# Menu System TDD

## V\_00.01

## Author: Finn Varvell

# Introduction

## Rationale

This TDD is for making a menu system, and will include a main menu, options menu, pause menu, loading screen and the code that allows the user to navigate through the menus and use them. It will also include custom elements such as a splash screen, a game icon and a custom cursor.

## Background

Menu systems are in every game and are what lets users navigate through the game and interact with the different parts. Some of the most important parts of games such as saving, changing settings, and exiting the game all require menu components to let the user control them.

## Terminology

UI – User interface

GUI – Graphical user interface

TDD – Technical design document

UML – Unified Modeling language

API – Application Program Interface

## Non-Goals

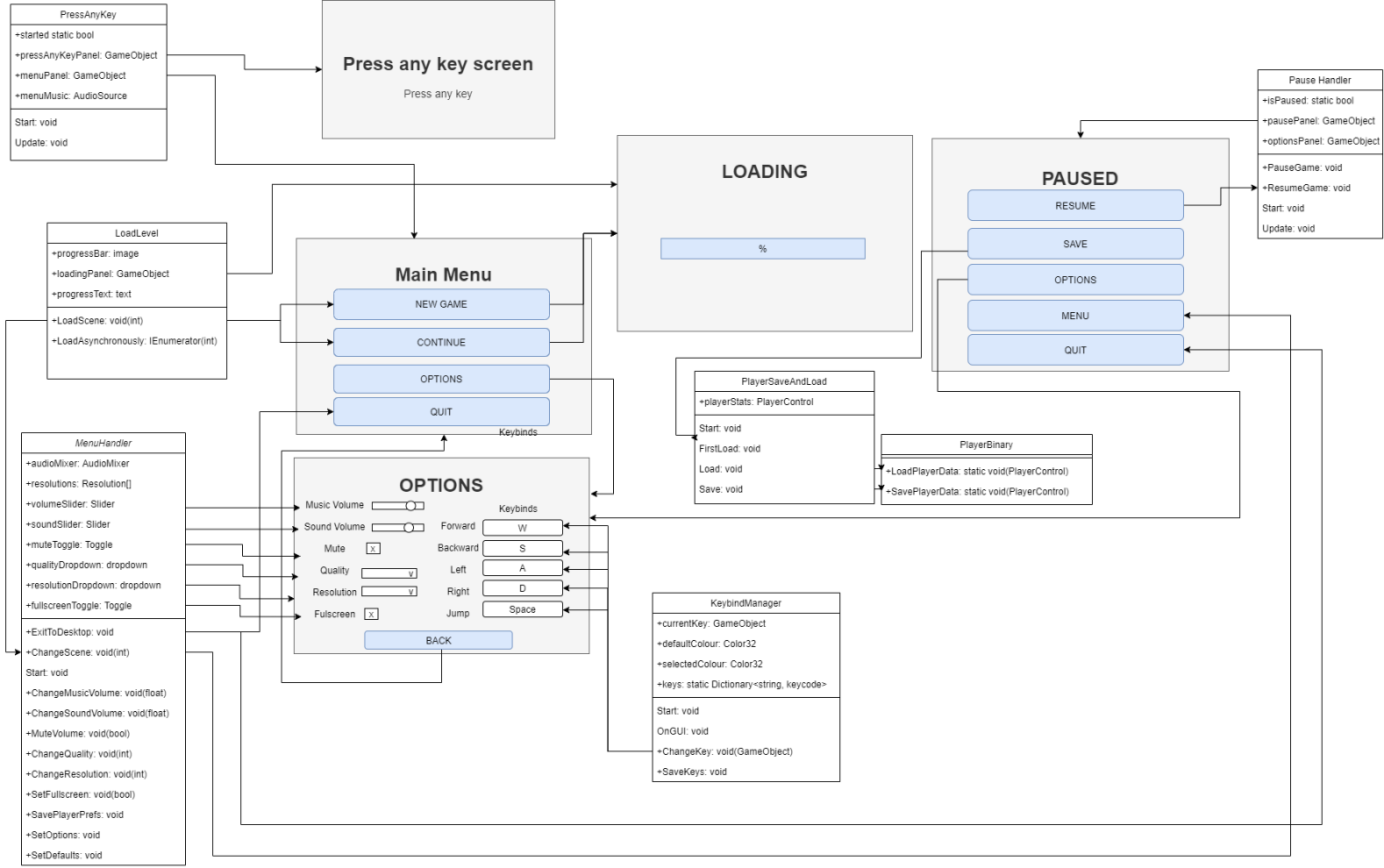
- Toggle for muting volume

- Animating the menu buttons

## Proposed Design

* Custom cursor, splash screen and exe icon
* Press any key screen at start
* Main Menu
  + New game
    - Resets everything saved to its default and then plays
  + Continue
    - Loads using saved data, both continue and new game go to loading screen
  + Options
    - Opens the options menu and sets all the values to saved player preferences
  + Quit
    - Exits to desktop, quits play mode in editor
* Options Menu
  + Sliders for music and sound volume
    - Changes separate music and sound parameters
  + Mute all volume toggle
    - Changes master to -80, silencing music and sound
  + Quality dropdown
    - Can change between low, medium, and high
  + Resolution dropdown
    - Gets resolutions on start to list
  + Fullscreen toggle
  + Key bindings
    - Forward, backward, left, right and jump can be changed to any key
  + Back button
    - Saves everything to player preferences
* Pause menu in game, opened with escape and freezes time/unlocks and shows cursor
  + Resume
    - Unfreezes time and locks/hides the cursor
  + Save
  + Options
    - Opens up a similar options menu to the one on the main menu that uses the same player preferences
  + Back to menu
  + Quit

# System Architecture

If the design consists of a collaboration between multiple large-scale components, list those components here — or better, include a diagram [UML].

## Data types

### Press Any Key

* Static bool for if the game has been started
  + Bools can be set to true or false, and since it is static it will only have a single value that can be accessed by anything for whether the game has started and is off the press any key screen
* GameObjects for the menu and press any key panels
  + GameObjects are objects that can store components, which are what makes the game work. By accessing the panels as game objects, they can be turned off and on to let the player move between screens.

### MenuHandler

* UI elements for the options menu, including Sliders, Dropdowns, and Toggles
  + Sliders can be moved up and down by the player, and changing it changes a value (in the case of the options menu, the volume of the music or sound) from a range, -80 to 20 for volumes where 0 is default.
  + Dropdowns are set to a value, and the player can click on them and change the value from a list of possible options. Dropdowns will be used for the quality (low, medium or high) and the resolution.
  + Toggles can be turned on or off by the player. Toggles will be used for mute and fullscreen.
  + References to the UI elements allow the script to interact with the elements in the menu so that they can be changed based on player preferences.

### KeybindManager

* Static dictionary that holds the controls and which keys are set to them.
