

# Lecture 3: Constructive Generation Methods for Dungeons and Levels

Procedural Content Generation, Autumn 2013

Noor Shaker and Antonios Liapis

What makes  
a good level?

- Long?
- Unpredictable?
- Branching?
- Even level of challenge?
- Affords different playing styles?
- Beautiful?
- Realistic?
- Balanced?

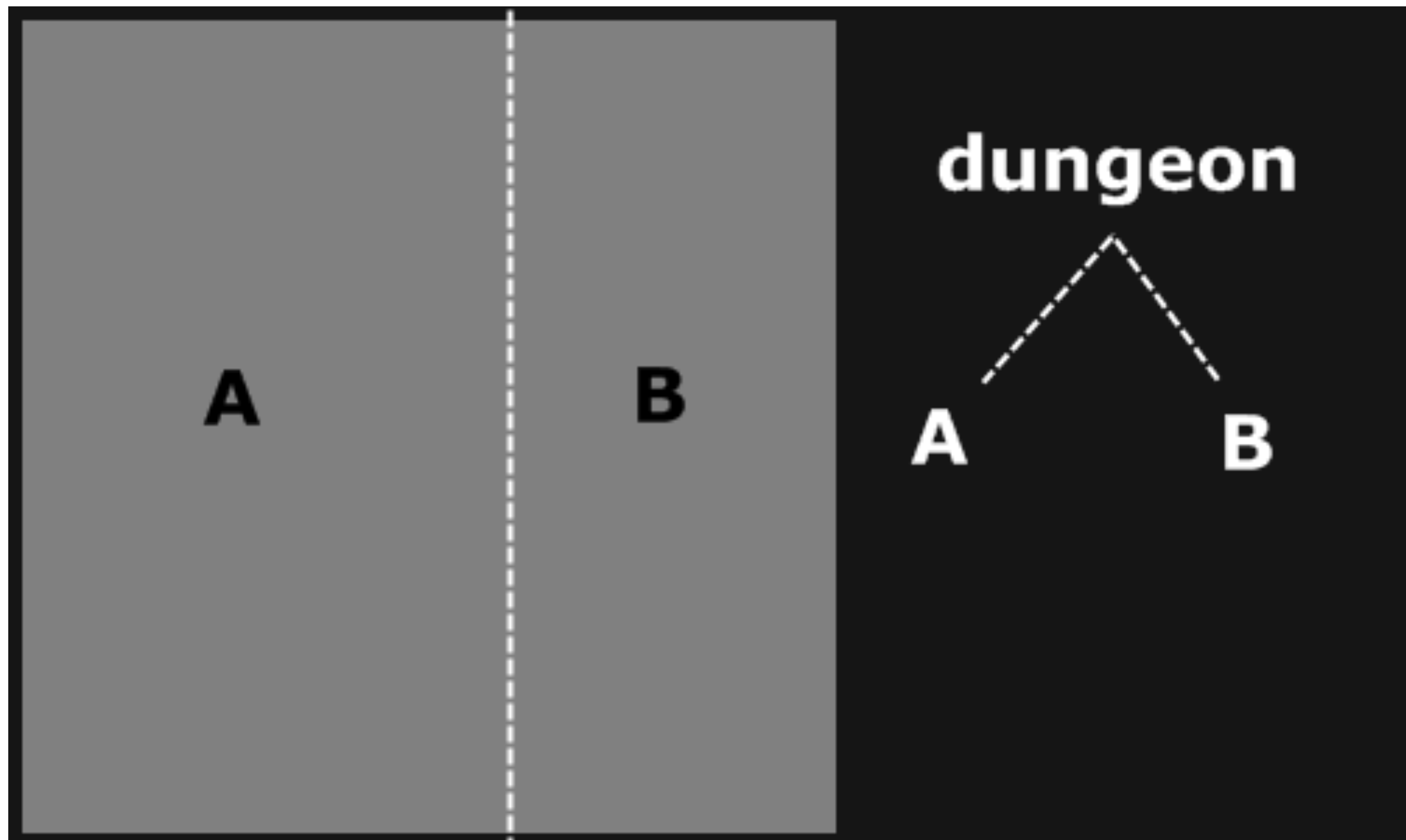
How could we  
generate levels with  
these properties?

# Simple Roguelike dungeon generation algorithms

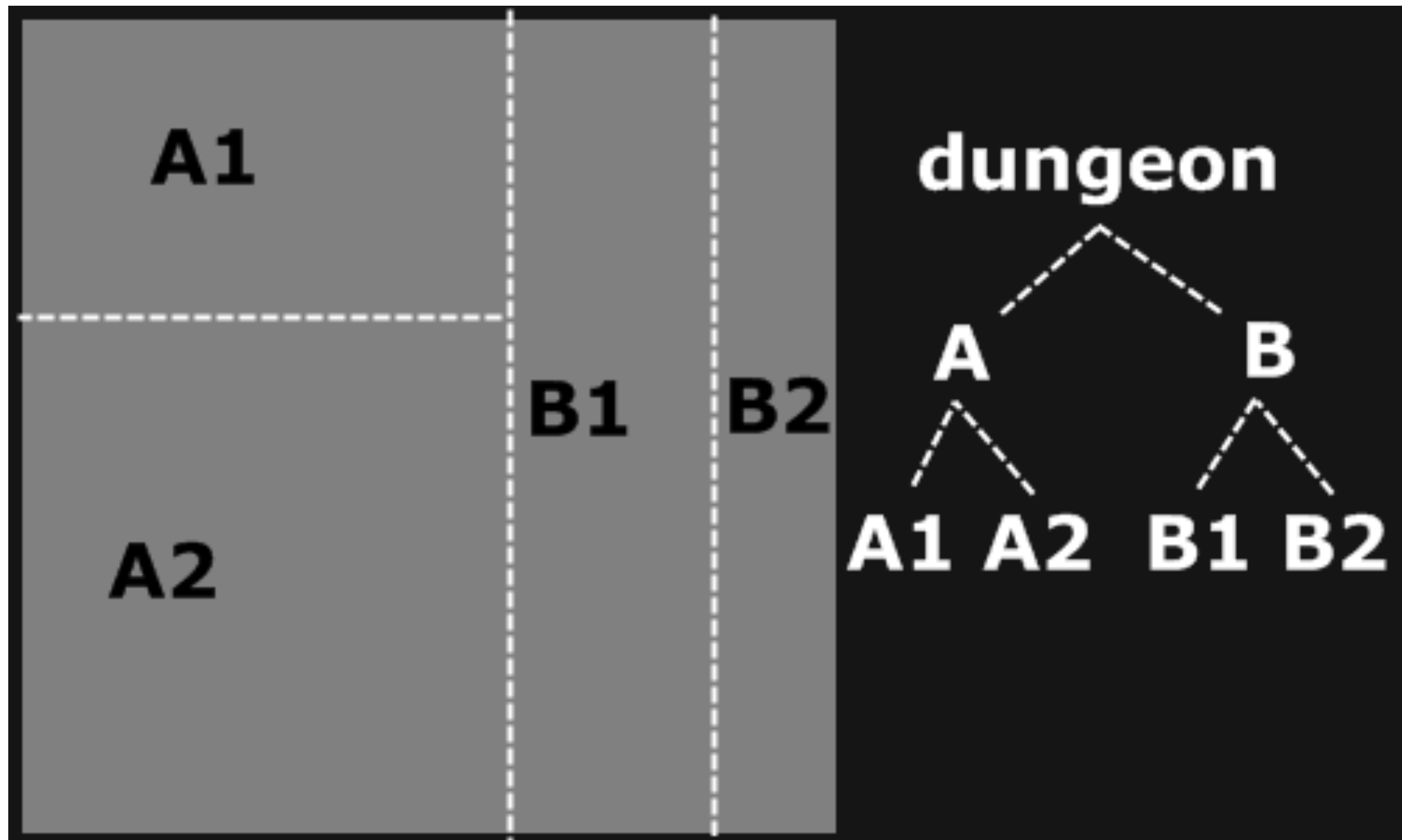
# Binary space partitioning

- choose a random direction : horizontal or vertical splitting
- choose a random position (x for vertical, y for horizontal)
- split the dungeon into two sub-dungeons
- call the same procedure for each sub-dungeon until finished
- Finally, create a room in each leaf and connect siblings

