RetroRemakes Documentation

How to Install **4**

About this Project **6**

The Games **6**

Roller Maze 7

Lemurs 7

Oil Tycoon 7

Where to Get Help 7

Project Structure **8**

Roller Maze **10**

Gameplay 10

Scenes 11

Game Flow 11

Main Features 11

Scripts 12

Electric Barrier 12

Explosion 12

Extra 12

FollowPlayer 12

GameManager 12

Highscores 13

HighscoresRepository 13

IDamagingObstacle 13

LevelInfo 13

MainMenuPanel 13

MapGenerator 13

ParalaxMovement 13

PlayerController 13

ReplaceTilesWithGameObjects 14

ScaleOverTime 14

SlidingDoor 14

SoundManager 14

Tags 14

TeleporterData 14

TileCollection 15

tilemapBoundsCompressor 15

TileSwapper 15

TubeData 15
UIManager 15

Lemurs **16**
Gameplay 16
Scenes 17
Game Flow 19
Main Features 19
Scripts 20
CameraMovement 20
EnterCodeMenu 20
ExtendedButton 20
GameManager 20
IAbility 20
Lemur 20
LevelData 20
LevelManager 20
LevelUIManager 21
Map 21
MapData 21
Minimap 21
MouseCursor 21
SoundManager 21
UIManager 21
Editor/ExtendedButtonEditor 21

Oil Tycoon **23**
Gameplay 23
Oil drilling minigame 25
Pipelining minigame 26
Scenes 26
Game Flow 27
Main Features 28
Scripts 28
AnimateRamp 28
Calendar 29
CalendarRenderer 29
CompanyLogoOfPlayer 29
Contract 29
DistortedImage 29
DrawLineTexture 29
Economy 29
EventManager 29
GameManager 30
InfoTextOfPlayer 30

Judgement 30
MapSector 30
Oilfield 30
Player 30
Screen 30
ScreenProvider 30
SectorSelector 31
SelectableItem 31
Tank 31
Ticker 31
UIManager 31
Window 31
WindowProvider 31
Scriptable objects 32

Common Scripts 32
Extensions 32
GameManager 32
PointerClickDetector 32
SceneDescriptor 32
SpriteSwitcher 32
ToggleActive 33
UppercaseValidator 33

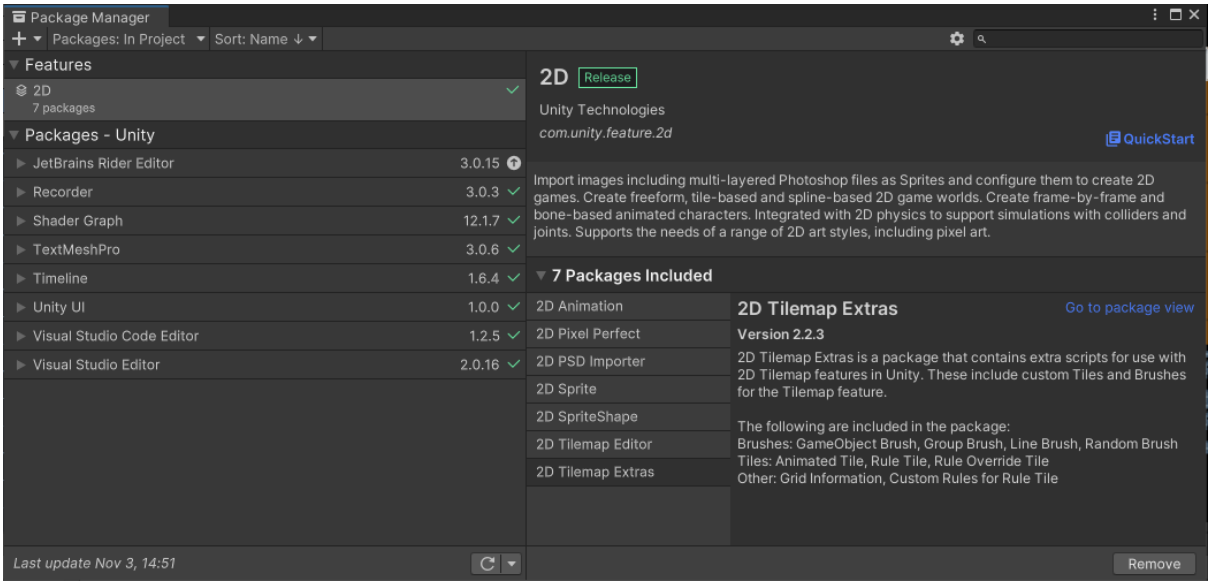
Where to Go From Here? 33

Last Words 33

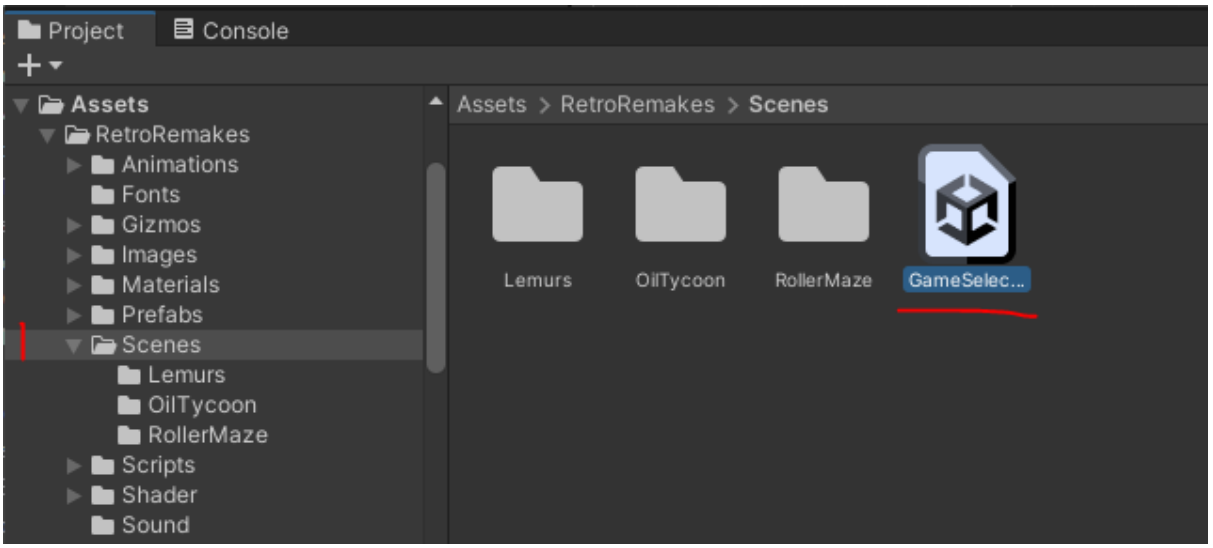
How to Install

Just import the package from the asset store. You will be asked if you want to create a new project with it and you should do so. Otherwise, some folders might get mixed up and the game won't work.

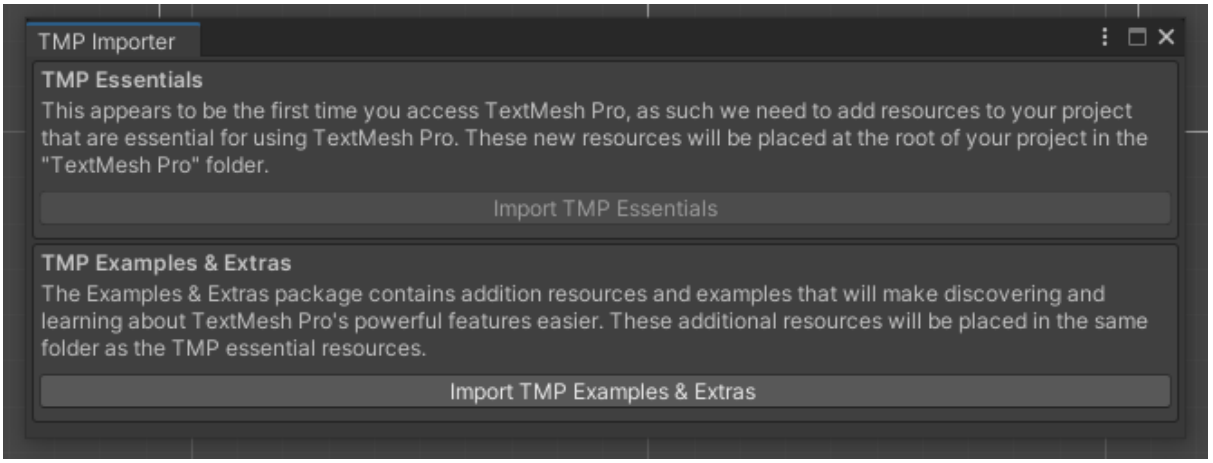
The project uses TextMeshPro, Tilemap, and Tilemap Extras. If these packages are not installed automatically or if you are not getting prompts to install them, you might need to go to the Unity package manager and add the necessary packages to the project:



Once everything is set up you can open the GameSelection scene from the scenes folder which is your entry point to start all games. You can return to that scene by hitting SHIFT+ESC anytime when you are inside one of the games.



