Thank you for your purchase! The most recent documentation (with better formatting) can be found online. If you have any questions feel free to post on the forum.

Overview

The Infinite Runner Starter Pack contains everything that you need in order to make an infinite runner game - scripts, models, textures, audio, shaders and animations. It is arranged to make everything as logical as possible. Any object that can appear in the game is placed within a prefab. That prefab is then spawned from a managing script. This script determines what objects to spawn based on its appearance probability as well as its appearance rules. Probabilities are used to say that at x distance the object n has a probability p of occurring. On the other hand, rules are used to determine if an object should be more likely to spawn because there is another object nearby, or if it shouldn't spawn at all because there are too many of the same objects nearby.

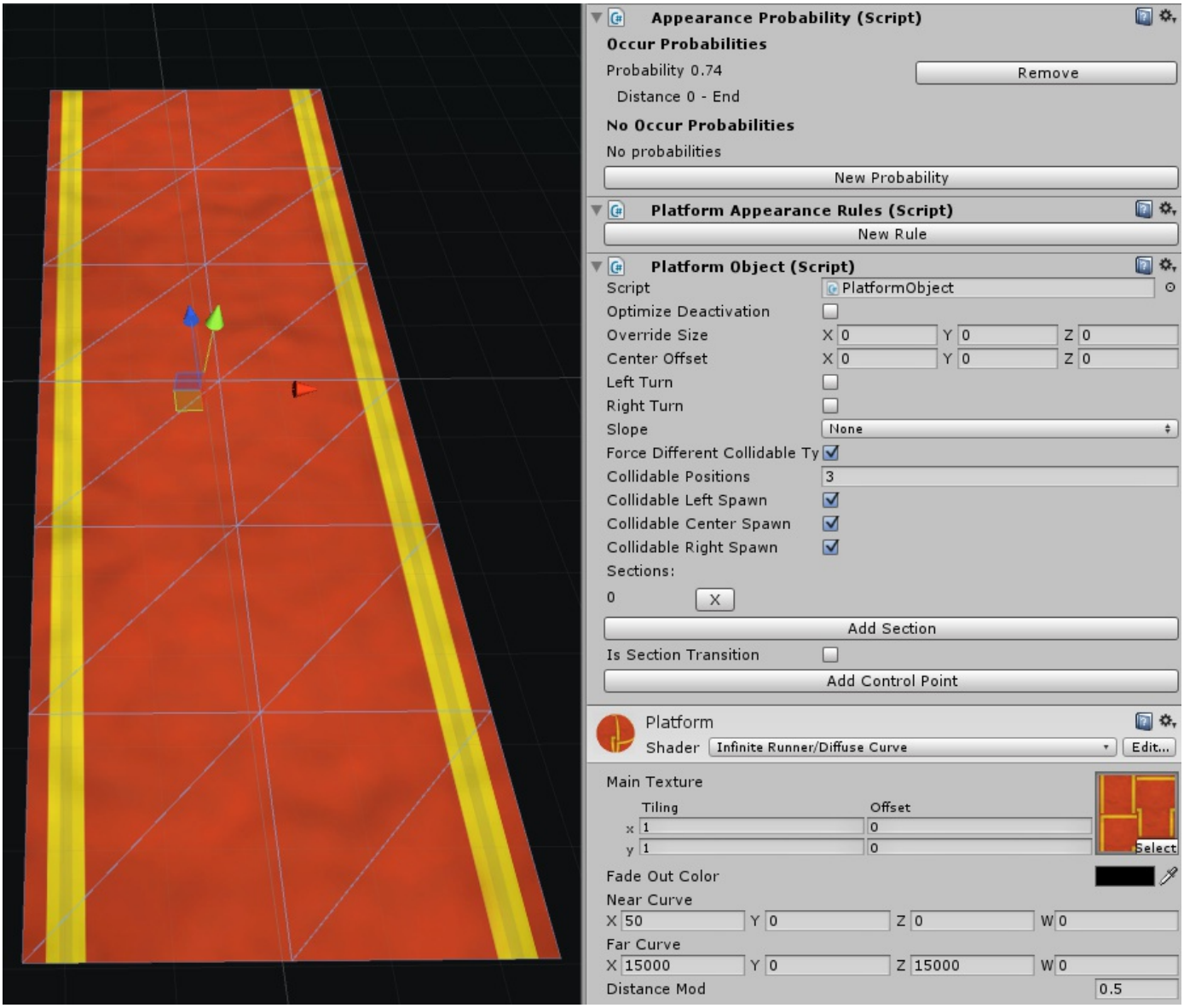
In order to make things as efficient as possible, these objects are placed in a pool so they don't have to be instantiated every time a new object has to appear. As the player runs, all of the objects around the player are actually moving rather than the player himself. This is done to avoid floating point errors with players who get a really high score. The player has to avoid jump and duck obstacles, all the while collecting coins and power ups. Coins are used to purchase power ups. There are three different types of power ups - double coin, invincibility, and coin magnet. Scene objects then act as the background objects and are used as the games setting. In this starter pack the scene objects are rooms inside a castle.

The pack is created like most other infinite runner games where there is a three slot system within the track. The player can be positioned in a left slot, center slot, or right slot. Obstacles, coins, and power ups (or collidables as referred to in the code) can appear in one or all of these slots. In addition to the slots, players will also have to turn left or right when there is a dead end ahead. All of these controls are made with the keyboard on the computer or a flick of the finger / accelerometer on a mobile device. The tutorial objects at the start of the game will also give an indication of what to do. Related to this, it is also possible to have a set of objects always occur at the start of the game.

Audio has also been included in this pack, and sound effects will play over the looping background music as events occur. The pack also includes a sample set of player animation to give a visual that the player is running, ducking, jumping, etc. The GUI was created with NGUI to show the different menus - Start, In Game, Tutorial, Pause, End Game, Power Ups, and Stats. Game and data managing classes are used to control the state of the game as well as any game related variables - score, coins collected, etc. A pretty cool shader effect has also been created to show all a gentle right curve when in reality all of the objects are flat.

It wouldn't be a complete starter pack without any editor tools. We've included multiple editor inspectors which will make creating the appearance probabilities and appearance rules really easy. And if this doesn't answer your question or you can't understand the code we'd be happy to help through email or the forums.

Infinite Objects

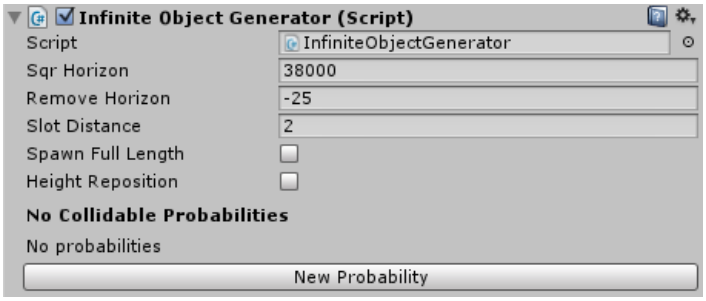


If an object is dynamically created it has a base class of InfiniteObject. Infinite Objects are platforms, scenes, obstacles, coins, and power ups. The last three types - obstacles, coins, and power ups - also share a parent class called CollidableObject because they all spawn on top of the platform. We'll get into more detail later on the specifics of each type of object, but all of these objects are created and managed through the Infinite Object Generator.

InfiniteObjects have the option of optimizing their activation/deactivation in the object pool by only disabling the renderer/box collider. Instead of deactivating the entire object using SetActive(), just the two components will be disabled (along with the InfiniteObject component). This option is especially useful if your object has a big animation and there is a lot of lag when an object gets activated.

Related Files: InfiniteObject.cs, CollidableObject.cs

Infinite Object Generator



The Infinite Object Generator can be considered the heart of the Infinite Runner Starter Pack. It manages the Infinite Objects on the screen as well as what Infinite Objects are about to spawn. When Infinite Objects are spawned, they are created at the *horizon*. When they spawn, they are placed in a list called *activeObjects*. As the player moves, the player tells the Infinite Object Generator the distance that it moves and in return the Infinite Object Generator moves all of the Infinite Objects. When a Platform or Scene Infinite Object is spawned, they are place in a hierarchy to increase performance. When one of these objects are spawned, the most newest object is set as the parent of all of the other objects of the same type and this one object is moved by the Infinite Object Generator

within *moveObjects()*. The Infinite Objects keep moving until they reach the *removeHorizon*. At this point the Infinite Objects will be removed and added back to an object pool within the Infinite Object Manager.