# Binary space partitioning

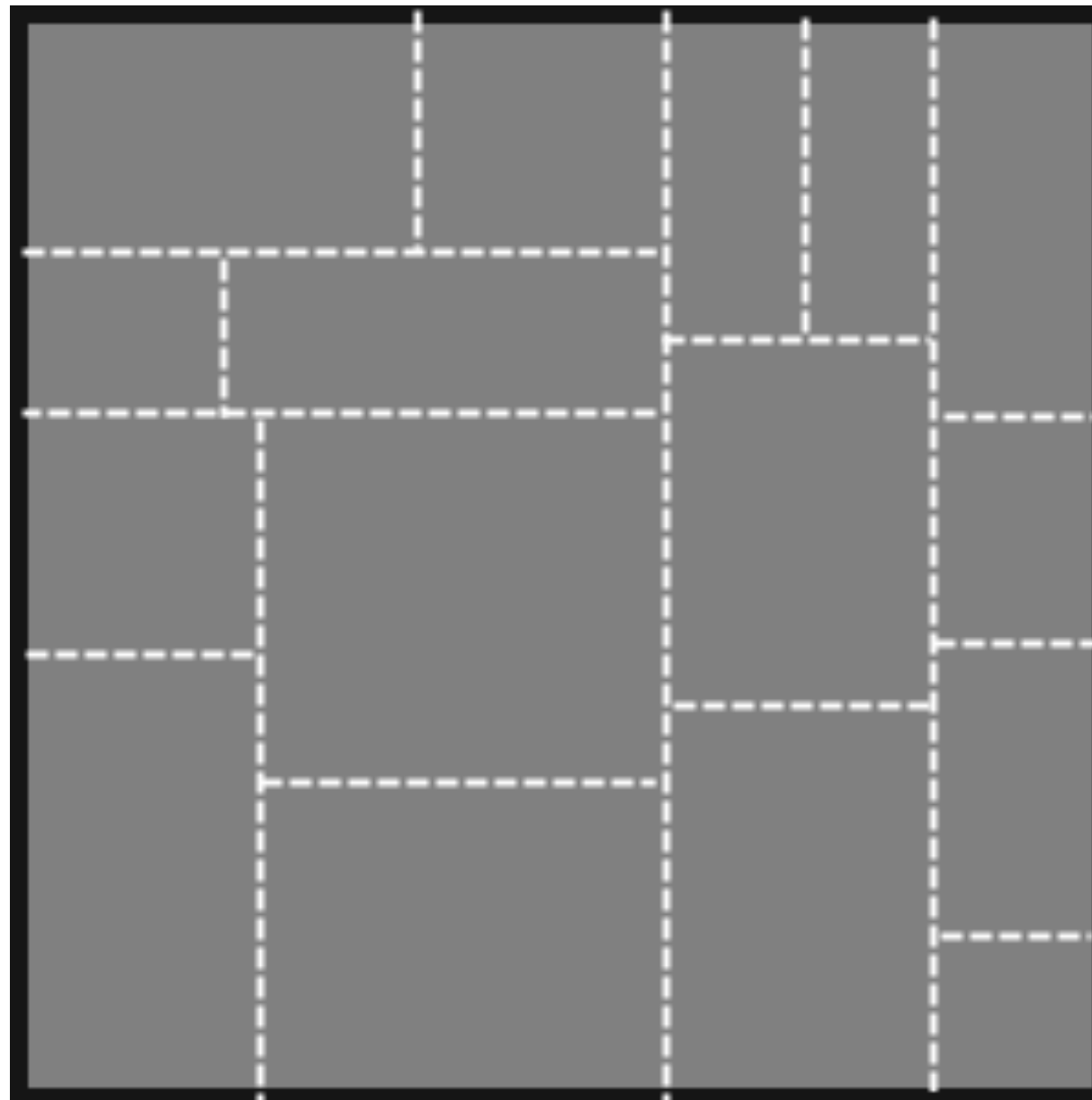


# Binary space partitioning



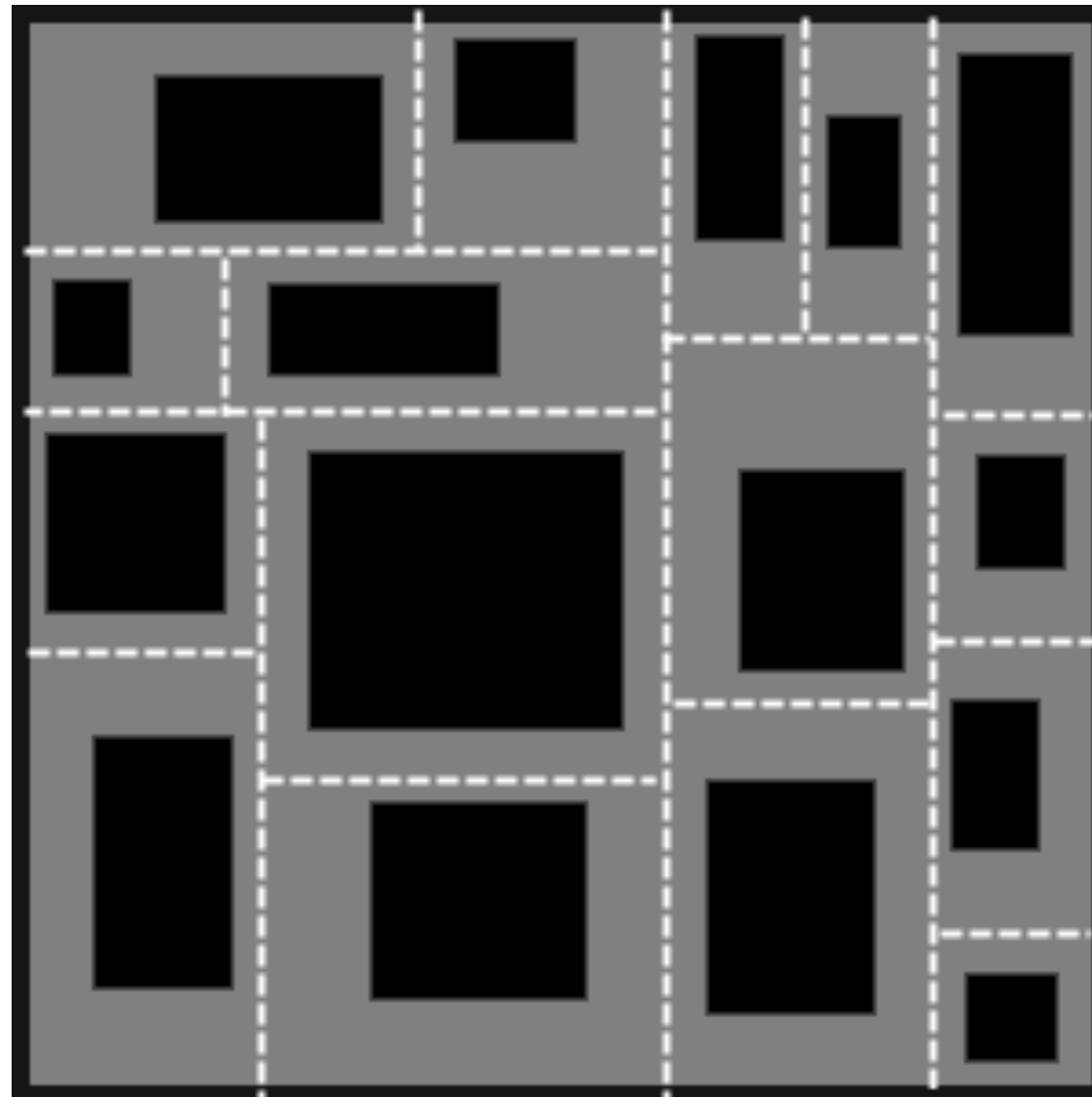


# Binary space partitioning



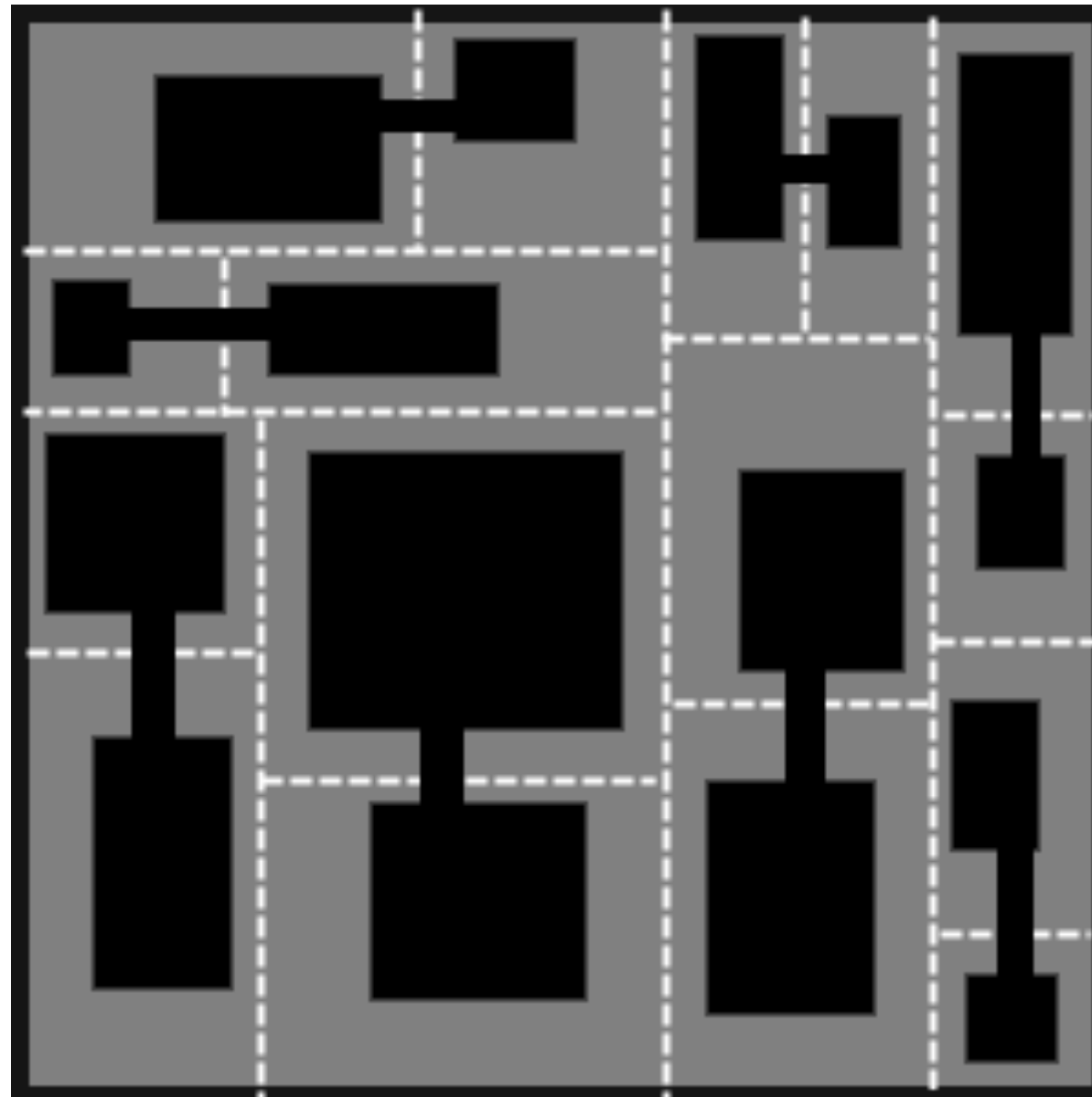
After 4 splitting

# Binary space partitioning



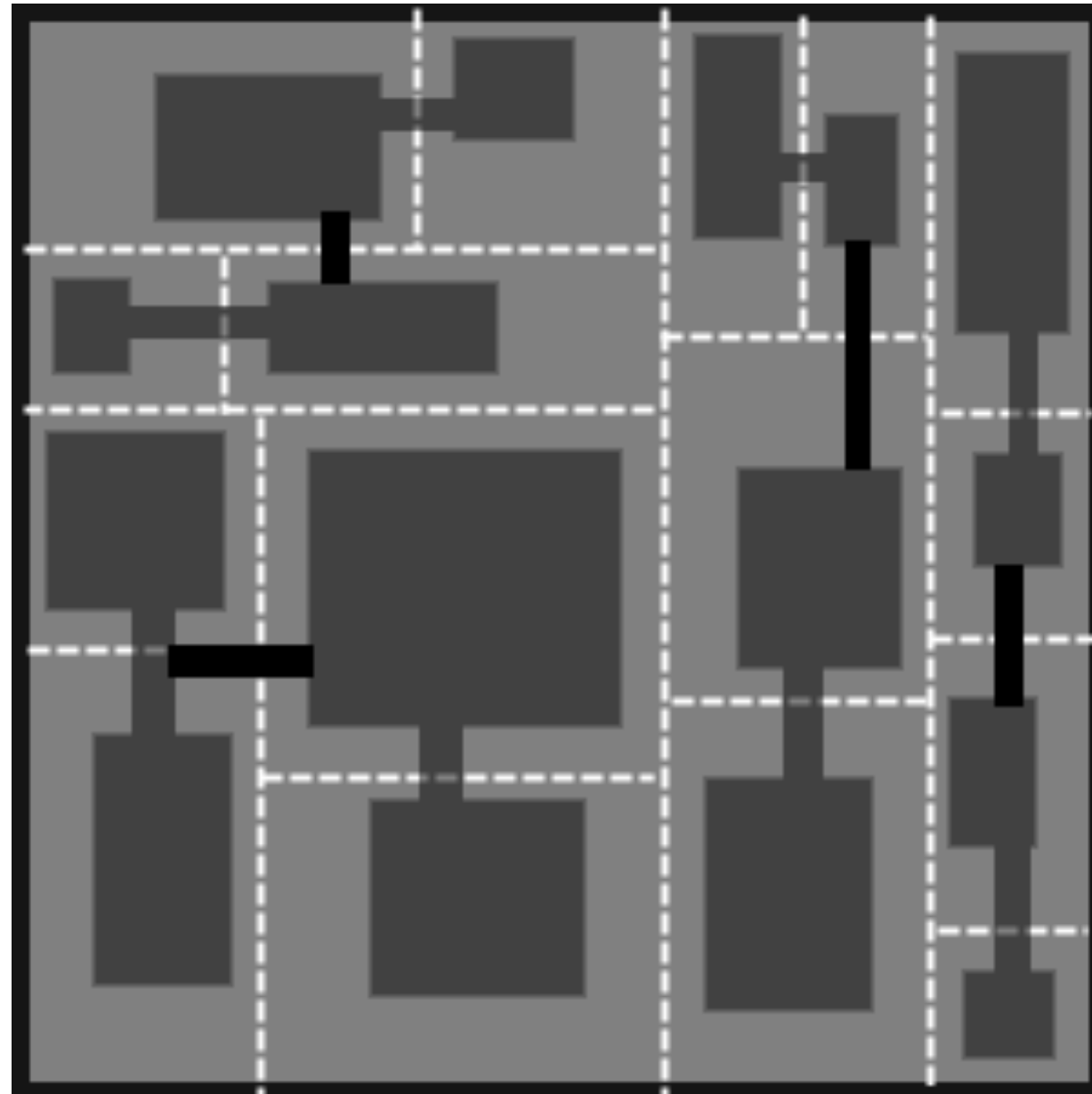
Creating the rooms

# Binary space partitioning



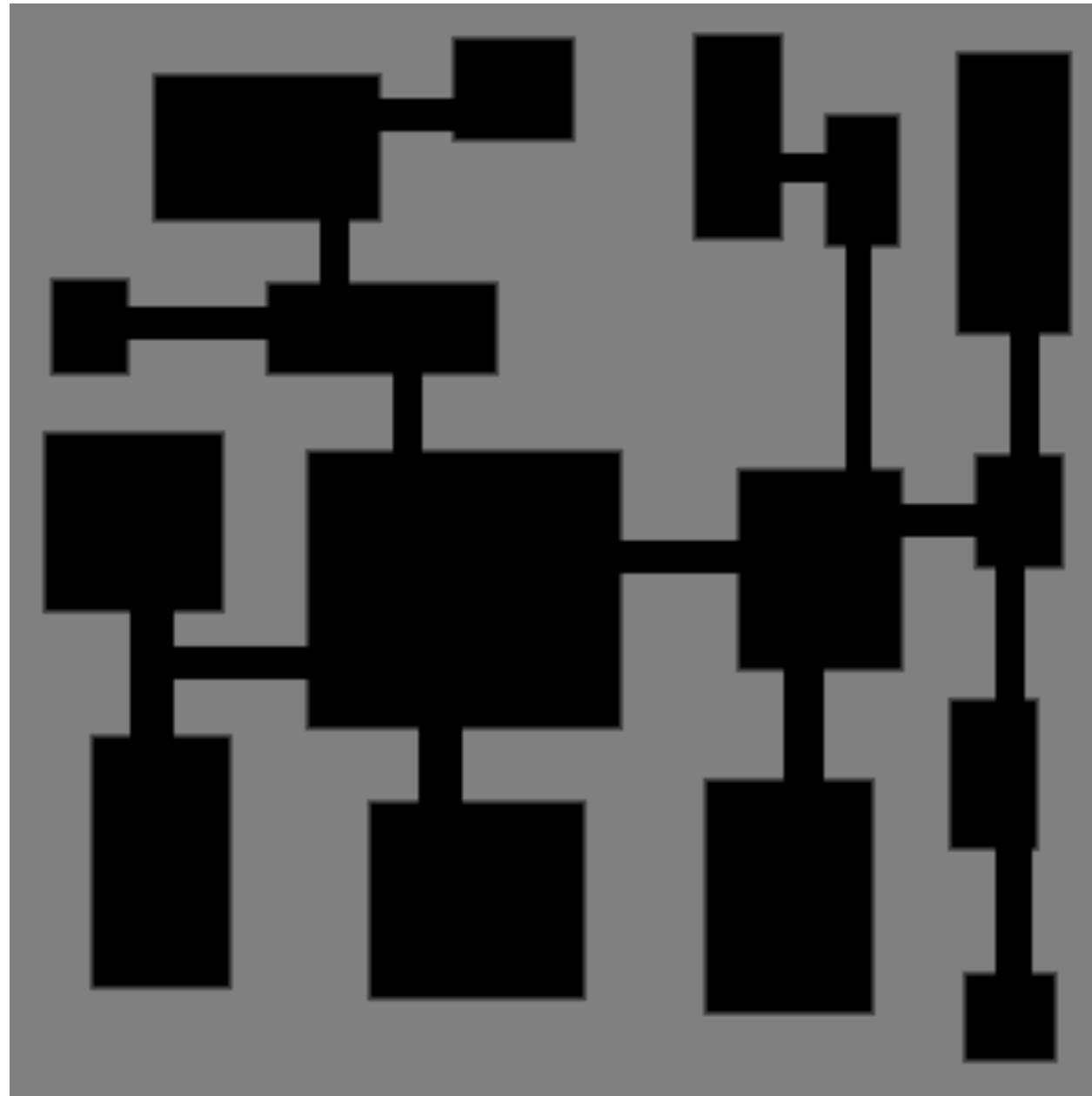
Adding level-1 corridors

# Binary space partitioning



Level-2 corridors

# Binary space partitioning



The complete dungeon

# Procedure

- 1: start with the entire dungeon area (root node of the BSP tree)
- 2: divide the area along a horizontal or vertical line
- 3: select one of the two new partitions
- 4: if this partition is above than the minimal acceptable size:
- 5:   go to step 2 (using this partition as the area to be divided)
- 6: select the other partition, and go to step 4
- 7: for every partition:
- 8:   create a room within the partition by randomly  
    choosing two points (top left and bottom right)  
    within the partition's boundaries
- 9: starting from the lowest layers, draw corridors to connect  
    rooms in the nodes of the BSP tree with children of the same  
    parent
- 10: repeat 9 until the children of the root node are connected

# Bros vs cons

- Bros:
  - easy to implement
  - no overlapping rooms or corridors
  - easy to create group of rooms
- Cons
  - very neat



