Procedural Generation: Markov Chains

Nico, Tyler

Topics

Spawning the start room

- Generating doors
- Room spawning availability check
- Room transition mechanics

Randomly generating rooms

- Tagging rooms with attributes
- Determining room metrics from transition matrix

Markov chains

- History of Markov Chains
- How we implement Markov Chains

Markov Chains

- State based process
- Stochastic - Randomly Determined but statistically analyzable
- Probabilities based on current state and a transition matrix

How do we store rooms?

An Array!

An array of **tuples** will allow us to store room information without **physically** storing tiles.

0	**2**	**4**	**6**	**8**	**10**	**12**	**14**

This helps save on space & overhead!

Saving Room Dimensions

When the start room is generated at index 0, we send into that index a tuple with:

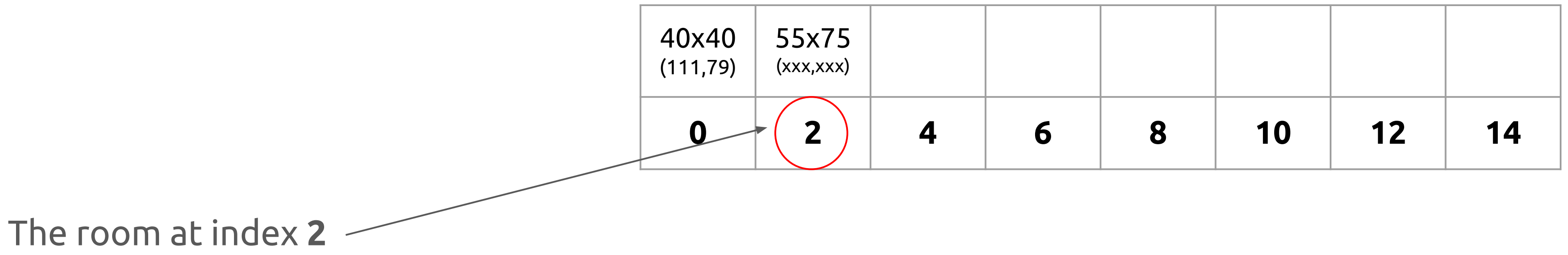
- Room size (width, length)
- Start point (x,y)

40x40 (111,79)							
0	**2**	**4**	**6**	**8**	**10**	**12**	**14**

Odd indexes are **ignored** and **never accessed**

Saving Room Dimensions

Every time a room & its dimensions are generated, we send it here.



Saving Room Dimensions

Every time a room & its dimensions are generated, we send it here.

40x40 (111,79)	55x75 (xxx,xxx)	64x52 (xxx,xxx)					
0	**2**	**4**	**6**	**8**	**10**	**12**	**14**

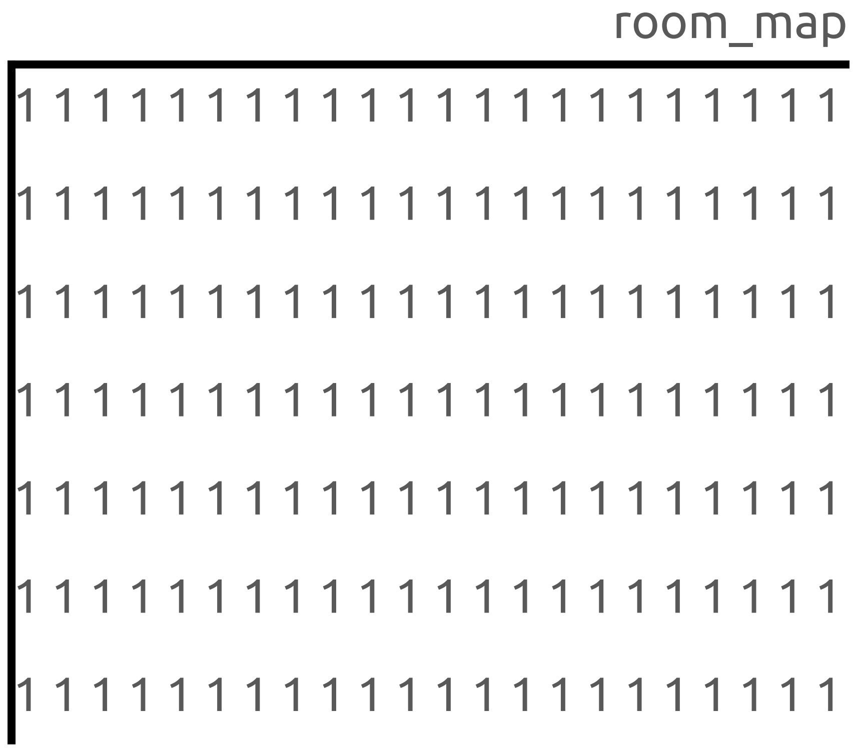
These values are utilized in the…

World Map

The world map is initialized as an 800x800 **2D Array** of 1’s, which indicate an empty square.

Each index is a single tile in the game world.

