X-Racer Documentation

Thank you for buying X-Racer, an advanced 3D game template for Unity.

Overview

X-Racer features high-speed action over an infinite scrolling terrain. Players can collect credits as they race, then use them to purchase items in the integrated shop. X-Racer features a flexible procedural generation system, a multi-screen menu system, looping intro and game music for you to use, and a complete set of models and textures for you to use as is or customise for your own style.

Example Scenes

X-Racer comes with 3 example scenes. The default scene uses custom models and lightmaps to get a minimalist, futuristic look. The mobile scene is a mobile optimised version of the default scene with simpler materials and no full-screen effects. The Scifi scene is specifically built to be easy to customise. It uses simple meshes and no lightmaps, allowing the textures to be replaced very quickly for a totally different look.

Level generation

X-Racer features a flexible procedural generation system that allows the level designer to control the flow of the level through various sections, varying the difficulty and creating interest to keep the player engaged.

Infinite Scrolling System

In order to support an infinitely generating world, we can't simply move the player forward through the scene. Due to the limited precision of floating point values, beyond about 10,000 metres, there isn't enough precision between values to accurately represent the geometry. If we kept moving the player forward as they flew, then once they are more than 10,000 units from the origin (at the default speed about 1.5 minutes in to the game), then physics calculations would start to become unstable, the rendering might appear to stutter and all kinds of other artifacts would arise.

For this reason, instead of moving the player, we simply scroll the contents of the world past the player (who remains stationary at the origin). In this way we can support an infinitely sized world without any floating point precision issues.

Using this system has a number of implications which will be discussed in more detail in the following sections.

Level Sections

The world as a whole is composed of multiple sections, with each section being made up from prefab blocks of 64x64 units. It is the **LevelManager** object that is responsible for generating all the blocks, scrolling them past the player, and then removing them from the scene again once they are out of view. Parented directly beneath the LevelManager are any number of Level-Sections that ultimately make up the flow of the level. Each section has a specified length, which determines how many blocks the LevelManager will generate using this section before it is considered complete, at which point the LevelManager will start generating blocks from the following section.

Each section should have a script attached that is sub-classed from **LevelSection** (see Scripts/LevelGeneration/LevelSection.cs). X-Racer comes with three such sub-classes that each

implement a different manner of selecting blocks to use. The simplest is **ConstantSection** - it always generates the same block. This section is used underneath the menus for example. The **RandomSection** class picks a block to generate randomly from amongst a set of potential blocks. Lastly, the **SectionSequence** class generates blocks in a specified order (using the same block for an entire row). You can easily create other types of section by sub-classing **LevelSection**, and have them generate blocks in any manner you wish.

Camera Motion-Blur and Full Screen Effects

Because the player and camera remain stationary, we can't just turn on camera motion blur. The camera doesn't move from frame to frame, so by default no blur will be calculated. Instead we require some tricks to make the camera think it has moved a certain distance each frame - in particular we use the 'isPreview' setting of the camera motion blur script and explicitly set the delta movement of the camera we wish to use for calculating motion blur.

This has the unfortunate side-effect of also blurring the player's ship, which we would like to avoid. To solve this, we render the player ship using a second camera - parented underneath the main camera. By attaching the motion blur effect to the main camera we can ensure that the player's ship remains crisp.

Any full-screen effects we want to apply to both the world and the ship (for example color-correction) should be placed on the PlayerCamera, if they should only apply to the world and not the ship, then they should be placed on the main camera.

Power-ups

Power-ups can be placed in sections of the level by attaching a **PowerupSpawner** component to the section objects (that is, the child objects of the **LevelManager**). This allows you to set the frequency at which they should spawn (where 0=never, and 1=spawn in every single block). You can also specify a set of power-up prefabs to randomly choose from, so you can limit which power-ups are found in different sections of the level. If you have two power-ups (say PowerupA and PowerupB), and you wish PowerupA to be spawned twice as often as PowerupB, then you can simply add PowerupA to that list twice, and it will be picked twice as often.

Power-up Placement

We must create a set of valid power-up spawn points for each block prefab, so we can avoid placing them in impossible to reach positions - such as directly in-front or behind an obstacle. To do this, for each block prefab we simply create a number of empty game objects and attach a **PowerupSpawnPoint** component to them to mark them as valid spawn point positions. You can look at the example block prefabs for reference on how to do this. Make sure they are positioned at a height that the player will collide with.

Power-up Hierarchy Structure

Each power-up prefab needs to have a structure something like this:

- PowerupPrefabObject - should contain a **PowerupCollectable** component, and a collider.
  - Visual representation object(s) - what the player sees
  - Subclass of **Powerup** - the item that actually performs the power-up if the parent object is collected.

If the **PowerupSpawner** decides to spawn a power-up prefab of a particular type, a new copy of the prefab is instantiated and placed at the spawn point. If the player collides with it, then the visual part of the power-up is destroyed, and the subclass of **Powerup** is notified so it can perform whatever it needs to do to make the power-up happen. In addition to having a **PowerupCollectable** component, the power-up prefab should also have a collider of some sort (it doesn't need to be especially accurate so a sphere is a good choice).

Creating New Power-ups

To create a new type of power-up, you will need to subclass **Powerup**, and override the **ApplyPowerup**() and **Cancel**() methods. **Cancel**() is called if the player picks up a new power-up before the current one has completed. It is important to implement **Cancel**() if your power-up alters a property over time (for example giving the player a speed boost for a short time), this will ensure that collecting two power-ups in quick succession won't cause issues.

