Ratóncio

by Juan David Payán

✨Description

Ratóncio is a little 3D prototype where you control a hamster in its ball, the prototype is designed towards the use of physics to control movement, handle interactions and solve puzzles.

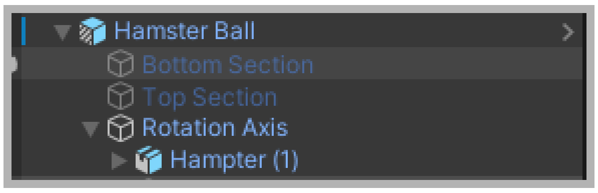
❓Q/A

- **What should we look for in your project?**: You should definitely take a glance at the Assets/Variables folder. There you will find scriptable objects that hold single values. Think of these values as global variables. This architecture will be explained better in its own section.
- **What are you most proud of?**:Even though the player was requested as a simple ball, I always loved to make the player an actual character. That is why I decided I could make a hamster on its little sphere. It didn't take me more than two hours to 3D model-it, rig it, texturize it and make the ball's shader. But I love the end result. More explanation later.
- **What could have been done better?**: The isGrounded functionality is currently using a little trigger at the bottom of the ball rather than a raycaster solution. This actually works perfectly based on the requirements of the prototype, but it could give issues if there were more complicated "trigger" interactions, due to OnTrigger events not being able to "ask" what collider was the one that entered the other one. I would also like it if I could make a more robust Scritable Variables system (More into that later) with the use of Generics. To quickly create new Scriptable Variable Types easier.
- **How much time was spent on each section/ task**: The main task (Everything about movement) took like 70% of the 12 hours this prototype took, just fine tuning it to simulate the hamster a "hollow sphere", and make it physically accurate. Try balancing the hamster on top of another sphere and watch the torque work as it should.

🐹THE HAMSTER

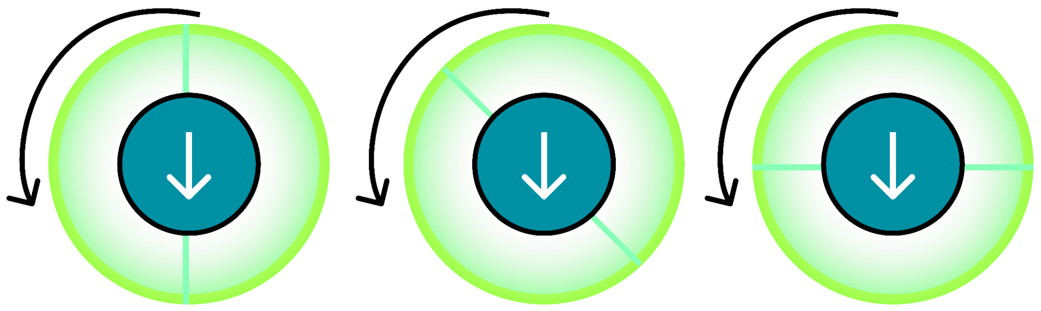
The main challenge in the looks and physics of the hamster and its ball, is the challenge of having hollow colliders and having a father-son relationship between two RigidBodies. At first I wanted to make the hamster an actual physical entity, and somehow recreate a hollow sphere collider combining multiple plane colliders just as there are faces on a very polygonal sphere. This was clearly nonsense for a prototype and such a simulation would take a lot of effort for not a lot of value. Instead the only physical object is the ball itself.

There are some neat tricks to simulate the physicality of the hamster itself though. The hamster, the camera and the Ratcopter are child of the ball transform, like this:

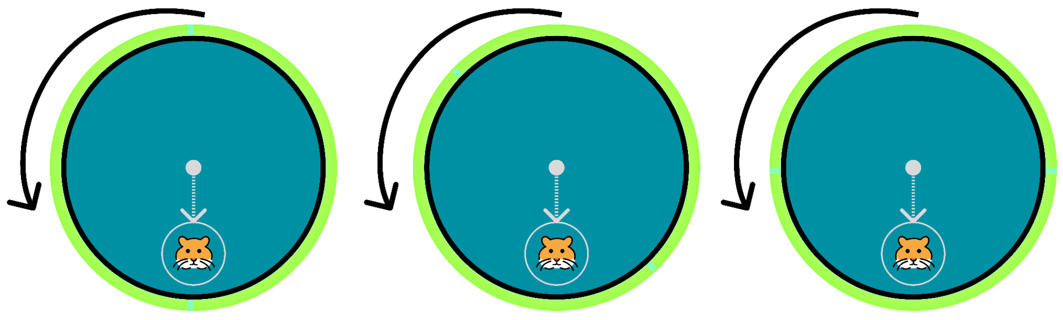


- **Hamster Ball:** The hamster ball is a non-mesh object that contains the movement script, it has a sphere collider and a RigidBody. Which makes it the sphere itself. This rigidbody has the scripts that control the ball movement and other types of forces like the RatCopter thrust or the forces of a LaunchPad. This also means the Hamster Ball Transform will be subject to a lot of rotation and forces and all its children will obviously attempt to rotate with it. That is the importance of...

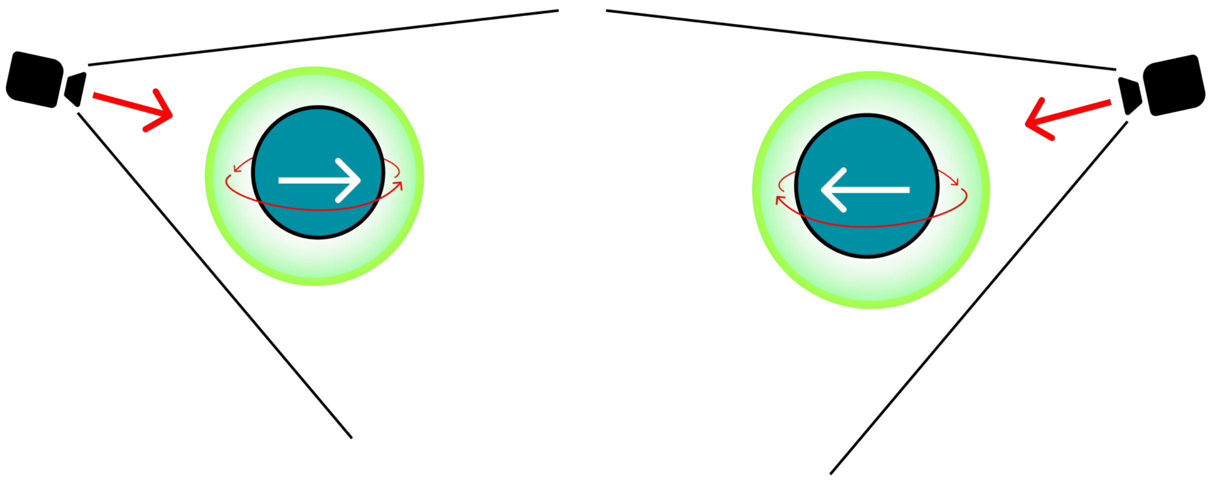
- **Rotation Axis:** Imagine the well of a car, even though it rotates around the axle itself, connected to the center of the wheel will remain static to that rotation. Such an axis in 3D could only be possible on the absolute center of the sphere. This GameObject has a local position of 0,0,0 right at the center. It also contains an Script Identity Locker, it locks the Transform rotation, ignoring completely the rotation of its parent, think about it as an anchor. That always points downwards.



Now, as mentioned, in order to an axle to keep its position unchanged it needs to be directly on the center of the ball, but, any element **INSIDE** the axle itself will not only preserve its rotation unchanged, but also its position without accounting how close or far the elements is from the axle center. That's why the Hamster is inside the axle, and is able to remain on the lower part of the ball without rotating even if the ball is doing crazy things.



Finally, to account for “moving the ball forward means go where the camera is looking at” , the Axle rotates around its Y Axis in a way that the Z of the Transform is always facing wherever the camera is pointing.



