CHAPTER ONE

INTRODUCTION TO PROGRAMMING, AND PROGRAMMING WITH SCRATCH

Chapter Objectives

At the end of the chapter, the students should be able to;

- **understand how to create a computer program.**
- **use Scratch to create computer programs.**
- **create a Scratch project.**
- **create a basic computer program with Scratch**

What is Computer Programming?

A computer is a device that stores and processes (acts on) data and instructions given to it, usually by a human. The primary function of a computer is to carry out an instruction or a set of instructions. For example, if you want to take down a note with an application called "Notepad", on your computer, you simply open up the computer, look for the Notepad application icon, double-click (click twice) on the application icon to open it, and start taking down your notes. What was just described is a set of instructions you give to the computer because you want to open an application. Let us look at these instructions one by one:

- Open up the computer.
- Look for the Notepad application.
- Double click on the Notepad application icon.
- Start taking down notes.

This set of instructions given to a computer to perform specific actions is called a computer program. A computer program is used to instruct the computer to carry out some instructions, and whoever writes the computer program, that is whoever gives the computer the set of instructions to carry out, is called a computer programmer. The process of writing a computer program and running it on a computer, that is writing a set of instructions and giving it to the computer to execute, is called **Computer Programming**. Computer programs are also called **applications**.

One important thing to note is the order in which the computer carries out the set of instructions listed above; it starts with the first instruction, then moves to the next one and carries it out, and then the next one, until it gets to the end, at that point, we say the program has ended. This order of carrying out instructions is called **sequential**, i.e. one after the other. As we go through this textbook, keep in mind that the computer programs we will be writing will carry out their instructions sequentially, that is, one after the other in the order in which we write the program.

Human beings need a language to speak to each other, be it English or other languages. In the same way, computers need to be spoken to in a language they understand. To write a set of instructions, that is, a computer program, you need a **programming language**. Humans speak **words**, while programmers write **code** for the computer to execute.

There are numerous programming languages available today, and we will be focusing on one throughout this textbook: **Scratch**.

Introduction to Scratch

Scratch is a programming language that allows us to code computer programs using visual components. Instead of typing out code using full words, you use a set of predefined blocks and other visual components to build an application. Scratch is very easy to use and understand, it allows you to create many types of games and fun applications that you can share with your friends.

We will create a lot of games and applications with Scratch throughout this textbook, but before we get started, let us look at how we might write an application that simply speaks some words.

For our first-ever computer program, we will be making an application that speaks out your full name. For this program, assuming your name is “Peter Piper”, we will have a single instruction which is:

- Say Peter Piper.

Starting a Scratch Project

To create a Scratch Project, simply launch the Scratch 2 application on your computer. If this is your first time of opening the app, an empty project will be automatically created for you.

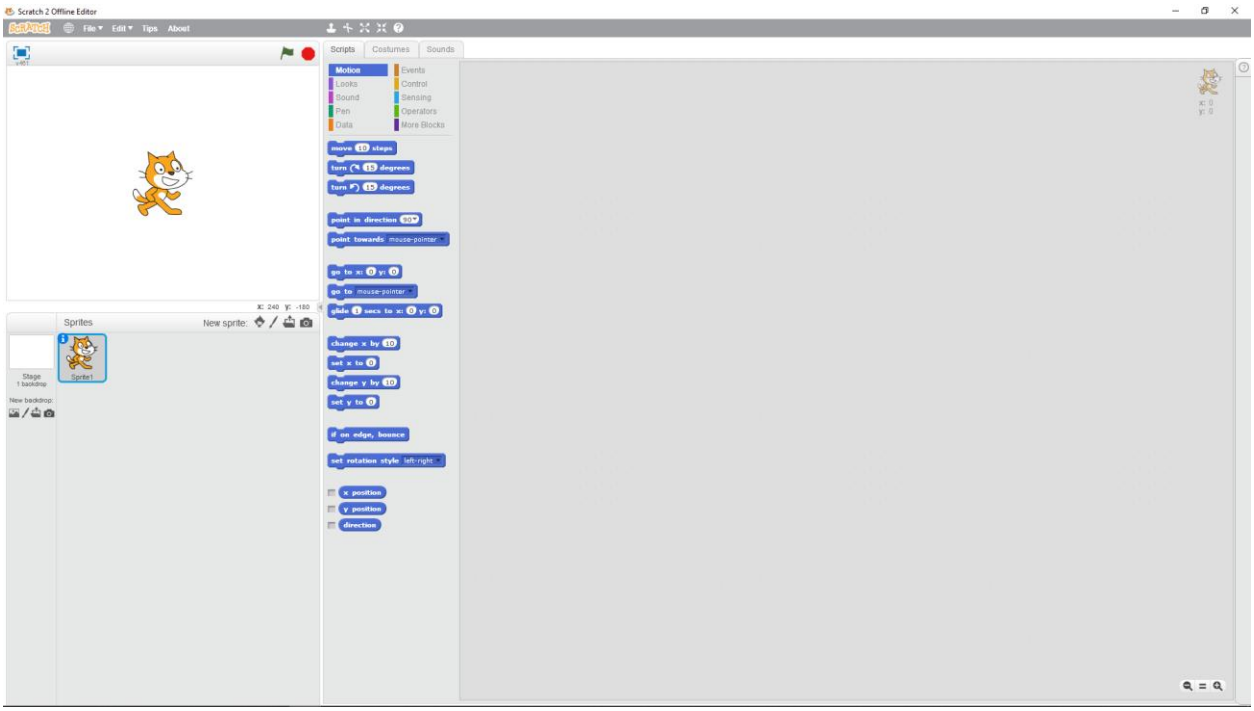


Figure 1.0

A project is your work environment for creating a single program.
Now, the screen that opens up might look overwhelming as there are different sections, but we will get around it in a bit. For now, we just want to create something cool to see.

The section to the left that contains the cat image is where the result of running your program, that is, executing the set of instructions that you specify in your program, is displayed. The cat has been added automatically for you, so, we will program the cat to say our name. Later on, we will learn how to create and add other things apart from the cat to our program.

Remember that our simple program contains just one instruction. Here it is again:

- Say Peter Piper (replace Peter Piper with your name).

To specify an instruction for our program, we use the blocks contained in the section to the right of the section containing the cat. These **blocks** help us control what happens in our program by dragging them to the empty space at the right side of the screen.

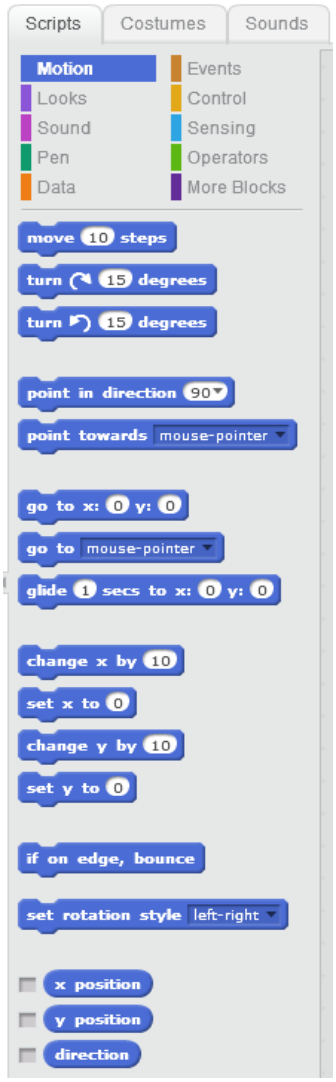


Figure 1.1

Now we have to look for a block that can do what we want, which is to say something. The block we need is in the “Looks” section, so click on “Looks” on the upper pane and look for a block that says “say Hello!” Then click and drag this block to the blank space on the right side of the screen.

Figure 1.2

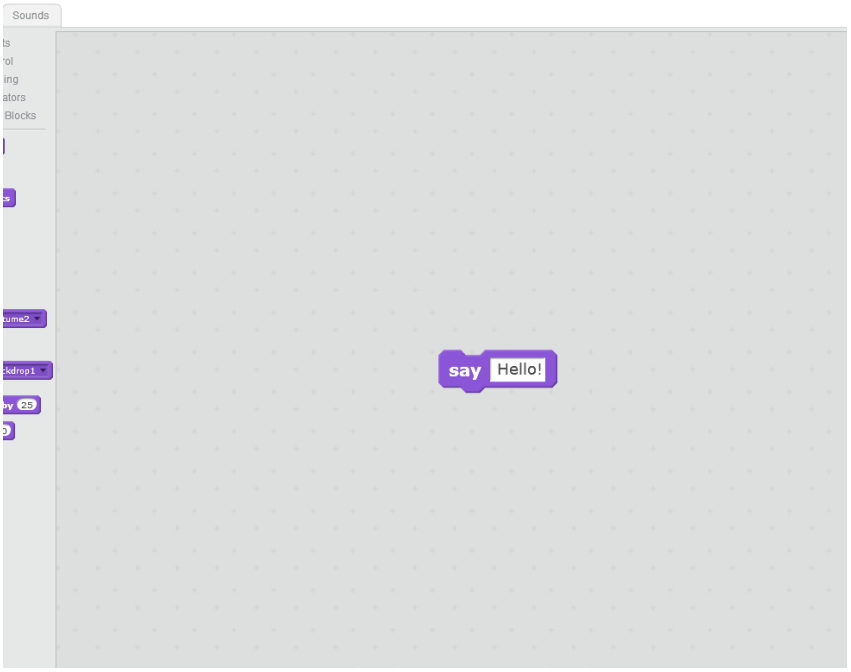


Figure 1.3

We have now specified an instruction in our program. To make the program carry out this instruction, click on the "say Hello!" block you just dragged to the blank space on the right side of the screen. What happens to the cat on the left?

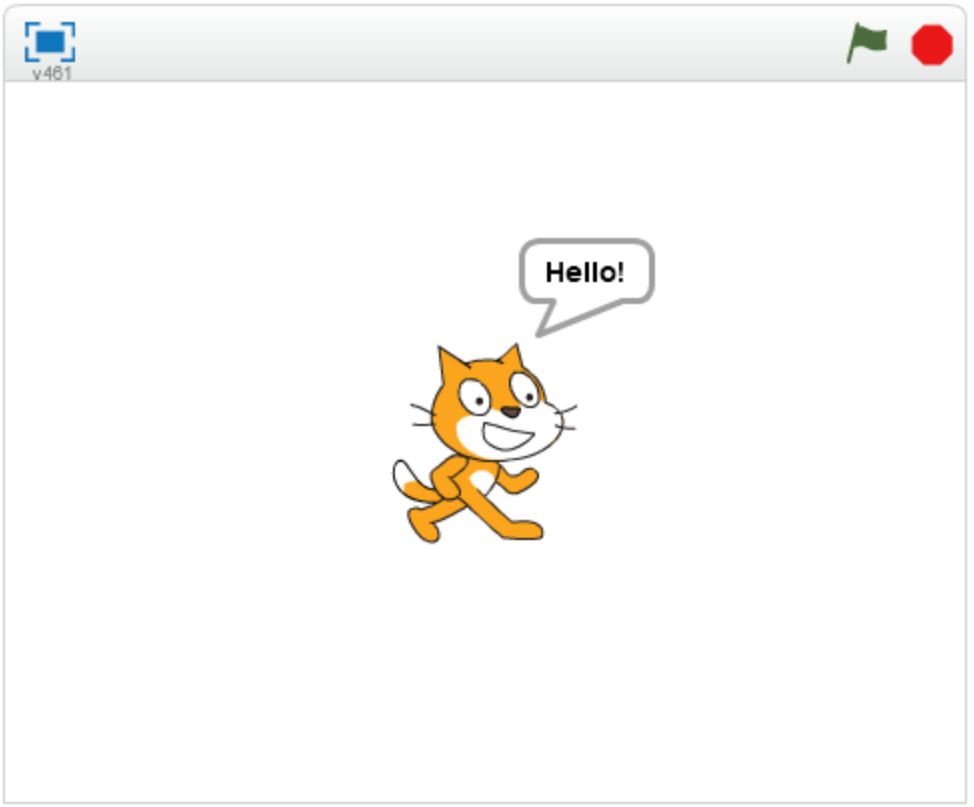


Figure 1.4

It says Hello!. This is great, but not correct because we want it to say our name (which is Peter Piper).

To correct this, click on the white box that contains the “Hello!” text in the block you have on the centre of the screen, change the text to your name and click anywhere on the screen outside the block to save your changes.

Figure 1.5

Finally, click on the block to carry out the new instruction we have just given our program. What does the cat say this time?

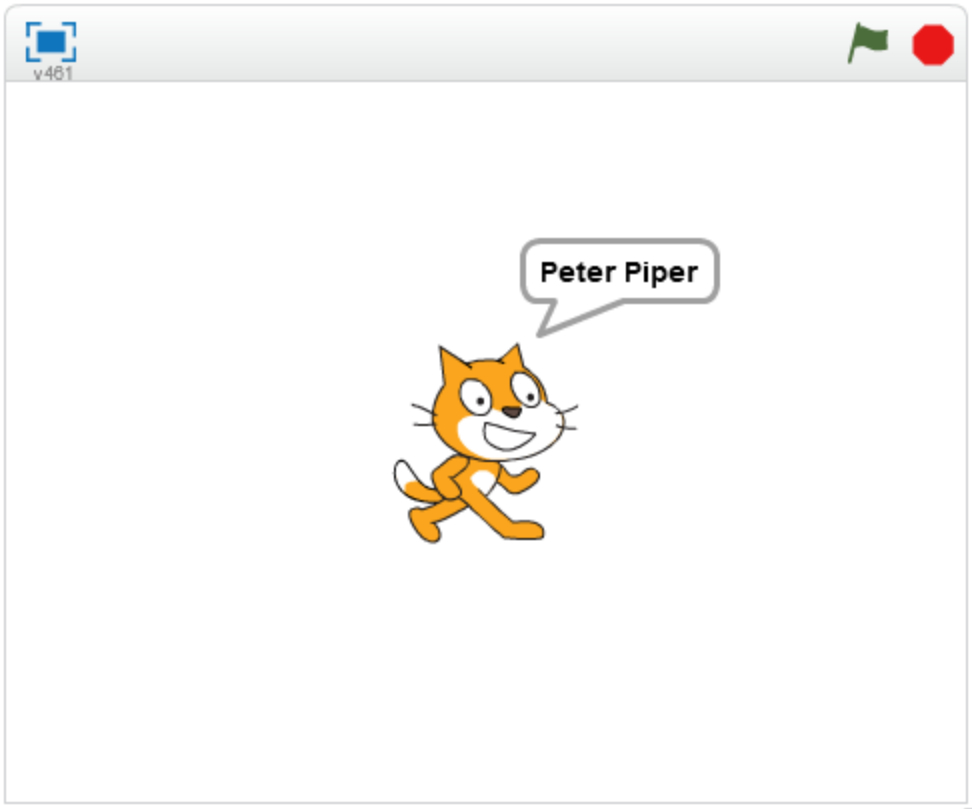


Figure 1.6

Great. You just built your first computer program. You can now call yourself a computer programmer.

Before you close the application, however, you must save your project. If you close the app without saving your changes to the project, everything you have done will be lost and you would have to start over if you wish to run this program again.

To save the program, click on “File” in the toolbar at the top of the app screen and choose “Save”.
Choose a location in your computer storage in which you wish to save the file, give the project a name and save the file.

Summary Of Chapter One

You have learnt that a computer program, also called an application, primarily carries out a set of instructions. The process of writing a computer program and running it on a computer, that is, writing a set of instructions and giving it to the computer to execute, is called Computer Programming.
You also learnt that a computer program carries out the set of instructions given to it in a sequential order, unless otherwise programmed. Do not forget that programming languages are the languages used to instruct the computer to perform actions through computer programs.

Finally, you learnt that Scratch is a programming language that uses visual components to code computer programs. A Scratch project is a work environment for creating a single computer program, and Blocks are used to specify instructions for a computer program in Scratch.

CHAPTER TWO

DEEP DELVE INTO THE SCRATCH EDITOR- PART 1

Chapter Objectives

At the end of the chapter, the students should be able to;

- **familiarize themselves with the Scratch Editor and its many sections.**
- **utilize the various capabilities of the Scratch Editor with little or no help.**

The Scratch Editor

In the previous chapter, you learnt a brief introduction to the Scratch Editor, the base from which we create computer programs using the Scratch programming language. The Scratch Editor can be accessed by opening the Scratch 2 application on your computer. When you open the app, a new Scratch project will be automatically created for you, and you can start creating a program right away.

The Scratch Editor is composed of many different parts that are used together to create computer programs, and an understanding of these parts and how they work together equips you with the ability to create highly functional applications.
The Scratch Editor is made up of the following major parts:

- The Stage
- The Sprite List
- The Scripts tab
- The Tips Window
- The Toolbar

The basic components of every Scratch project and program are: **Sprites, Costumes, The Stage** and **Backdrops**.

Sprites & Costumes

Figure 2.0

Sprites are the primary components of your Scratch applications. The cat in *Figure 2.0* that is included in every new Scratch project you create, is a Sprite. If your Scratch applications were to be movies, then Sprites would be the actors and actresses, the items used by the actors and actresses (for example, swords and guns for combat movies), and almost everything else. If your Scratch applications were to be football matches, then Sprites would be the players, the football, the referees, the goal posts, linesmen, the coaches, and the spectators.

In the Scratch Editor, a sprite is simply a container for a character or object you wish to use in your application. What the sprite actually looks like in the application depends on its current **costume**.

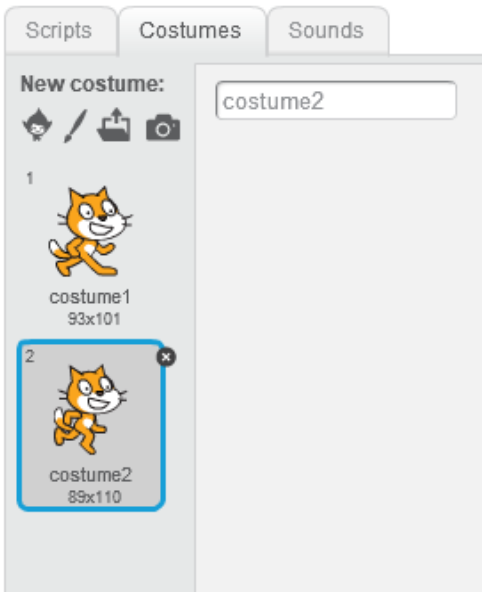


Figure 2.1

For example, if we were to create a football game, we would need the following objects:

- Player: there would be 22 of them in total, 11 for each football team.
- Referee.
- Goalpost: there would be 2 of them in total.
- Spectator: The number needed would depend on the capacity of our football field, but for the purpose of this illustration, we will need 100 spectators.
- Football: There's usually just one football needed for a football match.
- Linesman: Let us assume that we would need four linesmen.
- Coach: We need two coaches, one for each football team.

The above are the components we would need for a football match application, and these components all interact with one another. The players pass the football to each other, the referee talks to the linesmen and the players, the coach talks to his players, the players play the football on the field, and so on.

To develop this football application in Scratch, you would need to create a **Sprite** for each component. This means that you would have twenty-two sprites for the twenty-two players, one sprite for the referee, two sprites for the two goalposts, hundred sprites for the hundred spectators, one sprite for one football, four sprites for the four linesmen, and two sprites for the two coaches. That is 132 sprites in total.

The number of **costumes** you would need, however, would be much less than the number of sprites. You only need to create one costume for one type of look or appearance of your sprites, and you can give multiple sprites the same costumes. For example, twenty out of twenty-two players would all look the same, so you would need to create just **one player costume**. You can then add this costume to each of the twenty player sprites. The remaining two players, that is, the goalkeepers, would also look identical. Therefore, you would need to create just **one goalkeeper costume** and then add the costume to each of the two goalkeeper sprites.

This process would be repeated for all other identical sets of sprites, and we would finally have the following list of costumes:

- 1 Player costume.
- 1 Goalkeeper costume.
- 1 Referee costume.
- 1 Goal Post costume.
- 1 Spectator costume.
- 1 Football costume.
- 1 Linesman costume.
- 1 Coach costume.

That is just eight costumes in total, in contrast to 132 sprites.

This might be a lot to take in, but you only need to keep the following basic things in mind:

- A Sprite represents a single component in a Scratch application. This component could be anything.

- Sprites' looks are defined by costumes.
- A sprite can have more than one costume.
- A costume can be assigned to more than one sprite.

The Stage & Backdrops

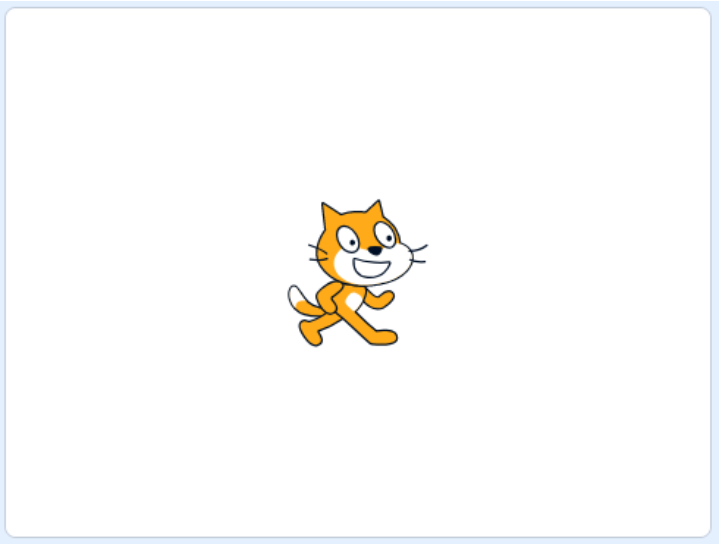


Figure 2.2

The Stage is a platform on which Sprites are placed to interact with one another. Every sprite you create in a Scratch application has to be added to the stage. The stage is only a container for sprites; its look is defined by its current **backdrop**. Unlike sprites, you can only have one stage for a Scratch project or application, but like sprites, the Stage can have more than one backdrop.

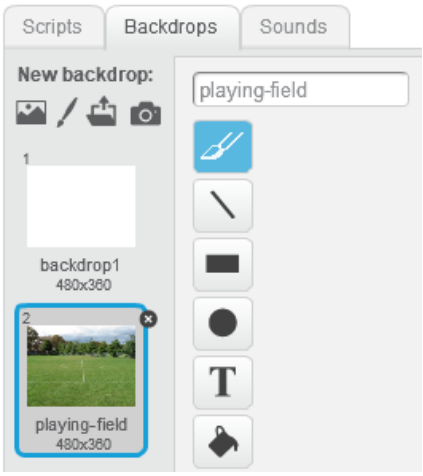


Figure 2.3

Using the analogy of a football match application, a football match has to be played on a football field, and all the sprites created, that is, the players, goalkeepers, coaches, referee, linesmen, football, goalposts

and spectators, all carry out their different functions on the same football field. The football field can be likened to The Stage.

There is, however, not just one football field for all football matches played all over the world; there are different football fields, and what differentiates each of these football fields is how they look. Hence, in our football match application, we would have one Stage, which is the football field, and different backdrops to make the football field look different for different matches. As you start making more complex applications with Scratch, you would need to change the backdrop displayed for the Stage as the application progresses. You can do this with the use of code blocks and scripts, as we will later see in this chapter.

For example, let us assume you were making an application about a human sprite moving around a house. A house has different types of rooms, and these rooms do not look similar. You might then have different backdrops for the different types of rooms, that is, the sitting room, the dining room, the kitchen, the bedroom, the toilet and the bathroom. You would then use code blocks to change the backdrop for the stage to the corresponding backdrop for each room as the human sprite moves into these rooms.

These are what you need to keep in mind:

- Each Scratch application can have only one Stage.
- The Stage is a container for all sprites used in an application.
- The look of a stage at a particular time is defined by its backdrop at that time.
- The Stage can have more than one backdrop.

With this basic knowledge of the basic components of every Scratch project and application, let us now look at the different parts of the Scratch Editor and see how we can manipulate these components, that is, the Sprites, Costumes, Stage and Backdrops to create exciting and functional applications.

The Stage Area

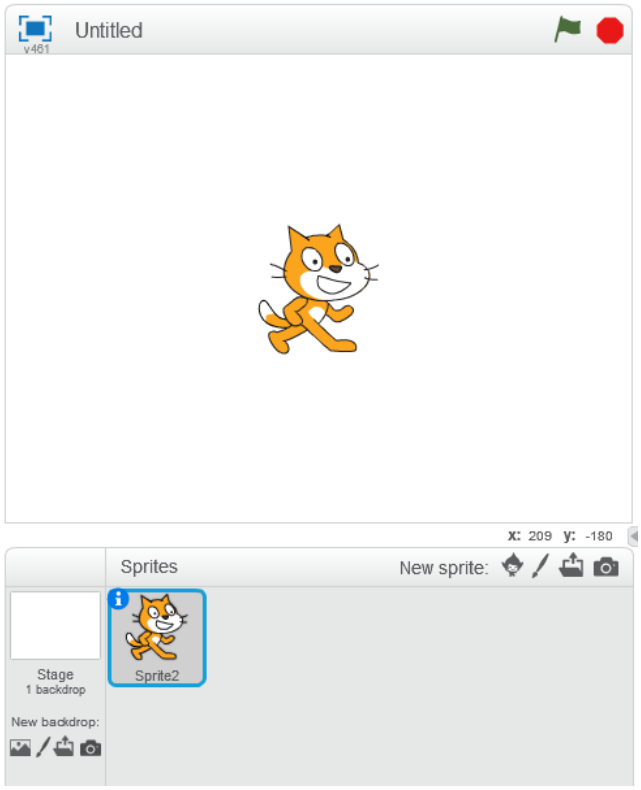


Figure 2.4

The Stage Area is the container for every visual component of our application, that is, everything that can be seen with the eyes that we have added to our app. The Stage is the first section contained in the Stage Area. There is a small bar at the top of the stage which contains some buttons; the first button is used to switch the Stage between fullscreen mode and the normal mode. You can try clicking on it and see what happens.

You usually switch to fullscreen mode when you are done creating your application and want to run it, or when you are still developing and want to test run what you have done so far.

Figure 2.5

To the right of the fullscreen switch is the name of the currently-opened Scratch project. Every new Scratch project is given the name “untitled”.

.

Figure 2.6

The *green flag* and the *red stop buttons* at the right are used to start and stop your application scripts respectively. Note that they are still present on the bar when you switch to fullscreen mode. The section below the stage is called the **Sprite List**.

The Sprite List

The Sprite List contains entries for all the Sprites you have in your project. Each Sprite entry has a thumbnail (a small image of the sprite) and the name of the sprite.

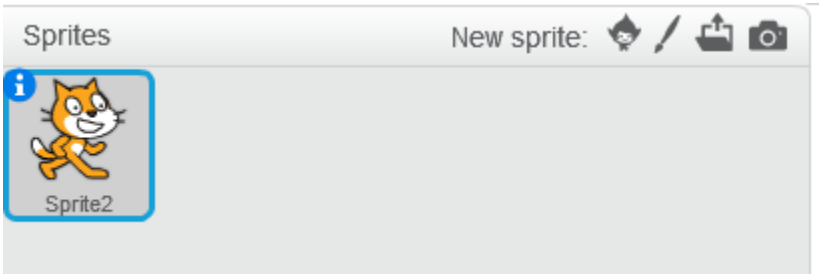


Figure 2.7

The ⓘ icon on a sprite entry is used to view and edit the sprite's properties. Each Sprite has a set of properties that determine the sprite's position, direction, visibility and behaviour on the stage. Click on the icon to see what these properties are.

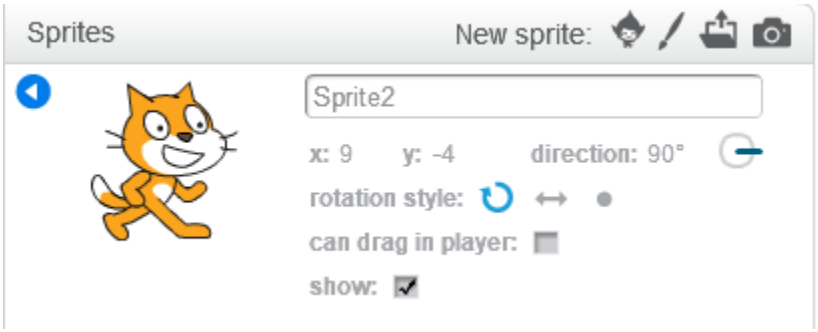


Figure 2.8

The sprite's properties are listed to the right of its thumbnail, as shown in *Figure 2.8*.

The first property is the name. You can click in the box and edit its contents to rename the sprite.

The ***x*** and ***y*** properties indicate the sprite's current horizontal and vertical position respectively on the stage. The default values for these properties are 0, meaning the centre of the stage. Move the cat sprite to another position on the stage and notice how the values of these properties change.

The ***direction*** property indicates the direction in which the sprite is currently facing on the stage. The default value for this property is 90, meaning the sprite is facing the east (i.e., the right side). Use the wheel to rotate the sprite and see how the value of the *direction* property changes.

The ***rotation*** *style* property is used to control how the sprite rotates on the stage. There are three options, with the first one which allows the sprite to rotate in any direction being the default. The second option restricts the sprite to rotating left and right only, while the third option prevents any form of rotation. Try changing the value of the *rotation style* property and then, using the wheel to change the value of the *direction* property to see how the sprite is affected by these properties on the stage.

The ***can drag*** *in player* property is used to control whether the sprite can be dragged around the stage with the mouse when the stage is in fullscreen mode. Switch the stage to fullscreen mode and try moving the cat sprite with your mouse. Now check the box and try again. See the difference?

The ***show*** property is used to control the sprite's visibility on the stage, that is, sprite's appearance and non-appearance on the stage. Check the box to render the sprite visible, and uncheck it to render the sprite invisible. Note that making a sprite invisible does not remove it from your project or from the stage; the sprite would still be in its current position on the stage, but you just cannot see it.

When you are done editing the sprite's properties, click the left arrow button at the top left of the thumbnail to close the properties window.

Right-clicking on a sprite in the Sprite List brings up some options, some of which we have already seen.

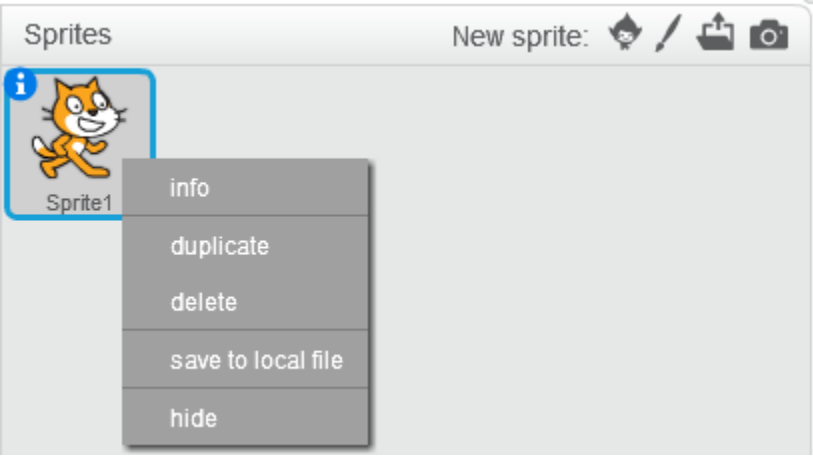


Figure 2.9

The ***info*** option works like the ⓘ icon on a Sprite entry to view and edit the sprite's properties.

The ***duplicate*** option creates a copy of the sprite and adds it to the Sprite List, but with a different name.

The ***delete*** option is used to remove a sprite. Unlike the *show* option which simply hides the sprite from the stage, the *delete* option removes the sprite entirely from the Scratch project.

The ***save to local file*** is used to export a sprite to a file and save it on your computer. You might want to do this if you created your own sprite with the Paint Editor (more on this later) and want to reuse it in other projects.

The ***hide*** option renders the sprite invisible on the stage.

The buttons at the top of the sprite list, to the right of the label "New sprite" are used to create and add new sprites to the project. There are four buttons, meaning there are four ways to add sprites to Scratch projects.

Figure 2.10

The first button is used to add sprites from the Scratch **Sprite Library**.

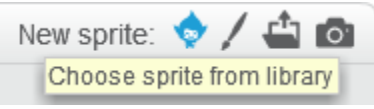


Figure 2.11

There are numerous sprites to choose from in the Sprite Library, and you can go ahead and try adding another sprite to your project.

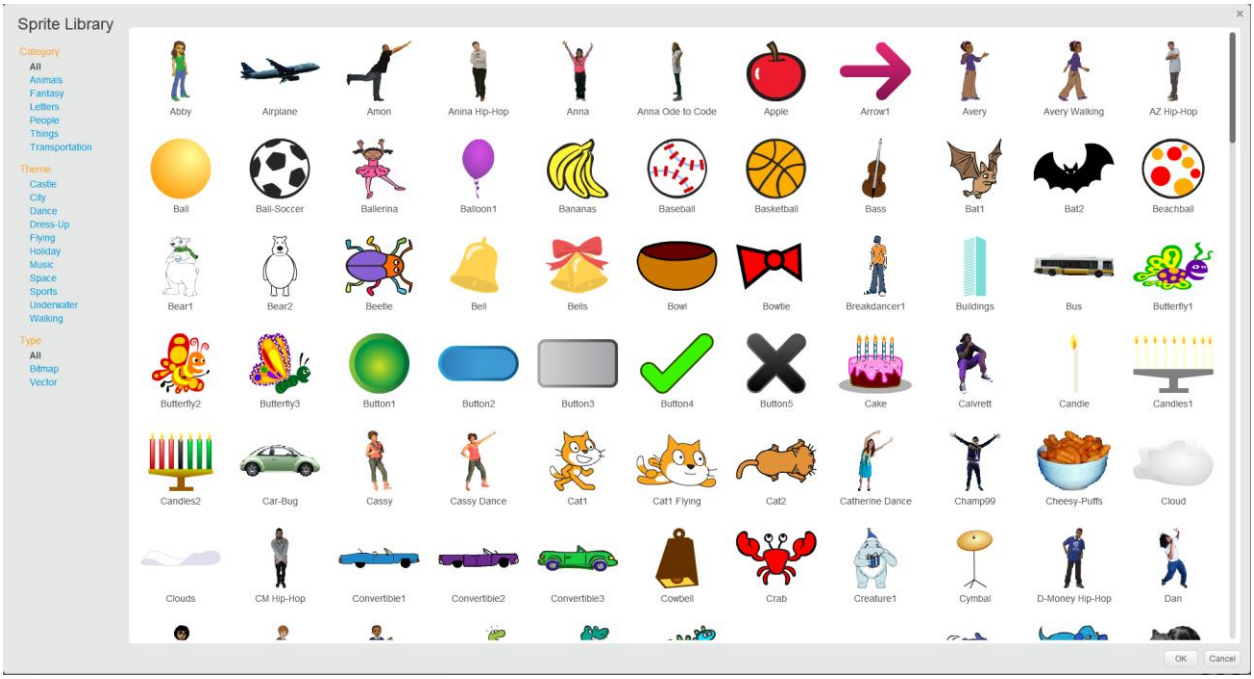


Figure 2.12

The second button is used to create a new sprite from scratch using the Paint Editor.

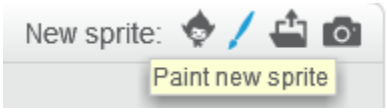


Figure 2.13

The Paint Editor is a tool built into the Scratch Editor that allows you to create custom graphics for use in your projects. We will explore the Paint Editor in the next chapter.

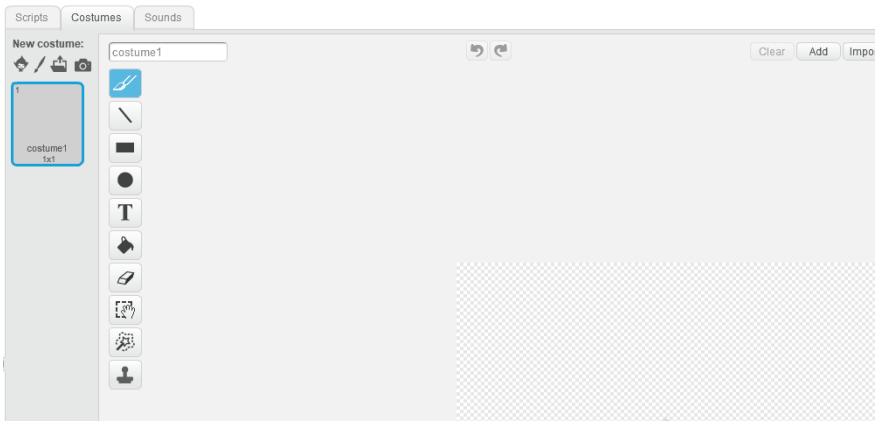


Figure 2.14

The third button is used to import sprites into projects from your computer.

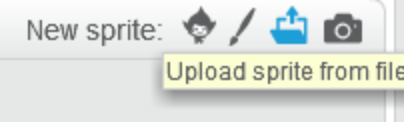


Figure 2.15

This is the opposite of the export option for sprites that we have seen earlier, which allows you to export sprites as files to your computer.

The fourth button allows you to create new sprites from images taken with a camera.

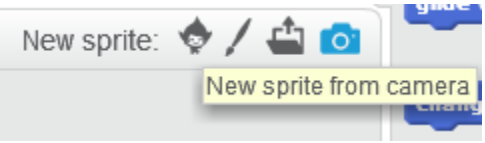


Figure 2.16

The camera must either be part of (internal) or connected to (external) your computer.
There is a small section to the left of the Sprite List, which contains a thumbnail for the backdrop currently displayed on the stage, the number of backdrops available for the stage, and the buttons needed to add new backdrops to the stage.

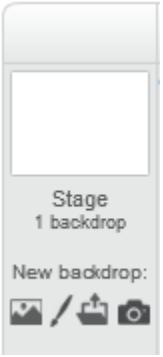


Figure 2.17

Like sprites, there are four ways to add backdrops to the stage. The first button lets you select a backdrop from the Scratch **Backdrop Library**.

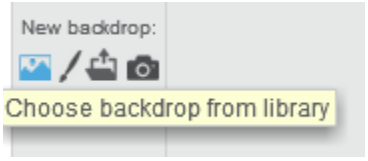


Figure 2.18

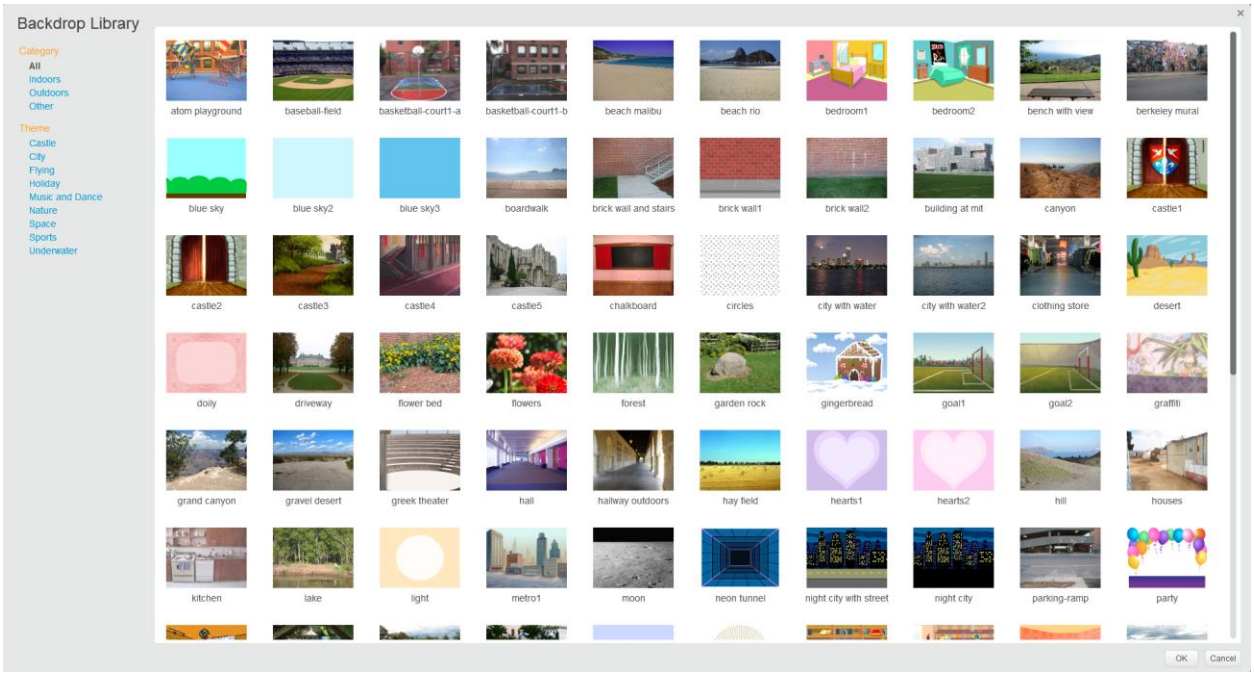


Figure 2.19

The second button allows you to create a new backdrop from scratch using the Paint Editor.

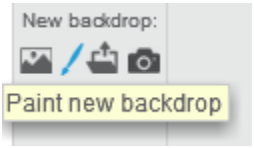


Figure 2.20

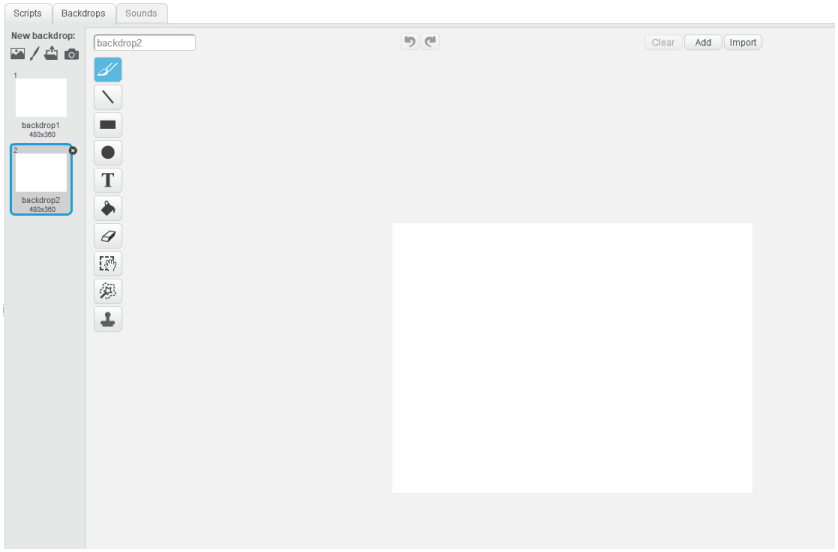


Figure 2.21

The third button allows you to upload a backdrop file from your computer,

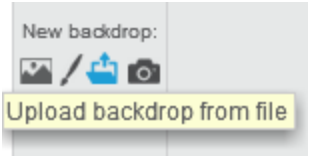


Figure 2.22

and the fourth button creates a new backdrop from an image taken from a camera present on or connected to your computer.

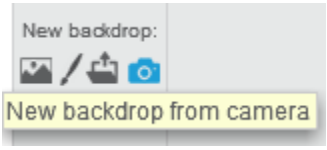


Figure 2.23

The Stage Area, which contains the Stage and the Sprite List, takes up about ¼ of the Scratch Editor window by default. The amount of space occupied by the Stage Area can be reduced when desired with the aid of a small arrow button located at the right side of the space between the stage and the sprite list.

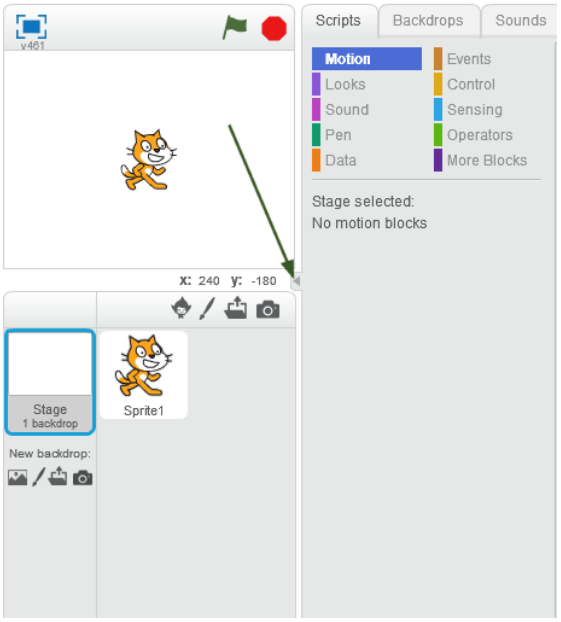


Figure 2.24

Summary Of Chapter Two

You have learnt that the Scratch Editor is the main screen that contains everything needed to create a computer program with Scratch. It is made up of the stage, sprite list, scripts tab, tips window and toolbar. Remember, sprites are the primary components of Scratch applications. They are containers for an object/character used in an application. A sprite has several properties that affect its looks and behaviour, and all sprites in an application are contained in the Sprite List.

You learned that the visual appearance of a sprite is determined by its current costume. A Sprite can have more than one costume, and a costume can be assigned to more than one sprite.
Then, Sprites can be added from the Scratch Sprite Library, created manually using the Paint Editor, imported from a local file, or created from images taken with a camera.
Finally, you learned that the Stage is a platform on which Sprites are placed and interact with one another. A Scratch project/application can have only one stage. The visual appearance of the Stage is determined by its current backdrop. The stage can have more than one backdrop. Backdrops can be added from the Scratch Backdrop Library, created manually using the Paint Editor, imported from a local file, or created from images taken with a camera.

CHAPTER THREE

DEEP DELVE INTO THE SCRATCH EDITOR - PART 2

Chapter Objectives

At the end of the chapter, the students should be able to;

- **create basic Scratch programs from simple scripts.**
- **reasonably explain the functions of most of the code blocks.**

Apart from the properties of Sprites which we have seen earlier, a Sprite can be given additional attributes to modify either its looks or behaviour. One of these attributes, **Costume** which has been talked about, is used to modify a sprite's look. There are two other such attributes, namely **Blocks** and **Sounds**.

The Stage can also be given other attributes, one of which is **Backdrop** which has been examined earlier in this chapter. Backdrop is to the Stage as Costume is to Sprites; both of these attributes modify the look of their respective components.

The Stage can also be given the **Blocks** and **Sounds** attributes. These attributes are contained in the tabs located at the right side of the Stage Area.

Tabs

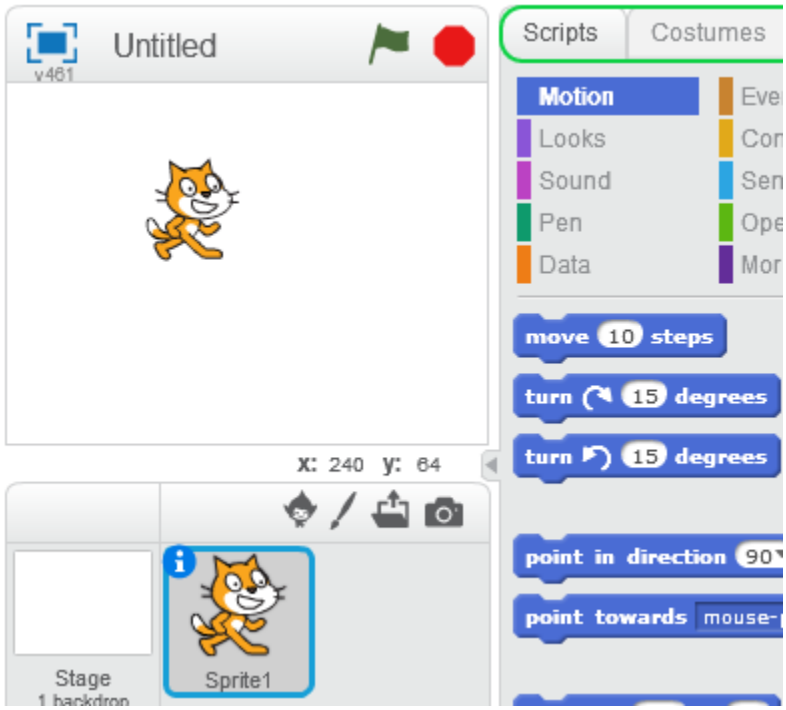


Figure 3.0

There is a tab for each of the **Scripts, Costumes, Backdrops** and **Sounds** attributes.

