Puzzle Game Collection Project - Technical Documentation

NatsuYuki

03.01.2022

Contents

1 Project Scope

The scope of this project is to provide a collection of varying puzzle games, all accessible from the same executable (on 64-bit Windows computers, but that range might be extended to other operating systems). For now, the planned games are: Sudoku, Minesweeper, Klondike Solitaire, 2048 and Battleship. This list may be extended if time allows. If time allows, certain games will receive online functionality (such as Battleship allowing player battles over the internet).

2 Introduction

The whole project shall be developed using Unity as the game engine, with coding done in C#, using Visual Studio as an IDE through Unity's Scripts system. The main menu, as well as each individual game will be part of a "Scene", all interconnected through Unity's Scene Management system. Each game will make use of 2D sprites for graphics, and will be controlled using either the keyboard, mouse or both depending on the game. Users will be able to change the game's window size, as well as the volume and window type settings from the main menu and from each game's title screen, which will be created using Unity's UI system. Here are a few key elements which will be used:

1. Scene - contains every single part of a stage/menu. In this project, each game menu as well as the Main Menu is a separate scene. Each game itself is contained within its separate scene;

2. Game Object - any entity existing in the scene. Can be a camera, a sprite, text, anything.

3. Component - attached to a Game Object, it gives functionality to a Game Object (i.e. the Sprite Renderer allowing for a Game Object to have a sprite attached to it);

4. Script - piece of code written in C# (in this project's case). It bridges the gap between the developer's code and the Unity Editor. Scripts are added as Components to Game Objects to add custom functionality.

3 Main Menu

Creating the Main Menu makes heavy use of Unity's UI system, however this only gives a skeleton without any functionality to the project. If we want the buttons, sliders, toggles etc to have any functionality, we make use of Scripts.

Here is how all UI elements are handled through the Scripts:

1. Button: calls a public function with void as the return type; Which function is called is assigned through the Unity Editor, and one button can

call several functions at the same time (this is how we can both have it play a sound and do what it is supposed to do at the same time);

2. Slider: updates a public, static numeric variable each frame the slider itself is updated. By using a TextMeshProUGUI component, we change the number attached to the slider to reflect the current value better and thus give the user a better idea of what they have selected;

3. Toggle: updates a public, static boolean variable each time the box is clicked;

4. Dropdown menu: upon selecting an option, calls a public, void return-type function taking an int variable as an argument.

All text is created through the TextMeshPro variant Game Objects. For the main menu, all text except the audio volume numbers is static and cannot be changed through any means.

Out of the 7 buttons available, 5 of these redirect the user to the Scene containing the respective game that the user clicked on. Out of the remaining two, one quits the game and the other brings forth a Settings menu. This is done by simply enabling the Game Object attached to the Settings menu and disabling the one attached to the Main Menu. In fact, this is how all menus which belong to the same scene are handled.

The Settings menu has the same features for every game as well as the Main Menu, mainly: audio and display settings. The audio settings offer two sliders which control the BGM (background music), respectively the SFX (sound effects) volumes. The display settings offer two dropdown menus which change the display mode (fullscreen, fullscreen borderless, windowed, switched through on a case-by-case basis with the switch statement), respectively the display resolution (ranging from 800 X 600 to 1920 X 1080, switched through on a case-by-case basis with the switch statement). All settings remain the same, even after the user quits the game and enters it again.

Finally, the user can input their name in the username section for the high-scores function. The limit is 8 characters.

From here on, we will not discuss any menu elements further, as all the other menus follow the same principles as the Main Menu, using the techniques described in this section.

4 Minesweeper

4.1 Creating a valid board

First off, to create a valid board, we need to create the tiles that make up the board. To do this, we use what is known as a Prefab, which is simply a Game Object which we can use to make several copies of it, with varying values. This is akin to Classes and Objects, where Prefabs are the Classes and Game Objects are the Objects. In our example, the Tile Prefab will have a

Transform component and a Sprite Renderer component with the Default Tile sprite attached from the get-go.

