



AOZ STUDIO

USER GUIDE



TABLE OF CONTENTS

1. THE MAGIC BEGINS.....	9
2. THE BIG PICTURE	13
3. OUR FIRST MAGIC PROGRAMS	17
Display Actor's Images	17
Let's see an Actor	17
Moving the actor	20
Pause, Do...Loop, If...Then	22
Move the actor with the keyboard	27
Move the actor with the Joystick	31
4. INTRODUCTION TO ANIMATIONS	33
LookAt\$.....	33
The HotSpots	34
AUTO\$.....	37
Use Actor to scroll	38
Where are my images saved?.....	41
5. MORE MAGIC	44
6. THE EDIT MODE	53
The top menu buttons	54
The project folders.....	55

7. CREATE A PROJECT APPLICATION.....	57
8. ERRORS, BUGS AND HELP	60
9. THE DIRECT MODE	65
10. GOING FURTHER WITH ACTOR	67
The various possible controls	67
Keyboard	67
Joystick	68
Mouse	69
Automatic	70
Collisions.....	72
Collisions with Actor Col.....	72
Collision precision	73
Collision with the Actor instruction.....	74
Mice and Actors	77
OnMouse\$......	77
Drag & Drop.....	79
11. THE LISTENER AND EVENT SYSTEM	80
OnCollision\$......	81
OnControl\$......	81
OnLimit\$......	82
OnAnimChange\$......	82
OnChange\$......	84
How to use the Listeners	85
MORE CONTROLS OVER Actor	86
Reset Actor	86
Del Actor.....	86

Visible Parameter	86
Enable Parameter	86
12. ANIMATION	87
What's a sprite sheet?	87
13. ACTOR SETTINGS	89
14. LET'S MAKE A GAME	94
15. PUBLISH IN A CLIC	104
Some explanations.....	104
Now we publish our work!	105
16. THE DEMOS AND GAMES	106
17. KEYBOARD INPUTS.....	108
Keyboard variable inputs	109
Input	109
Keyboard Buffer	111
Keyboard character input	112
Inkey\$.....	112
Input\$(N).....	113
Reading the keyboard state.....	114
Wait Key.....	114
Wait Input	114
Wait Click.....	114
Key State	115
Key Name\$	115
ScanCode	116
Key Shift	117
ScanShift	119

18.	FORMATTING TEXT	121
19.	STORAGE OF IMAGES AND SOUNDS	123
	Storage in a Memory Bank.....	123
	Loading On Demand.....	125
	Assets.....	125
	Compatible file formats	127
	Default folder	129
20.	AUDIO MAGIC	130
	Sound effects.....	130
	We need to talk	131
	Speech synthesis	131
	Speech Recognition	133
	Speech extra features.....	134
	Samples.....	136
	Music.....	137
21.	MOUSE INPUTS.....	138
	Using the Mouse	138
	Change Mouse Shape.....	138
	Hide.....	139
	Show.....	140
	Hide On	140
	Show On	140
	Limit Mouse X1,Y2 To X2,Y2	141
	= Mouse Click	142
	= Mouse Key (Button).....	143
	= Mouse Wheel.....	143

= Mouse Zone	144
= Mouse Screen	145
= ScIn(X,Y)	146
= X Hard(X_SCREEN)	148
= Y Hard(Y_SCREEN).....	149
X Mouse =	149
= X Mouse	150
= Y Mouse.....	150
= X Screen(X_HARD).....	151
= Y Screen(Y_HARD)	151
22. INCLUDE	152
23. AOZ TAGS	153
What is a tag	153
How to use the tags	154
List of available tags.....	155
Transpilation tags	155
Tags on applications	169
24. AMOS USERS	175
Compatibility	175
Bobs et Sprites	176
AMAL.....	178
25. INSTALLATION D'AOZ STUDIO.....	179
26. THANK YOU	180



1. THE MAGIC BEGINS

Hello!

Glad to welcome you to the magical world of AOZ Studio.

If you know nothing about programming but want to publish the app, game, or website of your dreams, you have come to the right place. If you are already a pro and want to develop much faster, you have come to the right place.

If you want to start programming right away, you have come to the right place, otherwise you can start by reading: *AOZ Studio - Introduction*.

INSTALLING AOZ STUDIO

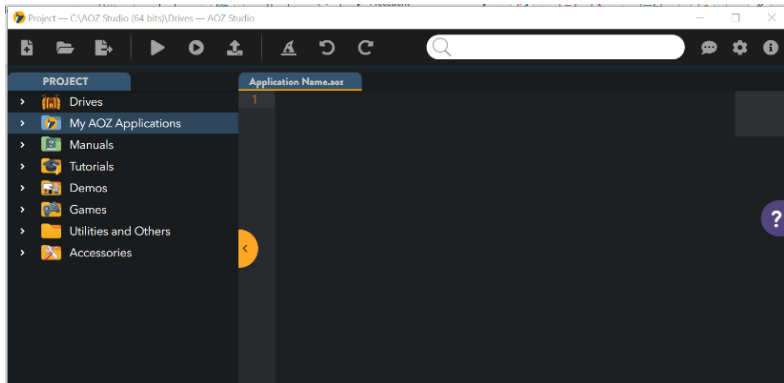
Very good news! Only one installation of the "AOZ Studio" app on your PC or MAC is required. Everything is included, and once the installation is completed you are up and running to start programming. I confirm you don't have to read 20 pages on how to Install, configure editor, compiler, libraries, SDK, API...

To get the latest version of AOZ Studio, guess where:

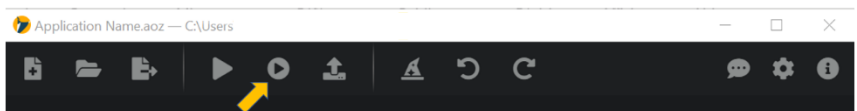
www.aoz.studio

YOUR FIRST PROGRAM

Please launch AOZ Studio on your PC or MAC.



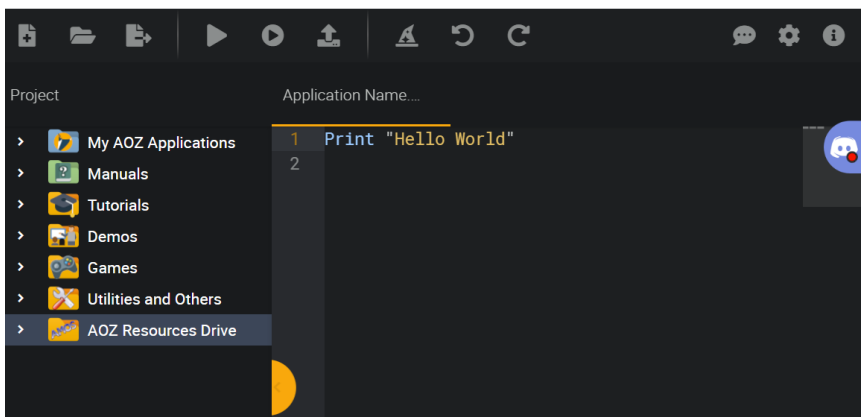
You will discover at the top the most important buttons like **RUN**. That's the one below the play icon:



Now look in the large black, empty, window on your AOZ Studio screen and type in your first line of code:

Print "Hello World"

Like this:



Now Click the RUN button (or hit F2 key)



Wait a few seconds, click, and now you can see printed "Hello World".

Congratulations! You are now a computer programmer.

If you think that's magic, you're absolutely right. What you did there was give your computer a simple instruction to print something on the screen, and to do that you used **Print**, which is an instruction your computer understands in the AOZ language that you have begun to learn.

OK, let's move forward.

MORE...

Type in these lines of program, using the Return key to create each new line:

```
1 Print "Hello World"
2 Wait Key
3 Boom
4 Print "goodbye"
5
```



You'll notice that AOZ tries to help you every step of the way by offering help and hints as you type, and it will even display different elements of your computer code in their own colors. For example, instructions are shown in blue.

Ready? Here comes the magic again.

Click on one of the **RUN buttons**. Yes actually we have two of them at the top of the AOZ Studio screen, they look like this:



There are two of them, because your program may run on:

- a Web Browser  (like Chrome, Edge)
or
- in the AOZ viewer  (inside AOZ Studio)

Once again wait a few seconds and your program will run automatically.

Do you have any questions about your new program?

- The instruction **Wait Key**, simply tells the computer or smartphone to wait until a key is pressed
- The instruction **Boom**, guess what...

So this program will print “Hello World”, wait for a key press, then play the boom sound and politely say goodbye. We like to keep your machines polite.

Now you know the main principles. We are going to have much more fun, but first let’s see some quick boring stuff.



2. THE BIG PICTURE

This chapter gives you an overview of AOZ Studio and the AOZ language. If you are impatient and want to move on to the next hands-on chapter to create your own program, that's fine.

KEYWORDS AND SYNTAX

Every big picture needs the perfect frame. The framework for AOZ the language, consists of a load of simple instructions (or keywords), and like any other language it uses them and a set of rules called syntax to compose everything together. Instructions and syntax - that's all you need to create your wonders. Like any other language there is a huge vocabulary of instructions, more than 700 in AOZ , and a wide range of simple syntax, but don't worry about them. As you create your magic, AOZ Studio will not only recommend instructions, it will also help you with the syntax.

BASIC

We use a type of language known as BASIC, which stands for Beginner's All-purpose Symbolic Instruction Code. Or it could stand for Build A Super Incredible Creation.

All instructions are defined within the AOZ language itself, and everything is what we call modular, which is just another way of saying that all the building blocks you need for your creations are built in and you can combine them and arrange them as you wish.

AOZ uses the BASIC standard as a base, but has extended it very widely in these capabilities (including with the ability to do what is called the object language).

SIMPLE AND POWERFUL

AOZ is easy to learn, It's also very powerful, which makes it ideal for hardened professionals too. It is very simple and it is also very deep, with a superfast development cycle, for you to create your dream game, mobile application or website, all in a fraction of the time needed with other tools. But just because it's easy, that doesn't mean to say AOZ isn't very powerful. You have the power to control as many objects on your screen as you want. Best of all, because AOZ lets your computer handle huge amounts of data and screen pixels all at the same time, your applications will run very fast indeed. Well, one of our team is British.

GETTING SERIOUS

If your interest is not so much in computer gaming and more in serious matters, then you won't be disappointed. You will be able to create some very serious applications with AOZ, including databases, utilities, user interfaces and DOM extensions for webby stuff like HTML and XML.

MODERN, UNIVERSAL AND FAST

Your AOZ applications may talk to computers, smartphones, and any devices using a Web browser, with modern doodahs like node.js and JavaScript libraries.

AOZ Studio also allows a simple way to define your own new instructions and functions. In other words, if you need something then you can invent it yourself. Your new instructions and functions can be then be used in your own applications, published on the Internet and reused by others.

Best of all, you can program your new instructions in AOZ itself using nothing but simple AOZ instructions, but you also can use JavaScript if you like. Make new tools with the existing tools, program your language using our language.

TRANSPILER

AOZ Studio includes what professionals call a transpiler.

A transpiler is a smart utility that takes code from one language and converts it into another language, like a translator. In this case the AOZ transpiler takes code from the AOZ language, the one that you are already using, and translates it into something called JavaScript and something called HTML5 code. This is the code that fancy websites run on. Your AOZ code becomes superfast and very portable, it does all the hard work for you, and it gives you amazing control over graphics, animations and audio. Very importantly, your code will be compatible to run on all Web browsers on any machine like a PC, MAC and smartphones, and you will see later that it will also run as standalone apps (.exe for PC, .apk for Android,...).

So to summarize, your program made with the AOZ language is converted by AOZ Studio into JavaScript, the most commonly used language. It was important for us to help you move from one language to another, you can mix AOZ and Javascript, but we bet you'll mostly use AOZ, like many professional developers because programming is twice as fast, and fun, with it.

FREE OR FEE

We are so nice that we have made an AOZ Studio free version and it is open source. That means you are welcome to use AOZ for free. We hope you like it so much you'll want to pay us an extremely modest license fee to use the transpiler, because we have to eat and make the magic potions!

- The free version is complete and you can create an unlimited number of programs with it. It includes some advertising and you, only you, are allowed to use your own wonders.

- The license fee (Paid version) has no adverts, and it allows you to share your creations, publish them or sell them to anyone and everyone without paying us any royalties. After all, you should keep all the rewards.

And we are so nice that it is our policy that non-profit and humanitarian organizations will always get the full product for free.

If you are an advanced coder, remember the AOZ code is Open Source (Public Domain), you can create your own branches, optimize the code, and insert your own coding ideas and achievements. You will be more than welcome to help in the development of AOZ Studio.

No matter if you are advanced coder, a lazy programmer or a rookie, you are very welcome to join us on DISCORD (it's a Chat app), you will meet a great community of Magicians who will help you and talk directly with the AOZ Studio team: <https://discord.gg/6Cuy3vtj8N>



3. OUR FIRST MAGIC PROGRAMS

Display Actor's Images

Can't wait to get images moving on the screen?

We understand you, and that's why we created the **Actor** instruction, so you can be the director in your Actor Studio, and direct your Actors.

There are several AOZ instructions for manipulating images, but the **Actor** instruction is the most powerful. It simplifies display, manages animations, it controls collisions and it will allow you to do magical things.

An actor is a graphic element that can be animated. For example, in a game it's the character that the player controls, an element of the scenery, or an enemies within the game. In other words it's a visual element, essential for a game as it is for a professional application.

Notes for more experienced programmers:



- As soon as you want to view or move an image, animated or not, you can use either the BOB instruction, which is AMOS-compatible, or the Actor statement we're talking about in this chapter.
- If you want to program in Object mode (OOP) the BOB statement is also OOP-compatible with settings similar to Actor.

Let's see an Actor

The Actor instruction has many parameters that themselves have many properties or values. But don't worry, you can just define what's useful to you, in fact all settings have default values that allow you to change only the settings that interest you.

Here we go then! Tap the following code, having cleared all other lines of the programs that may already be on screen:

Actor "magician", Image\$="magic.png"

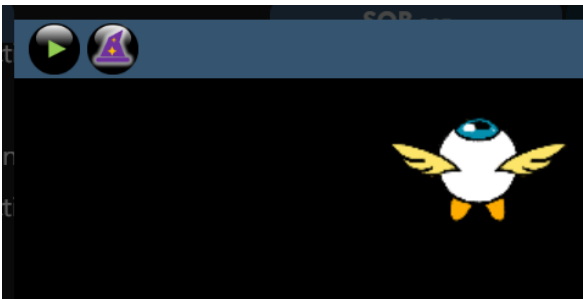
Now execute this program by clicking on the **RUN** button to display the «magic.png» image on the screen.



You have created an actor, his name is "magician", and his image is called «magic.png»: and he's alive. It's as simple as that!

Now add two parameters called X and Y to your code, like this:

Actor "magician", Image\$="magic.png", X=400, Y=40



Let's take a closer look at the different parameters used in this example:

"name of the actor"

The name of the actor, which is "magician" in this example, is the first parameter and you have to specify it. Most of the other parameters for the Actor instruction are optional. You can give this parameter a name or a number, for example:

Actor 10, Image\$="magic.png", X=400, Y=40

We will see later that it can be convenient to use numbers for moving several actors at the same time. You are free to choose names or numbers, but avoid using spaces and characters with accent marks. By the way, if a name is already used by another actor, then it is the other actor that will be affected by your settings.

Image\$ = "name of the image"

The Image defines the appearance or the "suit" that the actor will wear. It is literally their image.

As usual, after the image is set in the Actor instruction, then that defined image will be used for this actor. And if no image is set then the Actor instruction won't show anything.

We will see that many image formats can be used like PNG, BMP, JPEG.

X- horizontal position, Y- vertical position

These two parameters define the position to display the actor on the screen, the value of X and Y are in pixels (the smallest point displayed on the screen). In our example the image appears in position 400 (horizontal), 40 (vertical) from the top left corner of the screen.

Remember, these parameters are optional, just like the others, and if they are not defined then the last X and/or Y positions are used. For example, say you have placed an actor at 50, 50 previously in your program, if you do not assign values to X and Y in a new **Actor** instruction, then the start position it will be displayed at displayed is 50,50.

Note: by default if X and Y are not set, the position of the actor will be 0.0. Yes I know it's logic.

AOZ adapts to your logic

AOZ lets you chose the order you use for your magic potion.

If you prefer:

Actor 10, X=400, Y=40, Image\$="magic.png"

Rather than:

Actor 10, Image\$="magic.png", X=400, Y=40

The magic will still work!

MOVING THE ACTOR

Now that we know how to display an actor, let's change our code to move it across the screen.

We're going to change our program and add a new parameter, called **EndX** like this:

Actor "magician", X=0, Y=0, Image\$="magic.png", EndX=1980

Run the program by clicking on RUN (or hit the F2 function key), and your actor will move from left to right of the screen.



Note: to achieve the same thing in most other languages, it can take a lot more lines of code, 20 to 100 lines. Each language has its characteristics, strengths and weaknesses, AOZ has been designed to offer powerful programming simplicity,

Let's break down our program. We've set the new **EndX** parameter with 1980. **EndX** tells our actor that it must move from its current position to an end position defined by **EndX**. When this position is reached, the actor's movement stops.

Remember: you can put your parameters in any order you want, so you can also write:

Actor "magician", Image\$="magic.png", X=0, EndX=1980, Y=0

To sum up: the starting point and end point of the movement is automatically taken care of by the **Actor** instruction, and it starts immediately, in this case at X=0, Y=0 until the X position is equal to the value defined by the **EndX** parameter, which is 1980.

We'll look at the **Duration** parameter that changes the speed of movement very soon.

It's your move 😊

- Stop the actor in the middle of the screen.
- Move the actor vertically using the same principle as the code above.
- Move the actor from the right of the screen to the left.
- Move the actor diagonally
- Use the image name lucie.png

PAUSE, DO...LOOP, IF...THEN

Here are some other important concepts for a programmer.

By default the actor movement starts automatically, but in some cases you will need to control it. For example, to pause it when you want. So let's update our code :

```
Actor "magician", X=10, Y=0, Image$="magic.png", EndX=1980
Do
  A$ = Inkey$
  If A$ = "t" Then Actor "magician", ActionMove$="pause"
  If A$ = "y" Then Actor "magician", ActionMove$="play"
Loop
```

Run the program (RUN), and our actor will be displayed on the screen and move. Now quickly press the **t** key on your keyboard, and the movement will pause. Next press on the **y** bar, and the move will continue. Do it again, press **t** to pause then **y** to continue. Now stop! Understandable, but not understood by your computer which will repeat the loop indefinitely without getting tired.

But at AOZ Studio we are very kind with computers and magicians, so it's possible to stop a program by pressing the **Ctrl** and **c** keys simultaneously. (The Ctrl keys are located at the bottom left and right of your keyboard.)

Let's break the program down.

-Lines 2 and 6: the **Do ... Loop**. This is an important new concept. It's a two-part instruction to indicate that the part of the program between the Do and the Loop is to be repeated in a loop. In other words the loop is a part of code that gets repeated forever until the program, or the use Ctrl plus c, decides to end the madness.

-Line 4 and 5: a new parameter is set for the Actor:

ActionMove\$. This allows you to affect the actor's movement. There are two possible states: "**play**", which is the default value, and "**pause**", which interrupts the move.

-Line 3: the variable **A\$** stores the value of the pressed key, returned by **Inkey\$**

Note: If no key is pressed, **Inkey\$** returns an empty value or "". If the key is the space bar the **Inkey\$** function returns a space or " ". (Note that there is a space between the 2 quotation marks.)

Inkey\$ will return "t" or "T" if the keyboard is capitalized, but only the lower-case t will be recognized here.

Now let's look in detail at a very important instruction composed of three words, of which only the word **If** is mandatory.

Introducing the famous and indispensable **IF... THEN... ELSE**:

- **if A\$** contains the letter "t" it's because the key has been pressed, **then** the **ActionMove\$** of the **Actor** instruction is set with the "**pause**" property which stops the animation.
- **if A\$** contains a "y" then we change the **ActionMove\$** of the **Actor** instruction with the "**play**" property which relaunches the animation.

We have now covered two very important instructions: the **Do...loop** and the **If... Then... Else**.

If you don't understand it all, no worries. We'll review this in more detail later, but we suggest that you have a go at modifying this program to understand it for yourself.

It's your move 😊

- Change which letters of the keyboard pause and play.
- Add a **Print A\$** in the loop. Have a go.



As you can see with AOZ it is very easy to move actors on the screen and control them. If we have several dozen actors, the Actor instruction provides the parameters to make it easy to manage them.

Come on, let's keep going and try something more complicated. Change the program as follows:

```
Actor "magician", Y=100, Image$="magic.png"  
PX=0  
Do  
  PX = PX + 16  
  Actor "magician", X = PX  
  Wait Vbl  
Loop
```

Run the program (RUN). The actor gradually moves from the left to the right of the screen.

I know what you're thinking. You're thinking *that code achieves the same result as when we used the simple line:*

Actor "magician", Image\$="magic.png", Y=100, EndX=1980

So why make it complicated when you can make it simple?

Because it's important to understand what's going on behind an instruction, in terms of the value of a parameter.

The use of a powerful command such as Actor should not let you ignore the mechanics that allow it to produce its effects.

Let's break down our program once more:

Line 1: Our actor is positioned on the screen (at 0,100).

Line 3 and 7: The Do ... Loop, you recall, indicates that we repeat the code between the Do and the Loop continuously.

Line 4: The PX variable value when the program reach the Line 3 first is equal to 0 (value by default). This line will add 16 to PX which will then be worth 16 (new PX = 0 + 16).

Line 5: We call the **Actor** instruction to update the X setting with the value of the PX variable, now X will be worth 16.

Note: As we have already defined the value of Y and the Image in Line 1, there is no need to redefine them here, so we keep the same values (Y=100).

Line 6: Wait vbl is a needed small instruction to smooth the animation synchronizing it with your screen, we will explain later...

Line 7: The Do... Loop ends with Loop indicating that we are closing the loop and returning to the beginning of it (i.e. line 2) to run the same code again.

In summary, this loop adds 16 to the variable PX each time. Thus our actor will advance by 16 pixels in X (horizontally), without changing Y (vertically) in each pass of the loop.

It's your move 😊

- Move magician slower and faster by changing $PX = PX + 16$
- Move magician in diagonals
- Move in all directions with 4 keys using **Inkey\$**

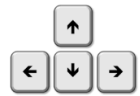


MOVE THE ACTOR WITH THE KEYBOARD

Now for even more magic, here you will automatically move your actor with the arrow keys of the keyboard using just one line of code. Clear all the previous program and please type:

Actor "magician", Image\$="magic.png", Control\$="Keyboard"

Run that program (RUN) and you'll see the actor move in different directions when you press the 4 arrow keys of your keyboard.



