Guide to the Orx Console and Commands

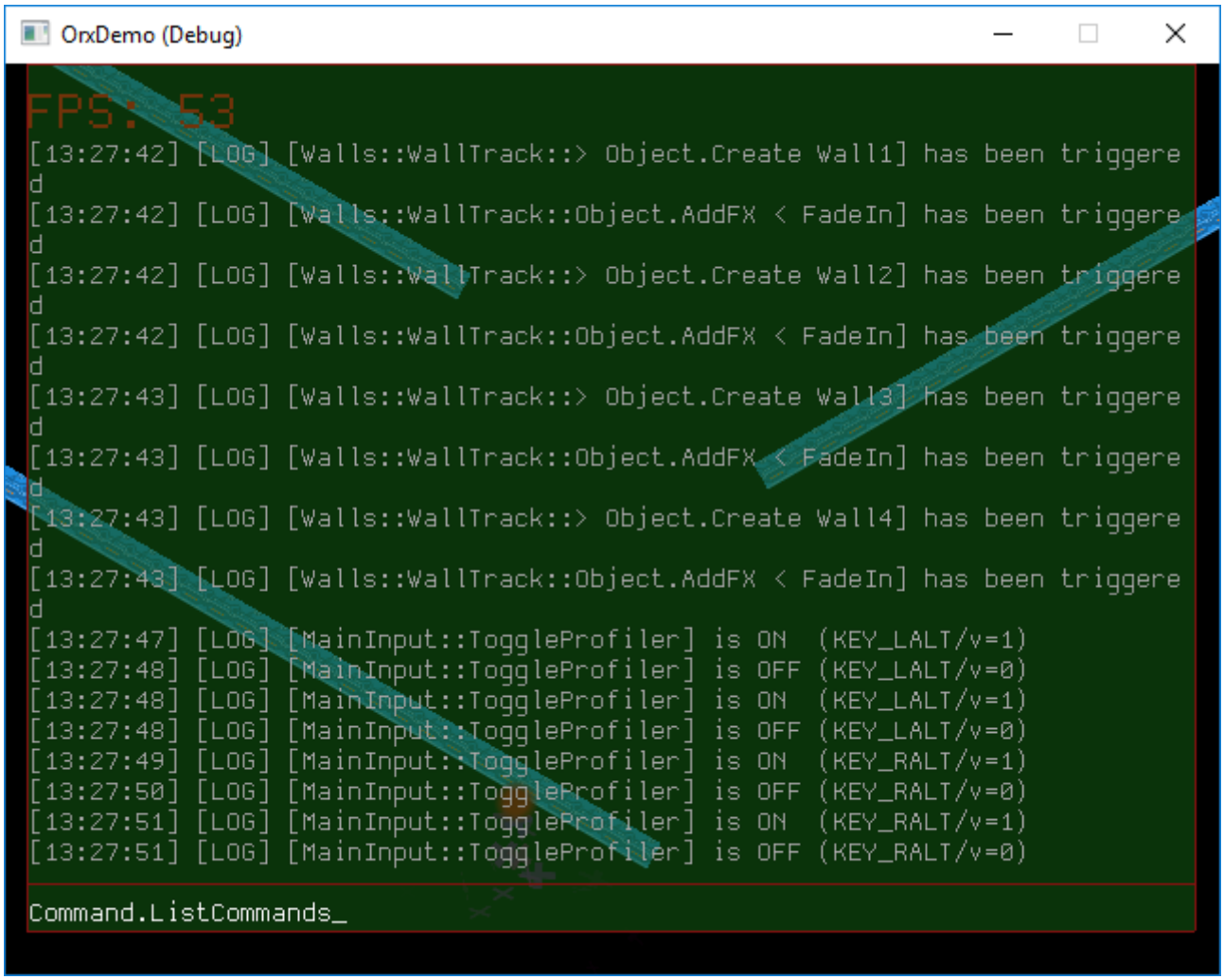
Beginners and seasoned Orx developers may not have tapped into the power that the Orx Console gives you. Or may not even have realised it exists. This tool can give you access to your game objects and config values during live execution in realtime.

