Co-funded by
the European Union

INTRODUCTORY GUIDE TO VIDEO GAME DESIGN AND DEVELOPMENT FOR SOCIAL CHALLENGES WITH "SCRATCH"

FVMP
Municipalistas por la Solidaridad y el Fortalecimiento Institucional
MUSOL
access

www.bgz-berlin.de

Netzwerk für betriebliche Integration und Sozialforschung e.V.
www.bisev-berlin.de

Gesellschaft GDLB für duales Lernen Berlin gGmbH

www.suedwind.at

DIE WIENER VOLKSHOCHSCHULEN
www.vhs.at

Municipalistas por la Solidaridad y el Fortalecimiento Institucional
www.musol.org

www.fvmp.org

About the Project:

"Access –Social Space Orientation in Youth Work 4.0" is supported by the European Union under the Erasmus+ program. The project addresses the limited participation opportunities for young disadvantaged people and develops intervention models along the three focal points of the EU Youth Strategy –inclusion, sustainability, and the digital world. In Berlin, Vienna, and Valencia, project tandems, consisting of youth work practitioners and experts in inclusion, sustainability, and digitalization, explore educational and participatory offerings for three selectively chosen target groups.

In Valencia, the Access Case Study endeavoured to cultivate a digitally inclusive society rooted in European values such as human dignity, freedom, democracy, equality, and the rule of law. This initiative empowers young citizens, particularly those from low-income areas, to exercise their rights and freedoms while influencing domains beyond their local communities. Participants are equipped with the tools to enjoy freedom, social protection, and equitable access to resources, ensuring that no one is left behind. Aligned with Europe's Digital Decade objectives, these young individuals are encouraged to acquire essential digital skills and leverage everyday technology to shape their future prospects in employment, education, training, and social integration.

1

INTRODUCTION

Welcome to the guide on integrating environmental and social themes into video game development!

Through this guide, the steps to create a prototype of a simple video game with the Scratch game engine are explained. This guide aims to be a pedagogical tool that allows trainers to address environmental and social challenges in their classrooms through video game programming. The functioning of the Scratch game engine is explained, and the steps to follow in order to develop a basic video game in six sessions are detailed. Designed to be used by an audience with no previous experience in video game programming, this guide features simple language and visual materials support. It can be used in classrooms to collaboratively develop, with an attractive and innovative tool - Scratch - video games that address challenges related to the Sustainable Development Goals while promoting students' technological and digital skills.

Join us as the student embark on this exciting journey of creativity, empathy, and digital innovation. Together, the student will harness the power of video games to inspire change and foster meaningful connections within our communities.

TRAINING OVERVIEW

In the first class, it will be conducted a collective dynamic with the students to identify and discuss the existing challenges in the neighbourhood. Next, the students will select the environmental or social themes to be addressed in the video games. The trainerwill introduce the students to narrative in a brief and concise manner so that students can develop the story of their choice at home. The trainerwill propose that students create the narrative design of the game themselves: what do they want to convey? Video games progress through Quests (missions); the trainerwill create a main Quest with a very basic structure:

1) A character asks for our help and gives us an important mission. To do this, students must create this character narratively, decide who he/she is, his/her needs, how to express him/herself, and write down a dialogue.
2) The main character must fulfil the mission (there will be no narrative contribution as this is the playable part).
3) The main character will complete the mission and then speak again with the initial character to conclude the story. Students will have to create this ending story on their own. With these three simple steps, students will be able to add depth to their video game through narrative, giving each project a different personality. As this is a project focused on development and programming, the students themselves must develop the dialogues alongside the tasks, keeping in mind the story they want to tell and writing them at the end of the project. This is an ongoing background task throughout the course where students' ideas will be heard and doubts resolved.

CHECKLIST TO PREPARE A WORKSHOP

From the Scratch website:

1) **Download Scratch: Start here:**
https://scratch.mit.edu/scratch_1.4

2) **Read the Scratch introductory guide:**
https://sip.scratch.mit.edu/scratchathome/

3) **See Scratch introduction videos:**
https://scratch.mit.edu/projects/editor/?tutorial=getStarted

4) **Make sure participants have Scratch accounts: Participants can sign up for their own Scratch accounts at:**
https://scratch.mit.edu,
or you can set up student accounts if you have a traineraccount. To request a traineraccount, go to:
https://scratch.mit.edu/educators

5) **Set up computers or laptops Arrange computers so that participants can work individually or in pairs.**

6) **Set up a computer with projector or large monitor You can use a projector to show examples and demonstrate how to get started.**

USE YOUR IMAGINATION!

In the first class, the trainerwill conduct a collective dynamic with the students to identify and discuss the existing challenges in the neighbourhood. Next, the trainerwill select the environmental or social themes to be addressed in the students' video games. The trainerwill introduce the students to narrative in a brief and concise manner so that they can develop the story they want to tell at home. The trainerwill propose them to create the narrative design of the game themselves: What do they want to convey?

In our first class, the trainerwill embark on a collaborative journey with the students to explore and dissect the challenges present in their neighbourhood. By engaging in collective dynamics, the traineraims to uncover the pressing issues that resonate within their community.

Following this exploration, the trainerwill carefully select environmental or social themes to serve as the focal point for the students' video game projects. These themes will not only spark creativity but also prompt critical reflection on the world around them.

As the trainerdelves deeper into the creative process, the trainerwill introduce students to the art of narrative storytelling. Through brief yet concise guidance, students will be empowered to craft compelling narratives that drive the gameplay experience. Encouraged to explore their own voices and perspectives, students will shape the direction and message of their games, conveying the stories they are passionate about.

CREATE YOUR STORY

Video games progress through Quests (missions); the student will create a main Quest with a very basic structure:

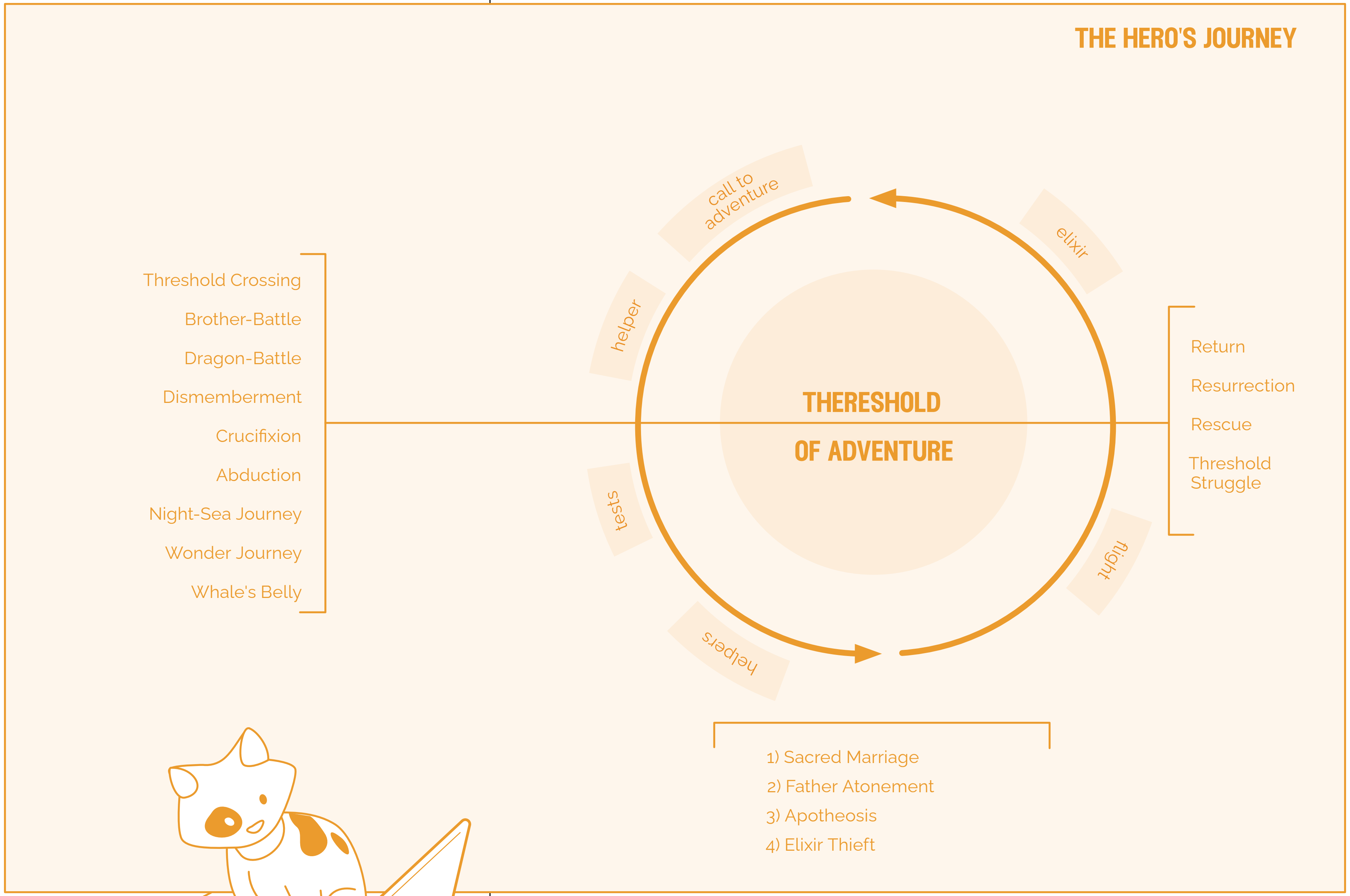
1) A character asks for our help and gives us an important mission. To do this, they must create this character narratively, decide who they are, their needs, how they express themselves, if the student know them previously, and write down their words.
2) The main character must fulfil the mission (there will be no narrative contribution as this is the playable part).
3) The mail character will complete the mission and then speak again with the initial character to conclude the story. Students will have to create this ending story on their own.

