Dungeoneer Asset Package

Dungeoneer is a First Person RPG with a turn-based menu driven combat system, stat based power structure, and grid based maze exploration. Before picking what code you want to use I'd recommend running the MainSample scene found in the DungeoneerAssets root directory and trying out the game. Due to licensing all art assets are placeholders and not what was included in the game release. On the main camera is the Player script which has some options for starting with higher stats, gold, and spells so you can try out some of the different features without progressing through the game.

Turn Based Battle:

Every step the player takes has a chance of setting off a random encounter with an enemy. After a battle occurs the player has a lower chance to encounter an enemy for a certain number of steps. Once the player enters combat an enemy is created based on the metadata for the floor and the game enters battle mode. In battle mode navigation is disabled and the battle menu becomes available. The player can use normal attacks, spells, items, defend, and retreat. The player and enemy alternate combat turns, choosing actions to decrease the opponent's HP to 0. If the player wins they get XP and Gold. If the enemy wins the player is presented a choice, return to town or load a previous save.

Primary Scripts Used:

- BattleManager.cs
- Player.cs
- MonsterBase.cs (and derived classes depending on enemy type)

Stats

Both players and enemies have 4 core stats, Strength, Vitality, Intelligence, and Agility.

- Strength governs physical attack power.
- Vitality governs defense and affects the amount of HP.
- Intelligence affects spell power and the amount of MP.
- Agility is used to determine who gets the first move in a fight as well as the odds of dodging attacks.

The player can allocate 1 point into a chosen stat every time they level up. The player levels up by winning battles and gaining XP.

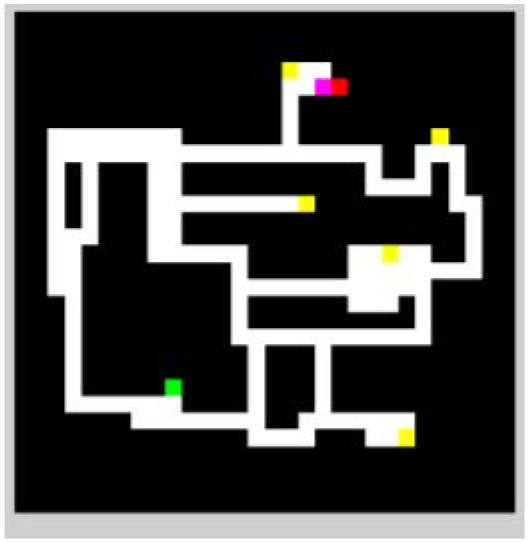
Primary Scripts Used:

- Stats.cs
- Player.cs
- MonsterBase.cs

Generated Maze Levels

Each floor of the dungeon is a square maze which is created using a png image. The pixel color in each position defines the location of walls, open spaces, the entrance and exit, treasure chests, and boss encounters. Green and Red indicate the entrance and exit respectively. Purple is the location of a Boss.

Yellow is a treasure chest. The B value on the RBG of yellow acts as an ID to determine what treasure is at that location. Each map file has a xml file of the same name which defines enemy and boss information, power of enemies, encounter rate, at what position the player triggers the entrance or exit, and what direction they should be facing when they come out of it. These files are located in the Resources folder so they can be read at runtime.



Primary Scripts Used:

- FloorManager.cs
- Floor.cs

Enemy Creation

Based on the metadata file for the current floor the system determine a random range of stat points to give to an enemy. Stats are allocated based on the enemy type, so Fighters will focus on strength, Mages on intelligence, and so on. The enemy metadata also includes display name as well as a reference to the image to be used. The enemy type correlates to the AI pattern that will be used as well.

Primary Scripts Used:

- MonsterFactory.cs
- MonsterStatWeights.cs

- MonsterStatCalculator.cs

Equipment / Items / Spells

Equipment, items, and spells share a common set of metadata which includes a lookup key, display name, description, and purchase cost.

Equipment has 4 categories; Sword, Shield, Armor, and Helmet. Equipment will directly affect Physical Attack, Magical Attack, Physical Defense, or Magical Defense, which have a base value calculated from the player's stats.

Items have a Func<Player, bool>, CanUse, which given the player instance will determine if the player can use that item. Health items can't be used when the player has full health and similar situations are taken into account using that function. The Func<int>, CalculatedEffect, is used to provide a power of an item such as how much health a type of potion restores.

Spells are similar to items in that they require a CanUse and CalculatedEffect function. There is an additional StatRequirement property that defines some prerequisite stats that the player must have to use the spell. In addition the CalculatedEffect function takes into account the player's intelligence to determine the resulting strength of a spell.

Primary Scripts Used:

- EquipmentDefinitions.cs
- ItemDefinitions.cs
- SpellDefinitions.cs
- Inventory.cs

Stores

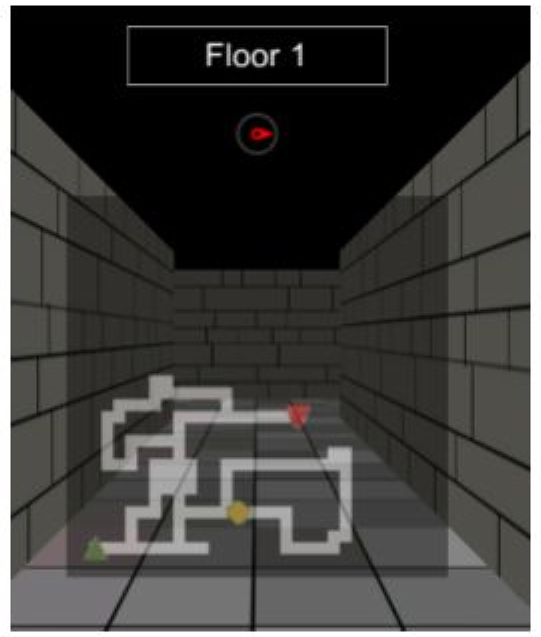
Stores can be accessed in town to buy equipment, items, spells, or restore the player's HP and MP for a cost. These things cost gold, which is gained after each successful battle. They use a paginated display to determine the number of items that can fit onto a page based on screen resolution and will determine the number of pages and spacing required at runtime.