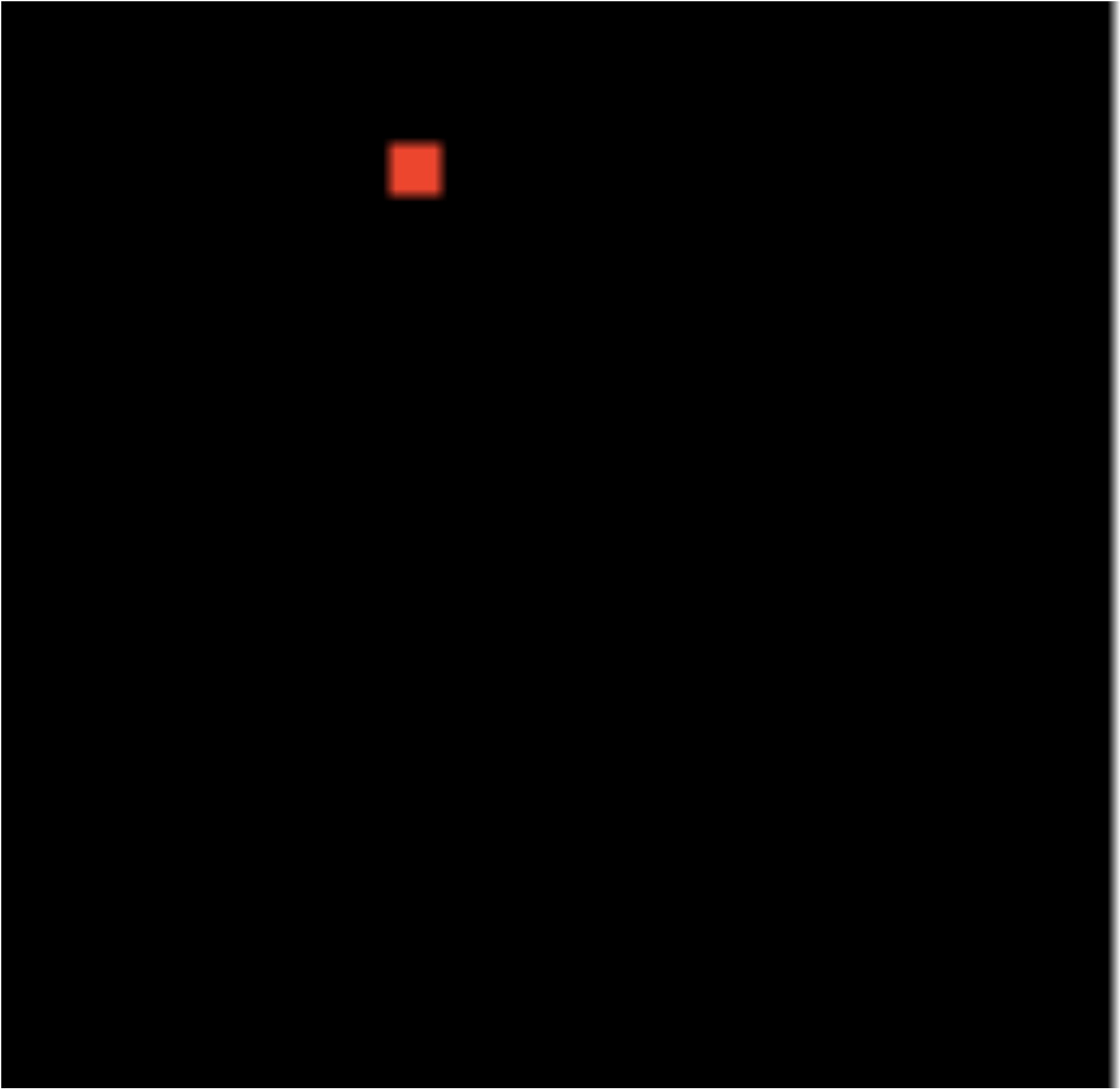


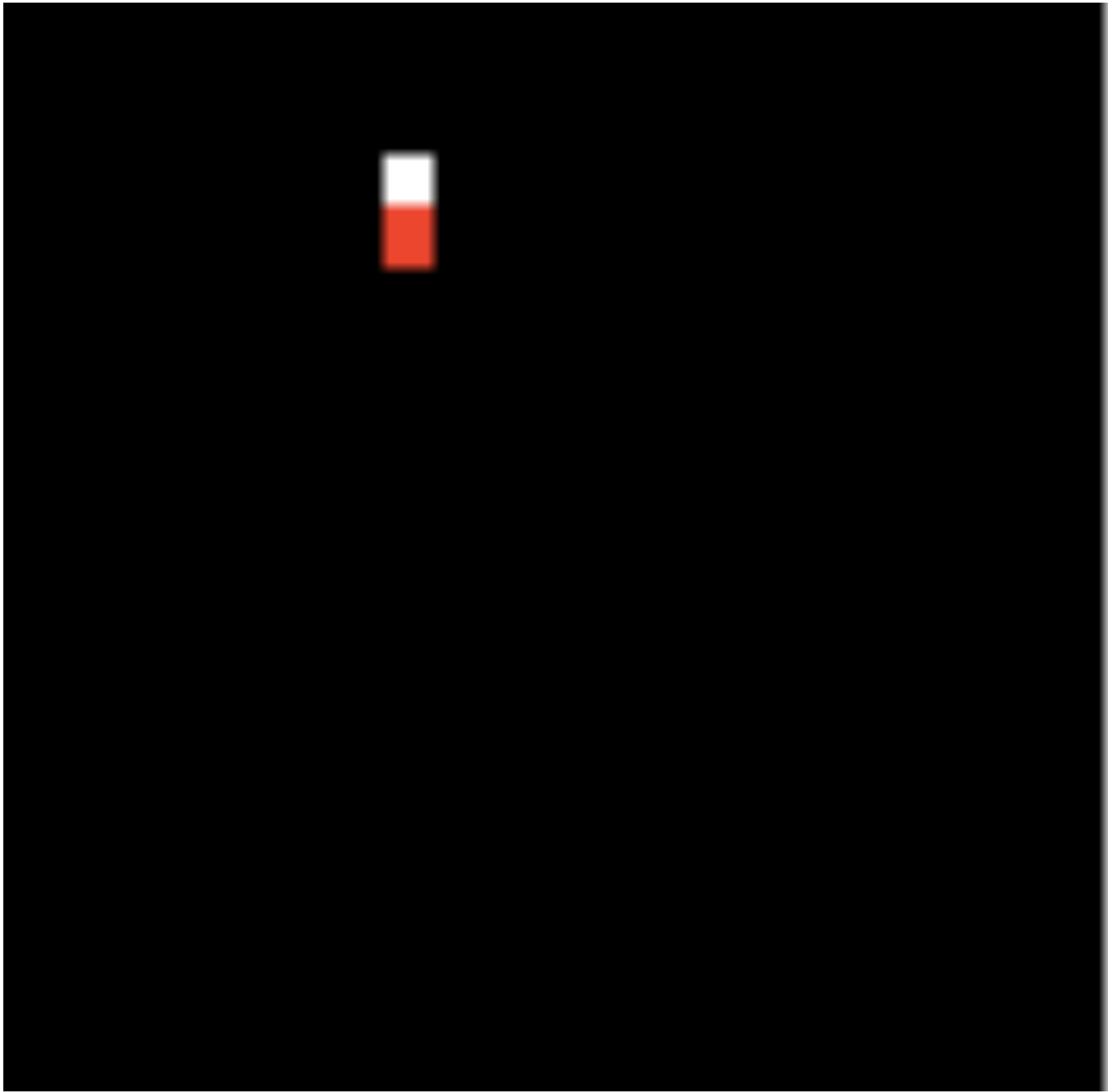
# Agent-based methods

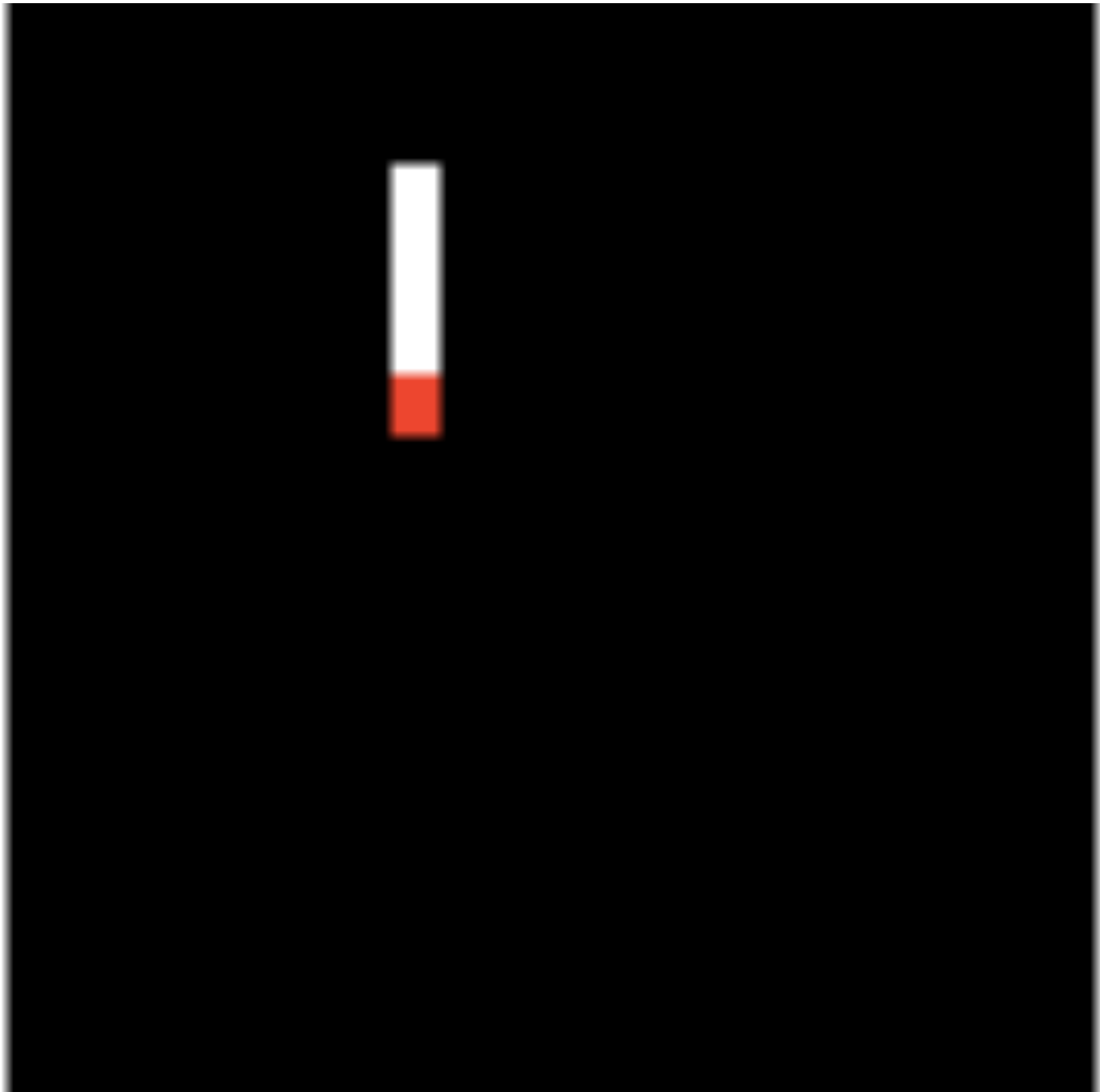
- Less predictable
- Less organized

# A highly stochastic method

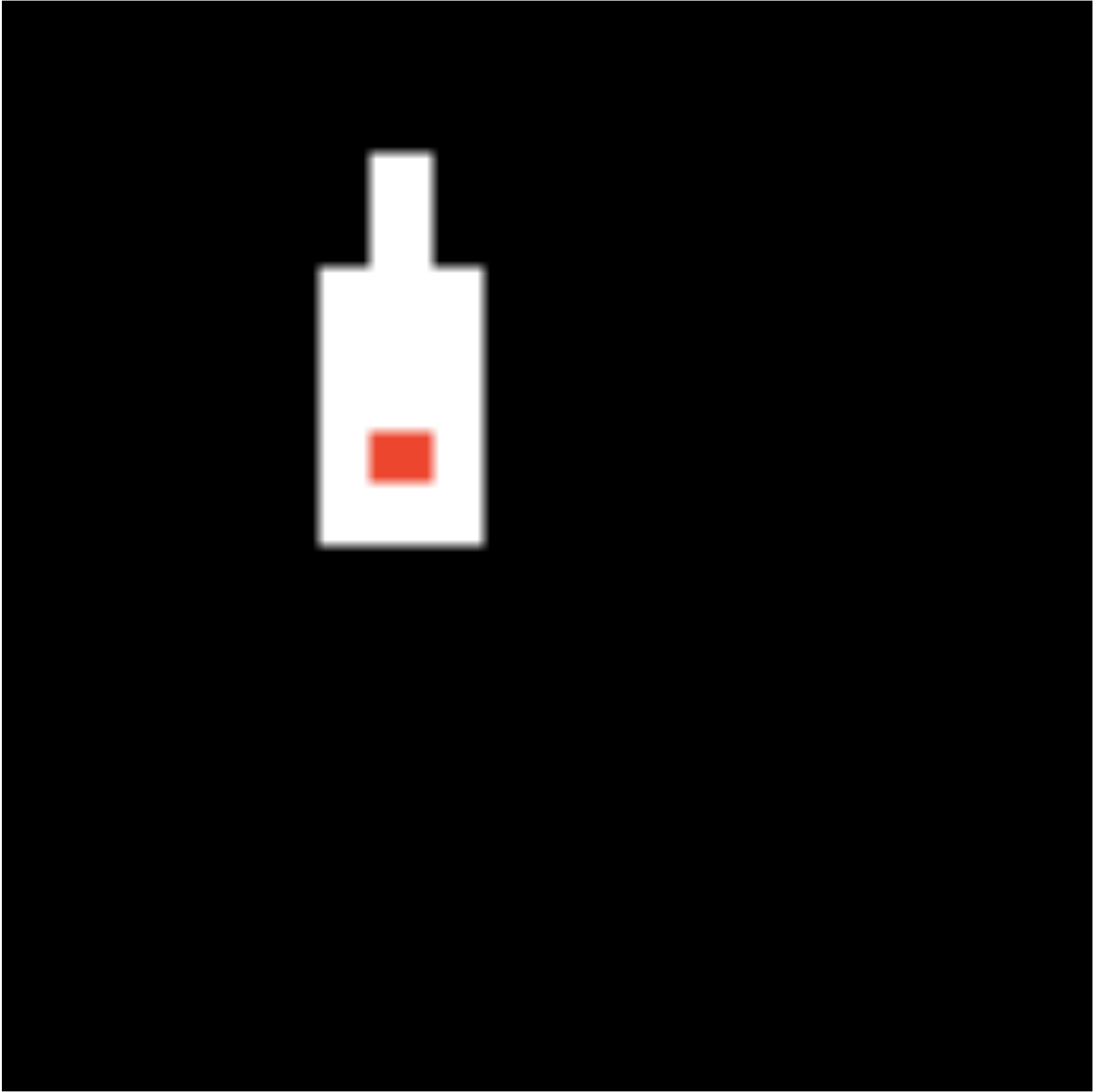
```
1: initialize chance of changing direction Pc=5
2: initialize chance of adding room Pr=5
3: place the digger at a dungeon tile and randomize its direction
4: dig along that direction
5: roll a random number Nc between 0 and 100
6: if Nc below Pc:
7:     randomize the agent's direction
8:     set Pc=0
9: else:
10:    set Pc=Pc+5
11: roll a random number Nr between 0 and 100
12: if Nr below Pr:
13:    randomize room width and room height between 3 and 7
14:    place room around current agent position
14:    set Pr=0
15: else:
16:    set Pr=Pr+5
17: if the dungeon is not large enough:
18:    go to step 4
```