Next, we run two for loops ranging from x = 0 to x = width - 1 and y = 0 to y = height - 1, in which we create new tiles using the Tile Prefab using the Instantiate() function. Next, we add a TileBehaviour script component to the newly instantiated tile by using the AddComponent<>() function. This script contains all the important information about a single tile, such as its position on the grid and in the scene, its sprite renderer component (for changing between sprites), the number of adjacent mines, if the tile has a mine itself as well as its current state through the use of an enum. We do this for every tile.

Though we have generated the board itself, it is simply full of default tiles until the user clicks once (so that the user doesn't lose immediately, with the exception of one case detailed further). Upon the first click, we have two distinct cases:

1. We have the maximum amount of mines: in this case, we randomly choose one free position, then fill all other positions with mines. This way, we ensure the user doesn't always win the game.

2. Any other amount of mines: in this case, we randomly choose x positions, where x is the amount of mines, to have mines under, other than the tile the user clicked on. However, so that the game doesn't take forever to load for high amounts of mines, there is a failsafe: if we couldn't find a non-mine tile 10 times in a row, we simply go from left to right, top to bottom and fill in the next available tile with a mine (unless that tile is the one the user clicked on first).

Now, we finally have a valid Minesweeper board, and the game has begun.

4.2 Implementing game over conditions and some QoL additions

4.2.1 Implementing game over due to loss

This is as simple as checking if the tile the user just clicked on has a mine under. If it does, then the user has lost.

4.2.2 Implementing game over due to win

If the user reveals all non-mine tiles, then they have won. So we iterate through each tile, then check if it does not have a mine and its state is Revealed. If any tile does not meet this criteria, we return false and the game continues. We only do this check upon the user clicking on any tile (unless it's a tile with a mine under, in which case they immediately lose and this check is skipped as it's senseless). If all tiles pass the check, however, then we return true and the game is won.

4.2.3 Implementing game over

Once the game is over, we have two cases:

1. User has won: we display "You won!" through the use of a GameObject with the TextMeshProUGUI component by editing its text value with the function SetText() in the script. We also display a "Play again?" button, which sends the user back to the Minesweeper menu.

2. User has lost: we reveal the entire board, so the user knows where they failed and where they did right, then display "You lost..." in the same way as we did the "You won!" text in the case above, along with the button.

4.2.4 Quality of Life additions

As QoL additions, I added the amount of time spent in seconds and the mines left on the top of the screen, using two separate GameObjects with the TextMeshProUGUI component attached to them, modifying the text on a frame-by-frame basis using the SetText() function in the script. To display the time in seconds, I use the UnityEngine static variable Time.timeSinceLevelLoad.

Another QoL addition is revealing all safe tiles surrounding the clicked tile and beyond, until we reach tiles that we are unsure of. To do this, we recursively call the function UncoverEmptySpace(), which receives a tile as a parameter. Upon each call, we set the sprite according to the amount of mines surrounding the tile, we set its state to being Revealed and we check if the amount of adjacent mines is different than 0. If it is, we return without checking anything else. If it is equal to 0, however, we check all adjacent tiles and reveal the ones which have yet to be revealed and are not flagged or question marked.

4.3 Implementing controls

4.3.1 Camera controls

To implement camera controls, we make use of the UnityEngine's InputManager. In the Editor, we can create different axes that can be modified through pushing a button or moving a stick/the mouse/the mouse's scrollwheel. The Editor already defines 3 axes that are important for us in this case:

1. the "Vertical" axis: ranges from -1f to 1f, with 0f as the neutral value. By pressing W or Arrow Up, we change the value towards 1f. By pressing S or Arrow Down, we change the value towards -1f.

2. the "Horizontal" axis: ranges from -1f to 1f, with 0f as the neutral value. By pressing D or Arrow Right, we change the value towards 1f. By pressing A or Arrow Left, we change the value towards -1f.