Primary Scripts Used:

- ShopBase.cs
- PaginationBase.cs
- ItemShop.cs

Map

Information about where the player has navigated through in the dungeon is saved and can be viewed at any time. The map keeps track of where the entrances and exits of each floor are at and show all spots where the player has stepped. That data is used to render a texture on demand.



Primary Scripts Used:

- PlayerMap.cs
- MapRenderer.cs

Content Replacement

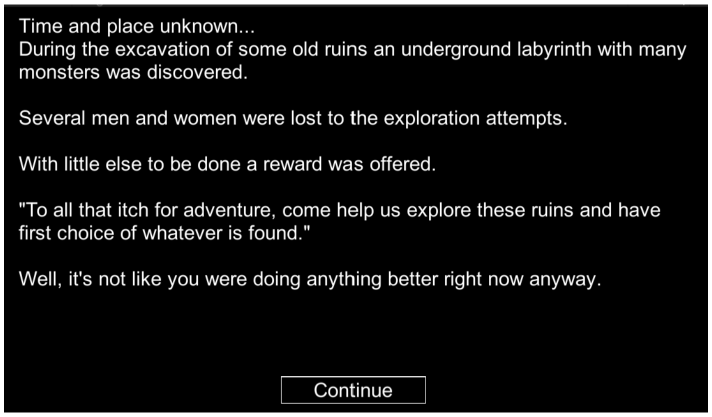
Since the package is a straight copy of an existing game I sell, including the non-licensed content in that game, it would be good to point out where some of the content that hasn't already been described is and how to replace it with your own content.

Credits

The items on the credits screen can be set from the CreditsScreen panel there is a "StockPanel" which has 5 text boxes. Each of these text boxes are a line on the credits screen.

Opening

Once you hit new game a text with some story appears. This text is under the "OpeningScene" panel.


Time and place unknown...
During the excavation of some old ruins an underground labyrinth with many monsters was discovered.

Several men and women were lost to the exploration attempts.

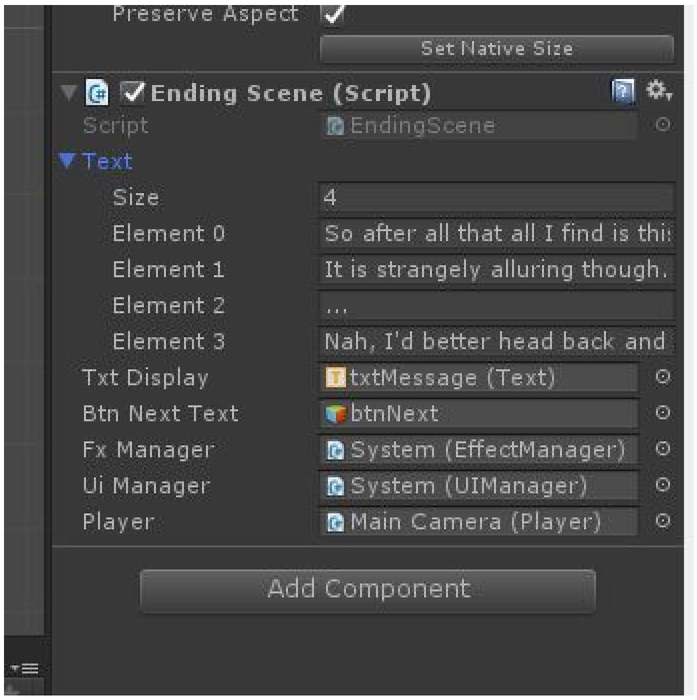
With little else to be done a reward was offered.

"To all that itch for adventure, come help us explore these ruins and have first choice of whatever is found."

Well, it's not like you were doing anything better right now anyway.

Ending

At the end of the game a bit of dialogue/narrative appears on a timer. This “EndingScene” panel hosts this. The text content is in the text array in the inspector except for the last line which due to the complex timing is hard coded in the EndingScene.cs script file. The ending scene is enabled through the floor manager by using the special floor name “Ending”


Preserve Aspect ✓

Ending Scene (Script)

Script	EndingScene
Text	
Size	4
Element 0	So after all that all I find is thi
Element 1	It is strangely alluring though.
Element 2	...
Element 3	Nah, I'd better head back and
Txt Display	txtMessage (Text)
Btn Next Text	btnNext
Fx Manager	System (EffectManager)
Ui Manager	System (UIManager)
Player	Main Camera (Player)

Inn

The UI for displaying the Inn section of town is just called “Inn” in the inspector. Some of the dialogue is defined in the Inn.cs file, and the button for displaying chatter which gives the player random tips pulls from text lines defined in the ChatterDefinitions.cs script file.

Spells, Items, Equipment

The “SpellShop”, “ItemShop”, and “EquipmentShop” panels in the canvas each pull text content from the scripts of the same names. The actual listed items in each shop are found in the EquipmentDefinitions.cs, SpellDefinitions.cs, and ItemDefinitions.cs script files. Each defined object has a DisplayName and description which becomes available when the player clicks on the name in each listing. In addition to visual information the Equipment define an EquipmentValue object that defines what stats get increased when the equipment is worn and Items and Spells define both a CalculatedPower and CanUse Functions that are used as the names imply.

Spell usage and the text associated with it are defined in the SpellManager.cs script file. This gets called from the PlayerSpellMenu script which is on the menu shown when the player presses the spell button while in battle or from the dungeon menu.

Items work similarly to spells and are called from the PlayerItemMenu script which is used by the menu shown when the player hits the item button from the dungeon menu or in battle. That script uses the ItemManager.cs script to define usage.

