Game AI (?)

Simple Maze Generation

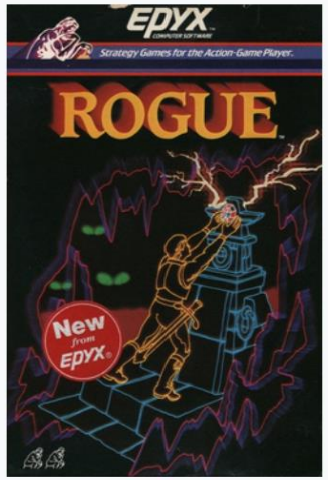
CS 415: Game Development

Professor Eric Shaffer

Source: Millington, Ian. AI for Games, Third Edition. CRC Press.

A Very Brief History of Maze Generation in Games

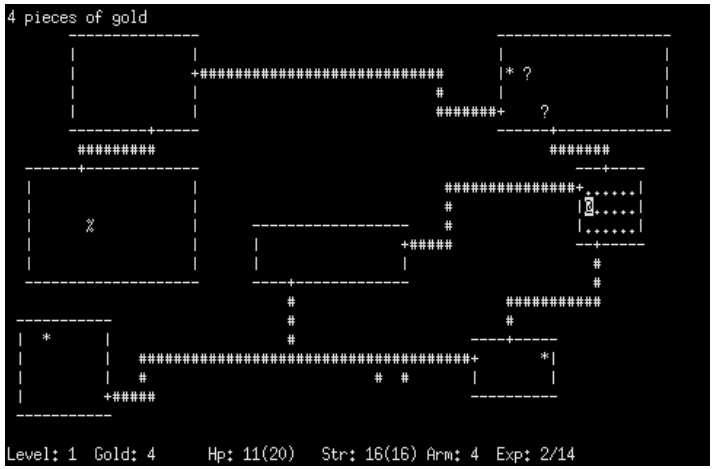
Rogue



Developer(s)	A.I. Design (Non-unix versions)
Publisher(s)	Epyx (Non-unix versions)
Designer(s)	Michael Toy Glenn Wichman Ken Arnold Jon Lane
Platform(s)	Amiga, Amstrad CPC, Atari 8-bit, Atari ST, Commodore 64, CP/M, DOS, Macintosh, TOPS-20, TRS-80 CoCo, Unix, ZX Spectrum
Release	1980
Genre(s)	Roguelike
Mode(s)	Single-player

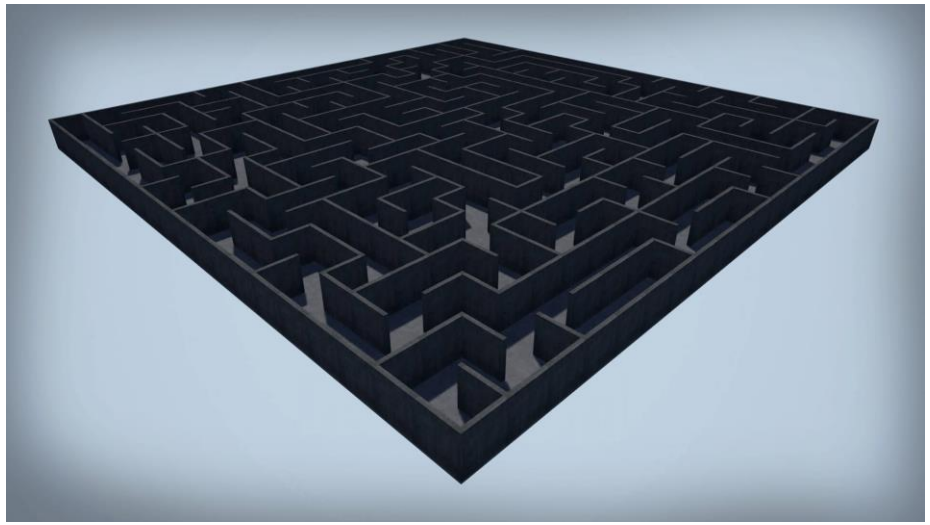
Rogue (also known as ***Rogue: Exploring the Dungeons of Doom***) is a dungeon crawling video game by Michael Toy and Glenn Wichman with later contributions by Ken Arnold. *Rogue* was originally developed around 1980 for Unix-based mainframe systems as a freely distributed executable. …Moreover, no game is the same as any previous one, as the dungeon levels, monster encounters, and treasures are procedurally generated for each playthrough.