A Script is a group of Blocks stacked together. This is a Block:

Figure 3.1

And this is a Script:

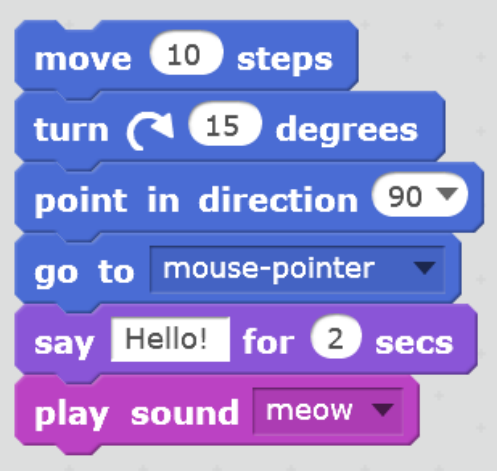


Figure 3.2

However, only three of these tabs are shown at once; either **Costumes** or **Backdrops** is shown depending on which type of component is selected on the Sprite List. If a sprite is selected, the *Costumes* tab is shown since only sprites can have costumes, while the *Backdrops* tab is shown if the Stage is selected, since only the stage can have a backdrop.

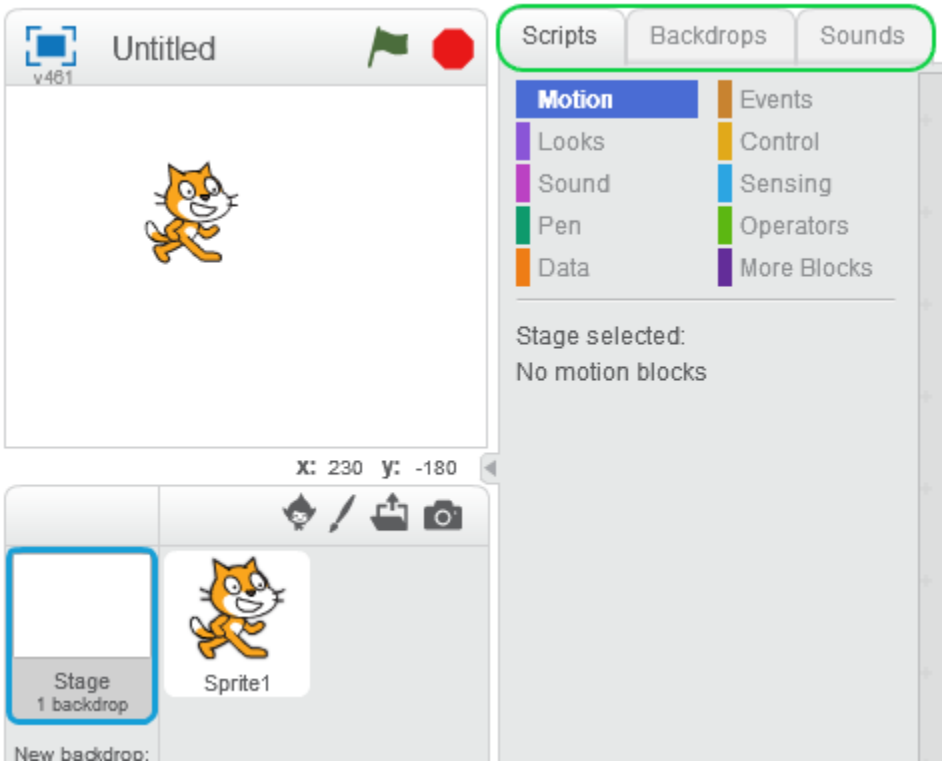


Figure 3.3

The Scripts Tab, The Blocks Palette & The Scripts Area

In the last chapter, we learned that computer programming is the process of specifying instructions for computer systems to follow or carry out. We also learned that we need to write **code** with a **programming language** that the computer can understand, and that Scratch is a programming language.

Blocks are the codes used to program applications in the Scratch programming language.

A component (sprite or the Stage) can be programmed, that is, its behaviour can be modified by assigning a set of blocks and scripts to it. Scripts are assigned to components by selecting them in the Sprite area and dragging blocks from the Scripts tab to the Scripts Area and stacking them with other blocks to form scripts.

To view the contents of the Scripts Tab, select the tab in the Tabs section.

Figure 3.4

There are two sections in the Scripts Tab: the Block categories, and the Blocks Palette.

Block Categories

There are nine categories of blocks that can be used in the Scratch programming language; clicking on a block category reveals all the Blocks available under that category below it, in the Blocks Palette.

Each block has some texts that describes what the block does, and some blocks have one or more input boxes or dropdowns that you can edit to modify the behaviour of the blocks. The blocks shown in the Blocks Palette depends on the type of component selected in the Sprite List, because not all blocks are

available for all components. For example, the Stage does not have any *Motion* block because the stage cannot be moved.

Each Block category has a colour, which is also the colour of all the blocks under it (shown in the Blocks palette when the category is selected). This helps you to quickly identify which category a block falls into.

You can quickly see how a particular block affects the selected component by clicking on it right there in the Blocks Palette. Try clicking on the **move 10 steps** block and see what happens to the Scratch cat on the stage.

These are the functions of the nine block categories available in Scratch:

- **Motion:** for modifying the position (horizontal and vertical) properties of a sprite on the stage.
- **Looks:** for modifying the visual properties of components.
- **Sound:** for modifying the audio properties and state of components.
- **Pen:** for drawing on the stage with sprites.
- **Data:** for storing information about components.
- **Events:** for reacting to triggers.
- **Control:** for controlling the order of operation.
- **Sensing:** for detecting stage and sprite conditions.
- **Operators:** for manipulating numbers and strings.

All blocks in Scratch are available in one of four kinds, which are: **Command** blocks, **Control** blocks, **Trigger** blocks and **Functions** blocks.

1. Command Blocks: These are the most common kinds of blocks, and are used to directly give commands to components.

Figure 3.5

2. Control blocks: They are blocks that other blocks within them. An example is the forever block in figure 3.6 below.

Figure 3.6

3. Trigger blocks are also called hat blocks because they have a rounded top, thereby making it impossible to stack a block on top of them. Hence, trigger blocks are usually used to start a script.

Figure 3.7

Command and Control blocks are called stack blocks, because they have attachment points both above and beneath them, and can be stacked with any other block in any position.

4. Function blocks: These blocks do not have attachment points, and hence cannot be stacked with other blocks. They are used as inputs for other blocks (hence, they are also called reporters), and come in different shapes:
    a. Those with rounded ends: used to report numbers and strings

Figure 3.8

    b. Those with pointed ends: used to report true and false (Boolean) values

Figure 3.9

We will see how these blocks are combined together to make scripts and run programs in future sections of this textbook.

In the same way a block is added for a component (sprite or stage) by dragging it from the Blocks palette to the Scripts area, blocks can be removed from a component by dragging it from the Scripts Area to the Blocks palette, or by right-clicking on it in the Scripts Area and selecting “delete”.

If you are unsure of what a block does, you can right-click on it either in the Blocks palette or the Scripts Area and selecting “help”; information about the block would be displayed in the Tips Window at the right side of the screen.

Costumes Tab

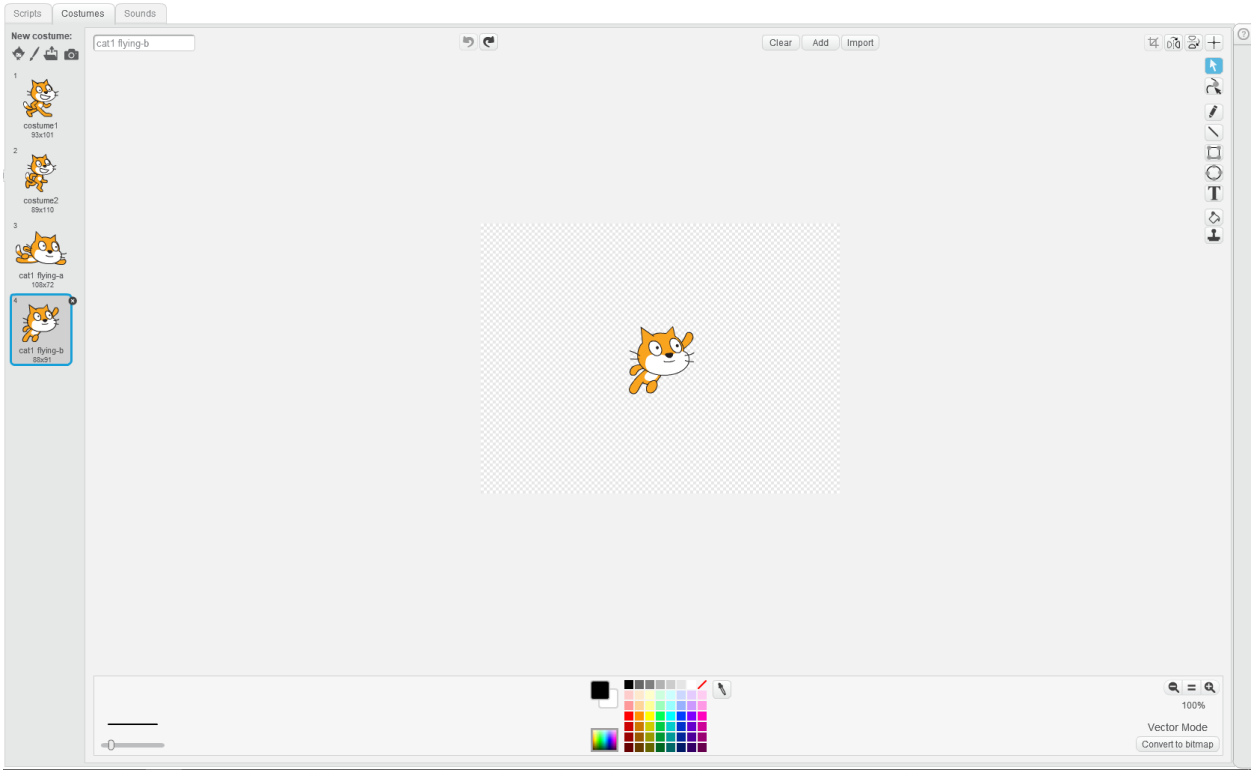


Figure 3.10

The Costumes Tab displays a list of all costumes for the sprite selected on the Sprite List. The name of each costume can be edited, and the appearance of a costume can be modified with the Paint Editor.

The Backdrops Tab

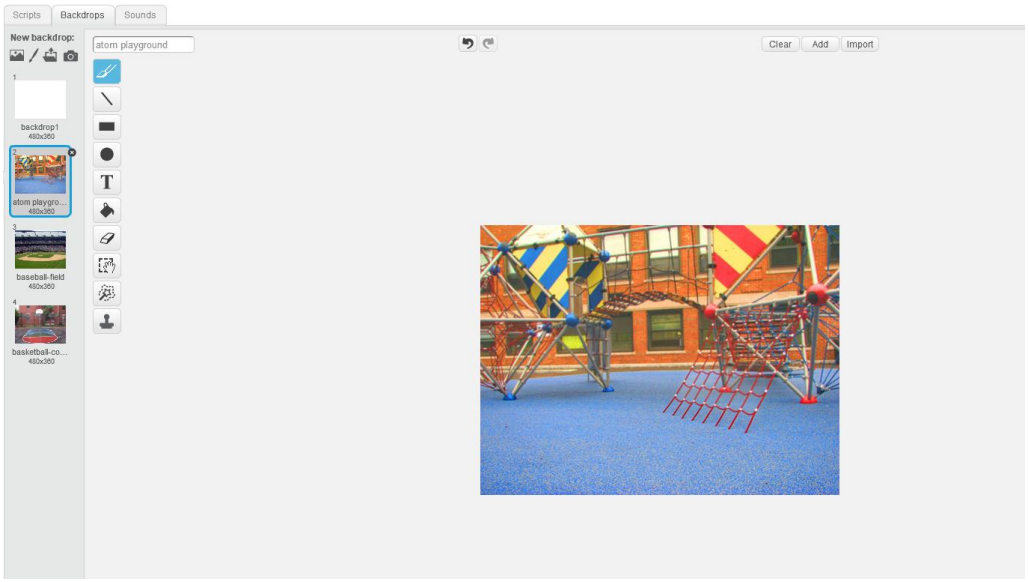


Figure 3.11

The Backdrops Tab displays a list of all backdrops for the stage. The name of each backdrop can be edited, and the appearance of a backdrop can be modified with the Paint Editor. Buttons for adding new backdrops to the stage are also present here, like on the Sprite List.

The Sounds Tab

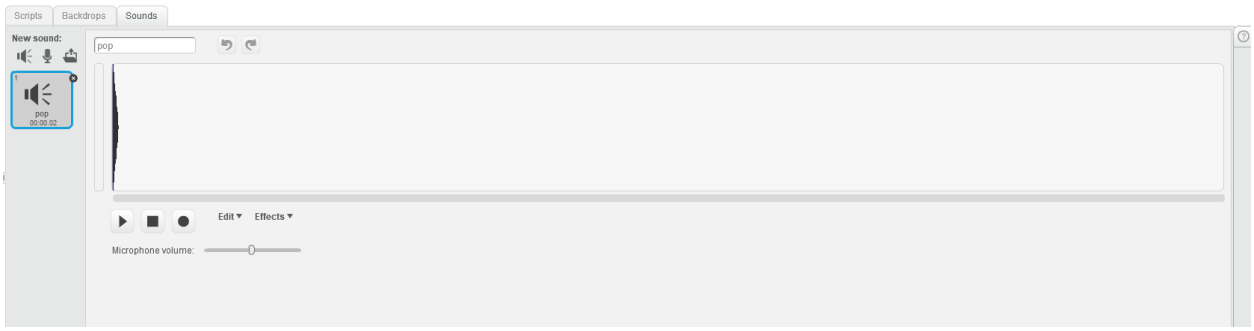


Figure 3.12

The Sounds Tab contains a list of all audio files for a component (sprite or stage). The stage has the "pop" sound file added to it by default, while the Scratch cat has the "meow" sound file added to it by default. Most sprites come with a default sound file, but you can add any sound file to any sprite from the library, make a recording of your own, or upload an audio file from your computer. You can also edit the names of sound files on the list.

The Tips Window

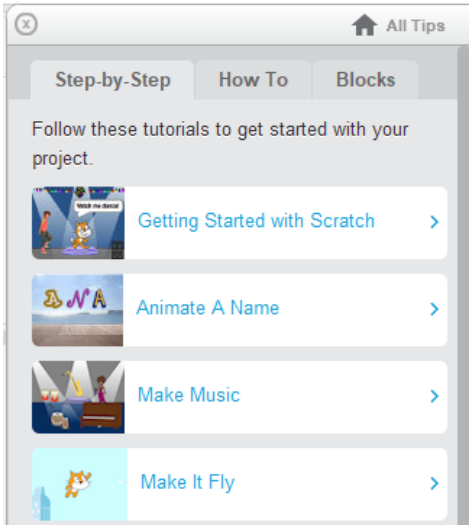


Figure 3.13

The Tips Window is located at the far right of the Scratch Editor. We have seen the tips window in use previously - for viewing information about Blocks. The Tips Window can be used to view various tutorials

and **how to's** for different functions on the Scratch Editor, in addition to displaying information about Blocks.

The Toolbar

Figure 3.14

There are two sets of buttons that would be of use for us in the course of this textbook; one set on the left, and one on the right, as shown in *Figure 3.14*.

The "File" button contains options for creating new Scratch projects, and opening and saving existing ones, while the "Tips" button opens the Tips Window. Explore these options to see what they each do.

The set of buttons at the right of the toolbar are used for manipulating sprites and blocks. The first button is used for duplicating, the second for deleting, the third for enlarging sprites, the fourth for shrinking sprites, and the last in the set for viewing tips about several parts of the Scratch Editor.

Chapter Project: Patrol Wizard

To put what we have learnt so far into practice, we will build a simple application that features a wizard patrolling a castle. The wizard is the guardian of the castle, and he is patrolling it to prevent unauthorized outsiders from getting into the castle.

Here is what the final application should look like:

Figure 3.15

The wizard is initially at the left side of the castle.

Figure 3.16

When the green flag is clicked, the application starts running and the wizard starts patrolling the castle (moves from left to right and vice versa repeatedly).

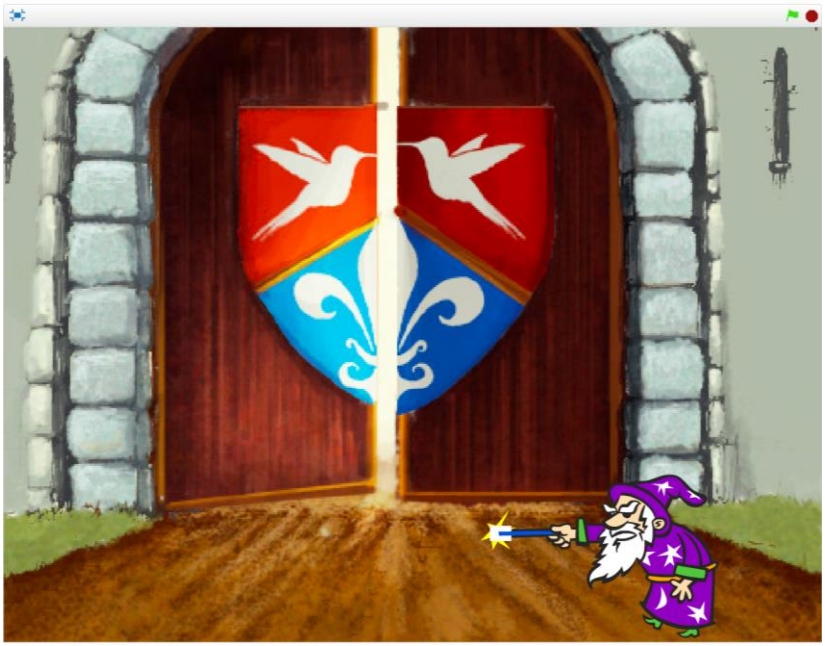


Figure 3.17

When the wizard gets to the right end of the castle, he turns back and patrols from right to left. The same thing happens when the wizard gets to the left end.

Even, an app as basic as this has several steps involved in its development, so it is necessary to break down the entire development process into small steps that would be tackled one by one till the app is complete. Most of the apps you would build would require the following steps:

- **Determine** all the functions (i.e. the actions performed) needed in the application.
- **Decide** which components (backdrops, sprites, costumes and sounds) are needed to perform these functions.
- **Add** these components to the application.
- **Program** the components (with blocks and scripts) to perform their functions.

This is the process we would follow every time we build an application in this textbook, so run through it again and bookmark this page for easy reference.

1. **Determine all the functions needed in the application**

   As we have seen from the final version of the application shown in *Figure 3.15*, our app involves just one action: a wizard moving from the left side of the castle to the right side. Let us note that down:

   - A wizard moving from the left to the right side of the castle.

   Also, notice that the wizard was originally facing the left direction (90 degrees) at the start of the application, but then switches to the opposite direction (-90 degrees) when it gets to the other edge of the castle (the right side). The same thing happens when the wizard gets back to the left

edge of the castle: its direction changes back to 90 degrees. This pattern of movement is repeated over and over again while the application is running.

Thus, we can conclude that:

- the wizard turns to the opposite direction when it hits the edge of the castle.

We have now determined all the functions needed in our Patrol Wizard application.

2. **Decide which components (backdrops, sprites, costumes and sounds) are needed to perform these functions**
   Now look closely at the application and see if you can point out the different backdrops, sprites, costumes and sounds present.

   From the final version of the application shown in *Figure 3.15*, we can see that there is just one castle backdrop, one wizard sprite, one wizard costume (that shows a wizard pointing forward with a wand in hand), and no sound in the application.

   Let us organize our findings into a table:

   PROJECT COMPONENTS

COMPONENT	**FINDINGS**
Backdrop	● 1 Castle
Sprite	● 1 Wizard
Costume	● 1 Wizard pointing forward with a wand in hand
Sound	None

   Organizing your findings this way makes it easy to determine what you need to add to your application when you eventually start building it in the Scratch Editor. It also helps you to make sure you are not leaving anything out, which might happen if you choose to think about these things along with your development process.

3. **Add these components to the application**
   It is now time to jump into the Scratch Editor. Open the Scratch 2 Offline Editor application on your computer; a new project containing the Scratch cat will be automatically opened for you.

   There is no need for the Scratch cat sprite, so go ahead and delete it from the Sprite List.

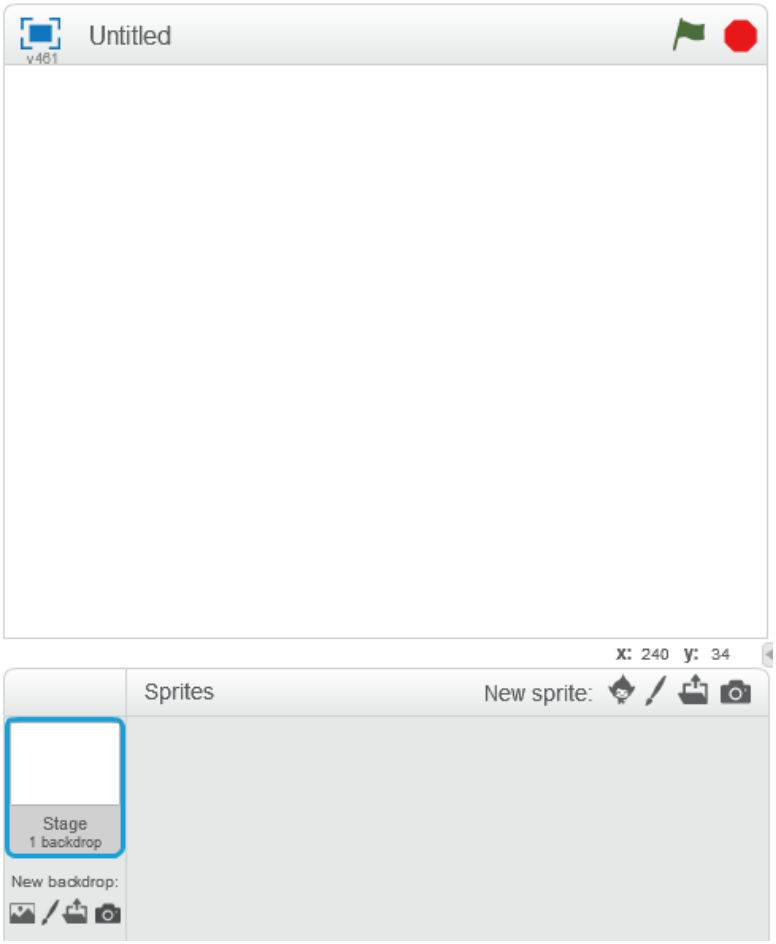


Figure 3.18

We now have an empty stage, and no sprites on the Sprite List

The first component on our **Project Components** table is a castle backdrop. Let us add this backdrop to the Stage.

Click on the first icon in the backdrop area of the stage to add a new backdrop to the Stage from the Backdrop Library.

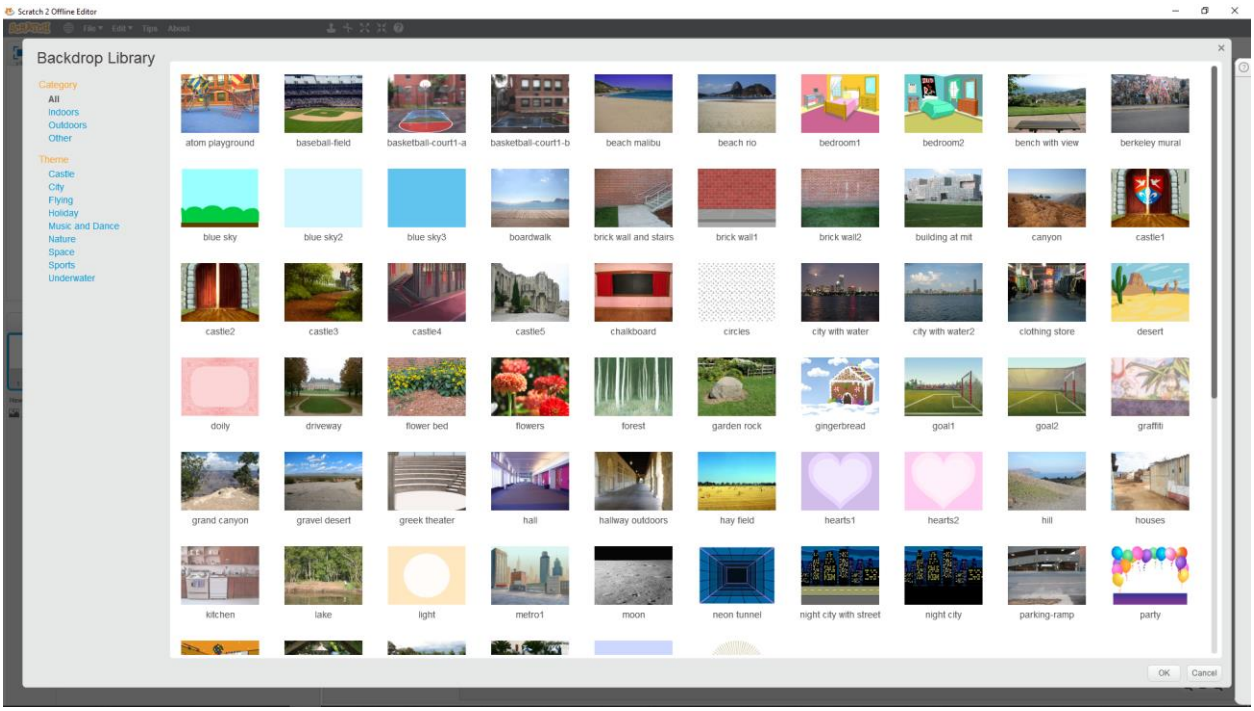


Figure 3.19

The backdrops in the Backdrop Library are sorted by name. The section on the left also contains categories and themes to sort the backdrops. The backdrop we need is a castle, and the Backdrop Library has five castle backdrops. Our final application uses the first one named “castle1".

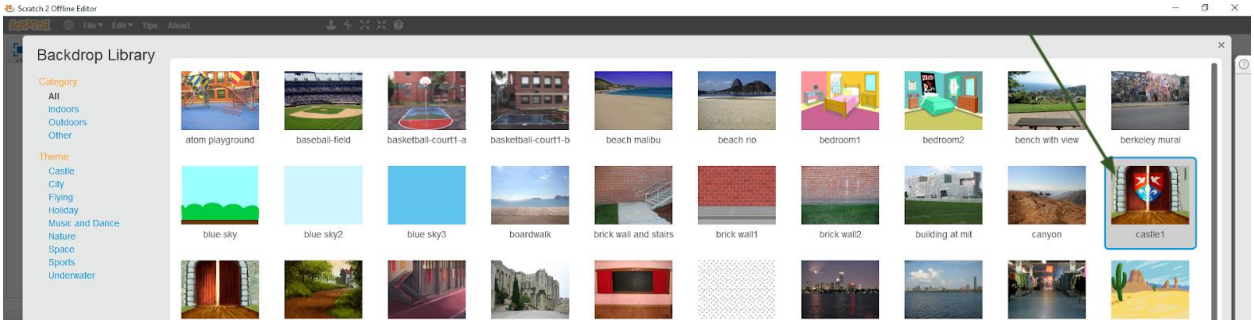


Figure 3.20

Select the backdrop and click “OK” to add it to the Stage.

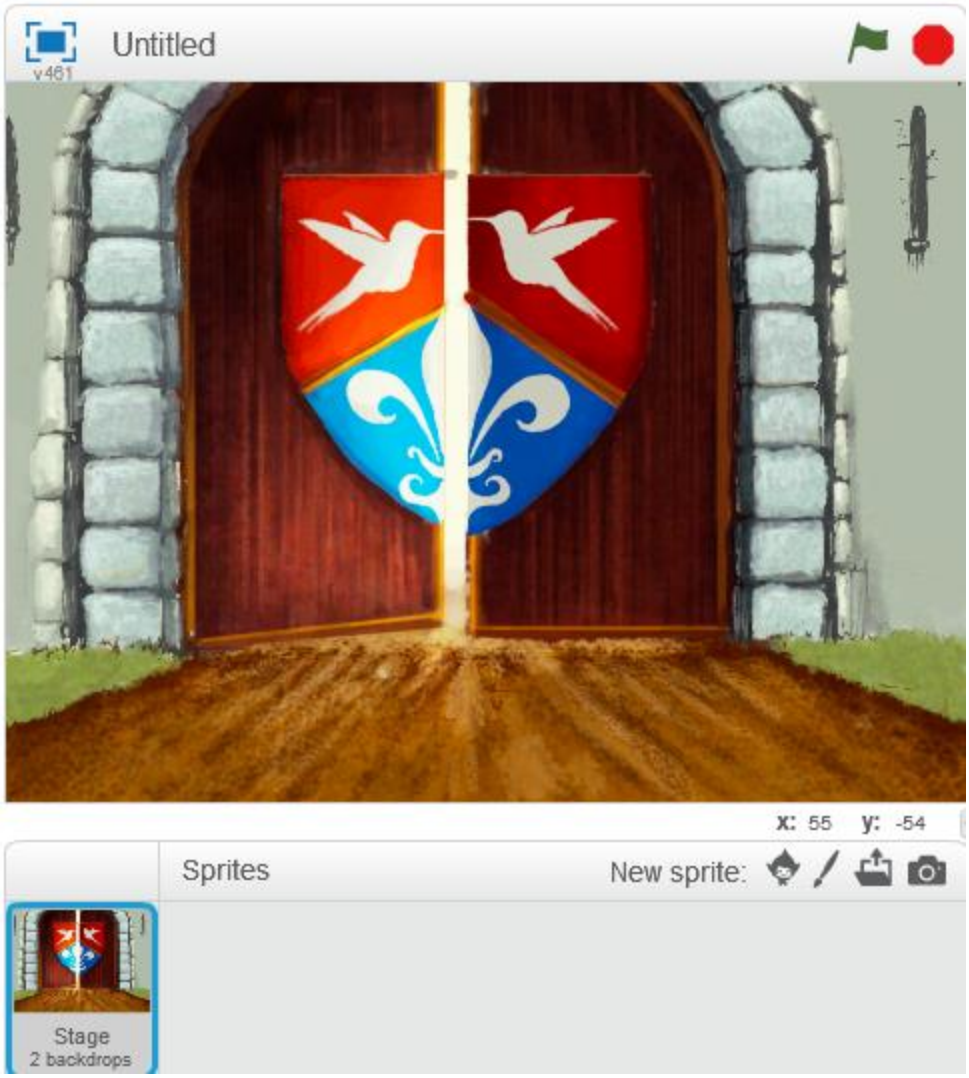


Figure 3.21

We now have two backdrops on the Stage. There is no need for the first one (the default one, with the blank background), so you can go ahead and delete it in the Backdrops tab.

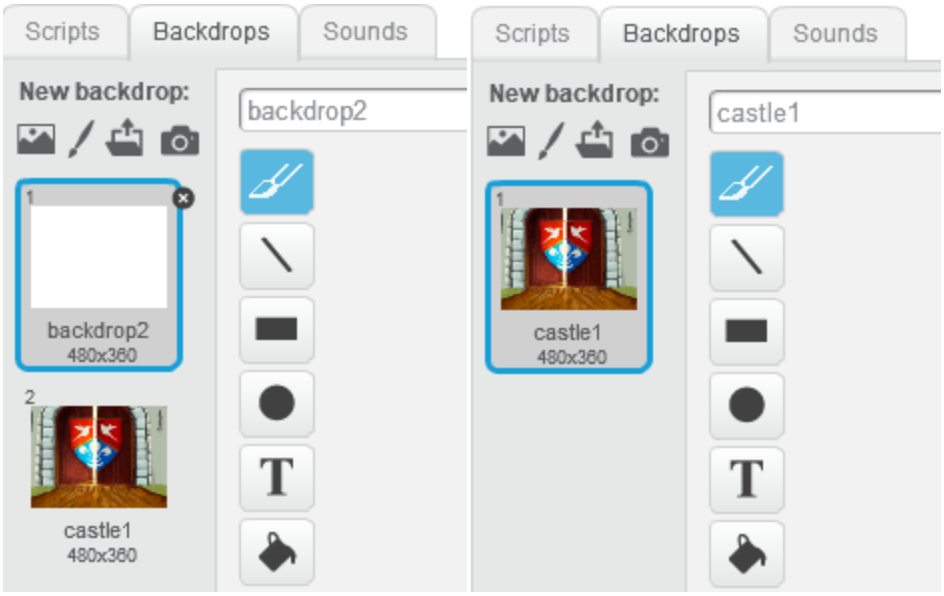


Figure 3.22

We have just added our first component, a backdrop named *castle1*, to the application. Let us rename the backdrop to better describe its functions in the application. Since the castle belongs to a wizard, we can call it "wizarding castle". Rename *castle1* to *wizarding castle*.

Note: Renaming components is not a mandatory step and your application would still run fine if you do not, but it helps to better understand the different components that make up your app. Large applications are comprised of numerous components, and having generic names like "Component1", "Component2", etc. would lead to confusion.

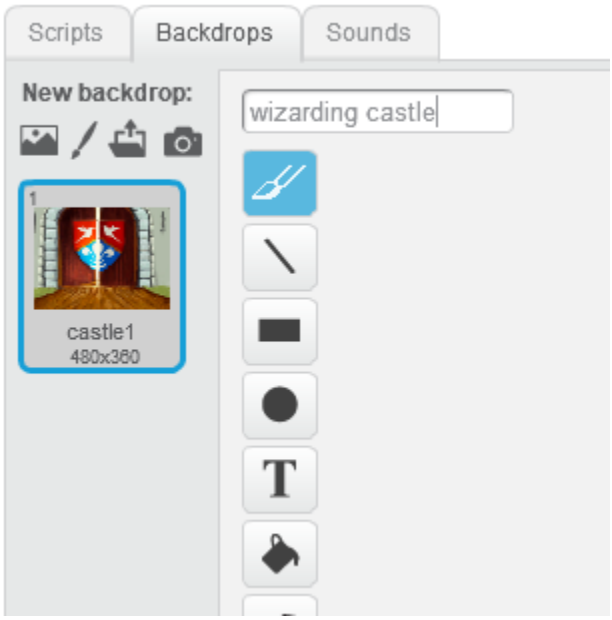


Figure 3.23

TRY IT OUT: Whenever you start a new project and add a component to it, it is a good idea to save the project and give it a name so that you can always work on it later. Use the toolbar tools you learnt earlier to save your project with the name of the application - Patrol Wizard.

The second component on our **Project Components** table is a wizard sprite. Let us add this sprite to the Sprite List.

Click on the first of “New sprite” icons above the Sprite List to add a new sprite from the Sprite Library.

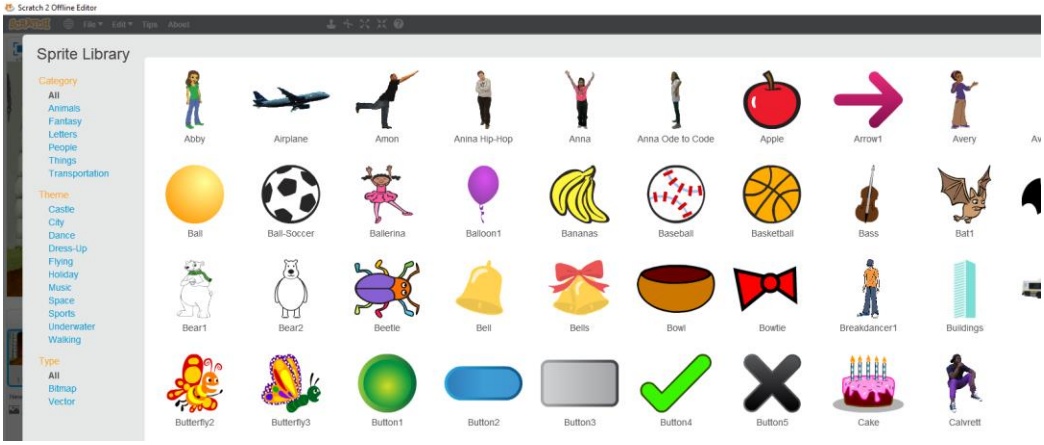


Figure 3.24

Like in the Backdrop Library, the sprites in the Sprite Library are sorted by name. The section on the left also contains categories and themes to sort the sprites. The sprite we need is a wizard, and the Sprite Library has three wizard sprites. Our final application uses the first one named “Wizard".

Figure 3.25

Select the sprite and click “OK” to add it to the Sprite List.

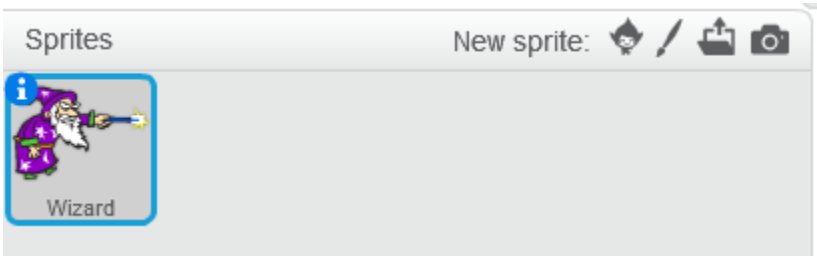


Figure 3.26

The sprite we added is named “Wizard”. This describes the function of the sprite enough, so we do not need to rename this component like we did for the castle backdrop.

The next component on our **Project Components** table is one wizard costume (that shows a wizard pointing forward with a wand in hand). Let us head over to the Costumes tab.

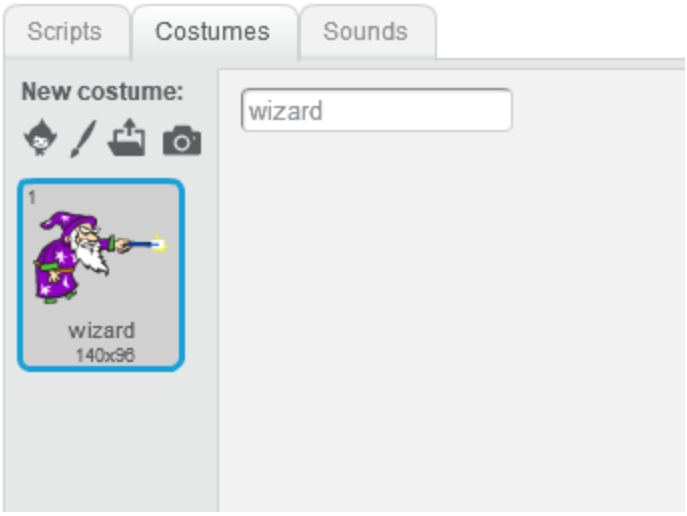


Figure 3.27

Unlike backdrops, sprites can have more than one costume, and each costume has its own name. Therefore, we would need to rename each costume separately if need be. The Wizard sprite has just one costume which is the exact one listed on our Project Components table named "wizard" which is fine.

Note: We have made further changes to our project. Save your changes.

There are no sounds listed on our Project Components table, so we are skipping that component. We will learn how to work with sounds in forthcoming chapters.

Now to the last step in our process for building applications.

4. **Program the components (with blocks and scripts) to perform their functions**
   These are the two functions we derived from the first step of the process:

   - A wizard moving from the left to the right side of the castle.
   - The wizard turns to the opposite direction when it hits the edge of the castle.

   We need to find the right blocks and scripts (combination of blocks) that would make our components perform these functions in our application. Let us take them one by one.

   A. A wizard moving from the left to the right side of the castle.

      There are two things to note about this action: it is a **motion action**, and it is performed on the **Wizard sprite**.

Therefore, we need blocks from the **Motion** category of blocks, and we need to select our *Wizard* sprite on the Sprite List. After doing that, switch to the Scripts Tab and select the Motion category to view all the motion blocks available for the sprite you selected.

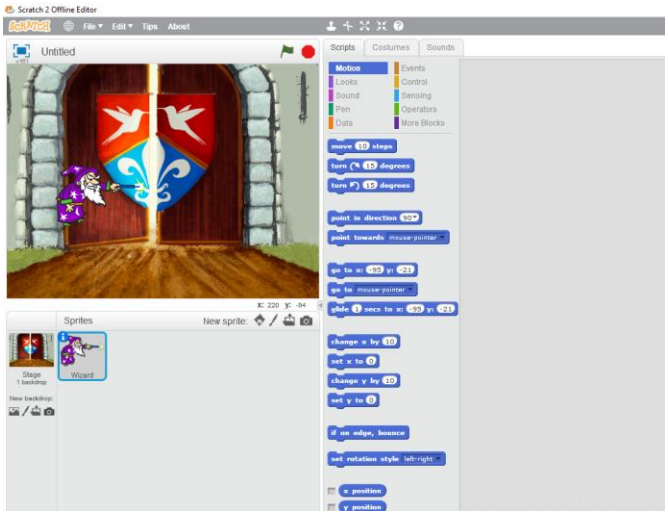


Figure 3.28

The first block in the Blocks Palette for the Motion category is called “move 10 steps”. This seems perfect for what we want, which is to make the wizard move from one point to another. Drag the block to the Scripts Area to add it to the Wizard sprite’s scripts.

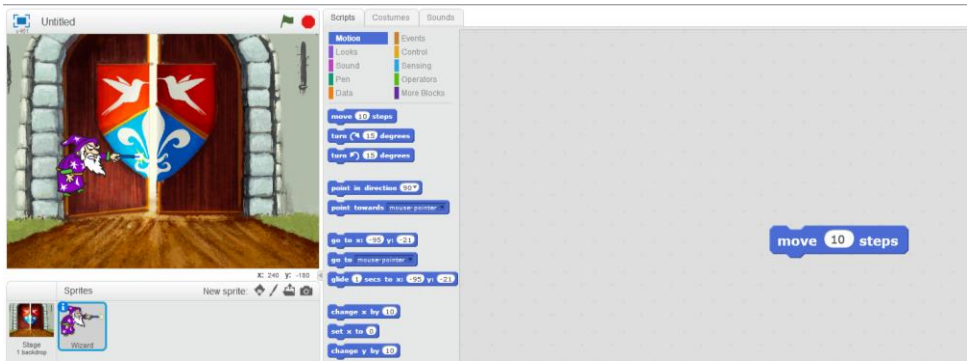


Figure 3.29

TRY IT OUT: You can execute your blocks right away by clicking on them either there in the Blocks Palette, or in the Scripts Area after adding it there. Test it: does the wizard move when you click on the “move 10 steps” block?

Notice that the wizard moves in the direction it is currently pointed at.

TRY IT OUT: Change the direction of the Wizard sprite in the Sprite List and click on the “move 10 steps” block. Does the sprite move in the new direction?

The “move 10 steps” block has a small white area in which the “10” is contained. This is called an input to the block, and the value of the input can be modified to change the block’s behaviour.

TRY IT OUT: Click on the white box containing “10” in the “move 10 steps” block and change the value to “5”. Now test the block again: what changes do you notice in the wizard’s movement?

We have made further changes to our project. Save your changes.

Programming the flag to run the application

Do you remember the fullscreen mode which is used to run an application? The stage covers the whole screen when you are in fullscreen mode, hence you cannot click on the “move 10 steps” block in the Scripts Area or the Block Palette to make our wizard move.

Figure 3.30

Fortunately, the fullscreen stage provides us with two tools to control the running of our application: the flag (start) and stop buttons. We can program the flag to run a script when it is clicked by using a block from the Events category.

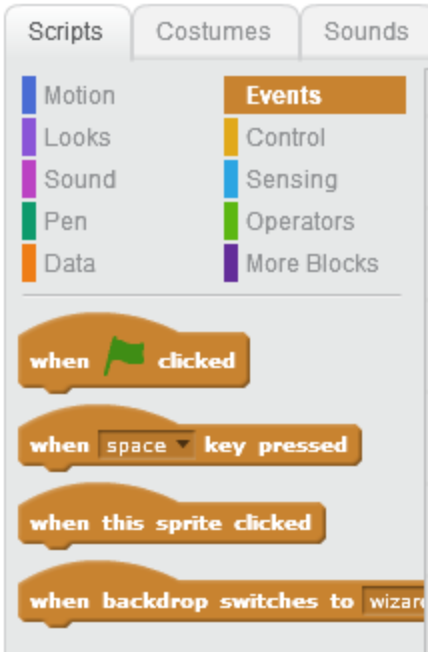


Figure 3.31

Can you see the block ?

Drag the first block in the palette (when clicked) to the Scripts Area.

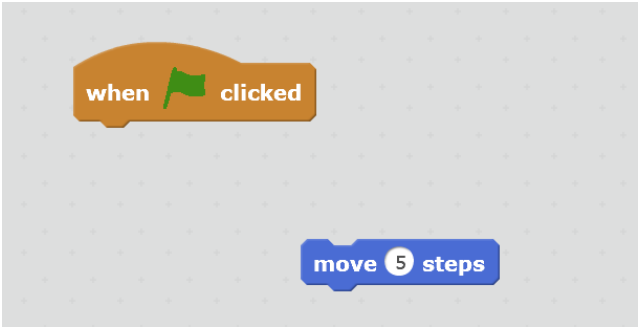


Figure 3.32

This type of block is called a trigger block, as it can only be used to start a script. Hence (and due to its "hat" structure), you can only place it at the top of a script. Snap the "when flag clicked" block to the top of the "move 5 steps" block.

Figure 3.33

Now, switch to fullscreen mode and click on the flag at the top right of the stage. Does the wizard move 5 steps?

Note: We have made further changes to our project. Save your changes.

We have succeeded in making the wizard move. But there is a problem: it moves only once. This is not like the final application where the wizard keeps moving left and right until the application is stopped after we click on the stop button beside the flag button.

Therefore, we need a block that can help us control the "move 5 steps" block to run forever as long as the application has not been stopped. Let us look at the **Control** Block Category to see if we might find what we need there.

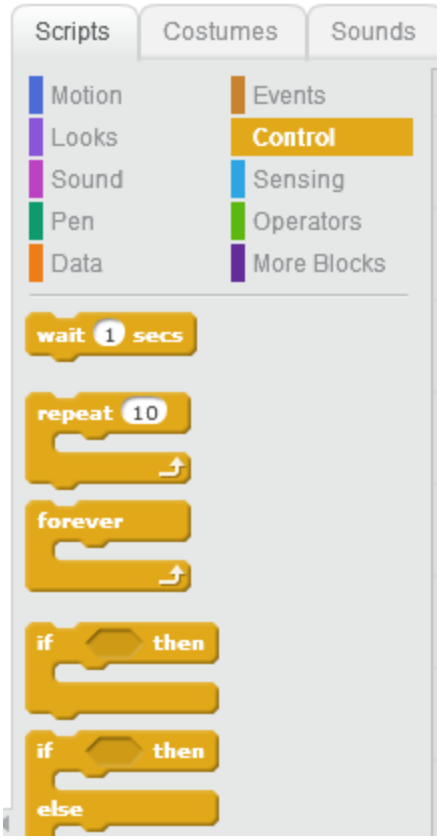


Figure 3.34

Can you see the block?

We will learn more about these Control blocks in forthcoming chapters, but for now, the block we need from here is the "forever" block. It is a stack block, so scripts can be nested inside it, and any script or block nested inside it would be run over and over again until it is stopped by another block, or until the application is stopped.

Drag the forever block to the Scripts Area.