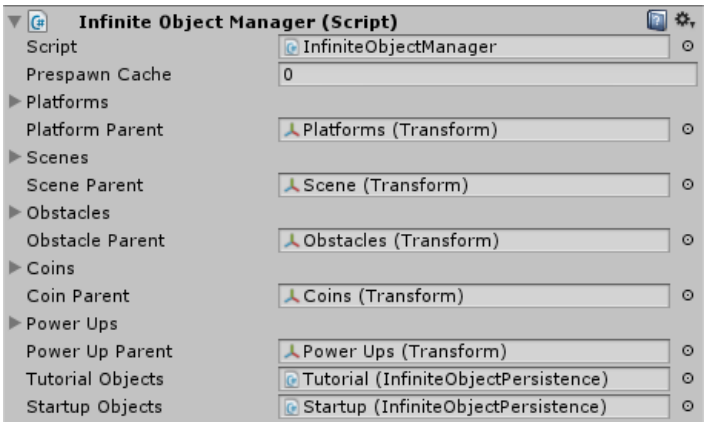
The Infinite Object Manager is the class that is in charge of keeping track of all of the Infinite Objects. When the Infinite Object Generator requests a new Infinite Objects, all it has to specify is the *ObjectType.* The Infinite Object Manager will then find a good Infinite Object to spawn based off of each Infinite Objects Appearance Probabilities and Appearance Rules. For these appearance classes to make a decision they need to have some history and that is obtained through the Infinite Object History. This class keeps tracks of anything that occurred in the past, such as the distance at which the last jump obstacle spawned or what is the latest platform type that has spawned. There is also a chance that no object can be spawned and that is controlled through the *noCollidableProbability* variable. This variable is of type Distance Probability List so you can specify different probabilities based on the distance.

In addition to spawning and removing Infinite Objects, the Infinite Object Generator also manages what do when a turn occurs and where to place Infinite Objects on top of the track. As Collidable Objects are spawned, the Infinite Object Generator will decide if the object should be placed in the center, left, or right side of the platform. In addition, the platform can also specify how many Collidable Objects can spawn on top of it and the Infinite Object Generator will take care of that as well.

Related Files: InfiniteObjectGenerator.cs, InfiniteObjectManager.cs, InfiniteObjectHistory.cs

---

Object Pooling

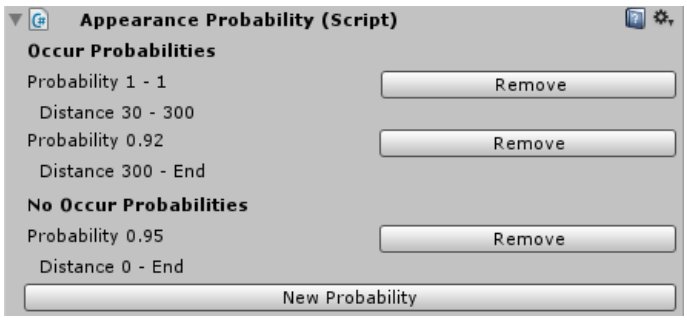


Object Pooling provides a huge performance boost which is one of the reasons why it can run smoothly on mobile devices. Instead of instantiating and destroying game objects as they appear/disappear, the game manager will activate an Infinite Object when the Infinite Object should spawn and deactivate when the Infinite Objects should disappear. An Infinite Object will only have to be instantiated when no Infinite Objects of the particular type have been created or all of the Infinite Objects within the pool are already on the screen.

The pool is called *objectsPool* and is located within the Infinite Object Manager. The Infinite Object Generator will request a new Infinite Object of a particular Object Type through *objectFromPool()*. *objectFromPool* takes two parameters - first an index and second the Object Type. There are three different index types within this starter pack and they are all discussed in detail within the file InfiniteObjectHistory.cs. Object Type is a platform, scene, obstacle, power up, or coin.

Related File: InfiniteObjectManager.cs

Appearance Probabilities



One of the more advanced topics of this starter pack is determining when Infinite Objects can spawn. Each time the Infinite Object Generator asks the Infinite Object Manager to spawn an object, the Infinite Object Manager loops through the list of objects of a particular Object Type. It will then use the Appearance Probability of the Infinite Object in order to randomly choose an Infinite Object. The Appearance Probability returns a probability based off of the current distance. All of the probabilities that an object can occur are placed in the *occurProbabilities* list and if you want to decrease the chances that an object occurs at a certain distance you can do so through *noOccurProbabilities*. Both of these objects contain a list of type Distance Value.

DistanceValue is a relatively simple class in that it takes the *startDistance* and *endDistance* along with the *startValue* and *endValue*

in order to determine the value at the specific distance. A variety of objects use the DistanceValue class so value is meant to be a generic type. Value may be a probability, speed, section, etc. If the current distance occurs before *startDistance* or after *endDistance* then the object does not have a chance of occurring with those values. In addition, you can specify through *useEndDistance* if you want to actually use the *endDistance* or if this value will occur for the rest of the game after *startDistance*. DistanceValue also gives you the option to *Loop* the list of distance values. In order to be able to loop, an end distance must be specified for every item. When the distance is greater than the last item in the list, it will simply loop back to the start of the list.

So that the probabilities can quickly be accessed, they have to be in order according to start/end distance and cannot overlap. To assist with this requirement, an inspector editor script called AppearanceProbabilityInspector has been created to allow you to quickly create these probabilities.

Related Video: Infinite Runner Starter Pack Appearance Rules/Probabilities

Related Files: AppearanceProbability.cs, AppearanceProbabilityInspector.cs

---

Appearance Rules

Appearance Rules can be thought of an advanced Appearance Probability. For each object, Appearance Rules allow you to specify the *minDistance* that an Infinite Object should appear from another Infinite Object. If *minDistanceSameObjectType* is checked then the Infinite Object who has this Appearance Rule component attached to will require that it has a *minDistance* from any object of the same type. For example, if you want to require the jump obstacle to have a minimum distance of 20 units away from the duck obstacle, you'd specify a *minDistance* value of 20 and the *targetObject* the duck obstacle. If you want the jump obstacle to be at least 20 units away from any other obstacle then you would set *minDistanceSameObjectType* to true. If *minDistanceSameObjectType* is true a *targetObject* does not need to be specified.

In addition to specifying the distance, you can also specify a *minObjectSeperation* value. *minObjectSeperation* allows you specify the minimum number of objects that should appear between the current Infinite Object and the Infinite Object specified by *targetObject*.

So far the only rules that we've discussed prevent objects from spawning. It is also possible to do the reverse using the same variables. The lists that store all of the rules are kept in *avoidObjectRuleMap* and *probabilityAdjustmentMap*. *avoidObjectRulesMap* is used for when you want to prevent objects from occurring and *probabilityAdjustmentMap* is used when you want to adjust the probability based on objects that have occurred within the minimum distance or separation. Both of these lists are of type Object Rule Map. The Object Rule Map contains the *targetObject* that the rule is being applied to as well as a list of Score Object Count Rules. Score Object Count Rule is the actual object that contains the minimum distances and separations. Object Rule Map contains a list of Score Object Count Rules because different rules can be applied to the *targetObject* at different distances. Similar to the Appearance Probabilities, Appearance Rules uses a Distance Probability object in order to determine which rules to apply based off of the current distance.

As if that wasn't complicated enough, collidable objects, scene objects, and power ups extend the rules to add some extra functionality. In the case of Collidable Appearance Rules, you have the option of specifying which platform(s) the attached collidable object cannot spawn on. Scene Appearance Rules adds functionality in the case of turns. If the there is a T turn ahead, certain scenes may not be able to fit without clipping the platform. Scene Appearance Rules will automatically determine if the attached scene object can fit within the space remaining. The Power Up Appearance Rules determines if the power up can spawn based on if the player has purchased the power up.

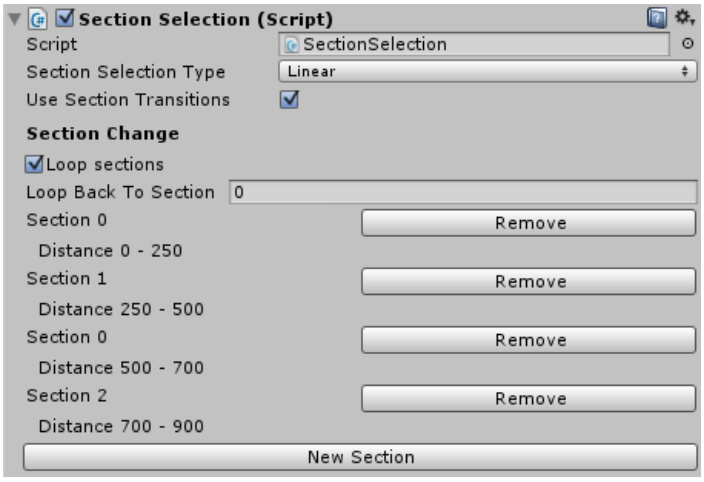
Appearance Rules can get pretty complicated and just like the Appearance Probabilities, Appearance Rules has its own set of restrictions such as the probabilities must be in order and cannot overlap. To make your game easier to design, another editor inspector class has been created named AppearanceRulesInspector.