With these three simple steps, students will be able to infuse their video game with narrative, imparting distinct personality to each of the projects. As this is a project centred on development and programming, it is the students themselves who must develop the dialogues alongside their tasks, keeping in mind the story they want to tell and writing them at the end of the project. This is an underlying task that will be carried out throughout the course, during which students' ideas will be heard, and any doubts resolved.

For those who are interested, a brief explanation of the hero's journey will be provided, and they will be recommended to read Joseph Campbell's book "The Hero with a Thousand Faces," as well as search for related information online to enhance their stories. Scriptwriting, character creation, and storytelling are important aspects of the video game industry that do not require significant resources to develop. While this aspect will not actively be part of the course, it is relevant information that may motivate some students.

The hero's journey:

https://www.jcf.org/learn/joseph-campbell-heros-journey



INTRODUCTION TO SCRATCH

THE CODE EDITOR

In order to start working on the project, students will first need to understand how Scratch works and what elements it has. To do this, students will go to the official Scratch page where, at the top along-side the logo, students will click the "create" button to create a new project and will familiarize ourselves with the interface, keeping it in English since if students want to continue exploring, he/she will use the standard language.

On the **left side,** it contains all the blocks we will use to program the project's code. These blocks are simple instructions that we will combine to make our game work. We will explain them in detail later on.

On the **right side,** it is used to manage the graphical part of the project. The top part is the window where we can see the game while the bottom part is the list of all the sprites (2D images) of the project, where we can rename them or change their position, rotation, and scale.

Sprites are any type of actor (interactable elements) that we can include in the game, each with its own programming code.

These can range from characters to boxes or any type of playable element in the project. Finally, next to the list of sprites is the list of backdrops. A backdrop is the background on which we place the elements of the game, which can also have its own programming code, although we will only use it for aesthetic purposes to create the stage background.

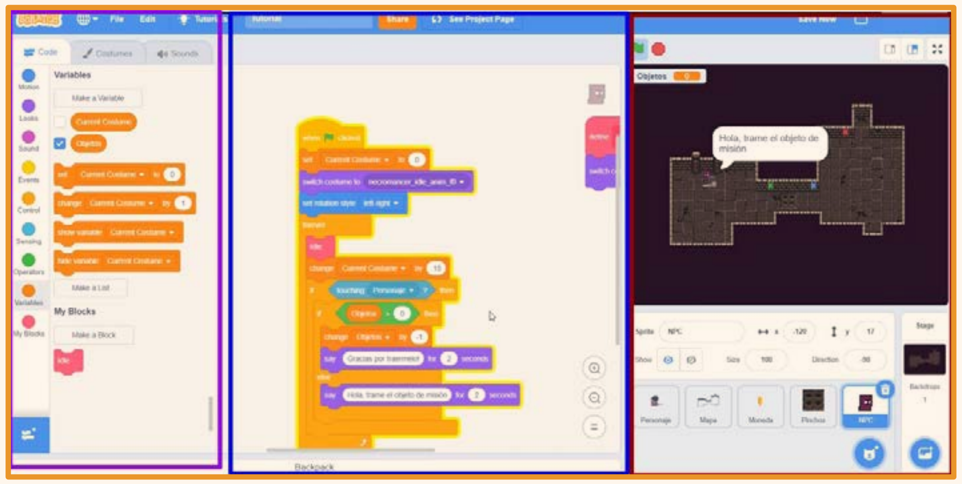
Lastly, in the **central part,** there is the workspace where we will place the blocks (mentioned earlier) to program our game. The programming seen here is for the sprite or backdrop that we have selected at that moment.

In addition to these three panels, in the upper left corner, students can see that there are also three tabs. Everything students have seen so far is within the "Code" tab, which is the one the student will use the most.

The next one, "Costumes" or "Backdrop", is used to change the appearance of the sprite or backdrop that the student have selected. From here, the student can modify the visual appearance of all the elements of the project. Mainly, the student will use it to animate or decorate the background. The student will see how to work in-depth with this editor later on.

The last tab is "Sounds", which is used to edit the sounds of the project. Since the student are creating a prototype, the student won't need to work with it.

Sprite Editor



BLOCK TYPES

In the "Code" tab, we have several colored dots, each dot representing a type of block. Let's proceed to explain them:

Motion (dark blue): This category manages the character's movement, allowing us to modify or understand things like its rotation or displacement.

Looks (purple): It allows us to manage the appearance of the character. Here, we can include dialogues, change the size of the character, add some effects, and even animate it.

Sounds (mauve): These blocks are used to execute and modify sounds, although we won't use them for this project.

Events (yellow): This category defines when things happen, such as what happens when you press a key, when the game starts, etc. It is one of the ones we will use the most, although we will try to keep it as simple as possible.

Control (orange): These blocks are closer to typical programming, although we will also try to use them in the simplest way possible. With this, we can manage everything that happens in our code, making it repeat other instructions several times, deciding whether to execute one set of instructions or another, etc.

Sensing (light blue): With this, we can know what is happening within the game, such as if the character is touching another element of the scene, or what keys we are pressing at that moment. It may sound similar to "Events", but we will see the differences as we apply it.

Operators (Green): This part is also closely related to typical programming and may be a bit more complicated to use than others. It allows for logical comparisons and mathematical operations.

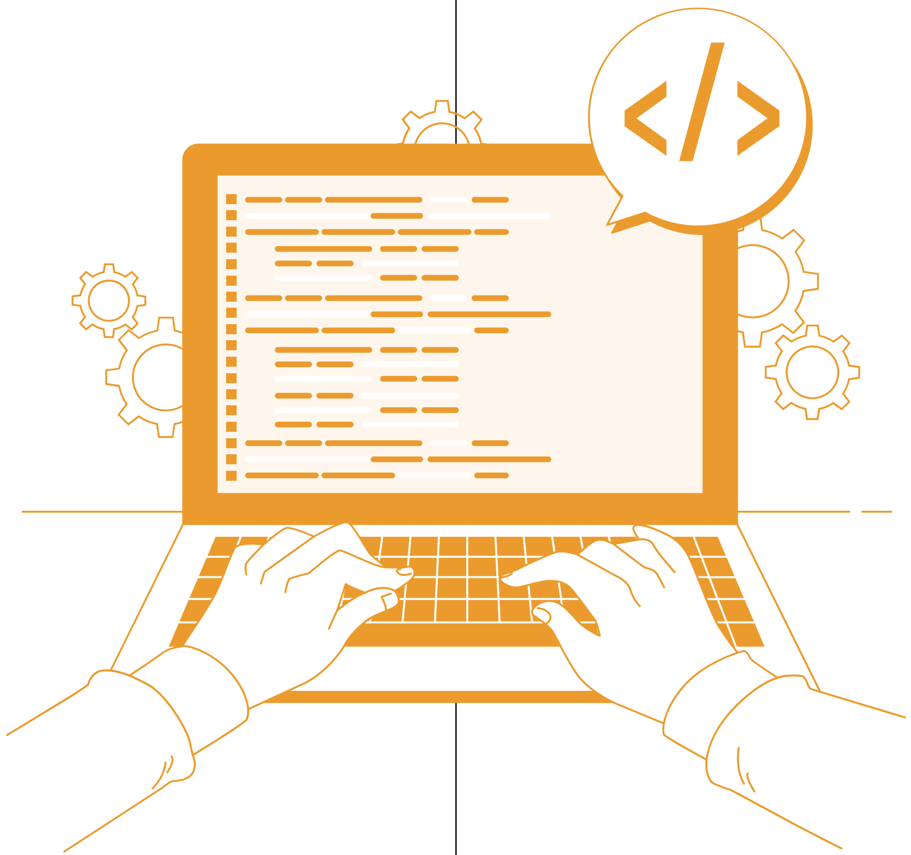
Variables (Dark Orange): Here we can create and edit the variables of our project. Variables are all the information we want to store, such as how many coins our character has collected, how much health is left, etc.

My Blocks (pink): "My Blocks", we can create our own blocks to reuse code. If there are pieces of code that we will repeat many times, we can turn it into a single pink block to make it easier for us to work. In programming, this is known as "functions". It is similar to "Events", although we will see the differences as we work with both. In programming these are known as “functions”. It is similar to “Events”, although we will see the differences when working with both.

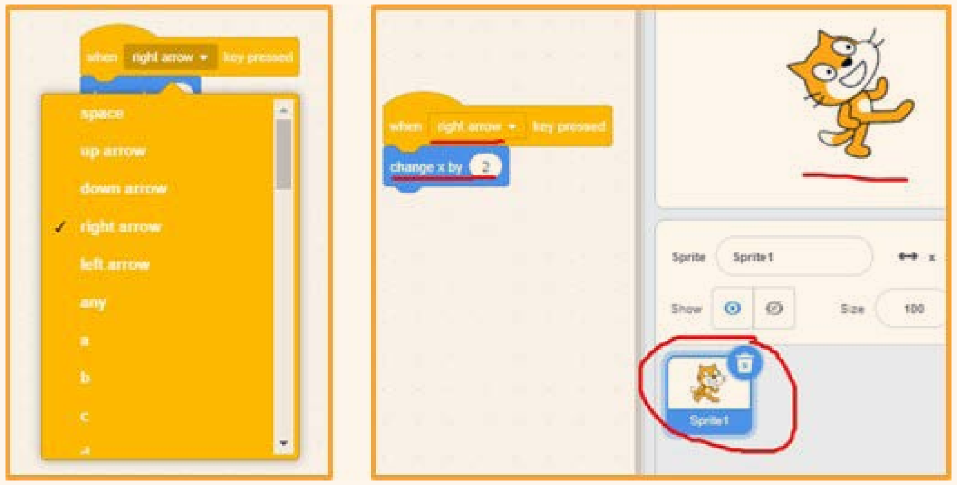
With this knowledge in mind, we start designing our prototype.

