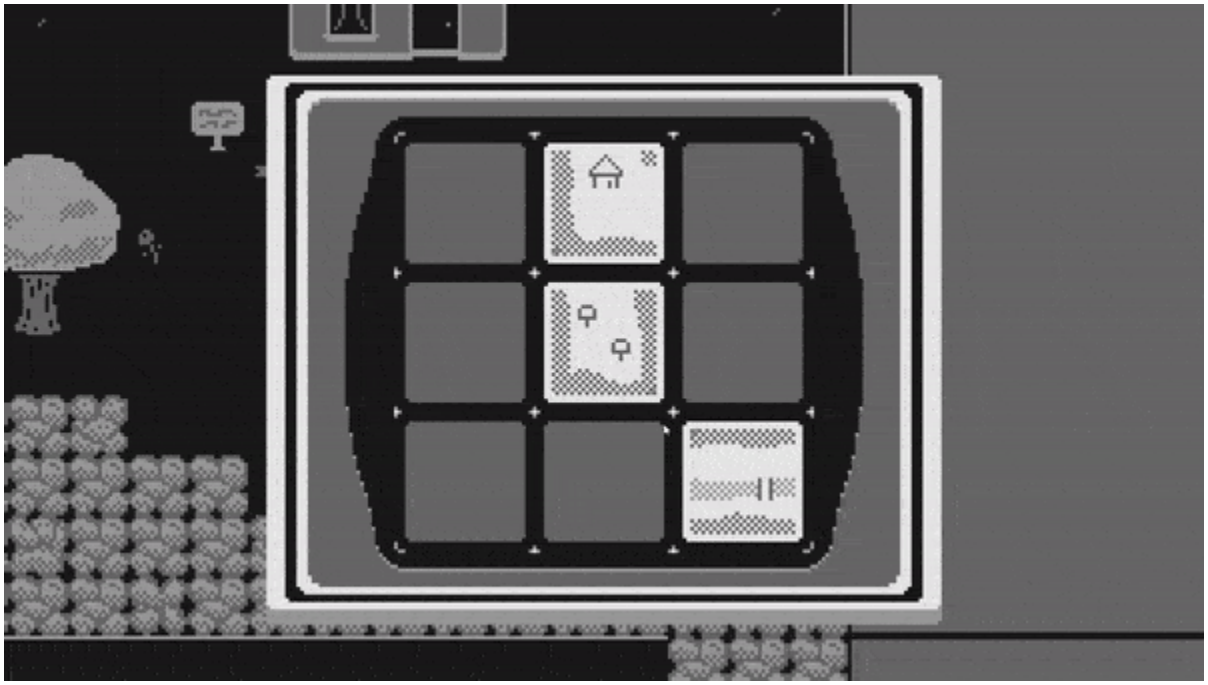
Slider Documentation



Github: https://github.com/randomerz/Slider
Last updated 7/10/22

*As of right now, we have ~~55~~ **217 (As of: 7/1/22)** code files in our code base. I don't think I'll be able to cover them all.*

Overview

Slider is a 2D puzzle exploration game. You control a player with a top-down view and can open an Artifact to manipulate the map to solve quests and puzzles. The map takes place on a Grid, and you move various Slider Tiles with the artifact.

Prominent Packages we Use:

- TextMeshPro
- FMOD
- Steam
- URP
- New InputSystem

IMPORTANT INFO

If you are working on this game, do not edit the same scenes! This will cause a lot of merge conflicts and make me sad. Create your own folder for scenes and duplicate someone else's with Ctrl+D. "`Scenes/Dev/Boomo/Base Scene`" is a good scene to start out with.

When possible, edit prefabs rather than scenes themselves. For example, if you're changing values on the `GameManager` component, try to do so in prefab edit mode rather than in the scene. Of course, sometimes this isn't possible, so just be courteous of others and try to avoid issues as best you can!

Summary

There are 5 parts this document will cover.