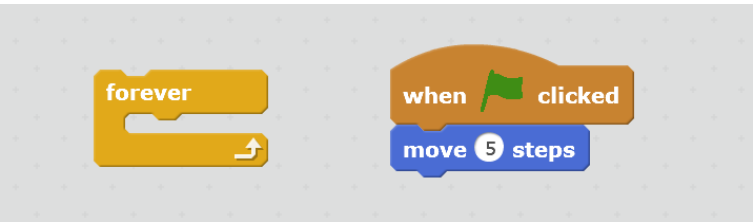


Figure 3.35

We want to run the "move 5 steps" block forever. Detach the block from the event block and nest it between the *forever* block.

Figure 3.36

TRY IT OUT: Click on the forever script... What happens to the wizard as it gets to the right side of the castle?

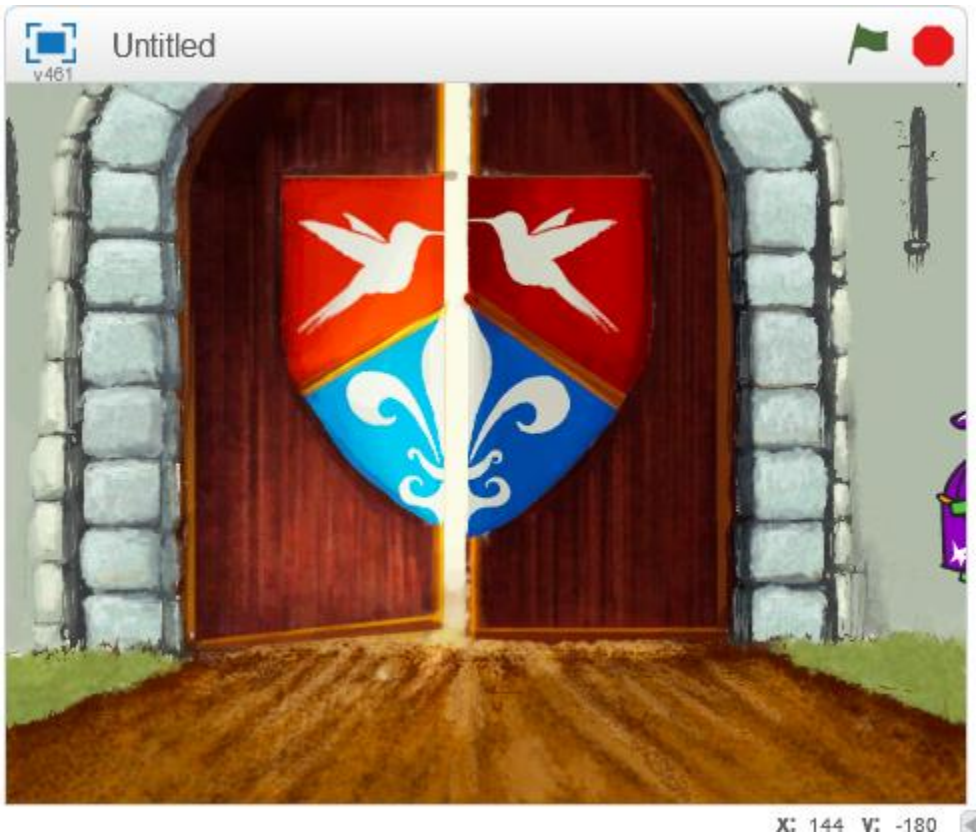


Figure 3.37

Uh-oh! The wizard does not stop when it gets to the right side of the castle; instead, it keeps going until it can no longer be seen on the screen. This is not what we want, and it is not how it is in the final version of the application. Let us fix that.

This is a problem with motion, so let us switch back to the Motion category and look through the blocks in the Blocks Palette to see which one can help us fix this issue.

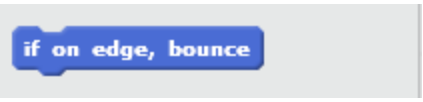


Figure 3.38

This says “if on edge, bounce”; that is exactly what we want! We want the wizard to turn back when it gets to the edge of the castle, and start patrolling the other way. Let us add this block to our forever script.

Figure 3.39

TRY IT OUT: Now click on the forever script in the Scripts Area to run our updated script. What happens as the Wizard gets to the edge of the stage?

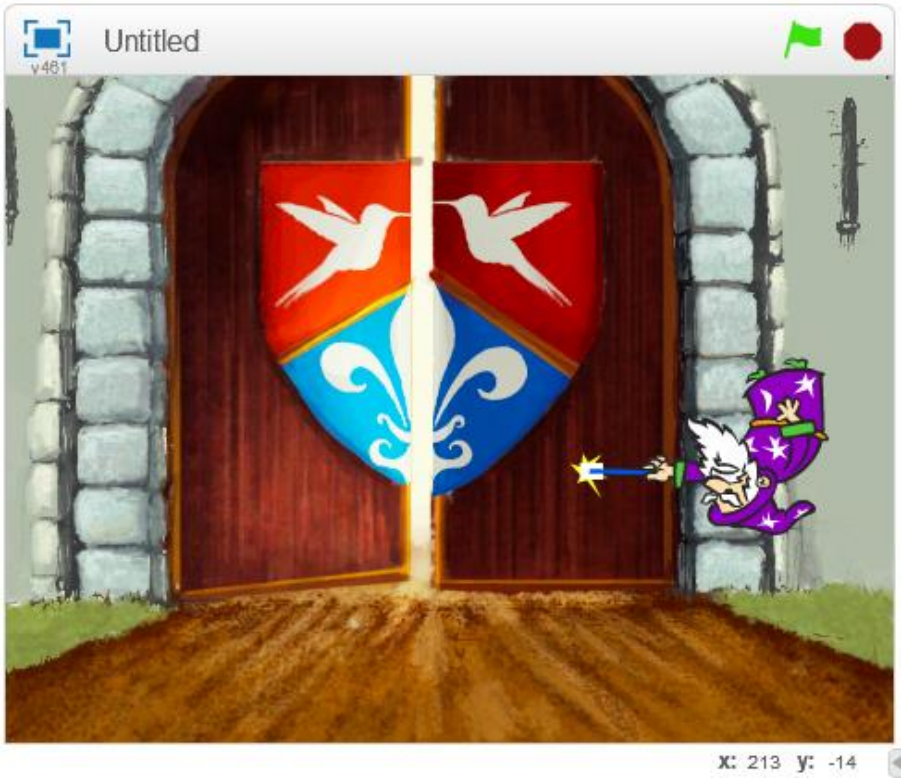


Figure 3.40

The wizard turns to the other direction and keeps patrolling (which is what we want), but it also turns upside down (which is not what we want).

FUN BIT: Now we know that wizards are capable of turning upside down and walking with their heads (hahaha), but let us keep things simple for now for learning purposes😁.

This indicates that there is a problem with the rotation of the wizard sprite. Remember the *rotation style* property of sprites that was discussed in the last chapter? Let us check it out to see if we can fix this issue there.

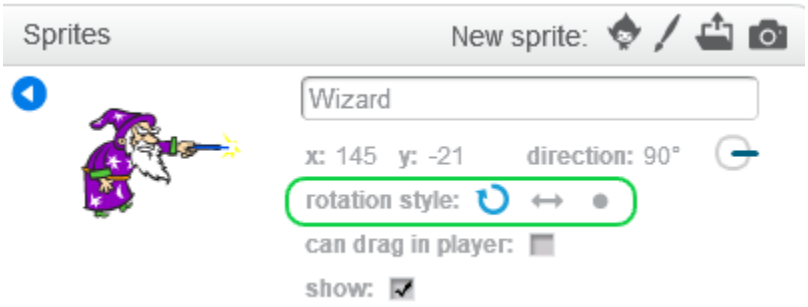


Figure 3.41

Looking at the *rotation style* property, the wizard sprite is currently set to rotate in any direction. However, we want our wizard to move only in the left and right directions.

TRY IT OUT: Change the value of the *rotation style* property and look at the stage. Has our rotation problem been fixed?

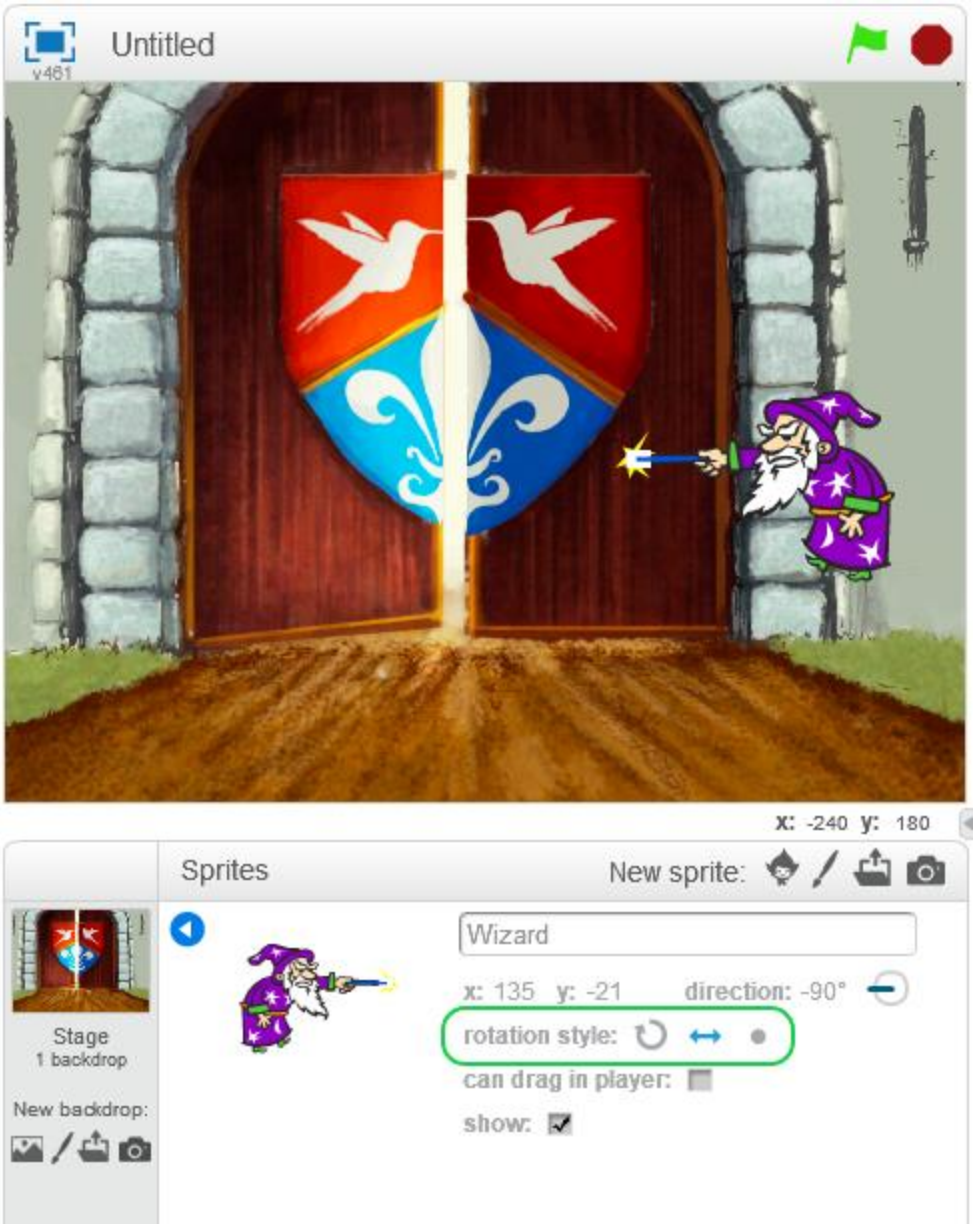


Figure 3.42

Yes! The issue has been fixed and the wizard now patrols the castle in the right direction.

Let us now switch to fullscreen mode to view our application properly. Remember that we cannot view the Scripts Area in fullscreen mode, so we need to snap our flag control block back on top of the forever script.

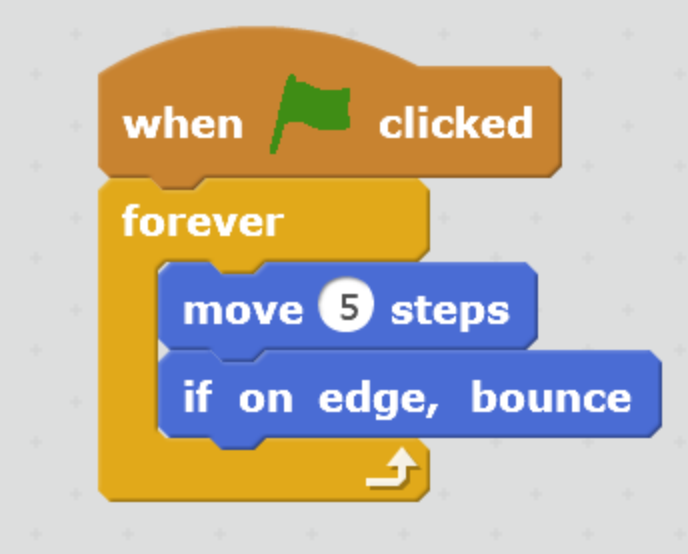


Figure 3.43

Figure 3.44

Great! Our application now works as it should, but there is one more thing that should be fixed; the wizard is not in the correct position on the Stage. Move the wizard down to the ground in front of the castle.
FUN BIT: Again, we know a wizard can walk on air, but let us keep things simple for learning purposes.

Figure 3.45

The application now runs exactly like the final version that was shown at the beginning of the project. Save your project so that you do not lose the progress you have made so far.

Summary of Chapter Three

Apart from properties, you must remember that a Sprite can be given additional attributes to modify its looks or behaviour. These attributes are grouped into Costumes, Blocks and Sounds. The Stage can also be given additional attributes to modify its looks and behaviour. These attributes are grouped into Backdrops, Blocks and Sounds. A Script is a group of Blocks stacked together. Also, the Stage Area contains the Costumes, Backdrops, Scripts (Blocks) and Sounds attributes in tabs. Only one of the Costumes and Backdrops tabs is shown at a time.

You learned about the nine Block Categories in Scratch, and they are contained in the Scripts Tab. There are four kinds of blocks: command blocks which are used to directly give commands to components, control blocks which can contain other blocks within them, trigger blocks used to start scripts, and function (reporter) blocks used to perform actions and as input to other blocks.
And finally, you learnt that the Tips Window contains tutorials and how-to's for different functions on the Scratch Editor, in addition to information about blocks. The Toolbar contains buttons for controlling components of the Scratch Editor, creating new Scratch projects, and opening and saving existing ones.

CHAPTER FOUR

PROGRAMMING APPS WITH MOTION AND DRAWING

Chapter Objectives

At the end of this chapter, the students should be able to:

- **understand how the stage is composed in terms of dimensional and how to utilize its dimensional properties to manipulate components on and around it.**
- **identify how the different blocks from the Motion category can affect behaviours related to motion in app components.**
- **Understand how the different blocks from the Pen category can affect behaviours related to drawing in app components.**

In the last chapter, we created an application featuring a wizard moving from left to right repeatedly in front of a castle. Most, if not all, of the applications and games that you would build with Scratch, require some form of movement. In this chapter, you will learn how to use blocks from the *Motion* category to program sprites to perform different types of motion-related activities. You will also learn how to use blocks from the *Pen* category to program sprites to draw on the stage.

Both motion and pen blocks control a sprite's activities across the Stage, so before jumping into the different blocks offered by these two block categories, let us revisit the Stage and see how its composition affects the position and placement of components (sprites) on it.

Composition of the Stage

The Stage is a rectangular platform on which sprites are placed, and interact with one another. The Stage can appear in three sizes in the Scratch Editor:

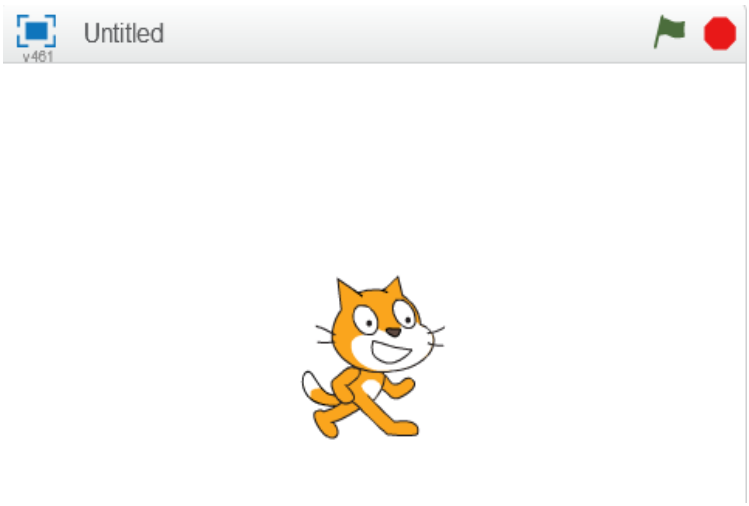


Figure 4.0: Default size

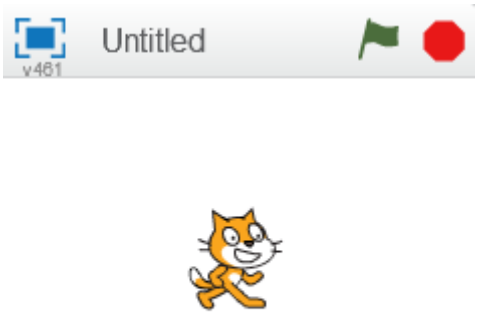


Figure 4.1: Small size

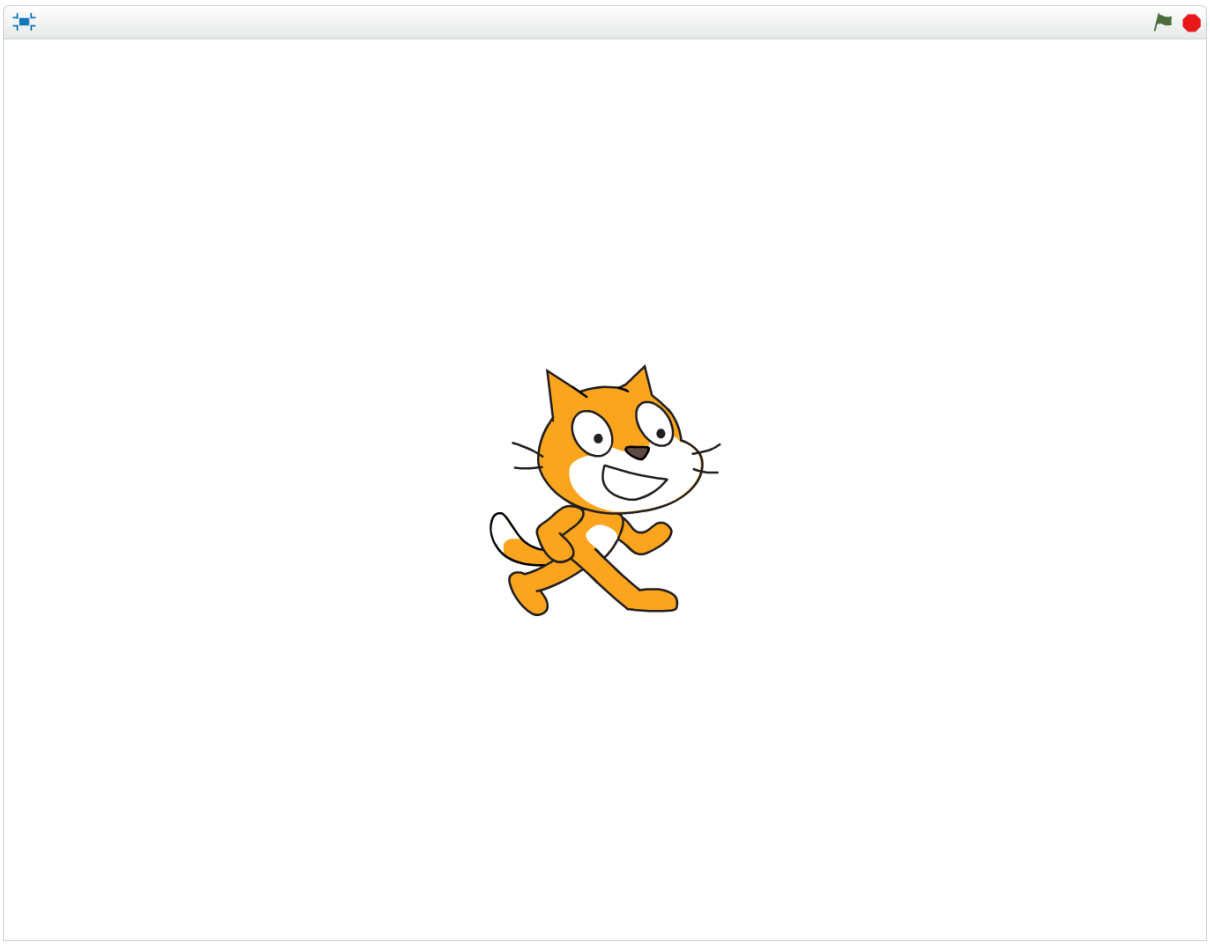


Figure 4.2: Fullscreen size

Regardless of its size at any time, the stage has a constant dimension, which is 480 steps by 360 steps. That is, the stage is 480 steps wide and 360 steps tall.

A coordinate is the position of a point on a surface; most surfaces have horizontal coordinates (also called x-coordinates) and vertical coordinates (y-coordinates). The total number of x-coordinates on the stage is 480, and the total number of y-coordinates on the stage is 360. You can find the x-coordinates and y-coordinates of any point on the stage by moving your mouse to the point, and you will see them at an area below the stage, at the bottom left.

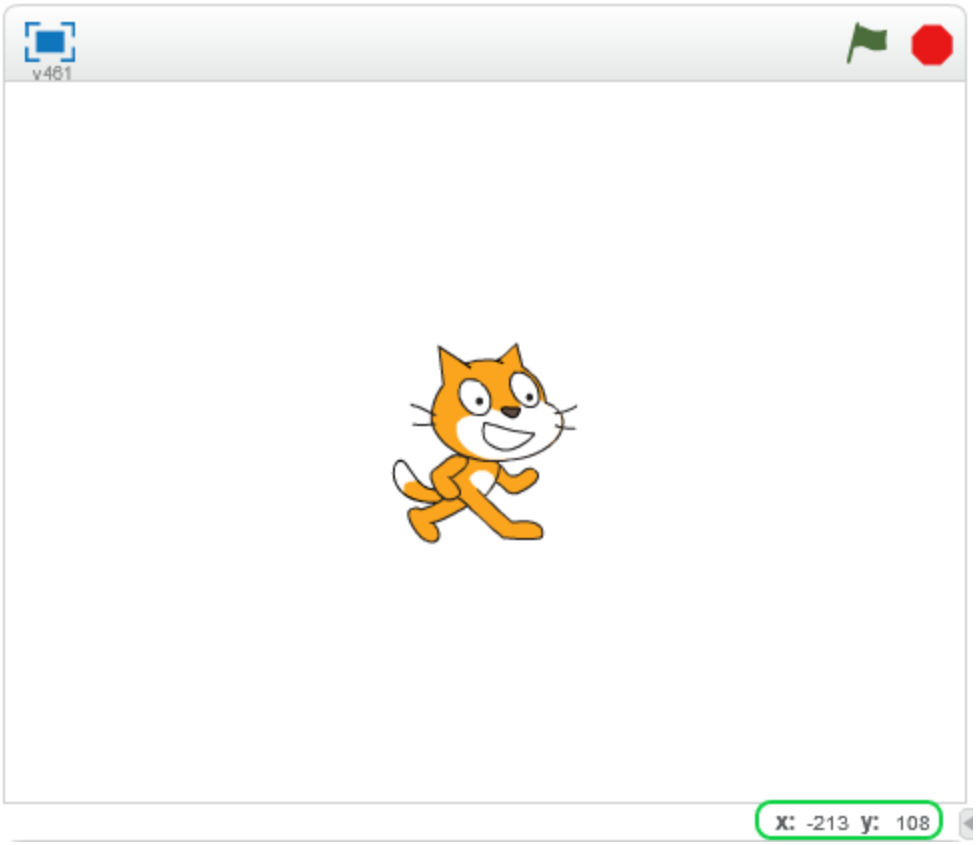


Figure 4.3

The center of the stage has an x-coordinate of 0 and a y-coordinate of 0. Therefore, every point from the center upwards lies between 0 and 180, and every point from the center downwards lies between 0 and -180. Similarly, every point from the center to the right end of the stage lies between 0 and 240, and every point from the center to the left end of the stage lies between 0 and -240.

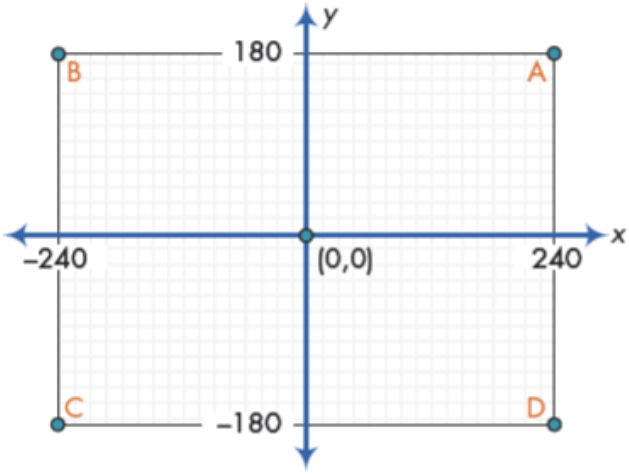


Figure 4.4

A Sprite's position on the stage can be seen on the Sprite List.

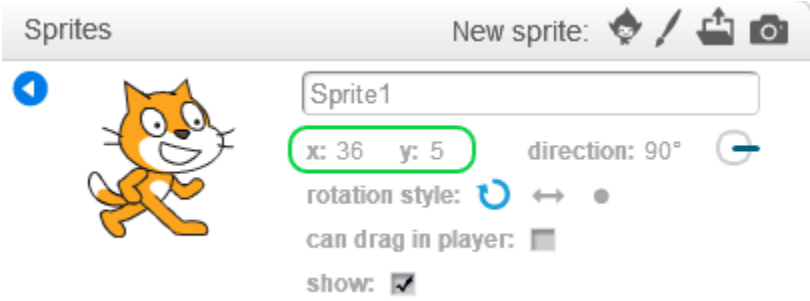


Figure 4.5

In the Patrol Wizard application built in the last chapter, we used a Motion block, *move 5 steps*.

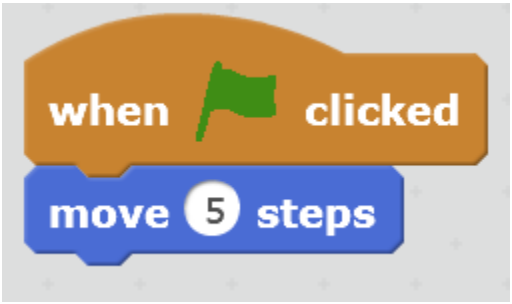


Figure 4.6

When this block is executed, the sprite for which it was created moves 5 steps towards the direction it is facing on the stage. With our newfound knowledge of the stage's dimensions, if the sprite was at point (0,0) (that is, an x-coordinate of 0 and y-coordinate of 0), and facing the right side before this block was executed, the sprite would be at point (5, 0) after its execution. That is, the sprite has moved five steps to the right, and remains on the same coordinate on the vertical axis.

Most sprites in the Sprite Library have costumes that are large and can occupy an area covering more than a single x-y coordinate on the stage. How, then, is it possible to place a sprite at a specific x-y coordinate on the stage?

The Center of a Costume

Every costume has a center point, which determines the xy-coordinates on which any sprite that has it as its current costume is placed on the stage. When we say a sprite is at point (0,0), what we are really saying is that the center of its costume is at point (0,0).

The center of a costume can be set in the Paint Editor.

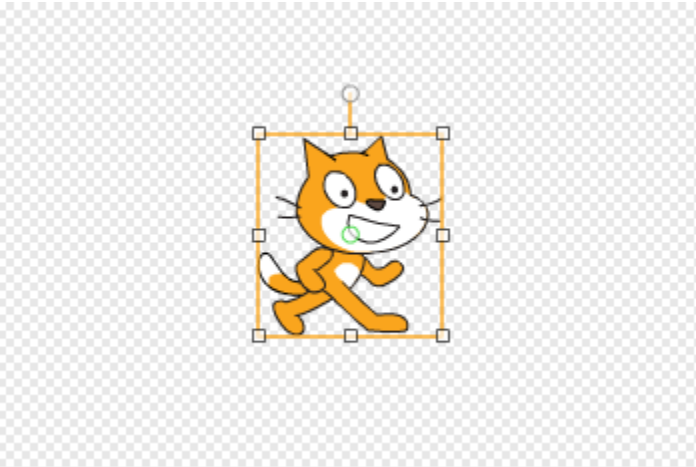


Figure 4.7: The center of the costume is the small green circle at the center.

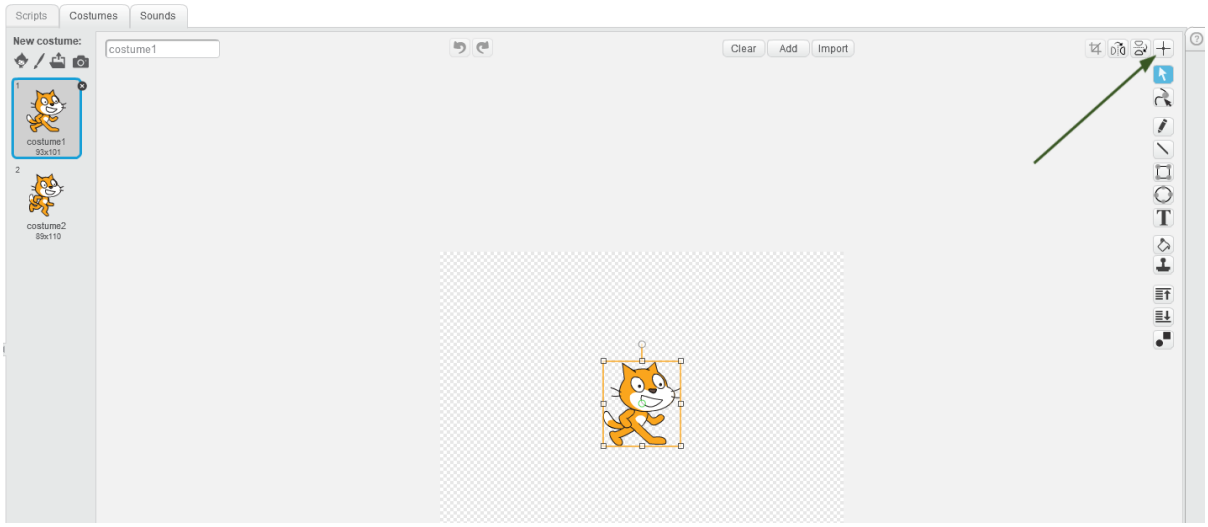


Figure 4.8

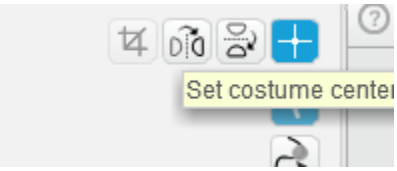


Figure 4.9: The "+" button at the top right of the Paint Editor is used to set the center of a costume.

When the button is selected, one horizontal and one vertical line appear on the costume. The point where these lines meet is the center of the costume. Move the lines around to set the center of a costume to your desired point.

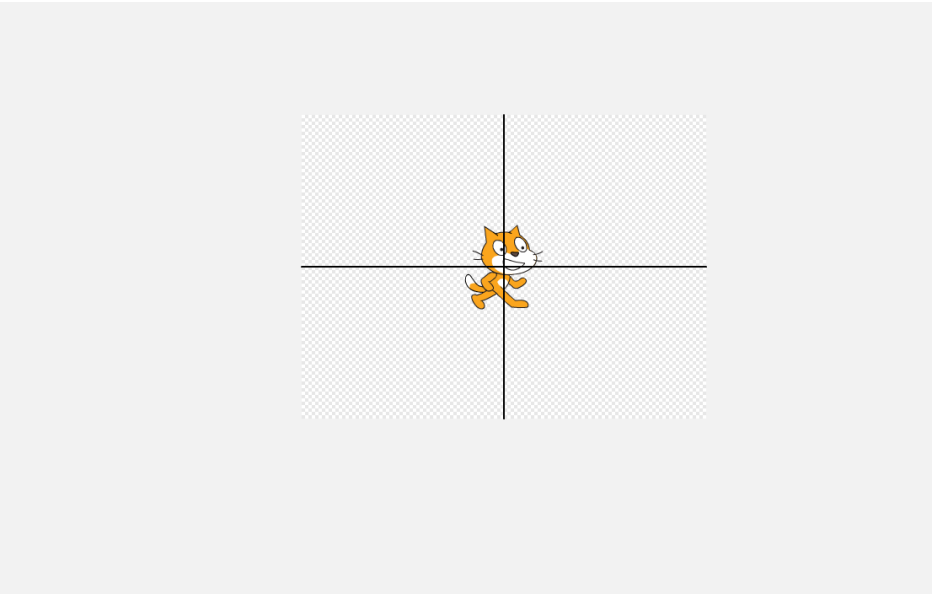


Figure 4.10

The center of a costume also determines the way it turns when its direction property is modified. The default direction of a costume is 90°, that is, it is pointing to the right. So, when a sprite is commanded to change direction, it does so with reference to its costume's center point.

You can observe this behaviour by turning the direction wheel on the Sprite List.

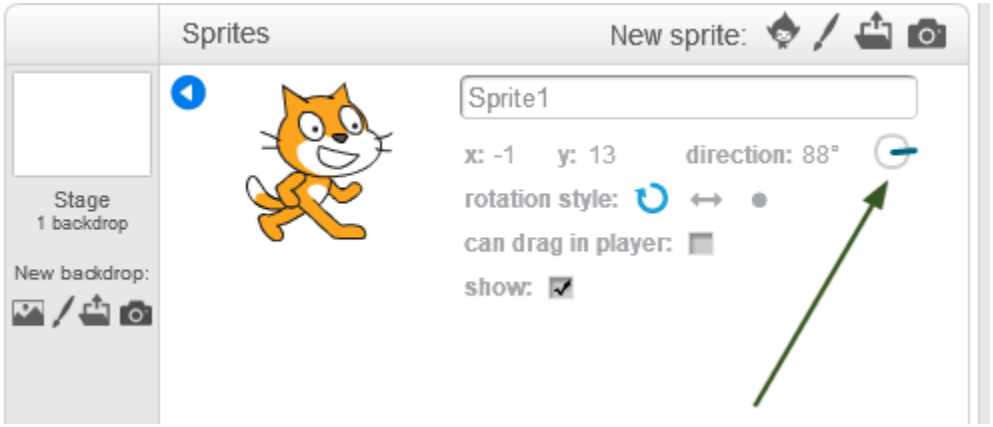


Figure 4.11

The Scratch cat's costume is directed to the right in the Paint Editor. When its *rotation style* property is set to "all around", rotating the direction wheel rotates the sprite around the center point. This is why the wizard sprite in the Patrol Wizard application built in the last chapter was initially walking on air when its direction was changed. See figure 4.12

Figure 4.12

When the *rotation style* property is set to “left right”, the sprite is constrained to rotate only in the left and right directions, so, rotating the direction wheel simply flips the sprite over horizontally.
Therefore, if you have a costume, and you want it to to rotate as if it is the real object, make sure it is facing the right side in the Paint Editor, so that it aligns with a costume’s default direction of 90°.

For example, let us use a beetle sprite in an application.

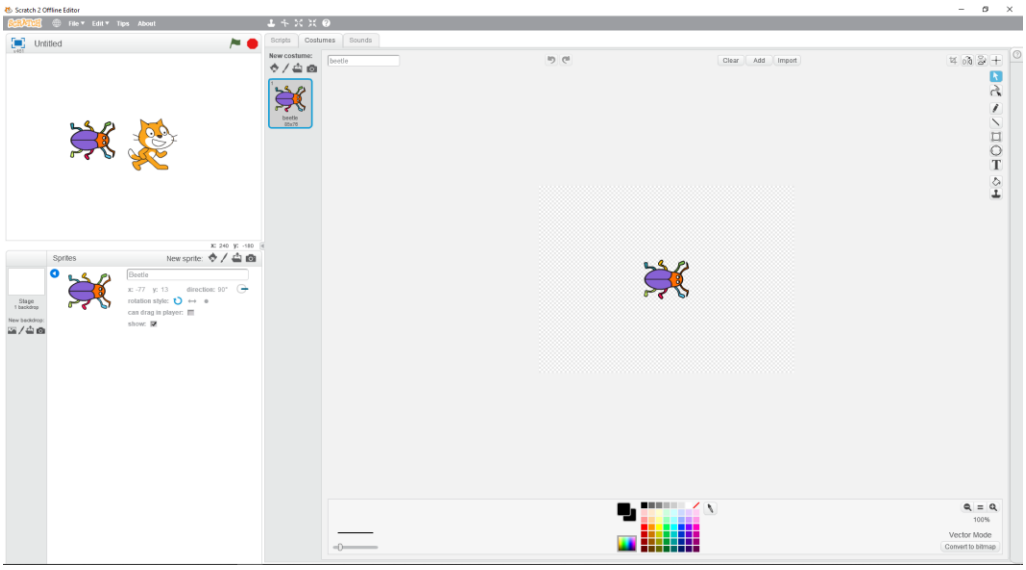


Figure 4.13

The beetle’s costume faces right by default. Select the costume in the Paint Editor and rotate it to face upwards.

Figure 4.14

Notice that the direction wheel still points to the right, even though the beetle looks like it is facing upwards. Now try rotating the wheel to the right. What direction does the beetle face?

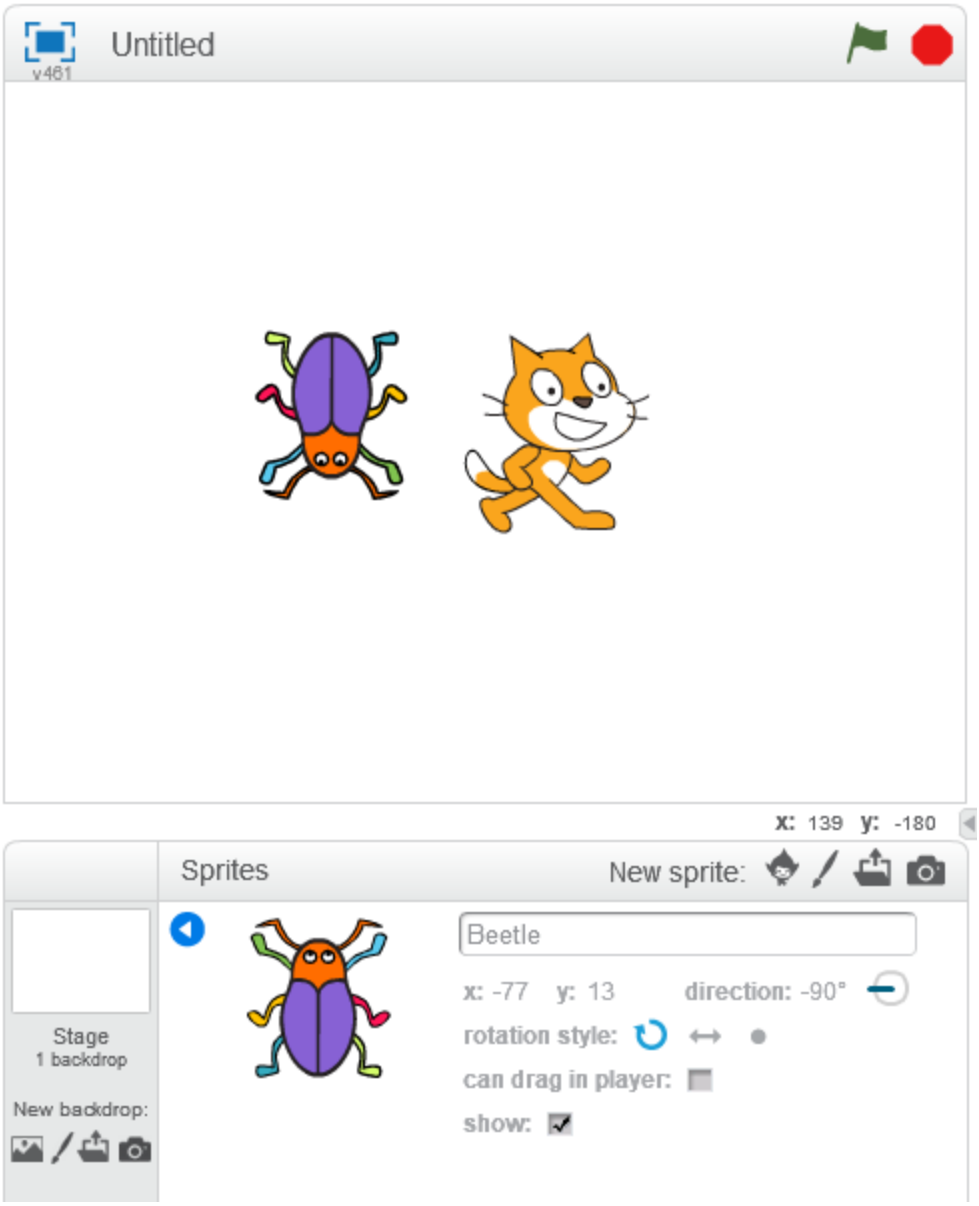


Figure 4.15: The wheel points to the left, but the beetle costume is facing downwards.

Rotating the beetle's costume back to the right in the Paint Editor will fix this.

The Motion Blocks Category

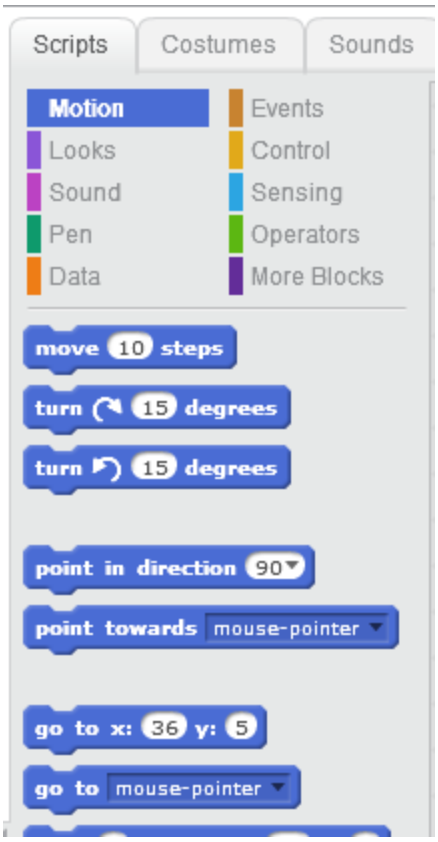


Figure 4.16

Everything you need to make a sprite move around in different ways, is available in the *Motion* blocks category.

The *move n steps* block

Figure 4.17

The ***move n steps*** block is used to move a sprite by a number ("***n***") of steps across the stage, in the direction faced by the sprite at the point of execution of the block. We used this block in the previous chapter to move the patrol wizard from left to right.

This block has an input parameter, which determines the number of steps moved by the sprite when the block is executed. The default is the number "10", and the value can be changed. If *n* is a positive number, the sprite moves forward by *n* steps, and if *n* is a negative number, the sprite moves backwards by *n* steps.

Figure 4.18

The *turn clockwise/anticlockwise n degrees* blocks

Figure 4.19

As shown in figure 4.19 above, first block is used to rotate a sprite in the clockwise direction, and the second is used to rotate a sprite in the anticlockwise direction. Each block has an input parameter, *n*, which determines the number of degrees through which a sprite is rotated when the block is executed. The default is the number "15", and can be changed.

If *n* is a positive number, the sprite moves in the direction specified by the arrow on the block. If *n* is a negative number, however, the sprite moves in a direction opposite to the arrow on the block.

The *point in direction* block

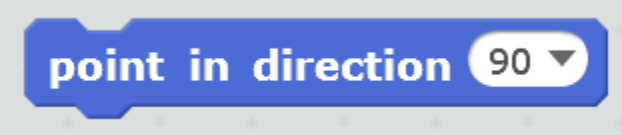


Figure 4.20

The ***point in direction*** block is used to rotate a sprite to a specific direction. Unlike the *turn n degrees* blocks, you can only point a sprite upwards, downwards, left and right with this block. Its input parameter determines the direction in which a sprite is pointed when the block is executed. The default point is the number "90", and can be changed by selecting one of the options in the dropdown. See figure 4.21 below.

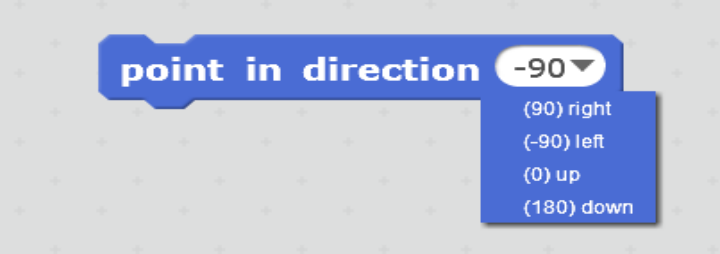


Figure 4.21

The *point towards n* block

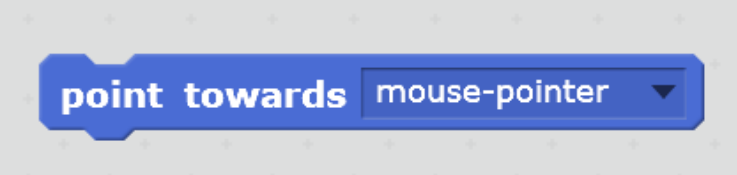


Figure 4.22

This block is used to rotate a sprite to face a specific component. The default input parameter is your computer's mouse pointer, and every sprite added to your project will appear in the input options dropdown.

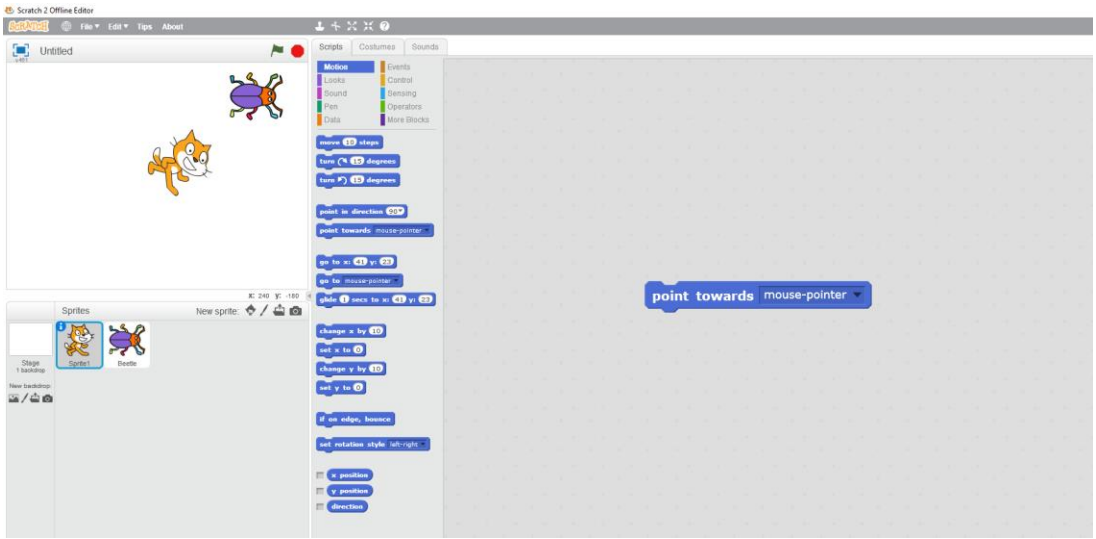


Figure 4.23

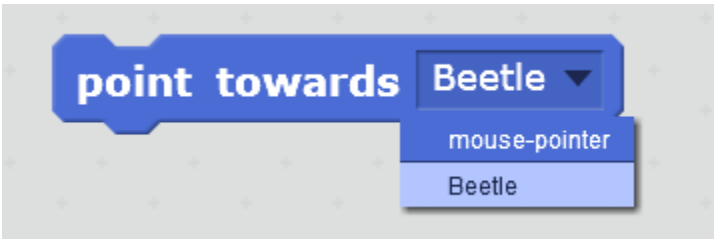


Figure 4.24

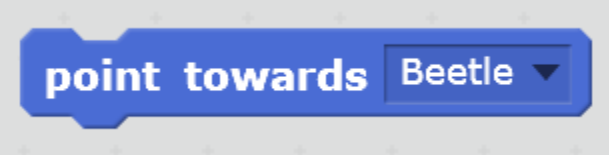


Figure 4.25

The *go to x, y* block

Figure 4.26

This block is used to move a sprite to specific x and y coordinates on the stage. It is useful when you know exactly where you want a sprite to be on the stage. The first input parameter sets the x-coordinate of the sprite on the stage, and the second parameter sets its y-coordinate.

