Powerslide Kart Physics

by Justin Couch / JustInvoke

Support email: justin@justinvoke.com
Support terms: https://justinvoke.com/contact/

Table of Contents

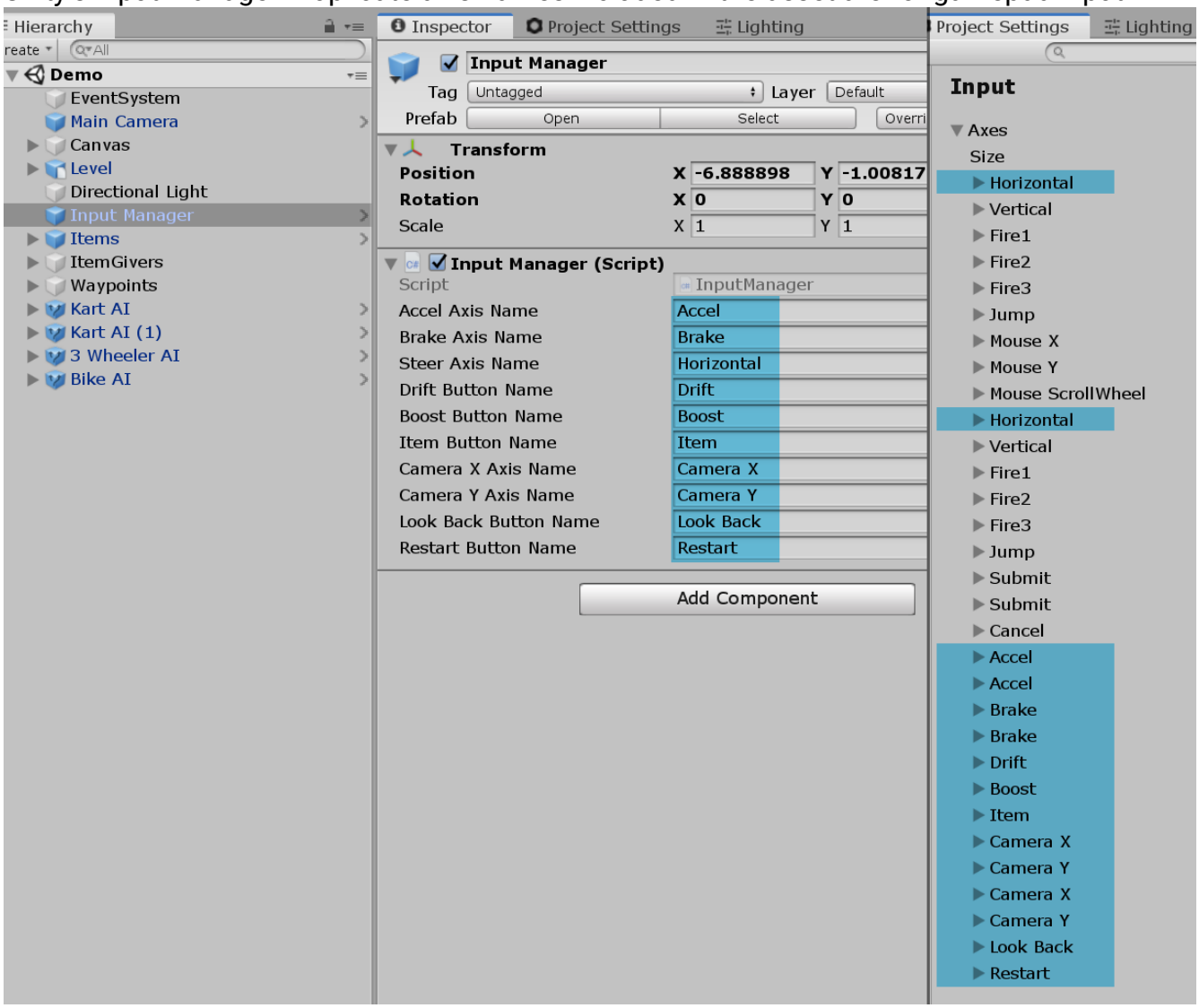
Project Settings

This asset comes with project settings already configured “optimally,” but they can still be changed to suit your project. Be careful when importing the package into your own project as it will override your existing project settings. Settings not listed in this section can be configured freely.

Input (Legacy Input Manager)

In the demo scene, there is an object named “Input Manager” containing the *InputManager* script with references to input axis names in Unity's Input Manager. If these axes are not set up, you will get warnings stating that they are not set up. By default this asset comes with Input Manager settings containing these input axes.

Depending on the needs of your project, you may resolve conflicts by either changing the input axis names in the script in scene, or changing the names of the actual input axes in Unity's Input Manager. Duplicate axis names included in the asset are for gamepad input.



Input (Input System)

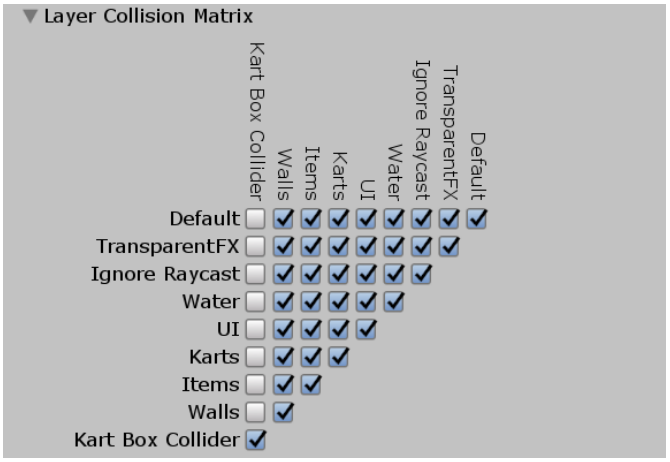
If you're using the new Input System package, see the Input System section below.

Physics

Physics settings in the asset are basically the same as the default settings in Unity, aside from the layer collision matrix. In the matrix, there should be a layer named “Kart Box Collider.” This is to provide a way for karts to collide without launching each other in the air. See the Kart Box Collider section below for more details.

This layer should only be used for child objects of karts containing a box collider, separate from a kart's mesh collider. Notice how the box colliders only collide with each other and nothing else. This allows karts to have stable, predictable collisions with each other while maintaining smooth collisions with everything else.

Other physics settings can be relatively flexible, within reason. Karts and projectile items have their own options for custom gravity on top of Unity's gravity. (See Custom Gravity section).



Tags and Layers

This asset makes optional use of tags and layers for physics purposes. Tags and layers included with the project are not required, but are strongly recommended (you may use your own project's “equivalent” layers).

Tags:

Walls – Optional tag used for tag-based wall collision detection. (See Wall Collision Detection section.)

Layers:

Karts – Layer for karts and their child objects, excluding kart box colliders.

Items – Layer for projectile items.

Walls – Layer for walls for layer-based wall collision detection.

Kart Box Collider – Layer for the kart box collider object within each kart.

Time

By default, the fixed timestep in the asset is 0.01. Other values are supported, but timesteps that are too large may result in instability. Karts and items use continuous collision detection by default in the example prefabs, so larger timesteps should not result in objects passing through static geometry. If continuous collision detection is disabled, then the fixed timestep should be decreased.

Kart Setup

This section covers setting up a kart from scratch. You can find example kart prefabs in the *Prefabs* > *Karts* folder.

From Scratch – Basic Hierarchy

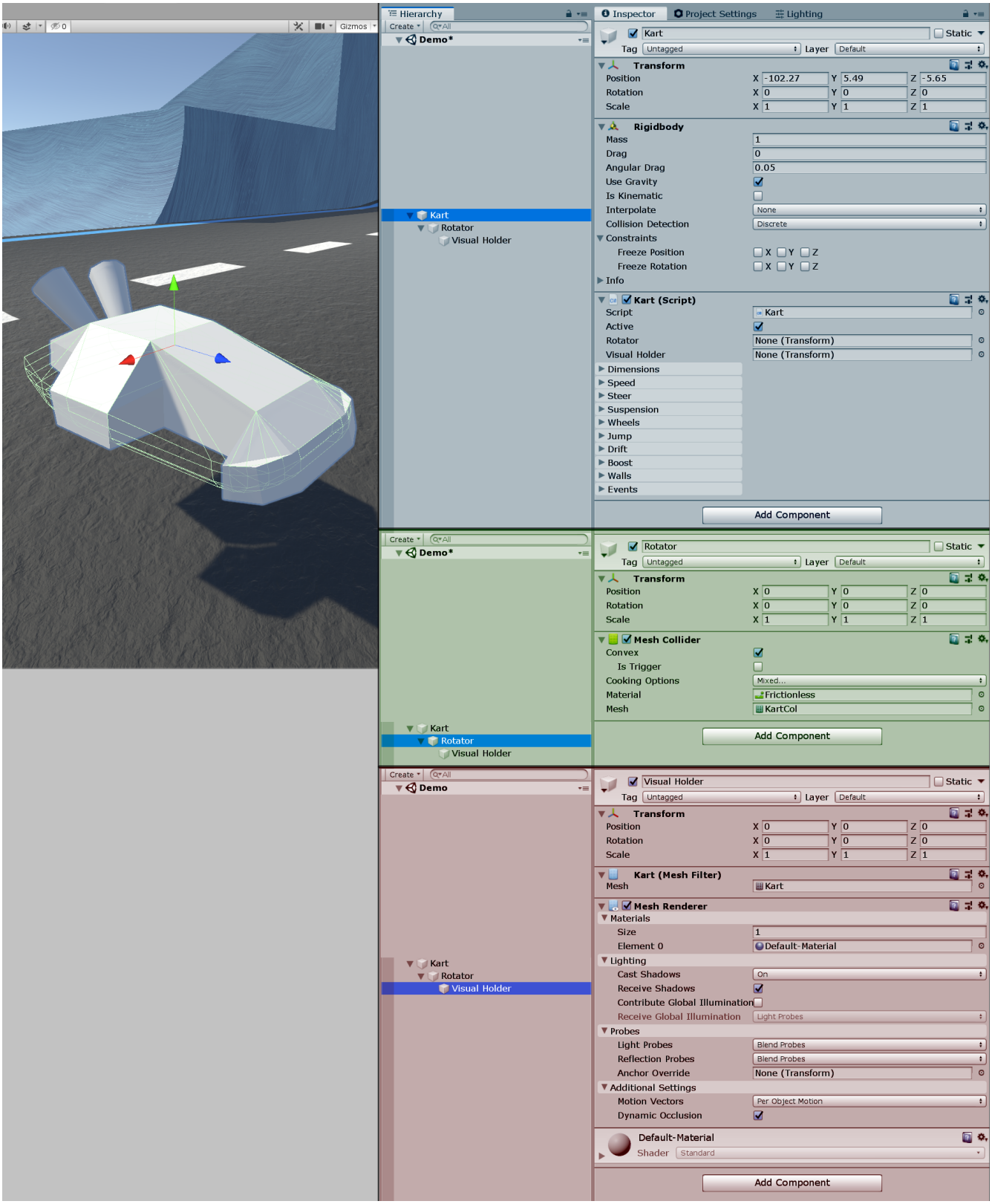
Begin by creating an empty object named “Kart.” Then create an empty child object inside of the kart named “Rotator.” Then create an empty child object inside of the rotator named “Visual Holder.”

Add a *Kart* component to the kart object. This is the main script for controlling kart behavior.

Add a collider component (preferably convex mesh collider) to the rotator object. This is the object representing the rotation of the kart. Rotation of the kart's rigidbody is automatically constrained because rotation of the kart is handled by rotating the transform of the rotator object. The rigidbody only deals with movement. The rotator's collider allows the kart to collide with world geometry and should use the included “Frictionless” physic material.

Add a mesh filter and mesh renderer component to the visual holder. This object holds all visual parts of the kart and rotates in local space for cosmetic features such as tilting/wheelies.

Current hierarchy of the kart:

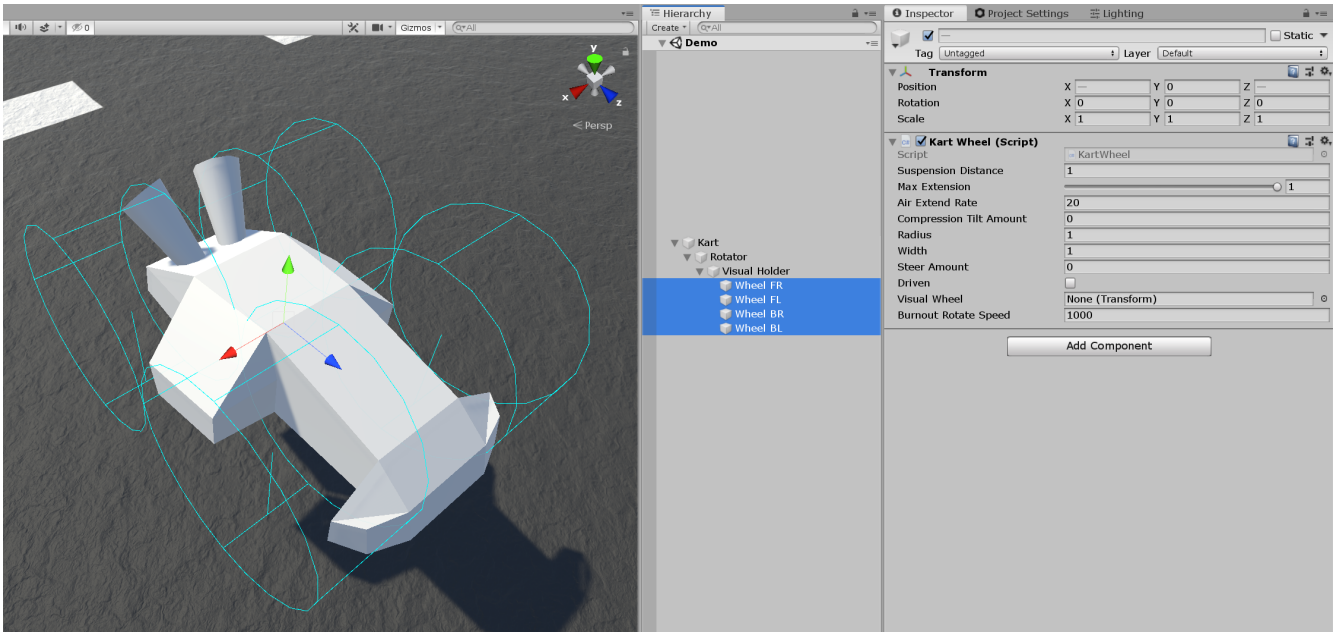


Now within the visual holder create a few empty child objects to represent the wheels of the kart (this example will use four wheels). The wheels are named with the suffixes FR (front-right), FL (front-left), BR (back-right), and BL (back-left) indicating their local positions. Position them accordingly.

Karts can have any number of wheels (at least one) and they are purely cosmetic aside from determining where the wheel raycasts start and how long they are, based on the suspension distance.

Add a *KartWheel* component to each wheel.

