Cave Game

AQA Computer Science NEA

Freddy Heppell

Table of Contents

1 Analysis

1.1 Background

1.1.1 Text-Based Games

A text-based game is a game that uses text characters instead of 2D or 3D graphics. Many early games used text graphics because they are much less demanding to render. Text-based games typically use a fixed-width environment (i.e. the terminal or command line on modern operating systems), controlled by keypresses or key commands (opposed to mouse or controller input).

1.1.2 Procedural Generation

Procedural generation is the process of generating data using an algorithm rather than pre-creating the data. Rogue (1980) was the first game to use procedural generation. There are two primary approaches to procedural generation:

Random Selection Decisions (for example which enemy should appear in a room) are made based on random number generation.

Algorithmic Generation Noise algorithms (e.g. Perlin, Simplex) generate random data which is interpreted to build the world's terrain. Used in games such as Minecraft to generate the world.

Many games use a combination of both. For example, Minecraft uses algorithmic generation (using the Perlin noise algorithm) to create the terrain, but random selection to place enemies.

1.1.3 Cellular Automata

A Cellular Automaton is a model for the simulation of complex problems. Each cell abides by certain rules, typically based on the number of neighbouring cells. The most common form of 'neighbourhood' is the Moore Neighbourhood (shown right) which includes the 8 cells (red) surrounding the home cell (blue). The home cell itself is technically part of the neighbourhood but is usually ignored.
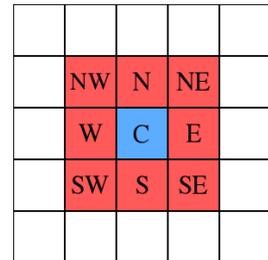
The most famous use of Cellular Automata is Conway's Game of Life. This game uses the ruleset:

- A live cell with < 2 live neighbours dies
- A live cell with 2 or 3 neighbours remains live
- A live cell with > 3 neighbours dies
- A dead cell with exactly 3 live neighbours becomes a live cell

This ruleset can be described as a "*2,3 ruleset*".

These changes take place simultaneously to all cells, not to each cell in turn. As this is difficult in an algorithm, the algorithm should compute it for each cell in turn but not immediately apply

changes, but rather make the changes to a copy of the grid which replaces the main grid once the iteration is complete.

1.2 Description of Problem

A game that procedurally generates an infinite cave map using cellular automata. The player can navigate the cave, collect rewards and defend themselves against monsters.

1.3 Identification of Third Party

For my third party I have selected **removed** as he enjoys playing similar games. He assisted me with devising requirements for the game.

I enjoy playing games of this genre, however most are very graphically demanding. I would like a game of this genre that uses simpler graphics, so it can run on lower-specification computers such as laptops. I'd like the game to have fairly simple mechanics, so I can play it casually.

1.4 Requirements

1. The user should be able to load an existing world.
   1.1. The player should be able to choose the world from a list of worlds currently stored on disk.
   1.2. Upon loading, the world and character should be identical to when it was saved, including:
      1.2.1. Position of player
      1.2.2. Inventory of player
      1.2.3. World and world state
      1.2.4. Enemies
      1.2.5. Chest status (opened or unopened)
2. The user should be able to start a new game
   2.1. They should be asked to input a seed
      2.1.1. A seed may consist of any character
      2.1.2. Two worlds created with the same seed should be identical
3. The world should consist of a 2-dimensional, top-down grid
   3.1. Floor cells and rock cells should be clearly differentiated
   3.2. Treasure and monsters should be clearly differentiated from the environment
   3.3. The game should generate new grids when the player leaves the current grid, so the world generates infinitely.
   3.4. Each of the grids should be connected to their vertical and horizontal neighbours
   3.5. The grid should be procedurally generated based on the world seed
   3.6. Only an area around the player should be shown, this should be set by the user's screen size
4. Treasure and enemies should be placed at random on the grid
   4.1. They may only be placed on floor cells

5. The player should be able to move through the grid using the WASD keys
   5.1. The game should take this input without requiring the ENTER key to be pressed
6. The player should be able to collect treasure
   6.1. They will either receive a random amount of gold, or
   6.2. They may receive a random item reward
7. The player should be able to attack monsters
   7.1. Each monster has should have a sight radius
   7.2. If the player enters the sight radius and is not obscured by an obstacle, the monster should attack the player. If they are obscured by an obstacle, the monster should not attack.
   7.3. The game should simulate a fight between the player and the monster
       7.3.1. The player and monster should take turns attacking each other
       7.3.2. If the player has armour equipped, it should reduce the amount of damage the monster does to the player
           7.3.2.1. When an armour point is used, it will regenerate in 2 turns time
       7.3.3. If the player wins, the monster should die and the player should receive a random reward
       7.3.4. For each turn, there is a random chance of 5% the attacker will stun the defender
       7.3.5. If the defender is stunned, the attacker can attack again and the defender regenerates no armour
   7.4. The rewards should be generated in the same way as chests
   7.5. If the player dies, they should return to a nearby point and lose an item
8. The player should be able to see their inventory
   8.1. The player should be able to check which weapon and armour is equipped
   8.2. The player should be able to select which weapon they wish to equip and which armour they wish to equip
9. The player should be able to close the game
10. The game should automatically save periodically
    10.1. The save should include data defined in 1.2
    10.2. The save data should be placed in an appropriate location for each operating system

2 Documented Design

2.1 Overall System Design
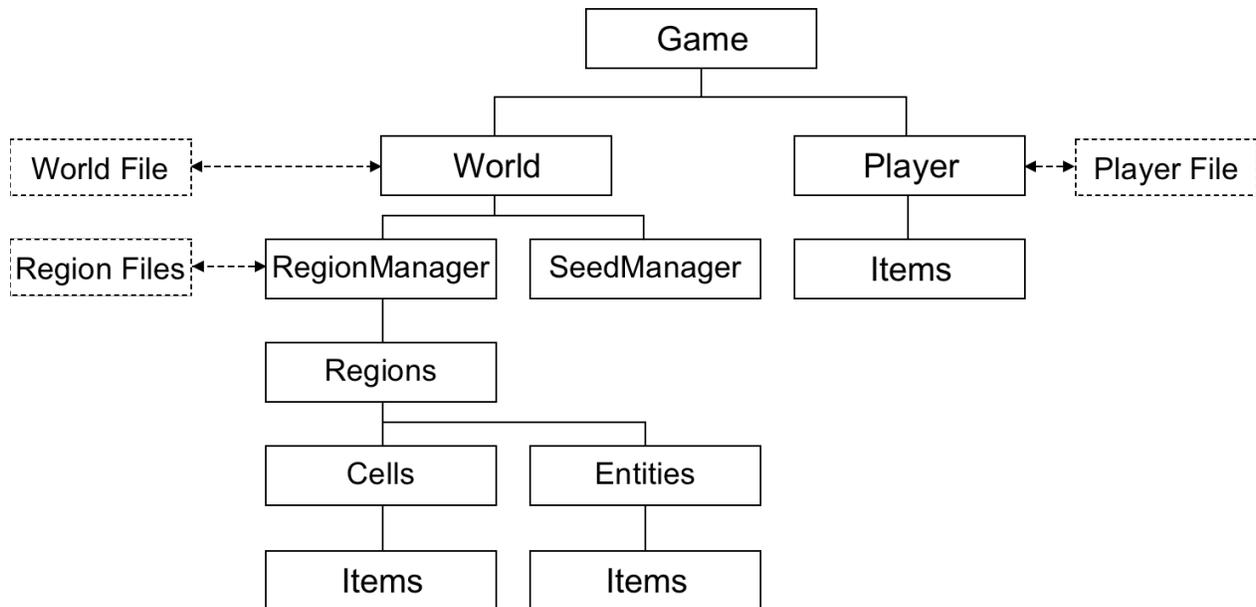
The hierarchy of classes used to store the world file.

