Procedural Content Generation for Unity Game Development

Harness the power of procedural content generation to design unique games with Unity

Ryan Watkins

[PACKT] PUBLISHING

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 **'Generating an Endless World'**
- A synopsis of the book's content
- More information on **Procedural Content Generation for Unity Game Development**

About the Author

Ryan Watkins was digitized and absorbed into his computer at an early age. In the digital realm, he learned the importance of video games and the balance they brought to the ecosystem of computing. Video games strive to always push the boundaries of what we know to be true while being a super-charged source of fun. Ryan formed friendships with many of the video games he encountered on his digital journeys, and in return, they shared the secrets of their creation with him. He has since returned to the physical world to share those secrets with the rest of us.

Preface

This book is an introduction to Procedural Content Generation (PCG) and how it can be applied in the popular game engine, Unity3D. PCG is a powerful programming practice that is trending in modern video games. Though PCG is not a new practice, it has become even more powerful as technology has advanced and it looks to be a prominent component of future video games.

Throughout the course of this book, we will be learning the basis of procedural content generation, including theory and practice. You will start by learning what PCG is and what its uses are. You will then move into learning about pseudo random numbers and how they work with PCG to create unique gameplay.

After your introduction to PCG, you will dive in and build the core functionality of a 2D *Roguelike* game. This game will be heavily based on PCG practices so that you can experience what it takes to design and implement PCG algorithms. You will experience level generation, item generation, adaptive difficulty, music generation, and more. Lastly, we will move into 3D object generation by generating a 3D planet.

The aim of this book is to teach you about the theory of PCG while providing some simplified practical examples. By the end of the book, you should have a fundamental understanding of PCG and how it can be applied using Unity3D. This will all facilitate your further learning, research, and practice of PCG methods in video game development.

What this book covers

Chapter 1, *Pseudo Random Numbers*, teaches you about the theory of procedural content generation (PCG). We will cover what PCG is and how it is used in video games. You will then learn about a useful component of randomization called Pseudo Random Numbers (PRN). We will cover what PRNs are, how they are used, and how they can help us implement PCG algorithms.

Chapter 2, *Roguelike Games*, teaches you about a prime example of procedural content generation, *Roguelike* games. We will cover some history of the origin of PCG and *Roguelike* games. We will then set up the Unity project foundation of our very own *Roguelike* game.

Chapter 3, *Generating an Endless World*, begins the implementation of your 2D *Roguelike* game. We will be creating a level that generates itself at runtime while the player explores. We will cover PCG algorithm design and useful data substructures. Then, we will put it together to implement the game world.

Chapter 4, *Generating Random Dungeons*, implements the sublevels of our Roguelike game. We will cover a different approach to level generation as we generate a full level layout at runtime. You will learn about some common approaches to this technique and implement one for yourself.

Chapter 5, *Randomized Items*, teaches you about randomly generating items. The items you generate will have differing properties so we will use some techniques to communicate this to the player. We will cover item spawning, interaction, and inventory storage.

Chapter 6, *Generating Modular Weapons*, teaches you about and how to implement a random modular weapon system. You will build upon what you learned in the previous chapter to add more complexity to item generation. These items will comprise a small set of pieces that are assembled at runtime.

Chapter 7, *Adaptive Difficulty*, crosses over into the field of Artificial Intelligence (AI) and teaches you about how AI and PCG are similar and related. You will learn about the PCG idea of adaptive difficulty, which is one part AI and one part PCG. You will then implement an adaptive difficulty system for your *Roguelike* game.

Chapter 8, *Generating Music*, shows you how PCG can even contribute to the music and sound content of a game. You will learn a little music theory; just enough to design a PCG algorithm for music generation. Then, you will implement a music generator for your Roguelike game that can generate music at runtime.

Chapter 9, *Generating a 3D Planet*, switches gears from 2D-based PCG to 3D-based PCG. We will have finished our core 2D Roguelike functionality and be working on a new project. This chapter will introduce the fundamentals of 3D object generation. You will then implement a 3D planet generator. Plus, as a bonus, you will implement a first person controller to take a closer look at your generated world.

Chapter 10, *Generating the Future*, discusses the most common methods of PCG used today and some ways to further your learning in the subject. We will also summarize some of the key points of what you learned throughout the book and how they relate to these PCG methods. We will lastly take a look at some ways that we can improve these PCG methods for the future.

3

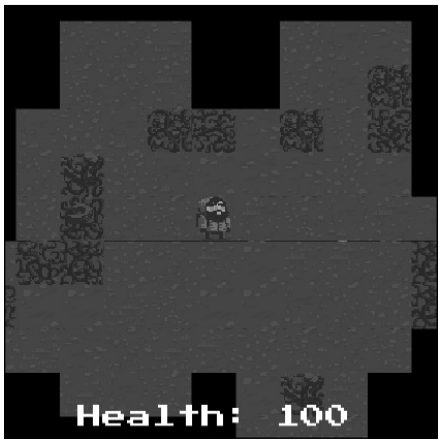
Generating an Endless World

Our base project is setup and ready to be expanded upon. You received an intro to PRNs and PCG by making a quick `Hello World` program. However, it is time to develop a fully functional PCG algorithm that can be directly applied to our *Roguelike* game.

Our game has the base functionality of a player character capable of movement by player input. In this chapter, we will create the tile-based Game Board that the player will explore. This Game Board will expand itself as the player moves, and as there are no bounds, the board is potentially infinite. In this chapter, you can expect:

- To learn about dynamic data structures
- To design our first PCG algorithm
- To set up a scene that allows for an ever expanding game world
- To develop our PCG Game Board

By the end of the chapter, you will have developed a PCG game world that is unique with every play. Plus, you will have achieved this with a relatively small amount of assets. We have to do some learning and planning first. So let's get started on the endless PCG Game Board.



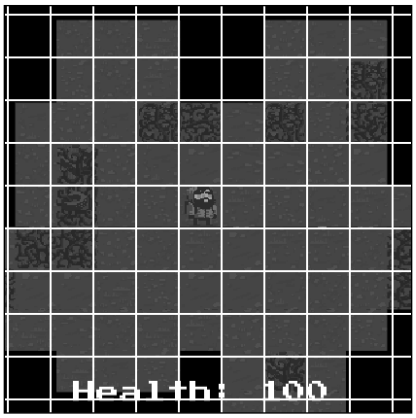
This is what our Roguelike endless PCG Game Board will look like

Data structure choice

The first PCG algorithm we will develop will create the game environment that our player will explore. We are going to build a game world that never ends; that is, as long as you have enough memory. As the player explores the world, our algorithm will create and place more pieces of the Game Board.

