INM379 Computer Games Architecture Report

Vignesh Gokul Ramanan Asokan

Part 1

For loading the level assets, a data-driven design approach was accomplished. The tiles in the game are loaded from a .txt file based on the character that is read. Each character is assigned to an asset that is then initialized in Level.cs. The Loader.cs reads the .txt file and returns it as a string. So, the developer does not have to go through the trouble of coding the level design each time. This was referred with Lab 02 code.

The player is controlled with an event-driven architecture. First each key is assigned to an GameAction with the use of Dictionary and when the key is pressed, or released the respective action is invoked. Referred to Lab 03.

Code Snippet:

```
private void InitializeKeyBindings()
    {
        commandManager = new CommandManager();
        commandManager.AddKeyboardBinding(Keys.Escape, StopGame);
        //commandManager.AddKeyboardBinding(Keys.C, EnableCameraSpring);
        commandManager.AddKeyboardBinding(Keys.W, level.Player.Jump);
        commandManager.AddKeyboardBinding(Keys.A, level.Player.TurnLeft);
        commandManager.AddKeyboardBinding(Keys.D, level.Player.TurnRight);
    }
```

There are 3 types of collision detection in the game, Passable, Impassable, and Platform. Passable collision doesn't not affect the motion of the player. Impassable does not allow the player to go through it in any way. Finally, the platform collision allows the player to jump through the platform only from the sides and cannot go through from above.

The player and NPC characters both have animations that are played with the AnimationPlayer.cs and Animation.cs respectively.

Part 2

The GameInfo class is a Singleton that uses the data driven architecture to store all the game variables values that can later be used. This information is saved in info.xml file and loaded using Loader.cs. The info.xml contains the all the values for Enemy, Power Ups and Collectible variables. This help when debugging as all the values are stored in an xml file and the code does not have to be compiled each time a value is changed.

The power ups, collectables, and the enemy has collision detection responses that are event listeners that is triggered on collision that the object is removed if it's a power up or a collectible. In the case of enemy the player is reset back to it start position.

The scoring system uses an event listener architecture since multiple events have to be called in different cases of a score is attained. For example, each time a fish is collected by the player it simply calls the event to increment the score, but in the case of a power up is collected the score is incremented followed by the activation function of the power up is invoked.

Code Snippet:

```
public event EventHandler<ScoreEventArgs> FishCollected = delegate { };

    public event EventHandler<ScoreEventArgs> EnemyKilled = delegate { };

    public event EventHandler<ScoreEventArgs> PowerUpCollected = delegate { };

public ScoreManager()

   {

     FishCollected += this.incrementScore;

     PowerUpCollected += this.incrementScore;

     PowerUpCollected += this.ActivatePowerUp;

   }
```

The High-score table is implemented using serialisation. Every time the game is closed the current score checks with the table off high scores and replaces if it's greater than one of the high scores. The high scores are saved as an array of size in a json file is the bin folder of the game in the following format:

"SaveData.json"

```
{"Level":-1,"highScores":[4270,1230,650,640,65]}
```

For loading and Saving data:

```
public void LoadSavedData()

   {

     var filecontent = File.ReadAllText("Savedata.json");

     SaveDataInfo.Instance = JsonSerializer.Deserialize<SaveDataInfo>(filecontent);

   }

   public void SaveData()

   {

     string Serializedtext = JsonSerializer.Serialize<SaveDataInfo>(SaveDataInfo.Instance);

     Trace.WriteLine(Serializedtext);

     File.WriteAllText("Savedata.json", Serializedtext);
```

```
    }
```

The saved data is then deserialized into an instance of SaveDataInfo Singleton. This also saves the current level the player is in and loads that level next time the game is launched.

