

Features of landscapes

- Heightmap
- Passable/impassable areas
- Points of interest (e.g. items, base locations, spawn points)
- ...other features possible

Example: flight sim



Example: StarCraft



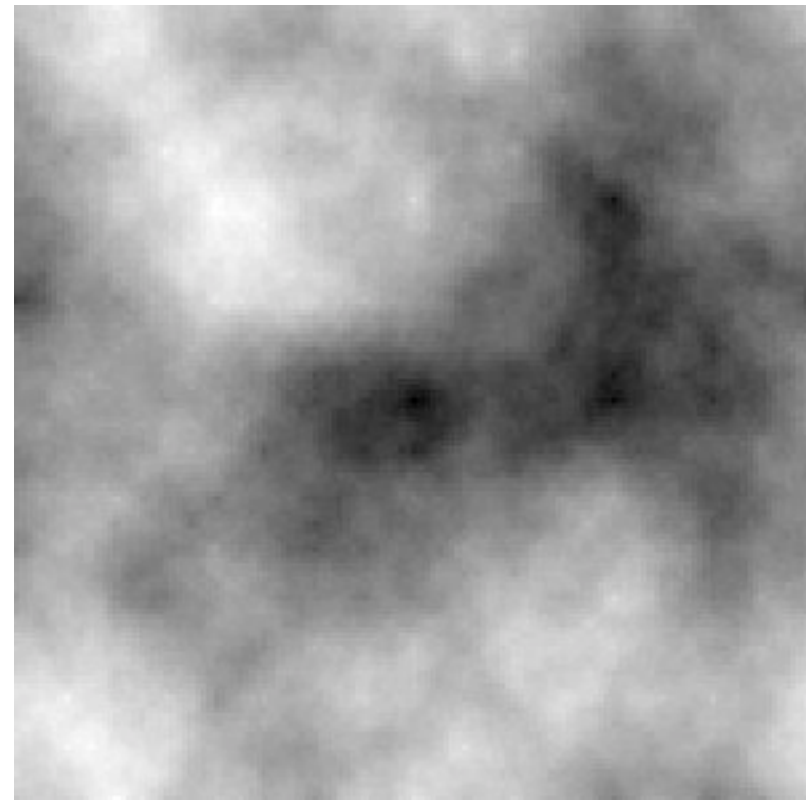
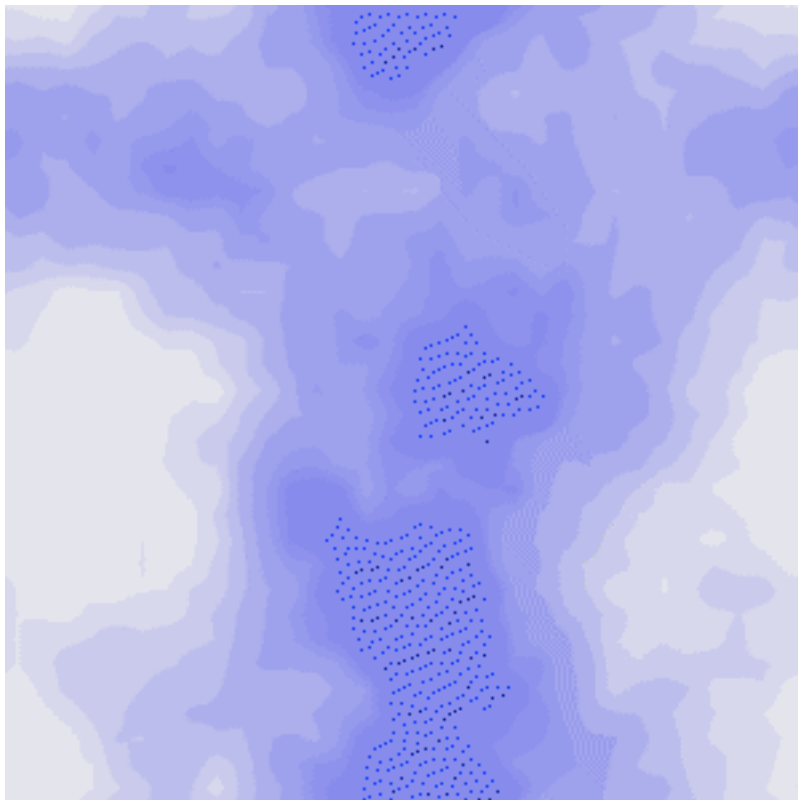
Example: Halo Reach



Heightmap representation

- Essentially a 2D grid with height values in cells

Cloudy skies, plasma...



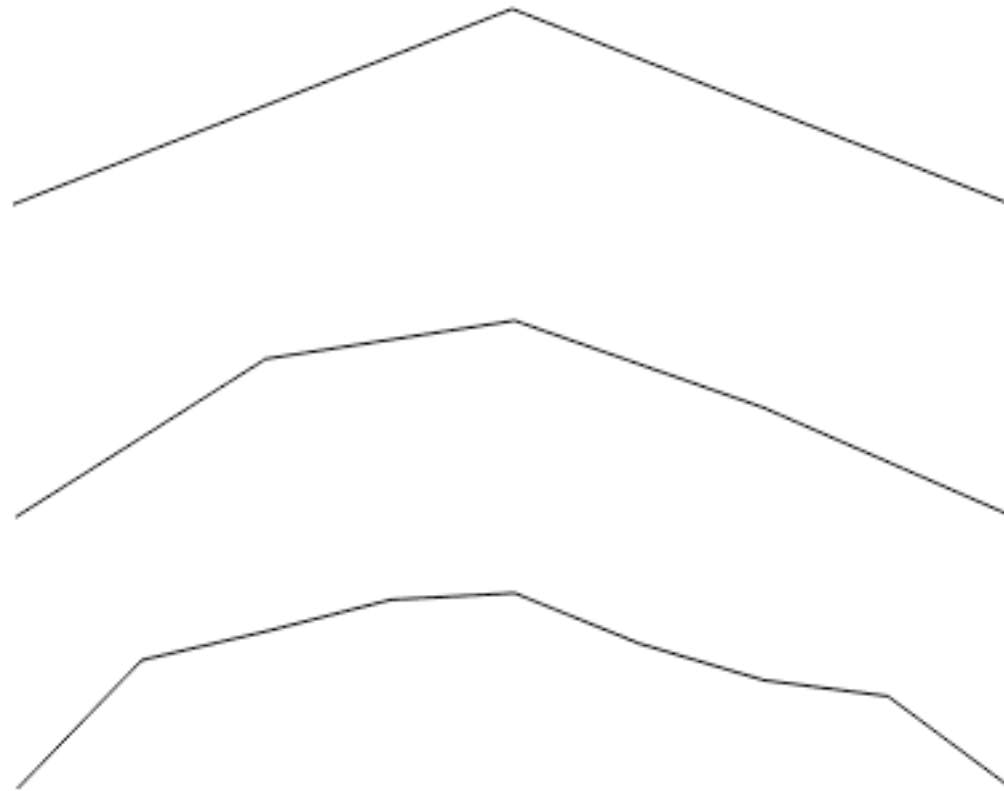
Fractal methods

- Based on self-similarity
- Did you know that all coastlines are of infinite length?

Midpoint displacement

- 2D algorithm - creates “silhouettes” through offsetting a line
- At each iteration, find the midpoint between two points on a line, and raise or lower this point a random amount
 - parameter H : factor by which random number range is multiplied each iteration
- This results in two new lines; do the same for these lines (using range $* H$), etc.