The **Player** section discusses how Player is implemented and code related to controls. It also goes into Collectibles and Items.

The **Map and Artifact** section discusses how the world is set up on Grids and STiles. It also talks about how UIArtifact is used to control these Grids. Finally, it talks about how the world is set up in the unity editor.

The **UI** section covers the remaining portions of the UI, such as the pause menu.

The **NPC** section talks about dialogue and NPCs.

The **Systems** section talks about the miscellaneous systems used in the game.

Useful Design Patterns

This project uses a lot of the observer and singleton design patterns. The observer pattern (or publisher subscriber model) is extremely useful for creating event-based mechanics. Examples in Unity are `OnTriggerEnter2D` and various input system methods. Examples in our game are `Player.OnAction` and `SGridAnimator.OnTileMove`.

Player

The Player controller is fairly simple and spread across `Player.cs`, `PlayerAction.cs`, and `PlayerInventory.cs`. The controls defined in the Input System are movement (WASD), actions (E), and cycling inventory (Q).

Player

Type: Singleton
Player is primarily used for the few player functions we do need. It is also used by many other classes to get information relevant to the player, such as position and the STile the player is on.

PlayerAction

Type: Singleton
PlayerAction controls the various actions a player can perform. Currently, this is just picking up objects and throwing them (the code relating to `pick`). PlayerAction also sends out an event, OnAction, which is primarily used by a listener to be attached to various objects. The listener is `PlayerConditional.cs`, found under Utility. It can determine when the player walks into a trigger collider, subscribe to OnAction, and then invoke an event if OnAction is invoked.

These new event listeners will likely be attached to the various puzzle objects in the Jungle for changing Jungle Sprite Shapes, in the factory for the Levers and Gates, and in NPCs for following up with interaction.

PlayerInventory

Type: MonoBehavior (Singleton Conversion Candidate)
PlayerInventory is used to keep track of which Collectibles and Items the player has acquired. We still need to serialize this for saving!

Collectibles are pickups like the Sliders that add a new piece to the map, and quest items like coffee in the Village. These are (probably always) one time pickups and will remain in the player's inventory for the rest of the game.

Items are objects that can be picked up and put down. Additionally, some special items will have effects on gameplay and the world. These special items can be stored in the inventory, and should be sent back to the inventory on scene change. They can be taken out and used, cycling through them with the cycle inventory key (q).

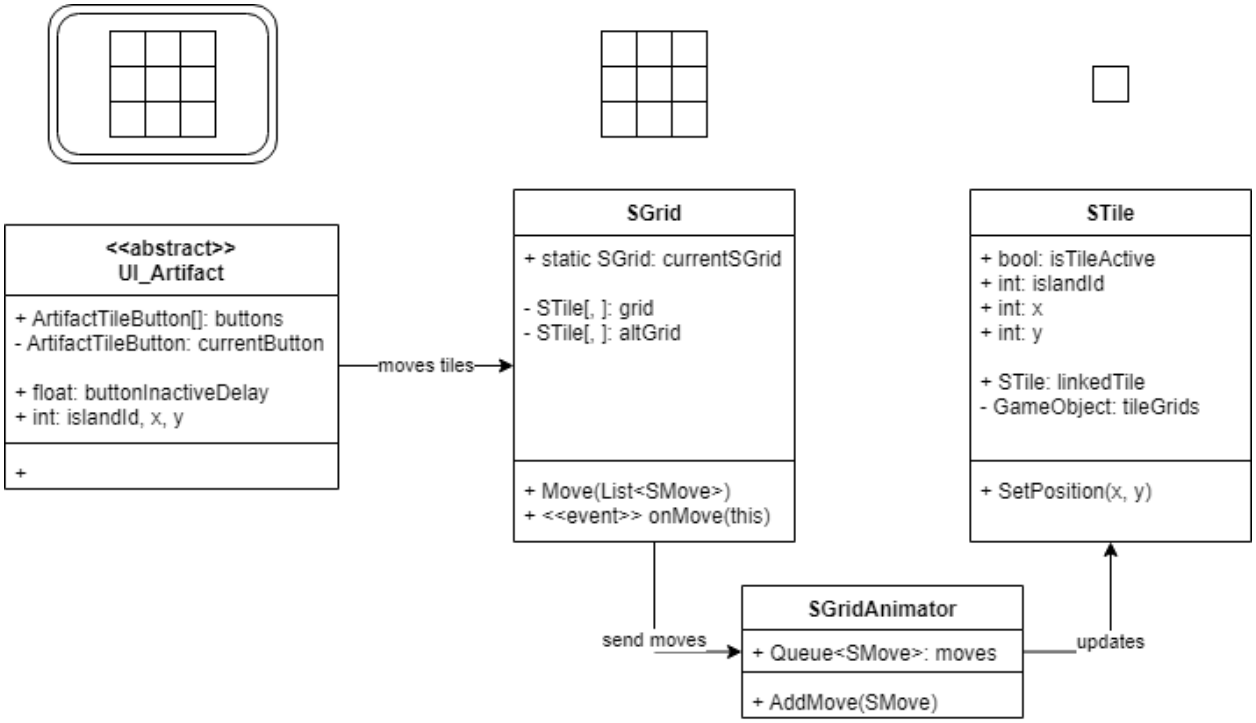
The anchor is a special item that locks a tile in place when dropped on it.