The root class of the project (which contains the `main()` method executed by the JVM) is the `CaveGame` class. This creates an instance of `Game`, which is stored statically so it can therefore be accessed by any class within the hierarchy. For example, when a chest cell is opened it accesses the Game instance and instructs the region it is in to resave. This is also used in the Entity system so that entities can instruct regions which cells cause the player to enter combat.

The Game class stores an instance of the active `World` and `Player`. The world stores an instance of `RegionManager` and `SeedManager`. The region manager is responsible for loading regions from disk and caching them in memory. The active regions are stored in a linked hashmap (a dictionary that maintains order), which automatically deletes the eldest entry when the maximum size is reached. The SeedManager generates the per-region seeds used to ensure that any random generation is consistent.

The Region stores the cells in a 2D-array and the entities within the region. Both of these may contain items. Cells are given a reference to the Region they are in, which most cell types will discard (some, for example Chest cells will keep this), to instruct the SaveManager to update the save file. An example of when this is used is when the player claims a reward from a chest. Once

7

the reward has been claimed it cannot be claimed again so the save file should be updated to reflect this.

The item system is implemented using the Singleton design pattern, which means that one instance of the class is automatically instantiated when the class is first accessed, and all future usages of the class use this first instance. This means that the instance of the class can be used anywhere in the codebase without needing to pass the instance into every class that requires it. Whilst this can be problematic in some cases (for example, it is easy to accidentally access the class prior to setup methods being run), for a class of this type it significantly simplifies code.

2.2 Coordinate Systems

The game uses several coordinate systems. The primary coordinate system is world coordinates (x_w, y_w). These are continuous across the entire world. Region coordinates (x_r, y_r) represent the regions that divide the map. Cell coordinates (x_c, y_c) denote the coordinates within that region relative to the bottom left. For world and region coordinates, all integer values are valid. For cell coordinates, only integer coordinates within the size of a region are valid. Given a world coordinate, it is possible to calculate the other coordinate pairs, where S is the region size.

$$r = \left\lfloor \frac{w}{S} \right\rfloor$$

$$c = w - S \left\lfloor \frac{w}{S} \right\rfloor$$

In Java, these operations can be performed using the `floorDiv` and `floorMod` methods from the `Math` library.

The world coordinate can be found from the region coordinate and cell coordinate by calculating the coordinate of the bottom-leftmost cell of the region (from which the cell coordinates are numbered) and adding the cell coordinate.

2.3 Data Structures

2.3.1 Game Class

Property	Type	Explanation
world	World	The game's World instance.
player	Player	The game's Player instance.
seedManager	SeedManager	The game's instance of the SeedManager. Used to generate seeds for regions.
gameName	String	The user's chosen name for the world
logger	Logger (Final)	Log4j's logger class

2.3.2 World Class

Property	Type	Explanation
regionManager	RegionManager	The region manager is responsible for loading regions from disk and caching them.
logger	Logger (Final)	Log4j's logger class

2.3.3 Region Class

Property	Type	Explanation
cells	2D array of Cell	Stores the cells within the region. Note that this array stores the child types of Cell rather than Cell directly. The first index is the x value, the second is the y value.
random	Random	The instance of Java's Random class used to make random decisions for this region. It is seeded with the region's generated seed.
entityCoordinates*	Array WorldCoordinate	An array of the coordinates of the entities within the region
entities*	Array Entity	An array of the entities within the region

* These pairs of arrays are used like a dictionary, so the nth entry in one corresponds to the nth entry in the other. This is to bypass a limitation with Gson serialisation and the JSON format. Dictionaries do exist in JSON, but they must have a key of type String. As the keys in this case are objects, it is not possible to serialise this directly. One option would be to convert between a dictionary and two arrays as the region is serialised/deserialised, however this could be a resource intensive process that would have to be performed every time the region is loaded or saved. It is far more efficient to store them as two separate arrays.

2.3.4 Cell Class

Cell is an abstract class. This means that it cannot be directly instantiated, and most of methods (tagged with abstract) *must* be overridden.

Property	Type	Explanation
listener	WorldCoordinate	The location of the entity that needs to be informed if a player enters this cell

The ChestCell class, which extends Cell, contains the following properties:

Property	Type	Explanation
reward	Array Item	An array of the items to be given to the player when the chest is opened.
claimed	Boolean	If the chest has been opened or not

2.3.5 Entity Class

Property	Type	Explanation
visibleCells	Array WorldCoordinate	A list of the cells visible to the entity
health	Float	The player's current health
alive	Boolean	If the player is alive and has not been resurrected
armour	Integer[1]	The amount of armour the player has
armourChange	Queue[2]	How much armour must be restored on the next turn. In the constructor, 0 and 0 is pushed to the queue so the player receives no armour back on the 1st and 2nd turns. When the armour lost in the 1st turn is added to the queue it will be restored in the 3rd turn.

2.3.6 Player Class

Property	Type	Explanation
inventory	Array Item	The player's current inventory
iEquippedWeapon	Integer	The index of the player's currently equipped weapon
iEquippedArmour	Integer	The index of the player's currently equipped weapon
location	WorldCoordinate	The current location of the player
moveCounter	Integer	The number of moves the player has made since the last save. Used to determine when the game should next autosave.
logger	Logger (Final)	Log4j's logger class

2.3.7 CombatManager Class

Property	Type	Explanation
player	Player	The player involved in the combat
enemy	Entity	The entity involved in the combat
overallMultiplier	Float	The precomputed value of *em*
Turn	Enum	An enum of who's turn it is, with options: PLAYER, ENEMY
random	Random	The instance of random used to make random decisions
extraTurn	Boolean	If the current turn is an extra turn

2.3.8 SaveManager Class

Property	Type	Explanation

---

[1] Note that this is the `Integer` class opposed to the primitive `int` because this cannot handle popping potentially null values from a queue.
[2] In Java, `Queue` is an interface. This property uses the `LinkedList` implementation of Queue