PROGRAMMING THE BASIC MOVEMENT

Before going any further, it is convenient to look at a practical example of how to program in Scratch. To do this, the student will make the Scratch pet move in a very basic way, he/she will face some problems that may arise and the student will understand why some blocks are used instead others. The first thing to do is to extract the necessary material from the blocks divided by colored balls that have been mentioned before and drag it to the central area of the project. The student will start with the **yellow ball "events",** then he/she will add the yellow box (these boxes are called blocks) that says "When [-----] key pressed" in the space where there are now hyphens. A drop-down menu can be expanded and select the option "When [right arrow] key pressed" then with the sprite of the selected character the student adds the **motion blocks** "Change X by [2]" and puts it right under the previous block. Now if you press the right key, the cat moves in that direction. The student can change the number to edit the speed at which it moves. It looks like he/she could use the "Move Steps" block, but this will make the character always move in the direction he's facing, which doesn't suit us for this project. The problem with using "When [right arrow] key pressed" is that if you press that key it will always move, whether you want it to or not. It is more convenient for us to separate when the student starts the game and when it ends, so that all the programming, such as the controls, works only while playing.



Extract the necessary material from the blocks



The student switches tasks, deletes the previous blocks, and places the "When Green Flag Clicked" event. This event is activated when the student presses the green flag on the game screen, marking the start of the game. If he/she presses the red signal, the game stops, stopping the code from running.

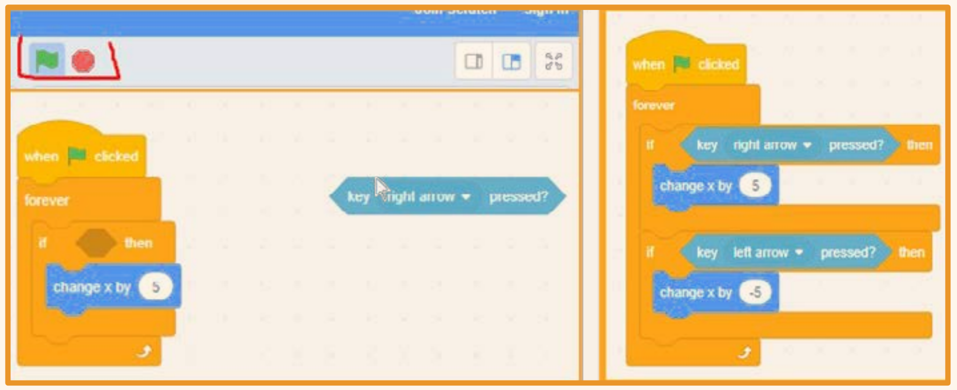
Next, place the "Forever" **control block.** It's a slightly different box, whatever the student puts in it will be executed continuously, many times per second.

Inside "Forever", the student place the **control blocks** "If [] then", inside this new node the student will add as before the **motion blocks** "Change X by [2]". This checks a condition, and if it is met then the character moves.

As the student can see, the boxes have shapes that connect to each other, to add the condition the student have just mentioned he/she must add the **sensing blocks** "key [] pressed?", which has a hexagonal shape, and he/she must introduce it into the hexagon inside the box "If [] then" leaving "if [right arrow] pressed".

Really, block programming is like writing orders. Looking at this the student can read that when starting the game, it continuously checks if the right key is pressed, and if so it moves the character in that direction. The student can repeat this process by duplicating the blocks within "Forever", this time putting the left arrow, and changing the speed by the same number in negative, so that it moves to the left.

Block programming

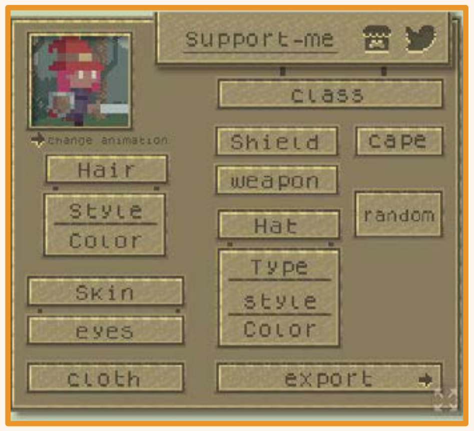


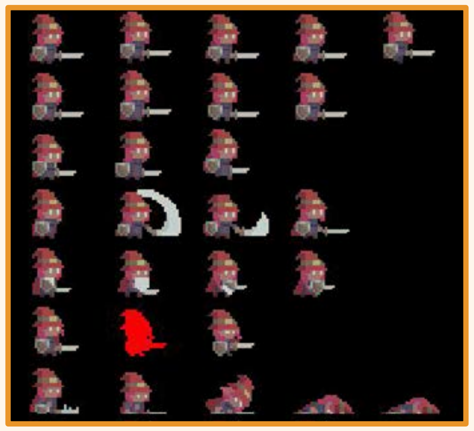
CHARACTER CREATION

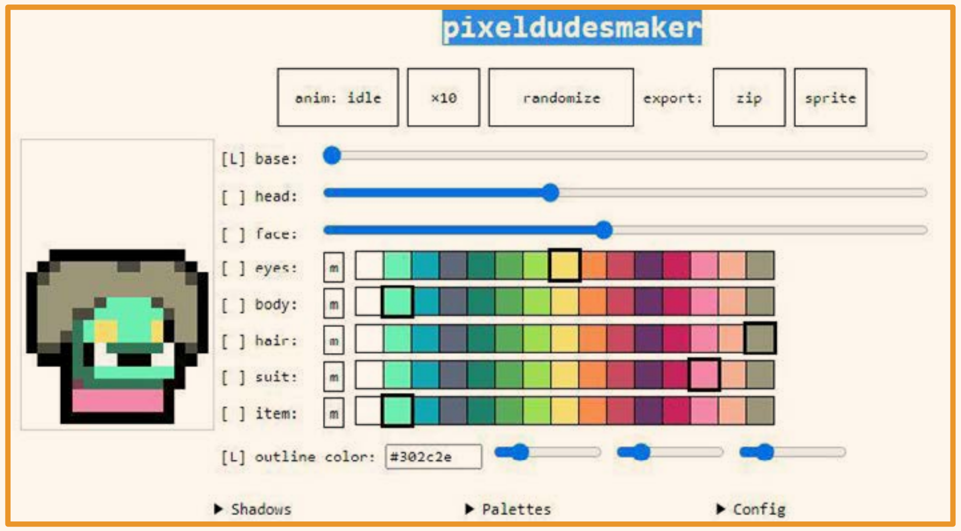
The student have two different tools to design our character, both equally valid: **pixeldudesmaker** and **pixel art rpg character creator.** Each has its own style and parameters for editing the character, and includes an option to create it randomly.

Now the student can have fun for a while trying out the different options and creating the character the student like the most. Once the student has the character, export it ("export: sprite" for pixeldudesmaker and "export/save" for Pixel Art Character Creator). This will create a sprite sheet, which is an image with all the animations of the character.

Creating the character







To import the character and its animations into Scratch the student will have to chop up that sprite sheet, so that he/she have each image of the animation separately. **Ezgif.com** allows the student to do it easily. Within the "Split" section is the "Sprite Sheet Cutter" tab, where he/she can drag our sprite sheet to chop it up.

After uploading the image and pressing "Upload", the student will access the image editor. He/she selects the "by columns/rows" method and puts the number of "columns and rows" that our image has so that it crops it correctly.

Make sure that the chosen format is PNG, click on "Cut!" and save it as a ZIP. When unzipping the images (with **7zip** or similar) it is advisable to make folders and save the two animations that are going to be use in them.

1) Idle (images 5-8 for pixeldudesmaker, and 7-10 for Pixel Art Character Creator): Animation for when the character does not move.
2) Walk (images 9-12 for pixeldudesmaker, and 1-6 for Pixel Art Character Creator): Animation for when the character moves.

IMPORTING THE CHARACTER

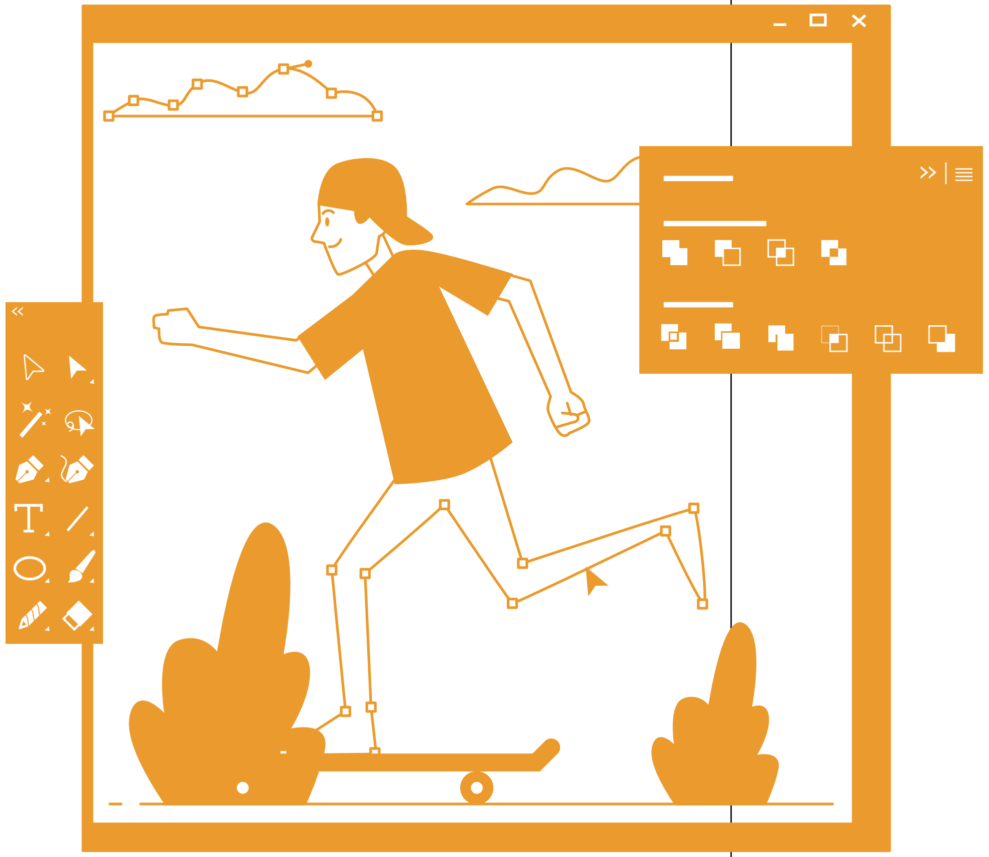
Within Scratch, at the bottom right where the list of Sprites and Backdrops is, there is a drop-down that allows to import the sprites.