When you open the scene for the first time, you should get the following dialog to import TextMeshPro Essentials. You just need to click the first button. The “Examples & Extras” are not needed.



About this Project

RetroRemakes is a complete Unity project that contains three remakes of old 16-bit classics. Making a remake has the advantage that you can focus on the actual creation of the game in Unity and don't need to spend too much time coming up with a game idea. It also provides you with some challenges you have to overcome (if you want to stay true to the original), so you can't decide to eliminate a feature because it is "too hard to do".

The games try to keep tech features at a rather simple level to respect the original possibilities that existed back in the 90s and also to keep the entry barrier low.

The games in this project are more or less feature complete but they are missing some content. For example, two games only have two levels to play because that is all that is needed to show how level switching and game ending can be implemented. The third game is missing one of the minigames left as an exercise for you.

That means you have the opportunity to add content and your own ideas, which is a big point of this project. After all, you will only learn new things if you try them out. Having a starter project might help you with that because you don't have to create a lot of objects and code before you have a good foundation for a game. Look through the code and try to understand the structure, play the games, and come up with ideas on what could be changed or added. Those can be really small things, and by adding them you will understand the project better and ultimately learn things for your own project.

The other goal of this project is to give you a working game so you can learn how to structure your objects and code in a project. Many tutorials teach a specific topic, and that's great if you want to solve a certain problem. But when I was starting using Unity, I was a bit uncertain where to put the code and where everything starts and how to go from one level to the next etc. Since the project is quite large for a "let's build it from scratch" video tutorial, you can access videos that will explain the project structure and code in greater detail which might help you better understand the decisions that were made. You can find the videos here (more videos will be released over time): https://www.youtube.com/@BitbeastGames

Please always keep in mind that these games were developed within a span of a few weeks. It takes time to make a game. Don't be intimidated by tutorials that make a game in 2 hours. Such videos just show coding of an already finished game by experienced developers.Take your time. Look at the project and experiment with it. And most importantly: have fun!

The Games

I chose these games in particular as I wanted them to be somewhat different from each other. So you will learn new things in every game, but you will also see how the same task can be solved in different ways. By doing so, you learn a few different methods that you can apply to your own project. There are always many possibilities to solve a problem depending on the game and your preference, so the more options you know, the better decisions you can make.

The games and their topics are as follows:

Roller Maze

In this game, you have to steer a ball through a labyrinth. Find your way to the exit in the shortest time possible. Use items you can pick up or buy such as keys to open doors or tools to build bridges. Avoid traps like explosions, acid pools, or holes that will send you into an infinite void.
The game makes heavy use of the TileMap feature of Unity. The levels are created with that editor, and the map data is also manipulated in code. We will read the data to check for items the player can collect, and we will change that data to alter the level following the player's interactions. It also does not utilise the usual collision detection (e.g. OnCollisionEnter2D) but a different approach that is good to know and might come in handy in one of your own games.

Lemurs

Here, you have to deal with a bunch of not-very-intelligent lemurs that run through the level independently. Your goal is to lead all of them safely to the exit. To achieve that, you can assign a limited amount of tasks to some of the lemurs, for example, digging a hole, building some stairs, or climbing a wall.
The game has pixel-precise collision and fully destructible terrain, so normal colliders can't be used efficiently. Instead, the game reads and writes directly from the image that is used as the level background. It also manages up to 100 lemurs at the same time. Another special feature is that the game only has one scene that displays all the different levels (versus one scene for each of the levels). The content of each level is configured in scriptable objects which are quite useful for that.

Oil Tycoon

In this game, you have to manage an oil company, buy oilfields and sell the oil at the best price. Sabotage your fellow players and win the game by achieving one of the goals that can be set at the start of the game.
The game is a typical management game that is purely made in the UI. It has custom screens and windows that can be configured before opening them and stores a lot of different data like who the players are, what oilfields they possess, what contracts they have signed etc. The data is also evaluated quite often in the game. This will introduce you to the beauty of LINQ, which is a powerful way to access data that is stored in collections like Lists or Dictionaries. The game also uses quite a lot of events with which windows can be configured (what should happen if the player presses OK?) and with which some of the timings in the game work very well (open the new window AFTER the closing animation of the old window is finished).

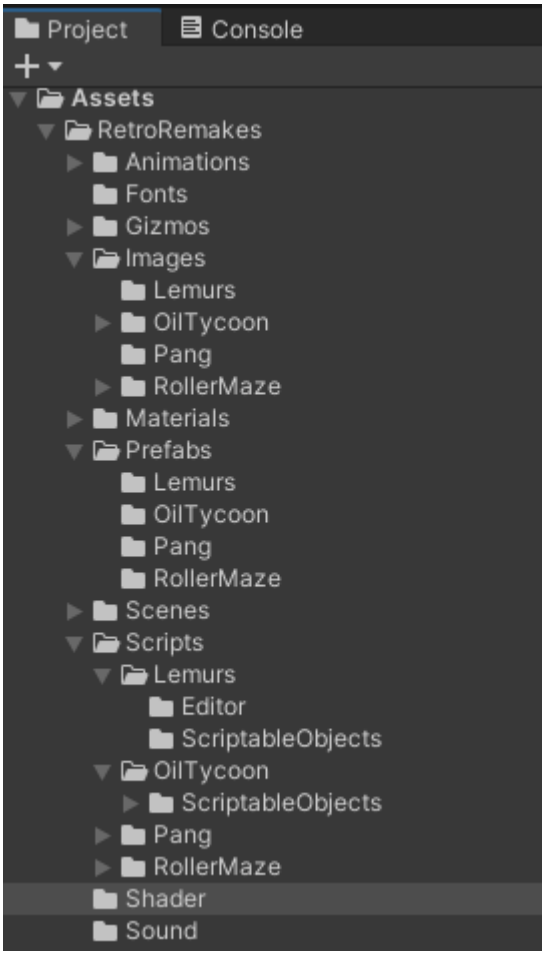
Where to Get Help

If you have any questions about the project or the code, you can visit the #retro-remakes channel BitBeast Games Discord: https://discord.gg/grjdJvM

For explanatory videos that will shed light on every corner of the code, you can visit the YT channel: https://www.youtube.com/@BitbeastGames
More videos will be added to the channel over time.