It will let you alter any of these elements and values. You can experiment with commands which can then be used in Tracks within your game.

All Orx applications and games have an Orx Console compiled directly in.

This guide will take you on a tour of some of the power of the Console to help you get comfortable using it. Hopefully after working through this guide you will see many possibilities that are available, and how they can be applied.

The Orx Console will save you many hours of fiddly experimentation.

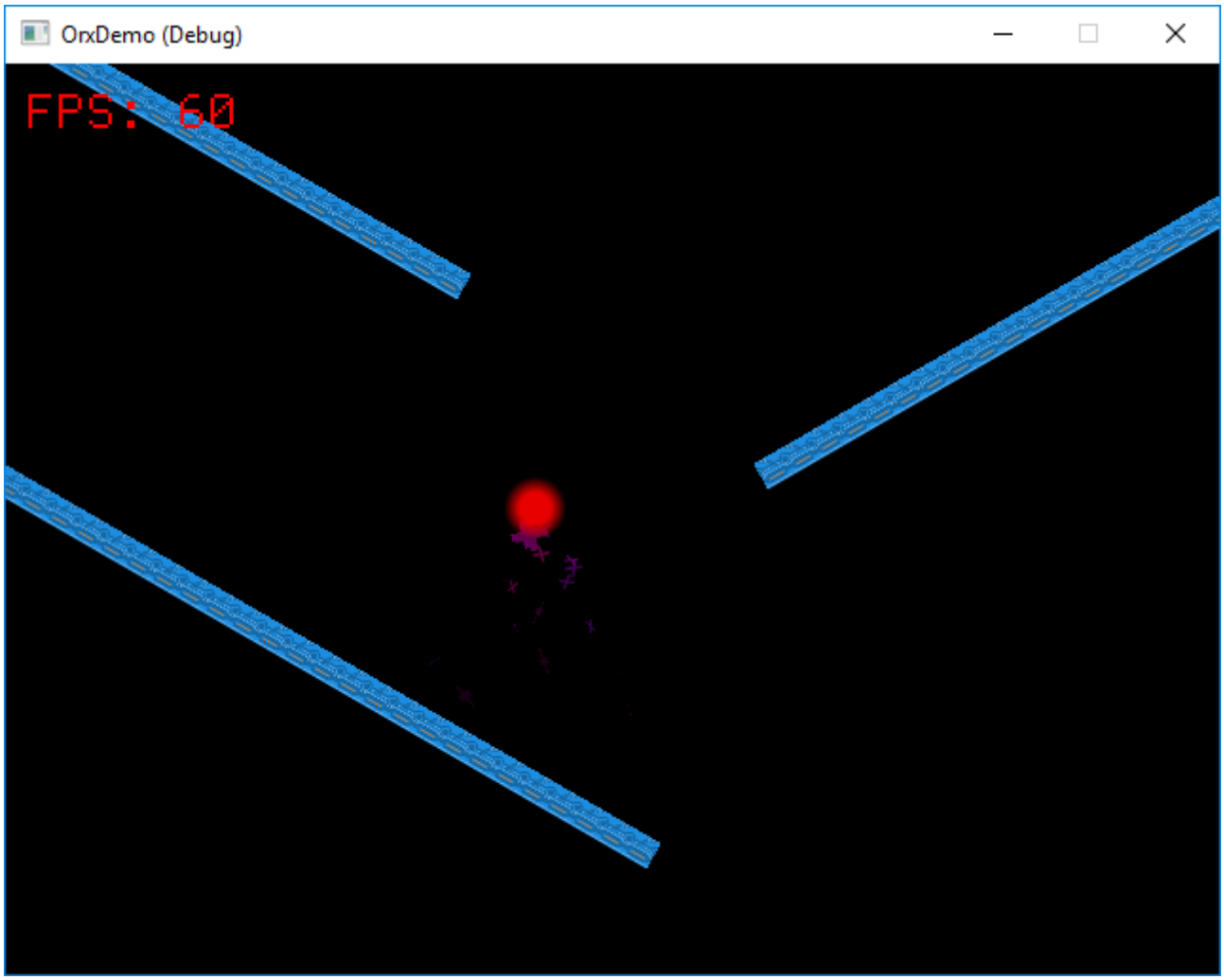


Starting with a Common Project

It will be helpful to use a common project that is available to everyone so that you can work with commands and configuration demonstrated in this guide.

If you are using the git repo version of Orx, you can use the Bounce Demo. Start it by executing: `code/bin/orxd.exe`

The Bounce Demo is not available in the downloadable release versions of Orx. Otherwise just use a project of your own.

