Procedural Content Generation, continued

2018-04-05

N-3: PCG intro

1. PCG can be used to p____ or a____ game aspects
2. What are some reasons to use PCG?
3. What are some risks / concerns of PCG?
4. Design-time vs run-time PCG?
5. How does the use of a random seed in PCG effect development and gameplay?
6. What is flow theory? How does it relate to dynamic difficulty adjustment & drama management?
7. How do you know you are generating something interesting?

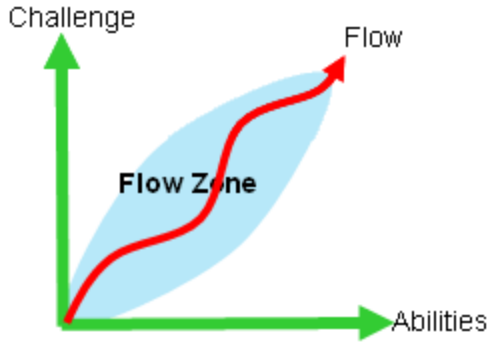


Figure 2 Player in-game Flow experience

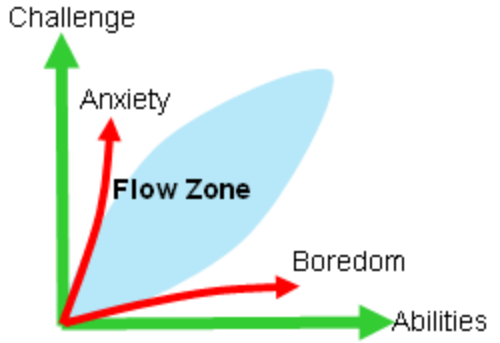


Figure 3 Player encounters psychic entropies

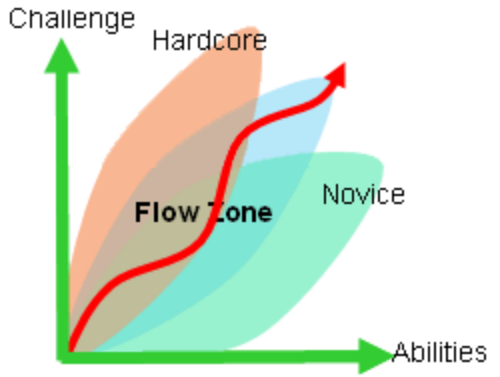


Figure 4 Different players and Flow Zones

http://jenovachen.com/flowingames/designfig.htm

N-2: PCG as Search

1. What “search” is happening? Do we seek a path to goal?
2. What is the state space? How many states do we save?
3. How memory efficient is this search?
4. Hill climbing: (PCG as parameter search part 1)
    1. L____ search
    2. What is the “landscape”?
    3. Need a function that maps p____ to f_____
5. GAs: (PCG as parameter search part 2)
    1. Good in _____ domains, where _D.K.__ is scarce or hard to encode
    2. Can also be used for ____ search
    3. Also needs a f_____ function (maps c____ to f_____)
6. Other local search techniques
    1. Gradient Descent, Simulated annealing, Local beam, Tabu, Ant Colony Optimization, …

N-1: Player models

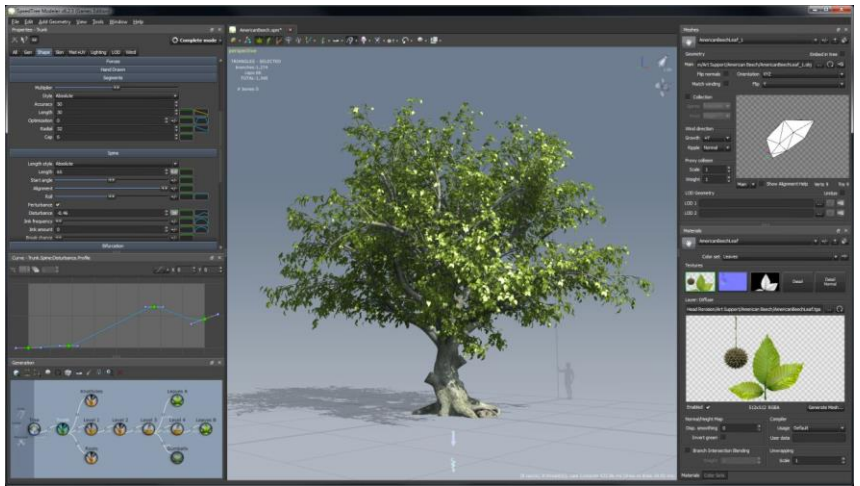
1. What is a player model? What does it allow?
2. What are two high-level categories of modeling?
3. What are a couple major types within the first category?
4. What are ways to get a player model?

Game Bits

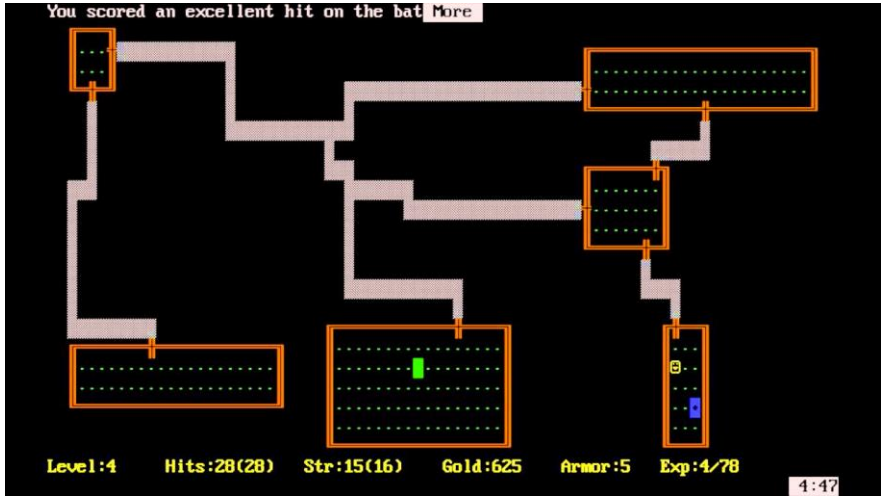
Borderlands series

No Man's Sky

SpeedTree



Game Space



Rogue (1980)

Bloodborne, Chalice Dungeons

Spelunky

Minecraft

Game Scenarios

Skyrim, radiant quests

Dwarf Fortress

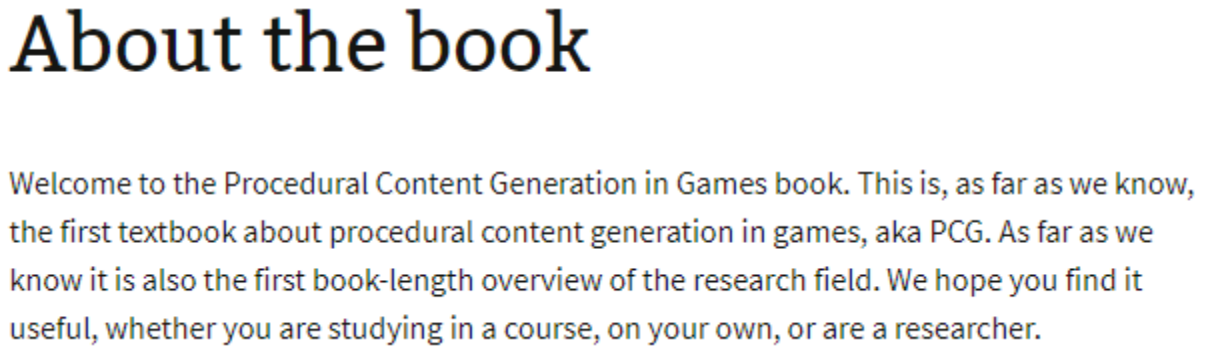
Diablo 2, randomizing situations

PCG in General

Indie/AAA games folks use as touchstones

- Rogue (1980)
- Dwarf Fortress (2006)
- Minecraft (2009)
- Spelunky (2012)
- No Man’s Sky (2016)

- Stolen terms (bits, space, scenarios): Procedural Content Generation for Games: A Survey
  - https://course.ccs.neu.edu/cs5150f13/readings/hendrikx_pcgg.pdf
- Search-Based Procedural Content Generation: A Taxonomy and Survey
  - https://course.ccs.neu.edu/cs5150f13/readings/togelius_sbpcg.pdf
- PCG in Games: A textbook and an overview of current research (2016)
  - http://pcgbook.com
- http://pcg.wikidot.com/



Content is king!

PROCEDURAL CONTENT GENERATION

PCG high-level Methods

- Search
- Rule systems
- Generative Grammars
- Constraint Solving

http://pcgbook.com/wp-content/uploads/chapter02.pdf

JUMPED IN TO SEARCH ALREADY…

Genetic Algorithms

- Loosely inspired by Darwin's Theory of Evolution
- Sometimes called "Evolutionary search"
- Consistently pretty popular (if only for the tagline "evolving artificial intelligence to solve…")

Genetic Algorithm Pseudocode

population = set of random points of size *X*
time = 0
while E(population)<*threshold* and time<max:
    time ++
    Mutate(population)
    population = **Crossover**(population)
    population = **Reduce**(population)
return bestE(population)

GA Pseudocode cont

- **Mutate**: Given some probability, randomly replace a member of the population with a neighbor
  - Make a random change
- **Crossover**: Take pairs of the initial population (chosen based on fitness), and combine their features randomly till population grows up to size *Y* (where *X*<*Y*)
- **Reduce**: Reduce the size of the population back down to *X*

Genetic Algorithms

Pros

- Middling authorial burden (more than hill climbing, less than generative grammars/rule system)
- High likelihood of finding global optima-ish

Cons

- Takes skill to pick and balance mutation/crossover
- Over-reliance

Search-based PCG

Pros

- Allows for designers to specify high-level desires in heuristics rather than low level content authoring

Cons

- Coming up with a good heuristic is hard
- Many devs will prefer to just use generative grammars

PCG: RULE SYSTEMS

PCG high-level Methods

- Search
- Rule systems
- Generative Grammars
- Constraint Solving

Rule Systems

- Similar to the rule systems discussed before
  - If (world has certain conditions)
  - Then (make change to world)

- Now focused on building game bits/spaces, with each rule focusing on one feature
  - Example: *if* y<groundheight *then* set voxel to ground
  - This is how Minecraft world generation works

Rule Matching

Query

```
{ who: nick, concept: onHit, curMap:circus, health: 0.66, nearAllies: 2, hitBy: zombieclown }
```

PASS Rule1: { who = nick, concept = onHit } → *"ouch!"*
FAIL Rule2: { who = nick, concept = onReload } → *"changing clips!"*
FAIL Rule3: { who = nick, concept = onHit, health < 0.3 } → *"aaargh I'm dying!"*
PASS Rule4: { who = nick, concept = onHit, nearAllies > 1 } → *"ow help!"*
PASS Rule5: { who = nick, concept = onHit, curMap = circus } → *"This circus sucks!"*
PASS Rule6: { who = nick, concept = onHit, hitBy = zombieclown } → *"Stupid clown!"*
PASS Rule7: { who = nick, concept = onHit, hitBy = zombieclown, curMap = circus}

Rule Systems Methods

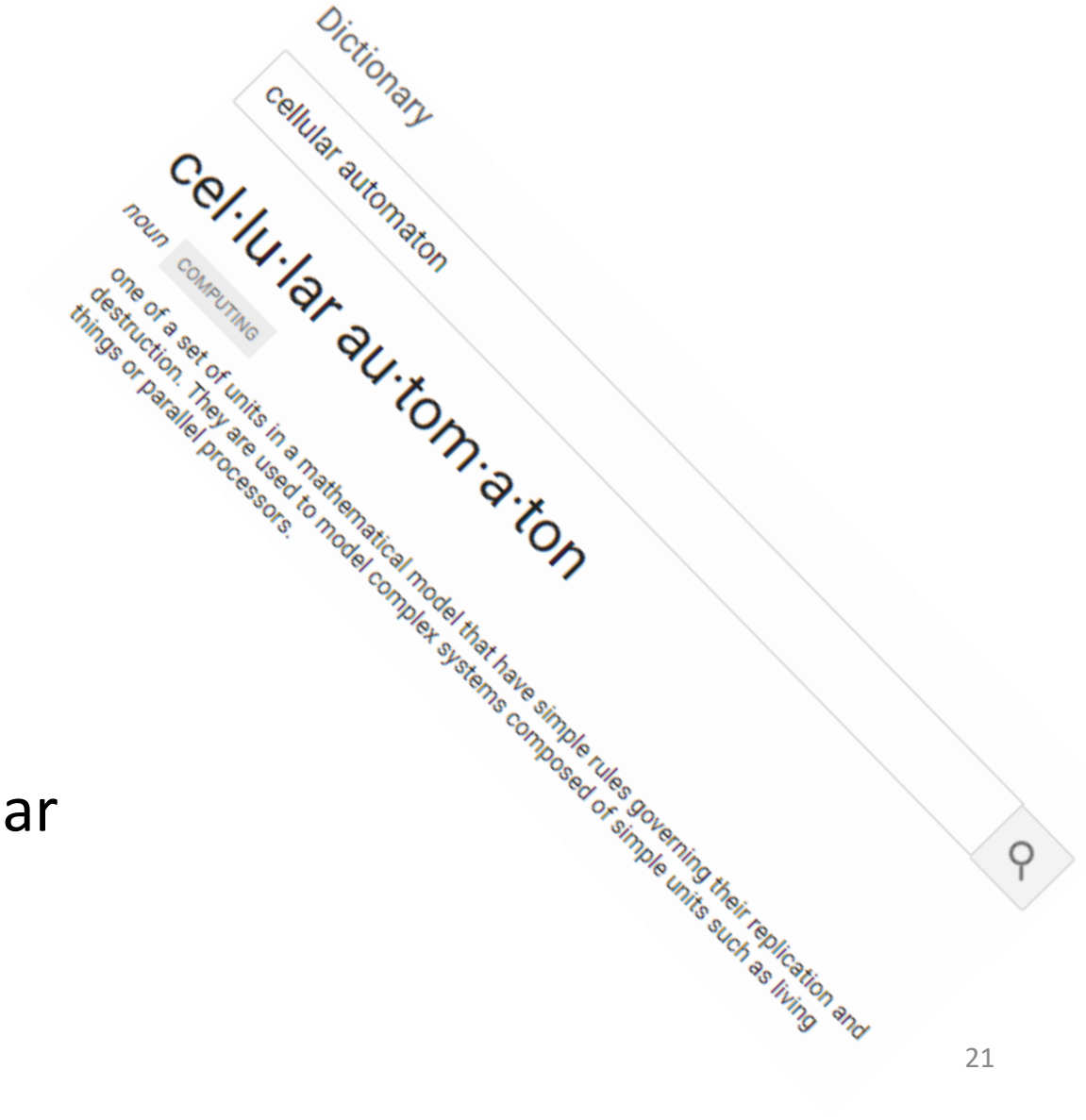
Cellular Automaton

1. Take noise/random values
2. segment bits/space into grids
3. Each grid updates until stable/time limit

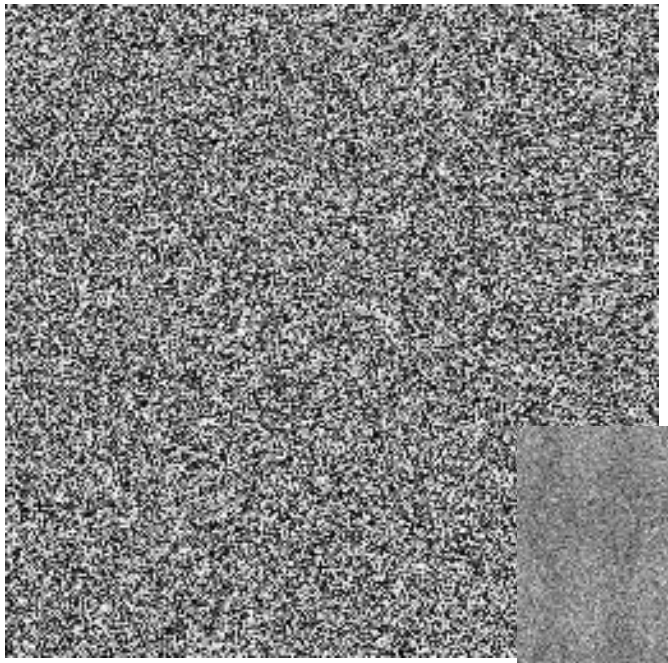
Agent-based simulation

1. Take noise/random values
2. Iterate each agent over the world
3. Continue until stable/time limit

Where are the rules? In each grid cell (cellular automaton) or in an agent (agent-based simulation)

