Project Documentation - SDEV-435 Applied Software Practice I

Kendall Hudson

The Ravening Toad

Project Documentation - SDEV-435 Applied Software Practice I

Executive Summary

I am creating a dungeon exploration game in which the player takes on the role of a hideous toad-like creature which runs a horrible little cafe in an ancient cellar beneath the crumbling remains of a dying, degenerate city in a grim fantasy world. The creature makes its living by exploring the labyrinthine sewers beneath the city and harvesting whatever semi-edible things it can find there to cook and sell at its cafe. This quest for ingredients makes up the primary experience of this game.

Journeys into the sewers will occur as the toad-creature wanders the tunnels and becomes lost, with the player taking over at that point and exploring a procedurally generated maze until an exit is found at which point they escape with whatever harvest they've managed to find. These mazes are filled with dangers in the form of aggressive creatures and possibly environmental hazards.

As a secondary element, when the player returns from a successful foraging / hunting trip there will be a simple mini-game in which the found ingredients are prepared and sold. This mini-game works by placing ingredients into categories which then fit into recipes, such as a recipe which requires three different ingredients from specific categories, and the resulting dishes then selling for predetermined amounts of money. The money is then spent on new equipment to help with future foraging and possibly upgrades for the cafe as well. After a shift at the cafe, once all the food is sold and the shopping is done, the player then returns to the sewers for more resources.

At this time, to keep testing simple, there is only one type of monster, one ingredient, and one recipe.

% Complete

This depends on how we look at it. If we want it to be fully playable and error free then it's at 100% right now. However, if we want it to actually be fun then I'd say it's about 80%. I'd say it needs another two weeks / one month of work to be fleshed out to the point that the game is actually fun to play.

Links

Github Profile
https://github.com/khudson13

Project Repository
https://github.com/khudson13/The-Ravening-Toad

Documentation
https://github.com/khudson13/The-Ravening-Toad/blob/main/The%20Ravening%20Toad/Hudson_SDEV-435_System_Documentation.pdf

Video Walk Through and Catchall of Use Cases
https://drive.google.com/file/d/15YF-E5ied0RmMpT09nKghLSyMyOXKZtb/view?usp=sharing

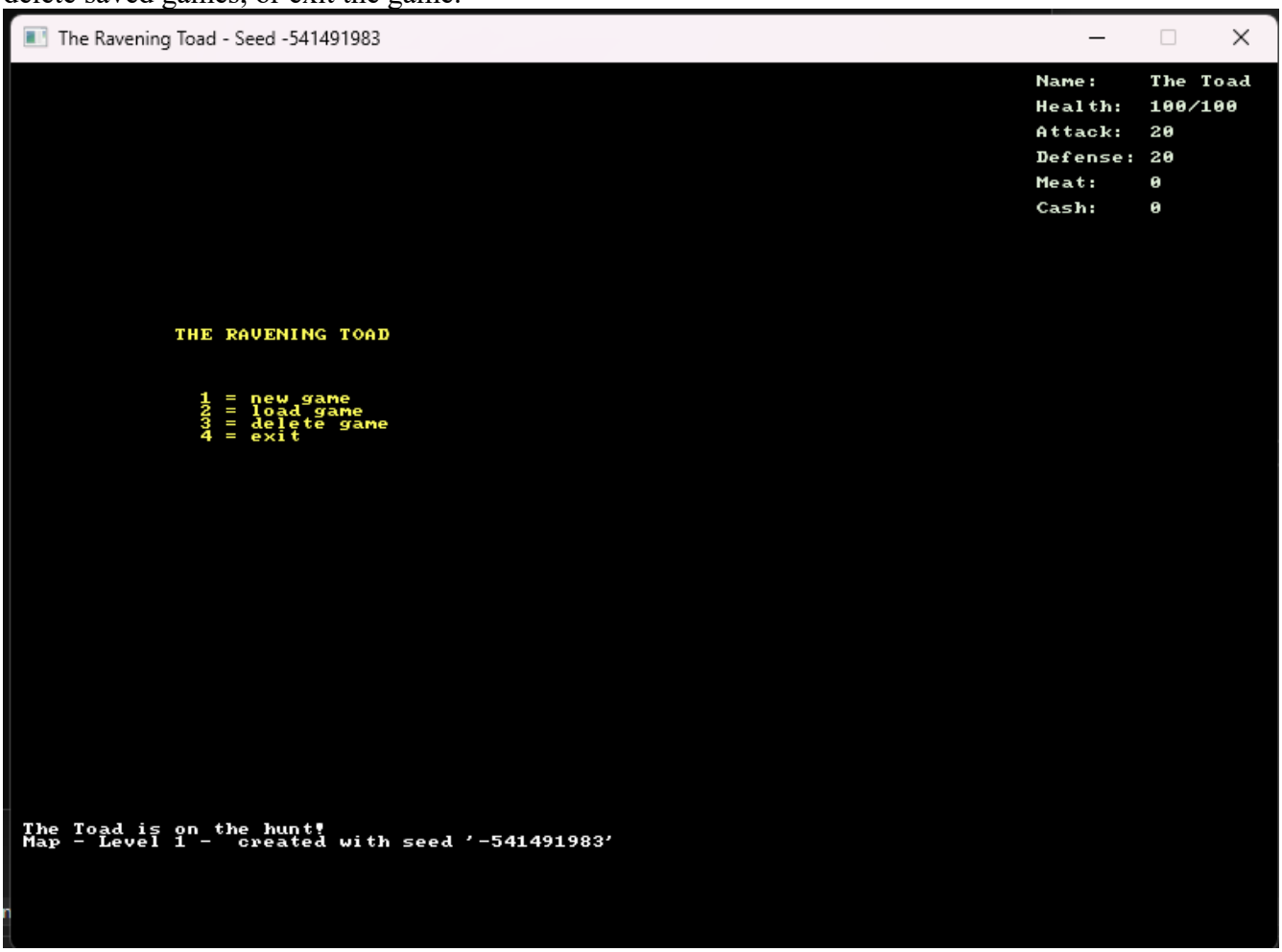
Source-Code Deliverables Video (3 in 1)
https://drive.google.com/file/d/1mDaT1ibpgAe1W_ADmGOB8YVxP6OatvER/view?usp=sharing

