**Quick Start Guide: Basic Design and Development Principles**

*Build 220724*

0. Foreword

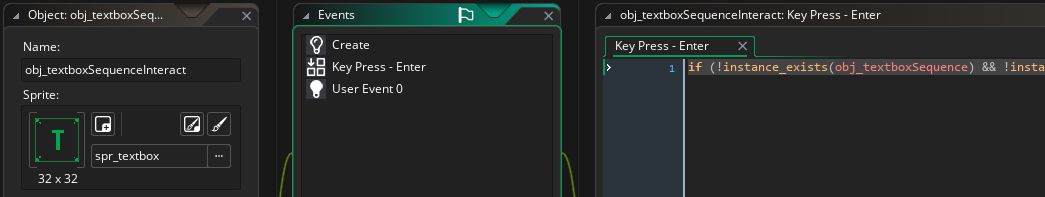
When developing a game with The OQRP Engine, there are several developmental guidelines that deviate from standard Game Maker 2 creation practice. This guide will show you the most basic and fundamental principles to help get you started on your project. Keep in mind that you need to follow these guidelines **strictly,** and **throughout the entire development of your game.**

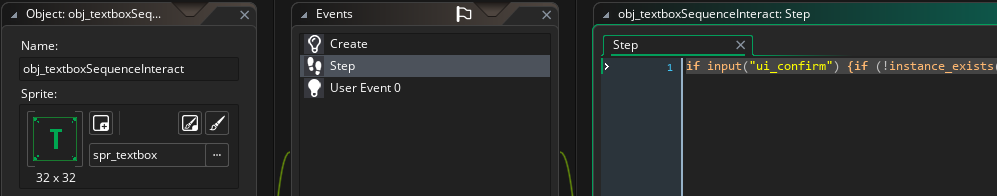
1. Using Controls in your Game

The OQRP Engine overrides the default Game Maker control scheme and adds additional functionality that is not normally supported. When implementing controls in your game, there are a few things you need to take into consideration.

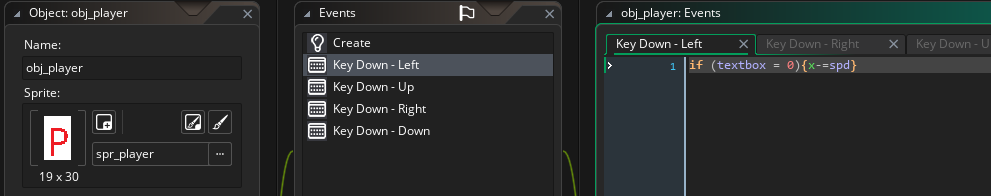
**1a. Events and Checks**

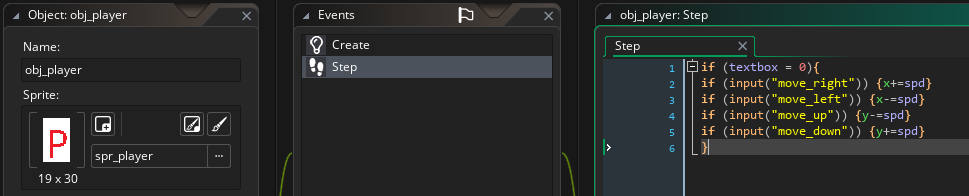
Instead of using Game Maker’s default control events, use the function input(action\_id) inside your regular events (most commonly Step, Begin Step and End Step but you can also use others for single-frame checking). These functions return a boolean value. Here are some examples:

**Textbox Interaction** – Here’s how you would do it in regular GMS2:

and here’s how to do it in the OQRP Engine:

In this case, if input(action\_id) completely replaces the key press event.

