CS4303 150009974 29 Oct

RogueLike

Overview

The assignment asks to implement a variant of the classic "Roguelike" roleplaying game. In this game, the player controls a character and explores an endless dungeon, fights monsters, and collects treasure. The goal of the practical is to practice Procedural Content Generation (PCG) and Artificial Intelligence (AI) in video games. I have completed all basic requirements and have implemented the following extensions: 'Fog of War', a common feature for dungeon-delving games, where the player can not see the whole map, until it has been explored; various monsters with different abilities and movement patterns; the ability to create items with text files, with neither editing nor recompiling code.

In order to compile and run the program, simply execute 'make' from within the submission directory. The commands 'make compile' and 'make run' can be used separately in order to compile or run the code, respectively. The core.jar file is the Processing library used by my Java program.

Design

The following subsections go over the design of each aspect of the game. They reflect the ones in the practical specification. The last subsection describes how default values can be changed without re-compiling the code.

Dungeon

The specification refers to one dungeon split into multiple levels. However, the word 'level' is also used in character advancement. To differentiate between the two, the level of a dungeon is known as depth. In the source code, the game and hereby in this report, the word 'level' is only used to refer to character level.

The game keeps track of the depth the player has reached and generates a new dungeon for every depth. The complexity of the dungeon increases with its depth. The dungeon is procedurally generated using the third approach described in the PCG lectures. Here is an overview.

The window in which the game is displayed is the root 'region'. Randomly select one of two options: split the current region into two regions (known as child regions); or place a room in the current region. Repeat with each new region created. Stop the loop if all regions have rooms in them. Do not split the region if it has reached a minimum size. At the end of the loop, the screen has variably sized regions with variably sized rooms inside them.

The next loop connects the regions. Every region, except the root, has a sibling region and a parent region. The parent region is the one that was split into two to obtain this region and its sibling. Connect every pair of siblings with a corridor. Start with the regions that have no children and work the way up the tree until the children of the root region are connected. This ensures that there is a path between any two points in the dungeon.

Each room is composed of rectangular cells. The dimensions of a cell are calculated as a percentage from the width and height of the play window. Almost all game elements have position and size in terms of number of cells. The maximum width and height of a room are one less than the width and height of the region it occupies. This ensures that no two rooms will touch each other, as there will be at least two cells between them (1 for each region).

A ‘free’ cell is one where the characters can move into. The dungeon is first comprised of non-free cells. The PCG algorithm then ‘frees’ some cells to create rooms and corridors.

In my game, the difficulty of an empty dungeon is the number of rooms in it. A dungeon with one room is much more easily explorable than a dungeon with five rooms. The depth of the dungeon is taken into account when deciding whether to split a region in two (and thereby increase the number of rooms). The deeper a dungeon is, the more likely is it that its regions will be split. However, this means that at the first few depths, the region and room count will go up drastically. To avoid this, the chance to split a child region is a fraction of the chance that its parent is split. This means that the room count will gradually increase with the depth of the dungeon.

Movement

The player character and the Non-Player Characters (NPCs) all use the kinematic movement algorithm discussed in the lectures. This creates a smooth movement from one point to another. The characters are not bound the cells and can thus occupy multiple cells at once. This type of movement allows sharp turns which is appropriate for an exploration game where the player might wish to quickly run away from a monster. The steering algorithm would increase the difficulty of navigating the dungeon (which is already dangerous enough on its own).

The player controls the movement of the their character with the arrow keys. Pressing a key causes the character to move in that direction. When in combat, the movement keys act a little differently, as explained the combat section of this report.

The size of each character is half that of a cell. This ensures that any character can move through any corridor. This, in turn, guarantees that a character can move from any one place in a dungeon to any other. Collision is detected by comparing the distance between two characters. If that distance is small enough and one of these characters is the player, combat is initiated.

Attributes

The characters in my game have four attributes. These are Health, Strength, Agility, and Awareness. These attributes influence different aspects of the game. Health measures how much damage a character can take. Damage is dealt only in combat. If a monster’s health drops to 0, it dies and the player is awarded experience points (XP). If the player drops to 0, then the game is over.

Strength measures how hard a character hits in combat. Higher strength means more damage to the opponent. Agility measures both speed and evasiveness. High agility means faster movement and more damage reduction in combat. Awareness measures how much is conceived from the world surrounding the character. Higher awareness means larger field of view.

These attributes are inspired by popular RPGs (both table-top and video games).

The player starts at level 0 – completely inexperienced in the dungeon. Health increases by 3 at each new level. Starting at level 1 and every 4 levels after (levels 1, 5, 9, 13, …) the player's strength increases by 1. Starting at level 2 and every 4 levels after (levels 2, 6, 10, 14, …) the player's agility increases by 1. Starting at level 3 and every 4 levels after (levels 3, 7, 11, 15, …) the player's awareness increases by 1. Starting at level 4 and every 4 levels after (levels 4, 8, 12, 16, …) the player's health increases by an additional 3 (on top of the one already received).

This progression means that the player's dungeon exploring capabilities increase gradually. In order for the player to level up, they need to collect XP equal to 5 times the next level. So, to go from 0 to 1, the player needs 5 XP, from 1 to 2 – 10 XP, from 2 to 3 – 15 XP. The amount of XP a player has is reset to 0 after each level up. If the player gains more XP than needed, any excessive XP is transferred to the next level progression. For example, the player is level 2 and has 12 XP. The player than gains 7 XP. This puts the player at 19 XP which is enough to reach level 3. 15 XP is subtracted and the player now has 4 XP remaining.

Combat

Combat in my game is turn- and text-based. Text is displayed as an overlay on top of the dungeon. The character with the higher agility goes first. The player wins ties. The player can use the keyboard to select an action. The button 'A' is used to make an Attack. The button 'U' - to open the inventory and Use an item. The Arrow keys are used to jump in a direction. If the cell in that direction is free, the player immediately moves there. Usually after that combat ends, as the two creatures are further apart. A quick-acting user will immediately run away and try to leave a monster's field of view.

When attacking, the attacker's total 'attack' value is compared to the defender's total 'defence' value. The attack value is equal to the character's strength. A player can boost that value by wielding a weapon. The defence value is equal to the character's agility. A player can boost that value by wearing armour. The difference between the two is how much damage is dealt. For example, an attacker with an attack value of 5 will deal 2 damage to a defender with a defence value of 3.

