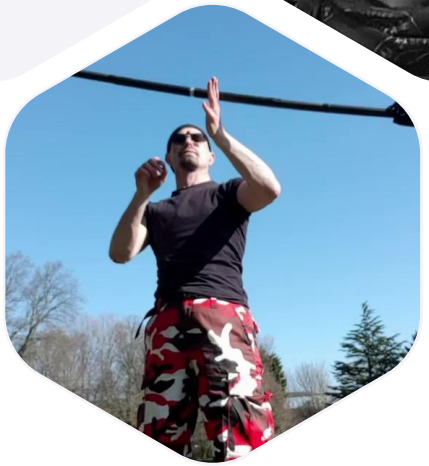
All work and no play makes Gophers...

... something something

https://github.com/necrophonic/3d-gopher-maze

WHO AM I?



John Gregory

Senior Software Engineer at Admiral Money, artist, fire spinner and proud Gopher

linkedin.com/in/necrophonic

3D Gopher Maze

The hows and whys of writing a retro styled Gopher game

github.com/necrophonic/3d-gopher-maze

Coming up!

- Getting here - history and inspiration
- Designing the game - rules, constraints and setting goals
- Building the game - solving problems and writing code

1 GETTING HERE

History and inspiration

Humble beginnings

My first computer was
my brother's **Dragon 32**

- Released in 1982
- Made in Swansea, Wales 
- 32K of memory
- 0.89MHz, 8 bit CPU
- Cassette tape storage

https://en.wikipedia.org/wiki/Dragon_32/64

First steps

Getting started with BASIC on the Dragon

SOME CLASSIC GAMES

- Jet Set Willy
- Chucky Egg
- Donkey King*
- *... and many more!*

*not a typo!

https://www.retrogamer.net/top_10/top-ten-dragon-32-games/

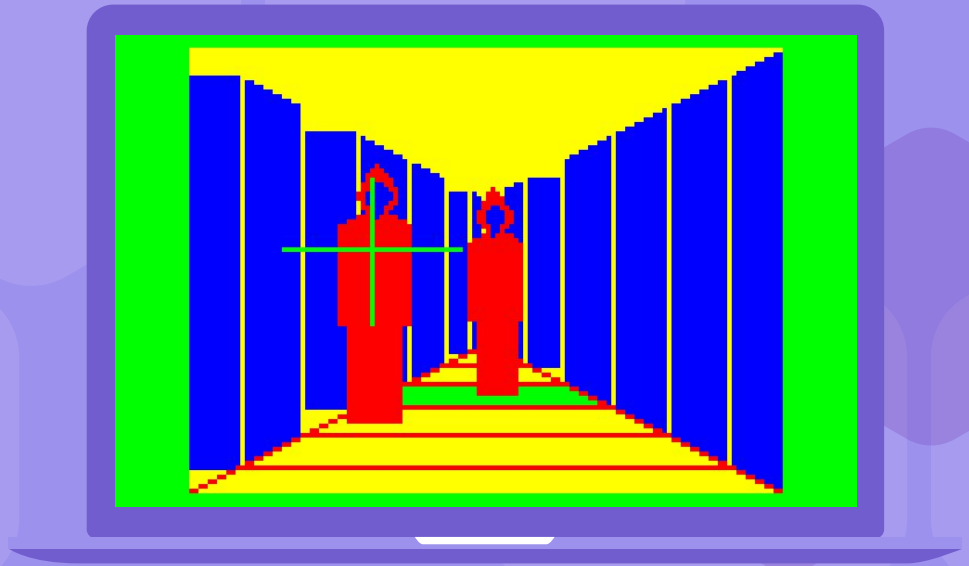
Doing a lot with a little

Doing clever things with limited resources

Inspiration:

Phantom Slayer

- First person **3D maze** game
- From **Microdeal** in **1982**



https://en.wikipedia.org/wiki/Phantom_Slayer_(video_game)

2

DESIGNING THE GAME

Rules, constraints and setting goals

Where to start?