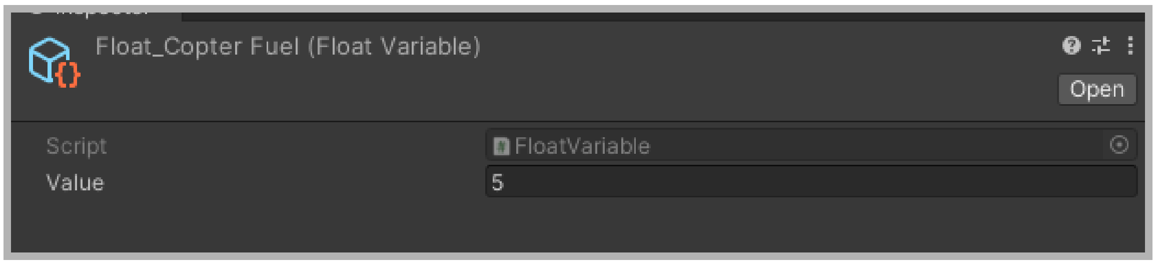
- **Hamster:** The hamster is just a rigged mesh, the player won’t do anything to control it directly, the hamster has a reference to the ball rigidbody velocity vector. With that information the hamster is able to adapt itself accounting for the speed of the ball, and will also change its Animator State (Idling, Moving, Falling).
- **Ratcopter:** The ratcopter will not handle any physics nor boost by itself. What the RatCopter does is manage all the Fuel system (Fuel level, Fuel Capacity and Refill rate). If all the conditions are met for the Ratcopter to start propelling. All it does is raise a flag (An scriptable variable), or lower it in case the conditions for flying are not met. The propulsion itself it’s handled by the ball itself which depending on the state of said flag will emit some force making it aerial.

📚SCRIPTABLE VARIABLES

The architecture of the prototype relies on a decouple system, all modules handle their own behavior and typically will not need a reference to the module of another object, usually they only handle their own components. To share information between different objects, we use Scriptable Object, or a system that I live to call

Scriptable Variables. This Scriptable Object only has a Value of a given type, this allows the ability to Store variable values on Assets that can be shared, read, and written on by decoupled components. You can use them declaratively as a way to store Game-Broad constants or particular values that get reused by many systems, and my favorite part as reactive variables that execute logic every time someone changes the value.

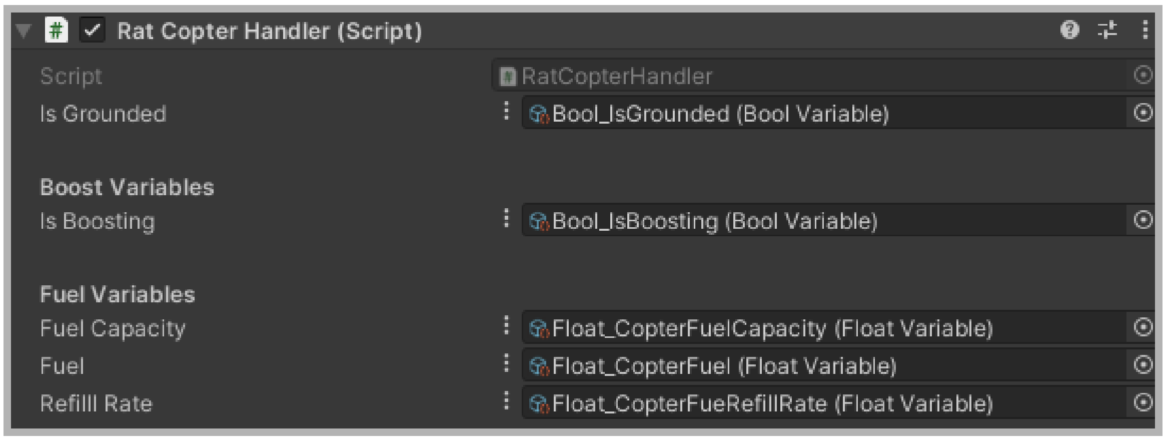
Take for example the Fuel Level, we have two independent objects that need to share this information. The RatCopter which will Write and Read that value, and the UI element that will show you in a progress bar the fuel amount. Instead of referencing objects between each other. We wire both components to the same scriptable object that holds that value.


Float_Copter Fuel (Float Variable)