3. the "Mouse ScrollWheel" axis: ranges from -1f to 1f, with 0f as the neutral value. By moving the wheel upwards, we change the value towards 1f. By moving the wheel downwards, we change the value towards -1f.

Next, we simply change the camera's size and position by checking if each axis' value is different than the neutral value, as such:

- if "Vertical" axis > 0f, move camera up;
- if "Vertical" axis < 0f, move camera down;
- if "Horizontal" axis > 0f, move camera right;
- if "Horizontal" axis < 0f, move camera left;
- if "Mouse ScrollWheel" axis > 0f, zoom in;
- if "Mouse ScrollWheel" axis < 0f, zoom out.

Now, because the axis values themselves only range between -1f and 1f, we make use of another variable for each axis which determines the speed at which they go and we multiply it by the axis' value. Finally, we multiply the result by Time.deltaTime, which ensures that the speed at which the camera moves and zooms is relatively similar between all machines, since what deltaTime represents is the physical time taken between two in-game frames.

Also, to make sure the user doesn't accidentally lose track of where the board is, we set some limits on all of the axis, such that the camera cannot move or zoom in/out past those values. These values were set through trial and error, however they work on all board sizes.

4.3.2 Game controls

By moving the cursor on top of a tile, the user selects that tile.

By left clicking on a non-flagged, non-question marked tile, the user reveals what is under that tile.

By right clicking on a non-revealed tile, the tile switches through the following states:

- Default -> Flagged;
- Flagged -> Question Marked;
- Question Marked -> Default.

5 2048

5.1 Creating the board

To create the board, we use the same technique described in the Minesweeper "Creating a valid board" section. By making use of a Tile prefab, we spawn in new tiles for the board. This time, the tiles only contain information about the sprite renderer, position (only regarding the scene), TextMeshProUGUI and number. We don't store information about each tile's position in the grid, as we have an array with a constant size of board_width * board_height, where each index tells us exactly where the tile is on the grid (the x value is given by tile_index % board_width and the y value is given by tile_index / board_width). Any free tiles now lead to a null pointer. At the start, we place a random tile with the number 2 on it.

Now our board has been created and the game can begin.

5.2 Implementing the gameplay

Each time the user moves the tiles in a direction, a new tile is created in an empty slot, bearing either the number 2 or 4 (note that the first tile is always numbered 2). To move the tiles, we define a destination tile as:

- Free - there is nothing standing between the border and the tile we want to move. We move the tile to the border according to the direction given;
- Occupied - there is at least one tile standing between the border and the tile we want to move and the first tile to come in contact with the current tile does not have the same number as the current tile. We move the tile one position prior to the first occupied tile;
- CanConvert - there is at least one tile standing between the border and the tile we want to move and the first tile to come in contact with the current tile does have the same number as the current tile. We remove the current tile from the array and multiply the number of that first tile by 2 (effectively merging the two tiles together);
- NULL - the current tile is already at the border of the direction we want to move in, thus nothing happens.

We simply run this check for every tile and act accordingly by changing its position (Free/sometimes Occupied) or removing it and changing the number and sprite of the tile it was merged into (CanConvert). There are 10 predefined sprites that the tiles cycle through based on the number. The formula for choosing the sprite is: $log_2 number \% 10$, where the number given is used as the index of the array storing all the sprites in memory.

5.3 Implementing game over conditions and some QoL additions

5.3.1 Implementing game over due to loss

It is not enough to only check if the board is full, though that is a preliminary condition to continue checking for a game over due to loss. If the board is full, we run the following check: look at every tile and see if there are any able to merge. If there are, then there are moves left and we continue. Otherwise, we flag a game over due to loss.

5.3.2 Implementing game over due to win

NOTE: This condition is only checked if the user does not have Endless ticked on.

The program looks if any of the tiles have the same number as the one specified before starting the game. If there are any, flag a game over due to win. Otherwise, continue playing.