First need to break down goals and constraints

GOALS AND CONSTRAINTS

Some decisions to reduce to problem space:

- Pseudo 3D - not attempting “true” 3D rendering or lighting
- Grid based movement - can limit the rendered view, and simplify navigation
- Run in terminal - only using terminal for controls and rendering
- Turn based - wait for player input

3

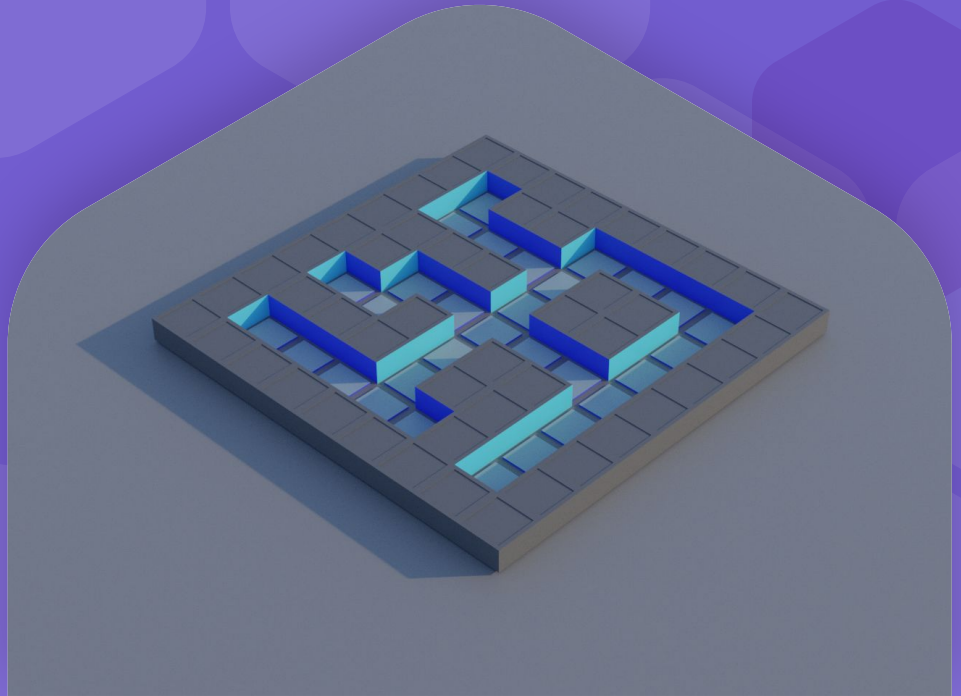
BUILDING THE GAME

Solving problems and writing code

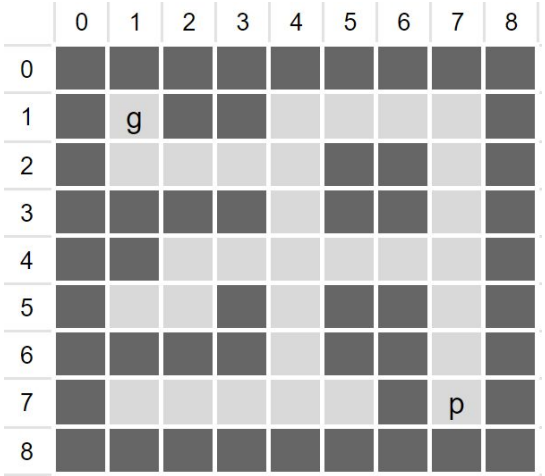
The Maze

Get lost!

But in a good way...



Get on the grid



- Maze as simple grid
- Represent as 2D array
- Zero indexed top left

```
type (
  spaceType uint8
  mazeDefinition [][]spaceType
)

var mazes = []mazeDefinition{
  {
      // Simple maze
      {'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'},
      {'X', 'g', 'X', 'X', ' ', ' ', ' ', ' ', 'X'},
      {'X', ' ', ' ', ' ', ' ', 'X', 'X', ' ', 'X'},
      {'X', 'X', 'X', 'X', ' ', 'X', 'X', ' ', 'X'},
      {'X', 'X', ' ', ' ', ' ', ' ', ' ', ' ', 'X'},
      {'X', ' ', ' ', 'X', ' ', 'X', 'X', ' ', 'X'},
      {'X', 'X', 'X', 'X', ' ', 'X', 'X', ' ', 'X'},
      {'X', ' ', ' ', ' ', ' ', ' ', 'X', 'p', 'X'},
      {'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'},
  },
}
```

Getting to the point

When the game starts, the selected map gets imported into a `Maze`.

In addition this also stores the `panel` set used to display the view

(more on that later!)

Given a point (`struct{X,Y int}`), we can fetch any `space` from the grid.

```
// Space only contains its type for now, but could
// be extended in the future (items present etc)
type space struct { t uint8 }

// Pseudo "screen pixel" matrix for display
elements
type pixelMatrix [][]uint8

// Maze stores the map grid definition and the
// set of display panels for rendering
type Maze struct {
    grid   [][]space
    panels map[int]map[string]pixelMatrix
}

// Fetch a space from the maze given a point
// ( struct{X,Y int} )
func (m *Maze) getSpace(p Point) space {
    return m.grid[p.Y][p.X]
}
```

A-mazing!