If you're a beginner you'll think this is obvious, but if you're experienced you'll know that normally it would take a complicated program to achieve this. In AOZ this simple line is enough.

As you can see, to achieve this we have added the **control\$** parameter, which as the name suggests determines how to control the actor. Here control is with the keyboard property, but you can do exactly the same for a joystick, a mouse or a touch screen.

With AOZ you can make it simple using the default settings, then do what you want by adjusting a few details.

Note: by default the keys used by the `Control$="Keyboard"` are the **4 arrow keys** and the keys **A S** and **W D** on an English QWERTY keyboard and **Q S** and **Z D** on a French AZERTY keyboard.

Now let's take more control to do some more accurate things. Here's another example:

```
Actor "magician", Image$="magic.png", Control$="ArrowRight:
offsetX = 4; ArrowLeft: offsetX = -4"
```

Run the program (RUN). On screen, our actor is static. By pressing the right and left arrow keys of the keyboard, our actor moves.

We see that we have defined, two keys in the `Control$`: `ArrowRight` and `ArrowLeft`. These properties are unique codes associated with each key on your keyboard (right and left).

Let's break down our program again:

The actor is positioned at coordinates 0.0 which are the default values.

We set the **Control\$** parameter value with a string (remember a string of text starts and ends with " in AOZ), in that string we assign the values of the 2 keys (`ArrowRight` and `ArrowLeft`). And for each, we have established the values to use for the `offsetX`, which indicates the number of horizontal moving pixels for the actor. A negative value has been defined for the Left key in order to move it to the left of the screen. Reminder: the top left of the screen is the position 0.0.

Each property is separated by a semicolon ;

To summarize to assign properties to a key here's what we do:

"Key code : property1 = xxxx; property2 = yyyy,..."

Ex: "ArrowRight: offsetX = 4; ArrowLeft: offsetX = -4"

If you don't understand right away, try to change those values and see what happens.

It's your move 😊

- Change the speed of your actor's movement.
- Add the properties ArrowUp: offsetY = -4; ArrowDown: offsetY = 4 to move the actor in all 4 directions.
- Reverse the movement: left key to move right, and right key to move left.

We can always make it more complicated...

Well, to continue our series of making it complicated when we can make it simple, we will now control our actor without using the Control\$ parameter. It's possible, here's an example:

PX=0 : PY=0

Actor "magician", Image\$="magic.png"

Do

If Key State(37) = True Then PX = PX - 4

If Key State(39) = True Then PX = PX + 4

If Key State(38) = True Then PY = PY - 4

If Key State(40) = True Then PY = PY + 4

Actor "magician", X = PX, Y = PY

Wait Vbl

Loop

Run the program (RUN).

Using the arrow keys of the keyboard, you discover that you control the movement of the actor very well 😊.

Let's break down our program:

Lines 4 to 7: These 4 lines allow you to test each of the arrow keys on the keyboard. The function **Key State statement (Key_number)** returns true as if the corresponding key_number is pressed on the keyboard.

If this is the case the code that follows the **then** instruction is executed.

You can of course press two keys simultaneously. Left and Top, for example, will move diagonally.

Note: When you have the chance, try the little "ScanCode Tester" program that's in the AOZ Studio folder under "Accessories", or check the site <https://keycode.info/> to find the Key State codes (or Key Code) of the keys.

Line 8: Update our actor by assigning the new values of PX and PY variables to the X and Y parameters. You see that we only update the parameters that change, remember it's not useful to reuse parameters that do not change, as for example the image «magic.png».

We've seen how to control our actor by keyboard using **Control\$** or **Key State**, but that's just the beginning. We can work wonders with our Actor with all sorts of behavioral instructions, and all is revealed in the chapter called "**Going further with your Actor**".

MOVE THE ACTOR WITH THE JOYSTICK

We saw how simple it was to use the default keys or to assign keyboard keys to manipulate our actor. Now let's see how complicated it is with the joystick.

As always with AOZ you can make it simple with the default settings and do what you want by going into details.

The simple version of the move with the Joystick is:

Actor "magician", Image\$="magic.png", Control\$="Joystick"

Connect a joystick or gamepad to your computer, execute the program, and there you are, moving your actor. Great or not?



And the more controlled version:

Actor "magician", Image\$="magic.png", Control\$= "JoyRight0: offsetX = 10; JoyLeft0: offsetX = -10"

Here the Actor's **Control\$** parameter has been set so the speed on the X axis is +10 going to the right, and -10 going to the left.

Control\$="JoystickN" is the syntax because we number our joysticks **N**, as you can have several Joysticks connected to your PC or device:

- **Joystick** (or **Joystick0**): For all directions of the 1st joy
- **JoystickN**: for the second, third,... joystick connected
- **JoyLeftN**: Left direction.
- **JoyRightN**: Right direction.
- **JoyUpN**: Up direction.
- **JoyDownN**: Down direction. We're not sure why we're telling you this, it's so obvious!
- **JoyButtonN** (button nb): is the number of the joystick action button. This is needed because there are Joysticks, Gamepads, Controllers for flight simulators that have a lot of buttons, so each needs a number.

Example: **JoyLeft0** uses the first joystick connected (well, we know the numbering starts at 0 and not 1, it's an old habit.)

It's your move 😊

- Change the speed from 10 to 6
- Add the JoyUp0 properties: JoyUp0: offsetY = -6; JoyDown0: offsetY = 6 to move the actor in all 4 directions
- double the speed of your actor's movements
- Reverse directions

As said we will see in the chapter **GOING FURTHER WITH ACTOR** even more possibilities to manipulate your actors, including how to manage collisions between several actors, and animations.



4. INTRODUCTION TO ANIMATIONS

If you've got this far then you've probably started playing with the Actor instruction parameters. maybe you've even thought of some game ideas, or applications. We hope you are starting to feel the power of AOZ Studio in your hands and the fantastic creative possibilities you now have.

LOOKAT\$

We are going to talk about about Hotspots and a fun new instruction: **LookAt\$**

Here's a brand new program so you can see what **LookAt\$** does for a living.

```
Actor "ship", X=Screen Width/2, Y=Screen Height/2,  
Image$="ship.png", LookAt$="Mouse"
```

Run the program: Our actor is displayed on the screen as you would expect. Now move your mouse, and notice how your actor seems to look in its direction!

Let's break down our program:

Line 1: We display the actor at the center of the screen which is the Width and Height divided by 2. The **LookAt\$** parameter has the property "Mouse". You need to understand that this tells the actor to orient itself in such a way as to always face the tip of the mouse.

Screen Width and **Screen Height** are 2 functions that returns the size of the screen in pixels. By default it is 1920x 1080 (full HD, 2M. pixels) but you can change the resolution of the screen.



But, but, your actor is not at all centered, and it looks like it's circling around something. Weird.

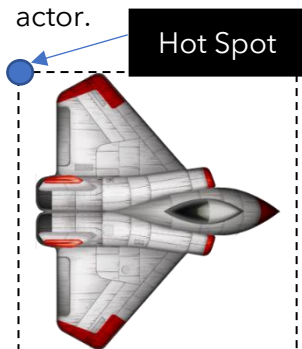
THE HOTSPOTS

To make animations you have to understand the principle of the hotspot (and Sprites sheets that will be seen later in the Animation chapter).

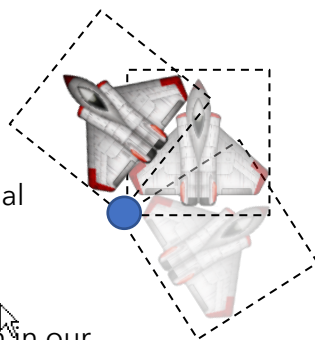
The hotSpot or "anchor point" is an invisible point, like a center of gravity that each actor's image has. The hotspot serves to:

- Position the actor precisely on the screen in X and Y
- Hold the actor's rotation center
- Center a zoom on the actor

By default, the Hot Spot is positioned at the top and left of each actor.



As it is located at this point, the rotational movement of our actor is around it:



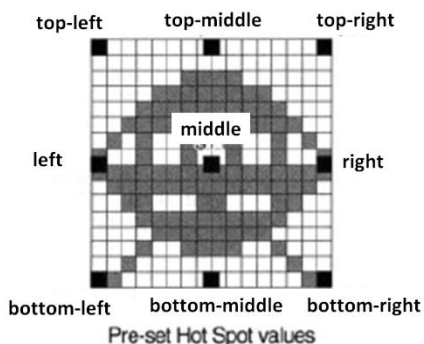
So to get the expected result of rotation in our program we need to have the hotspot of our actor at the center of it.

Nothing simpler in AOZ, and we will use a new instruction that will allow us to move this point to the right place. And this instruction, you'll never have guessed, is called Hotspot. Let's change the line of our code like this:

Actor "ship", X=Screen Width/2, Y=Screen Height/2,
Image\$="ship.png", LookAt\$="Mouse", Hotspot\$="middle"

Run the program, and see how the display problem is solved. Your actor is perfectly centered, and orients itself from its center. This is due to the **Hotspot** setting you added. This setting defines the location of the actor's hotspot in the image. Here, giving it the value **middle** this indicates that you have to take the center.

The following diagram determines the values of the Hotspot:



The hot spot is set from the top-left corner of the image. We see that the central point of the image is defined by the value **middle**.

When displayed by the Actor instruction, the hotspot's lag is added automatically by AOZ to

the coordinates X, Y, which is very nice because you have nothing to do. I like it.

Note: It is perfectly legal to position the Hot Spot outside the current display of the screen. This can be used for example for a game where Sprites disappear and reappear off the edge of the screen.

Remember that the Hotspot is an anchor, i.e. the X and Y positions are the position of your actor's Hot Spot. For example, if you change your line of code by putting X=0 and Y=0:

Actor "ship", X=0, Y=0, Image\$="ship.png", LookAt\$="Mouse", Hotspot\$="middle"

You will see that your actor will be partially hidden by the edge of the screen because the point displayed at 0.0 is actually the center of the Sprite not the top left corner.

To find the position of the original hotspot, simply set the Hotspot setting as the value **"top-left"**.

The Hot Spot may not be the primary topic of this paragraph, but it's important to understand it.

So now let's go back to **LookAt\$**.

The **LookAt\$** instruction tells the actors that they need to look in a specific direction. Earlier, we asked the actor to follow the mouse pointer by defining the **property** = "Mouse."

Let's look at the different **properties** that can be given to **LookAt\$**:

- **Mouse:** As we have just seen, the actor follows the mouse pointer.
- **N:** The actor looks at the other actor (good for actors in love). N is the number or name of the actor to look at.
- **X,Y:** The actor looks at a fixed point on the screen. If the actor moves, it will always follow this point. X and Y are the coordinates of the point to look at.

Examples:

- LookAt\$="Mouse"
- LookAt\$="MyLove"
- LookAt\$="100,350"

AUTO\$

Let's go further and change your code like this:

```
Actor "ship", X=Screen Width/2, Y=Screen Height/2,  
Image$="ship.png", LookAt$="Mouse", Hotspot$="middle",  
Auto$="forward=4"
```

Run the program. Your actor continues to look at the tip of your mouse, but this time it's chasing it, so watch out!

Auto\$: is an other great parameter of Actor, it gives the instructions to do automatically what is indicated by it's property, here **forward=4** will move automatically (towards the mouse pointer).

USE ACTOR TO SCROLL

Type this little 3 lines program please:

```
Actor 1, X=0, Y=0, EndX= -1920, Duration=10000, LoopMove=True,  
Image$="bg.png"
```

```
Actor 2, X=0, Y=880, EndX= -1920, Duration=7000, LoopMove=True,  
Image$="ground.png"
```

```
Actor "magic", X=Screen Width/2, Y=Screen Height/2, Image$="magic.png",  
LookAt$="Mouse", Hotspot$="middle", Auto$="forward=8"
```

Run the program, do you see:



Yes with 3 lines of AOZ code only!

You are a magician, I told you, did you believed me?

We have added a new parameter: **LoopMove**, when the value of it (or property) is **True**, here, the Actor's image will scroll in a loop, it repeats the movement X,Y to EndX, EndY.

We are scrolling horizontally 2 actors, named 1 and 2, with each an image.

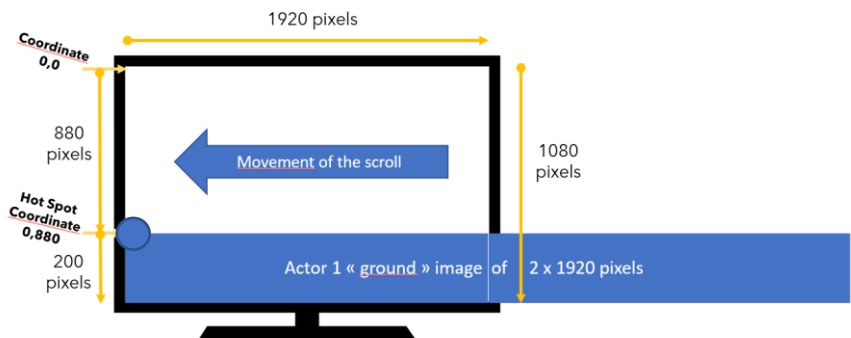
Note: the forest (Actor 2) starts at the top left corner ($X=0, Y=0$), while the ground image (Actor 1) is placed at position 0,880 to start with, and move to -1920 ($\text{EndX} = -1920$). The movement last 700ms so quicker than the forest (the scrolling is faster).

WHY -1920?

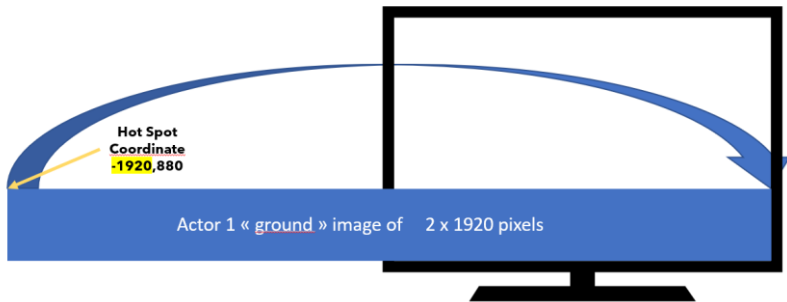
First you have to know:

- The screen horizontal resolution is 1920 pixels (1920*1080 are AOE Studio default screen sizes but you can change it),
- In this example the 2 images are 3840 pixels or 2×1920 long,
- When **LoopMove** is at True, it means that when the requested animation of the image is ended it is starting over again.

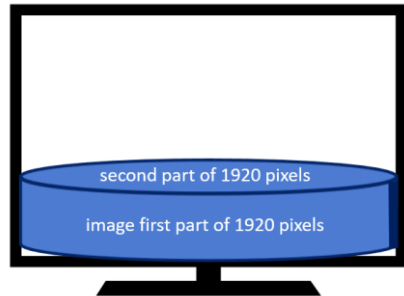
Remember the hotspot? Hope so it was just the previous chapter... The "anchor" or hotspot of the ground image is at the top left of the image, so from there to move the image to the left the EndX have to be less than 0 (here -1920).



When half of the image (1920) disappeared to the left of the screen the **Actor** instruction will start to loop:



And the loop continues, it's like a spinning wheel.



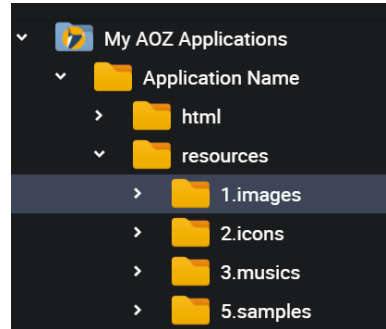
But that's enough for now, and there will be plenty of brilliant and bonkers things to see later. Let's continue to explore AOZ Studio and become the master of the computer science universe.

WHERE ARE MY IMAGES SAVED?

At this stage you may have wondered, or asked on the AOZ Studio Discord channel: *But where are all these images saved that I am playing with in these examples? How do I add one?*

Here are the answers, right away.

Each program you create will have its own resources folder in which you will also find the resources sub-folders. That's where the 1.images folder lives, and that's where you can upload your new images, see here:



Once an image, a sound,... is there, the program have a direct access to it, just by its name. (and you do not even have to give the extension, like lucie and not lucie.png)

-You can copy/paste sounds, music, images in the corresponding resources folder of your application by dragging the file in it.

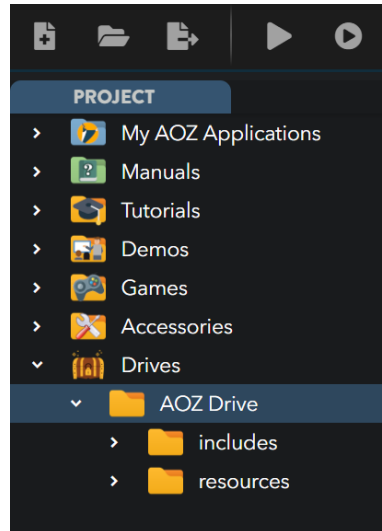
-You also can copy a file by dragging it from an other AOZ Studio folder while the Ctrl key is pressed.

- You may find those folders on your hard drive by default: Documents/My AOZ Applications/**your program**/resources/



OK but I need to tell you a secret, the images, sounds that we are using in this User guide are coming from a secret folder... What? Yes I know, we did create a secret global shared assets folder: the "AOZ Drive", it includes already the shared generic images that we are using in this User Guide (btw thanks Romain for our great magicians, Lucie, Magic, and all AOZ characters).

Thanks to globally shared assets when you create new apps you do not even have to copy images on your Application/resources folders, AOZ Studio is doing it for you so you can directly use all the ones that are in the AOZ Drive global shared folder.



You got it? If you save say an image in that global folder you can use that image (or sound) in every applications just by its name. This is really for the ones you use always like your logo, photo,... stuff that you need to reuse.

This secret folder, I can now reveal where it is, for the images:
"C:/AOZ Studio/AOZ Drive/resources/images

How it works?

When you RUN your application AOZ Studio copy the resources you need (the one used in your program) from that secret AOZ Drive folder to your application resource folder. And when it is there, your program may use them directly.

AOZ uses several image file formats. .PNG, .GIF or .SVG are the ones we recommend, because they allow for transparent backgrounds, which are required for the best results with Bobs and Actors. You can also use other graphic formats, such as .JPG, .BMP, .IFF, or .ILBM graphics, but you'll have to manually mask the color(s) used for transparency.

Note: Another method for loading images is via the Load Asset instruction. This command loads a file from the path you select THAT MUST be in the resources folders of AOZ (not on the hard drive so we can package them with the save, aozip save and publish). You can also use Load Asset to load sound and video files either directly from the Resources/Assets folder of your application or using a path. We'll talk more about Load Asset later.



OK let's take a breath, because that Actor instruction is super-powerful and as such it will take time to master. Now it's time to learn more classical programming using a few key instructions to start with. Let's create a game together!

5. MORE MAGIC

A little recap before continuing?

You now know how to use:

- Actor to:
 - display images **Image\$**
 - position them precisely with **Hotspot\$**
 - move images by their coordinates with: **X, Y, EndX, EndY**
 - animate them automatically with **Auto\$**
 - choose how to control them with **Control\$**
 - on the joystick with: **JoyStick**and the associated commands:
 - trigger their movement with **ActionMove\$**
 - orient them automatically with **LookAt\$**
 - scroll through them with **LoopMove**

- **Print** to display a message on the screen.

... and you have started to use:

- **If... Then** to define conditions and act accordingly
- **Do... Loop** to repeat actions.
- **InKey\$** to retrieve a key pressed by the user

May be you are wondering here the differences between:

- **Inkey\$** is a function that returns the next key as a string or an empty string if the keyboard buffer is empty.
- **Key State(n)** is a function that checks if the key with the ID of n is currently pressed, and returns true if it is.

Not bad for a start ! We continue in this chapter with:

- **Input** to enter text on the user's keyboard
- **Goto** to go to part of your program
- **If... Then... Else** to complete the above, to master the conditions... and to initiate actions accordingly.

Let's go !

Please erase all your previous code and type:

```
Input "Tell me your name";NAME$
Print "Hello ";NAME$
Input "How old are you?";age
days=age*365
Print "Wow! In days that's ";days
Bell
Print "I think you are old enough to be a Magician!"
Print "Goodbye ";NAME$
Bell 2
```

You basically knows all the instructions already. What is new is the **Input** instruction and the **calculation**.

Input is a great instruction to ask for something specific, it can be a number or text, for example **Input NAME\$**. Here we combine Input with a Print (it is possible with this instruction), so:

```
Input "Tell me your name" ; NAME$
```

- Will first display the request *Tell me your name*, followed by a question, and what you answer by typing on your keyboard will be stored in the variable NAME\$

Input "How old are you?" ; age

- Another question, and we Input the age. Say you typed in your age as 20, in the memory of the variable age the value 20 is then stored

days = age*365

This line will calculate the number of days in our example 20*365, so now in the memory of the variable days we have 7300.

You may have noticed that some variable names finish with a \$, like NAME\$, but some variable names don't, like age. There's a reason for this, and our tutorials will teach us how to create different type of variables and why. For now just note that when you set a variable with text you should add a \$ at the end of its name, but if it is a value then there is no \$. It's that simple.

We already know that a computer program is nothing more than a bunch of instructions that tell your machine what to do. If a computer only obeyed your list of instructions one after the other, your programs would be very limited and very boring, so we already know that the magic only begins when you teach your machine to start making decisions. These decisions are all based on simple conditions.

DECISIONS

The quick way to get a computer to make a decision is to learn it something and offer it a choice of what to do, depending on what it knows. If computers understood plain English we would say something like, "Hey computer, look out the window. If it's daytime then let's go to the gym. But if it's not daytime then let's go to bed." When you ask AOE to give this sort of choice to your computer, it will look at the choices on offer and decide if the condition is true or false. If it's true then your computer decides to take one course of action, but if it's false then another course of action will be taken.

One course of action could be to jump to a new place somewhere in your list of instructions, using for example what is called a PROCEDURE. But there are other solutions on offer.

One is to mark the place you want to jump to in the code before telling your computer's brain to jump to it and carry out whatever instruction is waiting there. With AOZ this is dead easy. You simply mark the place in your list of commands by giving it a name or a label, and tack on a colon character so your computer can recognize it and not confuse it with anything else. You can use any letter or number characters you like for this label, including the "underscore" _ character. So an example of your label could be **THE_GYM:** or **thegym** or **THEGYM:**

By using labels like this, you can now go to anywhere you like in your program using a special Goto command. Type in the following computer program. This example includes THEGYM label, the new command Goto, and the Wait command with every second you want the computer to wait. Normally you would have to wait 10 seconds to find out what time it is, but you can tell the computer to jump over that wait and go straight to the gym.

Print "I wonder what the time is"

Goto THEGYM

Wait 10

Print "bed time?"

THEGYM:

Print "It must be daytime!"

