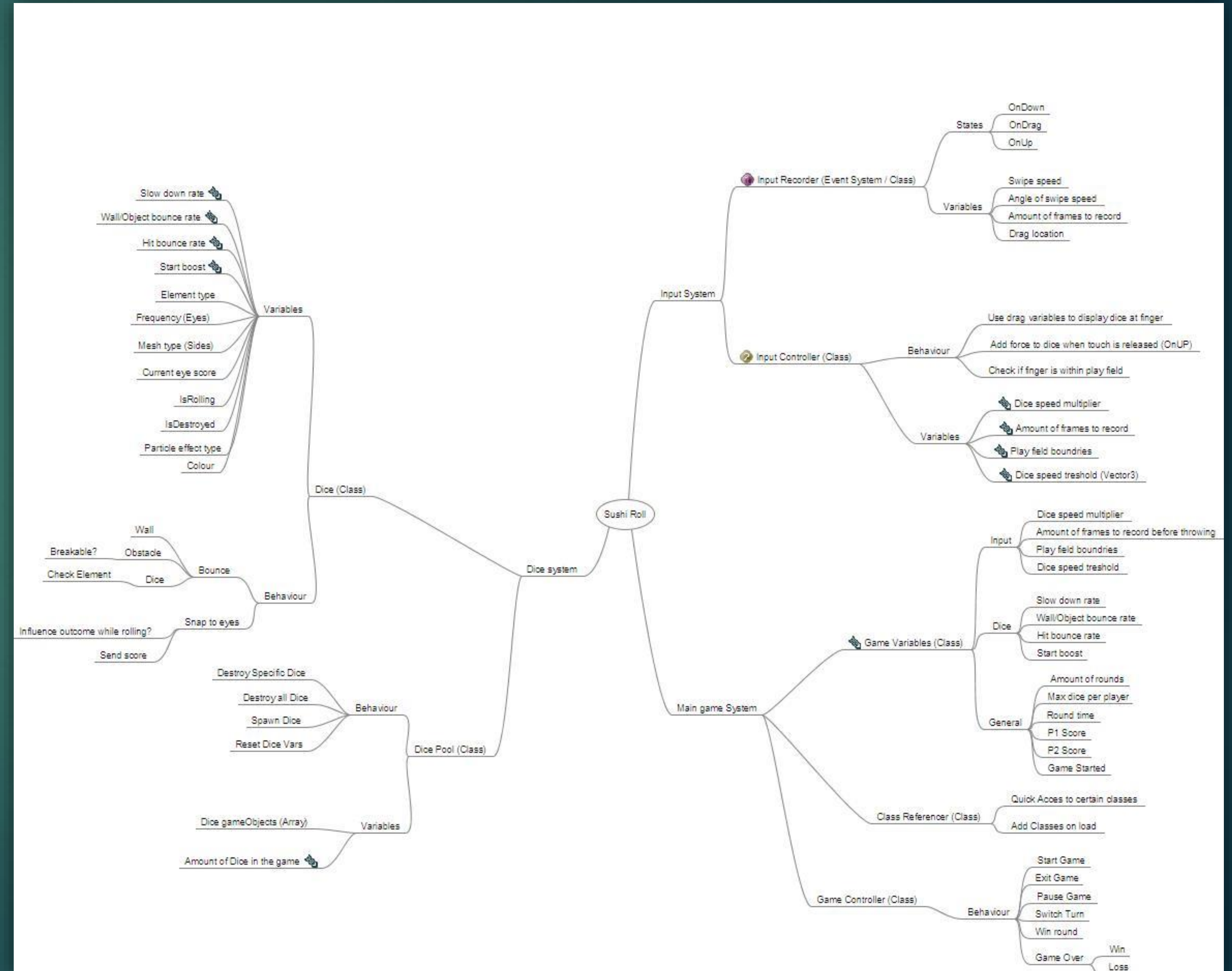
SUSHI ROLL MOBILE GAME – ALEX MEESTERS (142783)