The PlayerEquipmentMenu script is also used when the player opens the equipment menu from the dungeon menu, and works similarly except there is no additional manager class to use. The Equipment and Item screens are both used when selling items or equipment at the store, and have text for dealing with selling to the shop in the corresponding scripts.

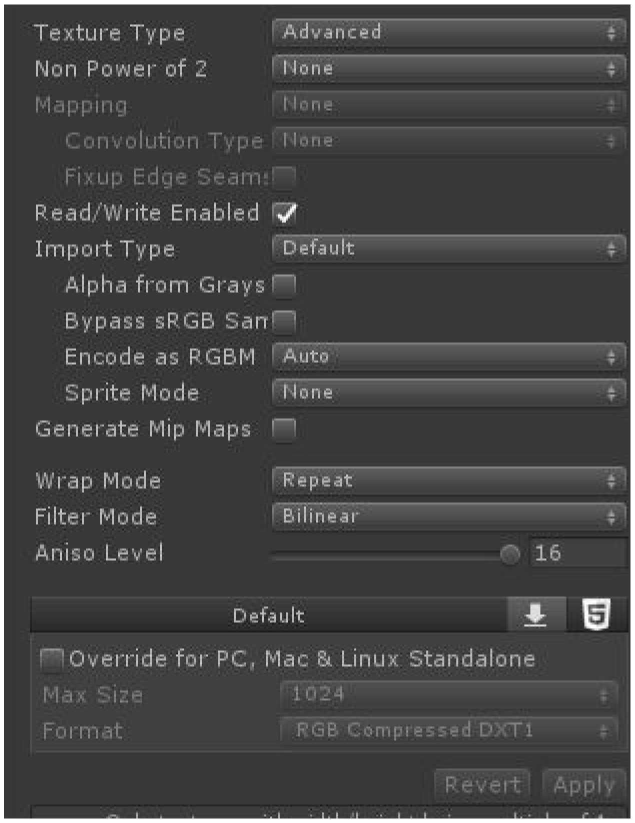
New Floors from Scratch

Each floor in the maze consists of a png image and an xml file. The image uses each pixel to determine which prefab to put at that x,z location on the generated floor. Black represents spaces where the player cannot walk, white represents floor spaces that can be walked on. Other colors represent some other stuff but to start with you’ll want to

1. Create a new png image in your image editor of choice.
2. Fill the entire image black.
3. Draw the path you want the player to be able to walk in white.
4. Add the Entrance to the map by adding a single green pixel which is connected to a white walkable area.
5. Add the Exit to the next floor by adding a red pixel somewhere that is reachable by the white walkable area.

Save your image with a distinct name and put it in the resource directory of the project. Alongside the image you’ll need to create a text file which will have our xml data about the floor. For this example we’ll call it “TestFloor”. At this time your resource directory should contain TestFloor.png and TestFloor.txt.

When you add a new floor image to the resource directory Unity will import it as a sprite image type. You need to change the import settings of the file to match the following so that the FloorManager code can read the pixel data from the image.



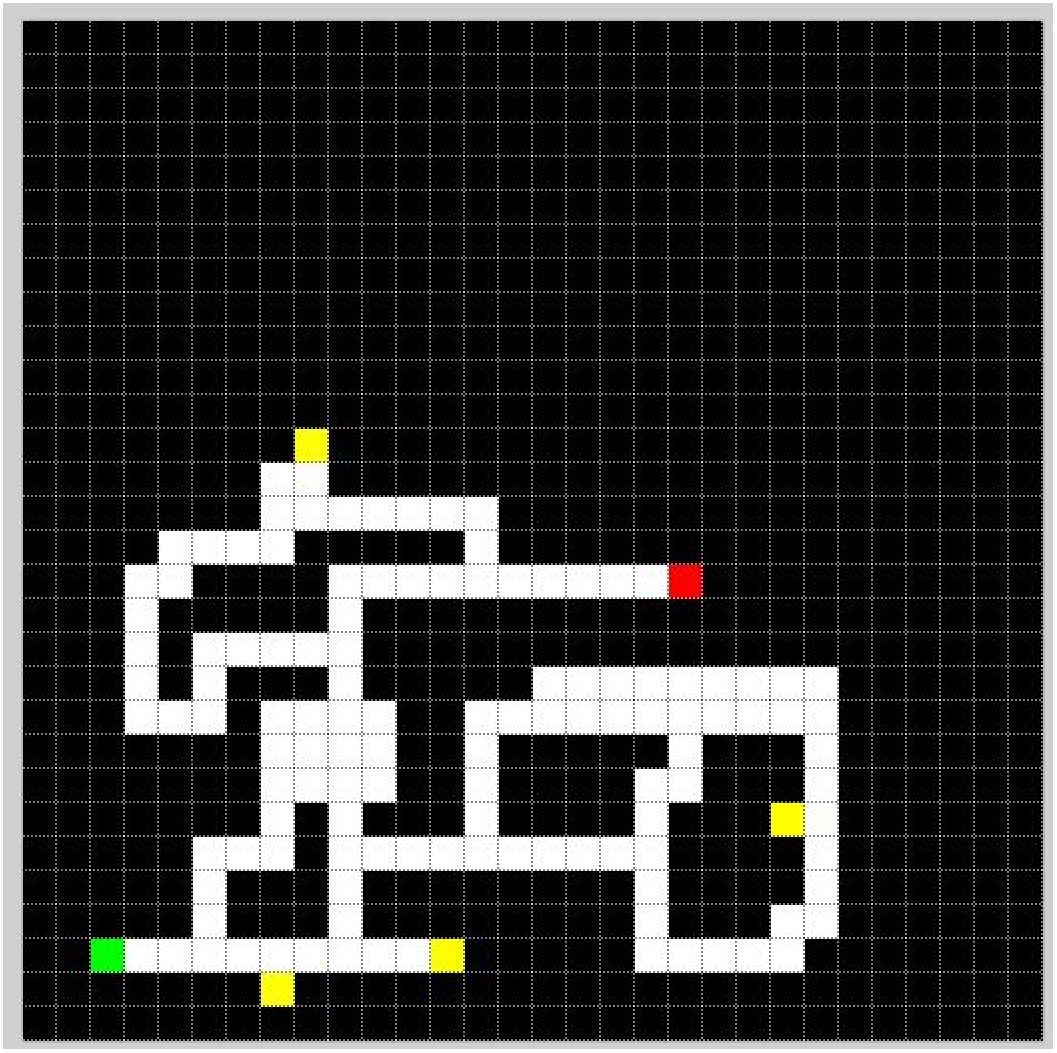
Texture Type	Advanced
Non Power of 2	None
Mapping	None
Convolution Type	None
Fixup Edge Seams	☐
Read/Write Enabled	✓
Import Type	Default
Alpha from Grays	☐
Bypass sRGB Sam	☐
Encode as RGBM	Auto
Sprite Mode	None
Generate Mip Maps	☐
Wrap Mode	Repeat
Filter Mode	Bilinear
Aniso Level	16