**Player Movement** – Here is a basic player character:

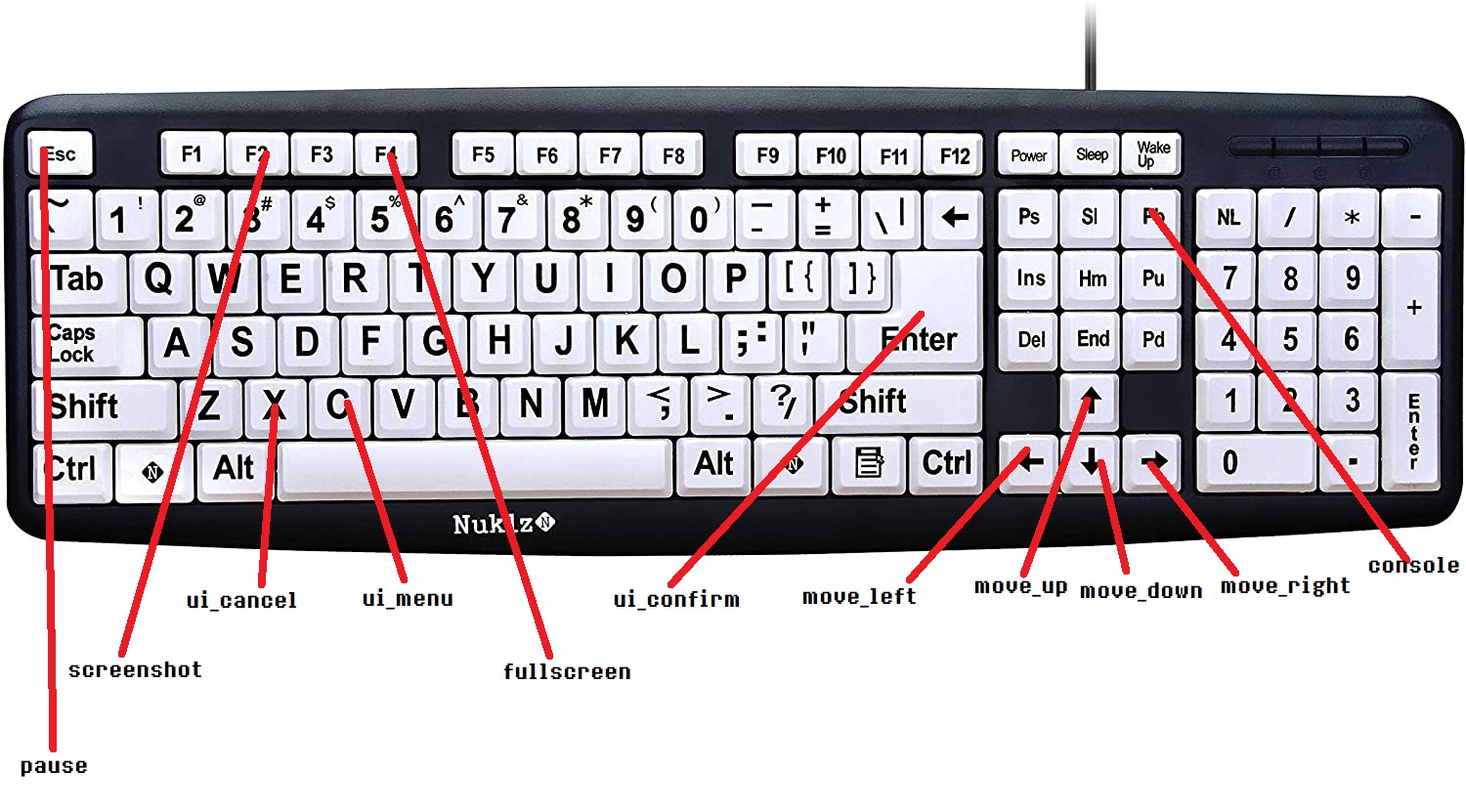
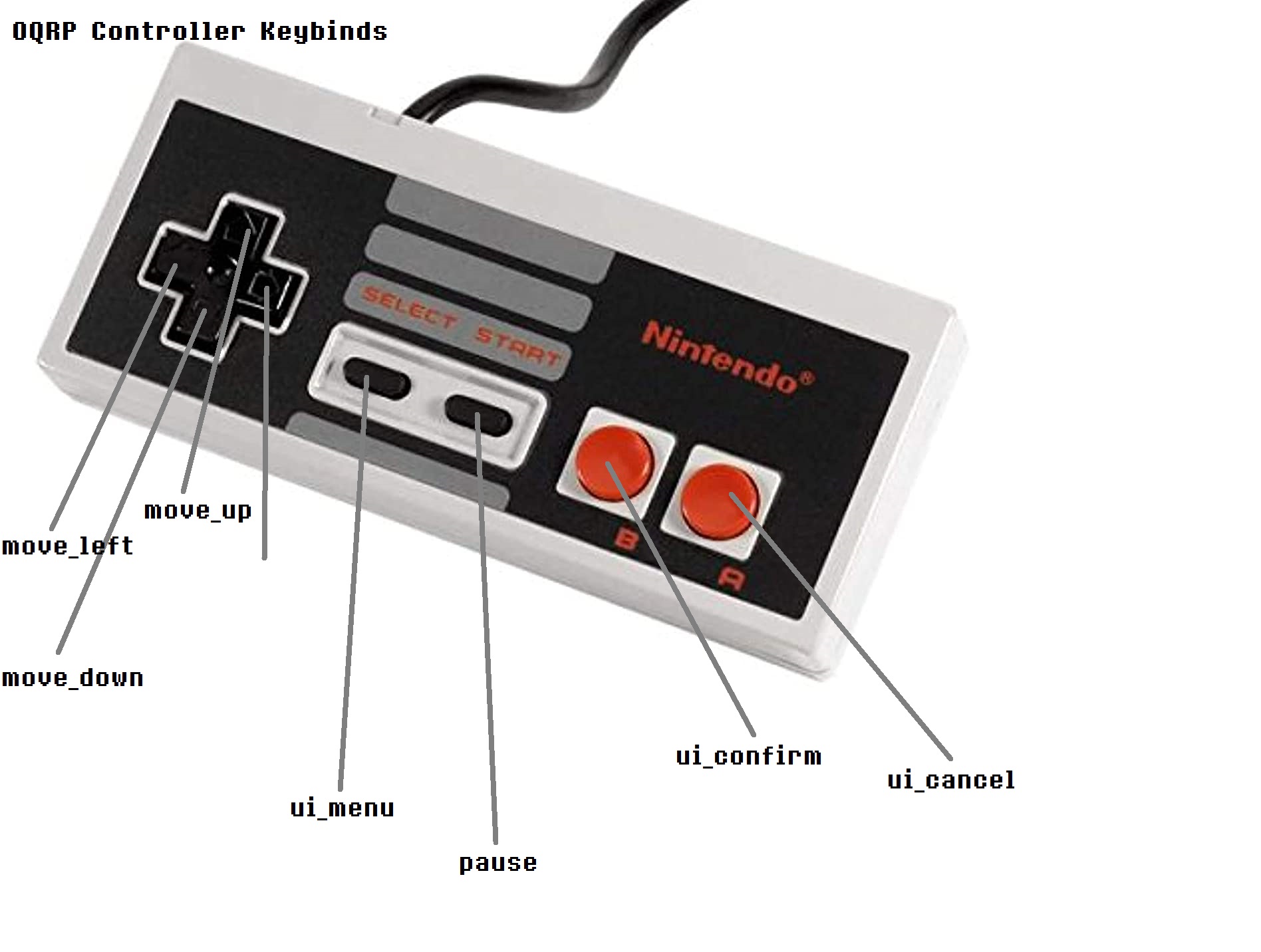
and here’s the OQRP implementation:

Once again, if input(action\_id) completely replaces the key down event.

While this may make your code messier, it allows you to effortlessly use custom dynamic single- or multi-keybinds, use other input devices like controllers and switch between them „on the fly“. This is not possible with Game Maker’s standard key events.

**1b. Action IDs**

The OQRP Engine uses Action IDs to determine player interaction regardless of platform or input device. Below are images of default control schemes and mappings, using a PC keyboard and NES controller as references.



2. The Saving/Loading System

The OQRP Engine uses the .OQRPSAV file format to store saved games. This is an encrypted and signed file format with the purpose of detecting and protecting against possible safe file corruption and modification, i.e. cheats. This approach has a few downsides. To prevent the player from downloading someone else’s completed game, the save files are tied to the player’s device. This may unfortunately break or invalidate a player’s save files in a number of scenarios, including a hardware change or game reinstall. The system is slot-based; the developer can choose how many save slots their game can use.

**2a. Initialization**

Upon startup, the engine initializes the save system, slot system and file system, and detects a first launch scenario (can be checked with global.firstlaunch). In the case of an FLS, the save system builds a persistent save file (persistent.oqrpsav), otherwise performing an integrity check on the persistent file and loading it. The persistent file, as well as all other save files with the .OQRPSAV format are essentially .INI files with three sections:

* Config: Keys containing the game configuration, versions, TOB and VKEY of the file.
* Info: Gameplay information about the save file, such as progression or the name.
* Data: The encrypted save file data itself.

**2b. The slot system**

Preload information about a save file can be obtained using SaveInfo(num), where num is the save slot. This function returns a ds\_map containing the preload information chosen by the developer. To edit the preload information for a given slot, use SaveInfo\_Edit(num, key, value).

To delete a slot, use Delgame(num). To reset the save file system, use SaveReset(full), where full is a boolean indicating whether or not to wipe the persistent save file. If a full reset is executed, the next launch of the game will trigger a FLS.