The *go to x* block

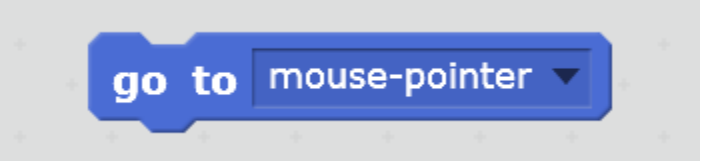


Figure 4.27

This block is used to move a sprite to the exact position of another component or the mouse pointer on the stage.

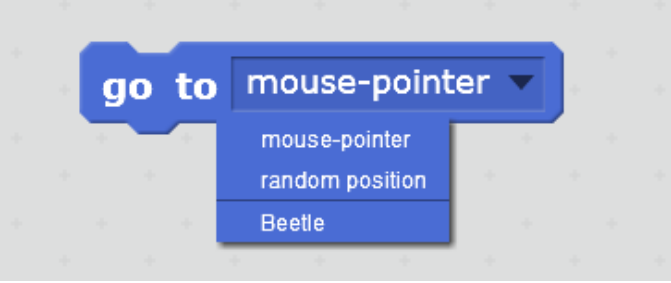


Figure 4.28

The *random position* input option moves the sprite to a random position on the stage every time the block is executed and the block is selected.

When a sprite is selected as the input parameter and this block is executed, the sprite on which it is executed moves so that its x and y coordinates match those of the sprite in the input parameter.

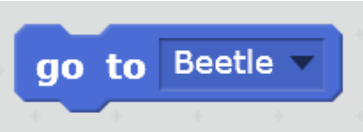


Figure 4.29

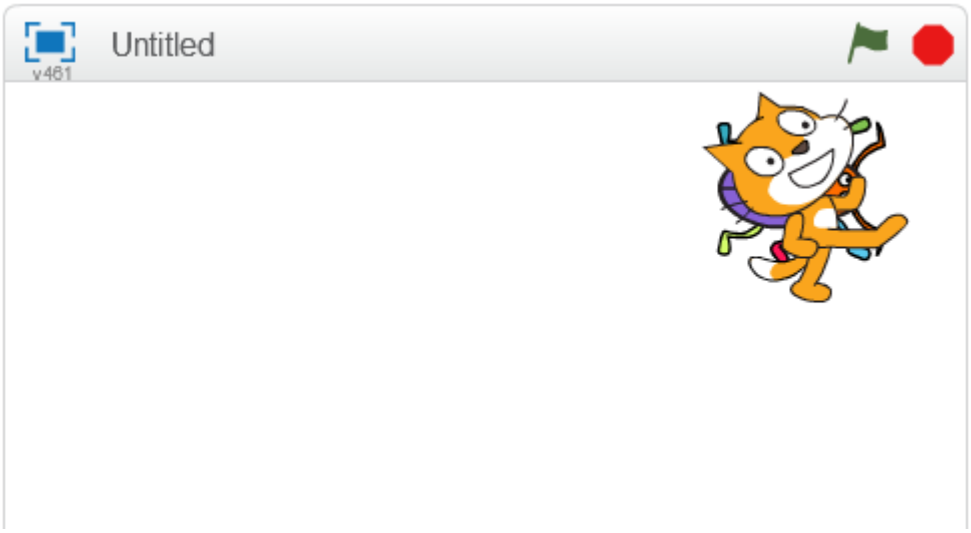


Figure 4.30

The *glide n secs to x, y* block

Figure 4.31

This block operates like the *go to x, y* block, but unlike the latter which moves the sprite to the x and y-coordinates in the block's input parameter abruptly, the '*glide...*' block animates the movement. The speed of the animation depends on the *n* input parameter and the distance between the sprite's current position and the destination coordinates, specified by the *x* and *y* input parameters. *n* is in seconds and the total time of animation from the current location to the destination is equal to this value, so the farther the sprite is from its destination, the faster it moves to get there. If *n* is a positive number, this block behaves like the *go to x,y* block.

The *change x/y by n* blocks

Figure 4.32

These blocks are used to change the x and y coordinates of a sprite on the stage by a number, specified by the *n* input parameter. If *n* is a positive number, the x/y coordinate increases by that number, and if *n* is a negative number, the x/y coordinate decreases by that number. Increasing or decreasing the x and y coordinate of a sprite moves the sprite forwards and backwards, horizontally and vertically, respectively on the stage.

The *set x/y to n* blocks

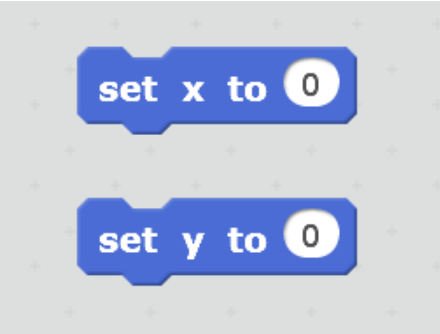


Figure 4.33

These blocks are used to set the x and y coordinates of a sprite on the stage to a specific number, specified by the *n* input parameter.

The difference between these '*set…*' blocks and the '*change by…*' blocks is that the former modify the x and y properties of the sprite directly, while the latter increase or decrease the x and y properties of the sprite by a number. If the '*set…*' blocks are called repeatedly with the same *n* parameter on a sprite, the sprite remains in the same position, but if the '*change…*' blocks are called repeatedly with the same *n* parameter on a sprite, the sprite keeps moving forward or backward because its x or y coordinate is being increased or decreased.

The *if on edge, bounce* block

Figure 4.34

We have seen this block in use before. It is used to prevent a sprite from going into the edges of the stage, and bounce back in the opposite direction instead.

The *set rotation style* block

Figure 4.35

Figure 4.36

This block is used to modify the *rotation style* property of a sprite. Blocks like this one, that modify a component's properties directly with the *set* command, are also called *property setter* blocks.

Motion property reporter blocks

The blocks we have seen so far in the Motion blocks palette are command blocks. The palette also contains reporter blocks that report a sprite's properties.

Figure 4.37

The *x position* block reports the current x-coordinate of a sprite on the stage, The *Y position* block reports the current y-coordinate of a sprite on the stage, and the *direction* block reports the current direction a sprite is facing on the stage. Blocks like this one, that report a component's properties, are also called *property getter* blocks.

These property getter blocks can also be used to track the properties they report in real-time on the stage. To do this, tick the checkbox next to them in the Blocks palette as shown in figure 4.37.

Figure 4.38

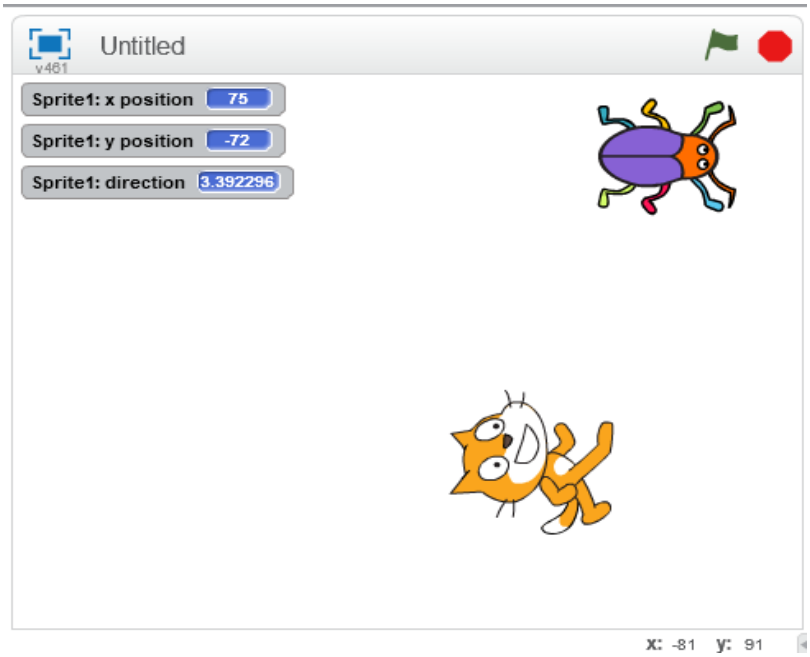


Figure 4.39

Motion blocks for the Stage

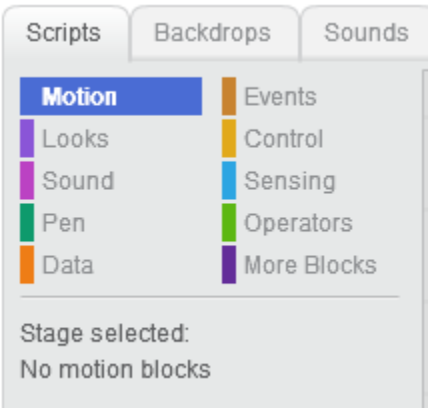


Figure 4.40

The Stage is a stationary object, and therefore has no motion blocks.

Every Sprite has an inbuilt pen that can be used to make colourful drawings on the stage.

The Pen Blocks Category

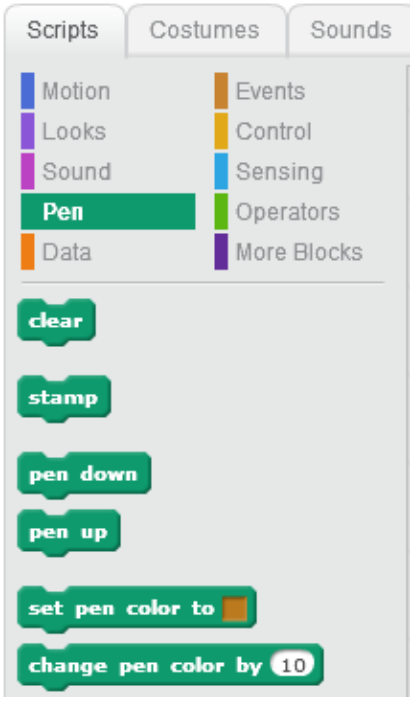


Figure 4.41

Everything you need, to control the usage of a sprite's inbuilt pen is available through the blocks contained in the *Pen* blocks category. A sprite's inbuilt pen is invisible, and is up by default. This is why all of your motion commands so far have not resulted in the sprites drawing anything on the stage while they move around it. When the sprite's inbuilt pen is down, it will draw on the stage as it is moved around it using motion blocks.

The *pen down/up* blocks

Figure 4.42

These blocks are used to put a sprite's inbuilt pen up or down; when the pen is down, the sprite does not draw as it moves around the stage, and vice versa.

TRY IT OUT: Execute the ***pen down*** block on a sprite and move it around the stage with one of the blocks from the motion category. What happens?

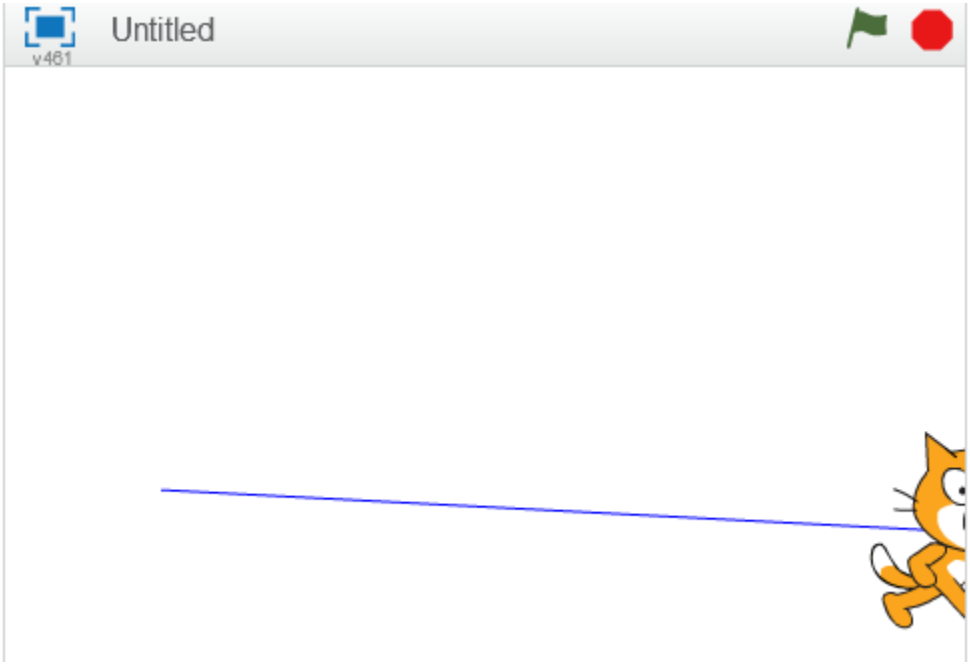


Figure 4.43

The costume's center comes into play here again. The inbuilt pen is located at the center of the sprite's costume, so when the pen is put down, the sprite draws along its center point as it moves around the stage.

The *set pen color to* blocks

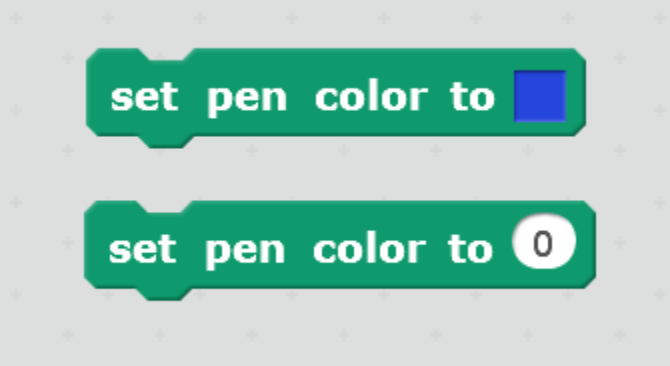


Figure 4.44

These blocks are used to set the color of the lines drawn with a sprite's pen.

For the first block, a colour can be picked from any object in the Scratch Editor and passed as an input parameter to this block by clicking on the color picker (the colored box on the block) and moving your mouse cursor to the desired colour.

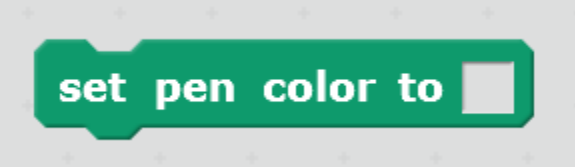


Figure 4.45

The colour picker is used to set the pen colour to the background colour of the Scripts Area.

For the second block, a number can be passed as an input parameter to set the pen's colour. The colours that can be used in Scratch are the colours of the rainbow (Red, Orange, Yellow, Green, Blue, Indigo, and Violet). A number between 0 and 200 can be chosen; 0 and 200 represent red, and all other colours fall in between in different hues, in the order in which they appear in the rainbow.

TRY IT OUT: Set the colour of the pen to different numbers between 0 and 200 and move the sprite across the stage with a motion block. Observe how the colour of the lines drawn change.

The *change pen shade by n* block

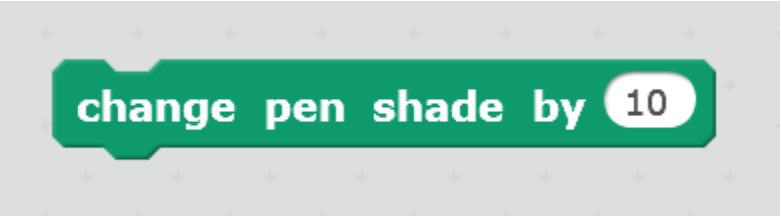


Figure 4.46

A pen's shade refers to how dark or light the pen's colour is drawn. The pen can have a shade between 0 and 100, and the default pen shade for every sprite is 50 (a higher number makes the colour darker, and a lower number makes the colour lighter). This block increases the pen shade by *n* if it is positive, and decreases it by *n* if it is negative.

The *set pen shade to n* block

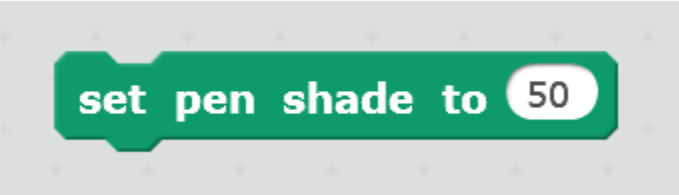


Figure 4.47

This block directly sets the shade of a sprite's pen to *n*.

The *change pen size by n* block

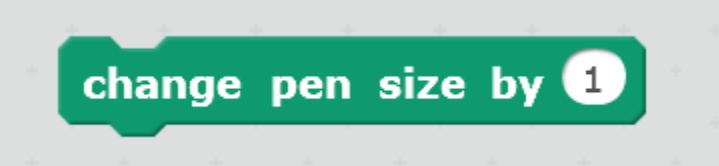


Figure 4.48

The last property of a sprite's pen is size. The size of the pen determines the size of the lines drawn by the sprite as it moves across the stage. The default size is 1, which is why the lines drawn by the sprite so far are so thin. This block increases the size by *n* if it is positive, and decreases the size by *n* if it is negative.

The *set pen size to n* block

Figure 4.49

This block directly sets the size of a sprite's pen to *n*.

Asides drawing lines on the stage as it moves about, a sprite can also draw images of itself on the stage. This is called *stamping*, as the sprite draws an exact replica of its costume on the stage.

The *stamp* block

Figure 4.50

This block is used to instruct a sprite to stamp its costume in its current position on the stage.

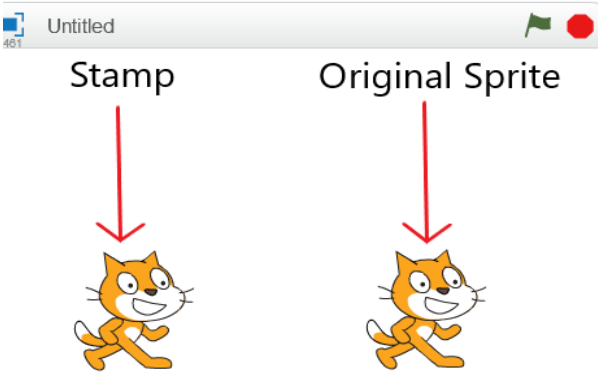


Figure 4.51

The stamp is made at the exact x and y coordinate and center point of the sprite's costume. So, you have to move your sprite to another location after stamping to see the stamped image.

Note that a stamp does not represent a sprite; it is treated just like the lines drawn by a sprite's pen. The stamp command is executed whether the sprite's pen is down or not.

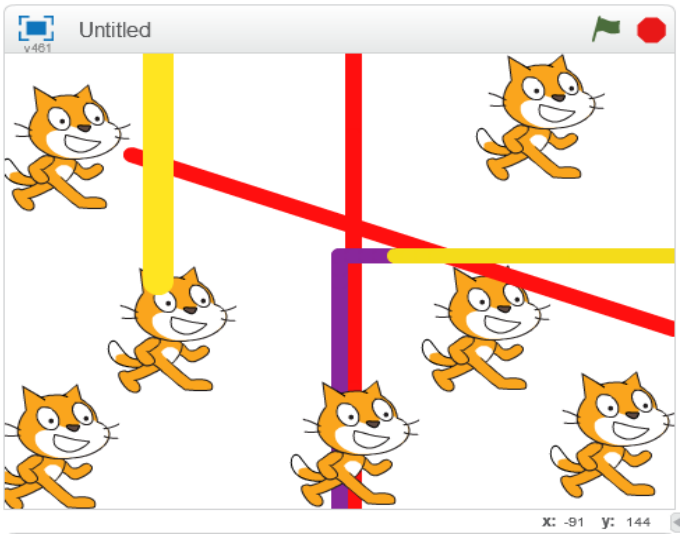


Figure 4.52

We have now made a mess of the stage, and cannot seem to remember which of the cat images is the real sprite. Is there a block that can help us fix this?

The *clear* block

Figure 4.53

This block removes all drawings and stamps made by all sprites from the stage.

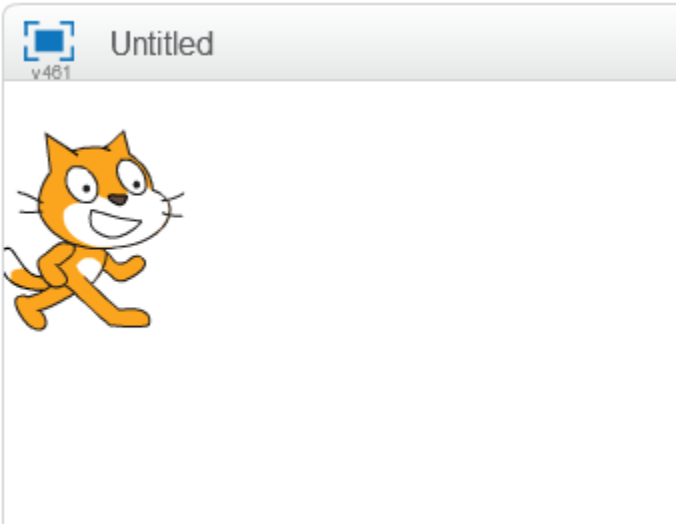


Figure 4.54: Ahahaha! This is the original sprite.

Pen blocks for the Stage

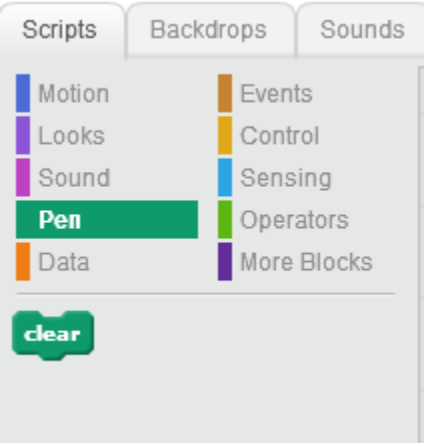


Figure 4.55

The Stage has just one block in the *Pen* blocks palette: the *clear* block. This block works exactly like that on the palette for sprites.

Summary of Chapter Four

You have learned that the stage can appear in three sizes (default, small & presentation) in the Scratch Editor, but regardless of its size at any time, the stage has a constant dimension of 480 steps by 360 steps. The center of the stage has an x-coordinate of 0 and a y-coordinate of 0.

Then, you learned that every costume has a center point, which determines the xy-coordinates on which any sprite that has it as its current costume is placed on the stage. It can be set in the Paint Editor and it also determines the way a sprite turns when its direction property is modified.

And know that everything needed to make a sprite move around in different ways is available through the blocks contained in the *Motion* blocks category. The Stage is a stationary object, and therefore has no motion blocks.

Finally, every Sprite has an inbuilt pen that can be used to make colourful drawings on the stage. The Pen Blocks category contains everything needed to control the usage of this inbuilt pen.

MODULE 5

Project: The Drawing Bee application

Table of Contents

Learning Objectives

By the end of this module, the student would:

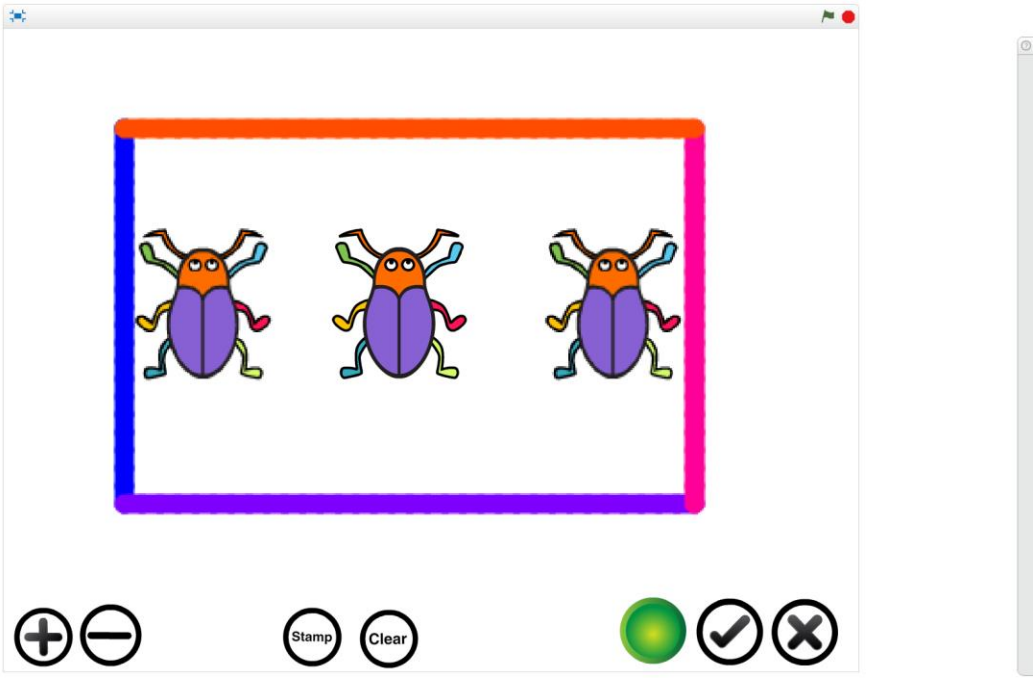
- Be able to build the Drawing Bee application.
- Be able to use various tools in the Paint Editor to modify a costume's looks.
- Be able to program a component based on other components' blocks and scripts.

Module Prerequisites

- Scratch 2 Offline Editor set up on the computer with access to local storage.

In the last module, we explored the blocks in the *Motion* and *Pen* block categories. These are the blocks used to program sprites to move around the stage in various ways and make artistic drawings while at it. We will now apply the knowledge gained from the last module to build an application called Drawing Bee.

The Drawing Bee application features a Bee which can draw lines and stamp itself on the stage. The bee is moved around the stage with the computer keyboard's arrow keys and by clicking on a point on the stage. The bee's pen is put up and down with the ✓ and ✕ buttons at the bottom right of the application, and the multi-coloured button to their left are used to change the pen's colour. The buttons at the bottom centre of the application are used to stamp the bee on the stage, and clear the stage. The buttons at the bottom right of the application are used to increase and decrease the size of the bee's pen.



We would follow the application development process established in the previous module to build the Drawing Bee app.

Determine all the functions needed in the app

All the functions needed in the Drawing Bee app have been stated above. Let us outline them for easy reference:

A bee that:

- Moves when the keyboard's arrow keys are pressed.
- Moves when the stage is clicked.
- Draws lines on the stage when its pen is put down with a button.
- Stamps itself on the stage when the *stamp* button is clicked.
- Changes its pen colour when the multi-coloured button is clicked.
- Changes its pen size when the + and - buttons are clicked.
- Clears the stage when the *clear* button is clicked.

Decide which components are needed to perform these functions

Look closely at the final version of the application shown above and point out the different backdrops, sprites, costumes and sounds present. List them in your Project Components table as you identify them.

PROJECT COMPONENTS

COMPONENT	FINDINGS
Backdrop	• 1 White Background
Sprite	• 1 Bee • 7 Buttons
Costume	• 1 Multi-coloured bee • 1 circle with a white background and black border with a "-" icon in it • 1 circle with a white background and black border with a "+" icon in it • 1 circle with a white background and black border with the text "Stamp" written in it • 1 circle with a white background and black border with the text "Clear" written in it • 1 multi-coloured circle • 1 circle with a white background and black border with a "✓" icon in it • 1 circle with a white background and black border with a "✕" icon in it
Sound	None

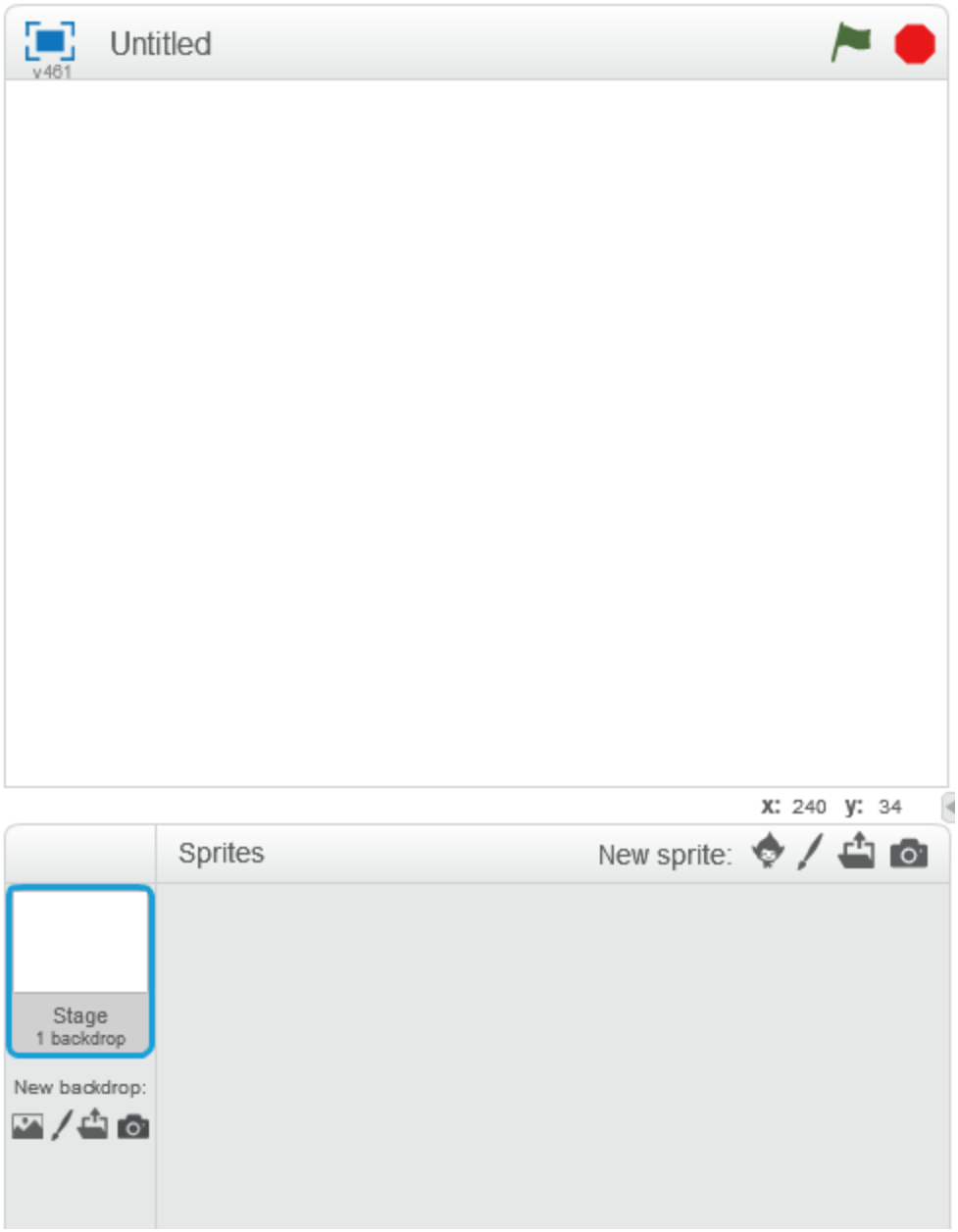
Add these components to the application

It is now time to jump into the Scratch Editor.
Open the Scratch 2 Offline Editor application on your computer and a new project containing the Scratch cat will be automatically opened for you.

We do not need the Scratch cat sprite, so go ahead and delete it from the Sprite List.

Backdrops

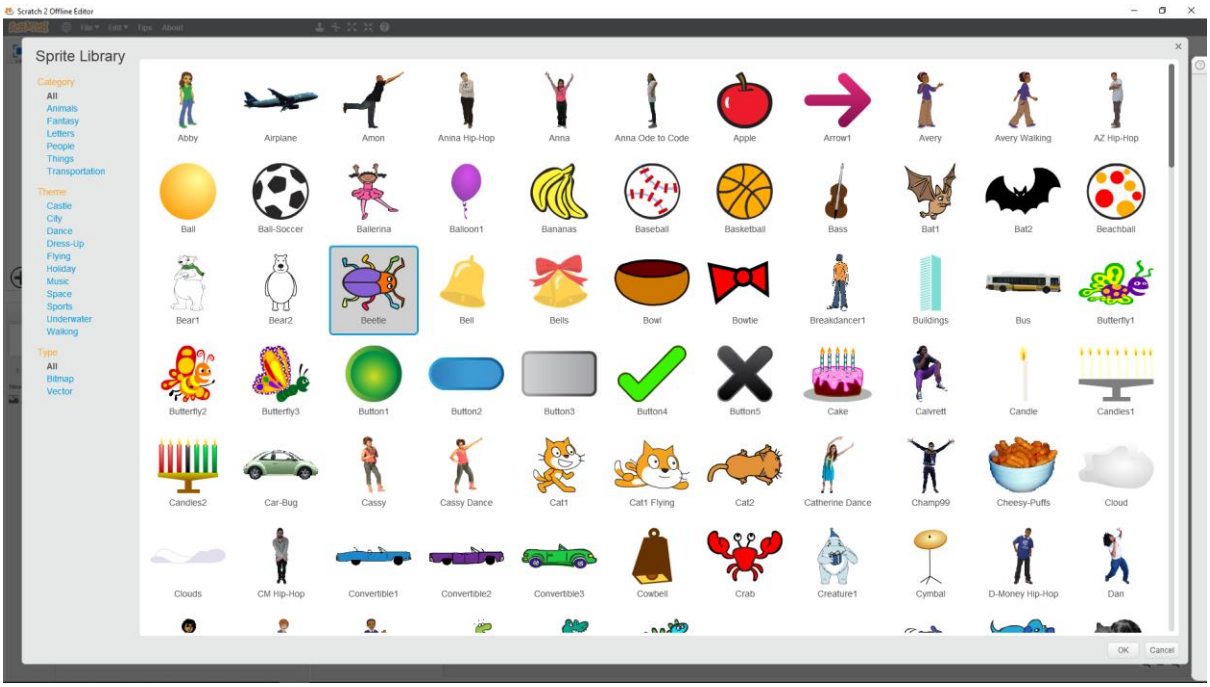
The first component on our **Project Components** table above is a backdrop with a white background.

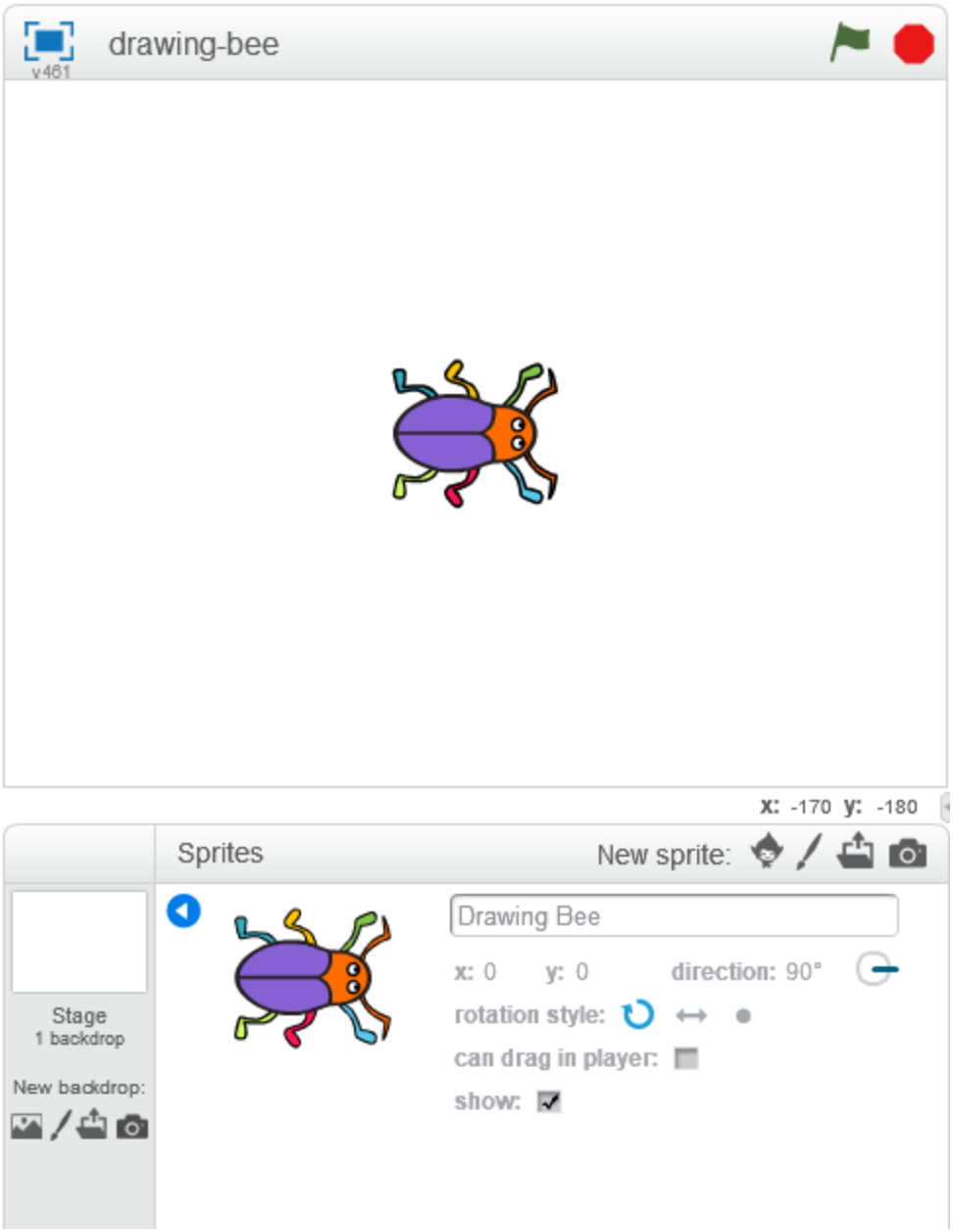


This is added by default to the new project created, so save the project with the name of the application - Drawing Bee.

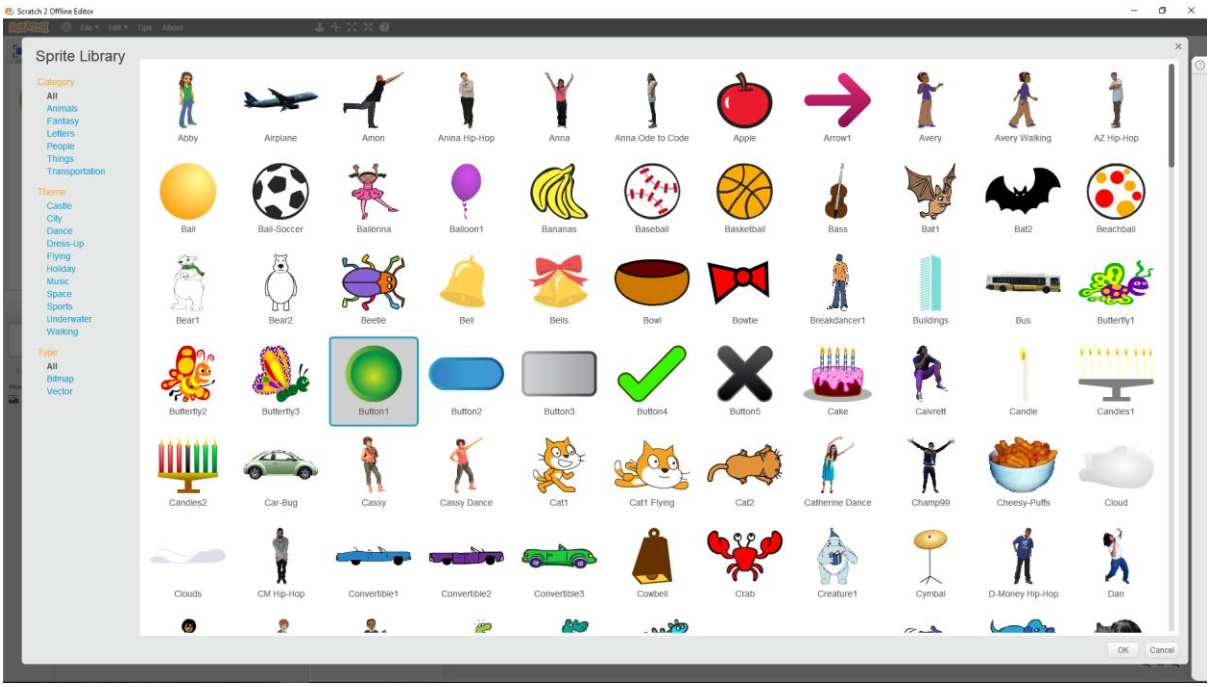
Sprites

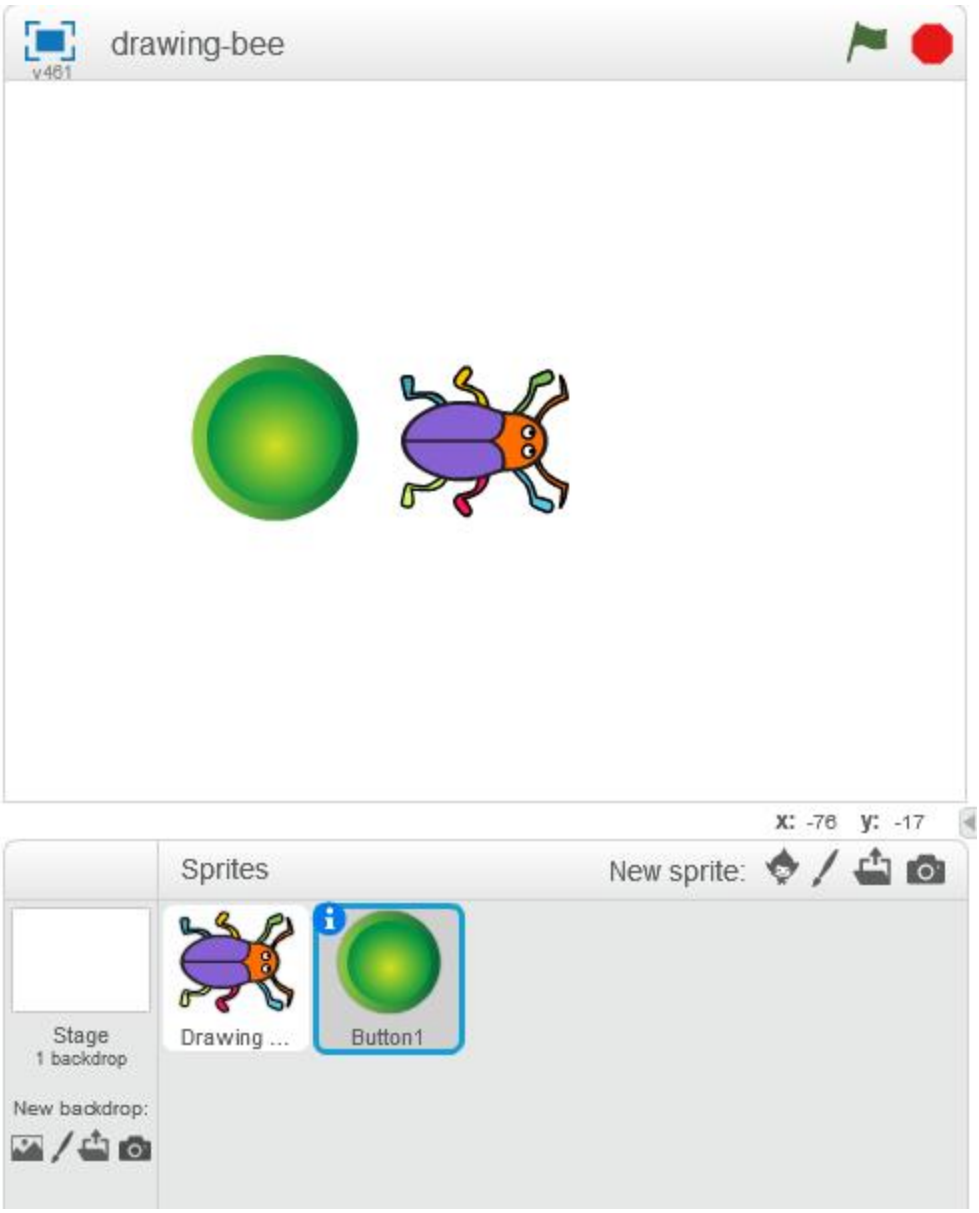
The second component on our **Project Components** table is a bee sprite. Add this sprite to the Sprite List from the Scratch Sprite Library and change its name to “Drawing Bee”.