Scenario | Program Usage Description

"The Ravening Toad" is a game and as such there is really only one usage scenario: playing the game. The steps involved in playing the game are presented as follows.

The Start Screen

Players first arrive at the start screen where they can start a new game, load a saved game, delete saved games, or exit the game.

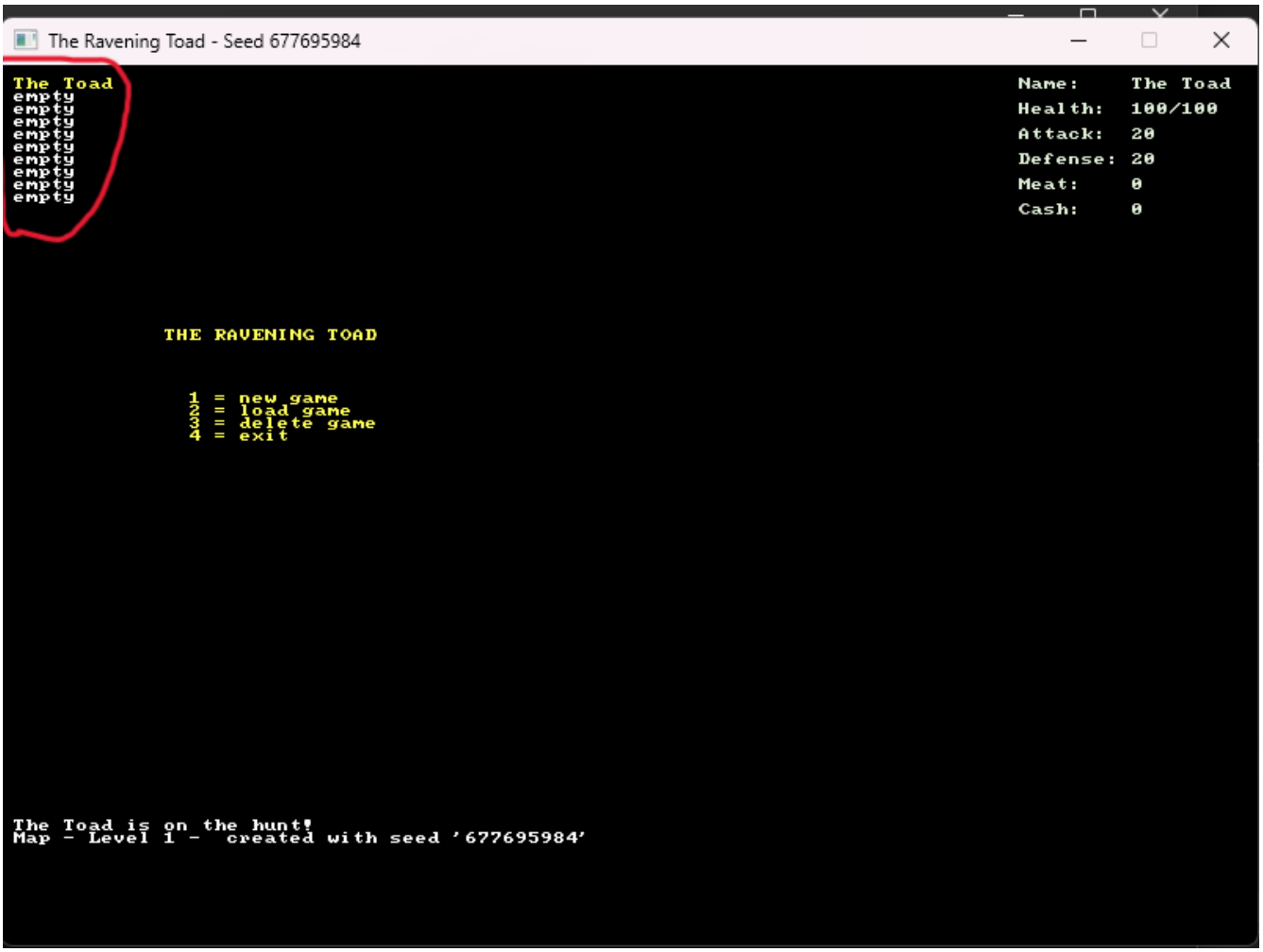


The Loading Screen

The loading menu appears at the top of the screen and is available from the Start Screen and from the Exploration Screen (see below). This menu is used to load saved games and resume previous sessions.