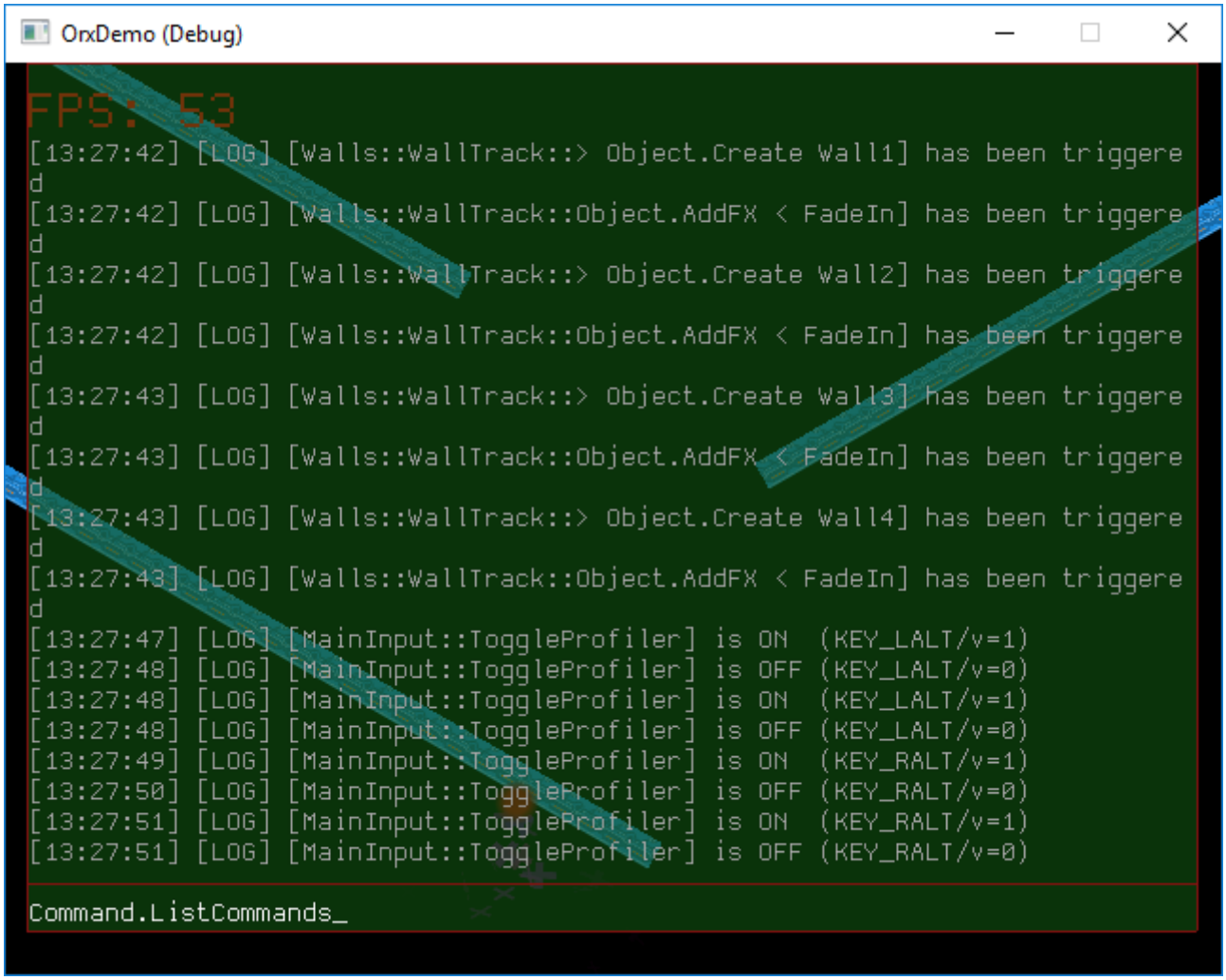


Starting the Orx Console

When running your game, to start the Console, press the `backquote` key:

`

The backquote key is below the ESC key and above the Tab key. Press backquote again to close the console.



Finding Commands and Navigating

There are over 170 commands available in the console. You can list them all with:

`Command.ListCommands` (then press enter)

All commands will be printed to the screen. The list can be scrolled with the mouse wheel or the PageUp and PageDown keys.

Partial commands can be typed and then use the Tab key to auto complete. For example, if we wanted to get the Orx version, the command would be:

`Command.Version`

If you type: `Command.V` <tab>

the Console will autocomplete to `Command.Version` for you, as this is the only command starting with V.

The tab key also cycles between multiple possible commands listed. For example, if you wanted the command:

`Command.ListAliases`

You could type: `Command.Lis` <tab>

Both `Command.ListAliases` and `Command.ListCommands` would be listed.

Press `Tab` to cycle between both commands.

Press `Space` to select the command.

Command history can be accessed by pressing the `Up Arrow` or `Down Arrow` keys.

Different executables (debug/release) will have different histories. So when you key up or down through histories, these will be tied to a different project configurations/executables.

Command histories are saved in the same folder as your executable, with a `.cih` extension. If your exe is called: mygame.exe then you'll have a mygame.cih created/updated whenever you quit your game. This is just a text file. Very handy for getting a copy of any command that you have used in the console. We'll cover some uses later.