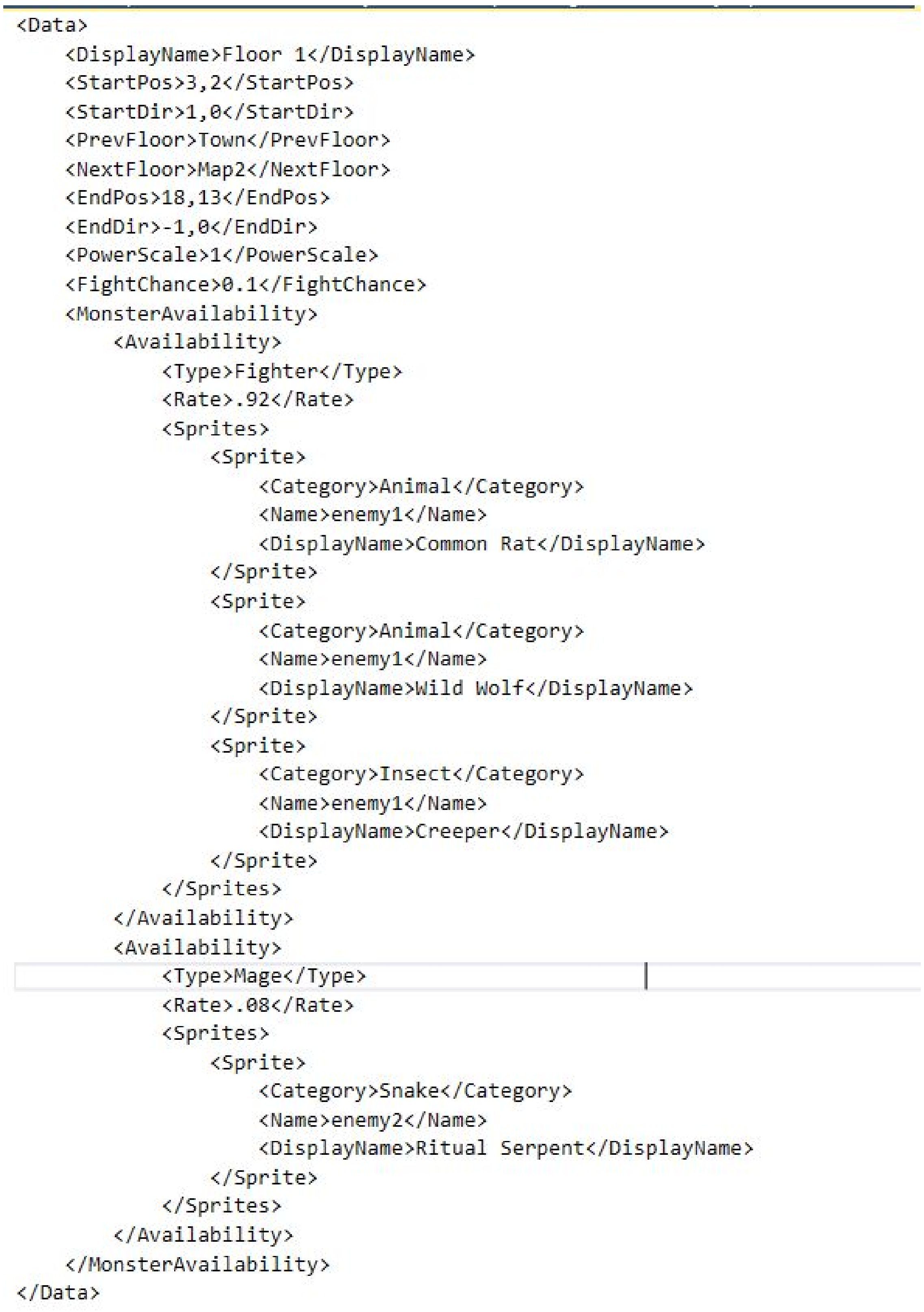
Default

☐ Override for PC, Mac & Linux Standalone

Max Size	1024
Format	RGB Compressed DXT1

Inside of TestFloor.txt we'll copy the xml from the existing Map1.txt from the sample and replace the information that will work with our floor. For reference here is the image and text from Map1 of the sample. In the example verbiage I'll be referencing an underground dungeon where going further down is progress and going up is backtracking. Additionally we'll refer to the next floor in terms of progress (going down) and previous floor will refer to the previous completed floor. A floor is considered completed when the player has passed through the exit to the next floor.