Project Structure

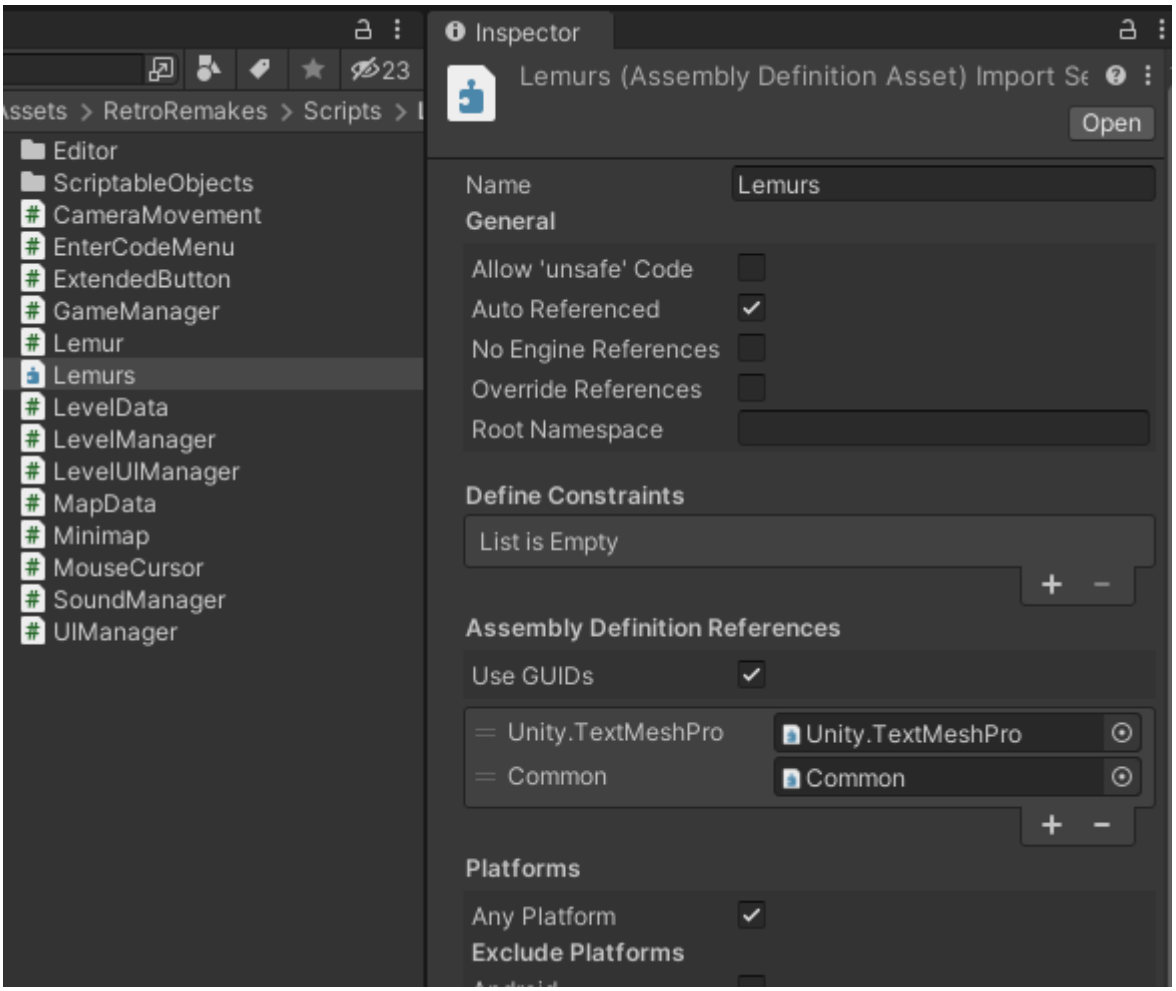
The project is structured as follows:



As you can see, there are folders for each asset type containing subfolders for each game. Since every game needs some classes that are similar - for example, a GameManager - the scripts are also organised in packages/namespaces so classes with the same name won't collide.

```
namespace RetroRemakes.Lemurs
{
    //manages the game flow like loading a level, going to the main menu,holding all level data etc.
    //it works as a singleton in the popular design für Unity
    public class GameManager : MonoBehaviour
    {
```

In addition to that, the code is organised in assemblies. That way, every game's code will be compiled into its own dll-File. Changes to the code of one game will only force a recompile of the classes for that game which results in shorter compile times.



The following chapters will explain the details of each game in greater detail.

Roller Maze

RollerMaze is designed for a resolution of 640x512 pixels, so please set the aspect ratio accordingly!

Gameplay

The goal is to navigate a ball through a labyrinth full of obstacles to find the exit.
You can collect items like jewels, coins, and keys which you can use to open doors. With the coins, certain items can be bought to make your life easier:

- Parachutes: will protect you from falling into the bottomless void
- Bombs: can be placed at certain spots to eradicate a 3x3 grid of obstacles such as walls
- Hammers: will close one hole next to the player to fix a broken passage
- Spikes: will allow you to travel on ice without sliding too much for a short time
- Armoured ball: once activated your ball will be of metal which allows you to bump into walls with cracks and if you have enough speed, the wall might come down; only lasts a limited amount of time
- Eyes: can be found in the level and will give you a crude map of your surroundings

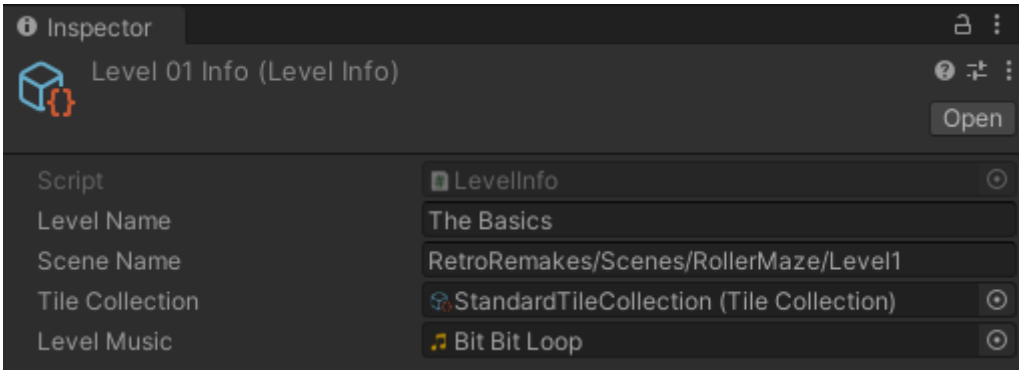
You can select these items by clicking the right mouse button. The selected item will be shown in the UI and can be used with a left click.

There are several dangers that will drain your energy, such as explosions, acid pools, or electric barriers. Once out of energy, you will lose one of your lives. When you are out of lives, the game ends and the highscores will be shown.

Hitting SHIFT+ESC brings the player back to the game selection menu where one of the three games of this project can be chosen.

Scenes

The game has one scene for the main menu and one scene for each level.
The main menu allows you to start a game or to review highscores for each level. When you start a game, you have to enter your name and the first level scene will be loaded. Some meta information for each level is stored in scriptable objects of the type LevelInfo:



The level scenes each contain multiple tile maps for several layers of the labyrinth (floor, items, obstacles, etc.). Most objects in the level scene are prefabs, so they stay the same for each level once a change is made to them.

Game Flow

The game starts in the MainMenu scene and loads the first level after starting a new game. When the player dies, the GameManager loads the MainMenu again. Once the player finds the exit of a level, the GameManager decides which level to load next depending on the list of LevelInfo objects that are assigned to the GameManager in the editor. If a next level is available, the corresponding scene is loaded (the name of the scene is stored in the LevelInfo scriptable object). If the last level is completed, the GameManager returns to the MainMenu.

Main Features

RollerMaze heavily relies on tilemaps. Each level has four tilemaps representing the infinite void, the floor, obstacles the player will collide with, and lastly the items map with everything the player can collect or use (e.g. coins or tubes to ride). The tiles come with individual colliders as set in the sprite editor and are combined into one big collider/trigger. The collisions are not handled by the usual OnCollisionEnter2D. Instead, a different method from the physics engine is used (Physics2D.OverlapCollider). So you will maybe learn a new approach that might be useful someday. We will also deal with getting bumped off of a wall manually without the need for the help of the physics engine which can give a greater amount of control.

In the code, the tilemaps are read and written in order to know where the player is and with what object they have interacted with. That way, we don't need to have many gameobjects in the scene (e.g. for items) which is advantageous when building a map because you can just "paint" items on the map and don't need to place gameobjects by hand. This has its limitations, and we work around that by replacing some tiles with gameobjects automatically as you will see in the code. The main reason for that is removing colliders for a single frame (like with the explosion) or applying dynamic damage (like with the electric barriers).

Sadly, you can't store any additional data on tiles (at least in the standard tilemap that comes with Unity; there are some assets on the Asset Store you can check out). So we have to replace some of the tiles with gameobjects as mentioned above, or we can add "data gameobjects" on the map. You will find some of these as a child of the items map.The gameobjects are used, for example, to store the information where a teleporter is supposed to move the player to. When hitting a teleporter tile, the game is looking for a data object in the same spot and gets the data from the gameobject it found. There is also a gizmo drawn on the map where a teleport starts and ends which can be colourized to allow for more clarity when placing many of them (see TeleporterData for details).