The `Insert` key switches between Overwrite and Insert modes.

Getting Help on a Command

You can get help on any command if you are unsure of how to use it. In the next section we will make use of the `Config.GetValue` command. If you don't know how to use it, you can get help on it with:

`Command.Help Config.GetValue`

```
: {orxSTRING Value} Config.GetValue (orxSTRING Section) (orxSTRING Key)
[orxS32 Index = -1] [orxBOOL Verbatim = false]
```

The info in between the `{}` is the return value. Then the command itself. Values in between `()` are required values. Values in between `[]` are optional values.

Commands utilize the following types:

- `orxBOOL`: Must be either 'true' or 'false'.
- `orxFLOAT`: Any floating point value.
- `orxS32`: signed 32-bit value
- `orxS64`: signed 64-bit value
- `orxU32`: unsigned 32-bit value
- `orxU64`: unsigned 64-bit value
- `orxNUMERIC`: Any numerical type, ie. a string that can be interpreted as a vector, an int (decimal, hexa, octal, binary) or a float
- `orxSTRING`: Any String value.
- `orxVECTOR`: a 3-component vector (mathematics)

Copy/Paste Commands

Rather than typing all your commands, you have the choice to copy commands from this tutorial directly into the console using the clipboard. Select the following command and copy it into your clipboard:

`Command.Version`

Now paste with Control-V into the command bar of the Orx Console. Your command will be pasted in, and you can press enter to execute it.