End

LOGIC

It's time to let your computer begin to think for itself using simple logic, and make a decision without any help from you. To do this you can use two commands that have the same meaning in the AOZ language as they do in the English language. These commands are **If** and **Then**. We told you we were going to see them again 😊.

So **If** something is true **Then** the computer will decide to take one action, otherwise it will do something **Else**.

Run this little routine and watch the computer decide what time it is:

```
NIGHT=1
NOW=1
Print "What time is it now?"
Wait 3
If NOW = NIGHT then Goto BED
If NOW <> NIGHT then Goto SOCCER
BED:
Print "I think it is bed time"
End
SOCCER:
Print "Where is the ball?"
```

Now change the value of **NIGHT** to another number, like **NIGHT=0**, and **RUN** that routine again.

If you are sure you understand how your computer reaches its decision **Then** *Goto* the next paragraph, or *Else* try again.

AOZ also uses the word **Else** when telling your computer how to decide if something is true or false, so you could change that last routine to something like this below, then change the values of **NIGHT** to see what happens:


```

NIGHT=0
NOW=1
Print "What time is it now?"
Wait 3
If NOW=NIGHT Then Goto BED else Goto SOCCER
BED:
Print "I think it is bed time"
End
SOCCER:
Print "Where is the ball?"

```

How about you create some more magic by writing a simple computer game to test out what you've learned so far? It's a genuine game of logic and the computer responds to anything that you throw at it. To do this you will have to use these mathematical symbols.

=	means "is equal to"
<>	means "is different than"
>	means "is greater than"
<	means "is less than"

This is a game of Guess The Secret Number, so we're going to use a new command to clear the screen, otherwise the number won't be secret at all. The instruction to clear your screen is **Cls**. One more thing, you already know **If...Then...Else**, but there is a different way of using the **If** without the **Then**.

Here we use **If** several times, and you have to type **End If** the appropriate number of times at the end of your logic tests, otherwise your computer will be left wondering what to do for ever.

OK, when you are ready type in this game and go and find a victim to inflict it on.

```
Print "Let's play Guess The Secret Number"
Print " Ask your victim to shut their eyes"
Wait 2
Input "Now type a secret number between 1 and 10 ";A
Cls
Print "Ask your victim to open their eyes"
Wait 2
```

SECRET:

```
Input "Victim, find the secret number ";B
If B=A
    Print "YES. CONGRATULATIONS!"
Else
    If B<A
        Print "WRONG! Go higher" : Goto SECRET
    Else
        If B>A
            Print "WRONG! Try lower" : Goto SECRET
        End If
    End if
End If
```

Do you see the : this separate instructions on a line, you can write the same think like this:

```
    If B<A
        Print "WRONG! Go higher"
        Goto SECRET
    Else
        If B>A
            Print "WRONG! Try lower"
            Goto SECRET
        End If
```

Optimization is important in programming. Let's change the SECRET: procedure part of the code, and try to understand why it still works. Once you understand then you're the Master of the lfs!

```
SECRET:
Input "VICTIM, FIND THE SECRET NUMBER";B
If B=A
    Print "YES! CONGRATULATIONS!"
Else
    If B<A
        Print "WRONG! Go higher" : Goto SECRET
    Else
        Print "WRONG! Try lower" : Goto SECRET
    End If
End If
```

Now let's try to achieve the same results using the previous more traditional **If...Then...Else** :

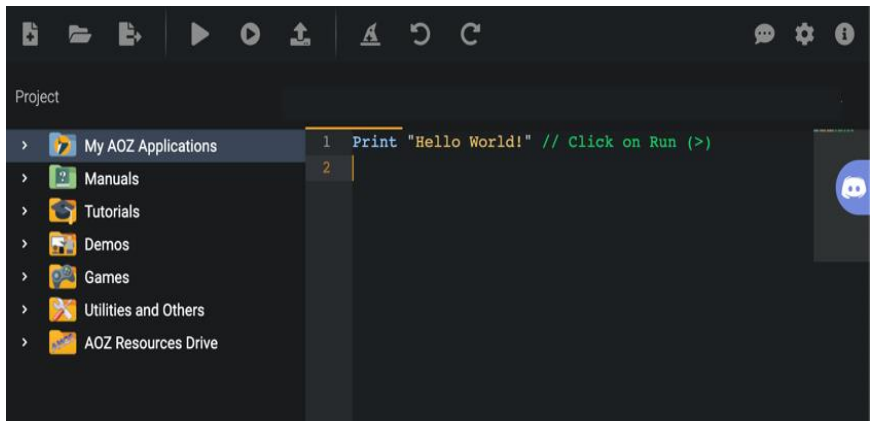
```
SECRET:
Input "VICTIM, FIND THE SECRET NUMBER";B
If B=A then Print "YES! CONGRATULATIONS!": End
If B<A then Print "WRONG! Go higher" else Print "WRONG! Try lower"
Goto SECRET
```

If you do not understand **Then** we have failed **Else** you are brilliant.



6. THE EDIT MODE

Finding your way around AOZ Studio has been made as simple as possible, with everything on one Edit Screen. The project folders are on the left, the main big working screen area is on the right and the Menu icons are at the top. Since the beginning of this User Guide we have been using this EDIT MODE and you are probably wondering what you can do with all these buttons.



The top menu buttons

Each of these provides a one-click action:



Create a new AOZ Application



Load an AOZ Application that already exists



Save an AOZ Application and export to share with others



Run this Application in a Web Browser



Run this Application in the AOZ Viewer



Publish this Application for the world to enjoy



Go into AOZ Direct Mode to test pieces of codes



Undo your last action



Redo your last action



Read your AOZ Messages



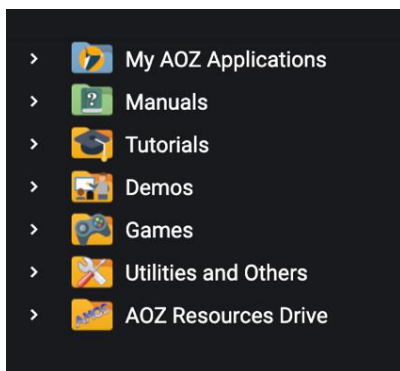
Change your User Settings



See your Documentation

The project folders

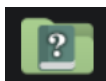
The Project Folders are all neatly stacked on the left side of the Edit Screen. Inside each folder is a whole host of sub-folders, and inside each sub-folder is a whole host of magic.



You can move, delete, rename and copy. Let's look at each folder in turn.



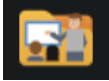
All your Applications are kept in this folder by default, with whatever names you have given them. Each Application folder can contain a load of other folders for Resources like images, icons, music and sound effects.



The Reference Manuals folder is where you can find this User guide, and old hands will find familiar items.



Tutorials are kept in this folder. They include a whole series of lessons about AOZ Studio along with learning via a series of classic entertainments.



The Demos folder contains classic games, special effects,... for you to take apart, adapt and put back together again as you like.



Games is where you will find a load of familiar and not so familiar ready-mades to enjoy, adapt and do what you want with.



Accessories/Utilities is the folder for useful links and resources like a font viewer, joystick and scancode testers.

INTERACTIVE MANUAL AND HELPS

The Interactive Manual is behind the **i** Documentation Icon on the top right of the screen, or when you hit the **F6** Key in your code (click on an instruction and press **F6**), which is very useful.

Try also to click on an instruction and press **F5**

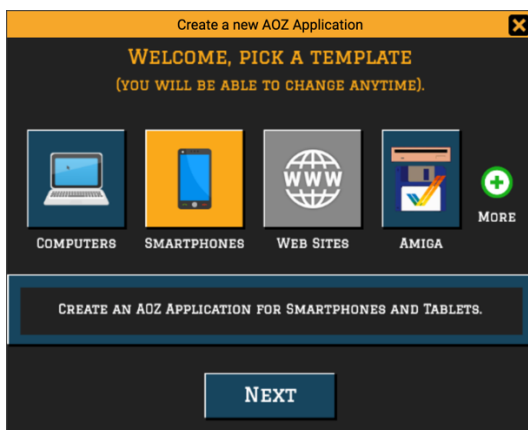


7. CREATE A PROJECT APPLICATION

Getting ready

This is where we get everything set up for you to create a new project. Click on the first icon in the row at the top of your screen, which says "Create a new AOZ Application", and a selection box appears with a choice of templates for your new project. There are ready-made templates that make sure it works for computer screens, smartphones, web sites, historic Amiga formats and more, so simply select the one you need.

Let's suppose you want your new AOZ Application to run as a project on smartphones. Make sure you select the SMARTPHONES option then hit NEXT.



You can now fill in your details, select which folder you want to keep your project in and give your project its own graphic icon if you like. Hit the NEXT button when you're ready to move on.

Create a new Aoz Application

APPLICATION PARAMETERS

TITLE

VERSION

COPYRIGHT

AUTHOR

FOLDER & ICON

FOLDER

ICON FILE

The right fit

Aoz Studio will help you as much as possible to make sure your project's application is the perfect fit for the device you want to use it on. You can choose the ORIENTATION which is the way you'd like your work to fit best on your smartphone's little screen - portrait, or landscape, or both. And you can select a TARGET device by brand, model and screen resolution.

Create a new Aoz Application

DISPLAY PARAMETERS

ORIENTATION

TARGET

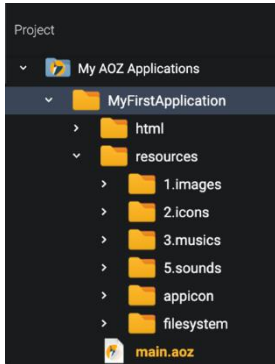
- ☒ 720p (1280x720)
- ☐ 1080p (1920x1080)
- ☐ 2K (2560 x 1440)
- ☐ 4K Ultra HD (3840 x 2160)
- ☐ True 4K (4096x2160)

SCREEN PARAMETERS

- ☐ ALLOWS FULL SCREEN
- ☐ ALLOWS FULL SCREEN
- ☐ SHOW THE FULL SCREEN

Your applications folders

When you create a new application it is given its own folder where everything gets stored for safekeeping and quick access.



By default, this folder is put in the "My AOZ Applications" folder on your computer, along with related sub-folders for all the graphic images, icons, audio, video, and data files.

As always, the application is fired up by clicking on main.aoz ("main" can be changed by the name you gave to your program)

At this point in your User Guide, we're going to start letting go of the steering wheel and get ready to hand over the guidance to you. Before we do that, we'll need to say a word or two about programming errors and bugs, and how we've done everything we can to help you deal with them. So let's move on to the next chapter.



Lucie says Hi!

8. ERRORS, BUGS AND HELP

Most programmers refer to the mistakes and glitches in their programs as "bugs". These little devils are the errors that are responsible for messing up the magic. They could be a single keystroke that has been typed in by mistake, or an instruction that's been left out, or some impossible task that blows the computer's mind. Sometimes it can take hours or even days to spot where these bugs are lurking and what they are doing before they can be put right, but AOZ is smarter than that and does most of the hard work for you.

If a you make a mistake when you write a program in AOZ, or when you ask your computer to do the impossible, AOZ will always do its very best to offer some first aid, and it will offer it automatically. It will not only help you spot the error, but will also try to explain what the problem is, and exactly where it is lurking, no matter how big or complex your program is.

If this happens while you are programming, then you'll be able to try and cure the bug immediately. If it happens when you test or run your program then AOZ will take you straight to any offending lines of code, one after the other, until they are perfect.

Apart from simple typing mistakes, some of the most common mistakes are in the way you type in the variables. A variable is a quantity or a number with an unknown value, where the value can change depending on what's is saved in it. It's important to remember that when you name these variables in AOZ they are case sensitive, so a name in all lower-case letters is a different variable, with a different value from the same name with a capital letter. For example:

<code>stuff\$="stuff"</code>	<code>STUFF\$="more different stuff"</code>
<code>Stuff\$="different stuff"</code>	<code>sTuFf\$="even more different stuff"</code>

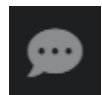
On help

AOZ will try and help you every step of the way. When you started to create your magic by typing in lines of code, you probably noticed your code appeared in different colors. Keywords, and comments and variables and twiddly bits all appear in their own distinct color. If something isn't quite right then AOZ will try to highlight the problem by changing color. Similarly, as you type in the first few characters of a command, AOZ will try and help save you time and avoid mistakes by showing you a list of the commands and keywords it reckons you may want to use. All you have to do is use your Up and Down arrow-keys to highlight the item you're interested in, and AOZ will automatically tell you about it, and help you put your commands in the right order and the right format. **You also may click on the instructions and type the F5 or F6 keys.**

Apart from coming round to your house and programming for you, what more can we do? Well, we can do quite a lot more as it turns out. For example an understandable User Guide. Please read on.

On error Help

If you make a mistake and there are errors that stop your program working properly, then the errors will appear on their own wall of shame in your editor window at the bottom, usually with a helpful message showing the exact line number and specific character where the bug is sitting. You may want to show these errors again so click on this button at the top of the screen or type Ctrl+Shift+c.



There are even more debugging options if you're writing your magic words in the built-in AOZ viewer by pressing the **F2** key and the magical hat at the top left of it (see next Chapter), so never be afraid to experiment and try out new ideas and

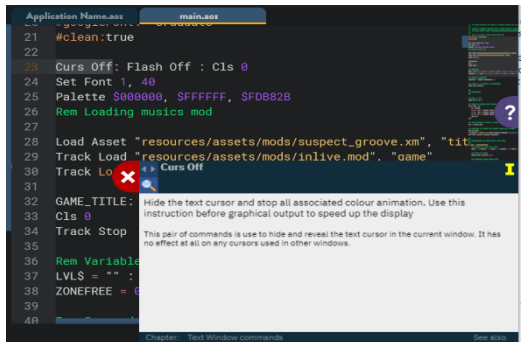
routines. AOZ is always there to help, and never to hinder. The contextual help is great:

Contextual helps F5 & F6

As you learn AOZ language you need to know what instructions are available, and there are so many (almost 750), you need to be helped, we all do; here comes the great contextual Help and Manual

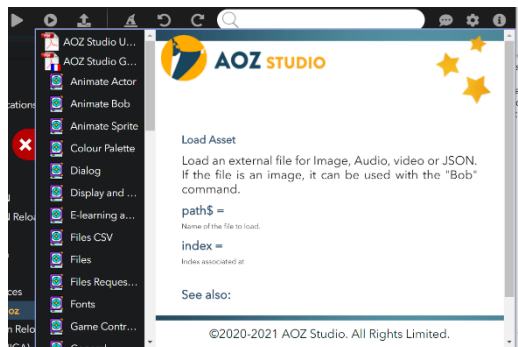
HELP (F5): If you click on an instruction then press the F5 function key, a popup will appear and gives you all details as well as the ones related to it. See also at the bottom the links

Close it with the big Cross in the Red Tab.



MANUAL (F6): If you select an instruction and press the F6 function key the manual search will run and tell you all what he knows about it.

Close it with the big Cross in the Red Tab.



Direct Mode help

And... you have the Magical Hat DIRECT MODE to check all sorts of things, like the value of your program's variables, we will see it soon in the next chapter.

Speeding up

There are a lot of ways to speed up a program. For example you can avoid using large graphics if you don't really need them. Anyway, often small is beautiful. You should also keep your audio and video files as small as possible.

Some instructions go faster than others, choosing the organization, the sequence and the instructions will have an impact which can be very important. This is called algorithmic.

For apprentice magicians who want to reduce the speed of their programs you can tell your computer to **Wait** for however long you like. Each full second you want to wait is represented by 1 unit, so the command **Wait 1** would slow the action down by a second, whereas **Wait 10** would grind things to a halt for 10 seconds. **Wait 0.5**





We are about to re-enter the magical realm where we make objects on a screen perform amazing, unfeasible, brilliant, scary, baffling and downright bonkers acts.

Yes, let's continue with the Actor Instruction we saw earlier and we will also have an overview of other AOZ movements and animations using the famous Bob instruction.

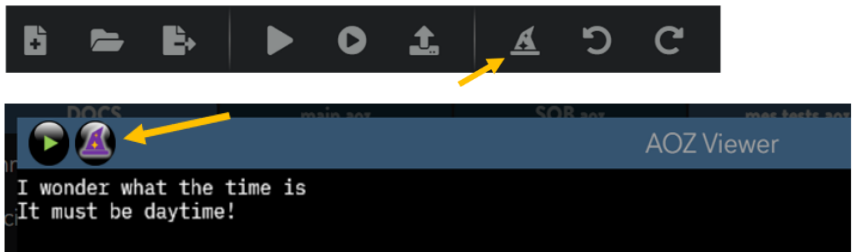
But before there is an other important help that needs it's own chapter: the famous Direct Mode.

9. THE DIRECT MODE

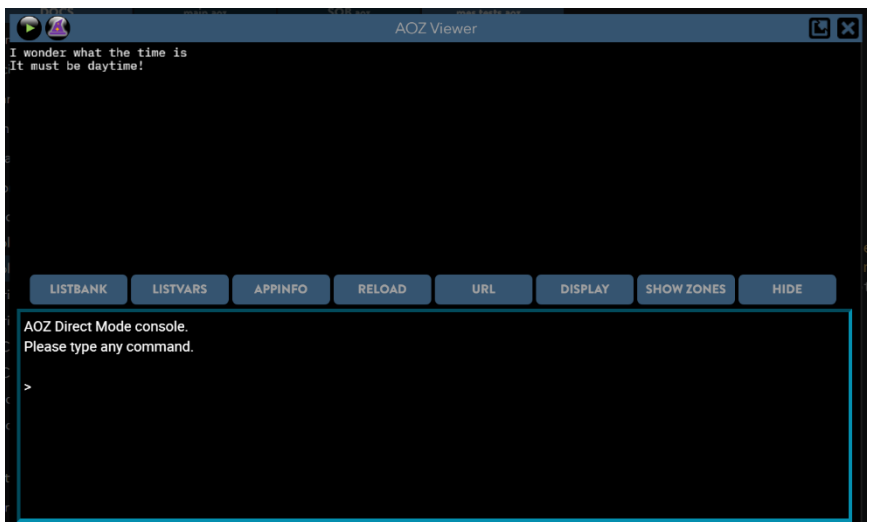
Up to now we have been using the Big screen and editing our code in the so-called EDIT MODE.

AOZ Studio also has a Direct Mode, and this is the place to test anything as you go along.

To switch from Edit to Direct mode, trigger the Magician Hat button from the top of the Menu of the Edit Mode, or on the AOZ viewer :



You will then enter the Direct Mode :



You will still see your code output in the main window but now have 2 added features:

- The blue buttons and
- the direct mode console.

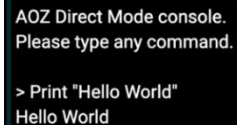
Let's try, type any program and RUN (F2) to view it in the AOZ Viewer (as you did until now), then click on the magical hat to enter the Direct Mode from the AOZ Viewer.

You will see your program, OK fine, but also the blue buttons, they are quite easy to understand, just try out,

and below the AOZ "Console".

Type your instruction in the console, like this:

Print "hello"

A screenshot of a dark-themed console window. The text inside reads: "AOZ Direct Mode console. Please type any command." followed by a prompt "> Print 'Hello World'" and the output "Hello World".

```
AOZ Direct Mode console.  
Please type any command.  
  
> Print "Hello World"  
Hello World
```

What? Yes I believe the Blue buttons of the Direct Mode are self-explanatory, and if not you are very welcome to send a complaint to [givemeabreak!@kjh0\\$£%.com](mailto:givemeabreak!@kjh0$£%.com).



10. GOING FURTHER WITH ACTOR

Please don't read this chapter before you have read Chapter 9. We appreciate your eagerness, but first things first. We have given this chapter a clear title for a reason!

THE VARIOUS POSSIBLE CONTROLS

Note: this parameter and chapter is work in progress

We saw that it was quite easy to control an actor with the **Control\$** parameter. It is possible to assign multiple controls to an actor (e.g., keyboard and joystick).

As a reminder, a control is always defined as:

<name of the control> : property1 = value1 ; property2 = value2...

The properties must be separated by a semicolon (;)

Here is a description of the different types of controls available with the Actor instruction.

Keyboard

Example: **Control\$="ArrowLeft: angle=4"**. Here, by pressing the left key the actor will rotate with a 4 degree angle change.

Here's how to sum up the **Control\$**'s properties:

- **offsetX**: value for horizontal movement of the actor.
- **offsetY**: value for the vertical movement of the actor.
- **angle**: value for the rotation of the actor
- **hrev**: true/false. Whether or not the horizontal flipping (mirror effect) of the actor is activated
- **vrev**: true/false. Whether or not the actor's vertical flipping (mirror effect) is activated

- **Image:** image to be used for the actor when typing this key on the keyboard
- **anim:** Name of the animation to play for the actor when typing this key. The Actor's **Actorsheet\$** parameter must be set.
- **forward:** numerical value for moving forward. The movement takes place according to the angle of rotation of the actor.
- **backward:** a numerical value for moving backward. The movement is done at the opposite angle of rotation of the actor.

Example:

Actor "magic", X=Screen Width/2, Y=Screen Height/2, Image\$="magic.png",
Control\$="ArrowUp: angle=-8; ArrowDown: angle=8; ArrowRight:
forward=10; ArrowLeft: backward=10"

Actor "lucie", X=Screen Width/2, Y=Screen Height-230, Image\$="lucie.png",
Control\$="ArrowRight: OffsetX=16; ArrowLeft: OffsetX=-16"

Your turn 😊 :

- Replace the backward property by forward but still moving backward (think -)
- Try the different Control\$ parameters

Joystick

As we saw in one of the early examples in Chapter 9: Move the actor with the Joystick, we can associate a joystick with our actor to control it. Many joysticks can be connected to the computer, and they are numbered by their login order. The first joystick will be the 0 (zero), the second will be the 1. Yes we know it doesn't make sense, but it comes from the origin of Personal Computers, in an age when people were weird.

Try those sample code (with a Joystick):

Actor "magician", Image\$="magic.png", Control\$="Joystick"

And this sample one:

Actor "magician", Control\$ = "JoyLeft0: angle=-4; JoyRight0: angle=4; JoyUp0: forward=4; JoyDown0: backward=4", Image\$="magic.png"

Note: if your joystick is well connected to your PC but is not recognize, close and relaunch AOZ Studio. If this is still not working check if it is recognize by your computer (for ex playing a game).

Mouse

Now let's see the final way to control an actor, using the mouse. This is the easiest control to define, because there is only one property for the instruction **Control\$ = "Mouse"**.

Try the following code:

Actor "magician", Control\$="Mouse", image\$="magic.png"

Our actor is positioned under the mouse pointer. And if you move the mouse on the screen, the actor moves with it. You can use this method to replace your mouse pointer with a custom image.