Script	FloatVariable
Value	5

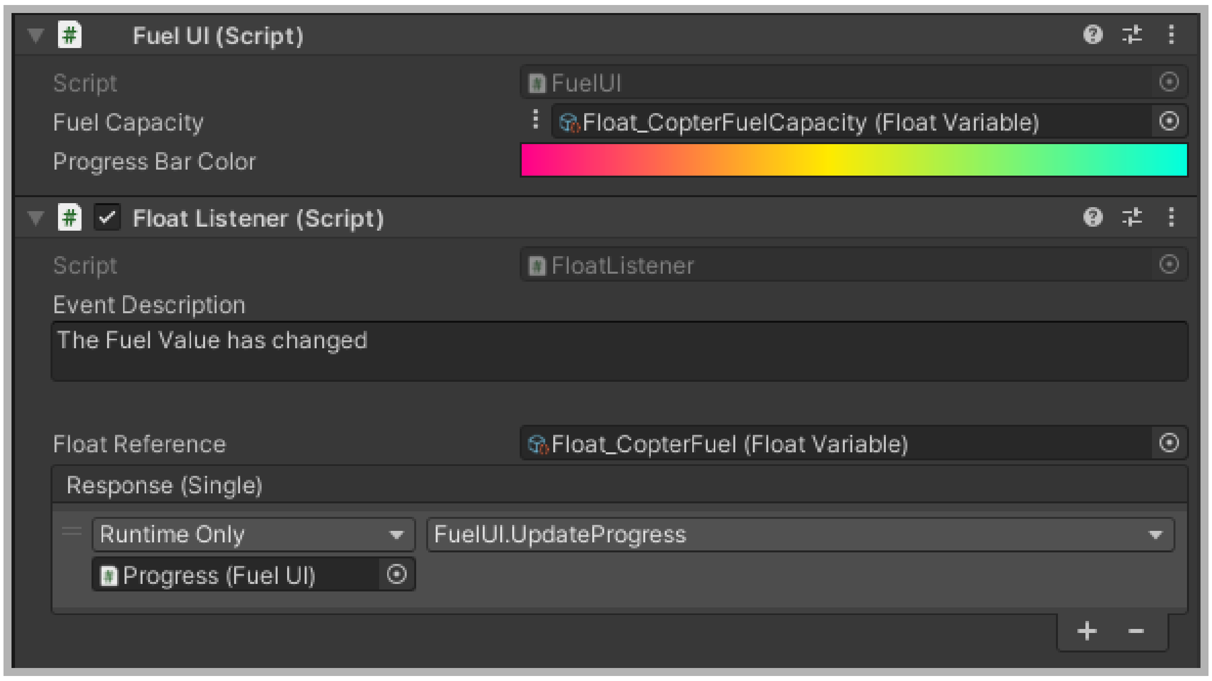
The **RatCopterHandler.cs** uses a lot of Scriptable Variables, Some are only read in the script itself, for example ***IsGrounded*** which is written by the ***BallMover.cs*** but RatCopter needs to read in order to handle refill fuel system, Some other variables like ***IsBoosting*** are written by RatCopter and Read by the Ball Mover.

Here we set a ***Fuel*** variable where the RatCopterHandler will write the fuel level.


Rat Copter Handler (Script)

Script	RatCopterHandler
Is Grounded	Bool_IsGrounded (Bool Variable)
Boost Variables	
Is Boosting	Bool_IsBoosting (Bool Variable)
Fuel Variables	
Fuel Capacity	Float_CopterFuelCapacity (Float Variable)
Fuel	Float_CopterFuel (Float Variable)
Refilll Rate	Float_CopterFueRefillRate (Float Variable)

And then the FuelLevel UI element will use a component called ***FloatListener,*** what this component allows is to invoke an Unity event every time the given Scriptable Variable has changed its value, passing that value as a dynamic argument to the functions that are expected to receive an argument of that type.



Fuel UI (Script)

Script: FuelUI
Fuel Capacity: Float_CopterFuelCapacity (Float Variable)
Progress Bar Color

Float Listener (Script)

Script: FloatListener
Event Description: The Fuel Value has changed

Float Reference: Float_CopterFuel (Float Variable)
Response (Single)
Runtime Only | FuelUI.UpdateProgress
Progress (Fuel UI)

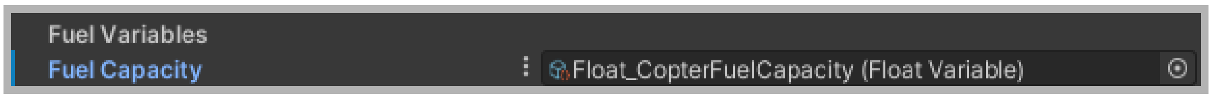
As you can see the FuelLevel UI element reads Scriptable Variables in two ways.