The loading menu is visually identical to the save menu and the delete menu. The save menu allows the user to select one of the ten files to store the data which defines their current session. The delete menu allows the player to delete save files, and prompts to player to verify they are sure they want to perform the deletion before it will proceed.

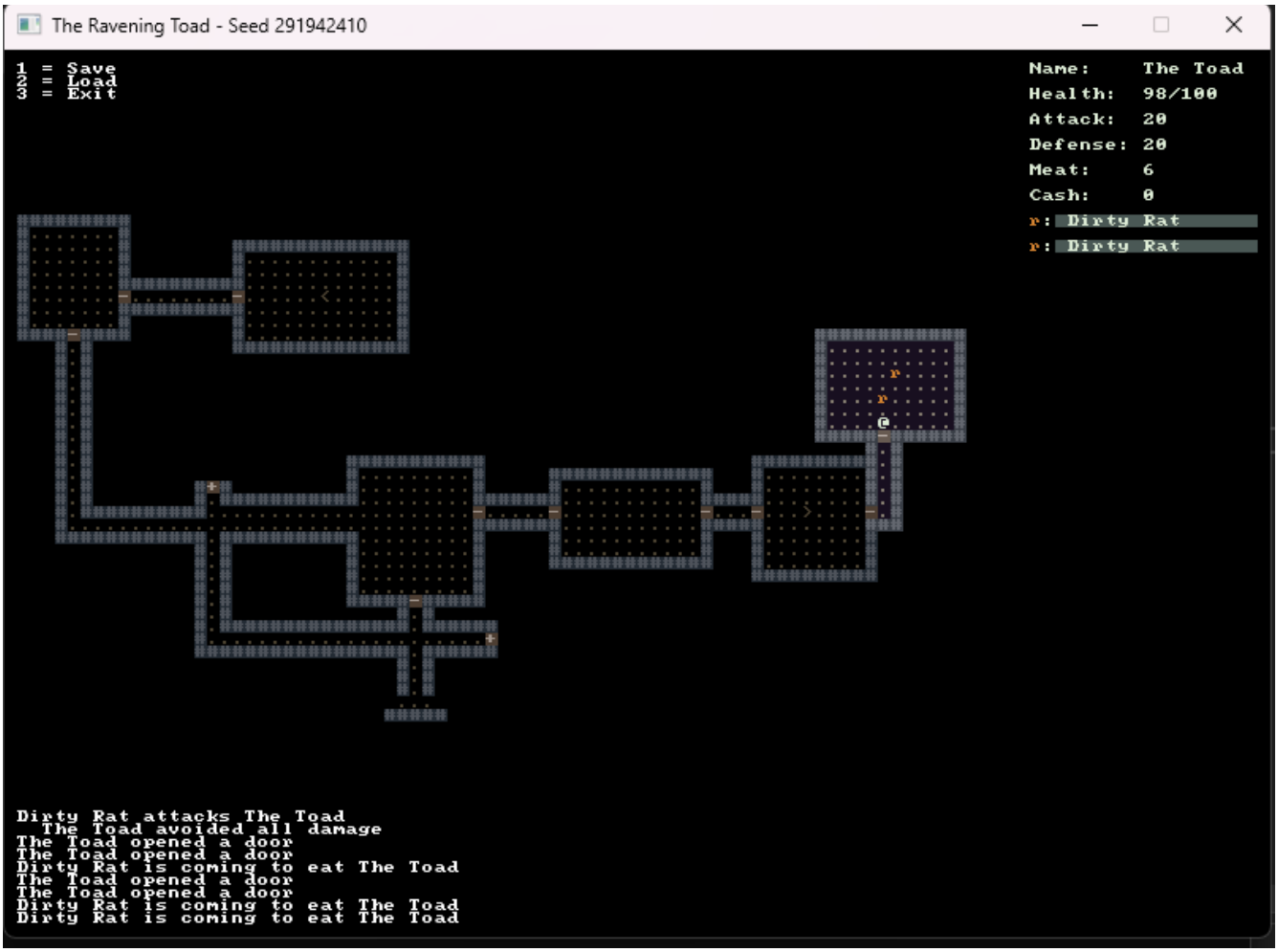
In the image below, the first save file contains data and the rest are empty. Attempting to load an empty file returns an error message and then disregards the attempt.



