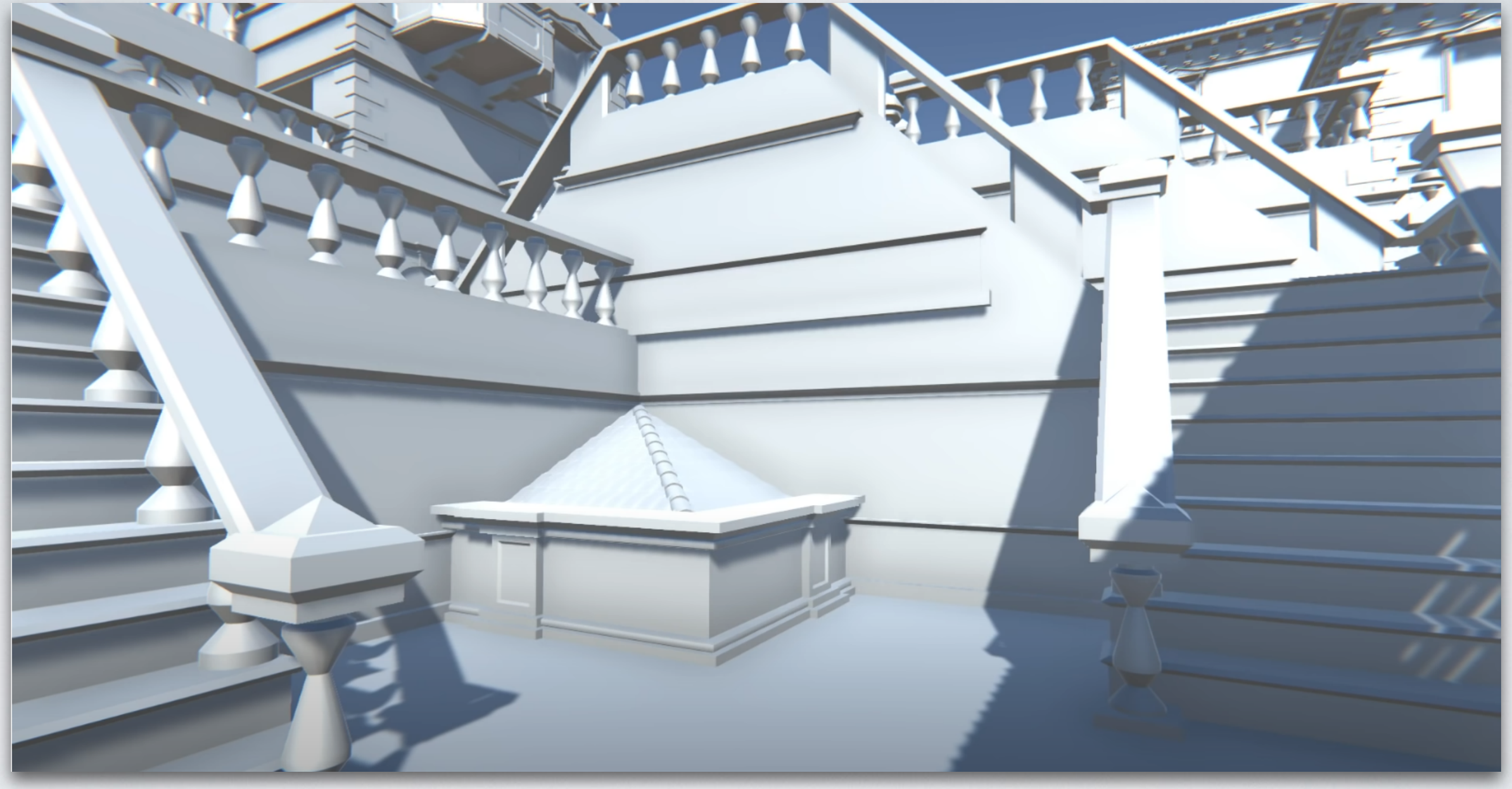
LEVEL GENERATION

Techniques for generating environments for games

Marian Kleineberg (source)

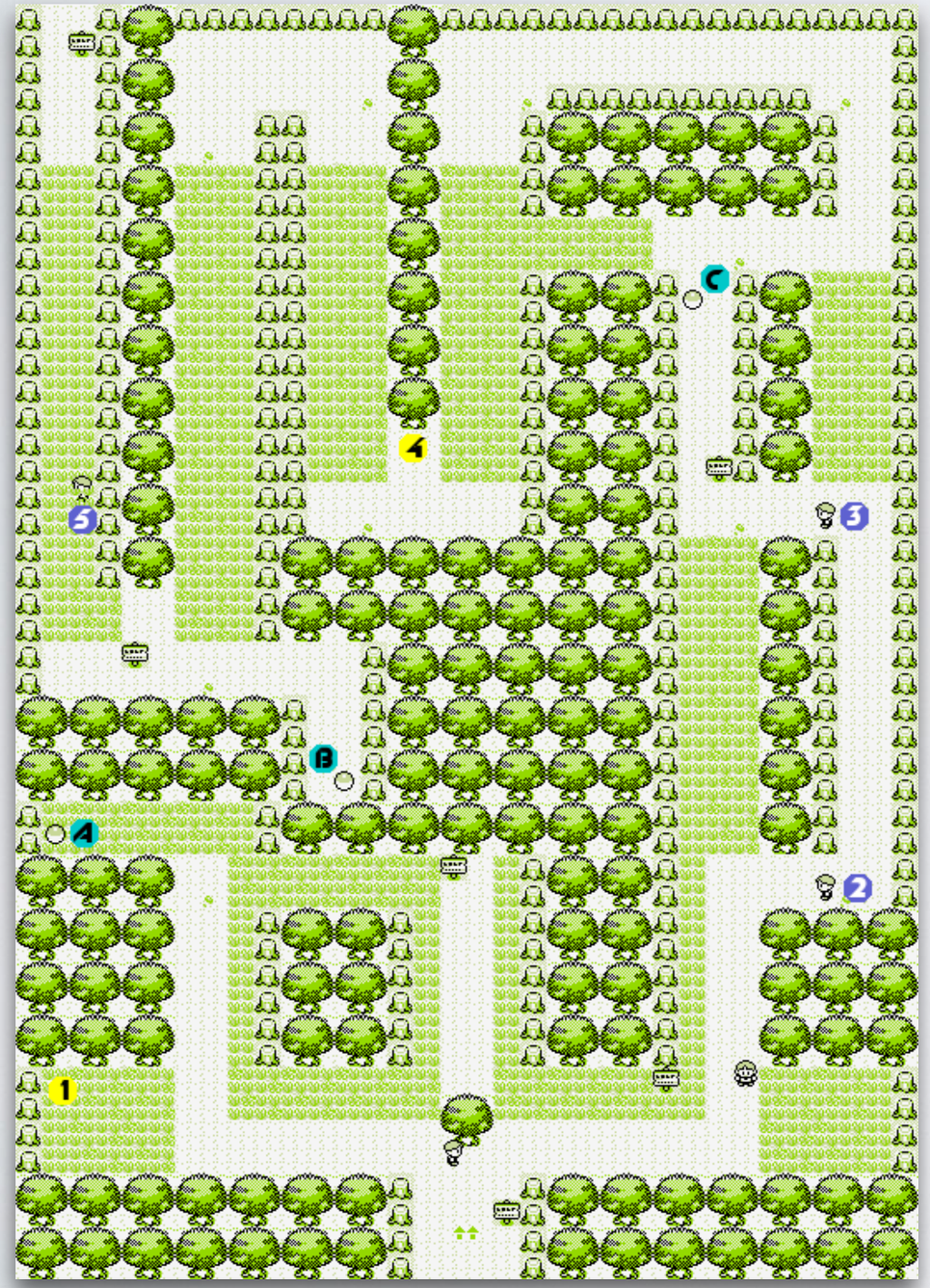
HOW TO MAKE THIS?



Marian Kleineberg (source)

Video Here

EXPLORE FOREVER



Pokemon Green via strategywiki (source)