The student imports the first image of the idle animation, which will create the sprite of his/her character. Now, in the "costumes" tab the student can find the same drop-down menu for import at the bottom left, and import the rest of the images from the idle.

To make it easier to work with, the student renames the first image of the idle to "Idle-1", and so on with all of them.

The student now takes the opportunity to import the images from the walk animation, and rename them in the same way.

With the character already imported, the student drags the code he/she created in the previous class for the Scratch character on the character's sprite to copy the code. Once done with this, he/she can now delete the Scratch cat to work only with the elements of our project.



ANIMATIONS

Since there are four motion inputs, one for each direction, but the student want the character do the walk animation in all of them, he/she is going to create a different block to not repeat the same code all the time.

Create a block in the **"My Blocks"** section. the student can rename it Walk_Anim, and the student is going to enable the bottom tab to run without screen refresh (this automatically creates a block called "define [Walk_Anim]" that we'll use later).

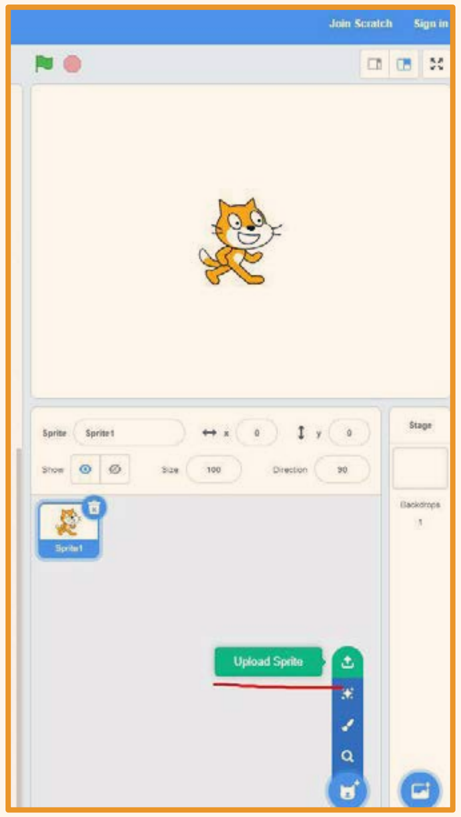
The student add this new block inside each of the "If [] then " **control blocks** at the end of the code.

But in order to manage the animation, you'll have to know if the character was already walking

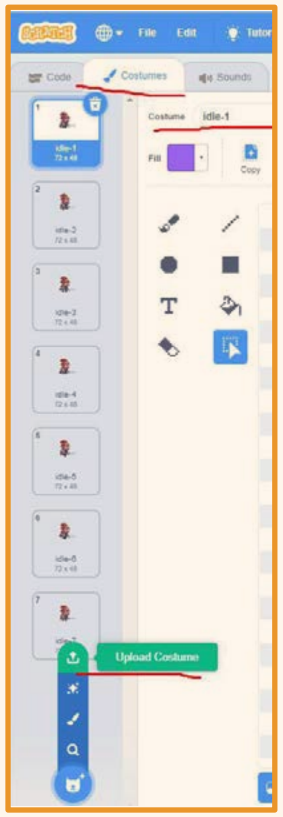
or not, so within the **"Variables"** section the student create one, just for this sprite, which the student will call "Current Costume". You can remove the tick that is in the list of variables so as not to view it on the game screen.

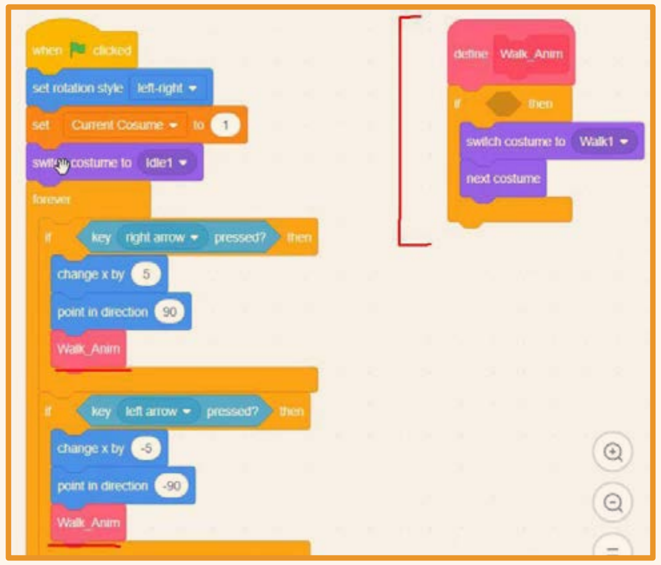
This variable represents which image the character now has, out of all the ones we've imported. Usually it is wanted to always start still when the user starts the game, so under the **motion blocks** "set rotation style [left-right]" can be placed another **variable block,** "set [Current Costume] to [0]", and another **looks,** "switch costume to [idle 1]".

Import the sprites



Animation of the character





```
when [flag] clicked
set rotation style [left-right]
set [Current Cosume] to (1)
switch costume to [idle1]
forever
    if <key [right arrow] pressed?> then
        change x by (5)
        point in direction (90)
        Walk_Anim
    if <key [left arrow] pressed?> then
        change x by (-5)
        point in direction (-90)
        Walk_Anim

define Walk_Anim
if < > then
    switch costume to [Walk1]
    next costume
```

The most intuitive way to approach animations would be with an "if [ ] then", checking if the character was already doing the walking animation, so if so, it continues, and if not, it starts from the beginning.

But it is going to be used a mathematical formula to calculate it automatically, improving the performance of the project. The student will need the "switch costume to [ ]"**looks,** the "Current Costume" **variable,** and **the operator blocks** "[ ] + [ ]", "floor of [ ]" (you can puts ABS instead of floor or similar since it is a drop-down menu) and "[ ] mod [ ]". The set of blocks would be something that:

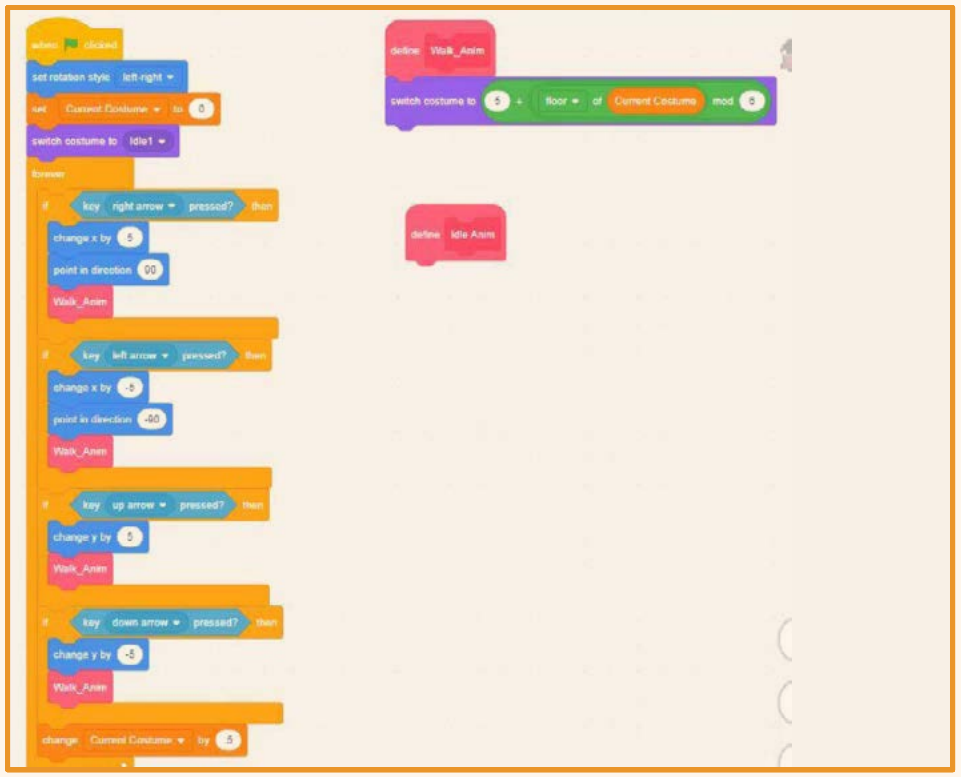
"Switch Costume to [[5] + [[floor of [Current Costume]] mod [6]]]"

5 is the image in which the walking animation begins, and 6 is the number of images that It has that animation, as it goes from 5 to 10. All of this can be seen in the "Costumes" tab.

"Floor" eliminates the decimals, while "mod" makes sure that the first number the student enter never exceeds the second, and finally he/she is going to use "Current Costume" to calculate the advance of time.

Once placed these blocks under the "define [Walk_Anim]" (which was automatically created before), then has to be introduced at the end of the code previously created inside **"forever"** the **variable block** "Change [Current Costume] by [0.5]". With this last number can be chosen the speed at which the animations have to go.