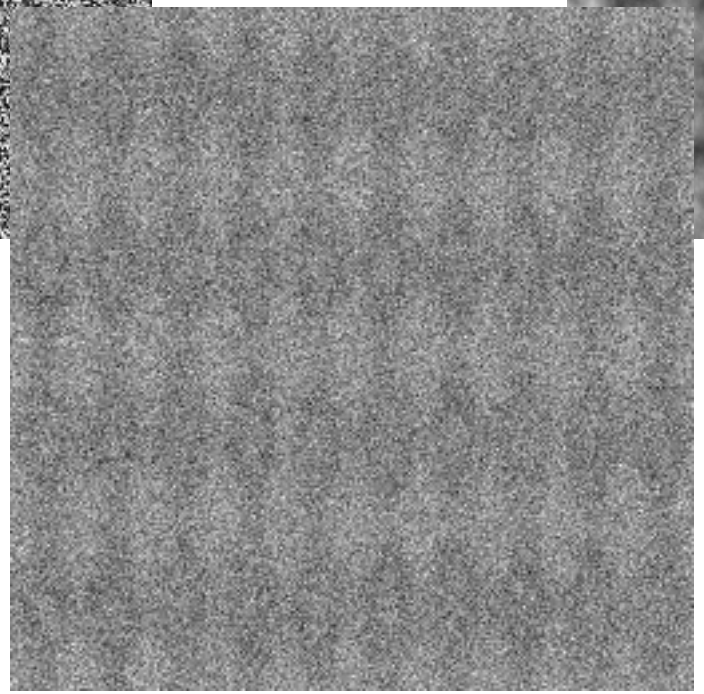


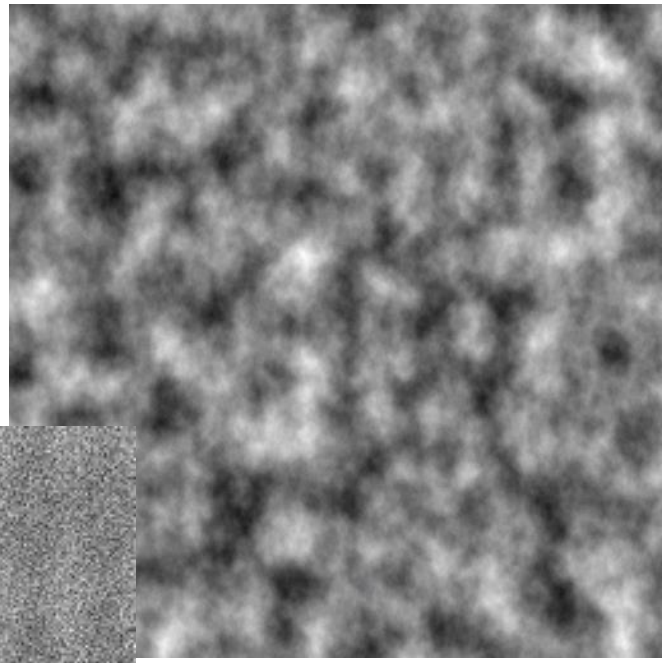
Quick note: Noise > Random



Pure “random”

Sin wave





Perlin noise

Cellular Automaton

Stanislaw Ulam and John von Neumann (1940)

Conway's Game of Life made famous (1970s)

Rules:

1. Any live cell with fewer than two live neighbors dies, as if caused by underpopulation.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Cellular Automaton Example

https://www.youtube.com/watch?v=OumDj9PNWjE

Cellular Automaton

Pros

- Easy to implement
- Complex behavior from small set of rules
- Intuitive mapping to game space generation
- Local coherence

Cons

- Reliant on guess and check for modifying
- Hard to spot edge cases
- No way to forbid undesired output
- Upper bound is slow
- Global incoherence

Question

Cellular Automata are good at local coherence and bad at global coherence, which limits its usage cases (caves, mazes, etc).

What could we add or change to this approach to allow for a designer to ensure global coherence?

What if…

- What if we had another set of rules, not for a single grid cell, but for the whole grid?
- Well, in that case we don’t have to use a grid at all. All the rules could be on this layer
- And we could even have rules that contradict one another, so we’d get…

Agents

- Instead of individual grid rules, have multiple agents that iteratively update a map.
  - Example:
    - Mountain agent: make high points higher
    - Valley agent: flatten out sections
    - River agent: make low points lower and add water
    - Manager: make parts of the map non-editable when they reach thresholds
- Harder to come up with rulesets that converge completely on their own, so dependent on time

Comparisons: CA vs Agents

- Similarities (rule system commonalities):
  - Rely on “emergent” output of simple rules
  - Hard to make strong designer restrictions
  - Requires a lot of guess and check
  - Can take a long time to converge
- Differences
  - Agents allows for some global assurances
  - CA more emergent in its output

Major Drawback of Rule Systems

- What cellular automaton could you construct to build a house? A sword? An NPC?
  - Local rules don't adapt well to generating bits that require strong global coherence
  - Said another way, no one wants an NPC that only looks locally like a human

http://pcgbook.com/wp-content/uploads/chapter05.pdf

PCG: GENERATIVE GRAMMARS

PCG high-level Methods

- Search
- Rule systems
- Generative Grammars
- Constraint Solving

Solution: Generative Grammars

- Noam Chomsky's study of languages in the 1960s
  - Rich taxonomy of grammars
  - Widespread application
- Key questions: determinism & order of expansion
- Components with rules about how they are allowed to be put together
  - Words that can be placed together in a sentence according to grammar rules
  - Limbs that can be combined to form creatures according to designer rules

http://pcgbook.com/wp-content/uploads/chapter05.pdf

Formal Grammars

- Grammar: set of **production rules** for rewriting strings
  - Rule: <symbols> → <other symbols>
- E.g. (left hand side vs right hand side; terminal vs nonterminal)
  - A → AB
  - B → b
  - Given ‘A’: [1] AB [2] ABb [3] ABbb
- Applying: for each sequence of LHS symbols in string, replace with RHS of rule

Determinism & Order

- Determinism
  - Deterministic grammar: one rule that applies to each symbol or sequence
  - Nondeterministic grammar: several rules may apply
    - Random
    - Probabilities & Priors
- Rewriting order
  - Sequential: left to right, rewrite string while reading it
  - Parallel: all rewriting done at same time

L-systems

- Class of grammars w/ parallel rewriting
- Aristid Lindenmayer, 1968: model growth of organic systems
- EG (Given A): A → AB B → A
  - A
  - AB
  - ABA
  - ABAAB
  - ABAABABA
  - ABAABABAABAAB

Application and Interpretation

- One approach: interpret strings as drawing instructions
  - F: move forward 10 pixels
  - L: turn left 90
  - R: turn right 90
  - EG: F → F L F R F R F L F

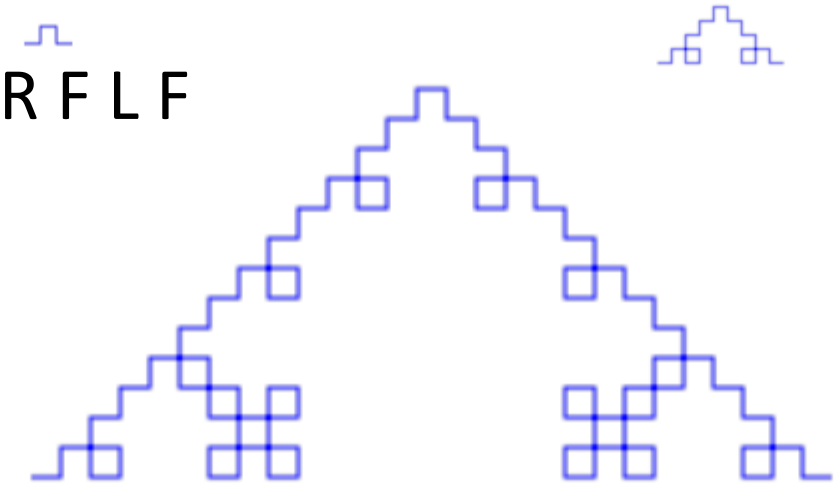
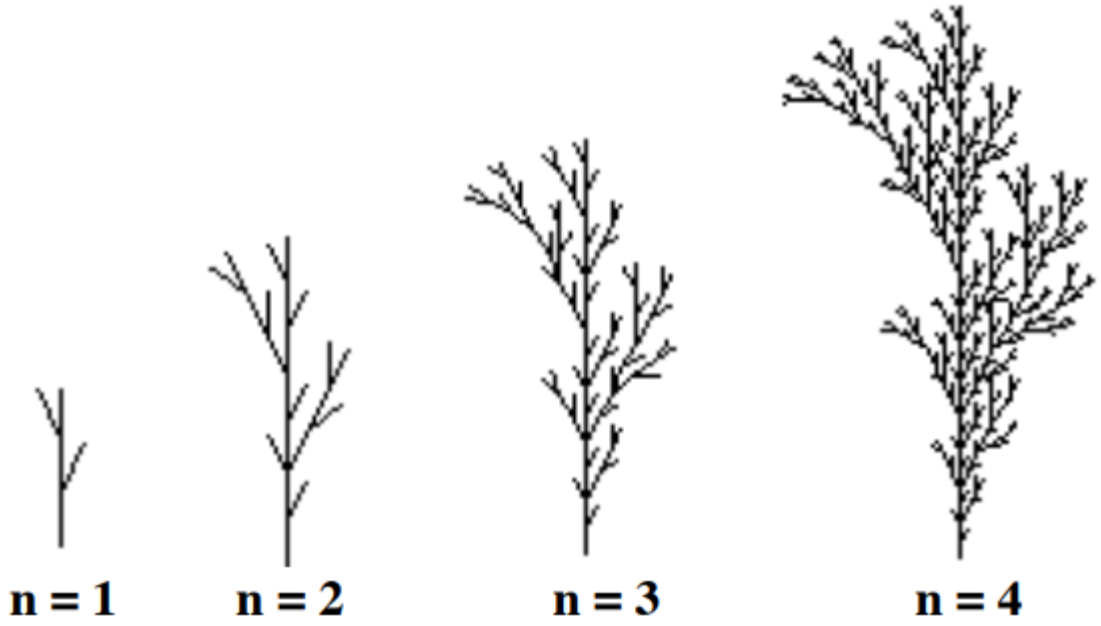


Fig. 5.2: Koch curve generated by the L-system $F \rightarrow F + F - F - F + F$ after 0, 1, 2 and 3 expansions

http://pcgbook.com/wp-content/uploads/chapter05.pdf

Bracketed L-systems

- How do we “lift the pen”?
  - Plants are branching, and branches end
- Introduce two extra symbols: ‘[‘ and ‘]’. A FILO stack
  - [push
  -] pop
- EG: F → F[RF]F[LF][F]



http://pcgbook.com/wp-content/uploads/chapter05.pdf

Beyond strings

- Generative grammars are not restricted to representation as strings
  - Graphs, tile maps, 2D/3D shapes, etc.
- Graph grammar:
  - Find subgraph in target that matches LHS; mark subgraph w/ IDs
  - Remove all edges between marked nodes
  - Transform marked nodes into corresponding RHS
    - Add a node for each node on RHS not present in target
    - Remove any nodes that have no corresponding node on RHS
  - Copy edges as specified by RHS
  - Remove all marks

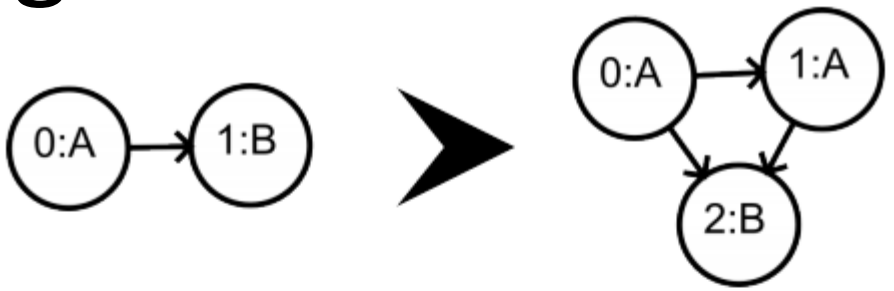
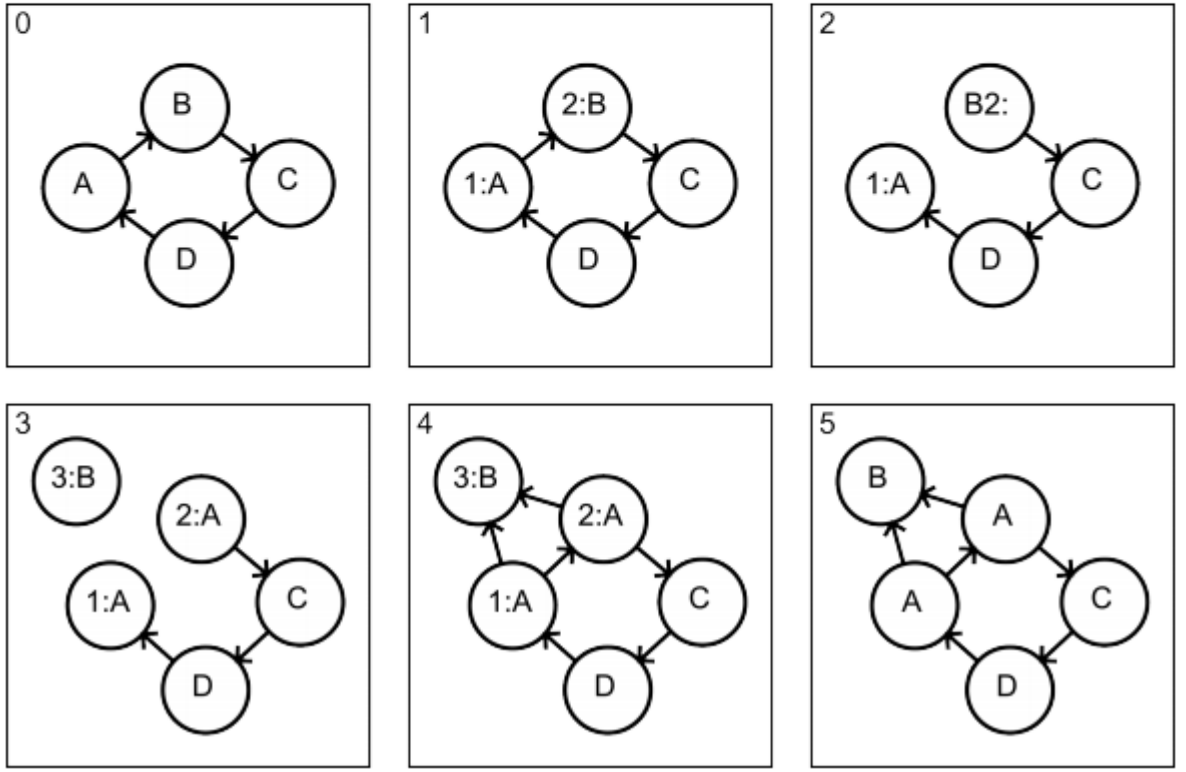