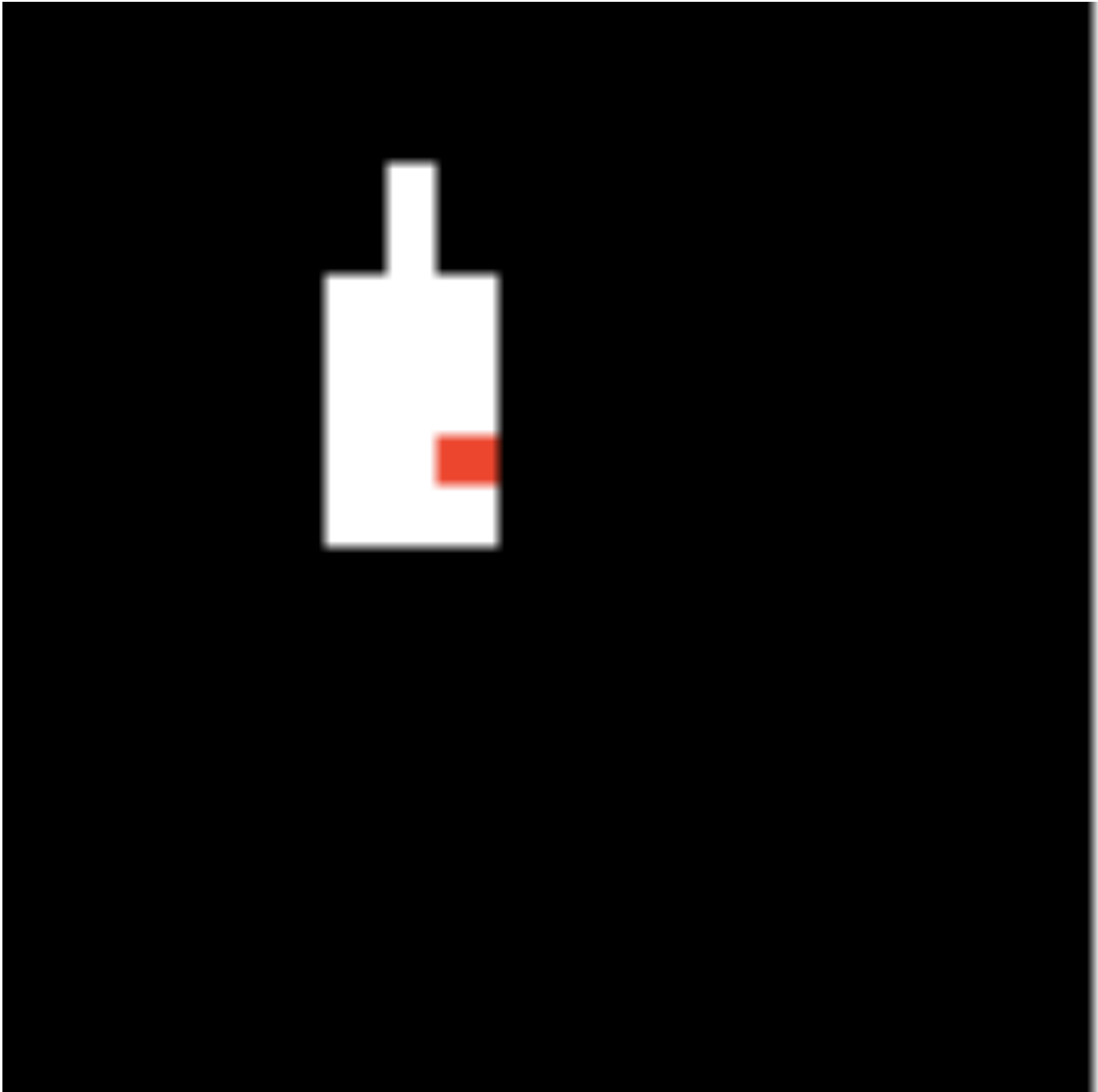




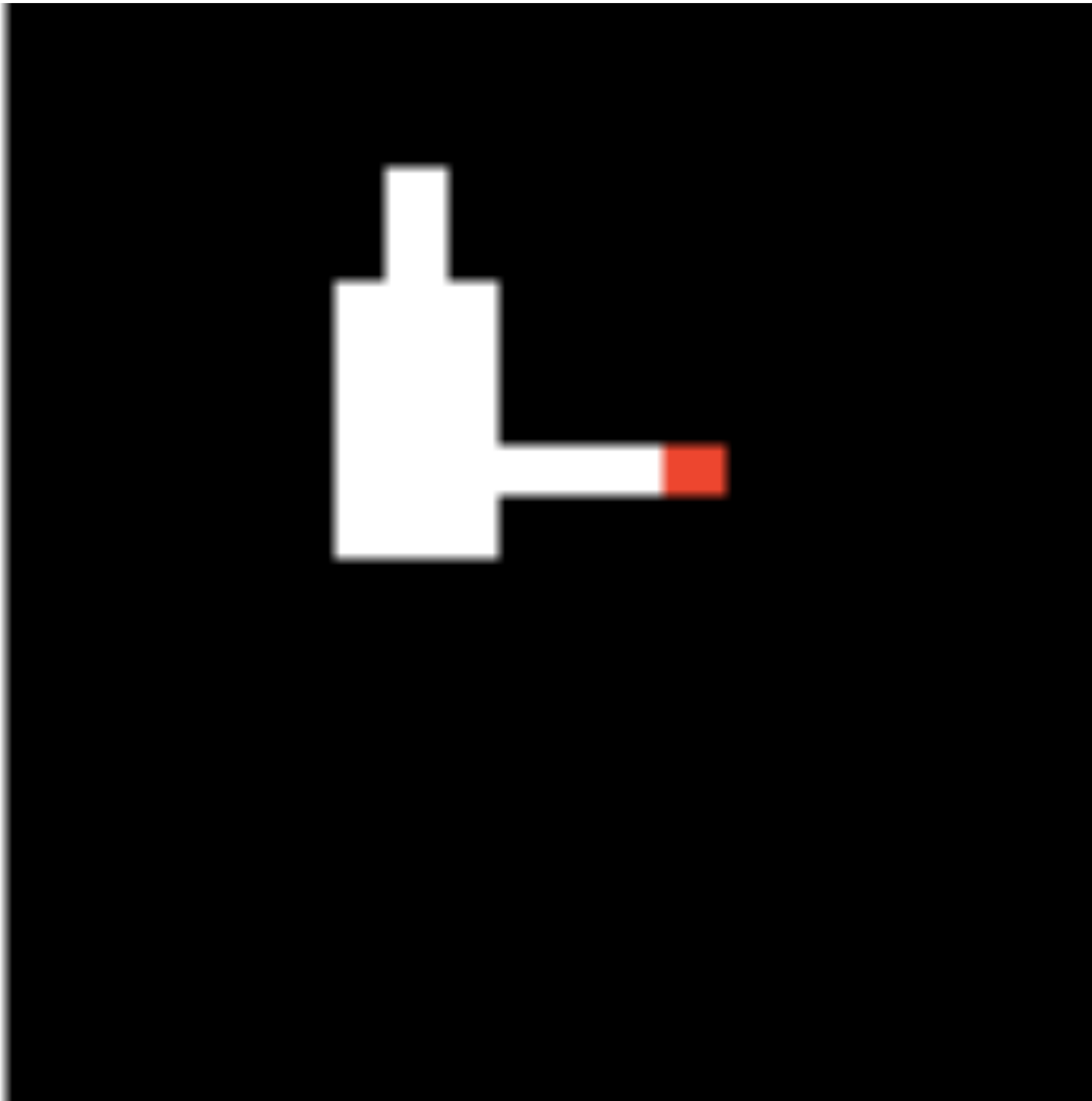


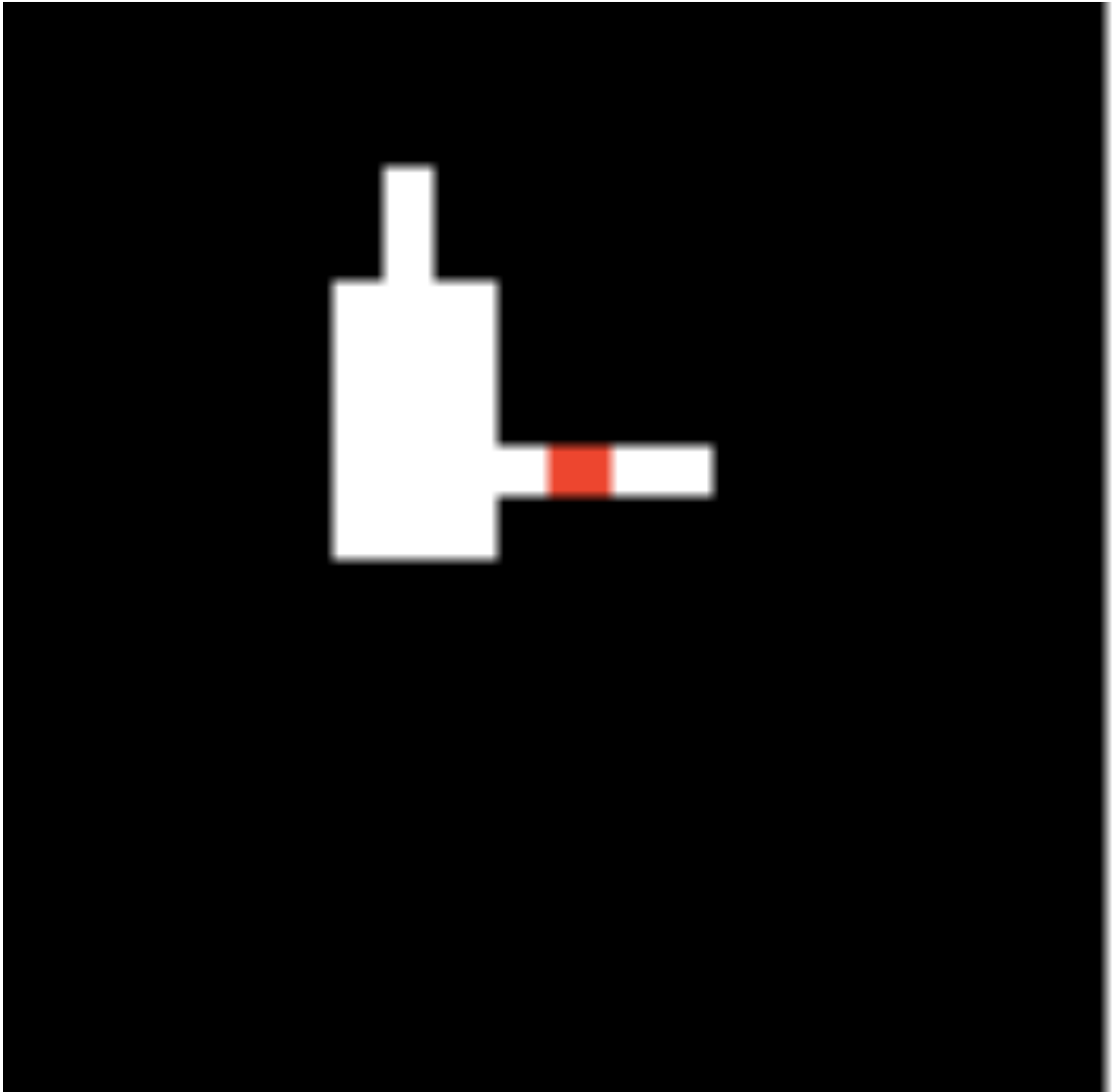














# Less stochastic method

- Use look ahead to avoid overlaps
- Make few changes in the direction

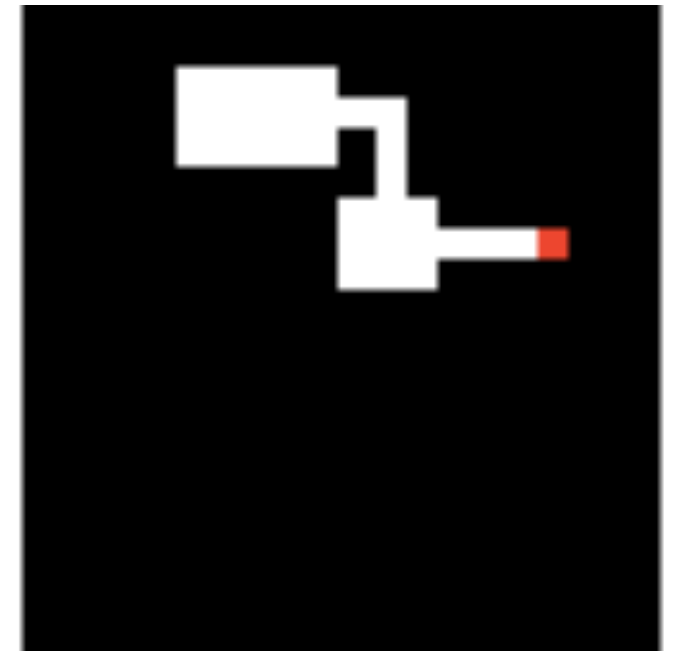
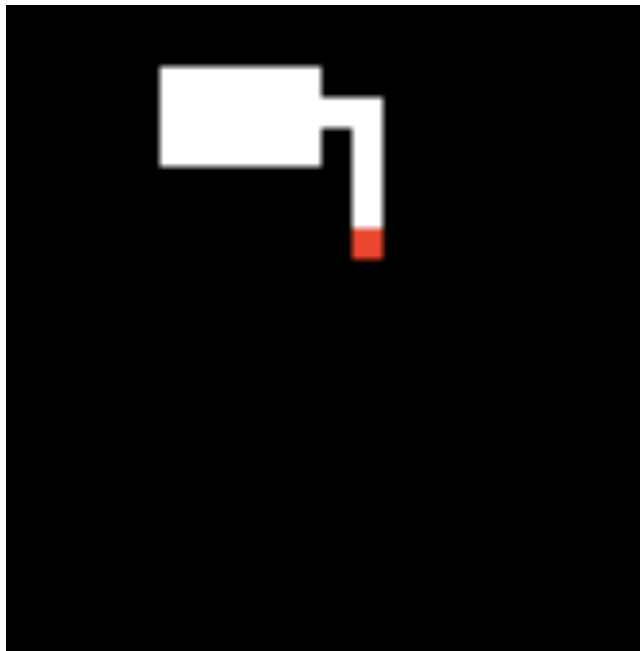
# Procedure

```
1: place the digger at a dungeon tile
2: set helper variables Fr=0 and Fc=0
3: for all possible room sizes:
3:   if a potential room will not intersect existing rooms:
4:     place the room
5:     Fr=1
6:     break from for loop
7: for all possible corridors of any direction and length 3 to 7:
8:   if a potential corridor will not intersect existing rooms:
9:     place the corridor
10:    Fc=1
11:    break from for loop
12: if Fr=1 or Fc=1:
13:   go to 2
```