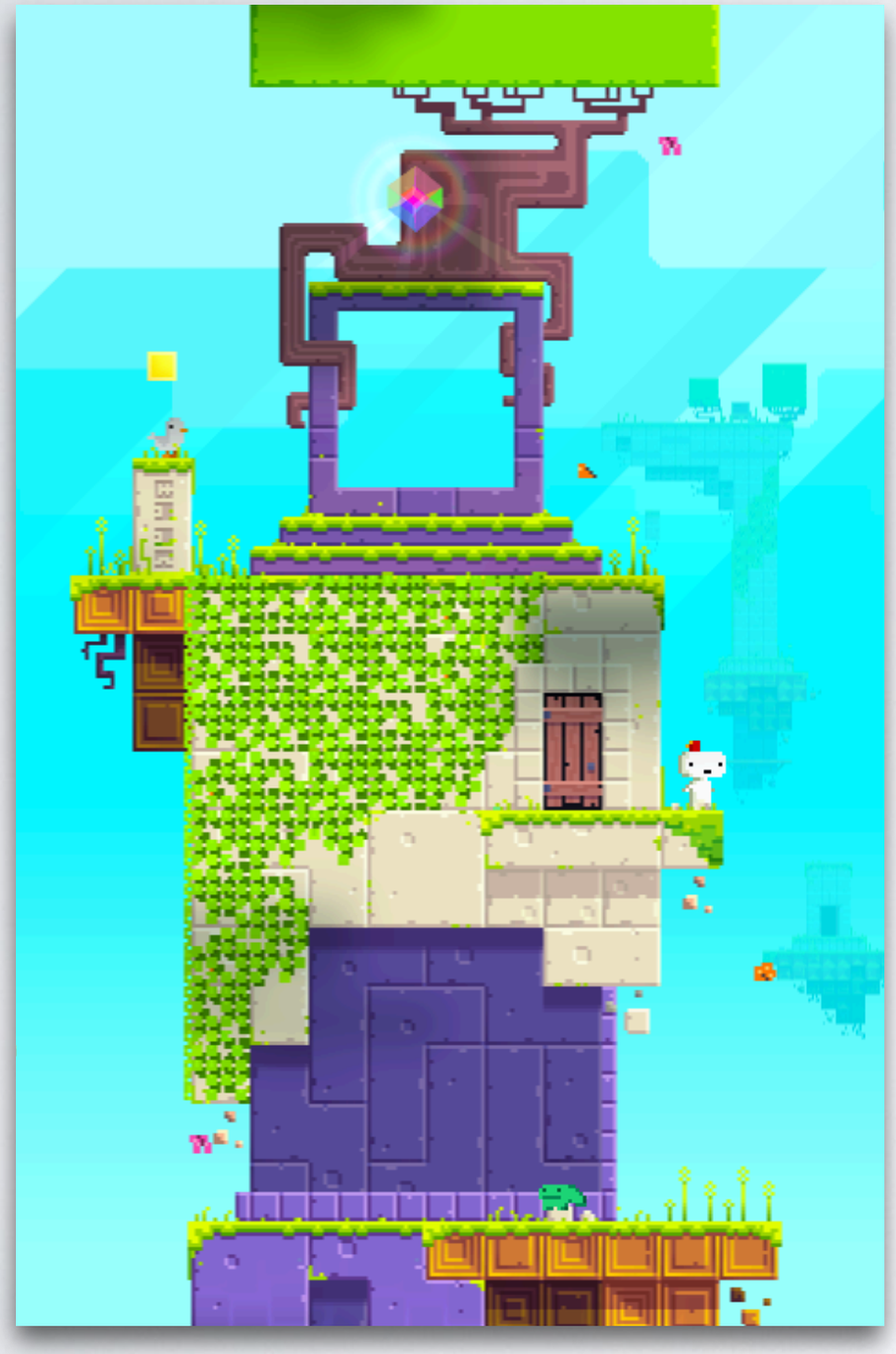
- Many games include structured environments, where a player must explore, navigate, find resources and avoid hazards. Bread and butter for many games, especially RPGs!

- But once a player knows a static map, there's no motivation to explore — this limits the replay value.

- Of course, here's where procedural generation comes in. Come up with good generation rules, and you have infinite levels! Explore forever!

- Rich history of procedural game levels: Rogue, Dwarf Fortress, Diablo, Infinite Mario Bros (common AI contest base)

LEVEL INGREDIENTS



Fez via PS4Home (source)

- This is a little different from our discussion of other content generation. Now we care about the player experience, not just the visuals
- Now, don't limit your imagination, but canonically, games of this type often contain:
  - **Goal(s)**: endpoint(s)
  - **Resources/rewards**: things to collect
  - **Obstacles**: things that block your progress
  - **Hazards**: things to avoid! Static, or dynamic
- **Constraints:**
  - **Feasibility**: can you beat it?
  - **Fun**: do you want to beat it?
  - **Challenge**: appropriate difficulty?

BASIC APPROACH

- Designing a single good game level is a difficult design challenge. How can we abstract such a system into a generation problem?
- Feasibility first. The game must be beatable*!
    - Figure out what your player's constraints are.
        - eg. avatar can't walk through walls, jumps a fixed distance etc.
    - Then either
        - Use those same constraints directly in your generation algorithm
        - Design an evaluation method for checking a generated level
- Difficulty / interestingness are much harder issues
    - Figure out what elements affect these eg. number of path turns, enemy frequency
    - Then as always, parameter tune!

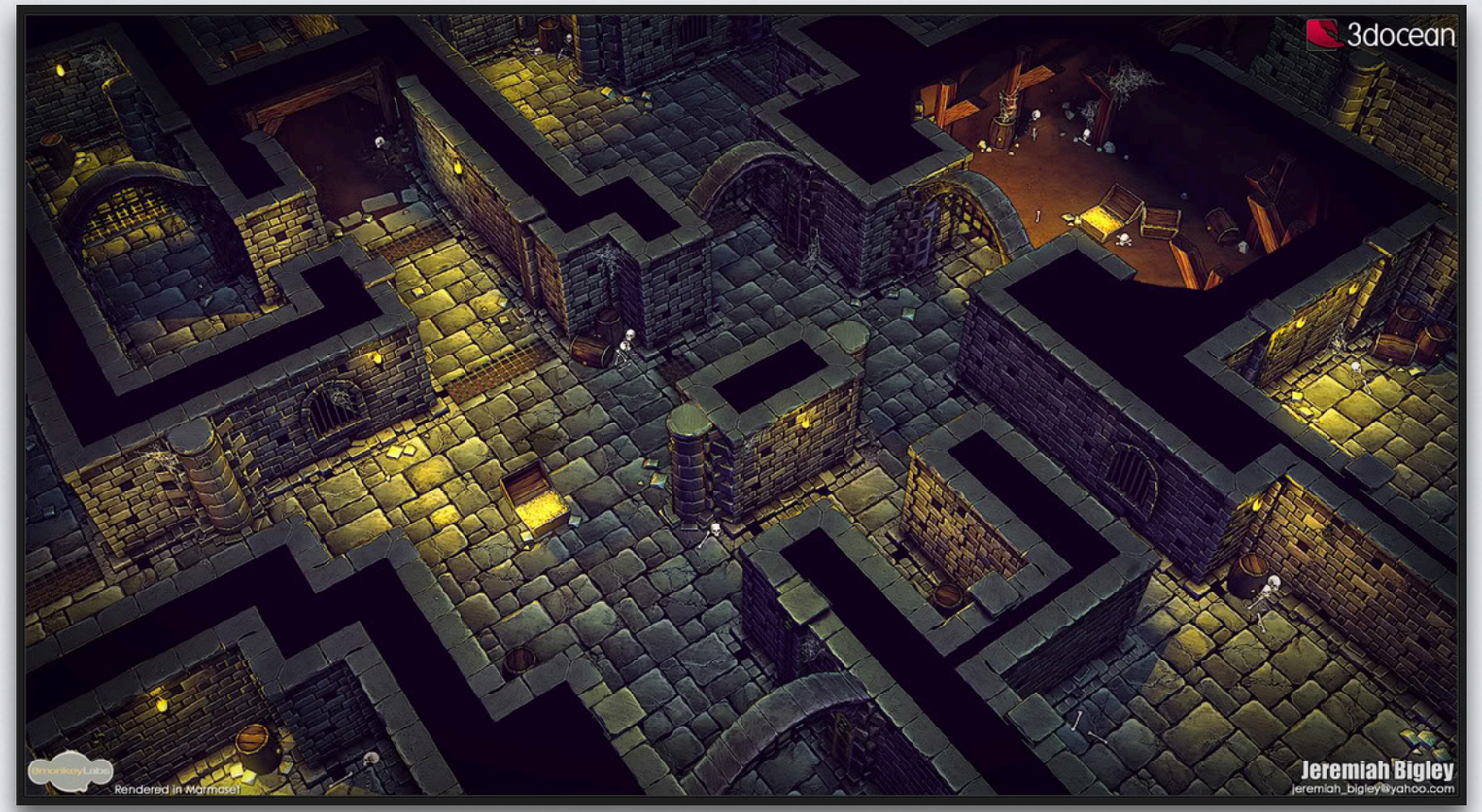
*Unless the challenge is to figure out if it's beatable?

BASIC APPROACH

- Most level generators consist of
    - A representational model: an abstract representation of a level, its elements and their positions/parameters
    - A method for generating that representational model
    - A method for converting the model into actual geometry for the level

From pcgbook (source)

DUNGEONS

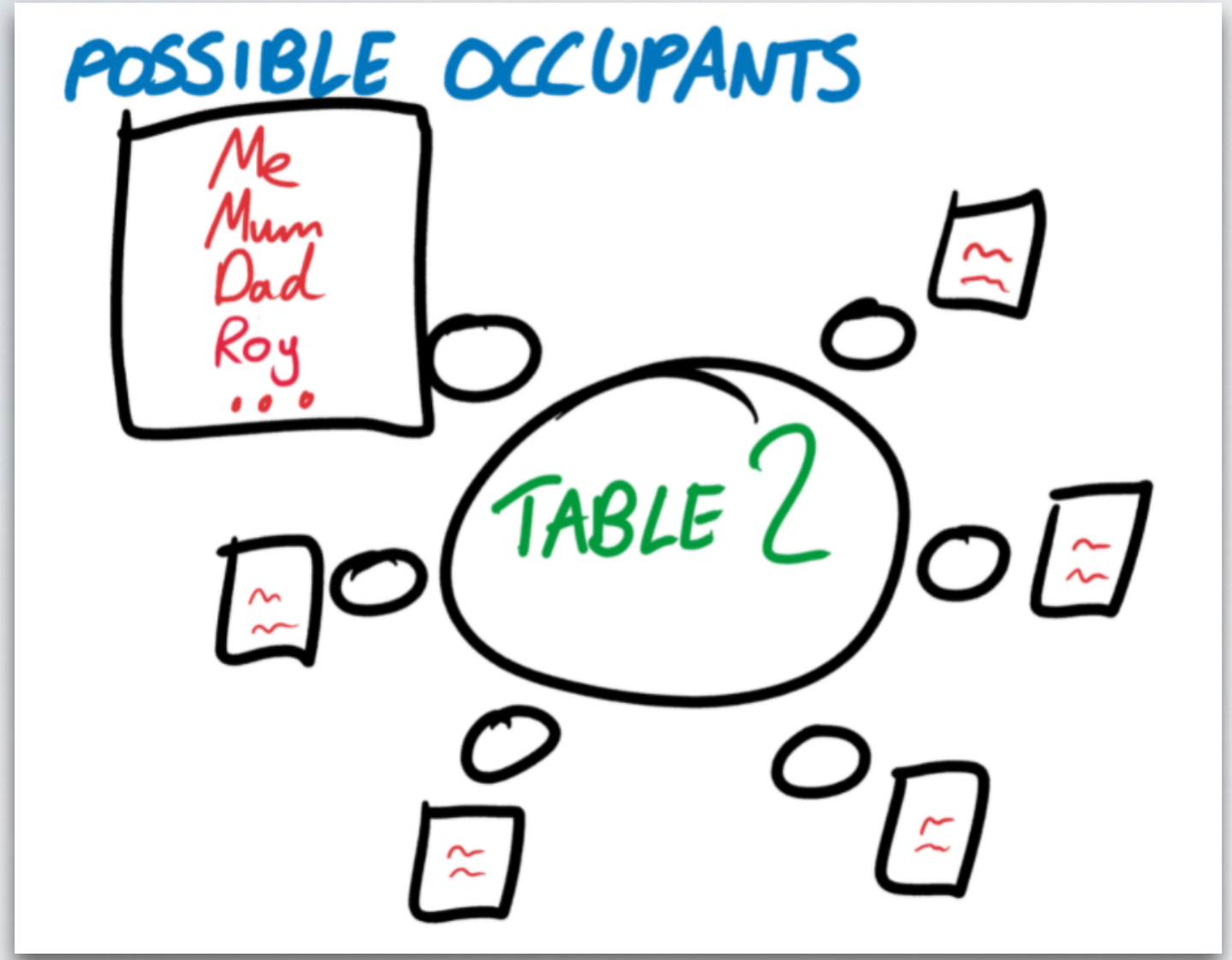


Jeremiah Bigley (source)