Remember, the Game Board is what we are calling the ground area in which the player will walk on. The Game Board is made up of small, rectangular, 2D sprites that we refer to as **tiles**. In this chapter, we will start with floor tiles, which the player will walk on. We will then randomly add wall tiles to a new layer as an obstacle to the player.

The concept of the 2D Game Board can be visualized as a grid. A grid can be easily implemented as an array or list data structure to track the tiles we layout for the Game Board. As the player explores, we will add to our list references to our newly created tiles. We can then use this list to look up any tile on our Game Board. The most important role of the tile list is such that we don't recreate a tile that is already on the Game Board. This process is usually referred to as **object pooling**.



Imagine the Game Board within a grid

Array

Because we are creating a large part of the Game Board while the player is playing the game, we need a data structure in which we can add tiles too dynamically. A two-dimensional array is visualized as a grid, which makes adding tile coordinates more natural. We name an index of a two-dimensional array like this: `myArray[X][Y]`. `X` and `Y` will be the 2D *x* axis and *y* axis coordinates.

Dynamic is another computer science term. We refer to things as *static* or *dynamic*. Static refers to something that has a hard-coded value such as: `const int num = 2;`. Dynamic refers to a value that is determined at runtime and might change over the life of the program such as: `int num = someFunction();`.

X	0	1	2	3	4
Y	0	0	0	0	0
	1	1	1	1	1
	2	2	2	2	2
	3	3	3	3	3
	4	4	4	4	4

Visualization of a 2D array

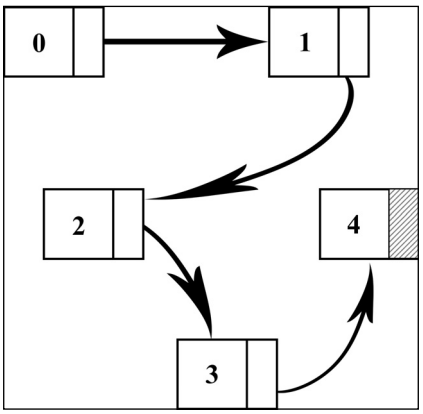
Arrays are great for fast lookup, which will be important as our Game Board expands. Arrays reserve a portion of memory and assign a call number called an index to every block of memory in the array. This gives the array a fixed size guaranteeing that we won't run into memory problems when generating the world. Also, looking up any index in an array is nearly instant.

The main issue with using a two-dimensional array is that we have to predefine the size. This means that as soon as the game starts, we will reserve a large chunk of memory for our game that we might not use. It is even worse if we end up filling the array before the game is over, as then we have to make a new larger array, transfer all the information to the new array, and deallocate all the memory of the old array.

The larger array then faces the same two problems as described previously. This method also means having to spend extra development time to write the logic that would perform the array rewrite task. Our game might then start to slow down as our memory expands and contracts to make new arrays and delete old ones. This will still be a viable option, but perhaps with a little more careful development work.

Linked list

The 2D array has the important feature of fast look up but the implementation would be inefficient and it also runs the risk of creating too much overhead. So another option would be a list, which is a form of linked list. The list doesn't need to reserve a chunk of memory because each entry in the list holds a link reference to the entry before it and after it in the list. This means that list entries can be stored anywhere in the memory, which eliminates the need to reserve any memory on startup.



Visualization of a linked list

The list solves the problem of having to create a new array when you run out of space. We can easily and continuously add to a list dynamically, as well. However, the fast look up will suffer slightly with lists and therefore cause the performance of the game to suffer. When an array is created, the structure of it is like a chart where each entry is adjacent to the next and the system can scan very quickly to a specified index. Lists, on the other hand, have entries littered throughout the memory so the system has to start at the first entry and follow each entry to the specified index.

One more caveat of a list is that we wouldn't be able to reference an index that we hadn't already filled with a tile. In our 2D array example, all of the indices of the array are predefined and can be referenced, even though they are empty. List entries don't have place holders like this. If we reference a list entry that doesn't exist, the system will throw an exception and most likely freeze the game if we aren't prepared for it. We would have to predefine all the entries in the list to circumvent this issue, which means a list is little better than an array at this point.

So now we are at the crossroads of choosing the lesser of two evils. In one hand, we have the array that needs some overhead maintenance to add tiles dynamically. On the other hand, we have a list that can be dynamically added to, but we are forced into a similar overhead to maintain our free flowing grid structure. Our ideal data structure can have dynamic entry additions with a fast lookup, but with none of the costs associated with arrays and lists. Again, this is a perfectly viable option, but perhaps there's another that will suit our needs better.

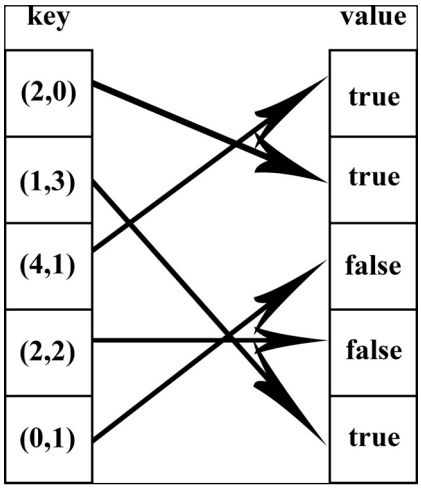
Dictionary

The compromise is something called a **Dictionary** in C#, which is a form of an associative array. An associative array is just a modified array to use some other data type like a string as the index instead of a non-negative number. The dictionary uses a key-value pair to store data and it can be dynamically added to and removed from. However, keep in mind that the dictionary will continue to take up as much memory as its maximum size. The dictionary key-value pair would look something like this:

```
Dictionary myInventory (string key, int value);
myInventory.Add("Gold", 10);
myInventory["Gold"];
```

The C# dictionary is a class that wraps an array. This class will notice when the internal array is about to become full and will perform the task of rewriting our data into a larger array automatically. The dictionary also has a fast lookup speed. Having an associative array as our data structure also benefits us by only needing to create a one-dimensional array instead of a two-dimensional array as in the previous examples.