Related Video: Infinite Runner Starter Pack Appearance Rules/Probabilities

Related Files: AppearanceRules.cs, CollidableAppearanceRules.cs, SceneAppearanceRules.cs, AppearanceRulesInspector.cs

---

Sections

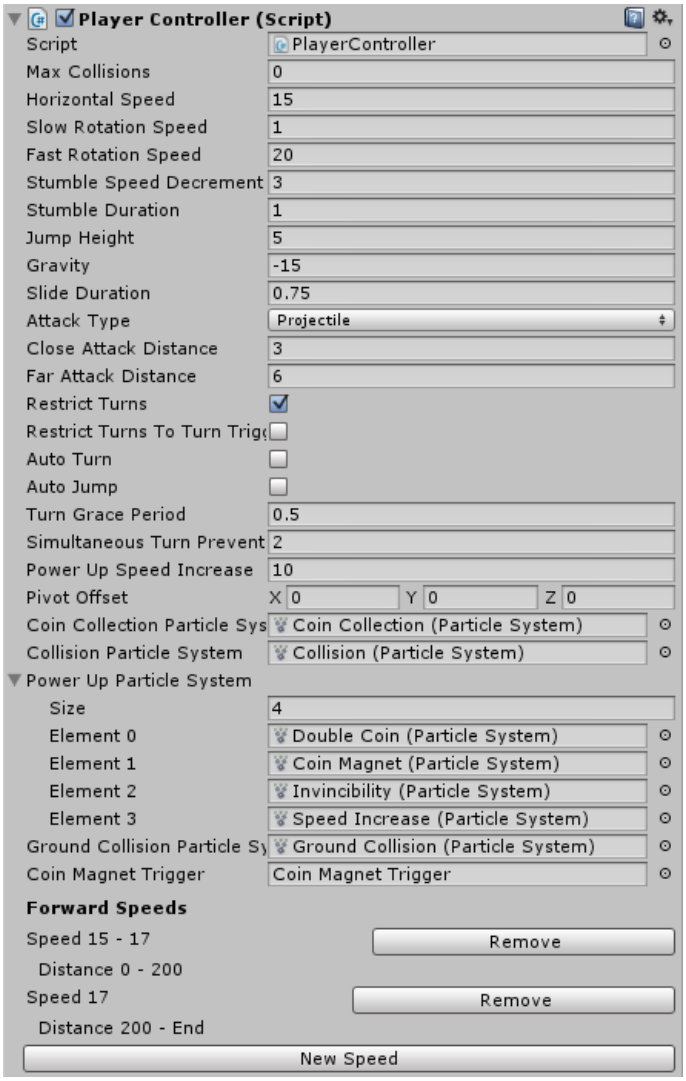


Sections allow you to specify when a particular object should spawn based on the distance. For example, you may want the fist 1000 meters to be inside and then switch outside after that with a complete different set of infinite objects. Sections allow you to do that. To setup sections, you first must decide how you want the sections to change. The most straight forward is to do a linear transition between the different sections (*SectionSelectionType.Linear*). You can also change sections based off of a probability. If at any point a probability value is less than the probability specified for the given distance, a new section can randomly be chosen (*SectionSelectionType.ProbabilityRandom*) or it can sequentially move through the sections (*SectionSelectionType.ProbabilityLoop*). When a section changes, you have the option to use *useSectionTransitions*. This allows you to place transition objects between the different sections. A custom inspector script has been created to allow you to easily change the section properties.

Related Files: SectionSelection.cs, SectionSelectionInspector.cs

---

Player



The main responsibility of the player is to determine the speed at which Infinite Objects move and to determine collisions with any Collidable Objects. Every update the Player Controller will tell the Infinite Object Generator to move its active objects by the value

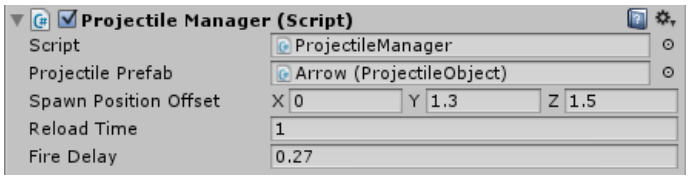
determined by *forwardSpeeds*. *forwardSpeeds* is of type DistanceValue to allow variable speeds throughout the game. *maxCollisions* is used to determine how many times a player can collide with obstacles before the game is over. When an input action has occurred, the Input Controller will call the Player Controller's corresponding jump, duck, attack, change track slots, or turn method. The force applied to make the player jump is determined by the *jumpForce*. Related to this, the amount of time a player ducks is determined by the *duckDuration*. Turns can be restricted to just turn platforms by enabling *restrictTurn*. A *turnGracePeriod* allows you to specify an amount of time that the player can swipe before the turn and have the character still turn. If the player collides with an obstacle, they can stumble and slow down for a moment of time. The player can automatically turn on all turns if *autoTurn* is enabled. The amount that the player slows down on a stumble is defined by *stumbleSpeedDecrement* and *stumbleDuration* determines the time that the player stumbles for.

attackType specifies if the player can attack. If the player can attack, there are two different types of attacks: Fixed and Projectile. Fixed attacks attack a fixed distance in front of the character. *closeAttackDistance* and *farAttackDistance* specify this attack range. If the AttackType is Projectile, a ProjectileManager component must be added to the character. The ProjectileManager creates a pool for all of the different projectiles. More information on the projectile system can be found here.

All of the walls have Collider setup so when the player runs into it, the Player Controller will call game over. If a player hits a Collidable Object, the Player Controller doesn't directly respond to this event. Instead, the Collidable Object has a trigger which responds when the player enters this trigger. The Collidable Object will then call the Game Manager and the Game Manager will decide what to do with the event. In the case of the player colliding with Obstacles, the Player Controller will call *obstacleCollision()* and play an animation along with some particle effects.

Related File: PlayerController.cs

Player Attack Projectiles

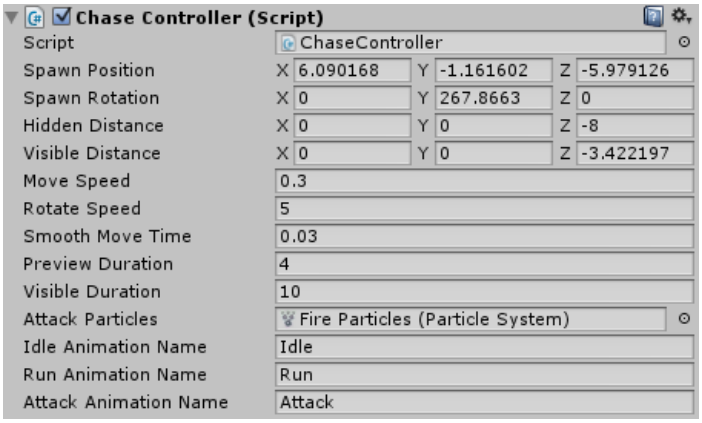


The ProjectileManager manages the object pool of all the projectiles for a particular player. The PlayerController requests a projectile to be spawned through the *fire()* method. When that is called, the ProjectileManager ensures a projectile can be fired and the player isn't currently reloading. *reloadTime* specifies how long it takes for a projectile to be reloaded. When *fire()* is initially called, the corresponding animation is started as well. fireDelay specifies the delay that the projectile should wait to stay synchronized with the animation. Similar to the *objectPool* within the InfiniteObjectManager, when a projectile actually needs to be spawned it will take a deactivated projectile within the pool if one exists. If one does not exist then it will spawn a new projectile.

A Projectile is created by the ProjectileManager and it is a simple class which manages the individual projectile. When the projectile is spawned, its forward direction is determined by *speed*. The projectile will keep moving until it either hits another collider or is further than a distance of *destroyDistance*. If the projectile collides with a ObstacleObject it will check to see if the ObstacleObject is destructible. If it is, it will then destroy that object.

Related Files: ProjectileManager.cs, Projectile.cs

Chase Object

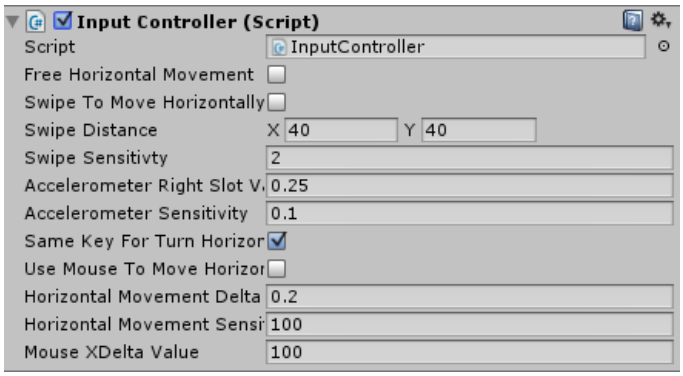


To start with, if you do not want a chase object within your game all you have to do is remove it from the static data component. The chase object uses the ChaseController class and that class exposes many options for determining how the chase object is visible. When the game first starts *spawnPosition* and *spawnRotation* determine the location where the chase object should spawn. When the game starts running *hiddenDistance* and *visibleDistance* are used to determine the offset of the chase object when it is outside of the view or when it is within view. When the chase object has to transition between any two positions it will use the *moveSpeed* and *rotateSpeed* to control its movement speed and rotation speed. When the game first starts the chase object will appear directly behind the player to indicate that it is chasing the player. It will stay behind the player for a duration of *previewDuration*. After the player

has hit an obstacle the chase object will move towards *visibleDistance* to indicate that the player can only collide with an obstacle one more time before the game is over. The duration *visibleDuration* is used to indicate how long the chase object should be in the *visibleDistance* position. *attackParticles* are played after the player has died and the chase object should attack the player. *idleAnimationName* is played at the start of the game when the chase object is just waiting for the game to start. *runAnimationName* is the main animation played when the game is playing and *attackAnimationName* is used when the chase object should attack the player because the player hit too many obstacles.