CELL_ADAPTER_FACTORY	RuntimeTypeAdapterFactory (Final)	The adapter factory the parser uses to serialise and deserialise the subclasses of Cell
ITEM_ADAPTER_FACTORY	RuntimeTypeAdapterFactory (Final)	The adapter factory the parser uses to serialise and deserialise the subclasses of Item
ENTITY_ADAPTER_FACTORY	RuntimeTypeAdapterFactory (Final)	The adapter factory the parser uses to serialise and deserialise the subclasses of Entity
PLAYER_FILE_NAME	String (Final)	The name of the player data file. Set to "player.json"
WORLD_FILE_NAME	String (Final)	The name of the world data file. Set to "world.json"
logger	Logger (Final)	Log4j's logger class

2.3.9 RegionManager Class

Property	Type	Explanation
regionLookupCache	HashMap[3] RegionCoordinate → Boolean	A common operation is to look up whether a file exists on disk. This property stores cached values as to whether a region exists on disk.
regionCache	LinkedHashMap[4] RegionCoordinate → Region	Stores a cache of the most recently accessed regions. This reads the size from the config file and removes the eldest entry if the number of entries exceed this size.
saveDir	File	The save directory for this game.
seedManager	SeedManager	The game's seed manager instance. Used to create new regions with the correct seed
logger	Logger (Final)	Log4j's logger class

2.3.10 WorldCoordinate

WorldCoordinate is a class designed to be used as a custom datatype. In Java, for a class to behave like a data type it must implement certain methods (specifically `equals()` and `hashCode()`).

Property	Type	Explanation

---

[3] A HashMap is Java's dictionary implementation.
[4] A LinkedHashMap is a HashMap that preserves the order the values were added to it.

wx	Integer	The world x and y coordinates
wy	Integer	
rx	Integer	The x and y coordinates of the region
ry	Integer	
cx	Integer	The x and y coordinates within the region
cy	Integer	

2.3.11   Vector2 Class

Vector2 is a class designed to be used as a custom datatype. In Java, for a class to behave like a data type it must implement certain methods (specifically `equals()` and `hashCode()`). It represents a change in a coordinate.

Property	Type	Explanation
dx	Integer	Difference of x
dy	Integer	Difference of y

2.3.12   ItemRegistry Class

ItemRegistry uses the Singleton pattern so that it can be accessed from the parts of the codebase that need to.

Property	Type	Explanation
itemRegistry	ItemRegistry	The single instance of the Singleton class. This is set when the first method is called on this class and all subsequent method calls go to this instance.
Items	Array Item	A list of all the item types registered
totalWeight	Double	The sum of the weight of every item
logger	Logger (Final)	Log4j's logger class

2.4 Algorithms

2.4.1 Cave Generation Algorithm

This algorithm was defined in the 2010 paper *Cellular automata for real-time generation of infinite cave levels* [1]. This paper describes how the algorithm operates but does not provide specific implementation details such as code or pseudocode.

A grid of $d \times d$ cells is generated as the Base Grid, with four other grids to the north, south, east and west. Each cell can have status rock or floor.

The base grid is initialised with random rock cell, with each cell having probability r of becoming a rock cell. This random selection is seeded so that it can be reproduced given the same seed. If a cell has a Moore Neighbourhood value of $\geq T$, it is set as rock. Otherwise it becomes floor. This process is reproduced for a iterations. For each of the neighbouring grids the same procedure is performed. To ensure it is possible to walk between the grids, the closest two floor cells on the neighbouring grids are selected and a straight path of floor is dug between them. To smooth it the iteration is run again b times. Once the grids have been generated, rock cells that are directly adjacent to floor cells are labelled as wall cells. Each time a further base grid is generated and connected to the existing grid, the two adjacent cells are smoothed for b iterations. The paper uses $d = 50, r = 0.5, T = 5, a = 4, b = 2$, but these can be varied to change the appearance of the generated caves.

During prototyping, it was found that as the parameters used in the program were different to the paper, it was not necessary to check for connections as they appeared naturally in all cases.

2.4.2 Area-of-Effect Algorithm

Each entity has a circular area-of-effect around it, which must be registered to the EntityController. To calculate this circle, a simple algorithm is performed. As the general equation of a circle is $x^2 + y^2 = r^2$, if $x^2 + y^2 \leq r^2$, the point must be inside the circle or on the edge. As the radius increases this method becomes less efficient, however with the radius size used in the program (~5) this method is more efficient than specific algorithms.

Prototyping this algorithm found that the circles had an unsuitable shape. This is because some parts of the circle are outside of the range when shown on a pixel grid, when they should be inside the circle. Instead of using r^2 as the right-hand-side of the inequality, the algorithm increases this by using $1.8 \times r^2$

```
FOR y in (-r+ys) … (r+ys)5 DO
     FOR x in (-r+xs) … (r+xs) DO
           IF x^2 + y^2 <= r^2 THEN
                 ADD (x,y)
           ENDIF
     ENDFOR
ENDFOR
```

Algorithm to find circle radius r, center (xs, ys)

---

[5] Both bounds are inclusive

2.4.3 DDA & Bresenham's Algorithm
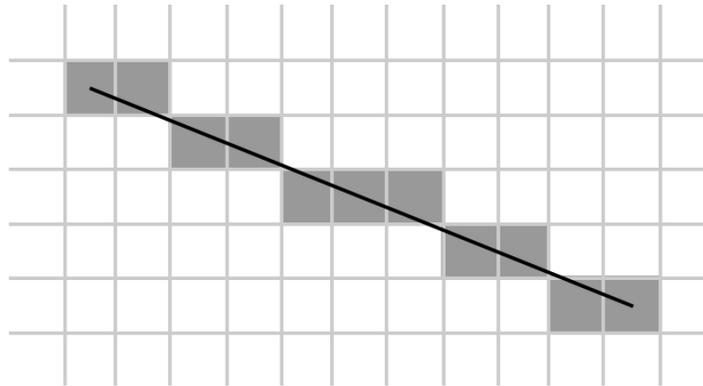
These two algorithms rasterise a line (shown in black above) into a set of pixels (shown in grey above).

The simplest method of rasterising lines is with the DDA algorithm

Let $dx = x_1 - x_0$, $dy = y_1 - y_0$

If $|dx| > |dy|$, $S = |dx|$ otherwise $S = |dy|$, where S is the number of steps to be taken. This means that if the line is moving horizontally or vertically, S is the horizontal or vertical difference respectively.

The increments can be calculated with:

$$I_x = \frac{dx}{S}$$

$$I_y = \frac{dy}{S}$$

Then, iterating from $i = 0 \rightarrow S$, add I_x and I_y to the coordinates starting from pair 0.

However, this algorithm is inefficient because it uses float calculations. A better method, Bresenham's algorithm, was developed which only uses integer calculations.

```
IF |y1 - y0 | > |x1 - x0 | THEN
      Swap the x and y coordinates within each pair
ENDIF
IF |x0 > x1| THEN
      Swap the coordinate pairs
ENDIF

E ← |y1 - y0|

IF y0 > y1 THEN
      S ← -1
ELSE
```

```
      S ← 1
ENDIF

dx ← x1 – x0
e ← dx >> 1 # This is a bitwise operator for dividing by 2

FOR x in x0 … x1 DO
      IF |y1 – y0| > |x1 – x0|
            PLOT (y,x)
      ELSE
            PLOT (x,y)
      ENDIF

      e ← e - E

      IF e < 0 THEN
            y ← y + S
            e ← e + dx
      ENDIF
ENDFOR
```
*Find a line between (x0, y0) and (x1, y1) using Bresenham's Algorithm* [2]

The typical version of Bresenham's algorithm works only where the gradient is between 0 and 1 (i.e. travelling rightwards more than upwards), and coordinate pair 0 is the leftmost pair (i.e. travelling from left to right). The first two IF statements normalise the line to meet these requirements and the IF statement within the FOR loop undoes this normalisation when the line is plotted.