Scratch 2 Offline Editor
Sprite Library
Category
All
Animals
Fantasy
Letters
People
Things
Transportation
Theme
Castle
City
Dance
Dress-Up
Flying
Holiday
Music
Space
Sports
Underwater
Walking
Type
All
Bitmap
Vector
Abby
Airplane
Amon
Anina Hip-Hop
Anna
Anna Ode to Code
Apple
Arrow1
Avery
Avery Walking
AZ Hip-Hop
Ball
Ball-Soccer
Ballerina
Balloon1
Bananas
Baseball
Basketball
Bass
Bat1
Bat2
Beachball
Bear1
Bear2
Beetle
Bell
Bells
Bowl
Bowtie
Breakdancer1
Buildings
Bus
Butterfly1
Butterfly2
Butterfly3
Button1
Button2
Button3
Button4
Button5
Cake
Calvrett
Candle
Candles1
Candles2
Car-Bug
Cassy
Cassy Dance
Cat1
Cat1 Flying
Cat2
Catherine Dance
Champ99
Cheesy-Puffs
Cloud
Clouds
CM Hip-Hop
Convertible1
Convertible2
Convertible3
Cowbell
Crab
Creature1
Cymbal
D-Money Hip-Hop
Dan
OK
Cancel

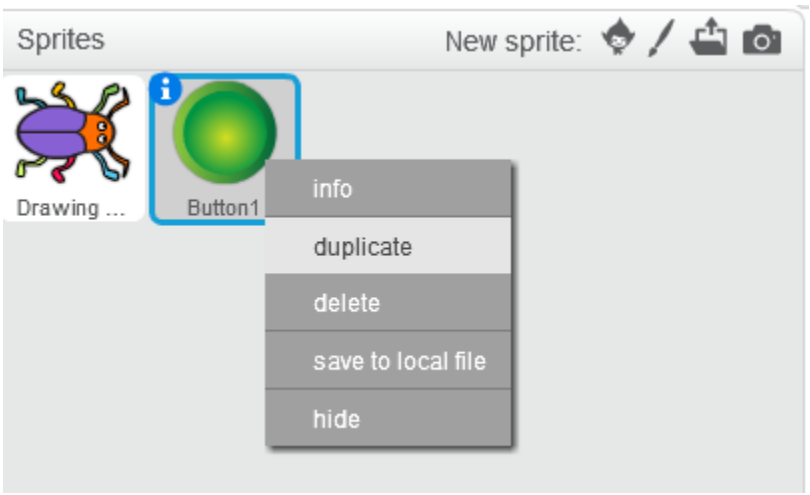


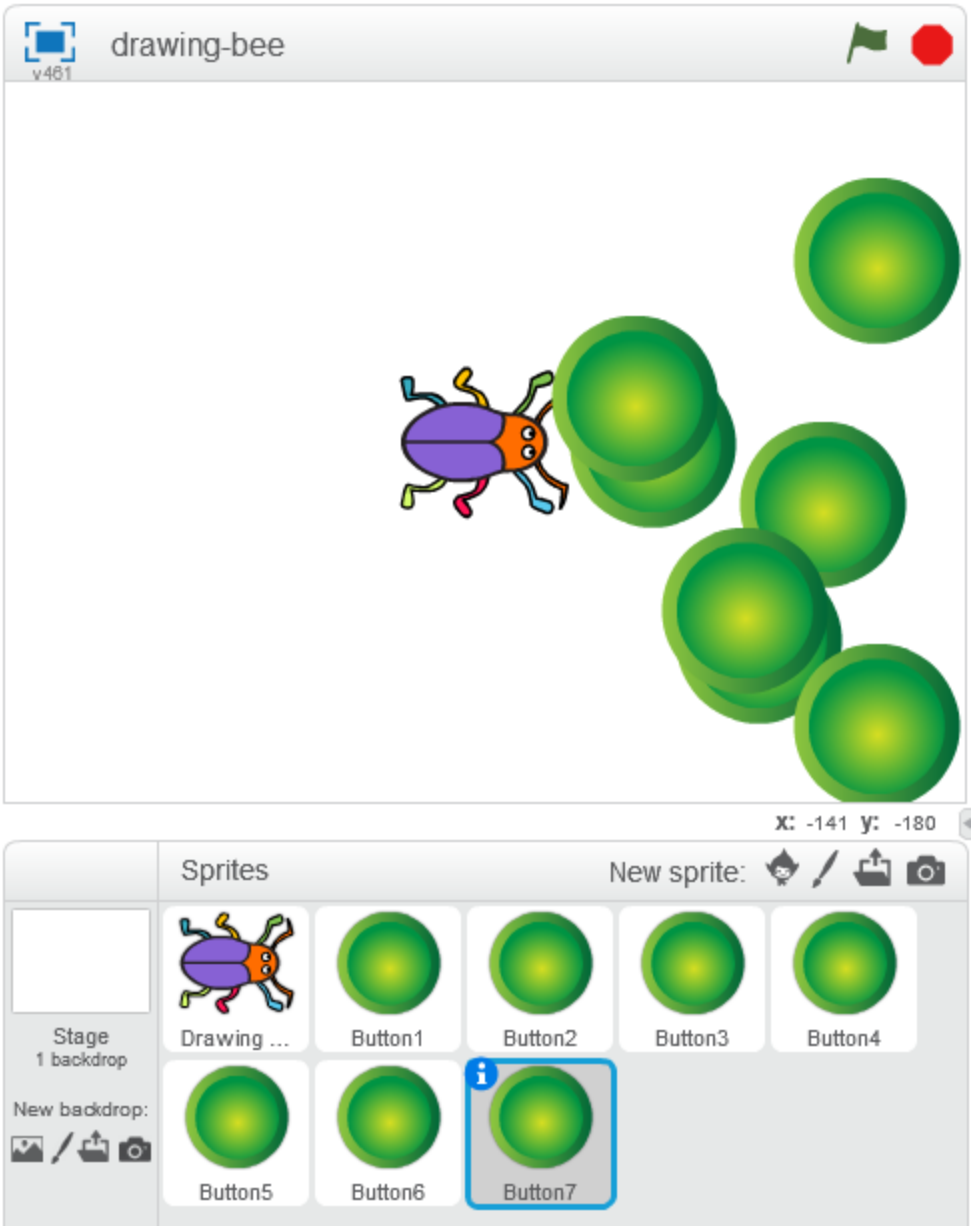
Next on our **Project Components** table are seven buttons. Add one round Button sprite from the Scratch Sprite Library.


Scratch 2 Offline Editor
Sprite Library
Category
All
Animals
Fantasy
Letters
People
Things
Transportation
Theme
Castle
City
Dance
Dress-Up
Flying
Holiday
Music
Space
Sports
Underwater
Walking
Type
All
Bitmap
Vector
Abby
Airplane
Amon
Anina Hip-Hop
Anna
Anna Ode to Code
Apple
Arrow1
Avery
Avery Walking
AZ Hip-Hop
Ball
Ball-Soccer
Ballerina
Balloon1
Bananas
Baseball
Basketball
Bass
Bat1
Bat2
Beachball
Bear1
Bear2
Beetle
Bell
Bells
Bowl
Bowtie
Breakdancer1
Buildings
Bus
Butterfly1
Butterfly2
Butterfly3
Button1
Button2
Button3
Button4
Button5
Cake
Calvrett
Candle
Candles1
Candles2
Car-Bug
Cassy
Cassy Dance
Cat1
Cat1 Flying
Cat2
Catherine Dance
Champ99
Cheesy-Puffs
Cloud
Clouds
CM Hip-Hop
Convertible1
Convertible2
Convertible3
Cowbell
Crab
Creature1
Cymbal
D-Money Hip-Hop
Dan
OK
Cancel



Since all seven buttons are the same, simply duplicate the button you just added six times.


Sprites
New sprite:
Drawing ...
Button1
info
duplicate
delete
save to local file
hide

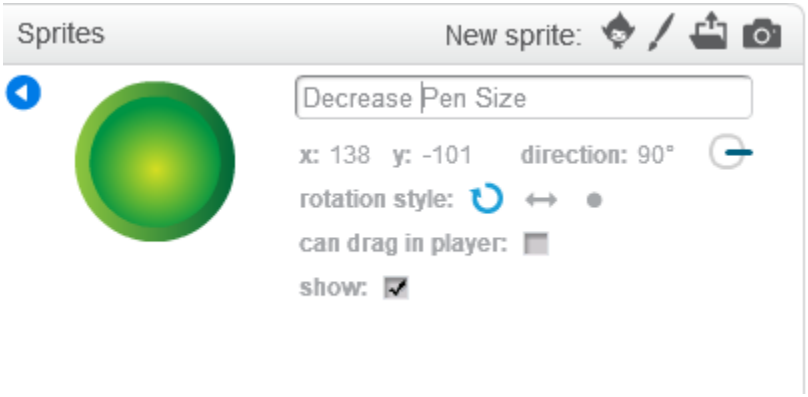


Costumes

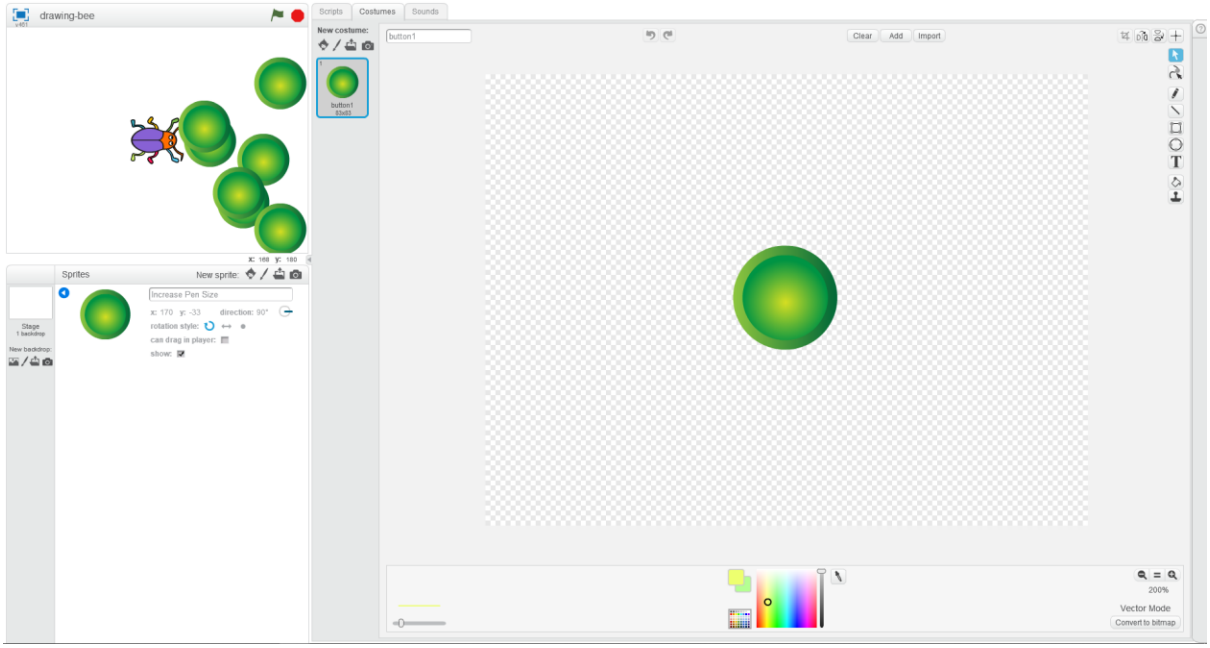
Now that we have all the sprites needed for the Drawing Bee application, we will now modify them to look just like they look in the final version with the Paint Editor. The *Drawing Bee* sprite is just fine as it currently is, so we will start from the first button.

From the Project Components table, the first button costume is “1 circle with a white background and black border with a “-” icon in it”.

The first thing to do is to rename the sprite so that we can easily identify its function when modifying its looks in the Paint Editor. Rename *Button1* to “Decrease Pen Size”.

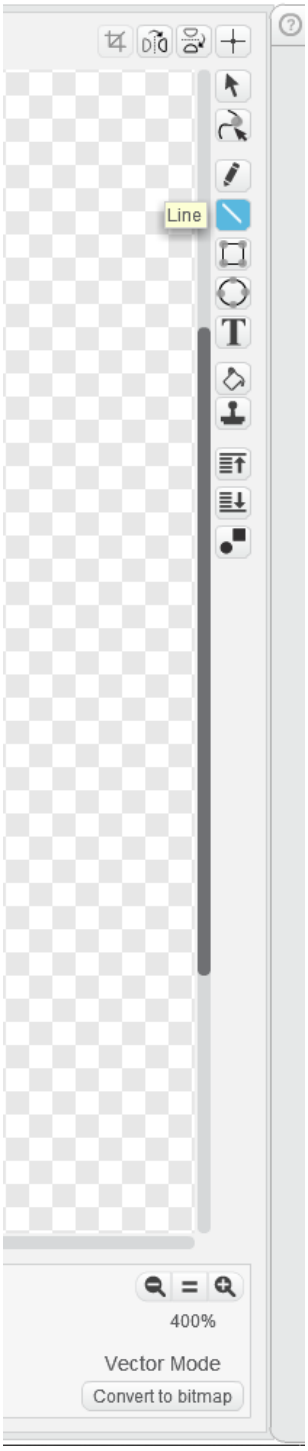


We already have a circle, but we need to make it smaller. Select the *Decrease Pen Size* sprite and switch to the Costumes Tab.

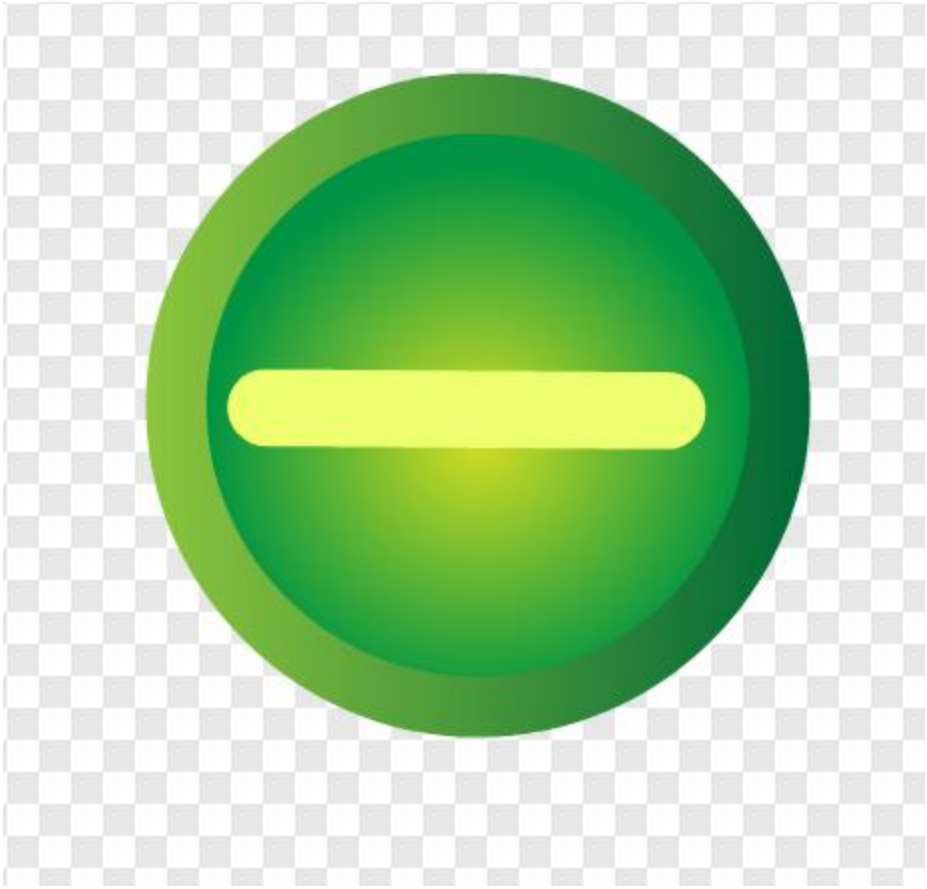


Note: Use the zoom buttons at the bottom right of the Paint Editor to enlarge the window. Editing a costume is easier that way.

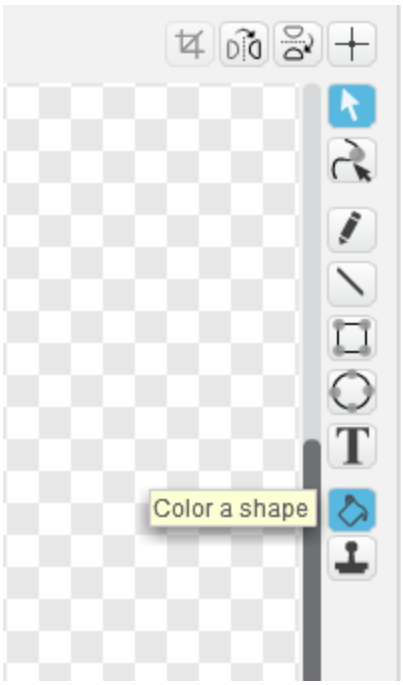
Let us start with the "-" icon in the button. We can draw this icon using the *line* tool of the Paint Editor. The tools are located at the right side of the Paint Editor.



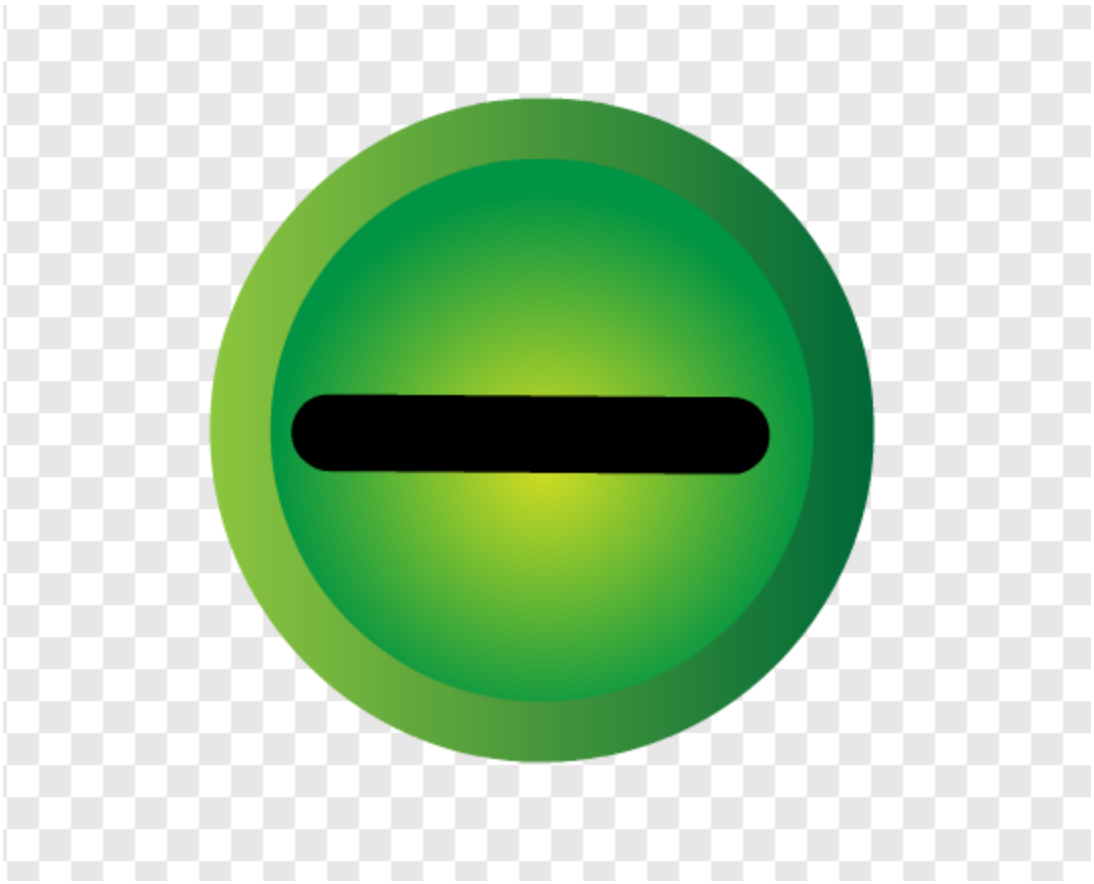
Choose the line tool and draw a line in the middle of the circle. Then choose the select tool, select the line you just drew and enlarge it with the controls that appear on it.



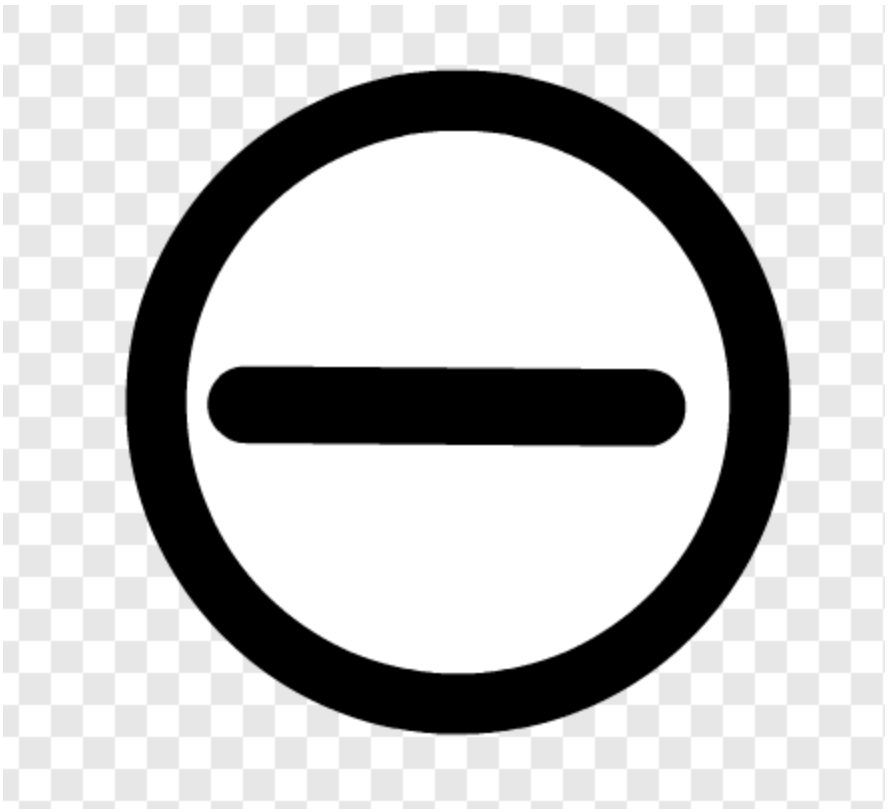
The “-” icon is black, so select the “color a shape” tool, select the black colour from the color palette at the bottom center of the Paint Editor, and click on the icon to change its colour to black.


Color a shape

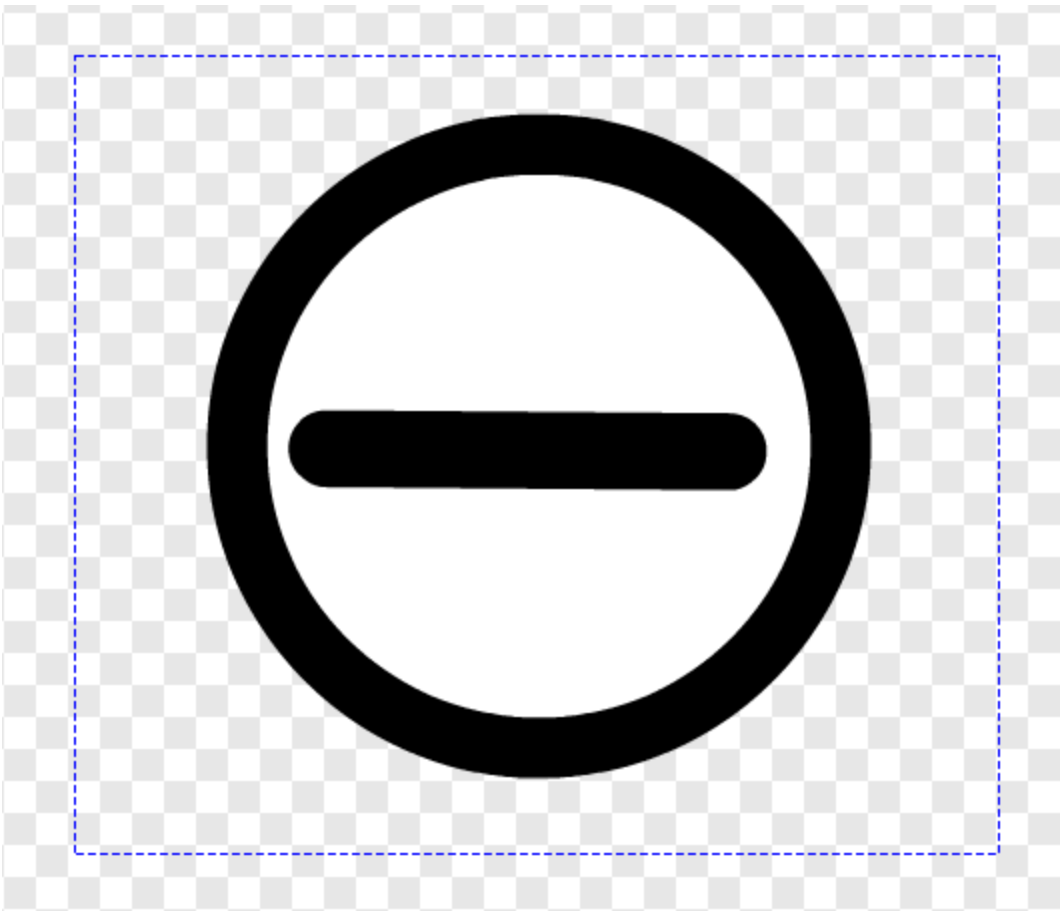

400%
Vector Mode
Convert to bitmap

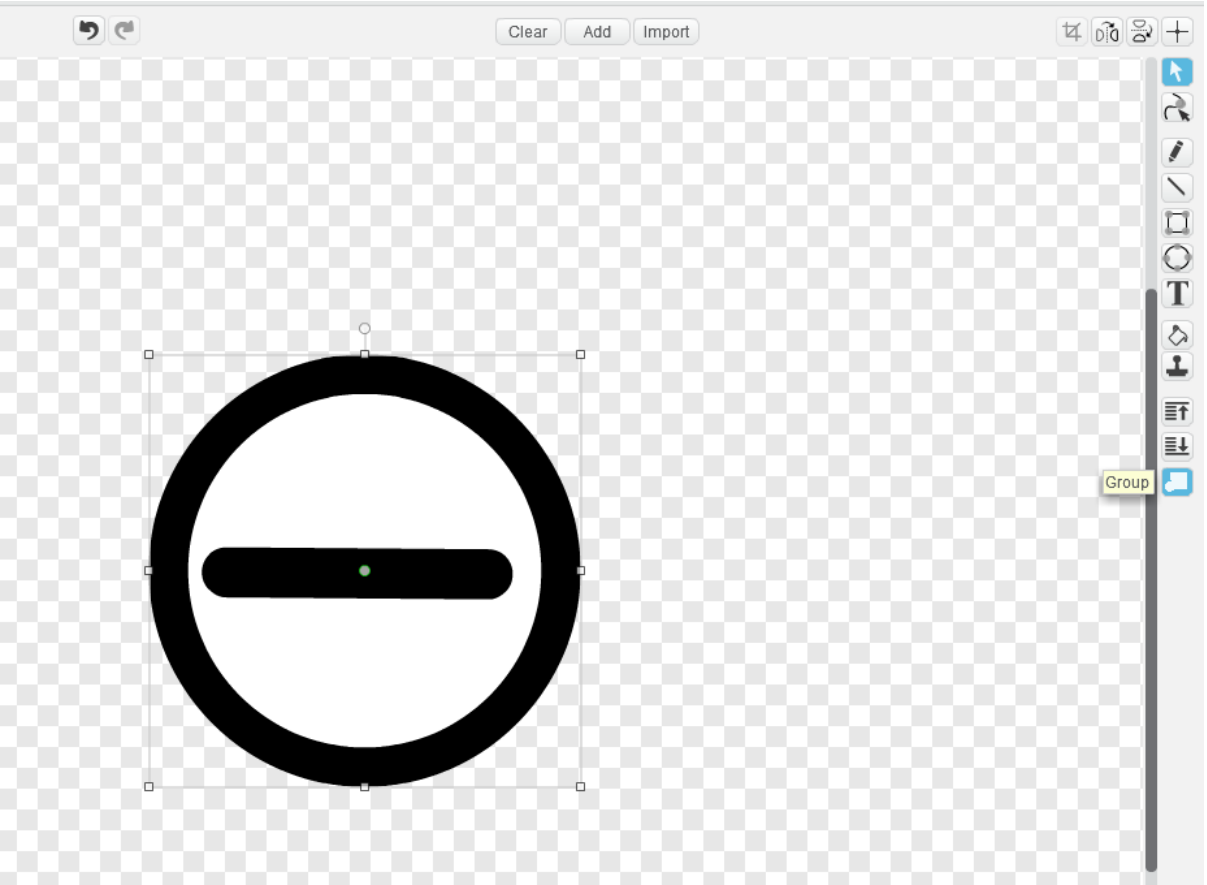


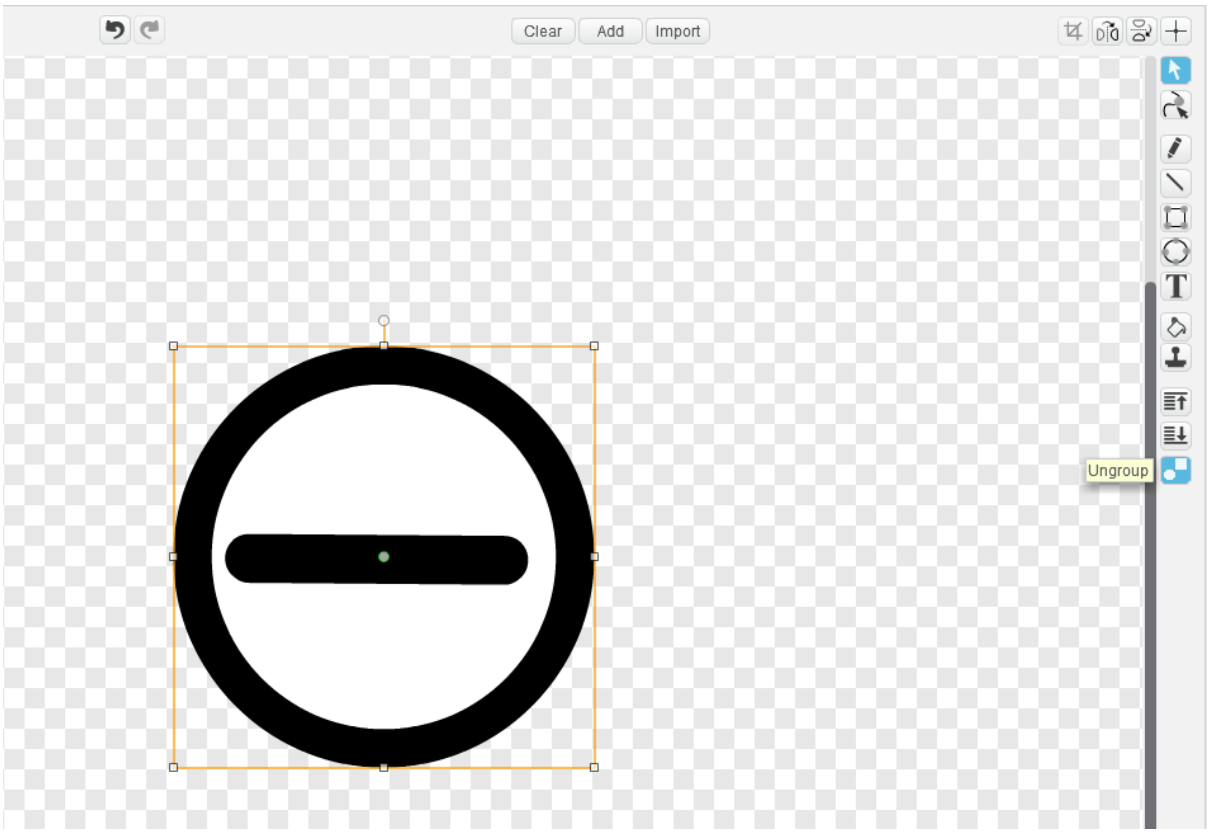
Use the same *color a shape* tool to change the circle's background to white, and the border to black.

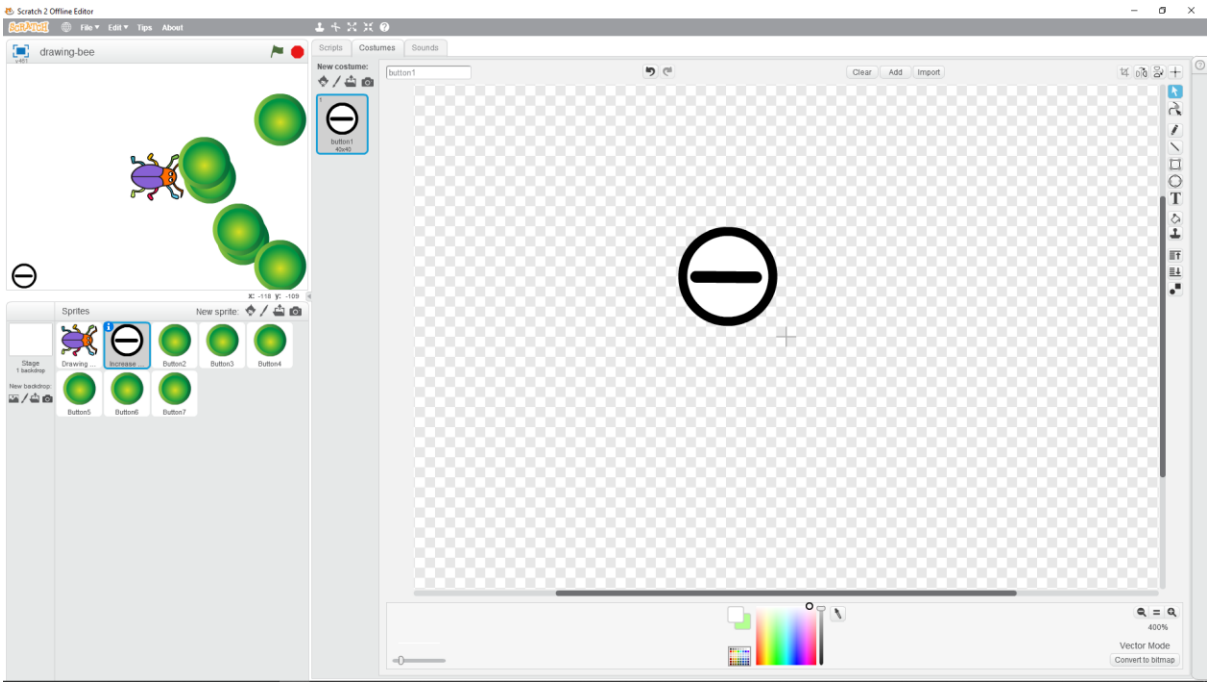


This now looks good, but it is bigger than what we have in the final version. Select all the shapes in the costume with the select tool, use the *Group* tool to group them all together, and then use the controls that appear on the grouped shape to reduce its size.



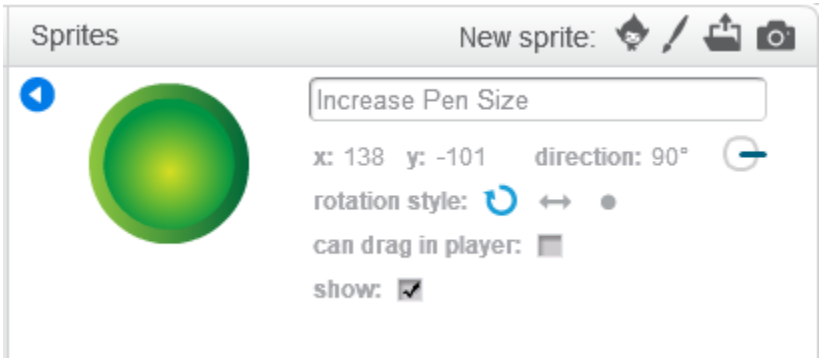

Clear
Add
Import
Group



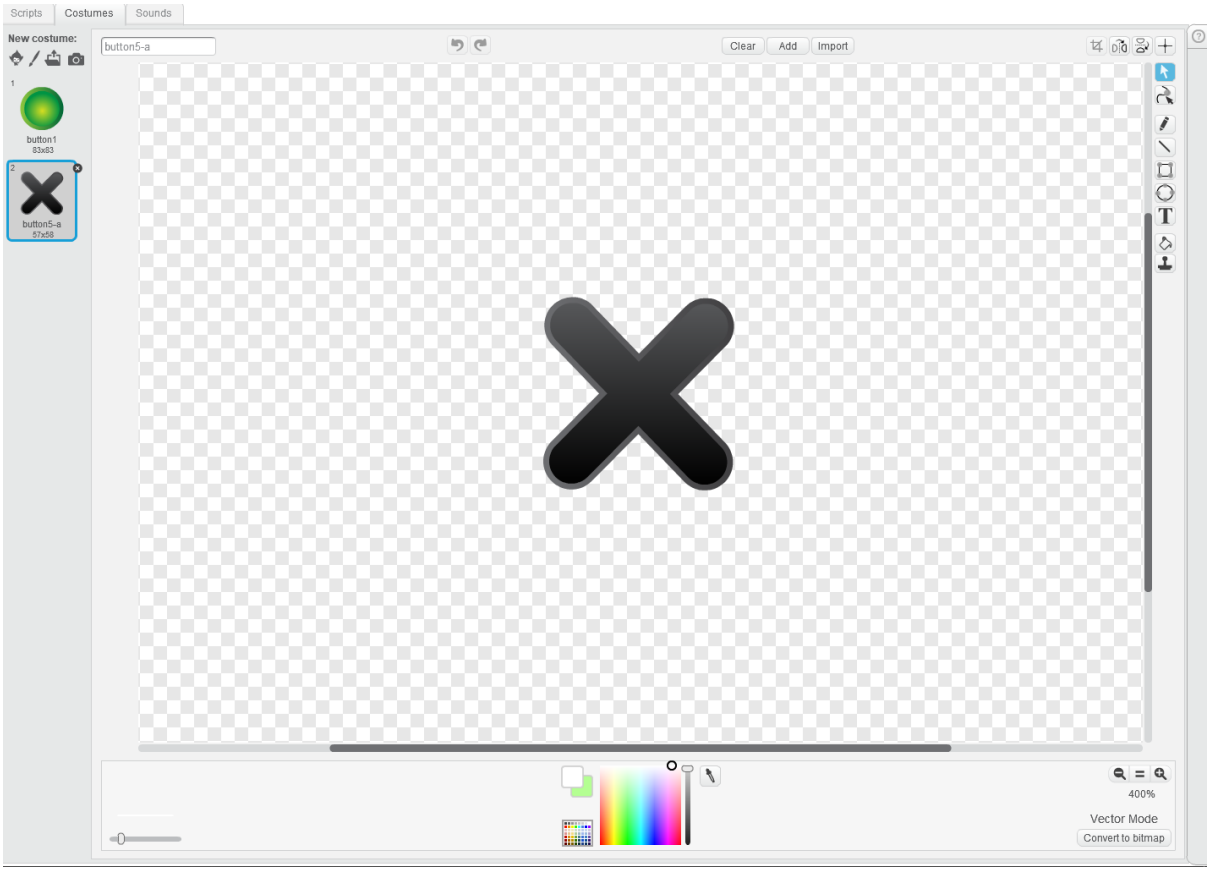


We now have the *Decrease Pen Size* button where it should be.

The next costume from the Project Components table is “1 circle with a white background and black border with a “+” icon in it”. Rename the *Button2* sprite to *Increase Pen Size*.

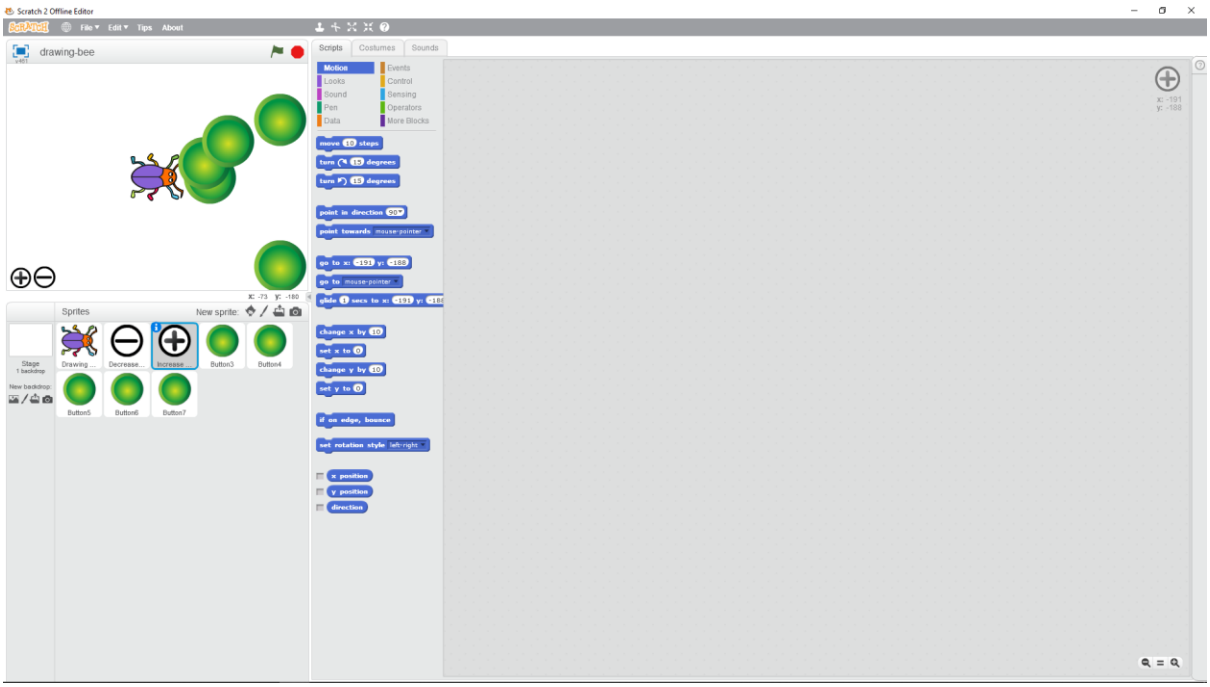


We can draw the “+” icon like we did for the first button, but there is an easier way. Add the *button5-a* costume from the Scratch Costume Library.



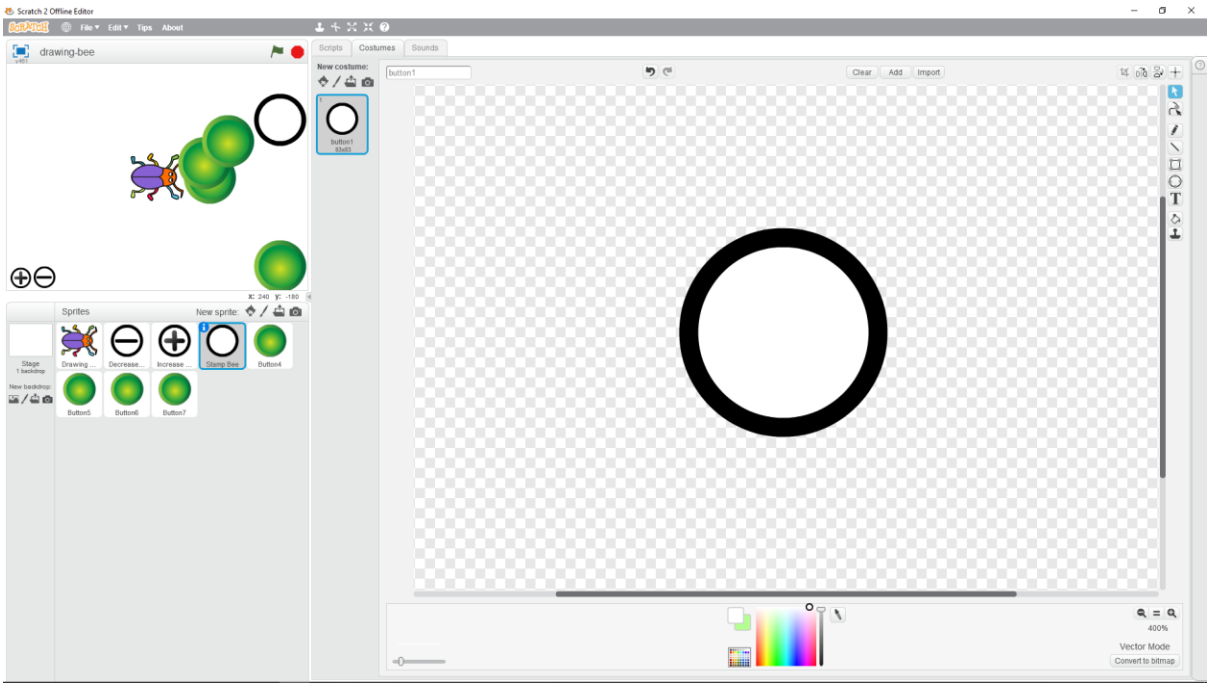
Now select the X image in the second costume and copy it, and then paste it in the circle in the first costume.

You can now use the controls that appear on the image to rotate it to look like a “+”, change the colours of the circle background and border, and group and resize all the images to look like the final version.

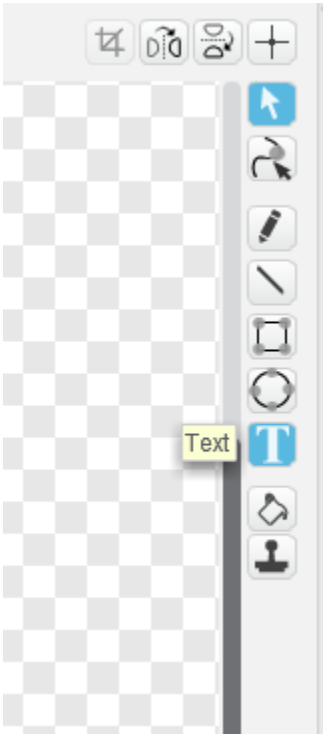


Two buttons down, five to go. 💪

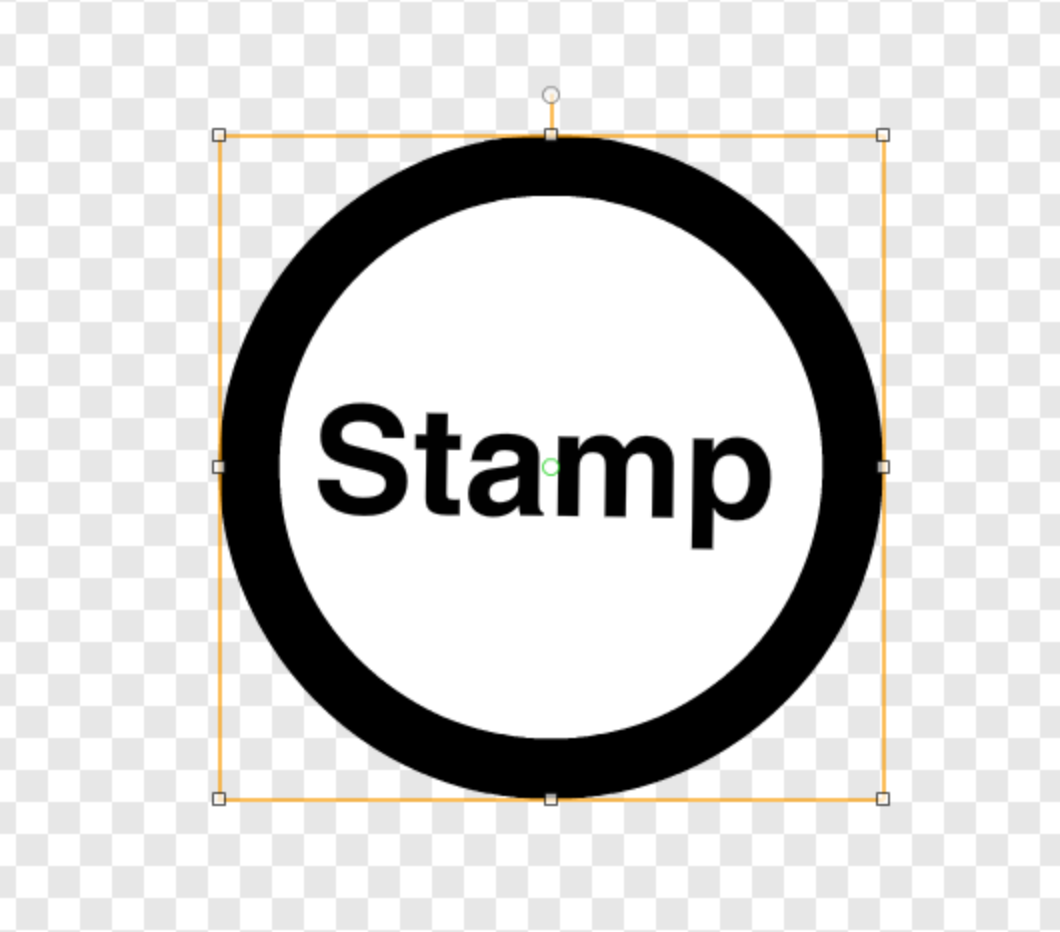
The next costume is “1 circle with a white background and black border with the text “Stamp” written in it”. Rename the button to “Stamp Bee” and change the circle’s background and border colours.