Related File: ChaseController.cs

Input



The starter pack supports keyboard input, touch input, as well as the accelerometer. On desktop computers the keyboard controls all input besides GUI selection. The player uses the keyboard to attack, jump, duck, change track slots, as well as turn. On devices that support touch, a tap will tell the player to attack and a swipe will cause the player to jump, duck or turn. The accelerometer is then used to determine which track slot the player should be running in. *swipeToMoveHorizontally* specifies if the character should move horizontally with a swipe (true) or the accelerometer (false).

The Input Controller interfaces with Unity's Input Manager to handle all of the in game input on both desktop and mobile platforms. For an introduction to Unity's Input Manager, take a look at this page. Unity's Input Manager maps a key press to a specific action, and that action is then called by the Input Controller.

Related File: InputController.cs

---

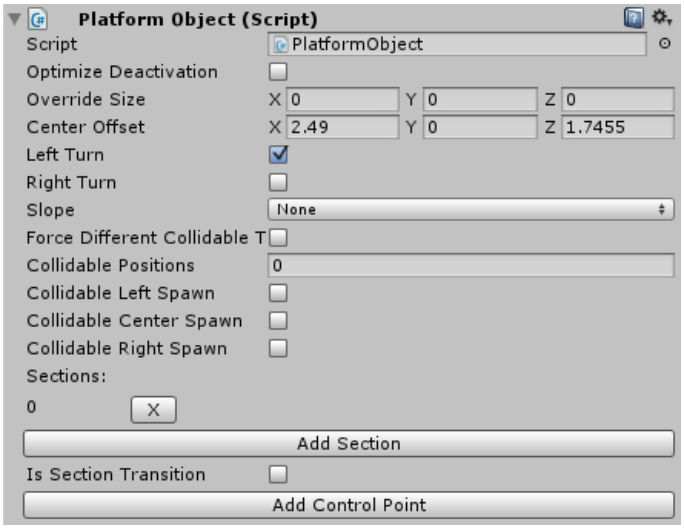
3 Slot System

One of the popular features of the infinite runner genre is the three slot system. The player can be running on the left side of the track, the center, or the right side. The distance in between these slots is controlled within the Infinite Object Generator called *slotDistance*. The speed at which the player changes slots is controlled by *moveSpeed.x* within the Player Controller class. The actual position translation happens within the update loop within the Player Controller.

Related Files: InfiniteObjectGenerator.cs, PlayerController.cs

---

Turns



The player will be running straight for most of the starter pack. However, every so often the player will have to turn left or right because a wall is straight ahead. Platforms can be marked as turning platforms through the Platform Object variables *isRightTurn* and *isLeftTurn*. To mark the platform as being able to turn left and right both variables should be enabled. When the invincibility power up

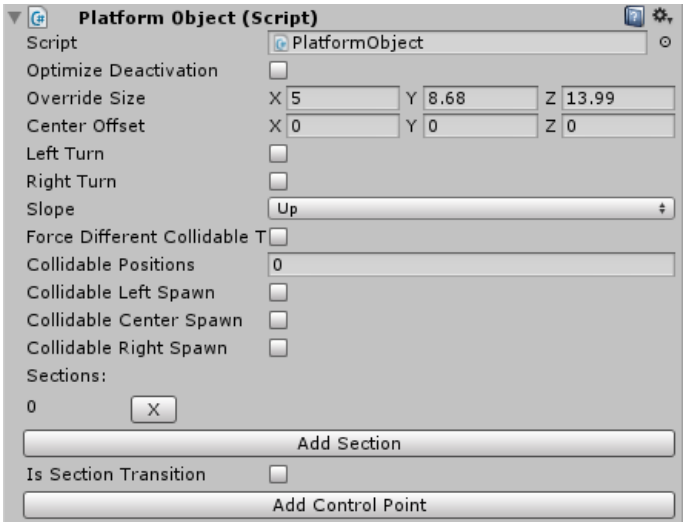
is activated the Turn Trigger will automatically turn the player on turns.

Once a turn is spawned, the Infinite Object Generator will spawn Infinite Objects to continue the track in the direction of the turn. *spawnObjectRun()* within the Infinite Object Generator will spawn the Infinite Objects in the direction of the turn. Object Location specifies the relative location of the Infinite Objects that are being spawned. For example, ObjectLocation.right specifies that Infinite Objects are being spawned for the track that is to the right of the current track. If the player turns right then the Infinite Objects that were in the right Object Location switch to becoming the center Object Location.

Related Files: PlatformObject.cs, TurnTrigger.cs, InfiniteObjectGenerator.cs

---

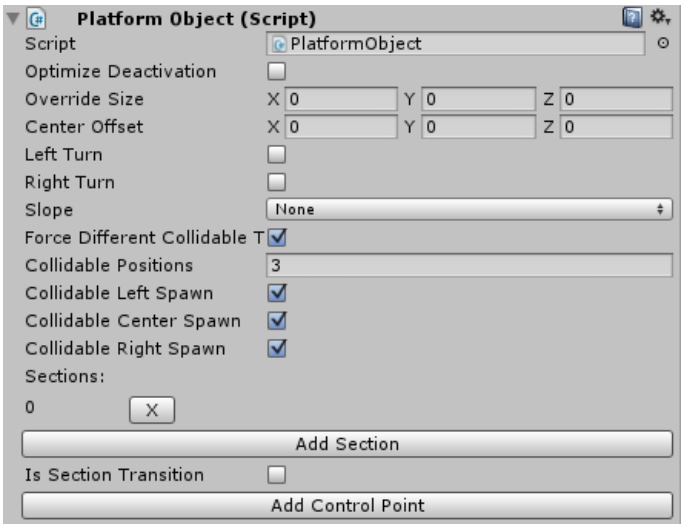
Height Variations



Platform Objects allow you to specify their slope - none, up, or down. This tells the InfiniteObjectGenerator if the platform causes a height change. An up slope indicates a positive height change and down indicates negative. The InfiniteObjectGenerator will then use this information to spawn all of the objects after the sloped platform with a height offset. When the platform is removed because it passed the *removeHorizon* the InfiniteObjectGenerator will then revert that height change to keep the objects in the same relative position.

Related Files: PlatformObject.cs, InfiniteObjectGenerator.cs

Platforms



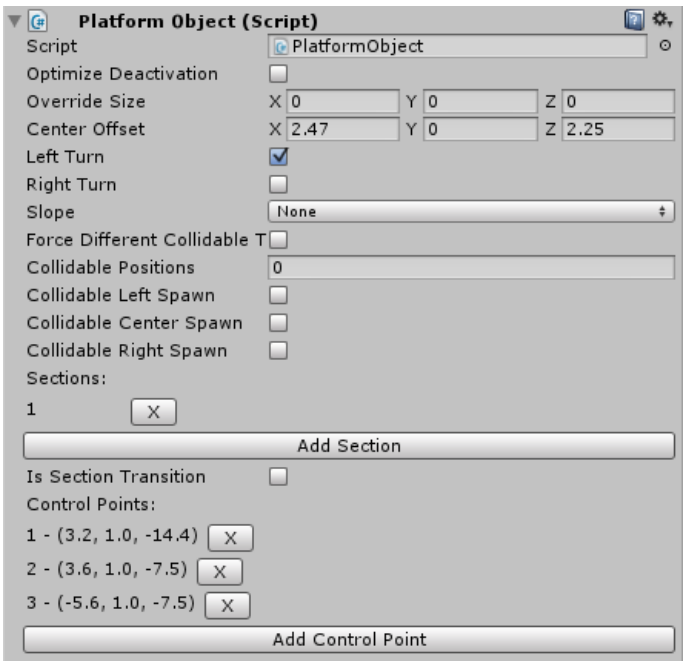
A Platform Object is derived from the Infinite Object and it specifies additional parameters that are unique to the platform object. The Infinite Object Manager calculates the size of the platform based off of its renderer. However, if this doesn't work for a particular platform, the size can be overridden with *overrideSize*. A platform has the option of having a turn, being a jump, or having a vertical slope. Turn platforms must specify the *centerOffset*. This value is the difference between the center and the amount that is extended out for the turn. If the platform is a jump it normally won't have a renderer attached to it. Therefore, *jumpLength* must be specified. Vertical slopes can either *Slope.Up* or *Slope.Down*. Platforms also specify if they can only spawn in particular section. If no sections are added then the platform will be able to spawn in any section. *isSectionTransition* must be selected if the platform section transitions from one section to another.