Fig. 5.7: A graph grammar rule



http://pcgbook.com/wp-content/uploads/chapter05.pdf

Example: Spelunky Levels

- Each Spelunky level starts life as a grid.
- Each grid can be filled in with potential templates designed by Derek Yu (designer)
- Templates are filled in one by one
- Later templates chosen according to rules about how templates can neighbor

Game Maker's Toolkit: https://www.youtube.com/watch?v=Uqk5Zf0tw3o

Generative Grammars

- Generative grammars are well-suited to both game bits and game spaces
  - Borderlands' guns
  - Bloodborne's chalice dungeons
- …but they are only as good as the components and rules the designer creates
  - No man's sky's ship and creature generation

Generative Grammars

Pros:

- General usage
- High-quality, “feels designed” quality
- Accessible
- Fast

- Cons:
  - Large burden on design
  - Hard to “debug”
    - How do you know if a fix actually fixed anything unless you generate infinite output?

Comparing the Approaches

- **Rule Systems** allow for emergent output from simple rules. But hard to control output.

- **Generative grammars** can create high-quality output, but depend upon expert authoring of components and rules

- **Search** cuts back on authoring burden to just a heuristic and searchable representation, but these are unintuitive for many designers

See http://pcgbook.com/wp-content/uploads/chapter07.pdf

PCG: CONSTRAINT SOLVING & PLANNING

Constraint Satisfaction

- Also called constraint solving/constraint propagation
- Similar to search *but* instead of a heuristic with a single value, designers give constraints
  - Constraints are facts that must be true in the final “answer” output
  - Constraints can be authored or learned from an example
- Shout out to Dr. Adam Smith at UCSC

Constraint Satisfaction Problem (CSP)

Input

- Set of variables/slots
- Set of all values that can go in the slots
- Set of constraints that relate slots to each other

Output

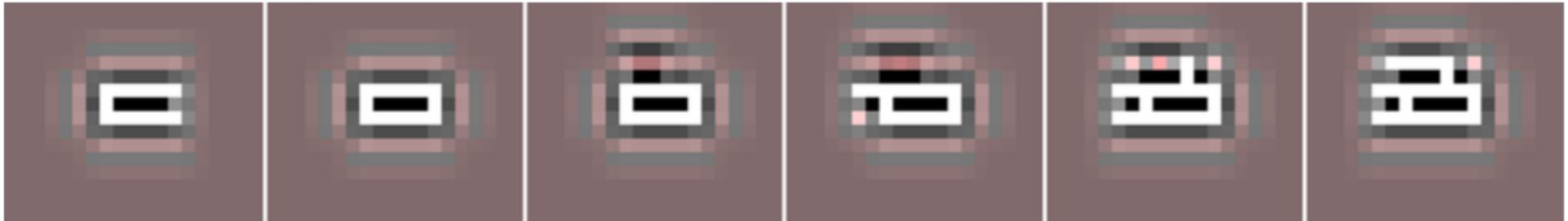
- All variables are filled with a specific value

Trivial Example: Name generator

- Variables: <First Name> and <Last Name>
- Values: List of names
- Constraints: Names cannot start with the same letter

CSP (General) Process

- Initially, all variables have a list of all values they could *potentially* have
- Each update, pick one variable and choose arbitrarily from its remaining values (collapse)
- Based on the constraints, remove values from the surrounding variables to represent that they are no longer possible (propagation)
- End when all variables have only one value



CSP related to Search/Planning

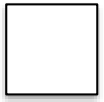
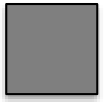
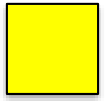
- We have seen the idea of iteratively solving constraints before in partial order planning
  - Use similar heuristics: most constrained variable, minimum remaining values

- CSP is essentially search, but is generally faster as we cut out vast parts of the search space due to constraint propagation

CSP: Dungeon Room Example

1-4	E	1-4	1-4
1-4	1-4	1-4	1-4
1-4	1-4	1-4	1-4
1-4	1-4	1-4	1-4

Values:

1. Blank 
2. Wall 
3. Enemy 
4. Treasure 

Constraints:

- There must be a path from the entrance to all treasure and all enemies
- There can only be at most 1 treasure
- There can only be at most 2 enemies
- Enemies cannot be next to each other

CSP Step 1

Collapse:

1-4	E	1-4	1-4
1-4	1-4	1-4	1-4
1-4	1-4	1-4	1-4
1-4	1-4		1-4

Propagate:

1-4	E	1-4	1-4
1-4	1-4	1-4	1-4
1-4	1-4	1,2, 4	1-4
1-4	1,2,4		1,2, 4

Constraints:

- There must be a path from the entrance to all treasure and all enemies
- There can only be at most 1 treasure
- There can only be at most 2 enemies
- **Enemies cannot be next to each other**

CSP Step 2

Collapse:

1-4	E	1-4	1-4
1-4	1-4	1-4	1-4
1-4	1-4	1,2, 4	1-4
1-4	1,2, 4		

Propagate:

1-3	E	1-3	1-3
1-3	1-3	1-3	1-3
1-3	1-3	1,2	1-3
1-3	1,2		

Constraints:

- There must be a path from the entrance to all treasure and all enemies
- **There can only be at most 1 treasure**
- There can only be at most 2 enemies
- Enemies cannot be next to each other

CSP Step 3

Collapse:

1-3	E	1-3	1-3
1-3	1-3	1-3	1-3
1-3	1-3		1-3
1-3	1,2		

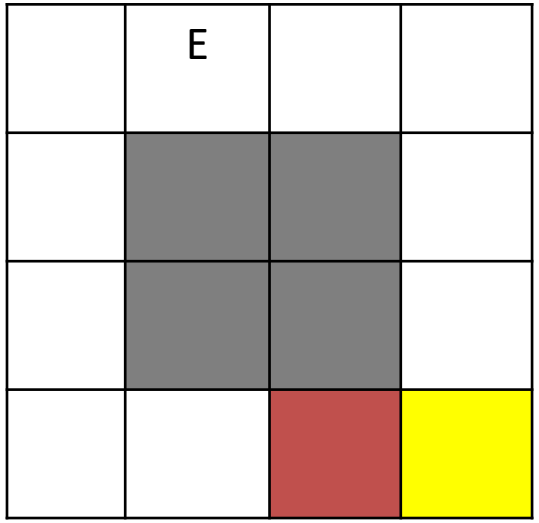
Propagate:

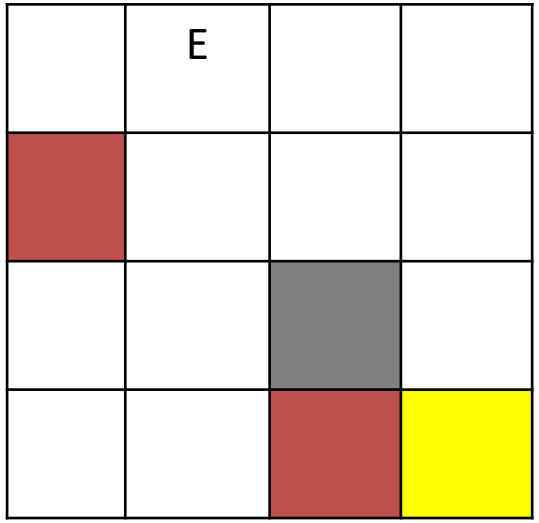
1-3	E	1-3	1-3
1-3	1-3	1-3	1,3
1-3	1-3		1,3
1-3			

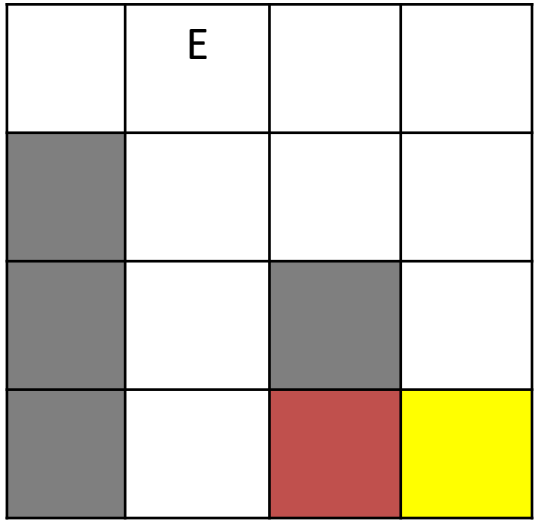
Constraints:

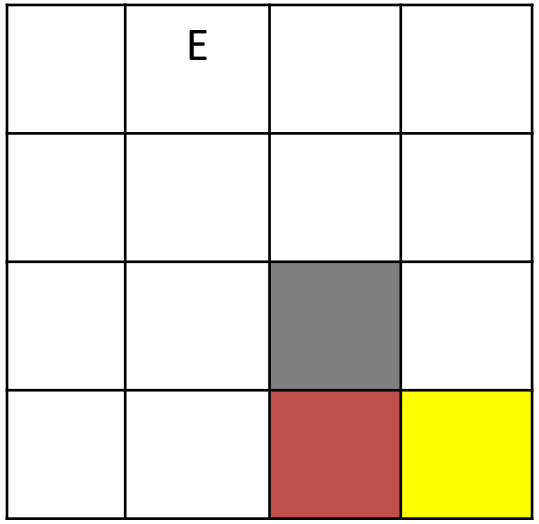
- **There must be a path from the entrance to all treasure and all enemies**
- There can only be at most 1 treasure
- There can only be at most 2 enemies
- Enemies cannot be next to each other

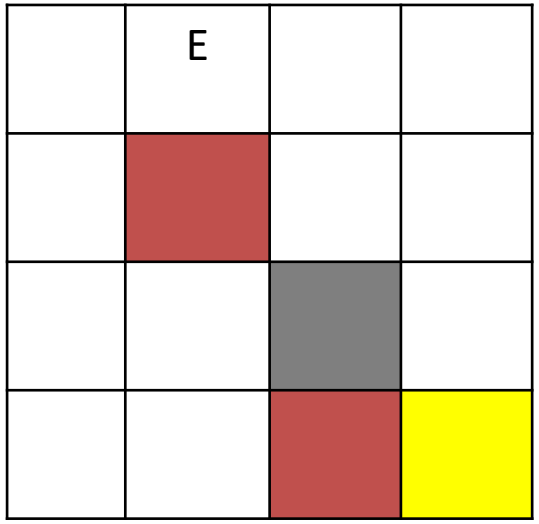
CSP Output

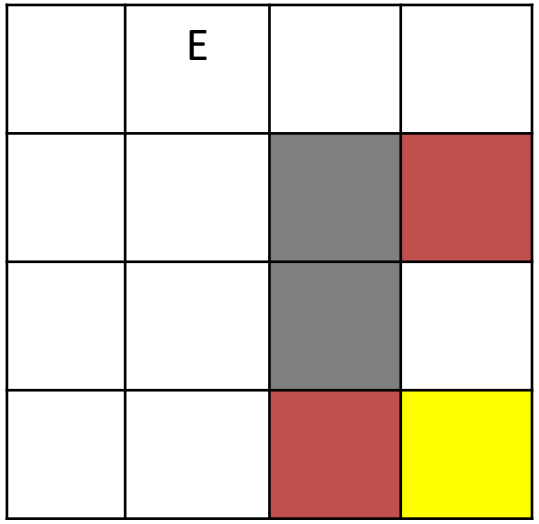












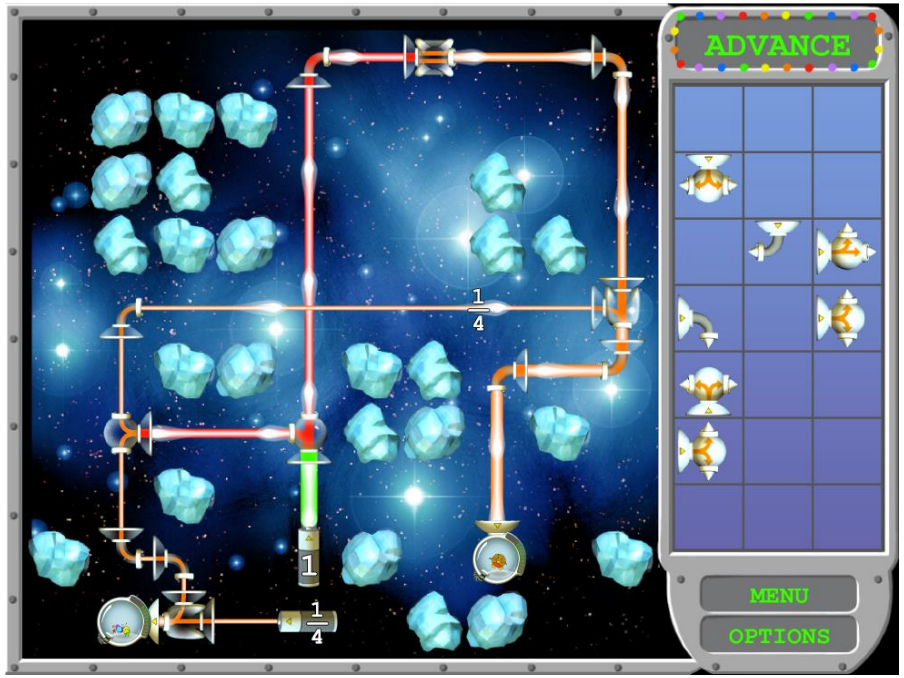
CSP Usage

If output doesn't look good, just add more constraints!