This grid helps us keep track of **where** rooms are in relation to each other.



```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Not to scale

Creating World Map Runtime

Our grid is **800x800** indexes long.

800 x **800** = 640,000

n * n = n²

Thus, the runtime is **O(n²)**

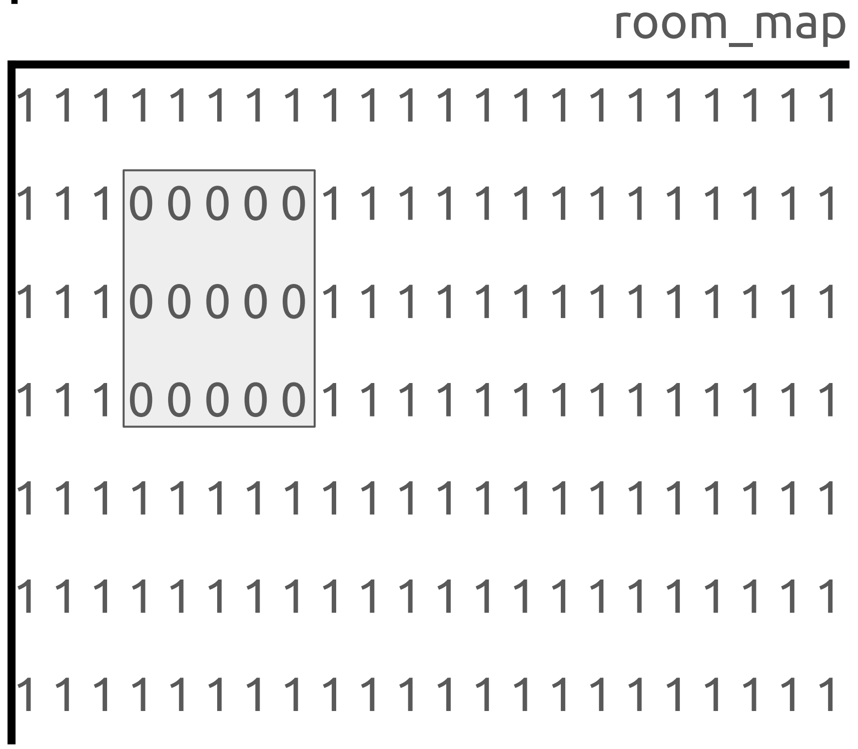
room_map

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Not to scale

Adding the Start Room to Map

The start room size is always 40x40 for simplicity’s sake.



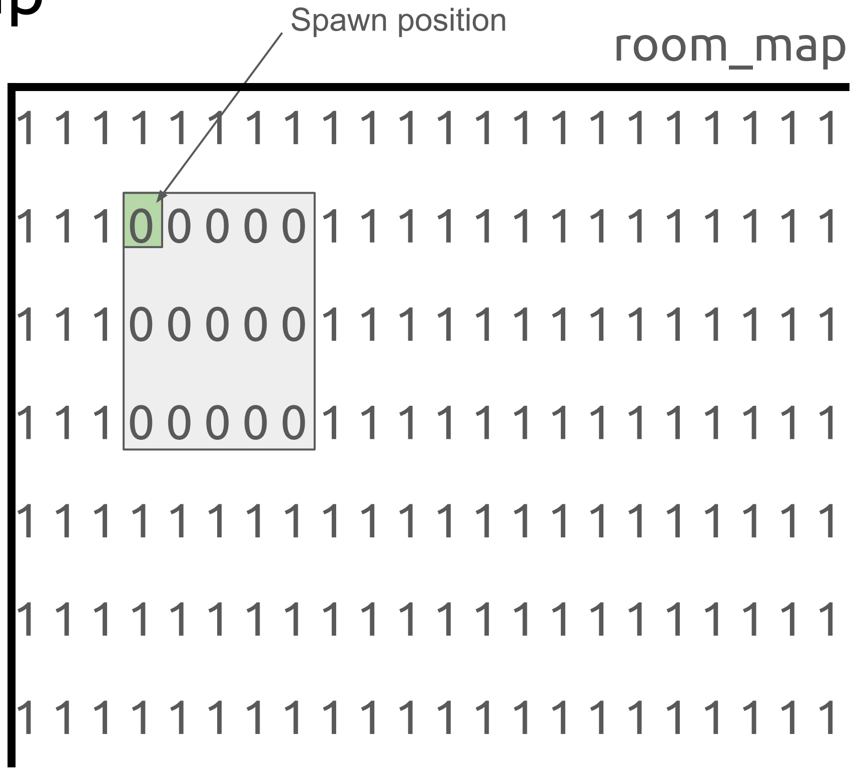
```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Not to scale

Adding the Start Room to Map

The start room size is always 40x40 for simplicity's sake.

A random grid position is randomly selected as the top-left corner of the room.



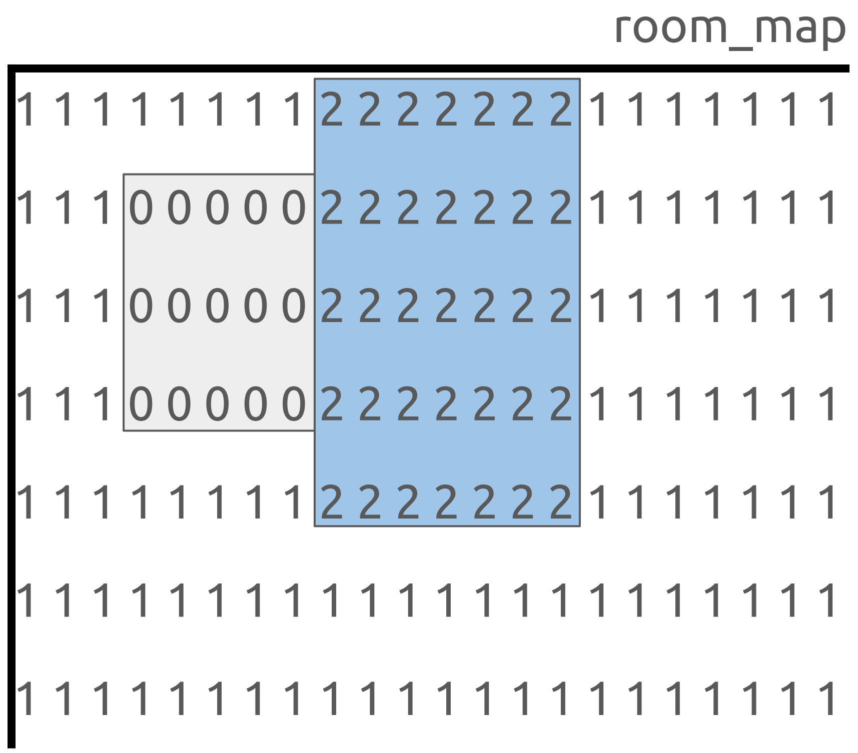
Not to scale

Store room in world map

Rooms are added starting from index 0

and each subsequent room's value is:

lastRoom + 2



Not to scale

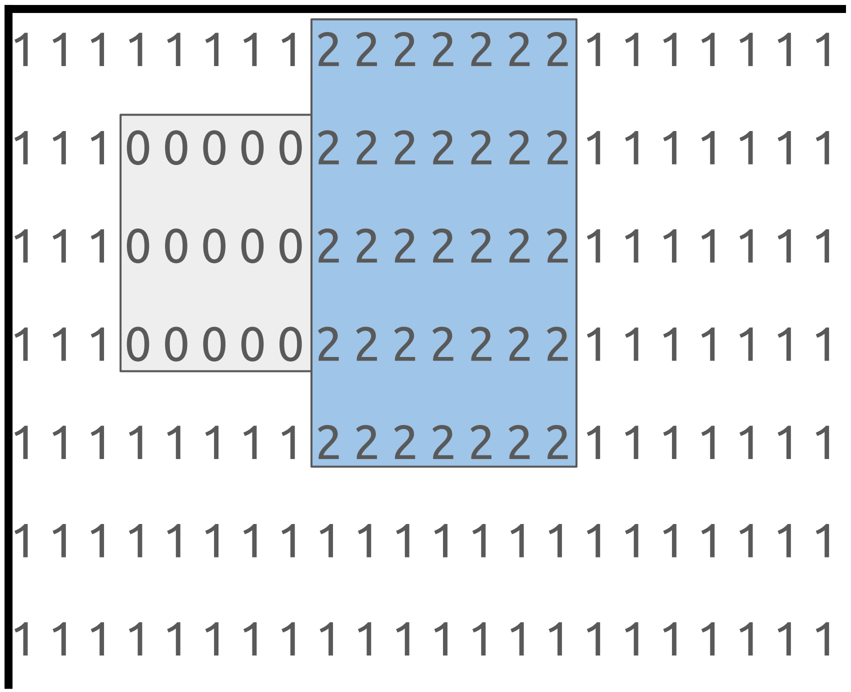
Spawning Doors in Rooms

Every time a room is created, we check the midpoints of each wall

To do this, we need to grab the current room's start point and size from the dimensions array.

```
// (x,y) together create the top left corner of the room
let  (width, height, x, y) = get_current_room_dimensions();
let left_x = x - 1;
let left_y = y + (height / 2);
```