Basic Commands

Getting a value from the config

In the `BounceAlt.ini` file, the following section is defined there:

```
[Ball]
LifeTime = 10.0
```

We can retrieve this value with:

`Config.GetValue Ball LifeTime`

The output is:

```
: 10.0
```

Setting a value in the config

We can change this value in realtime to affect the game:

`Config.SetValue Ball LifeTime 1.5`

The output is:

```
: 1.5
```

Press the ` key to close the console and press the left mouse button to spawn some objects. You'll notice each object now only lives for 1.5 seconds rather that 10. This is an instant change within the game.

In this way, you can try out all sorts of parameter changes without stopping and starting your game. These changes remain in memory. They are not changed in your original config files.

There is a way to save your changes out to a single file, which we will cover later.

More Getting and Setting: turning things on

Two very handy settings available in Orx are turning on physics debugging and showing the FPS. These settings can be changed from the Console

If you wanted to check if the ShowFPS property is on or off you can use:

`Config.GetValue Render ShowFPS`

```
Output: (true or false)
```

If you wanted to turn it off:

```
Config.SetValue Render ShowFPS false
```

or

```
Config.SetValue Render ShowFPS 0
```

In the same way you could turn on Physics ShowDebug with:

```
Config.SetValue Physics ShowDebug 1
```

Output

```
: 1
```

Or the profiler with:

```
Config.SetValue Render ShowProfiler 1
```

These are good examples, and you can also do exactly the same thing with all properties in your configuration files.

So far you might be thinking that the commands are little long. Soon we will address this by introducing aliases for commonly used commands. For now we'll stick to common longhand commands.

Getting and Setting Object properties

Retrieving values and setting values on Object sections is no different to the Physics and Render sections demonstrated above. Let's try some.

Getting a texture used by an object

```
Config.GetValue ParticleGraphic Texture
```

```
: x.png
```

This is the graphic used by the spawner controlled by the mouse. Let's change it:

```
Config.SetValue ParticleGraphic Texture ball.png
```

```
: ball.png
```

Notice that the objects being spawned are now balls.

Random values and Lists

`BounceAlt.ini` contains the following section:

```
[Fade]
SlotList = AlphaFadeIn#Scale#Wobble#Vanish
```

This command will get one value from the list at random:

```
Config.GetValue Fade SlotList
```

```
: Wobble (or AlphaFadeIn or Scale or Vanish)
```

You can retrieve a specific item in the list via an index value:

```
Config.GetValue Fade SlotList 1
```

```
: Scale
```

The first value starts at 0.

Objects and IDs

Not only can you find out information from the config files, you can also find objects running live in your game and affect them.

You can get an object using:

```
Object.FindNext Wall1
```

```
: 0x00000007000000EF
```

The return ID is your object.

We can create a second Wall1 object with:

```
Object.Create Wall1
```

```
: 0x0000000A000F12AF
```

You may not notice any change, and that's because the two Wall1's are on top of each other. We can change the position of the new Wall1 object with:

```
Object.SetPosition <press tab> (100, 100, 0)
```

```
: 0x0000000A000F12AF
```

When you press tab, it inserts the result of the last command in place. In this case, the object ID.

Now that there are two Wall1 objects, we can cycle between.

Start again with:

`Object.FindNext Wall1`

```
: 0x00000007000000EF
```

Then find the second one from the first with:

`Object.FindNext Wall1` <press tab>

```
: 0x0000000A000F12AF
```

Finally the last object can be deleted from the game with:

`Object.Delete` <press tab>

Objects and the stack

Not only can we get object IDs, and affect objects using their ID, objects can also be pushed onto the stack.

Values can be pushed onto the stack as well. They can then be popped and used with other commands.

`> Object.FindNext Wall1`

Notice the > symbol at the beginning of the command. This means: take the result of the command and push it onto the stack.

In our example above, the result of `Object.FindNext` is an object ID. This ID will be pushed onto the stack.

We can now use this value with our next command, changing the object position:

`Object.SetPosition < (100,0,0)`

The < symbol above means to pop the last saved value from the stack and use it within the command. This simply means: set the position of the object ID popped from the stack to (100, 100, 0).

The result of any command can be pushed onto the stack.