Scripts

The following list is a brief overview of the scripts and their respective roles used in the game. The exact details of each script are commented upon inside the scripts themselves.

Electric Barrier

Used on electric barriers on the map. Implements the IDamagingObstacle and is thereby recognized by the player controller as an obstacle that causes damage. See IDamaginObstacle for details.

Explosion

Used on explosions on the map. Implements the IDamagingObstacle and is thereby recognized by the player controller as an obstacle that causes damage. See IDamaginObstacle for details.

Extra

Base class for all extras the player can buy and activate. It already contains all common functionality which can be overridden and adapted as needed in concrete extras. This is done for example in ParachuteExtra, EyeExtra, ArmouredBallExtra, etc.

FollowPlayer

Used by the main camera to move with the player so they always stay in the middle of the screen. It's possible to set an offset for a bigger distance to the player.

GameManager

Controls the general game flow, e.g. loading new levels and handles game over/finished.

Highscores

Contains the highscores for each player and level. It's also capable of writing the data to a binary file and reading them back in. This is a static class because we only need this as a general functionality (meaning we don't need to instantiate objects from it). It's not a singleton because we don't need it to be on a gameobject in the scene.

HighscoresRepository

Handles access to the files and PlayerPrefs that contain the saved highscores.

IDamagingObstacle

This is an interface that other scripts can implement to work as an obstacle which deals damage to the player. The PlayerController just checks if the player is colliding with an object that has implemented the interface. That way, you can add new types of obstacles without the need to add checks for it in the player controller. (For example the player controller does not check for ElectricBarrier or Explosion objects, it just treats both of them as IDamagingObstacles.)

LevelInfo

Stores the meta information about each level such as level name, scene name, which tiles are used and which music should be played. The GameManager has a list of these objects to guide the player through each level.

MainMenuPanel

Used on the main menu at the start of the game. Contains the functions of the menu like showing the highscore panel and cycling through the highscores of each level. It also shows the name input panel for the player and asks the GameManager to start the game once the player has entered their name.

MapGenerator

Generates a texture from tilemap data. It will read the floor and obstacle tilemap and paint different pixel colours for different tiles. That way, the player can see a minimap when that extra is activated.

ParalaxMovement

Forces a gameobject to move with the camera to an adjustable degree. This is used on the background tilemap to create a parallax effect.

PlayerController

Used on the ball the player is moving. Deals with movement, collision detection, picking up objects, etc. It's quite long but allows for greater control of the interaction between the different mechanics e.g. what happens in what order.

It's not considered best practice to have huge scripts, but the PlayerController script also shows that you don't have to be intimidated by "best practices" if you are starting out. You will learn about those practices and why they are handy on the way. You can always come back and restructure your code later. And there are examples of very successful games that have awful huge scripts like *Celeste*. Check it out here:
https://github.com/NoelFB/Celeste/tree/master/Source/Player
(Disclaimer: this is not an encouragement to write huge scripts! ;))

ReplaceTilesWithGameObjects

Used to substitute certain tiles on a tilemap with gameobjects. These will be placed on the same position as the tile but may provide more functionality than a simple tile. This method allows you to paint those tiles in the tilemap editor instead of placing them by hand.

ScaleOverTime

Simple script that scales the gameobject it's used on over time. In the game it is used on the parachute to simulate its inflation.

SlidingDoor

Used on the sliding doors on the map. In addition to dealing damage, it triggers special movement when the player gets moved by the doors. So it's not one of the IDamagingObstacle objects because the PlayerController needs to know exactly what kind of object it is.

SoundManager

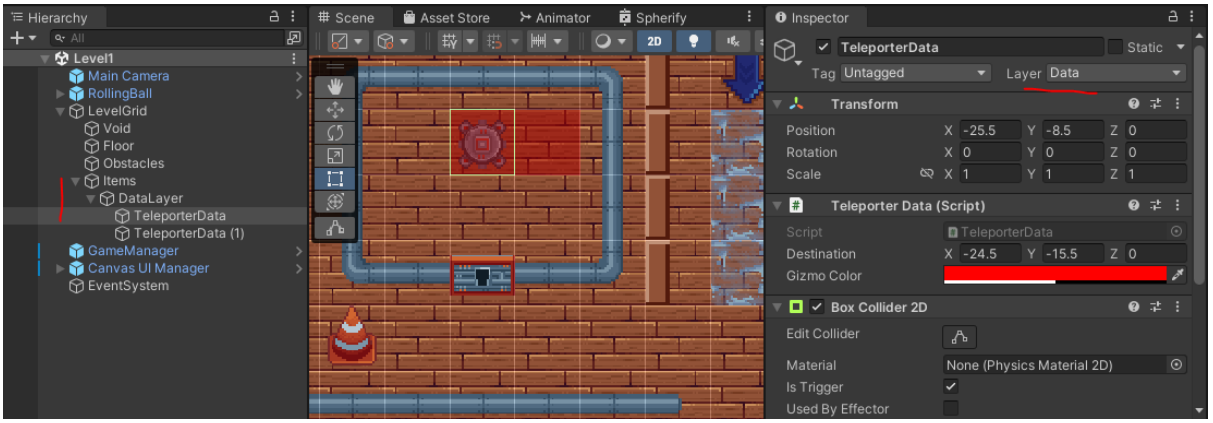
Used to play music and sound, can be called from any script due to its singleton design.

Tags

Static helper class to contain all used tags in one place to avoid having string literals all over the code.

TeleporterData

Contains data for teleporters, mainly the destination of a teleporter. GameObjects with this script need to be placed under the data layer under the items tilemap and also need to be on the physics layer "Data" so the game can find them when the player hits a teleporter tile. Since tiles can't store any data, this is one way to solve the problem.



TileCollection

Scriptable object that bundles all tile assets for a level. Is used in LevelInfo to let the PlayerController know for which tiles to check in a certain level.

tilemapBoundsCompressor

Just a helper class to reduce the dimensions of a tilemap. The dimensions of a tilemap will grow when tiles are added but don't shrink automatically when tiles are removed again. Can be run from the editor.

TileSwapper

Used to swap a tile once the player enters it. Can be used with a sequence of tiles that can also be looped. You can, for example, make a counter with it that the player can set to a certain number. It was used for the cracked floor tiles at one point, but the behaviour of those got too complex for the TileSwapper script (e.g. the tiles only change when the player stays on them for a certain amount of time) so it is unused at the moment.

TubeData

Similar to the TeleporterData, it was used to store a tube path the player would ride along when entering a tube. The code used for it is still in the PlayerController so you can see how that worked. The new version does not need data but instead just looks at the tilemap to find the next tube tile for the player to go to.

UIManager

Handles all UI tasks of a level. It holds references to the TextMeshPro texts and the images used in the UI and updates them as needed (e.g. advance timer or show the selected extra). It also shows the minimap and the game over or respawn panels.

Lemurs

Lemurs is designed for a resolution of 640x480 pixels so please set the aspect ratio accordingly!

Gameplay

The goal is to lead all Lemurs through the level to the exit. The Lemurs themselves don't know what to do and just walk until they hit an obstacle and change direction after that. The player can assign several tasks to Lemurs to help them. The available tasks and their available number of usages are shown in the lower UI. The player can select a task and, after that, click on a Lemur to assign the task to it. It's also possible to lower or raise the spawn rate of Lemurs, to pause the game or to explode every Lemur as a last resort.
The buttons in the UI are in order:

- lower spawn rate (hold mouse button for faster adjustment)
- raise spawn rate (hold mouse button for faster adjustment)
- assign climber task so a Lemur can climb walls
- assign floater task so a Lemur can survive long falls
- assign exploder task so a Lemur will explode and remove bits of the level as well
- assign stopper task so a Lemur will act as permanent obstacle hindering others to pass
- assign builder task so a Lemur will start to build stairs to reach unreachable places
- assign basher task so a Lemur will start to bash a tunnel in a horizontal direction
- assign miner task so a Lemur will pull out a shovel and dig a tunnel with a slope downwards
- assign digger task so a Lemur will start digging downwards vertically
- pause game
- explode all remaining Lemurs (needs a double click)

There is also a minimap on the right side of the UI which the player can click on to navigate in the level. Moving the mouse to the edge of the screen will also navigate the camera in the respective direction.
When no Lemur is left in the level, be it through escape or demise, the results are shown, and depending on the performance of the player, the next level can be played.
Hitting ESC brings the player back to the main menu at any point of the game.
Hitting SHIFT+ESC brings the player back to the game selection menu where one of the three games of this project can be chosen.