Loop through the **maze definition** (row, then column).

If the space is a special type (player or gopher), then note the position.

Then import the space type.

```
func (g *Game) importMaze(m mazeDefinition) error {
    // ... skipped var initialisation
    for y := range m {
        newMaze[y] = make([]space, width)
        for x, sp := range m[y] {

            switch sp {
            case SpacePlayerStart:
                playerStarts = append(playerStarts, NewPointInt(x, y))
                sp = SpaceEmpty
                playerFound = true

            case SpaceGopherStart:
                g.gopher.p = NewPointInt(x, y)
                gopherStarts = append(gopherStarts, NewPointInt(x, y))
                sp = SpaceEmpty
                gopherFound = true
                g.items = append(g.items, g.gopher)
            }

            newMaze[y][x] = space{
                t: spaceType(sp),
            }
        }
    }
    // ... skipped error if player and gopher not found
    // ... skipped random choosing of start points
    g.m.grid = newMaze
    return nil
}
```

The Player

Getting yourself into the game!

In a bit of a state!

Just a simple `struct`!

- **Current point** - defined by an x, y coordinate of a space in the maze grid
- **Orientation** - representing a cardinal direction (North, South, East, West) as an enumerated byte (`N=0`, `E=1`, `S=2`, `W=3`)

```
type Point struct {
  X, Y int
}

const (
    N byte = 0
    E      = 1
    S      = 2
    W      = 3
)

type Player struct {
  p Point
  o byte
}
```

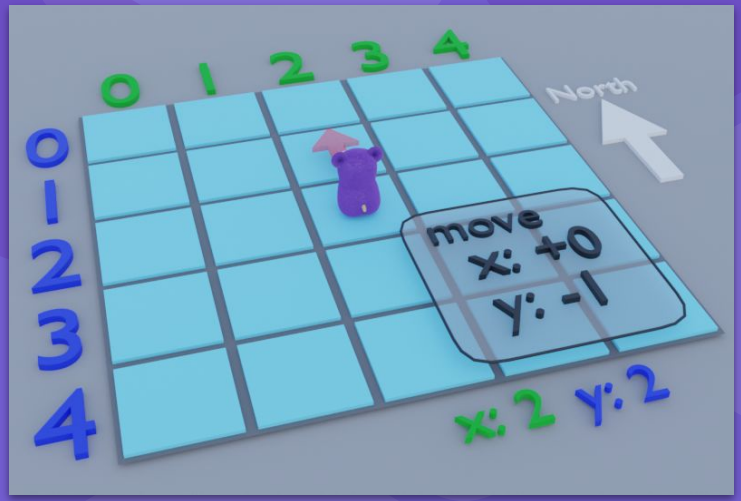
Move it!