- Don't have to consider balancing heuristic

Need to strike balance of overly constrained/avoiding bad output

- In general, designers prefer this over heuristic tuning



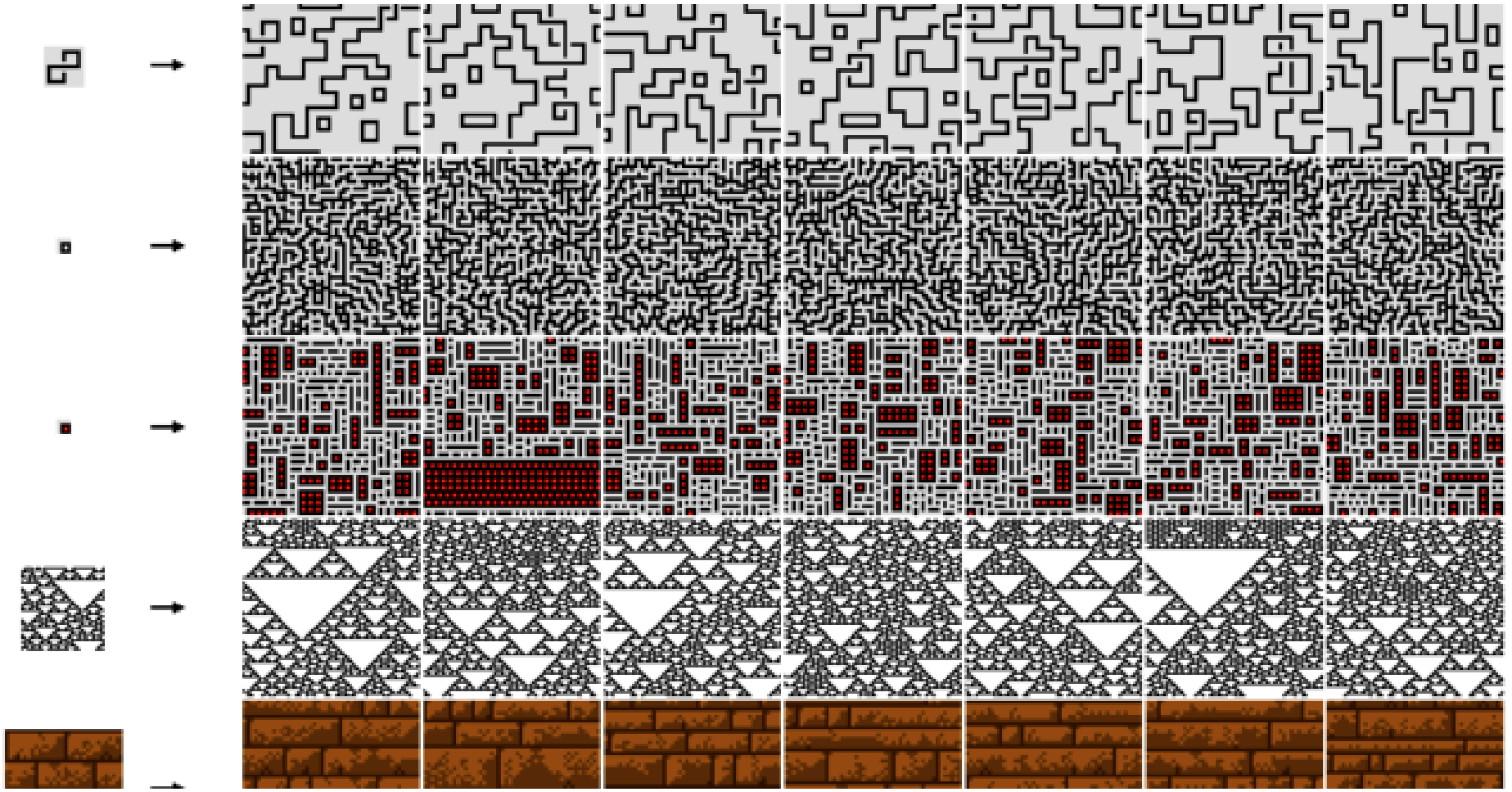
Question 1

- Come up with a very simple generation task to which you could apply constraint satisfaction
  - Give the variables for the problem
  - Give the values for those variables
  - Give the constraints

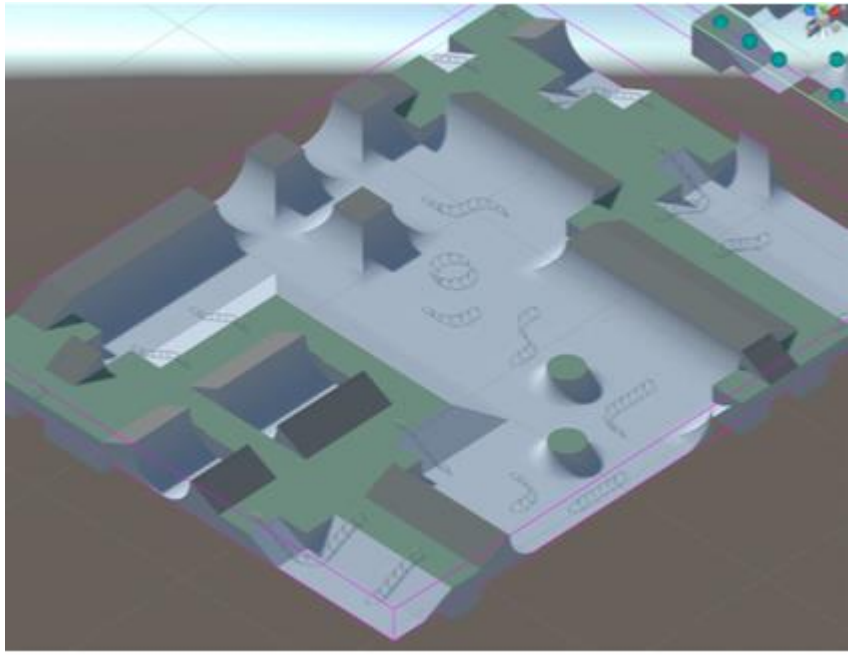
Writing constraints seems hard

Didn’t you say constraints could be learned?

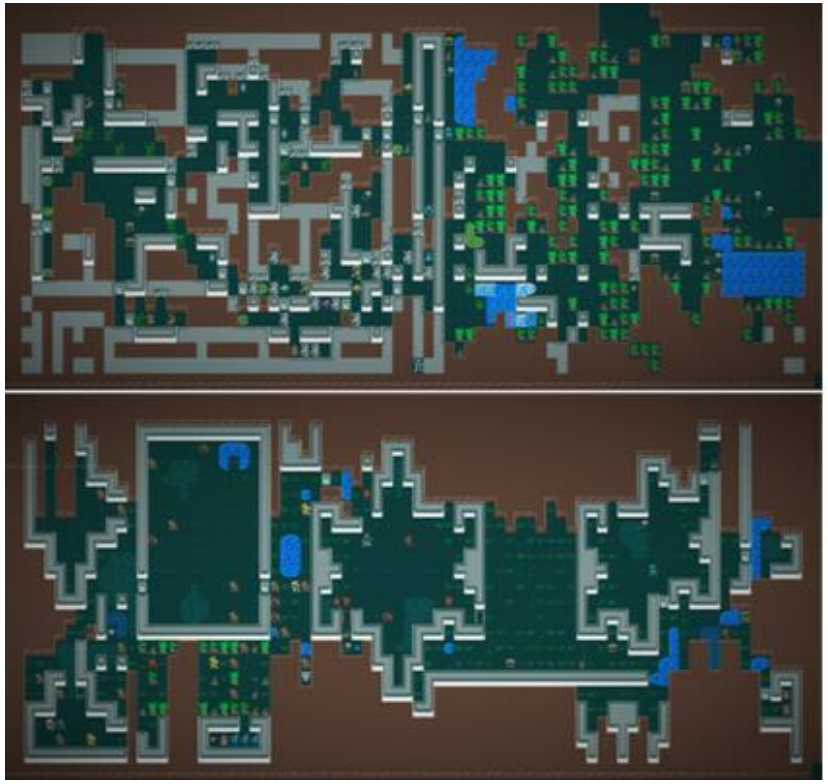
Wave Function Collapse



Wave Function Collapse: The New Hotness



Proc Skater, Joseph Parker (2016)



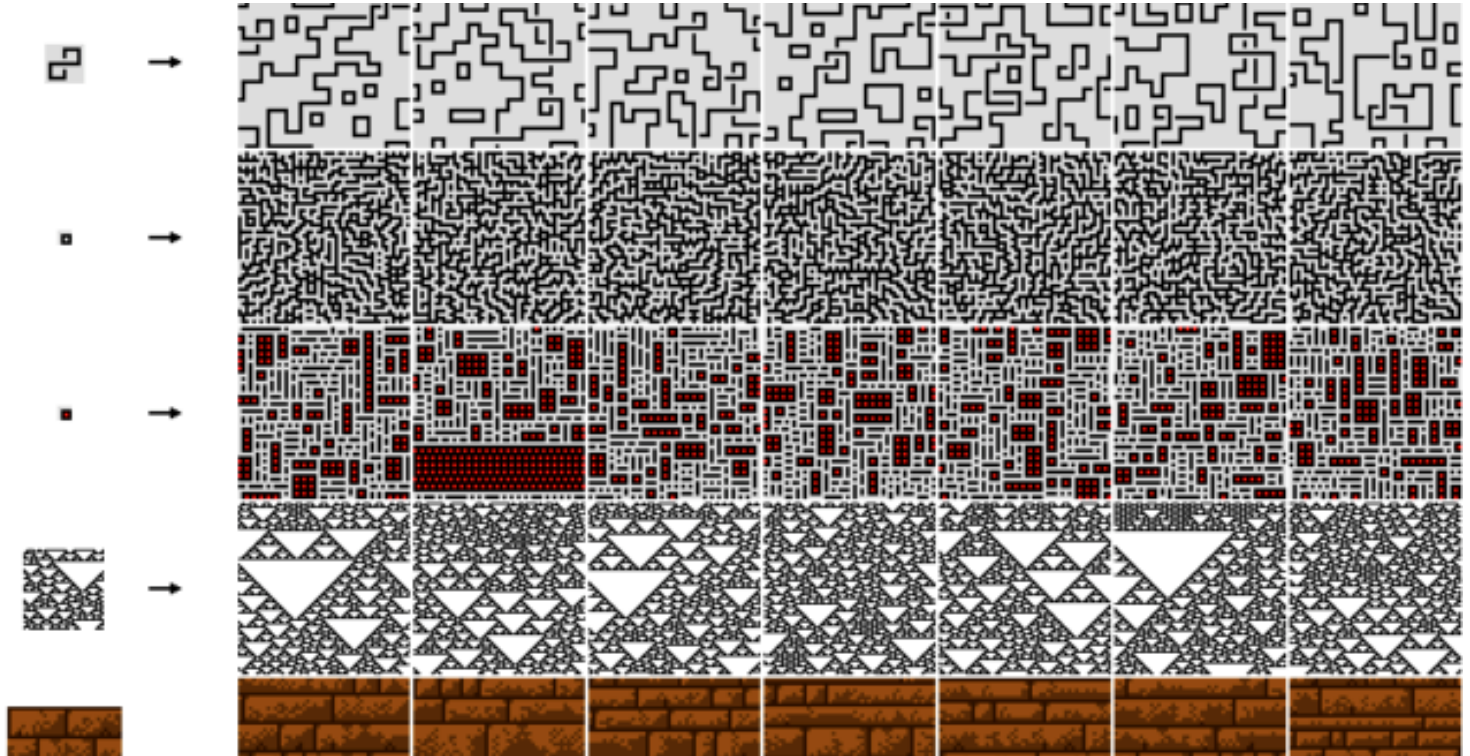
Caves of Qud, Freehold Games (2016)

https://github.com/mxgmn/WaveFunctionCollapse

WaveFunctionCollapse is Constraint Solving. 2017. Workshop on Procedural Content Generation

Wave Function Collapse Big Ideas

1. Derive legal patterns from a sample input
2. Determine legal intersections of patterns to get constraints



Wave Function Collapse Algorithm

PatternsFromSample()

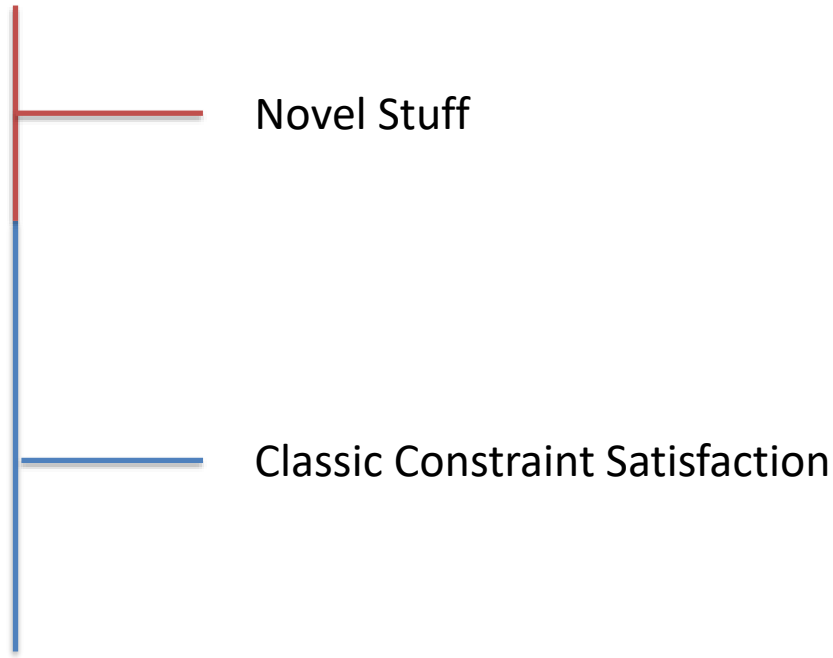
BuildPropagator()

While notFinished:

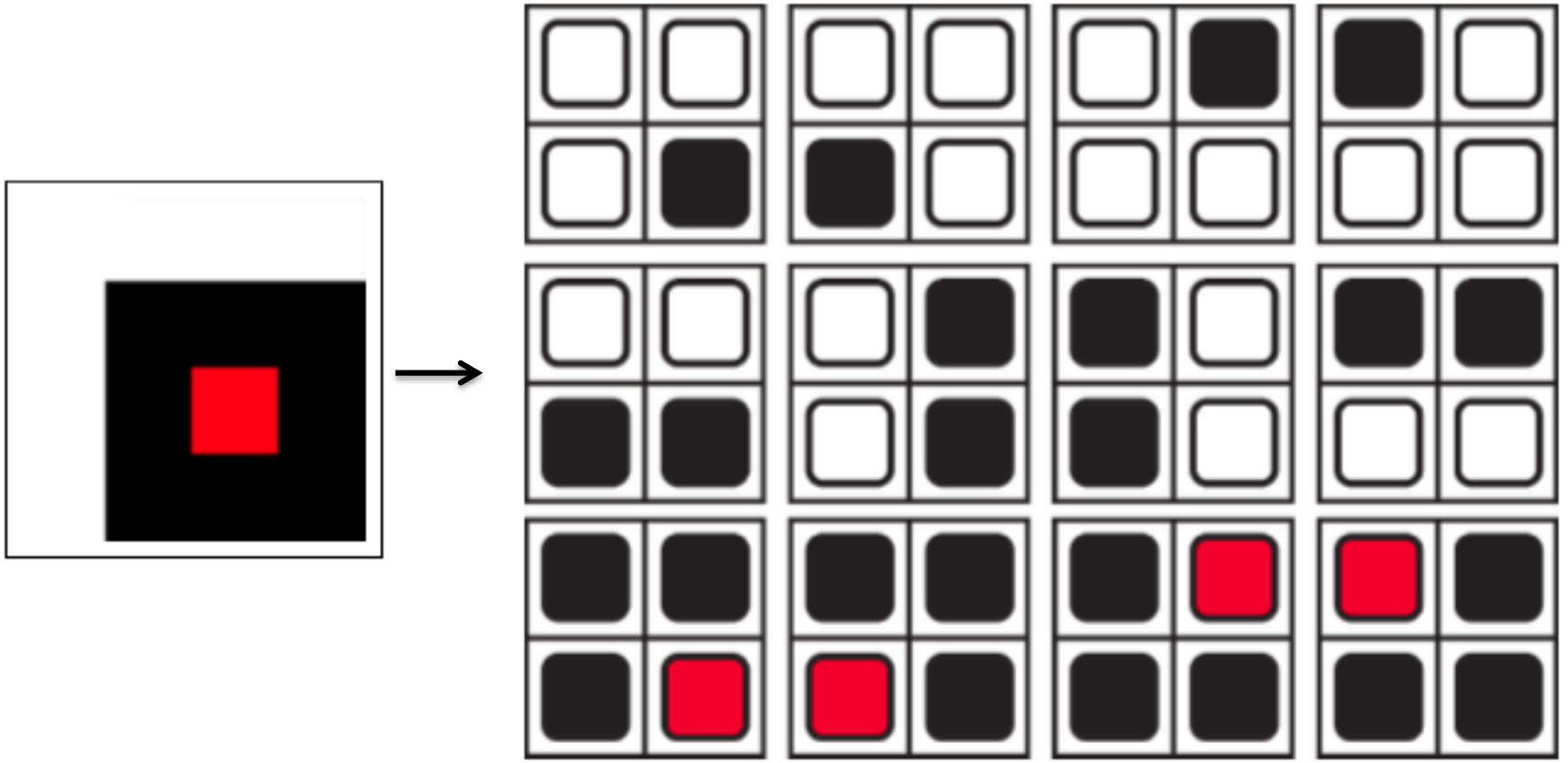
    Observe()

    Propagate()

OutputObservations()

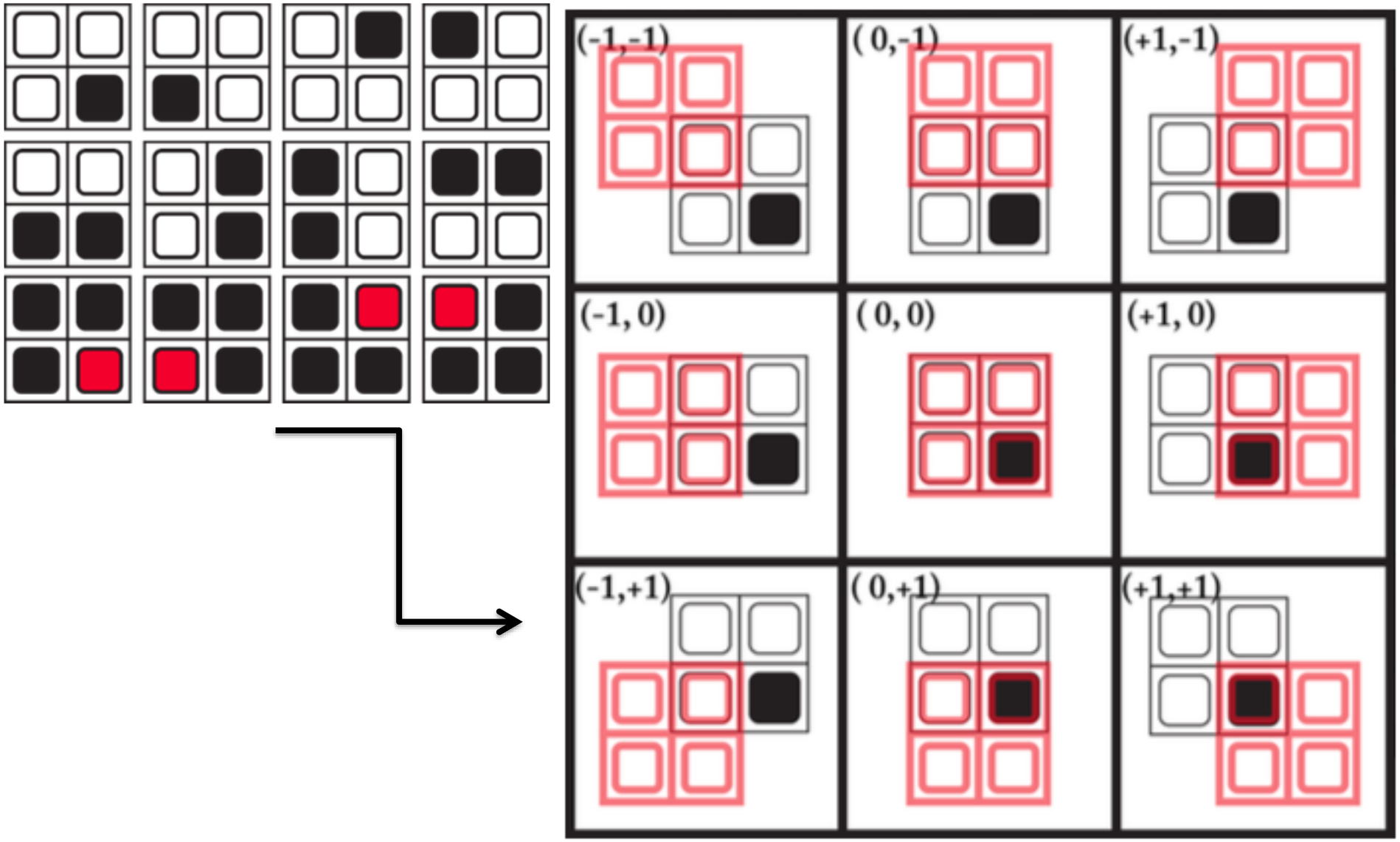


Patterns from Sample
(given neighbor size N=2)

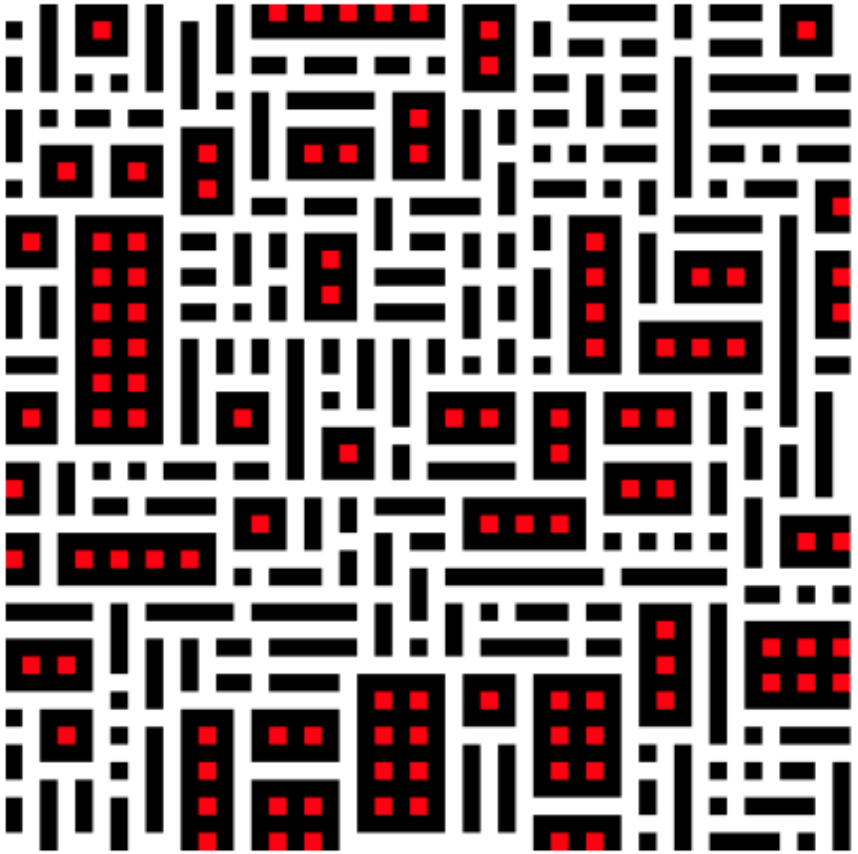


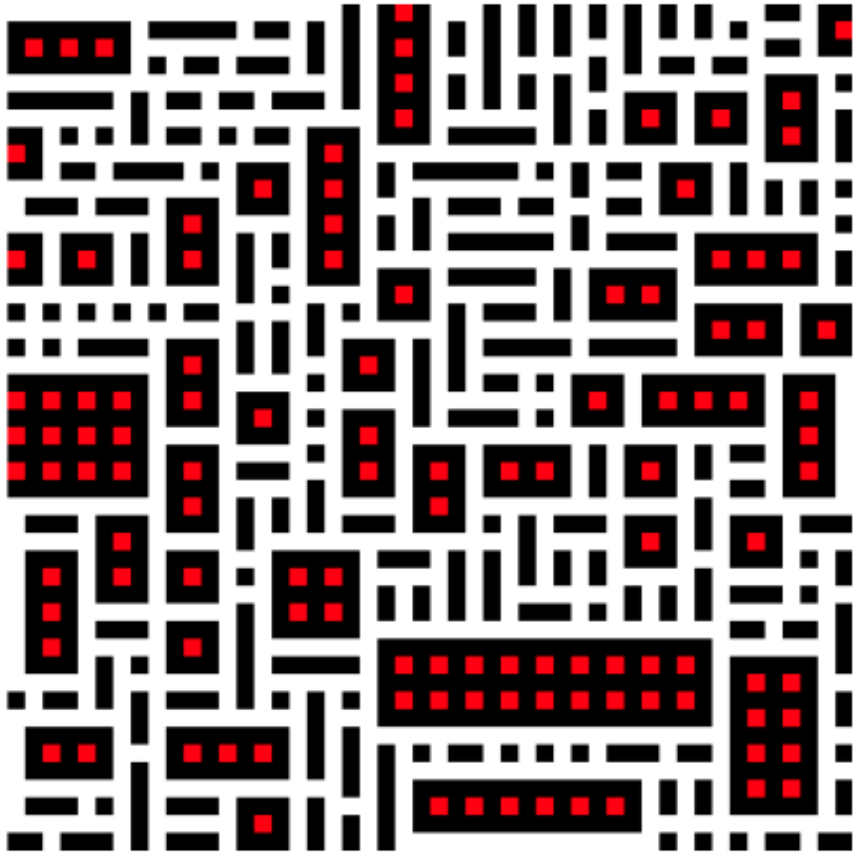
The larger the neighbor size, the more like the input sample

Build Propagator



Output





CSP Pros/Cons

Pros

- Constraints are more intuitive than heuristics
- Faster in many cases than search
  - Though slower than grammars due to propagation
- Can learn constraints from exemplar

Cons

- Must be able to represent problem in terms of formal logic

Story/Quests in Games

Story: Series of events players experience

- Linear: players experience the same story
- Multilinear/Non-linear: players experience variations on the same story based on choices

Quests: Give the player something to do for experience points, challenge, or the story

Why would we want to generate these?

- Add more content to a game

- Create new types of game experiences
  - More on this in the third PCG lecture

EA wants to use machine learning to create real-time game narratives

Quest Generation: Templates

- We can create a quest template based on standard quest types or to fulfill a specific in-game purpose

- Example:
  Assassination: Go to <LOCATION> and kill <Non-Essential NPC>

Quest Templates

- Variables: like <LOCATION>, slots that can be filled by specific values
- Values: The elements that can fill variables
- Rules: Constraints between variables that limit the types of values we can place in them.

- Generation is then trivial random selection

Quest Templates Example: Bethesda

- Collect values of types (LOCATION, NPC, etc) once the player has visited or been made aware of them
- On generation select a template and fill in with values from these types

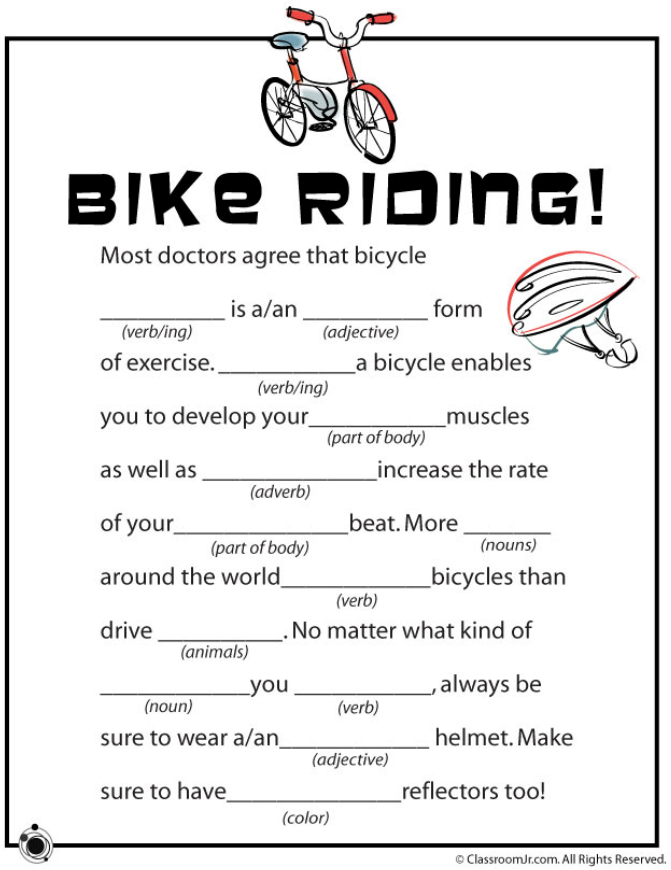
Quest Templates

Pros

- Fast (basically a generative grammar)
- Can be customized to player experience

Cons

- Basically madlibs where you do the same madlib over and over again
- Gets boring quickly



BIKE RIDING!