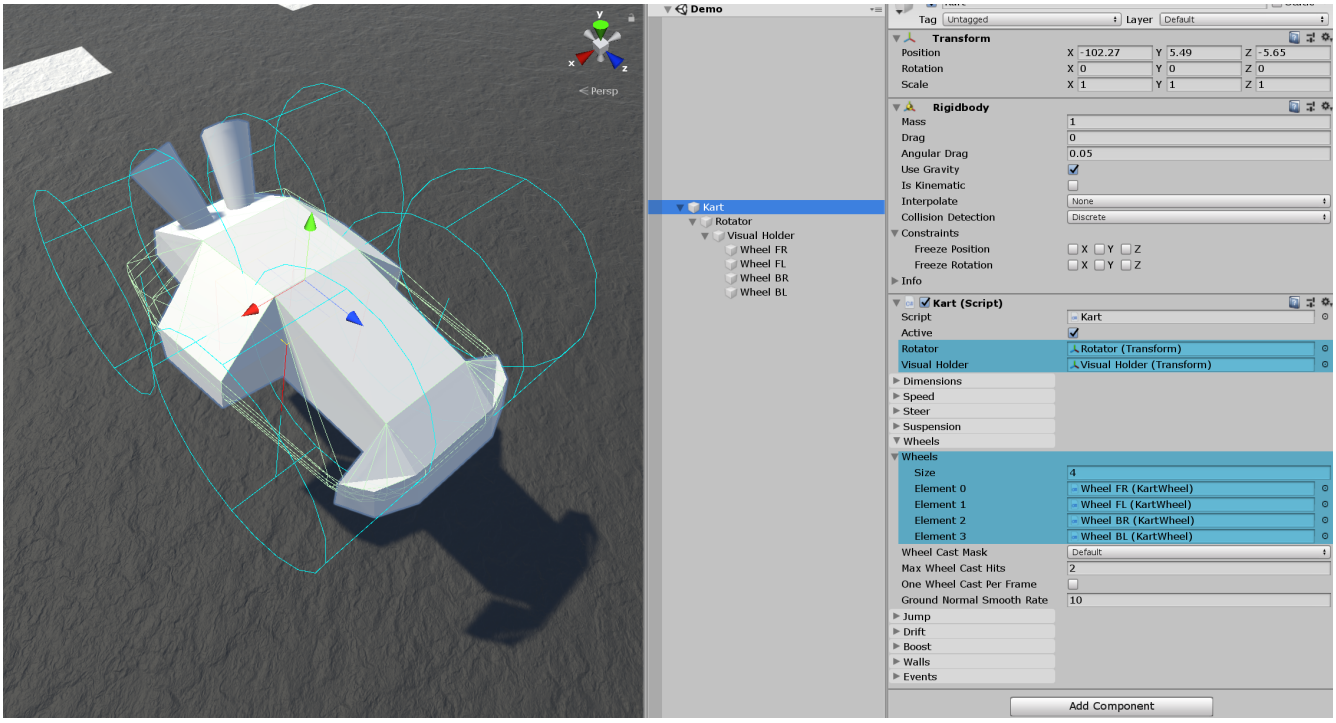
Current hierarchy:



Now go back to the root kart object. Drag the rotator object into the “Rotator” variable on the *Kart* script and the visual holder into the “Visual Holder” variable.

Open the “Wheels” foldout on the *Kart* script and drag each wheel into the “Wheels” array in the script.

The script should now look like this:



To make the kart controllable, just add a *KartInputPlayer* component to it and make sure there is an object present in the scene with the *InputManager* script attached (if you're using the legacy input manager). You can find a prefab of this object in the *Prefabs > Scene* folder. The demo scene already has this. Now when you enter play mode, the kart should respond to input (try the arrow keys/default input).

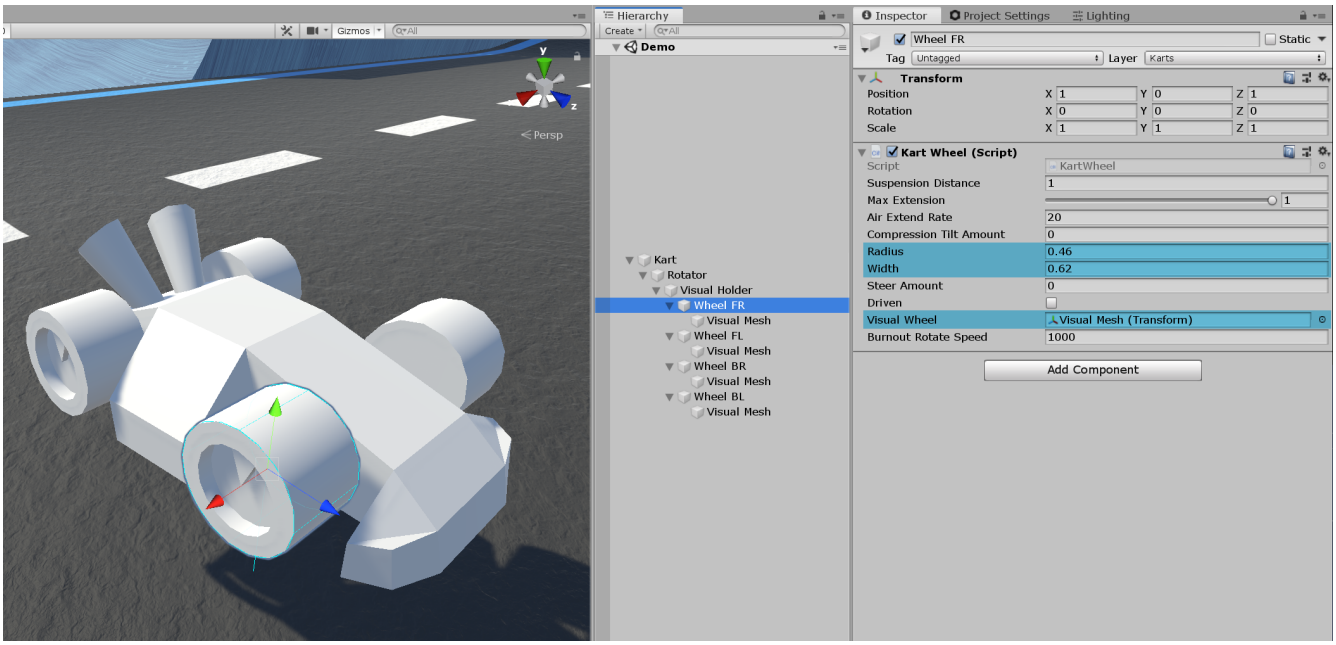
If you're using the new Input system package, see the Input System section below.

Don't forget to set the layer of the kart and its child objects to the “Kart” layer. This is important for excluding karts from the wheel raycast layer mask. See the layer mask below the wheels array on the *Kart* component.

Now we can set up visual meshes for the wheels and adjust the wheel dimensions. Add a child object inside of each wheel with a mesh filter and mesh renderer attached. Visual meshes for wheels are meant to rotate around their local z-axis. If your wheel mesh is not oriented such that it's local z-axis points out from the face of the wheel, you can place it as a child of the previously created visual mesh object and rotate it in local space to face the correct direction.

These visual mesh objects can be rotated +/-90 degrees on their local y-axis to face out toward the sides of the kart. This is purely for ease of setting properties in edit mode, as they are automatically rotated in play mode.

Now navigate to the *KartWheel* component on each wheel and set the “Visual Wheel” variable to the newly created child object for each one. Adjust the radius and width of the wheel to match the mesh (see the cyan cylinder gizmos).



You may want to set the “Driven” variable to true on the rear wheels to make them spin during burnouts and set the “Steer Amount” variable on the front wheels to a positive number to make them rotate with the kart's steering.

The suspension distance can be adjusted to extend how far the wheels can reach. Keep in mind that where the wheels are positioned in edit mode should be their most compressed position, where the suspensions would be at their shortest possible length.

For information about other kart wheel variables, see the *KartWheel* section below.

This concludes the bare necessities for setting up a kart that can be driven. Now we should set up a camera to follow the kart.

Camera Setup

There is a camera prefab you can use in the *Prefabs > Scene* folder, or you can use the existing camera in the demo scene. Once you have a camera in the scene with the *KartCamera* component attached, just drag the kart into the “Target Kart” variable of the script. Once you enter play mode, the camera should automatically follow the kart.

Customizing Performance

The kart script has many variables for tuning behavior, so this section will only cover using presets to adjust performance. See the *Kart* section below in the script reference for descriptions of all properties. Many properties are self-explanatory and/or can easily be tweaked in play mode to observe their effects.

Start by adding a *KartPresetControl* component to the kart object. Expand the “Save/Load Buttons” foldout to see buttons for working with presets.

There are several types of presets, each containing certain properties:

- Dimensions – Contains properties related to the kart's size and rotation.
- Speed – Contains properties related to speed and acceleration.
- Steer – Contains properties related to turning, tilting, and sideways friction.
- Suspension – Contains properties related to spring forces holding up the kart.
- Wheels – Contains properties related to wheels and ground raycasts.
- Jump – Contains properties related to jumping.
- Gravity – Contains properties related to gravity such as sticking to walls.
- Drift – Contains properties related to drifting and burnouts.
- Boost – Contains properties related to boost.
- Walls – Contains properties related to wall collision detection.

There is a variable for each type of preset on the script, and each preset can be created in the project as a *ScriptableObject* .asset file. If you navigate to the project window and choose *Create > Powerslide Kart Physics > Kart Preset*, you can find options for creating any type of preset.

There is a selection of presets included in the *Prefabs > Kart Presets* folder. Go through these folders and try picking different presets to place on the kart by dragging them into the corresponding field on the *KartPresetControl* component. To load the properties into the kart, click on the load button for each preset. Alternatively, you can check the box at the top of the script “Load On Awake” to automatically load the properties from all presets upon entering play mode.

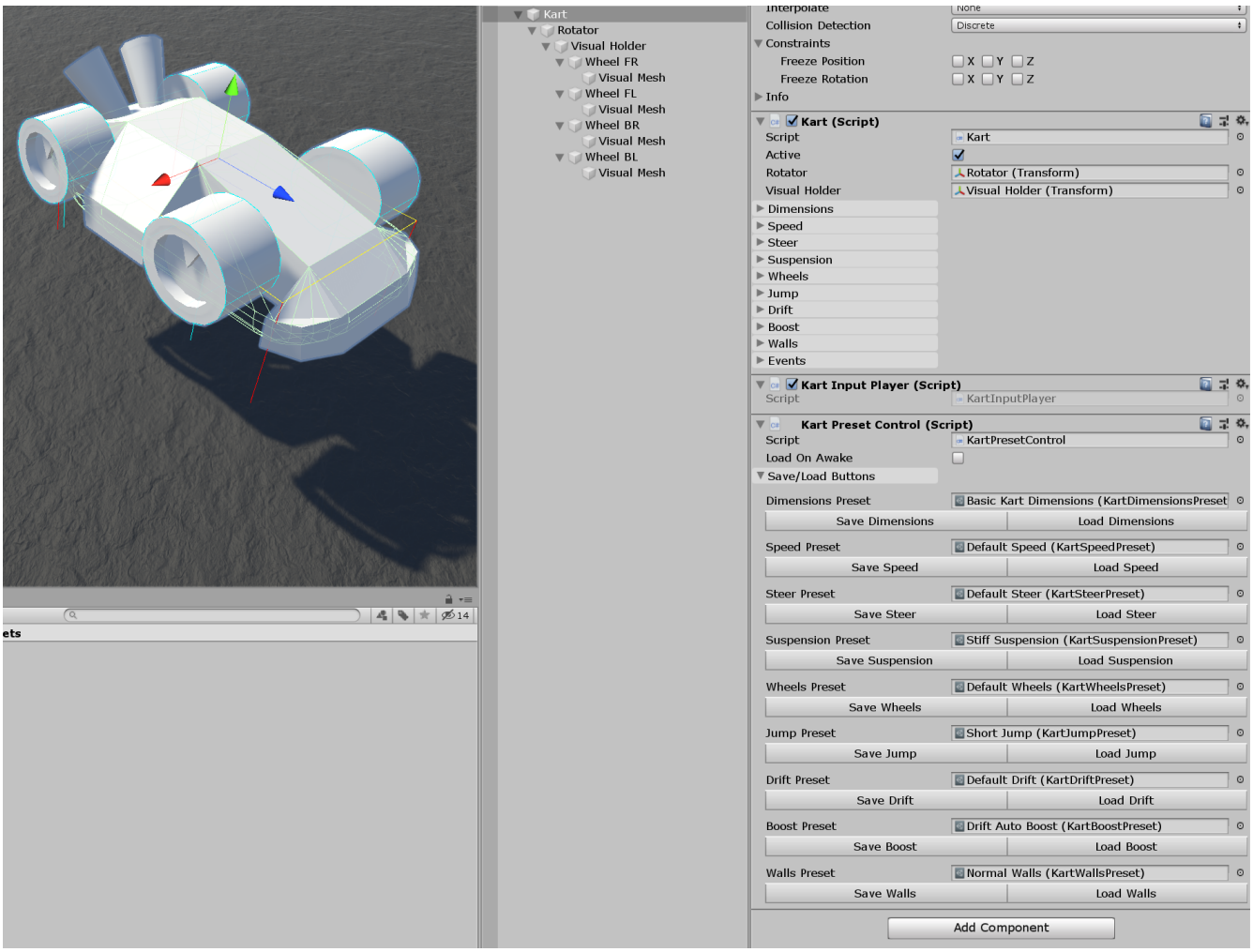
You can also save the kart's current properties to a linked preset by clicking the save button. This will copy the kart's variables from the corresponding foldout section on the *Kart* component to the linked preset. This is an easy way to save changes to properties made in play mode. Click the save button while in play mode, then click the load button after exiting play mode to reload the changes made in play mode.

Be careful with saving over the included presets as you may want to use them as a reference, but you can also reimport the original presets from the Asset Store. You can also fetch properties from example prefabs by linking presets to them and clicking the save buttons.

Custom Gravity

It's worth mentioning that custom gravity (adjusted on the *Kart* component) stacks on top of the global physics gravity applied to all rigidbodies. You will likely want to disable “Use Gravity” on a kart's rigidbody if the gravity is going to vary based on the driving surface angle or otherwise change from the global gravity. Make sure to adjust the “Gravity Add” variable to compensate for the disabled global gravity (add the magnitude of Physics.gravity to “Gravity Add”). “Gravity Add” should be negative in case the ground normal is used to adjust gravity.

KartPresetControl component with linked presets:



Note that the wheels array is not saved with presets as it depends on actual objects in the hierarchy. Some properties you may need to change for your unique kart include the corner cast variables in the “Dimensions” section. “Corner Cast Size,” “Corner Cast Offset,” and “Corner Cast Distance” are used to configure raycasts at the corners of the kart used to detect a special state of being “air grounded.”

This is like an intermediate state between being on solid ground and moving freely in the air. An example would be when a kart is just barely above the ground while jumping just before entering a drift. In kart racers, it's common for karts to retain their handling and behave as if they're still grounded during short jumps. However it is also common for karts to have reduced steering capability while making large jumps off of ramps, hence the distinction between “air grounded” and “not grounded at all.”

In the *Kart* script there are hidden Boolean variables “grounded” and “airGrounded” indicating these states. Both being false means the kart is completely in the air.

Corner cast properties are visualized with yellow rectangle and red line gizmos.

The “Front Length,” “Back Length,” and “Side Width” variables influence how a kart is offset when it tilts, so you may need to adjust these values if your kart either clips into the ground or floats above the ground while doing wheelies or tilting sideways. The “Steer” section also contains important properties that affect how a kart tilts as it turns. This is mainly useful for bikes.