Part 3

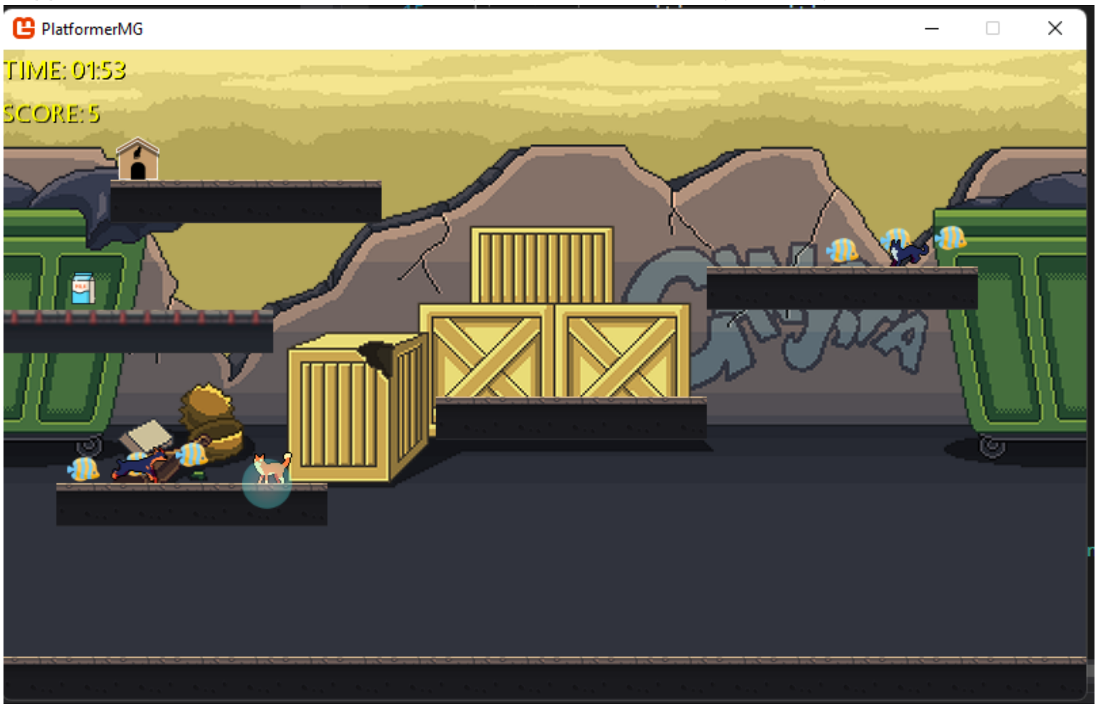
The entire game has three states, Menu, GamePlay and GameOver, that are switched between respectively. The game starts off in the Menu state and pressing to continue when Play is selected will switch the game to the GamePlay state. When all the levels are completed, the game switches the GameOver state and can be restarted by pressing “R”. All the conditions are checked in PlatformerMG.cs and switchState() is called.

Code Snippet:

```
protected override void Update(GameTime gameTime)
    {
      if(commandManager != null)
        commandManager.Update();
      // Handle polling for our input and handling high-level input
      HandleInput();
      switch (CurrentGameState)
      {
        case GameState.Menu:
          {
            menuSelected option = menu.Update(gameTime);
            if (option == menuSelected.Play)
              SwitchState(GameState.GamePlay);
            else if (option == menuSelected.Exit)
              Exit();
            break;
          }
        case GameState.GamePlay:
          // update our level, passing down the GameTime along with all of our input states
          level.Update(gameTime, keyboardState, gamePadState, touchState,
                 accelerometerState, Window.CurrentOrientation);
          break;
        case GameState.GameOver:
          if (gameover.Update(gameTime))
```

```
                {
                    levelIndex = -1;
                    SwitchState(GameState.GamePlay);
                }
                break;
            default:
                break;
        }
        base.Update(gameTime);
    }
```

There two types of power ups, shield, and speed power up. Shield power up mix the player immune two damage from the enemies. The speed power up makes the player move around faster. Both the power ups last for about 5 seconds each using a timer, this value can be changed in info.xml. The powerups implementations uses hierarchy where PowerUp.cs is the base class and Shield.cs and SpeedPowerUp.cs is derived from it. When the power up is collected even listener is triggered which increments the score by 5 and activates the power up.



The game has two types of NPC opponents, the Dog and the Cat. both the opponents are controlled by FSM with the states, Patrol, Chase, and Idle. The cat opponent switches between Patrol and Idle on its own and is not affected by the player's position. The Dog opponent is responsive to the position of the player as it will chase the player if its in a certain range.