room_map



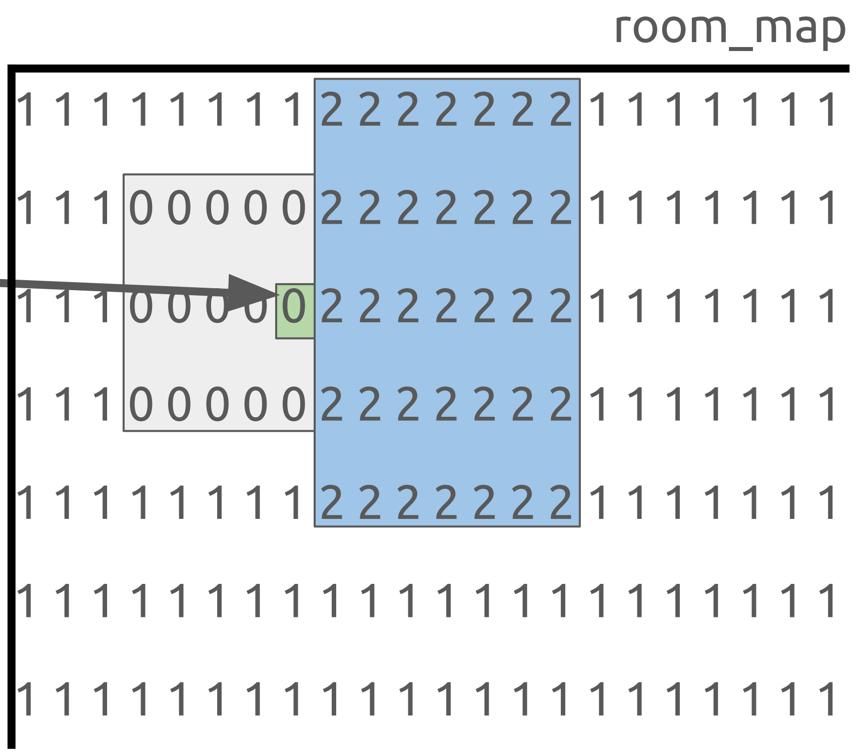
```
1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 1 0 0 0 0 0 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 1 0 0 0 0 0 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 1 0 0 0 0 0 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Not to scale

Spawning Doors in Rooms

Next, we check if a room already exists at that point

```
// (x,y) together create the top left corner of the room
let  (width, height, x, y) = get_current_room_dimensions();
let left_x = x - 1;
let left_y = y + (height / 2);

// check if room already exists in that location
let map_value= get_room_at_index(left_x  , left_y );

if (map_value == 1) {
            check_size_try_generate_door();
} else {
            generate_door();
}
```

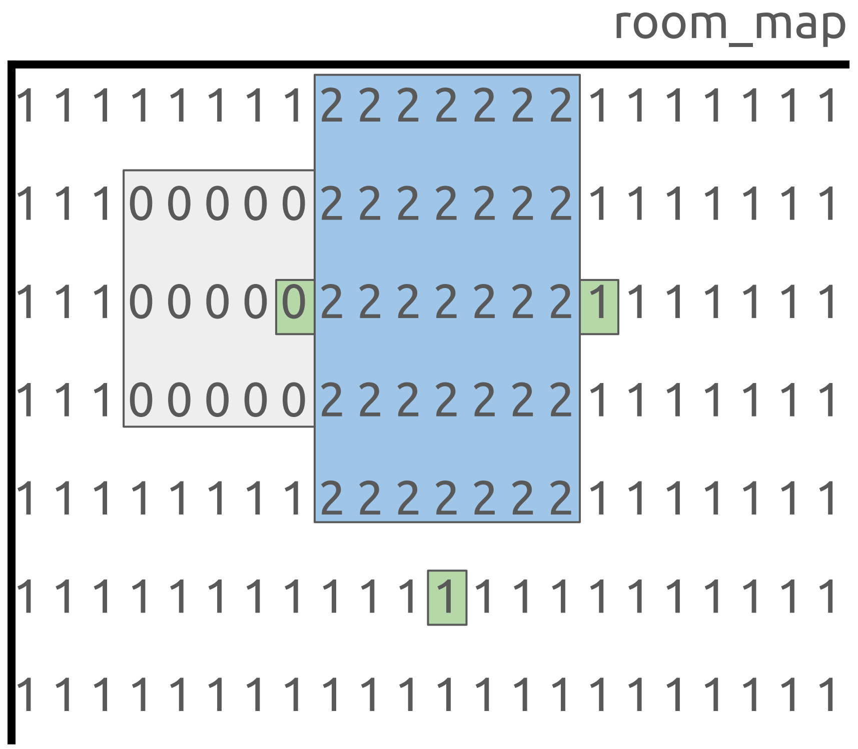


Not to scale

Spawning Doors in Rooms

Next, we continue checking each door

```
let  (width, height, x, y) = get_current_room_dimensions();

// bottom door
let bottom_x = x + (width / 2);
let bottom_y = y + (height + 1);


// right door
let right_x = x + (width + 1);
let right_y = y + (height / 2);

// check for rooms at indexes above
```

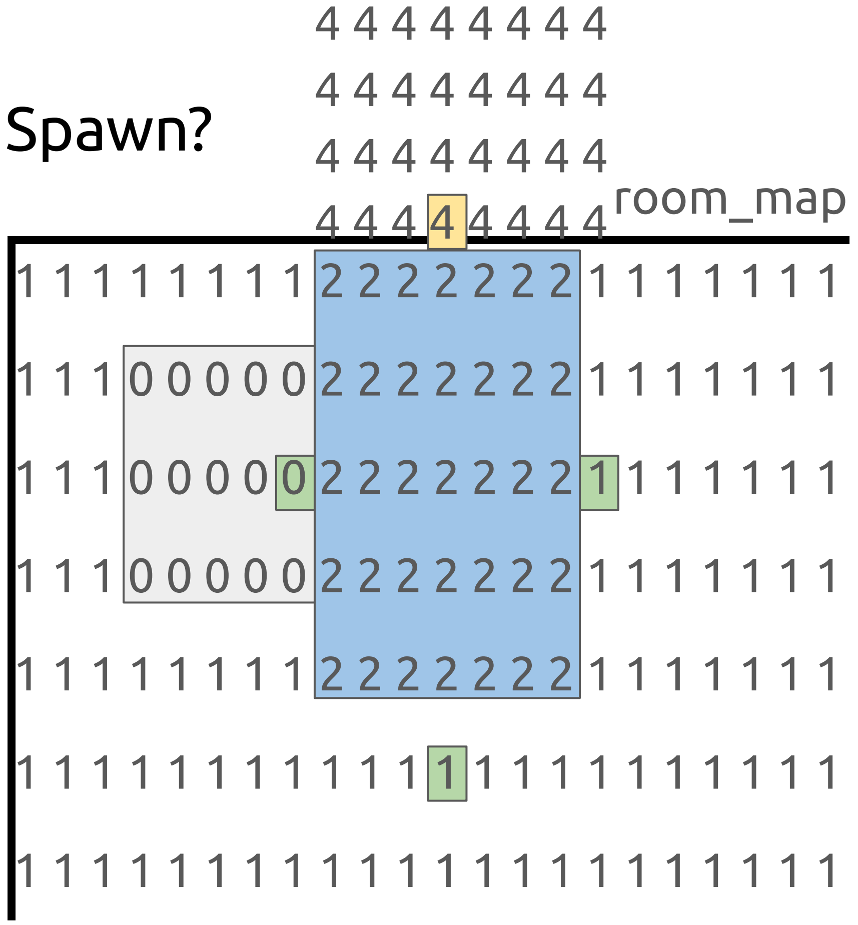


What Happens if a Room Can't Spawn?

Every door check, we see if a max-size room can spawn outside of the door