Map1.png



```
<Data>
    <DisplayName>Floor 1</DisplayName>
    <StartPos>3,2</StartPos>
    <StartDir>1,0</StartDir>
    <PrevFloor>Town</PrevFloor>
    <NextFloor>Map2</NextFloor>
    <EndPos>18,13</EndPos>
    <EndDir>-1,0</EndDir>
    <PowerScale>1</PowerScale>
    <FightChance>0.1</FightChance>
    <MonsterAvailability>
        <Availability>
            <Type>Fighter</Type>
            <Rate>.92</Rate>
            <Sprites>
                <Sprite>
                    <Category>Animal</Category>
                    <Name>enemy1</Name>
                    <DisplayName>Common Rat</DisplayName>
                </Sprite>
                <Sprite>
                    <Category>Animal</Category>
                    <Name>enemy1</Name>
                    <DisplayName>Wild Wolf</DisplayName>
                </Sprite>
                <Sprite>
                    <Category>Insect</Category>
                    <Name>enemy1</Name>
                    <DisplayName>Creeper</DisplayName>
                </Sprite>
            </Sprites>
        </Availability>
        <Availability>
            <Type>Mage</Type>
            <Rate>.08</Rate>
            <Sprites>
                <Sprite>
                    <Category>Snake</Category>
                    <Name>enemy2</Name>
                    <DisplayName>Ritual Serpent</DisplayName>
                </Sprite>
            </Sprites>
        </Availability>
    </MonsterAvailability>
</Data>
```

Map1.xml

I'll now describe each element in order.

- DisplayName - Appears at the top of the screen when the in game map is open.
- StartPos - When the player enters the floor from the above floor this is where the player will be standing. The coordinate is from the bottom left of the image. The very bottom left of the image is 0,0, so in the Map1 sample the starting coordinate is 1 pixel to the right of the entrance (green) pixel.
- StartDir - This is the X,Y direction that the player is facing when the enter the floor.
- PrevFloor - This is the name of the floor above the current one. When the player collides with the entrance a png and txt with this name will attempt to be loaded and used to generate a new floor.
- NextFloor - Same as the PrevFloor but in this case it's the next floor in game progression and is used to generate the floor after the player goes through the exit.
- EndPos - When the player enters the floor from the next floor, such as when the player is backtracking, this is where they will start at.
- EndDir - When the player enters from the next floor they will be facing this direction.
- PowerScale - This is a floatin point number used to determine the strength of enemies on this floor. This is used in both the MonsterFactory.cs and MonsterStatCalculator.cs script to generate monsters of a variable strength.
- FightChance - the chance of getting into a battle each step in ratio form. After a battle occurs this is lowered for a few steps and then returns to the value set here.

The MonsterAvailability section deals with what kinds of monsters the player can encounter. Under the resources directory there needs to be a folder called "EnemySprites". This will have the sprite representation of the monster. When a battle starts the availability rates will be calulated to determine which time of enemy the player will face, and after that a random sprite definition will be selected to be shown.

- Availability - This section represents a type of monster than can be encountered.
  - Rate - the chance of encountering this type of monster in ratio form.
  - Type - the AI pattern that this monster takes after. This is both a behavioral pattern in battle, and how the MonsterStatCalculator allocates stats. The possible classes are Fighter, Mage, Tank, GlassCannon, Balanced, SpeedDemon, which can be found by looking at the MonsterStatWeights.cs script as well as the corresponding AI script in the Monsters/AI script folder.
- Sprite - Each sprite defined is a different enemy that fits that set monster type.
  - Name - the name of the sprite file to use from within the EnemySprites folder.
  - DisplayName - what will appear in the name field during battle for this enemy.

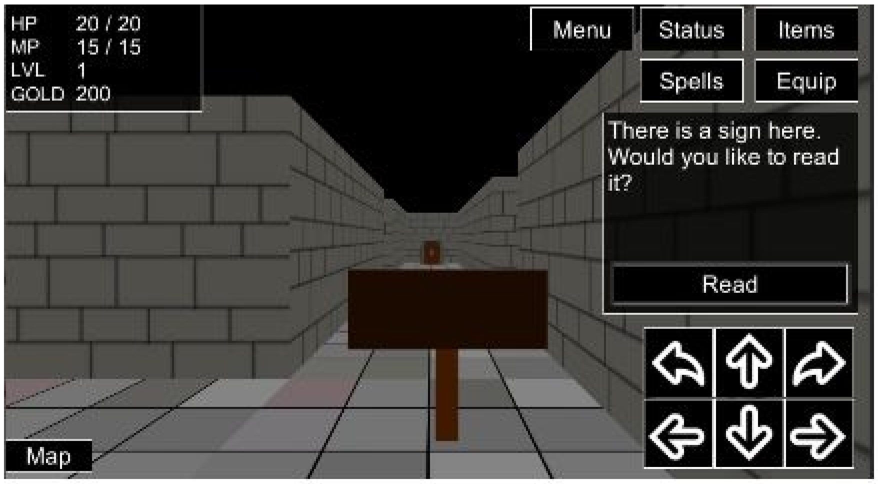
This gets us the basic setup of a floor that has a previous and next floor defined as well as a path and random encounters. It's important to note that the floor name "Town" is a special keyword that will take the player back to the town. This is defined in the FloorManager.cs script. In the DungeonEntrance.cs script there is a hard coded value for going to the first floor of the Dungeon that you'll want to change as well if you use a different naming convention.