Midpoint displacement



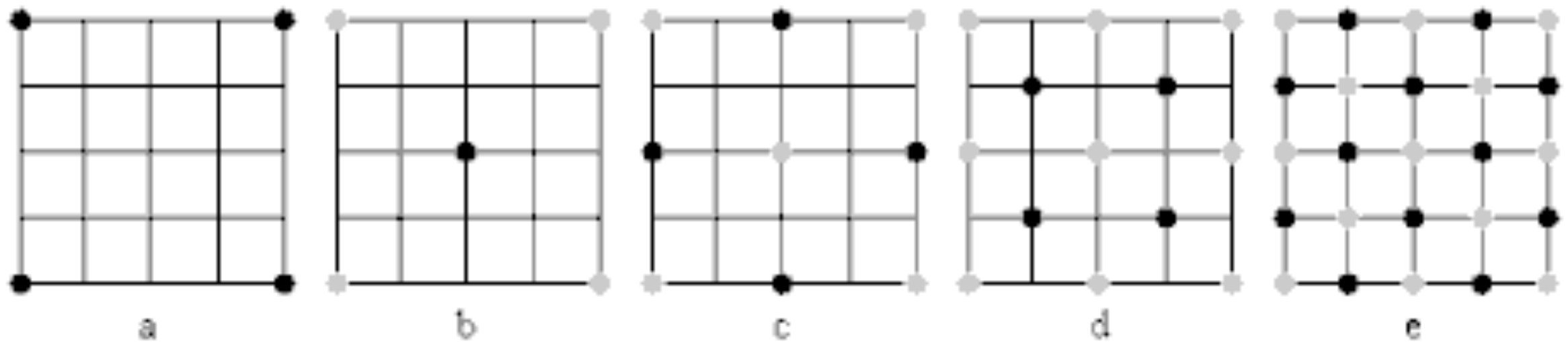
Midpoint displacement

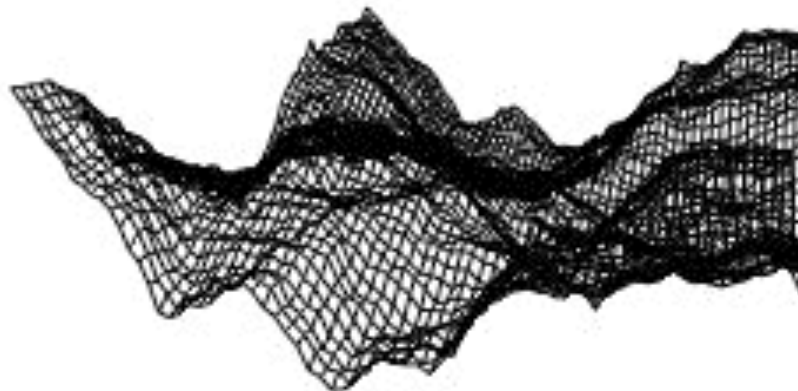
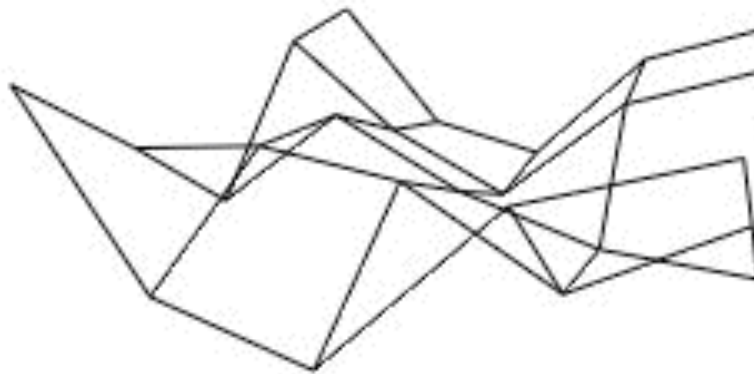
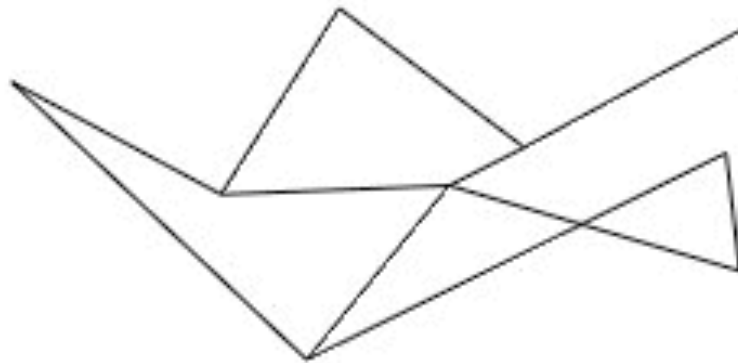


Diamond-Square

- Extension of midpoint displacement into three dimensions
- Offsets the midpoints of squares rather than lines
- Two steps for each iteration:
 - Diamond step: squares horizontal
 - Square step: squares turned 45 degrees

Diamond-Square

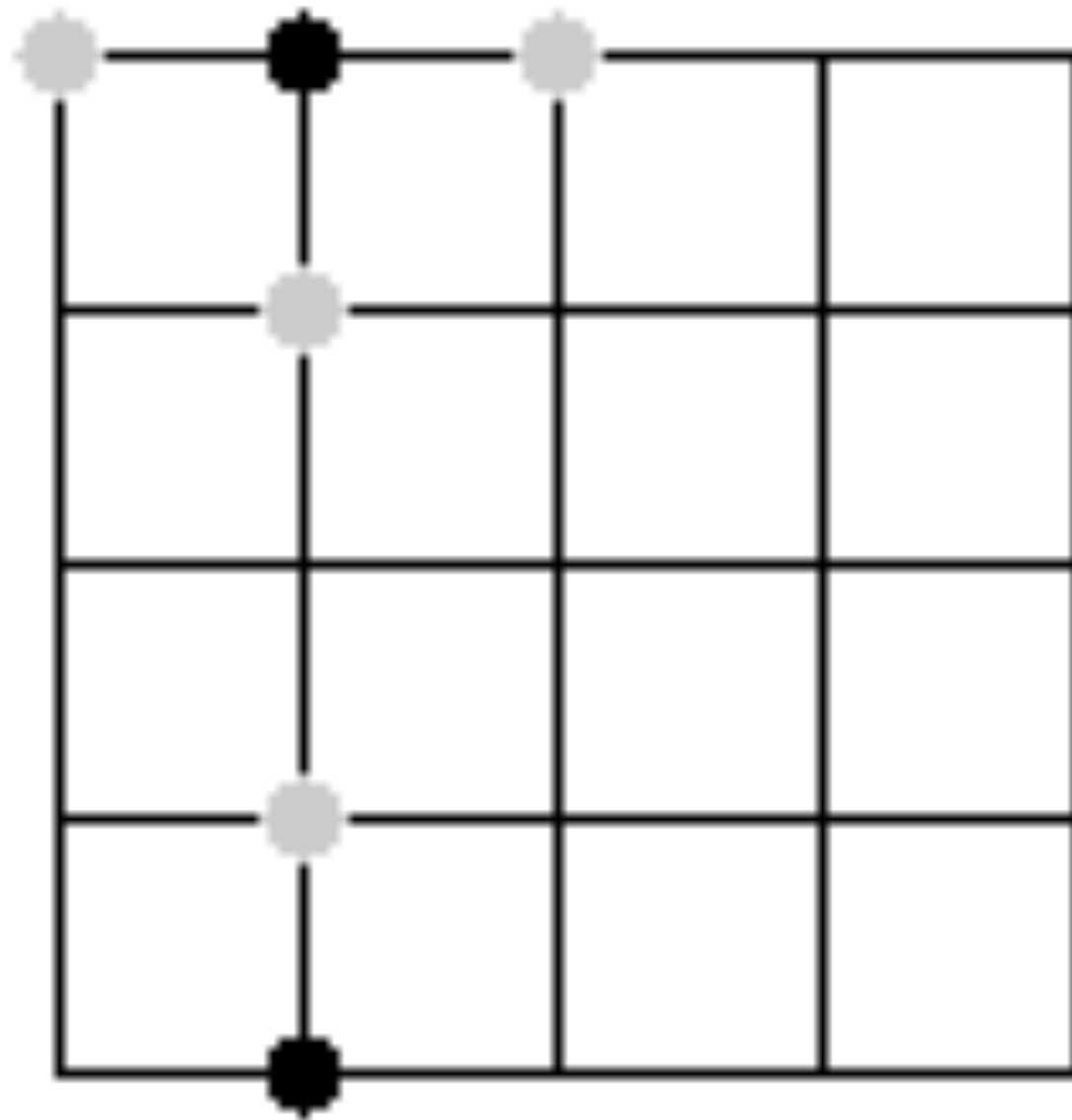




Recursive implementation

```
diamondsquare (square) {  
    midpoint = diamond (square);  
    for all 4 edges {  
        edge midpoint = square (edge);  
    }  
    randrange *= H;  
    for all 4 resulting squares {  
        diamondsquare (new square);  
    }  
}
```

Wraparound?