Multiple Collidable Objects can spawn on top of a single platform. The Infinite Object Generator knows how many to spawn based on the *collidablePositions* variables. With 3 *collidablePositions* spawned there is the potential to have up to three Collidable Objects

spawn on top of the platform. Similar to this, you can also specify the horizontal spawn location by enabling *collidableLeftSpawn*, *collidableCenterSpawn* and *collidableRightSpawn*. This goes along with the 3 Slot System and determines what slot the Collidable Object can be spawned over.

Related File: PlatformObject.cs

Platform Curves



Curved platforms use a quadratic bezier curve in order to determine the position of the character as the character moves down the platform. Bezier curve uses control points in order to form the curve. Control points can be added from the Platform Object. These control points are stored within the controlPoints array on the Platform Object. When the character reaches a curve, they will evaluate the curve based on the control points and the distance traveled within the platform. These calculations are made within PlayerController.getCurvePoint().

Related Files: PlatformObject.cs, PlatformObjectInspector.cs, PlayerController.cs

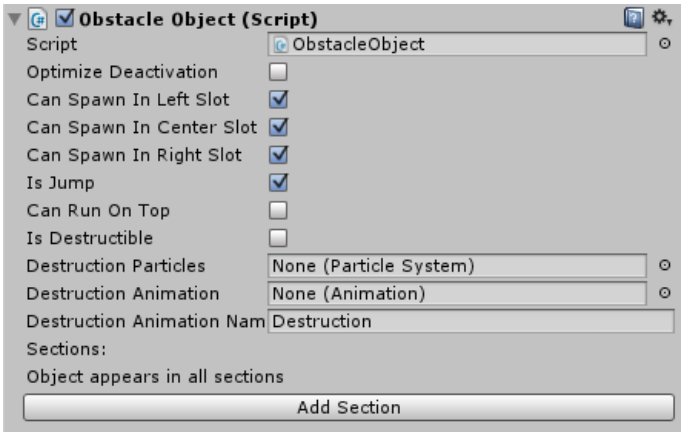
---

Collidable Object

The collidable object is the base class that all obstacles, power ups, and coins inherit from. Collidable object is inherited from and Infinite Object. When a Collidable Object is spawned, its parent is set to a Platform Object. When the Platform Object is deactivated, it is then placed back with its original parent to keep things organized. Similar to the platform, you can specify the horizontal spawn location on an individual collidable object by enabling *collidableLeftSpawn*, *collidableCenterSpawn* and *collidableRightSpawn*.

Related File: CollidableObject.cs

Obstacles



The Obstacle Object is inherited from the Collidable Object. The Obstacle Object does not offer much more functionality other then adding a trigger so that when the player enters this trigger the Game Manager then gets called to take action.

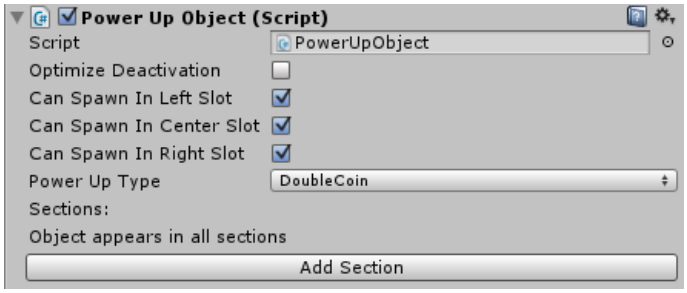
Related File: ObstacleObject.cs

Moving Obstacles

The Moving Obstacle Object is inherited from the Obstacle Object. A Moving Obstacle will start moving when it is within the *startMoveSquaredDistance* distance from the player. The squared distance is used for a less costly comparison than using the standard distance formula. *forwardSpeed* defines how fast the object should move towards the player. If *moveTowardsPlayer* is enabled, the object will move horizontally towards the player at speed *horizontalSpeed*. In addition, the obstacle can always face the player if *lookAtPlayer* is enabled. When the obstacle has passed (or collided) with the player the object will deactivate itself to prevent having to call the Update function. Similar to the PlayerController, the MovingObstacleObject applies a downward force to prevent the object from floating above platforms which have a slope.

Related File: MovingObstacleObject.cs

Power Ups

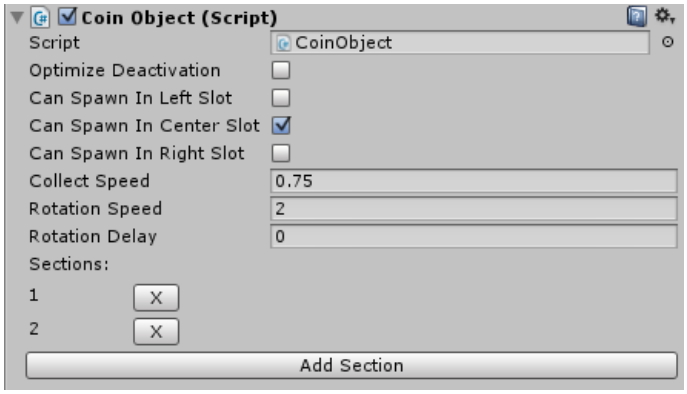


The Power Up Object is inherited from the Collidable Object and it is about as basic as the Obstacle Object. When the player enters the Power Up Object's trigger it'll activate the Power Up associated with *powerUpType*. The starter pack contains three types of power ups - invincibility, double coin, and coin magnet. The Power Up Manager handles what to do when each of these power ups is activated. All power ups have a duration associated with them for when the power up ends. This time is gotten through *powerUpLength* within Static Data. The player has to purchase power ups in order to enable the power up to spawn. This is done by purchasing the power up with coins, and the cost for the power up increases as the power up level gets higher. This cost is stored within *powerUpCost* also within Static Data.

When the double coin magnet is activated, instead of just counting one coin the Data Manager will count each coin collection as two. If the invincibility power up is active, the Game Manager will ignore collisions with obstacles. In addition, the Turn Trigger will automatically turn the player in the direction of the turn. The coin magnet is the last power up included in the pack and that will increase the radius that the player can collect coins. It does this by enabling the trigger Coin Magnet Trigger.

Related Files: PowerUpObject.cs, PowerUpController.cs, StaticData.cs, CoinMagnetTrigger.cs

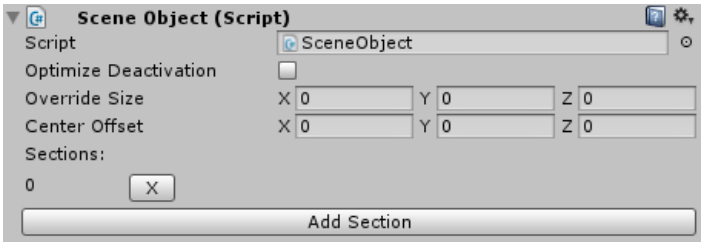
Coins



Coin Objects are again inherited from the Collidable Object and the only extra functionality that it provides is that when a coin is collected it will move to the center of the player to simulate the player collecting the coin. The speed at which the coin moves can be changed by *collectSpeed*.

Related File: CoinObject.cs

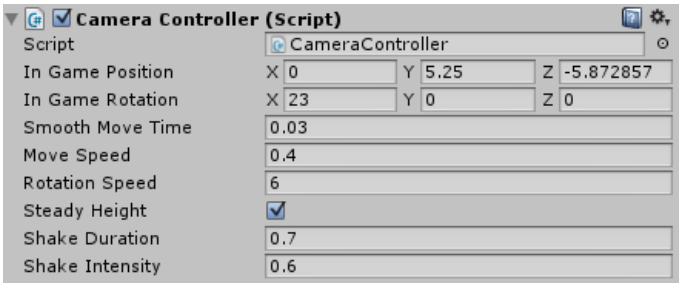
Scenes



The Scene Object is derived from the Infinite Object and adds the *centerOffset* property which is designed to be used for turn objects. If the center of the object isn't the scene center, then this offset is used to position the scene correctly in the game. The left and right corner scene prefabs make use of this property in the starter pack. The Scene Appearance Rules have been extended to be able to link to a platform. In previous versions you manually specified the type of scene (left or right turn, for example). Linking to a platform removes this requirement. The Scene Appearance Rules allow you to require a particular scene to always spawn with a particular platform (and one scene can spawn on top of multiple platforms). This is used in the starter pack for the turn, transition, and stair platforms. Similar to platforms, scenes allow you to *overrideSize* if the Infinite Object Manager did not calculate the size of the object correctly.