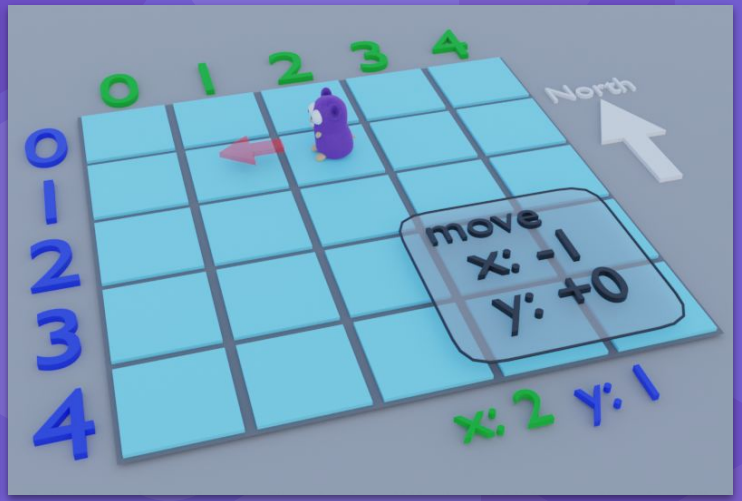
Can move by applying a **move vector**

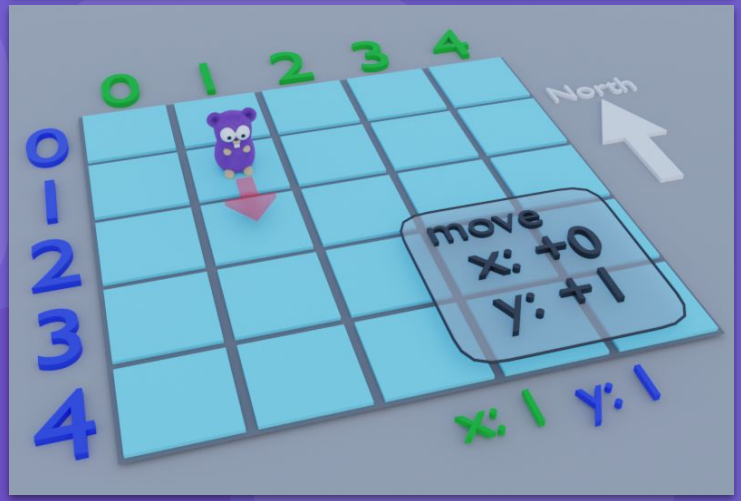
Each direction has its own vector:

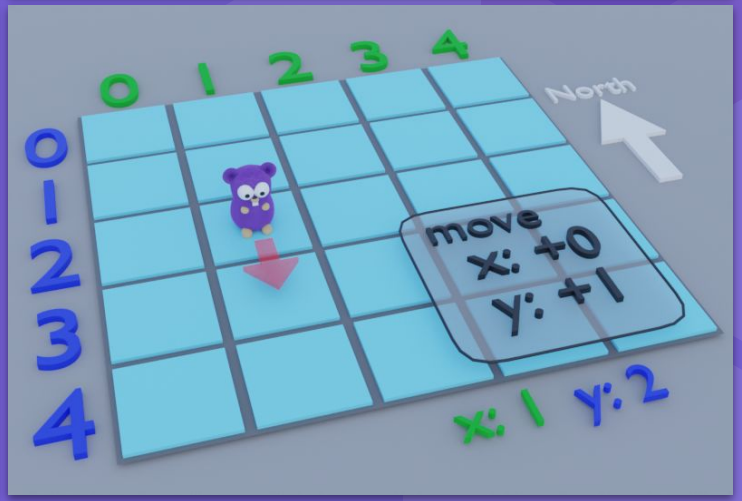
```
North: { x+0, y-1 }
South: { x+0, y+1 }
East : { x+1, y+0 }
West : { x-1, y+0 }
```

Apply appropriate vector depending on the direction we're facing.









There and back

- **Forward** is the basic application of the vector.
- **Backwards** is almost identical, except multiply the vector values by `-1`

In both cases we perform a **ghost move** to test if we've hit a wall.

When moving **forward** we also test to see if we've found the Gopher!

```
func (g *Game) moveForward() {
    if g.isMoveToWall(g.move.x, g.move.y) {
        g.Msg = "Can't go that way!"
        return
    }
    g.player.p.X += g.move.x
    g.player.p.Y += g.move.y

    if g.gopher.p.Is(g.player.FrontPoint()) {
        g.state = sWin
    }
}

func (g *Game) moveBackwards() {
    if g.isMoveToWall(g.move.x*-1, g.move.y*-1) {
        g.Msg = "Can't go that way!"
        return
    }
    g.player.p.X += (g.move.x * -1)
    g.player.p.Y += (g.move.y * -1)
}
```

Round & about

Turning resets our orientation and sets the move vector.

- Right is clockwise

```
N -> E -> S -> W
```

- Left is anticlockwise

```
N -> W -> S -> E
```

```
func (g *Game) rotateRight() {
    switch g.player.o {
    case 'n':
        g.player.o = 'e'
        g.move.x = 1
        g.move.y = 0
    case 'e':
        g.player.o = 's'
        g.move.x = 0
        g.move.y = 1
    case 's':
        g.player.o = 'w'
        g.move.x = -1
        g.move.y = 0
    case 'w':
        g.player.o = 'n'
        g.move.x = 0
        g.move.y = -1
    }
}
```