Perlin Noise

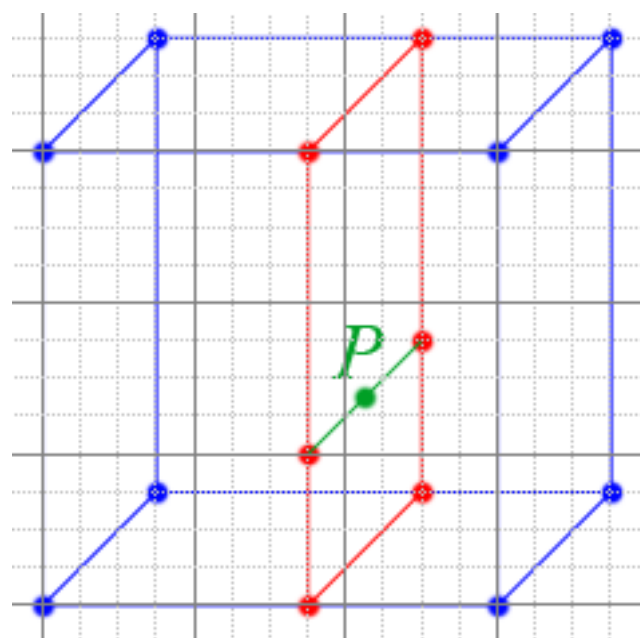
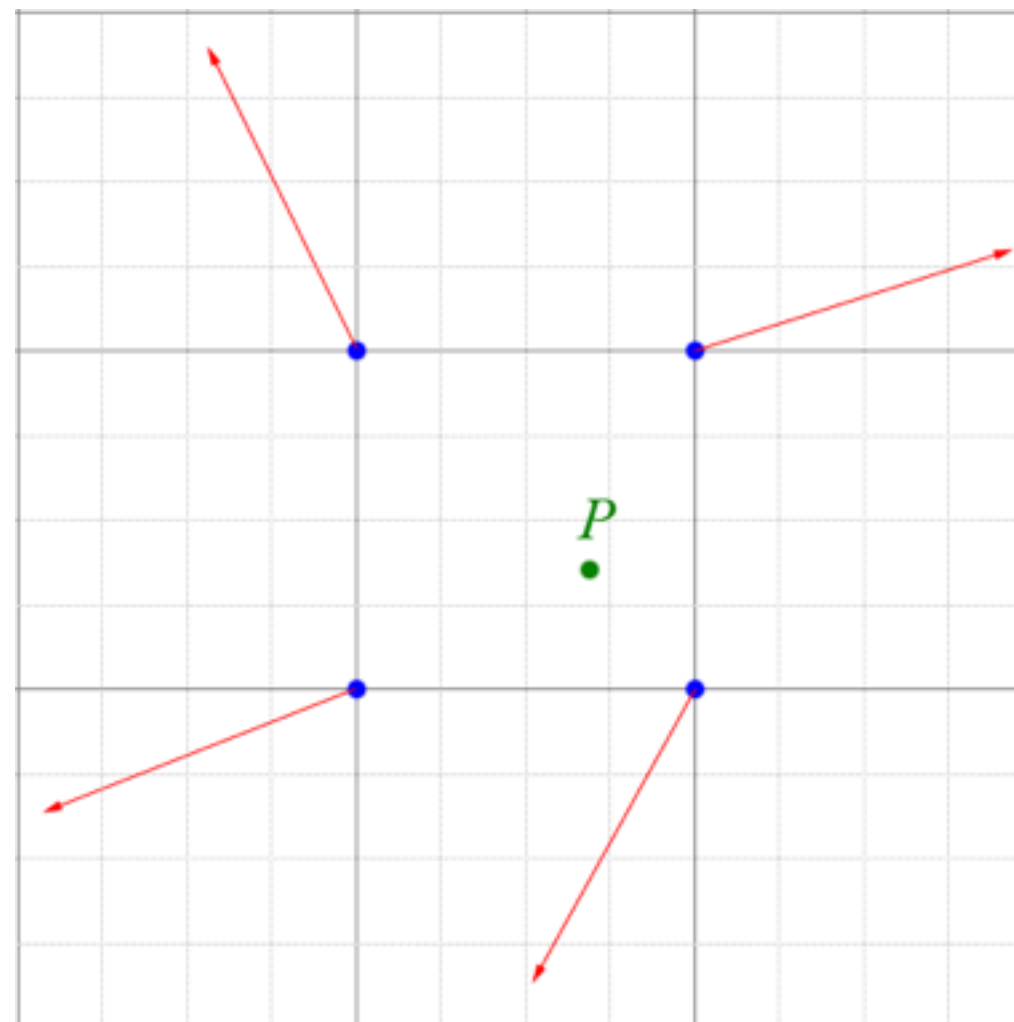
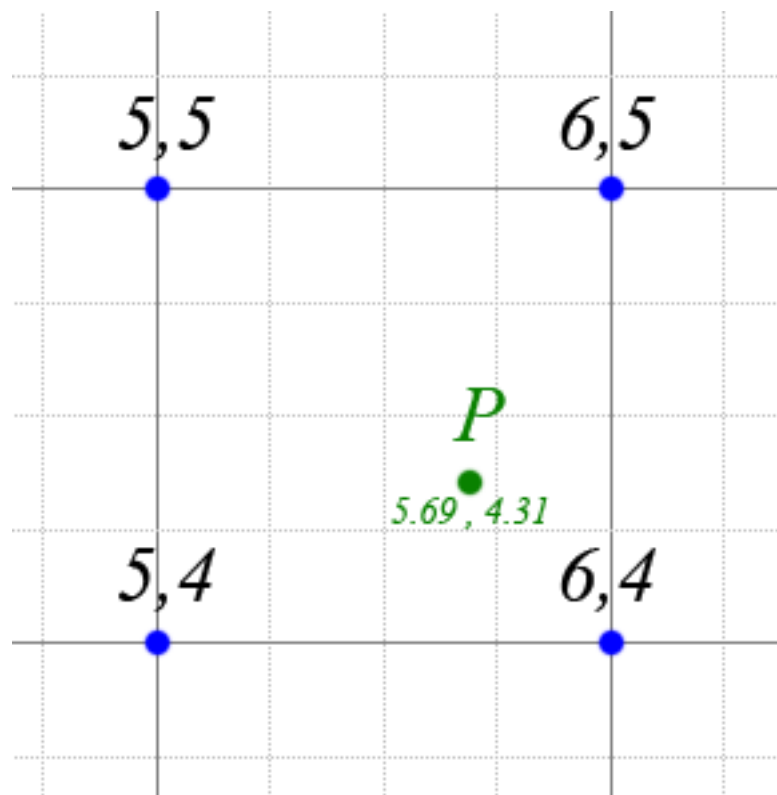
- A function for generating noise in n dimensions
 - Scale-free
- Very widely used in all computer graphics, including in games and movies
- Combinations of Noise() with itself and with other operators can give interesting results - see paper
- Ken Perlin received an Oscar for it!

Perlin Noise: basic idea

- Assign random *gradient vectors* to cells in a matrix (using standard random number generators)
- These gradients determine the height
- For values “between the cells” (non-integer coordinates): interpolate
- Repeat at higher resolutions (“octaves”) and lower amplitudes
 - Just like diamond-square

Perlin Noise in 2D

- Create a 2D grid with a randomly chosen 2D vector associated with each cell
- e.g., $[0, 0]$ is associated with the vector $[0.4, 0.3]$ and $[0, 1]$ is associated with $[0.1, 0.7]$ etc.
- Smoothly interpolate (using e.g. cubic interpolation) a surface that corresponds to the vectors



A taxonomy of PCG

- Online/Offline
- Necessary/Optional
- Random seeds/Parameter vectors
- Stochastic/Deterministic
- Constructive/Generate-and-test

In general,

PCG > randomness

Agent-based methods

- Use a number of “artificial agents” that construct the landscape by acting on it
- Agents of different types do different jobs
- Could be more controllable than diamond-square
- Could give rise to different types of landscapes

Controlled Procedural Terrain Generation Using Software Agents

Jonathon Doran and Ian Parberry

Published in IEEE TCIAIG, 2010

D&P's five agent types

- Coastline agents
- Smoothing agents
- Beach agents
- Mountain agents
- River agents

Rules for agents

- Each agent has a set number of “tokens” to spend on actions
- Each agent is allowed to see the current elevation around it, and allowed to modify it
- Agents don't interact *directly*

In the beginning...

...there was a vast ocean.

Then came the first coastline agent.

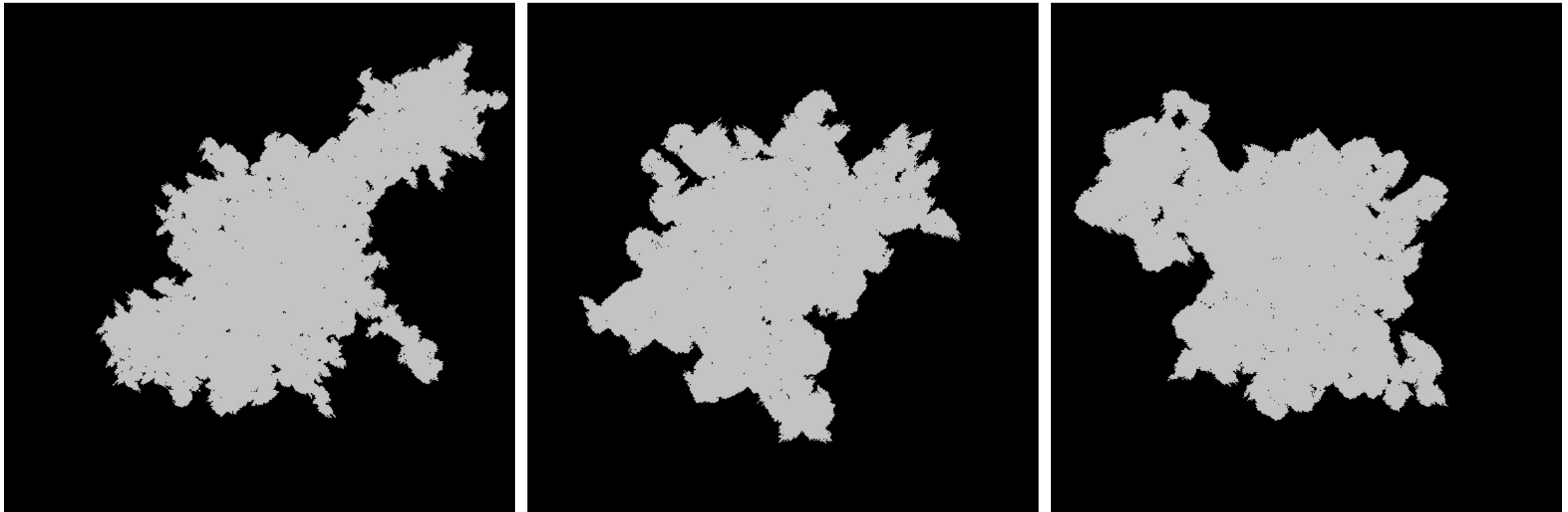
Coastline agents

- Multiply until they cover the whole coast
- Move out to position themselves right at the border of land and sea
- Generate a repulsor and an attractor point
- Score all neighbouring points according to distance to repulsor and attractor points
- Move to the best-scoring points, adding land as they go along

COASTLINE-GENERATE(*agent*)

```
1  if tokens(agent) ≥ limit
2      then
3          create 2 child agents
4          for each child
5              do
6                  child ← a random seed point on parent's border
7                  child ← 1/2 of the parent's tokens
8                  child ← a random direction
9                  COASTLINE-GENERATE(child)
10     else
11         for each token
12             do
13                 point ← random border point
14                 for each point p adjacent to point
15                     do
16                         score p
17                 fill in the point with the highest score
```