Here is the reasoning for this design choice. Strength shows how hard an attacker hits. Agility represents how much of the damage the defender can avoid. Weapons let characters hit harder. Armour protects characters from hits. Using fixed calculations (without any randomness) is intentional. The player needs to consider how much damage they can do to a monster and how much damage they can take. The different weapons inspire different strategies. I considered adding randomness to determine whether an attack hits based on the attack and defence values. Damage in that case would be dealt based on strength. However, I decided not use this approach because I did not want players to rely on luck. The game offers only close combat options, as I did not have time to implement ranged weapons and attacks. Design and implementation of that is discussed at the end of the report.

Some weapons and armours are 'magical'. A magical weapon ignores 'non-magical' armour when making an attack. This happens when fighting some monsters.

Items

Throughout the dungeon, the player can find several types of items: gold, weapons, armours, consumables, bag enchantments, and torches. Gold is the ‘treasure’ that the specification talks about. It has no purpose other than being valuable. At the end of the game, the total amount of gold is displayed.

Weapons and armour affect combat as described in the combat subsection of the design section of this report. Additionally, armour also introduces an agility penalty. Some heavy armour reduces a player’s agility for the reward of great combat protection. This implements the physics of wearing heavy armour slows you down.

Consumables are potions that restore the player’s health. Torches are special types of weapons that increase the player’s awareness while equipped (they do not affect attacks). Bag enchantments increases the inventory capacity for storing unused items.

The default inventory capacity is 5 and can only be increased. Each item takes up one space in the inventory.

Items take effect only when used. Using an armour or a weapon means equipping that item. If the player has equipped a weapon and equip a new one, the old one replaces the new one back in the inventory. Consumables and bag enchantments take effect immediately when used. They disappear from the inventory when used.

Weapons, Armours, and Consumables are loaded from the ‘assets/items’ folders. This means that the user can create new items by simply adding more text files to the folders. The text files describe all the information the game needs in order to add any of these three items to a dungeon.

Each weapon has a bunch of flavour texts that are displayed when the weapon is used. For example, the inventory description of a flaming sword is “This. Sword. Is on fireeeee!”, where as the attack description of a great axe is “Apply the axe DIRECTLY to the problem.”.

Every item has a ‘value’ which is used when determining what items to place in a dungeon. A dungeon has a maximum value that it can contain. This scales with the dungeon’s depth. Therefore, deeper dungeons contain more value in total. Powerful items (such as flaming sword and adamantine armour) have high value and therefore cannot appear early in the game.

Monsters

This section describes the design of the various monsters and their movement patterns, that are featured in the game. The basic specification requires that monsters can move in the dungeon and have attributes. Both of these points are covered in the respective sections of this report. All the design described here is an extension.

There are four monster templates. Specter, werewolf, arcanist, and gargoyle. Monsters spawned can have any number of these templates. This makes 2 to the power of 4 possible combinations. Therefore, there are a total of 16 possible monsters that the player can encounter. A monster that has none of these templates is a goblin. A goblin has default attribute values and no bonuses. It is used as the base for all other monsters.

A specter has no physical body and haunts a specific location. In game mechanics, it has slightly increased agility can move through walls an can be harmed only by magical weapons. It moves randomly, but stays within range of its spawn cell.
A werewolf hunts and eats pray. In game mechanics, it has slightly increased strength, and much higher agility and awareness, and restores health when it deals damage. It moves randomly and wildly, looking for food.
An arcanist is a practitioner of the arcane arts. In game mechanics, his attacks and defence are magical (which affects combat), and can find the player anywhere on the map. It always moves towards the player.
A gargoyle is hard stone statute. In game mechanics, it has higher health and strength. It does not move, unless it sees the player. Before that it pretends to be a statue.

When a monster is spawned with multiple templates (for example a Specter Gargoyle), it takes the attribute increases of all its templates. Movement is based on precedence: the magical tracking of an arcanist is best choice. The haunt of specter, moving around a space is second best. The wild movements of a werewolf are next in line. Lastly, gargoyles stand still. If the monster has none of these templates (aka, it is a Goblin), it patrols an area, moving either only horizontally, or only vertically.

I chose to do this extension because it allows me to be creative with the monsters' abilities while at the same time implement multiple movement algorithms discussed in the lectures.

Each monster has an XP value which starts at 1 and increases with every template. This XP is awarded to the player when the monster is slain. It is also used when determining what monster to place in a dungeon. Similar to items, the XP value of a dungeon depends on its depth. Deeper dungeons have more XP in total. Powerful monsters (such as the 'Specter Werewolf Arcanist') cannot appear early in the game.

User Interface

The UI is comprised of several screens. Each displays different information to the player. All elements on the screen take up percentages of it, so they can scale well with different screen sizes.

The Menu screen gives the player the quest to enter the dungeon. Two small buttons at the bottom corners of the screen can provide additional information. One of them presents the player with the lore of the game. That is simply a flavoured description of the monsters the player can encounter in the dungeon. The other button presents the controls for playing the game.

The Play screen is displayed during most of the game play. It contains the dungeon and lets the user move the character. An overlay appears during combat. The overlay textually describes combat. When the inventory is accessed, a dedicated screen is displayed. It shows a list of all items currently in the inventory. It can be navigated with the arrow keys. If there are more items than can be fit on the screen, the list will be split into pages. Player attributes and progression are displayed only on the play screen. They are either at the top or at the bottom of the screen and change location as the player approaches either end.

The Game Over screen is displayed at the end of the game. It shows the dungeon depth that was reached along with the total amount of accumulated gold.

I used basic shapes to display the characters and the items on the screen. I wanted to insert images but in the end, I decided to focus on features (monster movement and item creation). I left room for images as described in the implementation section of this report.

In my submission, all items are circular, and all characters are rectangular with dots to show their orientation. Everything is colour coded. Monsters that are a mix of templates, are coloured in a mixed way.

Configuration

A lot of the constants used throughout the program are stored in props files. Those are simple text files containing key-value pairs. The user can easily edit those. Here is a list of some properties that can be changed: full screen mode, window size, minimum room size, cell size, the chance that a region will be split (when generating the dungeon), some of the controls, the default character attribute values, the bonuses each monster gets, the player health bonus, the XP progression, the amount of health gained at each level, the initial inventory capacity, the locations and colours of most elements displayed on the screens, the colours for the different characters. All of these configurations can be found in the files under ‘assets/config/’.