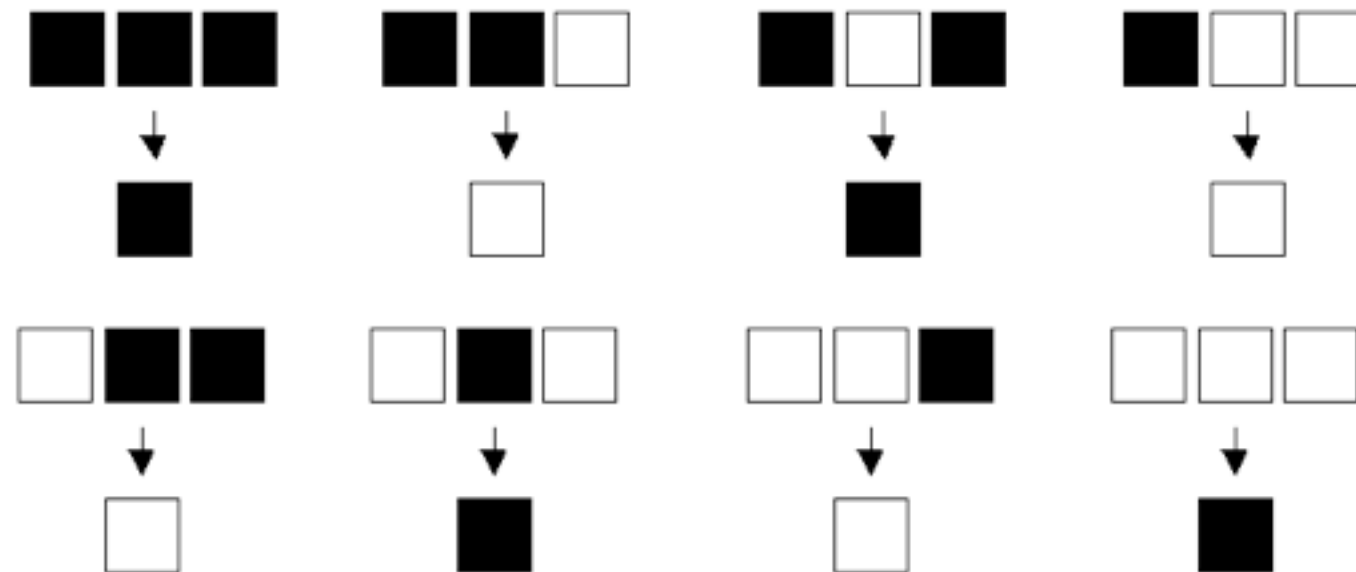
# Less stochastic method



# Less stochastic method



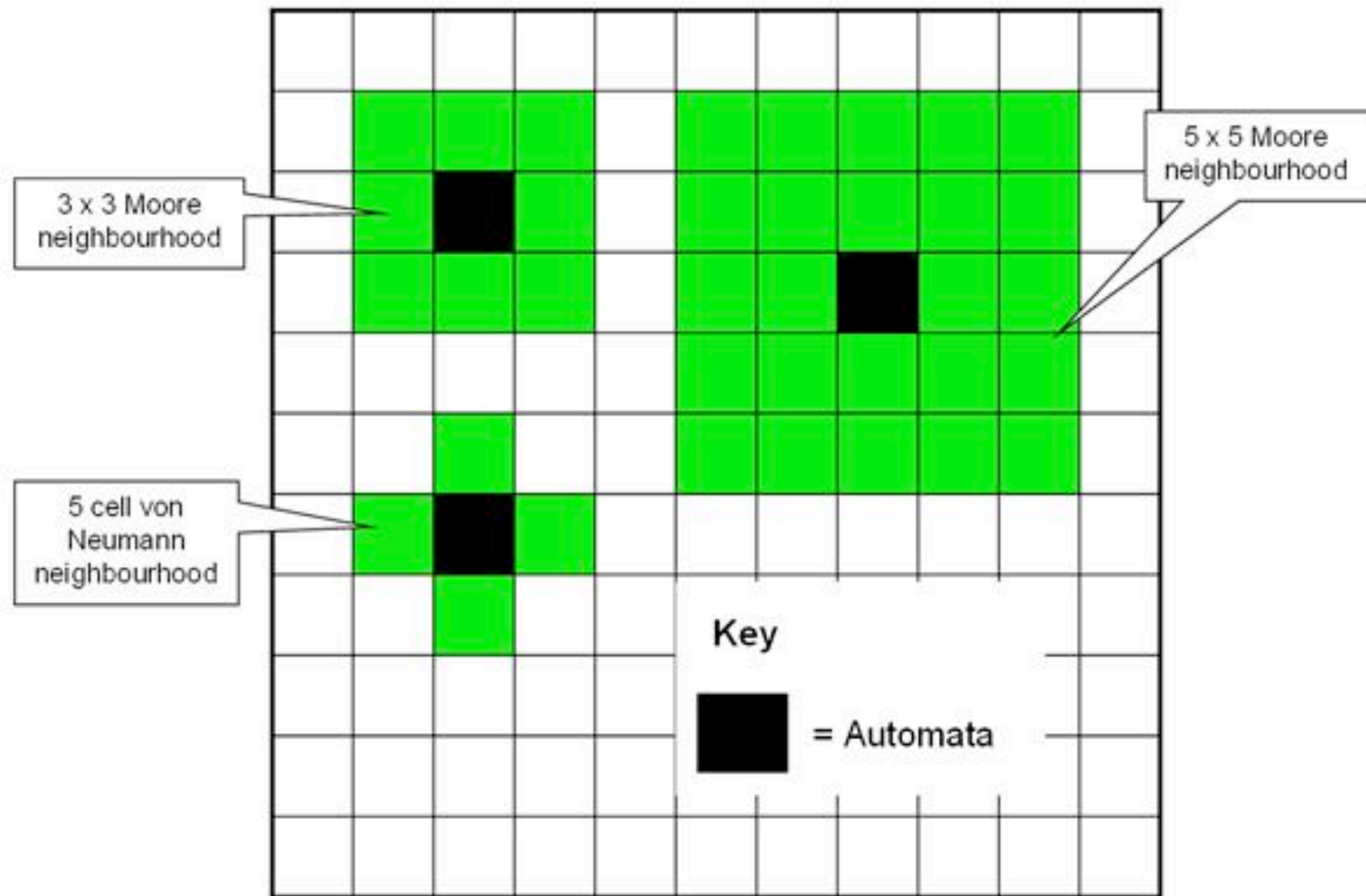
# Cellular automata



- Computational paradigm based on local interaction
- Used in artificial life and complexity studies
- The value of each cell in iteration  $n+1$  is based on the value of neighbouring cells in iteration  $n$  and some rule



# 2D cellular automata



# Cellular automata for real-time generation of infinite cave levels

Lawrence Johnson, Georgios Yannakakis and Julian Togelius

FDG PCG Workshop 2010

# This...

- A CA-based algorithm for generating infinite 2D caves
  - simple
  - realtime
  - looks good
  - *somewhat* controllable

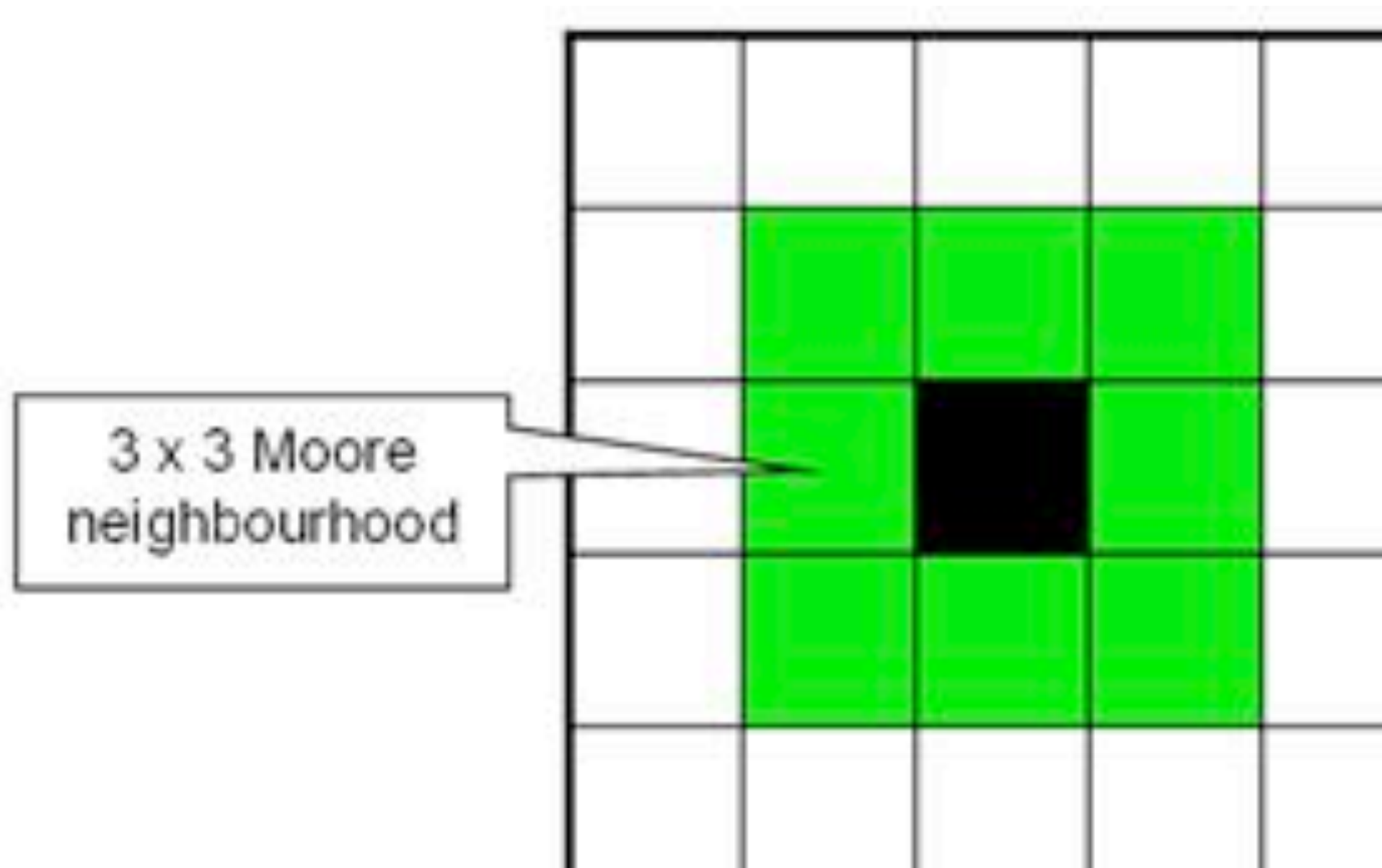
# The motivation

- *Cave Crawler*: a cooperative *abusive* dungeon crawler
- Never ends - therefore needs to produce infinite caves...

# CA cave generation

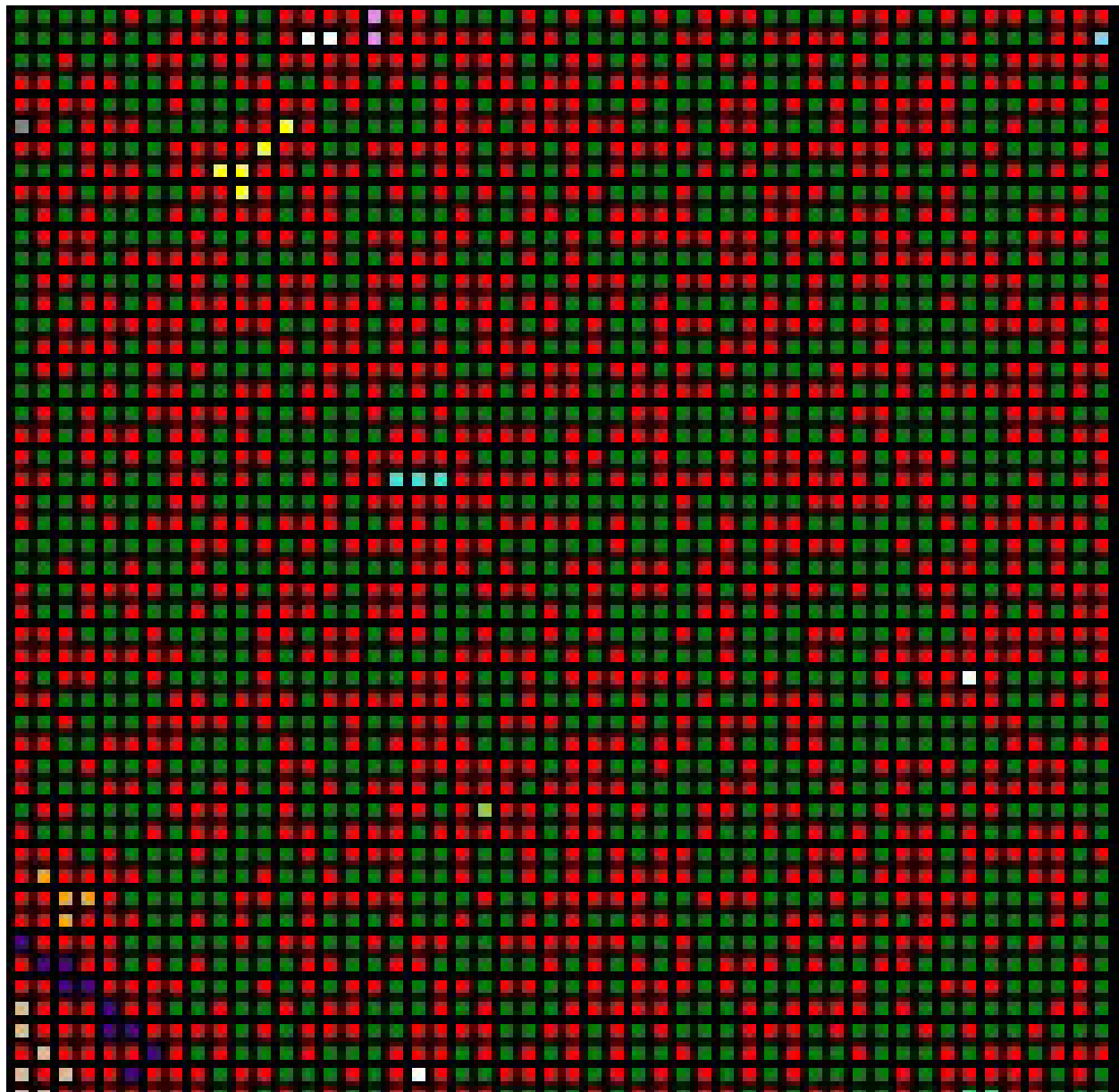
- Start with a square grid (e.g. 50\*50) - all floor
- Randomly switch a proportion of cells from floor to rock
- Run a CA  $n$  times, where each cell is set to:  
rock: if at least  $T$  neighbours are rock  
floor: otherwise
- Fill in the interior of rock formations