In some games, it may be necessary to move the actor only in the horizontal direction (like a Breakout racket) or in the vertical direction. No problem, AOZ has it all planned! Let's change the previous code a bit:

Actor "magician", Control\$="Mouse: honly=true",
Image\$="magic.png"

Note: you might be wondering why it's not honly\$ or true is not "true", you think hehe they make it wrong at AOZ Studio. That's it I'm the boss.

In fact for simplicity we use Aliases, true is 1 and false is 0, they are numeric values. It can be argued but we did this to make your life easier, so who's the boss?

Run the program. The actor now follows the friendly mouse pointer on the vertical axis only.

The Mouse has 2 properties:

- **honly**: True/False. Turns tracking the mouse's horizontal position only on or off.
- **vonly**: True/False. Turns tracking the mouse's vertical position only on or off.

So in this example the vertical is False (by default the property is set as False).

This is interesting for ex for a breakout game where the actor should only move left-right at the bottom of the screen.

Automatic

Ground control to Major Tom: your actor can be on autopilot. By setting the **Auto\$** parameter, all properties defined (placed immediately after the **Auto\$=**) will be automatically used without the need to press a button, waggle the joystick or move the mouse.

Look at this code:

Actor "magician", X=0, Y=150, Auto\$="offsetX=4, offsetY=4",
Image\$="magic.png"

Run the program to see that your actor is moving alone to the right of the screen.

This Control\$ thing is not super-easy, but once you understand, it's pure magic! Take a break and practice a bit.

It's your move 😊:

- Understand the syntax, the use of the "" with Control\$, try changing properties and values.



COLLISIONS

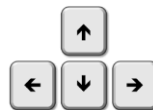
According to the movies, when a spaceship hits another ship it goes boom, even though space is silent. To know when things should go boom you have to detect the collision between the two actors. Fortunately it's dead easy to detect collisions with AOZ. There even 2 ways of doing this.

Collisions with Actor Col

Returns TRUE (or 1) if the actors are colliding. If no collision is detected, the function returns FALSE (or 0).

Let's type this new code to test if our magician encounters lucie, AOZ most famous actor(ress) and by setting up the collision detection:

```
Actor "lucie", 600,150, Image$="lucie.png",Control$="keyboard:  
OffsetX=12, offsetY=12"  
Actor "magic", 960,540, image$="magic.png"  
Do  
  A = Actor Col ("lucie", "magic" )  
  If A = True Then Boom  
  Wait Key  
Loop
```



Run the program. Move your actor with the keyboard and collide... Boom, love at first sight!

This program is quite clear I believe:

Line 4: The properties passed to Actor Col are 2 indexes. Indexes are a special form of variable. Here, as we use Actor with its name and not with a number so we have "".

Collision precision

With AOZ the collisions work to the nearest pixel ("pixel perfect") and not when the boxes surrounding the actors touch each other, so it's much more precise.



On the other hand, depending on the power of the machine which executes your code, this can slow down the display. So we made an instruction to tune the collision accuracy:

Actor Col Precision <delay>, <precision>

delay: Numeric value indicating the waiting time between 2 collision tests. By default its value is 300 milliseconds.

precision: Numerical value indicating the fineness of collision detection between 1 and X. 1 being the most precise precision. 2 is set by default. So if you want the display to be faster increase delay and precision.

There is also a way to set the collision area per Actor, with Rigidbody:

```
Actor "lucie",600,150, Image$="lucie.png", Control$="keyboard:  
OffsetX=12, offsetY=12", Hotspot$="middle",  
Rigidbody$="circle:radius=50"
```

```
Actor "magic", 960,540, image$="magic.png", Hotspot$="middle",  
Rigidbody$="circle:radius=50"
```

Do

```
  A = Actor Col("lucie", "magic" )
```

```
  If A = True Then Boom
```

```
  Wait Vbl
```

Loop

Note: I am sure you noticed that it works without specifying X=600, Y=150 as per above: Actor "lucie",600,150,

I told you at AOZ Studio we try to make your life easier.

Collision with the Actor instruction

We have just seen how easy it is to test a collision between two actor(s) with the function **Actor Col.**

The Actor instruction also allows you to test a collision using the **OnCollision\$** parameter. It is a special parameter, used as a "Listener", that is intended to monitor an event (in this case a collision) and alert the program. We'll come back to this Listener system in the next chapter.

Let's change our program as follows:

```
Actor "magician", Image$="magic.png", OnCollision$="COLLISION"  
Actor "magician", Control$ = "ArrowRight: offsetX=4; ArrowLeft: offsetX=-4;  
ArrowUp: offsetY=-4; ArrowDown: offsetY=4"  
Actor "lucie", 350, 350, Image$="lucie.png"  
Do  
  Wait Vbl  
Loop  
Procedure COLLISION [INDEX2$]  
If INDEX2$ = "lucie" Then Boom : End  
End Proc
```

Run the program. Move the actor using the arrow keys, collide with lucie and... Boom!

Let's break down the program.

Line 1: **OnCollision\$** has been added to the **Actor** instruction. Its property is a string of characters, containing the name of the procedure: **OnCollision\$="COLLISION"**

Note: A procedure is an independent program block, that here will be called when a collision occurs.

Line 3: We add our second «lucie.png» actor at position 350,350.

Lines 4 to 6: A simple loop to keep the program active. The Wait Vbl instruction as already said makes the display more fluid. (you can try without it)

Line 7: A procedure, here called **COLLISION** is created. To do this you have to put the **Procedure** statement followed between brackets by the parameter(s) to communicate to the procedure (here **INDEX\$2** -in capital letters-, whose name is not chosen at random, as we will see a little later), and then finish with **End Proc**.

So the syntax is as follows:

Procedure COLLISION[INDEX2\$]

....

... the instructions of your procedure

....

End Proc

When you put the name of your procedure anywhere in your program, it jumps into the quantum hyperspace and executes all the code of the procedure, and when it arrives to **End Proc** the execution of the program returns just after the call of the procedure.

We will come back to the procedures that allow programs to be effectively divided into independent parts, and show you how extremely useful this can be.

Note: another mean to divide your program in different parts is to use **INCLUDE**, we will discuss it later.

In the last program the procedure tests the variable named **INDEX2\$** which will contain the name of the colliding actor. Remember the name of the procedure is set in the **OnCollision\$** parameter (line 1).

When the "magician" actor collides with another actor the procedure **COLLISION** will be called and the name of the colliding actor will be passed to the variable **INDEX\$2**.

Line 8: We test whether **INDEX\$2** is the other actor: «lucie.png», if this is the case, a terrible bomb sound effect is emitted and the program stops

Line 9: End of the procedure, which means we return right after the call of the procedure.



Love collision at first sight

Mice and Actors

We have just seen that it is possible to control an Actor with the gamepad, keyboard or mouse, in order to move it around the screen.

We can also make our Actor sensitive to mouse actions (or to the touch screen it is seen by AOZ as the same to be compatible). It will be able to react when we click on it (but slowly we just said that it is sensitive), or when the mouse pointer passes over it.

OnMouse\$

For that, we are going to use the **OnMouse\$** parameter of the Actor instruction, it will save you a lot of time!

Let's see a simple example:

```
Actor "magic", X = 100, Y = 100, Image $ = "magic.png", OnMouse $  
= "CLICK_MAGIC"
```

Wait key

```
Procedure CLICK_MAGIC [EVENT $]  
  If EVENT $ = "mouseclick" Then Print "Click!"  
End Proc
```

Our first line shows the Actor "magic" at the coordinates 100,100. The **OnMouse\$** parameter indicates the name of the AOZ procedure to be called for each mouse action on this Actor. The AOZ procedure that will be called here is **CLICK_MAGIC**.

This procedure, as we will see, can retrieve a certain amount of information. In this example the **CLICK_MAGIC** procedure retrieves the **EVENT\$** information, which contains the type of action produced by the mouse on the Actor.

The content of our procedure displays the word "Click!" "If the value of **EVENT\$** is" mouseclick "(corresponding to a mouse click.)

EVENT\$ can take the following values:

- "**mouseclick**": when the user clicks on the Actor
- "**mousedown**": when the user keeps a mouse button pressed
- "**mouseup**": when the user lifts a mouse button
- "**mousemove**": when the user hovers the mouse pointer over the Actor.
- "**dragdrop**": when the user moves the Actor using the mouse

Now let's modify our **CLICK_MAGIC** procedure as follows:

```
Procedure CLICK_MAGIC [EVENT $, BUTTON, INDEX $, X, Y]
  Locate 1,1: Print "Actor:" + INDEX $ + "EVENT $:" + EVENT $ +
  "BUTTON:" + Str $ (BUTTON) + "X:" + Str $ (X) + "Y:" + Str $ (Y)
End Proc
```

When you operate "magic" with the mouse, information is displayed at the top of the screen.

The **CLICK_MAGIC** procedure offers several properties:

- **BUTTON**: is an integer corresponding to the mouse button that is pressed:
 - o 0: No button pressed
 - o 1: The left button
 - o 2: The right button
 - o 3: The central button or wheel (if there is one)
- **INDEX\$**: returns the name of the Actor concerned by the mouse action. If you also want to capture Actors identified by a number, you can use **INDEX**.
- **X and Y**: return the position of the mouse relative to the interior of our Actor.

Drag & Drop

Let's modify our procedure again:

```
Procedure CLICK_MAGIC [EVENT $, INDEX $, DRAGX, DRAGY]
  If EVENT $ = "dragdrop" Then Actor INDEX $, X = DRAGX, Y = DRAGY
End Proc
```

Here we test the type of action returned by **EVENT \$**. If the action is "dragdrop", the user has placed their mouse over the Actor, hold down a mouse button, and move.

The coordinates of the displacement are contained in 2 other information: **DRAGX** and **DRAGY**.

Our procedure, for each call, then places our Actor at these new coordinates.

You can for example make a window and move it in a single instruction.

11. THE LISTENER AND EVENT SYSTEM

Once the **Actor** Instruction is called, our actor can be "monitored" constantly by several Listeners. They check and scrutinize the activity of our actor and alert the program that a possible event has occurred.

An event can be:

- A collision
- A change of position
- A change of image
- A Key press

The name of a Listener always starts with **On...** and this type of parameter waits for a string containing the name of the AOZ procedure to call.

Thus, in our previous program, when we defined the **OnCollision\$** we asked the collision Listener to notify our program when a collision occurs, and to call our procedure by giving it some information (or properties). On this case it was the name of the actor that was hit.

Note that your AOZ procedure can exploit a few of these properties, you don't have to set them all.

We have several Listener each with a set of properties -to be written in capital- for the Actor instruction:

ONCOLLISION\$

If the actor collides with another actor, then this Listener calls the defined AOZ procedure by passing it the following properties if required:

- **EVENT\$:** Event name (always "**on_collision**").
- **INDEX1:** Value of the monitored actor index.
- **INDEX1\$:** The name of the supervised actor.
- **INDEX2:** Value of the collided actor index.
- **INDEX2\$:** The name of the actor we collided with
- **IMAGE:** Value of the image's index of the collided actor.
- **IMAGE\$:** The name of image of the collided actor.

ONCONTROL\$

Returns the values of the **Control\$** activated for the actor. Ex if **Control\$="keyboard"** this will give the values related to the keyboard. "mouse" for the mouse, ...

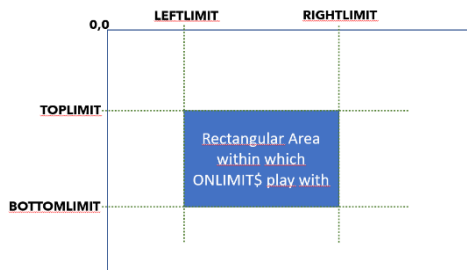
- **EVENT\$:** Event name (always "**on_control**").
- **CONTROL\$:** Name of the instruction of this event.
- **INDEX:** Value of the monitored actor index.
- **INDEX\$:** Name corresponding to the monitored actor.
- **BUTTON:** Index of the mouse button that was activated.
- **ALTKEY:** Returns True or False if the Alt key has been pressed or not.
- **CTRLKEY:** Returns True or False if the Ctrl key has been pressed or not.
- **SHIFTKEY:** Returns True or False if the Shift key has been pressed or not.
- **METAKEY:** Returns True or False if the "Windows" or "CMD" (Mac) button has been pressed or not.

- **MOUSEX**: Numerical value corresponding to the horizontal position of the mouse.
- **MOUSEY**: Numerical value corresponding to the vertical position of the mouse.

ONLIMIT\$

Moving an actor can be limited to a rectangular area of the screen, for example to prevent it from falling off the screen. When this area is delineated, the procedure defined is called when the actor hits one of the edges of the area, passing it the following properties:

- **EVENT\$**: Event name (always "on_limit").
- **INDEX**: Value of the monitored actor index.
- **INDEX\$**: Name corresponding to the monitored actor.
- **LIMIT\$**: The name of the edge reached ("left", "right", "top", "bottom")
- **LEFTLIMIT**: X coordinate
- **RIGHTLIMIT**: X coordinate
- **TOPLIMIT**: Y coordinate
- **TOPLIMIT**: Y coordinate
- **BOTTOMLIMIT**: Y coordinate
- **BOTTOMLIMIT**: Y coordinate



ONANIMCHANGES\$

When the actor is animated (when the **Spritesheet\$** setting has been entered), **OnAnimChange\$** calls the defined AOZ procedure by passing it the following properties:

- **EVENT\$**: Event name, "change" or "complete." If the name is "complete," it's because the animation is over.
- **INDEX**: Value of the monitored actor index.
- **INDEX\$**: Name corresponding to the monitored actor.

- **ANIM\$:** The name of the current animation.
- **FRAME:** The image number currently being played for the current animation.
- **TOTALFRAMES:** The total number of images for the current animation.



ONCHANGE\$

This listener calls the AOZ procedure defined, each time the actor undergoes a change in position, transparency, size or rotation, passing the following properties:

- **EVENT\$:** Event name: "change" or "complete." If the name is "complete," it's because the actor's move is complete.
- **INDEX:** Value of the monitored actor index.
- **INDEX\$:** Name corresponding to the monitored actor.
- **X:** Position X of the actor.
- **Y:** Position Y of the actor.
- **ALPHA:** Actor's opacity.
- **SCALEX:** Expansion/horizontal reduction rate
- **SCALEY:** Vertical expansion/reduction rate
- **ANGLE:** Actor rotation angle
- **HREV:** True if the actor is turned horizontally, False if not.
- **VREV:** True if the actor is returned vertically, False if not.
- **SKEWX:** Horizontal distortion of the actor
- **SKEWY:** Vertical distortion of the actor

Of course you don't have to listen out for every actor, you are free to monitor only those you reckon are important to watch.

HOW TO USE THE LISTENERS

Here is an example to use the Listener for a scrolling attached to the keyboard right-left keys:

```
Actor 1, Image$="bg.png", Y=0, Control$="ArrowLeft: offsetX=5;  
ArrowRight: offsetX=-5", OnChange$="ON_CHANGE"  
Do  
  Wait Vbl  
Loop  
Procedure ON_CHANGE [INDEX$,X]  
  If X > 0 Then Actor INDEX$, X = -1980  
  If X < -1980 Then Actor INDEX$, X = 0  
End Proc
```

You see with that example that:

- The listener **OnChange\$** is calling the procedure "ON_CHANGE" each time the actor undergoes a change in position (due to a Key press)
- The **Onchange\$** calls the procedure giving along the properties, automatically, here **INDEX\$** and **X**

So when actor 1 moved on the screen the procedure is called with the actor name in INDEX\$ and its X position, as properties that the code in the procedure may use (here to check if we reach the limit of 1980 pixels (remember it is half the size of the graphics) and to restart if we did..

MORE CONTROLS OVER ACTOR

Reset Actor

Reset all the actor properties. Example :

Reset Actor "magic"

Reset Actor 1

Del Actor

Delete an actor (so it will not be displayed). Example :

Cls 0

Actor 1, X=100, Y=50, image\$="magic"

Wait 2

Del Actor 1

Visible Parameter

To Display on/off the actor (Visible=True or False)

Actor "magic", X=100, Y=100, Image\$="magic.png"

Wait Key : Actor "magic", Visible=False

Wait Key : Actor "magic", Visible=True

Enable Parameter

If Enable=False (Enable=True or False) even if the Actor have Control\$ and animations the image stays displayed but all behaviors are frozen. Very useful to do a clickable button that you want to enable or disable on your user interface.

Actor "magic", X=100, Y=100, Image\$="magic.png",

Control\$="keyboard", Enable=False

Do : Wait Vbl : Loop

12. ANIMATION

It's very nice to display images but it's almost certain that you'll want to animate them.

In a game, heroes and enemies are rarely immobile, so let's see how to animate with a well-known technique of the video game known as a sprite sheet.

WHAT'S A SPRITE SHEET?

A sprite sheet is a large image containing a sequence of images that deceive the eye when they are displayed one after the other, and give the illusion of animated movement.

Here's an example:



The Actor instruction offers several parameters for using a spritesheet:

- **Spritesheet\$**: Is the name of the sprite sheet to use.
- **Anim\$**: Is the name of the animation to play.

How do we animate our actor?

This paragraph is not finished, indeed, more to come...



13. ACTOR SETTINGS

To sum up everything we've discussed and more, here's the full list of the various settings available with the Actor statement at the time of writing this manual.

- **X:** The actor's horizontal position on the screen. 0 by default
- **Y:** Vertical position of the actor on the screen. 0 by default.
- **StartX:** The start horizontal position of the actor's move.
- **StartY:** The start vertical position of the actor's move.
- **EndX:** The end horizontal position of the actor's move.
- **EndY:** The vertical position at the end of the actor's move.
- **Duration:** The duration of the actor's movement between 2 points.
- **Image\$:** The number or name of the image in the image to be used for the actor.
- **OnChange\$:** To assign the name of the procedure to be called when the actor undergoes a change (movement, form, transparency...)
- **Transition\$:** To assign the name of the effect to be given to the move of the actor. By default it is the "linear" transition.
- **OnMouse\$:** To create actions on mouse behaviors

AOZ is using the beautiful CREATE.JS API, it is a great product, for the **Transition\$** effects. The ones available can be seen by visiting this page:

https://www.createjs.com/demos/tweenjs/tween_sparktable

Ease Equations

backIn, backInOut, backOut,
bounceIn, bounceInOut,
bounceOut, circIn, circInOut,
circOut, cubicIn, cubicInOut,
cubicOut, elasticIn, elasticInOut,
elasticOut, linear/none, quadIn,
quadInOut, quadOut, quartIn,
quartInOut, quartOut, quintIn,
quintInOut, quintOut, sineIn,
sineInOut, sineOut,

Custom Eases

getBackIn, getBackInOut,
getBackOut, getElasticIn,
getElasticInOut, getElasticOut,
getPowIn, getPowInOut,
getPowOut

- **Scale:** The decimal value of the actor size scale. Default 1.0 (normal size)
- **StartScale:** The start size decimal value for moving the actor.
- **EndScale:** The end size decimal value for actor removal.
- **ScaleX:** The decimal value of the width scale of the actor. Default 1.0 (normal size)

- **StartScaleX:** The decimal value of starting width for moving the actor.
- **EndScaleX:** The decimal value of end width for the movement of the actor.
- **ScaleY:** The decimal value of the actor's height scale. Default 1.0 (normal size)
- **StartScaleY:** The decimal value of the start height for moving the actor.
- **EndScaleY:** The decimal value of the end height for the movement of the actor.
- **Alpha:** The decimal value of the opacity of the actor. Default 1.0 (totally visible).
- **StartAlpha:** The decimal value of the initial opacity for the movement of the actor.
- **EndAlpha:** The decimal value of the final opacity for the movement of the actor.
- **SkewX:** The digital value of horizontal distortion of the actor.
- **StartSkewX:** The digital value of the start horizontal distortion of the actor's movement.
- **EndSkewX:** The digital value of the end horizontal distortion of the actor's movement.
- **Skew Y:** The digital value of vertical distortion of the actor.
- **StartSkewY:** The digital value of the start vertical distortion of the actor's movement.
- **EndSkewY:** The digital value of the end vertical distortion value of the actor's movement.
- **Visible:** True/False Shows or hides the actor.
- **Spritesheet\$:** The name of sprite sheet to use for actor animation

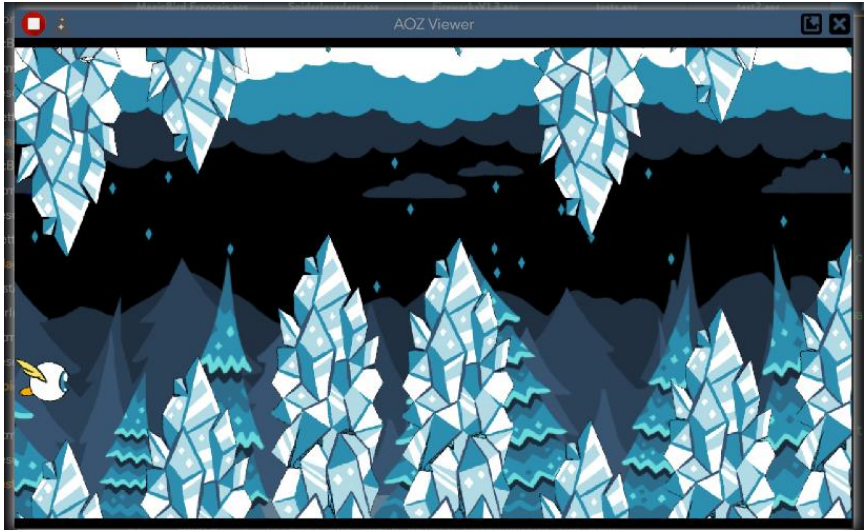
- **Anim\$:** The name of the animation to play (available with the sprites sheet)
- **LoopAnim:** True/False play a loop animation or not.
- **Hotspot\$:** Digital value that determines the position of the actor's "hot spot.". (For it's value see the Hotspot Chapter.)
- **HotspotX:** Digital value that determines the horizontal position of the actor's "hot spot"
- **HotspotY:** Digital value that determines the vertical position of the actor's "hot spot."
- **OnAnimChange\$:** Name of the procedure called for each change of the actor animation
- **OnCollision\$:** Name of the procedure called when the actor collides with another.
- **LoopMove:** True/False play a loop move.
- **ActionMove\$:** "play" or "pause" the mouvement
- **Control\$:** To direct the actor ("keyboard", "mouse", "joystick").
- **OnControl\$:** To assign the name of the procedure to be called to control the movement when control\$ is activated.
- **HRev:** True/False horizontal mirror of the actor
- **VRev:** True/False vertical mirror of the actor
- **LeftLimit:** Digital value defining the limit of the left edge of the actor
- **RightLimit:** Digital value defining the limit of the right edge of the actor
- **TopLimit:** Digital value defining the limit of the top edge of the actor
- **BottomLimit:** Digital value defining the limit of the bottom edge of the actor

- **OnLimit\$**: Name of the procedure called when the actor reaches one of the edges.
- **LookAt\$**: Object, actor or point of the screen that the actor must look towards.
- **Gravity**: Digital value defining the force of gravity attracting the actor. Default 0 (no gravity).