This algorithm can be used to find the shortest grid path between two cells which can be interpreted as line-of-sight. In the game, when the player enters the detection radius of an enemy the game draws a cell line from the enemy to the player and checks if any of the cells in this line are marked as impassable. However, there are some cases in which this algorithm returns the incorrect result. The algorithm can find a line of sight through diagonal walls even if it is blocked (see below). To avoid this, checks should be performed to ensure that the line does not pass through diagonal 'gaps'.
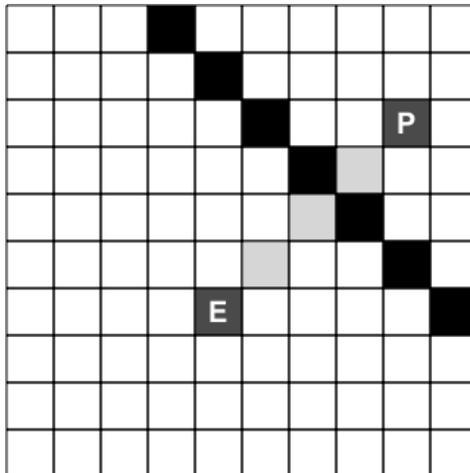
2.4.4 Player Spawning Algorithm

The player should be spawned at (0,0) when they first enter the map, however this may not be possible if the cell (0,0) is rock. Each child class of Cell defines an `isSpawnAllowed()` method which returns a Boolean value indicating if the player can be spawned on that cell.

17	16	15	14	13
18	5	4	3	12
19	6	1	2	11
20	7	8	9	10
21	22	23	24	25

To find the cell to spawn the player on, the game checks cells using a clockwise spiral pattern. This algorithm is normally used to construct an Ulam Spiral [3], this implementation removes the primality check.

```
a ← CEIL((ROOT(n)-1)/2)
t ← 2a + 1
c ← t^2
t ← t -1

IF n >= (c - t) THEN
     RETURN ((a – (c-n)), (-a))
ELSE
     c ← c – t
ENDIF

IF n >= (c - t) THEN
     RETURN ((-a), (-a + (c - n)))
```

```
ELSE
     c ← c - t
ENDIF

IF n >= (c - t) THEN
     RETURN ((-a + (c-n)), a)
ELSE
     RETURN (a, (a-(c-n-t)))
ENDIF
```

Finding the nth cell of the spiral

This algorithm is modified to spawn entities, but the *x* and *y* coordinates are added to the center of each region.

2.4.5 Damage Calculation Formula

The player can equip armour to give them protection against damage. Each point of armour reduces one point of damage by 30% ($= e$). Once an armour point has been used it is consumed for the next turn.

Let $d =$ the raw amount of damage done

$D =$ the actual amount of damage done

$a =$ the armour points available

$m =$ the damage multiplier

$e =$ armour blocking amount

$$D = d - em \cdot \min(d, a)$$

To minimum function is used to calculate the amount of armour used:

- If the amount of armour is greater than the raw damage, only armour up to the damage amount will be used
- If the raw damage is greater than the amount of armour, only the amount of armour that the player has can be used

em is a constant value throughout the game, so it can be pre-calculated.

After each turn, a random decimal value is computed. If this value is less than 0.1, the attacker of that turn can have another turn. The defender will not regenerate any armour.

Damage Against Player

		Attack Damage				
		Player				Monster
		Wood	Iron	Iron	Infused	Regular
		3	7	10	14	7
Shielding	**None** 0	3	7	10	14	7
	Leather 5	2.1	5.5	8.5	12.5	5.5
	Chainmail 8	2.1	4.9	7.6	11.6	4.9
	Iron 10	2.1	4.9	7	11	4.9
	Infused 12	2.1	4.9	7	10.4	4.9

Damage values for combinations of weapon and armour, assuming the full value of armour is available

2.5 Tools & Libraries

Several external libraries are used for the project. Most libraries are included using Gradle, except for those specified. These are the top-level libraries used for the project, they may themselves include further libraries which are not directly used by the project.

Gradle

The project is built using the Gradle build system. This allows packages to be easily included from Java's Maven repository and creates automatic builds of the project including all libraries.

gson

Gson is a JSON parsing library created by Google. The library can serialise objects into JSON strings automatically, and then convert JSON strings back into objects. The library is used to save and load game data.

RawConsoleInput

RawConsoleInput is a single-file library to read keypresses from Windows and UNIX-like (e.g. macOS, Linux) consoles. This library means that user inputs can be processed as soon as the key is pressed, without the ENTER key needing to be used. This is a single-file library and not available from Maven, so it is included in the codebase under the original biz.source_code.utils namespace.

JNA

JNA is a library to use native shared libraries entirely in Java. It is used by the RawConsoleInput library.

Log4j

Log4j is a logging package for Java. It allows the program to write various severities of logs to disk, and automatically moves old logs into dated files.

JLine3

JLine is a library to interact with the command line. This is used to get the dimensions of the command line. Usage of this library was discontinued due to an issue discussed in the Testing section.

Cloning

This library is used to clone Item objects when they are added to the player's inventory. This library includes the objenesis library.

2.6 Data Storage

2.6.1 Save Location

The game stores data in system folders typical for each operating system. On windows, this is `C:\Users\Username\Appdata\Roaming\CaveGame`. On mac OS `~/Library/Application Support/CaveGame` and `~/.CaveGame` on Linux. If the program is unable to determine the OS it will create a `CaveGame` subfolder in the current directory. This is the root folder in which the game stores all data including world folders and the preferences override file.

2.6.2 world.json

The world.json file contains data associated with the entire world. It contains the following data:

Key	Example	Explanation
worldName	"Testing World"	The world's name as a string. This cannot be found from the name of the world folder as not all characters are permitted in directory names.
configurationHash	XwWDK…	The hash of the merged configuration file at the time of the world's creation. Certain configuration parameters (e.g. region size) change how regions are loaded. If they are changed the program may not work properly. This hash is base64 encoded.
worldSeed	"abc123"	The world's global seed as a string

This file is *not* generated with Gson's automatic serialisation, as none of the properties of this class need to be saved directly. The JSON is generated with the `WorldSerialiser` class from the World object, and deserialised into a String → String HashMap.