Related Files: SceneObject.cs, SceneAppearanceRules.cs

Camera Controller



During the main menu the camera is facing the player. When the game actually starts it reorients itself so it is positioned behind the player. The position/orientation is determined by *inGamePosition* and *inGameRotation*. The camera also sets its parent to the player so these in game positions/rotations are relative to the player. The camera also has the option of staying at a fixed height if *steadyHeight* is enabled. When the player hits an obstacle the camera will shake according to the *shakeIntensity* for a duration of *shakeDuration*. The camera now is not required to be positioned behind the player making side scrolling infinite runners possible. To do make the game a side scroller all that needs to be done is to set the *inGamePosition*/*inGameRotation* to a side angle and that is it.

Related File: CameraController.cs

Startup Objects

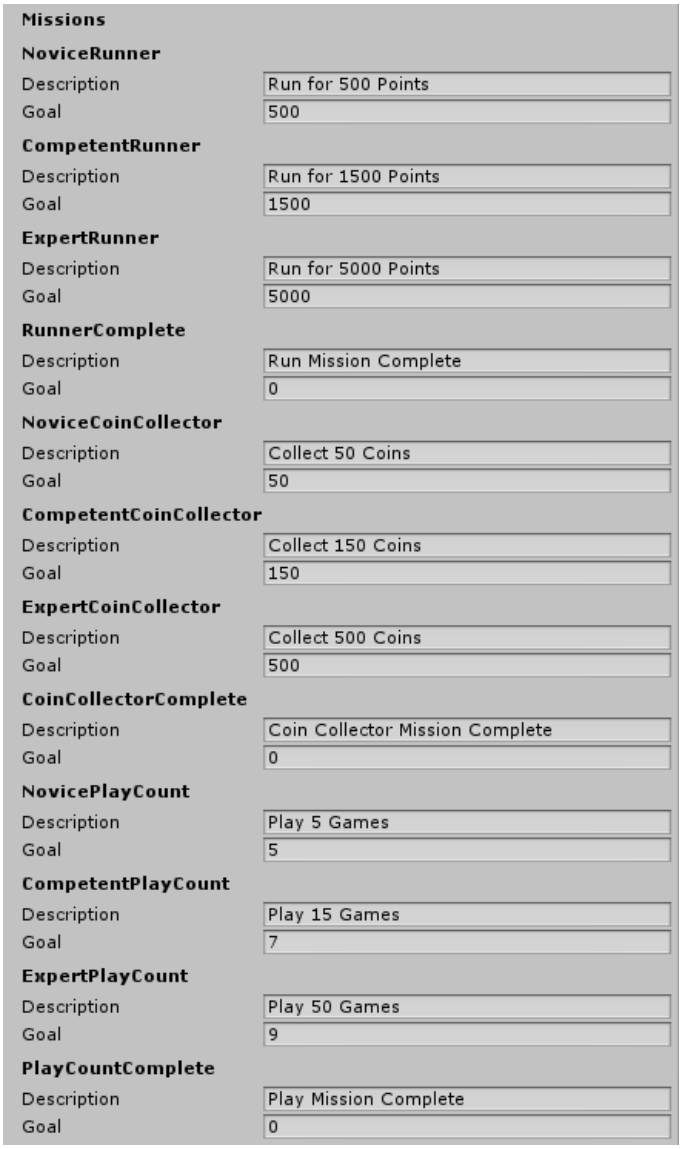
When most infinite runners start, there is a common set of objects that always appear at the beginning of the level. Most infinite runners also show the tutorial at the beginning of the level when the player hasn't played the game before. Both of these situations are referred to as startup objects. It would be possible to achieve the same objects run after run by modified the appearance probabilities and rules, however an easier method has been developed. Using the Infinite Object Persistence Editor, it is possible to create a prefab that always spawns at the start. The Game Manager will look for one of these prefabs at the start of the game, and if it finds it, it will place the objects in the correct positions within the infinite object hierarchy. Tutorial Triggers are used if you want to display text on the screen. When the player enters the trigger, the label appears. When they leave the trigger, the label disappears. Infinite Objects that appear within a startup prefab are marked to be destroyed after they pass the *removeHorizon* because they weren't instantiated from the object pool.

Related Video: Infinite Runner Starter Pack Startup Objects

Related Files: TutorialTrigger.cs, InfiniteObjectPersistenceEditor.cs

---

Missions



Missions, or objectives, define a set of goals for the player to achieve over multiple games. When the game ends, the Game Manager will tell the MissionManager that the game is over and to determine any missions that the player completed. The MissionManager will then loop through the *activeMissions* and determine if the player has completed any missions. If the player has completed a mission, the score multiplier will increase by *scoreMultiplierIncrement* and the next time the player plays a game their final score will be multiplied by this number.

Related File: MissionManager.cs

Pause

The starter pack has its own way of handling pause since setting Time.time to 0 within Unity prevents events from processing. When the game is paused (either through the pause button or the system callback OnApplicationPause), the game disables the appropriate scripts as well as stops any running Coroutines. The Game Manager uses the delegate Pause Handler to notify interested objects when the game pauses. Each object will then handle pausing/unpausing appropriately. In the cause of coroutines, the coroutine is stopped completely. The time remaining is saved off in an object called CoroutineData and that remaining time is then used when the game resumes from being paused. Objects such as the Player Controller and Input Controller are disabled because there is no reason for them to be running while the game is paused.

Related File: GameManager.cs

Game Manager

The Game Manager is used to act as a bridge between different game objects and is used to help manage events when they occur. The Game Manager is also responsible for changing game states, for example starting the game or calling game over.

Related File: GameManager.cs

Data Manager

The Data Manager keeps track of the data used throughout the game. It keeps the game score, the number of coins collected, the

number of obstacle collisions, as well as what level each power up is at. When *gameOver()* is called, the Data Manager will save off the score to PlayerPrefs if the score is higher than the previous high score. Similarly, the total coin count and power up level are saved to the PlayerPrefs so they can be retrieved later on.

Related File: GameManager.cs

GUI Manager

The GUI Manager keeps a reference to all of the different uGUI/NGUI panels and labels. The GUI can be in different GUI States (for example, main menu or in game). For each state the GUI Manager is responsible for making sure the correct panel is activated. The tutorial state is unique because it needs to be able to show at the same time as the In Game. The tutorial state has different types such as Jump, Duck, Strafe, and Turn for showing the different Tutorial Types. The GUI Manager is called by the Game Manager or the Data Manager to either update the GUI State or to update the label. For example, the Data Manager calls the GUI Manager to update the score.

IWhen a GUI button is pressed, the GUI Click Event Receiver handles the press. The GUI Click Event Receiver contains an enumeration of the different click types and sends the click to the appropriate object based on that click type. In addition, the buttons have a play animation or tween setup to seamlessly transition the GUI panels.

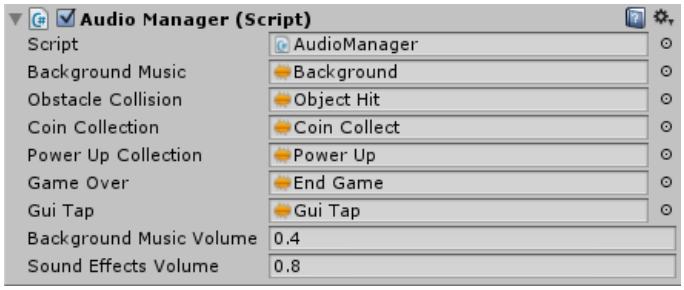
Related Files: GUIManager.cs, GUIClickEventReceiver.cs

Social Manager

The Social Manager is used to submit scores using Unity's Social API. Currently that API only supports Game Center on iOS and Mac. In order for this to work the *bundleIdentifier* must match that of the bundle identifier created within iTunes Connect. On a high score the Social Manager will submit the score to the leaderboard specified. In addition, any time a mission has been complete it will unlock the corresponding achievement. Both the leaderboard and the achievement identifiers must be setup within iTunes Connect for them to work properly.

Related File: SocialManager.cs

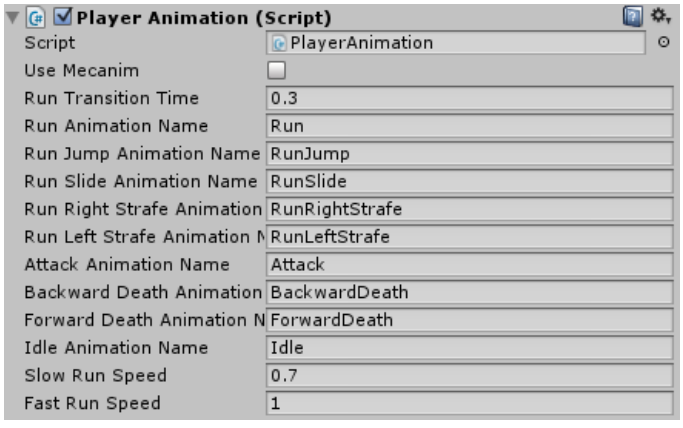
Audio



A background audio loop and six different sound effects have been composed for this starter pack. The background loop will play during the game and pause during any menus. The sound effects will play when a major event occurs such as activating a power up or hitting an obstacle. The audio is setup through the Audio Manager and it is also this class that manages how the audio is played. The Main Camera has two Audio Sources setup - one for the background loop and the other for the sound effects.