Map and Artifact

The game's heart revolves around manipulating grids in `SGrid.cs` and tiles in `STile.cs`. They are controlled by the various movements inputted by the user via `UIArtifact.cs`. We have two grids between SGrid and UIArtifact, so it's really janky working between them. This is in case we want to implement queues for movement and need a disconnect between them.

Moving Tiles

The diagram below shows a basic overview of the flow for how user input affects tiles.



UIArtifact isn't abstract, but it is extended upon!

STile

STiles are the various 17x17 tiles in a level. They contain information on where they are, and a few other things, but it is a little messy in terms of what it is responsible for. We will add an STile.OnMove some time soon!

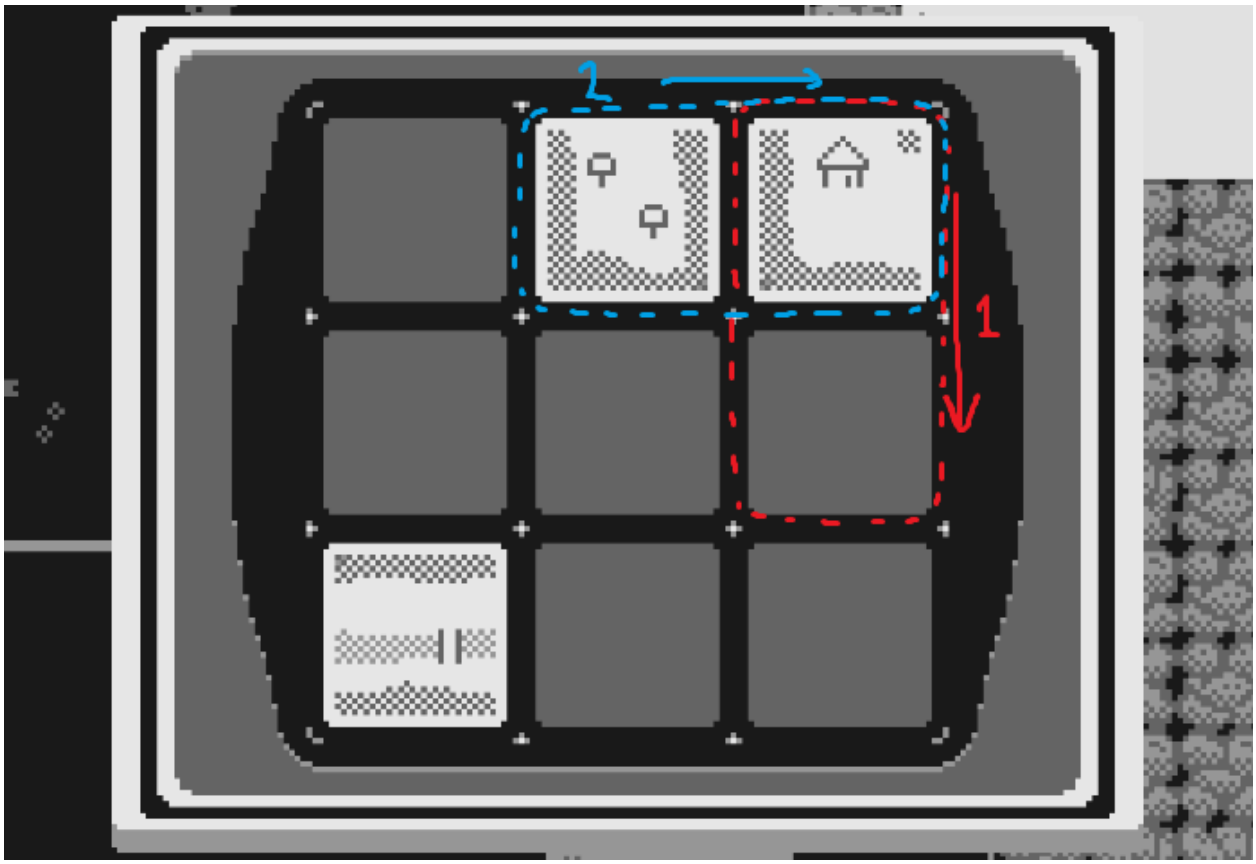
SGrid

Type: Singleton (Old System)

SGrid is where most of the action happens. This stores information about its relevant STiles in a 2D array, `grid[,]`. It receives various moves and immediately updates its internal storage of the grid. At the same time, it will send a move to `SGridAnimator.cs` which handles the movement in the world and cosmetic effects, like camera shake.

SGrid has an event, OnGridMove, and SGridAnimator has an event, OnTileMove. The former occurs immediately, and is useful for when you need to update UIArtifact (as done when checking for filling in completed tiles, like the final puzzle in the Village). The latter is useful if you are checking for something to occur in the world.

Borders are set up around the grid when a tile moves using `SGridBackground.cs`. There are four borders around each tile, and they are toggled on and off based on which STiles are involved in movement. See the image below for an example – When the first move is made, it enables the red borders, and when the second move is made, it enables the blue borders.