More interestingly, the properties of weapons, armours, and consumables, can be changed from the the files under ‘assets/items/*/*/des’. The ‘des’ files are property files that describe each item. Colour, size, value, drop chance, and name can be changed for all items. Some items have more properties. The user can create a new items. For example, to create a new weapon, go to ‘assets/items/weapons/’ create a new folder for that weapon, and then create a ‘des’ file for it. Fill it with the properties for the new weapon. You can also copy an existing weapon and edit its values. The process is similar for new armours and new consumables.

Implementation

I have split my code in five packages - ‘main’, ‘screens’, ‘terrain’, ‘creatures’, and ‘items’. he package ‘util’ contains java class files that I am re-using from previous practicals. They make programming easier in my opinion. Some of the classes in the ‘screens’ and ‘main’ packages are re-used from the Artillery practical this year. However, they have been adjusted to fit this new game. The ‘main’ package is smaller. It contains only two classes. The only difference in the Run class is a couple of new constants. The differences in the ScreenManager are the addition of full screen capability in the ‘settings’ method, the methods that handle user input (keystrokes and mouse clicks), and the globally accessible Player instance and Inventory. They are are static instances, so that all classes can access the player and the inventory whenever they need to.

Screens

The core of this package is the same as in the last practical. However, there are several big improvements. The re-used code is the classes ‘AScreen’ (abstract), and in the ‘components’ sub-package - ‘Button’, ‘Clickable’, ‘Label’. All other classes in these pacakge have been written for this practical. Specifically, the component ‘Message’ is practically a multi-line label that can be rendered on top of other objects. It is used to display attributes and combat overlay text. The component ‘ScrollList’ is entirely new. It contains code to render a list of text entries on the screen and allow navigation and selection of those. It is expected to be the only component on the screen. It is used by the ‘InventoryScreen’ to show a list of a items that the player has collected.

Given these components, the code for most screens is simply a constructor that creates all necessary elements and adds them to the collections of elements defined in the AScreen class. The two interesting classes are Play and Overlay. The Play screen contains a reference to the current dungeon. It also instantiates the player and inventory global instances. It takes care of displaying the dungeon, the player, and the player’s progress and attributes. The Overlay screen keeps a reference to the previous screen and also stores a single Message. When rendered, it first renders the previous screen and then its components. The position of the Message is determined based on the location of the player. If the player is in the upper half of the screen, the message is displayed in the lower half and vice versa.

Terrain

The terrain package makes up the dungeon, its regions, its cells, and defines the ground on which most other classes work. The abstract class ‘AObject’ defines basic functionality that is reused in other classes. Specifically, it says that an object should have an x and y integer coordinates. These correspond to locations in cells, not pixels. For example x=3 and y=5 means that this object is in 3 cells to the right and 5 cells down. The class provides methods to compare objects and calculate the Manhattan distance between them.

The ‘Cell’ class represents one rectangle of the dungeon. It is an AObject and its methods are mostly setters. The render method makes use of the fields to determine how to colour a cell. The ‘CellManager’ keeps track of all the cells in a dungeon. Its methods are used to mark some cells as being part of a room, find random free cells, determine line of sight between two cells, and apply a method to all cells in a slice.

The ‘Region’ class defines a single region from the PCG algorithm. It has methods to divide it in two, fill it a with room, and connect sub-regions. It keeps references to its child regions. When filling the region, it will first fill its children, then connect them. If it has no children, then it will create a room. This class keeps a reference to a CellManager and uses its methods when creating rooms and corridors.

The ‘AStar’ class keeps a reference to a CellManager and uses its methods to find paths between two cells. This class relies on the fact that two Cell objects with the same coordinates are considered the same. When the CellManager is given a Cell object, it uses its coordinates to find the its internal respective cell and determine whether a cell is free.

Finally, the ‘Dungeon’ class instantiates the root Region for the PCG algorithm, and the CellManager used to represent the rooms and corridors. It also instantiates a MonsterManager and an ItemManager (described further in this report). Most of the methods in this class deal with the correct creation of a dungeon. This includes invoking the Region’s methods to create rooms and corridors, setting start and end cells that are sufficiently far away (distance is determined via AStar), selecting cells for the monsters and treasures such that no are created in the same cell. Its public methods determine if a character is inside the dungeon (and not trying to move through walls), determine if the dungeon has been completed (player is standing on top of exit), and, of course, rendering all the monsters, items, and cells in the dungeon.

Creatures

This package defines all the properties of the characters. Two abstract classes define common functionality. 'Mover' represents a creature that moves. It has a position as a PVector, which is independent of cell. This class does not extend AObject. It is based on the KineticArrivalSketch on studres and similarities can be seen in its 'integrate' method. However, it has been heavily adapted for the purpose of this practical. It has methods to find the euclidean distance to cells and other creatures. It has a method that determines whether a cell is in front or behind the character for the purpose of field of view. It has methods to move and reorient the character, and, of course, to render the character.

The abstract class 'Stated' (the name comes from "stat", and so it implies that this is something that has stats (aka attributes)), defines characters that have attributes. This class inherits all the functionality of Mover. Methods handle attribute changes (such as level up or applying monster template), making an attack against another creature and applying a method for all cells with a creature's sight. Example usages of the last include determining if a monster can see the player and what cells should be illuminated for the player (as per the Fog of War extension). The abstract method 'act' is used to let a character take an action in combat.

The Monster class extends Stated and defines how monsters behave based on their templates. The templates are stored as boolean and methods simply check if a boolean is set to through before executing some functionality. The movement selection method select which movement method to call based on a monster's type (priority described in the monster subsection of the design section of this report). The selection is a sequence of if-else statements. The other methods overwrite some of the Stated methods to add the special features of monsters. For example, makeAttack is overwritten, so that the werewolf heals before returning from the method. A monster's 'act' method is always making an attack.

The MonsterManager stores all the living monster in a dungeon. It has methods to create a random monster based on a random roll, dungeon depth, and a maximum XP value. It also takes care of moving and rendering monsters. After each monster is moved, this class determines if it should initiate combat with the player. If combat is active, it does not move the monsters. At the end of combat, it awards XP if the monster is dead, or it moves to the Game Over screen if the player is dead. The player does not receive XP if they run away from the monster.

The Combat class deals with the combat in the game. It uses an overlay screen and sets the value of the message that is overlayed. It keeps a reference to the monster that is in combat with the player. The player can fight only one monster at a time. This classes makes use of several boolean varaibles to preserve its state. They keep track of whose turn it is, has that character taken an action, has it been described on the screen and acknowledged by the player. The Combat class has a turn method which is called every frame. The behaviour of this method depends on the state defined by the booleans. In most frames, it simply waits for the user to input their action (the state is player turn – true, action taken – false, action described – false, next pressed – false). The monster takes its action much more quickly. A couple of methods are used by the MonsterManager to check if combat is over (either combatant must be dead or the two of them are separated) and to set the screen back to the one that was overlayed.