Scenes

Despite being a game with potentially hundreds of levels, there are only two scenes that are needed in the game: the *MainMenu* and *LevelScene*.
The *MainManu* just shows the title screen and lets you choose to start the game or enter a code to start a specific level.
The *LevelScene* is a universal scene that can display a level defined by some data stored in scriptable objects. It's done this way because every level has the same functionality and same structure of game objects in the scene. Having only one scene for all levels makes it much easier to change levels. Imagine you had 100 levels, each in its own scene. You then decided on a new feature where you had to add a gameobject to the level. You would need to add it in 100 scenes which might be very tedious.
Instead, everything that defines a level is stored in scriptable objects. So every level results in one scriptable object of the type LevelData. The image of the level, the time limit, the available orders, etc. are stored there:

Once the level scene loads, the LevelManager reads all the data and makes the necessary adjustment to the scene.

You might notice that both scenes have a GameManager although it is supposed not to be destroyed during scene changes, meaning if you arrive in the level scene coming from the main menu, the GameManager is still available there.
The reason for the additional GameManager is so that you can test the level scene without coming from the main menu. It does not complicate things because if you arrive from the

main menu, the GameManager of the level scene will detect that there is another GameManager available and will destroy itself so you don't have two of them in the level scene.
To keep changes on the GameManager gameobject in sync between the scenes, it is a prefab.

Game Flow

The game starts with the *MainMenu* scene where you can start the game or select a level by entering a code. Both actions will load the *LevelScene,* the only difference being that entering a correct code will set the internal level number that is to be loaded accordingly. Once the LevelScene is loaded, the LevelManager will ask the GameManager about the data for the current level and will set the level image and initialize all values for the level. It will then display an intro screen for the level and, after a click from the player, will start the gameplay. Once the time is up or all Lemurs have escaped or are dead, a summary screen is shown with the results of the level. After that, the LevelManager tells the GameManager to load the next level.

Main Features

The main feature of the game is the fully destructible level that is made of a freely painted image. It doesn't use colliders but instead checks single pixels where black pixels are considered to be free of obstacles (hence the black background). If you want to have a more colourful background, you can make the black parts transparent and change the collision detection from black to transparent pixels. After that, you can add a second image behind the level. That way, you can have a colour ramp or some other pixel art in the background. The level image is imported with 1 pixel per unit so each pixel in the image is one Unity unit. This helps greatly to read exact pixel values from the image texture.
With that method of collision detection, it's possible to react instantly to level changes, for example, if a new hole is dug in the ground or any stairs are built.

The other bigger feature is handling multiple independent units (Lemurs) that all follow their own routine (walking or player-assigned task). You can call it some kind of AI (though I wouldn't stress too much on "intelligence" here ;)). No Lemur knows of the other. Instead, the LevelManager knows about every Lemur and decides what to do if one leaves the level or if a new one needs to get spawned. An extension of the game could be Lemurs being able to talk to each other which would lead to more unpredictable and possibly funny results.

A minor feature is the custom font used in the game. It's working with TextMeshPro but is based on a bitmap font where each letter is a sprite in a larger spritesheet. It's quite tedious to set up since you have to create settings for every letter, but since the result needed some work, I decided it could be of good use for you to put it in the game. You can find some discussion on that topic in the Unity forums as well (https://forum.unity.com/threads/creating-a-pixel-font-from-a-bitmap-image.1260656/).

Scripts

The following list is a brief overview of the scripts and their respective roles used in the game. The exact details of each script are commented upon inside the scripts themselves.

CameraMovement

Used for the movement of the camera depending on the mouse position of the player.

EnterCodeMenu

Used for the submenu where the player enters a code to jump to a certain level. This could be included in the main UI script as well but got a bit more complex due to the fact that input fields and custom formatters (which we need to check if ten letters have been entered) don't work well with a custom bitmap font.

ExtendedButton

Used to handle all possible input on buttons that the standard button can't handle. It builds on the Unity button and adds the possibility to trigger the action of a button repeatedly while holding the mouse button down and also supports double-clicking on a button.

GameManager

Small manager to handle the overall flow of the game. Stays active in all scenes and knows about the available levels, how to load a new level and how to quit the game. In contrast, the main gameplay of each level is handled in the LevelManager.

IAbility

Interface for the different abilities a Lemur can have. It is implemented by several classes like FloaterAbility, ClimberAbility, DiggerAbility, etc. Those encapsulate all the logic a Lemur has to perform when the according ability was selected for them.

Lemur

Contains everything that a single Lemur needs. It updates the behaviour of a Lemur, checks for collisions in its way, changes animations depending on the state, and is used to give orders/tasks to a Lemur.
It works closely together with the IAbility classes mentioned above.

LevelData

Scriptable object that contains everything that defines a level like the level image, time limit, available tasks, etc.

LevelManager

Deals with the game flow inside a single level. It will load the level data from the GameManager, set everything up, handle spawning and deleting Lemurs, keep track of the time and provide service functions to check level data like obstacle occurrence.

LevelUIManager

Deals with everything UI related inside a level. It has functions for the button clicks such as selecting a task, displaying level information, handling the minimap, changing the mouse cursor, providing a method to fade in and out panels, etc.

Map

Provides methods to manipulate and evaluate the map data for example to remove pixels when digging or to find the next free spot where a Lemur can walk.

MapData

Holds a copy of the level image and the pixel values in an array. This is what the LevelManager is operating on so our original image is not getting changed. Also, having the additional array allows for a more efficient reading of the level data.

Minimap

Handles the drawing of the minimap and also adjusts the camera depending on where the player clicks on the map.

MouseCursor

Helper script that changes the sprite of the mouse cursor. Is used by the LevelUIManager.

SoundManager

Used to play music and sounds. You can set one AudioSource for music and multiple AudioSources for playing sound effects, depending on how many you need simultaneously. Note that the game has no sound effects so far, check RollerMaze for that.

UIManager

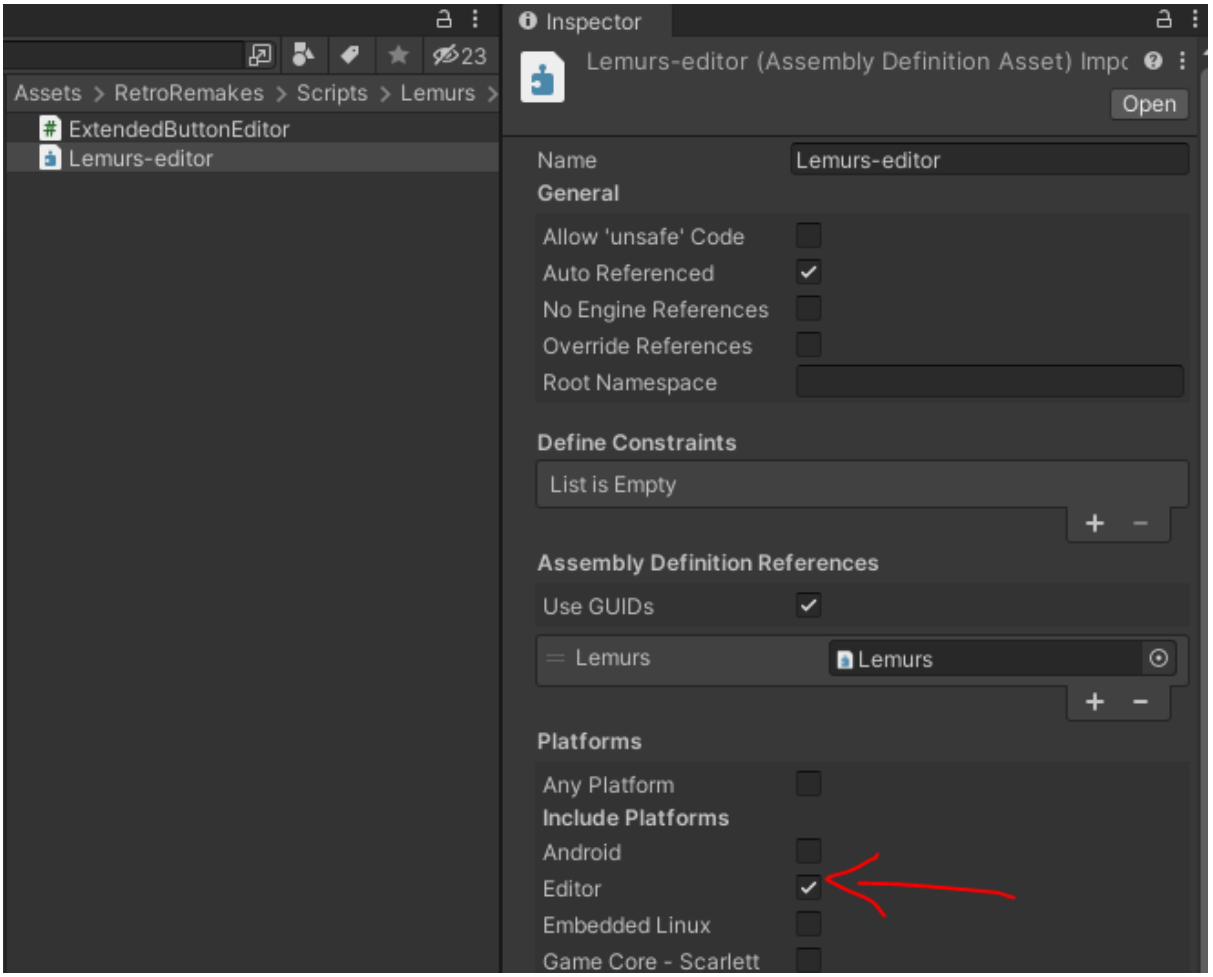
Very small script that deals with user input in the main menu.