Boost

There are three different boost types supported by this asset. You can adjust them in the “Boost” section of the *Kart* component and there are presets and example prefabs for each. They work as follows:

- ***Drift Auto*** (similar to boost mechanics in *MK*):
  - Begin drifting.
  - While drifting, the boost meter will fill up several times.
  - Each time the boost meter fills, a new boost level is reached and spark particles coming from the rear wheels change color.
  - The boost meter will stop filling once the maximum boost level is reached.
  - After ending a drift, boost is given to the kart based on the boost level reached.
- ***Drift Manual*** (similar to boost mechanics in *CTR*):
  - Begin drifting.
  - While drifting, the boost meter will fill up.
    - If the boost button is pressed while the meter is green, boost will be given to the kart based on the boost level.
    - If the boost button is pressed while the meter is red, or if the meter fills up all the way, the boost will fail and a new drift will have to be started.
  - With each successful boost, a new boost level is reached.
  - The boost meter will stop filling once the maximum boost level is reached or the boost fails.
- ***Manual*** (similar to standard boost mechanics in racing games):
  - If the boost meter is not empty, the boost button can be held down to give the kart boost (does not require drifting).
  - If the boost meter is empty, nothing will happen.
  - Drifting fills up the boost meter until it's full.

There is also a hidden “reserve” mechanic in the boost system. The way this works is that all boost sources fill up the reserve by varying amounts, but the reserve automatically depletes over time. This way skilled players can drift and boost efficiently to maintain continuous

boosting if your karts are configured to allow it (the example karts are). The “Boost Burn Rate” and “Boost Reserve Limit” variables affect how this feature works. They respectively change how quickly the reserve depletes and the maximum limit for the reserve amount. The hidden reserve variable is named “boostReserve.”

The boost mechanics are more apparent with UI, audio, and particle effects set up, so we will configure those features next.

UI Setup

There is a UI prefab you can use in the *Prefabs > Scene* folder, or you can use the existing “Kart UI” object in the demo scene (inside of the Canvas object). Once you have the UI object in the scene with the *UIControl* component attached, just drag the kart into the “Target Kart” variable on the script. Once you enter play mode, the UI should automatically reflect the kart's status. In the demo scene, you should also deactivate the “Demo Menu” object inside of the Canvas otherwise it will obstruct the game view.

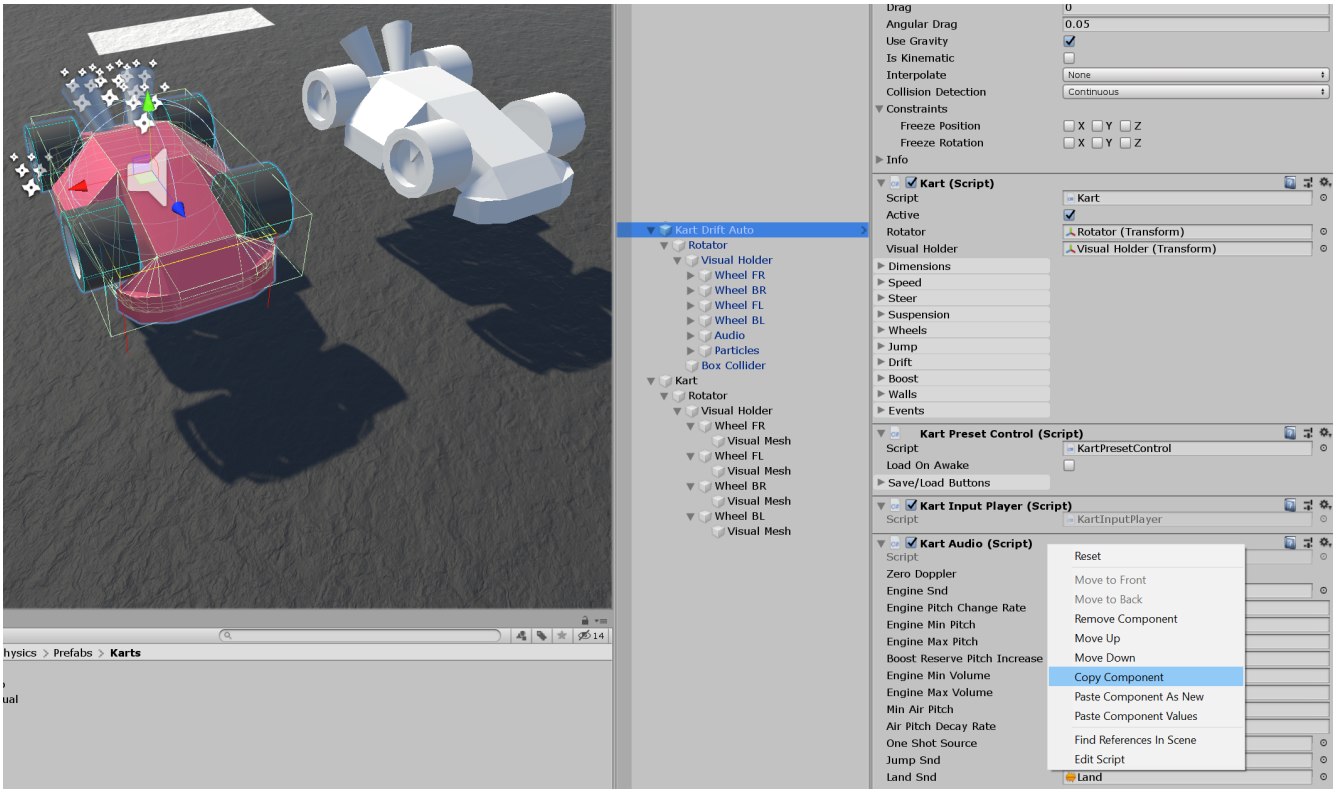
Audio Setup

The easiest way to configure sound effects for a kart is to copy them from one of the example prefabs. This is the process that will be described here, but you can manually set up the objects and components if you wish.

Start by navigating to the *Prefabs > Karts* folder and drag the *Kart Drift Auto* prefab into the scene next to the kart you're working on (this is a base kart prefab from which other prefab variants in the folder are derived).

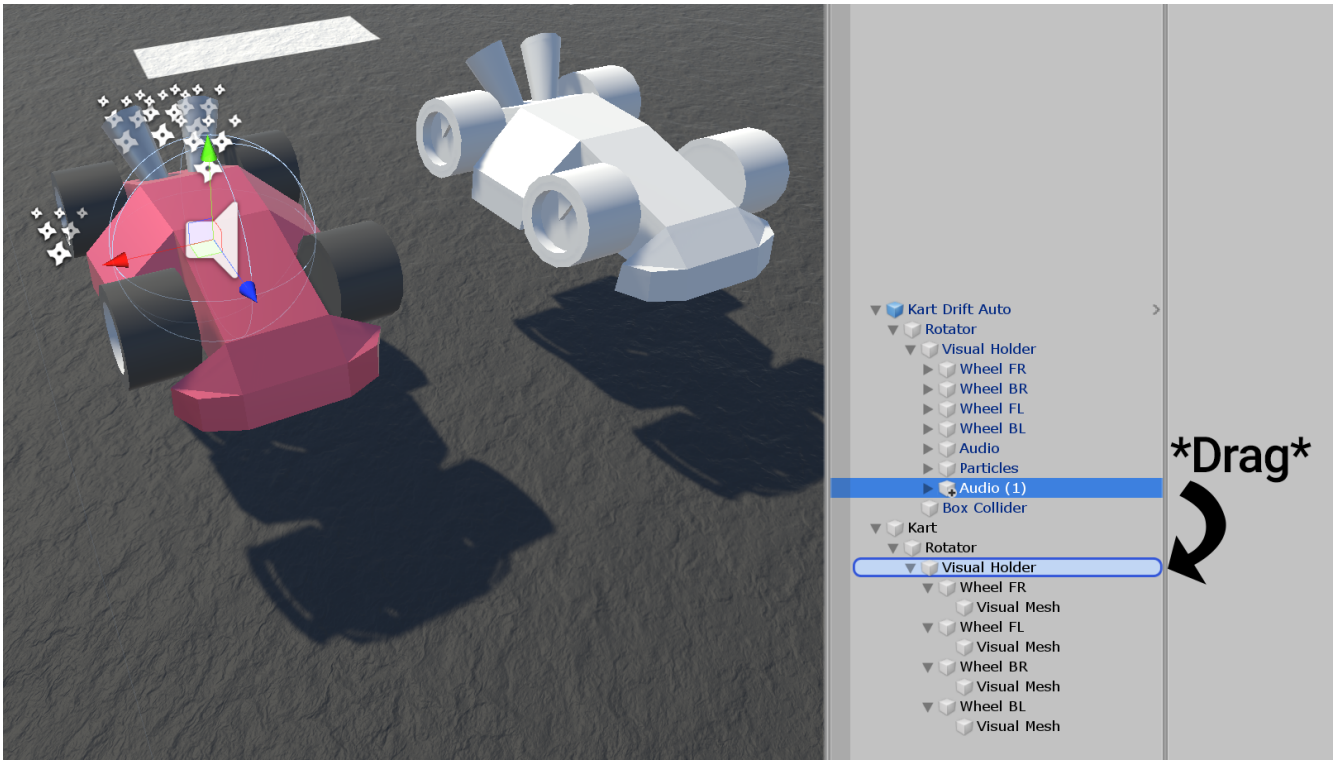
Copy the *KartAudio* component from the example kart and paste it as a new component on your kart.

Copying the *KartAudio* component:



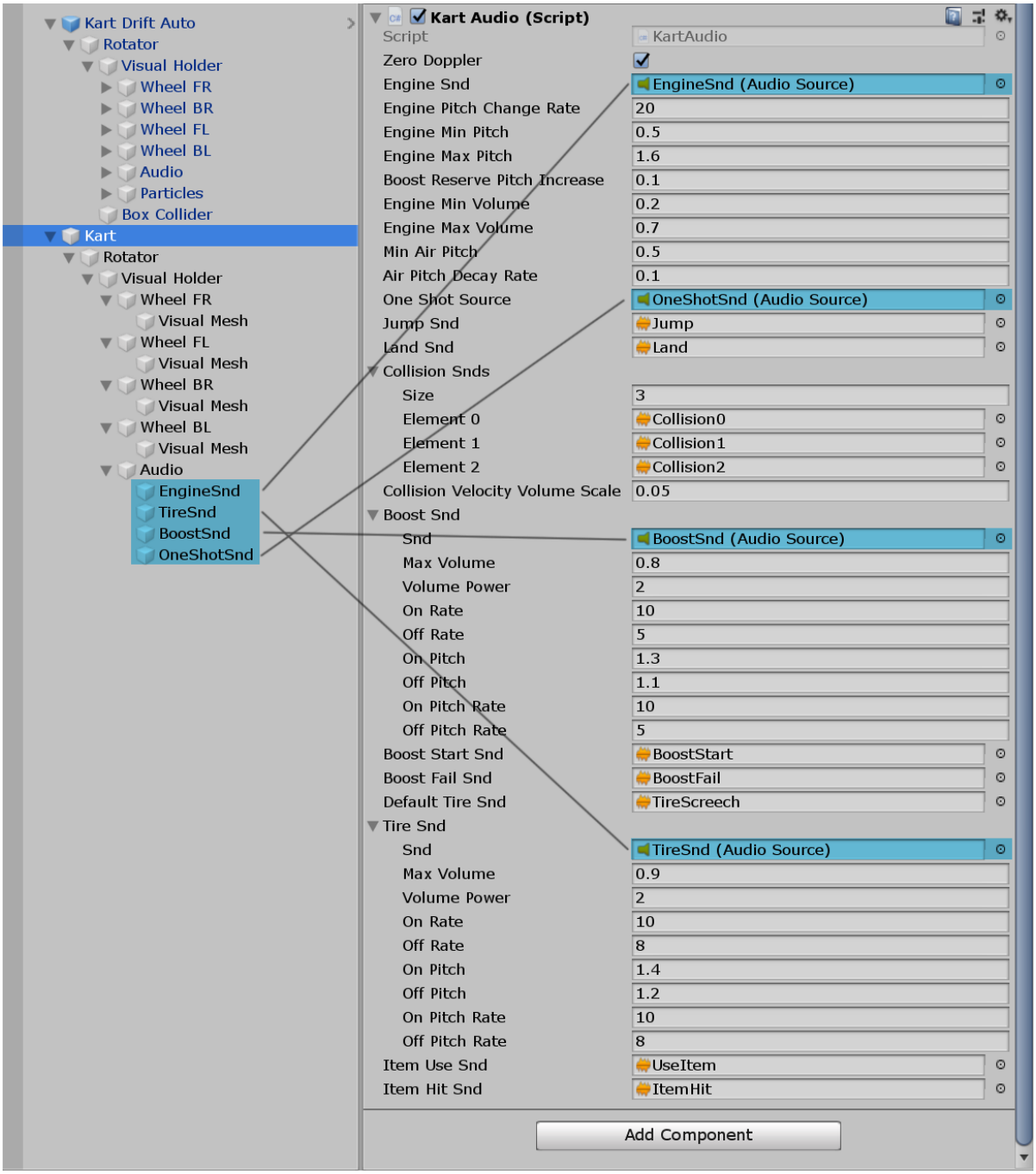
Now navigate to the “Audio” object inside of the example kart (child of visual holder). Duplicate it and drag it into the visual holder of your kart then reset its local position to zero.

Dragging the duplicated audio object:



Now go back to the *KartAudio* component on your kart. We need to link our duplicated audio sources to the script, since the copied script contains references to the audio sources on the example kart. Drag each audio source in your kart to the variable containing an audio source of the same name on the script.

Dragging the audio source references:

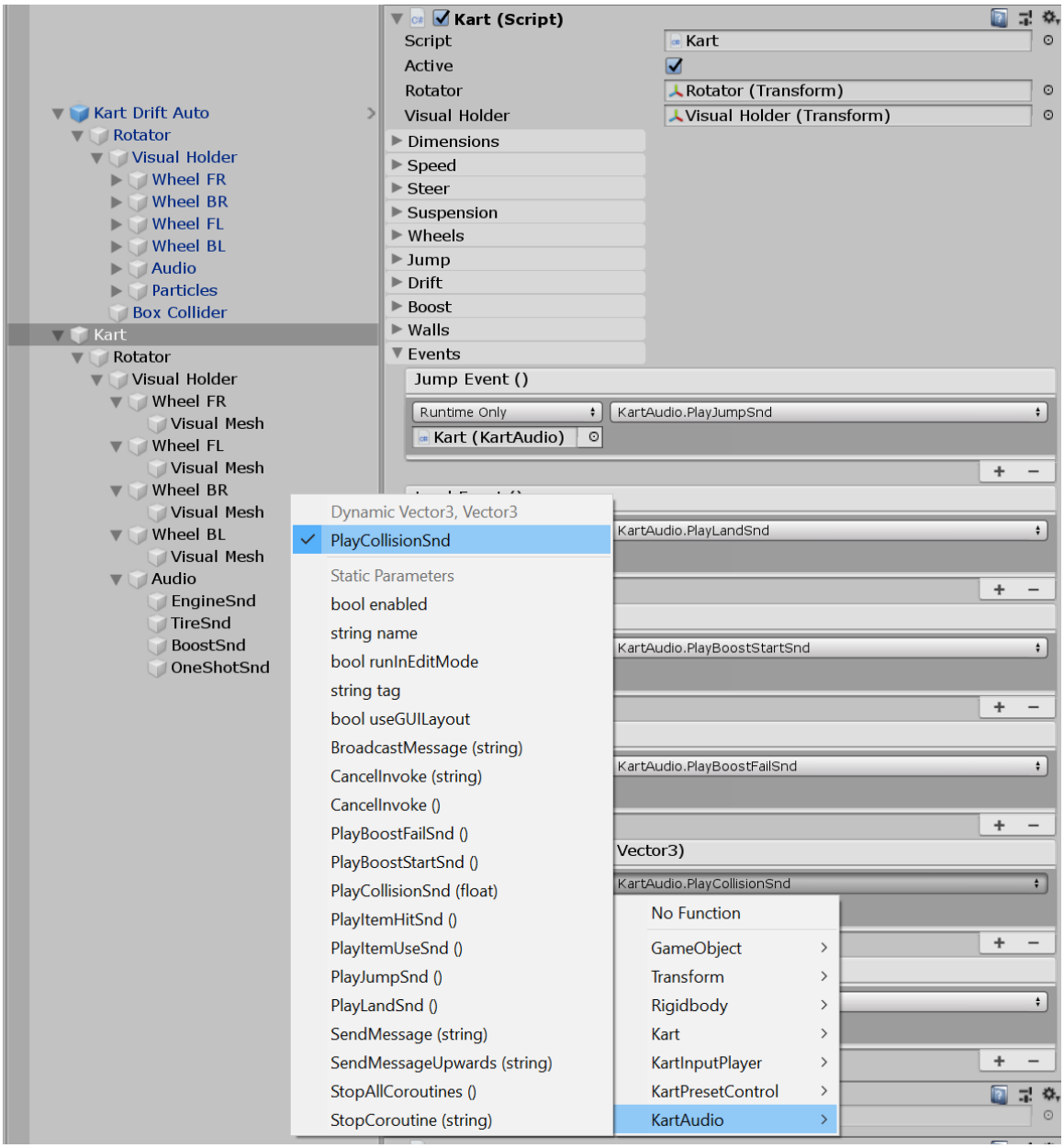


Now the looping sounds such as the engine sound, boost sound, and tire sound should all be working. We still have to connect some audio playing functions to events in the *Kart* script. These use the “OneShotSnd” audio source, used for sounds that play once and don't loop.