SliderColliders are also involved as a collider covering the entire tile. These can be activated to prevent the player from walking into the void.

If you want to check if the SGrid is in a certain pattern, you can use `CheckGridState.cs` to find if it matches any patterns.

SMove

When the player inputs a move into the UIArtifact, it will often create an SMove in order to pass to SGrid and move the tiles. This usually comes in the form of SMoveSwap, which will swap an active tile with an inactive tile.

UIArtifact

This is used to control the various STiles and movement. It's mostly UI code for how to select objects. The buttons have their information kept in `ArtifactTileButton.cs`, and include information on the position and sprites relevant. Some classes may extend off of this to change functionality, such as `OceanArtifact.cs`, which adds functionality for rotations. GetAdjacent and GetButton are some notable methods in UIArtifact.

Map Setup

The map is set up into a hierarchy as shown below. Map is an empty gameobject which holds the various other parts.

EightPuzzle

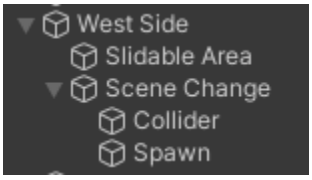
EightPuzzle has the level's SGrid attached to it, and childs the 9 STiles of the area. Each STile then has the various Tile Maps which will be filled with Tiles as needed. Floor, Wall, and Decoration are all purely visual while Colliders contains all the hitboxes. If you click on "Tile Maps", there is an editor function that allows you to convert all wall tiles into colliders for quick work.