```
let  (width, height, x, y) = get_current_room_dimensions();

// top door
let top_x = x + (width / 2);
let top_y = y - 1;
```

As you can see, the max-size room is outside of the map area. Thus…

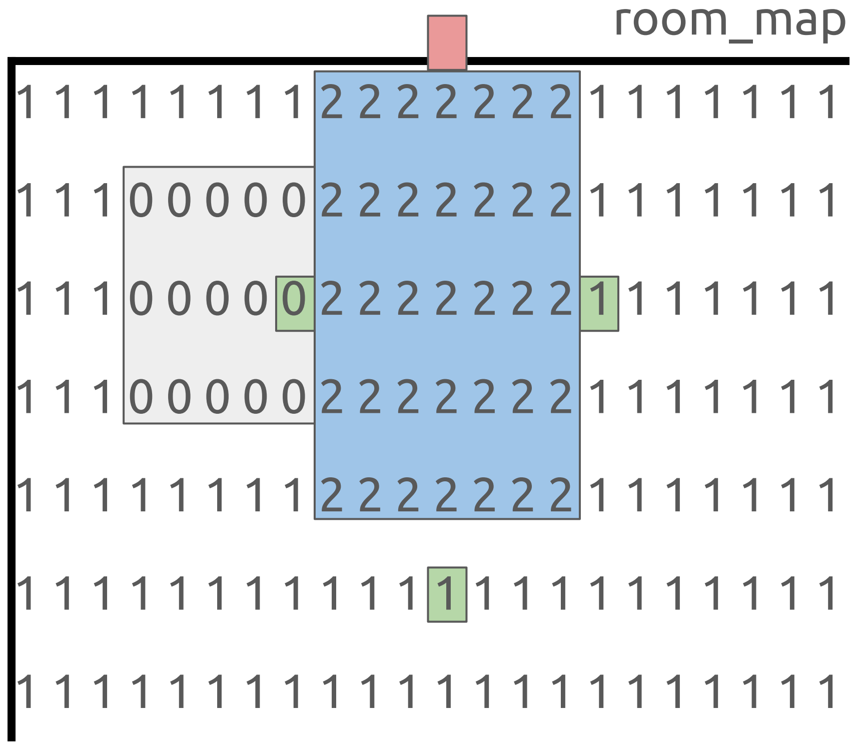


Top Door Deleted

The top door will not spawn as a max sized room could not physically fit there anyway.

Because we always know what index to check:

Every array access is **O(1),** and the room dimensions array is only accessed once during every room spawning process.

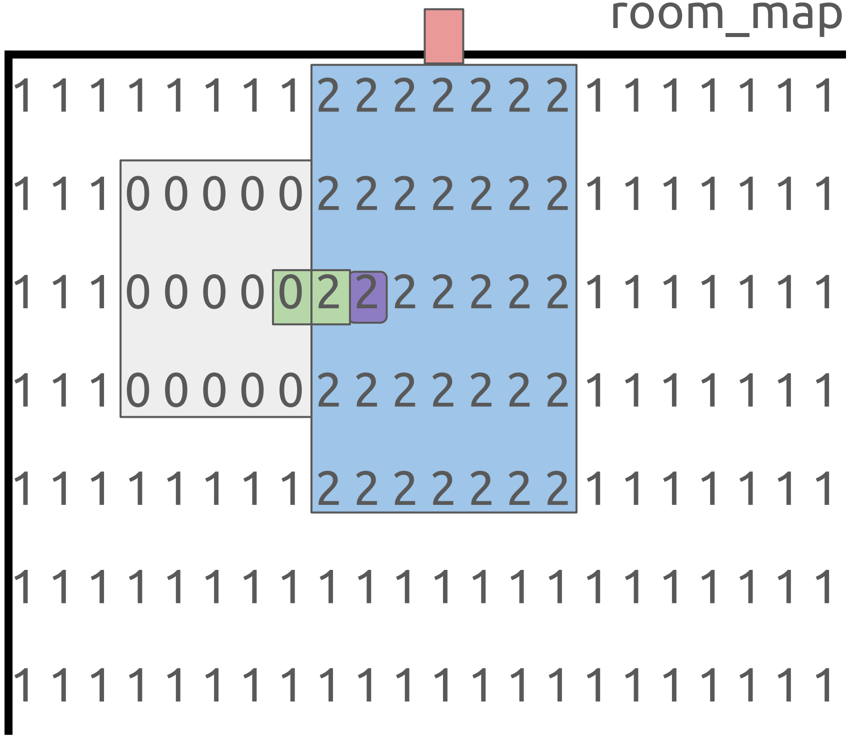


```
1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 1 0 0 0 0 0 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 1 0 0 0 0 0 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 1 0 0 0 0 0 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

What happens when we hit a door?

Room Transitioning

Let’s assume that this 🟪 square represents the player, currently standing in room 2.