2.6.3 rx.ry.json

The files for each region are serialised and deserialised by Gson. The primary advantage of this library is that it can automatically recursively serialise an object and the object's properties, and

then deserialise back into an instance of the same object completely automatically. For parts of the game where polymorphism is used (for example types of cells), Gson is unable to determine which child class to use as the property types only specify the parent class. For this, Gson provides RuntimeTypeAdapters, where each child-type is assigned a value which is then used to determine which child class should be used.

```
{
    "regionCoordinate":{"rx":0,"ry":0},
    "cells":[
        [
            {"t":"f"},
            {"t":"f"},
            {"t":"f"},
            ...
            {
                "t":"c",
                "reward":[
                    {
                        "t":"SwordItem",
                        "tier":"Steel",
                        "damage":8
                    },
                    {
                        "t":"ArmourItem",
                        "tier":"Leather",
                        "shielding":5
                    }
                ],
                "claimed":false
            },
            ...
    "entityCoordinates":[
        {"wx":22,"wy":22,"rx":0,"ry":0,"cx":22,"cy":22}
    ],
    "entities":[
        {"t":"m","health":30.0,"armour":0,"armourChange":[0,0]}
    ]
}
```

Example of a region file

2.6.4 Configuration File

The game stores a number of key values in configuration files. The default configuration file is bundled within the game's .jar file, however this can be overridden by adding the properties to the

20

`cavegame.properties` file created in the root folder. The .jar file also contains a default template for this override file which will be placed in the correct location if it does not exist.

Each file is a .properties file using Hungarian Notation for the keys, with each prefix corresponding to a Java datatype (e.g. i for integer, s for string).

```
iRegionSize=50
iRegionCacheSize=8
iRegionIterationCount=2
iCellDeathThreshold=5
fRandomBoundary=0.5
fChestSpawnBoundary=0.995
bShouldClearScreen=true
fFrameSleepTime=0.75
iChestMaxItems=2
iPlayerSaveFrequency=5
fArmourProtection=0.3
fDamageMultiplier=1.2
fCombatStunProbability=0.1
bShowDebugInfo=false
lTurnSleepTime=500
```
The default configuration file

2.6.5 Log Files

The log4j file stores output logs in a logs directory in the location that the executable is run. The active logfile is named cavegame.log, with older log files being renamed with the date they were finished.

A minimum log level to store is configured, for development this was set to TRACE, the lowest level so that everything would be logged. For an actual release this should be set to INFO.

Each log entry contains the following information:

- Severity level (DEBUG, INFO, WARN, FATAL etc.)
- Timestamp
- Active thread
- Class that triggered the log
- Log message

```
[DEBUG] 2018-01-01 09:00:00.000 [main] Region - Getting entity at
W:(102,138), R: (2, 2), C: (2, 38)
```
Example of a log entry

3 Technical Solution

The source code of the solution is listed below in alphabetical order of file name. The file *CaveGame.java* is the main class.

The key algorithms are in the following files:

2.4.1 Cave Generation Algorithm: `iteration()` method in `world/Region.java`.

2.4.2 Area-of-Effect Algorithm: `calculateVisibleCells()` method in `entities/Entity.java`.

2.4.3 Bresenham's Algorithm: `hasLineOfSight()` method in `world/World.java`.

2.4.4 Player Spawning Algorithm: `getSpiralCoordinate()` method in `world/coord/CoordinateProperties.java`.

2.4.5 Damage Calculation Formula: `calculateDamage()` method on `entities/Entity.java`.

Code listings removed
See:

https://github.com/freddyheppell/cavegame

4 Testing

Spawning Monsters

Test Of: The generateEntities() method of Region

Expected Result: Once monster is placed close to the centre of the Region in a valid cell

Actual Result: **Failed**, The game enters an infinite loop of creating the region the monster is in

Debugging Process: To find the source of the issue, I raised an exception within the method to see why it was being recursively called. I discovered that it was caused by the location-finding algorithm. When this algorithm tested if a cell was valid, it called the getRegion() method. This will create a region if it does not already exist. At the point that the generateEntities() method was being run, the region had not been saved, therefore the getRegion() method attempted to create it again, which itself called the generateEntities() method. This process repeated until the call stack overflowed. To resolve this, the entities are generated after the region has been saved, then the region is resaved.

```
[INFO ] 2018-02-21 17:22:22.372 [main] SaveManager - Loading region from disk R: (1, -1)
[INFO ] 2018-02-21 17:22:22.486 [main] SaveManager - Loading region from disk R: (1, -1)
[INFO ] 2018-02-21 17:22:22.602 [main] SaveManager - Loading region from disk R: (1, -1)
[INFO ] 2018-02-21 17:22:22.824 [main] SaveManager - Loading region from disk R: (1, -1)
[INFO ] 2018-02-21 17:22:22.824 [main] SaveManager - Loading region from disk R: (1, -1)
[INFO ] 2018-02-21 17:22:22.930 [main] SaveManager - Loading region from disk R: (1, -1)
[INFO ] 2018-02-21 17:22:23.022 [main] SaveManager - Loading region from disk R: (1, -1)
```

The log files showing that the region is repeatedly loaded

Revised Result: The entity is correctly generated
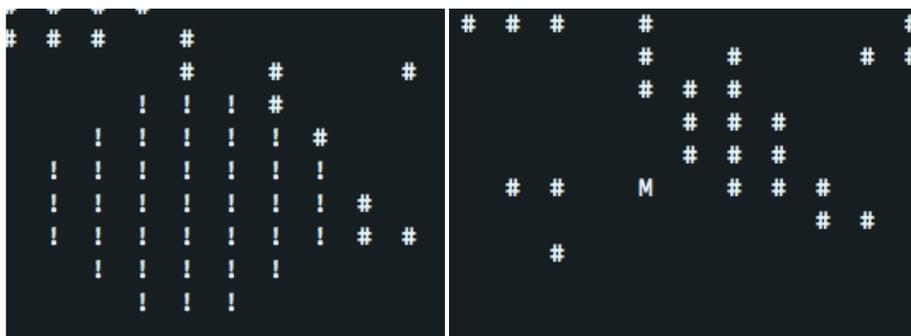
Calculating Monster Area-of-effect

*Test Of*: The system to detect if a player is within range of a monster

Expected Result: The monster is shown with a circle (radius 3) of exclamation marks surrounding it (added temporarily to indicate the area)

Actual Result: **Failed,** only the area-of-effect is shown until the game is restarted, then only the monster is shown.

Debugging Process: I began by inspecting the save file to see what was being saved. The location of the monster was being saved correctly in the region, but the fact that the cells were triggers were not saved. By first saving the region immediately after the entity is added and then again to each region

Revised Result: The game correctly shows both the area of effect and the entity
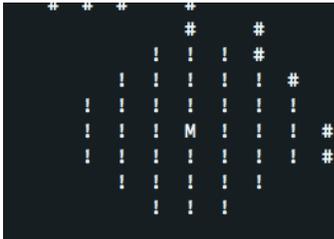
Checking Line of Sight

Test Of: world.HasLineOfSight()

Expected Result: If the line between the monster and player is not interrupted, the function will return true.