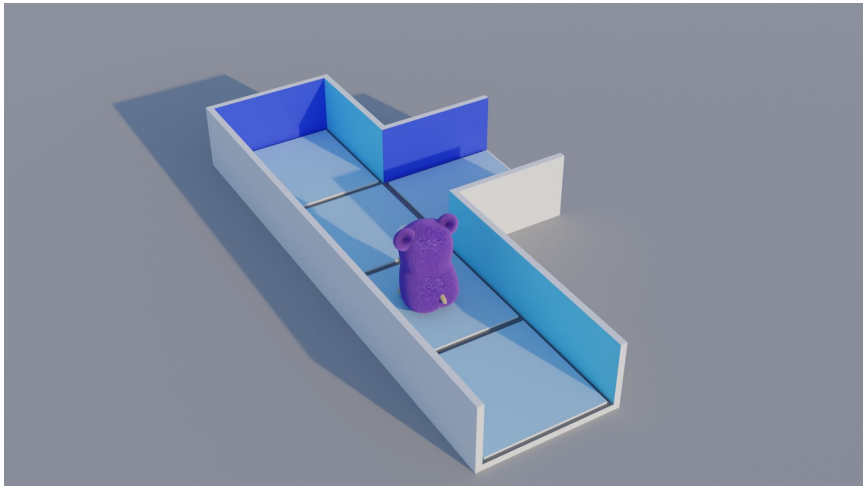
```
func (g *Game) rotateLeft() {
    switch g.player.o {
    case 'n':
        g.player.o = 'w'
        g.move.x = -1
        g.move.y = 0
    case 'w':
        g.player.o = 's'
        g.move.x = 0
        g.move.y = 1
    case 's':
        g.player.o = 'e'
        g.move.x = 1
        g.move.y = 0
    case 'e':
        g.player.o = 'n'
        g.move.x = 0
        g.move.y = -1
    }
}
```

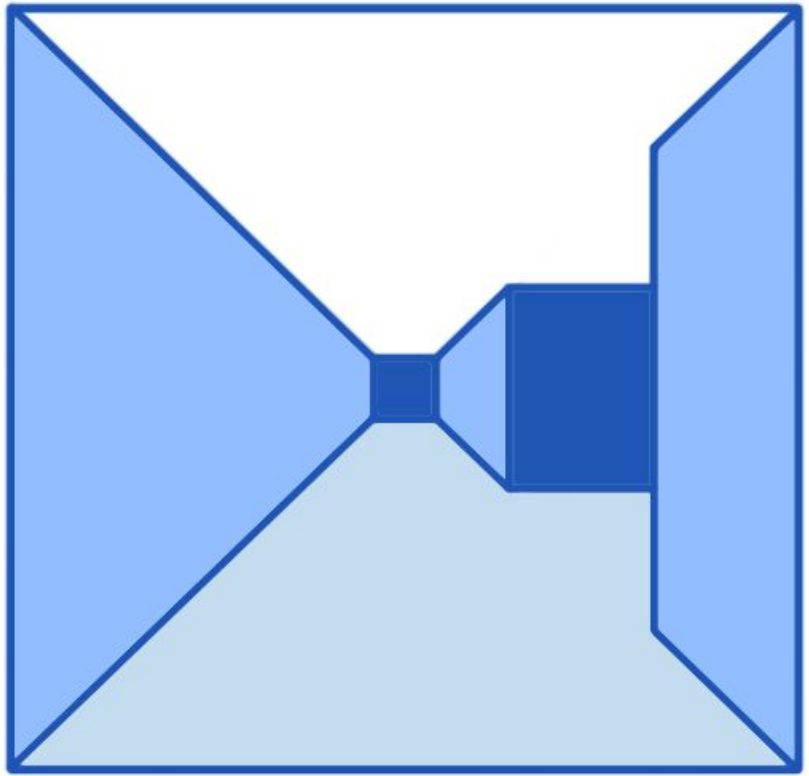
The View

Taking a good hard look at where we're going

Start simple

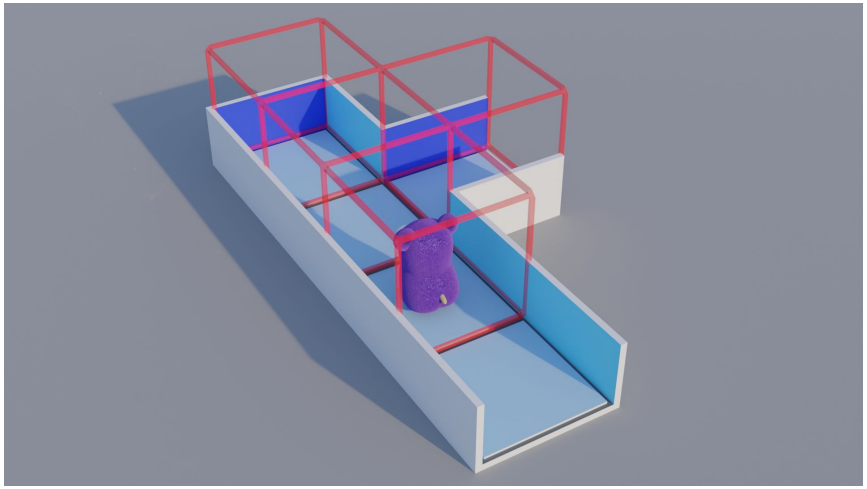
Consider a simple corridor with a Gopher in it