Code Snippet:

```
if (type == EnemyType.Dog && Vector2.Distance(player.Position, position) < 150)
        switchState(EnemyStates.Chase);
```

Overall game objective is to reach the player cat's home set in different positions in the level with being attacked by the opponents and collect as many food on the way for an higher score. Each level has a timer that is indicated in the HUD. Finished the levels faster will increase the score obtained for each level. The The level gets more difficult as it goes.

Part 4:

After doing some research, found that the preferred way to implement networking in the MonoGame engine is using the Lidgren.Network framework. Lidgren.Network is a networking library for .net framework which uses a single udp socket to provide a simple API for connecting a client to a server, reading and sending messages. The Lidgren.Network framework achieves networking via sending data across clients and host in the form of messages. This framework has 2 types of messages, Library messages telling you things like a peer has connected or diagnostics messages (warnings, errors) when unexpected things happen, and Data messages which is data sent from a remote (connected or unconnected) peer.

First a connection is made to the NetServer is established by *NetPeerConfiguration config = new NetPeerConfiguration("MyExampleName"); config.Port = 14242;* There should be an Async function to wait for all the players to join for the game to start. Hence the player will be sent to lobby menu to wait. And the following data has be synced with all the clients in the game:

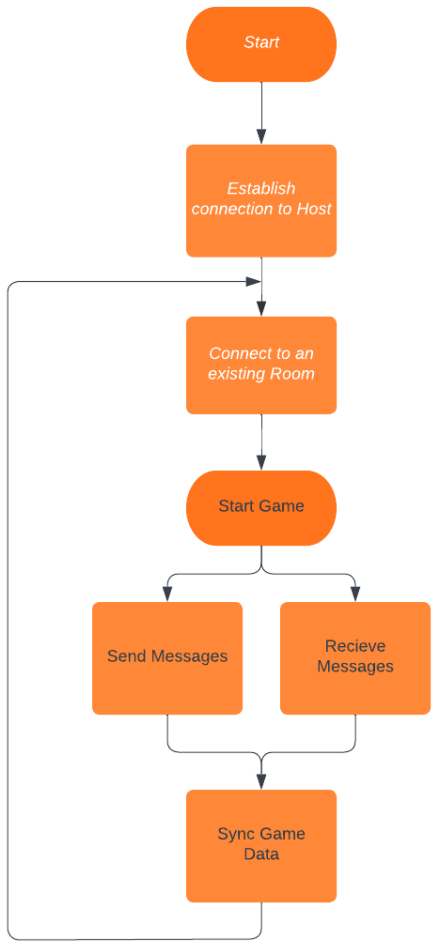
- Position of all the players
- Position of all the NPC opponents
- Score of each player
- Position and if collected information of all the Collectibles in the level
- Position and the status of the power ups

All the above-mentioned game data will be received and sent to and from clients in the game for each frame or when their values changed. The messages are usually of the type string. Hence the data will have to serialized from its sent and deserialized once it's received. The data is then processed and used appropriately. Players will have to wait for all the players to reach the hone position for the next level to be loaded and once the game is over the player will be sent back to the lobby where they can restart the game or quit.

Code snippet for receiving messages:

NetIncomingMessage msg; while ((msg = server.ReadMessage()) != null) { switch (msg.MessageType) { case NetIncomingMessageType.VerboseDebugMessage: case NetIncomingMessageType.DebugMessage: case NetIncomingMessageType.WarningMessage: case NetIncomingMessageType.ErrorMessage: Console.WriteLine(msg.ReadString()); break; default: Console.WriteLine("Unhandled type: " + msg.MessageType); break; } server.Recycle(msg); }

Game flow adding networking to the game:



References:

PlatformerMG project from Lab 02

Assets:

BG : https://craftpix.net/freebies/free-pixel-art-street-2d-backgrounds/ (10 March 2022)

Player: https://craftpix.net/freebies/free-street-animal-pixel-art-asset-pack/ (10 March 2022)

Shield: https://www.flaticon.com/premium-icon/award_1959460?term=shield&page=1&position=34&page=1&position=34&related_id=1959460&origin=search# (10 May 2022)

milk: https://www.flaticon.com/premium-icon/milk_869664?term=milk&page=1&position=1&page=1&position=1&related_id=869664&origin=search# (10 May 2022)