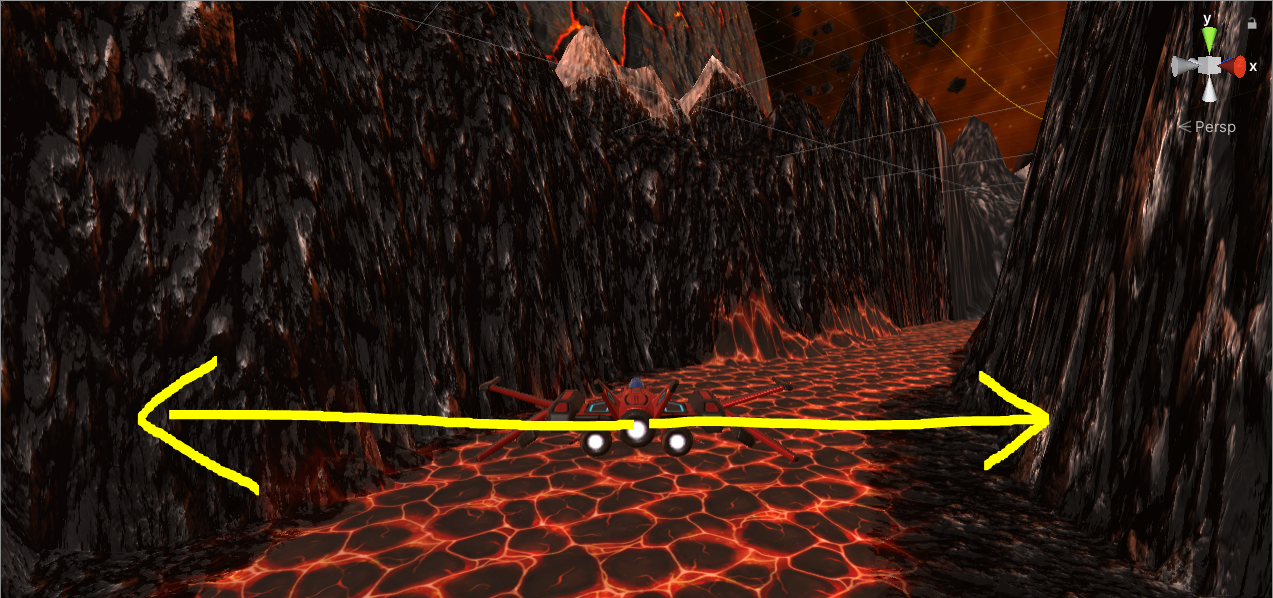
**Input Sensitivity and Gravity**

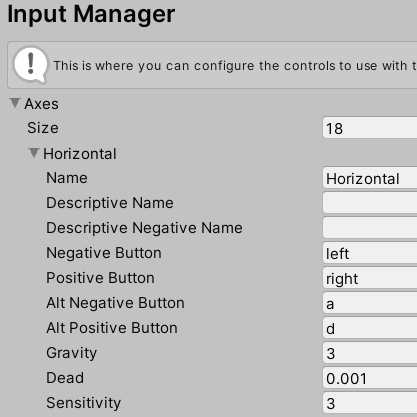
**Objective**: Students will lay the foundation for controlling the position of the ship on the screen by calculating an x offset that we need to apply to the ship every frame.

**Background:** We want the Spaceship to move smoothly throughout the level. If you remember in our previous game we used the current frame rate to determine how much the Rocket Ship should move.

The goal for this game is to get the Rocket Ship to move left and right



So how are we going to do that? If we get the fundamentals of this right now then it will serve you throughout your career.



**Sensitivity** determines how fast the throw changes when pressing A/D. **Gravity** determines the rate at which the throw goes back to neutral or zero.

The next phase is, how does that match the speed? Well, if we multiply our throw by some speed and then correct the frame rate, remember, time dot delta time in the last section when we dealt with rotation and thrust.

Then we can arrange that as our keys go from get pressed, or the stick goes from left to right, that the speed of the ship's movement left and right on the screen will also do something like this.



1. Change **horizontalThrow** to **xThrow**
2. Our designer might want to be in charge of the speed so lets create a **SerializedField** **floating** point variable called **xSpeed** and set it to **4**.

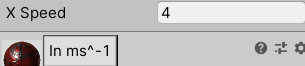


Now, why meters per second? They're the default units in Unity, and actually, I think that it's very important to think about the units. Now, you could put it in the variable **xSpeed in meters a second**, but very cumbersome. So why don't we do this instead.

1. There's a new attribute here for you in square brackets called **Tooltip**. And what we do is, we give it some brackets, and then we say, in meters per second.



1. Lets go back to **Unity**. Click on the **Spaceship** and in the **Inspector** window notice when you hover over xSpeed you get a hint at the units.



1. Under xThrow create a new floating point variable called **xOffsetThisFrame**and have it be equal to the **xThrow** times the **xSpeed**.



And because we understand that everything is happening inside the frame, because this is code that is executing on updates. So it's happening 70, 80, 100, 200 times a second. So we can only calculate an offset for this particular frame, which is the throw times the speed.

1. so lets multiply times the change in time since the last frame to have an effective offset.



1. print the **xOffsetThisFrame** variable right underneath and lets head to unity to see the console log.
2. Now for my setup I was login 0.01 meters or 1 centimeter every frame and around 1430 FPS.

If I multiply that I’m getting about 14.3 meters per second.

What we're trying to do here is a common sense check, a ball park check. Is this really the xSpeed in meters per second? Well, if we apply this particular offset every frame of a few centimeters a second at my current frame rate, then yes, it will be the xSpeed.

**Confused??**

If you are doing animations for work, imagine your game as an animation. What you see in the game screen is nothing but a picture. If you have a frame rate of 60, the picture gets rendered 60 times per second. To get the illusion of a smooth movement, you have to calculate the new position of the object. If you ever drew an animated stickman, you did that intuitively.

The problem in Unity (and other highly optimised game engines): The duration of a frame is not consistent, so you cannot say: move 1 WU per frame. If you had to draw everything, you would create 1000s of pictures for a simple rotation, so you can select the correct picture depending on the elapsed time.

Time.deltaTime is nothing but the elapsed time since the previous frame.