Technical documentation

Introduction

About

The SamSer app is a single-player 2D platformer game inspired by Nintendo's Super Mario. It uses a tile-based map system with JSON object serialization and point scoring system.

This game is made by Samuel Serafín as a software project for the Programming in C# and Advanced programming in C# classes at the Charles University in Prague. Credits to all the public resources and assets are in the ../Content/misc/z_credits.txt file.

Installation

The code is written in C# and while it does not necessarily limit the usage to a single platform (since I used multi-platform MonoGame framework), the primary target for this app is Windows.

Install the game by downloading the necessary files (either all files on my GitHub or all the files in ../. directory. Use the pre-built version for easier installation or build the project inside Visual Studio by either selecting a build option, or Release.

The user needs to have the newest version of the necessary Nuget/Myget packages installed:

- `MonoGame.Extended`
- `MonoGame.Extended.Content.Pipeline`
- `MonoGame.Extended.Animations`
- `MonoGame.Extended.Tiled`
- `MonoGame.Framework.DesktopGL`
- `MonoGame.Content.Builder.Task`

Hardware requirements

The game was tested on Windows 11 and Windows 10 machines with at least 4GB of RAM and x86-64 IA. Compiled using the C# compiler. The game needs to have a few MBs of free space for it to be stored locally, a screen (Monitor) and a keyboard and mouse.

The game does not have heavy hardware requirements and can run smoothly on any above specified machine.

Class design

The project has been made with non-strict OOP design, where there are only a few cases of polymorphism usage and mostly each class stands on its own.

Some classes do not respect privacy of other classes, where for example the Game State and Level Controller classes share some of their properties and fields.

Here is a section with summarization for each class type (interfaces and abstract classes will cover their descendants and more info is in the summaries in the code itself):

Game state

This class manages the game session's window and is partially responsible for interactions between entities, objects and the player, score handling, along with any action on the pause and game screen and the serialization system. Basically one of the two classes, which act as the **game engine**.

Main menu state

This class acts as a container for the main menu components, background sprite and the title text. Initializes the **level controller** and is responsible for updating and offsetting the components, along with assigning their respective events.

Player

This class represents the player and its **avatar**. It manages player rendering, collision with rectangles, player velocity and takes care of the player's respawning/health. Allows for access to global constants in the **Game Utilities** class. Initiates and controls the animation automaton (**SpriteStateProcessor**), and sets the necessary player specific constants for the proper animation.

IEntity (Enemies, Coins)

This class represents the base for all the **game entities** – collectibles and enemies. This interface declares common collision, physics methods and statically (in a static method) holds respective texture sheet. Each entity allows for both JSON and regular construction.

BaseEnemy - defines an abstract version of a movable entity which, upon collision with player, triggers the death event. These entities cannot be killed or picked up and act purely as an obstacle for the player. This class declares a common physics/collision system for the self-moving entities, which is similar to the player collision system. There are 2 implemented enemies – **zombie** and **flame**, each with its specific texture.

Coin - defines a **collectable** entity, whose type is differentiated in the **CoinType** enumeration. The game defines 3 types of Coin: **Gold** – 3 points, **Silver** – 2 points, **Copper** – 1 point. Upon collision with player, this entity is destroyed and the respective score is increased in the Game State using the Level Controller class.

Level Controller + Level

Level Controller - is a class that takes care of Level construction using either JSON deserialization or regular pre-set constructor. It is also responsible for entity – terrain collision and partially the player-entity collision, due to it having first-hand access to the current level and its index in the level list. Sets the new level upon completion, invokes the Win event. Deserializes the entities (specifically their initial level-specific states in the level). Updates the entities and takes care of their lifespan (with access to the level class)

Level - holds the TiledMap instance and its TiledMapRenderer. Contains all the level specific entities, takes care of the level specific music and draws/updates the TiledMap according to the Camera view matrix. Basically the representation of a single level in the game along with its specific properties and fields + physical bounds.

Component

This abstract class represents a base version of GUI object in the Game Session screen or the Main menu. Allows for both text-only and clickable objects on the screen.

Non-interactive (Text-only) – The text area and counter components both represent non-interactive objects on the screen. Text area (TextComponent) basically represents raw strings in a specific format, font and screen position(which are passed in the constructor). Counter is implemented generically with a specific updater event allowing for real-time updates of a count/text/any sensibly printable object with possibility of event handling outside of the class. It uses the base constructor for the text area and adds additional logic.