# Core CA mechanic

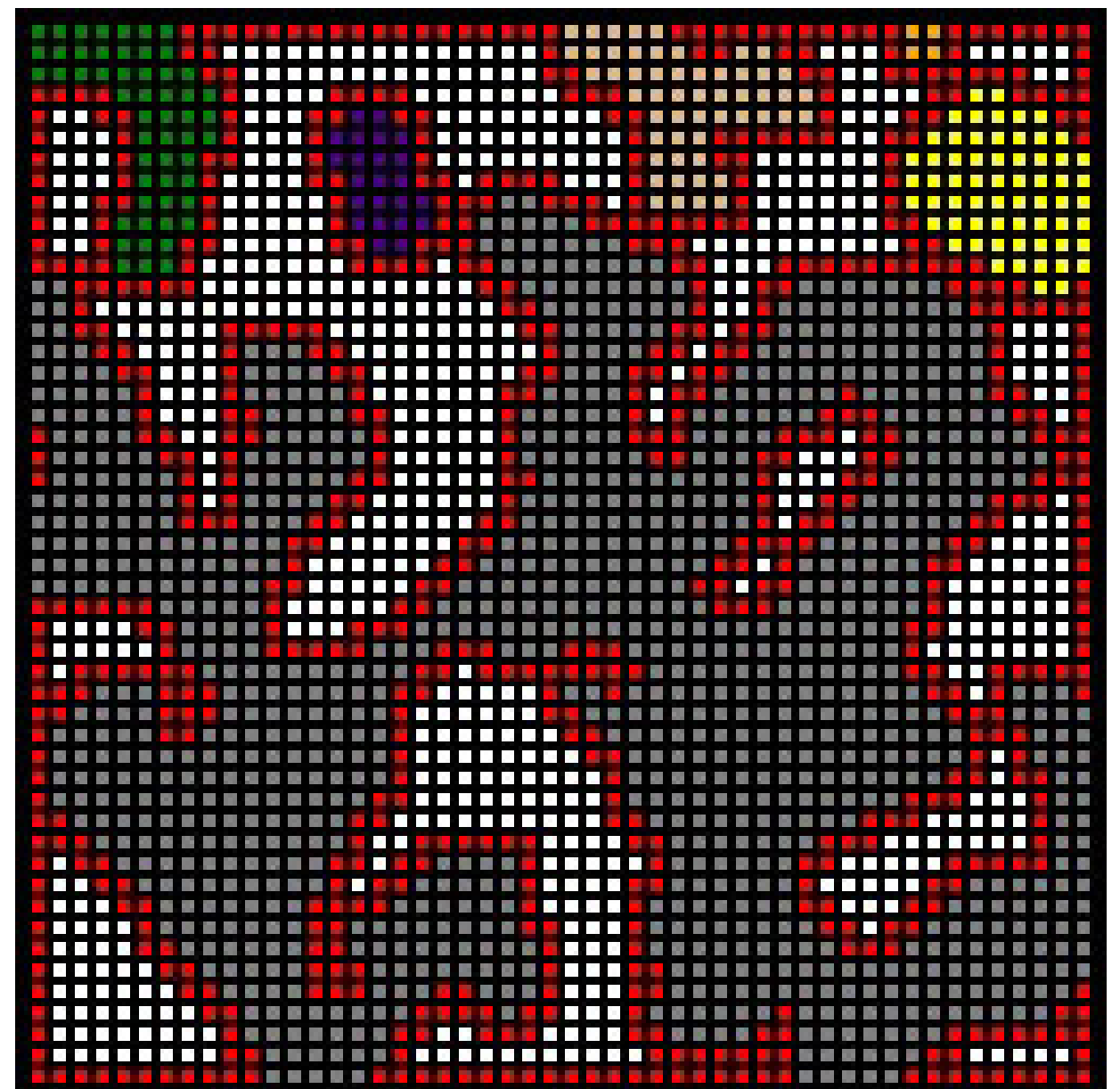


# Parameters

- $r$ : initial proportion of rock cells (0.5)
- $n$ : CA iterations (4)
- $T$ : neighbourhood value threshold that defines a rock (5)
- $M$ : Moore neighbourhood size (1)



(a) Random map

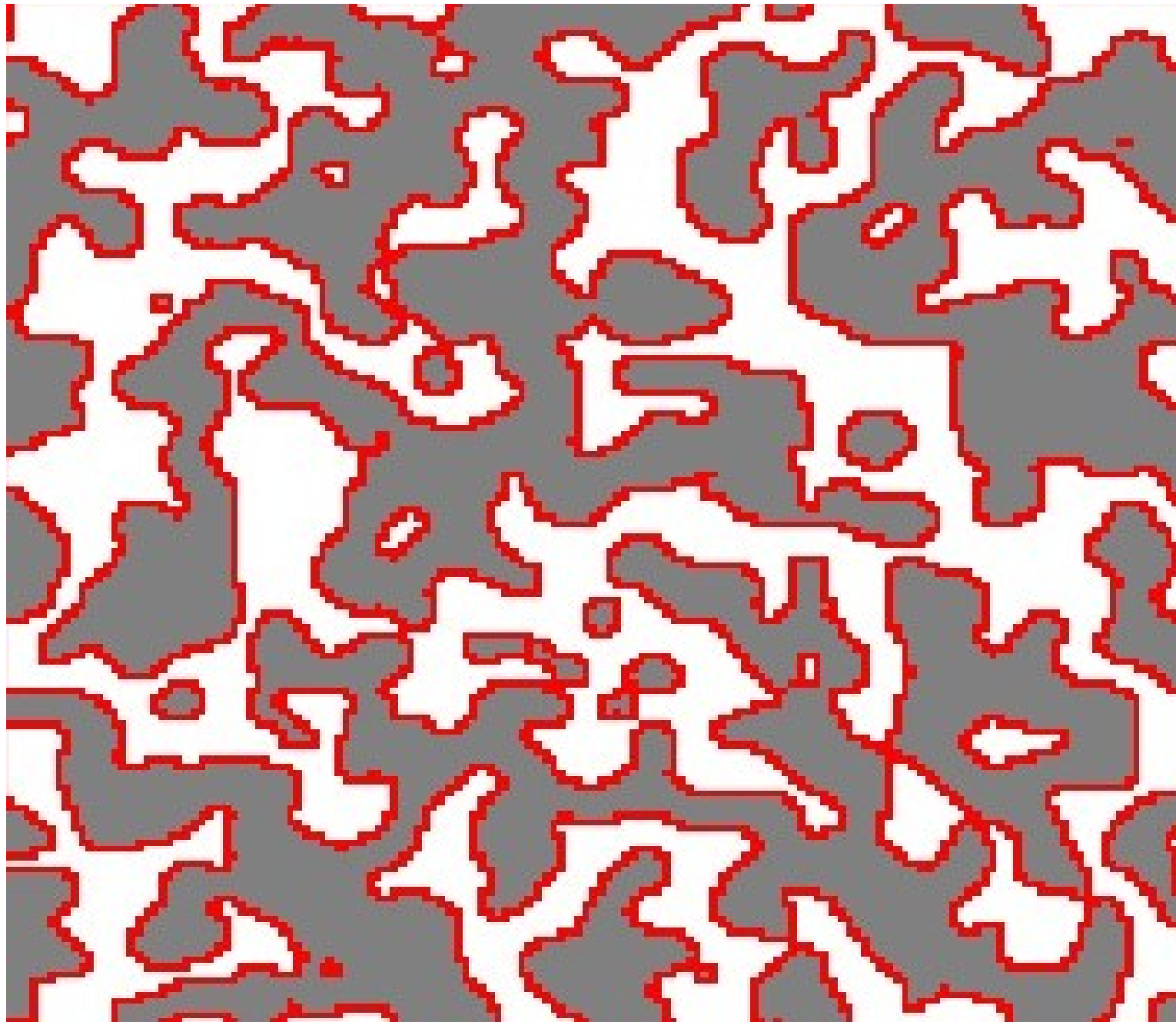


(b) CA map



# Adjacent rooms

- The infinite cave needs to be contiguous - and you need to be able to turn back!  
(Visited rooms stored as random seeds)
- Generate all four neighbours of a new room
- Dig tunnels from the central room to the new rooms at the shortest points
- Run the CA  $m$  times (2) on *all five rooms together* to smooth out edges

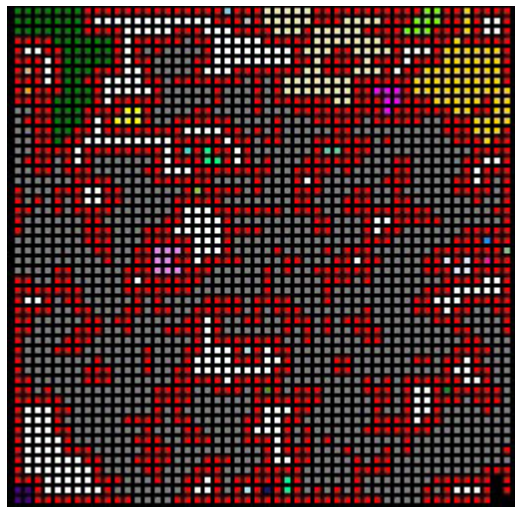


**Figure 3:** A  $3 \times 3$  base grid map generated with CA. Rock and wall cells are represented by red and white color respectively. Grey areas represent floor. ( $M = 2$ ;  $T = 13$ ;  $n = 4$ ;  $r = 50\%$ )

# Controllable?

Parameters can be varied,  
but what do they mean?