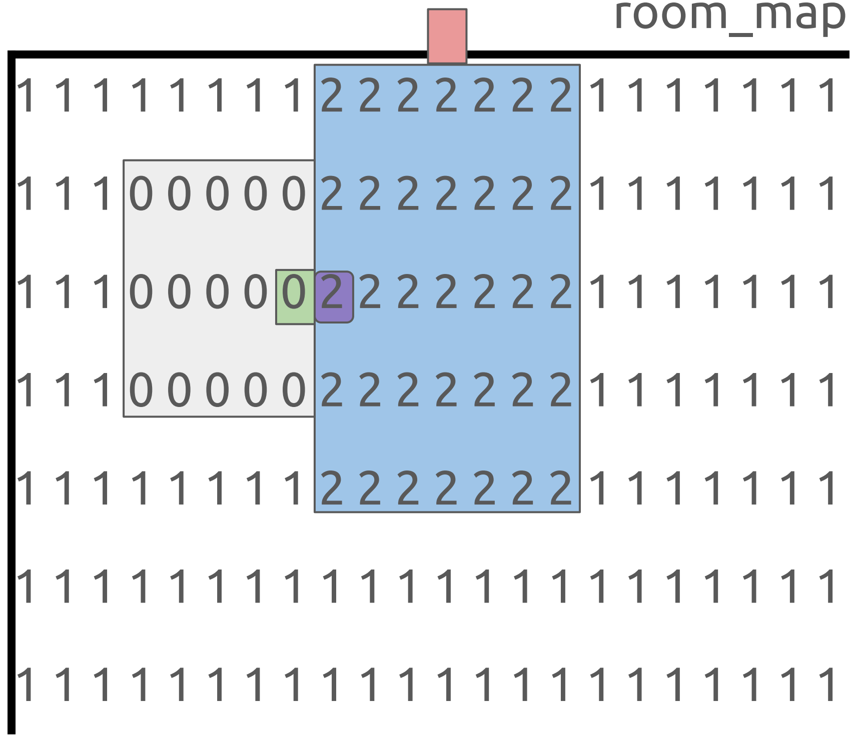
```
1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 1 0 0 0 0 0 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 1 0 0 0 0 0 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 1 0 0 0 0 0 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Room Transitioning

Let’s assume that this 🟪 square represents the player, currently standing in room 2.

When a player comes in contact with a door:

Check Room Value Outside of Door



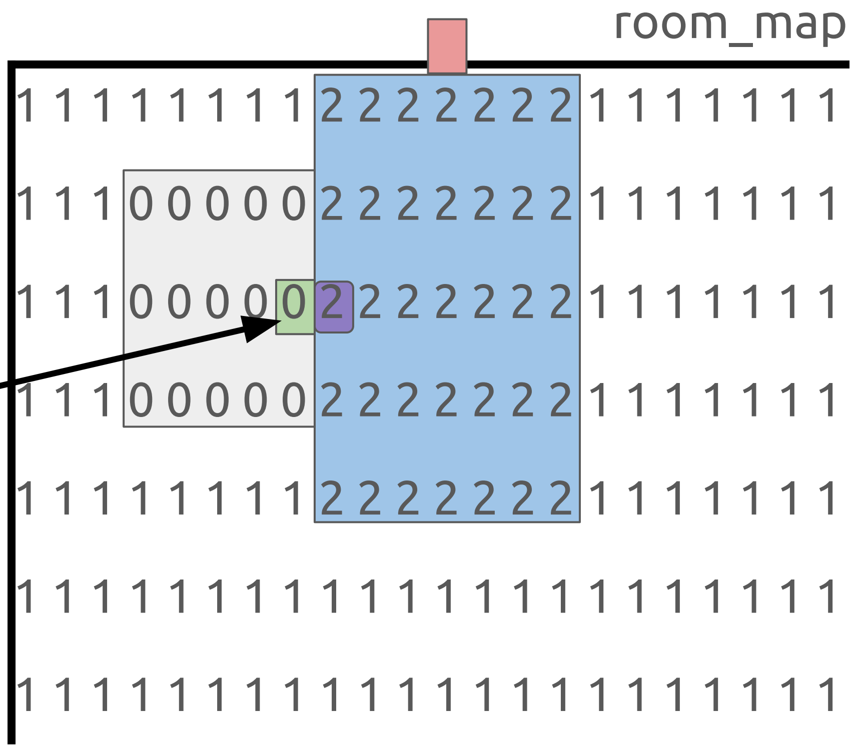
```
1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 1 0 0 0 0 0 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 1 0 0 0 0 0 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 1 0 0 0 0 0 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Check Room Value Outside Door

Using the same logic seen here:

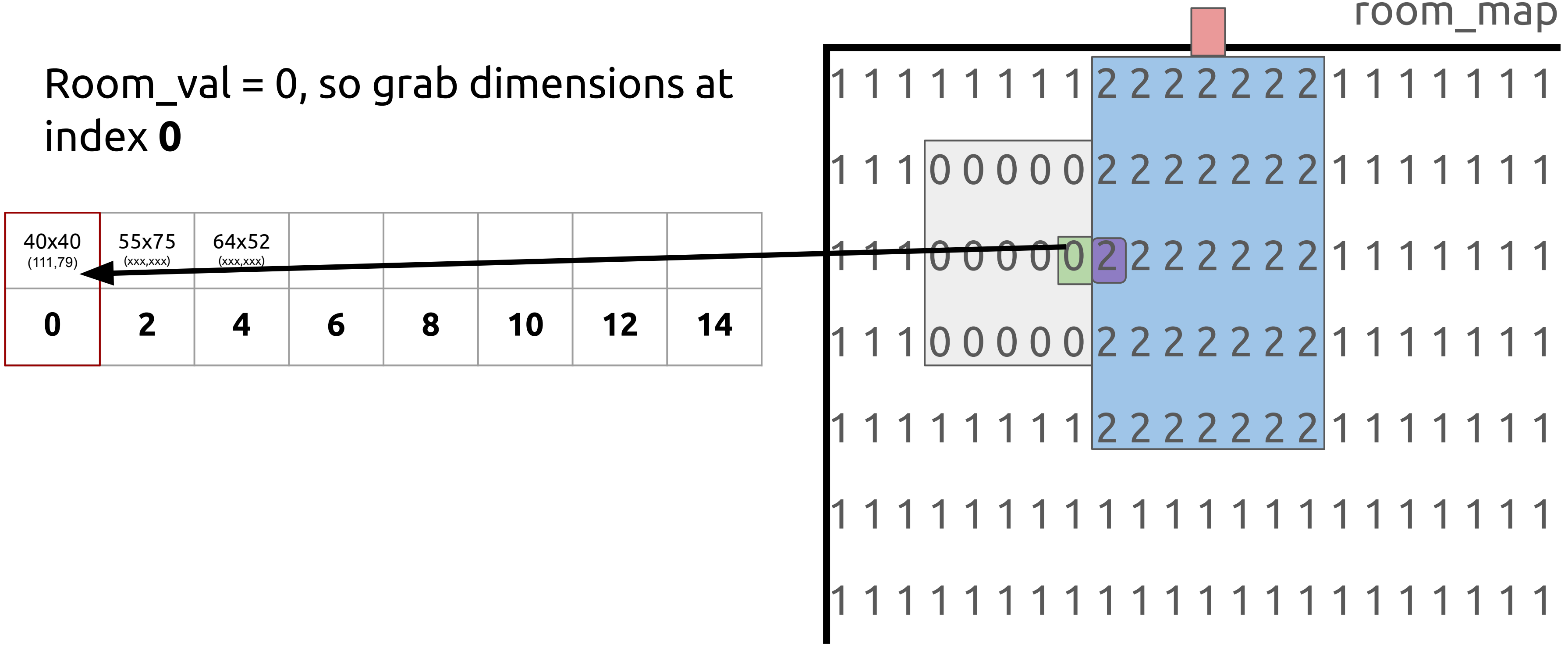
```
// (x,y) together create the top left corner of the room
let  (width, height, x, y) = get_current_room_dimensions();
let left_x = x - 1;
let left_y = y + (height / 2);

// check if room already exists in that location
let room_val = get_room_at_index(left_x , left_y );
```