Because the dictionary takes a key-value pair, we can make our key a Vector2 with our X and Y coordinates of each placed tile. This way, when we want to know whether a specific tile has been placed already, we can look up the X and Y coordinates directly. The `Dictionary` class has a method called `Contains()` making lookups as easy as `myInventory.Contains("Gold");`. This will give us a `true`/`false` value that we can check in an `if…else` or a `switch` statement.



Visualization of a dictionary/map

Here is a summary of our data structure choice:

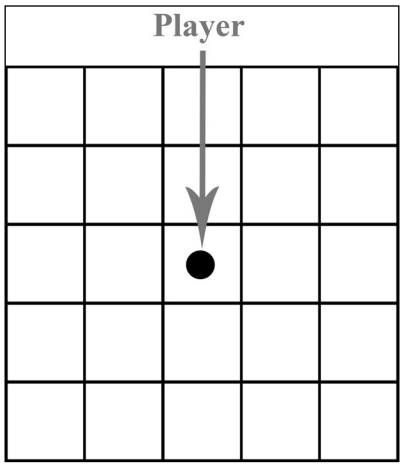
- **Array**: This is a simple data structure with fast lookup but it is difficult to add tiles dynamically.
- **List**: This is a linked list, which can have tiles added dynamically, but will have slower lookup as the list gets bigger.
- **Dictionary**: This is an associative array class with built-in array resizing and a fast lookup.

  Even though the dictionary has a little more overhead with it's extra class methods, it will be the most efficient data structure for our use. It is also really easy to use.

PCG algorithm overview

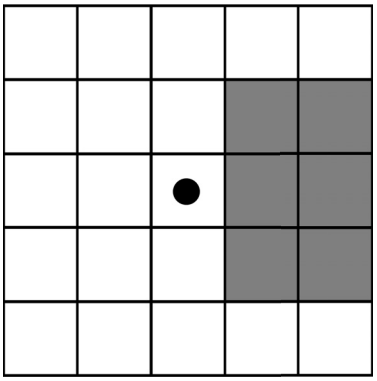
Now that we settled on our data structure for managing our Game Board grid, we need to design our algorithm for placing tiles. This algorithm will use two types of PCG. We will only create tiles that the player discovers, which is a form of player-triggered PCG. We will also use random numbers to dictate the look of the tile and to choose which floor tiles will have a wall tile placed on top.

To start our algorithm design, let's imagine and try to visualize a use case.
We want our player to start in a small area that has already been revealed and added to our data structure. When the game starts, let's create a 5 x 5 grid of tiles for an initial Game Board. We can then place our player character in the center of the grid initially.



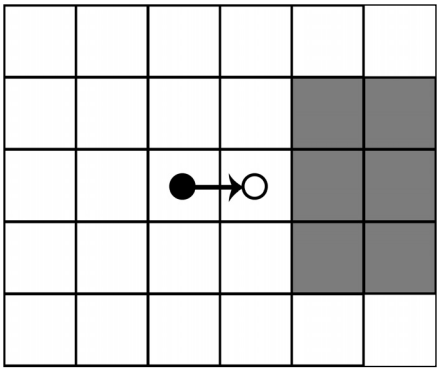
The initial Game Board grid with the player

As the player explores, our algorithm will reveal more tiles in the direction the player is headed. We will refer to this as the player's *line of sight*. We can use any arbitrary number of tiles to reveal ahead of the player. Six tiles feels like a good amount, though. This will give us a center point on which to place the player character.



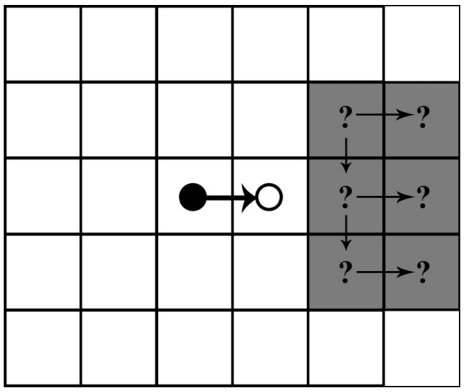
The line of sight grid squares are shaded

Every time the player moves into another tile, we will check the six tiles in front of the character. In order to accomplish this, we will need to track the player's position as a `Vector2` with an X and Y coordinate. We will also need to track the direction in which the player is moving. With the player position and direction, we can find the coordinates of the six tiles in front of the player.



Revealing more tiles as the player moves right

As the player explores and returns to the areas, the six tiles spaces in the player's line of sight might have already been revealed and in our dictionary. So, with every step the player takes, we need to check each of the six lines of sight tiles to see whether the player has already discovered them. If we find that a tile is already in our dictionary, we don't want to overwrite it with a new tile because it might change its look.



The tiles are iterated over to check whether they have already been discovered

When we check the player's line of sight, we will perform one of two actions on each tile. If the tile is undiscovered, we will add its coordinates to our dictionary, randomly choose a floor tile sprite to place, put it on the Game Board, and randomly add a wall tile on top of the floor tile. Otherwise, if the floor tile is already in our dictionary, we will just ignore it. We have to make sure that we are updating the player's position as well for this algorithm to work.

Here is a summary of our PCG algorithm:

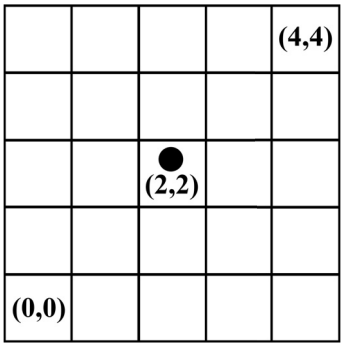
- The player moves one tile in any direction
- Get direction where the player moved
- Update the coordinates of the player's position
- Use the player's position to find and check the 6th line of sight tiles
- Add undiscovered tiles to our dictionary and place them on the Game Board
- Randomly add wall tiles to newly added floor tiles
- Ignore previously discovered tiles

Scene setup

Now that we chose a data structure and designed our algorithm, we need to set up our scene. At the moment, at the start of our game, the player character shows up in the corner of the screen with a black backdrop. The player can move in four directions but could potentially move off screen. The player can also move under the text that shows the player's health. We should fix this so that we can better see our PCG algorithm in action.