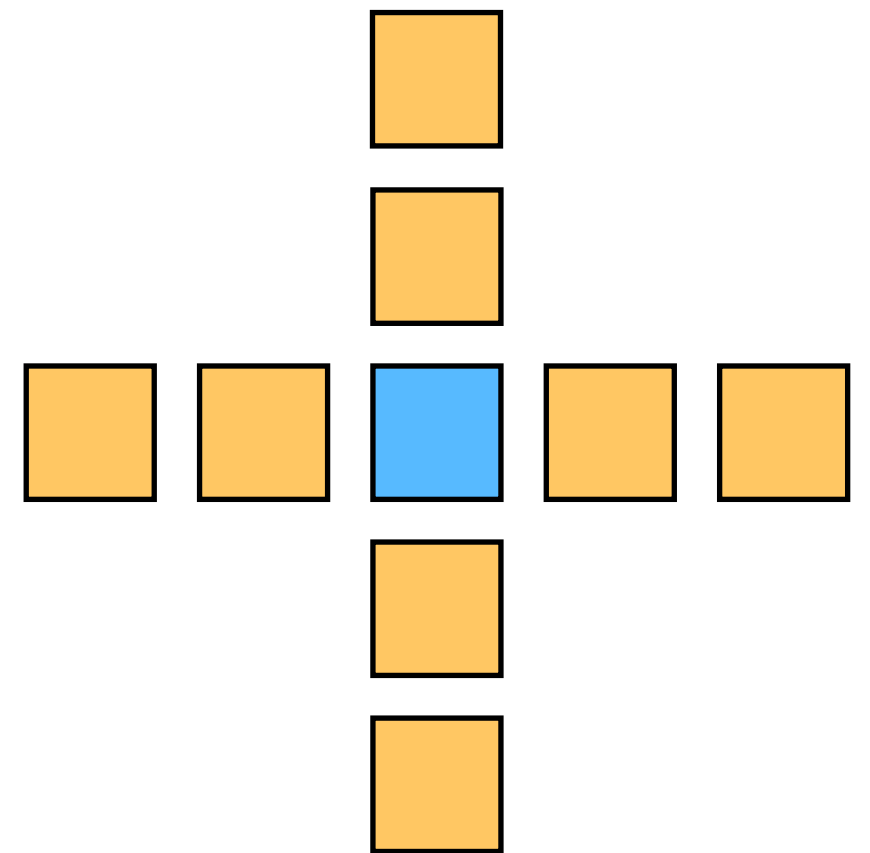
Coastline agents



Varying action sizes

Smoothing agents

- Take random walks on the map
- Change the elevation of each visited point to (almost) the mean of its extended von Neumann neighbourhood



Smoothing agents

SMOOTH(*starting-point*)

1 *location* \leftarrow *starting-point*

2 **for each** *token*

3 **do**

4 $height_{location} \leftarrow$ weighted average of neighborhood

5 *location* \leftarrow random neighboring point

Beach agents

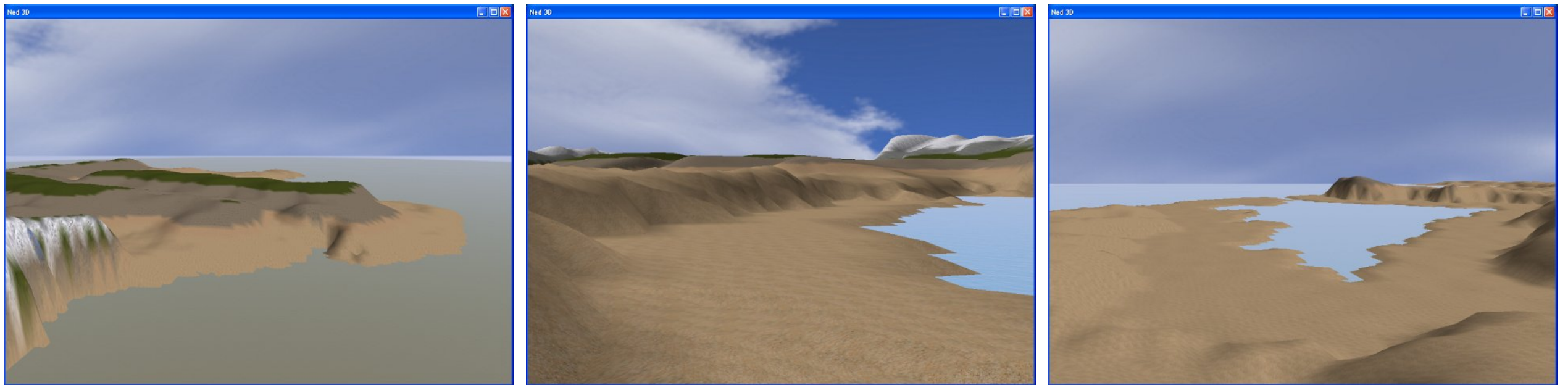
- Select random position along the coast, where coast is not too steep
- Flatten an area around this point (leaving small variations)
- Move randomly a short direction away from the coast, flattening the area

Beach agents

BEACH-GENERATE(*starting-point*)

```
1  location  $\leftarrow$  starting-point
2  for each token
3      do
4          if  $height_{location} \geq limit$ 
5              then
6                  location  $\leftarrow$  random shoreline point
7                  flatten area around location
8                  smooth area around location
9                  inland  $\leftarrow$  random point a short distance inland from location
10                 for  $i \leftarrow 0$  to  $size(walk)$ 
11                     do
12                         flatten area around inland
13                         smooth area around inland
14                         inland  $\leftarrow$  random neighboring point
15                 location  $\leftarrow$  random neighboring point of location
```

Beach agents



Varying beach width

Mountain agents

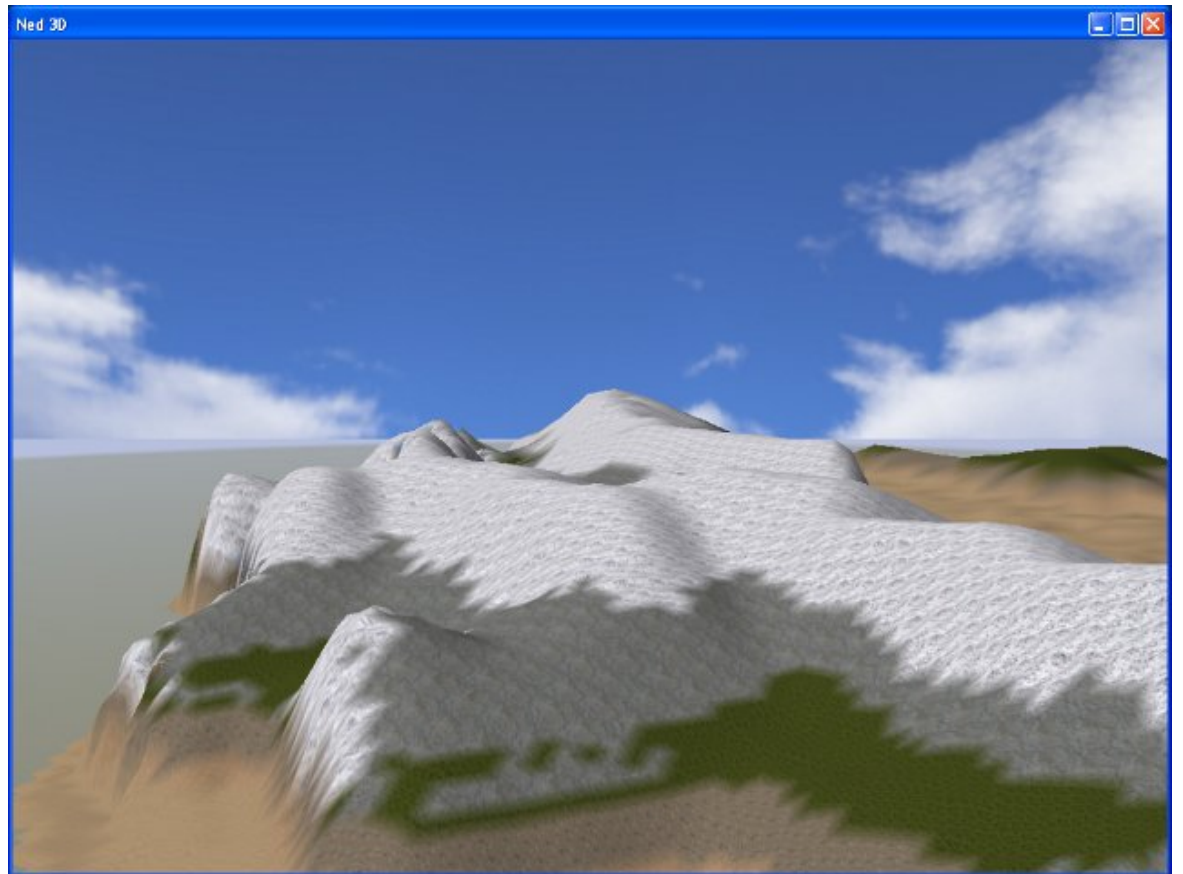
- Start at random positions and directions
- Move forward, continuously elevating a wedge, creating a ridge
- Turn randomly within 45 degrees from the initial course
- Periodically offshoot “foothills” perpendicular to movement direction

Mountain agents

MOUNTAIN-GENERATE(*starting_point*)

```
1  location ← starting-point
2  direction ← random direction
3  for each token
4      do
5          elevate wedge perpendicular to direction
6          smooth area around location
7          location ← next point in direction
8          every n-th token
9              do
10                 direction ← original-direction ± 45-degrees
```