- There is infinite variety in games. For simplicity, let's limit our discussion to dungeons, although included methods apply to platformers, etc.
    - Complex 2D/3D environment defined by navigable/impassable space
    - Spaces can be corridors, rooms of different types
    - Scattered objects, often distribution relates to room type

GENERATION METHODS

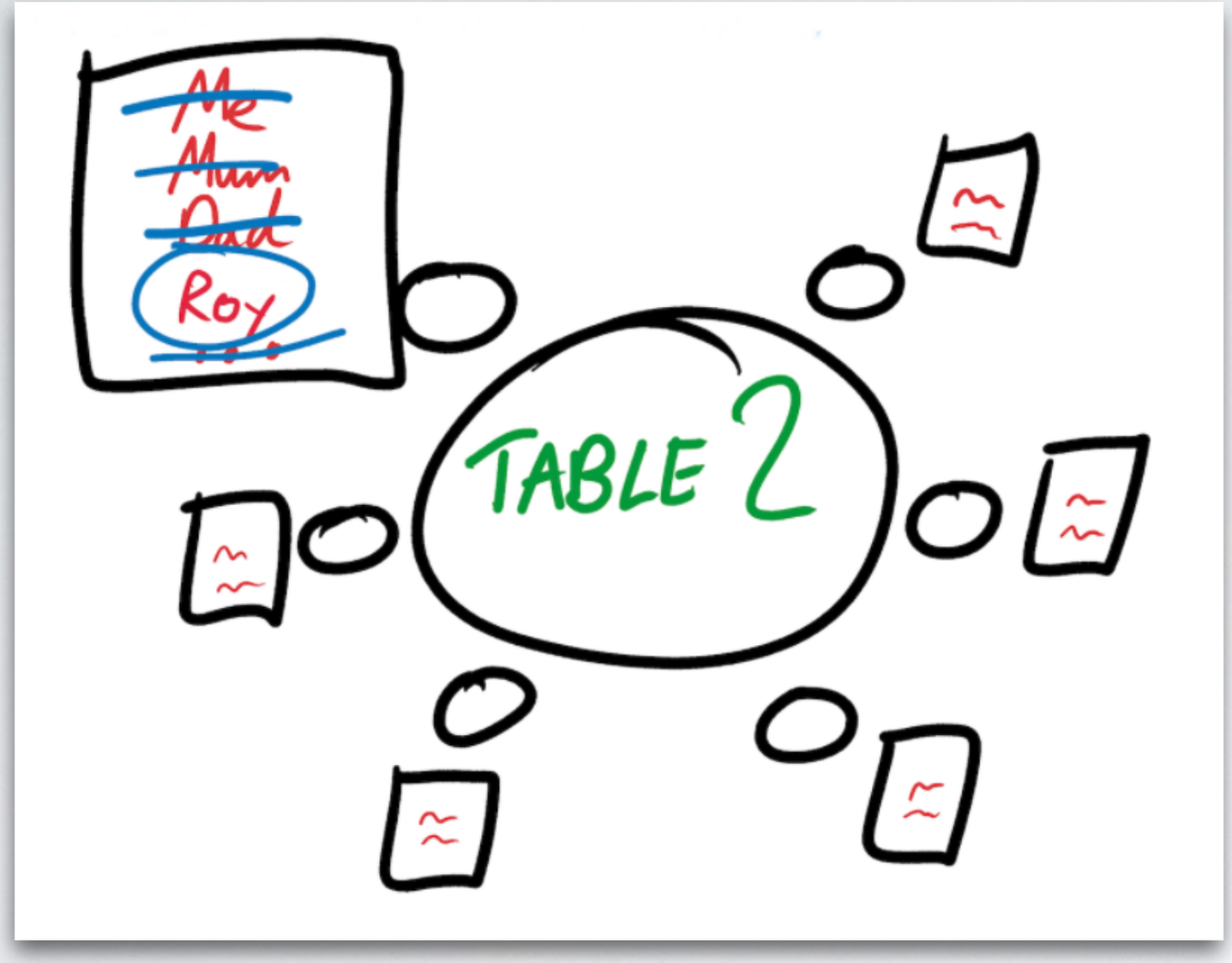
WAVE FUNCTION COLLAPSE



Robert Heaton (source)

- Imagine you have a problem like a wedding seating chart.... certain people can't sit next to each other
- How do you approach this problem?

WAVE FUNCTION COLLAPSE



Robert Heaton (source)

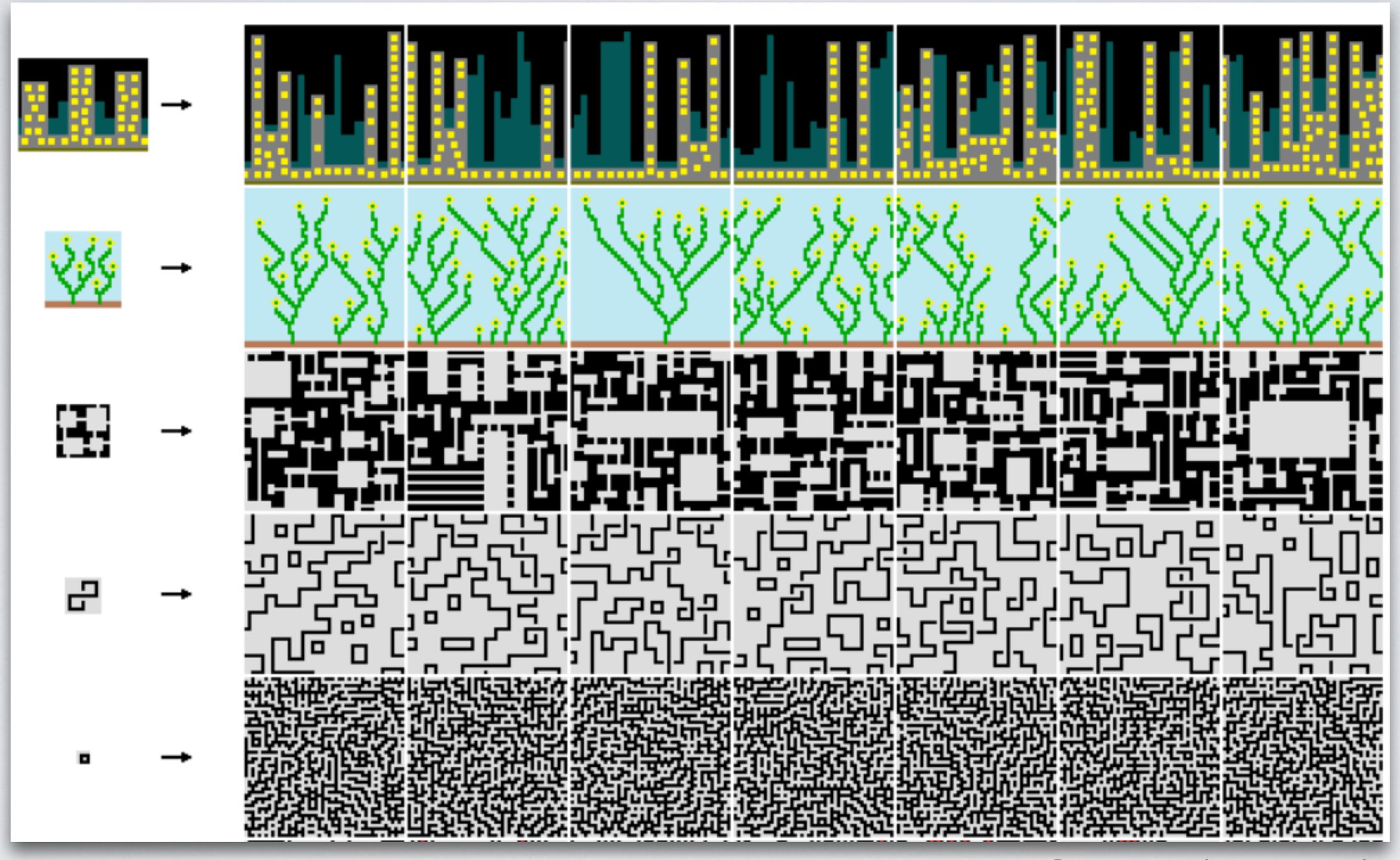
- Fix one person in place
- Then try one of the next viable candidates
- If you get stuck, backtrack

IT'S A LOT LIKE...

