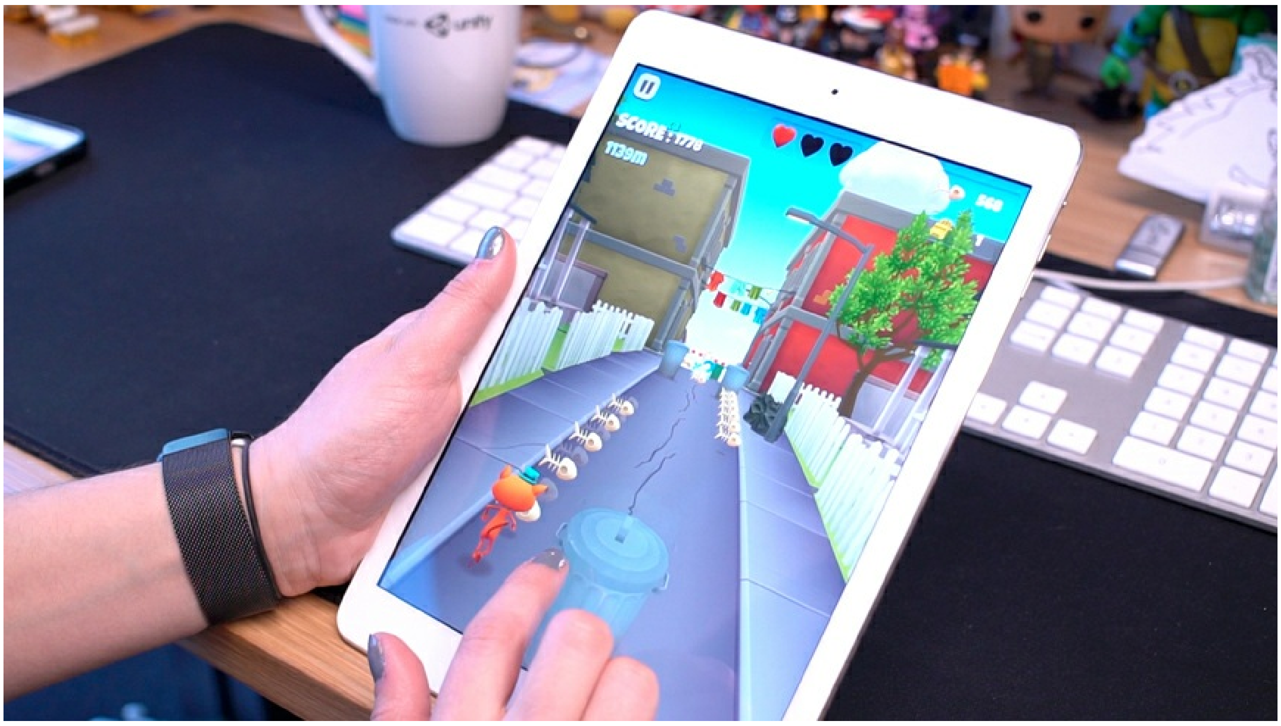
Mobile Development Techniques

learn.unity.com/tutorial/mobile-development-techniques

1.Trash Dash code walkthrough

Introduction

Trash Dash is a game that we created as an example of how to use Unity Services in a mobile game. It also contains some useful examples of coding architecture and techniques for solving common game programming problems. In this article we'll take a quick tour of the code for Trash Dash and learn about these features.



If you haven't already downloaded Trash Dash, you can get it here on the Unity Asset Store:

Endless Runner - Sample Game

免费

立即购买

Unity Services

Trash Dash uses Unity Services to take care of ads, analytics and IAPs. As well the services we can see in the finished game, our team used Unity Collaborate and Unity Cloud Build behind the scenes to build Trash Dash.
Using Unity Services makes it quick and easy to get these important features working: we only have to configure a few details in the Services Dashboard and write a small amount of code.

To see the code used to integrate IAPs, check out IAPHandler.cs. To see the code used to handle ads, see AdsForMission.cs and UnityAdsInitializer.cs. Analytics code can found throughout the project, but some good examples are found in GameManager.cs.To learn more about setting up Unity Services, see this section of the Unity Manual.

Architecture

When we talk about the architecture of a game, we mean how the code is organised. Good architecture can help us to avoid problems such as poor performance, long load times or code that's difficult to maintain.

The best architecture for any game depends on the game itself. An online multiplayer game that is being worked on by a large team has very different needs to a single-player offline game being worked on by one person.

Let's take a look at a couple of key points from Trash Dash's architecture. We will examine how the code is organised and what problems this solves.