The Exploration Screen

This is where the bulk of the game takes place. The player moves their character around the map in the center of the screen to explore the area, defeat enemies, gather ingredients, and ultimately to escape the area and return to the cafe. Enemies are attacked by attempting to move into their space.

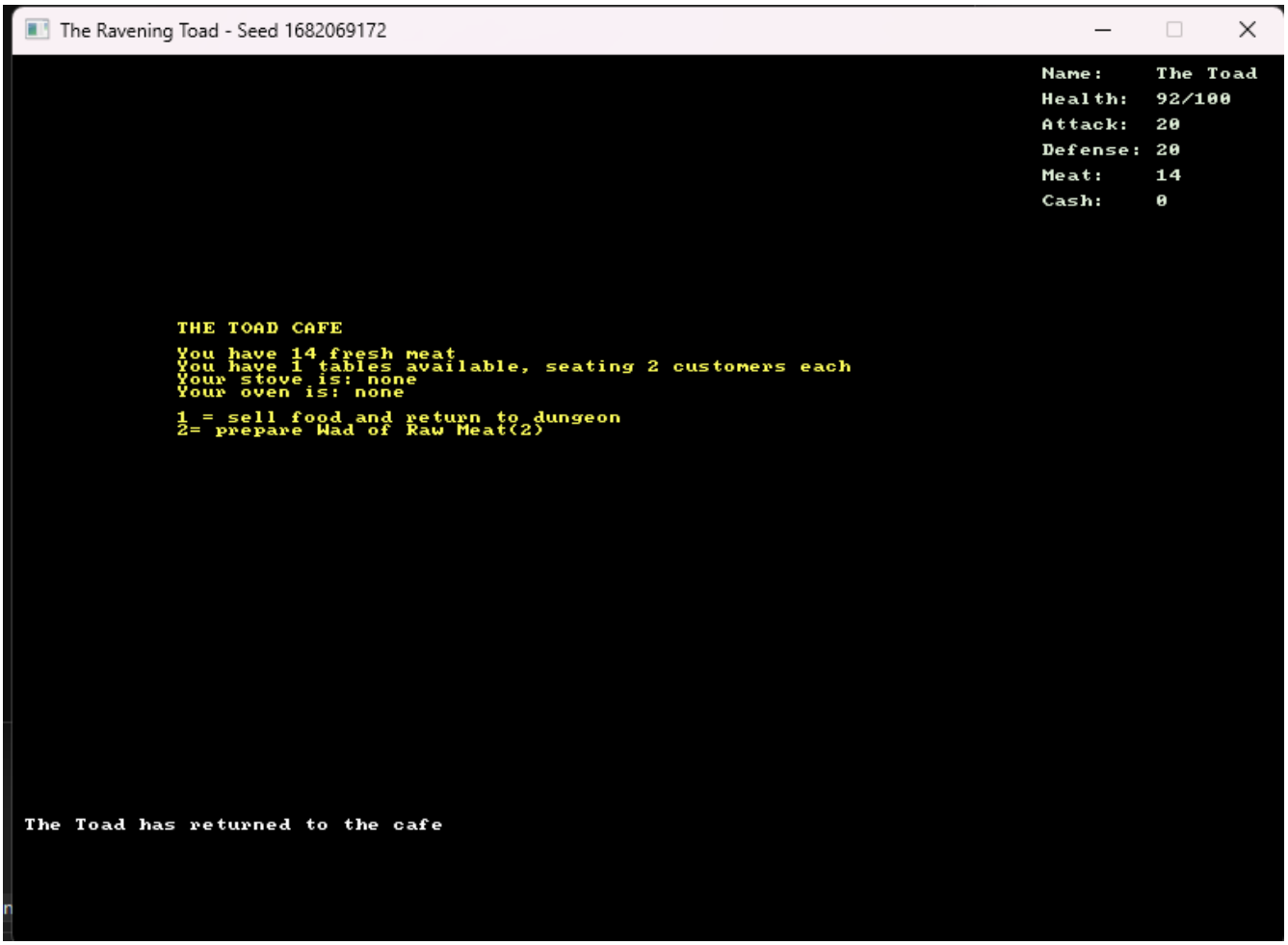
Data about the player character and nearby enemies is displayed on the right hand side of the screen. A message log is printed at the bottom of the screen to inform the player of events in the game. In the image below the pause menu is also visible at the top of the screen, from which the player can save their game, load a previously saved game, or exit the game. A portion of the map has already been explored to illustrate a partially revealed map, and enemy data is visible on the side panel. The player is represented as an '@' and the enemies are represented by 'r.'