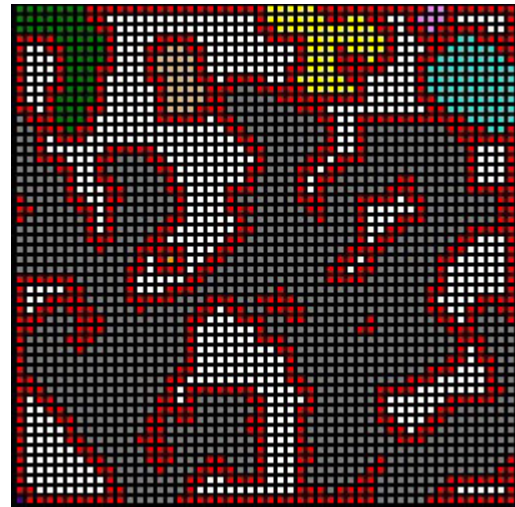




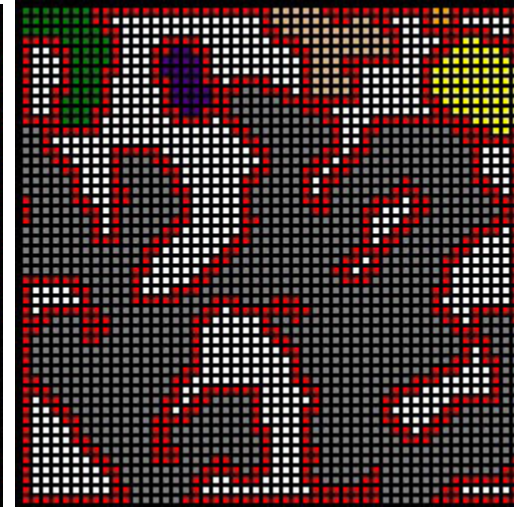
(a)  $n = 1, M = 1, T = 5$



(b)  $n = 2, M = 1, T = 5$



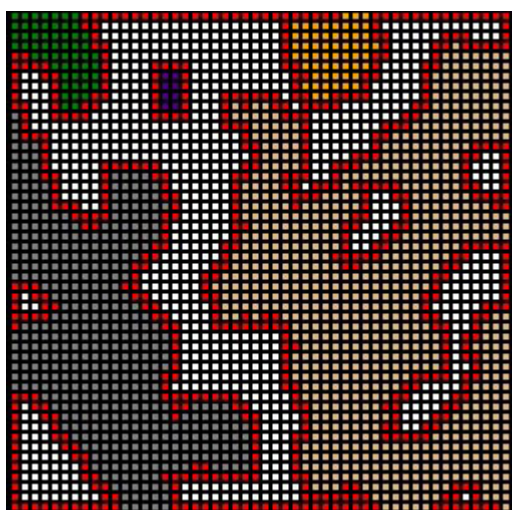
(c)  $n = 3, M = 1, T = 5$



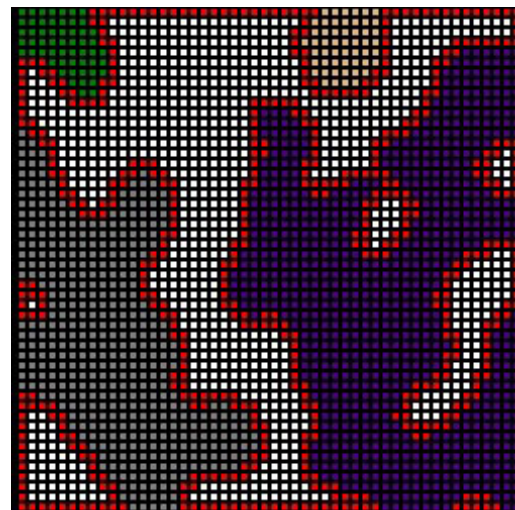
(d)  $n = 4, M = 1, T = 5$



(e)  $n = 1, M = 2, T = 13$



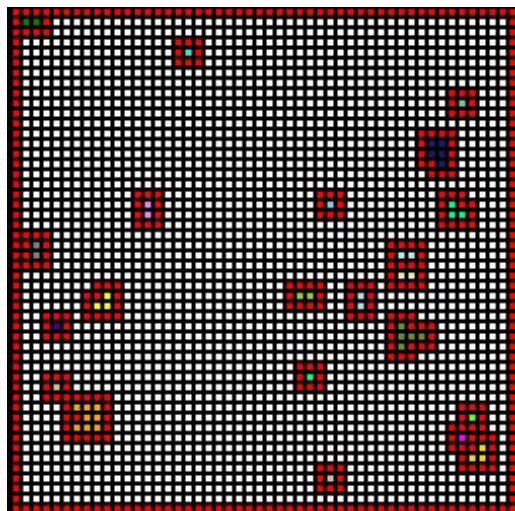
(f)  $n = 2, M = 2, T = 13$



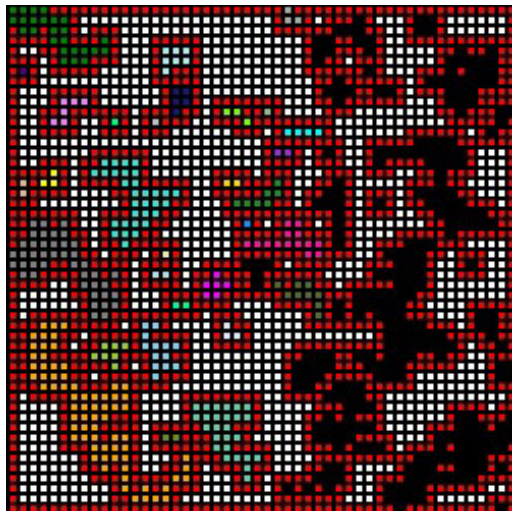
(g)  $n = 3, M = 2, T = 13$



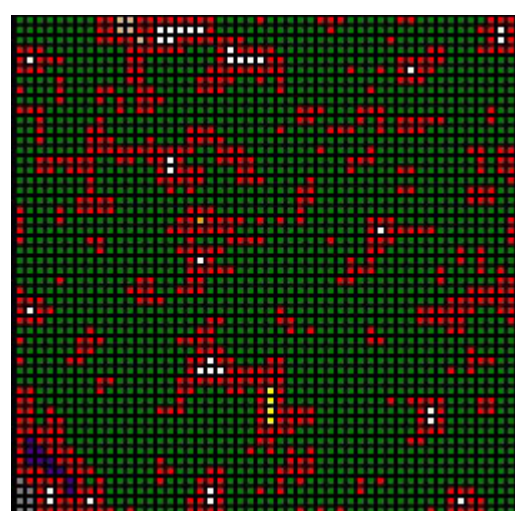
(h)  $n = 4, M = 2, T = 13$



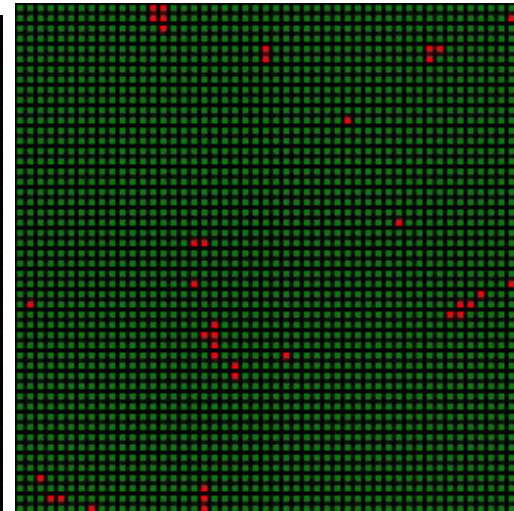
(i)  $n = 1, M = 1, T = 2$



(j)  $n = 1, M = 1, T = 4$



(k)  $n = 1, M = 1, T = 6$



(l)  $n = 1, M = 1, T = 8$

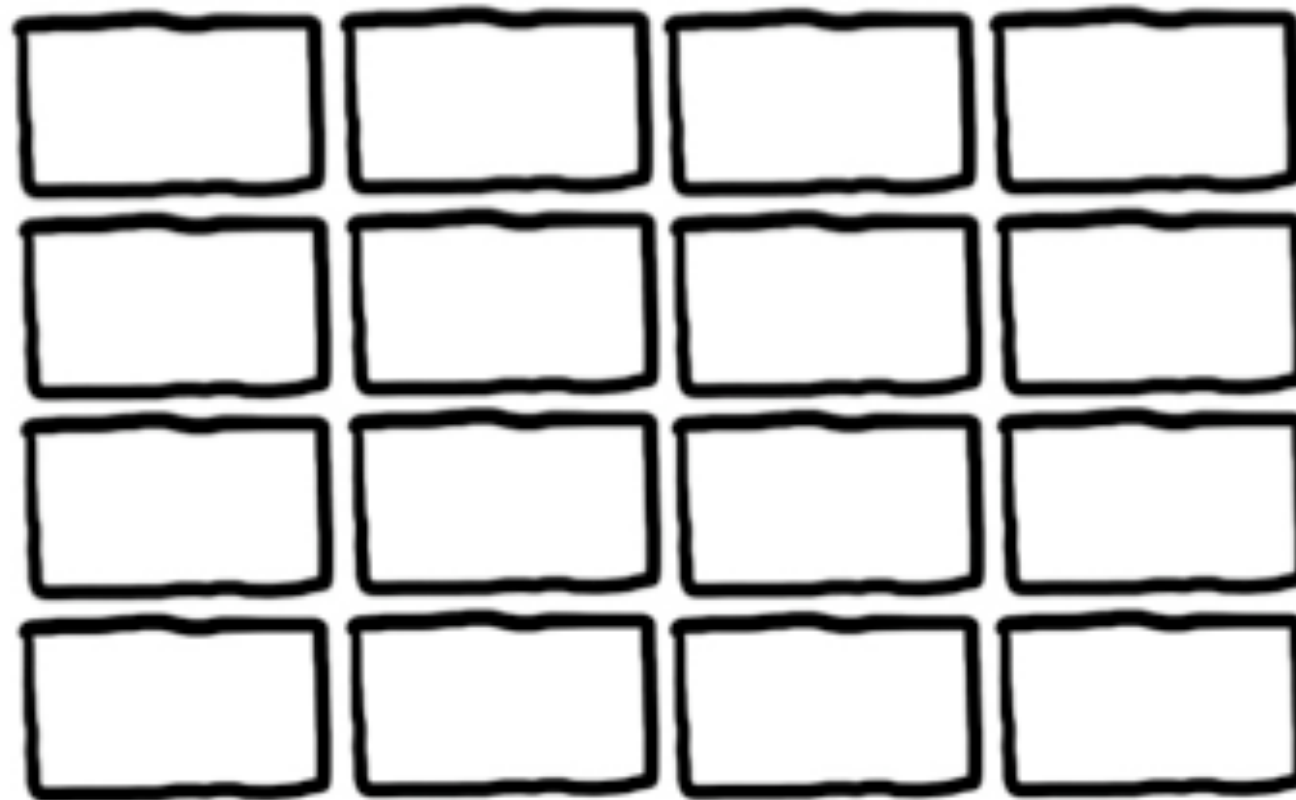


# Spelunky



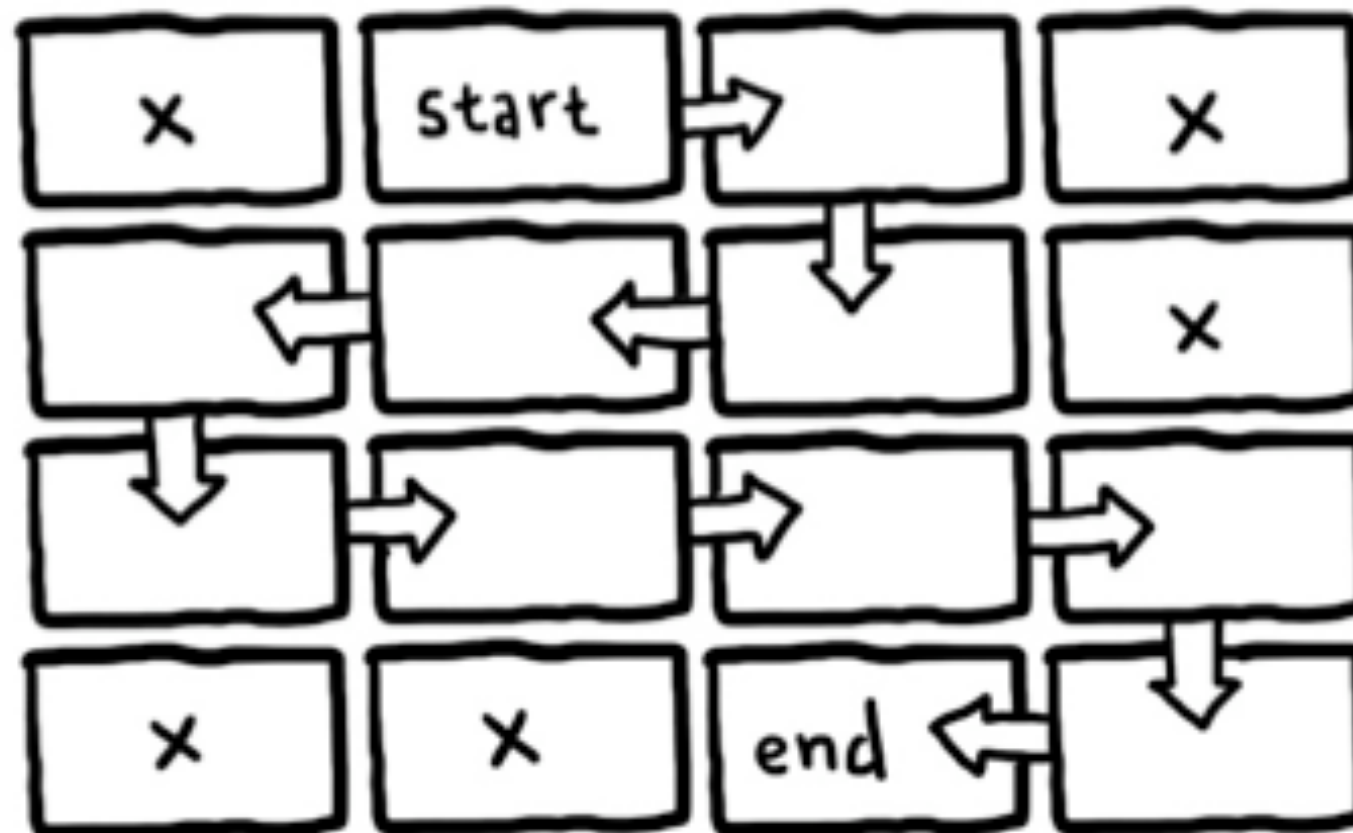
- Roguelike-like platformer
- Combines the fast pace of the platformer with the replayability and unpredictability of the roguelike

# Spelunky level generation



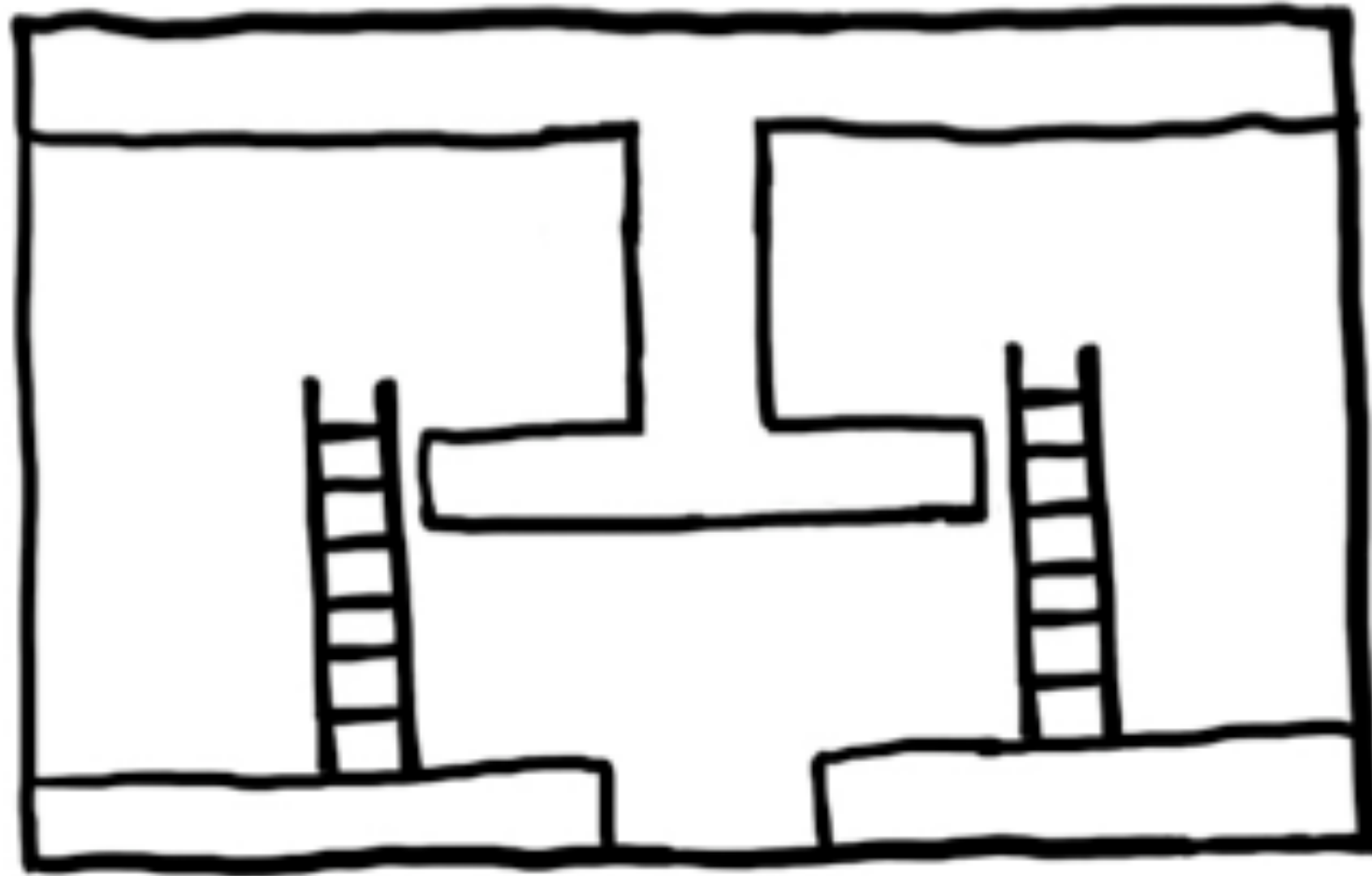
Each level is divided into a grid of 16 rooms.

# Spelunky level generation



A path is drawn from entrance at the top to exit at the bottom.

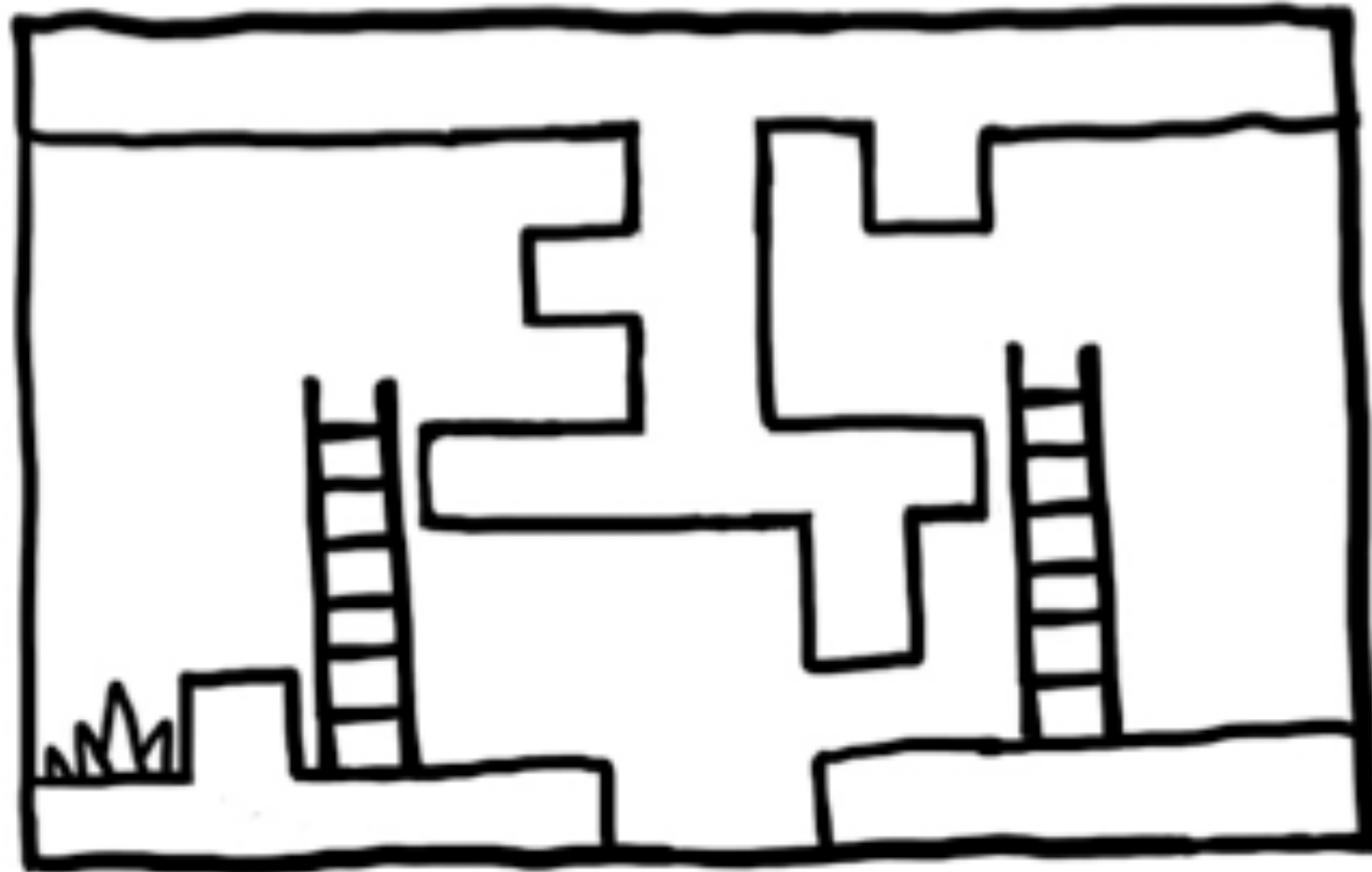
# Spelunky level generation



Each room is selected from a set of templates, so as to fit in the path drawn in the previous step (and also with the position in the level).