14. LET'S MAKE A GAME

You already know enough to make a game: Magic Bird it is.



The object of the game is to make the Magic bird fly by avoiding the stalactites. To play, a click on the mouse button (or press on the smartphone screen) allows Magic to fly away.

Without wasting any more time let's program it. You will find the corresponding code in the Tutorials folder, code that you can copy / paste for the more eager. For others let's learn step by step.

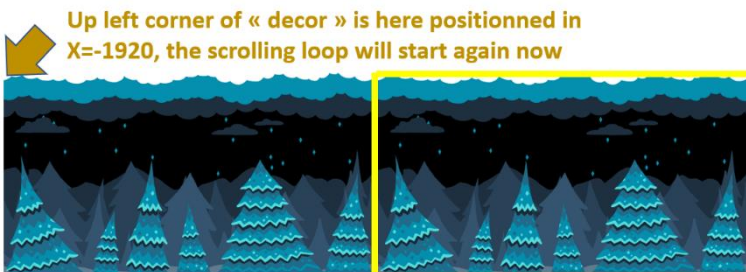
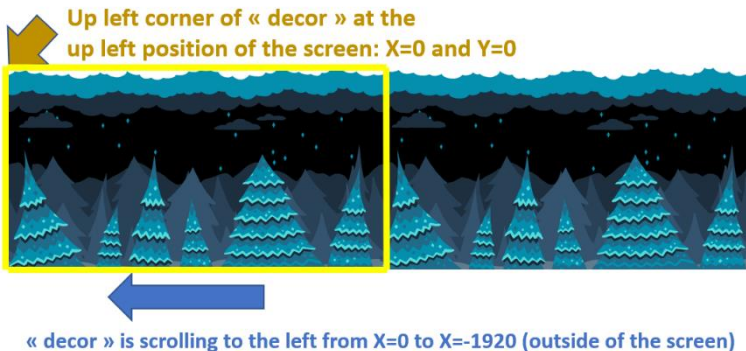
So, without commanding yourself, type this line of code (it's a single line):

```
Actor "decor", Image$="bg.png", X=0, Y=0, EndX=-1920,  
LoopMove=True, Duration=20000
```

If you have followed the previous chapters nothing here should surprise you.

By using the Actor instruction we therefore create an actor which is called here "decor" a nice name for background, and which uses the image which is called "bg.png". The aforementioned instruction then displays "decor" with the parameters that we are going to indicate to it:

- at the start we place "decor" at the top left of the screen, ie at coordinates $X = 0$ and $Y = 0$ and
- we tell "decor" to move its top-left corner:
 - from position $X = 0$ (top left of the screen)
 - until either in $\text{EndX} = -1920$ and
- to loop ($\text{LoopMove} = \text{True}$)
- with defined by the duration of the movement: $\text{Duration} = 20000$. (note: 20000 milliseconds = 20 seconds for 1 loop)



But you are probably wondering why he is rambling, we saw that at the start of this user guide.

1. So do RUN
2. See the scenery turn in loop, loop, loop, loop, loop, loop

We continue, now we need a hero, who better than Magic?
So we add this new line below the 1st line:

Actor "magic", Image\$ = "magicfly.png"

1. Do RUN
2. See the scenery turn in a loop and watch Magic appear at the top left, so in X = 0, Y = 0. Normal we did not tell him anything, he takes the default values.

I don't know about you but I don't like to see Magic stuck at the top, so I modify the code as follows by adding a **Do ... Loop** to move Magic on the vertical Y axis:

**Actor "decor", Image\$="bg.png", X=0, Y=0, EndX=-1920,
LoopMove=True, Duration=20000**

Do

If Mouse Key = 0 then PY = PY+7 else PY = PY-15

Actor "magic", Y=PY, Image\$="magicfly.png"

Wait vbl

Loop

No panic, I explain. And before you complain, know that to do the same thing in a language like JavaScript, C #,... you need at least 5 times more lines of code. Let's look at this code:

The Do... Loop allows you to execute - without stopping - the code inside. This code inside modifies the variable PY depending on whether or not the mouse button is pressed.



A quick reminder of the **If... Then... Else** statement:
If my test Then I do this if it's true **Else** I do this if it's false
Example: If $2 = 2$ Then Print "true" Else Print "false"

Understood? We continue with our game:

Explanation of: **If Mouse Key = 0 then PY = PY + 7 else PY = PY-15**
If Mouse Key = 0 it means that you have not press the Mouse button -> in this case PY increases by 7 pixels ($PY = PY + 7$). On the Y axis when it increases it means that we go down the screen (recall $Y = 0$ is at the top of the screen).

If Mouse Key is not equal to 0 it means that at this moment you do not press the mouse button then we are in the else -> PY goes up by 15 pixels ($PY = PY-15$) on the Y axis. Indeed, when PY decreases it means that we go up (towards the top of the screen).

The overall result of this line is that Magic goes down by itself by 7 pixels (when you do not press on the mouse) and goes up by 15 pixels if you press the mouse or the screen of the smartphone.

Explanation of: **Actor "magic", Y = PY, Image \$ = "magicfly.png"**
Okay there you understand, in each iteration of the loop Magic moves to the new PY coordinates and therefore goes up or down according to the previous line.

Explanation of: **Wait vbl**

This instruction synchronize the display correctly when it is frequently modified. Strongly recommended in any loop.

1. Do RUN
2. Watch the scenery revolve around and watch Magic move up and down with the mouse pressed. Cool !

Next step, now we are going to put obstacles, with stalactites which will move from right to left (we speak of horizontal scrolling) and gradually get closer to each other to increase the level of difficulty. Okay with you ? No ? No problem you can modify all of this, this is just an example.

```
Actor "decor", Image$="bg.png", X=0, Y=0, EndX=-1920, LoopMove=True,  
Duration=20000
```

```
For I= 1 to 50
```

```
  Actor I, X=400+(I*320), Y=-780+rnd(I*20), Auto$="offsetX=-5",  
  Image$="ice.png"
```

```
Next I
```

```
Do
```

```
If Mouse Key = 0 then PY = PY+7 else PY = PY-15
```

```
Actor "magic", Y=PY, Image$="magicfly.png"
```

```
Wait vbl
```

```
Loop
```

I'll explain the part I just added.



A quick reminder of the **For... Next** statement: This is another form of loop but conditional. **For** *variable = start value* **To** *variable = end value* **Next** (end of loop)

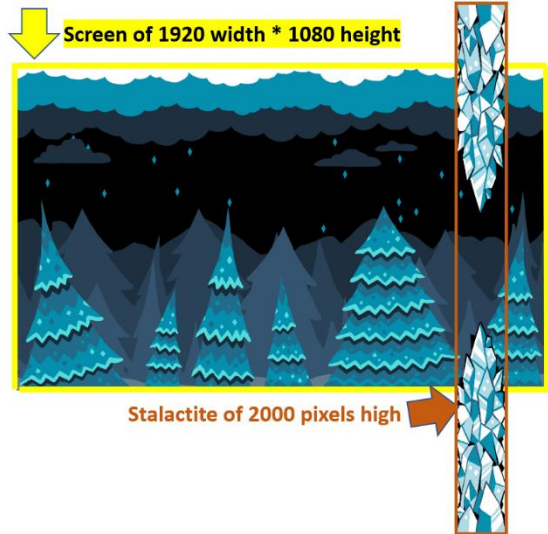
Example: For I = 1 to 10: Print I: Next I

Explanation of: Actor I, X = 400 + (I * 320), Y = -780 + rnd (I * 20),
Auto\$ = "offsetX = -5", Image\$ = "ice.png"

In the **For... Next** loop which counts up to 50, by increasing the value of I by 1 at each iteration, we therefore create 50 actors with the names: Actor 1, Actor 2, Actor 3,... Actor 50.

Note that the actors can have names like "magic" or "lucie" and then we put both ", but also numbers like in this game. It's very practical, like in this example where want to handle 50 actors.

Each of these 50 actors uses the same ice image (ice.png) which includes the upper part and the lower part and which is 2000 pixels high, more than the height of the screen which is 1080 pixels high.



To display the stalactites at different heights, we just move their top left corner vertically, so in Y. This is what we do with the parameter $Y = -780 + \text{rnd}(I * 20)$: We set Y with the value -780 and add a random value which is the product of $I * 20$.



The $\text{rnd}(x)$ function returns a random value between 0 and x. For example $\text{rnd}(10)$ returns a value between 0 and 10.

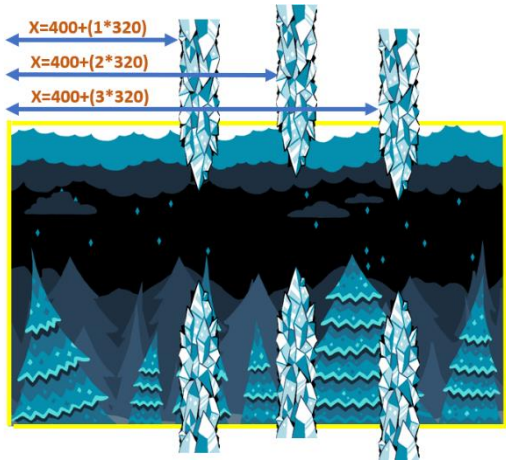
Thus for actor 1 (when $I = 1$) we will have a $Y = -780 + \text{rnd}(1 * 20)$, i.e. a value between $Y = -780 + 1 = -779$ and $Y = -780 + 20 = -760$. The more I increases, the stronger the amplitude of Y will be between two stalactites. So the difficulty of the game will increase.

Do the above calculation for $I = 50$.

To display the stalactites in different positions horizontally we also positioned them in X. This is what we do with the parameter:

$$X = 400 + (I * 320).$$

We set the X value at 400 plus $I * 30$.



Thus the more the value of I increases, the more the stalactite is created on the right.

To finish, it suffices to specify for each Actor that the displacement is automatic and to the left (therefore X decreases, and by 5 pixels) with the parameter Auto \$ = "offsetX = -5"

And voila with these parameters which modify the X and the Y in a single line of AOZ we display all our stalactites with randomness and a progression of difficulty.

Your turn to play 😊:

- Have fun changing the values of the parameters, this is the best way to understand.
- Understand that there is not only one method to achieve the result, it is the beauty of creation. I am suggesting a way to do this here, but another exercise would be to get there with a totally different code.

Are you ready, shall we continue?

If not, take your time, read again, modify,... Programming must be fun.

Well, we say to ourselves at this point that to finish our game something important is missing: collision management. That is, if Magic hits the ices or exits the screen, it has to, say... disappear, no, no, temporarily.



A little reminder about labels: anywhere in the code you can place a label, and ask the program to go there. For that the place to go is noted with a name here **START:** or **DEAD:** For a label use a name followed by a colon: to differentiate it from a variable or an instruction. And to ask the program to go to this label you have to call the **GOTO** instruction, here **GOTO START** or **GOTO DEAD** (without the colon:)

Let's see the completed code of our game and I'll explain it to you.

START:

```
Actor "decor", Image$="bg.png", X=0, Y=0, EndX=-1920,  
LoopMove=True, Duration=20000
```

```
For I= 1 to 50
```

```
  Actor I, X=400+(I*320), Y=-780+rnd(I*20), Auto$="offsetX=-5",  
  Image$="ice.png"
```

```
Next I
```

```
PY=Screen Height/2 // Magic set at the middle of the screen vertically
```

```
do // Loop start
```

```
  If Mouse Key = 0 then PY = PY+7 else PY = PY-15
```

```
  Actor "magic",Y=PY, Image$="magicfly.png"
```

```
  Wait Vbl
```

```

if Actor Col("magic", image$="ice") = True Then Goto DEAD
If PY>1000 Then Goto DEAD
Locate 0,7 : Pen 6: Print score: score=score+1
loop // loop end

DEAD: // If Magic collide that part of code is run
Actor "magic", Image$="Magic_Dead-0.png", Y=PY, ENDY=-20,
Auto$="offsetY=-15"
Actor "gameover", X=660, Y=400, Image$="gameover.png" //
display the GameOver banner
Wait click // Wait for a mouse click (or smartphone
touchscreen)
Del Actor "gameover" // delete the GameOver image

Goto START // go at the beginning of the program.

```

There are lots of new and interesting things to explain. It's going to be fine, we're progressing, that's good.

First of all I added two labels **START:** and **DEAD:** and we see the **Goto** which correspond. The last line is the **Goto START** which therefore simply says: go back to the very beginning of the program (where the **START:** is) and we will start playing again.

Explanation of:

if Actor Col ("magic", image \$ = "ice") = True Then Goto DEAD

We have already seen that, it is the function which tests the collisions, here between the actor "magic" (between "") and either another actor or (in our case here) an image which is called "ice" . Outcome? As soon as magic touches a stalactite, the function returns that it is True and Then we go at the DEAD label.

Explanation of:

If PY > 1000 Then Goto DEAD

We also go to the DEAD label if magic falls on the ground. The ground is at Y = 1080 pixels (the height of the screen), taking into account the height of magic whose hot spot is at the top left, testing a value > 1000 is sufficient.

Note: that in one case the if is all lowercase and not in the other, it's cool AOZ, we don't mess around with that.

OK now, we still have to see the DEAD code and we're done, see it's not that complicated!

Explanation of:

Actor "magic", Image \$ = "Magic_Dead-0.png", Y = PY, ENDY = -20, Auto \$ = "offsetY = -15"

Will display the image of the ghost of magic, starting from PY, which is the last vertical position of magic and ending the animation at Y = -20 ie just after the top of the screen. The ghost will therefore rise and disappear at the top of the screen.

Explanation of: **Wait click**

This instruction, as you might expect, is waiting for a mouse button click.

Explanation of: **Del Actor "gameover"**

This instruction stops the display of the actor (it disappears), so here we erase the Game over image, to then resume the game.

Your turn 🤖:

- Modify the code, the values... but step by step

15. PUBLISH IN A CLIC

AOZ Studio makes it easier to create, but also to publish and share your programs.

Rather than taking a server to host your program, mount it (install the software), create a URL, DNS,... notions that we will see later, AOZ Studio does all this for you automatically with the PUBLISH button.

So once with your program click on the **Publish button** (if you have the AOZ license):



Some explanations

Where are my apps stored when I use PUBLISH?

> Your programs are converted into HTML, Javascript (the language of internet browsers) and are stored in AOZ Studio's servers.

Is it associated with an account, a unique identifier?

> Yes soon, we are building the AOZ Store. A kind of digital store in which you can find the programs that you have published, those that you want to share, or even sale. This is where you can delete them for example. Right now you can only publish. Please note that for the moment the security management is very basic, always keep a copy of your source programs, we cannot guarantee the permanent presence of your published programs.

How to use the link and the QR code of my program?

> When you are going to use PUBLISH, it creates a unique URL (an internet link) and its QR code for your program, usable on Computers and Smartphones (take a picture of the QRcode this will activate the URL link in the default browser).

I do not have access to PUBLISH without the license of AOZ Studio 😞?

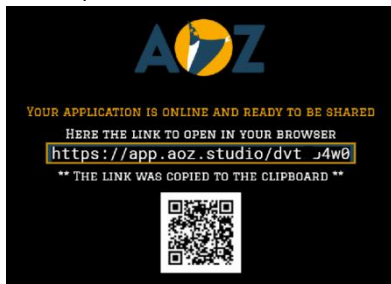
> And no. As you have seen you can use AOZ Studio without limit, without paying for any option and for all devices,... but to publish or make a save in .AOZIP format, to use AOZ Studio servers, or to have the right to publish your applications you must have a valid license. This will unlock those features and ensure you have the latest version with all the new stuff.

Now we publish our work!

Press the PUBLISH button (reminder with valid AOZ Studio license):



Wait a bit and you will have a screen like below with an internet link (URL) and the corresponding QRCode for computers and smartphones!



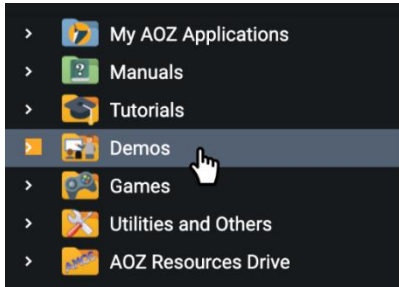
The Publish screen with:

- *The web link (URL)*
- *The QR code*

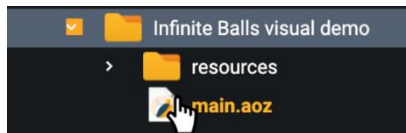
16. THE DEMOS AND GAMES

Now let's cut to the chase and have some instant fun!

Look near the top-left of your screen and click on the Demos or Games folder.



Open up a folder and click on the **main.aoz** (**main.** or any other name) then see all the computer code that was written for that program, many of them in just one morning between walking the dog and eating pizza. Please note: you do not necessary have to eat pizza to program in AOZ.



Next click one of the 2 RUN buttons, like this one to RUN this program in your AOZ Viewer.



We hope that AOZ Studio weaves it magic for you and becomes the future of your computing whatever your goal.

We're off to see the Wizards, the wonderful Wizards of AOZ, and leave you to discover the demos, games and magic provided with AOZ Studio.



17.KEYBOARD INPUTS

AOZ has a full range of keyboard input commands.

You can read keys one at a time, including modifier keys (Ctrl, Shift, Alt, Meta), and locking keys like Caps lock, etc. Some keycodes are stored in the input buffer, (an area of memory reserved for keys pressed, but not yet processed). You can even detect multiple keypresses at the same time.



There are 4 types of keyboard input instructions:

- **Variable** This input gets data from any simple type like a string, integer, or number, from the keyboard, and feeds it directly into each input variable. Multiple variables can be input using commas (or the **Enter** key) to separate the inputs.
- **Character** This type of input gets string only data from the keyboard. Character refers to any single letter, digit, special symbol, etc.
- **Buffer** This type allow you to interact with the keyboard buffer directly.
- **State** This type of input checks the current state of a key to see if it is pressed or not, but does not have to fill a variable with the result.

Keyboard variable inputs

Input

We have 4 forms of the Input instruction.

Input VARNAME { , VARNAME\$, VARNAME#, ... }

Description: This Instruction accepts keyboard input for simple variables. Input from the keyboard is inserted into the variable(s) specified by VARNAME, VARNAME\$, etc. The input will be displayed on screen as it is typed, with simple editing allowed using the backspace. Each input should be separated by a comma, or the **Enter** key. When the **Enter** key is pressed *after* the last variable terminates the input command it inserts the data into the specified variable(s).

Parameter(s): Any number of parameters may be requested, of any simple variable type: (**Integer**, **Number**, or **String**). Each variable must be separated by a comma or a the **Enter** key.

Example:

```
Do
    Input A$
    Print A$
```

Loop

Or you can type:

```
Do
    Input A,B$,C#
    Print A,B$,C#
```

Loop

Input TEXT\$;VARNAME { , VARNAME\$, VARNAME#, ... }

Description: An alternate form of the previously described Input Instruction, which also displays a text TEXT\$ just before the input. Otherwise, the command works identically.

Example:

Do

```
Input "Enter A,B$, and C#: ";A,B$,C#  
Print "Inputs: ";A,B$,C#  
Wait Vbl
```

Loop

With the exception of showing "Enter A, B\$, and C#:" just before accepting the inputs, this works Identically to the simple Input command.

Line Input VARNAME { , VARNAME\$, VARNAME#, ... }

Description: This Instruction functions identically to the Input command, except that it accepts all printable characters *including* the comma as part of the input. The **Enter** key is used to separate each input. See the description of the **Input** command for more details.

NOTE: If the "Please redo from start" error occurs, input starts again with the first variable.

Line Input TEXT\$; VARNAME {, VARNAME\$ VARNAME#, ...}

Description: As with the standard **Input** command, **Line Input** also has the option of displaying a prompt string before accepting inputs. Otherwise, this works identically to the **Line Input** command. See the description of **Line Input** for more details.

Keyboard Buffer

These commands allow you to interact with the keyboard buffer directly.

Put Key K\$ *Put Key is NOT CURRENTLY IMPLEMENTED, but will be soon, so:*

Description: This Instruction inserts the specified string (**K\$**) into the input buffer directly, as if it was typed from the keyboard. This is often used to pre-fill values for the **Input** instructions.

Example:

Do

Put Key "YES"

Input "Do you like AOZ? ";A\$

A\$=Upper\$(A\$)

Loop

Clear Key

Description: This Instruction clears the contents of the input buffer. This can be called immediately before **Wait Key** or other keyboard commands to ensure that we're waiting for new input.

Key Speed

Key Speed TIMELAG, DELAYSPEED

Description: This **Instruction** sets the speed at which input is accepted. **TIMELAG** is the time before repeat starts.

DELAYSPEED indicates the speed at which each keypress is accepted.

Keyboard character input

Character inputs accept only strings from the keyboard, usually 1 character at a time.

Inkey\$

Description: This Function checks the input buffer to see if a key has been pressed. If yes it puts that character into the function result, and removes it from the input buffer.

Return Value: The result (X\$ below) is a single character string containing the value of the next character from the input buffer. If the input buffer was empty at the time **Inkey\$** was called, then the result is an empty string: "".

Example#1 of Inkey\$:

```
Centre "Press d or f " : Print  
Do  
  A$ = Inkey$  
  If A$ = "d" Then print "d";  
  If A$ = "f" Then print "f";  
  Wait Vbl  
Loop
```

Example#2 of Inkey\$:

```
Result$=""  
Do  
  X$=Inkey$ : If X$ >= "@" And X$ <= "~" Then Print X$ ;  
Result$=Result$+X$  
  If X$=Chr$(8) Then Result$=Left$(Result$,Len(Result$)-1)  
  If X$=Chr$(13) Then Exit  
  Wait Vbl ' give the program time to run  
  Cls : Print Result$  
Loop  
Cls : Print Result$
```


The above code will get printable characters one at a time, and append them to **Result\$**. The backspace and enter special characters are also handled.

Input\$(N)

Description: This Function returns a string of **N** printable characters long. Once the specified number of characters (**N**) has been reached, the result is returned, and processing continues with the next statement. No editing is possible.

Return Value: The *String* result (**X\$**) returned is exactly **N** characters long.

Example:

Do

X\$=Input\$(3)

Print X\$

Wait Vbl

Loop

The above code accepts a 3 character string from the keyboard, and puts it into **X\$**.

Reading the keyboard state

Wait Key

Description: This Instruction waits for any keyboard key to be pressed before acting on the next instruction.

Example:

Print "Please press a key" : Wait key : Print "Thank you!"

In the example above, "Thank you!" will not be printed until a key is pressed.