Mountain agents



Narrow versus wide features

River agents

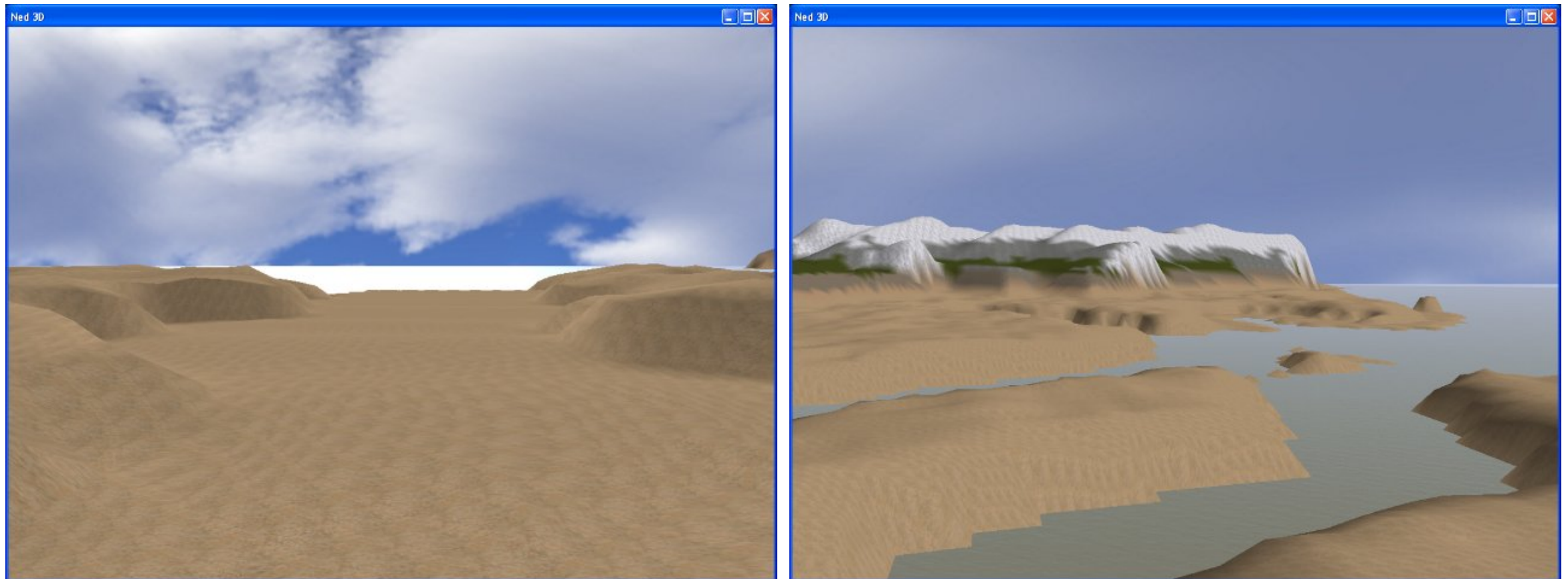
- Move from a random point on the coast towards a random point on a mountain ridge
- “Wiggle” along the path
- Stop when reaching too high altitudes
- Retrace the path down to the ocean, deepening a wedge along the path

River agents

RIVER-GENERATE()

```
1  coast ← random point on coastline
2  mountain ← random point at base of a mountain
3  point ← coast
4  while point not at mountain
5      do
6          add point to path
7          point ← next point closer to mountain
8  while point not at coast
9      do
10         flatten wedge perpendicular to downhill direction
11         smooth area around point
12         point ← next point in path
```

River agents



A dry river, and the outflow of three rivers

Further questions

- Parameters... what parameters?
- What features of landscapes do we want to be able to specify?
- How can the human and the algorithm interact productively?

Search-based methods

- Use an evolutionary algorithm, or other stochastic optimization algorithm
- Evaluate candidate pieces of content using a fitness function

Issues in search-based PCG

- Content representation and search space
 - Direct or indirect?
- Fitness function
 - Direct, simulation-based, interactive?

Multiobjective Exploration of the StarCraft Map Space

Julian Togelius, Mike Preuss,
Nicola Beume, Simon Wessing,
Johan Hagelbäck and Georgios N. Yannakakis

Our approach:

- Direct/indirect map representations
- An ensemble of fitness functions
- Multiobjective evolution

StarCraft

- Classic real-time strategy game
- Korea's unofficial national sport
- Two or three player competitive matches
- Three distinct races





StarCraft map features

Traditional (constructive) map generation

- Place features on maps according to some heuristic
 - e.g. fractals, growing islands, cellular automata
- Hard or impossible to optimize for gameplay properties
- Restrictions on possible content necessary in order to ensure valid maps

Our approach

- Define desirable traits of RTS maps
- Operationalize these traits as fitness functions
- Define a search space for maps
- Search for maps that satisfy the fitness functions as well as possible, using multiobjective evolution
 - (visualize trade-offs as Pareto fronts)

Desirable traits of an RTS map

- Playability
- Fairness
- Skill differentiation
- Interestingness

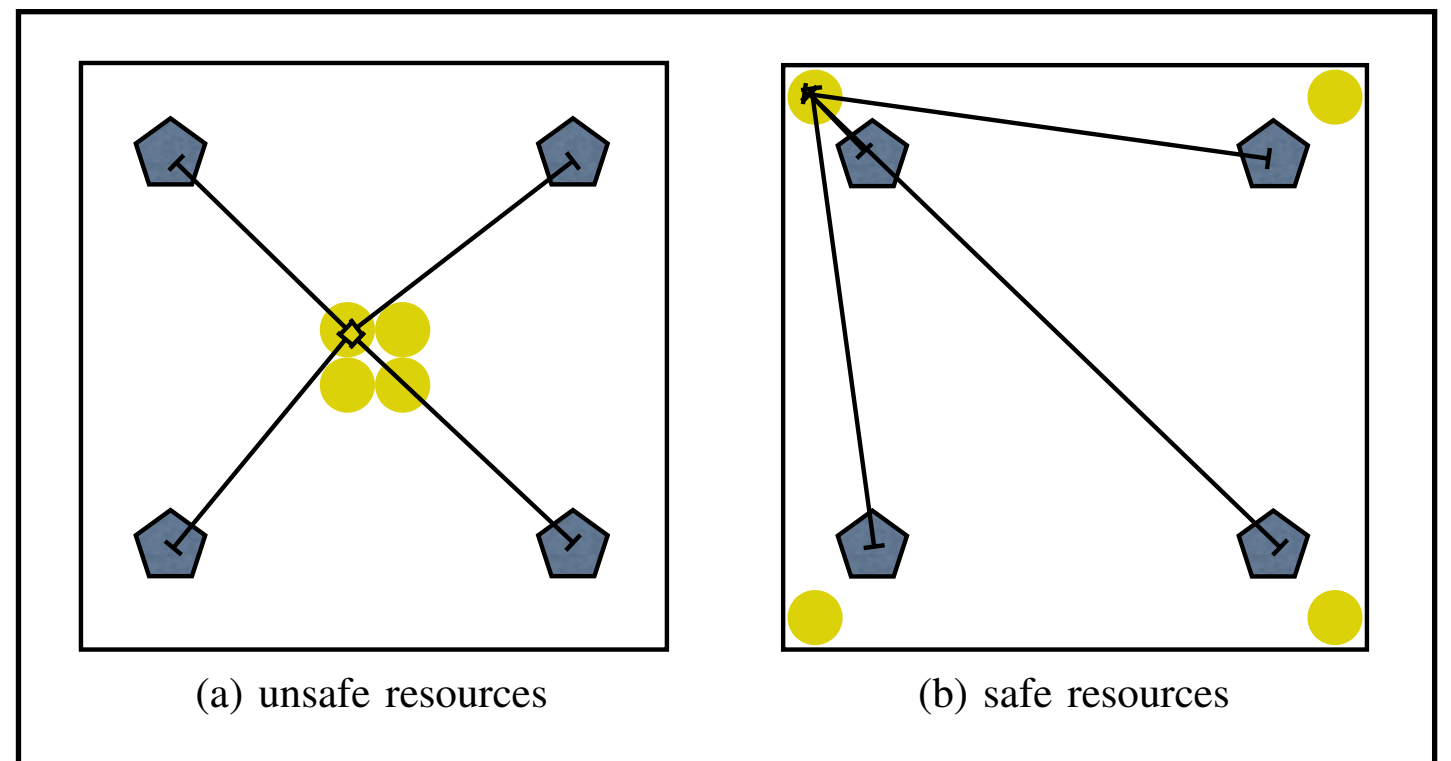
Playability fitness functions

- Base space: minimum amount of space around bases
- Base distance: minimum distance between bases (via A^*)

Fairness

fitness functions

- Distance from base to closest resource
- Resource ownership
- Resource safety
- Resource fairness



Skill differentiation fitness functions

(also contribute to interestingness)

- Choke points
(narrowest width of shortest path)
- Path overlapping

Dual map representation

- Indirect representation: a vector of real numbers in $\{0..1\}$
- Direct representation: a 64x64 grid corresponding to a StarCraft map, including impassable areas, bases, resource sites
- Genotype to phenotype mapping: before fitness calculation