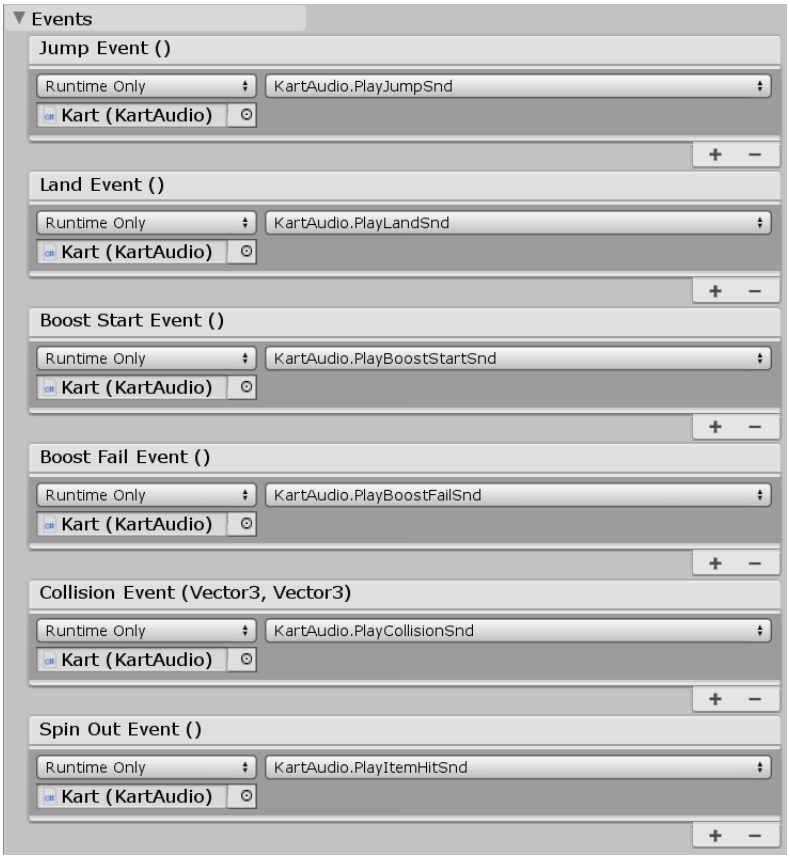
Navigate to the *Kart* component on your kart and open the “Events” foldout. Here you will see different events that are invoked when the kart does certain things. These provide a flexible way to connect the kart's behavior to audio and particle systems.

To add a function call to an event, click the “+” symbol in the bottom right of the event. Drag your kart object into the object field, since we need to call a function on the *KartAudio* script that's attached to your kart. Then click the drop-down to select the function to call. For each event, there should be a similarly named “Play____Snd()” function to choose from.

Example of linking the CollisionEvent(Vector3, Vector3) to the PlayCollisionSnd function:



Final audio event setup:



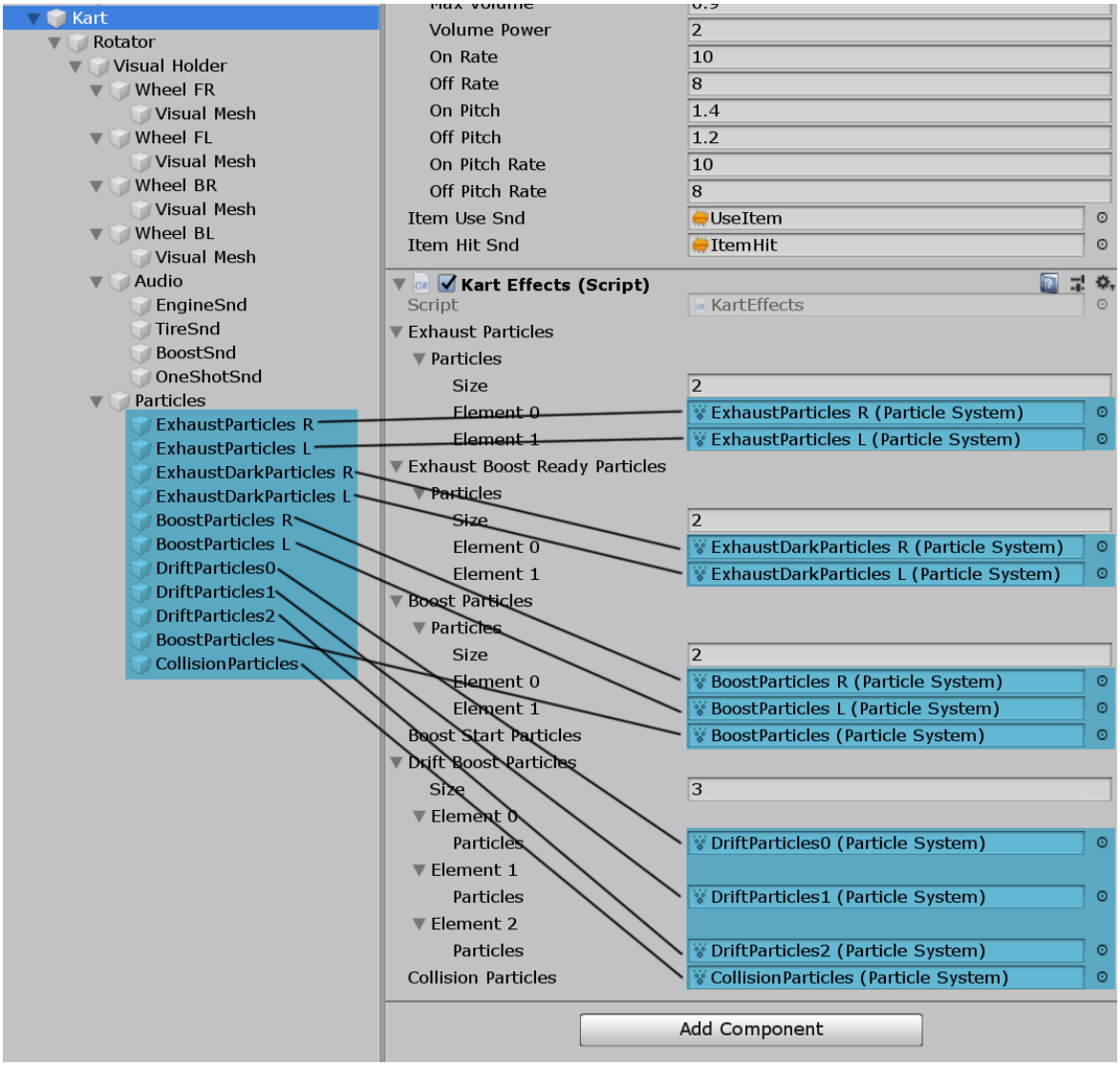
Now when you enter play mode, you should hear different sounds play for actions like jumping, boosting, and colliding with walls.

Particle Effects Setup

We will set up particles for your kart through pretty much the same method we used for the audio. If you have skipped to this section, read the previous one for a detailed walkthrough.

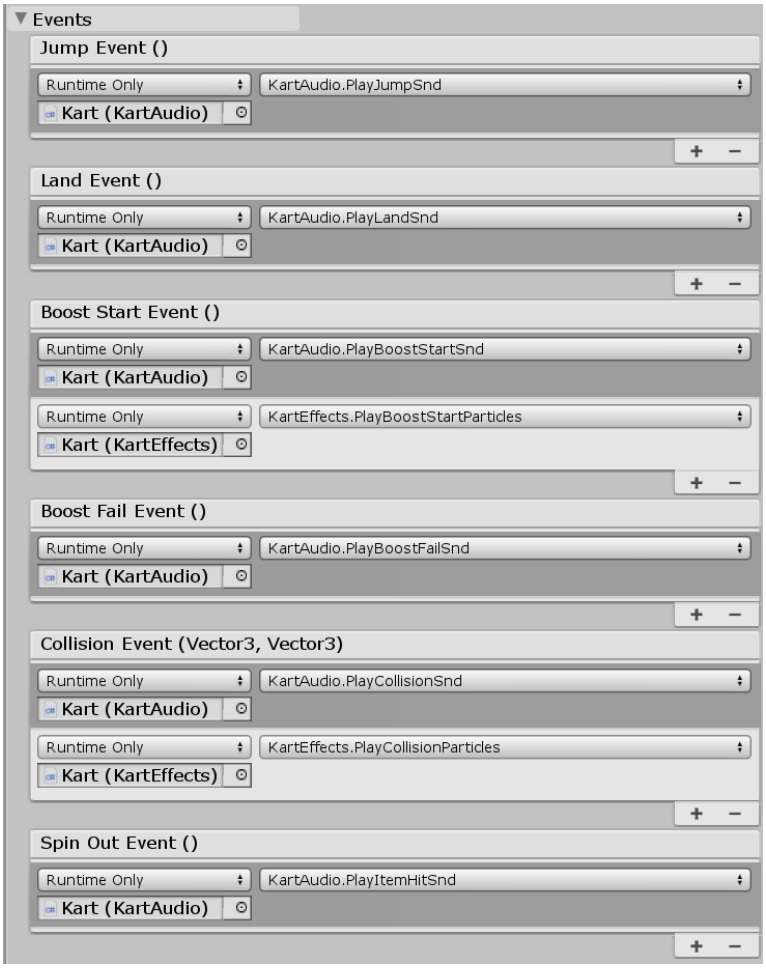
Copy the *KartEffects* component from the example kart to yours. Duplicate the Particles object, place it in the visual holder of your kart and reset its local position to zero. Drag each of the particle systems to a variable with an existing reference of the same name in the script.

Dragging the particle system references:



For the kart events, only the BoostStartEvent() and CollisionEvent(Vector3, Vector3) are meant to invoke functions in *KartEffects*. Link the events to the functions as was done with *KartAudio*.

Final event setup with particles:



The “Exhaust Boost Ready Particles” and “Drift Boost Particles” might not have a clear purpose at first. The boost ready particles are for indicating when boost is “ready,” which is a status that varies depending on the boost type.

For *Drift Auto*, this is true when at least one boost level has been reached (the meter has filled up once).

For *Drift Manual*, this is true when the boost meter has filled up enough for a successful boost to take place.

For *Manual*, this is true as long as the boost meter is not empty.

When driving around with the example karts, the boost being ready is indicated by exhaust particles turning black (hence the particle object names).

If you do not want to use particles to indicate boost ready status, you may leave the “Exhaust Particles” and “Exhaust Boost Ready Particles” arrays empty. This way the normal exhaust particles will always play.

The drift boost particles are for indicating the boost level with the *Drift Auto* boost type. Each

boost particle object corresponds to a boost level based on its position in the array. The example particles progress in the following order: blue, orange, then pink (similar to *MK*).

They also switch sides depending on the drift direction of the kart. When setting up these particles for your own karts, make sure to place them on one side of the kart (do not duplicate them for the opposite side). The *KartEffects* script automatically flips the particle system as needed.

Tire Marks Setup

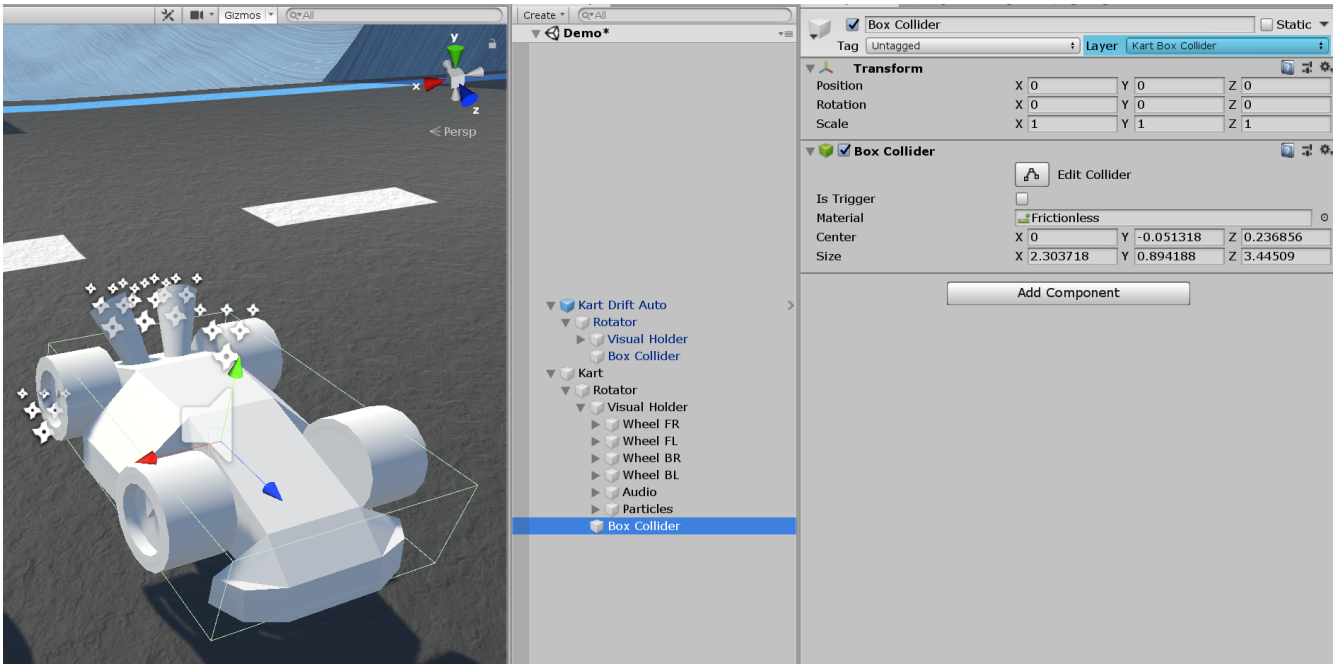
To enable tire mark creation for your kart, simply add a *TireMarkMaker* component to each wheel (specifically the same object with the *KartWheel* component attached). You can also copy the component from the example kart. Make sure to choose a material for the "Default Tire Mark Material" property. There is a tire mark material included in the *Materials* folder.

