

So you want to code in Fenix?

A Beginner's Tutorial by EvilDragon with a lot of help by Josebita

1. What is this?

It's a **tutorial** if you want to **begin to code in Fenix**. Fenix is a programming language specially designed to easily **code 2D games**. An interpreter exists for various platforms (e.g. DC, PC, GP32). This tutorial is mainly aimed at GP32 users, but can also be used to learn Fenix if you want to code for a different platform.

While writing this Tutorial, I am myself trying to learn Fenix – so basically it's a WIP Tutorial which will grow while I learn :)

Chapter 1 – Procedures, Functions and Variables

First off: I **KNOW** you want to start coding now... but I'm also sure that **we need to think some stuff through** before the actual coding starts (it starts in chapter 2 – **promised!**) So, this chapter may be boring to some of you (hey, who ever said that learning is always fun??) But it's very essential getting to know how something works before actually using it.

1. How coding basically works

Once upon a time, somebody invented a computer. From that time on, coding started... and it has changed A LOT!

Basically, **a computer runs programs line by line**. From the top to the end. Similar to reading a book.

So, if you created a shoot-em-up game, it should run like this:

Beginning:

Draw your ship.

Draw the enemies.

Draw the bullets.

Check for collisions.

If a bullet hits an enemy: Kill him.

If an enemy hits you: Kill you.

Check for any joystick push

Calculate your new position according to joystick movement

Create a new bullet if you push fire button

Calculate movements for enemies

Go to the Beginning.

That's **very basic** – and I think everyone understands it. It just does everything needed for the game **from the top to the bottom** and starts over, doing the same stuff again and again. Of course, we needed some more parts in the program what happens when you get killed or an enemy gets killed, but this is just to show you how coding worked earlier...

Well, the **advantage of THIS coding method is**: Very easy to understand for everyone if you look at the structure.

BUT as soon as your game gets **more complicated**, the loop it has to do gets longer and longer until it is **totally messy**... so you can mainly only create basic games this way.

So, **how does coding work today?** Multitasking is the keyword.

Multitasking basically means, that **more programs (also called processes) than one are running at the same time.**

So, **instead of one large loop** (like the above) where line after line is executed to make all the movements, collision checks, etc., we now have **more smaller programs/processes** that do all this continuously at once.

Basically, we would have the following processes **running at the same time** to achieve the same thing than the above one (each line represents process):

Your ship (joy polling, movement, drawing, creating Bullet process, collision checking)
Enemy (movement, drawing, collision checking)
Bullet (movement, drawing, collision checking)

Instead of one large, a bit messy code, we have three simple, not so messy codes, running continuously.

BTW: **Each enemy and bullet has its own process.** It's possible running **multiple instances of one process.** Nice, eh?

What does this mean for us?

Well, we can create A LOT more **flexible programs very easily.**

Bug tracking also is easy, because if everything works except ship movement, you know exactly where you have to search :)

But this also means, that **we can't just code from the scratch** (well, not if you create a complex game): **Before we start a game, we need to think about it**, take a pencil and a paper and write down **all the needed processes** for your game. Otherwise, it would end up in a messy code, adding something here and there because you simply forgot that.

Please keep that in mind. It's not just some talking, you certainly will realize how important that is once your stuck in the middle of a code wondering why it won't work the way you want it...

Some interesting thing you should keep in mind when working with processes in Fenix: If a **process** creates another process, the **first is called „Father“** and the **created one „Son“**. This has a reason: If you **pause** or **kill** the **Father**, you also **pause** or **kill** the **son**.

Why is this useful, you ask?

Well, imagine a situation in a shoot-em-up game:

There are about **20 bullets** and **10 enemies on the screen** (alone these are 30 processes). Then **you get killed**, the game is over and the **title screen** appears.

When the **title screen appears**, all enemies and bullets shouldn't be there – you'd have to kill them. What's better – killing the father which kills ALL the other processes at once or killing **ALL** the other processes **MANUALLY** – one by one?