Actual Result: The function inconsistently returns true or false if the line of sight is interrupted

Debugging Process: I began by logging what cell caused the line-of-sight to be broken, and specifically what type of cell that was. I found that the logs reported the line of sight was always blocked by a rock cell at the same location as the entity. I discovered that the checking process to see if an entity was valid was incorrect, it would only spawn entities on cells that were *not* valid for spawning.

Revised Result: The game correctly finds the line of sight when not interrupted, and does not find line of sight when it is interrupted.

Typing Delay issues in Windows

During testing it was noted that on Windows, there was a delay of 2-3 seconds where it was not possible to enter text at prompts (e.g. the movement prompt). Using Git, I was able to perform regression tests to discover when this issue was introduced. I found that the issue was introduced in the commit where I began autodetecting the height and width of the user's terminal. By systematically undoing the changes made in this commit, I was able to determine that this issue was caused by the JLine library. Binding an instance of JLine's Terminal class to the terminal caused a permanent delay on all input. I removed the library and implemented a manual method to set the width and height.

4.1 Requirement Testing

The requirements are reproduced below. The requirements that are being specifically tested are in **bold**.

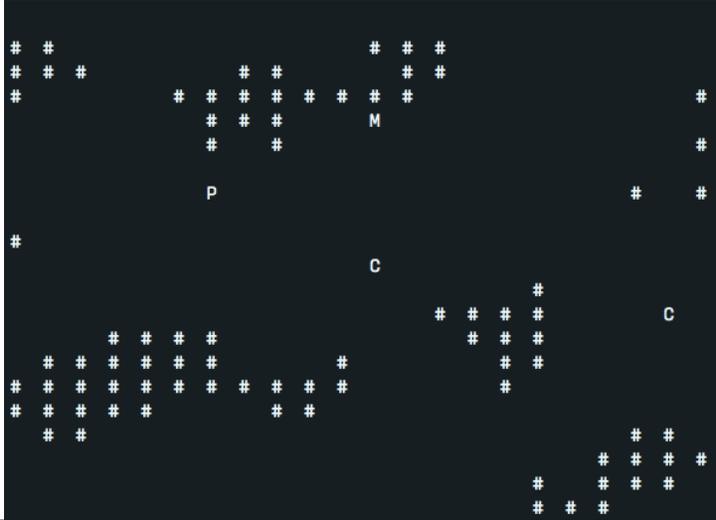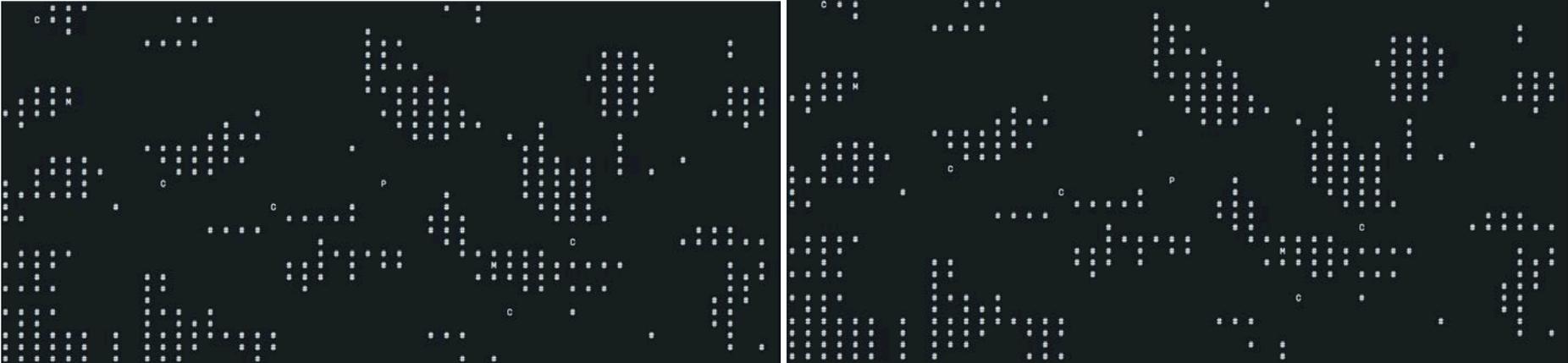| Inputs	Expected Result	Actual Result	Notes
1. The user should be able to load an existing world. **1.1. The player should be able to choose the world from a list of worlds currently stored on disk.**			
Enter 1 on main menu	The user is shown a list of worlds that they select from. Selecting a world loads it	As expected	
	Select a world to load 1. test 2. anothertest 3. andanothertest Enter your choice:		
1.2. Upon loading, the world and character should be identical to when it was saved, including: **1.2.1. Position of player** **1.2.2. Inventory of player** **1.2.3. World and world state** **1.2.4. Enemies** **1.2.5. Chest status (opened or unopened)**			
Create a new world, collect a nearby chest, fight a nearby enemy. Close the world and reopen.	The world state is identical	As expected	Due to the way saving is implemented, the position may not be exact. This is because the player's location is saved every 5 moves. In this case I intentionally made the necessary moves to reach this threshold.
Before reloading		*After reloading*	

Inputs	Expected Result	Actual Result	Notes
| ```
Your Inventory
1: Leather Armour (5 Prot)
2: Wooden Sword (1 Dmg)
3: Wooden Sword (1 Dmg)
4: Wooden Sword (1 Dmg)
5: Leather Armour (5 Prot)
6: Iron Armour (10 Prot)
7: Gold (10 G)
8: Wooden Sword (1 Dmg)
9: Infused Armour (15 Prot) (Equipped)
10: Iron Sword (5 Dmg) (Equipped)
11: Chainmail Armour (8 Prot)
12: Wooden Sword (1 Dmg)

x Exit, e<num> to equip
> |
``` | | ```
Your Inventory
1: Leather Armour (5 Prot)
2: Wooden Sword (1 Dmg)
3: Wooden Sword (1 Dmg)
4: Wooden Sword (1 Dmg)
5: Leather Armour (5 Prot)
6: Iron Armour (10 Prot)
7: Gold (10 G)
8: Wooden Sword (1 Dmg)
9: Infused Armour (15 Prot) (Equipped)
10: Iron Sword (5 Dmg) (Equipped)
11: Chainmail Armour (8 Prot)
12: Wooden Sword (1 Dmg)

x Exit, e<num> to equip
> |
``` | |

2. The user should be able to start a new game
    2.1. They should be asked to input a seed
        2.1.1. A seed may consist of any character
        2.1.2. Two worlds created with the same seed should be identical

Inputs	Expected Result	Actual Result	Notes
Select 2 on the main menu. Enter the world name "test", with seed "123". Close the world. Create a world with name "test2" and seed "123"	The worlds should be identical	As expected	

Inputs	Expected Result	Actual Result	Notes

3. The world should consist of a 2-dimensional, top-down grid
   3.1. Floor cells and rock cells should be clearly differentiated
   3.2. Treasure and monsters should be clearly differentiated from the environment

Generate a world	Floor, walls, chests and enemies should be displayed as different characters.	As expected	"#" shows walls, floors are shown by empty space, "C" shows characters and "M" shows monsters

Inputs	Expected Result	Actual Result	Notes
4. **Treasure and enemies should be placed at random on the grid**			
4.1. **They may only be placed on floor cells**			
Generate a world	Monsters and chests will have been placed randomly	As expected	
5. **The player should be able to move through the grid using the WASD keys**			
5.1. **The game should take this input without requiring the ENTER key to be pressed**			
Create a new world and enter W, A, S and D	The player should move up, down, left and right	As expected	Screenshots show result of pressing D

Inputs	Expected Result	Actual Result	Notes

6. **The player should be able to collect treasure**
    6.1. **They will either receive a random amount of gold, or**
    6.2. **They may receive a random item reward**			

Walk onto a chest cell	The player's inventory gains gold or an item reward	The player gained Iron Armour	In the screenshots below, the player gains leather armour from a chest.