Walking animation



```
when 🏁 clicked
set rotation style [left-right ▾]
set [Current Costume ▾] to (0)
switch costume to [Idle1 ▾]
forever
    if <key [right arrow ▾] pressed?> then
        change x by (5)
        point in direction (90)
        Walk_Anim
    if <key [left arrow ▾] pressed?> then
        change x by (-5)
        point in direction (-90)
        Walk_Anim
    if <key [up arrow ▾] pressed?> then
        change y by (5)
        Walk_Anim
    if <key [down arrow ▾] pressed?> then
        change y by (-5)
        Walk_Anim
    change [Current Costume ▾] by (.5)

define Walk_Anim
switch costume to ((5) + (([floor ▾] of (Current Costume)) mod (6)))

define Idle Anim
```

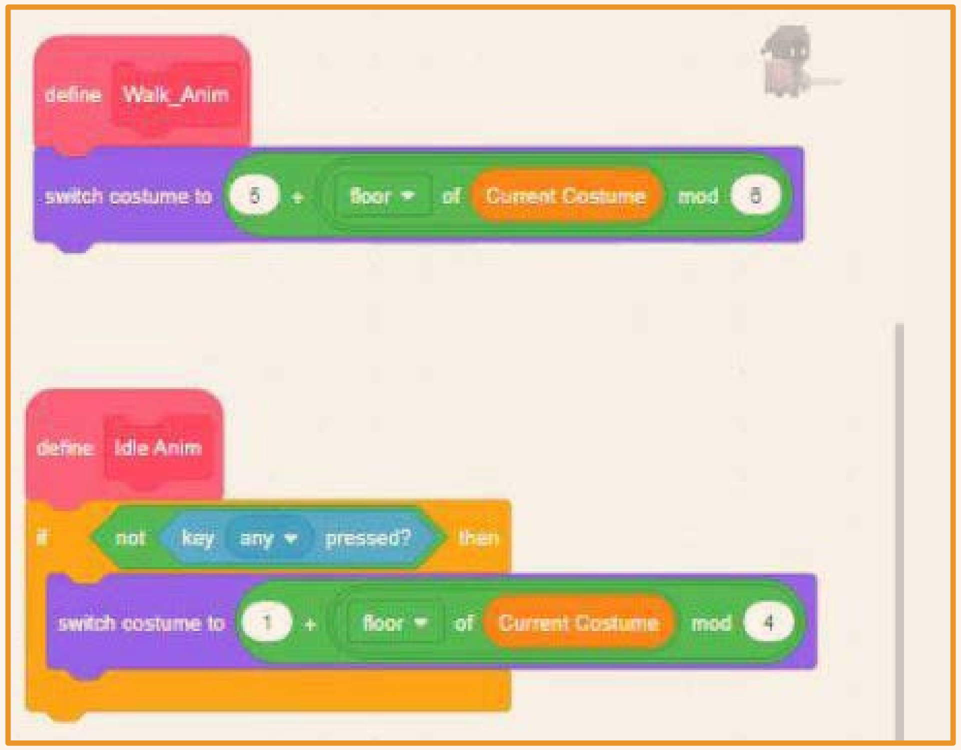
The student can also create the Idle animation. To do this the student will create from **"My Blocks"** a block called Idle_Anim, again he/she will make it run without image refresh. When defining it, the student have to check if the character is not moving, in order to execute the animation. To do this, the student will puts a **control block** "If [ ] then", and in the tester (the diamond between if and then) the student will puts an **operator block** "not [ ]", and inside the space of the "not [ ]", a **sensing block** "key [any] pressed?" (In any you can puts space or another word since it's a drop-down menu.) The student puts s them together so that the block looks like:

"If [not[key [any] pressed] then"

Inside this "If [] then " the student puts s the same operation he/she did for the walking animation, changing the values of the start image and the number of images that the idle animation has, so that it looks like this:

"Switch Costume to [[1] + [[floor of [Current Costume]] mod [4]]]"

The Idle animation



```
define Walk_Anim
switch costume to ((5) + (([floor ▾] of (Current Costume)) mod (6)))

define Idle Anim
if <not <key [any ▾] pressed?>> then
    switch costume to ((1) + (([floor ▾] of (Current Costume)) mod (4)))
```

COLLISIONS AND OBJECTS

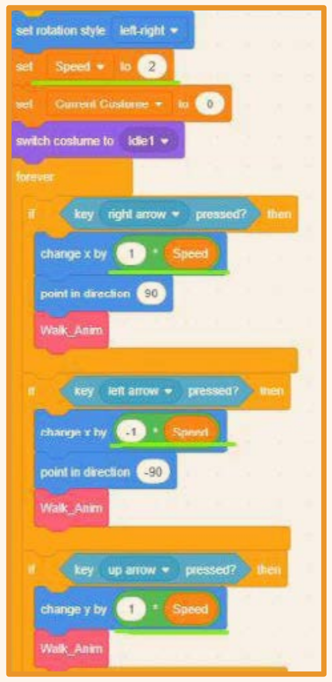
PROPERLY STARTING THE CHARACTER

To make it easier to modify the character's speed, we're going to create just for this sprite a **variable** that we'll call "Speed". After the initialize **events** the student puts the **variable block** "set [speed] to [2]", and in each **blocks** "change x by[ ]" and "change y by [ ]" that the student puts for the move, in those gaps "[]" the student puts this variable speed, next to the **operator block** multiplication, "[ ] * [ ]". In the up and right movement the student will multiply Speed by 1, and in the left and down the student will multiply Speed by -1.

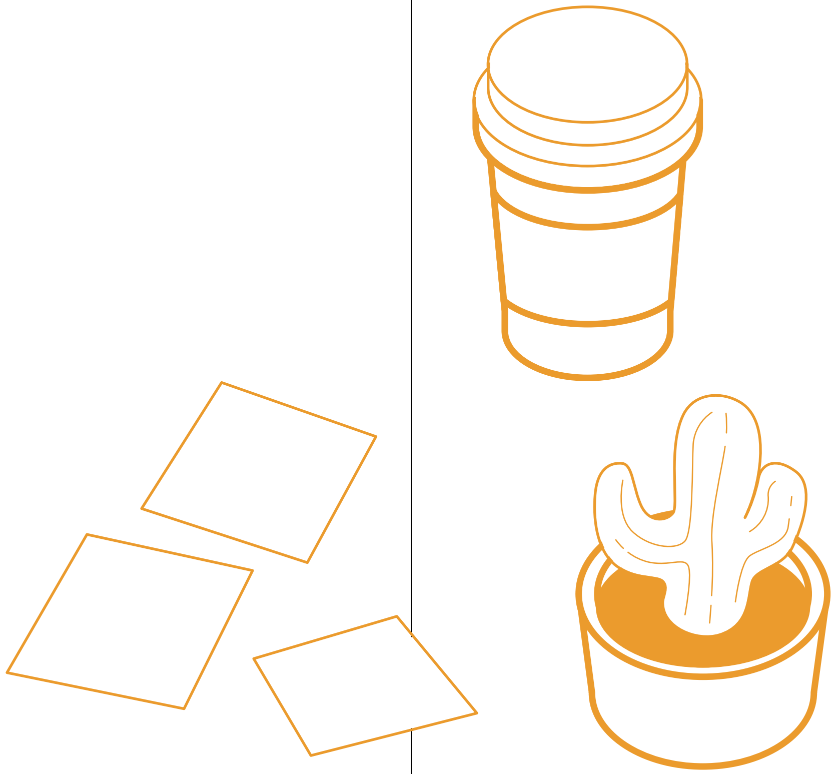
In addition, it is convenient to puts after the event of starting the game **the blocks** looks "show", "switch backdrop to [floor_tiles]" and "go to [front] layer". This way the student make sure that the student start on the map it touches on and that the character will always be visible.

It is also advisable to puts behind these blocks when starting the game the **motion blocks**"go to x [ ] y [ ]", where the student will puts the coordinates where the student want our character to always start.

Modify the character's speed



```
set rotation style [left-right ▾]
set [Speed ▾] to (2)
set [Current Costume ▾] to (0)
switch costume to [idle1 ▾]
forever
  if <key [right arrow ▾] pressed?> then
    change x by ((1) * (Speed))
    point in direction (90)
    Walk_Anim
  if <key [left arrow ▾] pressed?> then
    change x by ((-1) * (Speed))
    point in direction (-90)
    Walk_Anim
  if <key [up arrow ▾] pressed?> then
    change y by ((1) * (Speed))
    Walk_Anim
```



REDO CHARACTER MOVEMENT

Right now, the character can walk through the walls of the stage. To avoid this, it has to be created a collision so that he/she can crash into them.

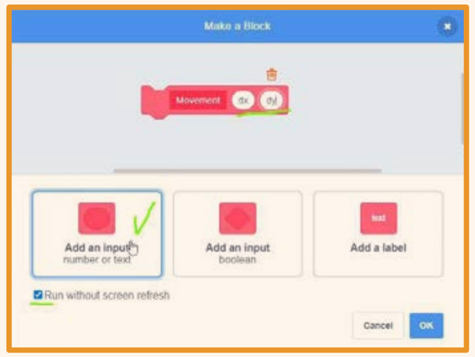
Start by creating another block from **"My Blocks"**, also without screen refresh, called Movement, but this time we'll add one of the options offered by Scratch "add and inputs, number or text". These two inputs s will be called "dx" and "dy", and they represent the speed of the character on each of the axes.

We're going to replace the "Change X by [ ]" blocks that the student used for the movement with this new "Movement" block that we've created. The value the student used, the **blocks**"[Speed] * [1]" for the right movement and "[Speed] * [-1]" for left, the student puts them in the first entry of the Movement **blocks,** and in the second the student puts a 0. For example, for the movement to the right it would look like this:

"Movement [[Speed]*[1]] [0]"

This is because the student will make the first value the speed with which he/she moves on the axis horizontally, and the second with which the student move vertically. Now the student repeat the process to replace the "Change Y by [ ]" blocks that the student have used for vertical movement, by puts ting their value in the second entry of the "Movement"**blocks,** leaving the first as 0. For example, this time the upward movement would look like this: "Movement [0] [[Speed]*[1]]".