The Cafe Screen

After the player completes a dungeon the Cafe Screen is where they turn their hard won ingredients into meals to sell to customers. As of now the player cannot upgrade the cafe and so only one recipe is possible, and only one table exists for customers.

When the player is done selecting recipes and preparing food they sell to customers, get their money, and return to the Exploration Screen to play through another level. This is the full game-play loop as of now.



Reports

There are two reports within the game, both of which are essential to successful game play. The right hand side of the screen is devoted to reporting data about the player and visible enemies. The data, in order of appearance from top to bottom, is the player character's name, health, attack and defense ratings, collected meat, and earned money. Below that enemies are listed starting with the letter or keyboard character which represents them, their name, and a colored bar behind their name representing their remaining health which diminishes are they are wounded.

At the bottom of the screen a message console prints useful information about combat events, enemy actions, cafe events, and other data.

System Architecture

Main()

Main() can be found in Game.cs and is primarily responsible for instantiating the objects responsible for the rest of the program. These objects include the game map (ToadMap.cs), the system which processes player input (CommandSystem.cs), the system which keeps track of who's turn it is and processes Non-Player Character actions (SchedulingSystem.cs), as well as the various menu handlers (MenuControls.cs, MainMenu.cs, LoadMenu.cs, etc.), as well as the various output consoles and others. At the end of main control is passed to the console function from the RLNet library: _rootConsole.run().

Main() also contains the two functions which drive the player experience: OnRootConsoleUpdate() and OnRootConsoleRender(), both of which are required by _rootConsole and which change the state of the root console and update console output respectively.

Interfaces and Entities

The system makes use of various interfaces which are then implemented by the various entities within the game world. These interfaces, such as IActor, IBehavior, IDrawable, etc. serve as guarantees that active entities within the game world can be operated in consistent and reliable ways.

For example, IActor is implemented by Actor which is extended by Monster which is further

extended by DirtyRat. This means that DirtyRat, and any other monster as well, can be operated using the methods standardized in IActor. The approach is used for all active elements within the game world and can be broadly extended to include nearly anything the user might want to implement into that world.

Map Creation

All maps exist in the same size of space. A list of rectangles is created and then placed at random within that space. If the rectangle trying to be placed overlaps with a rectangle already placed then the new rectangle is simply discarded. Rooms are connected by 'L' shaped hallways which link the centers of rooms together. Doors are placed wherever appropriate openings occur in walls (walls on two sides, open space on the other two sides). A random number of enemies is determined per room and placed at random within the room. The player is set at the center of the first room which was placed. The exit is located in the last room which was placed.

Primary Loop

The game loop is focused on the scheduling system which organizes all instantiated actors into groups according to '_time' which is an abstract measurement of time divided into turns. Each list within the scheduling system contains all the actors who need to be moved on that list's associated turn.

When the player completes their turn the scheduling system moves on and tells each Non-Player Character to move. Moves are handled in CommandSystem using data found in the respective Actors. Once the system has built the space in which the game is played, the main loop is actually very simple in concept:

Player Acts → Enemies Act → Repeat

Player actions are also very simple, comprising movement, attacking, and opening doors. All these actions are executed by pressing the arrow keys and are interpreted contextually by the system based on what exists in the space the player is trying to move into. Menus are accessed by pressing the escape key and then using number or arrow keys to select options.

Saving and Loading

There are ten files dedicated to holding game data. The first data in each file is a boolean indicating whether or not the file contains data. Empty files say “FALSE” at the top and trying to load them returns an error which is printed to the message console and lets the user try another.