I won't answer – **if you can't answer this one correctly, maybe it's better for you to play games than code games ;)**

It's similar if you pause the game: You don't want to pause all processes one by one – do you?

BTW: A process either loops somewhere or it will run only one time until the end and then kill itself.

That concludes our first look at processes. Don't worry if something is unclear yet – you'll learn how to work with processes while coding some games :)

2. Functions

Besides **processes**, you'll also use **functions** in your game.
What's that, you ask?

A function, once defined, basically **works like a command**.
e.g., imagine, you need a command which does the following:

Calculate the movement of an enemy (AI)

Well, if you only have **ONE** kind of enemy, you could **include the AI in the enemy process**.

But if you have written **different processes** for **different enemies**, but the **AI is the same** – then just write the **AI code as function** and call that function from the process.

Saves memory (because you need less code) and keeps the code easier to read (it's not that messy).

You can also pass variables to a function... which leads us to our last section today.

I know that the function part is rather short in this chapter – and again, you may not understand everything I told you.

It's easier if we go through example codes – and you can always use the boards to ask stuff you didn't understand :)

3. Variables

What's a variable, you ask?

That's easy: **A variable** saves numbers, text (strings), dates, etc. It's a placeholder. It is, for example, used for the **number of lives** you have left. Or your **score**. Or a **countdown**.

You can **calculate with variables**. For example, if you have a variable called „score“ and you add the value 2000, the new value in this variable will be the old one +2000.

You **need to define the variables** used in your program **before you use them**.

That's because Fenix needs to know where the variable should be used and what should be in them. Sounds complicated? Naaah, not at all.

There exist different types of variables:

GLOBALS

These variables can be **read / written anywhere in the game**.

This means: If the current value of the variable „score“ is 10000, it has **that value in any process or function**.

LOCALS

In this case, the variable with the **same name** has its **own value in every process**.

For example, if you set a value in the variable „score“ in Process 1, this value is **ONLY set** in Process 1. Process 2 has its own value for the variable „score“.

And if you change the value of „score“ in Process 2, it won't affect the value of „score“ in Process 1.

But: It's possible to read the value of „score“ from Process 1 in Process 2 (by telling Fenix it should use the variable from Process 1).

PRIVATE

Similar to **LOCALS**, with the difference that **it's not possible to access to this variable from a different process.**

This defined variable has its own values in each process / function.

CONST

A **fixed** variable that cannot be written to.

It always has the defined value, it's impossible to change it.

Basically, it's possible to **define a variable anywhere you want in your code** – however, **I advise you to define all variables at the beginning** of your program (except private ones: you can define it at the beginning of the process which uses the private variable). Again, this is useful if you **don't want to have a messy code** :)

What kind of data can be saved in a variable?

Now, we know about the different types of variables. But what about telling Fenix **what kind of data should be saved IN a variable?**

Usually, Fenix finds it out itself. For example, if you set the variable „test“ to 1, it automatically thinks that the variable „test“ is an integer (=numbers).

But what **if you don't want that „test“ is an integer**, but a **text string** and the 1 you set it to should be saved as text?

Well, then you need to tell Fenix it's a string, not an integer.

Fenix supports lots of variables: *integers, floats, chars, strings, byte, word, and more.*

If you want to know more about the variables: Either look into the [Fenix Language Reference](#) or wait for the next chapters of the tutorial – as I will explain the most important ones with example code :)

What are the names of the variables?

Well, basically, you can give a variable **every name you want** – it doesn't care.

However, I'd rather call the variable used for the player's score „score“ than „Paul“.

It makes it easier to remember what's that variable is used for.

Another thing should be know:

Fenix has some **pre-defined variables**.

For example, **x** and **y** used in a process where graphic is displayed, are the coordinates where the graphic should be shown on the screen.

Again, we'll learn some more with coding examples, but you can always look them up in the [Fenix Language Reference](#).

This concludes Chapter 1 of the tutorial.

In the next chapter, we will finally start to code something in Fenix :)

But, without the basics, it's hard to code something :) Now you know the basics – so start coding!

See ya soon, EvilDragon.