5.3.3 Implementing game over

Once the game is over, we have two cases:

1. User has won: we display "You won!" through the use of a GameObject with the TextMeshProUGUI component by editing its text value with the function SetText() in the script. We also display a "Play again?" button, which sends the user back to the 2048 menu.

2. User has lost: we display "You lost..." in the same way as we did the "You won!" text in the case above, along with the button.

5.3.4 Quality of Life additions

As QoL additions, I added the amount of time spent in seconds and the amount of moves done since the beginning on the top of the screen, using two separate GameObjects with the TextMeshProUGUI component attached to them, modifying the text on a frame-by-frame basis using the SetText() function in the script. To display the time in seconds, I use the UnityEngine static variable Time.timeSinceLevelLoad.

Another QoL addition is changing the tiles based on the number for an easier time judging between the different valued tiles. The way this is done is explained earlier in the Implementing the gameplay section.

5.4 Implementing controls

5.4.1 Game controls

Each frame, we check if one of the following keys is pressed:

- Arrow Up - queue the next direction to be up;
- Arrow Down - queue the next direction to be down;
- Arrow Left - queue the next direction to be left;
- Arrow Right - queue the next direction to be right.

Note that if the user presses all arrows down on the same frame, then the direction will be up, as these checks are made in several if ... else ... statements, meaning they are taken in order. Next, we move all tiles with the given direction that was queued earlier.

6 Battleships

6.1 Creating the boards

To create the boards, we simply place tiles as described in the Minesweeper "Creating a valid board" section. This is all we must do, as the users are the ones who decide where the ships go.

6.2 Implementing the gameplay

6.2.1 Preparation Phase

For the preparation phase, the users place their ships on the board by moving and rotating them. To achieve this, we have to take the ships as an object separate from a tile, though once in the Battle Phase, they are more or less the same. Each time a ship is selected, it is placed in the top-left portion of the board. The ship's position is always taken as the left or top-most tile. Using variables to determine its size and rotation, we check if the ship is within the boundaries of the board and if it can rotate. Once all ships are placed and the user has confirmed their choice, the ships are hidden by disabling the Sprite Renderer Component of these ships and either Player 2 gets to place their ships or the Battle Phase commences, if Player 2 has also confirmed their ship placement.

6.2.2 Battle Phase

For the battle phase, the current user's ships are displayed by enabling their ships' Sprite Renderer component, while the enemy's are hidden by disabling it. Upon clicking a tile, we use an overlay to display a cross if the attack failed, or a tick if the attack was successful. Each time a ship is hit, the ship checks if all of its tiles have been hit, declaring itself sunk if they have. Upon a ship being sunk, all ships are checked for their status. If all ships are sunk, then the opposing player wins.

6.3 Implementing the CPU

6.3.1 Preparation Phase

For the preparation phase, the CPU randomly places ships until it has found valid positions for all ships. The way it does this is by randomly selecting a rotation and position, then checking if that position is valid in a similar way to how the position would be checked for a real life player. Once it has managed to place all ships, the Battle Phase commences.

NOTE: If the CPU fails to place a ship 10 times, then it takes the next available place going from top-left to right, then bottom, given the rotation from the last try it made.

6.3.2 Battle Phase

For the battle phase, the CPU does try to make some educated guesses, but first, it randomly chooses a location until it finds one which wasn't hit before. (NOTE: If it fails to find one such location for 100 times in a row, it selects the next non-hit location starting from top-left going right, then bottom.). Once it has chosen its location and fires, two cases are possible:

- It didn't hit a ship, in which case the next time its turn is up the process begins anew;
- It did hit a ship, in which case it stores all adjacent positions which can be hit. The next time the CPU has to attack, it chooses from one of these locations. If it misses, it simply tries the next ones. If it hits, it adds those adjacent, valid locations as well. If the CPU runs out of educated guesses, then it's back to randomly trying until it hits something.