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

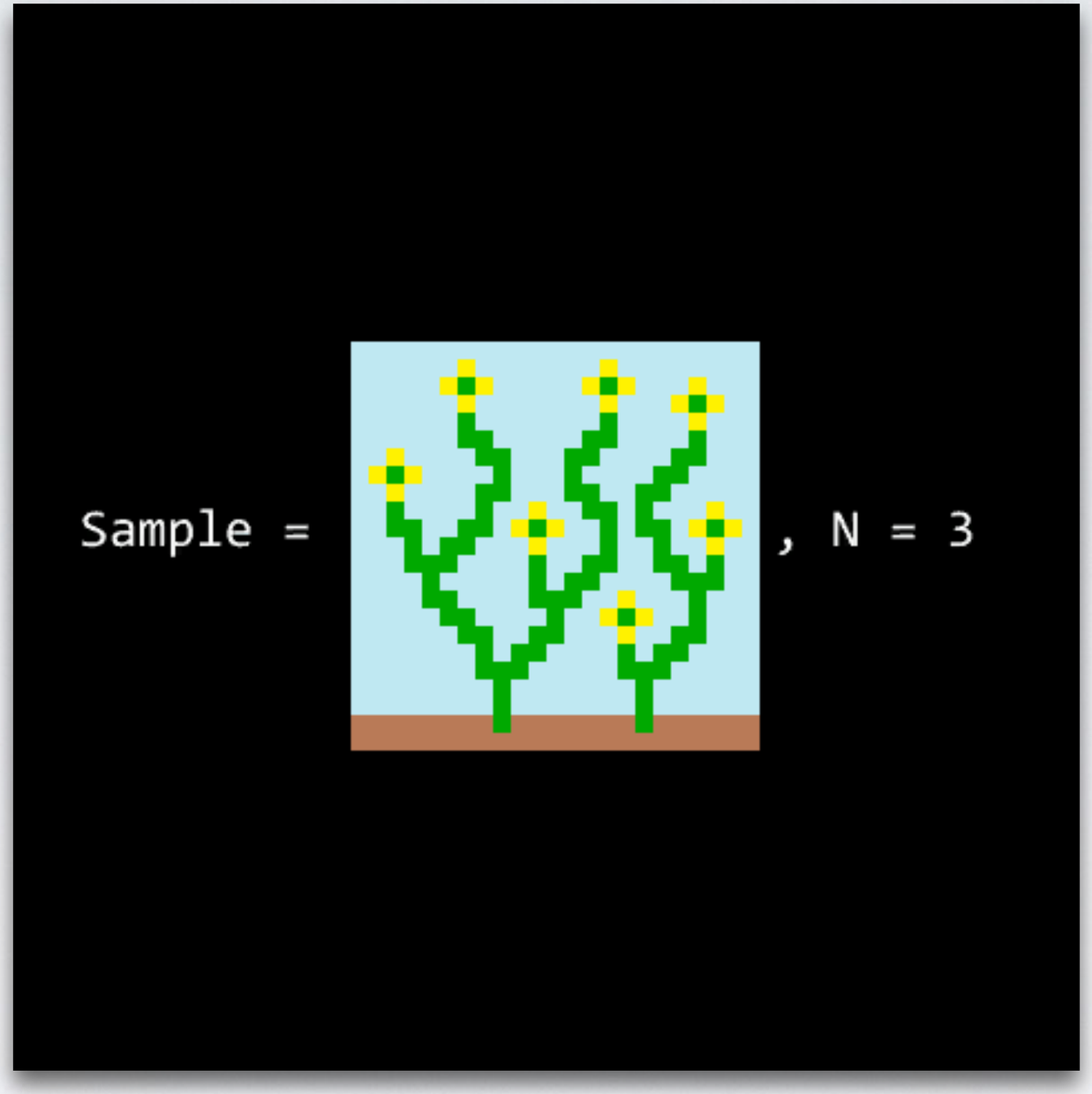
5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

RULES FROM EXAMPLES



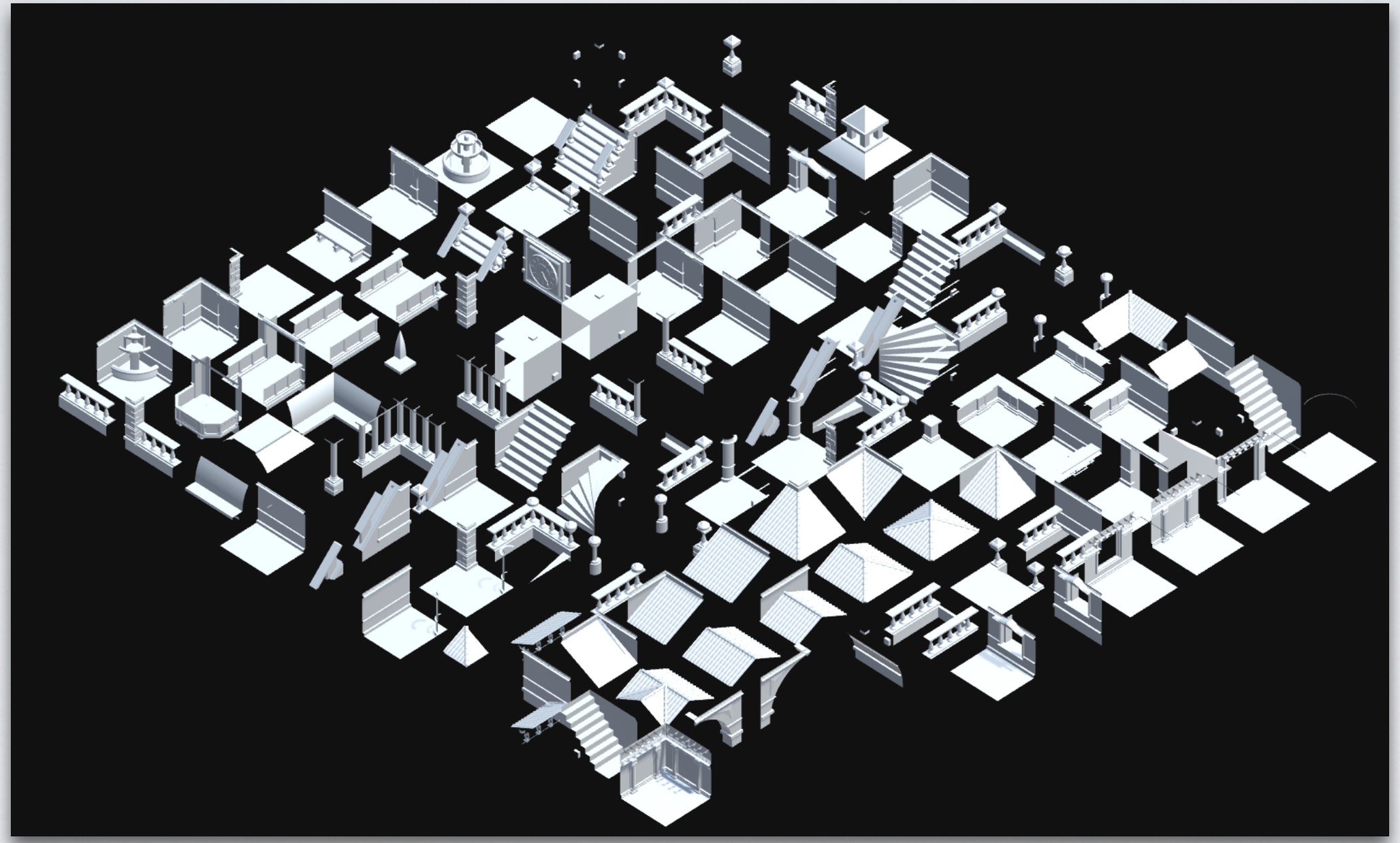
Maxim Gumin(source)

RULES FROM EXAMPLES



Maxim Gumin (source)

WAVE FUNCTION COLLAPSE



Marian Kleineberg (source)

WAVE FUNCTION COLLAPSE

- Read the input patterns (some NxN)

- Initialize an array of output dimension (a wave). Each element of that array should contain a state representing which of the possible patterns it contains (bool coefficients, which can be true or false)

Loop:

Find a wave element with the minimal nonzero entropy. If there are none, break

{

- Collapse this element into a definite state according to its coefficients and the distribution possible patterns from the input

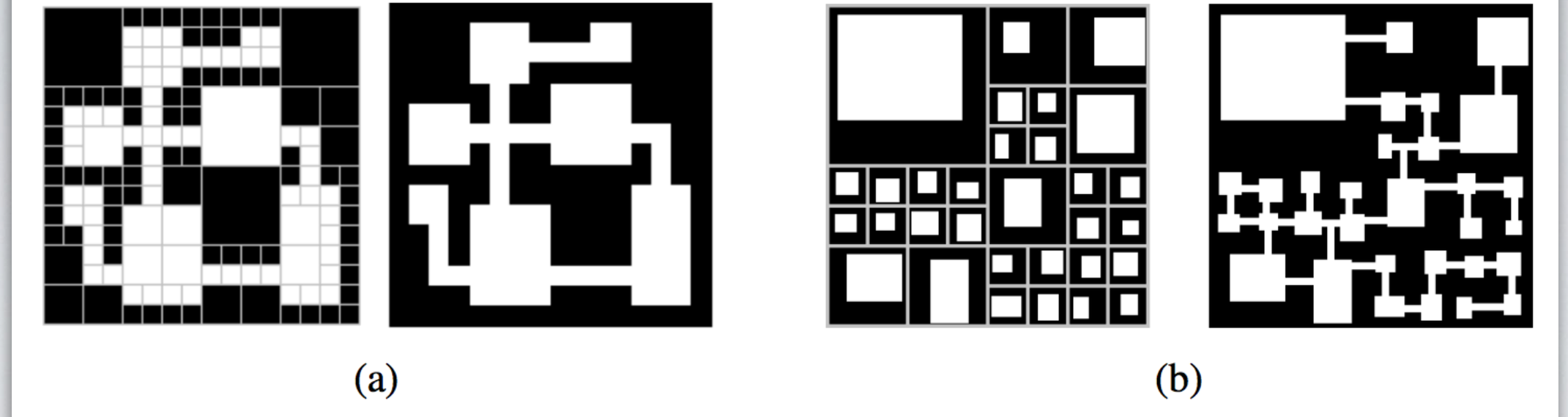
- Propagation: propagate information gained on the previous observation step.