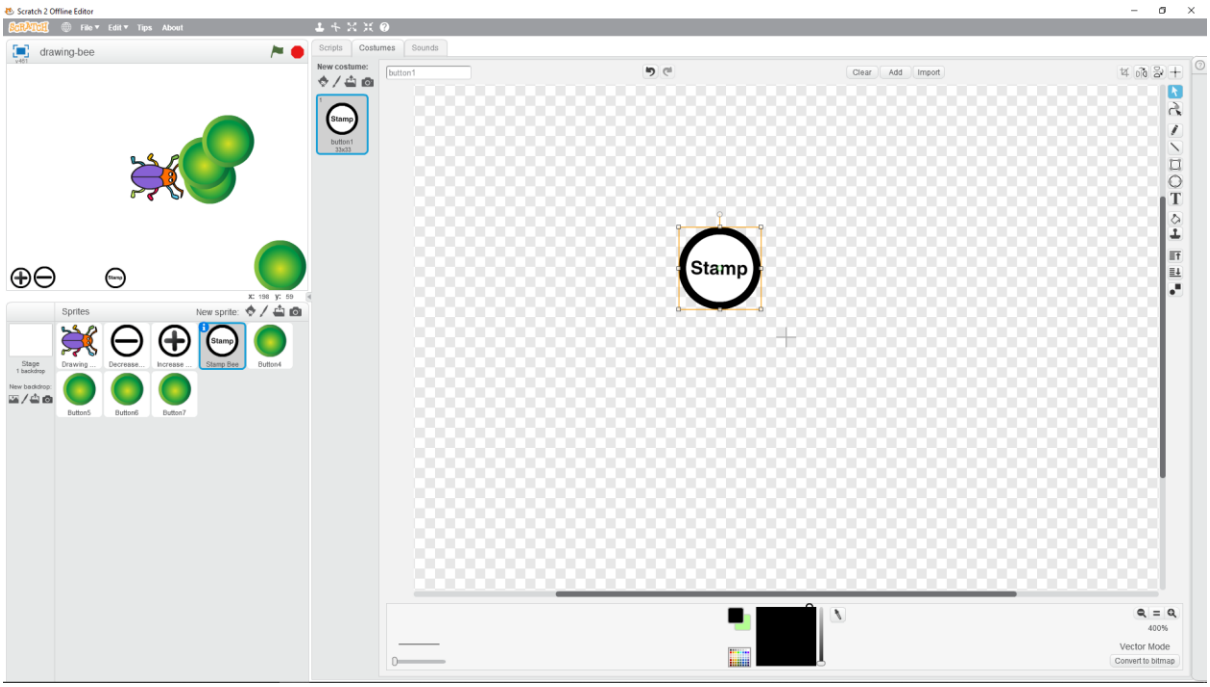
We can add text to a costume in the Paint Editor using the Text tool.



Add the text “Stamp to the circle, resize it and give it a black colour.

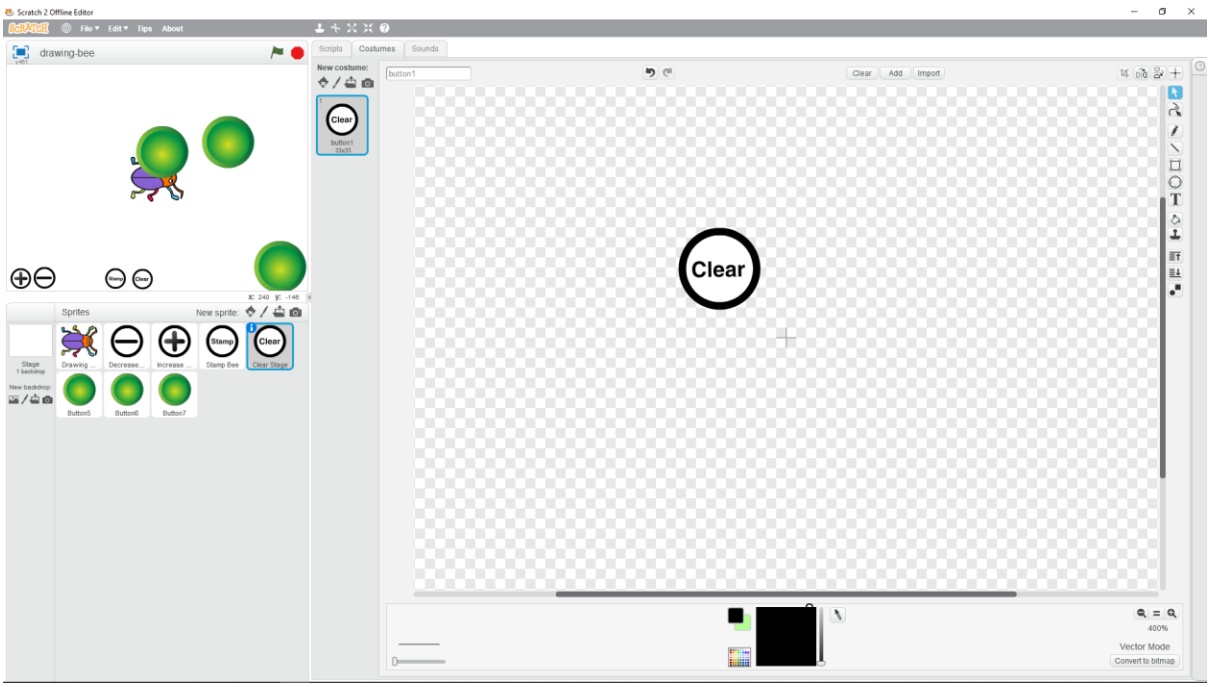


Now group both images together and resize the grouped image to fit the size in the final version of Drawing Bee.

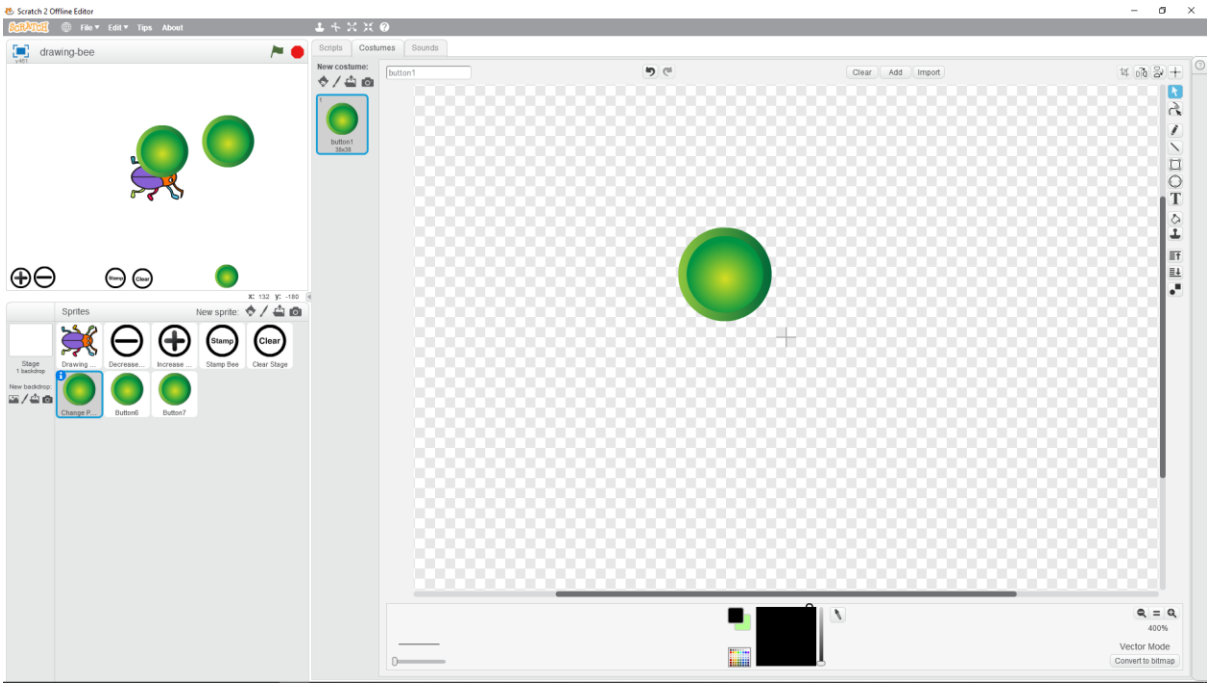


The next costume is “1 circle with a white background and black border with the text “Clear” written in it”. Rename this sprite to *Clear Stage* and edit it as we did above to get the desired result.

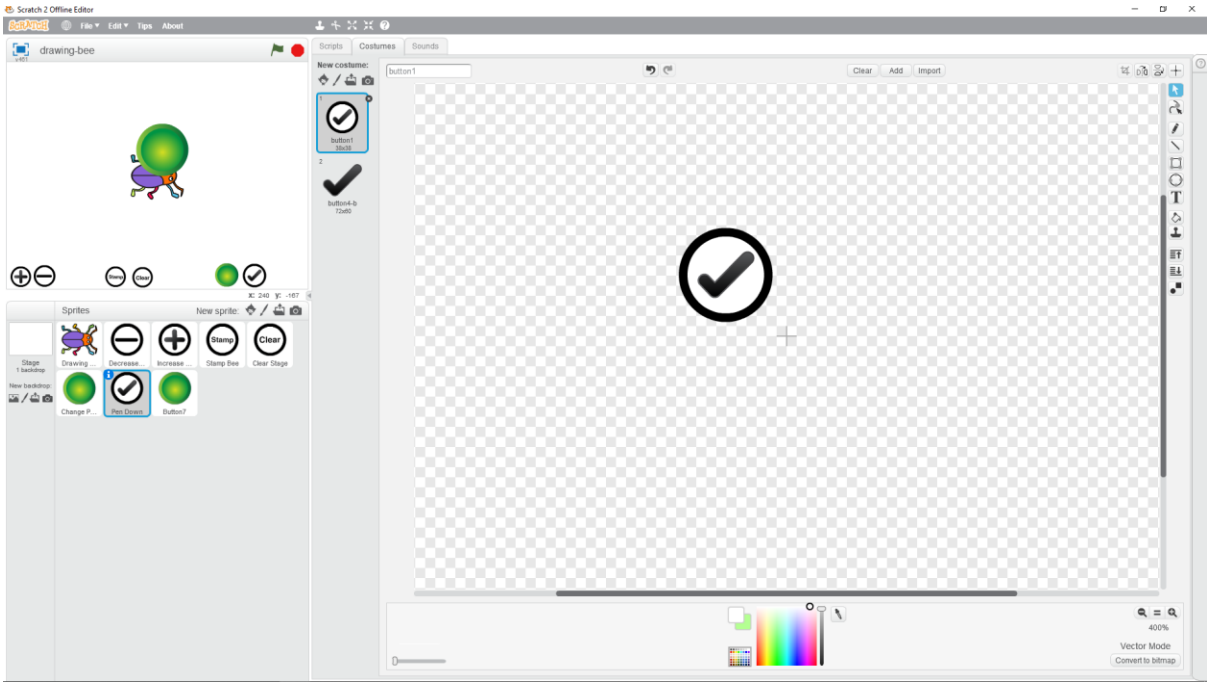

Clear


drawing-bee
Clear
Sprites
New sprite:
Clear
Vector Mode

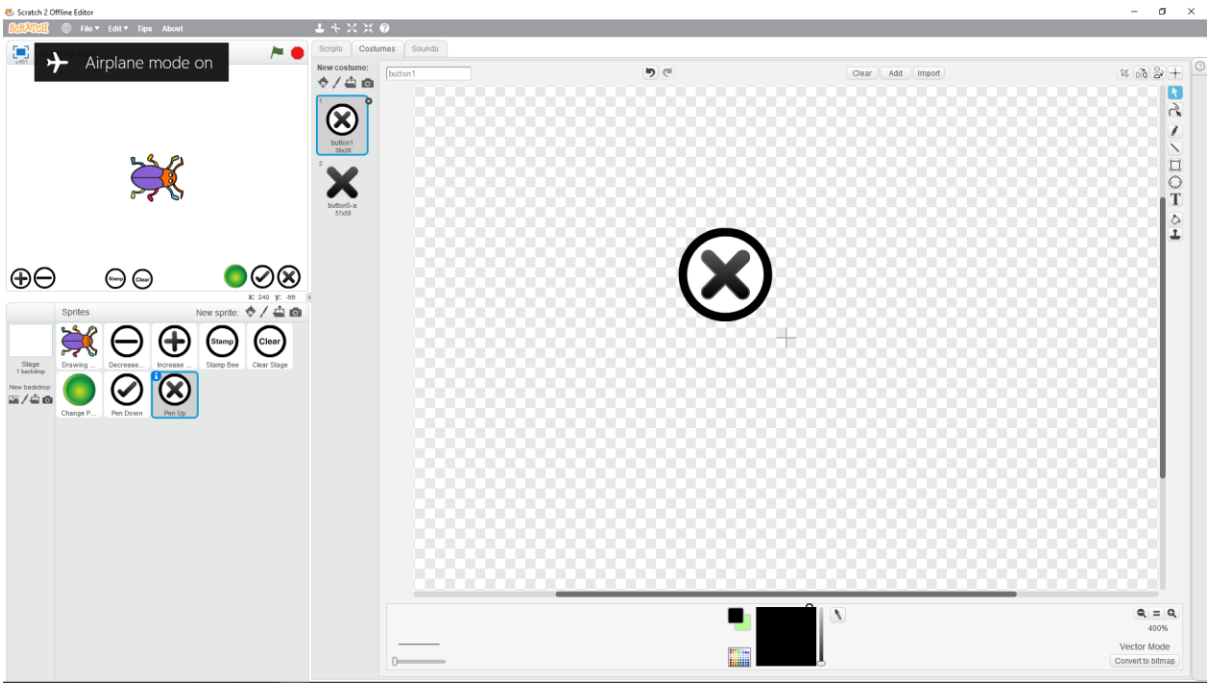
The next costume is “1 multi-coloured circle”. Rename the button to *Change Pen Colour* and switch to the Paint Editor. The button is already multi-coloured, so all you need to do is to resize it and place it at an appropriate place on the Stage.

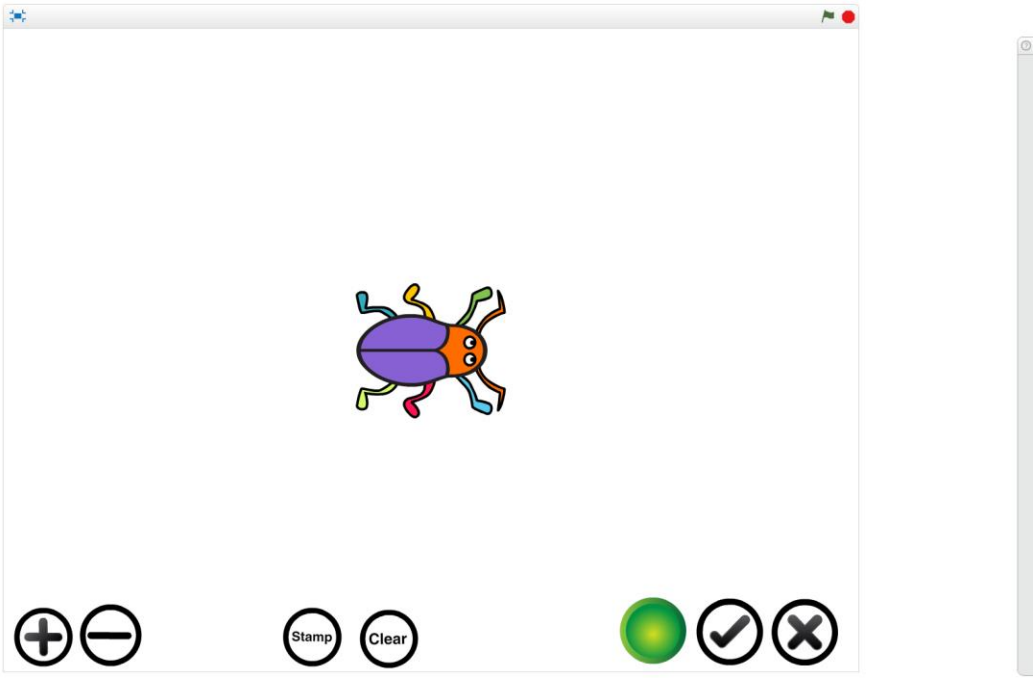


The next costume is “1 circle with a white background and black border with a “✓” icon in it”. Rename it to *Pen Down* and make it like we did for the Increase Pen Size button, using the *button-4-b* costume from the Scratch Costume Library.



The next costume is “1 circle with a white background and black border with a “✕” icon in it”. Rename it to *Pen Up* and make it like we did for the *Pen Down* button, using the *button-5-a* costume from the Scratch Costume Library.





We are set up! Now to the last step in our process for building applications.

Program the components (with blocks and scripts) to perform their functions

Here are the functions we derived from the first step of the process:

The Drawing Bee:

- Moves when the keyboard's arrow keys are pressed.
- Moves when the stage is clicked.
- Draws lines on the stage when its pen is put down with a button.
- Stamps itself on the stage when the *stamp* button is clicked.
- Changes its pen colour when the multi-coloured button is clicked.
- Changes its pen size when the + and - buttons are clicked.
- Clears the stage when the *clear* button is clicked.

We will tackle these functions one after the other.

1. Drawing Bee moves when the keyboard's arrow keys are pressed.

This can be further divided into two parts:

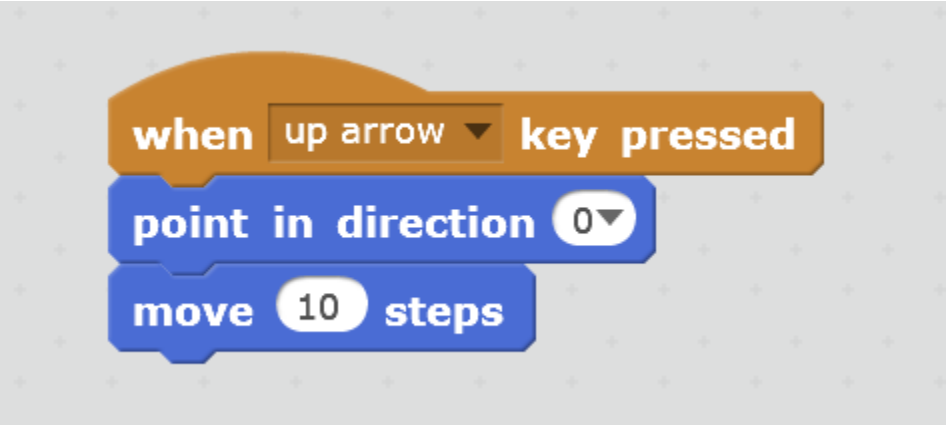
- Drawing Bee moves - this is a **Motion** action.
- When the keyboard's arrow keys are pressed - this is an **event** trigger.

We have learnt that trigger blocks are used to start scripts, so we will add the event trigger(s) first. Select the Drawing Bee sprite and switch to the Events block category and let us see which block can help us with this event.

The second block in the palette says “when space key pressed”. The space key is a keyboard key, but we want the arrow keys instead. Drag the block to the Scripts Area and check the input options to see if there is something about the arrow keys.

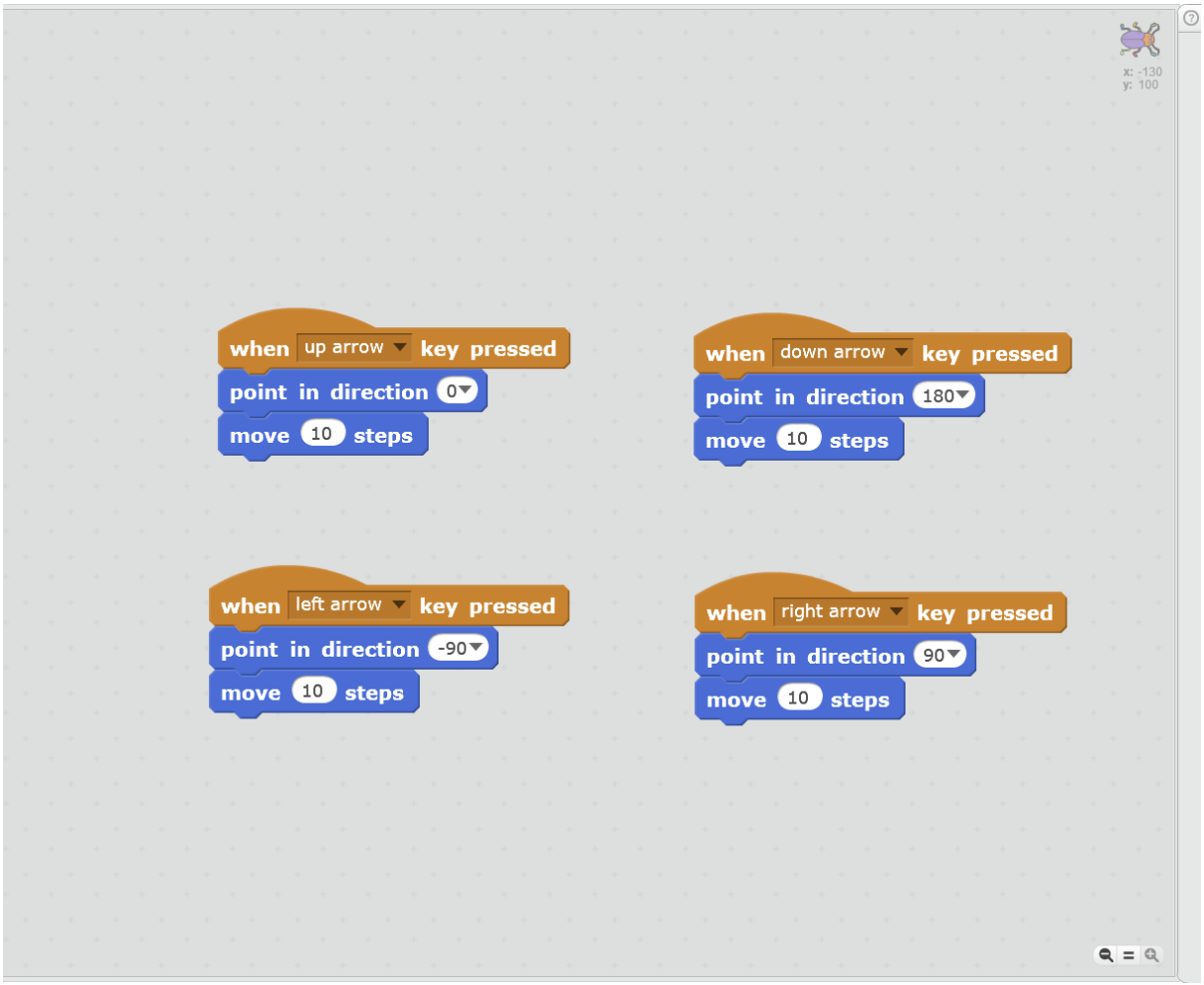
We have options for the up, down, right and left arrow keys, as well as for all letters of the alphabet and numbers 0-9. Select the up arrow option.

For the first part of the statement, switch to the Motion block category. When the up arrow key is pressed, we want to sprite to change its direction to 0° (up), and then move some steps in that direction. Which Motion blocks can be used to perform these actions?



Test the script: click on it and press the up arrow on your keyboard repeatedly. Does the bee move up?

Duplicate this script for the down, left and right arrows. Remember to change the direction for each script.

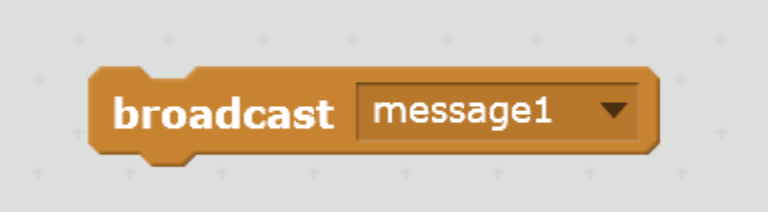


Let us move onto the next function.

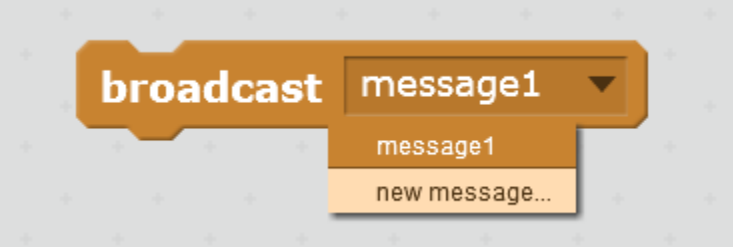
2. Drawing Bee moves when the stage is clicked.
We want the bee to move to any point on the stage that is clicked with the computer mouse; in other words, the bee should move to the mouse's x and y coordinates (which are equal to the x and y coordinates of the point on the stage that was clicked) when the stage is clicked. The stage has an event block that detects when it is clicked.

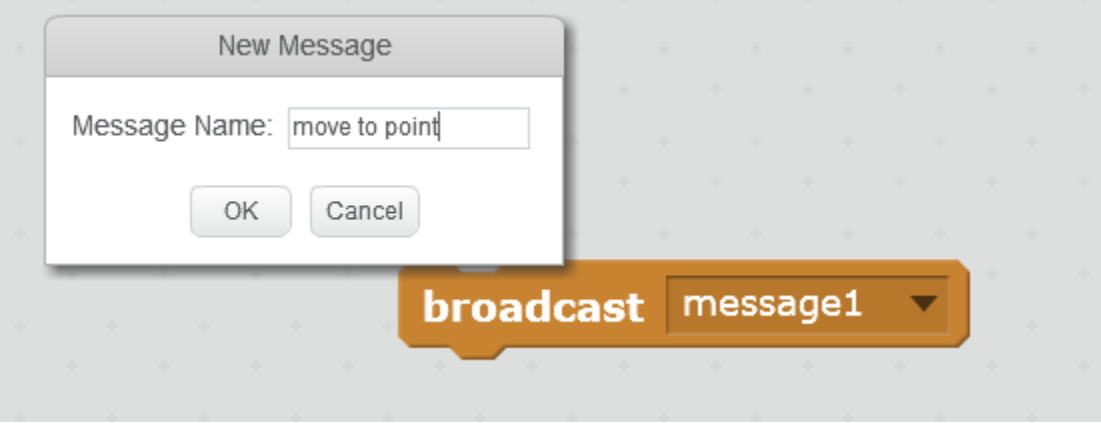
Remember, however, that we can only program blocks and scripts for the currently selected component, and hence, we cannot directly program the bee to move to a point when the stage is clicked from the stage's scripts. We need to find a way for the stage to inform the bee that it has been clicked and it should perform the motion action.

The Events block category has a "broadcast message" block that can be used to send a message out from a component to other components in an app.



We now know the importance of naming things for easy identification, so create a new message input option and name it "move to point".

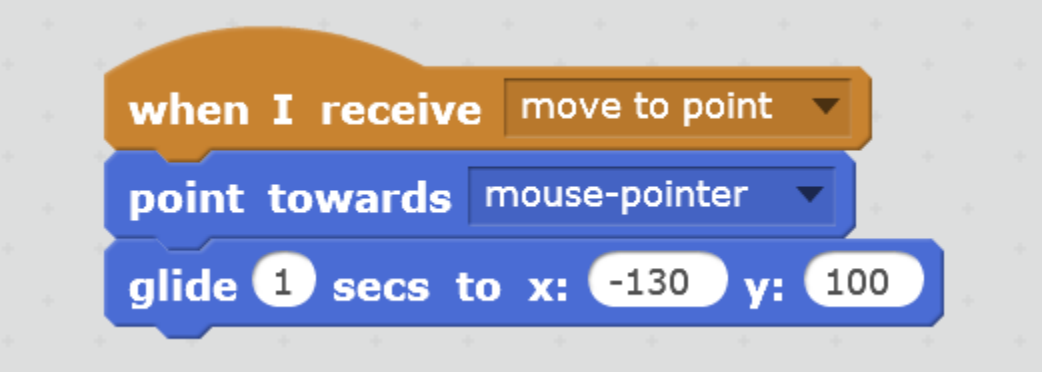




Snap this block to the event block.

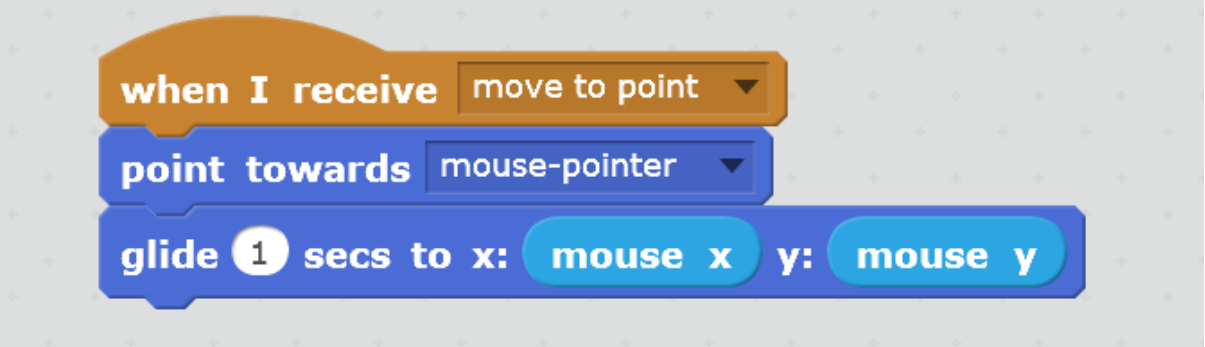
Now select the Drawing Bee sprite and use the *when I receive message* event block to detect when the stage broadcasts the "move to point" message when it (the Stage) is clicked.

We can then use motion blocks to move the bee to the point that was clicked on the stage using the mouse's x and y coordinates.



The first motion block points the bee in the direction of the mouse, and the second makes it glide to the specified coordinates. However, we do not want the bee to glide to specific coordinates; instead, we want it to glide to the coordinates of the mouse. These can be found as reporter blocks in the *Sensing* block category.


Scripts
Costumes
Sounds
Motion
Looks
Sound
Pen
Data
Events
Control
Sensing
Operators
More Blocks
touching mouse-pointer ?
touching color ?
color is touching ?
distance to mouse-pointer
ask What's your name? and wait
answer
key space pressed?
mouse down?
mouse x
mouse y
loudness
video motion on this sprite
turn video on
set video transparency to 50 %
timer
reset timer
x position of Change Pen Colour
current minute
days since 2000
username



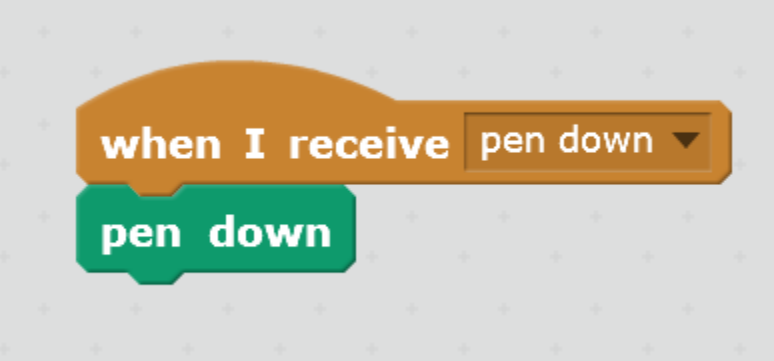
Test the script: click on it to execute it, and then click on any point on the stage. Does the bee move to the point clicked?

3. Drawing Bee draws lines on the stage when its pen is put down with a button.

The button that has the ✓ icon handles the action of putting the bee's pen down when it is clicked. Select the Pen Down sprite in the Sprite List and switch to the Events block category.

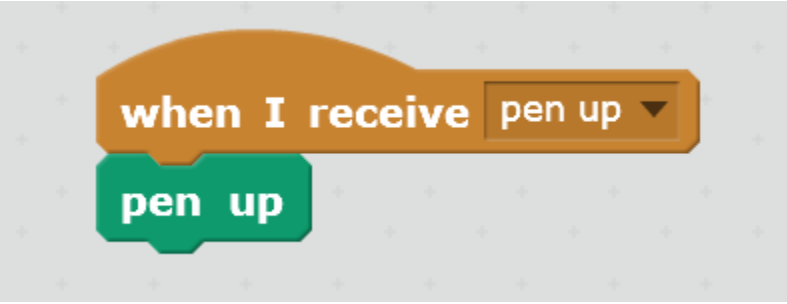
We now know that the only way to program a component based on an event triggered on another component is through broadcast messages. Create a broadcast message called "pen down" and send it when the *Pen* Down sprite is clicked.

Now select the Drawing Bee sprite and handle the broadcast event.



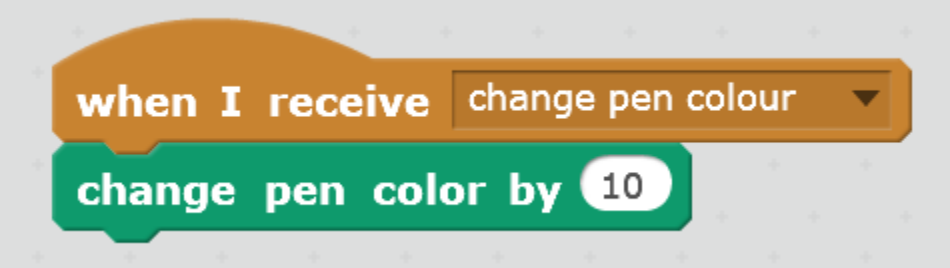
Test it out: execute this script and move around the stage with your keyboard arrow keys or the mouse. Does the bee draw on the stage as it moves around?

Since we now have a button that puts the pen down, we should program the *Pen Up* button to reverse this action. Follow the steps we took above and replicate this action for the *Pen Up* sprite.

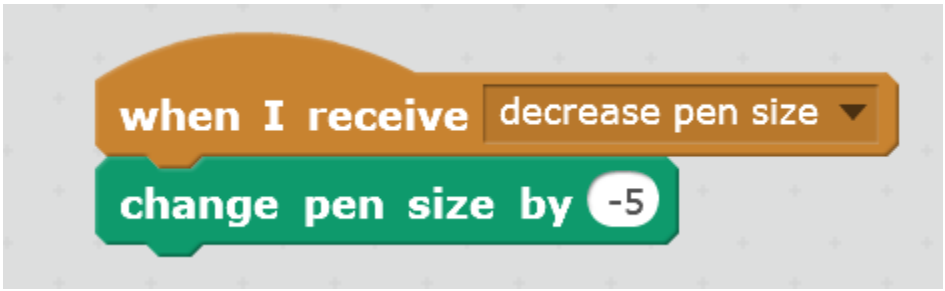


4. Drawing Bee stamps itself on the stage when the *stamp* button is clicked.
Follow the steps we used in number 3 above to program this function for the *Stamp Bee* sprite.

5. Drawing Bee changes its pen colour when the multi-coloured button is clicked.
Follow the steps we used in number 3 above to program this function for the *Change Pen Colour* sprite.



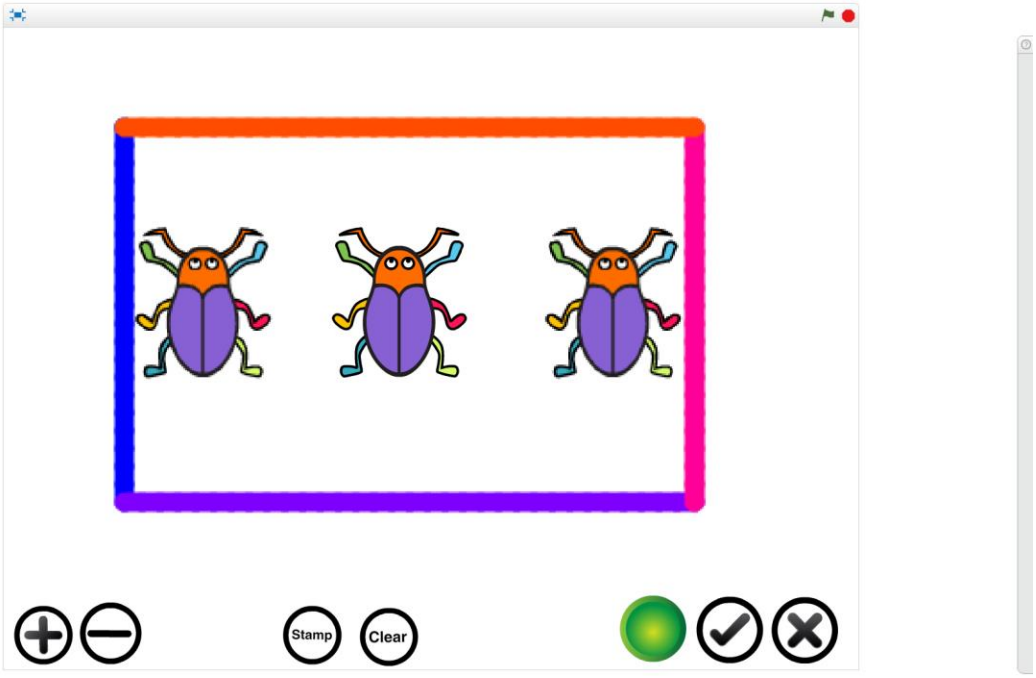
6. Drawing Bee changes its pen size when the + and - buttons are clicked.
Follow the steps we used in number 3 above to program this function for the *Increase Pen Size* and *Decrease Pen Size* sprites.



7. Drawing Bee clears the stage when the *clear* button is clicked.
We do not need to broadcast a message from the *Clear Stage* sprite for this action, as any sprite can execute the *clear* block to clear the stage at any time.

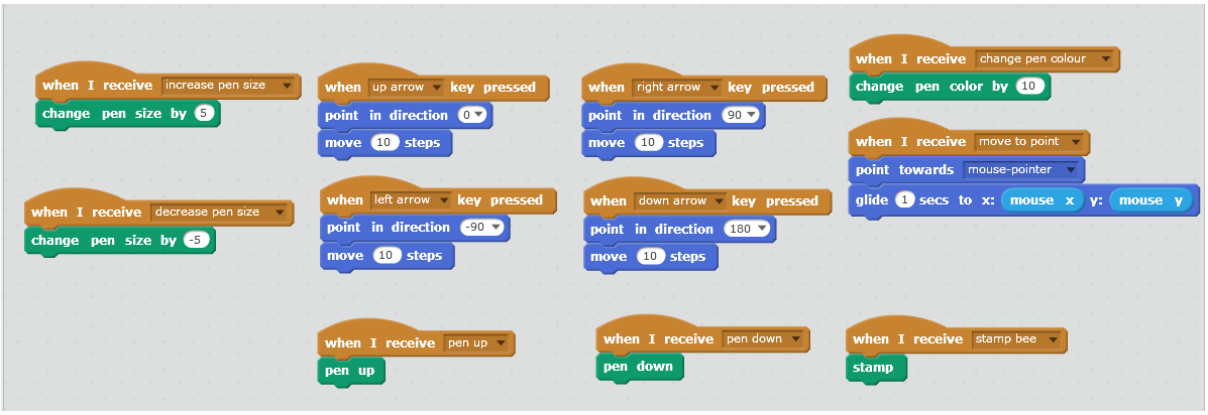
If you added all the blocks for the application functions as listed above, you would now have a complete Drawing Bee application.

Switch the stage to fullscreen mode and try to make the following drawing.

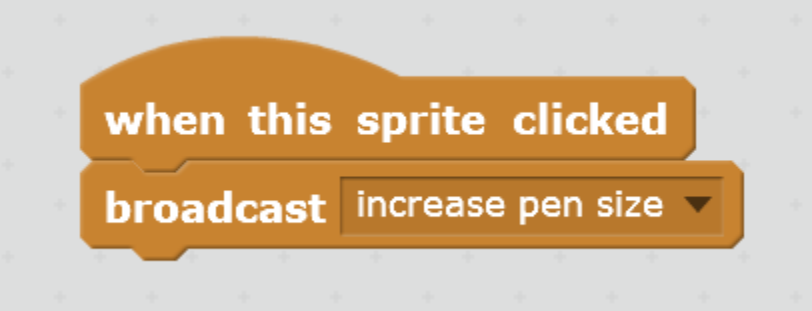


Project Blocks & Scripts

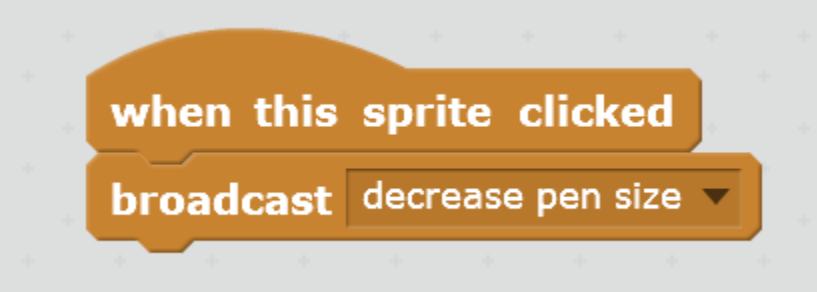
Drawing Bee sprite



Increase Pen Size sprite



Decrease Pen Size sprite



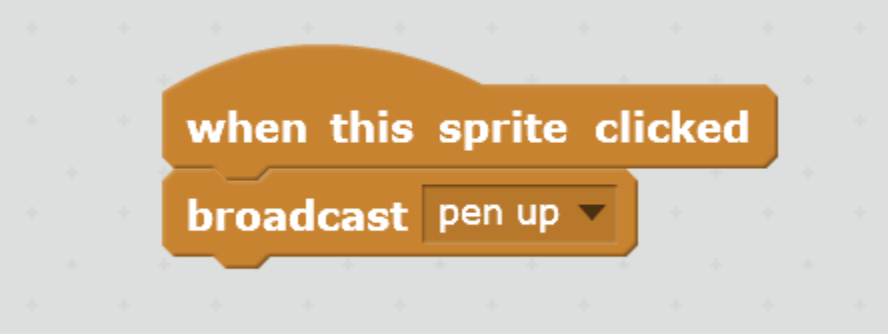
Stamp Bee sprite

Clear Stage sprite

Change Pen Colour sprite

Pen Down sprite

Pen Up sprite



The Stage

Summary of Module Five

The following are the key points covered in this module:

- Sprite costumes can be modified with different tools from the Paint Editor to achieve a different look. Images can also be copied across costumes to emulate a particular look.
- The various tools that can be used to modify costumes and backdrops with the Paint Editor are located on the right side of the Paint Editor window.
- The *line* tool of the Paint Editor is used to draw lines.
- The *select* tool is used to select images to perform operations like enlargement/reduction, rotation and grouping on them.
- The *color a shape* tool is used to paint the different parts of an image.
- The *group* tool is used to combine different images into a single entity.
- The *set costume center* tool is used to specify the center point of a costume, which determines how the costume is placed on and changes direction on the stage.
- The *Text* tool is used to write text.
- The *when ? key pressed* block is used to perform actions when a key on the keyboard is pressed.
- The *when Stage clicked* block is used to react to a click on any point on the stage.
- Blocks and scripts for a component can only be programmed when that component is selected. To program a component's blocks/scripts based on another component's blocks/scripts, use the *broadcast message* and *when I receive broadcast message* blocks.
- The Sensing block category contains reporter blocks that report the values of the computer mouse's x and y components on the stage at any time.

MODULE 6

Programming Apps with Looks

Table of Contents

        - The set effect to n block
        - The clear graphic effects block
    - Size
        - The change size by n block
        - The set size to n% block
    - Sprite Layers
        - The go to front block
        - The go back n layers block
- Summary of Module Six

Learning Objectives

By the end of this module, the student would:

- Understand how the different blocks from the Looks category can affect both internal and external looks in app components.
- Understand how the different graphic effects influence the looks of app components.
- Understand how layers are formed by overlapping sprites.

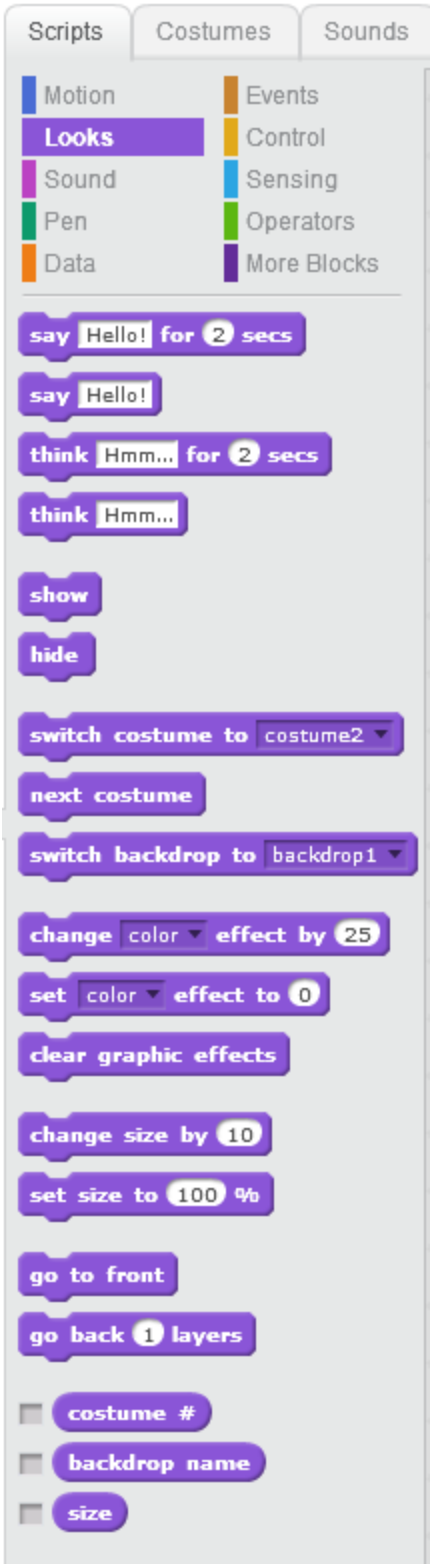
Module Prerequisites

- Scratch 2 Offline Editor set up on the computer with access to local storage.

In the last module, we created an application that can be used to move a bee sprite around, with motion blocks, and draw patterns on the stage with pen blocks. The application can be made even more user-friendly and exciting to use by playing certain sound effects when some actions occur, and giving the user the option to hide and show the bee sprite at will, while still being able to draw on the stage. What about being able to change the costume of the drawing bee to another animal entirely?

In this module, you will learn how to use blocks from the *Looks* category to program sprites to perform various operations related to a component's looks, both inherent (costumes & backdrops) and in relation to other components. After this, you will learn how to use blocks from the *Sound* category to program components to play different types of audio in the next module.

The *Looks* Blocks Category



There are different types of properties related to a component's looks in Scratch. Most of these properties are available to only sprites, and not the Stage.

Speech and Thought

Most games have sprites that act like characters and can think and speak to one another. This speech and thought behaviour can be emulated in Scratch applications with some blocks in the *Looks* category. When a sprite is programmed to think or speak some text, the text appears in a thought or speech bubble above the sprite on the stage. If you want a sprite to actually speak some words with a voice, you would have to program the sprite to play an audio file containing the spoken words. You will see how to do this in the next module when we talk about the *Sound* blocks category.

The *say* block

This block is used to program

a sprite to speak some text. The text to be spoken is passed as an input parameter to the block by typing into the white box on it. When this block is executed, the input text appears in a speech bubble above the sprite.

Once this block is executed and the speech bubble appears, it stays above the sprite indefinitely. To remove the speech bubble, delete all text input to the block and execute it.

The *say for n secs* block

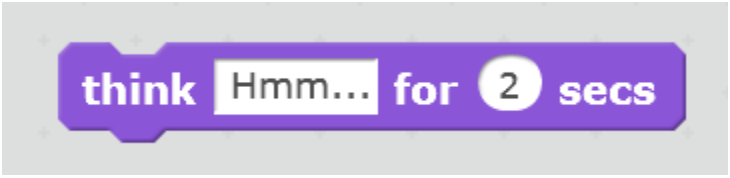
This block is used to program a sprite to speak some text for a specified period of time (seconds), *n*. When this block is executed, the speech bubble appears above the sprite for *n* seconds, after which it is removed.

The *think* block

This block is used to program a sprite to think about some text. The text to be thought about is passed as an input parameter to the block by typing into the white box on it. When this block is executed, the input text appears in a thought bubble above the sprite.

Once this block is executed and the thought bubble appears, it stays above the sprite indefinitely. To remove the thought bubble, delete all text input to the block and execute it.

The *think for n secs* block



This block is used to program a sprite to think about some text for a specified period of time (seconds), *n*. When this block is executed, the thought bubble appears above the sprite for *n* seconds, after which it is removed.

The stage is not capable of speaking or thinking, and therefore does not have speech or thought blocks.