Treasure

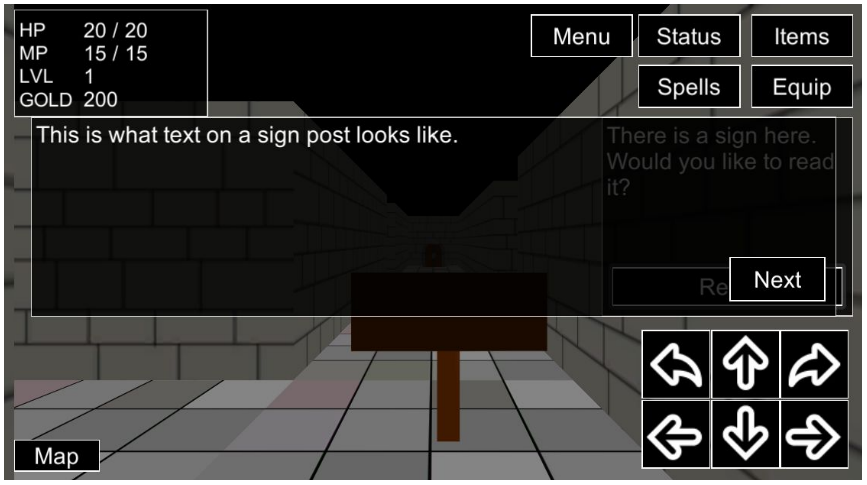
You can add treasure to the dungeon floor map by editing the image for that floor. To add a treasure add a pixel that has an RGB value of (255, 255, X). X in the blue position will be the id of the treasure that's in the chest which is defined in the TreasureDefinitions.cs script. So if you want a healing potion which is id 2 then your pixel would be (255, 255, 2).

Text Events

A text event will take two forms, either it will be a sign post type that can be read multiple times via the same interaction that a treasure chest has, or it will be a set triggered event which will be triggered when the player walks over a certain tile. Text events that are signposts are repeatable by nature, but triggered events can either be repeated or be one time only events.



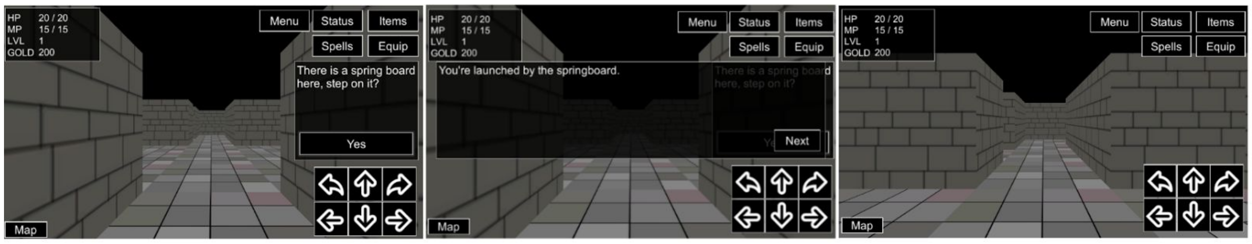
There is a sign here. Would you like to read it?



This is what text on a sign post looks like.

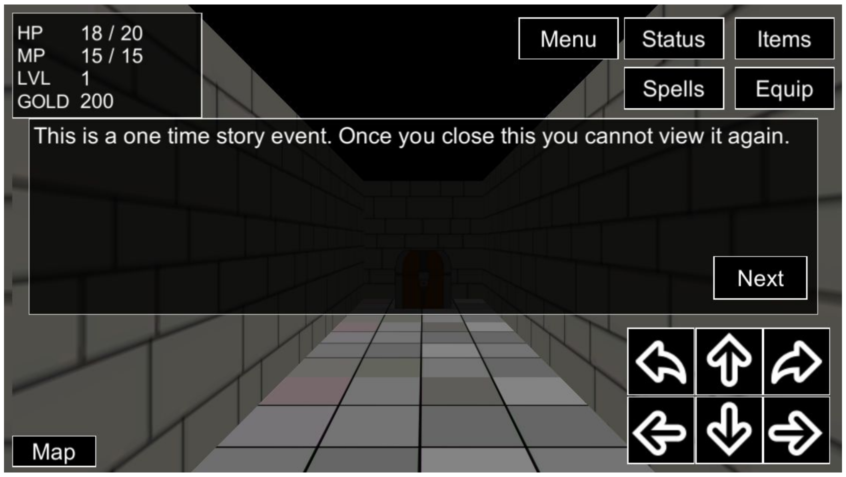
There is a sign here. Would you like to read it?

Some events can be triggered to teleport the player to another location on the floor. These are defined by the Teleport and TeleportTarget properties in the TextEventDefinition described below.



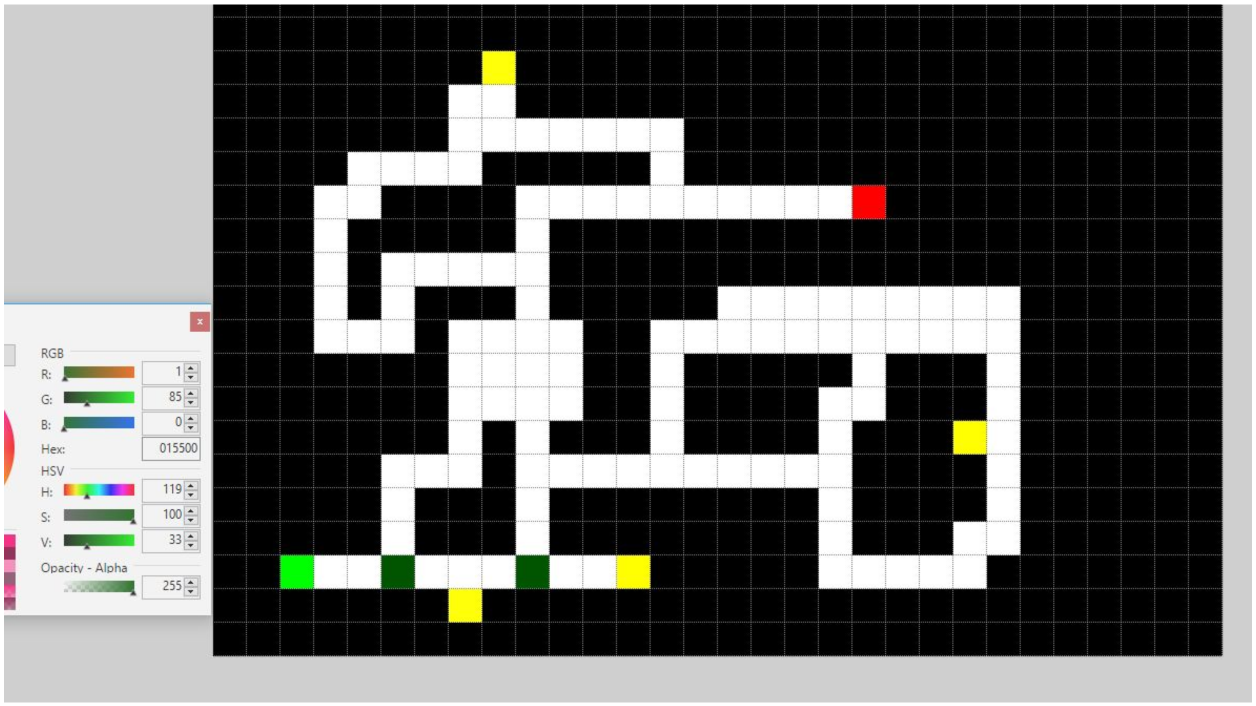
There is a spring board here, step on it?