Wait Input

Description: This very useful Instruction waits for any key or click on the Keyboard, joystick, mouse (not gamepad) to be pressed before acting on the next instruction.

Exemple :

Wait Input : Print "Moving forward"

Wait Click

Description: This Instruction specifically waits for a mouse click to be pressed before acting on the next instruction.

Key State

Description: This Function returns true when a *specific* key is pressed.

Parameter: The scan code (see next page) of the key to be tested.

Return Value: The *boolean* result is *true* if the key specified by the scan code (here \$42 in Hexadecimal or 66 in decimal) is pressed.

Example:

```
CODE=$42 : REM "it is the scan code of the letter B"  
Do  
  If Key State(CODE) Then Print "Key: ";CODE;" (" ;Chr$(CODE);")  
  was pressed."  
  Wait Vbl  
Loop
```

In the example above, the **Print** will not occur until the specified key: "B" is pressed.

Key Name\$

Description: This Function returns a *string* containing a textual name for the key pressed.

Return Value: The *string* result contains a textual description of the key pressed. Consult the appendix **Keyboard Codes** for a list of the key names and scan codes.

Example:

```
Do  
  Locate 1,1 : Print "hit gently your keyboard: ";
```

```

ik$ = Inkey$
If ik$ <> ""
    Print ik$;" ";
    k$=Key Name$ : Print k$;
End If
Wait Vbl

```

Loop

The above code will display the key names of the QWERTY keyboard as the keys are pressed.

ScanCode

Description: This Function returns the scan code of the *last key* in the input buffer as read by **Inkey\$**.

Return Value: The *integer* result contains the scan code of the *last key* pressed, or 0 if the input buffer is empty.

Example:

```

Locate 0,0 : Print Manifest$
Do
    I$=Inkey$
    If I$ <> "" ' Do we have a key?
        SC=ScanCode
        Locate 1,1 : Print I$, Hex$(SC,2) : SC=0
    End If
    Wait Vbl

```

Loop

The example above will display the printable key pressed and it's scan code in Hexadecimal.

Key Shift

Description: This Function determines which (if any) modifier keys are currently held down. Modifier keys are those that are usually intended to modify the value and/or function of another key being pressed at the same time.

Return Value: The integer result contains a bitmap indicating which modifier keys are currently pressed. The bitmap is filled as follows:

Bit Value Key Tested

0	\$0001	Left Shift
1	\$0002	Right Shift
2	\$0004	Left Ctrl (Caps Lock on Amiga)
3	\$0008	Right Ctrl (either Ctrl on Amiga)
4	\$0010	Left Alt
5	\$0020	Right Alt
6	\$0040	Left Meta (Amiga/Cmd/Windows)
7	\$0080	Right Meta (Amiga/Cmd/Windows)
8	\$0100	Caps Lock (AOZ mode only)
15	\$8008	Left Ctrl (Amiga mode only - bit 3 also set for compatibility)

Amiga manifest NOTE: More combinations of modifiers are allowed in AOZ than on an actual Amiga computer. For example, you can read both the *left* and *right* shift keys at the same time. Also, a new bit position (\$8000) has been added to represent the *Left Ctrl* key. For backward compatibility reasons, in Amiga mode, EITHER Ctrl key will set bit 3. (In the future, a tag may be added to only return *Left Ctrl* when it is pressed, instead of also activating the *Right Ctrl* bit.)

Example:

```
KS$=""
```

```
Do
```

```
KS=Key Shift
```

```
If KS <> 0
```

```
    If (KS & 1)=1 Then KS$=KS$ + "Left Shift| "
```

```
    If (KS & 2)=2 Then KS$=KS$ + "Right Shift| "
```

```
    If (KS & 4)=4 Then If Manifest$="amiga" Then
```

```
KS$=KS$+"Caps Lock| " Else KS$=KS$+"Left Ctrl| "
```

```
    If (KS & 8)=8 Then KS$=KS$+"Right Ctrl| "
```

```
    If (KS & 16)=16 Then KS$=KS$+"Left Alt| "
```

```
    If (KS & 32)=32 Then KS$=KS$+"Right Alt| "
```

```
    If (KS & 64)=64 Then KS$=KS$+"Left Meta| "
```

```
    If (KS & 128)=128 Then KS$=KS$+"Right Meta| "
```

```
    If (KS & 256)=256 Then KS$=KS$+"Caps Lock| "
```

```
Only for AOZ
```

```
    If (KS & $8000)=$8000 Then KS$=KS$+"Left Ctrl| "
```

```
Only for Amiga
```

```
End If
```

```
Locate 1,0 : Print Manifest$
```

```
Locate 1,1 : Print "KS: ";Right$(Bin$(KS,16),16)
```

```
Locate 1,2 : Print KS$
```

```
Wait Vbl
```

```
Loop
```

Hardware NOTE: Due to hardware differences in keyboards, certain combinations of modifier keys with or without other keys pressed may or may not be possible, so try to keep your keyboard interface simple, and test it thoroughly, in order to avoid compatibility issues.

ScanShift

Description: This Function determines which (if any) modifier keys were held down while the last key in the input buffer (read by **Inkey\$**) was pressed. The modifier keys are those that are intended to modify the value and/or function of the *last key* in the input buffer. This function operates exactly as the **Key Shift** function except that it relates to the *last key* in the input buffer vs. the current modifiers state.

Return Value: The *integer* result contains a bitmap indicating which modifier keys were pressed while the last key in the input buffer was pressed. The bitmap table is above with **Key Shift**.

Example:

```
SS$=""
```

```
Do
```

```
    I$=Inkey$ ' Get a key from the input buffer.
```

```
    SS=ScanShift ' Get that key's shift states
```

```
    Locate 0,0 : Print Manifest$
```

```
    If I$<>""
```

```
        Locate 0,1 : Print I$;" "
```

```
        Locate 0,2: Print "SS: ";Right$(Bin$(SS,16),16)
```

```
        If SS <> 0
```

```
            If (SS & 1)=1 Then SS$=SS$ + "Left Shift|"
```

```
            If (SS & 2)=2 Then SS$=SS$ + "Right Shift|"
```

```
            If (SS & 4)=4 Then If Manifest$="amiga" Then
```

```
SS$=SS$+"Caps Lock|" Else SS$=SS$+"Left Ctrl|"
```

```
            If (SS & 8)=8 Then SS$=SS$+"Right Ctrl|"
```

```
            If (SS & 16)=16 Then SS$=SS$+"Left Alt|"
```

```
            If (SS & 32)=32 Then SS$=SS$+"Right Alt|"
```

```
            If (SS & 64)=64 Then SS$=SS$+"Left Meta|"
```

```
            If (SS & 128)=128 Then SS$=SS$+"Right Meta|"
```

```
            If (SS & 256)=256 Then SS$=SS$+"Caps Lock|" ' Only
```

```
for AOZ
```

```
            If (SS & $8000)=$8000 Then SS$=SS$+"Left Ctrl|" ' 
```

Only for Amiga

```
End If
SS$=SS$+Space$(160) ' more than enough spaces to
erase previous states from display.
Locate 0,3 : Print SS$
SS$=""
End If
Wait Vbl
Loop
```

The example above will display the *last key* pressed along with the modifier keys which were pressed at the same time.

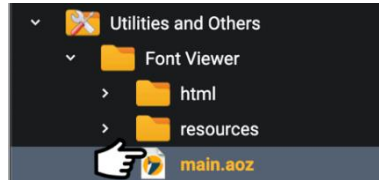


Here's a quick tip: to write an instruction over multiple lines, you can use the \ character. Example:

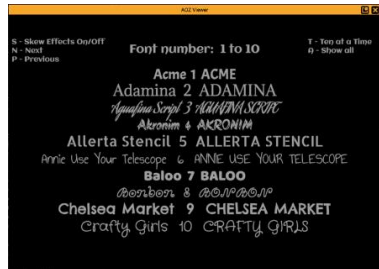
```
Actor \
"Player",\
HotSpot$ ="middle",\
Control$="ArrowRight: offsetX = 6; ArrowLeft: offsetX = -6",\
LeftLimit=40,\
RightLimit=Screen Width + 40
```


18. FORMATTING TEXT

Let's open an Application that parades text. Go to the "Utilities and Others" folder, open the folder called Font Viewer and trigger main.aoz as usual.



When you run this Font Viewer, you'll discover all the built-in fonts you are ever likely to need.



If you need more, then AOZ also welcomes any Google font with a simple tag. You can also create your own fonts using graphics.

Here's how:

1. Tell AOZ to embed a font in your application, using the **#googleFont** tag for a Google font or **#amigaFont** tag for an Amiga font. And if you don't know what a tag is, there is a dedicated chapter at the end of this guide.
2. The font must be installed in AOZ. Here is an example:

```
#googleFont: "Francois One"  
Set Font "Francois One", 80, "bold"  
Text Screen Width / 2, Screen Height / 2, "AOZ!", "# Center #"
```

3. In your application, choose the font with the "Set Font" instruction:

Set Font "font name", height, weight\$, italic\$, stretch\$

With weight\$: "normal" "bold" "light"

italic\$: "normal"

4. Display the Text with

Text x, y, "Mon texte", tags\$

With the tags:

horizontal alignment: #left #center #right

vertical alignment: #top #middle #alphabetic #hanging

#ideographic

direction (untested): #ltr / #rtl #inherit

or display with

Format Text MyBigText\$, x, y, width, height, tag\$

With the tags:

- '#left', '#center', '#right', '#start', '#end': horizontal align

- '#html': indicates a text formatted in HTML

- '#md': indicates a markdown text

- '#nozones': will ignore the = Zone \$ () AOZ commands

- '#animate': will render the links and images displayed on screen to be active, by starting a mouse-detection

background process

19. STORAGE OF IMAGES AND SOUNDS

Storage in a Memory Bank

A memory bank is an area of computer memory reserved for storing images, sounds, and other files that your AOZ program will need.

AOZ Studio uses banks. For each application, we find them in its "resources" folder:

- 1.images: The images used for Actor, Sprite or Bob instructions
- 2.icons: The icon images
- 3.musics: The sounds used for music instructions
- 5.samples: The sounds used for the sound effects

Note, in the resource folder you will also find:

- Assets: To store your images, sounds,... to be used directly by the Load Asset instruction (see later)

For example, if I put the "my_image.png" file in the "1.images" folder, and type this code:

```
// Display the image "my_image" on the screen  
Paste Bob 100, 100, "my_image"
```

Or this code:

```
Actor "image", X = 100, Y = 100, Image$ = "my_image"
```

My image will be displayed on the screen at coordinates 100,100. The name of my file is then used as a reference for my image.

If I now place the "1.png" file in the "1.images" folder, I could use this code:

```
// Display image number 1 on the screen  
Paste Bob 100, 100, 1
```

Or this code:

```
Actor 1, X = 100, Y = 100, Image = 1
```

The name of my file is automatically transformed into a number. So I can use this number as a reference for my image.

It also works for sounds. If I put the "my_sound.wav" and "25.wav" files in the "5.samples" folder and type this code:

```
Curs Off : Cls 0 : Paper 0  
Locate 0,10 : Center "Press 1 or 2"  
Do  
  A$ = Inkey$  
  If A$ = "1" Then Sam Play "my_sound"  
  If A$ = "2" Then Sam Play 25  
  Wait Vbl  
Loop
```

By pressing 1 or 2 on the keyboard the sound is played. As you can see, the Sam Play instruction plays the sound given as a reference. Here, as with the images, the sound is referenced by its file name ("my_sound") or its number (25).

- Any file placed in one of the bank folders will be automatically loaded when your program runs, and available immediately. Note that it increases the size of your program because it is loaded with it.

AOZ Studio banks support the most common file formats :

- Images: PNG, GIF, JPG
- Sounds / Music: WAV, MP3, OGG

Loading On Demand

You get the idea, memory banks are very handy because you don't have to worry about taking care of uploading files.

AOZ Studio offers another method which load the files on demand "on the fly", at a certain point in your program. In this case the image, the sound are not preloaded with your program (and its size is reduced accordingly).

For that, we are going to discuss "Assets".

Assets

An "asset" is a file that you can load from your program. Files used as assets should be placed in the "resources/assets" folder.

You can organize this folder as you wish, by arranging the files by categories (images, audio,...).

"It's the same as a database then? "

Yes, but unlike databases, assets are not loaded automatically when your program runs, but only when you choose to do so in your program. The example below loads the image "image1.png" from the resources / assets folder and displays it on the screen:

Cls 0

Load Asset "image1.png", "img1"

Paste Bob 0,0, "img1"

So in this example **Load Asset** loads the image "image1.png" previously placed in the "assets" folder, and gives it the reference "img1". The image is then available for the program.

When you call the Load Asset instruction, your program is "paused" while the file is loading and resumes when the file is loaded. This ensures that the file will be fully available for the future.

For sound or music, you can also use **Load Asset** to load your files "on demand":

```
Load Asset "music.mp3", 1
Load Asset "sfx.wav", 2
Curs Off : Cls 0 : Paper 0
Locate 0.10 : Center "Press d or f"
Do
  A $ = Inkey $
  If A $ = "d" Then Sam Play 1
  If A $ = "f" Then Sam Play 2
  Wait Vbl
Loop
```

In this example **Load Asset** loads the "music.mp3" and "sfx.wav" files with the names 1 and 2. When you press the "d" key on the keyboard, sound 1 is played. If you press the "f" key, it is sound 2.

NOTE: Sounds loaded with **Load Asset** restrict the use of Sam Play instructions as the sounds will be systematically played on all the channels, and at the frequency provided by the file.

Load Asset therefore allows, like Actor, to define a name for the files used, but also a number. Also, two assets can have the same name or the same number, provided that they are not of the same type (eg PNG and JPG). But if you name two images (for example) with the same name or number, the last loaded image will replace the previous one. Logic.

So, this code will work perfectly:

```
Load Asset "sfx.wav", 1
```

```
Load Asset "img1.png", 1
```

```
Curs Off : Cls 0 : Paper 0
```

```
Locate 0.10 : Center "Press Space bar"
```

```
Paste Bob 0,0,1 : // Our image 1 is displayed
```

```
Do
```

```
  A $ = Inkey $
```

```
  If A $ = "" Then Sam Play 1 : // Our sound 1 is played
```

```
  Wait Vbl
```

```
Loop
```

This example loads two assets: a sound and an image. Both are number 1. But AOZ differentiates between the two and does not point out errors.

Compatible file formats

Load Asset can load several file formats (some of which are not supported by the memory banks system seen previously).

- Music modules
 - o .mod: Pro-Tracker format
 - o .xm: FastTracker format
- Styles
 - o .css
- Javascript
 - o .js
- Data models
 - o json
- Images
 - o .gif
 - o .png
 - o .jpg
 - o .bmp

- o .svg: Vector format
- o .iff / .ilbm: Deluxe Paint Format

- Audio

- o .mp3
- o .wav
- o .ogg
- o .wma

- Video

- o .mp4
- o .mpeg4
- o .avi
- o .wmv
- o .ogv
- o .webm

Each format can be manipulated using dedicated AOZ commands.

"But how does AOZ know that I am loading a sound, an image or a video...? "

The Load Asset instruction automatically detects the type of the file thanks to its extension. So it's important to make sure your file extension is on the list above.

When you no longer need an asset, you can delete it with the Del Asset statement.

In the example below **Del Asset** permanently deletes the "audio" type asset with the number 1 (not the file).

Load Asset "sfx.wav", 1

Wait key

Sam Play 1 : // Our sound 1 is played

Wait 2

Del Asset "audio", 1 : // Deletes an asset

Default folder

The default folder in which your asset files should be placed is the "resources / assets" folder. It is from this folder that the AOZ Transpiler will load the assets. At this location you are sure that when you save your program in .AOZIP or publish it with PUBLISH the assets will be well integrated and therefore will not be missing.

If you want to use assets, for example an image1.png that sits in another directory, you can do this as follows:

1. use the **#useAssetsResources** tag by setting it to False. Thus, you will be able to load an asset file from a different folder than the default one. ****Warning**. This tag will later disappear.
2. Specify the path in the **LoadAsset** eg if the image1.png is in the C: / image directory

#useAssetsResources: False

Load Asset c: / image / "image1.png", "img1"

Well, good thing done, isn't it? If in doubt, try it out. Once this part is mastered, you are the boss.



20. AUDIO MAGIC

Sound effects

Every AOZ audio command acts independently from your gameplays and other routines, so they won't interfere with your programming. On the contrary, audio can enhance your work any way you like, to act as a marker, add realism, to soothe, shock or inject comedy and silliness. Let's start by ringing some musical bells, with 96 bells to play with and where Bell number 1 plays the lowest note up to Bell 96 for the highest note. Type in and RUN, then any key as much as you like to ring those bells:

```
For RING=1 To 96
Bell RING : Wait Key
Next RING
```

Here are some more built-in effects to type in and RUN:

```
Boom : Print "It was a dark and stormy night."
Wait 2
Bell 1 : Print "Count Dracula met a bat."
Wait 2
Shoot : Print "Ouch!"
```

There are AOZ commands to control volume, audio channels, wave patterns, white noise, sound samples, music modules and a whole lot more, so you are not just limited to bell-ringing. In fact you can liven up your creations with any sound effect or audio you like. Check the **Sam Play** instruction to produce sound as easily as possible.

We need to talk

Speech synthesis

You'll be asking AOZ to talk next, and you'd better believe that it can. Type in and RUN this:

```
Say "It was a dark, and stormy night."
```

We think that's a very attractive voice, and it will say anything you type in English, so have some fun with this, and please mind your language! Here's some more detailed info about the instruction:

Say "text", wait

Ask the system to speak your text with the simple Say instruction, followed by the string of characters representing the text to be heard. The wait parameter is a numerical value that is set to 0 by default and indicates the pause time after listening to the text.

Say "Hello world!" will talk in English by default

Or in French:

Speech Language "fr" : Say "Bonjour le monde!"

Note that you have 2 syntaxes:

Say "Hello world!" or **Say sentence\$="Hello world!"**

Before you make your machine say something, you can change the parameters:

Set Talk sex, fashion, pitch#, rate

Set the parameters for instruction **Say**:

- sex: 0 for female or 1 for male. By default, this setting is set to 0.
- mode: not used
- pitch#: Defines the pitch of the voice to speak the written text. The pitch setting is between 0.0 and 2.0 in AOZ mode, or 65 to 320 in AMOS Amiga mode.
- rate: Defines the speed of the voice, to dictate the written text. The value is between 0.0 and 10.0 in AOZ mode, or 40 to 400 in Amiga mode.

Set Talk sex=1, rate=0.5

Say "AOZ Studio is amazing!"

Talk Misc volume#, frequency

Equivalent of **Voice Volume** instruction.

(The frequency parameter is not used, for AMOS compatibility.)

Talk Stop

Stop the dictation of the written text.

Set Voice voice

Define the voice, among those available, to speak with.

Voice pitch#

Define the pitch of the voice from deep bass to high soprano, for speaking your written text. The pitch setting is between 0.0 and 10.0 in AOZ mode (65 to 320 in Amiga AMOS mode).

Voice Rate rate

Define the speed of the voice, for speaking the written text. The value is between 0.0 and 10.0 in AOZ (40 to 400 in AMOS).

Voice Volume Volume#

Define the volume of the voice. The volume setting is between 0.0 and 1.0 (40 à 400 in Amiga AMOS)..

You can try that:

Do

Voice Volume Volume#

Say "quick"

Volume#=Volume#+0.2

Loop

End

Speech Recognition

IMPORTANT NOTES:

- the audio recording works only in a web browser, not in the AOA Viewer. So make sure the microphone option of your web browser is active and RUN it with the F1 key (Run in a browser).
- To have sounds in the web browsers you first need to click in the program window, it is a protection from the web browsers.

Speech Recognition Start

Launches voice recognition

Here is an example using this instruction:

Speech Recognition Start

Print ">Start to say something"

Do

TXT\$ = Speech Recognition Value\$

If TXT\$ <> "" Then Print TXT\$

Loop

Press the F1 key to RUN, then speak English 😊

Speech Language

Defines the language spoken by the user. The value is a country code. By default, it is defined in English "en", "fr" is for French, like this:

Speech Language "fr"

Speech Recognition Start

Print "dis moi bonjour"

Do

TXT\$ = Speech Recognition Value\$

If TXT\$ <>"" Then Print TXT\$

Loop

Press the F1 key to RUN, then speak French 😊

Speech Recognition Stop

Stops voice recognition

Speech Recognition Reset

Resets all voice recognition settings.

Speech extra features

=Speech Synthesis Allowed

Returns True if audio text is allowed by the system.

=Speech Recognition Allowed

Returns True if voice recognition is allowed.

To test you can do something like that:

If Speech Synthesis Allowed

Print "Speech Synthesis allowed"

Else

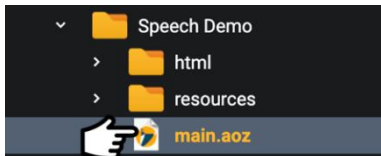
Print "Speech Synthesis NOT allowed"

End If

```
If Speech Recognition Allowed
  Print "Speech Recognition allowed"
Else
  Print "Speech Recognition NOT allowed"
End If
```

Speech Recognition Add Word, word\$ (not used yet)

There's more speech fun in Demos folder, if you click on the main.aoz file in the Speech Demo. Enjoy.



Samples

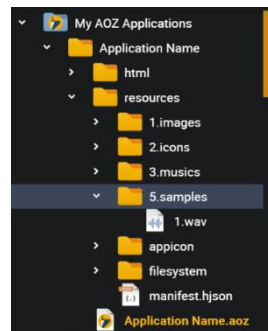
Most magicians will want to exploit audio samples in their computer games and routines. Samples are used for dialogue, atmospheric music and sound effects, just like in a movie, and with AOZ you can play them in any order, separately or simultaneously, at any volume, and with fades, crossovers and special effects.

The instruction to Play a Sample of audio is **Sam Play**, we hesitated with AltCtrlxx1 but after long discussions with the team we choose Sam Play.

As always, when you create an application, your audio files will be kept in a ready-made sounds folder. There's a separate folder for your application, so let's move on to an overview of that subject.

You can copy/paste sounds, music, likewise images in the corresponding resources folder of your application by moving the file in it. You can copy a file by dragging it in the folder while the Ctrl key is pressed.

Sam Play 1 will play the sound named 1 in the 5.samples folder ->



For sounds and music, it is possible to manage the volume, play in a loop, know the current playback position (See: Load Asset, Audio Loop On/Off, Play Audio, Stop Audio, Volume Audio)..

