Brady Kruse

Dr. Anderson

Honors Discrete Structures Report

April 10th, 2018

INTRODUCTION

I’ve said it before and I’ll say it again, this game challenged the heck out of me. At first, the code was difficult. If I’m being honest, I didn’t know what I was doing and spent a lot of time learning and understanding. After working with the code and becoming more skilled as a programmer, though, I struggled more with the sheer size of the game. This game is a whopping 1700 lines of code, in total: a fairly decent workload. I’m sure there are many ways to make it more efficient, but, that being said, the game itself is naturally large. A few dozen different tiles, a handful of monsters, and about ten items, combined with game and world loops makes for a substantial amount of work. I wish I could work more on CRIM and make it as large as possible, but, simply put, I don’t have the time. I’ve worked on this project for probably 30-40 hours in total already. Now, at the end of it all, I still have tons of ideas (including a GUI, an idea that I even experimented with a bit.) But, alas, to quote Pink Floyd “the time is gone.” Nonetheless, I’m still quite happy with my project and hope you enjoy it, also.

GAMEPLAY

I won’t pretend to be an incredibly creative person, so I decided to keep the story for my game relatively basic. In the end, it boils down to a simple fetch game. Explore the map, run into some obstacles along the way, collect the keys, then win. Of course, that is dull and not very much fun, so I decided to spice things up a bit.

First and foremost, every dungeon crawler needs some monsters. I toyed with the idea of an experience system for a while, but a grind within a text-based adventure game seemed boring more than anything. So, I only added a few monsters and made them not respawn. Each monster has a different name, amount of hitpoints, damage value, and item drop. Also, each monster stems from the same enemy class – something that I’ll delve into more later.

Also, I wanted to make items that the player can pick up. Again, I decided not to go overboard with the items, and only made twelve. Like monsters, every item is different and has different attributes. Every item can be equipped as a weapon. So, if you want to attack a rat with a coin, you can! Also, most items have some kind of special action, such as healing the player or bringing the player back to life. Items, like monsters, do not respawn, although there are several instances of the same item throughout the game, especially with the coin item.

Jumping back to the keys, I wanted to make them slightly more challenging to find than simple wandering the map. Each key is obtained a different way, hence the title: Cleverness, Riches, Intelligence, and Might. This is detailed more in the CRIM Guide.

As for winning the game, the game is complete when the player collects all four keys and returns to the treasure room. Losing, however, is a bit more complicated. Death is not the end of the game for the player, however, *permanent* death is. The only way to lose the game is to die after the Potion of Life has already been used. If this happens, the game ends upon the player’s return to the Potion of Life room. Honestly, I wanted to be a little cruel and make the player run all the way back to the Potion of Life room, hoping to be resurrected again, just to have their dreams crushed.

THE PLAYER

What good is a game without a player? None. Therefore, a player class was my starting point. Originally, I had the player saved in its own module. But, after some strange errors that I still can’t fully understand involving modules not being imported correctly, I eventually moved it to the game file.

Like I said, the player has quite a bit of data in the class, including an inventory list, a location variable, a victory variable, and a key counter. The player also has a number of actions such as moving, attacking, equipping weapons, healing, and fleeing. These will be described more later.

I mentioned above that the player has a hitpoints variable, but what happens if the player runs out of hitpoints? The player obviously dies, but I didn’t want that to be the end of my game, especially since one of the keys *requires* the player to be dead! Thus, the is\_alive() function was born. Is\_alive returns True or False based on if the player has more than 0 hitpoints. I used is\_alive a lot throughout the game, mainly before the player performs an action. If the player is dead, I didn’t want them to be able to pick up items or fight monsters, so there are several if-statements involving is\_alive() throughout the game.

TILES

The map is comprised of a number of tiles. Each tile is assigned an x and y coordinate in the map loop (mentioned below). There are several different tiles within the tile module, but they all stem from the parent class MapTile. MapTile has a few functions, specifically intro\_text and modify\_player. Intro text is exactly what it sounds like, text that is displayed upon entering the room.