You're launched by the springboard.



This is a one time story event. Once you close this you cannot view it again.

To add a text event you use the R value of the pixel RGB on the source image. For clarity I use a dark green so that the pixel doesn't appear too close to black to distinguish at a glance. Be aware that having a B value will cause a treasure chest to appear here instead since treasure is given priority.


RGB
R: 1
G: 85
B: 0
Hex: 015500
HSV
H: 119
S: 100
V: 33
Opacity - Alpha 255

The R value of the pixel will correspond to the index of a text event item defined in the TextDefinitions.cs script file.

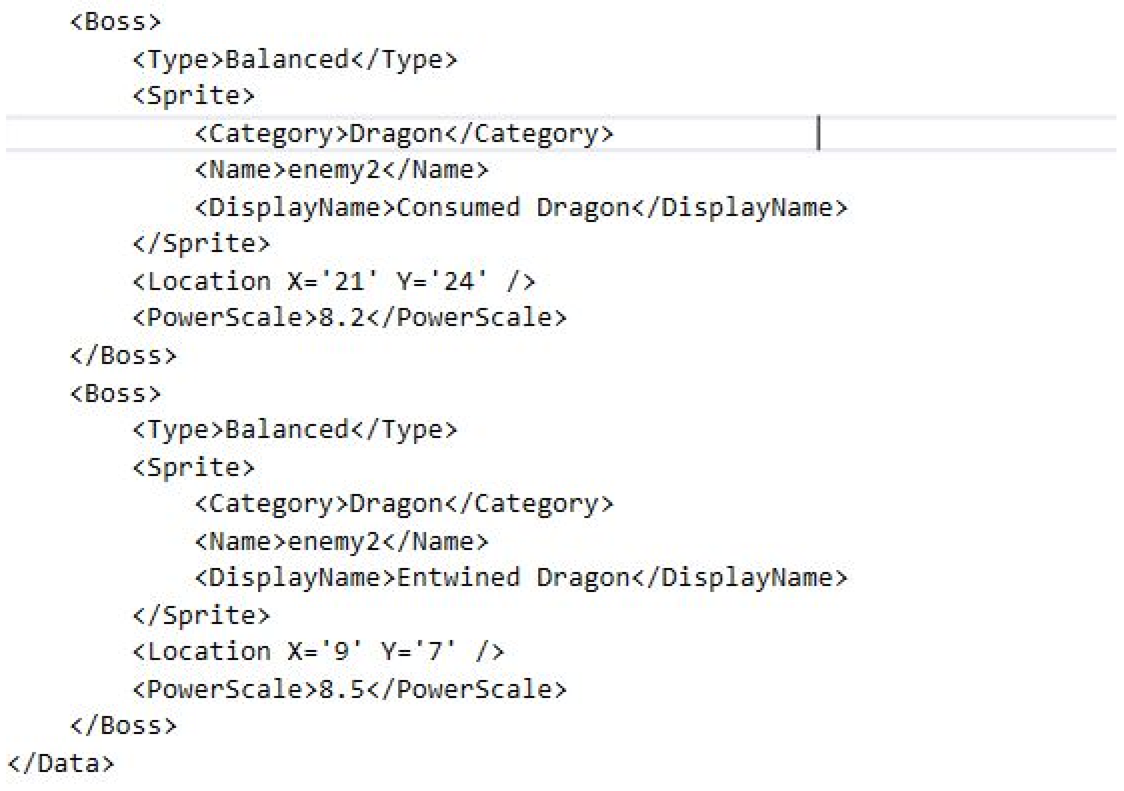
```
textEvents.Add(1, new TextEventDefinition
{
    Text = new string[] { "This is what text on a sign post looks like.",
     "It can be read multiple times" },
    ReadPromptText = "There is a sign here. Would you like to read it?",
    ShowSignpost = true
});

textEvents.Add(2, new TextEventDefinition
{
    Text = new string[] { "This is a one time story event. Once you close this you cannot view it again." },
    ViewOnce = false,
});
```

Text events are given the following properties:
Text: this is an array of text lines to be shown. Each element of the array represents a different page of the text to be shown. Pressing the next button will move to the next line of text with the last press closing the message box.
ShowSignpost: This will determine whether your text event is an interactable sign post that is read more than once, or a trigger when you step on a certain tile.
ReadPromptText: This is the text that appears when confirming that you want to read a signpost. Text and option appears when the player is facing the signpost.
ViewOnce: This is used by trigger text events. If ViewOnce is true then the game will record if the player has seen a text event already and not show it a second time.