The Game Manager

GameManager.cs is the class that controls the overall flow of the game.

GameManager uses a design pattern called the Singleton. The Singleton pattern is a way of ensuring that there is only ever one instance of a particular class in the game. This is useful for when a class has an important responsibility, such as GameManager. It also makes it easy for any other class to access this instance using a public static reference.
GameManager is also an example of a Finite State Machine. A Finite State Machine ensures that the game can be in only one state at any given time, and manages what happens when this state changes. The three possible states that Trash Dash can be in are: displaying the start menu, playing the game and displaying the game over screen. A Finite State Machine is a common way to manage how things behave in games, from the overall flow of the game to the behaviour of individual characters.
To see the code for the Singleton and Finite State Machine, explore GameManager.cs and read the comments. The code for entering, executing and exiting the three states can be found in the following files: LoadoutState.cs, GameState.cs and GameOverState.cs.

Object pooling

In Trash Dash, hundreds of coins may be spawned in a single play session. Instantiating and destroying a great many objects at run time can be a strain on performance: it can involve relatively costly code, and it can lead to frequent and time-consuming [garbage collection](https://en.wikipedia.org/wiki/Garbage_collection(computerscience)).

To reduce this overhead, Trash Dash uses a technique called object pooling. Object pooling is a technique where objects are temporarily deactivated and then recycled as needed, instead of being created and destroyed.
When the game begins, a number of inactive coin GameObjects are spawned and placed in a "pool". When a new coin is needed, one is requested from the pool and enabled. When a coin is collected or leaves the screen, the coin is disabled and returned to the pool.

To see the object pooling code in Trash Dash, take a look at TrackManager.cs, Coin.cs and Pooler.cs. For a general guide to object pooling in Unity, see this tutorial on the

Unity Learn site.

Techniques

While architecture decisions usually affect lots of classes or parts of our game, techniques are smaller in focus. Techniques might only affect a single function or file, but they can still help us to solve problems.

Let's take a look at a couple of techniques used in Trash Dash and see what problems they solve.

Origin reset

In any game where the player travels great distances - such as space exploration games or "infinite" games like Trash Dash - the developer must make a decision about how to handle the player's position. If we simply move the player GameObject, the values in the player's transform.position will get higher and higher over time. This can lead to problems due to something called floating point imprecision.
Floating point imprecision means that the larger the value of a floating point number, the less precise it is. This is a limitation of how computers store numerical data and is not unique to Unity. In a game with a large or infinite playable area, the floating point numbers used to store position could become large enough to cause problems. If GameObjects have imprecise values for their position, they may appear to move around, flicker or pop in and out.

There are several ways of solving this problem, and which one is best depends on the game. In Trash Dash, we solve this problem by using a technique known as origin reset. This means that once the player has moved a certain distance beyond the origin of the world (i.e., the world position 0, 0, 0), we move everything in the Scene back towards the origin. This ensures that the values used for positions always stay low and therefore are not prone to imprecision. The origin reset happens seamlessly and the player is not aware of it.

To see the code used for the origin reset technique, take a look at TrackManager.cs.

Curved world shader

In Trash Dash, we create an endless track by spawning track sections ahead of the player and removing them when they are behind the player. If the player could see a long way ahead, we would need to spawn a lot of track sections in advance. This could lead to performance problems.

In addition to this, the world is full of coins, obstacles and characters. Again, if the player could see a long way ahead of them, Unity would have to draw all of these objects to the screen long before they are close enough to interact with. This could also lead to performance problems, particularly on mobile.

To solve this problem, the the world curves away from the player with a horizon. This creates the illusion of an infinite world, hides the spawning of track pieces and means that we don't have to spawn coins and obstacles until the player is near them.

If we examine the Scene, we can see that the actual geometry for the world is flat, not curved. The curved effect is created by a shader. A shader is code that tells Unity how to draw an object to the screen. In this case, the shader calculates what the level would look like if it were curved, then tells Unity where to draw the pixels to the screen in based on that calculation.

The files to examine to see how this works are WorldCurver.cs, and CurvedCode.cginc. Shader code can be a little tricky to read at first, but this page of the Unity Manual is a helpful guide to reading and writing shader code.

Further reading

There are a few more interesting examples of approaches and techniques in Trash Dash. Take a look at the shader used to rotate the fish, the way that AssetBundles are used to load characters and themes and the way the player data is saved.

To learn more about performance optimization, read these articles on Unity's Learn site. To learn more about making mobile games using Unity, take a look at this section of the Unity Learn site.