# Planning out the systems for the game with a mind map

My plan was to divide the game into Three separate systems:

- Dice system
- Input system
- Game system

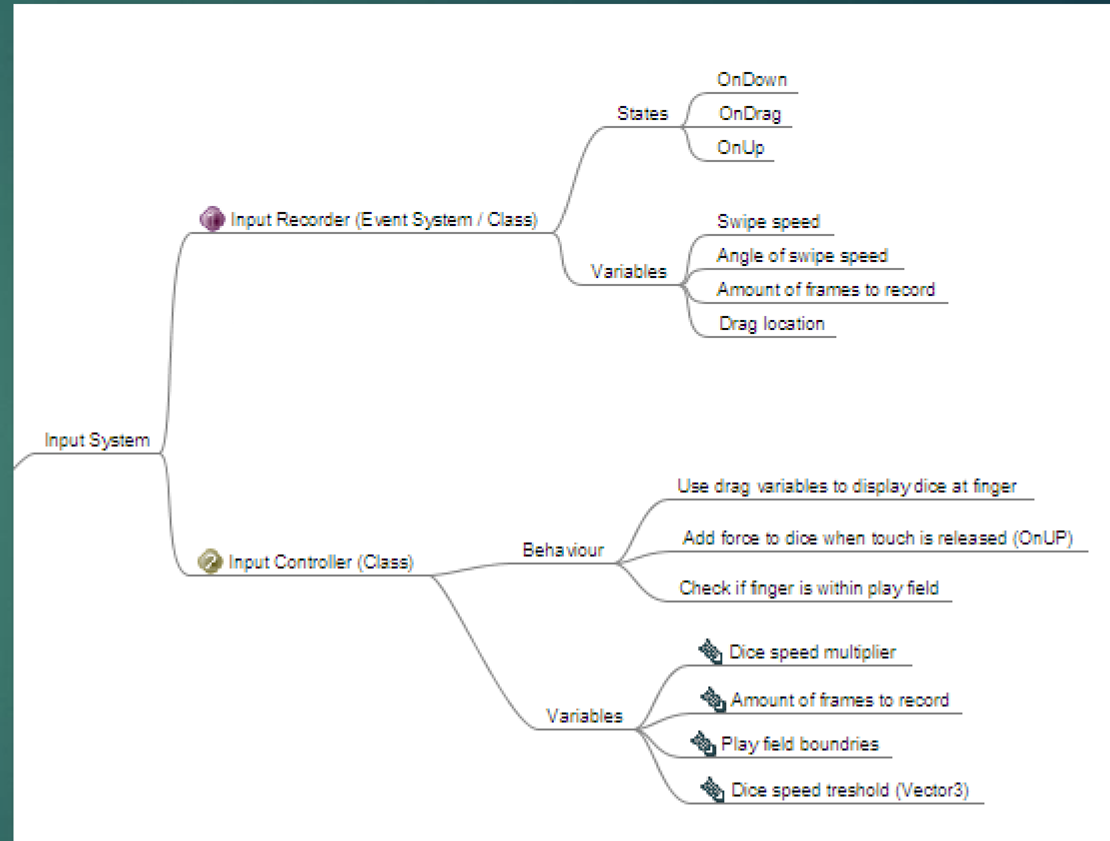


# Input system

The input system is the most important system for the core gameplay of the game. Because the game is a physics based game, where direction and velocity play a very big role.

I have separated this system into two classes:
- Input Recorder (input)
- Input Controller (output)

The reason for separating the classes is to maintain overview of the project.