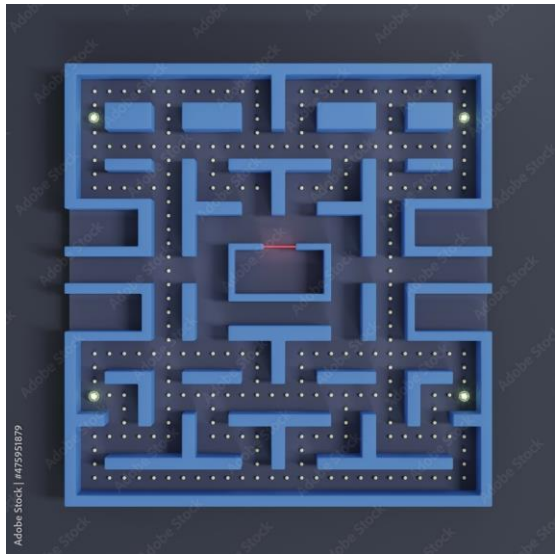
-Wikipedia



ILLINOIS

Maze Generation in Unreal





- Lots of visual variety is possible…be creative
- 2D that looks 3D would be interesting choice

ILLINOIS

Vocabulary

A ***dungeon*** is corridors and rooms

A ***maze*** is set of corridors with no rooms

- Formally, it's a maze only if it includes loops and/or dead ends
- A complicated path that zigzags without branching is ***labyrinth***
- In practice…this distinction is usually ignored

https://www.imdb.com › title

Labyrinth (1986) - IMDb

Sixteen-year-old Sarah is given thirteen hours to solve a **labyrinth** and rescue her baby brother Toby when her wish for him to be taken away is granted by ...

Directors: Jim Henson

★★★½☆ Rating: 7.3/10 · 137,527 votes

Depth First Backtracking

Start with a grid of cells; all of them initially unused.

Initially the entrance cell is excavated and this becomes the current cell.

At each iteration a random unused neighbor of the current cell is chosen.

- The current cell is connected to that neighbor
- The neighbor becomes the new current cell.
- If there is no unused neighbor, we return to the previous current cell.

Depth First Backtracking

When we return to the starting cell,
if it has no unused neighbors, then the algorithm is complete

The cells are stored on a stack.

When a neighbor becomes current cell, it is pushed to the top of the stack

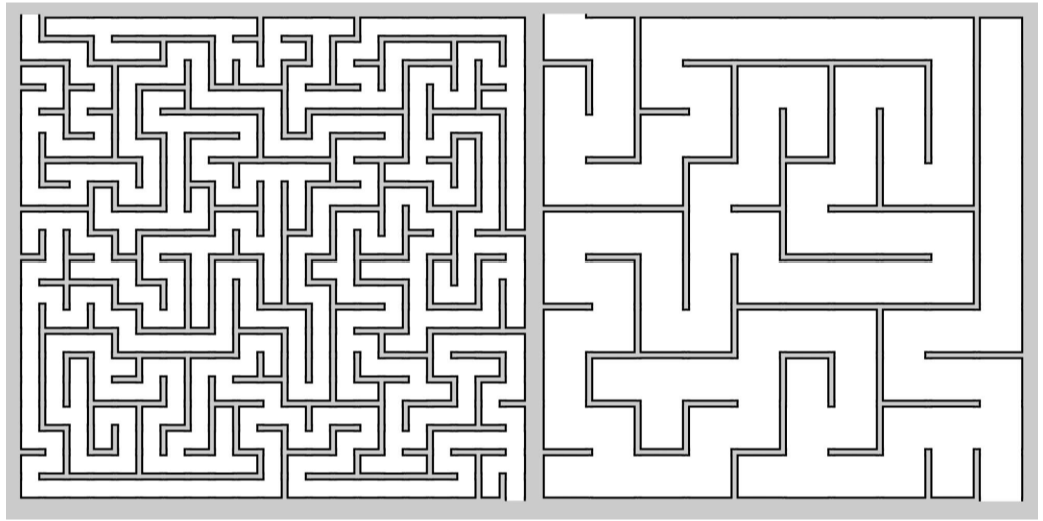
If the current cell has no unused neighbors, it is popped from the stack.

Exits!

Usually mazes have an exit as well as an entrance

- You pick the exit cell
- The exit appears as a normal unused cell while the algorithm is running
- Algorithm is guaranteed to spread the maze to all reachable locations in the grid
  - Including the exit.