Visibility

We learned, in the second module, about a sprite's *show* property, which is used to control whether the sprite is visible on the stage or not. The *Looks* category has blocks that can be used to modify this property programmatically.

The *show/hide* blocks

The *show* block enables a sprite's *show* property, which causes the sprite to be visible on the stage, while the *hide* block disables the *show* property, causing the sprite to be invisible on the stage.

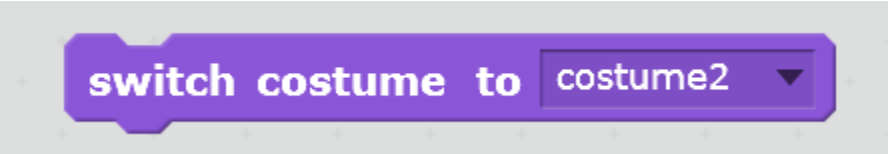
The stage does not have a *show* property and therefore does not have visibility blocks.

Costumes & Backdrops

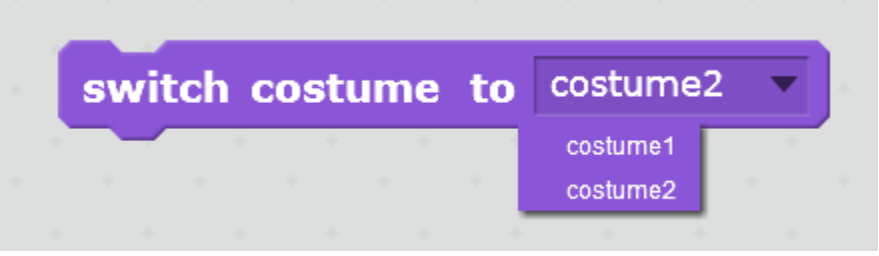
We know that a sprite can have more than one costume, and the stage can have more than one backdrop. In the applications we have created so far in this textbook, we always select the costume or backdrop to be shown before running the application. This is possible because we are the app developers, but when we eventually release an application for use by others, it is not possible for them to choose the costume or backdrop to be shown for a sprite or the stage before running the application.

The *Looks* category has blocks that can be used to programmatically change the costume or backdrop being displayed by a sprite or the stage.

The *switch costume to* block

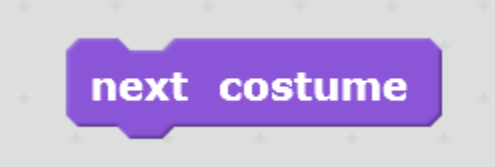


This block is used to change the displayed costume for a sprite. The names of all the costumes available in the sprite are shown in the input options, and the costume whose name is selected as the input parameter when this block is executed will be set as the sprite's displayed costume.



Since only sprites can have costumes, this block is available only for sprites.

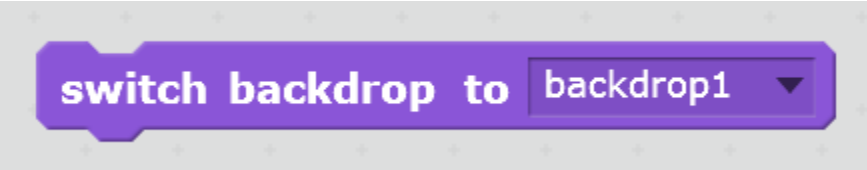
The *next costume* block



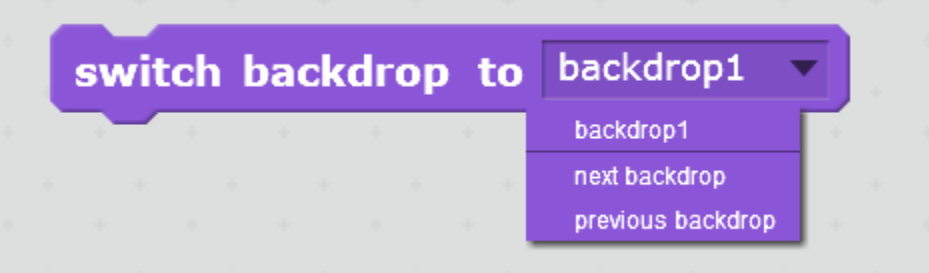
This block is used to change the costume displayed for a sprite to the costume next to the current costume in the Costumes Tab. If the displayed costume is the last one on the list and this block is executed, the sprite's costume is changed to the first one on the list, and the cycle continues.

Since only sprites can have costumes, this block is available only for sprites.

The *switch backdrop to* block



This block is used to change the displayed backdrop for the stage. The names of all the backdrops available in the stage are shown in the input options, and the backdrop whose name is selected as the input parameter when this block is executed will be set as the stage's displayed backdrop.



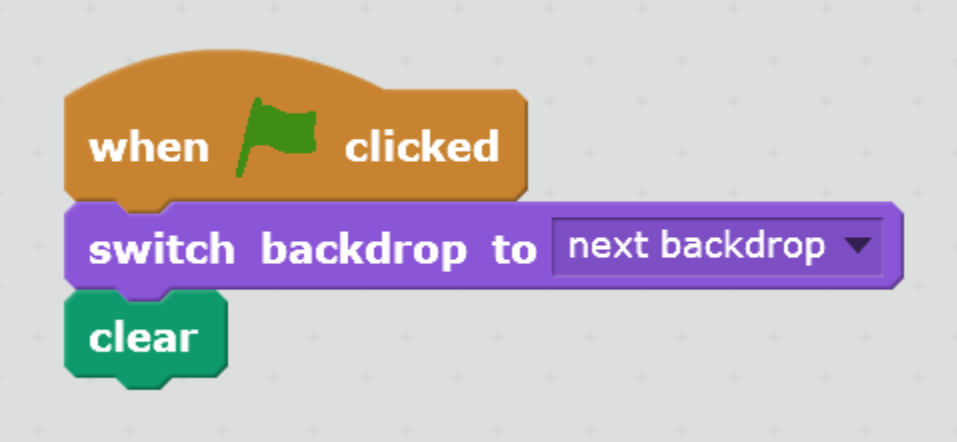
The *next backdrop* and *previous backdrop* options can also be selected to switch the stage's displayed backdrop to the backdrop next to or before the currently displayed backdrop in the Backdrops Tab.

This block is available for both sprites and the stage, meaning a sprite can be programmed to change the stage's backdrop.

The *switch backdrop and wait* & *when backdrop switches to* blocks

When the stage's backdrop is changed, an application can be programmed to react to this change using the *when backdrop switches to* event block in the *Events* category.

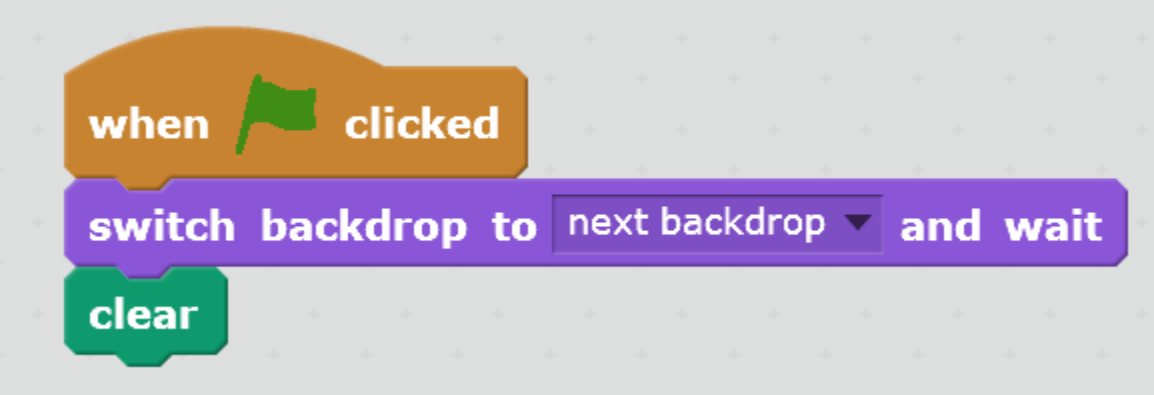
If the Scratch cat sprite has the above script that detects when the stage's backdrop is changed and moves some steps and says hello, and the stage has a script that changes its backdrop to the next one using the *switch backdrop to* block, and clears itself of any drawings made with the pen as below:



This is the sequence of actions that would be performed when the flag is clicked:

- The stage's backdrop is switched to the next one.
- The *when backdrop switches to* event is triggered on the sprite and it starts executing its scripts, starting with *move 10 steps*.
- **At the same time**, the stage continues executing its script. Therefore, the stage's *clear* block and the sprite's *move 10 steps* block are executed at the same time.

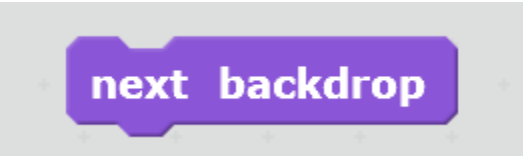
This might not be the behaviour we want in our app. We might want the sprite to move and say "Hello" when the backdrop is changed, and then after the sprite says "Hello", the stage should be cleared. We can achieve this behaviour with the *switch backdrop and wait* block. We will modify the stage's script as follows:



With this modification, when the flag is clicked, the stage changes its backdrop, but instead of going on to execute its *clear* block, it waits for any trigger for the backdrop switching event to complete its script before running the scripts below it. At this point, the sprite's *when backdrop switches to* event block is triggered and the sprite moves 10 steps and says "Hello". With the event trigger now done with its script, the stage continues its script and executes the *clear* block.

The *switch backdrop and wait* block is available only for the stage.

The *next backdrop* block



This block is used to change the backdrop displayed for the stage to the backdrop next to the current backdrop in the Backdrops Tab. If the displayed backdrop is the last one on the list and this block is executed, the stage's backdrop is changed to the first one on the list, and the cycle continues.

Since only the stage can have backdrops, this block is available only for the stage.

Graphic Effects

Any of eight graphic effects can be applied to components in Scratch to modify their look in an artistic way.

A graphic effect has a value, which can be positive or negative, that determines to what extent the effect influences the component's looks. Each sprite added to an application, and the stage, has all of these graphic effects applied on it by default, with an initial value of 0, meaning the graphic effects do not have any visible effect on the component. The higher a graphic effect's value is, the more the effect influences the component's looks, and vice versa.

The *color* graphic effect

A component can have different colours displayed on it. The scratch cat has two major colours: brown and white. Like the pen, all the colours on a component can have a value between 0 and 200.

The *color* graphic effect modifies all the colours on a component by the value supplied to it with reference to each colour's original value. For example, if the scratch cat has an initial value of 20 for the brown colour, 10 for the white colour and 0 for the *color* graphic effect, and the *color* graphic effect is changed to 20, then the sprite's brown colour value increases to 40, changing it to another colour, and the white colour increases to 30, also changing it to another colour. If the value of the *color* graphic effect is reset to 0, the brown colour value is reset to 20, changing it back to brown, and the white colour value is reset to 10, changing it back to white.

The *fisheye* graphic effect

This effect blows a component out (if its value is positive) or in (if its value is negative) from the centre.

The *whirl* graphic effect

This effect twists a component around its centre in a clockwise (for positive values) or anticlockwise (for negative values) direction.

The *pixelate* graphic effect

This effect increases a component's individual pixels in magnitude by its value.

The *mosaic* graphic effect

This block breaks down a component into several smaller copies of its own self.

The *brightness* graphic effect

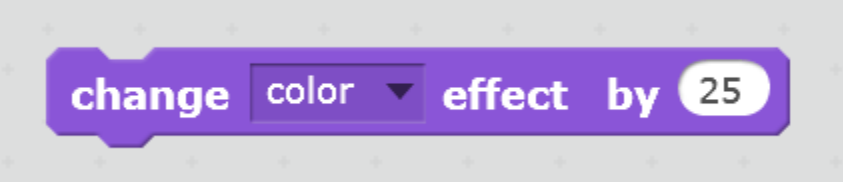
This effect makes a component brighter (for positive values) or darker (for negative values).

The *ghost* graphic effect

This effect makes a component appear fainter. It is affected by only positive values.

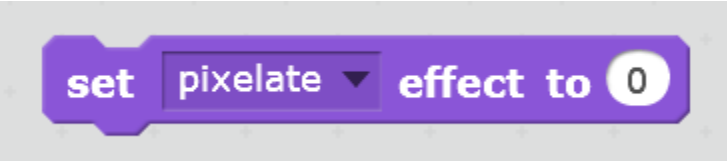
The *Looks* category has blocks that can be used to apply and remove these effects programmatically.

The *change effect by n* block



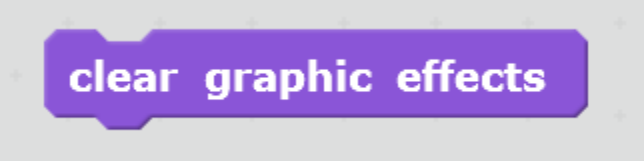
This block increases (for positive *n*) or decreases (for negative *n*) the value of a component's graphic effect by a number, *n*. The graphic effect whose value is to be changed can be selected in the dropdown.

The *set effect to n* block



This block directly sets the value of a component's graphic effect to a number, *n*.

The *clear graphic effects* block



This block resets the values for all of a sprite's graphic effects to 0. This restores the sprite to how it looked when it was added to the application.

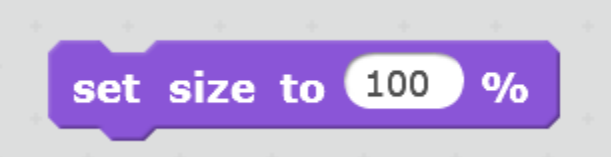
Size

The size of a sprite can be changed using numbers or percentages.

The *change size by n* block

This block increases (for positive values) or decreases (for negative values) a component's size by the number *n* passed to it as an input parameter.

The *set size to n%* block



This block increases or decreases a component's size by a percentage *n* of its original size.

Sprite Layers

Since multiple sprites can be placed on the stage at the same time, there would be situations in which sprites overlap each other on the stage. When this happens, we say they form layers, and the number of layers formed is proportional to the number of overlapping sprites; that is, two overlapping sprites form one layer, three overlapping sprites form two layers, and so on.

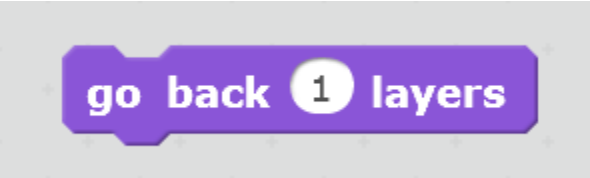
By default, sprites overlap in the order in which they are placed on the stage. They can be programmed to move to the front or back layers with blocks from the *Looks* category.

Three layers: the cat is in the 3rd (back) layer, the bear in the 2nd (back), and the beetle is in the first (front) layer.

The *go to front* block

This block moves a sprite to the front layer when it overlaps with another sprite. If this block is executed on the scratch cat in the initial overlapping above, we would have this:

The *go back n layers* block

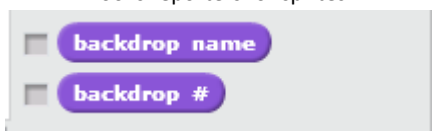


This block moves a sprite back by the number of layers specified by the *n* input parameter. If the *go back 1 layers* block was executed on the bear sprite in the initial overlapping image, we would have this:

Looks property reporter blocks

The blocks we have seen so far in the *Looks* blocks palette are command blocks. The palette also contains reporter blocks that report information related to a sprite and the stage's looks.

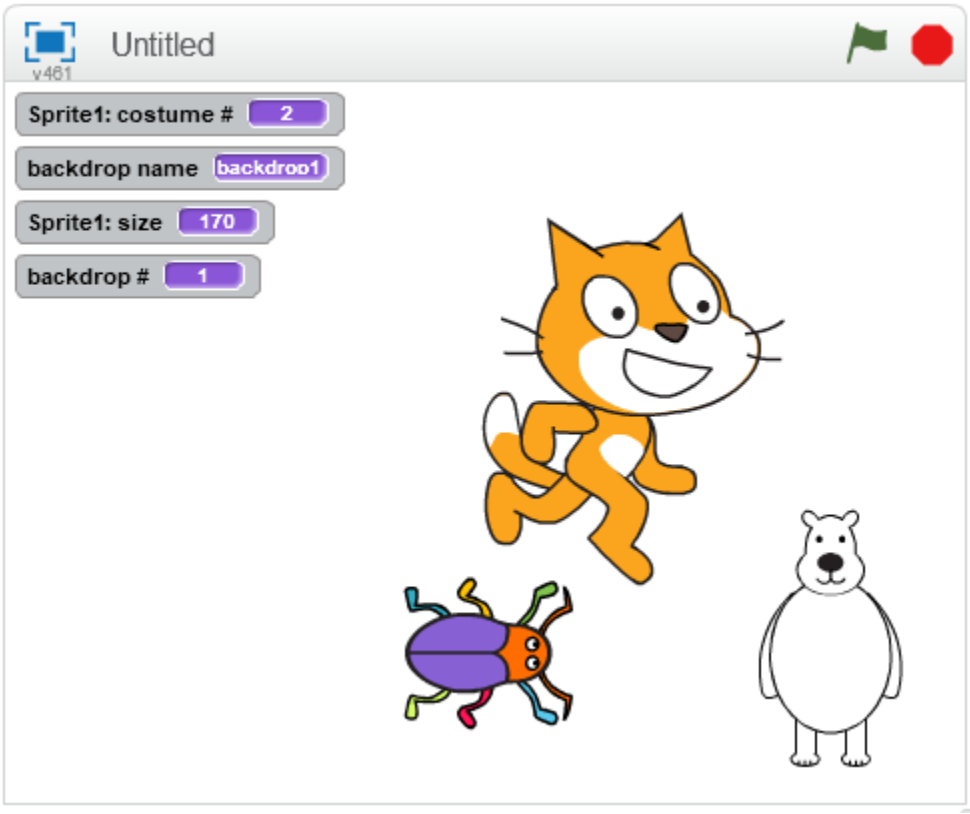
Looks reporters for sprites



Looks reporters for the stage

The *costume #* block reports the number of the currently-displayed costume for a sprite. The *backdrop name* block, which is available for both sprites and the stage, reports the name of the currently-displayed backdrop on the stage. The *backdrop #* block reports the number of the currently-displayed backdrop on the stage, while the *size* block reports the current size of a sprite in percentages.

These blocks, which are property getter blocks, can also be used to track the properties they report in real-time on the stage. To do this, tick the checkbox next to them in the Blocks palette.



Summary of Module Six

The following are the key points covered in this module:

- Everything needed to perform operations related to a component's looks, both inherent (costumes & backdrops) and in relation to other components, is available through the blocks contained in the *Looks* blocks category.
- Speech and Thought behaviours can be emulated in Scratch applications with the *say…* and *think…* blocks in the *Looks* category. When a sprite is programmed to think or speak some text, the text appears in a thought or speech bubble above the sprite on the stage.
- Any of eight (color, fisheye, whirl, pixelate, mosaic, brightness, ghost) graphic effects can be applied to components in Scratch to modify their look in an artistic way.

- When sprites overlap on the stage, they form layers, and the number of layers formed is proportional to the number of overlapping sprites.
- By default, sprites overlap in the order in which they are placed on the stage. They can be programmed to move to the front or back layers with blocks from the *Looks* category.

CHAPTER 7

Programming Apps with Sound

Table of Contents

Learning Objectives

By the end of this chapter, the student would:

- Understand the different forms of audio that can be possessed and played by a component in Scratch.
- Understand how the different blocks from the Sounds category can be used to program a component to play different forms of audio.
- Learn how to compose musical notes with Sound blocks.

Chapter Prerequisites

- Scratch 2 Offline Editor set up on the computer with access to local storage.

It is always best to make your computer programs as exciting to use as possible, as that is what will keep users coming back to the app frequently. Games, in particular, need to play music and provide audio response to certain events in the game, in order to keep the user interested. What would you think of a football game that has no commentary and sounds of fans cheering on the pitch? Or a music game in which the instruments and artistes move when interacted with, but do not produce any sound?

In the last chapter, we looked at how we can make a computer program more interesting visually with blocks from the *Looks* block category. In this chapter, you will learn how to use blocks from the *Sounds* category to program sprites and the stage to play music and give off sound effects in response to events in an application. You will also learn how to create your own audio files and modify existing ones, upload audio files for use in a Scratch application, and modify a component's sound properties.

There are three forms of audio that a component can have and play.

Sounds

A sound, which is simply an audio file, can be created and assigned to a component. All of the sounds that have been assigned to a component are contained in the Sounds Tab that we first learned about in the third chapter.

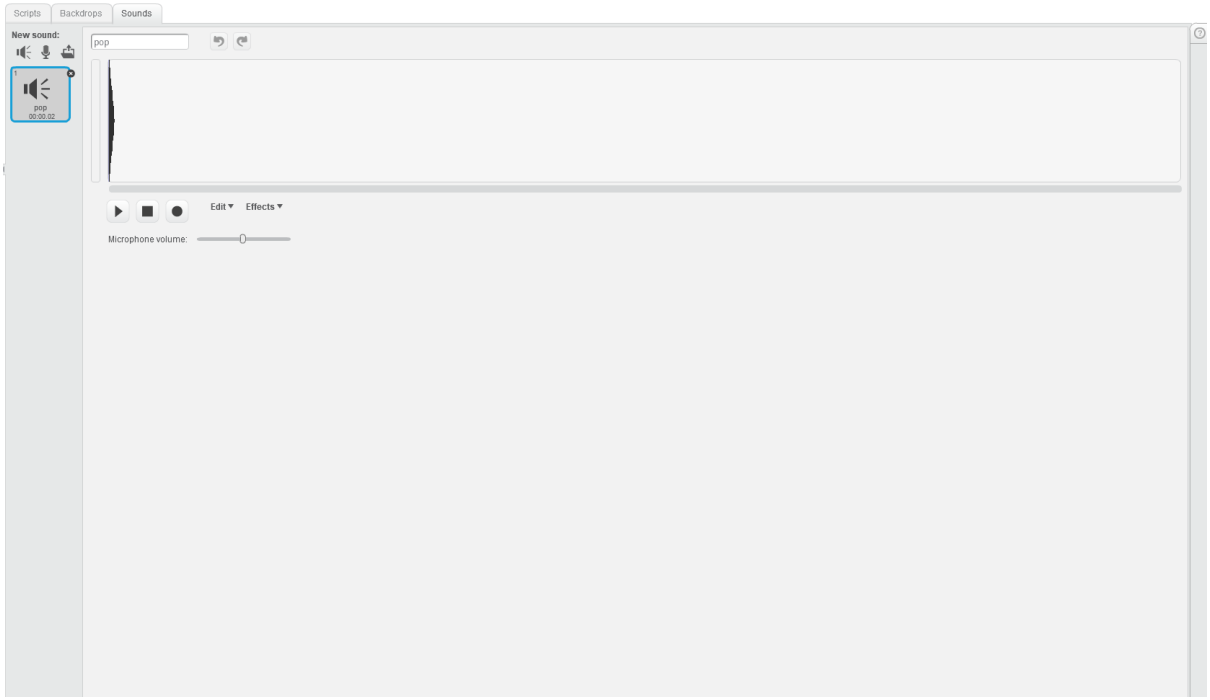


Figure 7.1

Most sprites in the Sprite Library have sounds assigned to them automatically when they are added to an application, and the stage has a single sound assigned to it by default. Sounds can be added to sprites and the stage using the *New sound* buttons in the Sounds Tab.

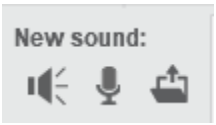


Figure 7.2

The first button is used to add a sound from the Scratch Sound Library.

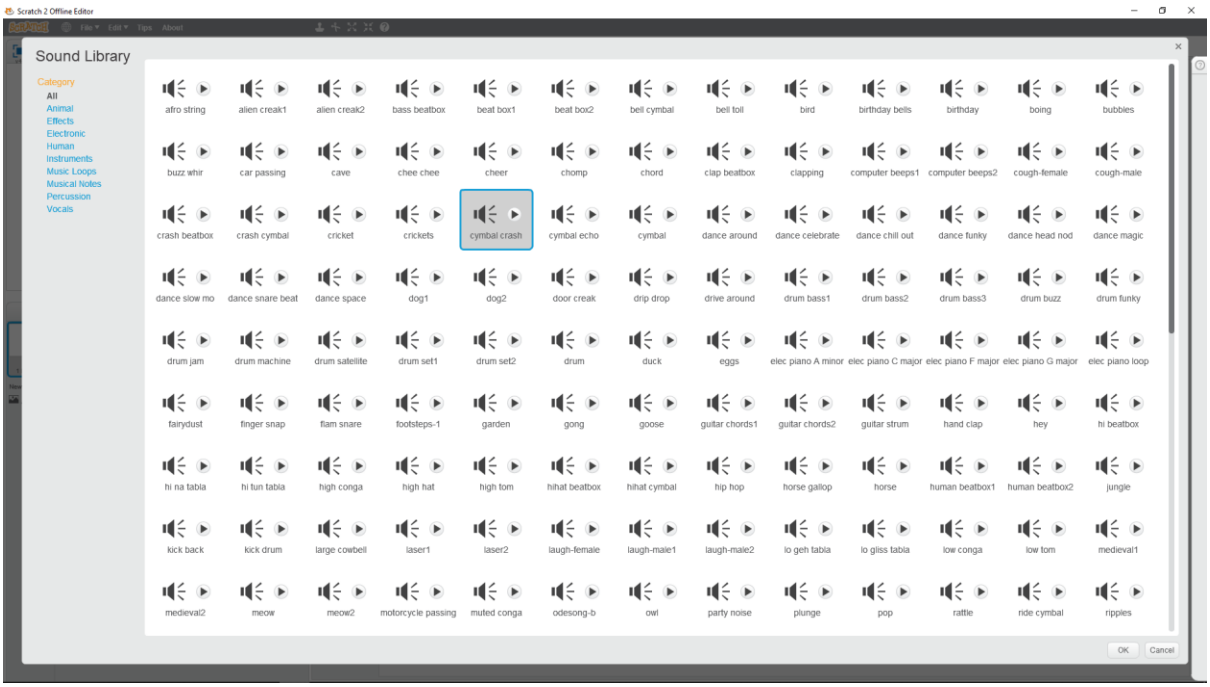


Figure 7.3: You can play a sound right there in the library before deciding whether to add it to a component or not.

When a sound is selected in the Sounds Tab, you can see the composition of audio waves in it.

Figure 7.4

You can select parts of the sound's audio waves and perform operations available in the *Edit* dropdown on them.

Figure 7.5

You can also add effects to the sound using the options in the *Effects* dropdown, and play/stop the sound while testing with the buttons in the bottom left. The third button allows you to add a recording to the sound's existing composition.

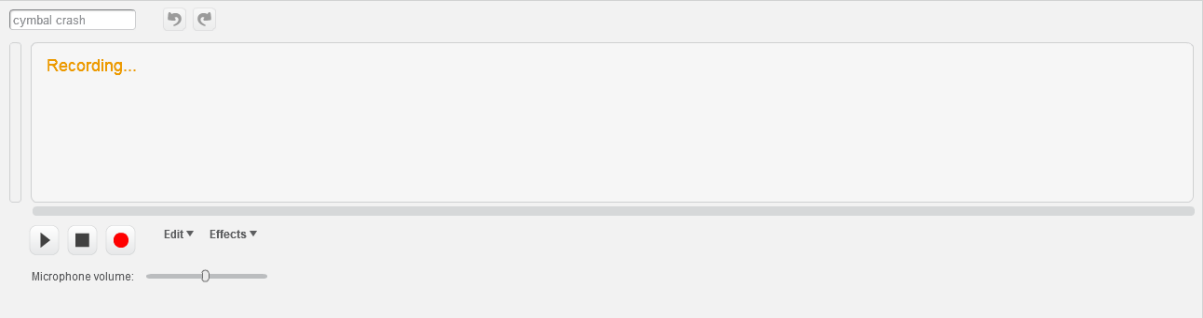


Figure 7.6

For recording, you can modify the level to which sound is picked up from the microphone using the *microphone volume* slider.

Every sound has a name that can be modified using the text box at the top left corner.

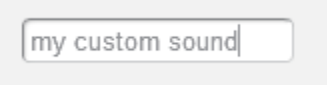


Figure 7.7

The second *New sound* button is used to record a new sound using your computer's microphone. Clicking on the button creates a new sound file with an empty composition.

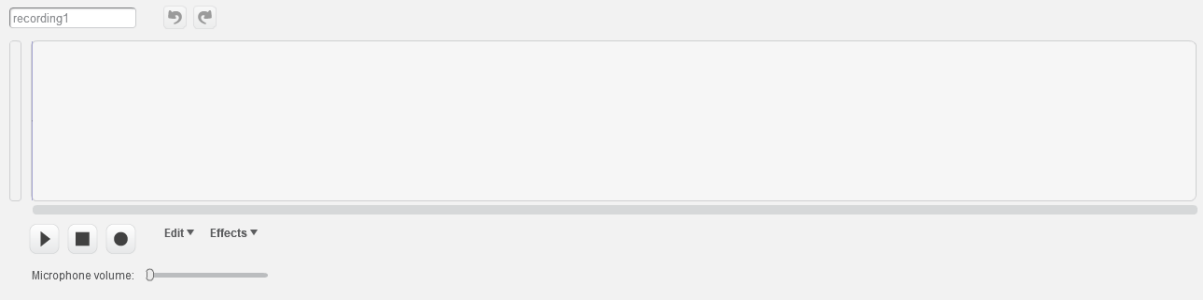


Figure 7.8

The third *New sound* button is used to upload a sound file to a component from your computer storage. If you have an audio file that you want to play using a sprite or the stage in your application, this button is what you need.

NOTE: Scratch can recognize audio files in only the WAV and MP3 formats (that is, the files have the .wav and .mp3 extensions). If an audio file that you want to use in your application is in a format other than these two, you have to convert it to either before uploading it to your application.

Drums

Scratch allows you to play up to 18 different drum sounds with the stage and sprites in an application.

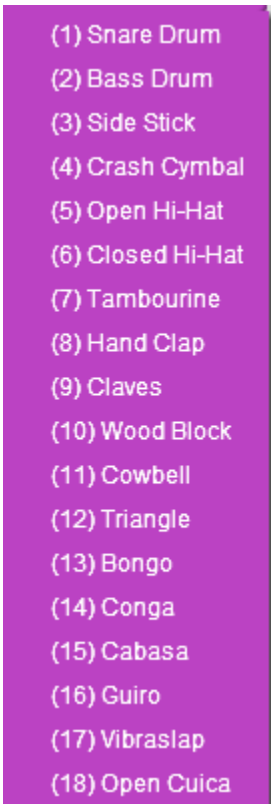


Figure 7.9

You do not need to add these drum sounds manually to a component before you can play it: Scratch does that automatically for every component added to an application.

Musical Instruments

Scratch allows you to play sounds for up to 21 different musical instruments with the stage and sprites in an application.

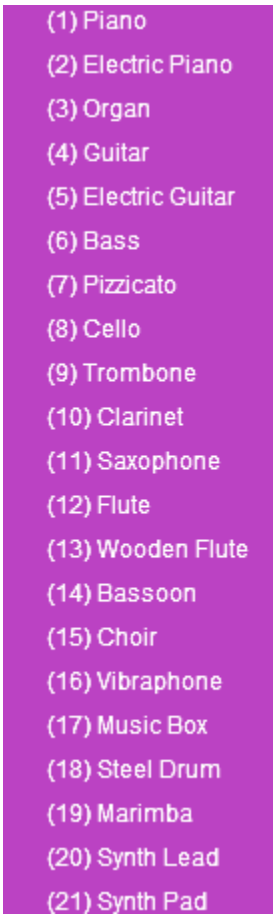


Figure 7.10

These instrument sounds are also added automatically for every component in an application.

Sound Properties of a Component

The way the drum and musical instrument sounds are played by a component depends on two of the component's sound properties, which are *beat* and *tempo*. A beat is a single strike of the drum or musical instrument, and the tempo determines the number of beats that are made for a component's drum or instrument per minute. If the tempo is set to 60 beats per minute (bpm) and the component is

made to play a drum or instrument for one beat, the component would play the drum or instrument for one second, since 60 bpm = 60 beats per 60 seconds = 1 beat per second.

Volume is another sound property that determines how loud a component plays its sounds, drums and instruments.

Now that we know the different ways in which sounds are composed in components, let us see how these components can be programmed with blocks to manipulate these sounds in a program.

The *Sound* Blocks Category

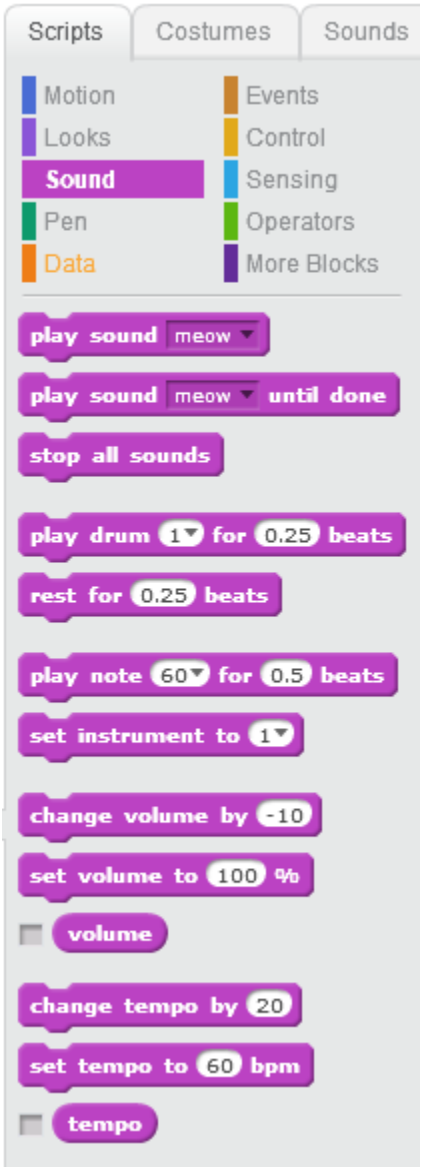


Figure 7.11

The *play sound* block

Figure 7.12

This block instructs a component to play the sound whose name is selected in the dropdown. The dropdown contains the names of all the sounds that are present for a component in the Sounds Tab.

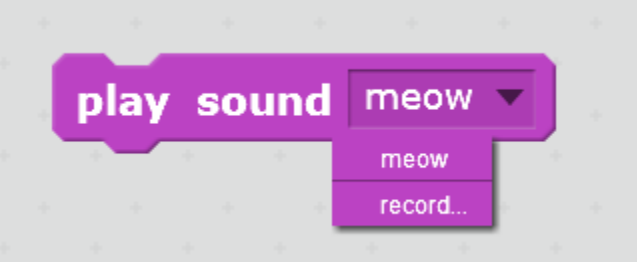


Figure 7.13

The *record* option can also be selected to create a new, blank audio file in the Sounds Tab for a component, for the user to record sound into. Executing this block will start playing the sound and then immediately continue executing all blocks/scripts below it, even if the sound is still playing.

The *play sound until done* block

Figure 7.14

This block works like the *play sound* block, but instead of continuing with the execution of the blocks and scripts below it, the *play sound until done* block waits for the sound to finish playing before executing other blocks below it.

The *stop all sounds* block

Figure 7.15

This block is used to stop all audio currently being played by the stage or a sprite. This applies to sounds, drums and instruments.

The *play drum for n beats* block

Figure 7.16

This block is used to play a drum sound selected in the first input parameter for a number, *n*, of beats entered as the second input parameter.

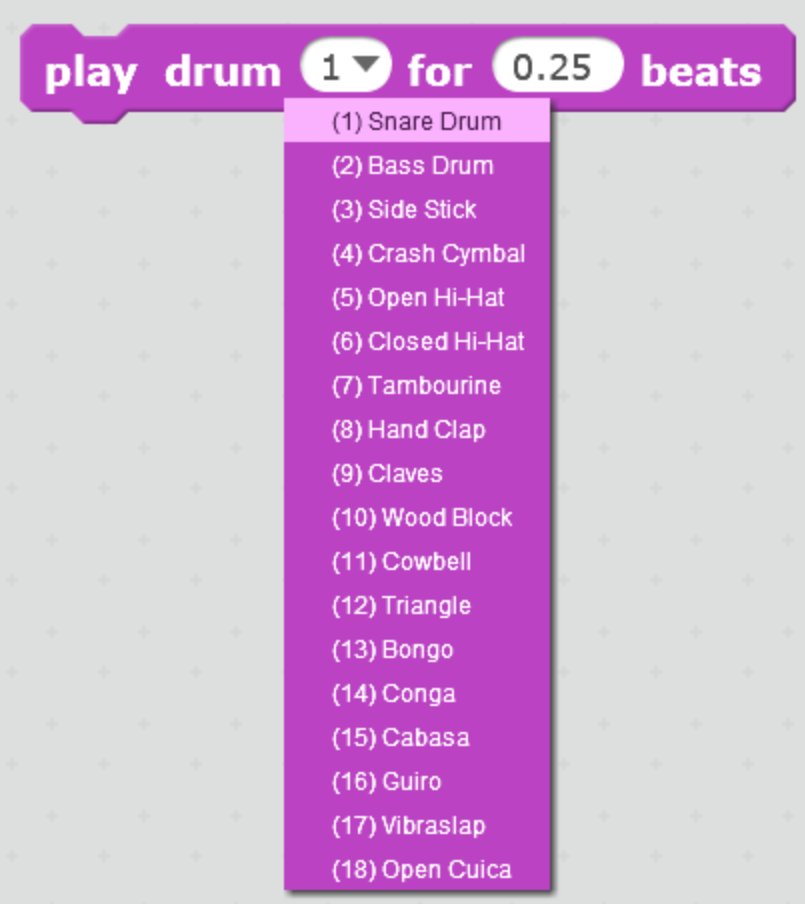


Figure 7.17

The *rest for n beats* block

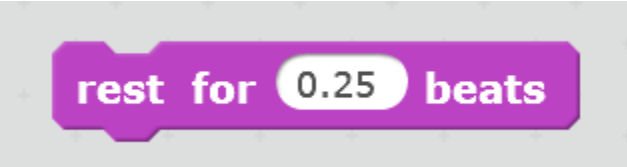


Figure 7.18

This block is used to pause all audio from a sprite for a specified number, *n* of beats. This applies to sounds, drums and instruments.

The *play note for n beats* block

Figure 7.19

This block is used to play a note from a component's currently-selected musical instrument for a specified number, *n* of beats. The note to be played can be selected from the dropdown for the first input parameter.

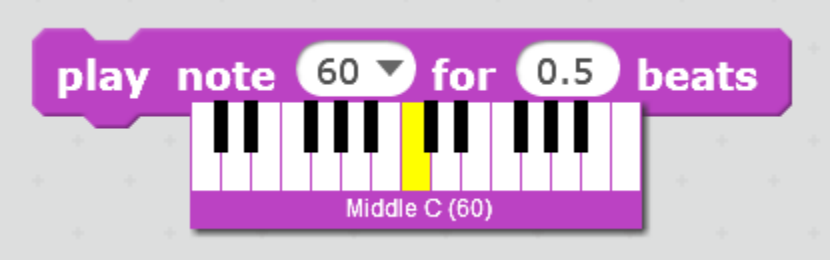


Figure 7.20

The *set instrument to* block

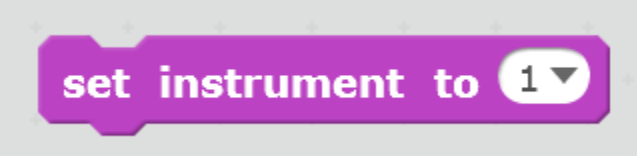


Figure 7.21

This block is used to select a musical instrument to be played by a component. The instrument selected for a component with this block is the one from which notes are played with the *play note for n beats* block.

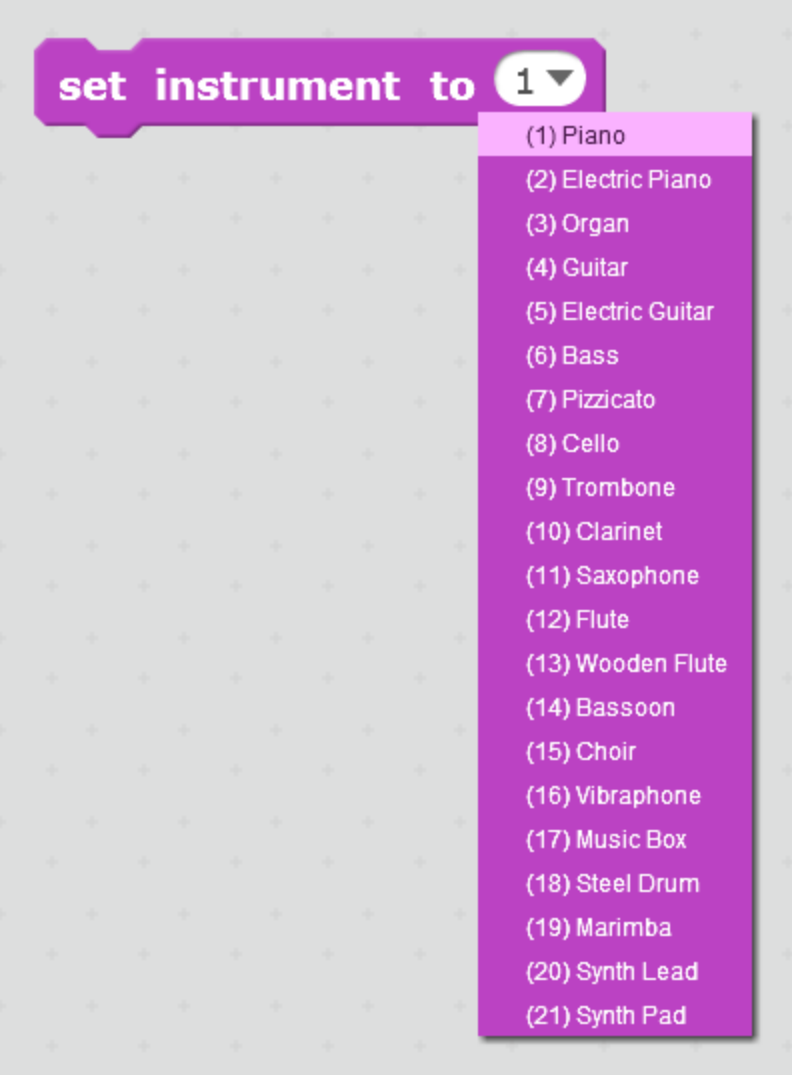


Figure 7.22

The *change volume by n* block

Figure 7.23

This block is used to increase (for positive values of n) or decrease (for negative values of n) the volume of all sounds played by a sprite by a specified value, *n*.

The *set volume to n%* block

Figure 7.24

This block is used to set the volume of all sounds played by a sprite to a percentage, *n%* of its original value.

The *change tempo by n* block

Figure 7.25

This block is used to increase (for positive values of *n*) or decrease (for negative values of *n*) the tempo (beats per minute, bpm) of a component's drum and instrument sounds by a specified number, *n*.

The *set tempo to n bpm* block

Figure 7.26

This block is used to set the tempo of a component's drum and instrument sounds to a specified number, *n*.

Sound property reporter blocks

The blocks we have seen so far in the *Sounds* blocks palette are command blocks. The palette also contains reporter blocks that report a component's sound properties - volume and tempo.

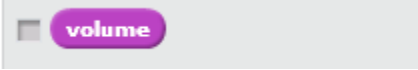


Figure 7.27

Figure 7.28

The *volume* block reports the current value of the volume property of a component, which determines how loud the sounds played by the component will be, and the *tempo* block reports the current value of the tempo property of a component, which determines how many the speeds (beats per minute) of the drums and instrument sounds played by a component.

These blocks, which are property getter blocks, can also be used to track the properties they report in real-time on the stage. To do this, tick the checkbox next to them in the Blocks palette.

Figure 7.29

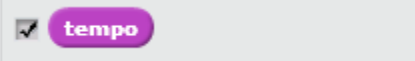


Figure 7.30

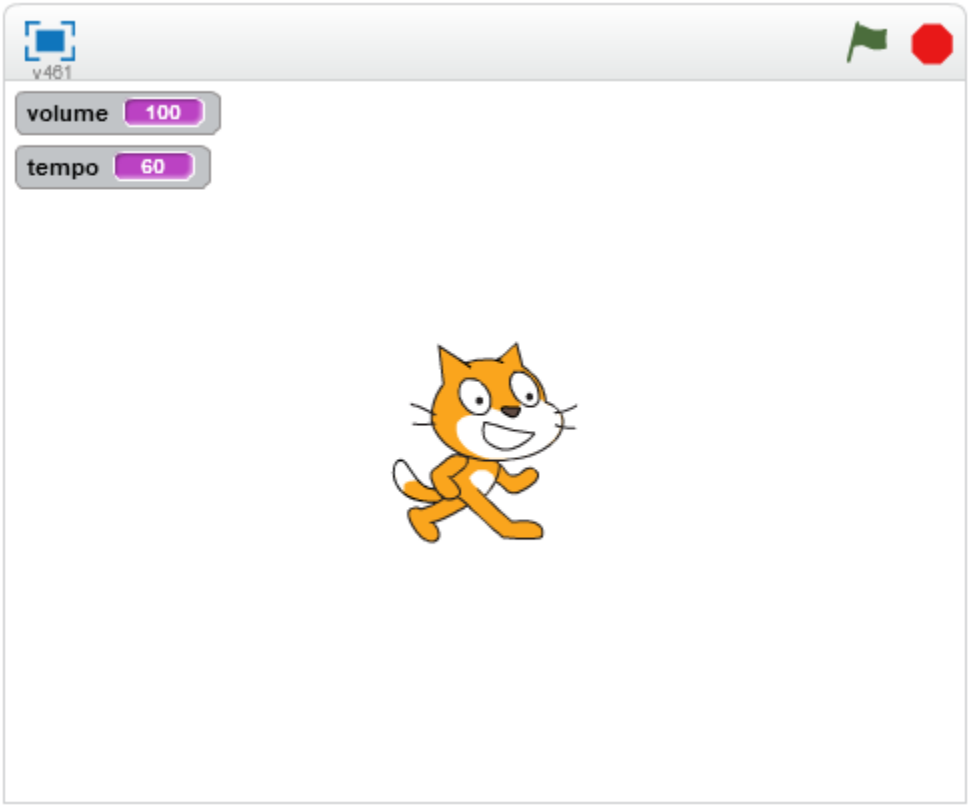


Figure 7.31

Chapter Project: 'Mary Had A Little Lamb'

To demonstrate the usage of the blocks in the Sounds category which we have learned about in this chapter, we will build a simple computer program that uses the Scratch cat to play the popular nursery rhyme, *Mary Had A Little Lamb*, on a saxophone.

We would program the application so that when the flag is clicked, the Scratch cat says "Playing: 'Mary Had A Little Lamb'" and starts playing the rhyme immediately using a saxophone musical instrument.