}

By now all the wave elements are either in a completely observed state (all the coefficients except one being zero) or in the contradictory state (all the coefficients being zero). In the first case return the output. In the second case, back up some number or steps and try again (or give up, with incomplete output)

Marian Kleineberg (source)

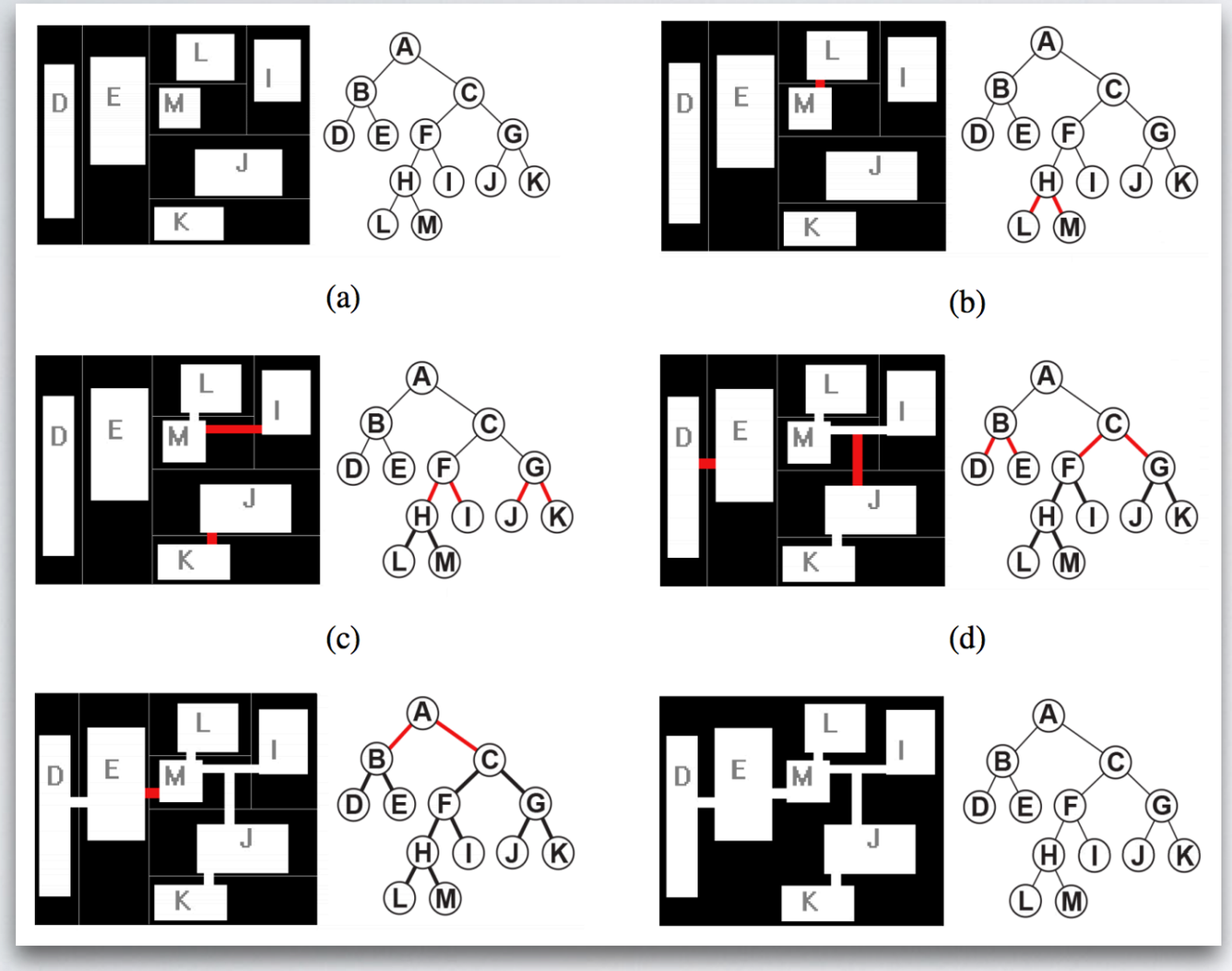
SPACE PARTITIONING



pcgbook (source)

- To carve out rooms, we can subdivide our space recursively using a Binary Space Partitioning (BSP) Tree, like a quad tree or oct tree. Nice, since we can easily avoid room overlap! This is a *macro* approach. Global knowledge of space.
    - At each tree node, randomly partition into further splits or stop and become a leaf
    - For each leaf node, create a room (the whole cell (a), or a subset of it (b))
    - As a post-process, we can join rooms with corridors. How?

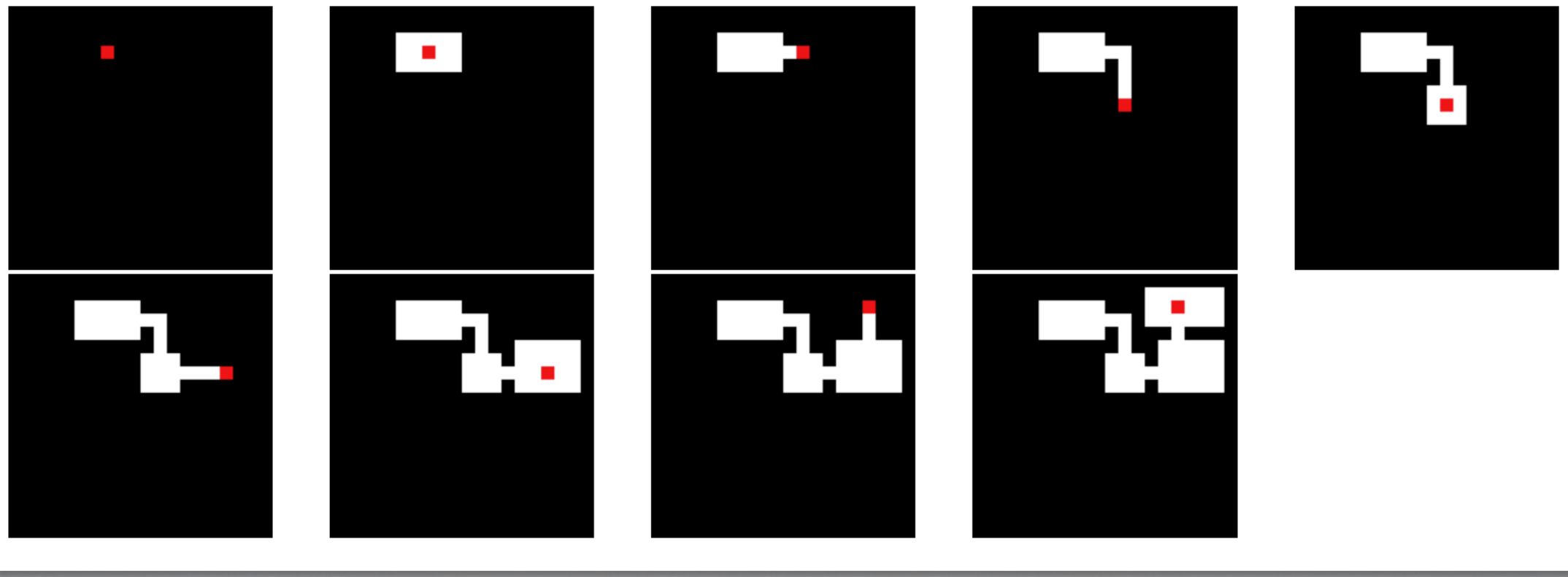
SPACE PARTITIONING



pcgbook (source)

- The BSP tree can organize our room joining process:
  - Join all children of a single node via corridors
  - Doing this recursively ensures a fully-connected tree.
- The tree can also help us categorize rooms. Each level of the tree can define an abstraction, since they're nicely grouped/connected
- Eg. this node is a prison, its children are jail cells.

AGENT-BASED



pcgbook (source)