Interactive – Slider, Button and Serial button are all interactive components, which trigger an event upon holding and moving or clicking respectively. All of these contain an event, which will come in handy with the trigger logic. Slider has a built-in slider container and a scroll. It calculates the value of its value according to the scroll position with respect to the container using a basic math formula. The button only triggers events upon clicking and has a built-in effect upon hovering. Serial button is inheriting from button and only adds context to the button (makes the event have a parameter, as opposed to the button event, which is only invoked with empty arguments)

Camera

This class represents everything currently on the screen. At first it is unlocked and the player has visually free-roam ability, but as soon as the player reaches the centre of the screen, the camera gets locked. It basically shows the currently-rendered objects and makes the game engine only update the necessary ones. It uses matrix transformation to follow an IFollowable object (Player)

Animation

This class offers mostly graphic service. The collection of spritesheet frames represents a simple animation, which is animated (interchanged) every game tick – calculated with pre-set FPS (10), with specific parameters (SpriteEffect or scale). Every entity has a set of animations, which are added into an Animation Automaton (SpriteStateProcessor) and this allows for contextual animation changes. Since I decided to allow only in-row or in-column sprite rotations, there are 2 classes inheriting from the BaseAnimation class (Horizontal and Vertical animation) – each with their specific animation handling. SpriteStateProcessor is basically a data structure storing the respective entity animations and setting the current ones (basically a finite automaton).

Game State

There are only 3 defined game states: Main menu, Game session (game state) and End screen. Main menu and game session were explained earlier and End screen is basically the screen which shows up after clearing all of the levels and invoking the Win event from the Level Controller class (which has a method that checks whether the index surpasses the length of the level list array and triggers upon being so)

Levels

Levels are .tmx TileMap files made in the Tiled app, they need a texture sheet in the form of .png file and a tileset (.tsx file). Each level has to contain 32x32 pixel textures and these layers: WaterLayer (TileLayer), DeadlyLayer(TileLayer), Platforms(ObjectLayer), Obstacles(ObjectLayer), TileLayer1(TileLayer) – solid blocks, TileLayer2(TileLayer) – decorations. Each level has its own initializing file in the format "Level{level-number}Data.json" located in the LevelEntityData directory. All that remains is to add the level files into the content pipeline (More information on that in the official MonoGame documentation) Using this, there is a support for user-made levels. The meanings behind the different tiles in the level can be found in the manual.

Serialization

The map, player, entities and their properties, score and assets can all be serialized into json files and deserialized back into a code form. This is all done using Newtonsoft.Json library, since it allows for MonoGame object handling.

The main serialization is happening in the GameState class, where I pass the necessary player and entity properties into a their respective specific serializable classes (PlayerData, EntityInfo, GameData) and serialize them into a save file in the 4 possible save slots inside the Saves directory, where each save file name is in the format "save{1-4}", if there are already 4 slots taken, the next save will be put in the 4th slot and the earliest save file will be deleted.

Deserialization happens in two separate occasions - first in LevelController, where the initialization of a LevelData from a file occurs and second inside the MainMenuState class, where we can load a game instance from the save file, by basically reversing the serialization process for the saves and initializing the deserializable objects with their JSON constructors. The same will essentially happen with the level data in the LevelController class, but the JSON file format is slightly different due to loading of the texture names and initial positions.

Physics

The player is controlled by WASD keys. He will be blocked by solid tiles or platforms and the 2D map bounds. There are also other collisions, which can be read about in the user manual. There are 2 types of physics environments (Air and Water, where each means different physics for the player)

I decided to update the entity and player in a check then move manner. This allows me for pre-emptive terrain collision, but also has a few downsides like inconsistency of specific collisions (for example top and bottom collisions need different parameters). Horizontal movement is fairly simple, with no acceleration and only constant velocity (specific for environment). Vertical movement in air uses gravity acceleration, which is set to 0.1 pixels a tick and jump impulses(which are only for the player), which eventually decay due to it being constant and gravity quickly over chasing them. This makes the player create the illusion of jumping in air. In water, the vertical movement for the player is constant and W/S keys change the direction (UP/DOWN). Every entity automatically stops downward vertical acceleration upon collision with a platform or reverses upward vertical acceleration . The vertical collision brings many problems with on-platform movement registration, so had to be implemented accepted ranges for OnPlatform entity state. More about player-entity collision can be found in the user manual.

Every entity has their specific physics for convenience.

Note

For UI information, there is information about it in the user manual.

What could be improved?

Better class design: the situation with GameState and LevelController classes being both partially responsible for the same thing could be improved, but would require an overhaul of the entire code body.

Sound bug: Main menu state initializes the Level Controller, so it starts to play level 0 music right after starting the app, which is avoidable, but would also require code overhaul.

More entities, sounds and more gameplay: there is a support for more entities due to the content folder having even more available textures (that are still not included in the game), but gameplay would require even more additional structures.

Physics system is not the best and needs support for more than 90 degrees collision.

Config file could be added, but could not be added in time.

I spent a large amount of time trying to make a level editor (unsuccessfully): would take me probably another month.

Entity movement and behaviour is very simple, there is a possibility to implement more entity functionalities.

Using a game engine instead of framework would make the timeframe of creating the game much shorter, but of course the code size would be a problem.