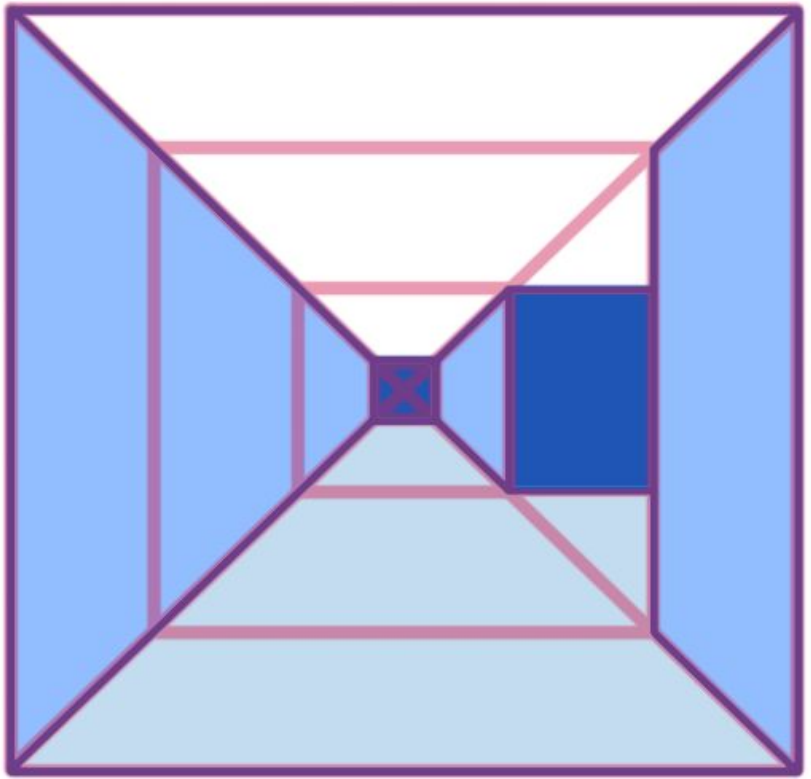




Box it up

Add the visible grid to the view



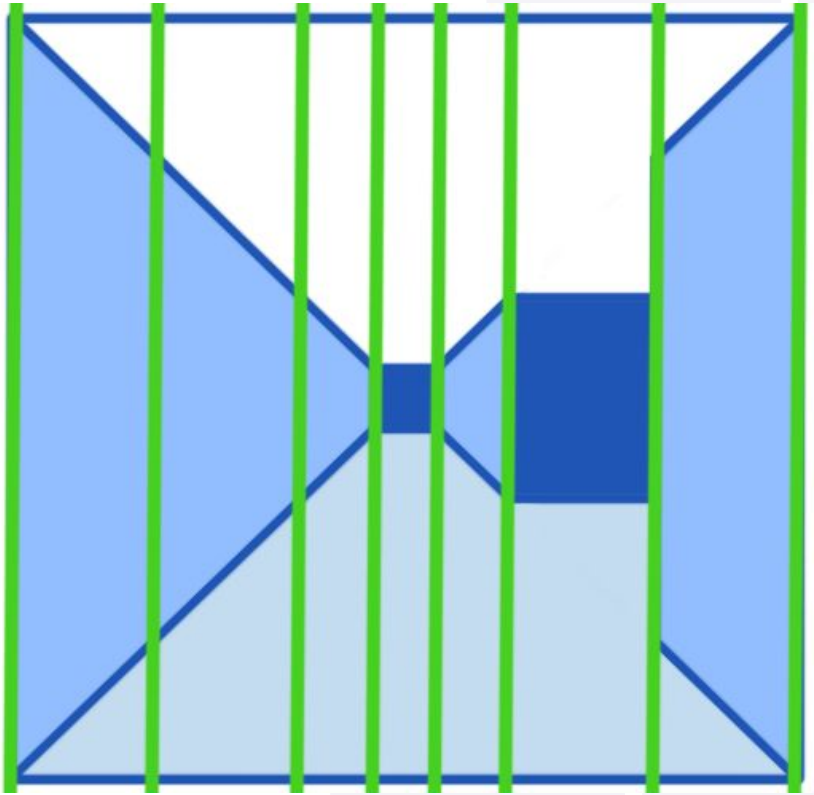


Slice ‘n dice

Can now split the view into vertical slices

- Can now render any view with a set of vertical panels
- Only need 13 panels for any view in the maze!*