Create a collision



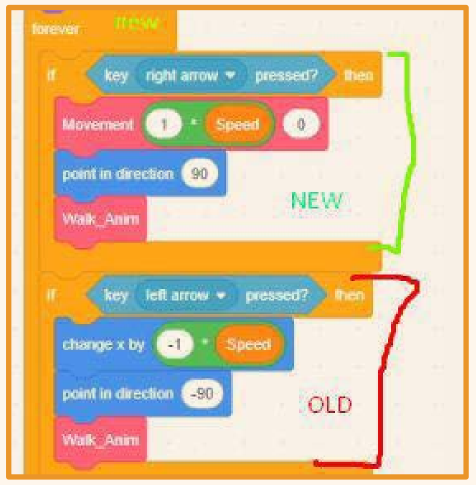
Make a Block

Movement (dx) (dy)

Add an input number or text ✓	Add an input boolean	Add a label

[✓] Run without screen refresh

Cancel OK



```
forever
  if <key [right arrow ▾] pressed?> then
    Movement ((1) * (Speed)) (0)
    point in direction (90)
    Walk_Anim
  if <key [left arrow ▾] pressed?> then
    change x by ((-1) * (Speed))
    point in direction (-90)
    Walk_Anim
```

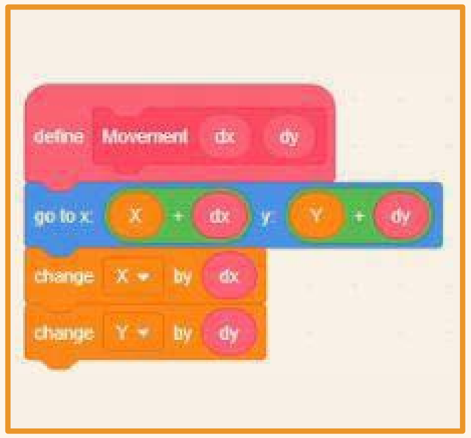
With this the student are grouping the movement of the character in a single box. In principle, it forces us to calculate the movement in a slightly more complicated way, but it will help us manage collisions. Next, the student creates two **variables** just for this Sprite, which we'll call "X" and "Y". With them, the student will mark the position of the character at all times. Under the initialize **events** the student will puts the **variable blocks** "Set X to [ ]" and "Set Y to [ ]", and assign them the

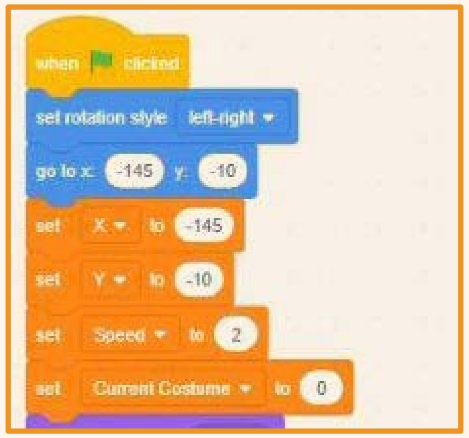
Now going back to the block **Defines "Movement [dx] [dy]".** The student can drag the "dx" and "dy" inputs s to be able to work with them, which represent the values the student apply each time the student have used this block. Let's define it by placing the **block of movement** "go to x [ ] y [ ]".

What the student want to do is add the movement to the character's current position, and update it. To do this, within the new block the student will add **the "[]+[]" of operations** in both slots and the student will use our variables "X" and "Y" (respectively) to occupy the first slot of the new operations box, and the student will add the entries of "dx" and "dy (respectively) that occupied the second slot of that box. The result would be something like: "go to x[[X]+[dx]] y[[Y]+[dy]]" Next to this the student puts the **variable block** "Change [X] by[ ]", and as a parameter in the gap the student add "dx", just below it the student puts the **variable block** "Change [Y] by[ ]", and as a parameter "dy".

With all this the student can see that the movement of the character feels exactly the same as it did before, with the difference that the student has unified it within a single block that the student can easily edit.

Unified movements within a single block


```
define Movement (dx) (dy)
go to x: ((X) + (dx)) y: ((Y) + (dy))
change [X] by (dx)
change [Y] by (dy)
```


```
when [flag] clicked
set rotation style [left-right]
go to x: (-145) y: (-10)
set [X] to (-145)
set [Y] to (-10)
set [Speed] to (2)
set [Current Costume] to (0)
```

CHARACTER COLLISION

Now that the student has unified the movement, the student is going to make the character stop when hitting the walls of our map. Between the "go to x [ ] and [ ]" and "Change [X] by[ ]" blocks of the movement, the student place **control block** "If [ ] then", as a parameter in the space the student enter **sensing block** "Touching [Map]?", to execute inside the if the character touches our map sprite.

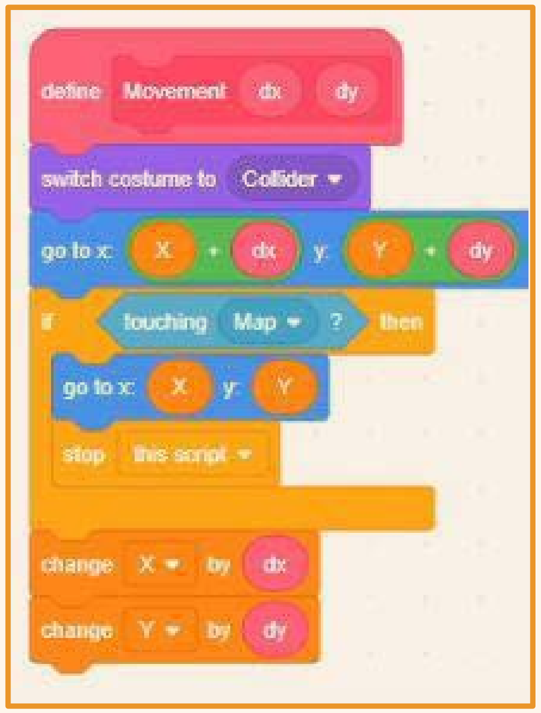
Inside of the **if** the student are going to introduce **motion block** "go to x [ ] and [ ]", and the student assign our **Variable** "X" and the and the **variable** "And", and following the **control block** " Stop [this Script]". With this the student make it so that when hitting the walls our character does not advance, and stops moving.

But if our character carries a sword or any other element, it will collide. To avoid this the student are going to create a Collider, which will be the only part of the character that collides with everything. Within the Costumes tab, the student duplicates the first image of the Idle and drag it to the end of the image list.

The student creates a rectangle above it that occupies the size that the student wants it to collide with things, and the student erase all the excess parts of the character around it with the eraser. The student will call this image Collider.

To apply it, in the code **defines "Movement [dx] [dy]",** the student puts before the rest a **block looks** "Switch Costume to [Collider]." What this does is puts the image of the collision to calculate when it hits the environment, and automatically change it to the one the student wants it to be seen, so fast that the student does not even notice.

The character stops when hitting the walls


```
define Movement (dx) (dy)
switch costume to [Collider]
go to x: ((X) + (dx)) y: ((Y) + (dy))
if <touching [Map] ?> then
    go to x: (X) y: (Y)
    stop [this script]
change [X] by (dx)
change [Y] by (dy)
```

OBJECTS

Now that our character has collision, he/she will be able to interact with other elements of the environment. For the environment the student is going to take elements from Dungeon Tileset II.

The student searches among its files for the coin_anim, a coin, and import it, along with all the images for its animation, as the student already did with the character. The student gives it a scale of 120 to adjust it to the size of the game.

The currency will also start working when you start the game, so the student added an **event block** "When Green Flag Clicked." The student creates a **variable** only for this sprite called "Current Costume", and under the event of starting the game the student puts the **variable block** "set [Current Costume] to [0]", and **block looks** "switch costume to [coin_anim_f1]" and "Show". Next the student puts a **control block** "Forever".

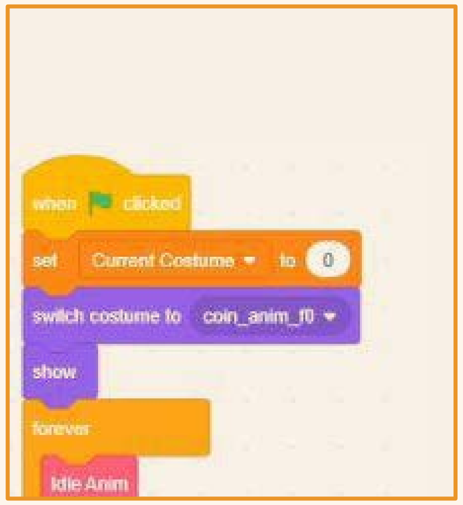
Now in **"My Blocks"** The student create a block called "Idle Anim", without a screen refresh, and place it inside the forever. Following the "Idle Anim" the student puts the **variable block** "change [Current Costume] by [0.2]" to say how fast the animation is going.

Now inside the **define block** "Idle Anim" the student puts the same series of blocks that the student already did with the character, with the appropriate values for the start image and the number of animation images, being something like this:

"Switch Costume to [[1] + [[floor of [Current Costume]] mod [4]]]"

.

Interact with other elements of the environment



```
when 🏴 clicked
set Current Costume ▾ to 0
switch costume to coin_anim_f0 ▾
show
forever
    Idle Anim
```

As the student can see, the same thing the student did for the character works for any animated element.

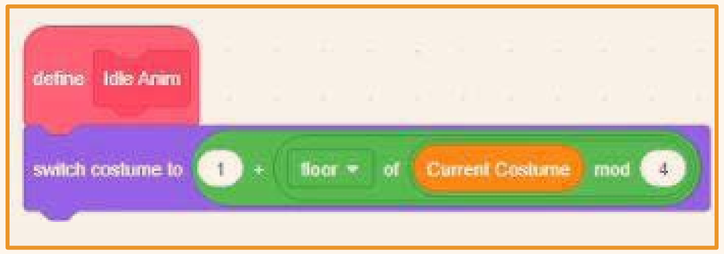
To manage animations, you will always have to repeat the same process.

