Урок Спринт 1.8

Создание игры

Сначала нам нужно как-то представить нашу сетку. Для этого мы создадим новый тип данных и назовем его `World`. Каждая клетка должна быть представлена двумя координатами, поэтому мы воспользуемся двумерным слайсом, или слайсом из слайсов:

```
type World struct {
    Height int // высота сетки
    Width  int // ширина сетки
    Cells  [][]bool
}
```

Значений у клетки может быть только два — живая или мёртвая, поэтому тип данных в клетках — `bool`.

Мы создали новый тип для описания игры. Теперь нужно выделить память под сетку:

```
func NewWorld(height, width int) *World {
    // создаём тип World с количеством слайсов hight (количество строк)
    cells := make([][]bool, height)
    for i := range cells {
        cells[i] = make([]bool, width) // создаём новый слайс в каждой строке
    }
    return &World{
        Height: height,
        Width:  width,
        Cells:  cells,
    }
}
```

Вот какие методы понадобятся нам для «Жизни»:

– `Neighbours(x, y int)` — определяет количество живых соседей у клетки

– `Next(x, y int)` будет использовать `Neighbours`, чтобы вычислить следующее состояние клетки на основе текущего и количества живых соседей.

Начнём с метода `Next`:

```
func (w *World) Next(x, y int) bool {
    n := w.Neighbours(x, y)       // получим количество живых соседей
    alive := w.Cells[y][x]       // текущее состояние клетки
    if n < 4 && n > 1 && alive { // если соседей двое или трое, а клетка жива
        return true // то следующее состояние — жива
    }
    if n == 3 && !alive { // если клетка мертва, но у неё трое соседей
        return true // клетка оживает
    }

    return false // в любых других случаях — клетка мертва
}
```

После того, как мы научились определять следующее состояние одной клетки, нам нужно применить этот же метод на сетку целиком:

```
func NextState(oldWorld, newWorld *World) {
// переберём все клетки, чтобы понять, в каком они состоянии
    for i := 0; i < oldWorld.Height; i++ {
        for j := 0; j < oldWorld.Width; j++ {
        // для каждой клетки получим новое состояние
            newWorld.Cells[i][j] = oldWorld.Next(j, i)
        }
    }
```

```
}
```

Итак, мы подготовили функцию, которая изменяет состояние клеток. Но для того, чтобы игра началась, нужно какое-то исходное состояние. Мы можем задать его вручную или написать отдельный метод Seed. Он заполнит сетку живыми клетками в случайном порядке:

```
func (w *World) Seed() {
// снова переберём все клетки
    for _, row := range w.Cells {
        for i := range row {
        //rand.Intn(10) возвращает случайное число из диапазона от 0 до 9
            if rand.Intn(10) == 1 {
                row[i] = true
            }
        }
    }
}
```

Теперь самое время написать основную функцию:

```
func main() {
    // зададим размеры сетки
    height := 10
    width := 10
    // объект для хранения текущего состояния сетки
    currentWorld := NewWorld(height, width)
    // объект для хранения следующего состояния сетки
    nextWorld := NewWorld(height, width)
    // установим начальное состояние
    currentWorld.Seed()
    for { // цикл для вывода каждого состояния
        // выведем текущее состояние на экран
        fmt.Println(currentWorld)
        // рассчитываем следующее состояние
        NextState(currentWorld, nextWorld)
        // изменяем текущее состояние
        currentWorld = nextWorld
        // делаем паузу
        time.Sleep(100 * time.Millisecond)
        // специальная последовательность для очистки экрана после каждого шага
        fmt.Print("\033[H\033[2J")
    }
}
```

Готово, наша «Жизнь» ожила! Теперь можем с гордостью наблюдать, как она развивается.

Стейт-машина, или конечный автомат, который мы реализовали, вам ещё неоднократно потребуется. Подробно работу с состояниями и переходами мы разберём в следующем модуле, посвящённом конкурентному программированию. А пока — давайте решим пару задач!

Далее