Player positioning

Previously, we said that a starting grid of 5 x 5 for our initial Game Board would be a good metric. So let's continue with a 5 x 5 Game Board in mind. If our 5 x 5 grid starts with the X-Y coordinate or (0,0) in the lower-left corner and (4,4) in the upper-right corner, then (2,2) will be the center of the grid. Select the **Player** prefab in the **Hierarchy** panel and set both the X and Y values to `2`.



The grid will correspond with the x-y plane

Each sprite in our sprite sheet is 32 x 32 pixels. When the sprite sheet was imported, the Pixels to Units import setting was set to 32 pixels equals 1 Unity unit of measure. So our Game Board will align precisely with the Unity x-y plane. We can then build our Game Board around our player character starting at (0,0) in the lower-left corner. This same unit of measure will be used to track our player's position.

All our sprites are 32 x 32, which is 1 unit of measure

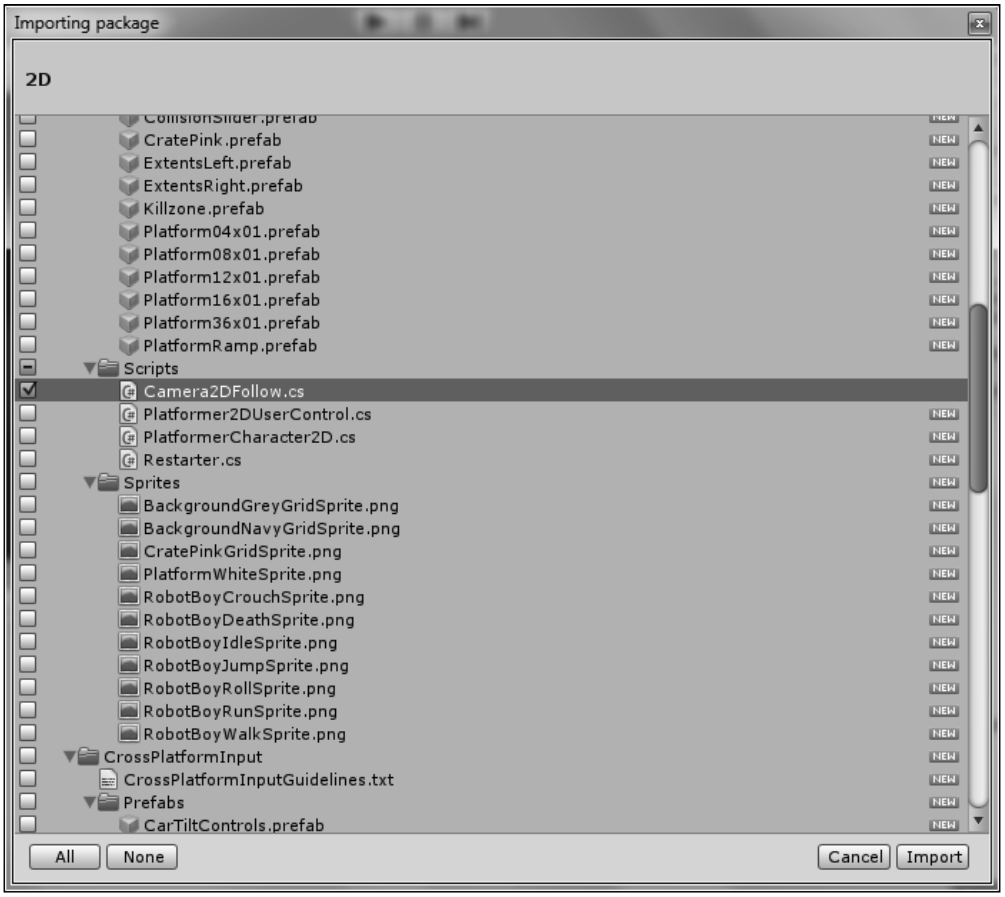
We changed the position of the player character, but that doesn't prevent the player from being able to walk off screen. We need a way to keep the player character in sight at all times. We can write a script that makes it possible for the player to only move as far as the screen edge but that doesn't make sense since our Game Board is infinite. Instead, we can have the camera move with the player.

Camera following

Unity makes it easy to have a camera follow the player. There is a script in the Unity Standard Assets called `Camera2DFollow.cs`. We will simply import the script, which is included when you download Unity. We can then adjust the settings to suit our needs.

To import the script, follow these steps:

1. In the top menu, navigate to **Assets** | **Import Package** | **2D**.
2. In the **Importing package** popup, select **None** to uncheck all the options.
3. Find and check the `Camera2DFollow.cs` package by navigating to the **2D** | **Scripts** directory.
4. Click on **Import**.

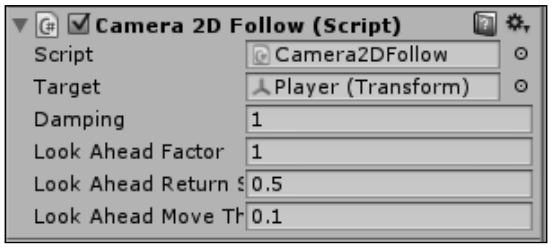


Import settings

You will have a **Standard Assets** folder added to your project. Inside the **Standard Assets** folder is **2D | Scripts | Camera2DFollow.cs**. Drag and drop the **Camera2DFollow** script onto the **Main Camera**. Then, from the **Hierarchy** pane, drag and drop the **Player** prefab onto the **Target** field in the **Camera2DFollow** script component of the **Main Camera**.

We are going to change the other settings as well:

- Set the **Damping** field to `1`
- Set the **Look Ahead Factor** field to `1`
- Set the **Look Ahead Return Speed** field to `0.5`
- Set the **Look Ahead Move Threshold** field to `0.1`



The Camera 2D Follow settings screen

You are welcome to experiment with the settings. There might be another set of values that you think looks better. However, this setup works without being too jerky.

So now, when you start the game by pressing the play button, the Main Camera will snap to the player character. You can try walking around to test the camera follow settings. Then, after you are satisfied with how the camera follows the player, we will need to make some layer adjustments.