`Your Inventory`
`1: Leather Armour (5 Prot) (Equipped)`
`2: Wooden Sword (1 Dmg) (Equipped)`
 	
`x Exit, e<num> to equip`
`> |`
	`Your Inventory`
	`1: Leather Armour (5 Prot) (Equipped)`
	`2: Wooden Sword (1 Dmg) (Equipped)`
	`3: Leather Armour (5 Prot)`
	
	`x Exit, e<num> to equip`
	`> |`		

7. **The player should be able to attack monsters**
    7.1. **Each monster has should have a sight radius**
    7.2. **If the player enters the sight radius and is not obscured by an obstacle, the monster should attack the player. If they are obscured by an obstacle, the monster should not attack.**			

Enter radius 3 of a monster whilst not in line of sight. *(a)* Enter radius 3 of a monster whilst in line of sight. *(b)*	An obstructed view should not trigger combat. An unobstructed view should trigger combat.	As expected	For screenshot *(b)*, moving up one cell triggers combat.
(a)	*(b)*		

7.3. **The game should simulate a fight between the player and the monster**
    7.3.1. **The player and monster should take turns attacking each other**
    7.3.2. **If the player has armour equipped, it should reduce the amount of damage the monster does to the player**
        7.3.2.1. **When an armour point is used, it will regenerate in 2 turns time**
    7.3.3. **If the player wins, the monster should die and the player should receive a random reward**
    7.3.4. **For each turn, there is a random chance of 5% the attacker will stun the defender**
    7.3.5. **If the defender is stunned, the attacker can attack again and the defender regenerates no armour**			

Inputs	Expected Result	Actual Result	Notes
Create a world, find a monster and enter its sight radius	The game simulates combat in accordance with the requirements	As expected	The full output of combat is reproduced below. The sections that provide evidence of each requirement is indicated.

```
Enemy goes first.
Enemy attacks Player for 1.92 damage, using 3 armour (2 remaining). Health: 48.08

Player attacks Enemy for 5.00 damage (7.3.1)
                          (7.3.2)
Enemy attacks Player for 2.28 damage, using 2 armour (0 remaining). Health: 45.80
You were stunned! The enemy may attack again (7.3.4)

Enemy attacks Player for 3.00 damage Health: 42.80

Player attacks Enemy for 5.00 damage

3 Armour Restored (7.3.2.1)
Enemy attacks Player for 1.92 damage, using 3 armour (0 remaining). Health: 40.88
You were stunned! The enemy may attack again

Enemy attacks Player for 3.00 damage Health: 37.88 (7.3.5)

Player attacks Enemy for 5.00 damage

2 Armour Restored
Enemy attacks Player for 2.28 damage, using 2 armour (0 remaining). Health: 35.60

Player attacks Enemy for 5.00 damage

Enemy attacks Player for 3.00 damage Health: 32.60

Player attacks Enemy for 5.00 damage

You have killed the monster! (7.3.3)
```

8.	The player should be able to see their inventory		
	8.1.	The player should be able to check which weapon and armour is equipped	
	8.2.	**The player should be able to select which weapon they wish to equip and which armour they wish to equip**	

Inputs	Expected Result	Actual Result	Notes
Collect several chests. Enter the equip command "e<number>" to equip armour and a weapon.	The armour and weapon should be marked as equipped in the inventory	As expected	The user has collected several items, the first two are the default items they spawn with.

	<pre>Your Inventory
	1: Leather Armour (5 Prot)
	2: Wooden Sword (1 Dmg)
	3: Gold (21 G)
	4: Chainmail Armour (8 Prot)
	5: Iron Armour (10 Prot) (Equipped)
	6: Leather Armour (5 Prot)
	7: Leather Armour (5 Prot)
	8: Iron Sword (5 Dmg) (Equipped)
	
	x Exit, e<num> to equip
	></pre>		

9. The player should be able to close the game			
Press "x" within a game	The game will close and return the user to their terminal	As expected	If the user is running the game from a provided build (i.e. the windows bundled executable) the window will just close.

	<pre>x
	Quitting game
	~/I/cavegame (master|●1✚1…) $</pre>		

10.   The game should automatically save periodically
    10.1.   The save should include data defined in 1.2
    10.2. The save data should be placed in an appropriate location for each operating system			

Move the player by five cells	Upon reopening the game, the player is in the same location	As expected	

111

Inputs	Expected Result	Actual Result	Notes

4.2 Robustness Testing

World Command Input

- Entering an invalid command (e.g. "q") causes the game to request input again. It does not, however, alert the player that their input is invalid.
- Entering a valid command that is invalid in this context (e.g. attempting to walk into a rock cell) causes the game to request input again

Inventory Command Input

- Entering an unknown command alerts the user that the command is invalid and requests input again
- Attempting to equip a non-existent item (e.g. equipping "5" when only 4 items are in inventory) asks the user for input again
- Attempting to equip a non-equippable item alerts the user
- An entry query crashes the game, which should be fixed in future

Main Menu Input

- Entering an out-of-range number asks the user for input again
- Entering a non-integer asks the user for input again

Load Screen Input

- Entering an out-of-range number, (e.g. loading world 5 when there are only 4 worlds) prompts the user again
- Entering a non-number prompts the user to enter the number again
- Entering the name of an existing world will load it instead

Create Screen Input

- Entering nothing is a valid input, which causes the world information to be placed in the root folder
- It is possible to enter a world which contains invalid characters for an operating system (e.g. "\" on Windows)

5 Evaluation

5.1 Meeting Objectives

Overall, the project meets almost all of the objectives. The only objectives that are not met are due to limitations with the Java language's ability to directly interact with the Operating System.

The solution's requirements are listed below, with comments as to how well the objectives have been met in *italics*.

1. The user should be able to load an existing world.
    1.1. The player should be able to choose the world from a list of worlds currently stored on disk.
    The user is shown a list of valid world folders (a valid world folder contains a world.json file) and they can choose from this list by entering a number. The game currently does not attempt to validate this data (i.e. checking if the JSON files are valid JSON), which would make the solution more robust.
    1.2. Upon loading, the world and character should be identical to when it was saved, including:
        1.2.1. Position of player
        1.2.2. Inventory of player
        1.2.3. World and world state
        1.2.4. Enemies
        1.2.5. Chest status (opened or unopened)

    All of the required data is saved in the region files and the player file.