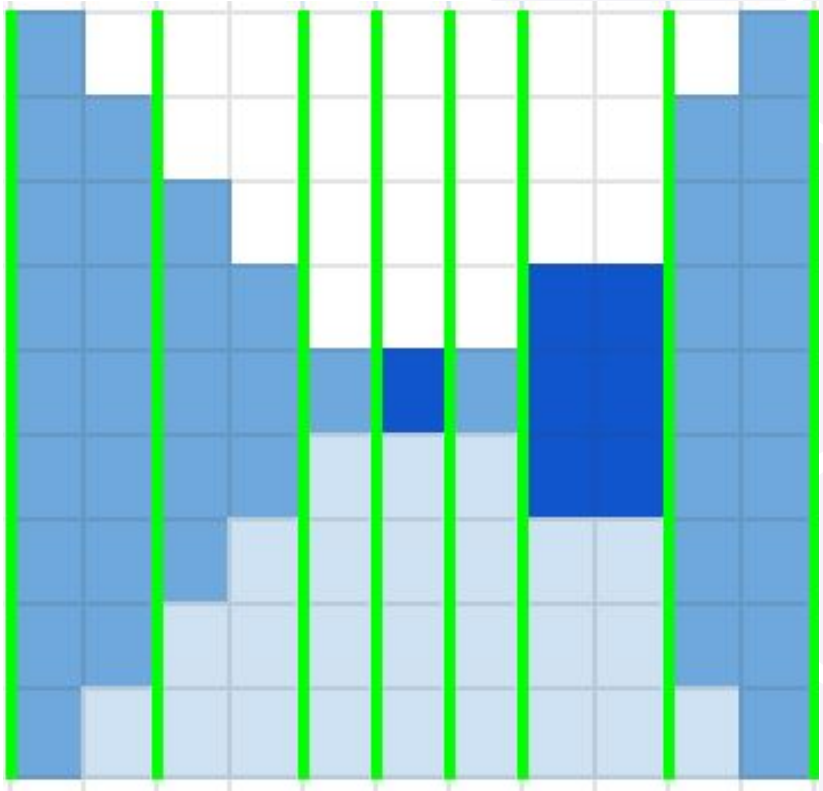
*Could be optimised further by mirroring, as currently left and right versions are defined separately.



Pixel perfect

Map the view onto a 11 x 9 “pixel” matrix

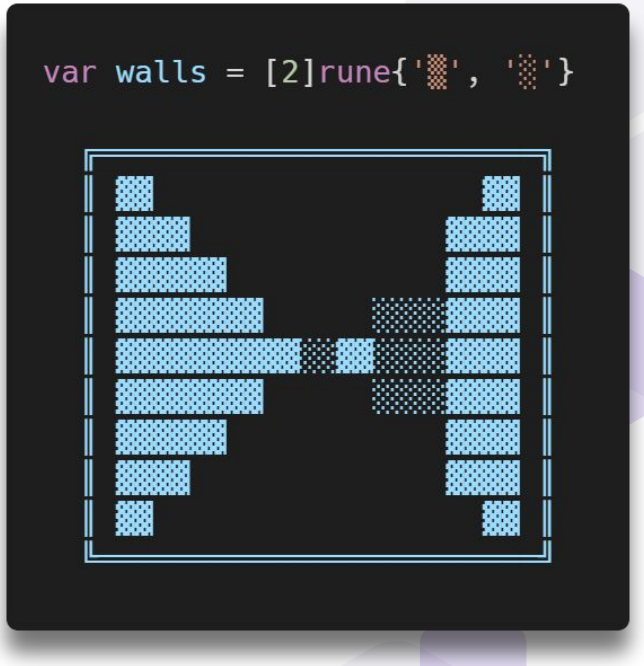
- Using character rows and columns in the terminal to simulate a bitmap
- Slice width differences limited by available resolution!