Genotype to phenotype

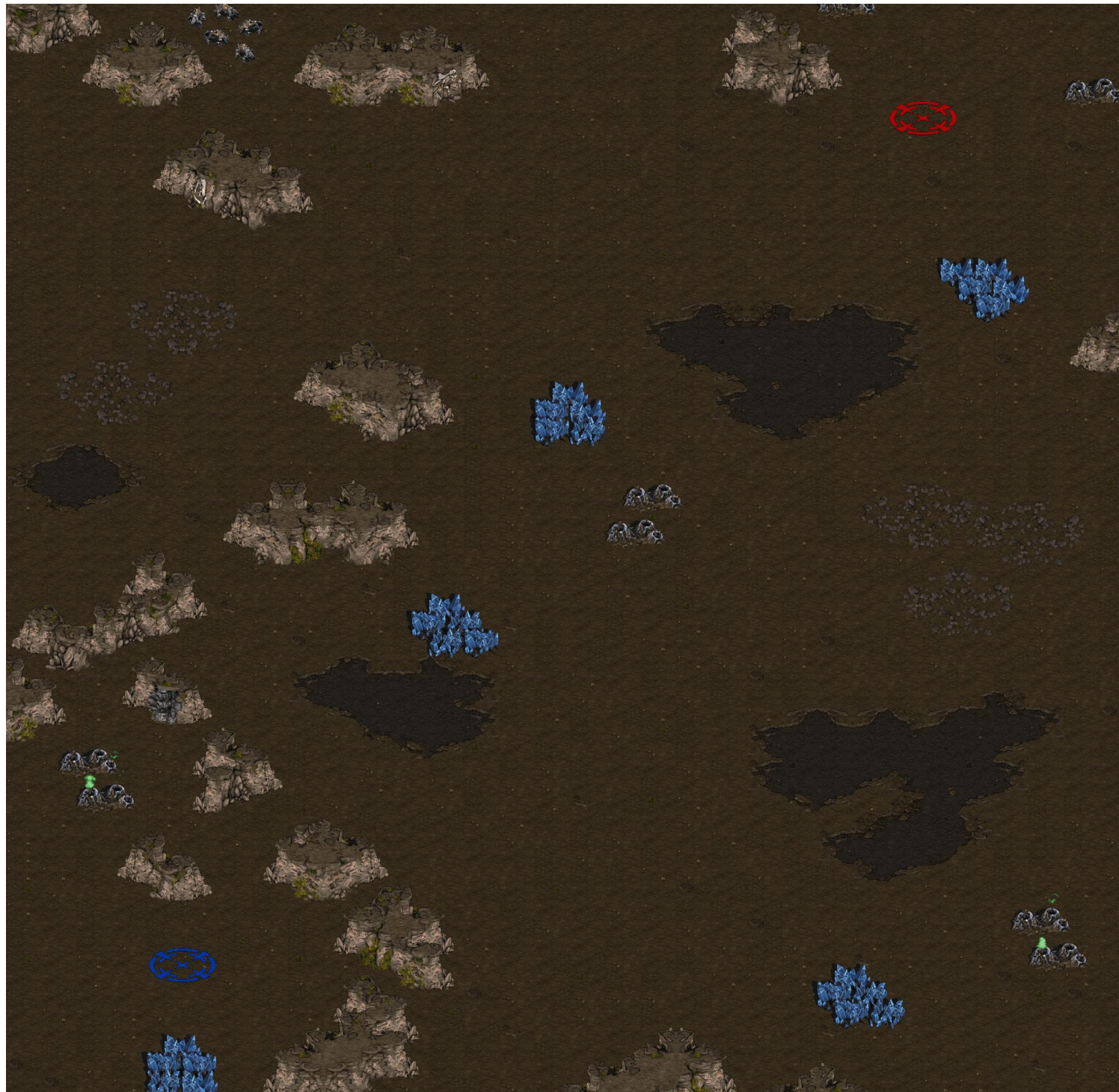
- Two or three bases, five mineral sources and five gas wells: (ϕ, θ) coordinates
- Rock formations represented indirectly using “turtle graphics”. Each formation has:
 - (x, y) starting position
 - probability of turning left/right
 - probability of gaps (“lifting the pen”)

Experiments

- Used the SMS-EMOA
 - fast hypercube-based descendant of NSGA-II
- Tried optimizing each *pair* of objectives
 - All at once not computationally feasible!
- Population 20, 50000 evaluations

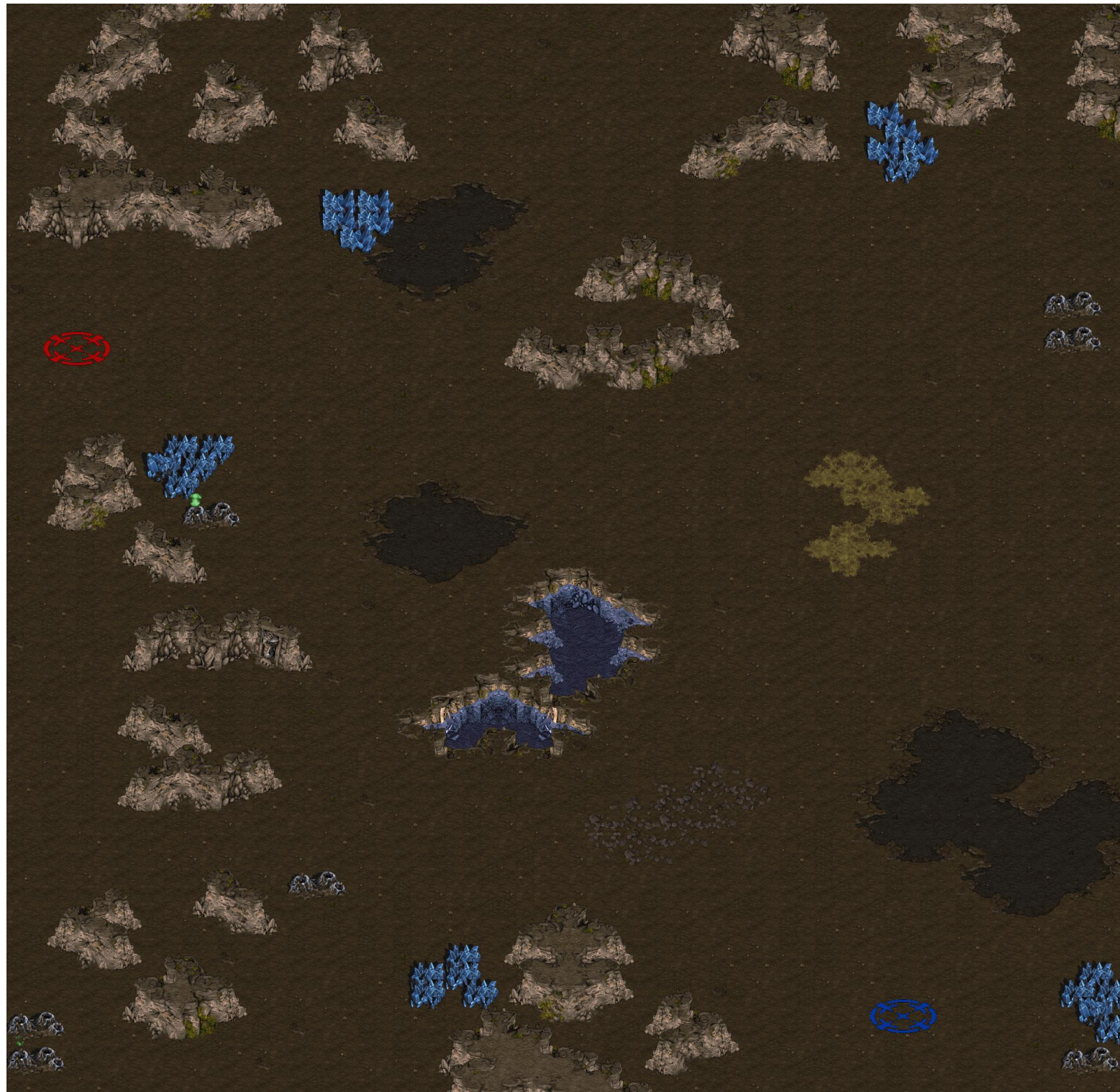
Last piece of the puzzle

- From phenotype to StarCraft maps via SCPM
- Further manual editing possible but not necessary
- (shown maps slightly aesthetically enhanced)