6.4 Implementing game over conditions and some QoL additions

6.4.1 Implementing game over

Once the game is over, we have two cases:

1. P1 has won: we display "Player 1 won!" through the use of a GameObject with the TextMeshProUGUI component by editing its text value with the function SetText() in the script. We also display a "Play again?" button, which sends the user back to the Battleships menu.
2. P2/CPU has won: we display "Player 2 won!" (or "CPU won!" if P2 is a CPU) through the use of a GameObject with the TextMeshProUGUI component by editing its text value with the function SetText() in the script. We also display a "Play again?" button, which sends the user back to the Battleships menu.

Either way, all ships are displayed for everyone to see where their opponent's ships were.

6.4.2 Quality of Life additions

As QoL additions, I added the player's turn on the bottom-left of the screen, using one GameObject with the TextMeshProUGUI component attached to it, modifying the text once a player's turn finishes using the SetText() function in the script.

6.5 Implementing controls

6.5.1 Game controls - Preparation Phase

Each frame, we check if one of the following keys is pressed and act accordingly:

- Number 1 - marks the size-2 ship as currently selected;
- Number 2 - marks the first size-3 ship as currently selected;
- Number 3 - marks the second size-3 ship as currently selected;
- Number 4 - marks the size-4 ship as currently selected;
- Number 5 - marks the size-5 ship as currently selected;
- Arrow Up - move the currently selected ship up;
- Arrow Down - move the currently selected ship down;
- Arrow Left - move the currently selected ship left;
- Arrow Right - move the currently selected ship right;
- R - rotates the currently selected ship;
- P - places the currently selected ship;
- Enter/Return - confirms the ship placement.

6.5.2 Game controls - Battle Phase

For the battle phase, the user clicks on a tile on the opponent's board. If the tile was not hit before, then it is marked as such and all the necessary checks are made.

7 Klondike Solitaire

7.1 Creating the table and deck

To create the table of Klondike Solitaire, we group up cards into piles. A single card, as well as a single pile, are different Game Objects with their own associated scripts. The card keeps track of its sprite, status and position, while the pile keeps track of what type of pile it is, the cards in it and moving any cards when necessary. We distinguish three types of rules for piles:

- DifferentSuits: these are used exclusively for the seven piles below, where cards have to go in ascending order, alternating between their colors;
- SameSuit: these are used for the waste and winning piles, where the cards have to be of the same suit (there's also an extra check for the winning piles that the card's number has to be the next one in that pile;
- NoRules: these piles have no rules whatsoever to them. This is for the drawing pile.

We also use two enums to distinguish between the suits of the cards and their numbers.

Now, to actually generate the board, we Instantiate each pile with some given offsets found through trial and error, then we generate the deck through a List of cards, we shuffle the deck (by swapping random cards a bunch of times) and finally put them into piles as the rules of normal Klondike Solitaire state: each bottom pile has one extra card than the other, only the top-most card is revealed. All other cards are placed in the drawing pile.

7.2 Implementing game over conditions and some QoL additions

7.2.1 Implementing game over

The game doesn't check for a game over, to allow the user to declare the game as over whenever they see fit, as I felt it is part of the experience of Klondike Solitaire. The game checks for a win by looking at the four winning piles and checking if they have exactly 13 cards. We don't care for the order in this case, as the cards cannot be placed in those piles unless they already meet the criteria. We then display "You won!" through the use of a GameObject with the TextMeshProUGUI component by editing its text value with the function SetText() in the script. We also display a "Play again?" button, which sends the user back to the Klondike Solitaire menu as well as a button to check the highscores.

7.2.2 Quality of Life additions

As QoL additions, I added the amount of time spent in seconds and the score on the bottom-left of the screen, using two separate GameObjects with the TextMeshProUGUI component attached to them, modifying the text on a frame-by-frame basis using the SetText() function in the script. To display the time in seconds, I use the UnityEngine static variable Time.timeSinceLevelLoad.