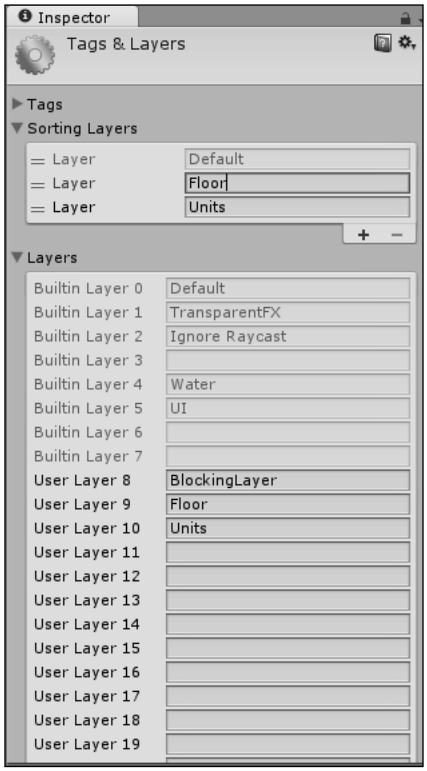
No Game Board

Layers

Even though our game is 2D and takes place on only one visible plane, we can still place game objects on different layers to manage how things interact. So, we will add some layers to manage how the player interacts with their environment:

- Select the **Player** prefab from the **Hierarchy** panel
- Select the **Layer** field dropdown
- Select **Add Layer…** from the dropdown
- Add `BlockingLayer`, `Floor`, and `Units` as layer labels to any empty field (do not overwrite any existing field)
- Then, select the **Sorting Layers** dropdown on the same screen
- Add `Floor` and `Units` as **Sorting Layer** labels, in that order

These layers might have been added from the import process in *Chapter 2*, *Roguelike Games* but this is not guaranteed.



The Layer settings

The layers will help us divide tiles into specific regions of interest. Game objects that are impassable or can prevent the player from moving will be placed on the `BlockingLayer`. The **Player** prefab and wall tiles will be on the `BlockingLayer` because the player should not simply be able to walk through walls.

The sorting layer is important because it will dictate which sprites are rendered first. We want the player character to render on top of the floor tiles. The layers at the top of the **Sorting Layers** list are rendered first. So we place the `Floor` layer higher in the list so that it is rendered before the player character.

Now, we have to select the **Layer** and **Sorting Layer** in the **Sprite Renderer** component of our **Player** prefab, floor tiles, and wall tiles.

For the **Player** prefab, follow these steps:

1. In the **Hierarchy** panel, select **Player**.
2. Select the **Layer** dropdown.
3. Select **BlockingLayer**.
4. In the **Sprite Renderer** component, select the **Sorting Layer** dropdown.
5. Select **Units**.

For the floor tiles, follow these steps:

1. In the **Project** tab, select the **Prefabs** folder.
2. Select **Floor1** to **Floor8** at the same time using the *Shift* or *command* key.
3. In the **Sprite Renderer** component, select the **Sorting Layer** dropdown.
4. Select **Floor**.

For the wall tiles, follow these steps:

1. In the **Project** tab, select the **Prefabs** folder.
2. Select **Wall1** to **Wall8** at the same time.
3. Select the **Layer** dropdown.
4. Select **BlockingLayer**.
5. In the **Sprite Renderer** component, select the **Sorting Layer** dropdown.
6. Select **Units**.

Unfortunately, layers in Unity do not carry over when making a package of your project. The reasoning behind this is that you might have created some layers in your project and then imported a package. If the imported package brought along it's layers, it might overwrite some of your originally created layers. For more on layers, visit the Unity Docs at `http://docs.unity3d.com/Manual/Layers.html`.

Initial Game Board

Now that we have our algorithm designed and the Unity Editor setup, we can start our code implementation. We'll approach the task in small pieces. First, let's put down a small starting area for our player. As stated before, we will make a 5 x 5 grid to lay floor tiles on. The lower-left corner will be placed at (0,0) and the upper-right corner at (4,4) with the player character at (2,2).

We'll start by building our `BoardManager` class. Open up `BoardManager.cs` for editing. Currently, there is only a public class called `Count`, but we are about to change that. *Code Snip 3.1* shows the additions we want to make to `BoardManager.cs`:

```
1 using UnityEngine;
2 using System;
3 using System.Collections.Generic;
4 using Random = UnityEngine.Random;
5
6 public class BoardManager : MonoBehaviour {
7   [Serializable]
8   public class Count {
9     public int minimum;
10    public int maximum;
11
12    public Count (int min, int max) {
13      minimum = min;
14      maximum = max;
15    }
16  }
17
18  public int columns = 5;
19  public int rows = 5;
20  public GameObject[] floorTiles;
21  private Transform boardHolder;
22  private Dictionary<Vector2, Vector2> gridPositions = new
    Dictionary<Vector2, Vector2> ();
```

```
23
24  public void BoardSetup () {
25  boardHolder = new GameObject ("Board").transform;
26
27  for(int x = 0; x < columns; x++) {
28    for(int y = 0; y < rows; y++) {
29      gridPositions.Add(new Vector2(x,y), new Vector2(x,y));
30
31      GameObject toInstantiate = floorTiles[Random.Range
        (0,floorTiles.Length)];
32
33      GameObject instance = Instantiate (toInstantiate, new
        Vector3 (x, y, 0f), Quaternion.identity) as GameObject;
34
35      instance.transform.SetParent (boardHolder);
36    }
37  }
38 }
39 }
```

So our original `BoardManager` class has doubled in size. Let's take a look at what we added:

- `Line 18-19`: Here are two public integer variables called `row` and `column`, which represent our starting Game Board grid.
- `Line 20`: `floorTiles` is a public GameObject array that will hold all the floor prefabs.
- `Line 21`: `boardHolder` is a private transform that will hold all the tiles.
- `Line 22`: `gridPositions` is a private dictionary, which is our chosen data structure to hold the list of references to every tile our game lays out.
- `Line 24-39`: `BoardSetup` is a public function that returns void. This function will create our initial Game Board and add the tile references to our dictionary.
- `Line 27-28`: This `for` loop nested within another `for` loop will iterate over every cell in our initial 5 x 5 grid.
- `Line 31`: `toInstantiate` will randomly choose a tile from our array of floor tiles.
- `Line 33`: `instance` will instantiate our randomly chosen floor tile and lay it at the coordinates provided by the `for` loops.
- `Line 35`: Finally, we make the instance of the floor tile a child of `boardHolder`, our Game Board transform.