Most doctors agree that bicycle

_____ is a/an _____ form
(verb/ing) (adjective)

of exercise. _____a bicycle enables
(verb/ing)

you to develop your_____muscles
(part of body)

as well as _____increase the rate
(adverb)

of your_____beat. More _____
(part of body) (nouns)

around the world_____bicycles than
(verb)

drive _____. No matter what kind of
(animals)

_____you _____, always be
(noun) (verb)

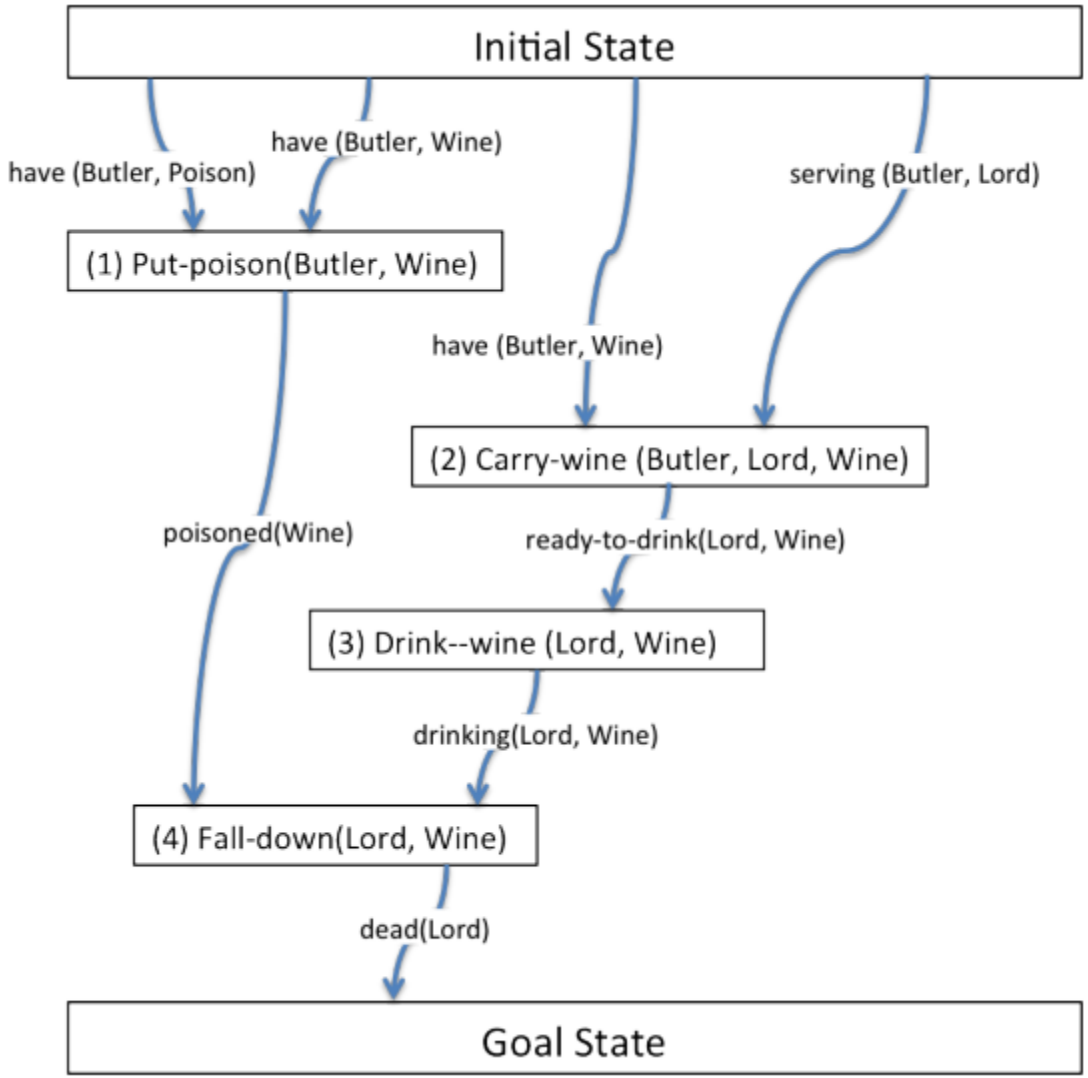
sure to wear a/an_____ helmet. Make
(adjective)

sure to have_____reflectors too!
(color)

Story Generation

- Templates
- Planning (forward, backward, POP, HTN)
- Talespin (1992)

One day Joe Bear was hungry. He asked his friend Irving Bird where some honey was. Irving told him there was a beehive in the oak tree. Joe walked to the oak tree. He ate the beehive.



http://pcgbook.com/wp-content/uploads/chapter07.pdf

Plan-Story Examples

One day Joe Bear was hungry. He asked his friend Irving Bird where some honey was. Irving told him there was a beehive in the oak tree. Joe threatened to hit Irving if he didn't tell him where some honey was.

Dr. Evil went to a bank. Dr. Evil withdrew cash from his account to buy a gun. Dr. Evil traveled to a gun store. Dr. Evil bought a gun. Dr. Evil traveled to the White House. Dr. Evil shot the president with his gun.

Planner Approaches

- Agent-based: Each agent is a planner, comes up with plans. Replan if they break
  - State: Current local state to an agent
  - Actions: Actions that agent is capable of

- Narrator-based: High-level story plans, with a much larger search space
  - State: The entire current story state (characters, locations, items, etc)
  - Actions: Everything any character or the world could do

Story Domain

- The model of possible actions, locations, characters that can occur from a specific domain

- This must be authored by a designer instead of a typical story to use one of these approaches

Question 2

This all works well for a linear story, but what if we wanted to make a multilinear/non-linear story?

My Answer

- Well really Fred Charles, Marc Cavazza and Steven Mead's answer
- Hierarchical Task Networks
  - Replan just the chunks of the story the player messes up
- Examples
  - https://www.youtube.com/watch?v=0erFey9hQTY
  - https://www.youtube.com/watch?v=3wzSb8fzDA4

Story Planning

Pros:

- Many possible stories
- Can adapt to the player

Cons:

- Slow
- Hard to get heuristics right (optimal stories rarely interesting)

From AI Game Programming Wisdom 2 (Rabin, 2004)

CASE STUDY: EMPIRE EARTH (SHOEMAKER'S MAPMAKER)

Stainless Steel Studios

- Dedicated exclusively to designing RTS games
- RT 3D game engine: TITAN 2.0
  - Windows, complete source (c++), 40+ person years
  - 3 million units sold (got cash?)
- Empire Earth
  - Released Nov 2001
  - "… *a new high-water mark for realtime strategy fans.*" GameSpy's Dave Kosak
  - Same lead game designer (Rick Goodman) as AOE
- Empires: Dawn of the Modern World

Birds-Eye

Fancy 3D View

If You Were Asked…

- Make a system to generate maps
  - Tile-based
  - RTS
  - Multi-player
- What qualities should these maps have?

These Maps Must be _____?

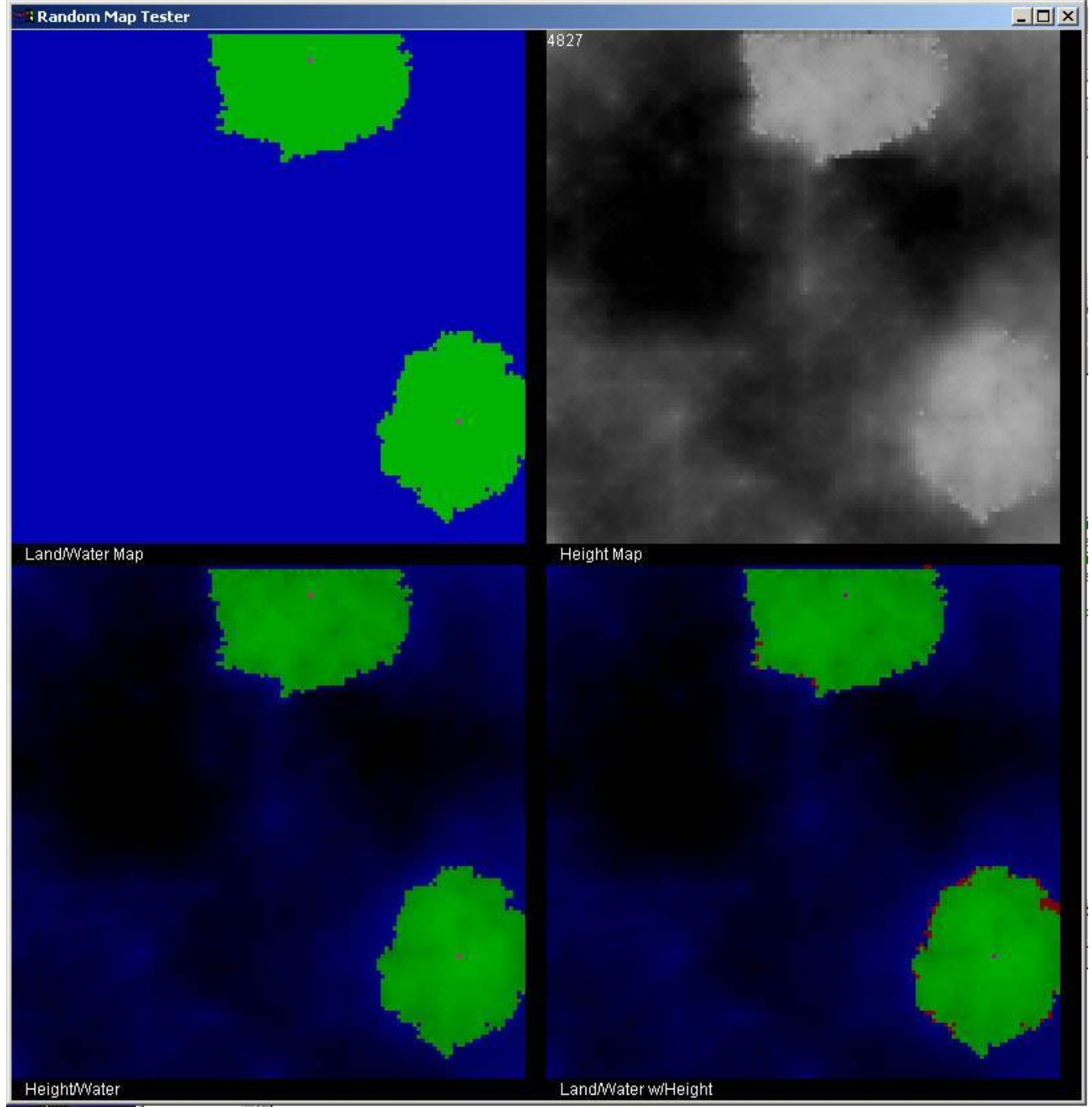
- Fair
  - Land per player
  - Resources per player
  - No strategic advantage
- Predictable
  - Similar by type
  - Allow for diff. outcomes
- Reproducible
  - Seed & type & size
  - Number of players
- Varied
  - Size
  - Type
- Relatively Quick
  - Many in-game systems not running
  - Impatient to start

Shoemaker's Mapmaker

- 1 .dll & 50+ script files
- Different .dll can be specified
- 7 scripts for each map type
- Inputs
  - # players, # teams
  - Size
  - Climate
  - Type
  - Random # seed

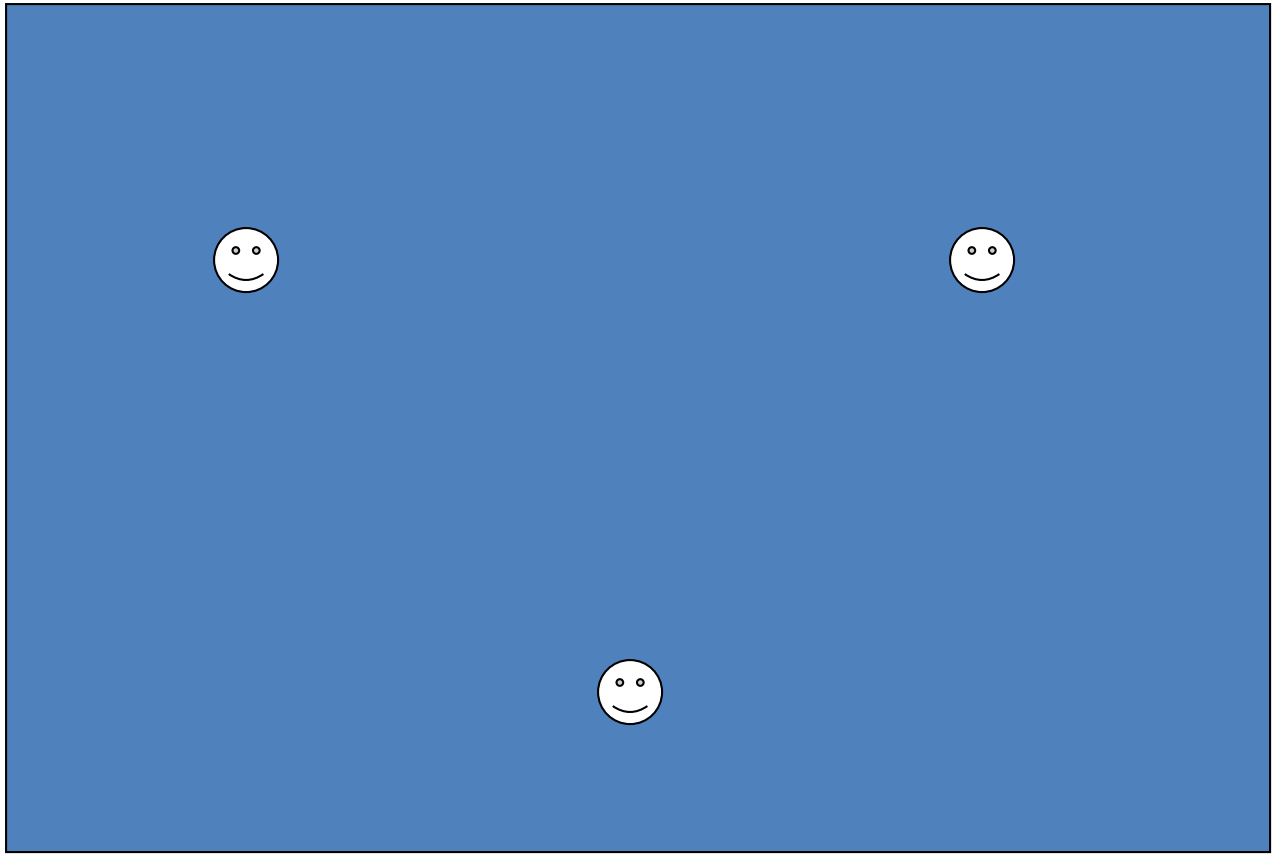
Intermediate Results

- Visualization tool created for 2D arrays
  - Land/Water
  - Height
  - Height/Water
  - Combined
- Map engine integrated late into game


Random Map Tester
4827
Land/Water Map
Height Map
Height/Water
Land/Water w/Height