2. The user should be able to start a new game
    2.1. They should be asked to input a seed
        2.1.1. A seed may consist of any character
        2.1.2. Two worlds created with the same seed should be identical

    All random generation used to generate the world either uses this seed directly, or from data based upon this seed (Regions append their coordinates to the seed). The only part of the application that does not use this data is combat, which uses an automatic seed.

3. The world should consist of a 2-dimensional, top-down grid
    3.1. Floor cells and rock cells should be clearly differentiated
    Floor cells are shown as blank spaces, rock cells are shown as "#"s
    3.2. Treasure and monsters should be clearly differentiated from the environment
    Treasure is shown as "C" on the map, until it is used, then it is not shown. Monsters are shown as "M" until they die, then they are not shown.
    3.3. The game should generate new grids when the player leaves the current grid, so the world generates infinitely.

The player can travel effectively-infinite distances. The only limit to this is Java's integer maximum, 2,147,483,647. If the game attempts to render a cell beyond this point, it will hang indefinitely. However this would take a significant amount of time and require at least 1TB of hard disk space.

3.4. Each of the grids should be connected to their vertical and horizontal neighbours
The regions appear seamless to the player, with no interruptions transitioning between regions.

3.5. The grid should be procedurally generated based on the world seed
The coordinates for each region are appended to the world seed to get the seed for each region

3.6. Only an area around the player should be shown, this should be set by the user's screen size
This feature was in the solution initially, using the JLine library. This works by binding a virtual terminal to the user's terminal, from which the height and width can be read. This solution worked on mac OS and Linux, but caused a significant delay entering inputs on Windows. Due to this, this feature had to be removed. Currently the height and width are hardcoded.

4. Treasure and enemies should be placed at random on the grid
One enemy is placed in each region, at a location closest to the centre as possible. Floor cells have a 0.5% chance of turning into treasure chests.

4.1. They may only be placed on floor cells

5. The player should be able to move through the grid using the WASD keys

5.1. The game should take this input without requiring the ENTER key to be pressed
This requirement has been partially met. The player can control their characters with the WASD keys, but ENTER-less movement is only works on Windows. On mac OS and Linux, this does not appear to be possible without code written in a lower level language.

6. The player should be able to collect treasure

6.1. They will either receive a random amount of gold, or

6.2. They may receive a random item reward
When the player enters a chest cell, they receive a random reward from the whole pool of available items including gold.

7. The player should be able to attack monsters

7.1. Each monster has should have a sight radius
The sight radius is configurable for each type of monster. Currently only one type exists with a radius of 3.

7.2. If the player enters the sight radius and is not obscured by an obstacle, the monster should attack the player. If they are obscured by an obstacle, the monster should not attack.
The game will test for line-of-sight if the player enters the sight radius of a monster. This test works for all unambiguous cases but can produce disputable results in some ambiguous cases, for example line of sight along a diagonal wall.

7.3. The game should simulate a fight between the player and the monster
  7.3.1. The player and monster should take turns attacking each other
  7.3.2. If the player has armour equipped, it should reduce the amount of damage the monster does to the player
    7.3.2.1. When an armour point is used, it will regenerate in 2 turns time
  7.3.3. If the player wins, the monster should die and the player should receive a random reward
  7.3.4. For each turn, there is a random chance of 5% the attacker will stun the defender
  7.3.5. If the defender is stunned, the attacker can attack again and the defender regenerates no armour
    The combat system works, however combat can last too long (20-30 seconds) in some cases. Ideally this should be shortened by either increasing attack damage or reducing the effect of shielding.
7.4. The rewards should be generated in the same way as chests
  These rewards are generated in the same way as the chest rewards
7.5. If the player dies, they should return to a nearby point and lose an item
  The player is respawned as close to the centre of the current region as possible. They do not lose an item as, during playtesting, it was found that it felt unfair for the player to lose an item at random.
8. The player should be able to see their inventory
  8.1. The player should be able to check which weapon and armour is equipped
  8.2. The player should be able to select which weapon they wish to equip and which armour they wish to equip
    The player can manage their inventories through an interactive menu. The user can type "eN" to equip item N into the appropriate slot.
9. The player should be able to close the game
10. The game should automatically save periodically
  The game saves every 5 moves. The project could be improved by adding manual save functionality, however this isn't particularly necessary for this type as game as there is not much that is actually modified other than the position of the player.
  10.1. The save should include data defined in 1.2
  10.2. The save data should be placed in an appropriate location for each operating system
    The save data is stored in the standard location for Windows, mac OS and Linux. In Windows, this is the AppData/Roaming folder, Application Support on macOS. On Linux, there is no real standard location, so it stores save data in a .CaveGame directory in the user folder.

5.2 Third Party Evaluation

How easy is it to use the game?

It is very easy to play the game because it has simple controls and mechanics.

How does the system meet the objectives?

The system meets all of the objectives except for it requiring the enter key to be pressed on non-Windows systems.

What improvements would you suggest?

With more time, adding more variety to the types of monsters and items to the game. More variety would allow the game's combat to be more advanced, for example to make some weapons better against types of monsters.

5.3 Enhancements

Certain areas of the project have been designed flexibly, to allow for the possibility of future enhancement. Some examples of possible enhancements are shown below:

Multiplayer: Allow a player to, as well as playing the game themselves, host the game on the local network so that other players can join. Implementing this within the existing code would be relatively easy, as most methods that interact with the player do so with passed parameters. If there are multiple players in the game, then the player passed to the functions can be modified to allow this.

Moving Monsters: Currently, monsters are stationary in the world, and therefore easily avoidable. The game could be modified so that monsters move around the world, requiring the player to actively avoid them or decide to kill them. It would be challenging to implement real-time movement into the game, as currently the game waits for user input before continuing.

Different Types of Monsters: In the current solution, all monsters are the same generic "Monster", with the same attack damage and shielding. The game could be improved by adding several different types. This would again be relatively easy, as a type of monster is simply a class that inherits from the abstract Entity class, with the monster's parameters specified by certain methods on the class.

Better Rendering System: Using the system terminal isn't an ideal environment for a game. If an emulator was used instead, keypresses without requiring ENTER would be possible on all platforms, as would using Unicode characters for cells.

Image Attributions

Moore Neighbourhood Diagram

https://commons.wikimedia.org/wiki/File:Moore_neighborhood.svg

Bresenham's Algorithm Diagram

https://commons.wikimedia.org/wiki/File:Bresenham.svg

Bibliography

[1] L. Johnson, G. N. Yannakakis and J. Togelius, "Cellular automata for real-time generation of infinite cave levels," 2010. [Online]. Available: http://julian.togelius.com/Johnson2010Cellular.pdf.

[2] K. I. Joy, "Breshenham's algorithm," [Online]. Available: http://www.idav.ucdavis.edu/education/GraphicsNotes/Bresenhams-Algorithm.pdf.

[3] Wolfram Research, Inc., "Prime Spiral," [Online]. Available: http://mathworld.wolfram.com/PrimeSpiral.html.