So, our Game Board is able to set up an initial 5 x 5 board, but the functionality isn't fully integrated yet. We need to make some adjustments to our `GameManager` script as well. We can see the changes needed in *Code Snip 3.2*. Keep in mind that *Code Snip 3.2* is not the full file:

```
7 public class GameManager : MonoBehaviour {
8
9   public float turnDelay = 0.1f;
10  public int healthPoints = 100;
11  public static GameManager instance = null;
12  [HideInInspector] public bool playersTurn = true;
13
14  private BoardManager boardScript;
15  private List<Enemy> enemies;
16  private bool enemiesMoving;
17
18  void Awake() {
19    if (instance == null)
20      instance = this;
21    else if (instance != this)
22      Destroy(gameObject);
23
24    DontDestroyOnLoad(gameObject);
25
26    enemies = new List<Enemy>();
27
28    boardScript = GetComponent<BoardManager>();
29
30    InitGame();
31  }
...
56  void InitGame() {
57    enemies.Clear();
58
59    boardScript.BoardSetup();
60 }
```

There's only a few key lines here to integrate the `BoardManager` class with the rest of the game. Let's see what the changes are:

- `Line 14`: `boardScript` is the variable we will use to keep a reference to our `BoardManager` script.
- `Line 28`: Inside the `Awake` function, we will have `boardScript` reference the `BoardManager` script that we will add to our `GameManager` prefab.

- `Line 59`: Inside the `InitGame` function, we will call the `BoardSetup` function from our attached `BoardManager` script.

So, now the `BoardManager` script functionality will be called from the `GameManager` script. However, when you press play, there is still no Game Board. We are still missing a few connections, which we will need to set up in the Unity Editor.

In the **Project** tab, follow these steps:

1. Select the **Prefabs** folder.
2. Select the **GameManager** prefab.
3. In the **Inspector** tab, select the **Add Component** button.
4. From the **Add Component** dropdown, navigate to **Script** | **BoardManager**.

These steps add the `BoardManager` script to our **GameManager** prefab, but we need to add the floor tile references now.

In the **Board Manager** script component of the **GameManager** prefab, follow these steps:

1. Set **Size** under **Floor Tiles** to `8` and press *Enter*.
2. Then, drag and drop **Floor1** to **Floor8** into the newly created **Element0** to **Element7** under **Floor Tiles**.

Now, we can press the play button and we'll see our player character standing on our initial 5 x 5 Game Board. Notice that because our initial Game Board is procedurally generated, it is made up of a different combination of tiles every time we play the game. However, this isn't the end of our PCG game world. We want the Game Board to expand as the player explores.

Initial Game Board

Connecting code

We now need to add the functionality of our expanding Game Board. As per our algorithm design, we need to track the player character's position. When the player moves, we need to send the player character's position and direction to the `BoardManager` class. So let's start with the additions needed in the `Player` script shown in *Code Snip 3.3*:

```
7 public class Player : MovingObject {
8   public int wallDamage = 1;
9   public Text healthText;
10  private Animator animator;
11  private int health;
12  public static Vector2 position;
13
14  protected override void Start () {
15
16    animator = GetComponent<Animator>();
17
18    health = GameManager.instance.healthPoints;
19
20    healthText.text = "Health: " + health;
21
22    position.x = position.y = 2;
23
24    base.Start ();
25  }
26  private void Update () {
27    if(!GameManager.instance.playersTurn) return;
28
29    int horizontal = 0;
30    int vertical = 0;
31
32    bool canMove = false;
33
34    horizontal = (int) (Input.GetAxisRaw ("Horizontal"));
35    vertical = (int) (Input.GetAxisRaw ("Vertical"));
36
37    if(horizontal != 0)
38    {
39      vertical = 0;
40    }
41    if(horizontal != 0 || vertical != 0)
42    {
```

```
43      canMove = AttemptMove<Wall> (horizontal, vertical);
44      if(canMove) {
45        position.x += horizontal;
46        position.y += vertical;
47        GameManager.instance.updateBoard(horizontal, vertical);
48      }
49    }
50  }
51
52  protected override bool AttemptMove <T> (int xDir, int yDir) {
53    bool hit = base.AttemptMove <T> (xDir, yDir);
54
55    GameManager.instance.playersTurn = false;
56
57    return hit;
58  }
```

In *Code Snip 3.3*, we are changing some of the structure of our base code. These changes will force us to change how some other functions operate. So let's see the new changes and how they'll affect the rest of our development:

- `Line 12`: `position` is a public static `Vector2` that will hold the current coordinates of our player. It is static so that we can access this variable from any script in the game.
- `Line 22`: We set the `x` and `y` value of position to `2` because we know that at the start of the game, the player character will always begin on (2,2) of our Game Board.
- `Line 32`: We create a Boolean variable called `canMove`, which will tell us whether the player is blocked from moving or not. We will calculate this value at every update so it is set to `false` by default.
- `Line 43`: Here, we set `canMove` equal to our `AttemptMove` function. However, `AttemptMove` returns void so this will be something we need to fix coming up.
- `Line 45-46`: `position` is updated by adding in the values we obtain for `horizontal` and `vertical`. `horizontal` and `vertical` come from `Input.GetAxisRaw`, which returns `1` if the player moves in the positive direction or `-1` if the player moves in the negative direction.
- `Line 47`: We call our instance of the `GameManager` class and invoke the `updateBoard` function, which doesn't yet exist. We will put this here as a place holder and write the `updateBoard` function later.
- `Line 52`: We need to rewrite `AttemptMove` to return a bool. We will start by declaring that the function will return bool.

- `Line 53`: We are going to create a Boolean variable to hold the bool that `base.AttemptMove` will return. This again is a place holder as `base.AttemptMove` does not yet return a bool. We also need to remove `RaycastHit2D hit`.
- `Line 57`: Return the newly created bool value.