An other method is to use the Load Asset instruction, here is an example, knowing that your sound file is in the sounds folder that you will have first create :

Load Asset "resources/sounds/my_audio.wav", "sfx"
Wait Key
Play Audio "sfx"
Pause Audio "sfx"
TM = Time Audio("sfx")

Music

You've got a complete range of facilities and commands for playing music composed with the AOZ system, as well as for music created with systems like Med Soundstudio and Tracker. If you can't write your own musical masterpieces, don't worry. AOZ allows you to take another composer's musical soundtracks and add them to your own games and utilities. There are tens of thousands of public domain soundtracks written with systems like SoundTracker, NoiseTracker, and all the commands to load them and use them are built in to AOZ.

And there are the big samples library from Miko that we supply free for (only) your AOZ Studio programs (no royalties to pay).

AOZ supports the following formats: wav, ogg, mp3, mod, xm.

CAUTION: To play a sound, the internet browser requires prior user action (browser protection). So you have to keep the AOZ Studio SplashScreen display or replace it with something that similarly requires you to click beforehand.

21. MOUSE INPUTS

USING THE MOUSE

AOZ provides full support for the mouse, trackball, and similar devices. We already discussed it in the Actor instruction chapter, in addition to returning and setting the position of the pointer, and reading the status of the mouse buttons, we can also change the mouse pointer's shape, limit the area of the display in which it is active, and more.

Change Mouse Shape

Description: This Instruction sets the shape of the mouse pointer to shape number N.

Parameter: The Integer Parameter (N) sets the shape of the pointer as follows:

Shape Nb	Pointer Shape
1	Pointer
2	Crosshair
3	Busy
4	Finger

Example:

For M=1 To 4

 Print M : Change Mouse M

 Wait Key ' new mouse shape when a key is pressed.

Next M

NOTE: On the Amiga, a mouse shape > 3 would pull the shape from the Sprite bank.

Hide

Description: This Instruction makes the mouse pointer invisible. Each time Hide is called an internal hide counter is incremented. The mouse pointer will remain hidden until the hide counter is 0 again.

NOTE: Although Hide makes the pointer invisible, it is still active, and it's position may be tracked and read.

The purpose of the internal hide counter is to simplify keeping track of who hid the mouse when. If you have multiple routines hiding or showing the mouse pointer, it can be cumbersome to keep track of whether it should be on or off. The internal counter automatically keeps track.

Example:

hides=0 'simulate the internal counter. Type H or S keys.

Curs Off 'Hide the cursor

Do

 I\$=Upper\$(Inkey\$)

 If Mouse Key=1 Or I\$="H" Then Hide : Inc hides : Repeat Until Mouse Key=0

 If Mouse Key=2 Or I\$="S" Then Show : Dec hides : Repeat Until Mouse Key=0

 Locate 0,0 : Print Using "-##";hides

 Wait Vbl

Loop

Show

Description: This instruction makes the mouse pointer visible. Each time show is called it decrements the *hide counter*. The mouse pointer become visible when the number of shows equals the number of hides. (when the *hide counter* becomes 0)

Hide On

Description: This instruction makes the mouse pointer invisible **all the time**. It *ignores* the internal *hide counter* used by the regular **Hide** and **Show** instructions. It is like a *show "override"*.

NOTE: Although **Hide On** makes the pointer invisible all the time, it is still active, and it's position may be tracked and read.

Show On

Description: This instruction will make the mouse pointer visible **all the time**. It *ignores* the internal *hide counter* used by the regular **Show** and **Hide** commands. It is like a *"hide override"*.

hides=0 ' simulate the internal counter. Type H or space bar.

Curs Off 'hide cursor

Do

 I\$=Upper\$(Inkey\$)

 If Mouse Key=1 Or I\$="H" Then Hide : Inc hides : Repeat Until Mouse Key=0

```
If Mouse Key=2 Or I$=" " Then Show On : Repeat Until Mouse Key=0
' Force visible ALWAYS
Locate 0,0 : Print Using "-##";hides
Wait Vbl
Loop
```

Limit Mouse X1,Y2 To X2,Y2

Description: This **Instruction** limits the movement of the mouse pointer to a specified rectangular area of the current screen.

Parameters: The **Integer** parameters represent the upper left and lower right corners of the rectangular area being defined.

X1 and Y1 are the X and Y coordinates of the upper left corner.
X2 and Y2 are the X and Y coordinates of the lower right corner.

The mouse pointer will not be displayed outside of this defined area.

Example:

```
Screen Open 0,800,600,32,Lowres
Limit Mouse 0,0 To 400,300
Do
    Wait Vbl
Loop
```

The above example will limit mouse movement to the specified area.

NOTE: When **Limit Mouse** is called with NO parameters, it restores the normal area of the mouse, allowing it to move around the entire screen.

= Mouse Click

Description: This **Function** returns whether mouse buttons have been clicked (a single click). The result is a bitmap which is returned as follows:

Bit	And	Description
0	\$01	Left Button
1	\$02	Right Button
2	\$04	Middle Button (if it exists)

Return Value: The **Integer** result is a bitmap with each bit position representing one of the mouse buttons as described above.

Example:

```
Do
  Btns$=""
  CLK = Mouse Click
  Locate 0,I :
  If (CLK & 1) = 1 Then Btns$ = Btns$+"Left": I=I+1
  If (CLK & 2) = 2 Then Btns$ = Btns$+"Right": I=I+1
  If (CLK & 4) = 4 Then Btns$ = Btns$+"Middle": I=I+1
  Print Bin$(CLK,8);" ";Btns$
  Wait Vbl
Loop
```

NOTE: Once a button has been pressed, the "clicked" state will not activate again until the mouse button is released and then clicked again.

= Mouse Key (Button)

Description: This **Function** returns the current state of the mouse buttons. This result is a bitmap which is returned as follows:

Bit	And	Description
0	\$01	Left Button
1	\$02	Right Button
2	\$04	Middle Button (if it exists)

Return Value: The **Integer** result is a bitmap, with each bit position representing one of the mouse buttons as described above.

= Mouse Wheel

Description: This **function** returns the current state of the mouse wheel. The state is returned as follows:

RESULT	Mouse Wheel State
0	Neutral
-1	Wheel moved up
1	Wheel moved down

Return Value: The **Integer** result indicates whether the mouse wheel has been moved up or down, or remains neutral.

Example:

PrevMW=0

Do

 MW = Mouse Wheel

 If MW <> PrevMW Then Print Using "-#";MW

 PrevMW = MW

Loop

The example above allows you to see changes in the mouse wheel status.

= Mouse Zone

Description: This **Function** returns the number of the Zone currently under the mouse pointer.

NOTE: A zone is simply a defined rectangular area of the screen reserved for collision or mouse position detection. (See also *HZone*)

Returns: The **Integer** return value is the Zone number under the specified coordinates. If there is no Zone under those coordinates, 0 will be returned.

Example:

Palette 0,\$FFFFFF,\$FF0000,\$00FF00,\$0000FF,\$FFFF00,\$00FFFF,\$FF00FF

Cls 0

Reserve Zone 5

MakeZone[1,20,0,100,100]

MakeZone[2,101,0,200,100]

MakeZone[3,20,101,200,200]

MakeZone[4,201,0,799,599]

MakeZone[5,20,201,200,599]

Do

Locate 20,5 : Print Using "-###";Mouse Zone
Wait Vbl

Loop

Procedure MakeZone[zn,x1,y1,x2,y2]

Ink zn+1 : Box x1,y1 To x2,y2

Set Zone zn,x1,y1 TO x2,y2

End Procedure

This example introduces new concepts, if it sounds very complicated don't panic, go on.

= Mouse Screen

Description: This **Function** returns the screen number under the mouse pointer. (*See also ScIn(X,Y)*)

Returns: The **Integer** return value is the screen number under the mouse pointer. If there is no screen under those coordinates, a negative number will be returned.

Example:

Global scrWidth,scrHeight
scrWidth=360 : scrHeight=220
#displayWidth: 1080
#displayHeight: 660

ScreenSample[1,0,0]
ScreenSample[2,scrWidth,scrHeight]
ScreenSample[3,scrWidth*2,scrHeight*2]
ScreenSample[4,0,scrHeight*2]
ScreenSample[5,scrWidth*2,0]

Do

Locate 1,1 : Print Using "##"; Mouse Screen

```

        Locate 1,2 : Print Using "-####"; X Mouse;" , " ; : Print Using "-####";Y
Mouse;" "
        Wait Vbl
Loop

Procedure ScreenSample[n,xPos,yPos]
    Screen Open n,scrWidth,scrHeight,32,Lowres
    Screen Display n,xPos,yPos
    Palette
0,$FFFFFF,$FF0000,$00FF00,$0000FF,$FFFF00,$00FFFF,$FF00FF
    Curs off : Ink n+1 : Bar 0,0 To scrWidth-1,scrHeight-1
End Procedure

```

This example introduces new concepts, if it sounds very complicated don't panic, go on.

= ScIn(X,Y)

Description: This Function **SCIN** returns the screen number at the specified coordinates. Typically used with **X Mouse** and **Y Mouse** to determine when the mouse pointer has entered a particular screen.

Parameters: The **Integer** parameters **X** and **Y** indicate the hardware coordinates we're checking.

Returns: The **Integer** return value is the screen number under the specified coordinates. If there is no screen under those coordinates, a negative number will be returned.

Example:

```

Screen Open 0,800,600,32,Lowres
#displayWidth: 800
#displayHeight: 600

SetPalette : Screen Display 0,0,0 : ClearIt
Ink 4 : BigX

```

```

S=1
Screen Open S,800,600,32,Lowres
SetPalette : Screen Display S,0,0 : ClearIt
Ink 2 : BigX

S=2
Screen Open S,800,600,32,Lowres
SetPalette : Screen Display S,1920-800,0 : ClearIt
Ink 3 : BigX

S=3
Screen Open S,1920-1600,1080-600-1,32,Lowres
SetPalette : Screen Display S,800,601 : ClearIt
Ink 5 : BigX

Screen 1 : Pen 1 ' Screen number display to screen 1
Do
  Locate 1,1 : Print Using "-#";ScIn(X Mouse,Y Mouse)
  Wait Vbl
Loop

Procedure SetPalette
  ' Set palette for current screen.
  Palette 0,$FFFFFF,$FF0000,$00FF00,$0000FF,$FFFF00,$00FFFF,$FF00FF
End Procedure

Procedure ClearIt
  Pen 1 : Paper 0 : Cls 0
End Procedure

Procedure BigX
  ' Draw box & big X in current color on full area of current screen.
  Box 0,0 To Screen Width-1,Screen Height-1
  Draw 0,0 To Screen Width-1,Screen Height-1
  Draw Screen Width-1,0 To 0,Screen Height-1
End Procedure

```

Try moving the mouse around over the various screen areas, and in the overscan area. You'll see the proper screen number appear for each screen, and a negative number when no screen exists under the mouse pointer.

= X Hard(X_SCREEN)

Description: This **Function** converts a horizontal screen coordinate into a hardware coordinate. **NOTE:** Hardware coordinates are relative to the upper left corner of the current **display area** with respect to the current screen. This is like the overscan areas of a video monitor.

Parameters: The **Integer** parameter, **X_SCREEN** is the screen coordinate to convert in the current screen.

Return Value: The **Integer** result is the horizontal hardware coordinate relative to the current **display area**.

Example:

```
// Set display size of the default screen.
#displayWidth: 1920
#displayHeight: 1080
Palette 0,$FF0000
Ink 1 : Pen 1 : Paper 0 : Cls 0
Box 0,0 To 1919,1079 ' Show the outer bounds of the display (in
red)
Locate 1,0 : Print "Display Area"
// Open a smaller screen.
Screen Open 1,800,600
Screen Display 1, 100,70,, ' Move screen 0 down and right a
bit...
Palette 0,$FFFFFF
Ink 1 : Pen 1 : Paper 0 : Cls 0
Box 0,0 To 799,599 ' Show bounds of Screen 1 (in white)'
Locate 1,0 : Print "Screen 1"
Locate 1,2 : Print X Hard(0),Y Hard(0)
' Above should display the hardware coordinates of the upper
left corner of the current screen.
```

= Y Hard(Y_SCREEN)

Description: This **Function** converts a vertical screen coordinate into a hardware coordinate. Cf X Hard(X_SCREEN)

X Mouse =

Description: This **System Variable** can be used to set the position of the mouse pointer using *hardware* coordinates.

NOTE: Your web browser may or may not be able to control the mouse position.

Example:

```
#displayWidth: 320
#displayHeight: 200
Screen Open 0,320,200,32,Lowres
X Mouse = X Hard(160) ' Middle of screen horizontally
Y Mouse = Y Hard(100) ' Middle of screen vertically
Wait Key ' Allow the user to see the mouse position until a key is
pressed.
End
```

The above code will open a 320 x 200 screen, and position the mouse pointer in the middle of the screen.

= X Mouse

Description: This **Function** returns the horizontal position of the mouse for the current screen.

Return Value: The **Integer** result is the horizontal position of the mouse on the screen. The result is expressed using hardware coordinates.

NOTE: *Hardware* coordinates take into account the entire display area including the overscan portion around the screen.

Example:

Screen Open 1,1280,800,32,Lowres

Screen Display 1,200,50

Locate 1,0 : Print "Coordonnées écran "

Locate 1,2 : Print "Coordonnées matérielles "

Do

Locate 25,0 : Print Using "X: -####";X Screen(X Mouse);

Print Using " Y: -####";Y Screen(Y Mouse)

Locate 25,2 : Print Using "X: -####";X Mouse;

Print Using " Y: -####";Y Mouse

Wait Vbl

Loop

= Y Mouse

Description: This **Function** returns the vertical position of the mouse for the current screen. See X Mouse

= X Screen(X_HARD)

Description: This **Function** converts a horizontal hardware coordinate into a screen coordinate.

NOTE: Screen coordinates are relative to the upper left corner of the current screen.

Parameters: The **Integer** parameter, **X_HARD** is the hardware coordinate to convert. (Hardware coordinates are relative to the entire display area include the overscan portion.)

Return Value: The **Integer** result is the horizontal screen coordinate equivalent of the hardware coordinate parameter.

Example:

Ink 1 : Pen 1

Box 1,1 To Screen Width-2,Screen Height-2

Do

Locate 2,2 : Print Using "X: -####";X Screen(X Mouse);" "

Locate 2,4 : Print Using "Y: -####";Y Screen(Y Mouse);" "

Wait Vbl

Loop

= Y Screen(Y_HARD)

Description: This **Function** converts a vertical hardware coordinate into a screen coordinate. See X Screen(X_HARD)

22. INCLUDE

Include allows to insert the source of one or several AOZ source code file(s) into the current file. It is a very powerful instruction which allows to create all kinds of procedures, or routines, or data in external files. Therefore this helps make your main code more readable and easy to debug.

The syntax is simple:

Include "Path to the source code to include".

You simply indicate the path to the file to include as a parameter. You can also, if the file is located somewhere in the AOZ Drive, just specify the name of the file. AOZ will perform a search in the AOZ Drive and use the file if it is found.

It is perfectly possible to have several include and to include some AOZ files in a file that is itself included (includes in cascade), AOZ makes sure that no code is duplicated.

Here is an example of inclusion of code from the "AOZ Drive/includes" directory:

```
// Include the code for facebook...
Include "fbdemo"
// The Program main loop
Do
...
Loop
// Call the code located in the fbdemo that is included now.
VIEW_INFOS
```


23. AOE TAGS

The AOE transpiler (who convert your AOE code in Javascript and HTML) offers a complete system of tags to control every aspects of the transpilation of your application. That put a lot of power into your hands.

We do not recommend to read this first, nor in second...

What is a tag

A tag is a string that begins with the "hash" character, (# as on Facebook or Twitter).

Tags are often followed by a parameter. If the tag needs a parameter, then you must separate it from the tag itself with a colon ":".

This parameter can be:

- **a boolean:** True or False. A syntax error will be generated if the value is different from those two.
- **a string:** the string must be enclosed within quotes.
- **a number:** integer or floating point
- **a system variable** that was declared with the #define tag, in which you can put either booleans, strings or numbers, and that can also be defined in the transpiler command line (future AOE Studio versions), the Application Manifest or in the application setup panel.

Note: The Manifest.json file is located in the resources folder within your application folder, very interesting reading. It is about all the settings of the application.

Examples of AOZ tags:

```
#splashScreen:False
#forceCompile
#displayWidth:3000
#include "../utilities/my_super_utility_procedure.aoz"
#ifdef LINUX_VERSION
    ... some code just for Linux
#endif
```

How to use the tags

Simply insert the tag in your source code at the appropriate location. Most of the tags have a "global" effect, and are detected during the preprocessing phase of the transpiler (usually the tags that define HOW to transpile).

Some other tags have only an effect on the function or method or procedure they are included in.

And other tags will have an immediate effect, like **"#include"**, that allows you cut your program in pieces, that you may want to reuse. Include loads and insert the code referred to in the path of the tag, at the position of insertion, before transpiling, explaining how this:

Say this below little program file is saved as "My Documents/AOZUtils/procedure.aoz"

```
Procedure TAG_DEMO
Print "This file will be included!"
End Proc
```

Then in another AOZ application, you can do:

```
#include "My Documents:AOZ Utils/procedure.aoz"
TAG_DEMO
End
```

List of available tags

Transpilation tags

This paragraph groups the tags used by the transpiler itself to transpile your application.

cleanExtensions

description: Deletes the object folder of all extensions before transpiling the application, forcing a complete re-transpilation
param: True or False to turn the feature on or off.

Eg: **#cleanExtensions: true**

You might need to use this tag if you have copied a new extension in your extension folder.

Also, keep in mind (at least for the first versions of AOZ) that this tag may solve some complex problems due to a de-synchronisation between the transpiler used to pre-transpile a certain extension and the current version of your transpiler.

If this happens, do one transpilation of your application with this tag set to true, all the extensions will be re-transpiled (this might take some time) and will be ready to use.

cleanModules

description: Deletes the object folder of all languages modules before transpiling the application, forcing a complete re-transpilation.

param: True or False to turn the feature on or off.

Eg: **#cleanModules: true**

Modules contain the core syntax of the AOZ language. They are represented in AOZ as simple AOZ source code visible in the aoz/language/v1_0 folder. All modules are pre-transpiled once for all at each new version (that's why the first boot of a new version of AOZ might take more time than the latter) to ensure the maximum speed of translation.

This principle works fine as long as all the sources have been synchronized with the same version of the transpiler. If they are not, some problems might occur, syntax errors, mismatch errors, or crashes of both the transpiler or the runtime.

The AOZ Studio transpiler is modular: by changing the path to the folders that contain the definition of the syntax, you change the syntax (more to come! :)). Normally you should not have to touch the code in it.

You might need to use this tag if you have manually copied a new module in your module folder.

Also, keep in mind (at least for the first versions of AOZ) that this tag may solve some incomprehensible problems due to a de-synchronisation between the transpiler used to pre-transpile a certain module and the current version of your transpiler.

If this happens, do one transpilation of your application with this tag set to true, all the modules will be re-transpiled (this might take some time) and will be ready to use.

clean

description: Deletes the HTML folder of the application before transpiling it, forcing a complete re-transpilation of all files.

param: True or False to turn the feature on or off.

When you work on an application, during development you often add files that you remove later (like images, fonts etc.) ...

As a default, and to avoid transpiling files that have already been transpiled, the AOZ Transpiler does not erase the destination folder where your application is built.

After a certain time, this folder can contain old files and stuff you do not use anymore and you may want to remove all the garbage: use this tag!

Please note that the transpilation will take a little longer as the transpiler will have to scan and re-transpile every file.

caseSensitive

description: Set the transpiler in case sensitive or insensitive mode

param: True or False to turn the feature on or off.

The default behavior of the AOZ transpiler is case-insensitive.

This allows you to type any combination of upper/lower case letters in the names of your variables, procedures, methods or objects.

But really, we suggest that after a while if you are a beginner you switch to case sensitive, as it is how all professional tools work.

You would never see a C transpiler being case insensitive, as for large programs it only brings confusion.

When is case sensitive mode, you need to know a few things:

- AOZ will reject names of instruction with improper casing. ALL instructions of AOZ begin with an upper case letter at the beginning of each word. Example "Print", "True", "Screen Open" will be accepted, but "screen Open" will not as "false" will be also rejected.
- Case Sensitivity also works for variables, for example if you define a variable like "MyVariable = 2", and later do "Print myvariable", your application will display "0" instead of "2" as the variable contained in the Print expression is recognized as a new variable, which are by default set to zero. This might be real dangerous and generate a lot of possible bugs with non-initialized variables. This is the reason why you should remove ALL warnings about "Non declared variable" when transpiling as this will for sure generate a bug or even a crash of your application.

syntax

param: nameOfSyntax\$:string: "loose" or "strict"

description: Defines the transpilation mode variables, such as case sensitivity and respect of "proper Basic syntax".

content:

AMOS Basic, AOZ grand-father was well know for having a loose syntax. Furthermore, the language was case-insensitive, which is today not recommended.

In order to be compatible and easy to use, the default mode when you create an application is "loose". This means that you can use upper or lower case letters in the names of your variables and procedures and they do not have to correspond to match.

For example: in "loose" mode, this would work:

```
score = 10 : Print SCORE
Display_Score[ score ]
Procedure DISPLAY_SCORE[ SCORE ]
    Print SCORE
End Proc
```

Let's not forget that we want to make learning easy and give programming freedom, but nothing prevents you from programming otherwise with these Tags.

endian

description: Sets the order of bytes in files and memory

param:type#:string: Either "little" (for pc) or "big" (for Amiga and Atari).

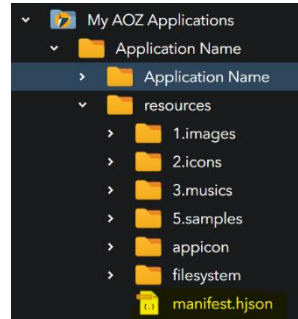
In the early days of computers, some computers preferred to store the lowest binary digital values first, and the highest last. This made the format of application and files on disc incomprehensible between computers of different brands. If your application is going to save fields that will be used on the Amiga or Atari, then set this flag to "big"... If not, then "little" is the right choice.

manifest

param:string:The name of the manifest to use

description:Indicates the manifest to use to transpile your application.

content:We already discussed this important Tag. The manifest is where all the information about the transpilation of your application are defined. Such value as the width of the application (display width and height) are stored, but you will find much more there.



The data is stored in the form of a HJSON object. HJSON is a readable version of JSON, with the same information, but presented in a very readable manner with comments. The most important values are available in the project setup wizard (when you create a new program with NEW button).

The V1.0 of AOZ supports the following manifest tags:

- "aoz": the application will be compiled for modern machines using the most recent version of the instruction set.
- "amos": if your application is designed from the Amiga computer. When set the AOZ renderer emulates the display of a CRT TV, including the dead zones on top, left, right and bottom. In AMOS, those areas were called "Hardware coordinate" and were usually not seen on the TV. Please refer to the original AMOS manual, included in this package in the "manuals" folder.
- "stos": (not available yet) To run applications from STOS on the Atari ST. It will emulate the display of the machine

platform

decription: Indicate the machine emulated by the renderer.

param:nameOfThePlatform\$:string: "aoz", "amiga" or "atari"

This is used by the renderer. In "normal" mode (understand "aoz"), the renderer access a modern machine, allowing hi resolution graphics and multitudes of sprites and bobs. There is

no such things as "Hardware coordinates", and sprites and bobs are displayed at their exact coordinates.