Finally, the Player class defines all the features that are exclusive to the player. It keeps a set of the key Codes of the buttons that are pressed. It also keeps a map of button key codes to respective methods. The class also keeps track of the current XP and level, and what items are equipped. Some methods here are used to retrieve these values and display them on the screen. There are methods that handle gaining XP and level, reacting to buttons, and retrieving flavour texts for in combat. Several methods handle equipping armour and weapons, and overwrite the 'attack' and 'defence' methods from 'Stated' to account for item bonuses. Finally, several methods implement the 'jump' action which the player can take while in combat.

Items

The items package contains two sub packages. One describes the properties of individual items, the other has classes that load and store items. The abstract class 'AItem' extends 'AObject' and defines the common fields of an item. These are name, inventory description, value, drop chance, colour, render factor. All of these are constants, as they do not change once an item is created. Since they are constants, I've made them globally available for all classes to reference. Some items have other fields that a global constants for the same reason. The methods in this class define how two items are compared for the purpose of storing them in a set, how they are displayed in the inventory, and how they are rendered. Each sub class must implement the following two abstract methods: 'use' and 'copy'. The purpose of the 'copy' will be explained in later in this section.

Gold is an AItem that is does nothing when used and is stored separately from other items. The Inventory of the player has only a single reference to an instance of this class. Gold introduces one filed – 'amount'. The Inventory's Gold instance's amount increases every time the player finds Gold in the dungeon.

A Consumable is an AItem that, when used, restores health to the player and removes itself from the Inventory. The amount restored is equal to the Consumable's value.

A BagEnchantment is an AItem that, when used, increases the capacity of the inventory and removes itself from it. The increase is equal to the BagEnchantment's value.

Armour is an AItem that has a defence bonus, an agility penalty, and a magical status. When used, it invokes the Player's equipArmor method.

Weapon is an AItem that has an attack bonus, a magical status, and a bunch of flavour text. When used, it invokes the Player's equipWeapon method.

Torch is a Weapon that has no attack bonus, is not magical, and has a bunch of hard coded flavour text. When equipped, it increases the player's awareness, until replaced with another weapon. The idea behind making this a weapon is that the player cannot wield both a Torch and an actual Weapon at the same time. However, a torch can be used as an improvised weapon.

The class 'ItemLoader' loads all items into a set. Armours, Weapons, and Consumables, are loaded from text files. This class has methods that, given a dungeon depth, a random roll, and a max value create an item for the dungeon. Possible items are all loaded items with a value less than or equal to the max, and a drop chance higher than the roll. The array of possible items is expanded with new Gold, Torch, and BagEnchantment instances. A random item is selected from the array and then its copy method is invoked. This way other classes have their own instance of the item that cannot affect the pre-loaded once. Specifically, the affect is change in the item's location (when placed in the dungeon, and when looted).

The class 'ItemManager' is used by the Dungeon to store all items that are generated for a particular dungeon depth. This class renders the items and checks if any of them have been looted. Looting happens by just walking over the cell that contains the item.

The class 'Inventory' keeps track of all items that have been looted by the player. It provides methods to add and remove items, and to list them for the inventory screen.

Further work

This is an open-ended practical and it gives a lot of room for extension. Here are several ideas that I would like to have implemented in my submission.

Images. Add images for the characters and items both in the Play screen and in the Inventory view screen. This would be easily implemented by simply editing the 'render' method to use images instead of shapes. Monster images would have to be cut and mixed to account for a monster with multiple types.

Ranged weapons. Everybody loves bows and arrows. Shooting an arrow would require handling separate from the combat that I have now. It would have its own speed and movement, and it would not trigger combat. A monsters behaviour would have to change when it is hit by a ranged attack.

Spells. A new attribute 'Intelligence' would be used to make spell attacks. There would be single use scrolls and permanently learned spells with a wide variety of effects. The arcanist would also be able to cast spells. In that case, the 'act' method in the Monster class would first check if the monster is an arcanist, if so, cast a spell, otherwise make a normal attack.

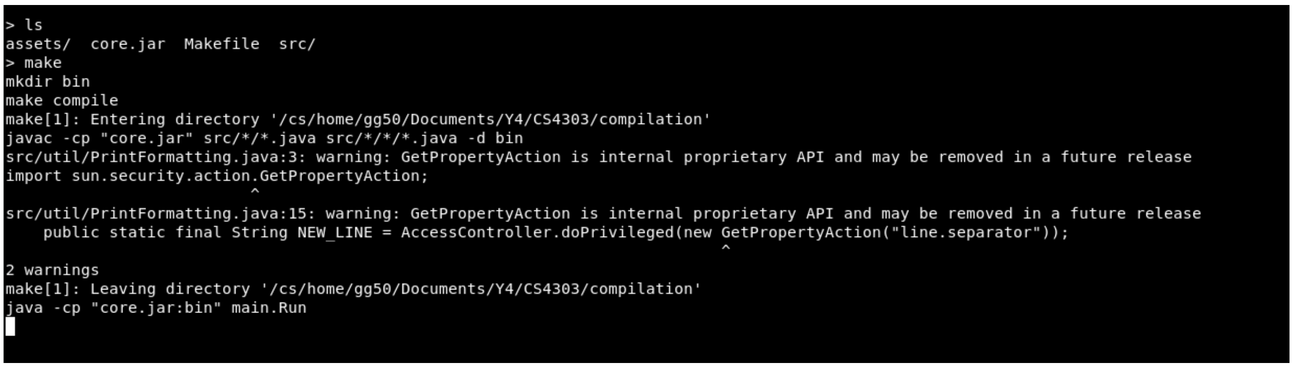
Traps and locked doors. Traps that damage creatures upon contact and doors that need to be unlocked with keys. Awareness could also be used to detect these. An adjusted A* search would have to find a path that goes from start to finish and passes through all necessary keys.

Item sizes. Bulky items (such as adamantine armour and great axe) should take up more than one slot in the inventory. In that case, I would have to keep track of total volume/weight carried.

Balance enemies. It takes time to fine tune the attribute values for such a wide array of monsters. I have to take into account their rarities, the drop chances of effective items, the player level, etc. Currently, the game seems balanced for the first couple of dozen depths but some enemy combinations (such as the Specter Werewolf Arcanist Gargoyle) do not get spawned until much deeper at which point the player is very high level and their attributes are too high to compare to monsters. Random monsters attribute values that depend on the dungeon depth is one way to solve this. Monsters could also be improved to use items and have levels as well. For example, a level 5 Gargoyle with Adamantine Armour would have huge defence.