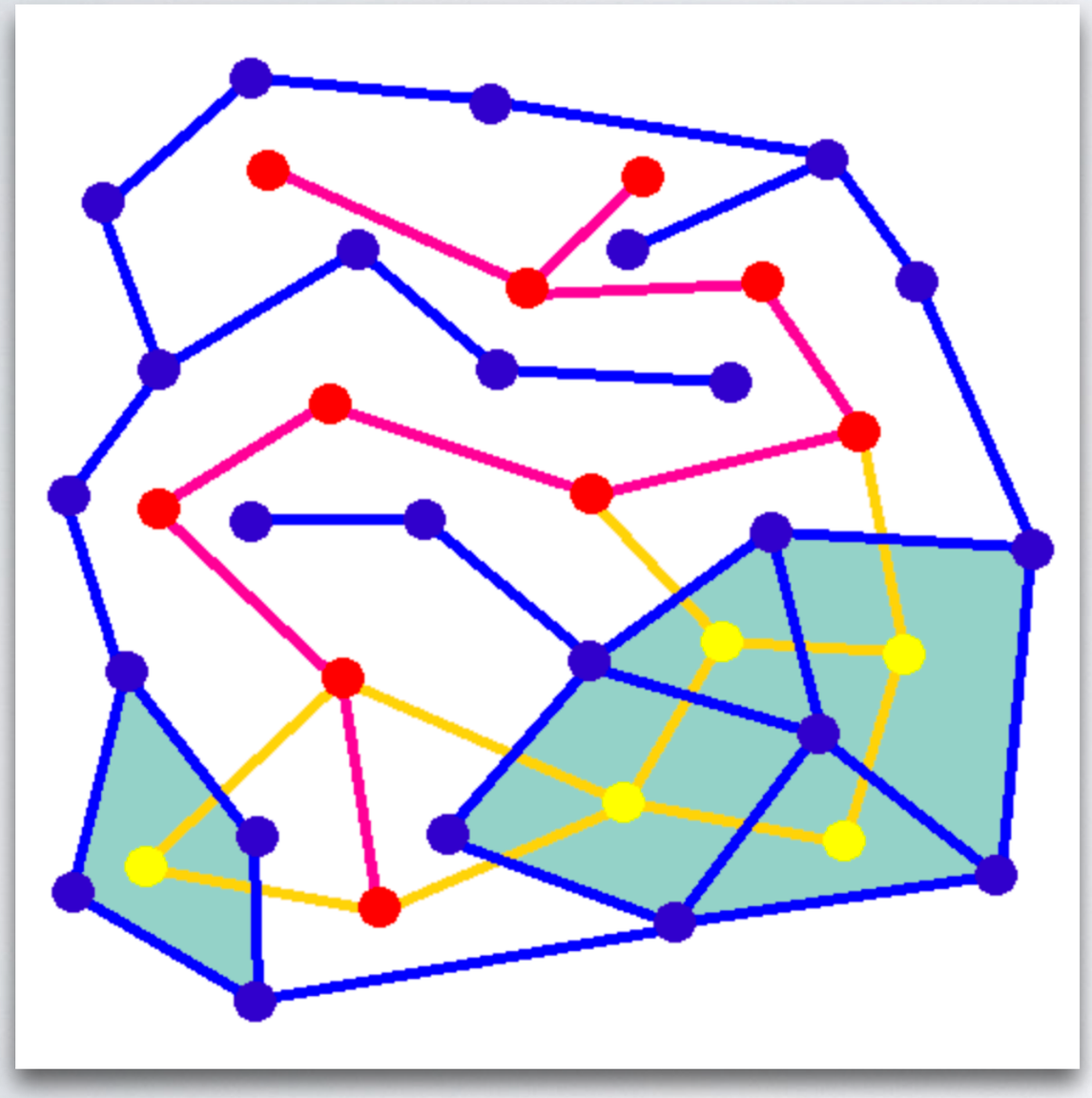
- We can use an agent to "dig" paths through a solid space. Define an agent's movement. Add some random component. Use its trail as paths
- A micro approach. No global knowledge — more chaotic.
- Paths depend on defined agent behavior. Could be like a random walker or could have some look ahead checks to avoid intersecting paths or overlapping rooms.

AGENT-BASED

- Initialize an agent at some position
- At each step:
    - Rotate (or not) a random direction
    - Walk forward some amount (mark your path as a corridor).
        - Optional: Check if corridor would intersect existing structures. Cancel if so
    - Drop a room of some dimension
        - Optional: Check if room would intersect existing structures. Cancel if so
    - Check for a termination condition
        - Distance traveled? No viable paths forward? Edge of map? Random termination?
- Other ideas: multiple agents, agents that spawn agents, attractors/repulsers… etc!

GRAPH-BASED

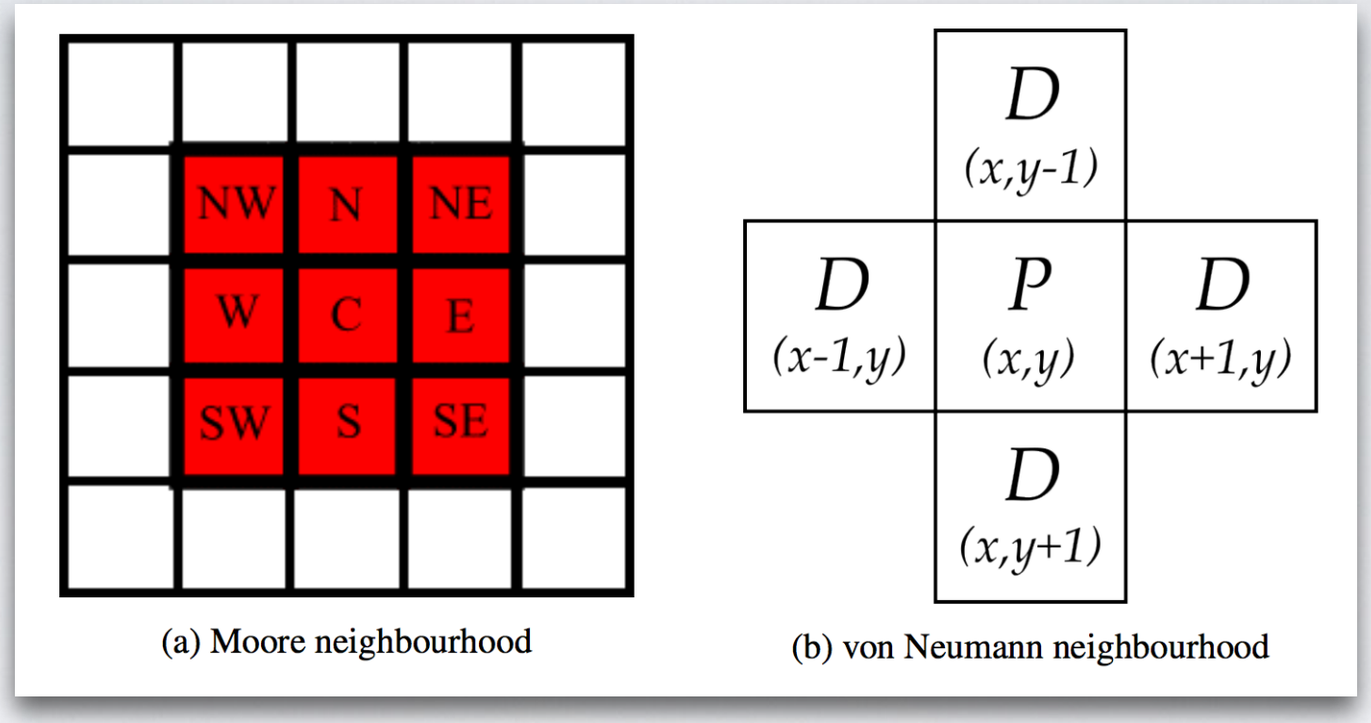
See cool animations



nBeverley (source)

- We can divide our space into cells (on a grid or not! Voronoi cells also work), surrounded by walls, then treat each cell as a node in a graph.
- Then create edges between nodes, removing the intersecting walls when we add an edge.
- Many possible applications of classic graph algorithms!
- Want every cell to be accessible?
    - Find a spanning tree!
- Want to connect two specific points?
    - Use a path-finding algorithm!
    - The graph traversal of say, depth-first search can create an interesting maze!
- Various algorithms guarantee cyclic/acyclic
- Can also place rooms first and plan around them.

CELLULAR AUTOMATA



(a) Moore neighbourhood

(b) von Neumann neighbourhood

pcgbook (source)

- Cellular automaton: a discrete computational model made of an n-dimensional grid, a set of states and transition rules.
- Each cell is in some state. Simplest: on/off, or free/full.
- Each cell is aware of its neighbors.
  - Two models: Moore and von Neumann neighborhoods. All cells use the same model.
- We simulate cellular automata for some number of time steps, starting with an initial configuration of cells
- At each time step, each cell's state is updated according to its neighbors' states.

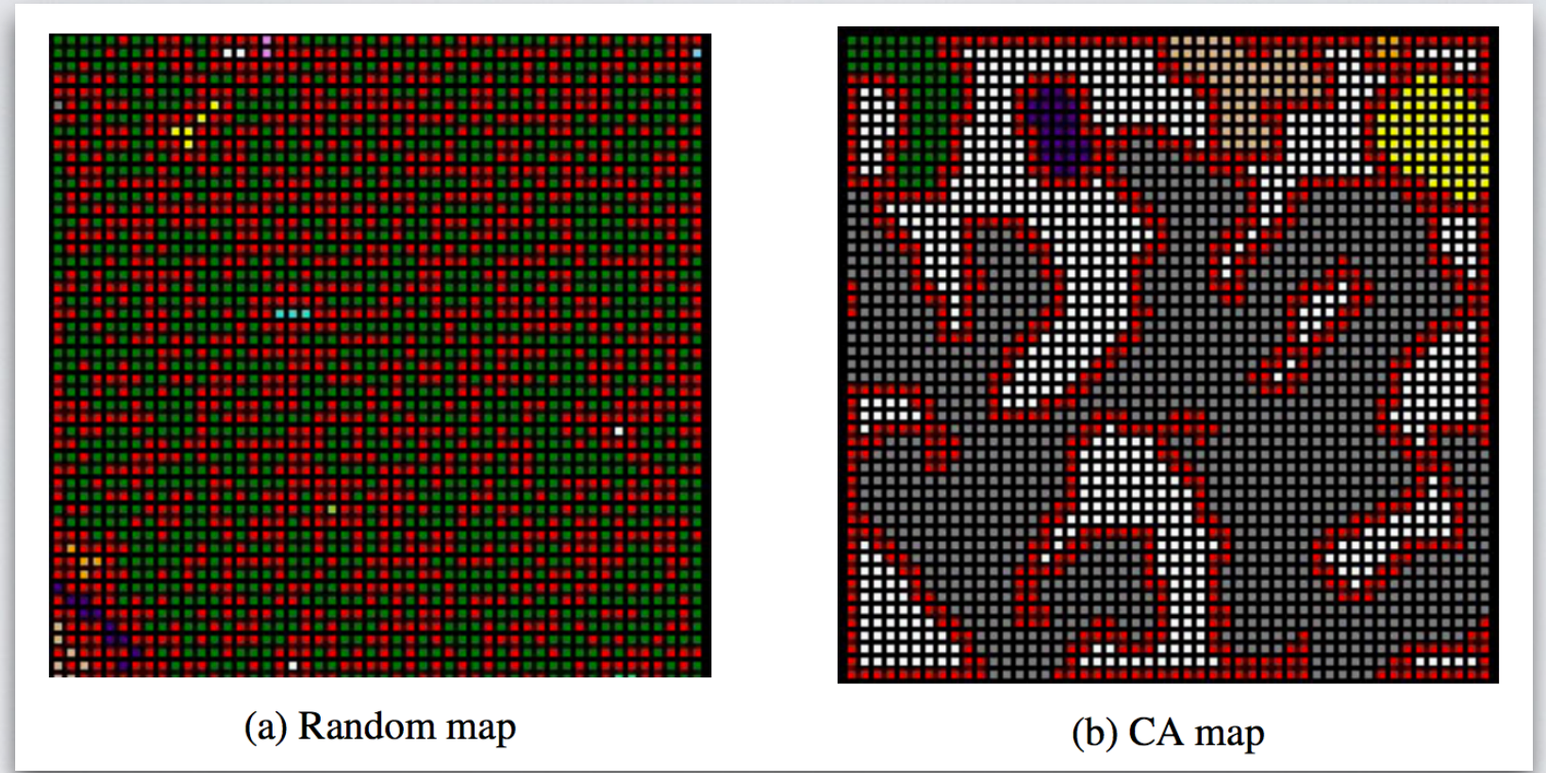
CELLULAR AUTOMATA

Conway's Game of Life (classic example)

