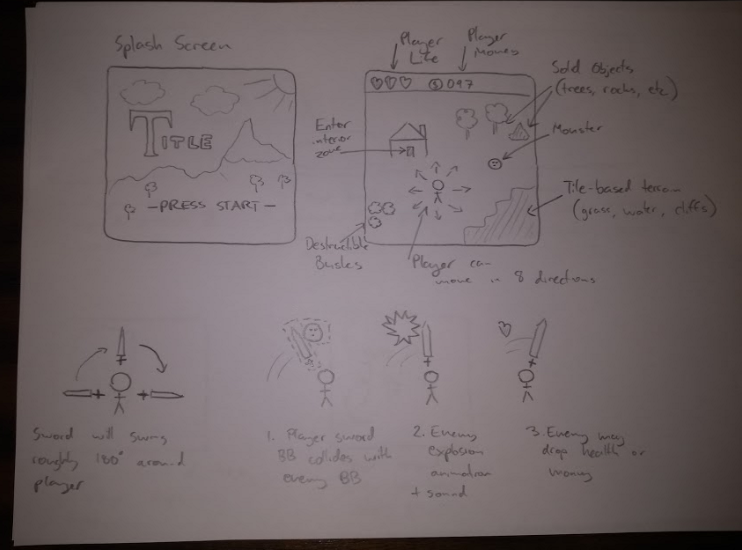
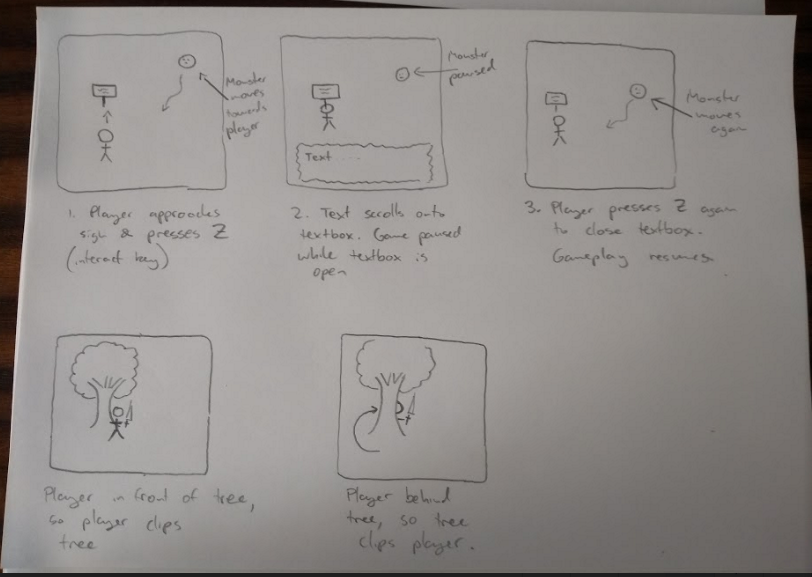
**Goal:** My aim was to create a 2D, top-down action/adventure game similar in design and atmosphere to an early *Legend of Zelda* game. I also wanted to build my own framework to achieve this.

**Storyboards:** The drawings below show my general plan for the splash screen, game levels, combat and dialog, and perspective.





Most of my planning, however, came from looking at and replaying old 2D *Legend of Zelda* games (mainly those on the Gameboy Color: Link’s Awakening and Oracle of Ages/Seasons), and thinking about how I might implement some of their features and general look and feel:



**Framework Outline**

Levels were designed in a helper program I wrote, LevelEditor (included in the submission). In brief, this allows objects to be positioned on a 20 \* 20 grid in five layers:

1. Tile
2. Prop
3. Mobile
4. ZoneTriggers (enable transition from one game zone to another)
5. Interactible

Each layer is outputted as a separate text file with one entry for each grid position. For tiles, props and mobiles, an entry consists of the name of the tile/prop/mobile in question. For zoneTriggers and interactibles it includes some additional parameters.

In Game1, game-specific objects (e.g., the Player, monsters, signs, bushes, rocks) are defined as prototypes. Levels are initialised based on the values in their text files. For example, to place a monster in a level:

1. The monster’s name is read from the Level file
2. The monster’s name is used as a key for a dictionary of monster prototypes
3. The relevant monster’s values are obtained via a copy constructor
4. The monster’s position in the level is set based on the location of its entry in the Level file

The remaining classes are essentially game-independent and could form the beginning of a general 2D game framework.