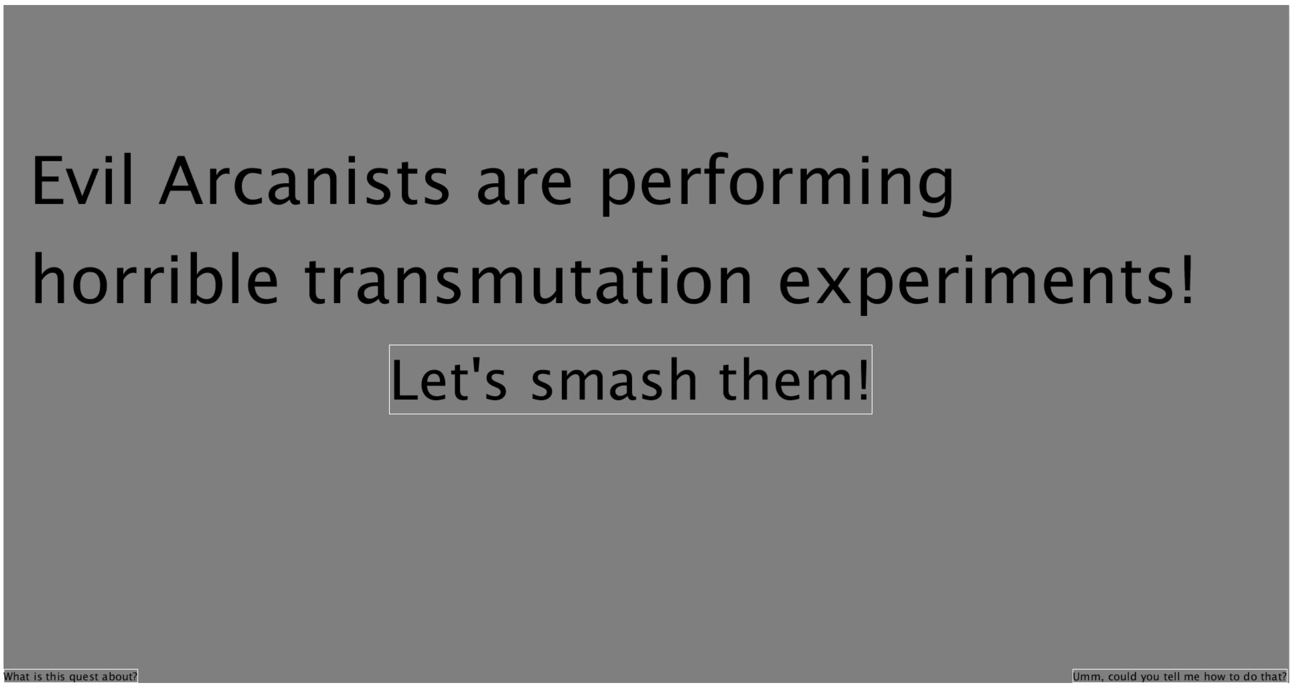
Appendix

Here are some screenshots of my working game.



```
> ls
assets/  core.jar  Makefile  src/
> make
mkdir bin
make compile
make[1]: Entering directory '/cs/home/gg50/Documents/Y4/CS4303/compilation'
javac -cp "core.jar" src/*/*.java src/*/*/*.java -d bin
src/util/PrintFormatting.java:3: warning: GetPropertyAction is internal proprietary API and may be removed in a future release
import sun.security.action.GetPropertyAction;
                          ^
src/util/PrintFormatting.java:15: warning: GetPropertyAction is internal proprietary API and may be removed in a future release
    public static final String NEW_LINE = AccessController.doPrivileged(new GetPropertyAction("line.separator"));
                                                                            ^
2 warnings
make[1]: Leaving directory '/cs/home/gg50/Documents/Y4/CS4303/compilation'
java -cp "core.jar:bin" main.Run
```

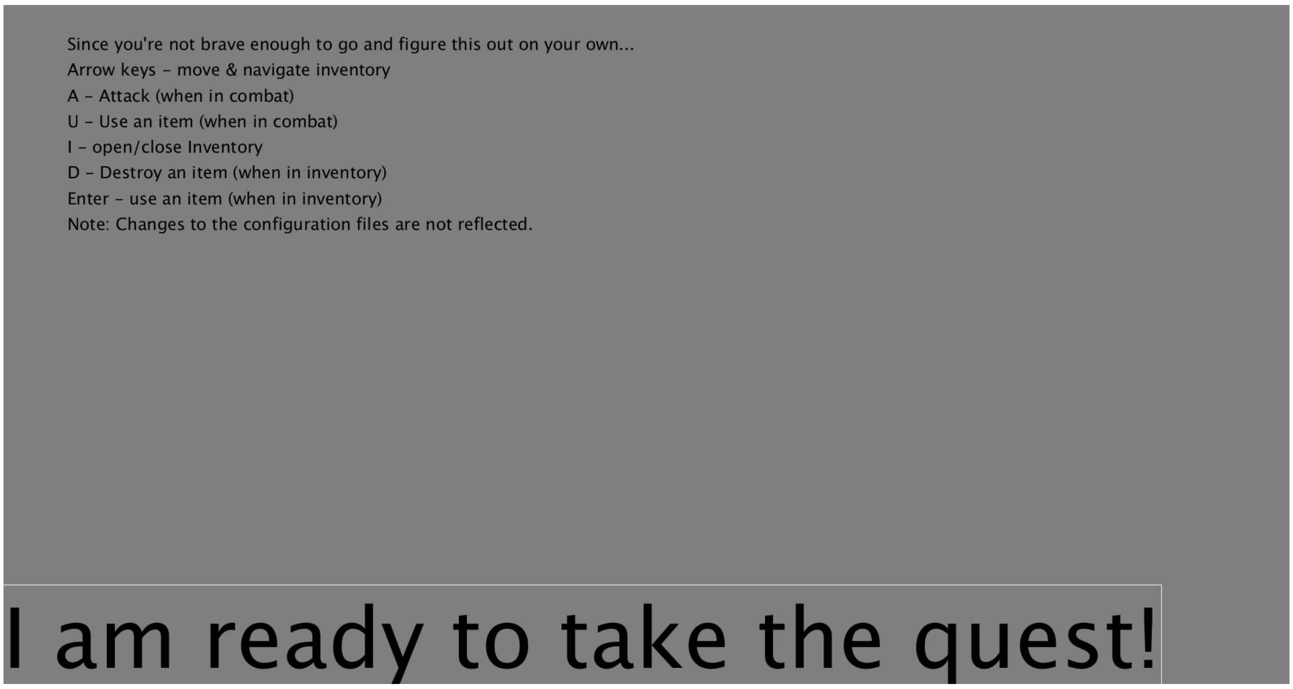
Running with the ‘make’ command.