Created with FreeMind software

# Dice system

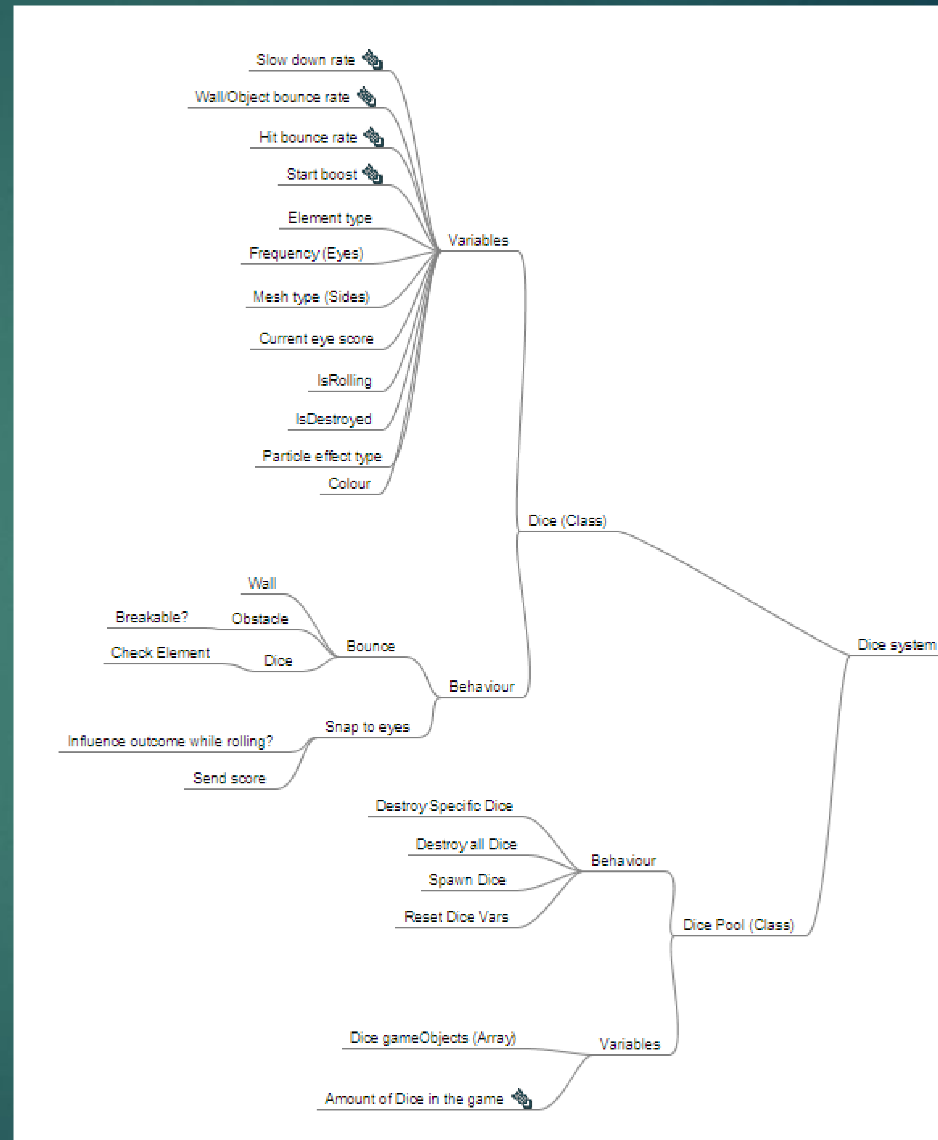
**This system is for management of the dice in the game. And controlling the behaviour of dice within the playing field.**

This is divided into two classes:
- Dice
- Dice Pool

**Dice**
Contains all needed information for interaction with other dice.

Also contains information about how it bounces towards other objects.

Snapping to eye behaviour is hardest to accomplish properly.