**2c. Saving and Loading**

Save data takes the form of a ds\_map. For an active save file, data is stored in global.save. The persistent save file’s data is stored in global.psave. These ds\_maps can either be accessed directly, or through the built-in functions savePut(name, value), saveGet(name), psavePut(name, value) and psaveGet(value). To save a game, use the void functions Savegame(slot) and SavegamePersistent(). To load a save file, use Loadgame(slot) and LoadgamePersistent(), although it should be mentioned that the latter is called automatically with initialization, so there is no need to call it a second time in-game. When loading, if a requested save slot doesn’t exist or cannot be found, the engine will throw an exception and the game will restart. If a save file fails the integrity check, the engine will throw an exception and the game will exit. This is different from when a persistent save file fails the integrity check, in which case the engine will perform a full save reset without warning, throw an exception, and restart the game.

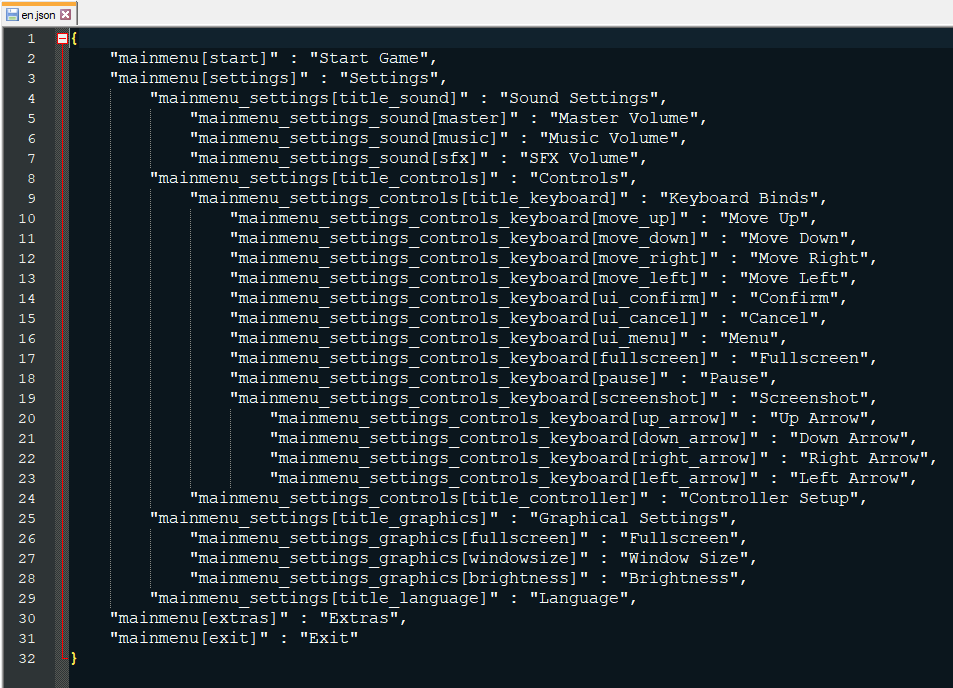
**2d. Cache system**

In addition to the save system, The OQRP Engine also has a temporary cache system. Accessing the cache is identical to accesing a save file, either directly via global.cache or with CachePut(name, value) and CacheGet(name). The cache is a temporary storage location that will remain in memory and not saved to disk, and it will be cleared upon exiting the game.

3. Written Text visible to the Player

One of the design goals for The OQRP Engine is an advanced localization system that supports switching between many different languages „on the fly“, affecting not just the text in-game, but also every aspect of the UI.

**3a. Understanding the script file**

Below is a sample of an english oqrp script file:

Language files are stored in the datafiles\lang\ directory. Notice that this file is stored in a JSON format, as it needs to be human-readable and human-editable, so that translators can quickly and easily make changes to the file. The easiest method for editing this file is to use a JSON editor, but it can also be edited by hand using any available text editor.

Keep in mind as this is very important: **Every line of text that the player can see in the game, including UI, textboxes, descriptions, items, titles, etc... needs to be stored in this file and acessed through the code! Do not insert player-readable strings into the game’s code! Refer to section 3b on how to use this file!**

**3b. Reading the Script File**

To acess contents of the script file, use the function s(key), where key is the unique identifier given to the string, i.e. „mainmenu[start]“. This function returns the requested human-readable string. Here is an example of how to use this function:

Drawing text to the screen:

draw\_text(0, 0, „Hello World!“) – incorrect

draw\_text(0, 0, s(„text[helloworld]“)) – correct

and here is how the text[helloworld] key looks like in the script file:

„text[helloworld]“ : „Hello World!“, 🡨 don’t forget the comma at the end (if there is another entry below it)

The obvious advantage of this system is that the key for the given string stays the same, regardless of selected language. Changing languages can therefore be done instantly, and no actual changes to the code have to be done. It also allows translators to modify the script file without needing to modify or even see the code. Translations to multiple languages can be done simultaneously.