The student only has to implement what happens when the character picks it up. The student creates a **variable** for all sprites called "Coin Amount". It is convenient for us to deactivate the tick of the rest of the variables, also of our character, and leave only this one active, to be able to see at all times the amount of coins the student has.

Inside of the **block** "Forever", above the other two, the student puts a **block** "If [ ] then", and how verification the student puts a **sensing block** "is touching [character]?" Inside the if the student puts the block. "Change [Coin Amount] by [1]", from **block looks** "Hide", and **of control** "Stop [this Script]". With this the student make it so that if the character touches it, the amount of his coins' increases, the student hide this coin so that it cannot be seen, and the student make it stop being able to be picked up again.

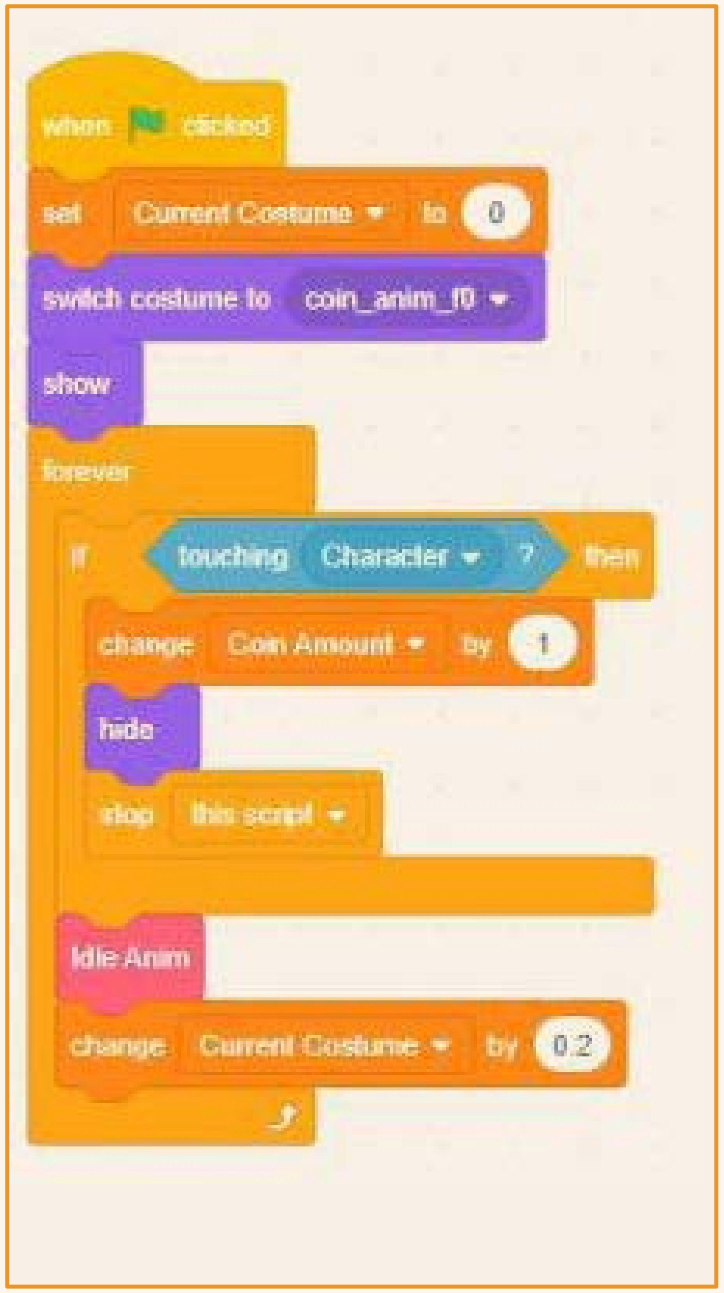
Finally, to make sure that our character always starts without coins, the student go back to his code and under the event of starting the game the student puts the **variable block** "Set [Coin Amount] to [0].

Manage objects animations



```
define Idle Anim
switch costume to (1 + (floor ▾ of (Current Costume mod 4)))
```

Manage objects animations



```
when 🏴 clicked
set Current Costume ▾ to 0
switch costume to coin_anim_f0 ▾
show
forever
    if touching Character ▾ ? then
        change Coin Amount ▾ by 1
        hide
        stop this script ▾
    Idle Anim
    change Current Costume ▾ by 0.2
```

DIALOGUES

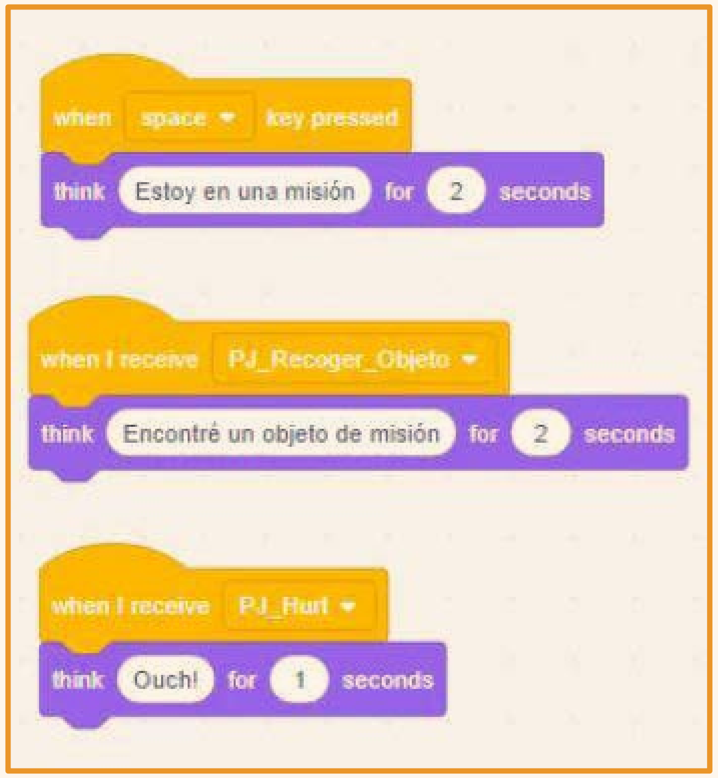
USE DIALOG BLOCKS

Scratch has a very integrated dialogue system, making it very easy to apply. A direct and simple example is, returning to our character code, puts ting other **block**"If [ ] then" inside the "Forever", to which this time the student add the **condition sensing**" is [spacebar] pressed?" Inside that "if" the student puts the **block looks** "think [ ] for [2] seconds", to which the student can puts the text that the student want the character to think when pressing the space bar.

The student can also have the character say something when picking up an object. For that the student add to the character code an **event block** "When I Receive [message]." Instead of message the student creates a new event, which the student will call "PC_Pick_Item". Now, below this event the student place the **block** "think [ ] for [2] seconds", with the text that the student want the character to think when picking up the object.

Now from the currency code, the student adds the **event block** "Broadcast [PC_Pick_Item] right above where the student placed the **block** "Stop [this Script]".

Apply dialogues



```
when space ▾ key pressed
think Estoy en una misión for 2 seconds

when I receive PJ_Recoger_Objeto ▾
think Encontré un objeto de misión for 2 seconds

when I receive PJ_Hurt ▾
think Ouch! for 1 seconds
```

NON-PLAYER CHARACTERS

Now that the student knows how the basic dialogue blocks work, the student can create other characters that interact with ours. The student imports a new character with its idle animation, which the student will call NPC. This can be created with the *pixeldudesmaker* or the Pixel Art Character Creator, or you can also choose one of the ones in the Dungeon Tileset II.

As with our main character, the student created the **variable** "Current Costume" only for this sprite, and the student add the **event** "When Green Flag Clicked", followed by **block** "Set [Current Costume] to [0]", and **block looks** "Switch Costume to [idle_0]" and "Show", from **motion block** "Set Rotation Style [left-right]" and "Forever".

The student believes in **my block** "Idle_Anim" without screen refresh. In the **definition of this block** the student would puts the animation formula, whose values will depend on our idle animation:

"Switch Costume to [[1] + [[floor of [Current Costume]] mod [4]]]"

Now the student puts inside the "Forever" our **block** "Idle_Anim", and the **variable block** "Change [Current Costume] by [0.15]".

With this the student have the NPC with its animation, now the student wants to make it interact with our character. For that, inside the "Forever", after the blocks that the student already have, the student puts a **block** "If [ ] then", and how **condition** the student puts "Touching [Character]?"

With this we have the NPC with its animation, now we want to make it interact with our character. For that, inside the "Forever", after the blocks that we already have, we put a **block** "If [ ] then", and how **condition** we put "Touching [Character]?"

In this way it will react when the student approach it. The next thing the student want is that if the student carries objects with us, receive them and give us a message, and if the student does not have them, ask us for them with a different message.

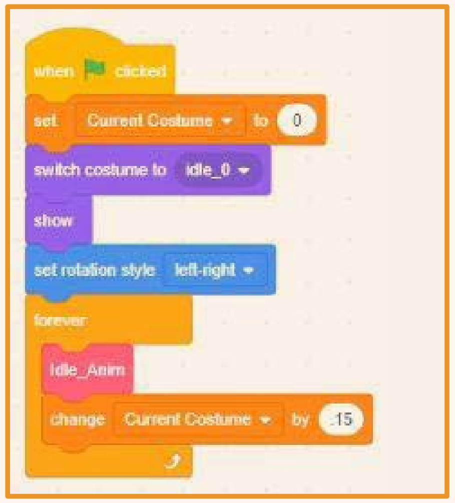
For that, inside that "If [ ] then" the student puts another **control block,** "If [ ] then else". This does that if the condition is met, execute a piece of code, and if not execute a different one. As condition the student puts "[Coin Amount] > [0]", using our **variable** "Coin Amount".

Within the first part of this "If [ ] then else" the student puts **variable block** "Change [Coin Amount] by [-1]", and **looks** "Say [ ] for [4] Seconds", where the NPC will be grateful that the student have given the object.