Created with FreeMind

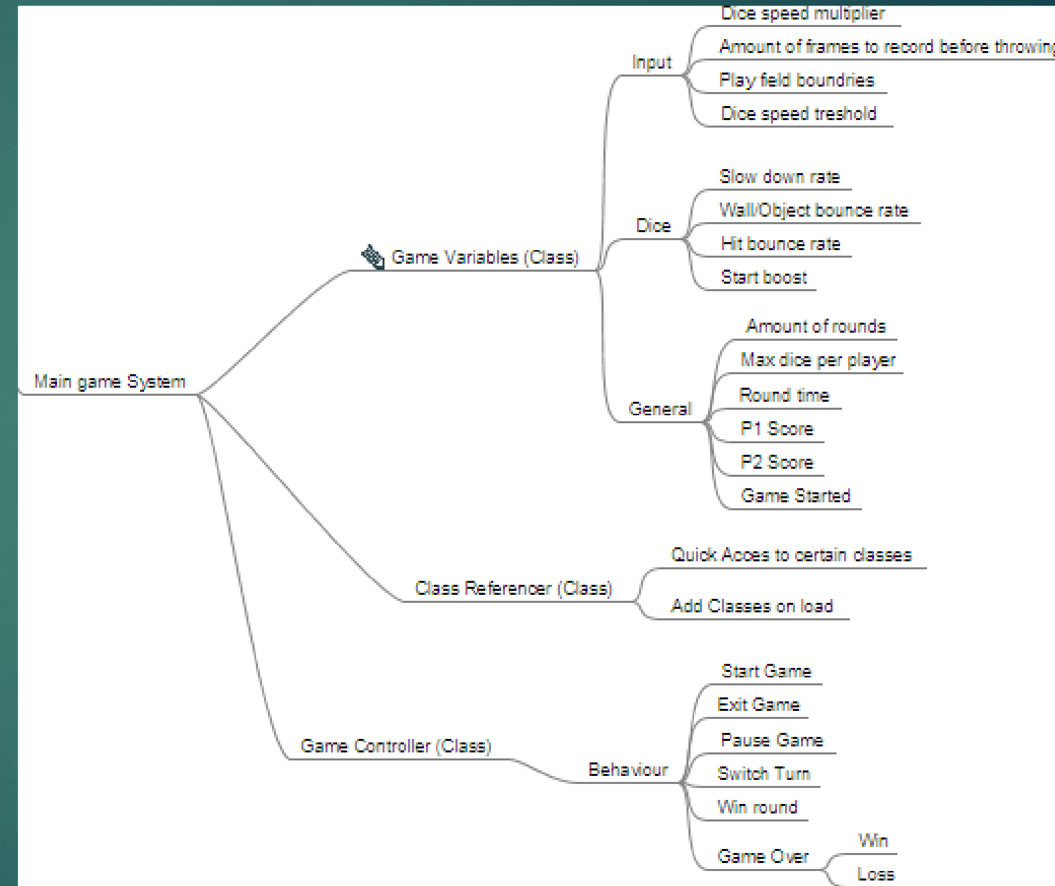
# Game system

**This system is for control of variables and game states.**

**Class Referencer:** gives easy access to all high level classes. By making a static reference towards them.

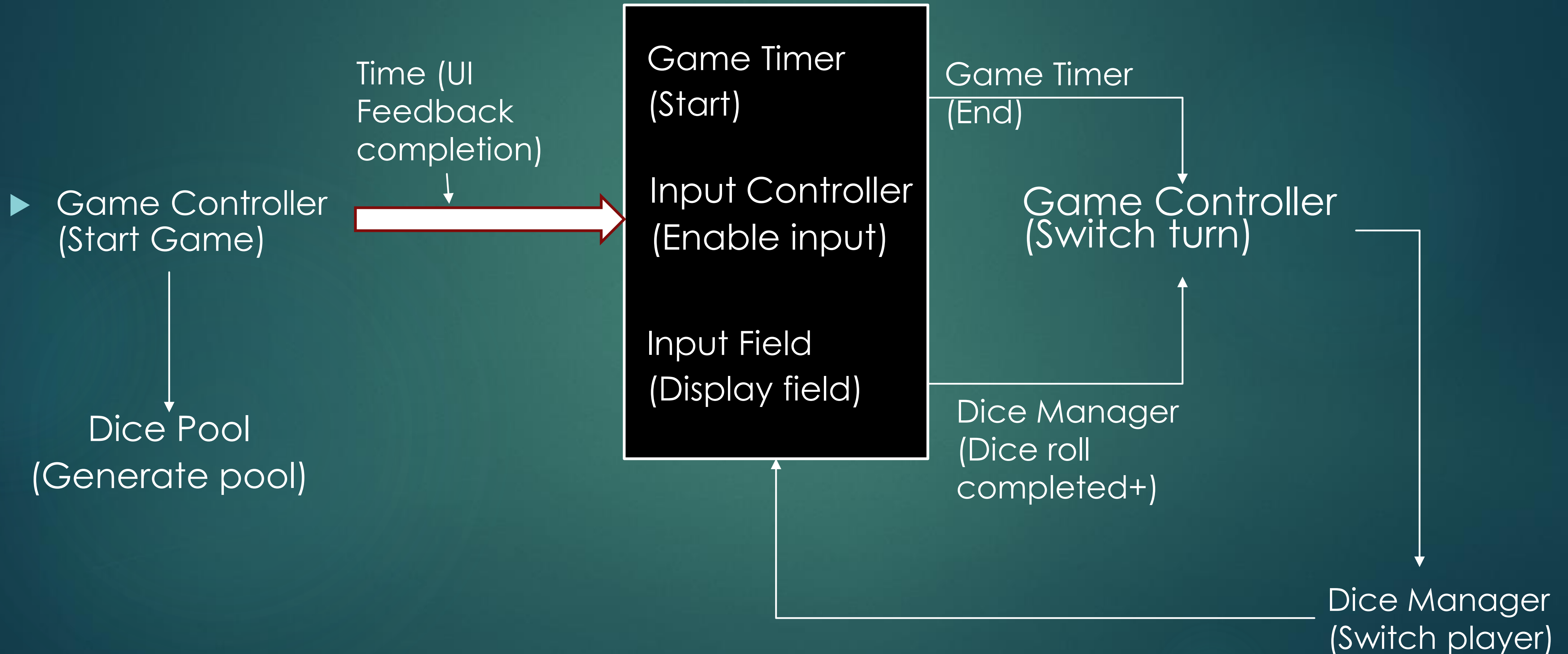
This is a singleton pattern, I have chosen to use it to speed up the development process. To make referencing of classes and variables to go as quickly as possible.

**Game Variables:** Stores dynamic and read-only variables. Storing all data within one class is for simplicity. Currently all read-only data is written by a scriptableobject.