Related File: AudioManager.cs

Animation (and Mecanim)



Similar to audio, when a major event occurs the player will animate to respond to that event. Player Animation manages when the different animations are played. The player has the option of using legacy animations or mecanim animations. The PlayerAnmation

class has a toggle *useMecanim*. If the toggle is disabled then it will use the legacy animation system (default). With the legacy animation system, animation layers and blending are setup on each Animation State to make the animations feel as fluent as possible. If *useMecanim* is enabled, it will use mecanims animator component to trigger each animation controller state. When the starter pack is first imported it imports the mecanim animation example to a seperate Unity package called CharacterMecanim.unitypackage. The starter pack will start to use the mecanim system if you extract this file and replace the character prefabs within the static data component with the mecanim character prefabs. Note that mecanim is not supported with version 3.5.7 so if you try to import the package for that version you'll get errors.

Related File: PlayerAnimation.cs

Curved Vertex Transformation Shader

One of the cool effects included in this starter pack is the illusion that the platform and scene are turning to the right. This is just a simple vertex transformation within the shader and the Infinite Objects are not actually curved. The three variables *_NearCurve*, *_FarCurve* and *_Dist* are used to control the degree of the curve.

Related File: Diffuse Curve.shader

Custom Editor Inspectors

The Appearance Probabilities and the Appearance Rules can be complex to setup correctly. To make this easier, two different inspector scripts have been created to make creating these probabilities and rules easier.

To add a new Appearance Probability, do the following.

1. Select the Infinite Object that you want to add the Appearance Probability to.
2. Select "New Probability" under the Appearance Probability component.
3. Select the "Occur" if you want this probability to be the probability that the Infinite Object will occur and add this probability to the *occurProbabilities* list. Selecting "No Occur" will add the probability to the *noOccurProbabilities* list.
4. Select "Use End Distance" if you want this probability to occur from the "Start Distance" until the end of the game.
5. Input a number of the "Start Distance". This number is the distance at which the probability starts to take effect.
6. Slide the slider to select a "Start Probability".
7. If "Use End Distance" is checked, two more fields will appear allowing you to add the distance at which this probability stops taking effect and what the probability should be at the end. Appearance Probability will interpolate between these two probabilities if the distance is between start and end distance.
8. Click "Add"
9. If you want to remove a probability then click "Remove" next to the probability that you want to remove.

Distances may not overlap. If you enter a distance that overlaps with a probability already setup then you'll get an error when you click add telling you what you did wrong.

Appearance Rules are even more complex because they have one more layer than Appearance Probabilities, but the custom editor inspector makes it relatively easy.

To add a new Appearance Rule, do the following:

1. Select the Infinite Object that you want to add the Appearance Rule to.
2. Select "New Rule" under the Appearance Rule component.
3. Select "Avoid Object" if this rule should be used to avoid objects based on the rules specified and be added to the *avoidObjectRuleMap*. Selecting "Probability Adj" will add the rule to the *probabilityAdjustmentMap*.
4. Select "Use Min Distance with same Object Type" if this rule uses the minimum distance and should be applied to any Infinite Object of the same Object Type.
5. Select a "Target Object". The "Target Object" does not need to be selected if you are selecting "Use Min Distance with same Object Type" because the rule will take effect for all objects of the same Object Type
6. If this rule affects the minimum distance, enter that distance in "Min Distance".
7. If this rule affects the minimum object separation, enter that separation in "Min Object Separation".
8. Continue with step 4 from the Appearance Probability steps. If this probability is being added to the probability adjustment list, the probability doesn't have to be within 0-1 because it is adjusting the probability. Any floating point number can be used.

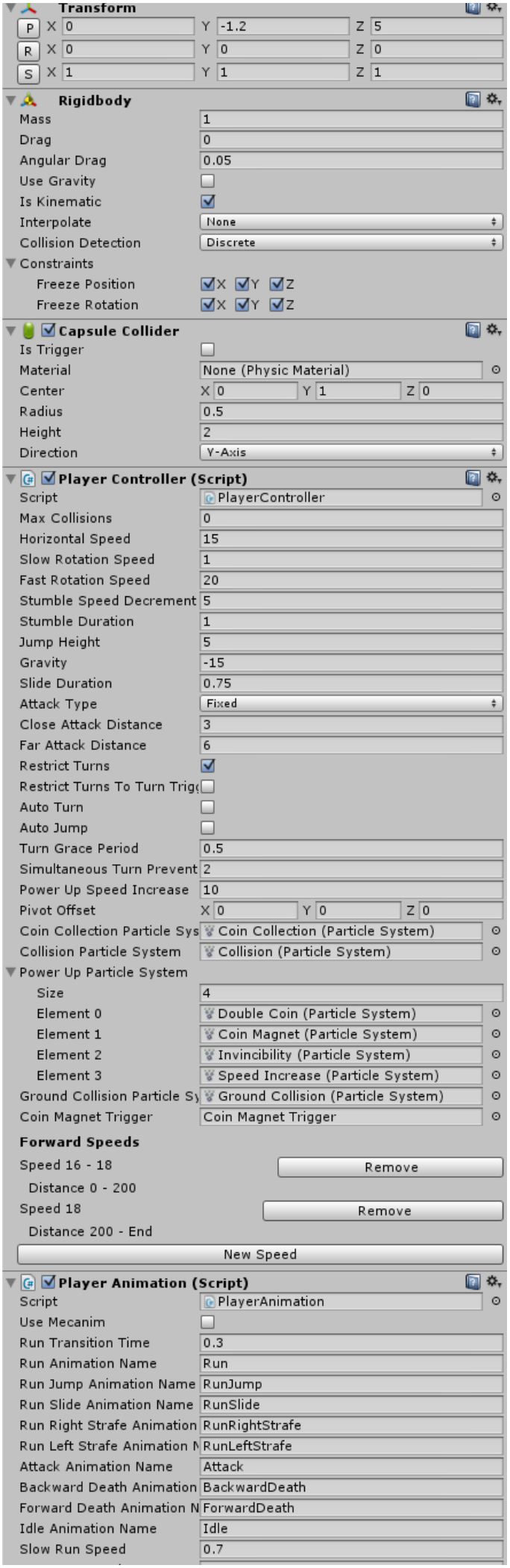
The Infinite Object Generator also uses a custom inspector script. The Infinite Object Generator allows you to specify the probability that a Collidable Object should not be spawned. This uses the Distance Probability List is very similar to adding a new Appearance Probability. To add a new probability, begin on step 4 from the Appearance Probability steps after clicking "New Probability".

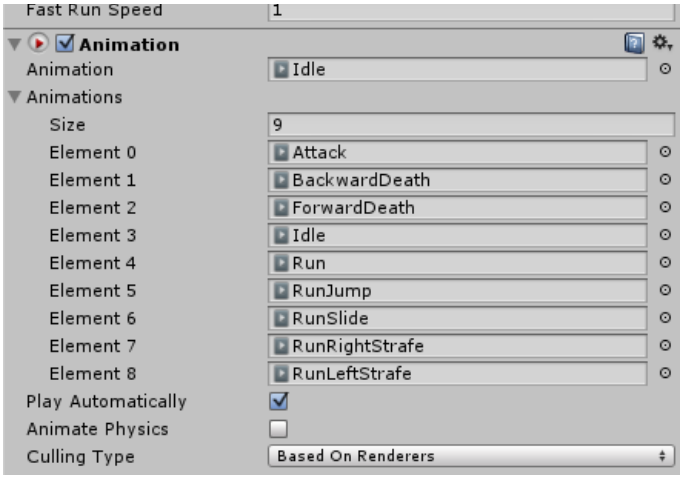
Related Video: Infinite Runner Starter Pack Appearance Rules/Probabilities

Related Files: AppearanceProbabilityInspector.cs, AppearanceRulesInspector.cs, CollidableAppearanceRulesInspector.cs, DistanceValueListInspector.cs, InfiniteObjectGeneratorInspector.cs, PowerUpAppearanceRulesInspector.cs, SceneAppearanceRulesInspector.cs