Evil Arcanists are performing horrible transmutation experiments!

Let's smash them!
What is this quest about?
Umm, could you tell me how to do that?

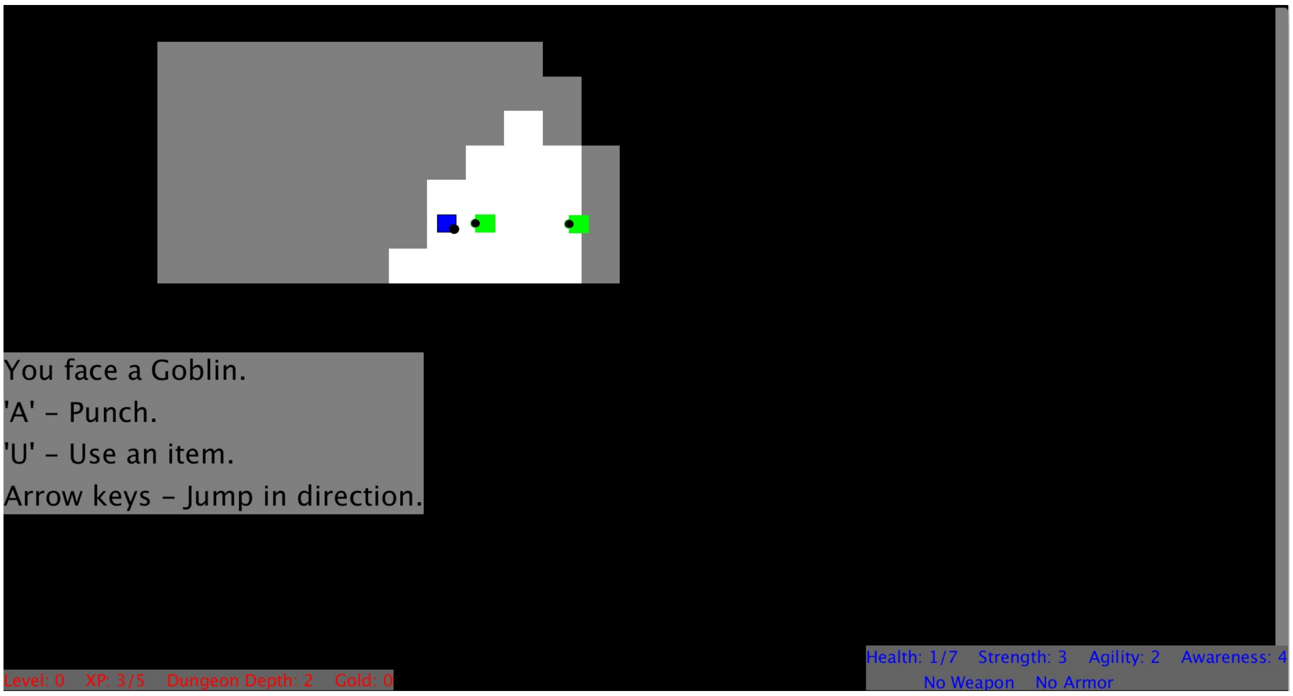
Menu screen. Notice the buttons at the bottom corners. They are small because the player should explore the game on their own, rather than read instructions.



Since you're not brave enough to go and figure this out on your own...
Arrow keys – move & navigate inventory
A – Attack (when in combat)
U – Use an item (when in combat)
I – open/close Inventory
D – Destroy an item (when in inventory)
Enter – use an item (when in inventory)
Note: Changes to the configuration files are not reflected.

I am ready to take the quest!

The controls screen, accessed via the bottom right button.

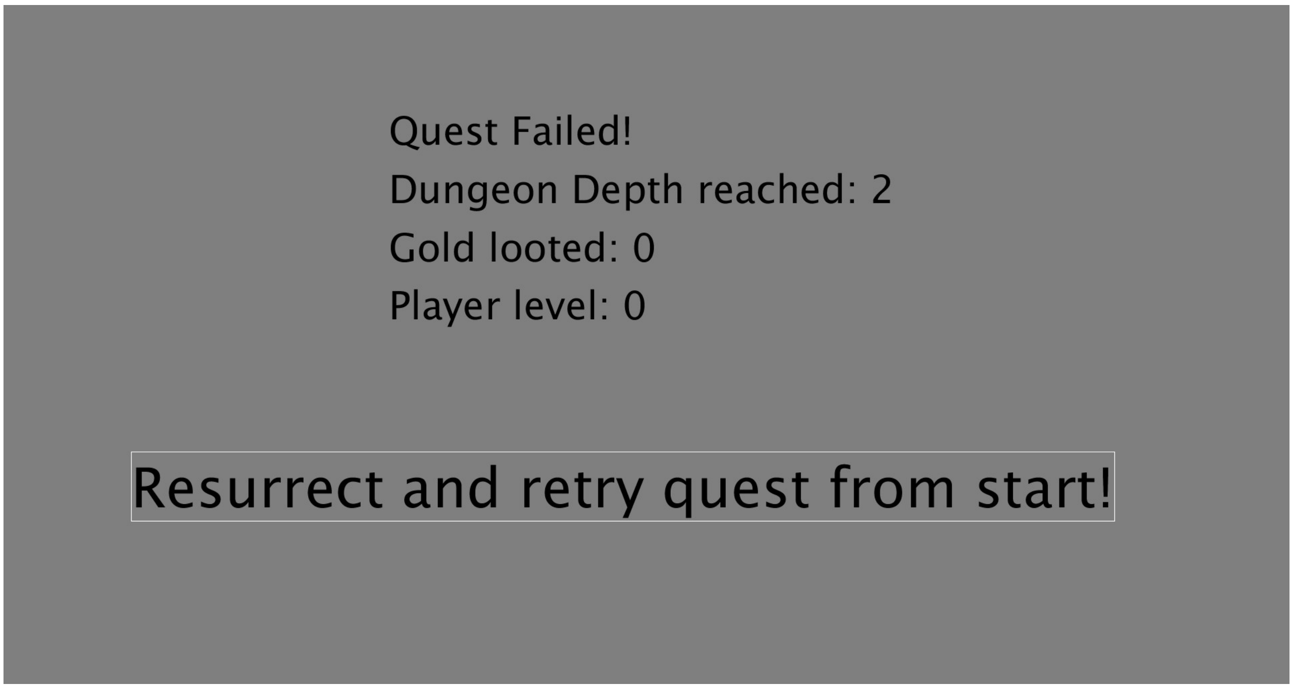


You face a Goblin.
'A' – Punch.
'U' – Use an item.
Arrow keys – Jump in direction.