If the room value != 1, we grab its dimensions



Check Room Value Outside Door

Room_val = 0, so grab dimensions at index **0**



40x40 (111,79)	55x75 (xxx,xxx)	64x52 (xxx,xxx)					
0	**2**	**4**	**6**	**8**	**10**	**12**	**14**

room_map

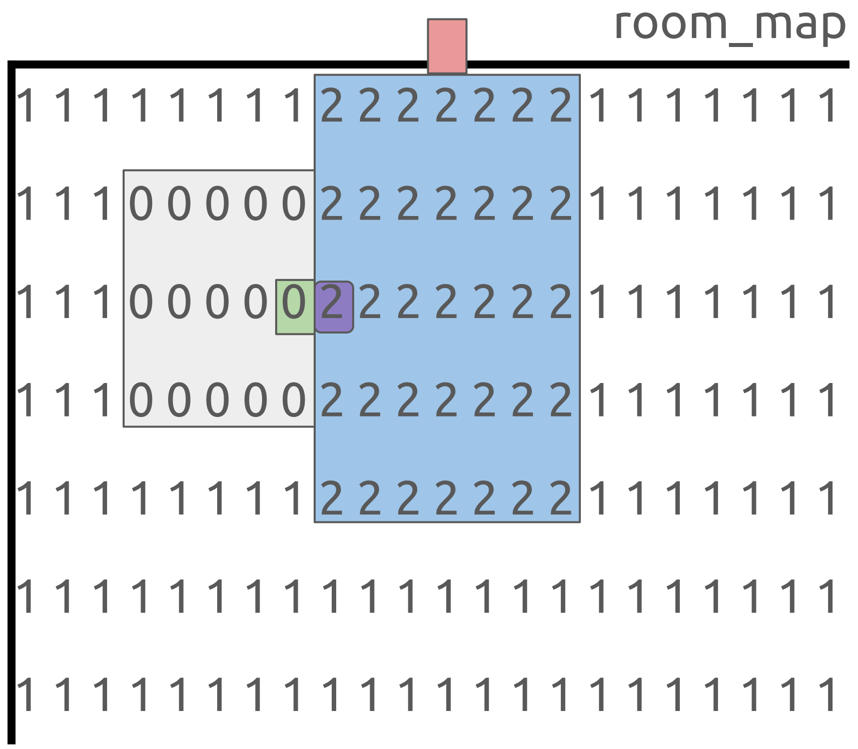
1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	1	1	1	1	1	1	1
1	1	1	0	0	0	0	0	2	2	2	2	2	2	2	1	1	1	1	1	1	1
1	1	1	0	0	0	0	0	2	2	2	2	2	2	2	1	1	1	1	1	1	1
1	1	1	0	0	0	0	0	2	2	2	2	2	2	2	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Check Room Value Outside Door

Room_val = 0, so grab dimensions at index **0**

40x40 (111,79)	55x75 (xxx,xxx)	64x52 (xxx,xxx)					
0	**2**	**4**	**6**	**8**	**10**	**12**	**14**

Redraw a room with the dimensions found.



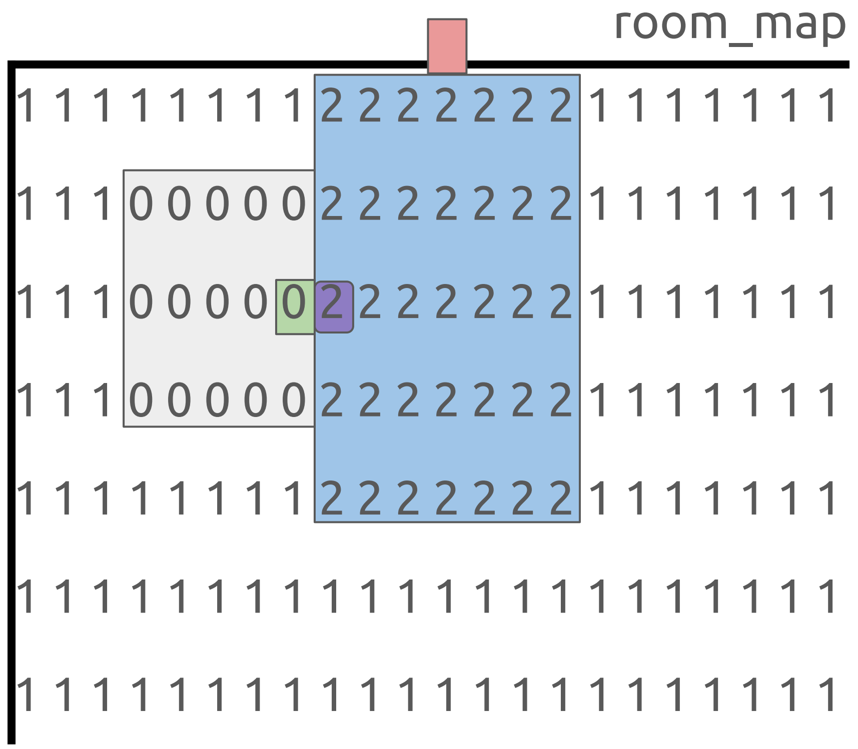
Check Room Value Outside Door

Room_val = 0, so grab dimensions at index **0**

40x40 (111,79)	55x75 (xxx,xxx)	64x52 (xxx,xxx)					
0	**2**	**4**	**6**	**8**	**10**	**12**	**14**

Redraw a room with the dimensions found.

What about if there is no room to transition into?



Generating Random Rooms

Now we are at the stage where random room generation will need to be **truly** random, including:

- Room Dimensions (width, height)
- Enemy Count
- Item Counts

How do we determine these metrics?

Generating Random Rooms

Now we are at the stage where random room generation will need to be **truly** random, including:

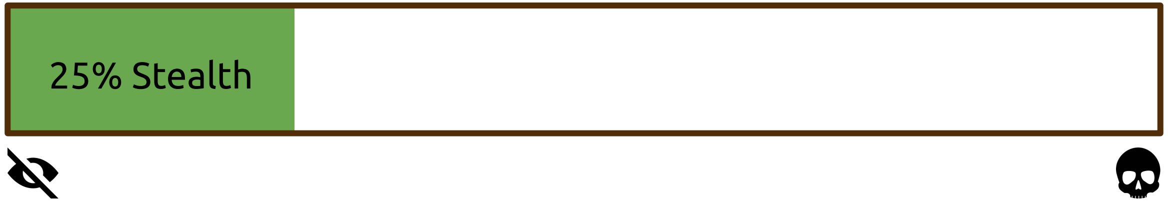
- Room Dimensions (width, height)
- Enemy Count
- Item Counts

How do we determine these metrics?

→ **The Carnage Bar**

The Carnage Bar

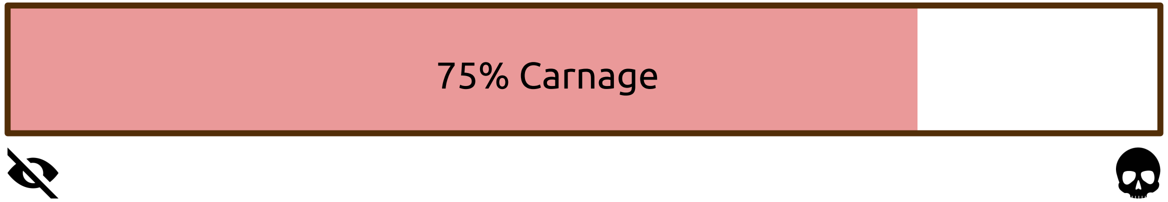
The **Carnage Bar** is a percent value from 0-100%.