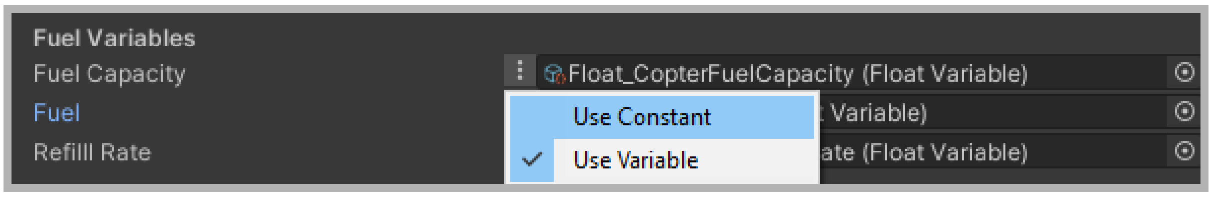
1. In the ***FuelUI.cs*** it has a direct reference to a Scriptable Variable, that tells the progress bar, which is the maximum level of fuel the tank can have, this only needs to be read once and thus why it is being directed assigned in the script.
2. The ***FloatListener.cs*** is listening to the Scriptable variable that holds the fuel level. Then the listener executes a function inside ***FuelUI.cs*** called ***UpdateProgress()*** everytime the value of the fuel level

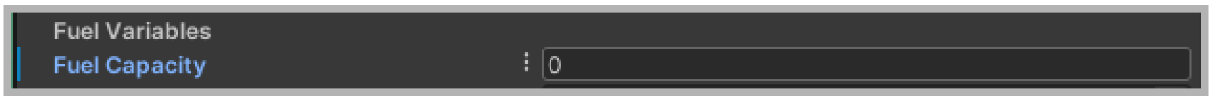
changes, and passes that value as an argument to said function.

```
public void UpdateProgress(float value)
{

    float normalizedValue = value / fuelCapacity.value;

    m_progressBar.fillAmount = normalizedValue;
    m_progressBar.color = progressBarColor.Evaluate(normalizedValue);
}//Closes UpdateProgress method
```

Scriptable variables are a potent and very designer friendly way to wire data in decoupled systems. If you see three dots in a Serialized Scriptable Reference variable in a script, you will notice that instead of using a Scriptable Variable to store for example, the maximum fuel capacity, Instead you can use constant value. This is implemented to allow quick prototyping without the need to create all the variable assets from the get go, but it also means that you'll need to reference the script itself in the conventional ways until you start using the reference as a variable.


Fuel Variables
Fuel Capacity ⋮ Float_CopterFuelCapacity (Float Variable)


Fuel Variables
Fuel Capacity ⋮ Float_CopterFuelCapacity (Float Variable)
Fuel
Refilll Rate


Fuel Variables
Fuel Capacity ⋮ 0

It also means that there won't be declarative ways to execute reactive unity events on values changed.

📚API

This is an overview on the main scripts on the prototype,

ScriptableVariables:
The scriptable variables repeat the same scripts but have different value types. All Variables created are in ***Assets/Variables***

- **Float**
  - **FloatVariable.cs**
  - **FloatReference.cs**
  - **FloatListener.cs**
  - **FloatReferenceDrawer.cs**
- **Bool**
  - **BoolVariable.cs**
  - **BoolReference.cs**
  - **BoolListener.cs**
  - **BoolReferenceDrawer.cs**
- **Vector3**
  - **Vector3Variable.cs**
  - **Vector3Reference.cs**
  - **Vector3Listener.cs**
  - **Vector3ReferenceDrawer.cs**

The ***{Type}Variable.cs***, is just an scriptable object which can be created as Right Click/Create/Variables. The scriptable object contains a single variable called ***Value*** of the given type.