Teleport: This denotes that the text event will teleport the player to another coordinate on the same floor.
TeleportTarget: The (x,y) coordinates for where the teleport will put the player. Like all other coordinates the bottom left of the map is (0, 0) and values increase positive towards the right and up.
Prefabs used are attached to the FloorManager on the system object and textures, materials, and prefabs for both trigger and signpost text events can be found in the Art/Models directory.

Bosses

Bosses are special monsters that appear in the dungeon. They remain stationary and will usually be used to block the player from reaching some valuable treasure or reaching the next floor. If you refer to the Map8 in the resource folder then you'll see the boss on the map which is marked by the purple pixel (255, 0, 255). There is an additional XML section that we need to add for the boss as well.



```
    <Boss>
        <Type>Balanced</Type>
        <Sprite>
            <Category>Dragon</Category>
            <Name>enemy2</Name>
            <DisplayName>Consumed Dragon</DisplayName>
        </Sprite>
        <Location X='21' Y='24' />
        <PowerScale>8.2</PowerScale>
    </Boss>
    <Boss>
        <Type>Balanced</Type>
        <Sprite>
            <Category>Dragon</Category>
            <Name>enemy2</Name>
            <DisplayName>Entwined Dragon</DisplayName>
        </Sprite>
        <Location X='9' Y='7' />
        <PowerScale>8.5</PowerScale>
    </Boss>
</Data>
```

This section is sibling to the MonsterAvailability section. You can define multiple bosses by adding additional boss sections. Just like the MonsterAvailability section the Sprite/Name and Sprite/DisplayName section represent the sprite file to use and the display name on screen during battle. Type represents the AI pattern to use.

- Location - the X, Y location on the map that the boss appears at. This corresponds to the position that the purple pixel is at. This information is used in the save data for determining if a boss has already been defeated or not.
- PowerScale - Allows you to define a powerscale for the individual enemy that is different than the normal enemies on the floor.

The bosses will appear on the overworld in a stationary position and won't respawn if the player leaves and re-enters the floor.

Town Background Images

The town section as well as the various stores have black backgrounds in the package, but for an actual game you should provide some kind of visuals to differentiate which area your player is in. I've used placeholder black backgrounds in the Art/Backgrounds folder but you should definitely replace each of these. The images in the project correspond to the following UI panels.

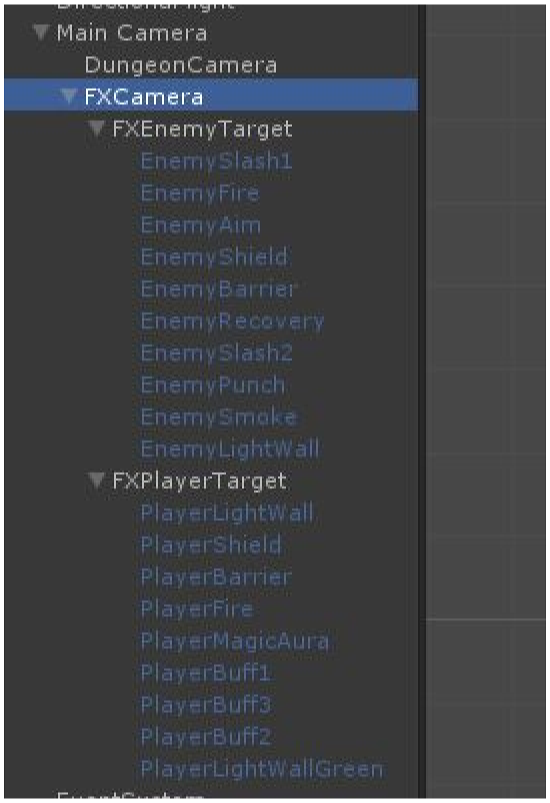
Outside	background_12
ItemShop	background_34
EquipmentShop	background_2
SpellShop	background_14
Inn	background_11
Ending	background_71

Audio

Music and sound effects are a large part of your game. In the scene hierarchy is a game object called the AudioManager. That has the AudioManager.cs script as well as several audio sources attached to it. One audio source is set in the audio manager as the BGM audio source, and the remainder are SFX audio sources. Only one source will play the background music, but it's possible for multiple sound effects to play at the same time depending on how fast your player is performing actions. The audio manager script also has listings for many different audio clips which in the latest version of this package are grouped by usage type in the editor. The audio clips are all played via the AudioManager PlayBGM or PlaySFX methods which provide an enumerated value for which audio to play.

Particle Effects

There is an effect for most actions taken in the game. This is primarily caused during battle but some effects will happen when the player casts a spell or uses certain items which can happen outside of battle as well. In the scene hierarchy there is an FXCamera under the MainCamera.



```
▼ Main Camera
    DungeonCamera
  ▼ FXCamera
    ▼ FXEnemyTarget
        EnemySlash1
        EnemyFire
        EnemyAim
        EnemyShield
        EnemyBarrier
        EnemyRecovery
        EnemySlash2
        EnemyPunch
        EnemySmoke
        EnemyLightWall
    ▼ FXPlayerTarget
        PlayerLightWall
        PlayerShield
        PlayerBarrier
        PlayerFire
        PlayerMagicAura
        PlayerBuff1
        PlayerBuff3
        PlayerBuff2
        PlayerLightWallGreen
```

This camera operates at a different depth and only renders items on the FX layer. Under that camera are two categories, FX that happen at a bit of a distance where the enemy should be, and closer to the actual perspective where the player would be. The positions of the particle FX defined under each target section might vary slightly depending on the effect, but for the most part they should fall around the location given. In the scene hierarchy is a “System” object. One of the components on this object is a EffectManager.cs script which operates similarly to the AudioManager. It has prefab references for each effect that is used in the game. Effects are played from a few different places in the code depending on usage via the ShowEffect method which is for effects that play once and don’t look, and StartEffect / StopEffect to control effects that loop and are active for a longer term such as shield spells. Like the AudioManager effects are referenced via enumeration.