ASCII anything

Take advantage of the ASCII character set for drawing

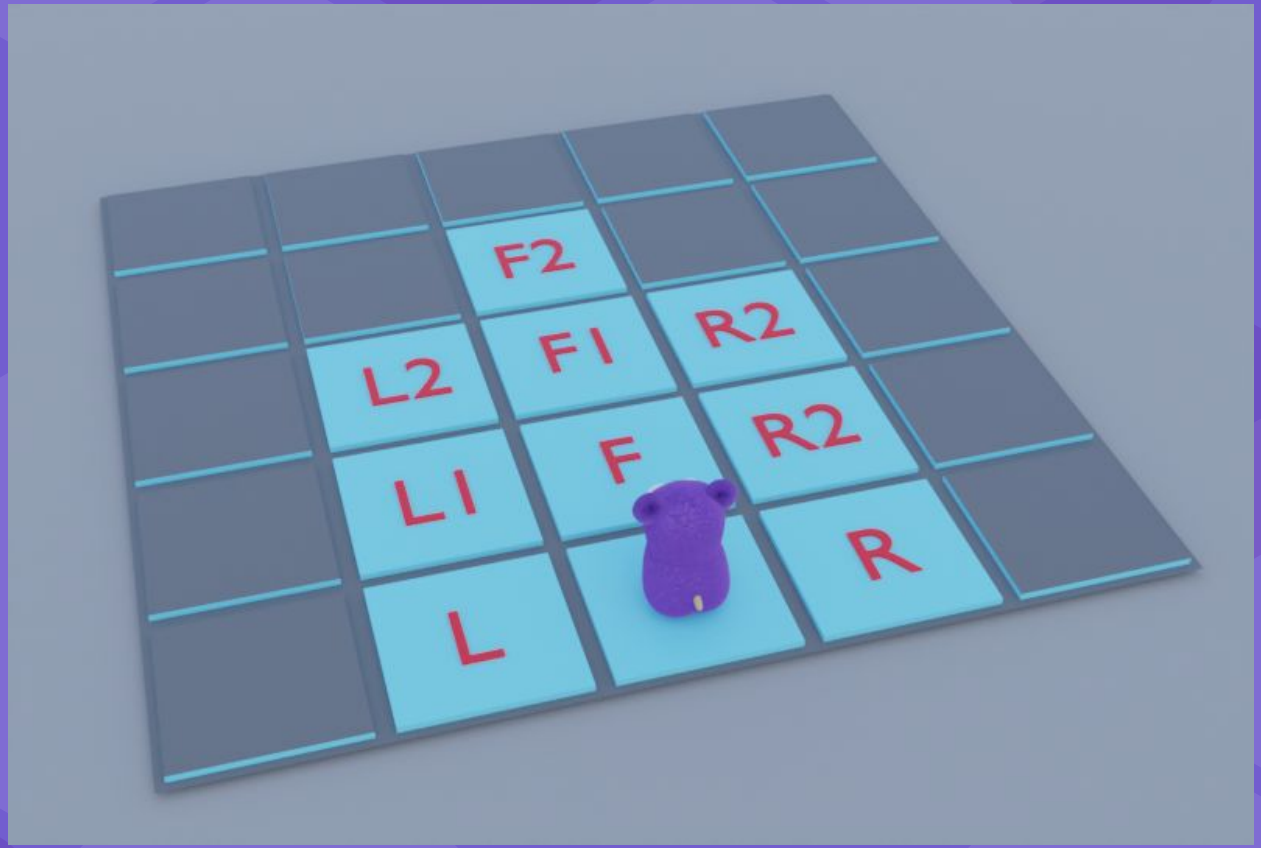
- Can then use a simple `walls[value % 2]` function to alternate colours
- “Render” to the view with a newline delimited string



Look out!

To render forward view, only need to consider a small set of potential spaces.

For each point, determine if a wall or empty space, starting **near** (L,F,R), then **middle** (L1, F1, R1), then **far** (L2,F2,R2)



Dealing with different architecture

```go
package terminal

import ( ... )

// Clear clears the terminal
func Clear() {
    cmd := exec.Command("cmd", "/c", "cls")
    cmd.Stdout = os.Stdout
    cmd.Run()
}
```

`terminal_windows_amd64.go`

Only compiles when architecture **is** Windows amd64

```go
// +build !windows

package terminal

import ( ... )

// Clear clears the terminal
func Clear() {
    cmd := exec.Command("clear")
    cmd.Stdout = os.Stdout
    cmd.Run()
}
```

`terminal.go`

Only compiles when architecture is **not** Windows

4

CONCLUSIONS

Some takeaways

Have fun!

You don't always have to be building the **next big thing**!

ANY QUESTIONS?

THANKS!

github.com/necrophonic/3d-gopher-maze

Find me at:

- @n3crophonic
- necrophonic
- in/necrophonic
- cafpanda

Credits

Special thanks to all the people who made and released these awesome resources for free:

- Presentation template by SlidesCarnival
- Icons by Font Awesome
- Dragon history by World Of Dragon
- Gopher images by Ashley Willis (github) and gopherize.me
- Maze renders created in Blender

References

- https://en.wikipedia.org/wiki/Phantom_Slayer_(video_game)
- https://www.retrogamer.net/top_10/top-ten-dragon-32-games/