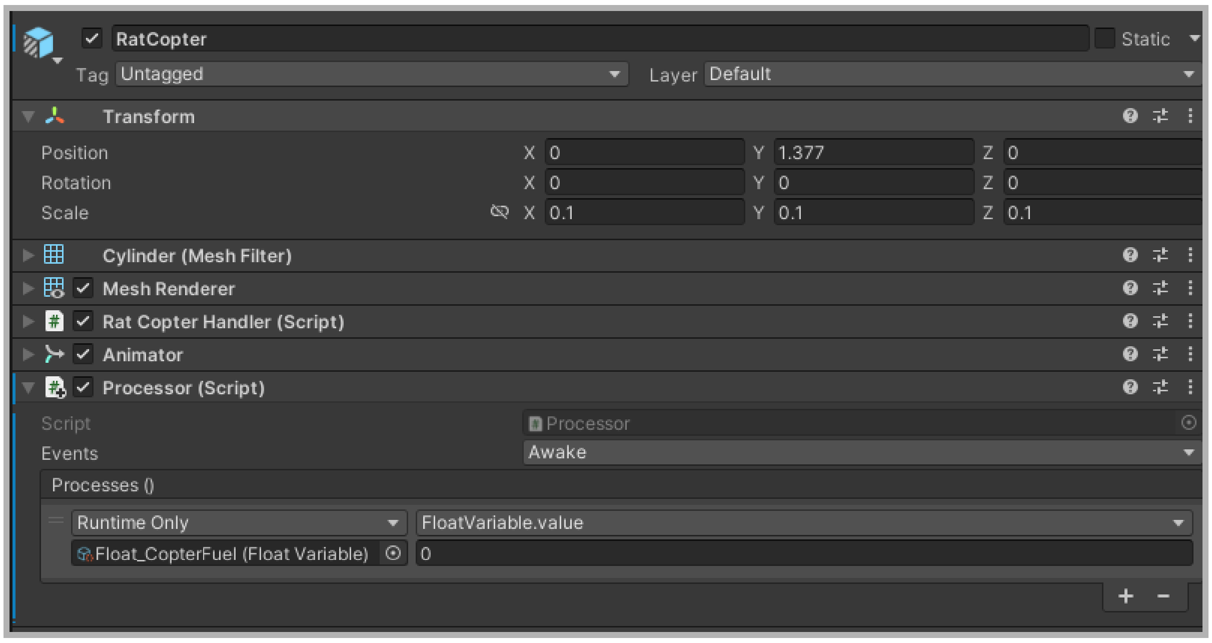
The ***{Type}Reference.cs***, is a class that has a property called ***Value*** of the given type. This property can reference the Value of a Scriptable variable or a value of a field, depending on a boolean called useStatic.

The ***{Type}Listener.cs***, is a MonoBehaviour component that will execute an unity event whenever the given Variable changes its value. The Value can be passed as a Dynamic Argument through the Unity event invocation if the method receives an argument of the same type.

The ***{Type}*ReferenceDrawer.cs**, is a custom editor drawer for the class ***{Type}*Reference,** *It conditionally renders fields based on the value of the UseConstant variable, rendering it like a selection input with two options. <UseVariable, UseConstant>*

Processor.cs:
The processor is an utility helper component that executes an UnityEvent whenever the Gameobject the script is attached to executes one of its most used Monobehaviour lifecycle methods.

For example, the RatCopter Gameobject has a Processor Module that will set the Float Variable that stores the fuel level to 0 when its Awake happens.


RatCopter
Static
Tag Untagged
Layer Default
Transform
Position X 0 Y 1.377 Z 0
Rotation X 0 Y 0 Z 0
Scale X 0.1 Y 0.1 Z 0.1
Cylinder (Mesh Filter)
Mesh Renderer
Rat Copter Handler (Script)
Animator
Processor (Script)
Script Processor
Events Awake
Processes ()
Runtime Only
FloatVariable.value
Float_CopterFuel (Float Variable)
0

IdentityLocker.cs:
The IdentityLocker is an utility helper component that forces the rotation of the Gameobject that has the component to Quaternion.Identity. It has only a boolean that activates or deactivates the Identity locker.

MouseRotation.cs:
The MouseRotation is an utility helper component that rotates a Gameobject around its Y axis following the Mouse Horizontal travel. It has a Float Reference that sets how much the object will rotate.

BallMover.cs:
This script handles all the HamsterBall physics. It has a bunch of Scriptable Variables, some are just read, for example BallSpeed, BoostForce, IsBoosting, JumpForce, others are written like BallVelocitym IsGrounded. If something about the ball movement like speed or jump force needs to be modified, it would be suggested to change the Scriptable variable that holds that information.

HamsterHandler.cs:
This script handles the Hamster inside the ball. It receives two Scriptable Reference which the script reads in order to understand the current state of the ball and animate the hamster to simulate that the hamster is the one affecting the ball direction and speed.

RatCopterHandler.cs:
This script handles the conditions of the ball and some Input events to detect if it should Raise or LowerDown the ***IsBoosting*** flag. This script handles all the Fuel system.