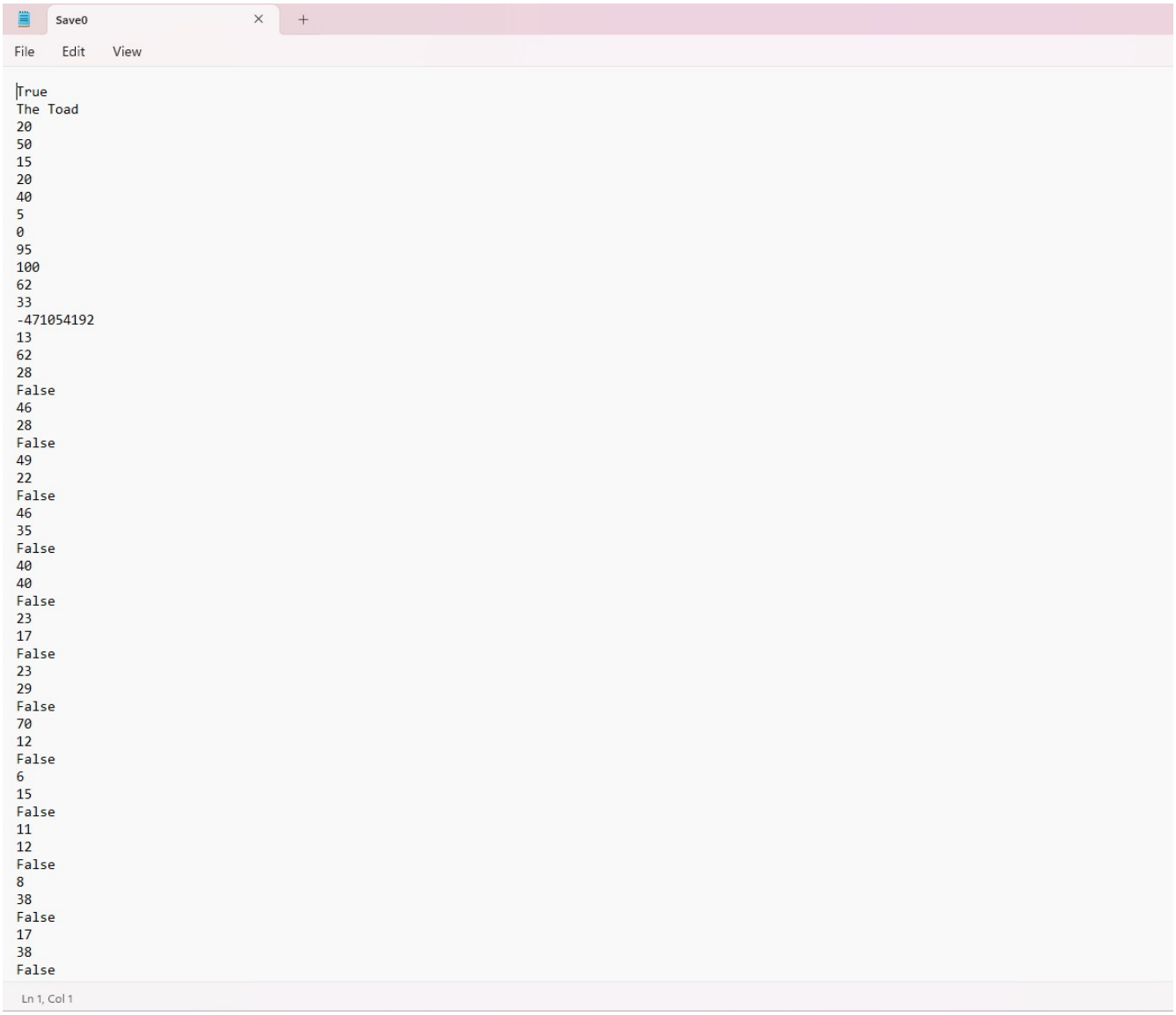
Next the file contains data concerning the player character, specifically name, health, attack and defense, meat, money, and location data. These data are written to the file in order and when the file is loaded they are read in the same order and stored in the Player instance.

After player data comes map data. The first, most important data is the seed number which is used to randomly generate the map. When the file is loaded this number is passed to the map generator which then uses it to generate an identical map. This is possible because generation is procedural rather than truly random, and is made to appear random by utilizing a number obtained from the system clock.

After the seed various data are stored concerning map elements which may change during game play. These include the open or closed status of door, the location and health of monsters, and data about whether or not each map cell has been previously explored. Each of these are respectively held in Lists until they are requested by the map generator, allowing changes to be maintained.

The largest amount of data in the file is regarding the exploration status of map cells and this

data is represented by a long string of true / false values. See below for an example of a portion of a save file.



Directory Structure

Directory	**Usage**
Core	The basic objects are here, such as ToadMap and Actor
Systems	Vital systems which drive the game, such as SchedulingSystem, CommandSystem, and MapGenerator
Interfaces	All the interfaces are gathered here for easy reference
Monsters	All the monster definitions
Recipes	All the recipe definitions
Save Files	Ten flat data files, no code here

Future of The Toad

While I've accomplished a lot in building The Ravening Toad, there is still a lot I'd like to do. The following is a short break down of the things I'd like to add to the game before I call it finished.

Items

Nearly every game of this type features a wide range of items. They may be useful in combat, or they may heal the player character or make them stronger, or they may be useful in solving a variety of other problems.

An old favorite of the genre is random potions, which the player can find without knowing what they do because descriptions are randomized. They may find a bottle of milky, sulfur smelling liquid and find that drinking it heals their wounds, but on their next game with a different character drinking the same liquid causes instant death!

I'd love to have something like that in my game, as it give the player a variety of extra mysteries to solve and makes each play different from the one before.