TeamGhostID (source)

- Simple rules can create incredible complexity
    - A 2-state automaton with Moore neighborhood radius 2 has $2^{25} = 33,554,432$ possible configurations
- Conway's:
    - live cells with > 3 living neighbors die (overpopulation)
    - live cells with < 2 living neighbors die (underpopulation)
    - dead cells with 3 living neighbors become alive (reproduction)

CELLULAR AUTOMATA



(a) Random map

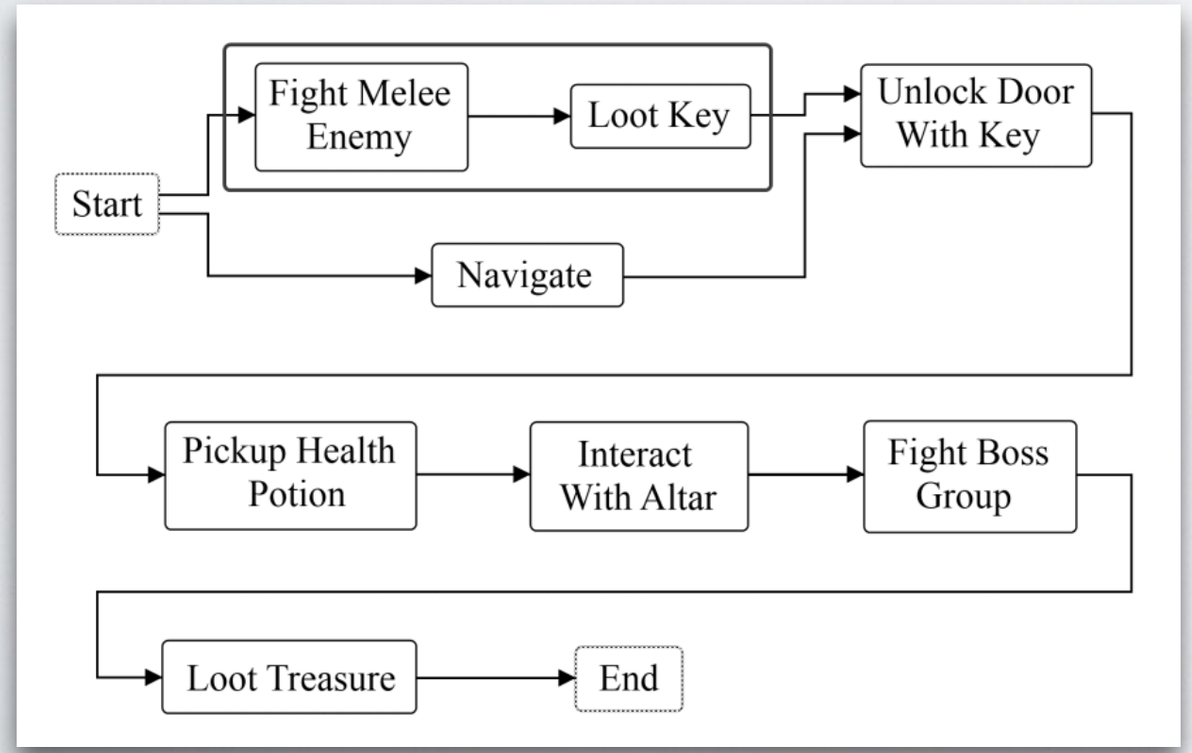
(b) CA map

pcgbook (source)

- We can use state to represent different environment elements, eg. Free space, rock, water
- Simple rules applied for few iterations can create very organic results.
  - Seed space with rock cells with some probability.
  - For n iterations, if a cell has more than 5 rock neighbors, it becomes a rock
  - Results in the above image — lots of local cohesion. Much better than random!
  - No global connectivity guarantees! But we can create tiles like this, then carve connecting paths. We can even run the algorithm again after connecting tiles to smooth.

GENERATIVE GRAMMARS

- We can of course use grammars!
- But we're not limited to specifying geometry. Unlike our previous study of grammars, we can use grammars to model a series of abstract events in our game
- We can represent a level as a connected graph of events, then use a separate process to translate that graph into actual level geometry
- Advantages: player experience driven!

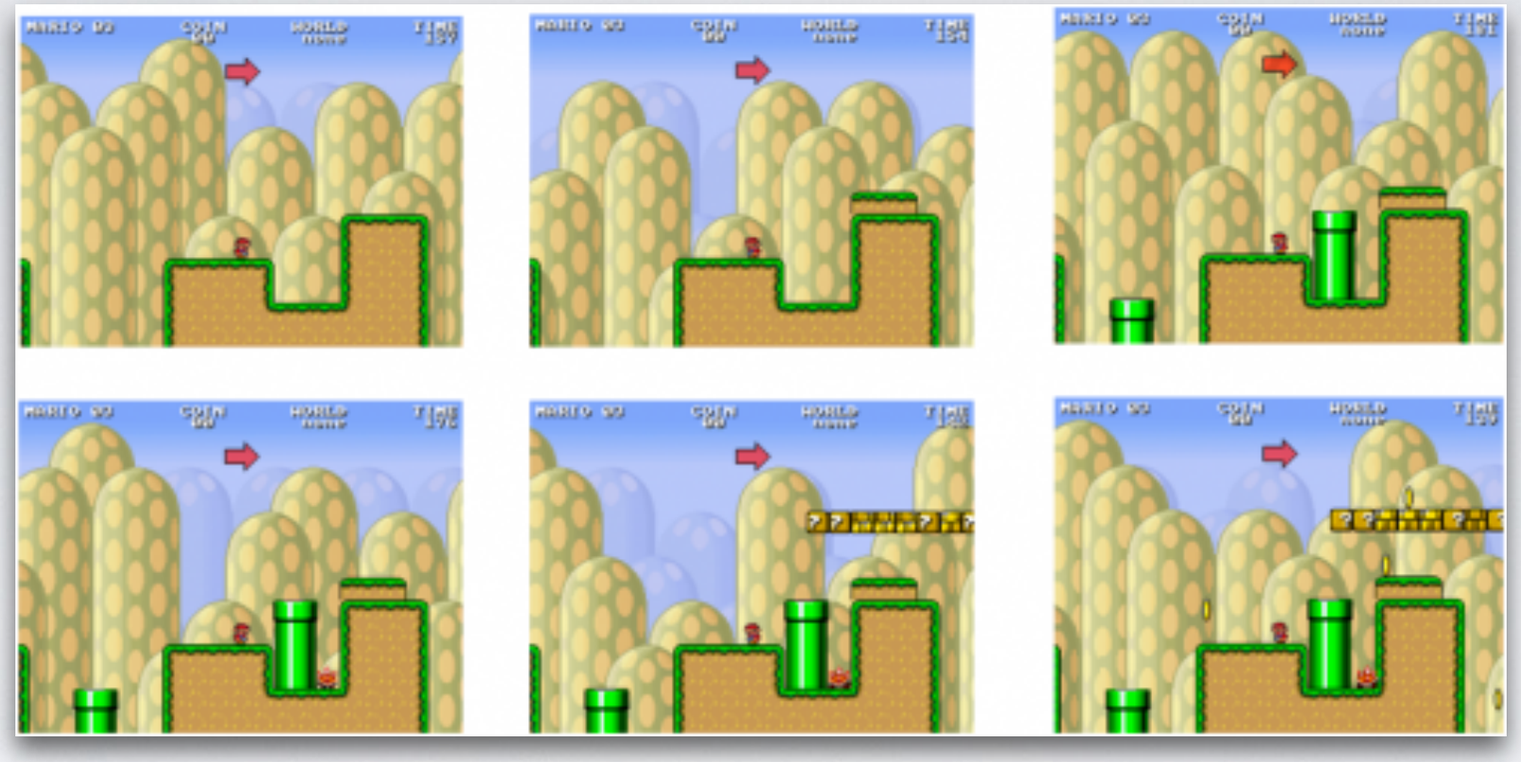


pcgbook (source)

Example system:

- Axiom: start > challenge > goal
- challenge => challenge*
- challenge = {spike pit, snake, fire, zombie, locked door}
- locked door => (key-quest > door)
- zombie => (get-weapon > combat > loot)