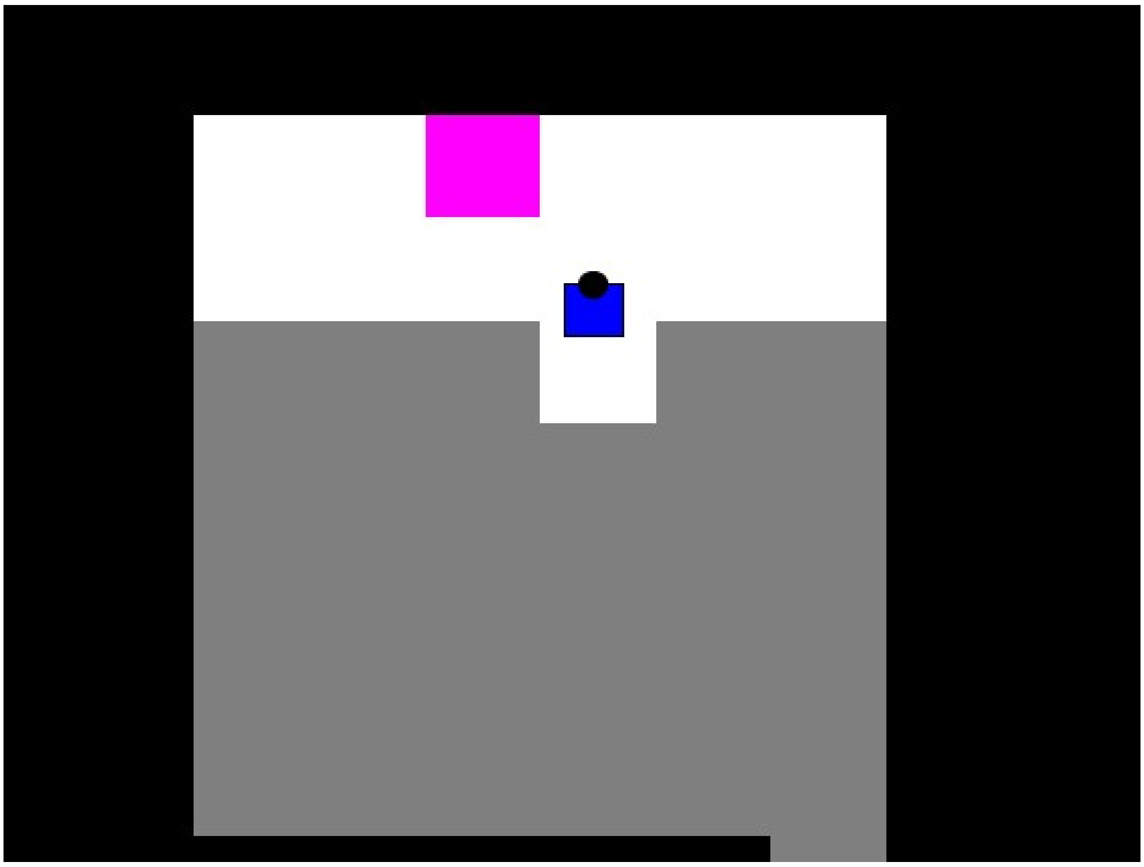
Level: 0 XP: 3/5 Dungeon Depth: 2 Gold: 0

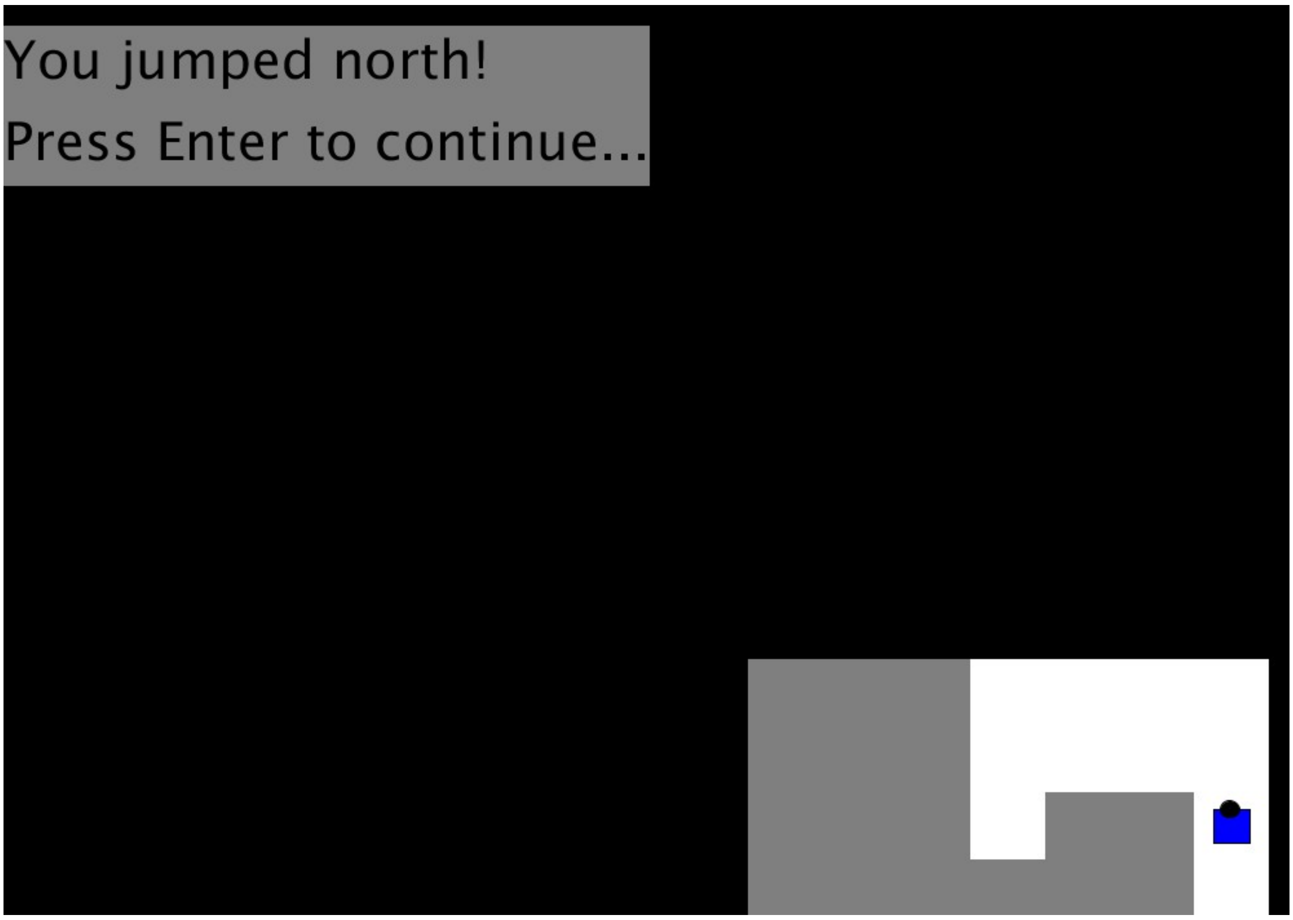
Health: 1/7 Strength: 3 Agility: 2 Awareness: 4
No Weapon No Armor