This would be implemented with a new class of object and an associated interface to represent the items. I imagine a base item class with multiple children for different sorts of items e.g. those which effect the player, those which are thrown at enemies, those which are put down as traps, etc. The player object would likely store a list of owned items.

Equipment

The Toad should be able to put on a helmet and some chain mail, or other gear, and carry a better sword to be more effective in combat. There should be a progression of better equipment as the player becomes more successful and faces greater challenges.

Implementing equipment would probably be fairly simple. It could build off of the items system, with equipment basically being a subset of player effecting items but with special limitations and continuous, rather than temporary, effect. By “special limitations” I mean e.g. the player can only wear one helmet, whereas they might be able to take several protection potions at the same time for extra effect.

Difficulty Progression

Speaking of greater challenges, the dungeons should become more challenging as the player progresses and the rewards should become greater. I'd like there to be a whole series of different themes such as magma filled dungeons or acidic swamp dungeons with their own special challenges and opportunities. This would add a layer of strategy and give the player reason to keep moving forward.

This would be a much more involved implementation than the above topics, being a major extension of the procedural generation system. Fortunately, that system is built with extension in mind but I'd need to have different sorts of cells (currently there are only walls and floors) and I'd need to be able to define “area effects” e.g. clouds of toxic gas or pools of dangerous liquid. I think this goal is within reach, but it will have a lot of moving parts.

Expanded Cafe System

There's a lot I'd like to add to this system. I want more recipes, cafe upgrades, and a system to control the sorts of customers the player can attract. The recipe system is already mostly in place, I just need to add more distinct recipes to it and probably extend it to be able to handle more complex sets of ingredients and required equipment.

The cafe equipment system barely exists right now. This will be another class of objects with another interface, and will mostly provide different “qualities” (probably defined in an enum) as required by different recipes. For example, if a recipe require the quality of “fine cutting” then the cafe will need to be equipped with a good set of knives.

Customers are the biggest challenge here. I need a system to determine how many customers are available and what they want, and what happens when the player can't satisfy them. That's going to be another class and interface for the customers. The cafe will need something to store customers waiting to be served, and what better place to store customers than a queue? I've already got a function which finds the most valuable recipe available and sells it, and I'll need to write another which determines what to do with remaining customers once all the food is gone.

Executables

There is only one executable file for this game, which handles all operations.
The Ravening Toad.exe

Programming Language | C#.NET

This project is written in C#, and utilizes the RLNet and RogueSharp libraries.

Project Classes

Main Class | Game.cs
This is where Main() lives and global variables are like Player and ToadMap are held.

Parent of Player and all Monsters | Actor.cs
Catch-all parent for anything which acts on the game map according to a schedule.

Player Class | Player.cs
Stores all vital player character data

Parent of all Monsters | Monster.cs
Stores methods relevant to all monsters.

My First Monster | DirtyRat.cs
The most basic, and currently only, monster.

Color Palette | Palette.cs
RGB values for several colors used in display.

Color Assignments | Colors.cs
Colors from Palette are assigned themse, like 'text' or 'background.'

Input Interpretation | Directions.cs
An enum storing values related to directional input.

Doors for the Map | Door.cs
All the data necessary to represent a door on the map.

Recipe Data | Recipe.cs
Everything needed to represent a recipe

My First Recipe | MeatWad.cs
The most basic, and currently only, recipe.

Stairs for the Map | Stairs.cs
All the data necessary to represent stairs on the map.

The Toad's Cafe | ToadCafe.cs
Stores the cafe equipment, recipe book, tables, etc.

The Dungeon Map | ToadMap.cs
All map related data, a lot going on in here.

Map Command Interpreter | CommandSystem.cs
Handles player and monster movement and actions on the map

Load Data From File | Load.cs
Reads data from a selected file and sends it where it needs to go.

User Control of Loading | LoadMenu.cs
Lets the user load save files.

Store Data in a File | Save.cs
Write necessary game state data to a selected flat file.

User Control of Saving | SaveMenu.cs
Lets the user save data.

The Pause Menu | MainMenu.cs
The primary in-game menu.