  + A dictionary is a collection of keys and values for each key. The keys will be the strings forward, backward, left, right, jump, crouch and sprint and each one will contain a keycode that can be changed in the options menu with its respective button.
  + Because the dictionary is static, there is only one of it and it can be accessed from the menu and the movement script with the same values.
* GameObject for currentkey and a color32 for selected
  + When a key button is clicked on, it will need to set a variable to hold its value so the script knows which key is being changed. A GameObject reference allows the script to reference the button object.
  + A color32 holds a colour value, which will be the colour that the button changes to to show the user that they are changing it.

## Data Model

* Variables for options carried over between scenes, including:
  + Screen resolution
  + Screen quality
  + Fullscreen
  + Volume for sound and music
  + Keybinds (which controls are assigned to which keys)
* Player preferences store all options and load them when the game is opened so the player doesn’t have to change them every time
* Save and load – saving the game stores things like the players position and name in a file to be loaded when the game is reopened

## Interface/API Definitions

Describe how the various components talk to each other. URL and the format of the data and parameters used.

Libraries used:

UnityEngine.SceneManagement – Allows SceneManager to be used, which can call functions such as LoadScene, which changes the scene to another one based on its parameter.

UnityEngine.UI – Allows references to the Unity UI such as the text, buttons and dropdowns/sliders/toggles. All the functions of them can then be used, and the values can be changed.

UnityEngine.Audio – The audio mixer and its parameters can be references and changed in the options menu using the audio library.

## Impact

Describe the potential impacts of the design on overall performance, security, and other aspects of the system.

* Performance – doing things to reduce the performance cost of running the game
  + Running events that only need to be run once a single time rather than every frame,
  + Changing the scene with LoadSceneAsync on the loading screen allows the loading to be done in the background while the bar fills up and improves performance
  + Coroutines to wait for responses over multiple frames
* Security
  + Public variables vs private variables – Public can be accessed outside of the class and changed from the editor and private can only be accessed from within the class.
  + Saving – Using PlayerPrefs to save saves the data to an easy place to access and change, so it’s fine for things like options. However, when saving character data like position and level binary should be used because it can save to any location with any name, making it harder to find, and it is also more difficult to edit.

## Risks

If there are any risks or unknowns, list them here. Also, if there is additional research to be done, mention that as well.

* I find binary saving a bit confusing

## Alternatives

If there are other potential solutions which were considered and rejected, list them here, as well as the reason why they were not chosen.

* Using PlayerPrefs instead of binary is an alternative but because it is easy to access and change it isn’t good for save data and learning and trying to understand binary saving would be a better alternative

# System Architecture

## Testing

Show progress, Error reports and explain fixes you used.

Show error reports

Explain fixes