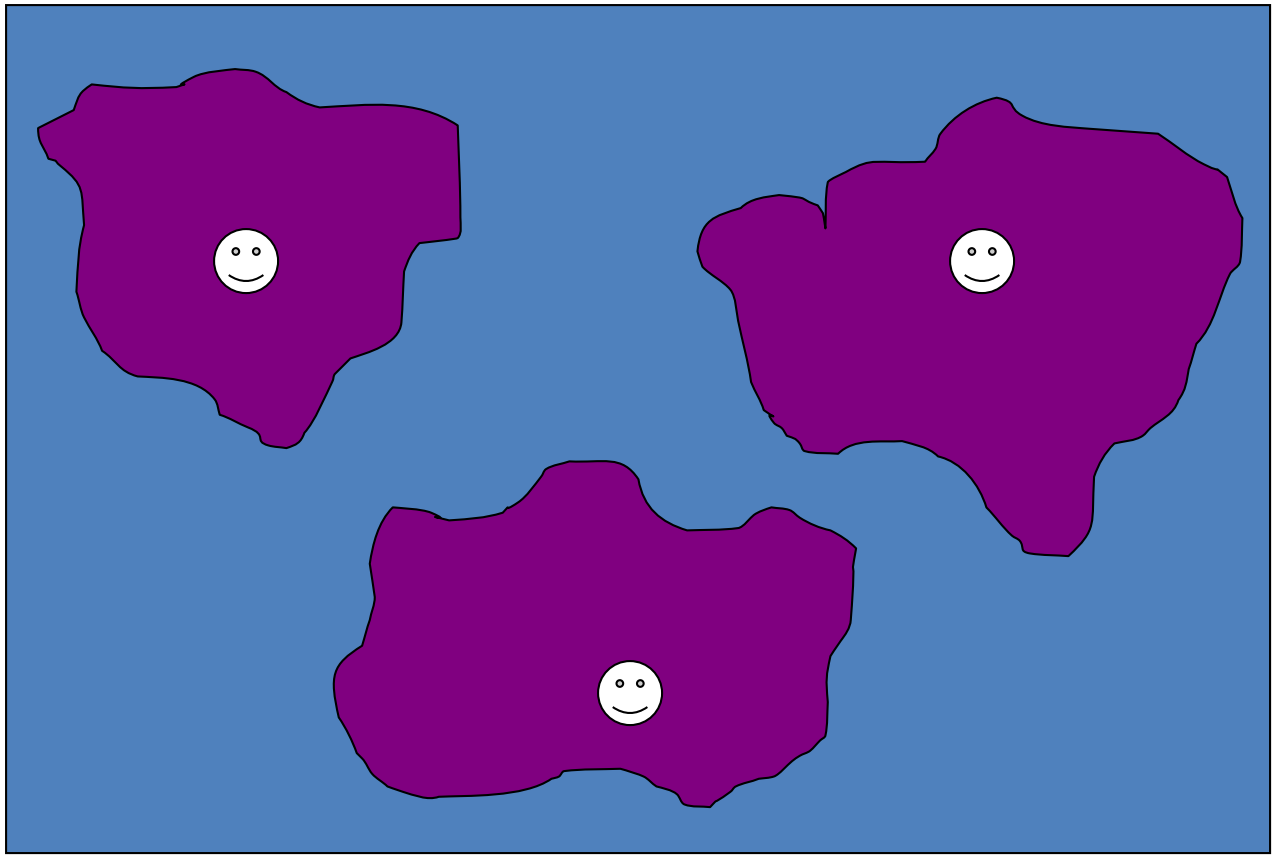
Solution Overview

- Step 1: Place players (blank map of H_2O)



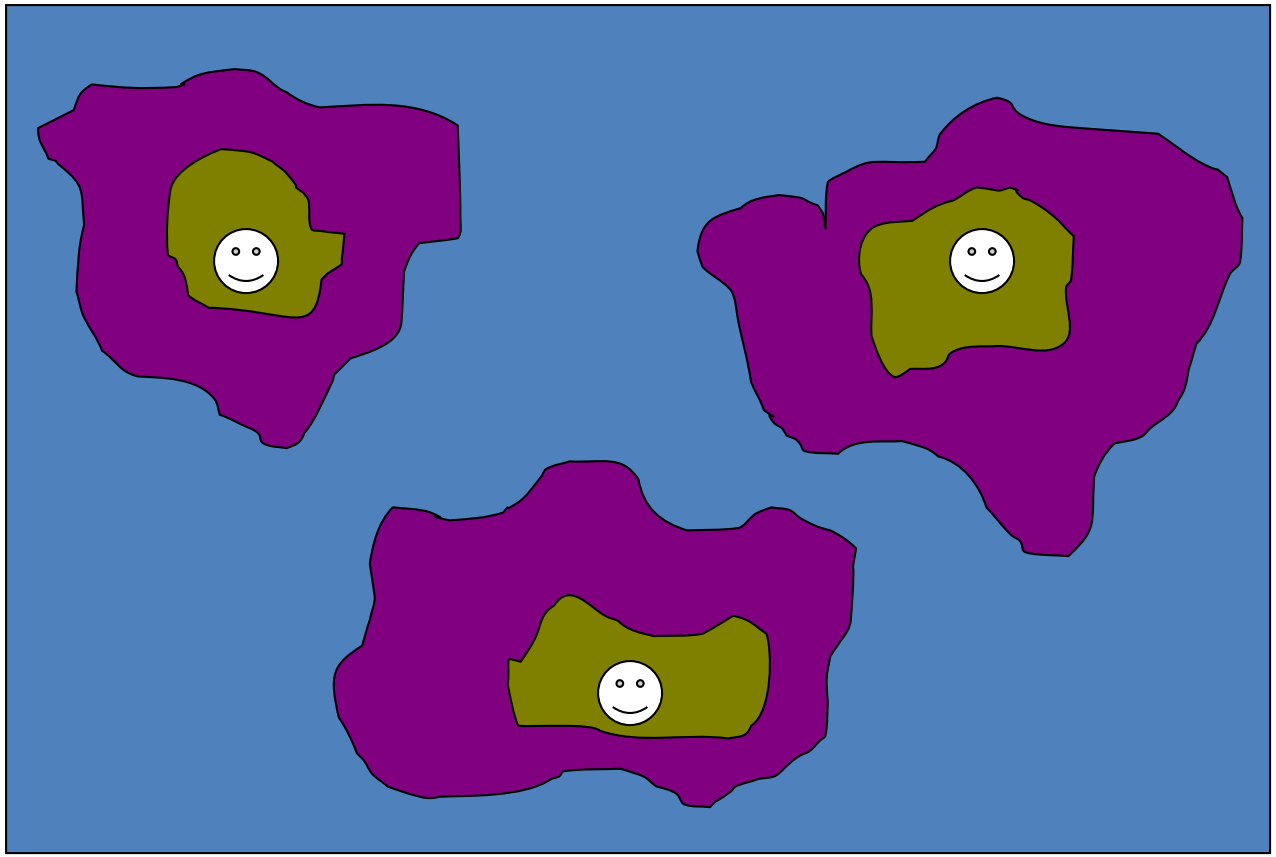
Solution Overview

- Step 2: Grow player land



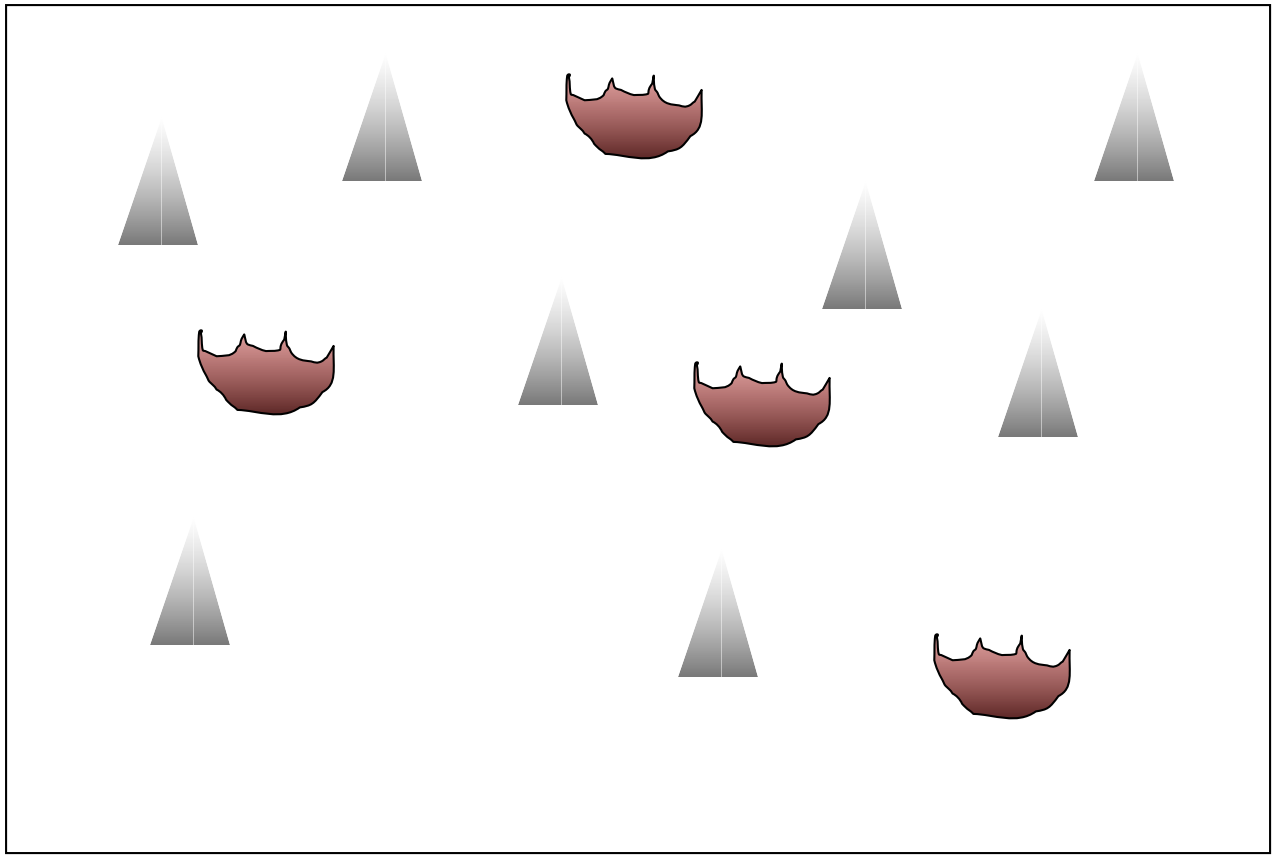
Solution Overview

- Step 3: Add flatlands to map



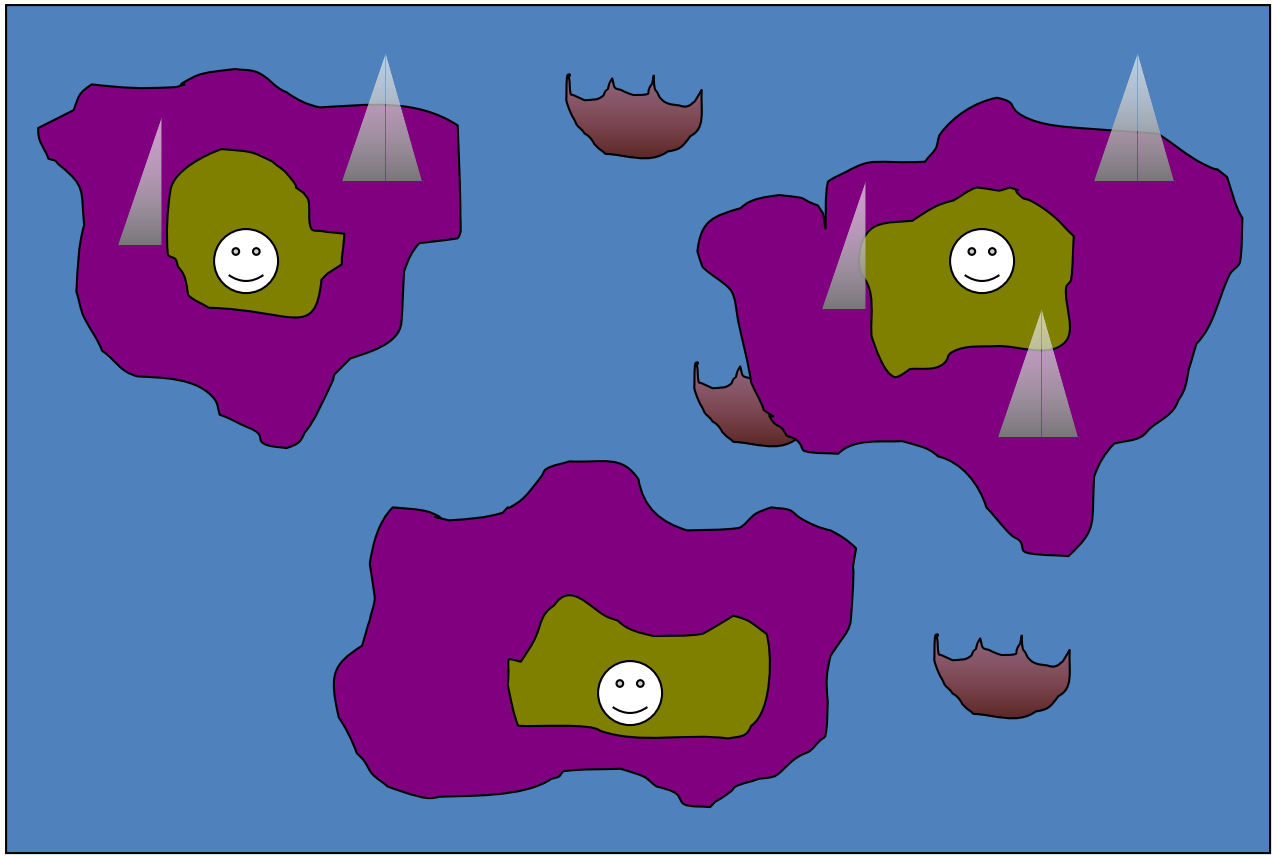
Solution Overview

- Step 4: Generate elevations map



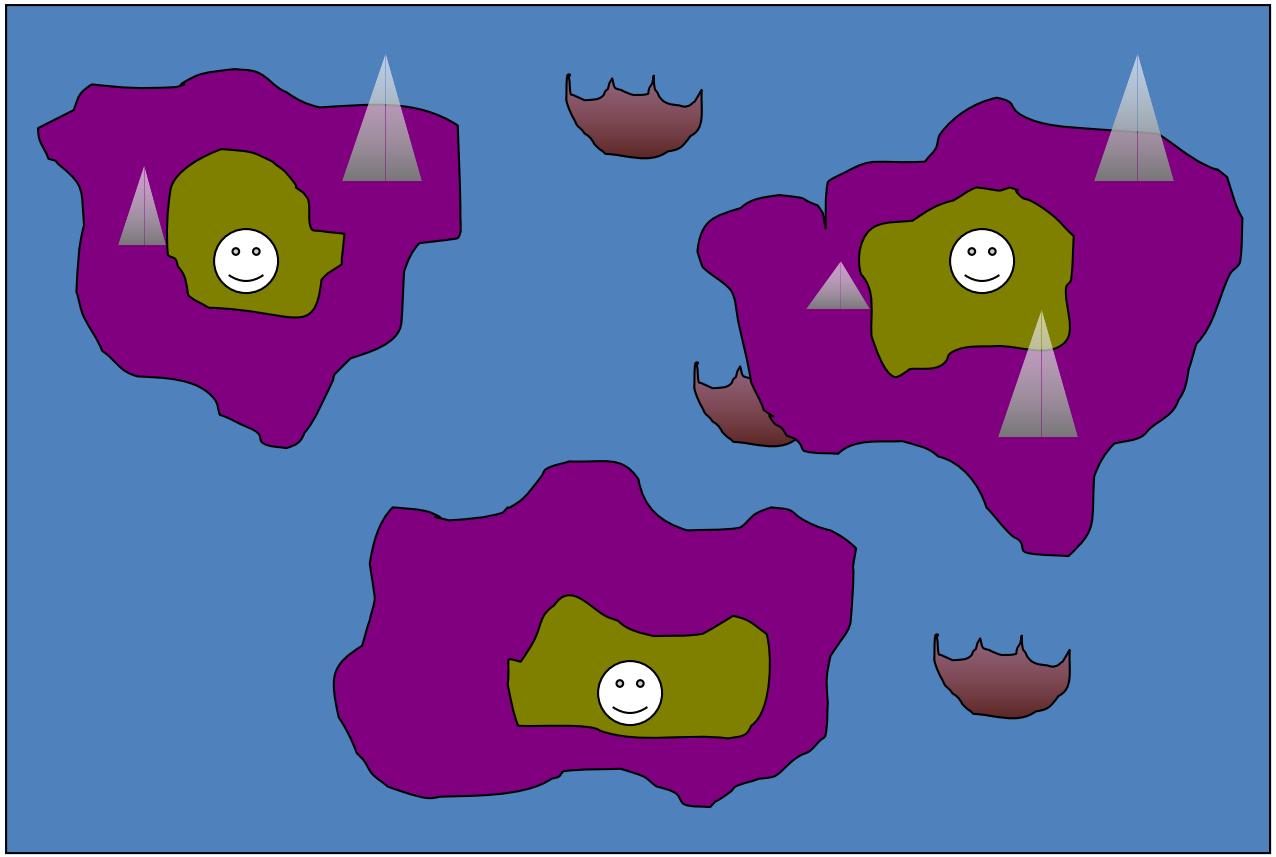
Solution Overview

- Step 5: Combine the two (obeying rules)