Creating an object with in-memory config

You can create any type of section or object in config without needing it to exist first.

`Config.SetValue MyNewBall Graphic @`

This means to create a config section + properties in memory like:

```
[MyNewBall]
```

```
Graphic = @
```

If you are not familiar with the @ syntax, this is a merging of another section. Specifically: MyNewBall is actually a regular object. The Graphic property would normally be set to a separate Graphic section. But rather than do that, @ lets us specify properties normally in a Graphic section, in the Object section instead. Very handy.

Next, we can set the Texture property:

```
Config.SetValue MyNewBall Texture ball.png
```

The following command will set the Scale property on the `MyNewBall` section:

```
Config.SetValue Scale 4
```

The config in memory will now look like this:

```
[MyNewBall]
Graphic = @
Texture = ball.png
Scale   = 4
```

Finally, you can create this object with the following command:

```
Object.Create MyNewBall
```

You should see a large ball in the middle of the screen.

Calculations

You can also perform math on values in the stack.

Addition on a value in the stack

In the `Bounce.ini` configuration file is the following section and property:

```
[Bounce]
BallLimit = 300
```

```
> Config.GetValue Bounce BallLimit
```

```
: 300
```

```
> + < 2
```

```
: 302
```

The above means: store the result of this calculation onto the stack. Use addition with the + symbol. Pop the last value from the stack which was the 300. Add 2 to this. 302 is stored in the stack.

Division on a value in the stack

```
> Config.GetValue Bounce BallLimit
```

```
: 300
```

```
> / < 2
```

```
: 150
```

In the same way as the addition example: store the result of this calculation on the stack. Use division with the / symbol. Pop the last value from the stack which was the 300. Divide this by 2. 150 is stored in the stack.

Subtraction on a position vector

Not just floats and integers. You can use maths on vectors too.

```
> Object.FindNext Wall1
```

```
: 0x00000007000000EF
```

```
> Object.GetPosition <tab>
```

```
: (-300, -250, 1)
```

```
> + < (-1, 0, 0)
```

```
: (-301, -250, 1)
```

In the above, the result is to be saved back to the stack. Addition is the mode selected with +. The vector from the stack is popped, and the (-1, 0, 0) vector is added. So only the first value in the vector is affected. The last two values in the vector, are 0 so no change.

As indicated in the output above, the resulting vector placed back on the stack is (-301, -250, 1).

Pressing tab will just give us the vector. Without losing our vector on the stack, get the Alien ID with:

```
> Object.FindNext Wall1
```

```
: 0x00000007000000EF
```

Set the wall's new position:

```
Object.SetPosition <tab> <
```

NOT Operator

You can perform operations on values, for example, a **not** operator.

In `BounceAlt.ini` is the following section and property:

```
[Bounce]
ShowCursor              = false
```

```
> Config.GetValue Bounce ShowCursor
```

```
: false
```

`not` the value that is popped from the stack, and store the result back to the stack:

```
> not <
```

```
: true
```

```
> not <
```

```
: false
```

Aliases

As mentioned earlier in this guide, commands, especially frequently used ones can become tiresome to type. Autocomplete is helpful, but an alias is often handier. Thankfully Orx allows ways to create aliases for commands.

You can see what current aliases are available with:

`Command.ListAliases`

The Orx Console comes with over 50 pre-packaged aliases. One of these is the question mark `?` which is an alias for `Command.Help`.

So instead of having to type, for example:

`Command.Help Config.GetValue`

You can simply type:

`? Config.GetValue`

We've used Config.GetValue and Config.SetValue throughout this guide quite a lot. It's a little mean to wait until now to mention that there was an alias for both of them:

`Get`

and

`Set`

Sorry about that. But it's good to know the full command first and be very familiar. Aliases work with each other. So we can further reduce getting help on `Config.GetValue` with:

`? Get`

Now, what about our own custom alias? A very handy one would be something like:

`d 1`

... to turn on physics debugging or:

`d 0`

... in order to turn off Physics debugging.