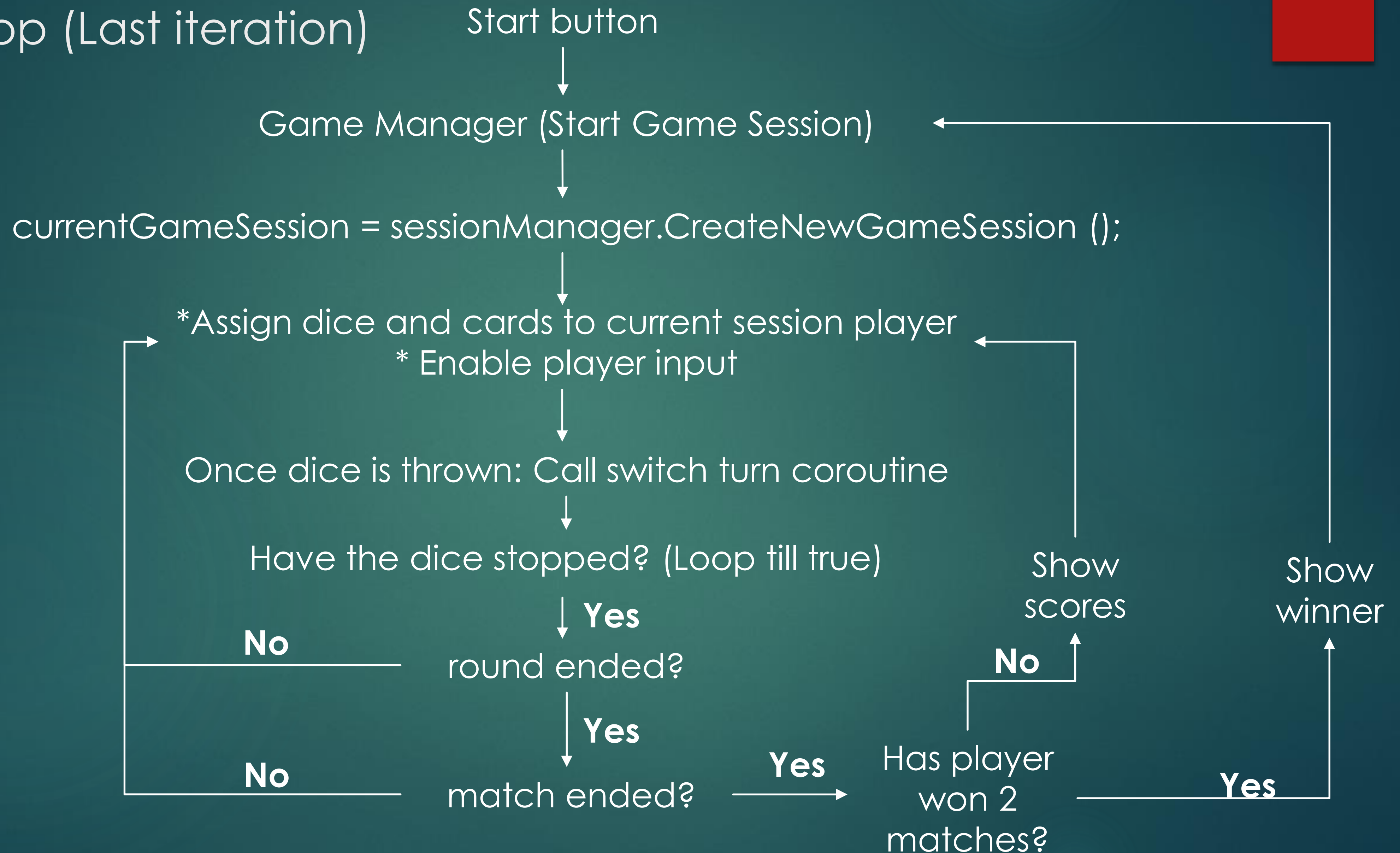


Created with FreeMind

# Code loop (First iteration)



# Code loop (Last iteration)



## Start Game Session

After the start button is called
It will use the StartGameSession function for the sessionManager.

This will create a new instance of GameSession.

A game session contains data such as:

- Amount of moving dice
- Match and round score
- Thrown dice count
- Round count
- Turn count
- Match count

All of this data is used to determine if there should be another game or another round. If all dice have stopped moving.

```
public void StartGameSession ()
{
    if (!hasDoneStartup) {
        StartupBehaviour ();
        hasDoneStartup = true;
    }
    soundManager.FadeInMusic (0.25f);
    currentGameSession = sessionManager.CreateNewGameSession ();

    ClearAssets ();
    StartGameMatch ();

}

void StartupBehaviour ()
{
    soundManager.Playmusic (0, 0.25f);
    Screen.sleepTimeout = SleepTimeout.NeverSleep;
    diceManager.CreateDicePool (gameVariables.MaxDicePerPlayer * 2);
}
```

```
void StartGameMatch ()
{
    playerFieldBlockadeManager.ResetBlockers ();
    soundManager.PlaySoundEffect (soundManager.soundDatabase.scGameStart, 1, 1);

    uiManager.AddNotification ("Round " + currentGameSession.round);
    uiManager.AddNotification (PlayerToColor.ReturnPlayerColorName (currentGameSession.activePlayer) + " Begins");
    uiManager.ShowNotifications ();

    uiManager.UpdateMatchScore ();
    uiManager.UpdateRoundScore ();

    EnableInput();

    System.GC.Collect ();
}

void EnableInput ()
{
    inputFieldOutput.AssignDice ();
    uiCardManager.SetCardsToPlayer (currentGameSession.activePlayer - 1);
    uiCardManager.ShowAllCards (true);
}
```

## Switch turn coroutine

When you throw a dice, the amount of moving dice will be increased, and decreased once the dice rigid body has gone to sleep.