We are using a public static variable for the player position so that we can access it from anywhere at anytime. It is, however, a best practice to make these types of values private and accessible through a `get` function. Having a variable be public static means it can also be changed from anywhere in the code.

This can cause problems if you have more than one person working on a single code base and another developer using the public static variable in a way you didn't intend. It is best to be very deliberate in your code. By forcing a variable to be private and only accessible via a `get` function, you protect the variable from changing in a way that it shouldn't.

With that said, we are going to use the player position as a public static because it is easier and makes our code less bloated. Of course, you are encouraged to revise the code later to make this variable private.

At this point, there are going to be some errors in the Unity Editor because of some conflicts we created. So let's work on clearing the errors. Once everything is working again, we can work on the new functionality.

First, we should fix our bool return value conflict in the `AttemptMove` function of the `MovingObject` class. Remember, `MovingObject` is the base class for both `Player` and `Enemy`. We will have to adjust the `AttemptMove` function in all three files, as it is a `virtual` function.

Let's start by fixing the `AttemptMove` function of `MovingObject`. Open `MovingObject.cs` for editing. *Code Snip 3.4* shows the changes that need to be made to the file:

```
91 protected virtual bool AttemptMove <T> (int xDir, int yDir)
92   where T : Component
93 {
94  RaycastHit2D hit;
95
96  bool canMove = Move (xDir, yDir, out hit);
97
98  if(hit.transform == null)
99    return true;
```

```
100
101 T hitComponent = hit.transform.GetComponent <T> ();
102
103 if(!canMove && hitComponent != null)
104  OnCantMove (hitComponent);
105
106 return false;
107 }
```

We only need to adjust the one function within the `MovingObject` class. Let's take a look at how it changed in *Code Snip 3.4*:

- `Line 91`: We need to change the `void` return type to `bool`
- `Line 99`: Return the bool value as `true` if the player hit an object
- `Line 106`: Return `false` if we reached the end of the function, meaning that the player didn't hit anything

If you return to the Unity Editor, you should see some new errors. One will be complaining that the `Enemy` class has implemented `AttemptMove` incorrectly.
So let's address this next.

Our `Enemy` class at the moment is only a place holder. If you remember from *Chapter 2*, *Roguelike Games*, we are using the `Enemy` class as a way to dictate movement turns. Because our game is turn based, we need the player to wait for each visible enemy to move before the player can move again. The base code has implemented a scan for enemies, each turn using the `Enemy` class place holder so we won't have to do it later.

So the adjustment to the `Enemy` class is fairly simple. *Code Snip 3.5* shows the changes:

```
6 protected override bool AttemptMove <T> (int xDir, int yDir)
7 {
8   return true;
9 }
```

The explanation for *Code Snip 3.5* is equally simple. Keep in mind though that this is still a place holder class and we will do a full implementation later on. Let's take a look at the changes made:

- `Line 6`: The return type is changed from `void` to `bool`.
- `Line 8`: Return `true` so that we are returning the correct value at the end of the function. We need to return a bool value at the end of the function so the compiler will pass this as a nonerror. However, the bool value doesn't matter as it is not used yet.

We are almost done mending our code. We have one more error to handle before we put in our expanding Game Board functionality. Returning to the Unity Editor yet again will reveal that we called a function (that doesn't exist) from the `Player` class to the `GameManager` class. We need to add the `updateBoard` function to the `GameManager` class as a connection from the `Player` class to the `BoardManager` class.

Add *Code Snip 3.6* at the end of your `GameManager` definition:

```
public void updateBoard (int horizantal, int vertical) {}
```

`updateBoard` is called from the `Player` class whenever the player makes a successful move. Since `updateBoard` is a method of the `GameManager` class, we can call a public method of the `BoardManager` class here. We will use this connection to develop our PCG Game Board functionality.

The PCG Game Board

We are all set to write the core functionality of our PCG Game Board. The goal is to have tiles laid out in the direction the player character walks. We designed our algorithm in such a way that the Game Board will expand as the player explores.

We connected our scripts so that when the player moves, the `Player` class will update the player position and send it to the `GameManager` class. The `GameManager` class will then call a method in the `BoardManager` class to update the Game Board and pass along the player position and direction. We now need to write the code that will update the Game Board based on the player position.

Let's start by adding the function that will update the Game Board in the `BoardManager` class. Open up `BoardManager.cs` for editing. *Code Snip 3.7* shows the function that needs to be added:

```
77 public void addToBoard (int horizontal, int vertical) {
78  if (horizontal == 1) {
79    //Check if tiles exist
80    int x = (int)Player.position.x;
81    int sightX = x + 2;
82    for (x += 1; x <= sightX; x++) {
83      int y = (int)Player.position.y;
84      int sightY = y + 1;
85      for (y -= 1; y <= sightY; y++) {
86        addTiles(new Vector2 (x, y));
87      }
88    }
89  }
```

```
90  else if (horizontal == -1) {
91    int x = (int)Player.position.x;
92    int sightX = x - 2;
93    for (x -= 1; x >= sightX; x--) {
94      int y = (int)Player.position.y;
95      int sightY = y + 1;
96      for (y -= 1; y <= sightY; y++) {
97        addTiles(new Vector2 (x, y));
98      }
99    }
100 }
101 else if (vertical == 1) {
102  int y = (int)Player.position.y;
103  int sightY = y + 2;
104  for (y += 1; y <= sightY; y++) {
105    int x = (int)Player.position.x;
106    int sightX = x + 1;
107    for (x -= 1; x <= sightX; x++) {
108      addTiles(new Vector2 (x, y));
109    }
110  }
111 }
112 else if (vertical == -1) {
113  int y = (int)Player.position.y;
114  int sightY = y - 2;
115  for (y -= 1; y >= sightY; y--) {
116    int x = (int)Player.position.x;
117    int sightX = x + 1;
118    for (x -= 1; x <= sightX; x++) {
119      addTiles(new Vector2 (x, y));
120    }
121  }
122 }
123 }
```

This function contains a switch driven by direction. The base code is set up to return only one directional value at a time. This means our player character can only move one direction at a time. Either the player moves horizontally in the positive or negative `x` direction forcing the vertical direction to return `0`, or vice versa along the `y` direction.