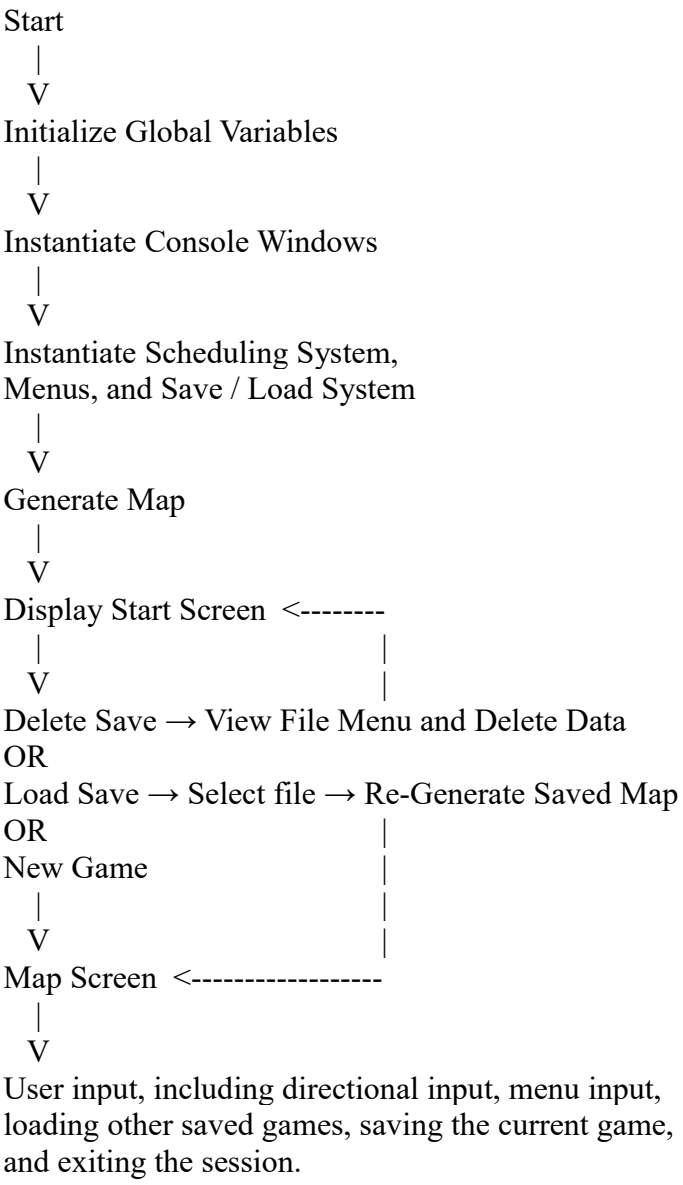
The First Menu | StartScreen.cs
The first menu the user encounters when starting the game.

General Menu Controls | MenuControls.cs
Interprets user input for all menus.

Create New Maps | MapGenerator.cs
All the systems used in creating new maps.

Decide When Each Actor Goes | SchedulingSystem.cs
Stores lists of actors in the order which they will act.

Program Start and End Flow



|
V
EXIT

Summary

The system begins at Game.cs where all the global variables are initialized. Fundamental objects like ToadMap, ToadCafe, the various menu systems, saving and loading systems, the scheduling system, and other objects fundamental to overall operation of the game are all instantiated here.

ToadMap (instantiated in Game.cs) holds all the data related to map features and monsters and is responsible for instantiating the Player.

Once all the necessary objects are instantiated the game is a loop of player input followed by enemy actions as dictated by the scheduling and command systems, with actions reflected in ToadMap. After each action the two methods in Game.cs (OnRootConsoleUpdate() and OnRootConsoleRender()) update the user interface.

Build and Release Process

Testing

Prior to release any new features should be thoroughly tested. The primary approach to testing is to run the game and interact with new features in any way possible. For new menu functions, the functions are simply selected and run from their respective menus and output is checked against expectations.

New actions on the map are somewhat more challenging to fully test due to the unpredictable nature of map generation, but since input on the map screen is restricted to movement the users can usually test new map elements simply by trying to move into or around them. If new actions are added to the map screen they should be tested just like menu functions: attempt the new action in various contexts and check the output against expectation.

New map elements should also be tested in the saving / loading system to ensure they remain the same after saving, shutting down, and reloading.

In general, developers should plan to play the game frequently and ask for feedback from other players. The system is, in some ways, unpredictable by design and so frequent and varied interaction is the best way to discover things going wrong.

Maintenance

The Ravening Toad is a self-contained game which should require only minimal maintenance. Developers should install and run the game on any new platforms (new or updated OS, etc.) on which the game would be expected to run in order to make sure the application remains compatible.

Deployment

The game is updated by compiling the new version of the game and replacing all game files with the new version. Users should take care to make copies of any saved games they don't want to lose as all files will be replaced. Once the update is complete users may copy their saved games back into the 'save' directory. If prompted to replace existing files, click 'yes.'

User Installation Instructions

The Ravening Toad is a completely self-contained game and no special tools or procedures are necessary to install or set up. Simply download and play!

Developer Setup Instructions

These instructions presume use of Microsoft Visual Studio. The 2022 Community edition was used for development up to this point. The .Net Desktop Development Workload must be installed in order to work with this project.

After loading the project file into the IDE, click “manage NuGet Packages” under the 'project' tab. Search for and install both RLNet and RogueSharp from nuget.org.

These are all the tools necessary to work on this project.