Editor/ExtendedButtonEditor

Needed so that Unity shows our ExtendedButton in the inspector even if it only calls for drawing the default inspector.

.

Note: the script is in its own assembly. That is needed because we are using our own assembly for the game, and Unity does not separate editor scripts out when building the project so this would end up in OUR assembly without having access to UnityEngine namespace which is not a problem in the editor but in the final build. So we need to put this script in its own assembly and only have it enabled in the editor.
More info here: https://stewmcc.com/post/unity-assembly-definitions/


Inspector
Lemurs-editor (Assembly Definition Asset) Impc
Open
Assets > RetroRemakes > Scripts > Lemurs >
ExtendedButtonEditor
Lemurs-editor
Name
Lemurs-editor
General
Allow 'unsafe' Code
Auto Referenced
No Engine References
Override References
Root Namespace
Define Constraints
List is Empty
Assembly Definition References
Use GUIDs
Lemurs
Lemurs
Platforms
Any Platform
Include Platforms
Android
Editor
Embedded Linux
Game Core - Scarlett

Oil Tycoon

Oil Tycoon is designed for a resolution of 640x400 pixels, so please set the aspect ratio accordingly!

Gameplay

Oil Tycoon is a typical game of the management genre that was very popular in the 90s (at least in Germany). The player is a manager of an oil company and needs to compete with three other (human or computer) players to win the game. The game is played in hot-seat mode, meaning players will take turns in interacting with the game after each other.
The tasks a player can perform are, for example:

- buy oilfields and tanks
- sell oil and oilfields
- sign contracts with customers, agents who sabotage other players and detectives who protect the player from sabotage attempts
- show detailed statistics about the own company and global data like market shares and oil prices
- play minigames to drill for oil and lay pipelines to get oil delivered successfully (see chapters below for more details)
- read the newspaper to gather information about the market
- check their phonecalls for important messages

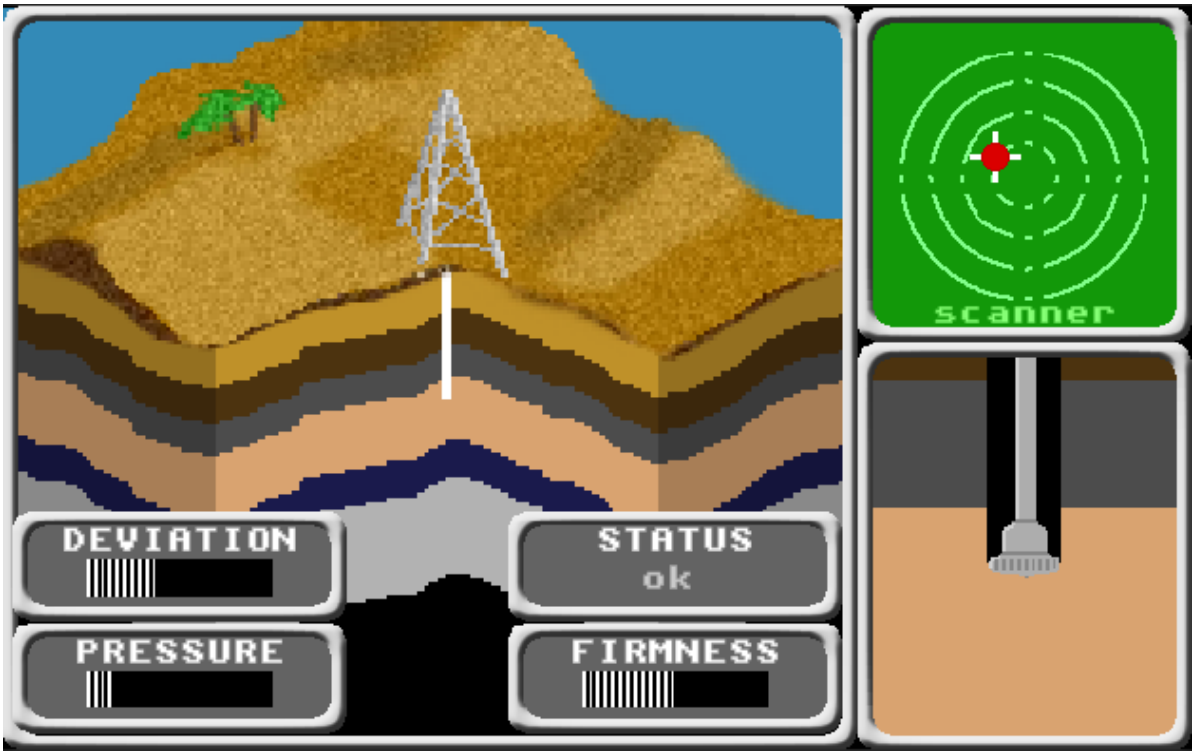
The goal for winning the game can be set before the game starts to give greater replayability.

The game is turn-based. Each player will play one in-game month where their actions take a varied number of days to be performed. When the month is over or the player decides to end their turn by clicking on the golden calendar at the bottom of the screen, the next player will play the same month. When every human player has finished their turn, a summary screen for that round will be displayed. The computer players are simulated and the current standings are displayed. A news ticker will also report the activities each player has performed (except the shady ones).

Unless the conditions for winning the game are fulfilled by one or more players, the next round gets started. Otherwise, the winner is announced and a new game can be started.