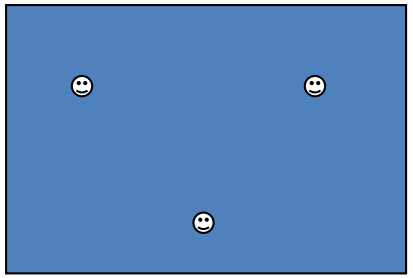
Solution Overview

- Step 6: Fix cliffs and other anomalies

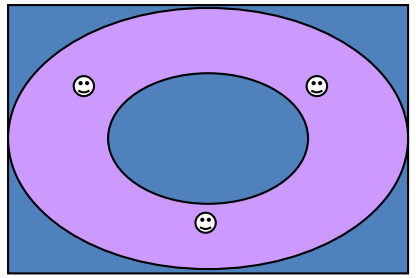


Solution Overview

- Step 7: Distribute resources to players
- Step 8: Place trees
- Step 9: Paint map with terrain textures
- Step 10: Place initial units for players
- Step 11: Place wildlife
- Steps 9 through 11 won’t be covered.



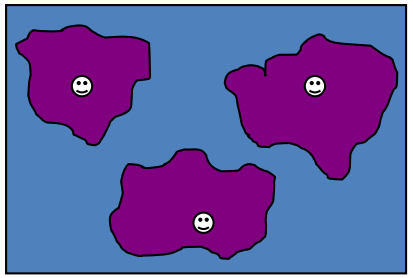
Step 1: Details



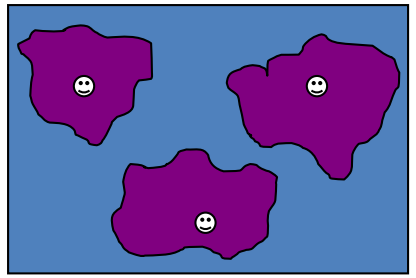
- Placement directly effects land growth
- How would you do this?
- Algorithm:
  - Inscribe one large disk onto map
  - Assign one random loc. per player w/in disk
  - Repeatedly push closest of player points apart until distribution sufficiently optimal
    - “optimal” is a map attribute
    - Specifies how evenly players must be placed

Step 1: Results

- Predictable player locations. Desirable?
  - Keeps maps balanced
  - Tunable through map attributes
- Assign player to points *after* placed. Why?
  - Teams adjacent to each other
  - Enemies somewhat symmetrically opposed
- What about colonization? “Dummy” players…
  - Placed after real players, on in/outside of disk. Why?
  - Dummies’ land grown after real players. Why?
  - Give the dummies resources too.



Step 2: Details



- Land is a resource in RTS games!
  - Allocate fairly
  - Must appear “natural”
- Clumps, modifiable script attributes
  - clumpSize
  - clumpNumber
  - Chaos level
- World starts as H_2O
- What’s a clump?

Step 2: Clumps

- Grow until
  - size == clumpsize
    - Create new clump some dist away at random, chaotic angle
  - Can’t grow any larger
    - Make new clump of *limited* size
- Continue until number == clumpNumber
- Two growth methods, step & completion

Step Clump Growth

- 1st tile is players starting location
- Choose one *valid* random edge per iteration

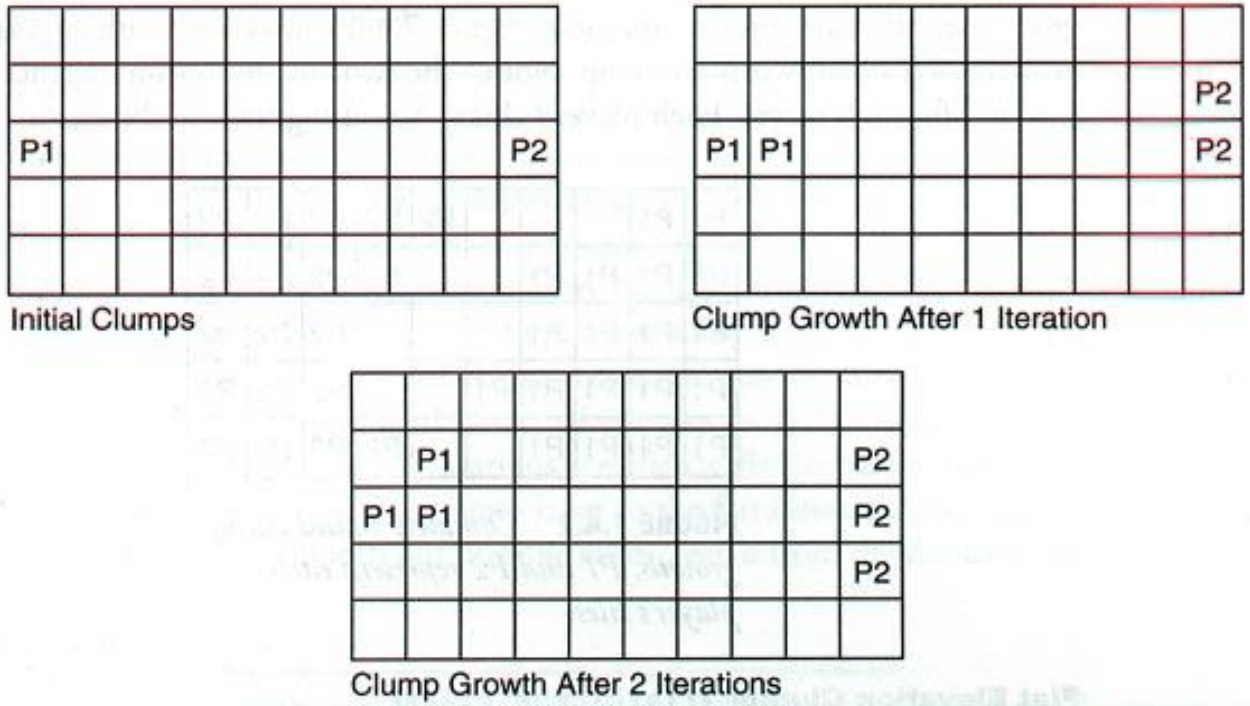


FIGURE 7.4.1 *Clump growth using the step method. P1 and P2 represent each player's tiles.*

Valid Clumps

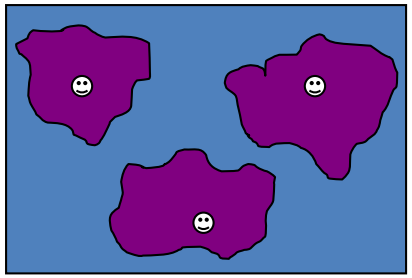
- Different types have different rules
  - Don't take owned land
  - Don't make peninsulas
- Island Clumps:

P1	P1				P2	P2	P2	P2	P2
P1	P1	P1	P1			P2	P2	P2	P2
P1	P1	P1	P1				P2	P2	P2
P1	P1	P1	P1	P1			P2	P2	P2
P1	P1	P1	P1			P2	P2	P2	P2

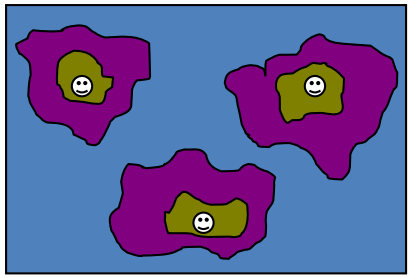
FIGURE 7.4.2 *Complete island clump growth. P1 and P2 represent each player's tiles.*

Step Clump Discussion

- Why is stepwise growth desirable?
  - Fair
    - each player has equal chance to grow
  - Preventative
    - Hinders one's land "surrounding" another's
    - Avoids "starvation"
  - Opportune
    - Makes good on the importance of placing players!
    - Remember the dummies!



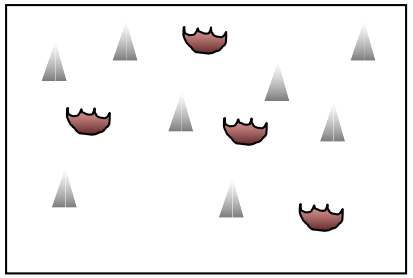
Completion Clumps



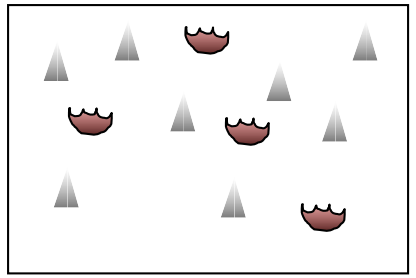
- Why bother, steps are awesome?!
- How about step 3?

P1	P1				P2	P2	P2	P2	P2
P1	P1	P1	P1			P2	P2	F	F
F	F	F	P1				P2	F	F
P1	F	F	P1	P1			P2	F	P2
P1	P1	P1	P1			P2	P2	P2	P2

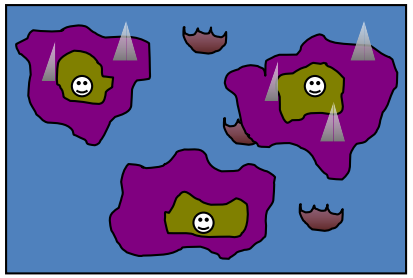
FIGURE 7.4.3 *Complete flat clump growth. P1 and P2 represent each player's tiles. F represents a flat tile.*



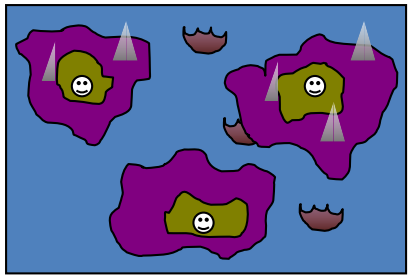
Step 4: Details



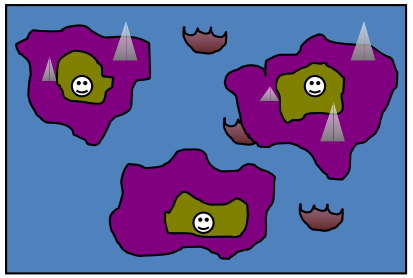
- 2D height map array
- Fractals (esp. diamond sq.)
  - Quick & easy
  - Realistic looking elevation
- Once again, map attributes
  - Min, max, initial elevation
- What about “noise”?
  - Filter used to smooth out jagged artifacts



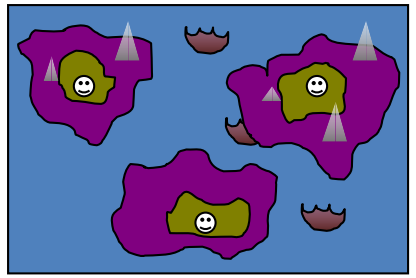
Step 5: Details



- Remember "obeying rules"?
- Algorithm:
  - if(landNH2O[i][j] == FLAT) ignore heightMap
  - if(landNH2O[i][j] == LAND &&
    height[i][j] == WATER)
    - Raise value in height[i][j] to preserve natural look
  - if(landNH20[i][j] == WATER &&
    height[i][j] == LAND)
    - Lower height[i][j] to landNH2O[i][j] level
- Why not negate in the last two cases?



Step 6: Details



- We’ve got anomalies again!
- Wouldn’t negating have worked?
- Re-apply filter

Step 7: Resource Allocation

- Each player gets same amt. of each w.r.t. other players
- Randomly placed within rings centered on players' starting locs.
- Validation
  - Some resources can't be placed on terrain features
  - Resources restricted in min. distance to other resources
  - Resources restricted in min. dist to start loc
- Once again, map attributes
  - Amount of each resource
  - Size of ring

Step 8: Place trees

- Plentiful; Restrict pathfinding
- Can the clump class be re-used?
- Map attributes
  - Number forests/player
  - Size of forests
  - Number of clumps/forest
- Some areas of map marked as forest free
- Player-owned land does not restrict growth

Scripts

- Language to manipulate map attribs
- Basic text parser and data structures
- Things like:
  - #resources per player
  - Max altitude in height map
  - Player land allocation
  - Climate (terrain/forest/animal texture set)
- Small changes yield map variations

Scripts: Pros

- Allow for incremental map type devel.
  - New attribs for new maps
  - Add attribs to parser
  - Matching code added to map generator
  - Offload some tasks from programmers to designers
  - Modability/extensibility/variety
- Next steps…
  - Island map w/teams: allow land for team-members to merge
  - Tweak elevations: island maps become canyon maps
  - Tweak clump size: island maps become river maps

Scripts: Cons

- Unfortunately:
  - One master script per type of random map
  - One script per size of type of map
  - In each script, subsections for each qty. of players (# berries? # gold?)
- You can imagine:
  - Scripts can become hard to manage
  - Manuals needed to explain types & attribs
  - Additional tools to edit attribs across files
  - Designers are not programmers
    - Programmers must create map “templates”
    - Designers simply “tweak”

Buyer Beware!

- Can’t explicitly place particular terrain features
  - “super-tile” design approach
  - Fairer/more visually appealing
- Designers are not programmers
- Chokepoints are possible & unfair
- Resource placement can still yield advantages