**Screen Management**

The interface **IGameScreen** defines the capacity to be drawn to the screen as a game state, and is implemented by a variety of classes:

**Level**: a game level (described in detail below)

**DisplayScreen**: a static screen like the title, defeat and victory screens

**OverlayScreen**: a screen that is drawn on top of the level. Currently used for the help screen

**DialogScreen**: a screen that is drawn on top of the level, with scrolling text

**HeadUpDisplay**: drawn above the level, displays player health and money

**FadeTransition**: fades between two levels, as when entering a house or cave

**ScrollTransition**: scrolls between two levels, as when walking between screens in the overworld or a cave.

The static class **ScreenManager** is used to set the current IGameScreen. The update / draw loop in Game1 simply calls ScreenManager’s Update and Draw methods, which then call the Update and Draw method of the current IGameScreen. Changing the current game screen is done via a call to ScreenManager. ScreenManager also controls playing music.

**Game Objects**

The following is a rough class hierarchy of the classes used within a single game level:

**WorldObject**: an abstract class; anything that exists within a Level and can be drawn

**Prop**: a static WorldObject. Eg, a rock. Has a **Sprite** object, which manages drawing a texture (static or animated) to the screen

**DestructibleProp**: a Prop that is destroyed when it collides with the Player’s sword. Eg, a bush.

**InteractibleProp**: a Prop that performs some function when the Player approaches it and presses Z. Currently, the only function implemented is to display some dialogue. Eg, a person or a sign.

**CollectibleProp**: a Prop that is ‘collected’ when the Player collides with it. Eg, a heart or a coin.

**Mobile**: a WorldObject that can move according to some MovementStrategy. May have multiple **Sprite** objects (eg, one for each direction).

**Monster**: a Mobile with health that damages the Player upon collision, and that the Player can destroy

**Player**: the entity controlled by the user.

**Level Management**

The static class **LevelManager** is responsible for holding all game levels and updating the current level. Each level belongs to a zone, and has an XY position within that zone, which is used to manage scroll transitions between levels in the same zone. Transitions between separate zones are achieved with the **ZoneTrigger** class, an invisible tile whose values are set in the LevelEditor. LevelManager contains a wide range of helper functions for handling different types of collisions, and manipulating lists in the current Level.

The **Level** class defines everything inside a single game screen. In brief, it holds lists of:

-Tiles and associated bounding boxes

NB: tiles are just references to textures, not real game objects

-WorldObjects: things that get drawn to the screen

-Props

-Collidable Props

-Mobiles

-ZoneTriggers

-Interactible Props

Some of these lists are subsets of the others (eg, collidable props are a subset of props) and are used to reduce processing time when updating the level (for instance, when determining if a collision has occurred). Adding and removing items from these lists (eg, a monster is killed, and it drops a coin) is handled within LevelManager

**Layering Strategy**

1. Game1 clears the screen and calls ScreenManager’s Draw method
2. ScreenManager calls the Draw method of the current IGameScreen
3. If the IGameScreen is a DisplayScreen, simply draw it
4. If the IGameScreen is an OverlayScreen or a DialogScreen, draw the underlying Level first, then the overlay
5. If the IGameScreen is a FadeTransition, draw the underlying Level with the colour filter determined by the length of time since the FadeTransition was initialised
6. If the IGameScreen is a ScrollTransition, draw two Levels and set the view matrix to be some point between the two, based on the length of time since the ScrollTransition was called
7. If the IGameScreen is a Level, or incorporates drawing a Level:
   1. Draw the tiles
   2. Draw the level’s WorldObjects by calling the draw method of each object in the list. WorldObject is a parent class to all drawable game objects. Different WorldObjects may draw themselves in different ways - for instance, a ghost will be drawn with partial transparency, and the player may be drawn with an attached sword sprite.
   3. Draw the HUD

Then main layering challenge within a level was to create a sense of ‘3/4' perspective. For example, the player should be drawn in front of a tree if they are in front of it in the level, but they should be blocked by the tree if they are behind it. I achieve this by sorting the list of WorldObjects based on the Y-position of each object’s bounding box, which was set to correspond to the bounds of the object at ground level. This achieves a reasonably good approximation of perspective.

**Major Challenges:**

**1. Building Levels**

Manually specifying the layout of a Level would have been extremely time-consuming and tedious. I addressed this by writing my own tile-based level editor (see above. A copy of LevelEditor1 is included in the submission).