.IsSleeping() can be a risk to use, because there can be situations when the dice falls through the floor, or it gets stuck in a wall. (Happens rarely, depends on how designers make the level)

One way of preventing it is to override the check by making a timer that checks the dice based on its positional difference. If it is still within the same radius after 5 seconds, then it is stuck within a object.

From DiceBehaviour.cs

```
IEnumerator Getfinalscore ()
{
    yield return new WaitForSeconds (0.05f);

    if (DiceRigidbody.IsSleeping ()) {
        Dicescore = Eyevalues [GetHighestPoint ()];
        HasBeenThrown = true;

        sessionManager.GetCurrentSession ().movingDice --;
    } else
        StartCoroutine (Getfinalscore ());
}
```

From GameManager.cs

```
///////////////////////////////////////////////////////////////
/// Waits for all the dice to stop moving /////////
///////////////////////////////////////////////////////////////

public void SwitchturnCoroutine ()
{
    StartCoroutine (WaitForSwitchTurn ());
}

IEnumerator WaitForSwitchTurn ()
{
    if (currentGameSession.movingDice != 0) {
        yield return new WaitForSeconds (0.15f);
        StartCoroutine (WaitForSwitchTurn ());
    } else {
        if (IsMatchEnded ()) {
            PrepareForNextMatch ();
        } else
            SwitchTurns ();
    }
}
```

# Round / Match ended

The scoreManager gets the score from the dice on the playing field, and applies Multipliers to them, afterwards the score manager adds the score to the current game session.

There are functions within the game session to determine if a session is done

ReturnMatchWinner () returns the player with highest score from the 3 rounds.

IsSessionDone () returns if a player has won two times.

ReturnSessionWinner () returns the player that has won 2 matches, -1 if this is not the case.

```
public void PrepareForNextMatch ()
{
    soundManager.PlaySoundEffect (soundManager.soundDatabase.scEndMatch, 1, 1);

    scoreManager.SetRoundScore ();
    uiManager.UpdateRoundScore ();

    int MatchWinner = currentGameSession.ReturnMatchWinner ();
    currentGameSession.EndMatch (MatchWinner);
    currentGameSession.ResetRoundData ();

    if (currentGameSession.IsSessionDone ()) {
        int sessionWinnter = currentGameSession.ReturnSessionWinner ();
        if (sessionWinnter != -1) {
            uiManager.UpdateMatchScore ();
            uiManager.UpdateRoundScore ();
            soundManager.FadeOutMusic ();
            StartCoroutine (uiManager.ShowVictoryScreenAfterTime (PlayerToColor.ReturnPlayerColorName (sessionWinnter + 1) + " Won!", 1.5f));

        } else
            StartCoroutine (StartNewMatch ());
    } else
        StartCoroutine (StartNewMatch ());
}
```

## Why did I use this loop

The current loop can be made compatible with switching of game sessions.

This is because all the required data for the other classes is accessed from the game session class.

Having multiple game sessions was not something that was required for the prototype, but it seemed like a comprehensive system to understand.

Also the current system does not use any Update () loop. Only coroutines, which can be made to run less frequently.

## What makes this approach bad

There are quite a lot of classes tapping into the game session class, which can make it harder to debug.

## What makes this approach good

The game manager class is a good starting point to learn how the code is structured.

## What did I learn?

During this block I had to do a lot of branching and renaming of classes. Doing this gave me more insight of how this improves the general structure of the code.
I also became a lot better with creating a good folder structure to know where certain classes/behaviour is found.