

# Sphere LithoniteTutorial

Abstract: Usage of lithonite vectors. Extending Object Oriented Javascript. DeObstruct() explained.  
Level: Intermediate users.(basic knowledge of javascript required)

## What is Lithonite?

Lithonite is an object with variables and functions that facilitates manipulation of your PC (Playable Character). Its very versatile, you can deobstruct, Push NPC's, Run, slide and create various other effects by extending it.

## Initializing Lithonite

First, we include the external lithonite.js script into our own and create an instance (a variable) of it, like this:

```
RequireScript("lithonite.js");  
var Lithonite= new LithoniteEngine("Lithonite");
```



Note that the name of the variable **Lithonite** must be given as a string to the Object **LithoniteEngine**.

We also need to give it the name of our PC, so if we have this:

```
var MainChar= "Celes";  
CreatePerson(MainChar, "Celes.rss", false);
```

You should initialize it like this (in the constructor):

```
var Lithonite= new LithoniteEngine("Lithonite", MainChar);
```

Or like this (through a method/function):

```
Lithonite.setInputPerson( MainChar);
```

Or even like this (references the variable directly, be careful!):

```
Lithonite.GIP= MainChar;
```

When MapEngine() is running, you can omit the parameter in setInputPerson(), it will then use GetInputPerson() to find the current PC. The speed penalty for calling setInputPerson() inside the update or render script is not that big and it is very save. Its handy; especially if you change from PC during the game.

## Lithonite Vectors

To have the latest info on what the PC is doing, these are calculated each time the screen is rendered. They have a value of -1, 0 or 1. Lithonite Vectors depend on the cursor keys the player is pressing.

<i>VectorName</i>	<i>Function</i>	<i>-1</i>	<i>0</i>	<i>1</i>
move_x	Current X direction of the PC	left	-	right
move_y	Current Y direction of the PC	up	-	down
hist_x	Previous X direction of the PC	left	-	right
hist_y	Previous Y direction of the PC	up	-	down

You calculate them like this:

```
Lithonite.calcVectors();
```

That's it! Now that we have the current and latest movement information. You can now do something with them, like deobstruct your PC if its colliding with something:

```
Lithonite.deObstruct();
```

So lets first take a simple example on how to use the vectors. To know if your PC just has stopped with walking, the move\_x and move\_y are 0, while the hist\_x and hist\_y are not. So, you could extend your variable Lithonite with this function:

```
Lithonite.hasJustStoppedWalking = function()
{
    return (    (this.move_x==0) &&
                (this.move_y==0) &&
                (
                    (this.hist_x!=0)||(this.hist_y!=0)
                )
            )
}
```

Note that to access the internal variables inside Lithonite you always need to use “this.” (well, not always, but its good practices. In the above example you could also have used Lithonite.move\_x) The problem with this code is that you are modifying your variable **Lithonite**, not the actual Object **LithoniteEngine**. So lets modify the Object:

```
LithoniteEngine.prototype.hasJustStoppedWalking = function()
{
    return (    (this.move_x==0) &&
                (this.move_y==0) &&
                (
                    (this.hist_x!=0)||(this.hist_y!=0)
                )
            )
}
```

Note that if the function already existed, it will be overwritten with your new version. (In the same way that `Lithonite.move_x=5;` will set `move_x` to 5)

Now, for this code to be included in your variable, it must be written before you initialize your variable **Lithonite** but after the `LithoniteEngine` Object is declared.

We do it like this:

```
RequireScript("lithonite.js");
RequireScript("MyChangesToLithonite.js");
var Lithonite= new LithoniteEngine("Lithonite");
```

And put all the changes in the **MyChangesToLithonite.js** javascript.

And if you want to define a new variable inside the Object `LithoniteEngine`, do it like this:

```
LithoniteEngine.prototype.myVar = "HelloWorld";
```

That's all the knowledge you need to understand the basics of Sphere ChiBi Tutorial. The rest (like making it snow) is just tricks and physics.

### ***DeObstruct(), how does it work?***

First we calculate the predicted position of our PC. We have `move_x` and `move_y`, which are the vectors of our current movement, so our predicted position in the next `UpdateMapEngine()` iteration is at `($GPXX,$GPYY)`, the vectors added to our current position, calculated as follows:

```
var $GPX = GetPersonX(this.GIP);
var $GPY = GetPersonY(this.GIP);
var $GPXX= $GPX+this.move_x;
var $GPYY= $GPY+this.move_y;
```

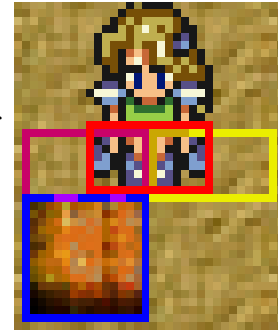
So, if `(IsPersonObstructed(this.GIP,$GPXX,$GPYY))` then we act now to deobstruct the PC.

But wait! If we are pressing 2 cursor keys and we can go into one of the 2 directions (say, facing a wall and pressing to the wall and sideways) we let Sphere take care of the movement, as it will slide the PC along the wall. In the example we press right and down. The sprite cannot go down as it collides with the blue obstruction, but it can go right (yellow arrow).

In this case, we do nothing. If it's not the case, then its time to deobstruct!



In this example we are pressing down. So first, we start sweeping along the path searching for a place where it is not obstructed. 8 pixels to the left (purple obstruction) we still are obstructed by the blue obstruction. If we go to the left our first not obstructed position is 24 pixels away. If we go right, our first not obstructed position is just 8 pixels away (yellow obstruction) and keeps being not obstructed further to the right. So our best option is to go.... right!



DeObstruct() isn't that intelligent. It will look 13 pixels to left and right and the victor is the side that has more non-obstructed tests. (That's what **\$domovePOS** and **\$domoveNEG** are, they just handle all the possible directions)

```
for ($i=0,$i<=13;++$i)
{
    $dHI=this.move_y*$i, $dVI=this.move_x*$i;
    $domovePOS+=!IsPersonObstructed(this.GIP,$GPXX+$dHI,$GPYY+$dVI);
    $domoveNEG+=!IsPersonObstructed(this.GIP,$GPXX-$dHI,$GPYY-$dVI);
}
```

Then if

```
$domovePOS && ($domovePOS >= $domoveNEG)
```

Why do we check \$domovePOS? Because if we are totally obstructed and cant really go anywhere (\$domovePOS is 0 and \$domoveNEG is 0, and 0 >= 0 so if we bump straight into a wall, we would be going left or right, while what we really want is stand in place. But if its not zero and \$domove\$POS >= \$domoveNEG, then we

```
SetPersonX(this.GIP,$GPX+this.move_y);
SetPersonY(this.GIP,$GPY+this.move_x);
```

Pinky... are you pondering what I'm pondering?

Why do we set the X position using move\_y?

It has to do with variables used long ago: move\_h (move horizontally) and move\_v (move vertically), which casually were:

```
move_h=move_y;
move_v=move_x;
```

so I replaced them.

if movement in the X axis is obstructed, then maybe we can go vertically (move\_y!) and vice versa. (Don't ask, it just works)

**End!**