Modify player is a bit more interesting. Modify player is run during every turn, allowing the tile (or the enemies in the room) to modify the player several times. I mainly use this for enemies: the player is attacked every turn until the enemy is dead or the player leaves the tile. But I also used it for when the player picks up keys. That way the player doesn’t have to leave the room then reenter in order to pick up a key. There are a number of other instances where modify player is used by a tile. Like I said, every tile is different in some way, therefore each tile has some sort of unique code used in it. Because the player is actually being modified, both intro text and modify player require the player to be a parameter.

Like I said earlier, there are several items within the game. Therefore, item tiles required a pick up item function, along with an item variable. The pick up item function usually appends the item to the player’s inventory and changes the item variable to false so that the player cannot pick up the same item twice.

In addition to all the other tiles, there are a number of tiles that are unlike any others and required specific code for each one. Most of the key rooms are like this. For example, one key room required a riddle function. Another required a money check to see if the player could afford to buy the key. Similar to that, there is a vendor tile that allows healing potions to be bought for ten gold, which also required a money check. Oddly enough, there also is a bar that required a check if the player is alive and, if so, the player is handed a beer which appends to their inventory. There were plenty of instances like this where the code was different from anything else in the game.

ITEMS

Like the tiles, each item is unique in its own way, but they all stem from a general item class. The general item class is fairly basic, not doing much more than assigning a name, damage, description, and overloaded string function. Mainly, these things allow the item to be printed in the player’s inventory, but not much else.

I started with weapon items. These didn’t differ too much from the item class except for one exception: sharpen. Every weapon can be sharpened which changes the name and the damage of the item.

Moving on to the more complicated stuff, I created potions. Potions were tricky because they modify the player in some way. For the healing potion, I created a player action called heal that, you guessed it, heals the player by increasing hitpoints. But what happens if the player doesn’t have healing potions, or more than one? Therefore, I also created some variables that kept track of how many healing potions the player has.

Staying in the vein of potions, I inserted a Potion of Life that raises the player from the dead. Like the healing potion, this item required its own action, so I made one: undead. Undead can only be called when the player has a Potion of Life and resets the player’s hitpoints back to 100. To make sure this worked properly, I implemented a number of if-statements that made sure the player wasn’t already alive and made sure that the player actually had a Potion of Life.

The rest of the items were fairly easy and all stuck to the base item class, save for one: the fire shield. Since the fire shield modifies an enemy’s damage, I added a variable that told whether or not the player had a fire shield which, in turn, determined how much damage the enemy would do.

ACTIONS

Since each action is very different from the others, I’ve decided to simply include a list of actions along with a brief explanation of what they do and the code behind them. All the actions are classes in the actions module which simply stores the action name, hotkey, and the corresponding player action. That way, whenever an action is called, it actually calls the action in the player class since all the actions involve the player in some way. Actions are made available based on functions within the tiles that add all possible actions to list, then returns the list. This prevents a player from attacking if there are no enemies in the room and such.

* Move – Exactly how it sounds: moves the player. This is done by modifying the player’s x and y coordinate, then restarting the game loop. Since 0,0 is actually in the top left corner of the map, Move North changes the y-coordinate by -1 and so on. However, the player cannot move any direction at any time. I added a function that checks whether or not the surrounding tiles actually exist, then adds the real tiles to a list called adjacent moves.
* View Inventory – Prints each item in the inventory, along with the current equipped weapon. Pretty basic for-loop.
* Attack – Ah, finally, the fun stuff. Attack does damage to the enemy’s hitpoints, so, naturally, it requires the enemy as a parameter. A standard damage amount is pretty lame, so I used the random module to spice things up a bit. The player’s damage is a random number from 20% below and above the current equipped weapon’s damage amount. So, a sword with 10 damage can hit anywhere from 8 to 12. The same applies to the enemy attack.
* Equip Weapon – Allows the player to equip any weapon in their inventory. The player types the name of whatever they want, then, I created some code that looks through each item in the player’s inventory. If the names match up, the item is equipped. If not, the player is told they don’t have whatever they typed.
* Pick Up Item – Appends an item to the player’s inventory. Since the items are actually stored in the tiles, this action simply calls the pick up item function within the room.
* Heal – Adds to the player’s hitpoints if they have a healing potion.
* Undead – Resets the player’s hitpoints to 100 if they are dead and have a Potion of Life
* Flee/Stumble Through The Door – Sends the player to a random adjacent tile using the tile exists function.
* Check HP – Simply prints the player’s hitpoints.
* Sharpen – Calls sharpen function for each item, since each item responds differently to this command. Only available in the dwarf room.
* Drink – The player drinks a beer which decreases hitpoints by three. Only available if the player has a beer.
* Do Action – How the player actually DOES the action. Do Action is called in the game loop (mentioned below), takes \*\*kwargs as a parameter (as each action requires a different number of parameters), uses getattr to find the proper attribute, then performs the actual method.