Let's take a closer look at the code:

- `Line 77`: `addToBoard` is a public function returning void. This will be our entry point from the `GameManager` class. From the `GameManager` class, we pass the player direction to this function as arguments.
- `Line 78`: This is our first switch point. If `horizontal` equals `1`, then we know `vertical` is `0`. This corresponds to the player moving to the right on screen.
- `Line 80-85`: We are using a `for` loop nested within a `for` loop to iterate over the player's line of sight. Remember the line of sight is the six tile spaces directly in front of the player's movement direction. The line of sight makes up a 2 x 3 grid.
- `Line 86`: For each tile space we iterate over, we will call the method `addTiles` and pass in the `Vector2` produced by our `for` loops. `addTiles` does not exist yet but we will be writing it next.
- `Line 90-122`: The rest of the function is simply a variation of `Lines 78-86`. If the player did not move to the right, then we check the other directions and set up the line of sight for that direction.

Next, we will complete our expanding Game Board functionality by writing the `addTiles` function used in the `addToBoard` function you just wrote. The main objective of this function is to check our dictionary for the line of sight tiles and if they are not there, we add them. *Code Snip 3.8* shows the function as part of the `BoardManager` class:

```
61 private void addTiles(Vector2 tileToAdd) {
62   if (!gridPositions.ContainsKey (tileToAdd)) {
63    gridPositions.Add (tileToAdd, tileToAdd);
64    GameObject toInstantiate = floorTiles [Random.Range (0,
      floorTiles.Length)];
65    GameObject instance = Instantiate (toInstantiate, new
      Vector3 (tileToAdd.x, tileToAdd.y, 0f), Quaternion.identity)
      as GameObject;
66
67    instance.transform.SetParent (boardHolder);
68  }
69 }
```

This code should seem familiar. We do similar calls in the `BoardSetup` function of the `BoardManager` class. `Line 62` is the main difference. Here, we check the dictionary for the tile before we proceed. If the tile is in the dictionary, we return out of the function. This prevents us from overwriting tiles that have already been placed in the game.

PCG Game Board

You can now return to the Unity Editor and test the new functionality. Click on the play button to try it out. As per our algorithm design, whenever the player moves, more tiles are revealed in that direction.

This Game Board isn't very interesting, though. We can walk in a single direction forever with no opposition. This terrain would also make it very easy to run away from enemies. We should add some wall tiles for obstacles.

Let's return to editing the `BoardManager.cs` file. We are going to add onto our `addTiles` function by putting in a condition that adds wall tiles to newly placed floor tiles. *Code Snip 3.9* shows the code addition:

```
29 public GameObject[] wallTiles;
...
62 private void addTiles(Vector2 tileToAdd) {
63  if (!gridPositions.ContainsKey (tileToAdd)) {
64    gridPositions.Add (tileToAdd, tileToAdd);
65    GameObject toInstantiate = floorTiles [Random.Range (0,
      floorTiles.Length)];
66    GameObject instance = Instantiate (toInstantiate, new
      Vector3 (tileToAdd.x, tileToAdd.y, 0f), Quaternion.identity)
      as GameObject;
```

```
67
68    instance.transform.SetParent (boardHolder);
69
70      //Choose at random a wall tile to lay
71    if (Random.Range (0, 3) == 1) {
72      toInstantiate = wallTiles[Random.Range (0,wallTiles.Length)];
      instance = Instantiate (toInstantiate, new Vector3
      (tileToAdd.x, tileToAdd.y, 0f), Quaternion.identity) as
      GameObject;
73    instance.transform.SetParent (boardHolder);
74    }
75  }
76 }
```

Let's take a look at what we added in *Code Snip 3.9*:

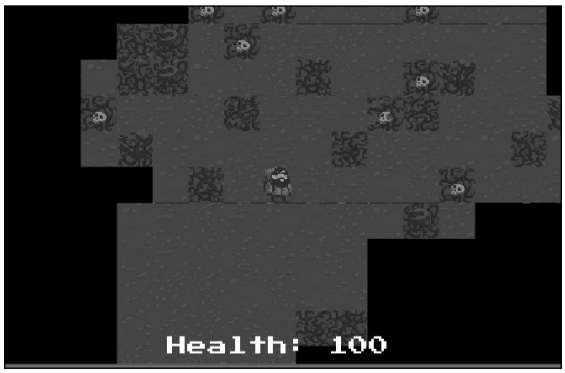
- `Line 29`: Like the floor tiles, we are going to add an array of `GameObject` to hold our wall tile prefabs.
- `Line 62-68`: This is our original `addTiles` function.
- `Line 71`: This condition uses random numbers to create a probability. We randomly choose a number between 0 and 2. If the number is 1, then we add a wall tile to the newly created floor tile. There is a 1 in 3 or 33 percent chance that a wall tile is added.
- Line 72-74: We instantiate the wall tiles as we do the floor tiles.

So, if we return to the Unity Editor and play the game, we get some errors. This is because we added the array for the wall tiles, but it is currently empty. You will need to add the wall tiles to the `GameManager` prefab the same way you did the floor tiles.

In the `BoardManager` script component of the `GameManager` prefab, follow these steps:

1. Set **Size** under **Wall Tiles** to `8` and press *Enter*.
2. Then, drag and drop **Wall1** to **Wall8** into the newly created **Element0** to **Element7** under **Wall Tiles**.

Finally, the Game Board is fully functional! Press the play button to test it out. The Game Board will expand as the player explores. Every time you play the game, you will experience a different Game Board.



PCG Game Board plus wall tiles

With the addition of high frequency wall spawning, there are plenty of obstacles. These walls will also make it more difficult to run from enemies. The PCG nature of the Game Board makes for a unique play of the game every time.

Summary

Our *Roguelike* game is coming along. You completed your first PCG feature from design to development. However, there is still plenty left to do.

In this chapter, you learned about and analyzed a few different data structures. You designed an algorithm that will expand the Game Board as the player explores. You set up the scene so we could implement our PCG algorithm by adding layers and a player tracking camera feature. And finally, you implemented our algorithm design and created a procedurally generated game world.

There's still more game world to procedurally generate. In the next chapter, we are going to develop a different kind of level building. We will be creating a random dungeon generator. This will present a new set of PCG algorithm challenges.

Where to buy this book