A % **LESS** than 50% means that the player is more heavily weighted toward stealth.

The Carnage Bar

The **Carnage Bar** is a percent value from 0-100%.



A % **GREATER** than 50% means that the player is more heavily weighted toward carnage.

Increasing Carnage & Stealth

To increase carnage, the player can:

Increasing Carnage & Stealth

To increase carnage, the player can:

- Attack enemies
- Take damage from enemies
- Collect items

Increasing Carnage & Stealth

To increase carnage, the player can:

- Attack enemies
- Take damage from enemies
- Collect items

To increase stealth, the player can:

Increasing Carnage & Stealth

To increase carnage, the player can:

- Attack enemies
- Take damage from enemies
- Collect items

To increase stealth, the player can:

- Get through a room undetected
- Have near-misses with enemies
- Distract enemies with a thrown object

Enemy Spawns

Our **4 enemy types** are spawned differently depending on the carnage value:

High Stealth:

- Weak, but long detection radius enemy
- Longest detection radius, but short memory enemy

High Carnage:

- Low health, but fast and lethal enemy
- Tanky and lethal, but slow enemy

Enemy Spawns with Varying Values

Singleplayer/Multiplayer:

High carnage rooms will generate a room with less enemies with more health and lower detection radii, encouraging stealth.

High stealth rooms will generate a room with weaker enemies, but too many to sneak around, encouraging carnage.

Multiplayer Only:

The game automatically spawns 2x as many enemies per room as a single player run, making the dungeon more dynamic and challenging.

Item Spawns

Items, such as potions and pots, will spawn more frequently in multiplayer lobbies to balance progression.

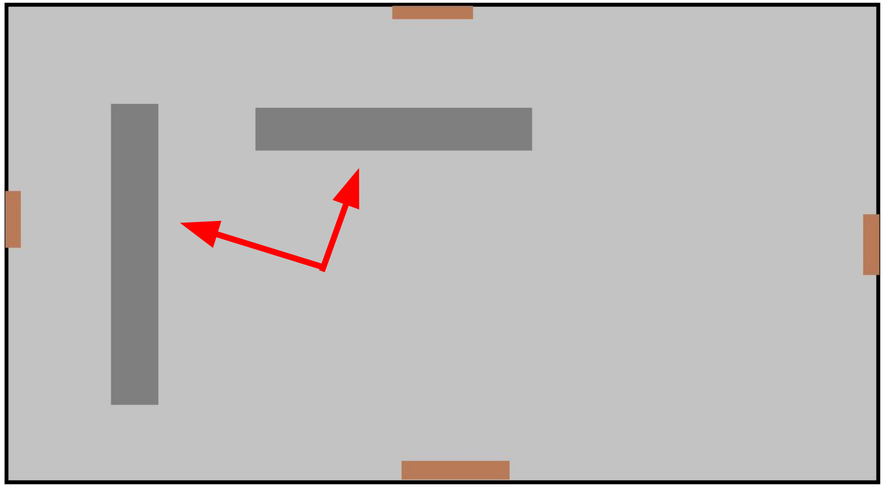
On top of this, a high carnage value will increase the number of potions spawned, as the player may need to be healing more often.

Inner Wall Spawning

Every room has 0-3 inner walls (capped at 3 so no closed walls could ever happen).

High carnage rooms will generate more inner-walls to encourage the player to be stealthy, and vice versa.

Now how do we actually calculate these metrics for each room?



Markov Chains!

Markov Chains: History

History

A probability theory developed in the early 20th by **Andrey Andreyevich Markov**, a Russian Mathematician who had a love for Russian poetry.

First real-world application in the 1940s and 1950s in optimizing telephone networks (Queueing Theory).

А. А. Марковъ (1886).

Markov Chains: History

The first implementation in video games was in the 1970s/80s for predicting player behavior, and is used in popular games today such as *The Sims* and certain roguelikes.

In *The Sims*, NPCs have a set of possible states that they can be in any given time, i.e., ***eating***, ***sleeping***, ***socializing***, etc.

Markov Chains: The Sims

A **Markov Chain** is used to model transitions between these actions based on the current state of the NPC (“***hungry***”, “***tired***”)

A **probability matrix** dictates how likely the NPC is to transition from one state to another.

For example, a “**hungry**” NPC is very likely to transition to “**eating**”.

Markov Chains

A Markov Chain:

- Only requires knowledge of the current state
- Already knows the probabilities of transitions between states
- Can be represented as a Finite State Machine (FSM)

In a Markov chain:

- Every row is a probability vector
- Each row has a sum of 1

Markov Chains

For Example: Auto-complete in word or google

- Every word is a state
- The probability of each state from the last is used to predict what you will write

Markov Chains: Example

What do you need?

What is it? Dragons?

Psst. Hey, I know who you are.

Try to hide it all you want. I know you're in the Thieves Guild.

My cousin's out fighting dragons, and what do I get? Guard duty.

I used to be an adventurer like you. Then I took an arrow in the knee.

What do

	do	is
What	2	

Markov Chains: Example

What do you need?

What is it? Dragons?

Psst. Hey, I know who you are.

Try to hide it all you want. I know you're in the Thieves Guild.

My cousin's out fighting dragons, and what do I get? Guard duty.

I used to be an adventurer like you. Then I took an arrow in the knee.

What is

	do	is
What	2	1

Markov Chains: Example

What do you need?

What is it? Dragons?

Psst. Hey, I know who you are.

Try to hide it all you want. I know you're in the Thieves Guild.

My cousin's out fighting dragons, and what do I get? Guard duty.

I used to be an adventurer like you. Then I took an arrow in the knee.

	do	is
What	67 %	33%

Markov Chains: Example

What do you need?

What is it? Dragons?

Psst. Hey, I know who you are.

Try to hide it all you want. I know you're in the Thieves Guild.

My cousin's out fighting dragons, and what do I get? Guard duty.

I used to be an adventurer like you. Then I took an arrow in the knee.

I know

	do	is	know	get	used	took
What	67 %	33%	0%	0%	0%	0%
I	0%	0%	40%	20%	20%	20%

Markov Chains: Example

What do you need?

What is it? Dragons?

Psst. Hey, I know who you are.

Try to hide it all you want. I know you're in the Thieves Guild.

My cousin's out fighting dragons and what do I get? Guard duty.

I used to be an adventurer like you. Then I took an arrow in the knee.

I know who

	do	is	know	get	used	took	who	you’re
What	67 %	33%	0%	0%	0%	0%	0%	0%
I	0%	0%	40%	20%	20%	20%	0%	0%
know	0%	0%	0%	0%	0%	0%	50%	50%

So how do we use these in Cuscuta?