ENEMIES

There are only a few enemies within CRIM, so this module didn’t take near as long as I originally thought. Like my previous modules, all my enemies are based out of the enemy parent class. Each enemy, like the player, has a name, hitpoints, and damage. All enemies also have a few functions such as attack and is\_alive, along with a few variables such as is\_hostile (which is used to determine if the enemy attacks on sight.)

There a few unique enemies in the game, however, specifically the key-bearers. These often required a bit more coding than the others. For example, the sphinx needed to ask a riddle, the goblin needed to ask for money, and the dragon needed to drop a key upon his death. Like the player, attacking uses the random module.

MAP

This is a pretty neat bit of coding that I was fairly pleased with. Essentially the world module is a loop that analyzes a txt file and assigns x and y coordinates based on separation by tabs, along with initializing each tile. So, the actual map is simply a txt file spaced out with tabs.

I won’t lie and claim I thought of this on my own. I originally saw the map loop on an online guide, and, surprise, it didn’t work very well. So, I made a number of modifications and now have a working map loop!

GAME

The game loop, by far, was the trickiest bit of coding I had to do for this project. I needed a lot of help from online guides but now, after implementing it myself, I fully understand it and think I could do it independently.

The game loop starts by printing some credits, initializing the world, then creating the player. Then, it jumps into a loop that doesn’t end until the player wins or loses. The loop prints the current room’s intro text, runs the current room’s modify player function, then performs whatever action the player chooses. This is essentially one “turn.” Also, within the loop is another loop: one that runs through all actions and, if the one the player entered is not found, displays an invalid input error. Outside of the loop, at the very end of the game, the credits are printed.

OTHER

To cap off my game, I did a number of little side projects. I added some ASCII art throughout the game, created a few readme files and a guide to help the player, inserted a number of easter eggs throughout the game, and did my best to format the text on the screen for readability.

Once my game was finished, I sent it off to some friends to check for bugs and playability. I also played it through several times myself and did my best to explore every possibility.

DISCRETE STRUCTURES

So, how does all of this tie in to discrete? In my opinion, how doesn’t it? Discrete is logic – considering if something is true or false and what happens for each option. There are *hundreds* of instances of this throughout my game: the player gets certain actions based on if they are alive, the game is complete based on if the player has the keys, the player can attack an enemy if and only if it is still alive. The list goes on. Honestly, I could write a thousand pages on all of the possibilities alone.

My game relies a lot on logic and certain conditions being fulfilled. So, yes, I think that discrete mathematics ties in quite nicely.

CONCLUSION

I’m finally done with CRIM and already I feel a load being lifted off my mind. Creating this game was a *ton* of fun and I often caught myself looking forward to sitting down, listening to music, and coding for a few hours. That being said, I’m definitely ready for a break. I plan on revisiting this game sometime in the near future, but, for now, it will hold a dear place in my “Completed Projects” folder on my Mac.

Once again, thank you Dr. Anderson for allowing me to partake in an assignment that isn’t strictly serious coding. I hope you enjoy my little game.

SOURCES

PLENTY of help from Phillip Johnson ([www.letstalkdata.com](http://www.letstalkdata.com))

ASCII Text from [www.patorjk.com](http://www.patorjk.com)

ASCII Art from Pranay Marella

[www.stackoverflow.com](http://www.stackoverflow.com)

Inspiration from Zork