Field of view captures two Goblins. Combat has started but the player has no weapons yet, so they can only punch.


Quest Failed!
Dungeon Depth reached: 2
Gold looted: 0
Player level: 0

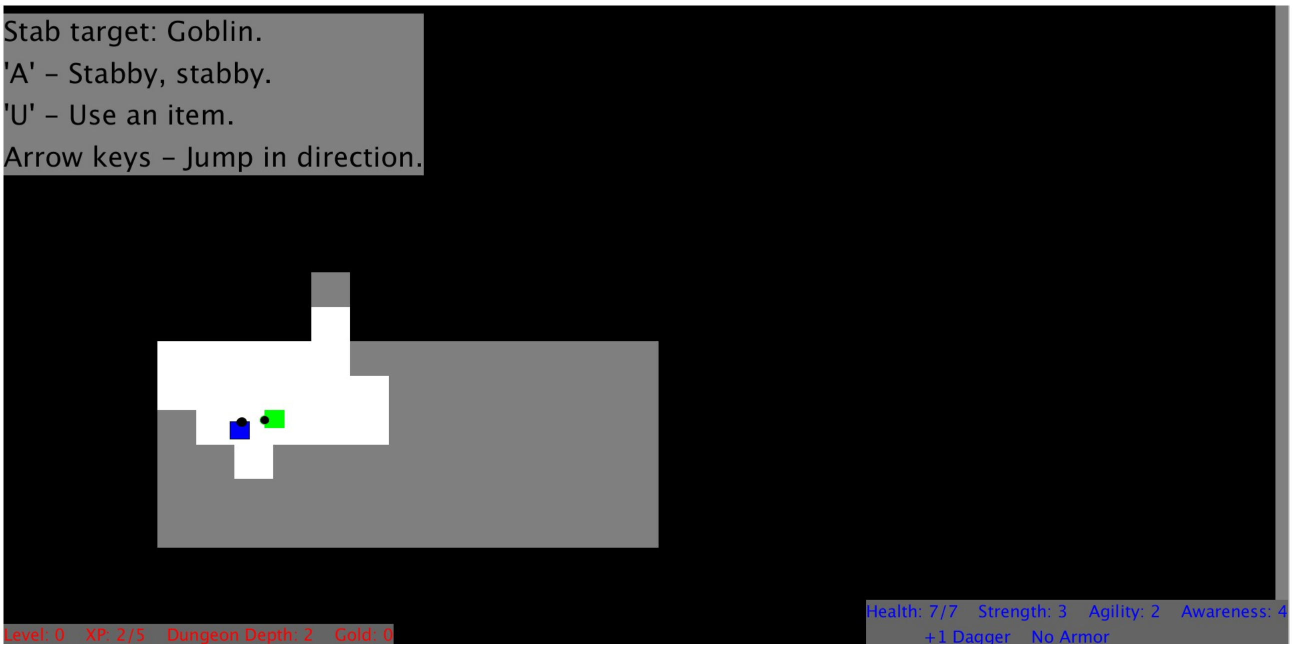
The player has died, Game Over screen.


The player has found the exit. White cells are within field of view, grey cells have been discovered, magenta cell is the exit.



```
You jumped north!
Press Enter to continue...
```

Jumped north to escape a fight.

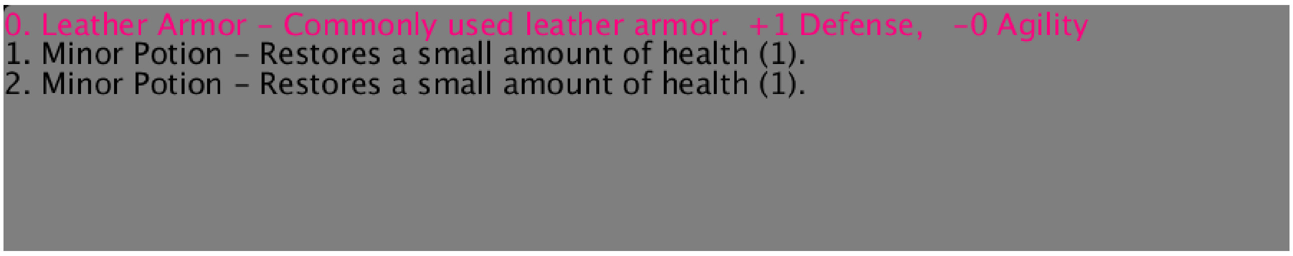


```
Stab target: Goblin.
'A' - Stabby, stabby.
'U' - Use an item.
Arrow keys - Jump in direction.

Level: 0   XP: 2/5   Dungeon Depth: 2   Gold: 0

Health: 7/7   Strength: 3   Agility: 2   Awareness: 4
      +1 Dagger   No Armor
```

Fighting a goblin with a dagger. Features appropriate flavour text.


```
0. Leather Armor - Commonly used leather armor.  +1 Defense,   -0 Agility
1. Minor Potion - Restores a small amount of health (1).
2. Minor Potion - Restores a small amount of health (1).
```

The inventory with a few item. Leather armour is selected. Pressing Enter will cause the character to equip the armour.