Decorations will parent all of the GameObjects tied to a tile, such as Collectibles and Items like Slider pieces and Anchors, or decorative GameObjects like trees and buildings.

BGFrame is also used by the EightPuzzle for colliders when moving, and sprites to fill the void.

World Grid and Scene Transitions

WorldGrid works similarly to STiles in terms of organization of the various tilemaps, only they are for the static world.



Scene transitions are handled with SceneChanger.cs and SceneSpawns.cs, which can be seen implemented in the game in various scenes.

User Interface

The pausing functions and opening/closing of menus are handled in `UIManager.cs`. Various screen effects are handled in `UIEffects.cs`.

UIManager

Type: Singleton
Most of UIManager is self explanatory in opening and closing the various panels. It uses the various controls in the UI part of the InputSystem.

UIEffects

Type: Singleton
UIEffects is a singleton that can be called to perform various effects. Some include fading the screen to black, picking up an item, and flashing the screen.

UINavigationManager

Type: Singleton
This component handles keyboard navigation for UI. It is attached to the `EventSystem` in the scene. Here are some notable features:
Properties:

- **CurrentMenu:** This should be set to a GameObject which has a `SelectableSet` component. Make sure this matches the currently active menu panel (ie. which panel the player is currently navigating) or navigation won't work properly.
- **InMouseControlMode:** Making a keyboard input switches to keyboard mode and clicking with the mouse switches to mouse mode. The key distinction is that buttons will not be put into the Selected state when we are in mouse mode. You really shouldn't need to change this unless you are doing something very strange.

Methods:

- **ClearSelectable:** Unselects whatever is currently selected. No, you can't just do `selectable.Unselect()` or anything similar. Yes, this drives me insane. This was the original impetus for this script which has grown larger over time to support more functionality.

- **ButtonInCurrentMenuIsSelected:** Use this to check if a button in the current menu panel is selected (such as for selecting a default button when a navigation key is pressed and none is currently selected.)
- **SelectBestButtonInCurrentMenu:** Selects the first select in the `SelectableSet` component on the CurrentMenu object,
- **LockoutSelectablesInCurrentMenu:** Apply a temporary lockout to all selectables in the current menu panel, making them un-interactable for the passed in duration. This has no effect if currentMenu is null.

SelectableSet:

Type: MonoBehaviour

This component should be attached to UI panels. In other words: if you set `UINavigationManager.CurrentMenu` to a specific GameObject, it should almost certainly have a `SelectableSet` component attached to it. The `Selectables` array should contain all selectables (eg. buttons) within the UI panel. The order matters: higher up items will be prioritized by `UINavigationManager.SelectBestButtonInCurrentMenu`.

NPCs

NPCs are currently split into 3 parts: `NPC.cs`, `DialogueConditionals.cs`, and `DialogueDisplay.cs`. The dialogue roughly follows a Model, View, Controller structure.

`DialogueConditionals.cs`

Represents a possible state the NPC could be in based on certain conditions with the player, map, other NPCs, etc. . Despite its name, these conditionals can control more than just dialogue, such as world events or NPC walking events. In the MVC, this would essentially act as the model.

`DialogueDisplay.cs`

As its name suggests, this part is used solely to display the dialogue above the NPC's head. It is fed the dialogue by NPC.cs. The typing of the dialogue is controlled by a custom TextMeshPro script. In MVC, this would be the view.

`NPC.cs`

Contains a list of DialogueConditionals (dconds) that control the actions that the NPC takes at any given time. This script checks the conditionals for each second and determines which dialogue should be selected based on the results. The dialogue also has a priority system, which prioritizes certain conditions over others. In MVC, this would be the controller.