We'd use the `Command.AddAlias` for this, but we'll check how to use the command first:

`? Command.AddAlias`

```
: {orxSTRING Alias} Command.AddAlias (orxSTRING Alias) (orxSTRING
Command/Alias) [orxSTRING Arguments = <void>]
```

So, the command becomes:

`Command.AddAlias d Config.SetValue “Physics Debug”`

Now you can do a `d 0` or `d 1`

This command is only temporary. If you exit your game or application, it is gone.

However, you can store aliases in your config to persist them. You can do this by adding a `Console` section to your config file (say: orx.ini), and placing an alias:

```
[Console]
d = Config.SetValue Physics ShowDebug
```

The above means: when using the d command and pass a value, it will turn on or off physics debugging. You can use any letter or command you wish if it is not already used. You will need to restart your application to pick up the new setting and re-enter the Orx Console.

You can even go further and have a tricky little toggle command to do the same thing without passing a value. If we used the following command sequence in the Console, Physics debug would toggle:

`> Config.GetValue Physics ShowDebug, > not <, Config.SetValue Physics ShowDebug <`

However, we can't add this to the `[Console]` section as a command. It's something we could add to a track perhaps (too messy). Alternatively we can add it to a section, and collect it as part of Orx's lazy execution:

```
[ComplexCommandSets]
TogglePhysicsDebug = % > get Physics ShowDebug, > not <, set Physics
ShowDebug <
```

I'm not covering lazy execution in this guide, as that is another subject. However, do note that the % symbol here means, if the value is retrieved, it be be executed.

Next, add the getting of the value (and execute it) as the d command:

```
[Console]
d = get "ComplexCommandSets TogglePhysicsDebug"
```

Restart your application, and type:

d as a command in your Console to toggle the physics debug.

Dumping all config to a file

When making lots of config changes in memory with `Config.SetValue` or just `Set`, it would be great to be able to dump this all to disk and pick out the wanted parts to include in our real config files. For now, Orx does not support changing the original file on disk where a section or property originated.

And in many ways, this approach is safer.

You save everything to disk with:

```
Config.Save myconfigfile.ini
```

Or with the alias version:

```
Save myconfigfile.ini
```

Disabling the Orx Console or changing the key that activates it

Sometimes you might want to choose a different key to toggle the Orx Console. You may not want to use the default backquote ` key.

You can do this by changing the `ToggleKey` with:

```
[Console]
ToggleKey = KEY_SPACE
```

There are times when you will want to disable the Orx Console entirely for your game or application. Particularly in the case when you are building a release version. You can do this with:

```
[Console]
ToggleKey =
```

Dynamic Values versus Static Values

Some values in the config are static, and some are dynamic. It is not always clear which is which. Some values such as the `ShowDebug` property in the `Physics` section is dynamic. If you change this value, Orx will reload it and apply it immediately. Random value ranges are dynamic. Lists are dynamic values:

```
[Fade]
SlotList = AlphaFadeIn#Scale#Wobble#Vanish
```

If this list was changed in the config, its change would be applied immediately.

Other values, like floats on a spawner are not dynamic. In the case of a spawner, unless you destroy and recreate a spawner, the changes in values are not applied.

There is a trick around this. See the following section and property:

```
[Spawner]
WaveDelay = 0.075
```

By converting this property into a list, the value will become dynamic:

```
[Spawner]
WaveDelay = 0.075 # 0.075
```

Conclusion

Hopefully this guide was helpful and is giving you some great ideas of what is possible. Anything you do here in the Console can be used in tracks giving you the same power in-game.

Check out some of the track tutorials and track examples to see how you can apply this.

From:
https://orx-project.org/wiki/ - **Orx Learning**

Permanent link:
https://orx-project.org/wiki/en/tutorials/config/guide_to_the_orx_console

Last update: **2025/09/30 17:26 (14 months ago)**