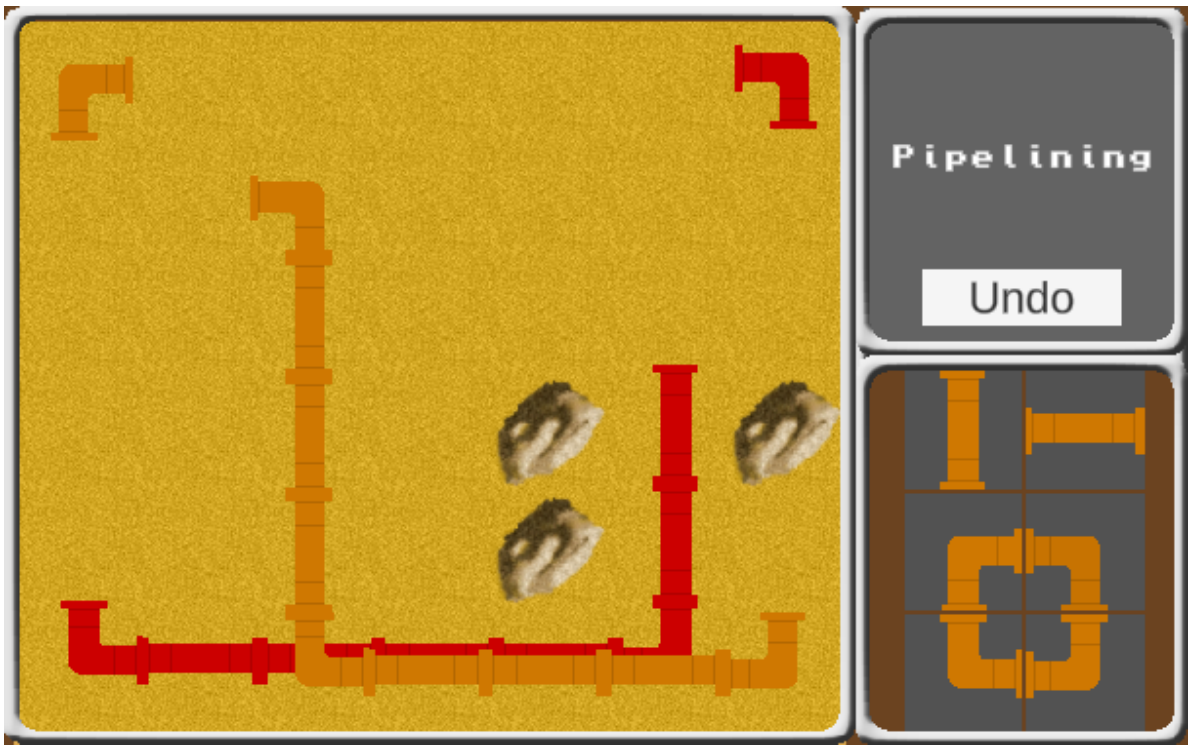
Hitting SHIFT+ESC brings the player back to the game selection menu where one of the three games of this project can be chosen.

Oil drilling minigame



This minigame will appear when the player chooses to drill for oil and does not want to spend the money on a contractor. The game starts on entering the screen. As you can see in the screenshot, a white line indicates the player's progress toward the oil at the bottom.
On their way, players have to pass several layers of earth, each having its own firmness. More firmness puts greater stress on the drill and makes it harder to control. The player has to use the arrow keys to keep the drill steady. This is indicated by the scanner in the upper right corner. The red dot has to be kept close to the center, otherwise, it might break and the drilling has failed.
To regulate the stress on the drill, the player can increase or decrease the pressure of the drill using the X and Y keys. Increasing the pressure will accelerate the process but causes greater instability for which the player has to compensate. Decreasing the pressure will make controlling the drill easier, but it will also slow down the process which means more days and more money are spent on the endeavor.
In the lower right corner, the player can see a magnification of the drill and the earth layer in which it is located.
Once the drill reaches the oil, the drilling has succeeded and oil will be pumped to the top. After that, the minigame ends, and the player returns to the buying screen.

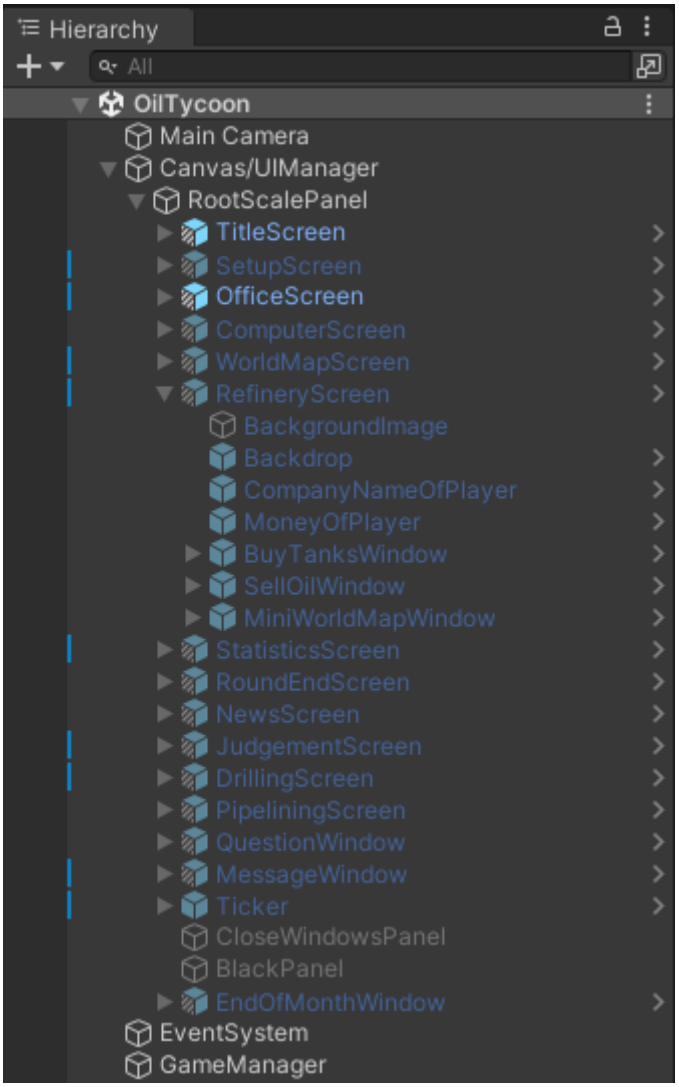
Pipelining minigame



This minigame will appear when the player tries to sell oil but the game decides to tell the player there is a problem with the pipeline (the probability of that is at 50%).
The goal is to connect the orange pipe in the lower right corner with the pipe in the upper left corner by laying pipes between them. The player has to watch out for obstacles like rocks which have to be bypassed. The player has a computer opponent who tries to achieve the same goal using the red pipes. The pipes of the opponent are not considered obstacles. Once the player has reached the final pipe, the game ends showing a success message. Should the computer outrun the player, the game is considered a failure, and no oil can be sold.
Laying pipes can be done by clicking the buttons on the right side of the screen. Only pipes that fit together will work here, of course. If the player makes a mistake, the undo button can be used to remove pipes in the reverse order of their placement.
The game will not start immediately to give the player a chance to plot the right path. Starting the game is then achieved by laying the first pipe.
When the minigame ends, the player is taken back to the selling screen.

Scenes

The game uses one scene for all the content. As a game that is working exclusively on the UI, apart from the GameManager and the Camera, in Oil Tycoon there is only a Canvas object containing all screens and windows which are shown and hidden by the game logic and the player interaction.



There are different philosophies to organize UI content such as using multiple canvases or even using multiple scenes so you can do some research to find out more. That way, you are perfectly equipped to make sound decisions for your own project.

Game Flow

The game starts with the title screen after which the player is presented with a number of windows where they can set up the game. Entering the number of human players, choosing a company, and setting the goal of the game is some of the information that needs to be entered there.
After the setup, the office screen is shown to the first player. From here, they can branch out to several subscreens to accomplish their needs. If a player is finished with their round, the next player will be able to do the same. If there are no more human players left, a summary screen is presented. When a player has won the game at this point, the game ends and can be restarted. Otherwise, a new round is started with the first player starting in their office screen again.

Main Features

The most obvious feature of the game is that it is purely made of UI elements. That might be a bit unusual in a 3D game engine like Unity but shows the wide variety of games you can make. Each screen and window is built from UI elements. For that to work efficiently and not to lose track of the many elements, the game for one uses prefabs and variants. For example, windows all have the same base look which can be changed by changing the underlying prefab. In the code, there is also a base class for the windows that ensures the same basic behaviour for all windows that are derived from it (the same goes for the screens).

Another feature is the use of events to generate the desired flow of the game. This is used for the windows where you can, for example, subscribe to events to trigger actions that should happen before a window gets opened or after it gets closed. The need for that came up after the windows got animated, and subsequent windows should only be opened when the closing animation was completely finished. After that, many more events were added to support special use cases.

Being a management game, Oil Tycoon there is a lot of data to store: economy data like price and demand, oilfields and their status, which sector is owned by which player etc. For that reason, there are several scripts/classes that are mainly used for encapsulating that data. They are not mono behaviours so they don't exist in the scene but they are "pure" classes that are created during the game (example scripts are Player, Oilfield, and Contract).