Examples and Pseudo-Code



```
function maze(level: Level, start: Location):
    # A stack of locations we can branch from.
    locations = [start]
    level.startAt(start)

    while locations:
        current = locations.top()

        # Try to connect to a neighboring location.
        next = level.makeConnection(current)
        if next:
            # If successful, it will be our next iteration.
            locations.push(next)
        else:
            locations.pop()
```

ILLINOIS

More Pseudo-Code

```
class Level:
    function startAt(location: Location)
    function makeConnection(location: Location) -> Location
```

```
class Location:
    x: int
    y: int

class Connections:
    inMaze: bool = false
    directions: bool[4] = [false, false, false, false]

class GridLevel:
    # dx, dy, and index into the Connections.directions array.
    NEIGHBORS = [(1, 0, 0), (0, 1, 1), (0, -1, 2), (-1, 0, 3)]

    width: int
    height: int
    cells: Connections[width][height]

    function startAt(location: Location):
        cells[location.x][location.y].inMaze = true

    function canPlaceCorridor(x: int, y: int, dirn :int) -> bool:
        # Must be in-bounds and not already part of the maze.
        return 0 <= x < width and
               0 <= y < height and
               not cells[x][y].inMaze
```

```

    function makeConnection(location: Location) -> Location:
        # Consider neighbors in a random order.
        neighbors = shuffle(NEIGHBORS)

        x = location.x
        y = location.y
        for (dx, dy, dirn) in neighbors:

            # Check if that location is valid.
            nx = x + dx
            ny = y + dy
            fromDirn = 3 - dirn
            if canPlaceCorridor(nx, ny, fromDirn):
```

```

                # Perform the connection.
                cells[x][y].directions[dirn] = true
                cells[nx][ny].inMaze = true
                cells[nx][ny].directions[fromDirn] = true
                return Location(nx, ny)

        # null of the neighbors were valid.
        return null
```

ILLINOIS

More Pseudo-Code

```
25
26      function makeConnection(location: Location) -> Location:
27          # Consider neighbors in a random order.
28          neighbors = shuffle(NEIGHBORS)
29
30          x = location.x
31          y = location.y
32          for (dx, dy, dirn) in neighbors:
33
34              # Check if that location is valid.
35              nx = x + dx
36              ny = y + dy
37              fromDirn = 3 - dirn
38              if canPlaceCorridor(nx, ny, fromDirn):
```

```
39
40                  # Perform the connection.
41                  cells[x][y].directions[dirn] = true
42                  cells[nx][ny].inMaze = true
43                  cells[nx][ny].directions[fromDirn] = true
44                  return Location(nx, ny)
45
46          # null of the neighbors were valid.
47          return null
```

```
5   class Connections:
6       inMaze: bool = false
7       directions: bool[4] = [false, false, false, false]
8
9   class GridLevel:
10      # dx, dy, and index into the Connections.directions array.
11      NEIGHBORS = [(1, 0, 0), (0, 1, 1), (0, -1, 2), (-1, 0, 3)]
```

ILLINOIS

Performance

O(n) in execution time

- n is the number of cells in the grid

O(k) in space

- k is the longest path in the maze
  - usually this is not the route from the goal to the endpoint
  - it is possible that k ∝ n, if the grid is almost a line, it is
  - more common that k ∝ log n, and performance may be given as O(log n) in space.

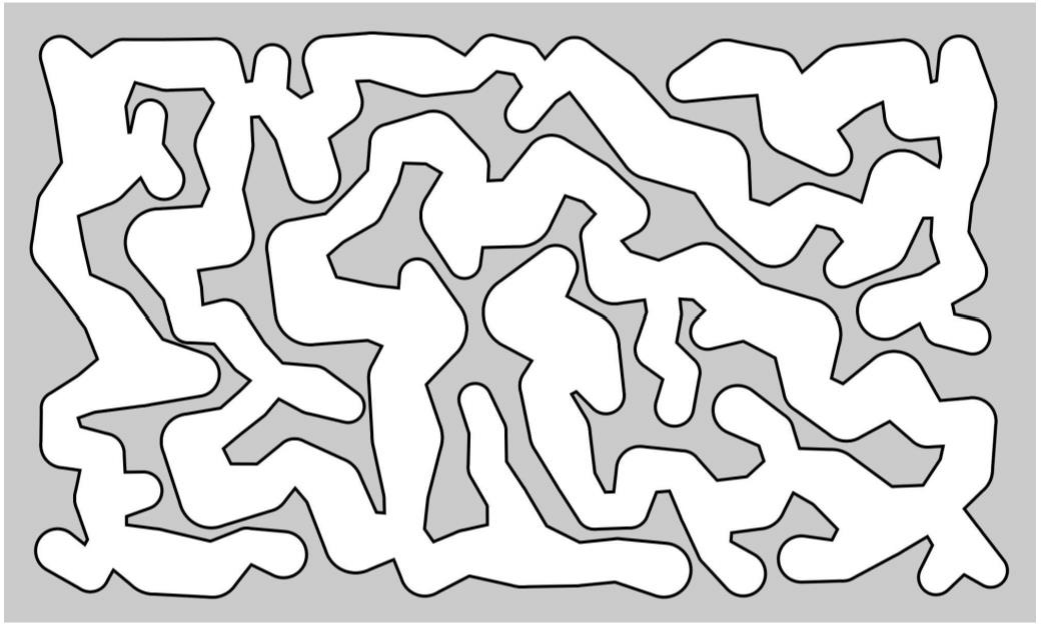
What data structure determines k? Why is this structure typically not O(n) in size?