Setting dconds in the Inspector:

Each individual dcond contains a number of options:

Conditions:

This list is inherited from the `Conditionals.cs` base class. All conditions here need to be satisfied in order for the NPC to speak the dialogue associated with the

conditions. **Do Not** use *OnSuccess* or *OnFail* with dialogue, as these are executed every frame and will interfere with the typing of the dialogue (especially for teleports). If you need to do something when the dialogue switches, use *OnDialogueChanged*.

Dialogue:

Although there is a string textbox for inserting dialogue in the base dconds, ***it is highly preferred to use dialogueChain instead (even if it is only one line).***
Each dialogue piece in dialogueChain is given one after the other based on the options that are checked. If waitUntilPlayerAction is not checked, the dialogue will wait for delayAfterFinishedTyping before continuing.

onDialogueStart and onDialogueEnd events are included for each dialogue in the chain and **will override** the events given in the base dconds.

Walking:

Walking is done through a list of walks. For each walk, you need to give it a list of transforms that lay out the path of the NPC, as well as indicate the movements between stiles, and which directions are valid to move between. The Desert scene has good examples of how this is done.

You can use StartCurrentWalk or StartValidWalk to make the NPC walk. These are usually called through the *OnDialogueChanged* event in the dconds. There are a number of additional events that you can call per walk. OnPathBroken and onPathResumed are used for when the path the NPC was walking is broken by a stile movement.

Systems

There are a number of miscellaneous systems that the game also uses.

Audio

Audio currently uses `AudioManager.cs` to keep track of and play various sound clips. This will eventually need to be updated to integrate with FMOD.

Saving

Saving is handled under `SaveSystem.cs` and responsible for keeping data between scene changes as well as file saving. It serializes information and saves it by area, but currently only works on grids via `SGridData.cs`. In the future, it will also be able to serialize Player information, like the inventory, and Missions. Missions are a method of keeping track of world events that have happened, such as the fish being in the river in the Village.

Effects

`CameraShake.cs` is a singleton that can be called at any time to shake the camera.

Input

Controls.cs

Type: Singleton

Don't manually subscribe and unsubscribe from Input Action events!

Until recently, this was the only way this project did things. This has several downsides:

- You have to create an independent instance of InputActions for each script, which is pointless and inefficient.
- You have to manually enable/disable these actions at the right time to avoid having controls active/inactive when you don't want them to be.
- You end up with a lot of duplicated code!

Instead of doing this, use the Singleton class `Controls` to add behavior to input actions. This class provides several useful features:

Properties

- **Bindings:** Returns an instance of InputSettings containing our current bindings. If the bindings are not yet loaded, this will load them before returning. However, this will not update the bindings to match the latest ones stored inside of PlayerPrefs if the bindings were already loaded previously. If you wish to do this, call `LoadBindings` This should not be necessary unless you are writing new bindings to PlayerPrefs in your code.

Methods:

- **LoadBindings:** Loads the latest bindings from PlayerPrefs. You should only need to call this when you write new bindings into PlayerPrefs in your code.
- **RegisterBindingBehavior:** This method supports two use cases: *managed binding behaviors* and *unmanaged binding behaviors*.
    - A managed binding behavior is automatically removed when the control is triggered while the owner is inactive. Until that happens, the behavior you pass in will be triggered every time the binding you pass in is triggered. In other words: call this method to add behavior to a particular input action, lasting until the owner you specify in the method call is disabled.
    - An unmanaged binding behavior is not automatically removed — you must remove it yourself! **You should only use this when a managed binding behavior does not work for your use case.** For an example, see the `SPECIAL BINDING BEHAVIOR SETUP` region inside of `ShopManager.cs`.
- **UnregisterBindingBehavior:** Undo the above method.
- **GetBindingDisplayString:** Use this to get a UI-ready display string for the bindings on the passed in action. **Use this instead of `action.GetBindingDisplayString` because this method considers the current control scheme.**