---

Using your own Character


Transform
P X 0 Y -1.2 Z 5
R X 0 Y 0 Z 0
S X 1 Y 1 Z 1
Rigidbody
Mass 1
Drag 0
Angular Drag 0.05
Use Gravity
Is Kinematic
Interpolate None
Collision Detection Discrete
Constraints
Freeze Position X Y Z
Freeze Rotation X Y Z
Capsule Collider
Is Trigger
Material None (Physic Material)
Center X 0 Y 1 Z 0
Radius 0.5
Height 2
Direction Y-Axis
Player Controller (Script)
Script PlayerController
Max Collisions 0
Horizontal Speed 15
Slow Rotation Speed 1
Fast Rotation Speed 20
Stumble Speed Decrement 5
Stumble Duration 1
Jump Height 5
Gravity -15
Slide Duration 0.75
Attack Type Fixed
Close Attack Distance 3
Far Attack Distance 6
Restrict Turns
Restrict Turns To Turn Trig
Auto Turn
Auto Jump
Turn Grace Period 0.5
Simultaneous Turn Prevent 2
Power Up Speed Increase 10
Pivot Offset X 0 Y 0 Z 0
Coin Collection Particle Sys Coin Collection (Particle System)
Collision Particle System Collision (Particle System)
Power Up Particle System
Size 4
Element 0 Double Coin (Particle System)
Element 1 Coin Magnet (Particle System)
Element 2 Invincibility (Particle System)
Element 3 Speed Increase (Particle System)
Ground Collision Particle Sy Ground Collision (Particle System)
Coin Magnet Trigger Coin Magnet Trigger
Forward Speeds
Speed 16 - 18
Remove
Distance 0 - 200
Speed 18
Remove
Distance 200 - End
New Speed
Player Animation (Script)
Script PlayerAnimation
Use Mecanim
Run Transition Time 0.3
Run Animation Name Run
Run Jump Animation Name RunJump
Run Slide Animation Name RunSlide
Run Right Strafe Animation RunRightStrafe
Run Left Strafe Animation N RunLeftStrafe
Attack Animation Name Attack
Backward Death Animation BackwardDeath
Forward Death Animation N ForwardDeath
Idle Animation Name Idle
Slow Run Speed 0.7



The Character Creation Wizard will assist you in creating your character. It will create a character prefab that you can immediately use in the game. The Character Creation Wizard can be accessed from the Window toolbar. If you do not want to use the Character Creation Wizard, the following steps are required to use your own character:

1. Drag the character model into the Unity editor
2. Add the PlayerController, CapsuleCollider, and Rigibody component. PlayerAnimation will be added automatically.
3. On the rigidbody disable gravity, set to kinematic, and freeze all position and rotation values.
4. Set the tag to "Player" and the layer to "Player"
5. Add a new game object with the script CoinMagnetTrigger
6. Create the following particle systems as new game objects: coin collection, coin magnet, collision, double coin, ground collision, invincibility, and speed increase.
7. Assign those newly created game objects to the corresponding values within the PlayerController. The power ups are assigned to an array named power up particle system.
8. Set the associated values within the PlayerController component. If you are unsure what value to use, compare your character to the screenshot above.
9. Rename the animations within PlayerAnimation to match your animation names (not required if the names are the same)
10. Save the character as a prefab in the Prefabs/Characters folder
11. Assign this prefab to the Static Data component which is attached to the "Game" game object. If you are only using one character you can assign it in the Character1 slot and remove the prefab from Character2.

Related Video: Infinite Runner Starter Pack - Using Your Own Assets (character begin around 8:00)

Creating Infinite Object Prefabs

The following steps will show what it takes to start using your own models as infinite objects:

Scene Objects:

1. Drag the scene model into the Unity editor
2. Add the SceneObject componenet. AppearanceProbability and SceneAppearanceRules will also come in with it.
3. Set the SceneObject parameters within the inspector
4. Mark the object as static
5. Create a child game object and name it "Collision Boxes"
6. A minimum of three collision boxes should be added. One for the left wall, one for the right wall, and one for the floor. Each collision box will be its own child game object under the "Collision Boxes" game object. Apply the "Wall" layer to these child game objects.
7. Create a new game object called "Baked Lights" and add any lights.
8. Create the prefab. Once you've created all of the scene objects, you can then drag all of them back in to the editor and do lightmapping

If your scenes are misaligned with other objects then take a look at the Object Misalignment topic.

Related Video: Infinite Runner Starter Pack - Using Your Own Assets (scenes begin around 16:30)

Obstacle Objects:

1. Drag the obstacle model into the Unity editor
2. Add the ObstacleObject componenet. BoxCollider, Rigidbody, AppearanceProbability and CollidableAppearanceRules will also come in with it.
3. Set the ObstacleObject parameters within the inspector
4. Mark the Box Collider as a trigger
5. The Rigidbody should not use gravity and it is kinematic
6. Select the "Obstacle" layer)
7. Create the prefab

Related Video: Infinite Runner Starter Pack - Using Your Own Assets (obstacles begin around 20:40)

Moving Obstacle Objects:

1. Drag the obstacle model into the Unity editor
2. Add the MovingObstacleObject componenet. BoxCollider, Rigidbody, AppearanceProbability and CollidableAppearanceRules will also come in with it.
3. Set the ObstacleObject parameters within the inspector
4. Mark the Box Collider as a trigger
5. The Rigidbody should not use gravity and it is kinematic
6. Select the "MovingObstacle" layer
7. Create the prefab

Platform Objects:

1. Drag the platform model into the Unity editor
2. Add the PlatformObject componenet. BoxCollider, AppearanceProbability and AppearanceRules will also come in with it.
3. Set the PlatformObject parameters within the inspector
4. Select the "Platform" layer
5. If the platform is a turn, add a turn trigger as the child with a turn trigger componenet attached to it. The Box Collider should be a trigger and the Rigidbody should not use gravity and be kinematic
6. If the platform is a jump, add a jump trigger as the child with a jump trigger component attached to it.
7. If the platform is a curve, specify the control points through the platform object inspector.
8. Create the prefab

If your platforms are misaligned with other objects then take a look at the Object Misalignment topic.

Related Video: Infinite Runner Starter Pack - Using Your Own Assets (platforms begin around 14:30)

Coin/PowerUp Objects:

1. Drag the coin or power up model into the Unity editor
2. Add the CoinObject/PowerUpObject componenet. BoxCollider, AppearanceProbability and (Collidable/PowerUp)AppearanceRules will also come in with it.
3. Set the CoinObject/PowerUpObject parameters within the inspector
4. Mark the Box Collider as a trigger
5. The Rigidbody should not use gravity and it is kinematic
6. Create the prefab

Related Video: Infinite Runner Starter Pack - Using Your Own Assets (coins begin around 22:20)

Object Misalignment

After you have created your platforms and scenes you may notice that the objects don't align with each other correctly. There isn't always one solution to this issue, so the following is the process that the starter pack goes through in order to place objects. It should give you a clue for where the true issue resides.

When the game starts up the Infinite Object Generator asks the Infinite Object Manager for the sizes of the platforms and scenes (InfiniteObjectManager:getObjectSizes). In order to determine the size of the object the manager uses the renderer's bounds. If your objects have children objects that contribute to the size of the platform then this method isn't going to work very well. In that case what you need to do is specify a size within the overrideSize property on the scene/platform. The Infinite Object Manager will then use that size for the object.

Once the size is retrieved the Infinite Object Generator has to actually place the objects. In order to place the objects in gets the position of the previous object and using both the previous and current object's size, it adds to the position. Before the current object is actually positioned the Infinite Object applies an offset to itself based on the transform's position (this is within InfiniteObject:orient). This works well for straight platforms/scenes but starts to cause a problem when you encounter a turn. A turn isn't necessarily symmetrical so it needs to have one more offset in order to determine the position of the objects after itself. That is what centerOffset is for. The centerOffset will be added to any object that occurs immediately after the turn object to help with alignment.

To generalize, if you only have one object repeating that the ends don't touch along the z axis then check the size of the object and use overrideSize if you need to. If the ends touch but they aren't aligned properly then change the position on the object's transform. If you have a turn object and things aren't aligning after the turn then play with the centerOffset.

---

uGUI and NGUI

Both uGUI and NGUI are supported. The uGUI scene is the default scene with the Unity 5 version of the Infinite Runner Starter Pack. The free version of NGUI is used for the Unity 4 version. If you'd like to use NGUI 3, perform the following steps:

1. Remove the free version of NGUI at /Infinite Runner/Third Party/NGUI
2. Import NGUI 3 from the Asset Store
3. Extract the package /Infinite Runner/Third Party/NGUI 3 Assets.unitypackage

4. Restart Unity if there are NGUI related errors in the console.

To use uGUI with Unity 4, extract the package /Infinite Runner/uGUI Assets.unitypackage. This will extract the uGUI scene as well as other uGUI assets. When you open the uGUI scene you'll notice that the GUI Type of the GUI Manager is set to uGUI and the uGUI objects are assigned.

When switching between uGUI and NGUI make sure the COMPILE_NGUI compiler definition is correctly set. If you only want to use uGUI the COMPILE_NGUI definition can be commented out. This needs to be set in both GUIManager.cs and GUIManageInspector.cs.

Support

We want the Infinite Object Starter Pack to be a great starting point for your infinite runner game. If you have any problems or suggestions for the starter pack please don't hesitant to email support@opsive.com or post on the forum.