Kart Box Collider

It is recommended that your karts have a special collider just for colliding with other karts. This is because the smooth collider used for colliding with environmental geometry is not suitable for kart collisions. The smoothness leads karts to launch each other upwards.

The kart box collider is just a box collider on another child within the rotator object. This is the only child object of a kart that is on a different layer, that layer being "Kart Box Collider." This layer is configured in the layer collision matrix in the Physics Settings in such a way that objects on this layer can only collide with other objects on the same layer.

This means that kart box colliders can only collide with other kart box colliders, and karts can still have smooth collisions with everything else. This way karts are much less likely to launch each other into the air.



Wall Collision Detection

In this asset (and kart racers in general) wall collisions are handled with special rules beyond basic rigidbody interaction. They usually slow down karts a bit or bounce them away and maybe even apply penalties like losing boost and ending drifts.

You can achieve the same effect with this asset, and there are several ways to have objects count as walls during collisions.

Open the “Walls” foldout on the *Kart* component to access options for dealing with wall collisions. You will see variables related to wall friction and bouncing that can be adjusted as needed.

There is also a “Wall Collision Props” foldout inside with properties for detecting walls. The “Wall Detection Type” drop-down shows the possible modes of wall detection:

- ***Normal*** – Walls are detected by comparing object surface normals to a reference up direction. The “Wall Dot Limit” variable determines the dot product limit for normal comparisons. Dot products under this value will register as walls (greater values mean less steep surfaces will count as walls).
- ***Layer*** – All objects on a layer contained in the “Wall Mask” layer mask are considered to be walls.
- ***Tag*** – All objects with the “Wall Tag” tag are considered to be walls.
- ***Component*** – All objects with the *Wall* component attached are considered to be walls.

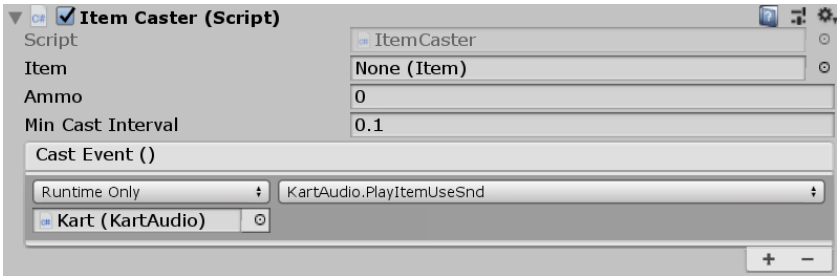
There is also a “Local Up Wall Dot Comparison” variable that if set to true, means that a kart's local up direction will be used for dot product comparisons, rather than the world up direction. This is useful for driving on sideways and upside-down surfaces like the loop in the demo, since you can drive on steep surfaces but still have walls/guardrails on the edges that are detected properly.

There are two more important options related to wall collisions within the “Boost” and “Drift” foldouts. These just allow you to set whether wall collisions should forcibly cancel boosts and drifts.

Using Items

This asset includes some basic item functionality. To allow a kart to use items, just add an *ItemCaster* component to it. To have a sound play when an item is used, connect the CastEvent() event to the *KartAudio* component as was done with the *Kart* component previously.

The configured *ItemCaster* component:



Now when your kart touches an *ItemGiver* (blue spheres in the demo scene), the kart will receive a random item to be used.

For more information on items including how to configure them, see the Item Setup section below.

Raycast Optimization

Two options for reducing the processing load of kart physics are “One Corner Cast Per Frame” and “One Wheel Cast Per Frame,” located in the “Dimensions” and “Wheels” sections of the *Kart* component respectively.

These options enable having only one raycast per frame as opposed to all simultaneously. This means all raycasts that would normally happen together are cycled through, one each frame. This is recommended for use with AI-controlled karts rather than player-controlled karts unless absolutely necessary since ground collision information will be somewhat delayed and less precise.

One more performance option is the “Max Wheel Cast Hits” variable in the “Wheels” section. This is the maximum number of hits to check with each raycast to get a valid ground hit. In some cases you may have a kart set up in a way that the wheel raycast could potentially hit the kart's own collider. Checking multiple hits allows for the physics system to discard invalid hits in favor of valid ground hits, unlike with typical raycasting where only one hit would be checked. (Internally, this is a result of using *Physics.RaycastNonAlloc()* to avoid garbage collection.)

Item Setup

Items in this asset should be considered a bonus feature, included because games with karts almost always feature items/powerups. The rationale is that kart physics should have at least some support for interactions with items.

Item Caster

The *ItemCaster* component is what allows karts to use items. See the previous section about configuring it.

Item Manager

The *ItemManager* provides functionality for accessing items to equip. In the *Prefabs > Items* folder you can find a prefab containing this component that can be dragged into your scene. It can also be found in the demo scene.

This object has a selection of child objects, each one representing an actual item to be used. The item manager automatically detects these objects and keeps them in a private list. Notice that each child object contains a component derived from the *Item* class. Each item has a name that can be used to reference it with scripting (refer to the script reference for more information).

You can add more usable items by creating more child objects with *Item* components attached. This asset comes with *BoostItem* and *ProjectileItem* types, derived from *Item*.

Boost Item

The boost item gives a kart boost. The boost item within the item manager object contains variables for adjusting boost behavior. “Boost Amount” is how much boost a kart receives, and “Boost Force” is how much force is applied to push a kart forward when using the item.

A new boost item can be created by creating a new child object of the item manager with the *BoostItem* component attached as described previously.

Projectile Items

Projectile items are spawned objects with rigidbodies that can move around to hit karts. Within the item manager object are several projectile item spawners that can be configured. Each one has an “Item Prefab” field for the actual spawned projectile item and a “Spawn Offset” field for offsetting the spawned item in local space relative to the kart.

The *SpawnedProjectileItem* component is what controls the spawned items. You can find prefabs for these items in the *Prefabs > Items* folder. This component has a fair amount of customizable properties allowing for behaviors including bouncing off of walls and following certain karts. See the script reference below for descriptions of every variable.

A new projectile item can be created by creating an empty object, adding a *SpawnedProjectileItem* component to it, adding a collider to it, and adding a renderer to it. You may want to restrict the rotation of the item and give its collider a “Frictionless” physic material. Its rigidbody should have a low mass so as to not push karts around upon collision and continuous collision detection enabled in order to not pass through objects. See the included item prefabs as examples.

Note that wall collision detection for items functions much like it does for karts as described in a previous section. Gravity settings also stack on top of global gravity as described here.

Item Giver

The *ItemGiver* is what karts can collide/overlap with to equip items. You can find an item giver prefab in the *Prefabs > Items* folder. To create an item giver, just create an empty object and add an *ItemGiver* component, trigger collider, and renderer to it. Item givers will only give items to karts that do not already have an item equipped (this includes the ammo being zero even if there is an existing item reference).

The *ItemGiver* component has a few properties for adjusting how items are given to karts. The “Item Name” variable indicates the name of the item to give, this being either the name of the item object inside the item manager or the name variable on the *Item* component. If this is left blank, then a random item will be given from the selection of all items within the item manager.

“Ammo” is how many times an item can be used by a kart once equipped and “Cooldown” is the duration in seconds that the item giver is deactivated after giving an item.

Environment/Level Features

This section covers features for having karts interact with environmental geometry.

Ground Surface Types

A ground surface type indicates how kart wheels should behave on different surfaces. This includes things like friction, speed, tire mark materials, and tire mark sounds. To enable the used of ground surface types, just add a *GroundSurface* component to any object with a collider.

This component has a single field for a *GroundSurfacePreset*, a class containing the ground surface properties. This is a *ScriptableObject* .asset file that can be used on any *GroundSurface* component to easily have multiple objects share the same surface properties. See the script reference below for information about these variables.

You can create a *GroundSurfacePreset* by navigating to the project window and clicking on *Create > Powerslide Kart Physics > Ground Surface Preset*. There are a few presets included in the *Prefabs > GroundSurfacePresets* folder.

Ground Surface Types on Terrain

With the *TerrainSurface* component, you can easily link ground surface types to different terrain textures. Just add a *TerrainSurface* component to any terrain object. This component should automatically populate a list of textures detected on the terrain as well as a list of associated ground surface types.

Just drag a desired *GroundSurfacePreset* to each open field in the list of corresponding ground surface properties. The index of each surface type is linked to the texture with the matching index in the list.

When driving on the terrain, each kart wheel will automatically use the surface type associated with the most visible texture at its position on the terrain.

Boost Pads

The *BoostPad* component is meant for any trigger that should act as a boost pad when a kart drives over it. This component has a few properties for configuring boost behavior.

“Boost Amount” is how much boost a kart is given upon overlapping the boost bad. “Boost Force” is the force used to push a kart forward when it touches the pad. “Delay Interval” is how long the kart must wait in seconds before it can touch another boost pad and receive a boost. Normally a kart will only receive an instant boost burst when it first touches the boost pad. “Continuous” can make it so that a kart will always get boost as long as it's touching the boost pad. If this is enabled, the boost amount given will be scaled by the fixed timestep.

Hazards

The *Hazard* component can be added to any object with a collider that should make a kart spin out upon touching it. This supports triggers too.

The “Spin Axis” variable determines which axis a kart spins about and the “Spin Count” variable is how many times a kart should spin after hitting the hazard.

Walls

See the Wall Collision Detection section above for detailed information about wall collisions. Depending on the wall detection method used, wall objects in your project may need to be either placed on a certain layer, given a special tag, or given the *Wall* component.

Basic Waypoint AI

This asset includes a very rudimentary waypoint-following AI system to demonstrate the kart physics in the demo.

Setting Up Waypoints

Create an empty object and add a *BasicWaypoint* component to it. The “Next Point” variable indicates the next waypoint in the path for a kart to follow. The “Radius” variable is the range determining how close a kart must get to the waypoint before moving on to the next waypoint.

Waypoint AI Input

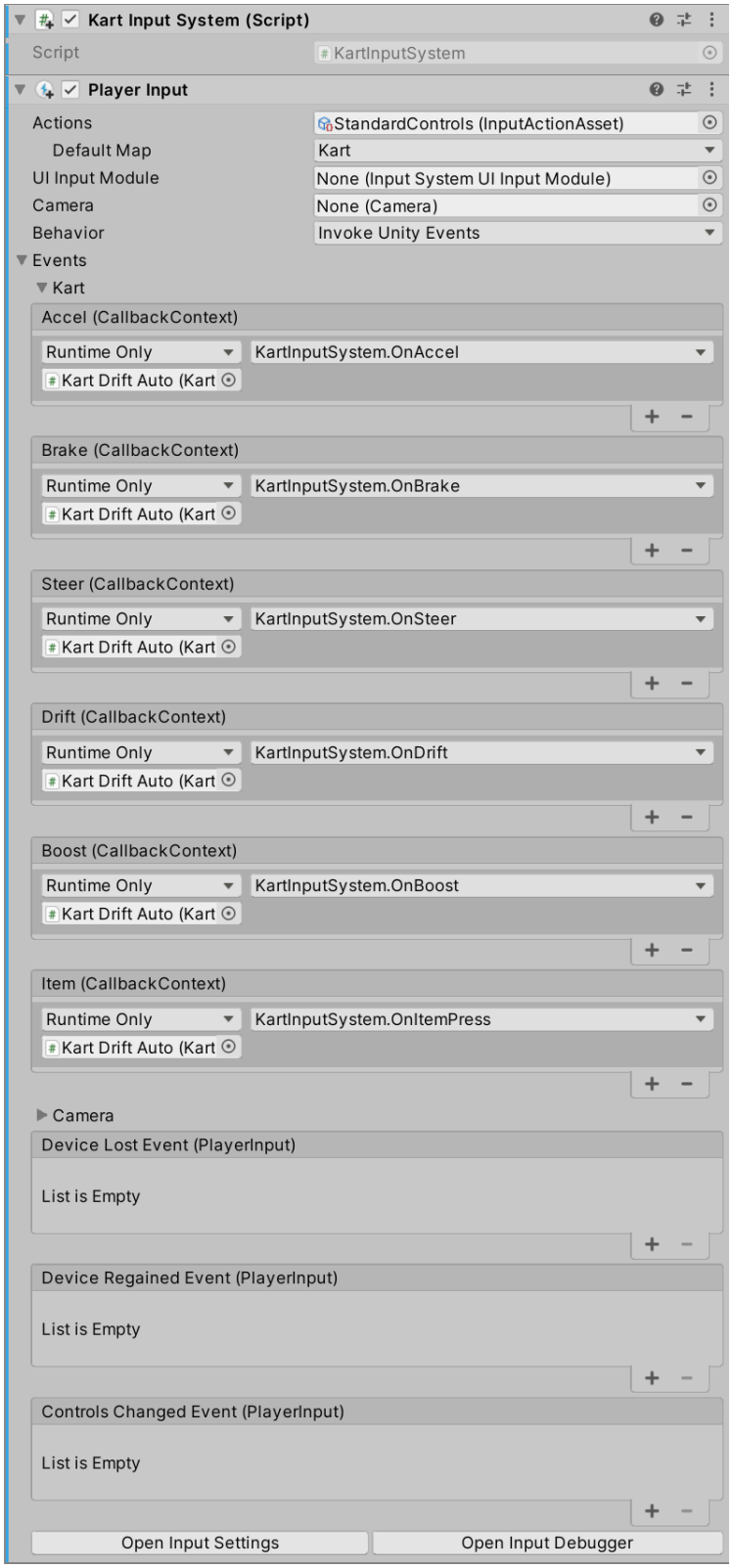
To make a kart follow waypoints, just add either a *BasicWaypointFollower* or *BasicWaypointFollowerDrift* component to it. If the kart already has a *KartInputPlayer* component attached, you must remove that first. Drag the first waypoint the kart should drive to into the “Target Waypoint” field. Once you enter play mode, the kart should automatically follow the waypoint path. See the script reference for more information about variables in these scripts.

Mobile Input

Mobile input is supported through the use of special functions in the *InputManager* class and the use of the *KartInputMobile* class. See the “Demo Mobile” scene for an example. The *Prefabs > Karts > Mobile* folder contains karts with the *KartInputMobile* script attached. This script fetches input from a static *MobileInputStruct* in *InputManager*. The static input is set by the mobile input functions in *InputManager*, which are linked to UI buttons in the “Demo Mobile” scene within the *Canvas > Kart UI > Mobile Buttons* object. See the script reference of the related classes for more information.

Input System

Use of the Input System requires that player-controlled karts have both a *KartInputSystem* component and a *Player Input* component attached. The *Player Input* component must have its behavior set to invoke Unity events, and the events for each input action must be hooked up as in the screenshot to the right.



The kart prefabs in the *Prefabs* > *Karts* > *InputSystem* folder are already configured as needed. The “Demo (Input System)” scene is also set up to work with the Input System.