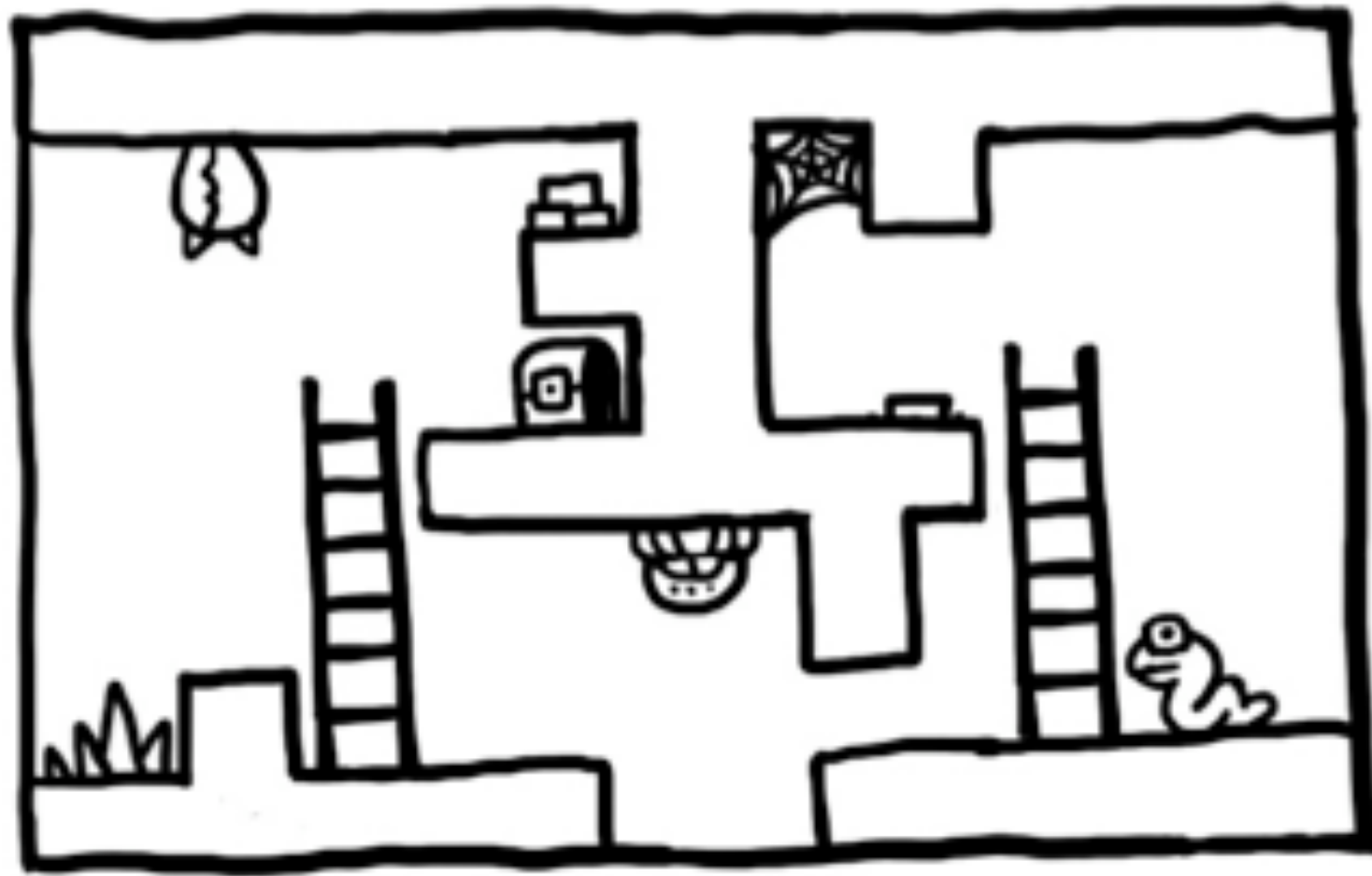


# Spelunky level generation



Randomised chunks in each room add variation.

# Spelunky level generation



Finally, critters, traps, treasures etc are added.

# Spelunky level generation

```
0000000011  
0060000L11  
0000000L11  
0000000L11  
0000000L11  
0000000011  
0000000011  
11111111
```

# The 2010 Mario AI Championship: Level Generation Track

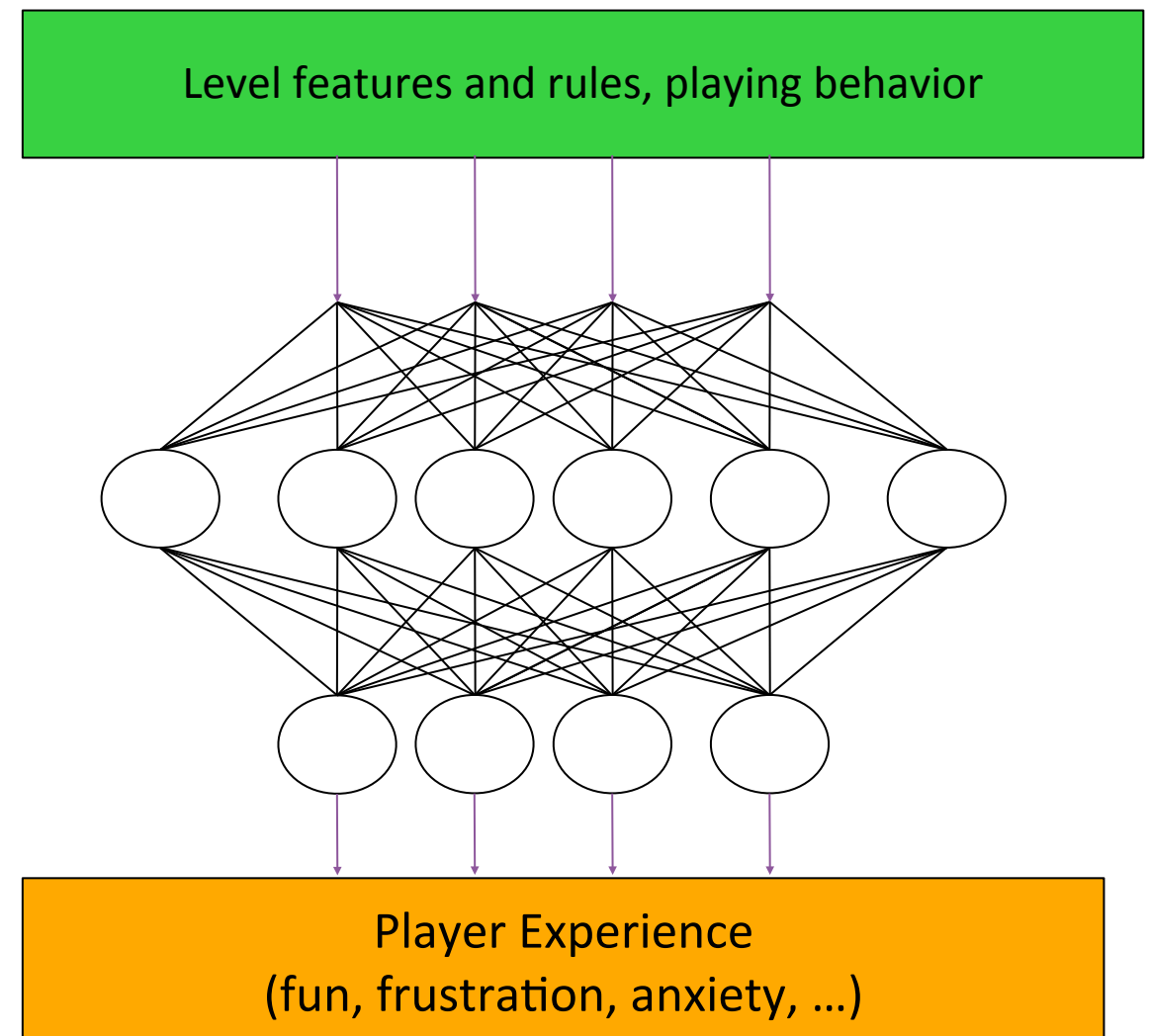
Noor Shaker, Julian Togelius, Georgios N. Yannakakis,  
Ben Weber, Tomoyuki Shimizu, Tomonori Hashiyama,  
Nathan Sorenson, Philippe Pasquier, Peter Mawhorter,  
Glen Takahashi, Gillian Smith and Robin Baumgarten  
IEEE TCIAIG, December 2011

# Infinite Mario Bros

- Open-source Java clone of Super Mario Bros (1? 3?)
- Developed by Notch (!)
- Infinite level generation - but levels are quite simple
- Developed into the *Mario AI Benchmark*

# Previous work: player level preferences

- Neuroevolutionary preference learning
- Player experience model 73-92%



C. Pedersen, J. Togelius, G. N. Yannakakis., **Modeling Player Experience for Content Creation** *IEEE TCIAG*, 2010

# The Mario AI Championship

- Based on different versions of the Mario AI Benchmark
- Gameplay track: controllers that play as well as possible
- Learning track: controllers that learn to play particular levels
- Turing track: controllers that play in a human-like manner
- Level generation track: level generators

# Level generation track 2010

- Six entries from three continents
- Each judge played a test level, and two levels generated by different generators, and then indicated a preference
- Generators were provided with information on how the judge played on the test level, to allow adaptation



# Ben Weber

- “Probabilistic multi-pass generator” (ProMP)
- A number of passes from left to right
- Adds a new feature every pass
- No adaptation

# Ben Weber

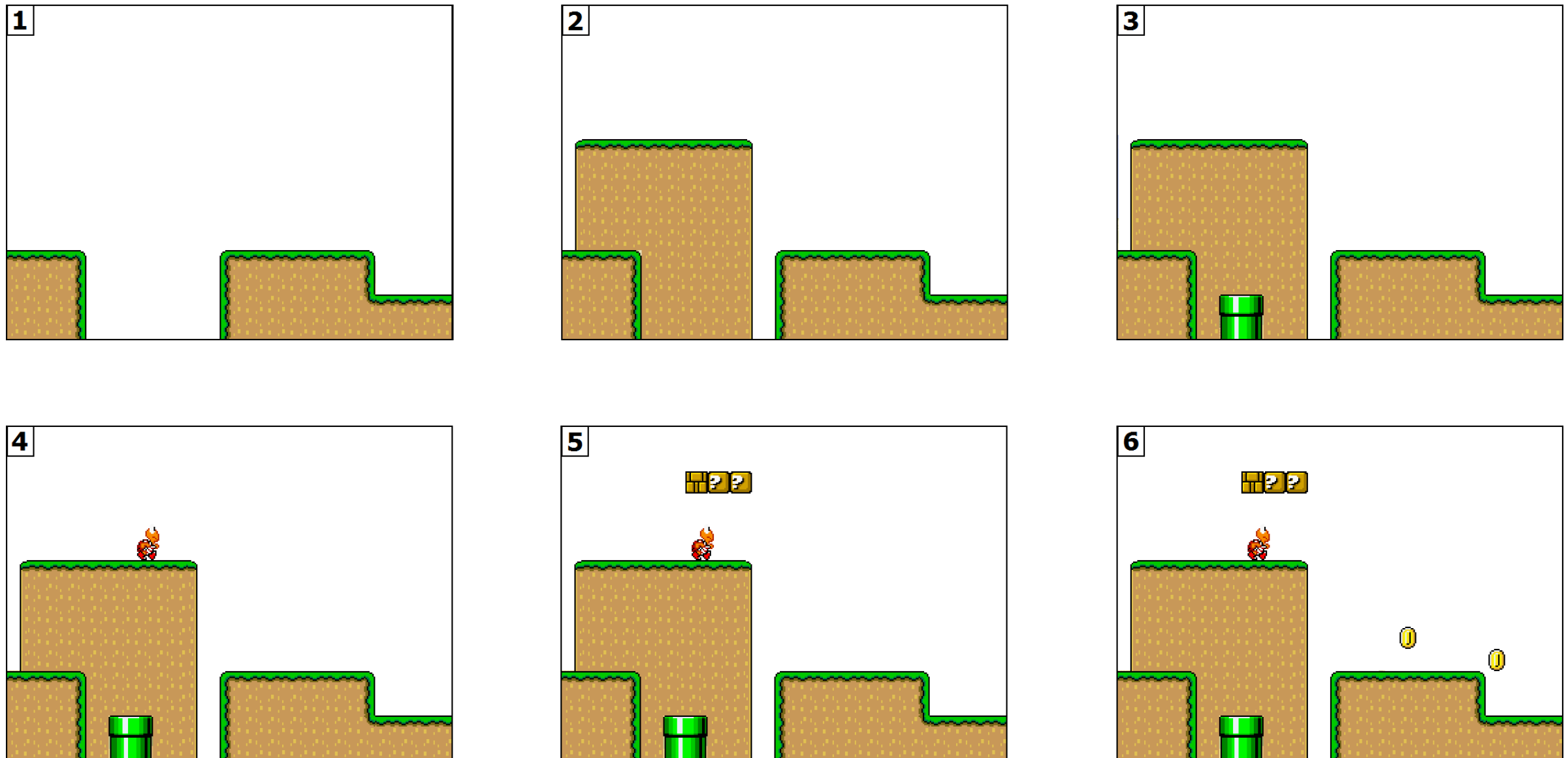


Fig. 1. Passes applied by Ben Weber's ProMP generator:(1) ground, (2) hills, (3) pipes, (4) enemies, (5) blocks, and (6) coins.

# Ben Weber

- Playability?

# This lab

- Get to know the Mario AI framework, especially the level generation version
- Build a simple level generator (re-implement any of the described, or do your own thing)
- Think about the strengths and weaknesses of your generator

Have you thought  
about the course  
project yet?