**2. Generating multiple, independent game objects within a level**

Early versions of my level constructor worked by referencing predefined game objects. This worked for objects that never changed (eg, trees and rocks), but it produced strange errors when different objects of the same type were expected to do different things. For example, all monsters in the level would move in exactly the same pattern, because it was actually just one monster being drawn multiple times. To fix this, I used the Prototype Pattern: game objects were given a Copymethod that returned a new object with identical values. This was called within the level constructor to create independent objects

**3. Adding scroll transition**

Scrolling from one screen to another is a highly recognisable feature of early Zelda games and one I was determined to implement. The difficulty was that in the early stages of game development, the Draw method consisted of drawing a Level directly to the screen, and it was unclear how this could be modified to incorporate scrolling between two levels without introducing a lot of complication (e.g., manually shifting every object on the screen)

Two major changes were required to implement scroll transitions. First, developing the ScreenManager class allowed a decoupling of game logic and drawing logic. Second, the development of a ScrollTransition class, drawn by ScreenManager that manipulates the game’s view matrix in order to create a scrolling effect. Basically, ScrollTransition works as follows:

1. When a level transition is called, draw the new level at whatever location is required. This will initially be entirely outside the view window
2. Each frame, apply a slightly larger translation to the default matrix. This will create an effect of sliding from one screen to another
3. When the translation is equal to the width/height of a level, the new level is entirely in view. The game can then go back to drawing a single level.

**4. Adding a flexible movement system**

The main way the game was given a sense of ‘life’ was by the player and monsters moving around the screen. I wanted to use a wide range of different movement algorithms (getting movement from the keyboard; following the player; running away from the player; moving randomly in any direction; moving randomly in any cardinal direction, etc.), I wanted to be able to switch movement algorithms easily, and I wanted some monsters (eg, the boss) to be able to change between movement algorithms in game. Creating a different Monster class for each type of movement would have been extremely unwieldy. Instead, I used the Strategy Pattern: I defined an interface, IMovementStrategy, which specified two methods, Copy and Resolve. Each mobile has an IMovementStrategy, and its Resolve method is called in each Update. I could then write classes implementing IMovementStrategy that controlled how movement actually took place.

**5. Adding ‘events’**

As the game progressed, it became clear that I needed a flexible way of defining how in-game events were responded to. For example, ‘when the player collides with a gold coin, remove the coin from the screen, play a sound and add 10 to the player’s money’. Or ‘when the boss dies, remove the rocks blocking the path to the final room.’ Trying to code these responses inside game objects would have either lead to a profusion of classes or a profusion of if-statements, since different instances of the same class were expected to do different things. I solved this by using C#’s delegate feature, which basically allows functions to be passed as values. This allowed me to remotely define small sequences of events (which I called Scripts), and pass them to game objects as required, or define them inline in the constructor.

**Resources Used**

**Sprites**

<https://ansimuz.itch.io/tiny-rpg-forest>

-Player, enemy destruction animation

<https://opengameart.org/content/zelda-like-tilesets-and-sprites>

-Tileset, most props, speech bubbles, dialogue boxes, log monster

<https://opengameart.org/content/battle-background-hazy-hills-0>

-Title screen background

<https://bakudas.itch.io/generic-rpg-pack>

-Slime monsters

<https://opengameart.org/content/pixel-art-minotaur-sprites>

-Minotaur

<https://jesse-m.itch.io/skeleton-pack>

-Skeleton

<http://www.java-gaming.org/topics/ghostr-2d-ghost-hunting-arcade-game/35262/view.html>

-Ghost

<https://opengameart.org/content/gold-treasure-icons>

-Treasure objects

<https://www.deviantart.com/phyromatical/art/Tons-of-Tileset-1-10-Light-jungle-trees-485775828>

-Trees

(not sure if open source)

<https://imgur.com/qYFvsmq?r>

-Jungle bushes

(not sure if open source; haven’t been able to find a better source)

<https://opengameart.org/content/screen-32x26-and-tablet-32x32>

-Computer screen

<https://opengameart.org/content/explosion-animations>

-Boss Explosion

**Sound Effects**

(I manipulated some of these in Audacity before using them in game)

<https://opengameart.org/content/battle-sound-effects>

-Sword swish

<https://opengameart.org/content/fantasy-sound-effects-library>

-Fanfare

<https://opengameart.org/content/rpg-sound-pack>

-monster splat

<https://opengameart.org/content/level-up-power-up-coin-get-13-sounds>

-Dialogue open

<https://opengameart.org/content/sound-effects-pack-2>

-Coin collect

<https://opengameart.org/content/512-sound-effects-8-bit-style>

-Bush destruction, pot destruction, heart collect, player hit, explosion, minotaur powerup

**Music**

<https://opengameart.org/content/path-to-lake-land>

-Title and intro screens

<https://opengameart.org/content/5-chiptunes-action>

-Boss fight

<https://tallbeard.itch.io/music-loop-bundle>

-Forest, Defeat, Cave, Home, Victory

**Code**

My Sprite class was initially based off R. B. Whitaker’s AnimatedSprite class, found at <http://rbwhitaker.wikidot.com/texture-atlases-2>. It has been significantly modified from this original.