To make a following camera respond to input, it must also have a *Player Input* component attached and the “Use Legacy Input” variable on its *KartCamera* component must be set to false. Callbacks on the *Player Input* component must be hooked up in a similar manner to karts. See the “Main Camera (Input System)” prefab in the *Prefabs* > *Scene* folder for an example, or look in the “Demo (Input System)” scene.

For mobile input, the only change necessary to work with the Input System is to make sure your scene contains an object with the *Input System UI Input Module* component attached, which is already the case in the “Demo Mobile (Input System)” scene.

To configure input bindings, just edit the *StandardControls* asset in the *Prefabs* > *InputSettings* folder. That same folder also contains general Input System settings in the *InputSystem.inputsettings* file.

Script Reference – Classes

Scripts in this asset are contained within the *PowerslideKartPhysics* namespace. When referencing associated classes and functions, make sure to add “using PowerslideKartPhysics;” to the top of your script(s).

Kart Classes

Kart

This is the main class for controlling kart behavior and physics, intended to be placed on the root transform of a kart object.

Enumerations:

KartBoostType – The type of boost a kart uses (declared outside of *Kart* class).

> **DriftAuto** – Drifting fills up a meter that can be filled multiple times while drifting. Each time the meter is filled, the boost level increases which is used to give the kart boost upon ending the drift. The boost amount awarded correlates directly to the boost level and not the meter fill amount. (Comparable to *MK*).
>
> **DriftManual** – Drifting fills up a meter once. The boost button must be pressed within a certain time window in order to give the kart boost, otherwise the boost will be canceled and a new drift must be started. Successful boosts can be repeated a certain number of times during a single drift. Boost awarded correlates to boost meter fill amount as well as the number successful boosts. (Comparable to *CTR*).
>
> **Manual** – Holding down the boost button gives boost to the kart as long as the boost meter is not empty. The boost meter decreases as boost is used. (Comparable to common boost mechanics in racing games).

SpinAxis – Axis about which a kart spins when spinning out (declared inside of *Kart* class).

> **Yaw** – Spin around local y-axis.
>
> **Pitch** – Spin around local x-axis.
>
> **Roll** – Spin around local z-axis.

Public Variables:

active – Whether the kart is listening to input. If false, all input is ignored.

rotator – The child object that is rotated in local space, representing the rotation of the entire kart. The kart's rigidbody does not rotate, only moves.

visualHolder – The child object containing the visual representation of the kart (child of *rotator*).

Dimensions Category:

rotationRateFactor – Multiplier for the rotator rotation rate.

minRotationRate – Minimum rotator rotation rate.

maxRotationRate – Maximum rotator rotation rate.

visualRotationRate – Rotation rate for the visual holder.

airFlattenRate – Rate at which the kart rotates upright in while airborne.

frontLength – Front size used for determining offset while tilting.

backLength – Back size used for determining offset while tilting.

sideWidth – Side size used for determining offset while tilting.

cornerCastSize – Rectangular size for positioning the corner cast points.

cornerCastOffset – Local offset for positioning the corner cast points.

oneCornerCastPerFrame – If true, only one corner raycast will be checked each frame, cycling through all of them.

cornerCastDistance – Distance checked with corner raycasts.

maxCollisionContactPoints – Maximum number of collision contact points that can be evaluated during OnCollisionEnter() and OnCollisionStay(). For optimization purposes, this cannot be changed at runtime.

spinRate – Rate at which spin out rotations progress.

spinHeight – How high the kart should raise off the ground during pitch and roll spin outs. (Yaw spin outs keep the kart grounded.)

Speed Category:

maxSpeed – Top speed of the kart.

maxReverseSpeed – Top speed in reverse.

acceleration – Acceleration rate.

brakeForce – Brake slowing rate.

coastingFriction – Rate at which the kart slows down while coasting with no input.

slopeFriction – Friction factor for slopes, lower values increase slippage on steeper slopes.

airDriveFriction – Friction factor for accelerating and braking while airborne.

autoStopSpeed – Speed under which the kart will automatically stop itself (useful for parking and preventing stopped karts from rolling freely).

autoStopForce – Force applied to automatically stop the kart.

maxFallSpeed – Maximum falling speed.

spinDecel – Rate at which the kart slows down while spinning out.

Steer Category:

steerRate – Rate at which the steering input changes.

maxSteer – Maximum steer amount (for slower speeds).

minSteer – Minimum steer amount (for higher speeds).

airSteer – Steer rate factor while airborne.

steerSpeedLimit – Speed above which minimum steer is applied.

steerSlowLimit – Speed under which maximum steer is applied.

brakeSteerIncrease – Increase in steer rate when braking.

dontInvertSteerReverseAccel – If true, steering will not be inverted while moving backwards and accelerating. This option gives the player more control while struggling up a steep slope. If false, the kart will always rotate in the opposite direction it's steering while moving backwards (typical vehicle behavior).

visualSteerRate – Steer rate of visual objects (See use of GetVisualSteer() below).

visualSteerSpeedLimit – Speed at which visual steer minimizes.

turnTiltAmount – Amount to tilt sideways while turning.

turnTiltReferenceSpeed – Speed at which sideways tilt is maximized.

turnTiltRate – How quickly the sideways tilt changes.

turnTiltSideOffsetFactor – Sideways offset proportional to sideways tilt.

invertTurnTiltHeightOffset – If true, the visual holder will be moved down instead of up with sideways tilting.

localTiltOffsetCompensation – Extra local up offset factor based on sideways tilt.

accelTiltAmount – Amount to tilt in any direction based on acceleration of the kart.

sidewaysFriction – Sideways friction of the kart.

airSidewaysFriction – Sideways friction of the kart while airborne.

brakeSlipAmount – Reduction in sideways friction based on braking.

Suspension Category:

springForce – Suspension spring force.

springDampening – Suspension spring dampening, counteracting the suspension velocity.

springDampVelMin – Minimum local velocity used for dampening calculations.

springDampVelMax – Maximum local velocity used for dampening calculations.

compressionSpringFactor – Proportion of how much the suspension compression affects the spring force.

groundStickForce – Force for keeping the kart on the ground.

groundStickCompression – Suspension compression amount above which the ground stick force is applied (normalized compression value).

Wheels Category:

wheels – The wheels of the kart.

wheelCastMask – Layer mask for objects that wheels can collide with.

maxWheelCastHits – Maximum number of raycast hits for each wheel. Every hit is checked for validity; more than one is useful for cases where the kart itself may have geometry that can be hit and should be ignored. For optimization purposes, this cannot be changed at runtime.

oneWheelCastPerFrame – If true, each wheel's raycast is checked on a separate frame, cycling through each wheel over a series of frames, one for each wheel.

groundNormalSmoothRate – Rate at which the smoothed ground normal changes.

Jump Category:

canJump – Whether the kart can jump.

jumpForce – Force for jumping.

jumpDuration – Duration for which jump force is applied.

jumpStickForce – Ground stick force applied immediately after a jump.

airJumpTimeLimit – Time in seconds being airborne during which a kart can still jump. This allows a kart to jump for a brief period after leaving ground.

Gravity Category:

gravityAdd – Continuous gravity force applied to the kart on top of the standard gravity.

gravityDir – Initial gravity direction for the kart, normalized at runtime.

gravityIsGroundNormal – If true, gravity for the kart is set to the ground normal direction while grounded. If false, the gravity remains at gravityDir.

airGravityMode – How the gravity should change when the kart is airborne. *Initial* leaves the gravity at gravityDir, or the ground normal when grounded and gravityIsGroundNormal is true. *NearestSurface* points the gravity toward the nearest surface point detected based on other variables. *LastSetDirection* leaves the gravity at what it last was when grounded, possibly varying based on the ground normal.

gravityCastLayers – Number of vertical “layers” of gravity sphere casts arranged around the kart.

gravityCastSegments – Number of gravity sphere casts in each layer.

gravityCastRadius – Radius of each gravity sphere cast.

gravityCastDistance – Distance of each gravity sphere cast.

gravityCastsPerFrame – Maximum gravity sphere casts checked each physics step. The total number of casts to find the nearest surface point is gravityCastLayers * gravityCastSegments, potentially resulting in many casts necessary to scan all around the kart. This helps to spread out the casts over multiple frames to improve performance at the cost of responsive accuracy.

drawGravityCastGizmos – Whether to visualize the gravity sphere casts.

Drift Category:

canDrift – Whether the kart can drift.

canDriftInAir – Whether the kart can drift while airborne.

minDriftAngle – Widest allowed drift turn rate.

maxDriftAngle – Sharpest allowed drift turn rate.

visualDriftFactor – Factor for turning the visual holder while drifting.

visualDriftAirFactor – Factor for turning the visual holder in the air just before drifting.

driftSwingDuration – Duration in seconds that an outward force is applied during a drift start.

driftSwingForce – The force amount applied during a drift swing.

minDriftSpeed – Minimum speed at which drifting is allowed.

wallCollisionCancelsDrift – If true, colliding with a wall while drifting will end the drift.

brakeCancelsDrift – If true, braking while drifting will end the drift.

burnoutSpeed – Movement speed while doing a burnout. Burnouts are initiated with simultaneous acceleration and brake input.

burnoutSpeedLimit – Maximum speed at which doing a burnout is possible.

Boost Category:

boostType – Type of boost to use, either *DriftAuto*, *DriftManual*, or *Manual*.

canBoost – Whether the kart can boost at all, including from boost items and boost pads.

boostSpeedAdd – Speed increase while boosting.

boostAccelAdd – Acceleration increase while boosting.

boostDrive – Drive input increase while boosting. This allows the kart to drive while boosting even without acceleration input.

boostPower – Multiplier for awarded boost.

boostRate – Rate at which boost time increases for *DriftAuto* and *DriftManual* (changing duration of boost levels). For *Manual*, this is the rate at which boost reserves increase while boosting.

boostBurnRate – Rate at which boost reserves decrease, and the rate at which the boost amount decreases for *Manual*.

boostGroundPush – Forward force applied while boosting on the ground.

boostAirPush – Forward force applied while boosting and airborne.

airLandBoost – Boost awarded after landing from jumping.

boostReserveLimit – Maximum boost reserve amount.

brakeCancelsBoost – If true, boost reserves are emptied with braking.

wallCollisionCancelsBoost – If true, boost reserves are emptied with wall collisions.

boostWheelie – Wheelie tilt amount while boosting.

maxBoosts – Number of boost levels for *DriftAuto* and *DriftManual*.

autoBoostInterval – Duration of boost levels for *DriftAuto*.

driftManualBoostLimit – Beginning of time window for valid boosting with *DriftManual*.

driftManualFailCancel – If true, premature boosts will cancel the entire boost level sequence until the next drift. If false, premature boosts will immediately begin the next boost level. This applies to the *DriftManual* boost mode.

boostAmount – Amount of boost to start with for *Manual*.

boostAmountLimit – Maximum boost amount for *Manual*.

driftBoostAdd – Rate at which boost is gained through drifting for *Manual*.

Wall Category:

wallFriction – Friction for wall collisions.

wallBounceTurnAmount – Rate at which the kart turns away from a wall upon collision.

wallBounceTurnDecayRate – Rate at which the wall bounce turn time decreases.

minWallHitSpeed – Minimum speed at which the wall bounce turn is applied.

wallHitDuration – Duration of the wall bounce turn.

wallDetectionType – Wall detection method.

wallDotLimit – Maximum dot product between the wall normal and relative up direction that counts as a wall collision. Greater values mean that flatter surfaces will count as walls. Only

used for normal-based wall detection.

wallMask – Layer mask for objects that count as walls. Only used for layer-based wall detection.

wallTag – Tag for objects that count as walls. Only used for tag-based wall detection.

localUpWallDotComparision – If true, the local up direction of the kart's rotator will be used as the up direction for normal-based wall detection, rather than the world up direction.

Public Events:

jumpEvent() – Invoked when the kart jumps.

landEvent() – Invoked when the kart lands after being airborne.

boostStartEvent() – Invoked when the kart starts boosting.

boostFailEvent() – Invoked when the kart fails boosting.

collisionEvent(Vector3, Vector3) – Invoked when the kart collides with a wall. The first Vector3 contains the collision point and the second Vector3 contains the collision velocity.

spinOutEvent() – Invoked when the kart spins out.

Public Functions:

public float **GetVisualSteer**() – Returns the visual steer amount. This is used for steering the visual wheels.

public float **GetJumpedAirTime**() – Returns the time in seconds the kart has been airborne, but only if the kart jumped beforehand. This function returns zero otherwise.

public void **AddBoost**(float **boostToAdd**) – Increases the boost reserve by **boostToAdd** in the case of *DriftAuto* and *DriftManual* boost types. For *Manual* boost, the boost amount is increased.

public void **AddBoost**(float **boostToAdd**, float **pushForce**) – Same as above, but **pushForce** is added to the kart's rigidbody as a force.

public float **GetBoostValue**() – Returns the normalized boost value depending on the boost type. For *DriftAuto* and *DriftManual*, this returns the progress of the current boost level. For *Manual*, this returns the amount of boost available.

public bool **IsBoostReady**() – Returns whether the boost is “ready” depending on the boost type. For *DriftAuto*, this returns true if at least one boost level has been fulfilled. For *DriftManual*, this returns true if the current boost time is within the range for a successful boost to be awarded if the boost button is pressed. For *Manual*, this returns true if the boost amount is greater than zero.

public void **SpinOut**(SpinAxis **spinType**, int **spinCount**) – Causes the kart to spin out about the **spinType** axis for the number of spins **spinCount**.

public bool **IsWheelSliding**() – Returns whether any wheel is currently sliding/skidding.

public GroundSurfacePreset **GetWheelSurface**() – Returns the ground surface type of any grounded wheel.

public GroundSurfacePreset **GetWheelSurface**(bool **onlySliding**) – Same as above, but if **onlySliding** is true, then only wheels that are sliding/skidding are observed.

public void **SetAccel**(float **accel**) – Sets the acceleration input to the value, between 0 and 1.

public void **SetBrake**(float **brake**) – Sets the brake input to the value, between 0 and 1.

public void **SetSteer**(float **steer**) – Sets the steer input to the value, between -1 (left) and 1 (right).

public void **SetDrift**(bool **drift**) – Sets the drift input to the value.

public void **SetBoost**(bool **boost**) – Sets the boost input to the value.

Kart Preset Control

This class controls saving to and loading from presets for kart properties.

Public Variables:

loadOnAwake – If true, all connected presets will be loaded during Awake().

dimensionsPreset – Associated preset for saving and loading dimensions properties.

speedPreset – Associated preset for saving and loading speed properties.

steerPreset – Associated preset for saving and loading steer properties.

suspensionPreset – Associated preset for saving and loading suspension properties.

wheelsPreset – Associated preset for saving and loading wheels properties.

jumpPreset – Associated preset for saving and loading jump properties.

gravityPreset – Associated preset for saving and loading gravity properties.

driftPreset – Associated preset for saving and loading drift properties.

boostPreset – Associated preset for saving and loading boost properties.

wallsPreset – Associated preset for saving and loading walls properties.

