Contents

1 Introduction

This Dungeon Crawler Sample is a simple top-down game to demonstrate the functionality of DunGen.

The player starts in a castle environment filled with treasure chests containing gold (purely cosmetic, no gameplay purpose) and a key used to progress to the second half of the dungeon – the graveyard.

At the end of the graveyard, there is an exit portal. If this were a real game it would take the player to the next floor, but for the sample we just regenerate the dungeon instead.

This documentation will explain how the sample project works but will gloss over how to actually make use of the described DunGen features. For instructions on how to use each of the features, see the included Readme document for DunGen.

The sample and this documentation were made to provide an example of how to use DunGen. Any gameplay code or other functionality unrelated to DunGen is outside of the scope of this documentation and we'll be unable to provide support. That said, the scripts included in the sample are well commented and can be used to understand how the sample works.

2 Project Structure

2.1 Scene

There are three main objects in the sample scene:

- **Dungeon Generator** – Contains DunGen's Runtime Dungeon component responsible for actually generating the dungeon layout.

  There's also the Dungeon Setup component which performs some tasks after the dungeon generation is complete. It's responsible for spawning the player and hooking up the UI.

  Finally, the Dungeon Crawler Nav Mesh Adapter component generates a dungeon-wide navigation mesh when the generation is complete. Normally, we'd use DunGen's built-in integration for the Unity NavMesh components, but in order to avoid requiring any dependencies for the sample, we build the nav mesh manually.

- **Render NavMesh** – Using the Render Nav Mesh component, we wait for the dungeon generation to complete, then convert the generated navigation mesh into a renderable mesh. Every frame, we tell Unity to render this mesh using *Graphics.DrawMesh()*.

- **UI** – Contains all the objects required for rendering the player's user interface. The functionality for all of this is contained in the Player UI component.

2.2 Layers

A few special layers are used in the sample for various purposes. The sample uses only built-in layers from Unity so there's no need to add custom layers to the project.

- **TransparentFX** – Used for objects that should ignore raycasts for player movement. This is used on the walls so they don't block player input but can still block rays used to determine if a wall is occluding the player (for hiding walls).

- **Ignore Raycast** – Used for objects that should not contribute to the navigation mesh.

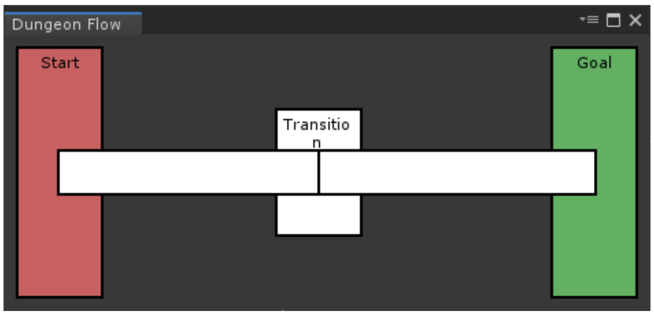
- **Water** – Rendered only by the minimap camera. This is used when rendering the navigation mesh so we have an outline of the walkable area for the minimap.

- **UI** – Rendered only by the minimap icons camera. Used to render minimap icons on top of the dungeon layout for the minimap.

3 Dungeon

3.1 Flow

The dungeon flow is fairly simple. The start, transition, and goal nodes each use a TileSet with a single tile designed for their respective purposes.



The flow is split into two archetypes. The first half of the dungeon is made up of tiles from the castle archetype, while the second half is from the graveyard archetype.

The transition node is marked to contain a locked door on the exit.

3.2 Tiles

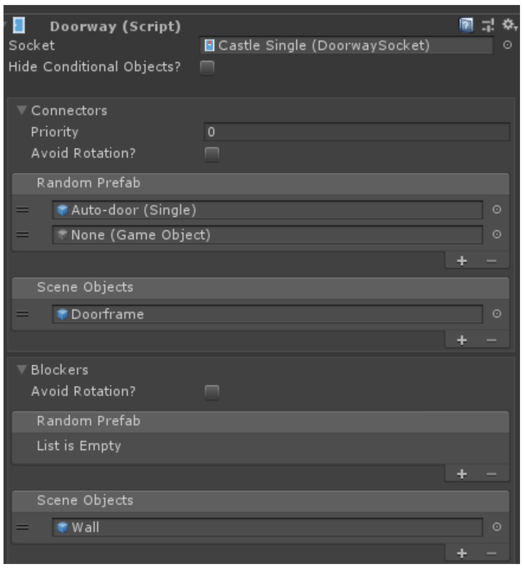
There are 6 TileSets in the sample:

1. **Start Tiles** – A single tile that acts as the spawn room
2. **Castle Tiles** – The basic castle tiles found in the first half of the dungeon
3. **Obelisk Tiles** – Contains a single special tile of which a single copy is injected into the first half of the dungeon
4. **Transition Tiles** – A single tile which was specially made to bridge the gap between the two halves of the dungeon. This is where the locked gate will be attached
5. **Graveyard Tiles** – The basic graveyard tiles found in the second half of the dungeon
6. **Goal Tiles** – A single tile that acts as the goal room

3.3 Doorways

There are three doorway sockets used in the sample. One for single castle doorways, one for double castle doorways, and one for graveyard doorways.

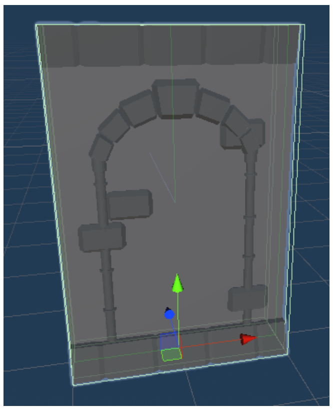
For convenience, there is a doorway prefab for each socket type. This lets us place the already set up doorways into our tiles with ease.



```
Doorway (Script)
Socket                     Castle Single (DoorwaySocket)
Hide Conditional Objects?  ☐

▼ Connectors
  Priority                 0
  Avoid Rotation?          ☐
  Random Prefab
    Auto-door (Single)
    None (Game Object)
  Scene Objects
    Doorframe

▼ Blockers
  Avoid Rotation?          ☐
  Random Prefab
    List is Empty
  Scene Objects
    Wall
```



The single castle doorway is set up with a mesh for both the doorframe and a wall. These are selectively destroyed at runtime based on whether the doorway is in use or not by referencing the meshes in the “Scene Objects” list of either the “Connector” or “Blocker” sections.

4 Gameplay

4.1 Player Movement

The player moves using a navigation mesh by left clicking on a point on the floor. Logic for this is handled in the PlayerInput and ClickToMove components.

4.2 Chest

The treasure chest prefabs are hand-placed inside each tile and use DunGen's Global Prop component to limit the number that can appear in the dungeon.

The logic is contained in the TreasureChest component which also implements DunGen's IKeySpawnable interface for allowing the key to spawn in a chest at random.

4.3 Obelisk

The obelisk room is placed using tile injection to guarantee that there is exactly one such room in the first half of the dungeon.

4.4 Transition Gate

Logic is contained in the TransitionGateway component which implements DunGen's IKeyLock interface which assigns a key to the lock.

4.5 Exit Portal

The exit portal prefab is hand placed in the goal tile.

5 Extras

These systems are unrelated to DunGen and are therefore outside of the scope of this documentation, but the basic idea of how they work will be explained here.

5.1 Hiding Walls

Walls are marked with a HideableObject component which is responsible for keeping track of relevant renderers and colliders, as well as caching hierarchy information so that hideable objects can propagate their state through their hierarchy. The DungeonSetup component refreshes these hierarchies when the dungeon generation is complete.

Hiding/unhiding objects is managed by the ObjectHidingCamera which performs a sphere cast between the camera and the player to determine which objects are occluding the player character.

5.2 Minimap

We use the navigation mesh as a quick and dirty solution to rendering an outline of the walkable area in the dungeon. The player prefab contains the GameObjects responsible for rendering the minimap. The process for rendering the minimap is not very performant, the steps include:

1. Render the navigation mesh from the minimap camera into an off-screen buffer
2. Take the previous results and render it to another buffer using a custom shader to convert it into a signed distance field
3. Use another custom shader to take the distance field and render the base minimap fill and outline
4. Render the minimap icons camera into an off-screen buffer
5. Layer the two output images on top of one another in the player’s UI