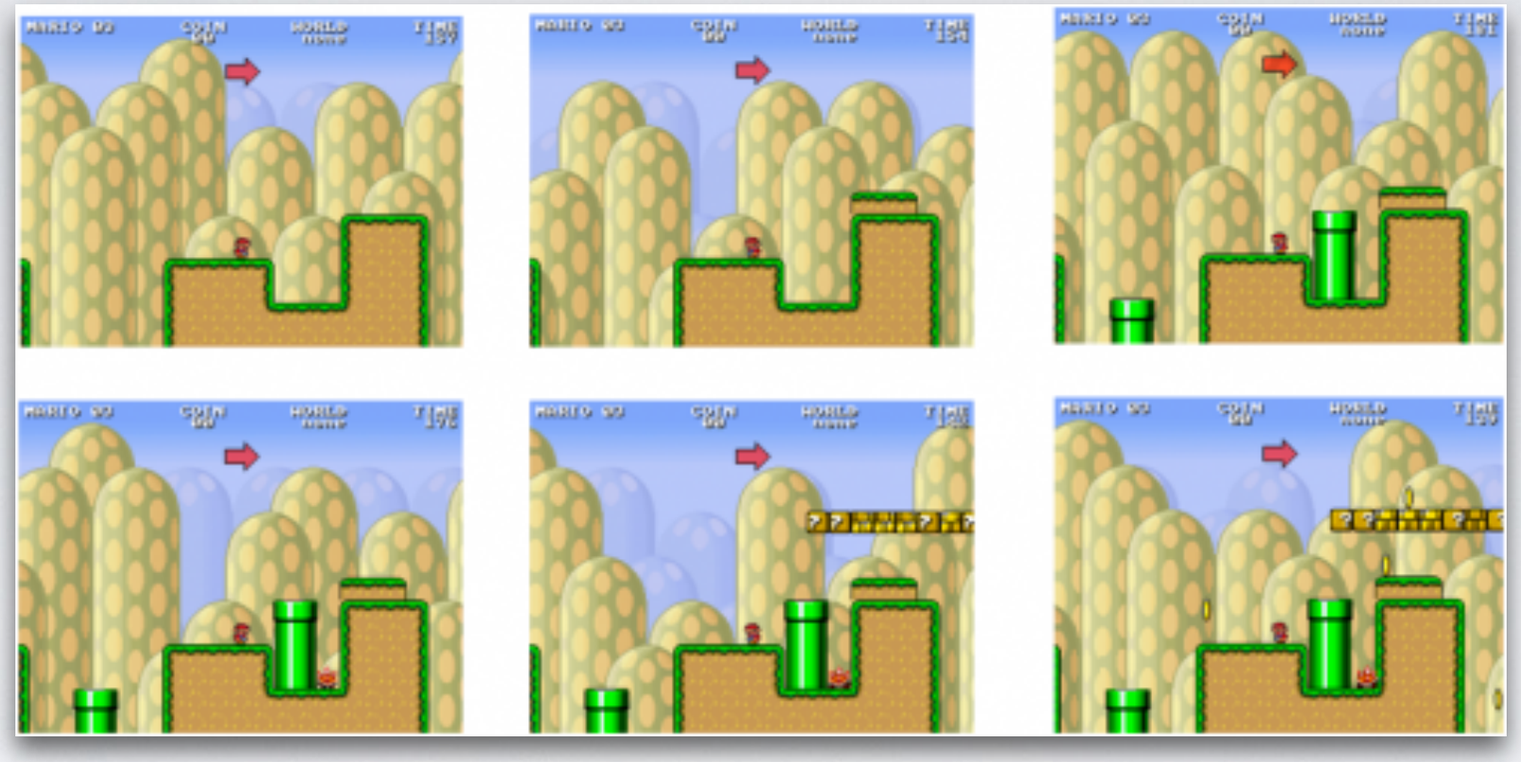
MULTIPLE PASSES



Ben Weber (source)

- For the game to be beatable, there must be at least one via path from start state to goal state. But no decision making can make for a boring game!
- Idea: using an agent-based approach, send the agent through multiple times from start to finish, adding elements on each pass based on what's a feasible action in context.
    - Eg. Generative Mario agent can jump, so adds a hole in the ground. Mario example explained here.
    - Each pass can add a different conceptual element, and previous layers can effect future layers.
        - Eg. terrain pass then enemy pass. Maybe enemies are more likely on lower ground.

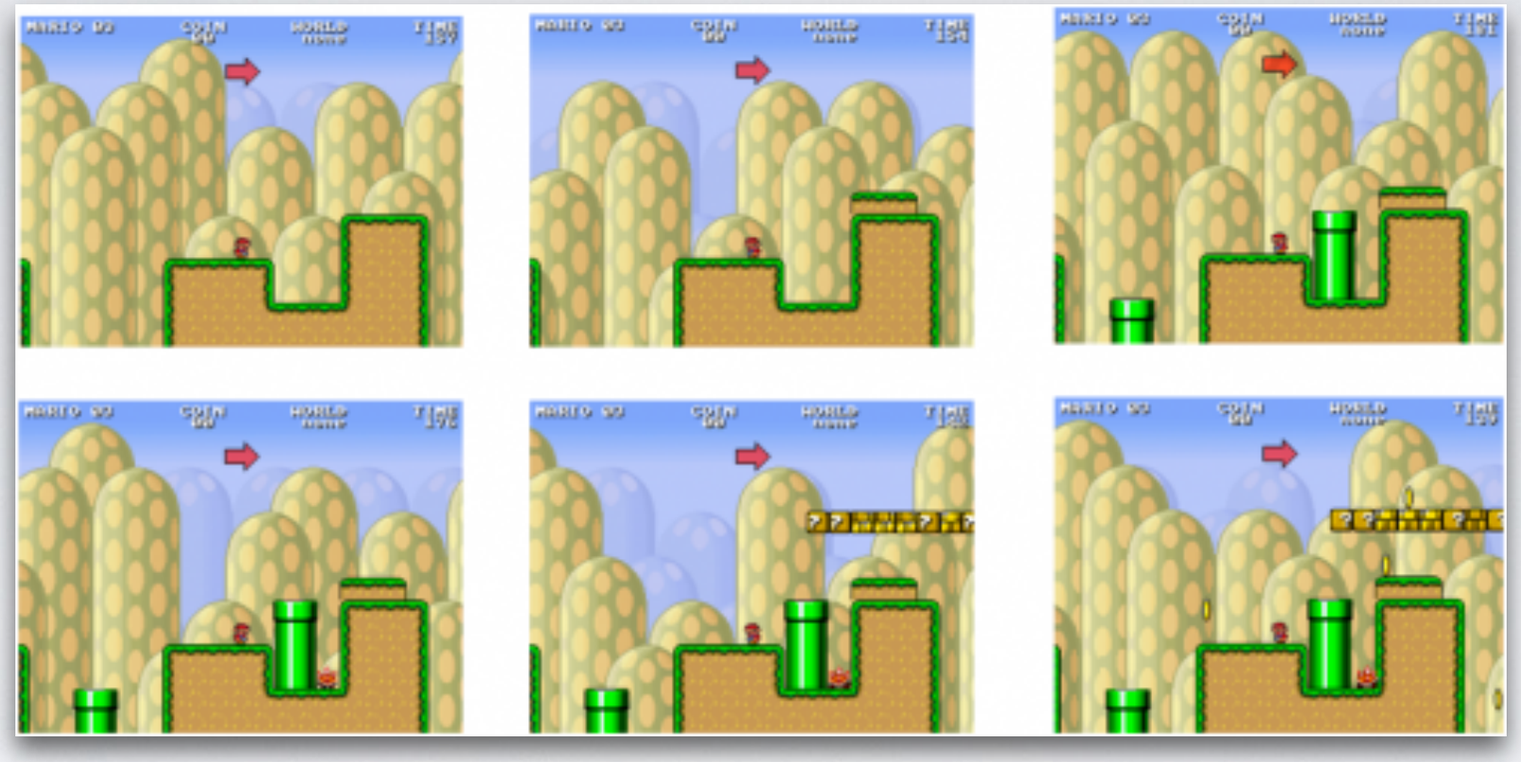
MULTIPLE PASSES



Ben Weber (source)

- For the game to be beatable, there must be at least one via path from start state to goal state. But no decision making can make for a boring game!
- Idea: using an agent-based approach, send the agent through multiple times from start to finish, adding elements on each pass based on what's a feasible action in context.
    - Eg. Generative Mario agent can jump, so adds a hole in the ground. Mario example explained here.
    - Each pass can add a different conceptual element, and previous layers can effect future layers.
        - Eg. terrain pass then enemy pass. Maybe enemies are more likely on lower ground.

MULTIPLE PASSES



Ben Weber (source)

- For the game to be beatable, there must be at least one via path from start state to goal state. But no decision making can make for a boring game!
- Idea: using an agent-based approach, send the agent through multiple times from start to finish, adding elements on each pass based on what's a feasible action in context.
    - Eg. Generative Mario agent can jump, so adds a hole in the ground. Mario example explained here.
    - Each pass can add a different conceptual element, and previous layers can effect future layers.
        - Eg. terrain pass then enemy pass. Maybe enemies are more likely on lower ground.

DYNAMIC UPDATE

Ben Weber (source)

- Another bonus of procedural level generation, unlike a traditional game, if your generation method is at runtime, you can dynamically tune difficulty!
- Check your players' performance and modify gameplay variables
- If you're using an agent-based generation method, can also modify the agent's behavior as you go.

IN SUMMARY

- There are many, many types of games and thus many, many generative methods
- In general though, we care about feasibility (beatable?), interesting design and difficulty, probably in that order.
    - Unlike our previous procedural discussions, our first priority is not visuals, but player experience
    - Feasibility is the easiest. Make sure there is a viable path from initial state to goal state literal path or not.
    - As always parameter tuning: figure out how parameters affect fun / difficulty
    - A game is a complex system. Work in layers of abstraction, slowly filling in layers of detail. Eg. Cave generation => enemy spawn => treasure placement.

REFERENCES

- Textbook
    - Wonderful survey chapter on which most of this lecture is based
- Articles
    - Good high-level discussion of procedural level design
    - Discussion of procedural level design via case study
    - In depth level gen tutorial with code
    - Detailed proc dungeon method explanation
    - Explanation of Spelunky's level generation
    - Proc room generation tutorial with Unreal blueprints
    - Maze-based proc dungeon with interactive examples
    - Many good articles on maze generation