Public Functions:

public void **LoadDimensionsPreset**(KartDimensionsPreset **preset**) – Loads the **preset** into the kart's properties.

public void **SaveDimensionsPreset**(KartDimensionsPreset **preset**) – Saves the kart's properties into the **preset**.

public void **LoadSpeedPreset**(KartSpeedPreset **preset**) – Loads the **preset** into the kart's properties.

public void **SaveSpeedPreset**(KartSpeedPreset **preset**) – Saves the kart's properties into the **preset**.

public void **LoadSteerPreset**(KartSteerPreset **preset**) – Loads the **preset** into the kart's properties.

public void **SaveSteerPreset**(KartSteerPreset **preset**) – Saves the kart's properties into the **preset**.

public void **LoadSuspensionPreset**(KartSuspensionPreset **preset**) – Loads the **preset** into the kart's properties.

public void **SaveSuspensionPreset**(KartSuspensionPreset **preset**) – Saves the kart's properties into the **preset**.

public void **LoadWheelsPreset**(KartWheelsPreset **preset**) – Loads the **preset** into the kart's properties.

public void **SaveWheelsPreset**(KartWheelsPreset **preset**) – Saves the kart's properties into the **preset**.

public void **LoadJumpPreset**(KartJumpPreset **preset**) – Loads the **preset** into the kart's properties.

public void **SaveJumpPreset**(KartJumpPreset **preset**) – Saves the kart's properties into the **preset**.

public void **LoadGravityPreset**(KartGravityPreset **preset**) – Loads the **preset** into the kart's properties.

public void **SaveGravityPreset**(KartGravityPreset **preset**) – Saves the kart's properties into the **preset**.

public void **LoadDriftPreset**(KartDriftPreset **preset**) – Loads the **preset** into the kart's properties.

public void **SaveDriftPreset**(KartDriftPreset **preset**) – Saves the kart's properties into the **preset**.

public void **LoadBoostPreset**(KartBoostPreset **preset**) – Loads the **preset** into the kart's properties.

public void **SaveBoostPreset**(KartBoostPreset **preset**) – Saves the kart's properties into the **preset**.

public void **LoadWallsPreset**(KartWallsPreset **preset**) – Loads the **preset** into the kart's properties.

public void **SaveWallsPreset**(KartWallsPreset **preset**) – Saves the kart's properties into the **preset**.

Kart Wheel

This class represents individual kart wheels. All wheel properties other than suspension distance are cosmetic and do not affect kart behavior.

Public Variables:

suspensionDistance – The distance that the suspension reaches.

maxExtension – Factor limiting suspension extension (maximum extend position for wheel).

airExtendRate – Rate at which the wheel extends while not grounded.

compressionTiltAmount – Amount to tilt the wheel as it moves up and down.

radius – Radius of the wheel.

width – Width of the wheel.

steerAmount – Amount the wheel is rotated with the kart's steering.

driven – Whether the wheel is driven. (Determines if it rotates during burnouts.)

visualWheel – The visual wheel that is rotated.

burnoutRotateSpeed – Rate of rotation during burnouts, if driven.

Public Functions:

public void **SetSurface**(GroundSurface **surface**) – Sets the current ground surface type to the given **surface**.

Kart Audio

This class controls a kart's audio sources.

Public Variables:

zeroDoppler – If true, all audio sources on the kart will have their Doppler level set to zero in Awake().

engineSnd – The engine audio source.

enginePitchChangeRate – Rate at which the engine pitch changes based on kart speed.

engineMinPitch – Minimum engine pitch.

engineMaxPitch – Maximum engine pitch.

boostReservePitchIncrease – Amount to increase engine pitch based on the kart's boost reserves.

engineMinVolume – Minimum engine volume.

engineMaxVolume – Maximum engine volume.

minAirPitch – Minimum engine pitch factor while airborne.

airPitchDecayRate – Engine pitch decrease rate while airborne.

oneShotSource – Audio source for one-shot (non-looping) sounds.

jumpSnd – Jump sound clip.

landSnd – Land sound clip.

collisionSnds – Collision sound clips.

collisionVelocityVolumeScale – Velocity factor for collision sound volume.

boostSnd – *ConditionalSound* for boosting.

boostStartSnd – Boost start clip.

boostFailSnd – Boost fail clip.

defaultTireSnd – Default tire screech clip.

TireSnd – *ConditionalSound* for tire screeches.

itemUseSnd – Item cast clip.

itemHitSnd – Clip for getting hit by an item.

Public Functions:

public void **PlayJumpSnd**() – Plays the jump sound.

public void **PlayLandSnd**() – Plays the land sound.

public void **PlayCollisionSnd**(Vector3 **pos**, Vector3 **vel**) – Wrapper for the next function. This is to allow collision sounds to be linked to the CollisionEvent() in the *Kart* class. The **pos** parameter is not used, but the **vel** magnitude is used for scaling the volume.

public void **PlayCollisionSnd**(float **volume**) – Plays a random collision sound with the given **volume**, scaled by collisionVelocityVolumeScale.

public void **PlayBoostStartSnd**() – Plays the boost start sound.

public void **PlayBoostFailSnd**() – Plays the boost fail sound.

public void **PlayItemUseSnd**() – Plays the item cast sound.

public void **PlayItemHitSnd**() – Plays the sound for getting hit by an item.

Kart Effects

This class controls particle effects for karts.

Public Variables:

exhaustParticles – *ConditionalParticles* for the exhaust.

exhaustBoostReadyParticles – *ConditionalParticles* for exhaust indicating that a kart's boost is ready. (See IsBoostReady() in the *Kart* class).

boostParticles – *ConditionalParticles* for boost.

boostStartParticles – One-shot particles for when boost starts.

driftBoostParticles – Array of *ConditionalParticle* instances for each boost level associated with the *DriftAuto* boost type. This array should be the same size as maxBoosts in the *Kart* class. One particle system will be activated at a time to match the current boost level of a kart.

moveParticlesWithDrift – If true, the drift boost particles will be moved to the side corresponding to a kart's drift direction.

collisionParticles – One-shot particles for collisions. This particle system will be positioned at collision points and rotated in the direction of collision normals.

Public Functions:

public void **PlayBoostStartParticles**() – Plays the boost start particles.

Public void **PlayCollisionParticles**(Vector3 **pos**, Vector3 **dir**) – Positions the collision particles at **pos** and rotates them to face **dir**, then plays them.

Tire Mark Maker

This class creates procedural tire marks for sliding kart wheels.

Public Variables:

markHeight – Height above the ground where tire marks are placed.

markLength – Maximum number of segments (vertex pairs) in a single tire mark mesh.

markGap – Time in seconds between the addition of new segments.

lifeTime – Duration in seconds that a tire mark remains after creation ends.

fadeRate – Rate at which tire marks fade when their life ends.

markOffset – Horizontal offset for mark creation, relative to a kart wheel's local space. (Useful for when wheel mesh origins are not in the center of the tire.)

calculateNormals – If true, mesh normals will constantly be recalculated.

calculateTangents – If true, mesh tangents will constantly be recalculated.

defaultMarkMaterial – Default material for created marks.

markShadowCastMode – Shadow cast mode for a tire mark's renderer.

markLightProbeMode – Light probe mode for a tire mark's renderer.

markReflectionProbeMode – Reflection probe mode for a tire mark's renderer.

Kart Bump

This class enables karts to bump off of each other upon colliding together. Simply place it on any kart (on the root game object with the *Kart* component).

Public Variables:

bumpFactor – Multiplier for bump force magnitude based on collision velocity.

minBumpMagnitude – Minimum bump force after taking bumpFactor into consideration.

maxBumpMagnitude – Maximum bump force after taking bumpFactor into consideration.

Public Functions:

public void **Bump**(Vector3 **force**) – Applies the force vector affected by the above variables and projected onto the kart's ground normal plane to avoid flying up in the air.

Input Classes

Kart Input

This is the base class for sending input to karts. On its own, the class has no public variables or functions and simply sends protected input variables to whichever kart it's attached to. See derived classes *KartInputPlayer, BasicWaypointFollower*, and *BasicWaypointFollowerDrift* for examples of active input.

Kart Input Player

Subclass of *KartInput*. This class connects karts to Unity's Input Manager (by means of the *InputManager* class within the *PowerslideKartPhysics* namespace, which itself uses Unity's Input Manager). There are no public variables or functions; it simply fetches input from static variables in the *InputManager* class.

Kart Input Mobile

Subclass of *KartInput*. This class connects karts to mobile input (by means of the *InputManager* class within the *PowerslideKartPhysics* namespace, which itself contains functions and a struct for handling mobile input). There are no public variables or functions; it simply fetches input from the static *MobileInputStruct* in the *InputManager* class.

Kart Input System

Subclass of *KartInput*. This class exposes functions that can be hooked up to callbacks on a *Player Input* component to make karts respond to the new Input System package. See the Input System section above for configuration instructions.

Public Functions:

public void **OnAccel**(CallbackContext **context**) – Sets the acceleration input based on the float read from the given **context**.

public void **OnBrake**(CallbackContext **context**) – Sets the brake input based on the float read from the given **context**.

public void **OnSteer**(CallbackContext **context**) – Sets the steer input based on the float read from the given **context**.

public void **OnDrift**(CallbackContext **context**) – Sets the drift input based on the button state read from the given **context**.

public void **OnBoost**(CallbackContext **context**) – Sets the boost input based on the button state read from the given **context**.

public void **OnItemPress**(CallbackContext **context**) – Invokes the item pressed input based on the button press read from the given **context**.

Basic Waypoint Follower

Subclass of *KartInput*. This class allows a kart to follow a path of waypoints. If a kart gets stuck, it will attempt to reverse and drive forward again.

Public Variables:

targetPoint – The current waypoint to follow. When the kart gets close enough to be within the waypoint's radius, it will begin driving to the next waypoint.

steerAmount – Amount to steer to face the target point.

maxBrake – Maximum brake input to apply.

minAccel – Minimum acceleration input to apply.

reverseSpeedLimit – If the kart's speed drops below this value, the reverse timer will begin to increase, otherwise it will reset to zero.

reverseTimeThreshold – If the reverse timer reaches this value in seconds, the kart will begin to reverse (and invert its steer input).

reverseDuration – Length of time in seconds the kart will reverse before driving normally again.

Basic Waypoint Follower Drift

Subclass of *KartInput*. This class is similar to *BasicWaypointFollower*, but can be configured to make karts drift around turns.

***Public Variables** (Following those shared with BasicWaypointFollower):*

driftStartThreshold – When the "sharpness" of the waypoint path exceeds this value, the kart will begin drifting.

driftEndThreshold – When the "sharpness" of the waypoint path falls under this value, the kart will stop drifting.

driftSpeedMultiplier – Multiplier for “sharpness” value based on the kart's speed, making it so that the kart will be more likely to drift if it's driving faster.

driftSpeedMultiplierCap – Upper bound on speed-multiplied “sharpness.”

distanceAdvanceFactor – How much the kart interpolates toward the next point based on how close it is to the current one. Decreasing this will make the kart turn sooner/when it's further away and increasing it will make the kart turn later/when it's closer to the waypoint.

Basic Waypoint

This class represents waypoints for a kart to follow.

Public Variables:

nextPoint – The next waypoint to guide a kart to.

radius – The radius of a waypoint. If a kart comes within this range, it will begin targeting the next waypoint.

Input Manager

This class stores input in static variables for easy access, fetched either from Unity's legacy Input Manager or mobile input functions, but not the new Input System package. The input fetching is wrapped in a try/catch block because it's the only way to prevent errors resulting from invalid input axis names (this only applies in the editor). Mobile input is set through the use of public functions that can be linked to UI buttons or other sources.

Public Variables:

useStandardInput – If true, input will be fetched from Unity's Input Manager.

accelAxisName – Name of the acceleration input axis.

brakeAxisName – Name of the brake input axis.

steerAxisName – Name of the steer input axis.

driftButtonName – Name of the drift input button.

boostButtonName – Name of the boost input button.

itemButtonName – Name of the item input button.

cameraXAxisName – Name of the camera x-input axis.

cameraYAxisName – Name of the camera y-input axis.

lookBackButtonName – Name of the look back input button.

restartButtonName – Name of the restart input button. Leave empty to disable reloading the scene when the button is pressed.

useMobileInput – If true, input will be fetched from mobile input functions (to **MobileInput**).

mobileAutoAccel – Constant acceleration input sent to karts even when no input is actually being sent from the player. This allows for karts to drive automatically and is counteracted by brake input.

mobileSteerAccel – Acceleration input applied based on steer input. This allows karts to accelerate while steering (for example, with buttons for right/left steering the player can steer and accelerate without also pressing an accel button). This is counteracted by brake input.

mobileDriftAccel – Acceleration input applied when drift input is being sent to karts. This allows karts to accelerate while drifting without extra accel input and is counteracted by brake input.

mobileBoostAccel – Acceleration input applied when boost input is being sent to karts. This allows karts to accelerate while boosting without extra accel input and is counteracted by brake input.

mobileDriftHoldSteer – Factor for how much karts should automatically steer during drifts.

Whenever a kart turns with mobile input, the last steer value is stored to allow karts to automatically steer in that direction when drift input is being sent. This allows players to only hold drift input to drift and turn, because drifting requires steer input on top of the drift input.

Static Variables:

accelInput – Current acceleration input value.

brakeInput – Current brake input value.

steerInput – Current steer input value.

driftButton – Current drift input state.

boostButton – Current boost input state.

boostButtonDown – True only if the boost button was just pressed.

itemButtonDown – True only if the item button was just pressed.

camRotInput – Current camera rotation input (Vector2).

lookBackButton – Current look back input state.

MobileInput – Struct containing current mobile input values (see below).

Public Functions:

public void **RestartLevel**() – Reloads the active scene.

public void **SetAccelMobile**(float **accel**) – Sets the mobile accel input to **accel**.

public void **SetBrakeMobile**(float **brake**) – Sets the mobile brake input to **brake**.

public void **SetSteerMobile**(float **steer**) – Sets the mobile steer input to **steer**, between -1 (left) and 1 (right).

public void **SetDriftMobile**(bool **drift**) – Sets the mobile drift input to **drift**.

public void **SetBoostMobile**(bool **boostIn**) – Sets the mobile boost input to **boostIn**.

public void **PressItemMobile**() – Sets the mobile item input to true for one frame.

Mobile Input Struct (Struct)

This struct is used for wrapping mobile input values.

Public Variables:

accelInput – Current acceleration input.

brakeInput – Current brake input.

steerInput – Current steer input.

driftButton – Current drift button input.

boostButton – Current boost button input.

itemButtonDown – True for a single frame if the item button was just pressed.

Item Classes

Item

This is an abstract class used for setting up items that can be used by karts with the *ItemCaster* class. Derived classes should be thought of as instructions for how items are initialized, not actual spawned items (like a projectile).

Public Variables:

itemName – Name for identifying the item.

Public Functions:

public virtual void **Activate**(ItemCastProperies **props**) – Activates the item with the given properties **props**.

public virtual void **Deactivate**() – Deactivates the item.

Boost Item

This class represents an item that gives boost to a kart.

Public Variables:

boostAmount – Boost amount given to a kart.

boostForce – Force used to push a kart forward.

Public Functions:

public override void **Activate**(ItemCastProperies **props**) – Gives boost to the kart using it.

Projectile Item

This class spawns projectile items with the *SpawnedProjectileItem* class attached.

Public Variables:

itemPrefab – The item to spawn.

spawnOffset – Offset relative to the kart's local space for spawning the item.

Public Functions:

public override void **Activate**(ItemCastProperies **props**) – Spawns the item.

public override void **Deactivate**() – Destroys the last spawned item.

Spawned Projectile Item

This class represents the actual spawned projectile items that can hit karts.