Having much data also poses the big question of what to do with it. The flow and many calculations of the game rely, of course, on the data. Be it to decide who won the game or to calculate how much oil a player produces in a certain number of days. The game uses LINQ (language integrated query) to ask questions about the available data in many places. LINQ provides extensions for collections that make this task very easy and intuitive. You will find examples of varying complexity throughout the game. Evaluating simple questions like "Does the player have a contract with a detective?" will be shown as well as more complex ones like "How much oil do the oilfields produce in a map region in x days, considering that they have not dried up?"
To learn more about LINQ, check out this link:
https://www.tutorialsteacher.com/linq/what-is-linq

Scripts

Since Oil Tycoon contains many more scripts than the other games, I will try to group them together in this documentation. These are most notably scripts derived from the classes Window and Screen. At the end of the list, there is also a compilation of several scriptable objects used to store data. Each script contains detailed information in the code.

AnimateRamp

Animates the _fade parameter of the RampShader.

Calendar

Used to simulate the passing of time in the game. Contains class definitions for Date and Month. Time can be passed by advancing the calendar. The flow of the game makes it necessary that the Calendar also fires events when the date gets changed and when the end of a month is reached.

CalendarRenderer

Shows the current date in a text field. Is subscribed to the CALENDAR_CHANGED event so it gets notified about any change of the current date to refresh the text field.

CompanyLogoOfPlayer

Used to show the logo of a player who can be specified. Gets refreshed every time it is enabled. For example, it always shows the logo of the currently active player in the office screen automatically.

Contract

Base class used to store data of a contract the player can sign. Objects of that type are created while the game is running. Scripts that use this base class are AgentContract, DetectiveContract, and SupplyContract.

DistortedImage

Adds a distortion to an image. Works with shearing and a small amount of perspective distortion.

DrawLineTexture

Used to draw lines on a given texture. Can be given a list of data points to draw a line chart as seen in the statistics screen of the game.

Economy

Simulates the economy in the game. Contains data about price and demand for each region of the map. The data is calculated for each month the players are playing. Some economic events like recession or war can be set to influence the calculation.

EventManager

Central point for scripts to subscribe to given event types and also fire events of those types (e.g. CALENDAR_CHANGED). The advantage is that no script needs to know who fires events and who is interested in them. May be a bit overkill for the game, but the smallness of the EventManager makes it easy to learn about its principle.

GameManager

Has knowledge of the game state and data like what players are playing and what map regions exist in the game. Also deals with the flow of the game like initiating a new round, checking the winning conditions, advancing player contracts, etc.

InfoTextOfPlayer

Similarly to the CompanyLogoOfPlayer, it shows textual information of the player. The player number and type of text can be configured (e.g. money of the current player).

Judgement

Stores the data for a judgement the player receives after a failed sabotage attempt. Is created by the GameManager as needed.

MapSector

Holds the information of each map sector of the game (there are 8 regions with 24 sectors each). The MapSector objects are created at the start of the game by the GameManager using data from the MapRegionData scriptable objects.

Oilfield

Is created when the player is able to drill successfully in a map sector. Holds all the information about the oilfield and is added to the list of oilfields in the Player class.

Player

Has all the information about a player and provides methods on how to interact with the data like adding new oilfields to their possession, sending phone calls, calculating maintenance costs for all assets of the player, etc. In contrast to the GameManager, the Player script does not know about other players, so every interaction between players is handled by the GameManager who calls methods of the players involved.

Screen

Base class for all screens in the game. Provides default functionality for showing and hiding a screen. Also provides several events you can subscribe to in order for actions to get triggered at some point in the screen's lifecycle like before showing or after closing a screen. Also contains virtual functions that can be overridden by child classes to customise the behaviour of each screen.
A child class of Screen usually contains references to all UI elements and all the functionality used in the screen like methods that are called by buttons on the screen etc.
Examples of the many screens in the game are DrillingScreen, OfficeScreen, and RoundEndScreen.

ScreenProvider

Keeps a list of all screens and delivers them when needed via the UIManager.

SectorSelector

Each sector in the map of a region contains a SectorSelector that detects mouse clicks and informs the RegionMapWindow about the sector that has been interacted with.

SelectableItem

Works similar to a button but also marks the item the player has clicked on. Used for example in the AgentActionWindow to choose between four possible actions the agent should execute or in the SelectMissionWindow to determine the goal of the game during the setup process.

Tank

When the player buys a tank, a Tank object is created based on the TankData the player selected. It is also associated with a map region so we can evaluate how much storage is available in each region.

Ticker

Shows messages at the bottom of the screen as text running from right to left. Can get started by using the UIManager.

UIManager

Organises all windows and screens and grants access to them. Also handles modal windows and fading between two screens for a smooth transition. Stores a list of sprites that are used for different icon types that can be used in the code (e.g. configuring the MessageWindow with an icon that fits the shown message).
Keeps a list of Windows and Screens by having a WindowProvider and a ScreenProvider.

Window

Base class for all windows in the game. Provides default functionality for showing, closing, and animating them. Also provides several events you can subscribe to in order for actions to get triggered at some point in the window's lifecycle like before showing or after closing a window. Also contains virtual functions that can be overridden by child classes to customise the behaviour of each window.
A child class of Window usually contains references to all UI elements and all the functionality used in the window like methods that are called by buttons in the window etc. Examples of the many windows in the game are QuestionWindow, RegionMapWindow, and SellOilWindow.

WindowProvider

Keeps a list of all windows and delivers them when needed via the UIManager.

Scriptable objects

There are several scripts that can be used to create scriptable objects that contain data used by the game. You can change the assets that are already created, or you can create new ones to extend the game. These scripts are:

- AgentData and DetectiveData: describe which agent or detectives are for hire along with their capabilities
- CompanyData and OfficeData: describe the companies and offices the player can choose from at the start of the game. They include the starting money, the logos/background images, etc.
- MapRegionData: describes the regions on the map including how much oil is to be expected there and how much oilfields cost
- TankData: describes the tanks the player can buy, including their capacity and price
- EarthLayer: describes the different layers the player will drill through when drilling for oil
- EarthLayerConfiguration: describes a group of earth layers, which can be used in the MapRegion to set the typical earth layers found in a region of the world

Common Scripts

There are a few scripts that turned out to be quite universal and could be applied in each game so they are in a common assembly and can be used in every game.

Extensions

Holds all extension methods of external classes.

GameManager

The overarching game manager that is responsible for starting the game the player selects from the menu.

PointerClickDetector

Detects mouse clicks on UI objects. There are three events where actions can be added that can be triggered depending on the type of click (i.e. which button was pressed).

SceneDescriptor

Encapsulates a scene name so it can be referenced in the inspector and is much more maintainable than using the scene name as a string directly in various places would be.

SpriteSwitcher

Can animate a given list of sprites in a configurable order. Usable for Image and SpriteRenderer components. Refreshes a PolyCollider2D for each change if present.

ToggleActive

Switches a configurable gameobject between active and inactive on a fixed interval.

UppercaseValidator

Scriptable object that can be used as a validator on TextMeshPro input fields. Does not really validate the input but will convert every entered character to uppercase.

Where to Go From Here?

As suggested at the beginning, after having read the documentation and maybe watching some videos, you should start playing the games to get to know them well enough that the code makes more sense to you. You should then decide on one game that interests you the most and start opening the scenes and take a look at how they are structured. Then you can take a deep dive into the code. This is probably the hardest part because you not only have to understand the code per se but also my thinking behind it. I hope I left enough comments to make sense of my decisions. So don't get frustrated and take your time. The best entry point in each game would be to look at the GameManager or LevelManager (or the manager scripts in general). Then go into the details, maybe try to find a feature you have seen in the game and you want to know more about. Feel free to modify the code and restart the game to see what happens. That is the best way to learn. You may fail at times, but you are always failing forwards ;). At some points in the code, I left suggestions for you to add functionality. These may be a bit more complex, so take your time.

Last Words

Thank you so much for buying this project! I hope you are having fun with it and you will learn some things from it. And to be clear: you may use all the code you find in the project for your own stuff, be it a personal or commercial project. I would be honoured ;)

I wish you all the best for your own projects! I would love to hear about them in the Discord.