- The deciding factor for how rooms generate
- Transition matrices are used to determine the likelihood of certain generations given the current state
- Our states define the bounds of our random generation
  - Inner Wall Count
  - Enemy Count
  - Room Size
  - Enemy Type
  - Item Count

Creating a Transition Matrix

- Room size bounds
- Let’s say current room is **small**

	Large Room	Medium Room	Small Room
Large Room	0.1	0.5	0.4
Medium Room	0.3	0.4	0.3
Small Room	0.5	0.2	0.3

Creating a Transition Matrix

- Room size bounds
- Let’s say current room is **small** -> Next room more likely to be **large**

	Large Room	Medium Room	Small Room
Large Room	0.1	0.5	0.4
Medium Room	0.3	0.4	0.3
Small Room	0.5	0.2	0.3

Determine room metrics using state & Transition Matrix

- Use previous transition matrix state
- I.e. Small Room
- Use matrix probabilities to define next state

	Large Room	Medium Room	Small Room
Large Room	0.1	0.5	0.4
Medium Room	0.3	0.4	0.3
Small Room	0.5	0.2	0.3

Effect of the Carnage Value

- Weights the probabilities in the transition matrix towards desired room types at carnage/stealth percentage
- Using weighted transition matrices (50% carnage shown below)

	Large Room	Medium Room	Small Room
Large Room	0.1	0.5	0.4
Medium Room	0.3	0.4	0.3
Small Room	0.5	0.2	0.3

Skewing results based on Carnage

Taking a **goal probability** and skewing the probabilities in the direction of the values using the Carnage percentage as influence:

- Maximum probability is 85%, minimum 5% to ensure random generation

The **skew** is how much we change the probabilities in the transition matrix.

- Max left skew [0.85, 0.10, 0.05]
- Max right skew [0.05, 0.10, 0.85]
- If Carnage is 50% **don't** skew
- Less than 50% skew **left**
- More than 50% skew **right**

Skewing based on Carnage in Action

Matrix = transition matrix

Matrix’ = **new** transition matrix

Left = goal probability if carnage percentage is 0

Right = goal probability if carnage percentage is 100

P = current carnage percentage (as decimal)

$P < 0.5$:

$$\text{Matrix'}[\text{row}][\text{col}] = (1 - 2P) \times \text{Left}[\text{col}] + 2P \times \text{Matrix}[\text{row}][\text{col}]$$

$P > 0.5$:

$$\text{Matrix'}[\text{row}][\text{col}] = (1 - 2(P - 0.5)) \times \text{Matrix}[\text{row}][\text{col}] + 2(P - 0.5) \times \text{Right}[\text{col}]$$

Effect of the Carnage Value

- Carnage is high, we want smaller rooms to be more likely

	Large Room	Medium Room	Small Room
Small Room	0.1	0.5	0.4
Medium Room	0.3	0.4	0.3
Large Room	0.5	0.2	0.3

Effect of the Carnage Value

- Carnage is high, we want smaller rooms to be more likely
- Using the carnage percentage to skew the probabilities
- Previous formula applied at 75% Carnage

	Large Room	Medium Room	Small Room
Small Room	0.1 -> 0.075	0.5 -> 0.300	0.4 -> 0.625
Medium Room	0.3 -> 0.175	0.4 -> 0.250	0.3 -> 0.575
Large Room	0.5 -> 0.300	0.2 -> 0.525	0.3 -> 0.175

How does this work with multiple markov chains?

- High carnage states will be on the right side of their transition matrices
- The same process can be applied to the different markov chains with different starting matrices

	Large Room	Medium Room	Small Room
Small Room	0.1	0.5	0.4
Medium Room	0.3	0.4	0.3
Large Room	0.3	0.5	0.2

Why do we use multiple Markov Chains?

- Limiting each specific attribute to its own markov chain allows for simpler modification
- Easier to track states

Enemies	Many	Medium	Few
Few	0.1	0.5	0.4
Medium	0.3	0.4	0.3
Many	0.3	0.5	0.2

Rooms	Large Room	Medium Room	Small Room
Small Room	0.1	0.5	0.4
Medium Room	0.3	0.4	0.3
Large Room	0.3	0.5	0.2

Items	Many	Medium	Few
Few	0.1	0.5	0.4
Medium	0.3	0.4	0.3
Large	0.3	0.5	0.2

Runtime of Markov Chains

Creation: **O(n²)** for each matrix: n is smaller when using multiple states

Skew: **O(n²)** for each matrix: multiplying each value in table by skew

Access: **O(n)**: Only need to grab values at given row

Enemies	Many	Medium	Few
Few	0.1	0.5	0.4
Medium	0.3	0.4	0.3
Many	0.3	0.5	0.2

Rooms	Large Room	Medium Room	Small Room
Small Room	0.1	0.5	0.4
Medium Room	0.3	0.4	0.3
Large Room	0.3	0.5	0.2

Items	Many	Medium	Few
Few	0.1	0.5	0.4
Medium	0.3	0.4	0.3
Large	0.3	0.5	0.2

So let's give an example…

Let’s say we just spawned in a **large room** with **few enemies** and **many items:**

Enemies	Many	Medium	Few
Few	0.1	0.5	0.4
Medium	0.3	0.4	0.3
Many	0.3	0.5	0.2

Rooms	Large Room	Medium Room	Small Room
Small Room	0.1	0.5	0.4
Medium Room	0.3	0.4	0.3
Large Room	0.3	0.5	0.2

Items	Many	Medium	Few
Few	0.1	0.5	0.4
Medium	0.3	0.4	0.3
Large	0.3	0.5	0.2

So let's give an example…

Let’s say we just spawned in a **large room** with **few enemies** and **many items:**

Enemies	Many	Medium	Few
Few	0.1	0.5	0.4
Medium	0.3	0.4	0.3
Many	0.3	0.5	0.2

Rooms	Large Room	Medium Room	Small Room
Small Room	0.1	0.5	0.4
Medium Room	0.3	0.4	0.3
Large Room	0.3	0.5	0.2

Items	Many	Medium	Few
Few	0.1	0.5	0.4
Medium	0.3	0.4	0.3
Many	0.3	0.5	0.2

So let's give an example…

Let’s say we just spawned in a **large room** with **few enemies** and **many items:**

As you can see, based on our input state and the probabilities, the outcome is most likely a **medium room** with **few enemies** and **many items.**

Enemies	Many	Medium	Few
Few	0.1	0.5	0.4
Medium	0.3	0.4	0.3
Many	0.3	0.5	0.2

Rooms	Large Room	Medium Room	Small Room
Small Room	0.1	0.5	0.4
Medium Room	0.3	0.4	0.3
Large Room	0.3	0.5	0.2

Items	Many	Medium	Few
Few	0.1	0.5	0.4
Medium	0.3	0.4	0.3
Many	0.3	0.5	0.2

Conclusions

The game initializes a start room and we spawn at 50% carnage with preset attributes.

We then check the player’s location to decide what room is being entered or generated.

Markov Chains allow us to skew our randomization with a carnage value.

- Makes generation more fun and dynamic.

Markov Chains can easily get large with too many states.

Dungeons are generated during runtime, using markov chains to influence room attributes such as items or room size.

Sources

https://brilliant.org/wiki/markov-chains/

https://en.wikipedia.org/wiki/Markov_chain

https://en.uesp.net/wiki/Skyrim:Guard_Dialogue

Thank You!

Any questions?