Public Variables:

groundMask – Layer mask for objects that count as ground.

groundCheckDistance – Distance to check under item to see if on ground.

launchHeight – How high the item is aimed upwards/arced at launch.

startSpeed – Starting speed for the item.

targetSpeed – Speed that the item continuously moves at.

inheritKartSpeed – If true, the starting speed will be increased by the launching kart's speed.

maintainKartSpeed – If true, the target speed will be increased by the launching kart's speed.

accel – Acceleration rate of the item.

moveInAir – Whether the item can accelerate and change direction while airborne.

gravityAdd – Continuous gravity force applied to the item on top of the standard gravity.

gravityDir – Initial gravity direction for the item, normalized at runtime.

inheritKartGravity – If true, the starting gravity will match the launching kart's gravity.

gravityIsGroundNormal – If true, the item's gravity will match the surface it's moving on.

resetGravityDirInAir – If true, the gravity direction will reset to gravityDir while airborne.

forwardFriction – Forward friction of the item.

sideFriction – Sideways friction of the item.

maxFallSpeed – Maximum falling speed.

fallSpeedDecel – Multiplier for decelerating force to limit fall speed.

wallBounceReflect – Whether to reflect the movement direction upon colliding with a wall.

itemBounceReflect – Whether to reflect the movement direction upon colliding with another projectile item.

bounceReflectForce – Speed after bouncing will be multiplied by this amount.

maxBounces – Maximum bounces before the item will be destroyed.

destroyOnWallHit – If true, the item will be destroyed upon colliding with a wall.

destroyOnItemHit – If true, the item will be destroyed upon colliding with another projectile item.

wallDetectionType – Wall detection method.

wallDotLimit – Maximum dot product between the wall normal and relative up direction that counts as a wall collision. Greater values mean that flatter surfaces will count as walls. Only

used for normal-based wall detection.

wallMask – Layer mask for objects that count as walls. Only used for layer-based wall detection.

wallTag – Tag for objects that count as walls. Only used for tag-based wall detection.

casterIgnoreTime – How long in seconds the item should not collide with its casting kart.

canHitCaster – Whether the item can collide with its caster kart.

fetchKartsDuringSpawn – If true, the item will find references to every kart in the scene. If false, the item will use the supplied list of karts from the casting properties.

homingAccuracy – How quickly the item changes direction to follow a kart.

prioritizeKartsInFront – If true, homing items will prioritize karts in the direction the item is moving rather than whichever kart is closest.

minHomingAngle – Minimum dot product between the item's movement direction and the direction to a kart that will allow the homing item to lock onto the kart. Greater values decrease the acceptable lock-on range, lower values expand it.

maxHomingDist – Maximum distance to a kart that will allow the homing item to lock onto it.

useLineOfSight – Whether a linecast will be used to check for valid karts to follow. This can prevent homing items from locking onto karts through walls.

lineOfSightMask – Objects than can block line of sight with karts.

findTargetWhileActive – Whether to automatically find a target kart if there is no current kart being targeted.

kartSpin – Which axis a kart that is hit should spin about.

kartSpinCount – Number of times a hit kart should spin.

Public Events:

collideEvent() – Invoked when the item collides with a wall.

destroyEvent() – Invoked when the item is destroyed.

Public Functions:

public virtual void **Initialize**(ItemCastProperties **props**) – Initializes the item with the given properties **props**.

public virtual void **SetHomingTarget**(Kart **target**) – Sets the homing target to the given **target** kart.

public virtual void **FindHomingTarget**() – Automatically finds a new valid kart to target.

Item Caster

This class activates items to be used by karts.

Public Variables:

item – The item currently equipped.

ammo – Number of times the equipped item can be used.

minCastInterval – Minimum time in seconds between item casts.

Public Events:

castEvent() – Invoked when an item is cast.

Public Functions:

public void **Cast**() – Casts the currently equipped item.

public void **GiveItem**(Item **givenItem**) – Equips the **givenItem** with an ammo count of one.

public void **GiveItem**(Item **givenItem**, int **ammoCount**) – Equips the **givenItem** with the given **ammoCount**.

public void **GiveItem**(Item **givenItem**, int **ammoCount**, bool **bypass**) – Same as above, but if **bypass** is false, the item **givenItem** will only be equipped if the current ammo value is zero (meaning no usable item is equipped even if the current item is not null). Otherwise, the currently equipped item will be replace by the new one.

Item Giver

This class gives items to karts (specifically the *ItemCaster* class).

Public Variables:

itemName – The name of the item to give (either the item's itemName variable or the name of the object that the *Item* class is on). If empty, a random item will be given.

ammo – Ammo count of the given item.

cooldown – Duration in seconds during which the item giver is deactivated after giving an item.

Item Manager

This class manages references to usable items. The intended use is to have individual items on child objects of the manager object, one for each. This class automatically creates a list of items from these child objects.

Public Functions:

public Item **GetRandomItem**() – Returns a random item from the list.

public Item **GetItem**<Item>() – Returns a certain type of item from the list if it exists.

public Item **GetItem**(string **itemName**) – Returns an item from the list with the given name **itemName** if it exists. This can be either the item's itemName variable or the name of the child object that the item is on.

Item Cast Properties (Struct)

This struct is used for sending kart and item launch properties to items that are cast.

Public Variables:

castKart – The kart that is casting the item.

allKarts – All of the karts in the scene.

castKartVelocity – Velocity of the casting kart.

castPoint – Casting kart's position.

castRotation – Rotation of the casting kart (kart's rotator).

castDirection – Casting kart's forward direction.

castSpeed – Not used, but still present for potential future uses. (*SpawnedProjectileItem* uses it, but the variable is not assigned by the caster).

castGravity – Casting kart's gravity direction.

castCollider – Collider attached to the casting kart's rotator.

Environment/Level Classes

Ground Surface

This class connects ground surface types to colliders in order to send ground information to kart wheels.

Public Variables:

props – The linked *GroundSurfacePreset* containing ground properties.

Public Functions:

public GroundSurfacePreset **GetProps**() – Returns props unless it's null, then the function will return a default *GroundSurfaceInstance*.

public virtual float **GetFriction**() – Returns the friction of the surface.

public virtual float **GetFriction**(Vector3 **pos**) – Same as above, but **pos** is ignored. The Vector3 **pos** is for an override in the derived *TerrainSurface* class.

public virtual float **GetSpeed**() – Returns the speed factor of the ground surface.

public virtual float **GetSpeed**(Vector3 **pos**) – Same as above, but **pos** is ignored. The Vector3 **pos** is for an override in the derived *TerrainSurface* class.

Terrain Surface

This class is derived from *GroundSurface* and associates ground surface types with terrain textures.

Public Variables:

groundSurfaces – List of ground surface types for each terrain texture. The inspector should display a list of textures found on the attached terrain component; these correspond to ground surface types in the list.

Public Functions:

public void **UpdateAlphamaps**() – Updates the private 3D array from the terrain alphamaps used to determine dominant surface types.

public GroundSurfacePreset **GetDominantGroundSurfaceAtPoint**(Vector3 **pos**) – Returns the *GroundSurfacePreset* from the list of ground surfaces with the greatest alpha at the position **pos** (greatest alpha value in 3D array). This corresponds to the most visible texture at the given position.

public float **GetFriction**(GroundSurfacePreset **surfaceProps**) – Returns the friction of the surface **surfaceProps**.

public override float **GetFriction**(Vector3 **pos**) – Returns the friction of the dominant ground surface type at the position **pos**.

public float **GetSpeed**(GroundSurfacePreset **surfaceProps**) – Returns the speed factor of the surface **surfaceProps**.

public override float **GetSpeed**(Vector3 **pos**) – Returns the speed factor of the dominant ground surface type at the position **pos**.

Ground Surface Preset

This class contains properties for ground surface types.

Public Variables:

friction – The friction of the surface.

useColliderFriction – Whether to use the friction value of the attached collider's physic material instead.

speed – The speed multiplier for karts driving on the surface.

tireMarkMaterial – Material used for tire marks created on the surface.

alwaysSlide – If true, a kart wheel driving on this surface will always leave tire marks, even if it's not actually sliding.

tireSnd – Sound used for tire screeching on the surface.

Boost Pad

This class represents boost pads that can give karts boost upon overlap. The class must be used on a trigger.

Public Variables:

boostAmount – Amount of boost given to a kart.

boostForce – Force used to push a kart.

delayInterval – Time in seconds until a kart can use a boost pad again after using this one.

continuous – If true, any kart sitting on the boost pad will continuously boost rather than get a single burst after initially driving onto it.

Wall

This class has no function other than basically acting like a tag to represent walls for the component-based wall collision detection mode.

Wall Collision

This abstract class provides different methods for detecting wall collisions. Discrete implementations of the class are used for each method of detection. There is a unique class matching each value of the enumeration below.

Enumerations:

CollisionType – The type of wall collision detection.

> **Normal** – Walls are detected based on their normal direction compared to an up direction.
>
> **Layer** – Walls are detected based on being in a certain layer.
>
> **Tag** – Walls are detected based on having a certain tag.
>
> **Component** – Walls are detected based on having the *Wall* component attached.

Public Functions:

public static WallCollision **CreateFromType**(CollisionType **colType**) – Returns a new *WallCollision* instance based on the collision type **colType**.

public abstract bool **WallTest**(WallCollisionProps **props**) – Tests for a wall collision based on the given properties **props**. Each derived class overrides this for their wall detection method.

Wall Collision Props (Struct)

This struct is used for sending collision data to *WallCollision* instances.

Public Variables:

contact – Contact point of the collision (Unity *ContactPoint* struct).

upDir – Up direction used for normal comparison.

dotLimit – Maximum dot product for normal comparison.

mask – Layer mask used for collision detection.

tag – Tag used for collision detection.

Wall Detect Props (Struct)

This struct is used for serializing wall detection properties in the inspector.

Public Variables:

wallDetectionType – The type of wall collision detection to use.

wallDotLimit – Maximum valid dot product comparison between wall normal and up direction for wall collision.

wallMask – Layermask containing layers for wall objects.

wallTag – Tag for wall objects.

Hazard

This class is used to make objects cause karts to spin out upon collision or overlap.

Public Variables:

spinAxis – Axis about which a kart should spin out.

spinCount – Number of times a kart should spin about the axis.

Other Classes

Kart Camera

This class controls a camera following a kart.

Public Variables:

useLegacyInput – Whether to use the legacy input manager instead of the new Input System package. See the Input System section above.

targetKart – The kart to follow.

initialDist – Distance to follow the kart from.

initialHeight – Height to remain above the kart.

maxVelDist – Maximum distance to drag behind proportional to the kart's speed.

castMask – Layer mask for objects that the camera should collide with.

smoothRate – Smooth movement rate of the camera.

rollWithKart – Whether to rotate upside-down when the kart does.

rollSmoothRate – Smooth roll rate of the camera to match the kart's roll.

inputDeadZone – The camera will only rotate if the input magnitude is greater than this amount.

Public Functions:

public void **OnRotate**(CallbackContext **context**) – Sets the rotation input based on the Vector2 read from the given **context** (for connecting to a Player Input component).

public void **OnLookBack**(CallbackContext **context**) – Sets the look back input based on the button state read from the given **context** (for connecting to a Player Input component).

public void **OnRestartPress**(CallbackContext **context**) – Reloads the current scene when the button from the given **context** is pressed (for connecting to a Player Input component).

Conditional Particle

This class controls a looping particle system that automatically starts and stops based on a given condition. This class is not a Monobehaviour and must be created within another script.

Public Variables:

condition – Anonymous method representing the condition that will cause the particle to play whenever true. This must be set with scripting.

particles – The particle system to control.

Public Functions:

public void **Update**() – Updates the particle system state, should be called in standard Update() from owning Monobehaviour.

Conditional Particles

This class is essentially the same as *ConditionalParticle*, except it uses an array of particle systems in order to control multiple particle systems with one condition.

Conditional Sound

This class controls a looping audio source that automatically starts and stops based on a given condition. The volume and pitch of the sound will change to transition between on and off states depending on the set properties. This class is not a Monobehaviour and must be created within another script.

Public Variables:

condition – Anonymous method representing the condition that will cause the sound to play whenever true. This must be set with scripting.

snd – The audio source to control.

maxVolume – Maximum volume of the sound.

volumePower – Power used for scaling the private linear volume value.

onRate – Rate at which the volume increases after starting.

offRate – Rate at which the volume decreases while stopping.

onPitch – Pitch of the sound while on.

offPitch – Pitch the sound changes to while stopping.

onPitchRate – Rate at which the pitch changes after starting.

offPitchRate – Rate at which the pitch changes while stopping.

Public Functions:

public void **Update**() – Updates the audio source state, should be called in standard Update() from owning Monobehaviour.

F

This class contains extra static functions.

Static Functions:

public static float **MaxAbs**(params float[] **nums**) – Returns the number with the greatest absolute value.

public static Component **GetTopmostParentComponent**<Component>() – Returns the topmost parent of the calling object with a certain component.

public static bool **Is**<Component>() – Returns whether the calling object or any of its parent objects contain a certain component.

public static bool **IsKart**() – Returns whether the calling object or any of its parent objects contain a *Kart* component.

public static bool **IsSpawnedProjectileItem**() – Returns whether the calling object or any of its parent objects contain a *SpawnedProjectileItem* component.

public static bool **IsWall**() – Returns whether the calling object or any of its parent objects contain a *Wall* component.

public static bool **Contains**(int **layer**) – Extension method for Unity's *LayerMask*, returns true if the calling layer mask contains the given **layer**.

Gizmos Extra

This class contains extra gizmo drawing functions.

Static Functions:

public static void **DrawWireCylinder**(Vector3 **pos**, Vector3 **dir**, float **radius**, float **height**, int **steps**) – Draws a wire cylinder gizmo at the position **pos**, with the caps facing in the direction **dir**, and with the given **radius** and **height**. The **steps** parameter indicates how many line segments are drawn to represent each cap circle.

public static void **DrawWireCylinder**(Vector3 **pos**, Vector3 **dir**, float **radius**, float **height**) – Same as above but drawn with the default 20 line segments for each cap circle.

UI Control

This class controls the UI in the demo scene.

Public Variables:

targetKart – The kart to use for updating the HUD.

boostMeter – Meter displaying the kart's boost amount.

boostMeterFill – The fill image of the boost meter.

boostReadyColor – Color for the boost meter fill when the kart's boost is ready.

boostNotReadyColor – Color for the boost meter fill when the kart's boost is not ready.

boostReserveMeter – Meter displaying the kart's boost reserve amount.

boostReserveCap – Value of the boost reserve corresponding to a full meter. If the boost reserve limit on a kart is less than infinity, the limit value will override this.

airTimeMeter – Meter displaying the kart's air time.

airTimeCap – Time in seconds of air time at which the air time meter should be full. (The maximum air time that the meter can show).

itemText – Text showing the kart's current equipped item.

ammoText – Text showing the kart's current equipped item ammo.

Demo Menu

This class provides functionality for the menu buttons in the demo scene.

Public Variables:

spawnPoint – Position to spawn selected karts at.

spawnDir – Direction that spawned karts should face.

uiContainer – Object containing the HUD.

kartCam – The camera that will follow a spawned kart.

antiGravPreset – *KartGravityPreset* applied to spawned kart if anti-gravity mode is enabled.

Public Functions:

public void **SpawnKart**(GameObject **kart**) – Spawns the given kart prefab at the spawnPoint position facing the spawnDir direction. This function will also activate the gameplay UI container and initialize the kart camera while deactivating the menu object.

public void **SetAntiGrav**(bool **grav**) – Sets whether to use anti-gravity on the spawned kart based on grav. This is for use with the anti-gravity UI toggle.