Note: If this flag is set to "amiga", the renderer emulates the display of a CRT Tv, with a limited number of pixels and blank areas on left, top, right and bottom.

Please note that for the moment, the copper is not emulated and that any application using the copper commands will simply have no effect on the screen.

If this flag is set to "atari", the diplays of the machine will also be emulated, with a maximum resolution of 640x400 in 2 colors.

useLocalTags

description: Allow local tags in extensions or modules to work. Local tags are tags that are only valid on files or extensions.

basicRemarks

description: Allow the use, or not, of the character ' (apostrophy) as a remark in your code

param: Switch the feature on or off.

content:

The original syntax of the Basic language allowed you to define remarks with the apostrophy character. Like this:

Print "Hello AOZ"

```
'      This is a remark with an apostrophy
```

```
Rem    This is a remark with the Rem instruction
```

```
//     This is a C-style remark
```

```
/*
```

```
    And this is a long C-Style remark,
```

```
    It can include any number of lines.
```

```
*/
```

We suggest to create your new applications using // and /* */ instead of the old-fashioned duo ' and Rem...

noWarning

description: Removes a specific warning from the transpiler syntax checking.

param:warningId:string:The identifier string of the warning.

comment:

The most frequent use of this instruction is to remove the "Variable not reserved" warning.

Although we highly suggest NOT to remove this warning, and if you are coding a new application to define all variables before using them, this tag may prove useful when running STOS and AMOS applications which were using the old syntax of remarks.

List of warning identifiers:

- font_not_found
- garbage_found_in_folder
- font_not_supported
- file_at_root_of_filesystem
- screen_not_multiple_of_font_size
- missing_folder
- missing_resources_folder
- creating_directory
- cannot_set_permissions
- illegal_bank_element_filename
- file_to_include
- copying_file_to_filesystem
- variable_not_declared
- duplicate_error_message
- instruction_not_implemented
- should_wait
- out_of_unique_identifiers

Example:

#noWarning: "font_not_found"

#noWarning: "variable+not_declared"

tvStandard

description: Indicate to emulate a NTSC display for the Amiga. Amiga emulation is by default in PAL mode.

param: Turn the features on or off.

export

description: Defines the output language of transpilation.,

param:exportType\$:string: The name to platform to export to. Today, this tag only support "html". In a near future version, it will also allow you to export direct to for ex Facebook (with the value "htmlFb"), create executables for Windows macOS and Linux ("windowsExe", "macOS", "linuxExe") and later for specific 3D engines like unreal or Unity.

saveTo

description: Indicates where to save the transpiled application.

param:path\$:string: Path to a folder where the transpiled application is saved.

As a default, AOZ saves the HTML folder with the transpiled application at the same level as the .aoz source code.

logTo

description: Change the folder where the transpiler logs are saved.

param:path\$:string: Path to a folder where the logs of the transpiler are saved.

As a default, AOZ does not save any log. You can activate this feature with the tag "#log:true", then AOZ will save it's transpilation logs into the aoz/logs folder. this tag allows you to define you own folder to save the logs.

log

description: Force the transpiler to save a log at each transpilation.

param: True to save the transpilation log, false not to. Default is false.

Logs can be useful in case of complex problems with the transpiler. It does not "compile" but why? It should! If this happens to you, activate the log feature to later after a transpilation inspect the files and probably find where the problem comes from.

forceTranspile

description: Force the transpilation of this module or extension. This tag is the contrary of #excludeFromBuild, it forces the transpilation of the module or extension each time you transpile your application.

Developers will appreciate the mental security to know that the code of your new extension is always compiled, specially in front of big debugging problems where the doubt comes "it might be the reason of that bug")

developerMode

description: Turn AOZ into developer mode.

param: True or False to turn the feature on or off.

AOZ contains a developer mode that can be useful when you are writing extension or want more deep features in AOZ.

It does not do much for the moment, but new options will be added for sure in the next month.

For the moment (v 1.0 of AOZ), being in developer mode bring the following advantages:

- when running your application, AOZ will **not** intercept the system keys of the browser, you will be able to open the Chrome debugger with F12 for example.
- when in developer mode, AOZ scans all the extension and modules folders looking for code that has changed since the last compilation. If you are not developing an extension and have not changed the content of the inside of the aoz system folder, you do not need it. But if you develop a new extension, a simple RUN of your AOZ test application will enforce the compilation of this very extension. Please note that this option will also make the transpilation process (a little) slower as AOZ

has to scan all the 50 sub-folders in the extension and module directories. As the number of extension will grow in the future, and reach hopefully the size of node.js list of packages (let's dream! :) it might become very handy then, and ensure everything is compiled with the latest version. You can turn on developer mode and off in the "Settings" panel of the AOZ IDE.

includeSource

description: Copy the source code of the application as remarks in the transpiled code

param: True or False to turn the feature on or off.

This option is true by default, and forces the transpiler to include in the code it produce the line of AOZ basic being transpiled.

Great care has been taken so that the source code of your transpiled application is as readable as possible.

The code of your transpiled application can be found in the my_application/html/run/application.js source code after a successful transpilation.

useSource

description: Replaces the entry script by the given string and proceed to compilation.

param:code\$:string: The code to replace with.This tag is for developer who use the AOZ transpiler as a node.js module (later to come).

With it, you can call the "transpile" function of the AOZ module replacing the code of the application refered to in the path of the "transpile" command by another code.

This new code will simple replace the original code in the project, the project keeping it's resources folder.

We use it internally for example when you open the Direct Mode window in the AOZ IDE: the current application is transpiled with an empty source code, which makes all the image and sound resources available in direct mode for experimenting.

define

description: Defines a variable to be used during transpilation.
param:variable: The name of the variable to create, without quotes.

AOZ allows conditional transpiling as C++ or C# transpiler. You define transpilation variables with the **#define** tag, and then use **#if**, **#else** or **#endif**, or **#ifdef** and **#endif** to define the zones in your application that will be transpiled.

This option, found in professional C or C# compilers is available in AOZ. This example demonstrates the use of this tag:

```
#splashScreen:false  
#define MACVERSION  
#ifdef MACVERSION  
    Print "This application has been transpiled for macOS"  
#else  
    Print "This application has been transpiled for another  
platform."  
#endif
```

If you replace the tag with:

```
#define WINDOWS
```

It will change the output. If you inspect the code of the transpiled application, you will see that only the selected code has been transpiled.

Such system also allows you to physically remove code, which may be important for demo applications until the user buys a license.

AOZ contains a preprocessor, called "Pass Zero" (say Hi to Pass Zero). It scans the source code taking the transpiler variables into account, and skip the part of the code that are nor active at this moment.

This VERY powerful option allows you to make multiple version of the same application, with only minor changes from platform

to platform, or different versions such as a free and commercial version with the SAME application.

let

description: Assign a value to a system variable

param:variable: The name of the variable.

param:value:any: The value to put in the variable, can be a string, a number or a boolean value.

Please note that this tag does not need a colon character after the name of the tag but instead the equal character, "="

Use this tag for conditional transpilation of specific parts of the source code.

```
#let:VERSION="0.99"
```

```
#if VERSION="0.99"
```

```
    Print "This version might crash!"
```

```
#else
```

```
    Print "All is debugged! :)"
```

```
#endif
```

if

description: Compares the value of a system variable to another value or system variable

param:variable: The name of the variable to test.

param:expression:any: The expression to compare to

System variable can carry values, and the #let tag allow you to define them. Please not that it is different from the #define tag in the sense that a real value is affected to the system variable, where as with #define the variable is created with a value of zero always.

The expression can include:

- number
- string
- boolean values
- other system variable containing values

Please note that for tags, there is no "type" associated with system variables, and that you do not need to add a \$ character after the name of a string system variable or a # after a floating point number.

```
#let VERSION = "0.99"  
#if version <> "1.00"  
  Print "Hmmm, it might crash!"  
#else  
  Print " YES! It wont crash, that's a promise (finger crossed :)"  
#endif
```

ifdef

description: Check if a transpilation variable is defined or not.

param:variable: The name of the variable to test.

Use this tag for conditional transpilation of specific parts of the source code.

```
#define IWILLNOTCOMPILE  
#ifdef IWILLNOTCOMPILE  
  A = "hello" // This line will generate a syntax error  
#endif
```

else

description: Indicates the code to transpile if the condition of a #if or #ifdef tag is NOT verified.

Use this tag for conditional transpilation of specific parts of the source code...

```
#define IWILLNOTCOMPILE  
#ifdef IWILLNOTCOMPILE  
  A = "hello" // This line will generate a syntax error  
#else  
  Print "All is fine! :)"  
#endif
```

endif

description: Closes a section of test started with #if or #ifdef

Use this tag for conditional transpilation of specific parts of the source code.

excludeFromBuild

description: Exclude the extension or module from the general build (use it for development).

This tag was created for developers of modules or extensions for AOZ. Once the code of the extension you are working on is open, it may prevent the chain of compilation of the other modules by generating errors.

Use this tag to remove the module or extension from the transpilation list, and you can use AOZ until you come back to your code.



Tags on applications

splashScreen

description: Activate or deactivate the splash screen at the beginning of your AOZ application. This is only possible for the paid license version of AOZ Studio, with the free version that Splashes will stay. Please support us and the future versions of AOZ Studio by buying a license.

param: Turn on or off the splash screen

Once again to try to motivate you to buy a license: As a default, when you run your AOZ application it displays an initial splash-screen.

If you use the free version of AOZ the splash-screen and/or some advertising of some sort will be displayed (check the license agreement if needed). The duration of the splash screen will increase.

If you have a valid Paid version/license you will be able to deactivate the splash-screen.

For applications that use sounds, a splash screen has a real important role though: it allows the sounds to be heard in your web browser. Modern Browser include a protection to prevent any page from playing loud sounds (imagine a pop page opening with a video yelling at you to buy a product).

Before ANY sound is played on any browser (being Chrome, Firefox, Safari... or their mobile equivalent), there MUST be a real user interaction, like a simple click or touch. This is the reason why if you use sounds in your application, the default splash screen asks for such confirmation. In this case the splash screen has a real role.

If you remove the splash screen, you will have to wait for the user to click somewhere BEFORE playing the first sound, otherwise the whole sound API will be de-activated.

displayWidth

description: Set the maximum width of the display in pixels.

param:width:integer: The width of the display in pixels.

This tag is equivalent to change the display.width value in the manifest of your application.

Eg: **#displayWidth:640**

displayHeight

description: Set the maximum height of the display in pixels.

param:height:integer: The height of the display in pixels.

This tag is equivalent to change the display.height value in the manifest of your application.

Eg: **#displayHeight:4800**

forceFullScreen

description: Force the application in full screen

param: True or False to turn the feature on or off.

This tag will force your application in full-screen, as best as possible depending on the exportation mode.

If you run your application in the browser, the application will be full PAGE first (not full screen, a protection from Browsers), and you will have to interact first with the application in order to turn it into real full screen. (user click).

If you run your application as a stand alone executable, then it will be full-screen at first.

Eg: **#forceFullScreen:true**

keepProportions

description: In full-page or full-screen modes, indicate if the dimension ratio of the application must be preserved.

param: True or False to turn the feature on or off.

If set to true, empty bars will be added on the left-right or on the top-bottom of the application, if set to false, the application will be resized to the entire browser area and graphics will be distorted.

Eg: **#keepPropotions:true**

fps

description: Displays a fps indicator on the top of the application's display.

param: True or False to turn the feature on or off.

You can set the width and height of the FPS indicator by editing the manifest used by your application.

appTitle

param:string:The new title to set

description:Allow you to change the title of the window where the application is running.

content:This instruction will have no effect if the application is running in the AOZ ide.

googleFont

description:Integrates a Google Font in your AOZ application.

param:fontName\$:string:The name of the font to integrate.

comment:

AOZ Studio supports Google Fonts as best as possible, and allows you to make applications that use Google Fonts and do not need to be connected.

To achieve this magic, the AOZ transpiler needs to know the name of the font to copy. The fonts are simple files downloaded from Google, and can be found in the aoz/fonts/google folder of your installation of AOZ.

To add a new Google Font please use the "Add Font" accessory, or do it manually, you will find the necessary template in aoz/templates/google font.

We have a paragraph TEXT in this User Guide to help you out. Be warned that some very precise Google Font need many files to work in all size and shapes and add one font only can significantly increase the size of you application.

amigaFont

description:Integrates an Amiga Font in your AOZ Application.

param:font string:The name of the font to add.

Amiga lovers will be happy to know that AOZ support black and white Amiga fonts. It is a joy to see those fonts again on a modern screen, some of them were really great.

The system works the same as for the #googleFont tag; indicate the name of the font and the files that define it will be copied in your application.

To add a new Amiga Font please use the "Add Font" accessory, or do it manually, you will find the necessary template in aoz/templates/google font.

Amiga font will work whatever the configuration of the transpiler (aoz or amiga or atari), but they might appear block in Full HD or higher resolutions.

We all remember Amiga fonts with style and precision, but on modern machines, as Amiga fonts are simple bitmap fonts (each character is in fact defined as a single image, where Google Fonts are vectorial) explaining why they appear blocky.

Amiga font contain one file for one size of the font. If AOZ cannot find the exact match for the height, it will take the biggest one and zoom it down. The "Text" and "Format Text" instruction will work even if you ask for a height that is not present in your application.

keymap

description: Set the keymap to use with the application.

param:keymap\$:string:"aoz", "amiga" or "atari"

A value of "aoz" will use the default Javascript keyboard. A value of "amiga" or "atari" will use the keyboard mapping of the specific machines.

tabWidth

description:Set the width of the tab character

param:numberOfSpaces:integer:the number of spaces to use for each tab. Default is 4.

displayEndAlert

description: Allow the display of an alert box at the end of your application if an error occurs.

param: Switch the feature on or off.

AOZ displays an alert box indicating the end of your application. If you inactivate this option with this tag, it will quit immediately (the program will halt in the browser).

Applications saved will return to the system with an error code of 0 (no error).

displayErrorAlert

description: Allow the display of an alert box at the end of your application if an error occurs.

param: Switch the feature on or off.

When an error occurs in your application, AOZ displays by default an alert box before quitting. If you inactivate this option with this tag, it will quit immediately after the fault (the program will halt in the browser).

Applications saves as stand-alone commands will report the number of the error as a result value.

sendCrashReport

description: Automatically sends a report after a crash of the application or the transpiler to AOZ Studio, with information about the crash.

param: True or False to turn the feature on or off.

The information transmitted is totally anonymous and only contain technical data about AOZ, your application, and your machine.

It helps us clean the product and remove as many bugs as possible, we appreciate that you share this information. but it is really up to you. ;)

useSounds

description: Forces the "sound" indicator and makes the splash-screen "Click to continue" upon start.

param: True or False to turn the feature on or off.

For applications that use sounds, the splash screen has a real important role though: it allows the sounds to be heard in your browser. Modern Browser include a protection to prevent any page from playing loud sounds (imagine a popup page opening with a video yelling at you to buy a product).

Before ANY sound is played on any browser (being Chrome, Firefox, Safari... or their mobile equivalent), there MUST be a real user interaction, a simple click or touch. This is the reason why if you use sounds in your application, the default splash screen asks for such confirmation. In this case the splash screen has a real role.

If you remove the splash screen, you will have to wait for the user to click somewhere BEFORE playing the first sound, otherwise the whole sound API will be de-activated.

Please read about the SplashScreen Tag.

insertIntoHead

description: Insert a string into the HEAD section of the index.html file generated by the transpiler.

param:CSS\$:string: A string containing code to insert in the HTML <head> </head? section of the index.html file that launches you application.

For advanced developers only, or people who want to make extensions, this tag gives you the freedom to have your code in this crucial file, index.html.

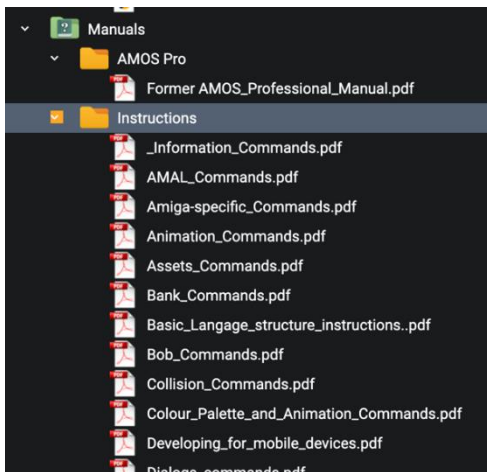


24. AMOS USERS

If you remember AMOS is the grandfather of AOZ created in the 1980s. This chapter is for our old friends who are familiar with AMOS, Easy AMOS and AMOS Professional, which are the ancestors of AOZ.

Compatibility

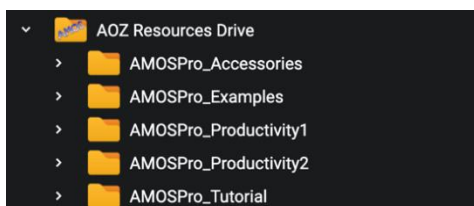
First of all, we are delighted to tell you that most of the code and banks from your old programs can be directly loaded into AOZ. If all is well, then everything will import automatically as soon as you load a .AMOS program.



The whole AMOS Professional manual is waiting for you in there, along with every instruction, command and module from A to Z. And you'll find a great range of brand new instructions, such as how to develop your work for use on mobile phones and how to manipulate HTML code for your websites.

We've even included the classic Amiga Game called CyBall and imported it into AOZ for you, as well as a full tutorial, how this was done on Youtube by Phill Bell [How I Ported My AMOS Game To AOZ Studio Start to Finish](#)

Remember, there is a dedicated AMOS folder for you to explore at your leisure, with masses more tutorials, oodles of examples and more accessories than you can wave a magic wand at.



There are more similarities than there are differences between your old AMOS and your new AOZ language, with the main difference being that AOZ

runs a lot faster, and is no longer restricted to one sort of computer. AOZ also supports object-orientation for ex and a huge range of controls and effects in your manipulation and control of animations.

AUDIO FILES

The tracker engine used supports not only the Amiga SoundTracker files, but newer formats supporting up to 32 channels, XM audio and more.

Bobs et Sprites

The world famous Bob and Sprite instructions, created with AMOS... You do not necessarily have to use these instructions in these modern times that we can use Actors, but you will find a load of demos and games that use them, and so many programmers still love them.

There are two types of movable objects, called "Bobs" and "Sprites". Let's deal with Bobs first.

A Bob can be moved around the screen, and there is no limit to the number of Bobs you can harness, apart from the amount of available memory of course. In other words you can have thousands of them performing on screen at the same time. You can control a Bob, track it and even give it special characteristics if it collides with something else or enters a special screen zone.

Best of all, you can animate a Bob to make moving cartoon characters, steampunk machinery, in fact anything that your imagination can dream up.

In the old Amiga time a Bob was stored as part of the current screen, so it can eat up quite a lot of memory, so in AMOS there was an alternative type of object called a Sprite.

A Sprite is also a graphic object that you can manipulate totally, and it's completely independent from the screen. In the 80s world of computers Sprites were manipulated directly by "hardware" which is why the Bob instruction was introduced, manipulated by software. Today this difference does not exist any more and the size limit is not really an issue anymore. But the name Sprite is widely use to define a moving graphic object. As you would expect, AOZ offers built-in routines to teach both of them magic tricks for intelligent movements. This chapter will continued unabashed, with examples using ~~Boobs~~ Bobs.

TO INFINITY AND BEYOND

With all these wonders for creating the movement and animation for your Actor, Sprite and Bob objects, you'll probably want to place them in a suitable electronic universe. Good news, your universes can be two-dimensional, three-dimensional and with effects like parallax scrolling where one layer of background graphics moves at a different speed from another layer to create a sort of 3D effect.

AMAL

AMAL is a little language within AMOS that was supporting the Amiga mouse as well. Now it is better to use the **Actor** instruction.

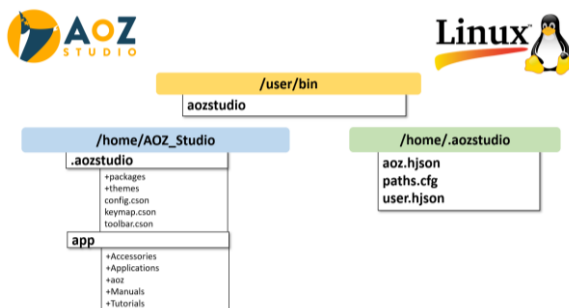
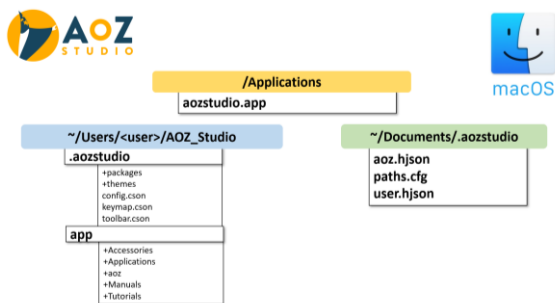
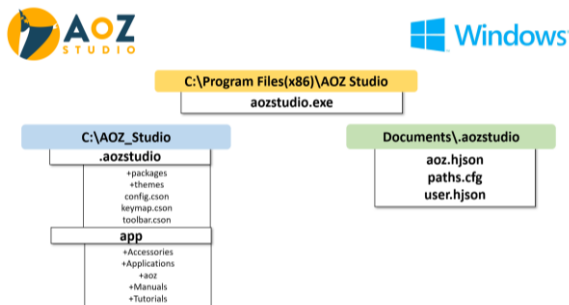
Discussing AMAL is beyond the scope of this chapter. Please consult the **AMAL** chapters in the former AMOS Amiga manual for more information.



25. INSTALLATION D'AOZ STUDIO

AOZ Studio is compatible with Windows, Mac OS and most of the Linux versions. We plan to expand its availability, for example to Raspberry.

Here are where the AOZ Studio files are in residence *****:



26. THANK YOU

To our fantastic community of early users. Nothing would have been possible without you during this almost 3 year journey in the making of AOZ Studio.

Please be patient with us, understand that we are working on making a better AOZ Studio, which is still a long and winding road. But we really hope you will enjoy what we've done together so far. And stay tuned for many more mind-boggling features.

Share your wildest desires on the Discord channel:
<https://discord.gg/6Cuy3vtj8N> in addition to benevolent magicians you will find us there to help you ...

...Until the day when it is you who will help others; you will then officially have the status of Magician. But if you are there it is because you now have the wand, well done!