Another QoL addition is self-clearing the board once the game is sure all the cards can be moved to the winning piles. This is done by checking if there are any cards left to draw and if there are any face-down cards left. If neither of these conditions are met, then the button for self-clearing appears and the user may or may not use it, as they desire.

7.3 Implementing controls

7.3.1 Game controls

By moving the cursor on top of a card, the user selects that card.

By left clicking on a card when another card is selected, the user moves all cards below that card, including the selected card, to the target card, if it is allowed by Klondike Solitaire rules. Otherwise, the selected card becomes unselected and nothing else happens. The same happens if the user clicks on an empty pile. If the user instead clicks on the drawing pile with no card selected, a new card is dealt into the waste pile. If the drawing pile is empty, then all cards are moved from the waste pile to the drawing pile instead.

8 Sudoku

8.1 Creating a valid board

Creating a valid and unique Sudoku board involves some backtracking and having a Sudoku solver, which also makes use of backtracking. Thus, the entire process can be quite resource-heavy, but thankfully, in real-life test cases, the board is generated almost instantly due to some optimizations: having a list of lists for each tile which tells us at all times which numbers we can place, from which we randomly selected a number. If there exists a solution for our board, we continue. Otherwise, we trace back our steps until we manage to find a valid board. Rinse and repeat until we have our Sudoku board. From here, it's all a matter of simply hiding some of the numbers, not too many to make impossible, but not too few to make it too easy. Thus, a random number is chosen between a set range for each type of board and we hide that many tiles.

To implement the Sudoku solver, we recursively go through every possible number at random, checking if every tile has a possible number. If the whole board can be filled, then the board is valid. If there is no solution, then the board is invalid. To check if every number we place is correct, we make use of a IsValid() function, which checks if there is a tile with the same number on the same row, column or box as the current tile, in this order. If it finds one which is the same, it returns false. If all tests pass, it returns true.

8.2 Implementing game over conditions and some QoL additions

8.2.1 Implementing game over

As the user cannot lose Sudoku, the game declares that the user has won when all the numbers they have placed match the numbers they have stored in the generation phase as being correct. For this, each tile remembers not only the number it has, but also a "correct number", which is the number given after generation. Once they have won, we display "You won!" through the use of a GameObject with the TextMeshProUGUI component by editing its text value with the function SetText() in the script. We also display a "Play again?" button, which sends the user back to the Sudoku menu, and a button with which they can view the highscores.

8.2.2 Quality of Life additions

As QoL additions, I added the amount of time spent in seconds on the top of the screen, using one GameObject with the TextMeshProUGUI component attached to it, modifying the text on a frame-by-frame basis using the SetText() function in the script. To display the time in seconds, I use the UnityEngine static variable Time.timeSinceLevelLoad.

8.3 Implementing controls

8.3.1 Game controls

By moving the cursor on top of a tile and clicking on it, the user selects that tile.

Once a tile has been selected, the user can press on any number to change that tile's number to the one they have pressed. To make sure the user doesn't change any pre-determined tiles, the tiles keep track of their status in an enum to check if they can be changed or not.

9 Highscores

Each game (except Battleships) keeps track of its score. At the end of the game, the main game checks if the current score can be entered into the Highscores Table and does so, if it can. To differentiate between users, the user is required to enter a name before entering any game. If they choose not to and they haven't ever entered a name before, then they are simply taken as "Guest". While each game's score is counted differently, the score is stored as an integer for every game, thus the same basis can be used for every game's Highscore Table. The same script is used, in fact, however the main game is told which game's table it has to access so it knows which to go for.

9.1 Adding to the Highscore Table

Adding to the Highscore Table loads the current list of entries we have, sorts it, and removes the 11th element (if it exists). This is done so we always have a maximum of 10 entries. Afterwards, we save the current list.

9.2 Saving/Loading the Highscore Table

This is done through a .json file. Thankfully, the UnityEngine includes a class called JSONUtility with two key functions: FromJson and ToJson, which combined with PlayerPrefs allows us to save and load the list of entries with ease.