Caves

What if we didn’t use a grid?

Algorithm still works!

Can move in a random direction in space instead…to generate something like this:



ILLINOIS

Pseudo-Code

```
class Location:
    x: float
    y: float
    r: float
    passageFrom: Location = null

class CavesLevel:
    locations: Location[]
    collisionDetector # Checks whether a cave is blocked.

    function startAt(location: Location):
        locations.push(location)
        collisionDetector.add(location)

    function makeConnection(location: Location) -> Location:
        # Sstart checking in a random direction.
        initialAngle = random( ) * 2 * pi

        # The next cave will have a random size.
        nextRadius = randomRange(MIN_RADIUS, MAX_RADIUS)
        offset = r + nextRadius

        x = location.x
        y = location.y
        r = location.r
        for i in 0..ITERATIONS:

            # Try to place the next cave.
            theta = 2 * pi * i / ITERATIONS + initialAngle
            tx = x + offset * cos(theta)
            ty = y + offset * sin(theta)

            # If there is space in the collision detector, add
            # it to the list of cave locations.
            candidate = new Location(tx, ty, nextRadius, location)
            if collisionDetector.valid(candidate):
                collisionDetector.add(candidate)
                locations.push(candidate)
                return candidate

        return null
```

Need to handle entrance and exit somehow…maybe find closest caves to randomly picked corners…

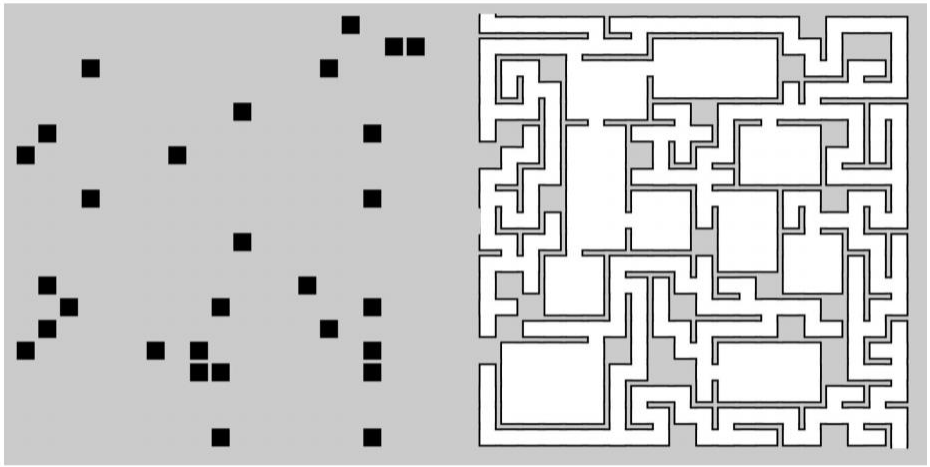
ILLINOIS

Rooms

The algorithm is flexible…initial state can be different

Can start out with areas that

- are unreachable
- contain resources
- add interest to the maze structure



What do the black dots represent?

ILLINOIS

Rooms

Nothing restricts locations to representing sections of corridor.

As long as the level data structure can treat them as a single location

- rooms can be placed
- the maze generated to connect them

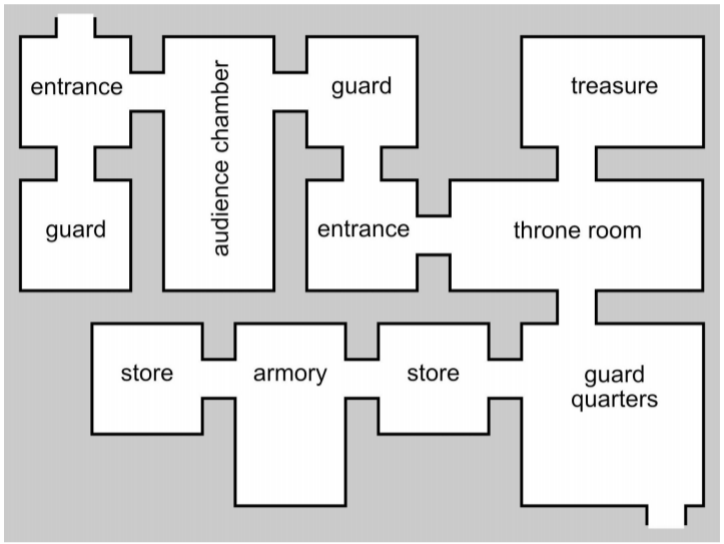
The level data structure generalizes to graph instead of a grid…