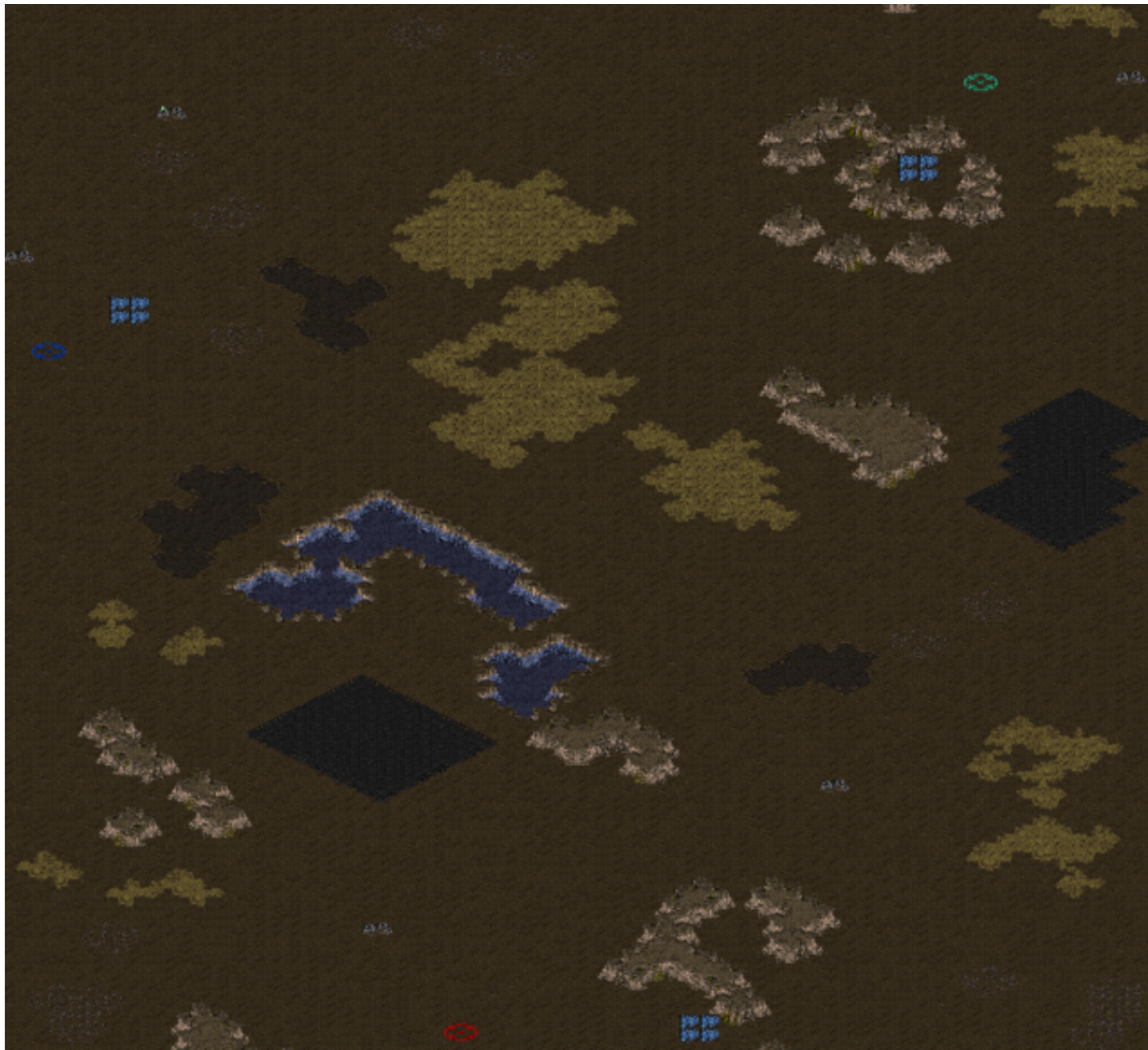
Evolved map

Resource fairness vs. choke points

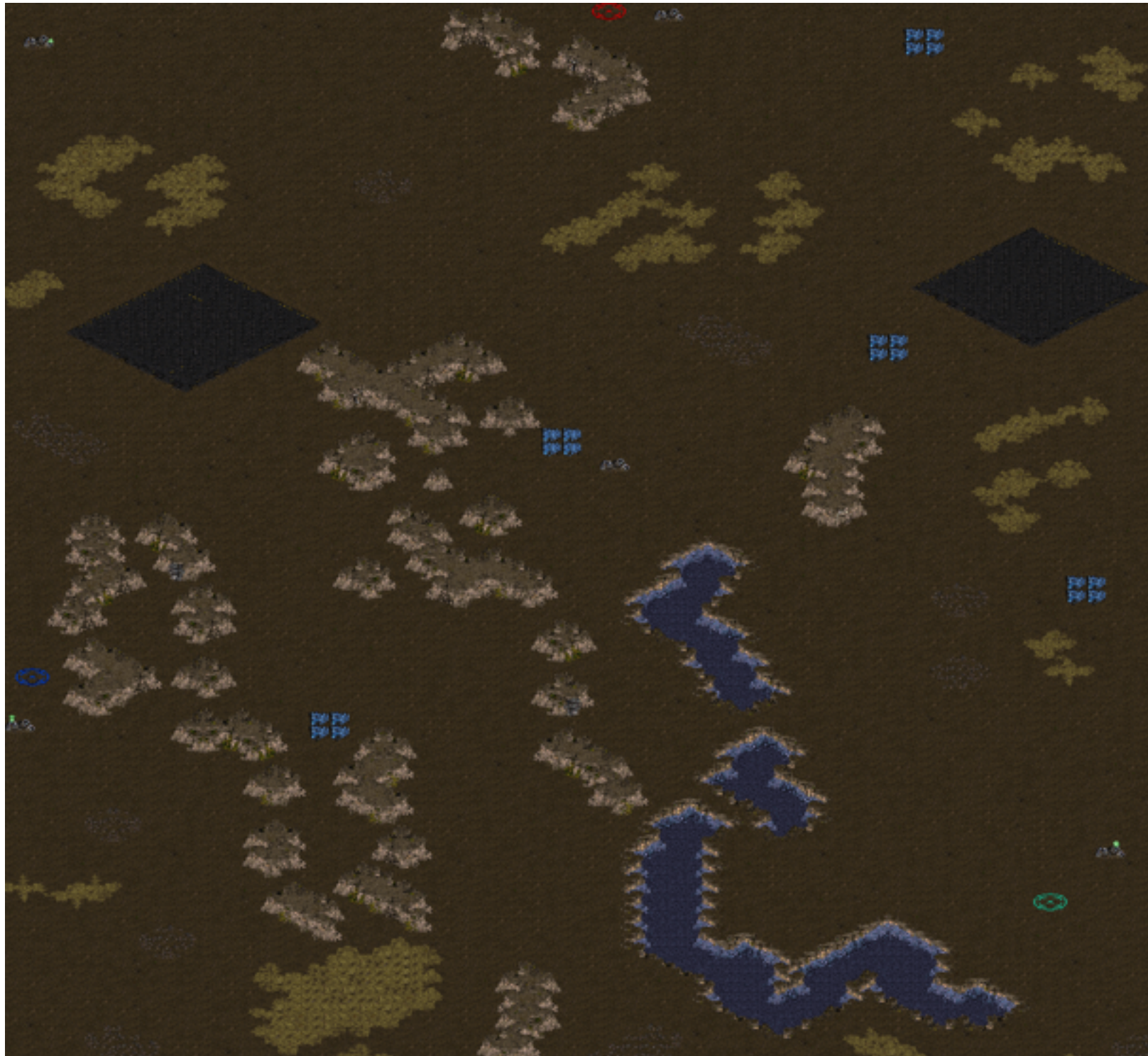


Another evolved map

Resource fairness vs. choke points



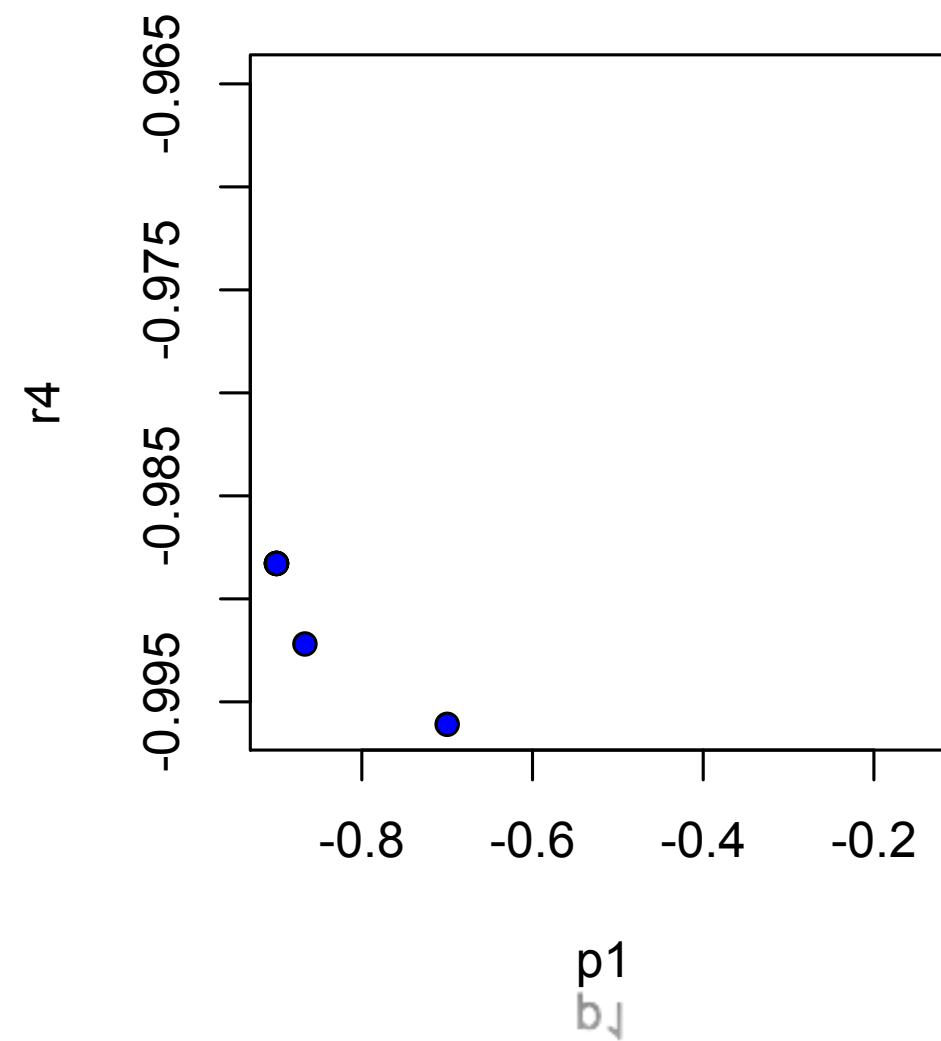
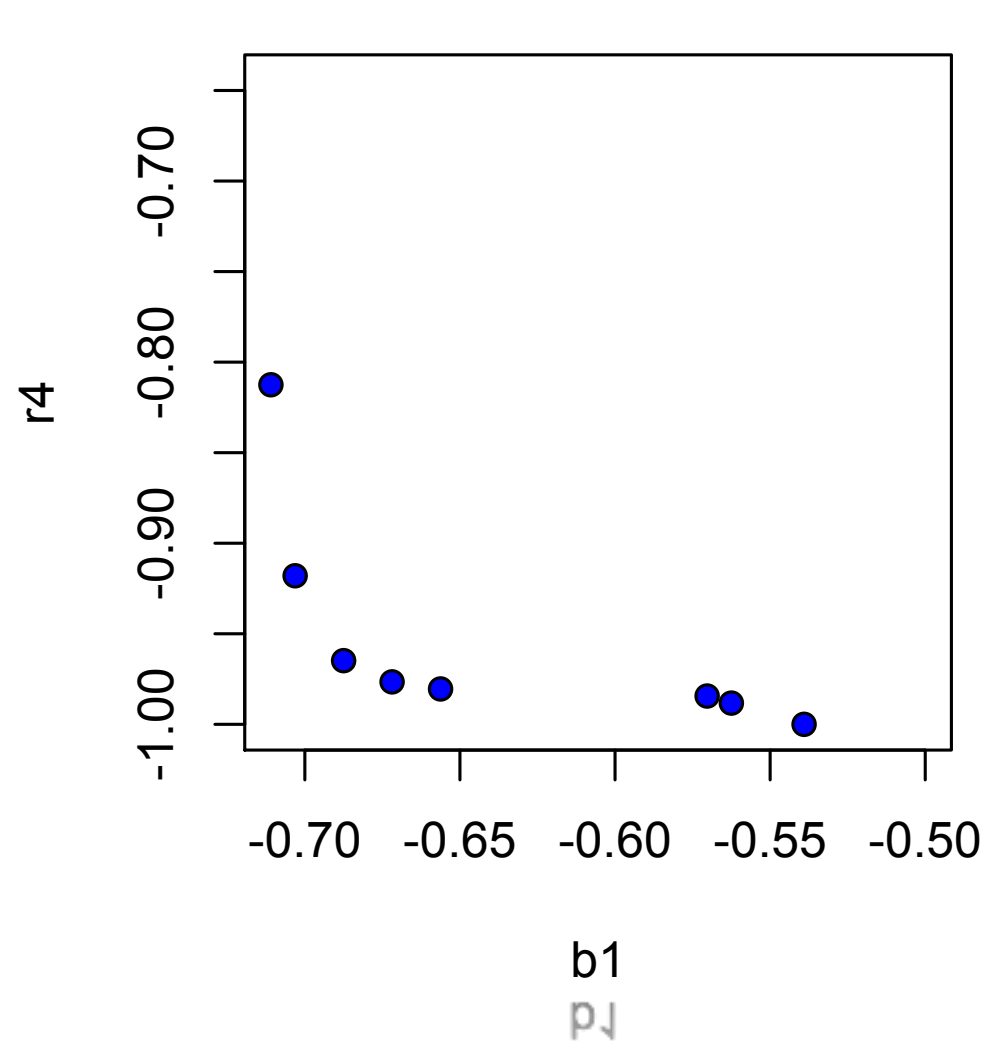
Three-player map