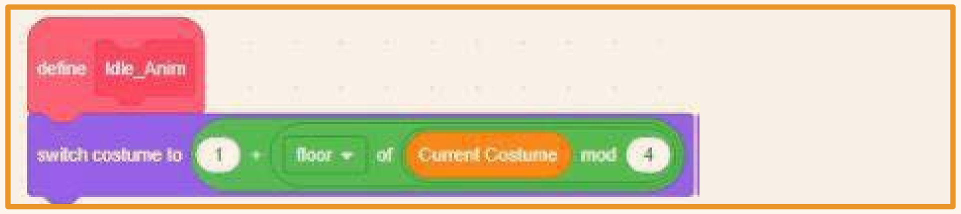
In the second part of this "If [ ] then else", which happens when the student don't have any objects, the student puts only the **block looks** "Say [ ] for [2] Seconds", where the NPC will ask us to go look for the object.

The student already has the NPC working, his only problem is that the student can walk over him. The student is going to try to make their collision faster and easier, since this NPC is not going to move from the site.

Creation of other characters that interact with the main character

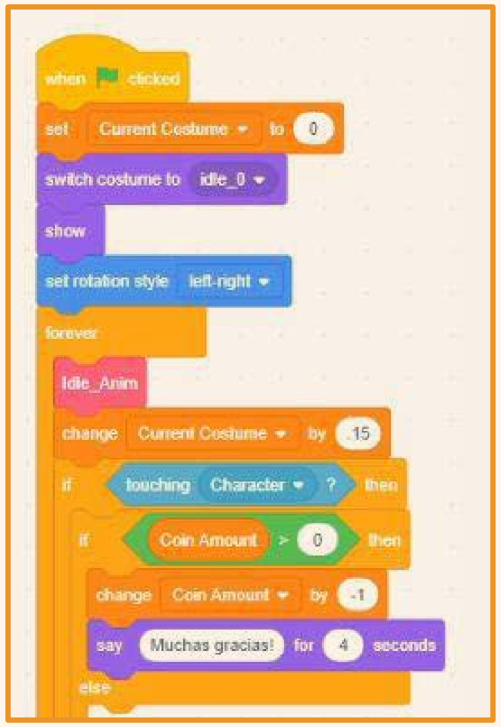


```
when 🏁 clicked
set Current Costume to 0
switch costume to idle_0
show
set rotation style left-right
forever
    Idle_Anim
    change Current Costume by .15
```



```
define Idle_Anim
switch costume to (1) + (floor of (Current Costume mod 4))
```

Create a message from the other characters



```
when 🏁 clicked
set Current Costume to 0
switch costume to idle_0
show
set rotation style left-right
forever
    Idle_Anim
    change Current Costume by .15
    if touching Character ? then
        if Coin Amount > 0 then
            change Coin Amount by -1
            say Muchas gracias! for 4 seconds
        else
```

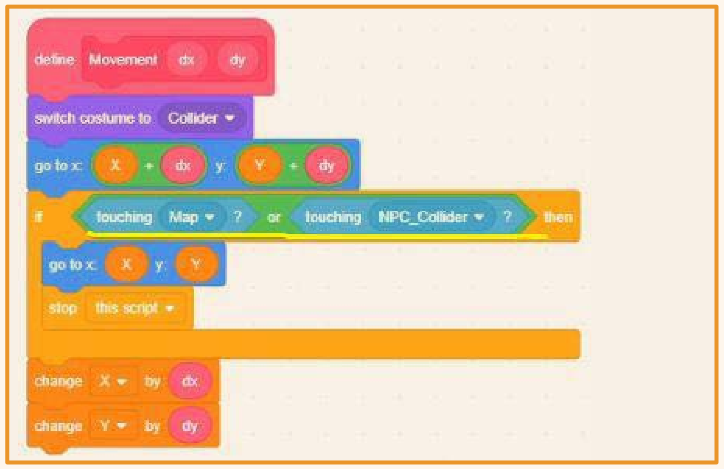
Where the characters are in the lower right frame of the screen, the student click on the cat-shaped icon and select paint. The student creates a new sprite that is a rectangle slightly smaller than NPC (the student will see in real time how it looks on the game map, the student must make it fit over the NPC), which the student will call NPC_Collider. Ideally this collision would have to be centred in the image editor so that it is easier for us to work with it.

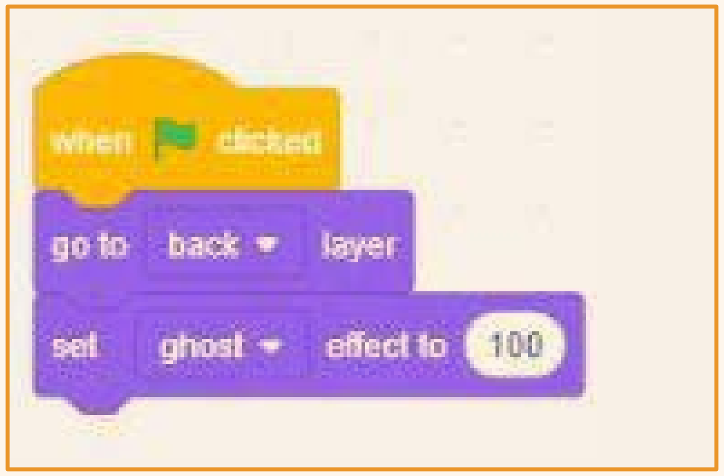
The student places it on the map above the NPC to see that it fits correctly. Now, the student will create the NPC_Collider code by adding the **block** "When Green Flag Clicked", and below **blocks looks** "Go to [Back] Layer", and "Set [Ghost] Effect to [100]". This way it will be there, but it will not be visible.

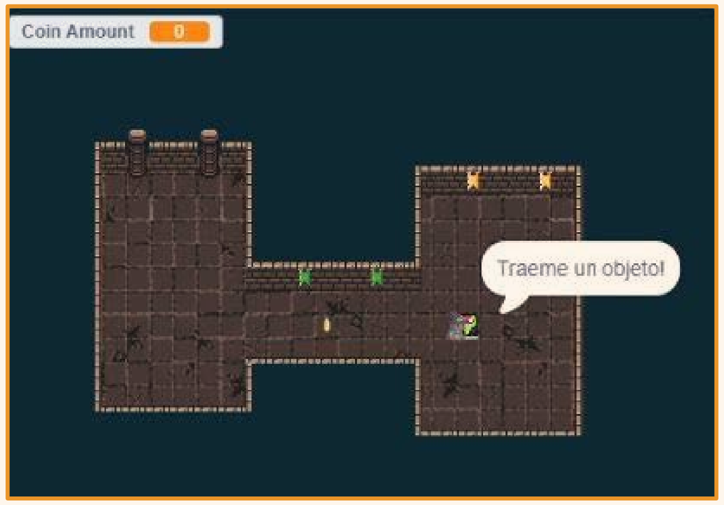
Now for this collision to affect our character, the student returns to the Character code. Where **the student defined the block** "Movement [dx] [dy]" the student see the **verification** from "If [ ] then" which was "touching [Map] ?". Let's add the **operator** "[ ] or [ ]" to check if it touches the NPC collision as well, so that it would be:

"[Touching [Map] ?] or [Touching [NPC_Collider] ?]"

Collision with another character that asks for an object


```
define Movement (dx) (dy)
switch costume to [Collider ▾]
go to x: ((X) + (dx)) y: ((Y) + (dy))
if <<touching [Map ▾] ?> or <touching [NPC_Collider ▾] ?>> then
    go to x: (X) y: (Y)
    stop [this script ▾]
change [X ▾] by (dx)
change [Y ▾] by (dy)
```


```
when ⚑ clicked
go to [back ▾] layer
set [ghost ▾] effect to (100)
```



5 FINAL TIPS FOR TRAINERS

In conclusion, as a trainer, it's essential to allocate time to provide clear instructions and ensure that everyone is following along at each step of the way. Offer support to any student encountering difficulty, fostering an environment of inclusive learning.

Scratch serves as an excellent tool for experimentation, enabling students to explore various programming approaches with ease. This flexibility is particularly beneficial; it allows advanced students to forge ahead and explore independently.

As the session draws to a close, encourage participants to showcase their projects to their peers. Prompt them with questions to stimulate reflection: What observations did you make while testing the games? What creative ideas might the student incorporate into his/her own project?

Above all, aim to cultivate a sense of enjoyment and fulfilment during the sessions. Encourage students to create their Scratch accounts, empowering them to continue their game development journey beyond the classroom walls. By embracing curiosity and perseverance, each student can unleash their creativity and thrive in the world of programming.

Bibliography

Olcina, Beatriz, *Guía de iniciación al diseño y desarrollo de videojuegos en Scratch,* Fundación MUSOL, 2023.

Scratch.mit.edu, "Scratch para educators", https://resources.scratch.mit.edu/www/guides/en/EducatorGuidesAll.pdf (last consultation on june 2024)

Publisher
BIS Netzwerk für betriebliche Integration und Sozialforschung e.V. (BIS), Gesellschaft für duales Lernen gGmbH (GDLB)

www.bisev-berlin.de
www.access-youth.eu

Project partners
BGZ Berliner Gesellschaft für internationale Zusammenarbeit mbH, Südwind - Verein für Entwicklungspolitik und globale Gerechtigkeit (Südwind), Die Wiener Volkshochschulen GmbH (VHS), Fundación MUSOL (Musol), Federació Valenciana de Municipis I Províncies (FVMP)

Authors
Fundación MUSOL and Federació Valenciana de Municipis / Províncies

Donors
European Union – Erasmus+ programme

Co-funded by the European Union

Graphics
Copyright © 2024 Scratch Foundation

Valencia, 2024

www.bgz-berlin.de

www.bisev-berlin.de
www.suedwind.at

www.vhs.at
www.musol.org
www.fvmp.org

WWW.ACCESS-YOUTH.EU