Room Pseudo-Code

```
class Room:
    width: int
    height: int

class GridLevelWithRooms extends GridLevel:
    unplacedRooms: Room[]

    function canPlaceRoom(room: Room, x: int, y: int) -> bool:
        inBounds = (
            0 <= x < (width - room.width) and
            0 <= y < (height - room.height))
        if not inBounds:
            return false

        for rx in x..(x + room.width):
            for ry in y..(y + room.height):
                if cells[rx][ry].inMaze:
                    return false

        return true

    function addRoom(room: Room, location: Location):
        for x in location.x..(location.x + room.width):
            for y in location.y..(location.y + room.height):
                cells[x][y].inMaze = true
                # If you're using connections to determine where walls
                # are drawn, then set all connections in the room.

    function makeConnection(location: Location) -> Location:
        # Try to fit a room.
        if unplacedRooms and random() < CHANCE_OF_ROOM:
            x = location.x
            y = location.y

            # Choose a room and work out its origin.
            room = unplacedRooms.pop()
            (dx, dy, dirn) = randomChoice(NEIGHBORS)
            nx = x + dx
            ny = y + dy
            if dx < 0: nx -= room.width
            if dy < 0: ny -= room.height

            if canPlaceRoom(room, nx, ny):
                # Fill the room.
                addRoom(room)

                # Perform the connection.
                cells[x][y].directions[dirn] = true
                cells[x + dx][y + dy].directions[3 - dirn] = true

                # Return nothing if rooms aren't part of the main
                # maze. Otherwise you might return the room exit.
                return null

        # Otherwise go through the neighbors as before.
        return super.makeConnection(location)
```

ILLINOIS

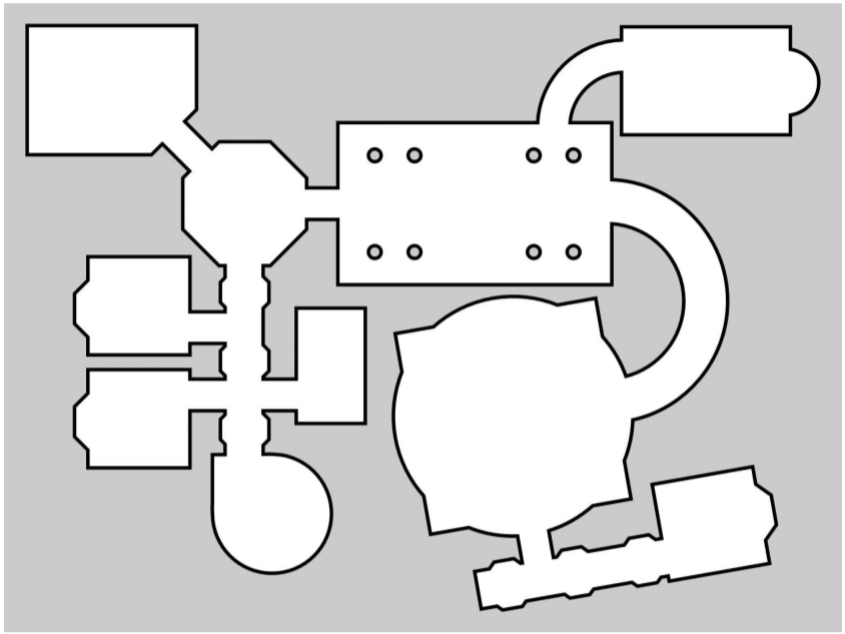
Only Rooms



Instead of the grid representing square sections of corridor, each location may be a room. The maze then consists of rooms with adjoining doors. This is similar to the dungeons in a game such as the original Zelda

ILLINOIS

Pre-fabricated Rooms



There is nothing that requires rooms to be placed on a grid.

If we have a set of prefabricated rooms or corridor sections, each with a list of possible exits, an implementation of the level interface can simply check, for each exit of the current location, whether a room can fit.

ILLINOIS