Another three-player map

Observations on the fitness functions

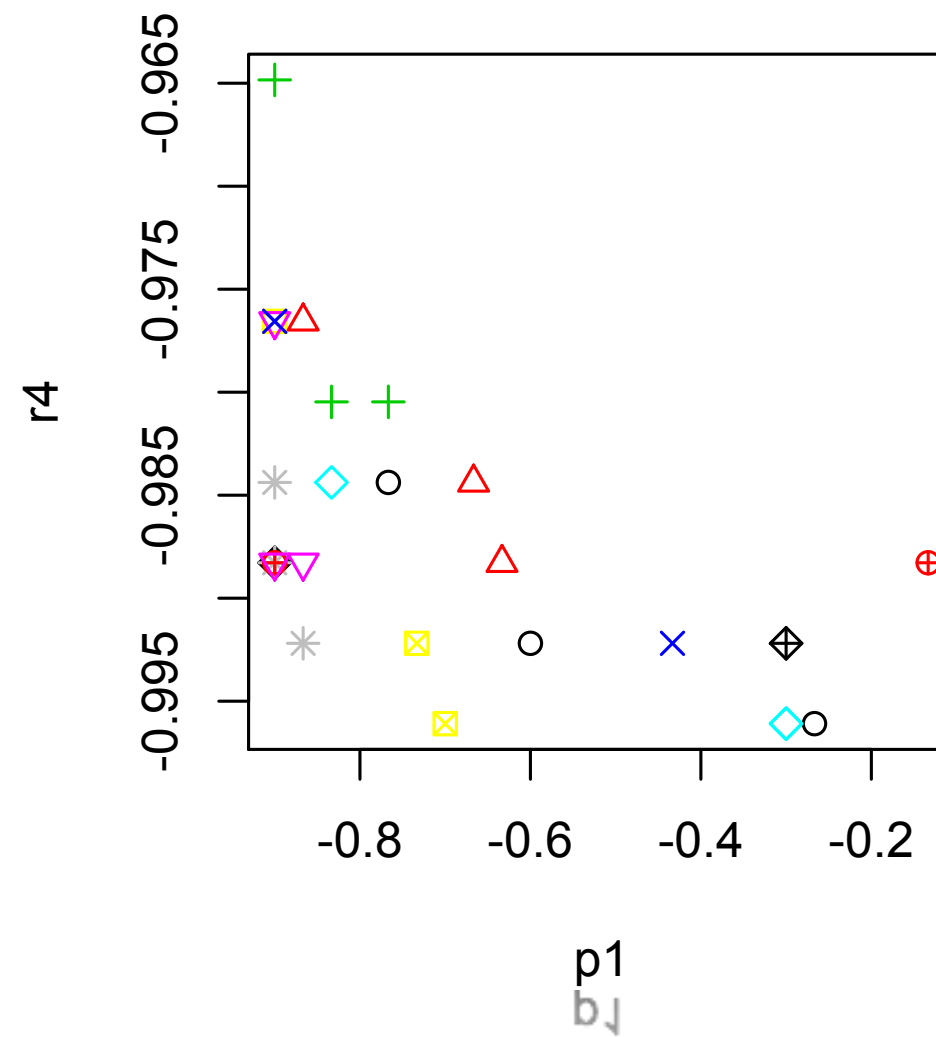
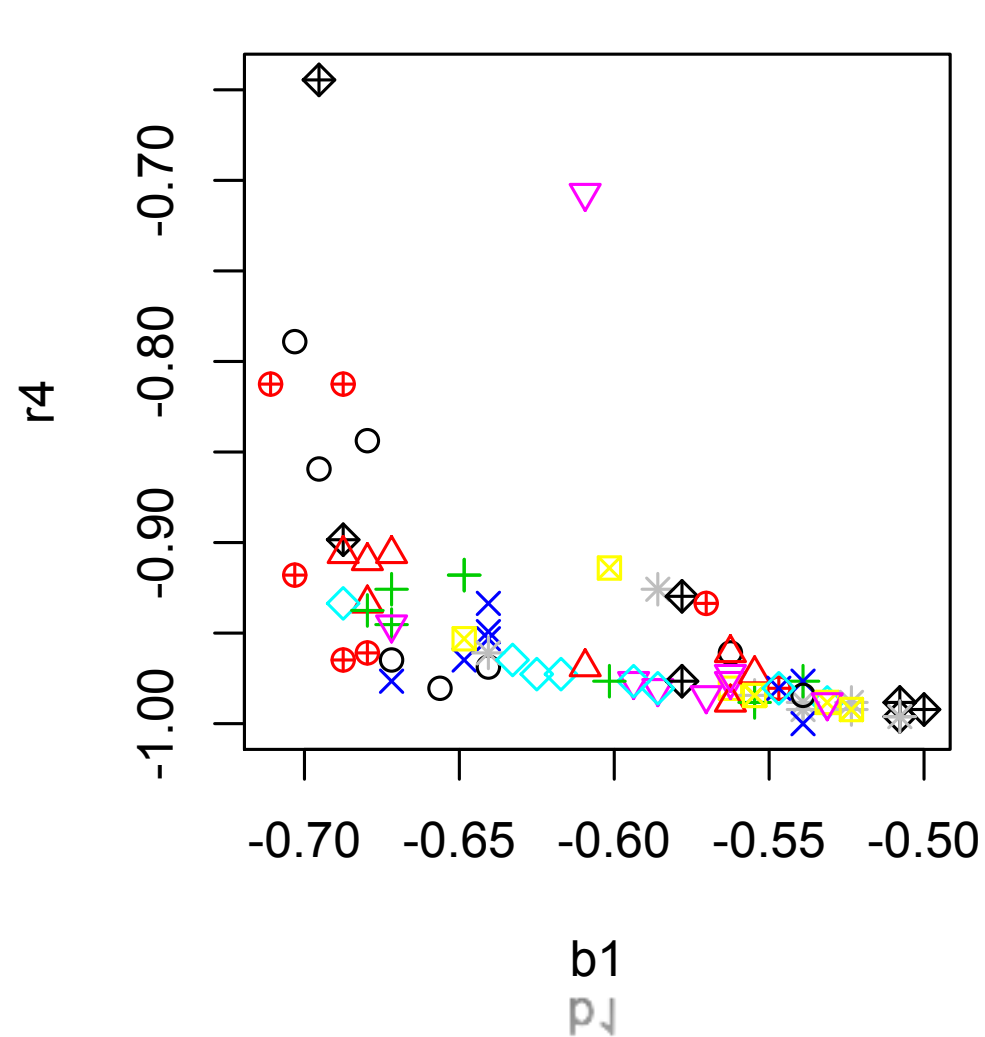
- Some objectives are trivial to optimise, e.g. base space was made into a constraint
- Some are easy and do not conflict with other objectives, e.g. resource ownership
- Some are easy but partially conflict with others, e.g. base distance
- Some are not so easy and highly conflicting, e.g. choke points



Single pareto fronts

Left: resource fairness vs. base distance

Right: resource fairness vs. choke points



Composite pareto fronts

Left: resource fairness vs. base distance

Right: resource fairness vs. choke points

What did we find?

- Our map representation works:
We can reliably evolve good-looking, playable maps
- Some of our fitness functions are good
- Some are not
- What to do with the Pareto fronts?

What to do with the Pareto fronts?

- Human designers might use Pareto fronts to understand the tradeoffs between different desirable properties
- Use evolved maps as starting points for further human design
- Or algorithmically select maps in automatic content generation

A taxonomy of PCG

- Online/Offline
- Necessary/Optional
- Random seeds/Parameter vectors
- Stochastic/Deterministic
- Constructive/Generate-and-test