Once you have your **Powerup** subclass, you'll need to create a power-up prefab with a structure as outlined above (a **PowerupCollectable**, with the **Powerup** subclass and any visual representation as child objects). Save that as a prefab and then you can reference it from any **PowerupSpawner** components you have attached to level sections.

Menus

The menus are controlled by the global **MenuSystem** component (by default this is attached to the **GameManager** object along with the other global manager components). This component maintains a list of all the menu screens and handles transitioning into and out of different screens.

Each menu screen is a child of the global Canvas object, and should have a **MenuScreen** component attached. It should also have a **CanvasGroup** component attached, to allow the screen to be faded out. Each **MenuScreen** can optionally have a default UI item, which is the item (usually a button) that will be given focus when the screen is opened. The **MenuSystem** component allows you to get a reference to individiual screens by name, so make sure you give any new MenuScreens an appropriate name.

Sound

Also attached to the global **GameManager** object is a simple **SoundManager** component. You can add sound effects and music clips to this component, then play them from any script by calling

```
SoundManager.PlaySfx(clipname)
```

or

```
SoundManager.PlayMusic(clipname)
```

Store

X-Racer comes with a simple built-in shop, where the player can exchange credits collected during the game for various items such as new ship skins, upgraded ships, and consumable items. The system is very flexible and new items can be added to the shop quite easily.

Creating New Shop Items

To create a completely new type of purchasable item, you will need to create a subclass of **ShopItem** and override the **Apply**() method. This function is called at the start of the game whenever your item is currently in the player's inventory. An item can be flagged as 'consumable' which means that the inventory count will be decremented after **Apply**() has been called, and the item will be removed from the inventory when the count reaches zero.

The maximum number of copies of an item that the player can own at once is specified with the maxInventoryCount parameter. For most things this will be 1, but for consumable items it may make sense for the player to be able to own several copies of an item.

To create a new purchasable ship or ship-skin, there's no need to create your own subclass of ***ShopItem****, you can simply use the* ***ShipSkin*** *component that comes with X-Racer.*

Attach your script to an empty game object, set any parameters you need such as cost, and whether the item is consumable. Remember to give each item a unique ID, which is used to reference it within the inventory. Now save it as a new prefab, we'll be needing this in a moment.

Next you'll need to create a new menu item in the store menu. You'll find all the store items under the global Canvas object - Canvas->StoreMenu->ScrollPanel->ShopItems. The quickest way to add a new one is simply to duplicate one of the existing items. Make sure it has a **ShopMenuItem** component (or subclass) attached. Set the itemPrefab parameter to the **ShopItem** prefab that you just created. That's it, you should now have your custom item available to purchase in the shop.

Storing the Player Inventory

By default, the items that the player owns, along with the total credits the player has, are all stored using Unity's **PlayerPrefs** class. Depending on the platform, this can be fairly easily edited by the end user, so if you plan to sell items for real money (or sell packs of credits for money), then you will probably want to implement a more secure method for storing this information.

Customisation

Swapping Textures

This is the simplest and quickest method of customising X-Racer. The Sci-fi example scene is specifically set up to make swapping the textures very simple and straightforward.

Open the XRacer-Scifi scene to start off, then in the project view navigate to XRacer/Models/Materials/Scifi - these are the materials we will need to change to customise the level. You will need some tiling textures to use (cgtextures.com is a good source for these, as is GameTextures.com). Simply drag your new textures over the base texture parameter in each of the 4 materials. Feel free to create new blocks and materials by duplicating and modifying the examples.

For a video tutorial showing you how to go about customising X-Racer by replacing textures, please see: https://youtu.be/zdrCt4cAZog

Altering the Sky Color

Navigate to XRacer/Materials and make a duplicate of the SkyboxConstant material. Next, open the Lighting window from the Window menu, and drag your new skybox material onto the Skybox parameter at the top. Now you can edit the Tint and Exposure values of your skybox material until you have a color you like. To make the fog match, in the Lighting window scroll down to the Fog options, click the color dropper next to 'Fog Color' and then click on the background within the scene view to select the sky color.

Creating New Level Blocks

Blocks should be 64x64 units in size (unless you change the constant kBlockSize in LevelManager.cs), with the origin at the center of the block. Remember that it is the blocks themselves that move, not the player, so don't mark them as 'static' in the inspector. This also means that concave mesh colliders won't work, so try to use simple box or sphere colliders where possible. Since the colliders are only used to determine if the player has crashed (and not for more complex physics simulations), then they can be fairly crude approximations of the geometry without major problems. Also since the player can never collide with the ground, the ground mesh(es) don't require colliders.

Creating New Ships

Creating a new ship is as simple as creating a prefab from your desired model. The model should face down the positive Z-axis, and as a reference to scale the default ship is about 3 units long and 1 unit wide.

Next, assign your new prefab as the modelPrefab on a **ShipSkin** shop item, so that it can be purchased in the shop. You can either modify an existing shop item or create a new one (see the section on creating new shop items for a more detailed description on creating new purchasable items, including new ships).

If you want your new ship to become the default ship that the player starts with, then also assign it as the defaultShipPrefab parameter on the Player, and make sure the shop item is marked as 'Own By Default'.

That’s all, I hope you enjoy using and modifying X-Racer and I wish you the best of luck with your game.