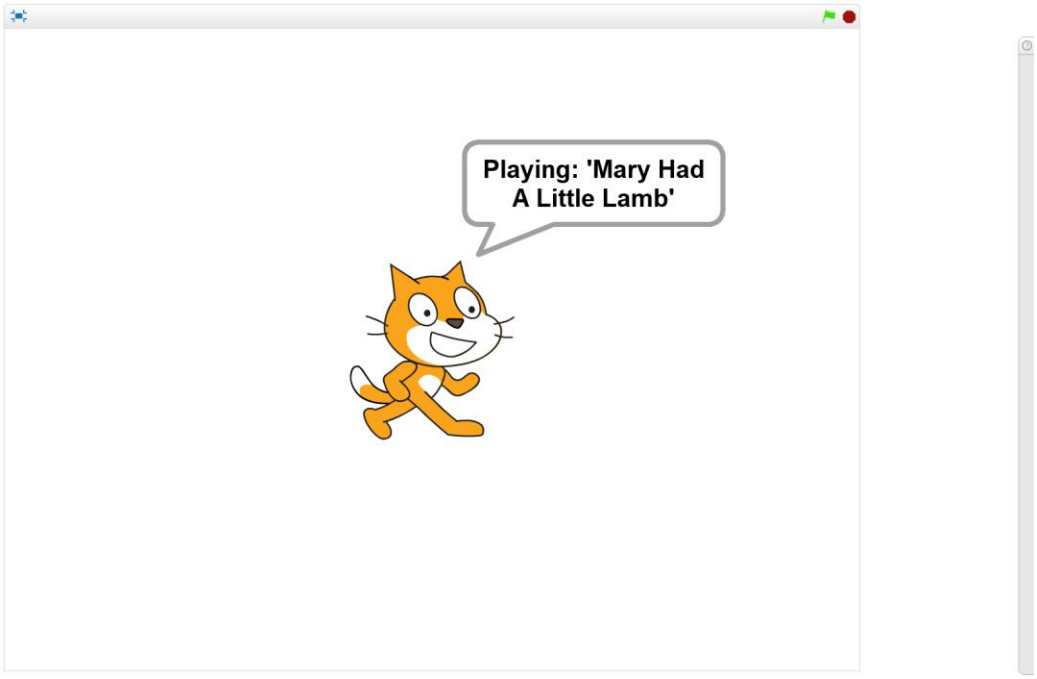


Figure 7.32

As usual, we would follow our application development process in building this app.

Determine all the functions needed in the app

This application is a simple one with just a few functions. These are:

- When the flag is clicked:
  - Scratch cat says "Playing: 'Mary Had A Little Lamb'".
  - Scratch cat plays rhyme using a saxophone.

Decide which components are needed to perform these functions

From the final version of the application in *Figure 7.32* above, the only components needed are the stage with a white background and the Scratch cat.

There are different ways to program the Scratch cat to play the rhyme. We could upload an audio file with the rhyme to our application through the Sounds Tab, or record the rhyme with the computer microphone right there in the Sounds Tab. We could then use the *play sound* blocks to program the sprite to play these audio files.

However, a more interesting method is to use the various musical instruments available to sprite to play the musical notes that comprise the 'Mary Had A Little Lamb' rhyme. This is the method we would use for this project, as it would help you to understand how to use these blocks to compose music.

PROJECT COMPONENTS

COMPONENT	FINDINGS
Backdrop	● 1 White Background
Sprite	● 1 Scratch Cat
Costume	● 1 Scratch Cat
Sound	None

Add these components to the application

It is now time to jump into the Scratch Editor.
Open the Scratch 2 Offline Editor application on your computer and a new project containing the Scratch cat will be automatically opened for you. A white backdrop is also added to the stage automatically. We now have every component we need for this application.

Program the components to perform their functions

Here are the functions we derived from the first step of the process:

- When the flag is clicked:
    - Scratch cat says "Playing: 'Mary Had A Little Lamb'".
    - Scratch cat plays rhyme using a saxophone.

The first statement is a familiar block from the Events block category. Go ahead and add this block for the Scratch cat.

Figure 7.33

For the *Scratch cat says "Playing: 'Mary Had A Little Lamb'"* function, we would need the *say* block from the *Looks* category.

Figure 7.34

Now to the last function. To program this function for the Scratch cat, we have to figure out the musical notes that comprise the rhyme. Here are the lyrics of the rhyme:

Mary Had A Little Lamb, Little Lamb, Little Lamb. Mary Had A Little Lamb, Its Fleece Was White As Snow.

Before we derive the musical notes, we have to break down the lyrics into single syllables, as each syllable in a musical tune has its own musical note.

Ma ry Had A Lit tle Lamb, Lit tle Lamb, Lit tle Lamb, Ma ry Had A Lit tle Lamb, Its Fleece Was White As Snow.

Now we can derive the musical notes. There are seven notes in music: C D E F G A B C. Every musical tune falls in between one of these notes. Here are the notes for each syllable in the 'Mary Had A Little Lamb' rhyme:

- Ma - E
- ry - D
- Had - C
- A - D
- Lit - E
- tle - E
- Lamb - E
- Lit - D

- tle - D
- Lamb - D
- Lit - E
- tle - G
- Lamb - G
- Ma - E
- ry - D
- Had - C
- A - D
- Lit - E
- tle - E
- Lamb - E
- Its - E
- Fleece - D
- Was - D
- White - E
- As - D
- Snow - C

We will now go ahead to program the sprite to play these notes with the saxophone instruments using blocks from the Sound category.

Firstly, set the instrument to *Saxophone* (11)

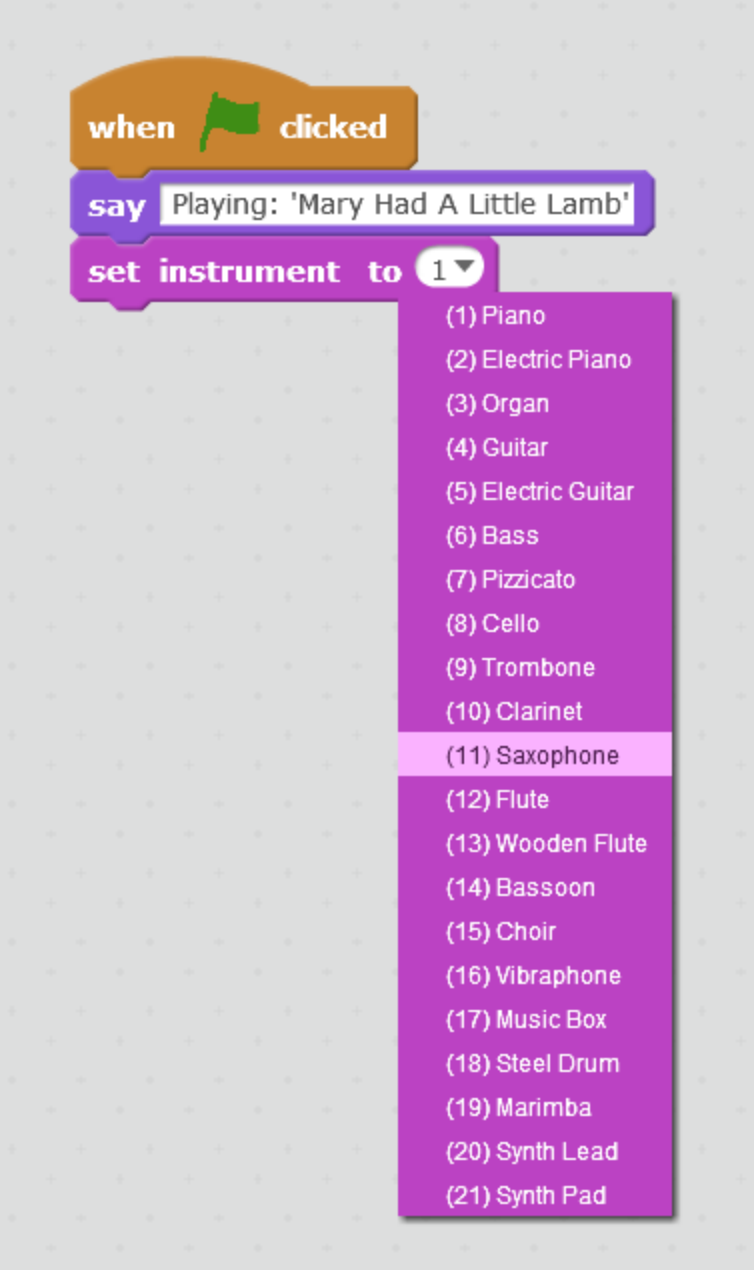


Figure 7.35

Next, set a desired volume and tempo for the sprite.

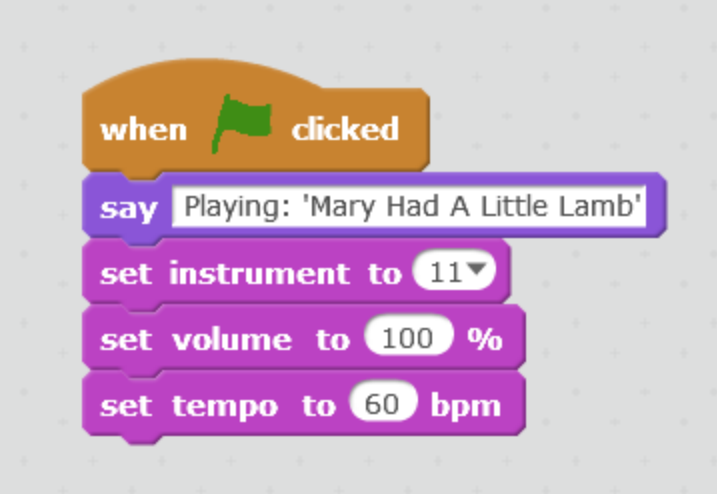


Figure 7.36

Add the *play note for n beats* block for the first note, an E note. This should be played for 1 beat.

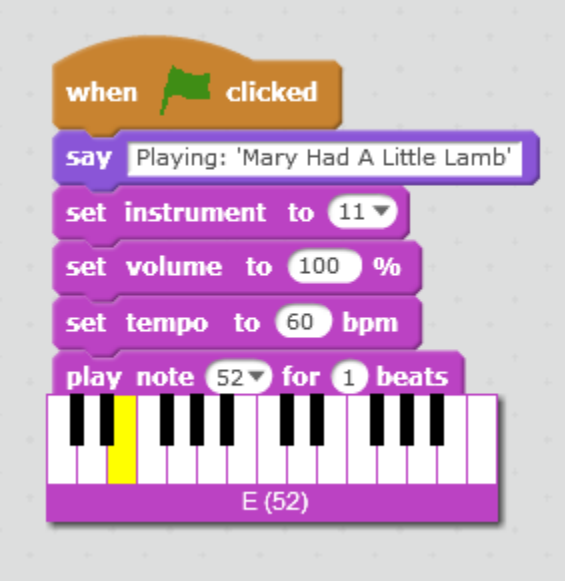


Figure 7.37

Do the same for all of the other notes.

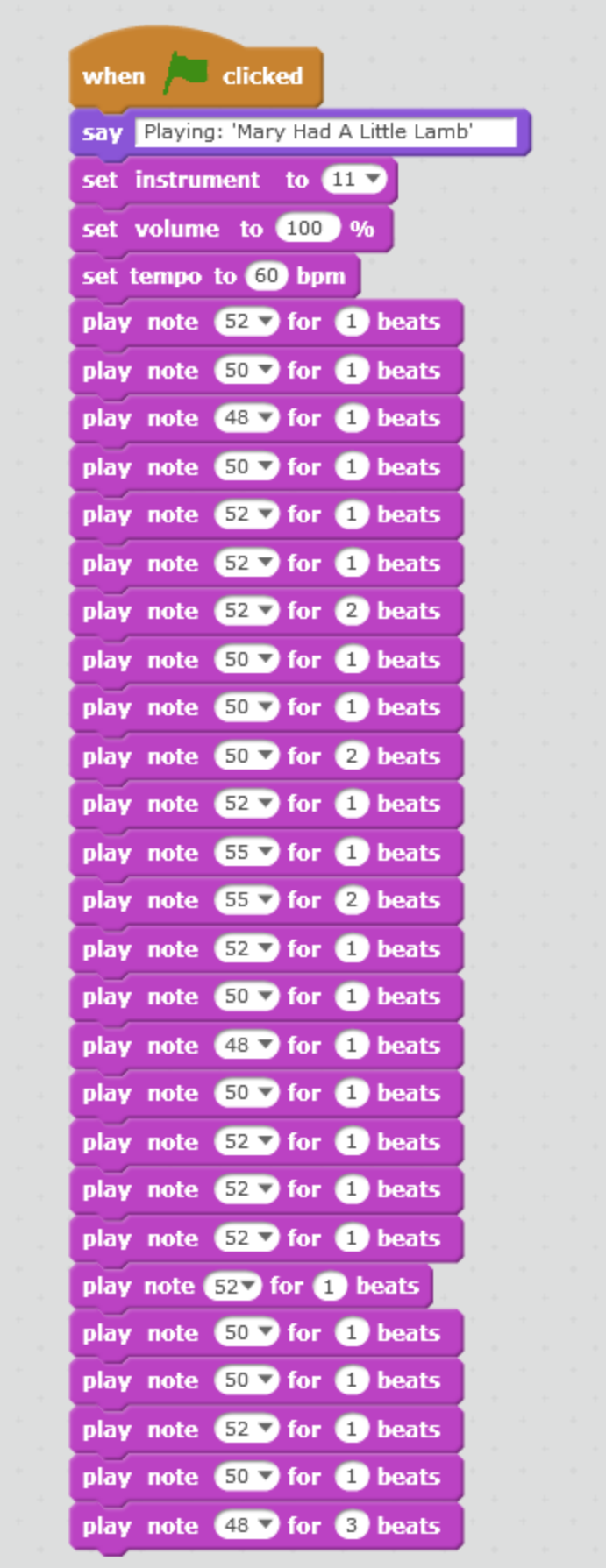

when clicked
say Playing: 'Mary Had A Little Lamb'
set instrument to 11
set volume to 100 %
set tempo to 60 bpm
play note 52 for 1 beats
play note 50 for 1 beats
play note 48 for 1 beats
play note 50 for 1 beats
play note 52 for 1 beats
play note 52 for 1 beats
play note 52 for 2 beats
play note 50 for 1 beats
play note 50 for 1 beats
play note 50 for 2 beats
play note 52 for 1 beats
play note 55 for 1 beats
play note 55 for 2 beats
play note 52 for 1 beats
play note 50 for 1 beats
play note 48 for 1 beats
play note 50 for 1 beats
play note 52 for 1 beats
play note 52 for 1 beats
play note 52 for 1 beats
play note 52 for 1 beats
play note 50 for 1 beats
play note 50 for 1 beats
play note 52 for 1 beats
play note 50 for 1 beats
play note 48 for 3 beats

Figure 7.38: Take notice of the notes with 2 and 3 beats.

Now switch to fullscreen mode and click on the flag. Does the Scratch cat play the 'Mary Had A Little Lamb' rhyme?

Summary of Chapter Seven

The following are the key points covered in this chapter:

- Programming an application to provide audio response to certain events, especially in games, is important in order to keep the user interested in the application.
- Everything needed to make a Scratch component play audio in different forms is available through the blocks contained in the *Sound* blocks category.
- There are three forms of audio that a Scratch component can have and play: Sounds, Drums and Musical Instruments.
- Sounds are audio files that can be created, modified and assigned to a component through the Sounds Tab.
- Audio files can be uploaded to an application and assigned to a component through the Sounds Tab. Scratch recognizes audio files in only the WAV and MP3 audio formats.
- Components can be programmed to play up to 18 different built-in drum sounds in Scratch.
- Components can be programmed to play sounds for up to 21 different built-in musical instruments in Scratch.
- Components possess three sound properties: beat, tempo and volume. A beat represents a single strike of a drum or musical instrument, the tempo determines the speed, in beats per minute, of the sound played by a component, and the volume determines how loud the sound played by a component is.

CHAPTER 8

Beefing Up The Drawing Bee Application

Table of Contents

- The Drawing Bee application.

Learning Objectives

By the end of this chapter, the student would:

- Be able to program an application to play sound effects.
- Be able to program an application to perform animations.
- Be able to build an advanced version of the Drawing Bee application.

Chapter Prerequisites

- Scratch 2 Offline Editor set up on the computer with access to local storage.

In the fifth chapter, we applied our knowledge of the Motion and Pen blocks to build Drawing Bee, an application that features a Bee which can draw lines and stamp itself on the stage. In this chapter, we will modify the Drawing Bee application, using our knowledge of the Looks and Sound blocks to make it more exciting. The Bee would make different sounds when it is moved with the mouse or keyboard, when its pen is put up or down, when its pen's colour is changed, when the bee is stamped, when the stage is cleared, and when the pen's size is changed. We will also add new functions to the app; increasing and decreasing the bee's size, hiding and showing the bee, and switching the bee's costume so that we can see the pen drawings underneath the bee. Lastly, we would make use of graphical effects to perform animations when the bee's size is increased or decreased, when the bee is stamped,and when the bee is hidden and shown.

This is what our Drawing Bee application would look like after we are done with these additions and upgrades:

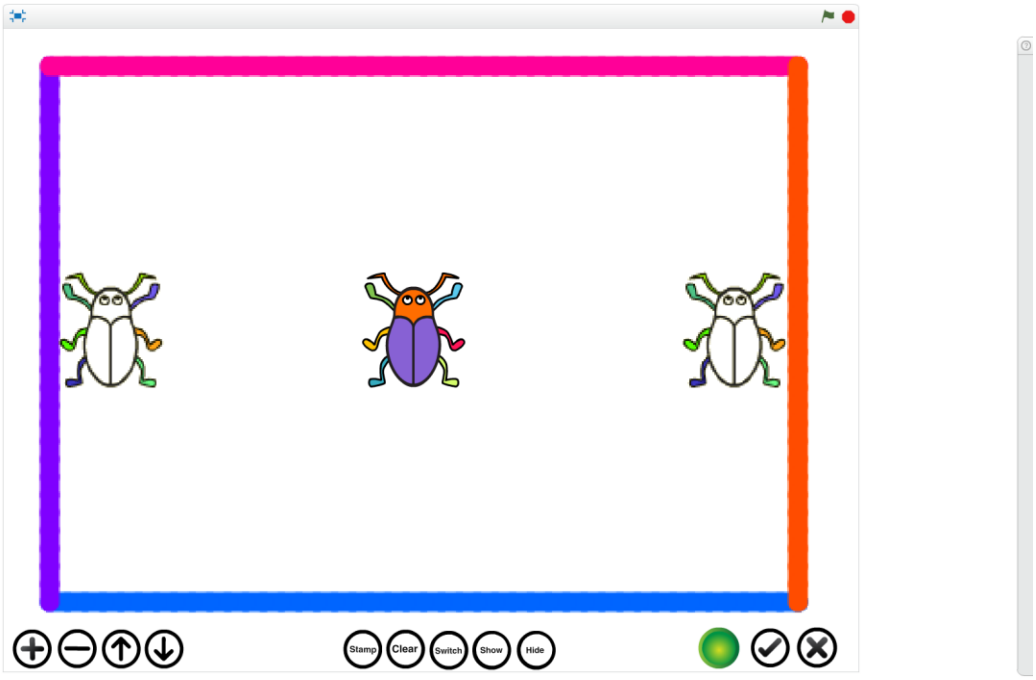


Figure 8.1

Since there are new components to be added to the app, we would follow all the steps in our app development process.

Determine all the functions needed in the app

These are the new functions to be added to the Drawing Bee application:

- Ability to hide and show the bee.
- Ability to switch the bee's costume.
- Ability to increase and decrease the bee's size.
- Play sounds when any function is performed on the bee, its pen, or the stage.
- Perform an animation when the bee's size is increased or decreased.
- Perform an animation when the bee is shown or hidden.
- Perform an animation when the bee is stamped.

Decide which components are needed to perform these functions

Look closely at the final version of the application shown in *Figure 8.1* and point out the new components that you need to add to the existing application. The new components are listed in bold in the project components table below.

PROJECT COMPONENTS

COMPONENT	FINDINGS
Backdrop	• 1 White Background
Sprite	• 1 Bee • **12 Buttons**
Costume	• 1 Multi-coloured bee • **1 bee with the legs and arms coloured, and the body and head transparent** • 1 circle with a white background and black border with a "-" icon in it • 1 circle with a white background and black border with a "+" icon in it • 1 circle with a white background and black border with the text "Stamp" written in it • 1 circle with a white background and black border with the text "Clear" written in it • 1 multi-coloured circle • 1 circle with a white background and black border with a "✓" icon in it • 1 circle with a white background and black border with a "✕" icon in it • **1 circle with a white background, black border, and an up arrow icon in it** • **1 circle with a white background, black border, and a down arrow icon in it** • **1 circle with a white background and black border with the text "Switch" written in it** • **1 circle with a white background and black border with the text "Show" written in it** • **1 circle with a white background and black border with the text "Hide" written in it**
Sound	• **1 sound for moving the bee with the keyboard arrow and number keys** • **1 sound for moving the bee with the mouse** • **1 sound for decreasing the pen size** • **1 sound for increasing the pen size** • **1 sound for stamping the bee**

	• **1 sound for clearing the stage** • **1 sound for changing the pen's colour** • **1 sound for putting the bee's pen down** • **1 sound for putting the bee's pen up** • **1 sound for increasing the bee's size** • **1 sound for decreasing the bee's size** • **1 sound for switching the bee's costume** • **1 sound for showing the bee** • **1 sound for hiding the bee**

Add these components to the application

Before we go ahead and add the new components to the app, notice that the buttons in *Figure 8.1* are smaller than they are in the first version of the application built in the fifth chapter. We need to reduce the size of the buttons to make it easier to add more buttons and fit them all nicely at the bottom of the application.

Use the Paint Editor to reduce the seven buttons currently present in your application.

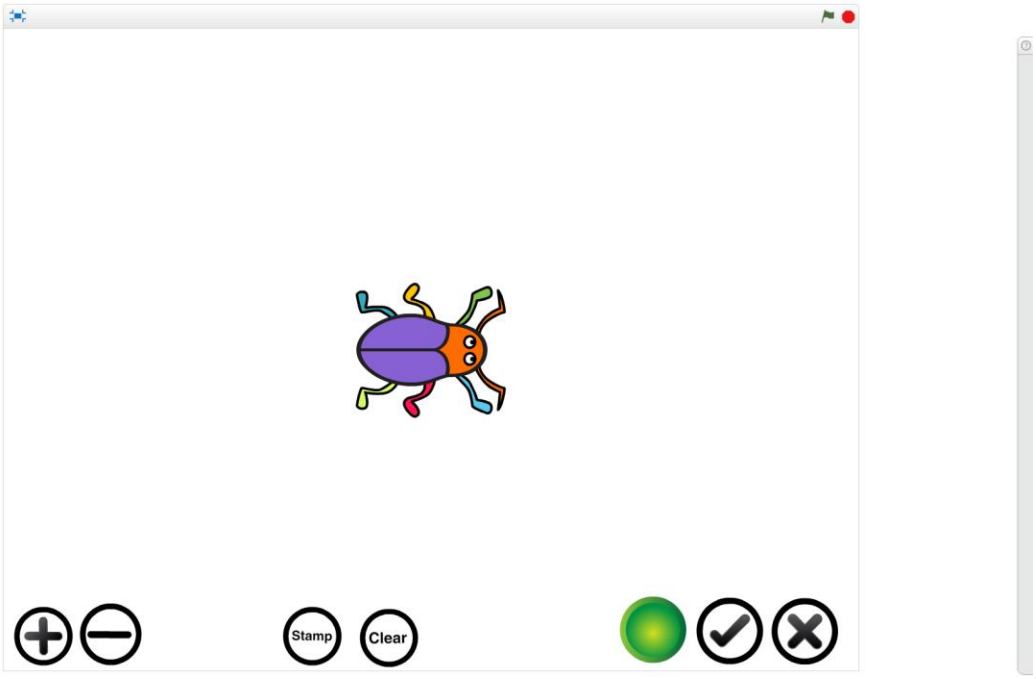


Figure 8.2

Before

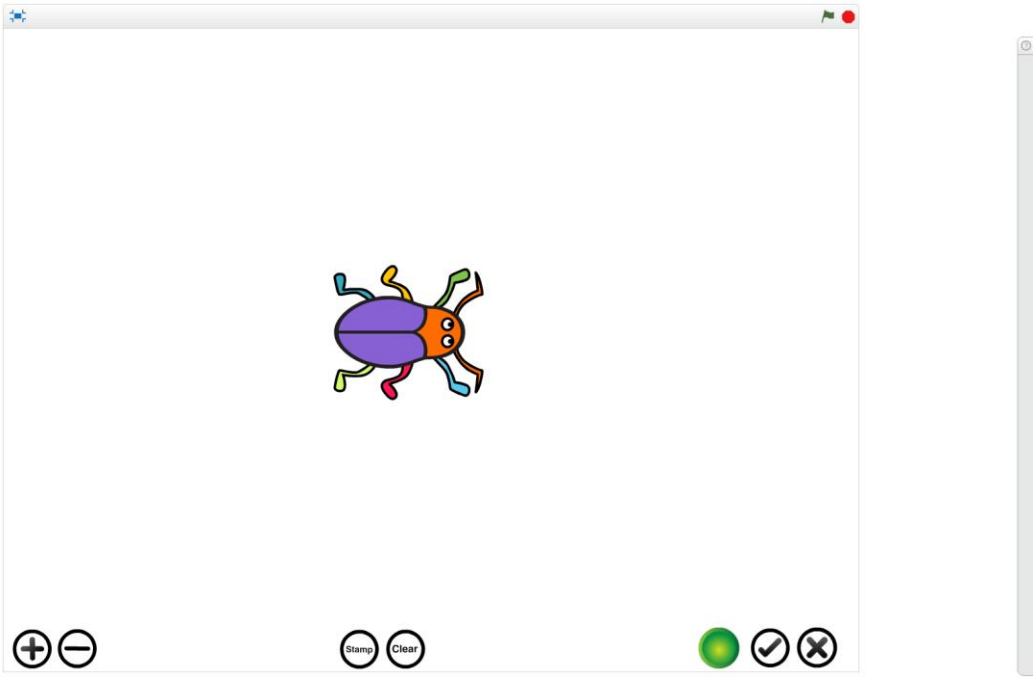


Figure 8.3

After

Now go ahead and add each new button to the app. You can copy images from the costumes of existing components and modify them to create the new components.

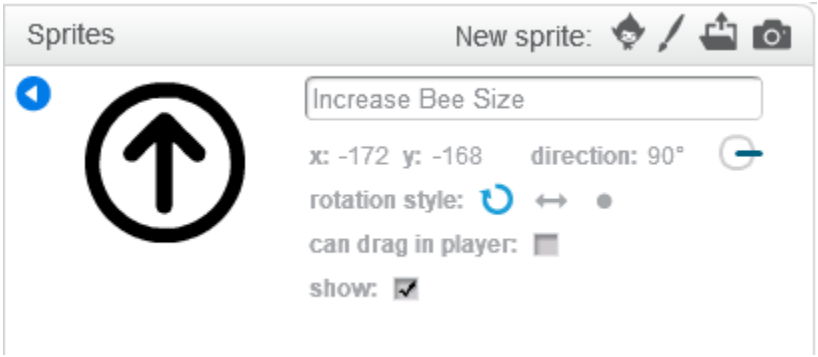


Figure 8.4

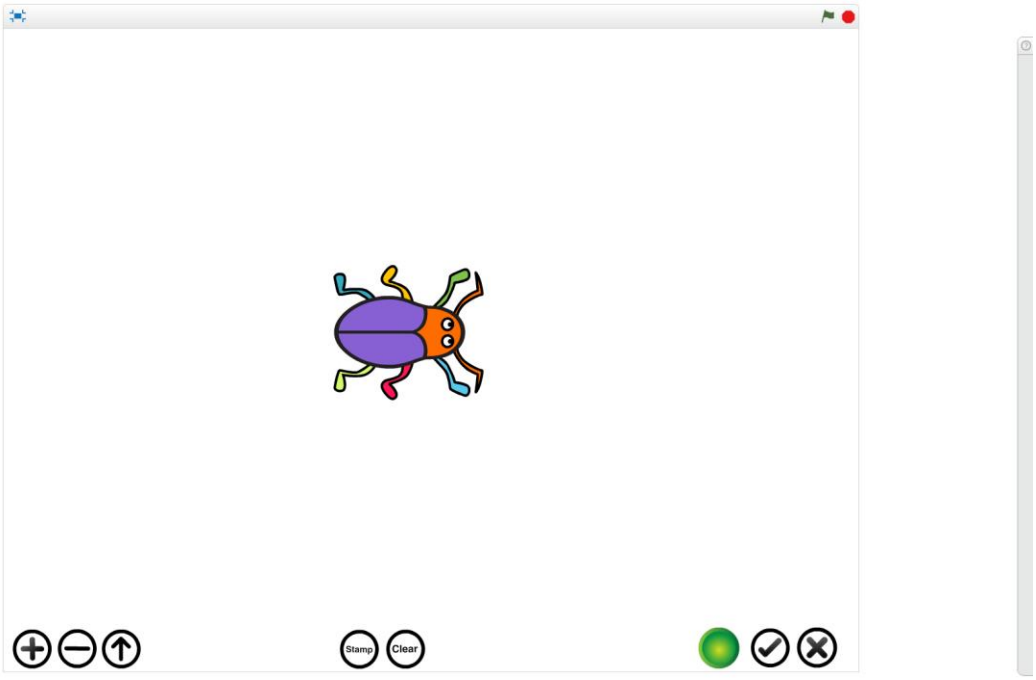


Figure 8.5

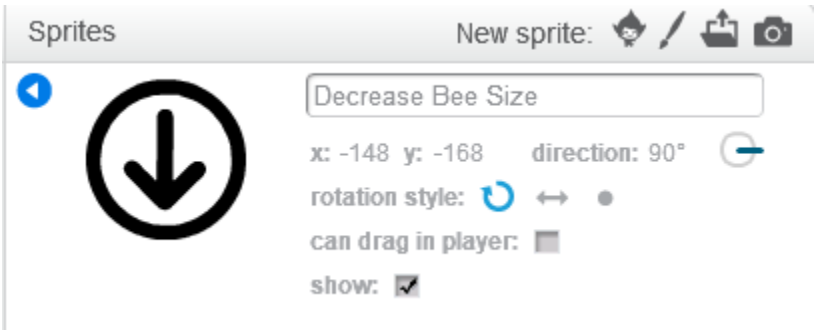


Figure 8.6

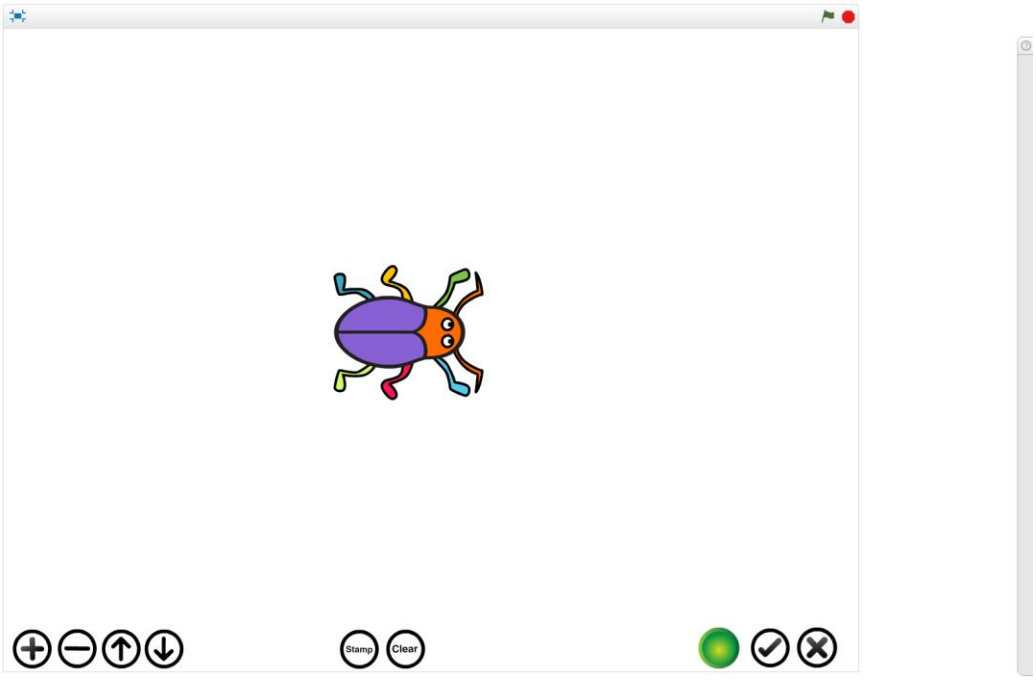


Figure 8.7

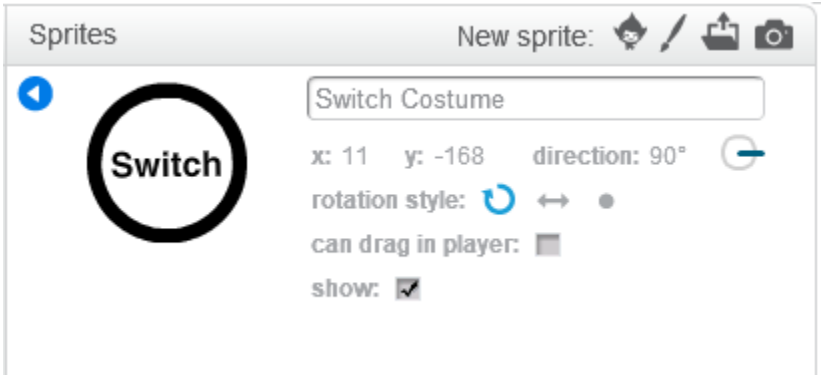


Figure 8.8

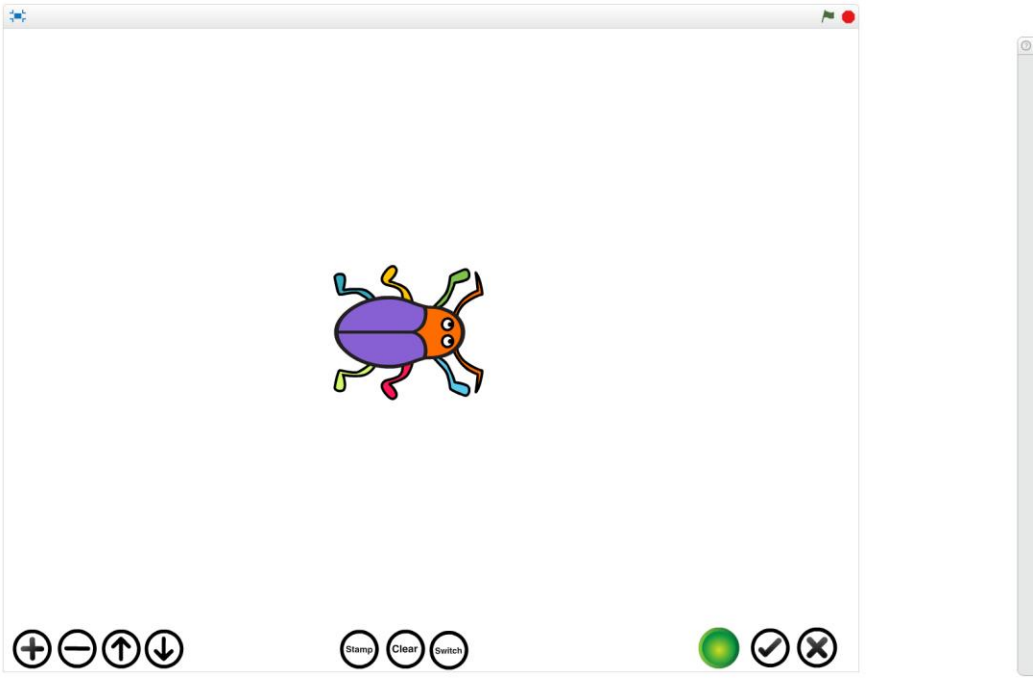


Figure 8.9

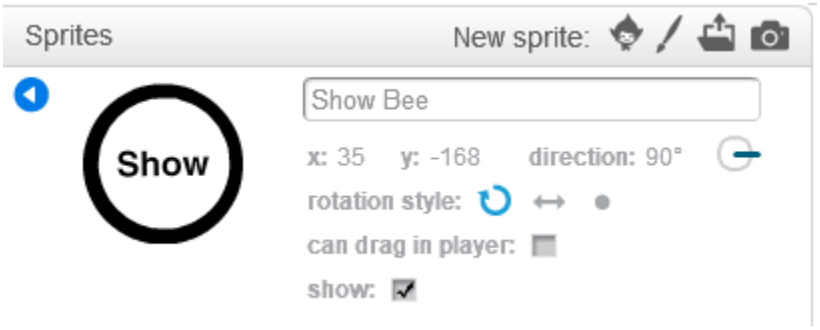


Figure 8.10

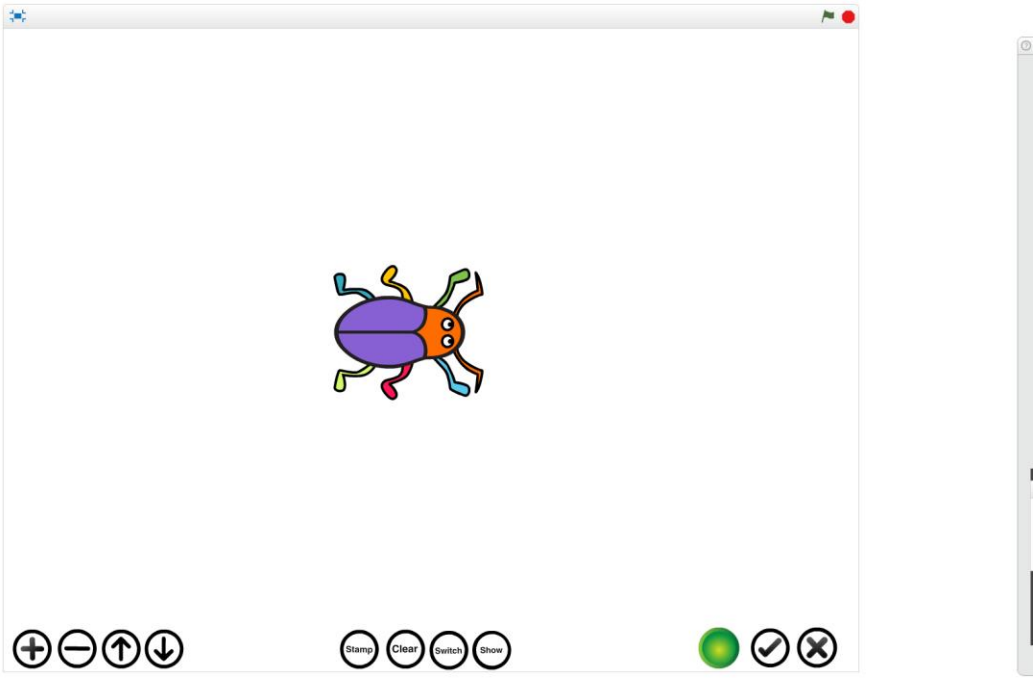


Figure 8.11

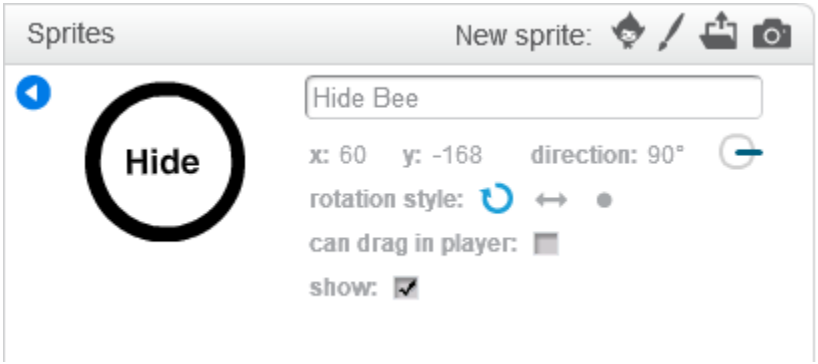


Figure 8.12

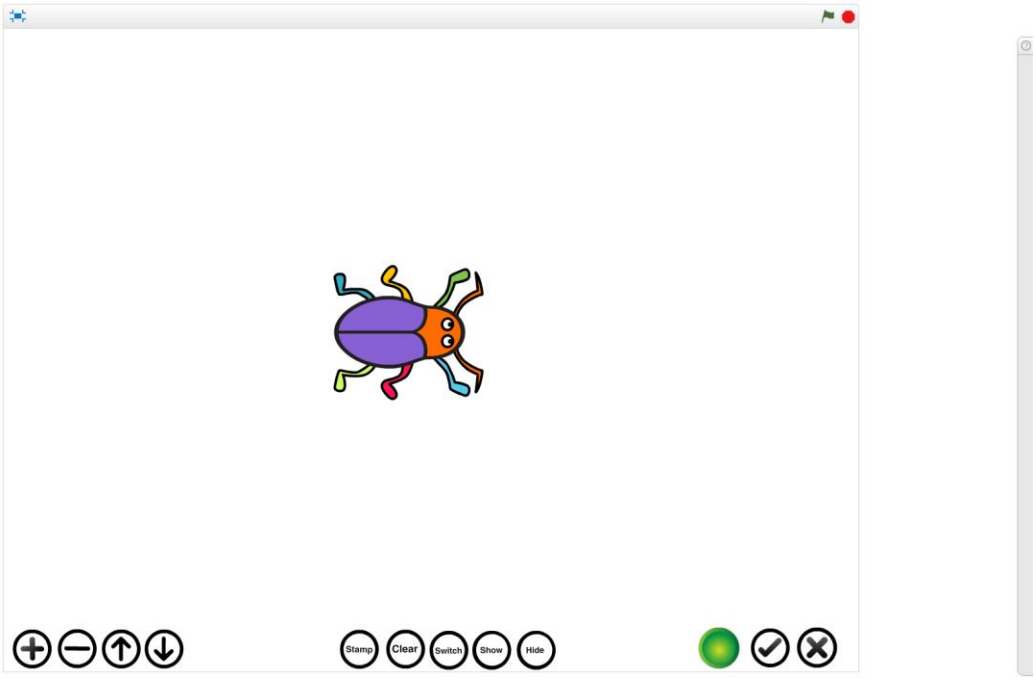


Figure 8.13

The Drawing Bee sprite also has a new costume, with the main body and head transparent, while the rest of the body remains the same as the existing costume. Duplicate the existing costume for the sprite in the Costumes Tab, and use the Paint Editor tools to modify the new costume to get the desired look.

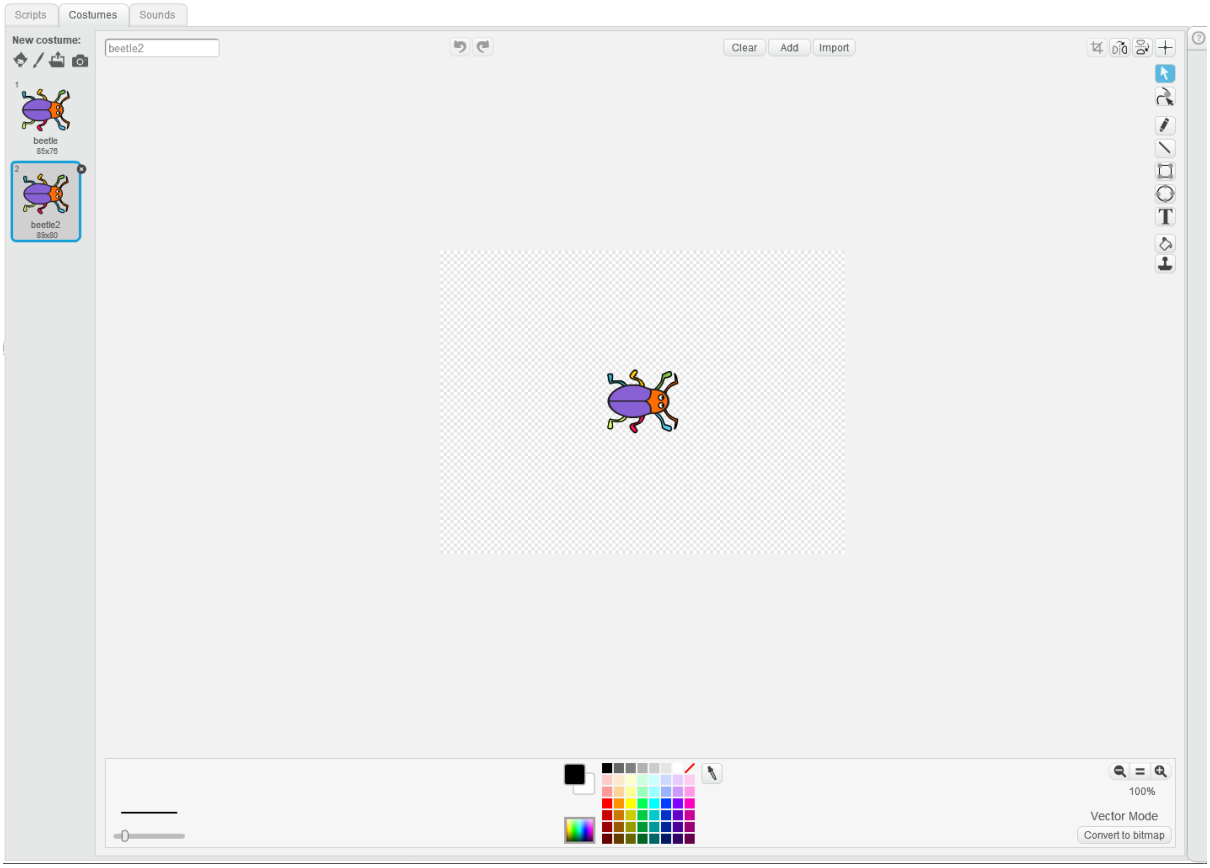


Figure 8.14

Select the image in the Paint Editor and ungroup it in order to edit the comprising parts separately. Then select the main body and head (grouped as one), pick the *color a shape* tool, select the *transparent* colour in the colour palette and apply it on the selected bee parts.

Figure 8.15

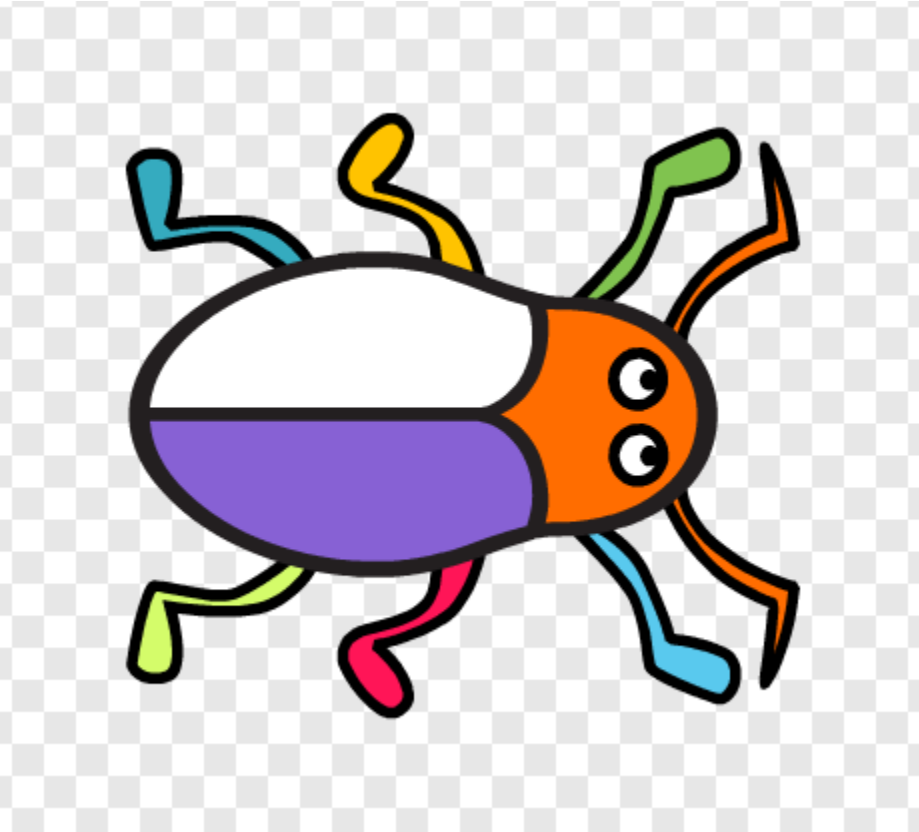


Figure 8.16

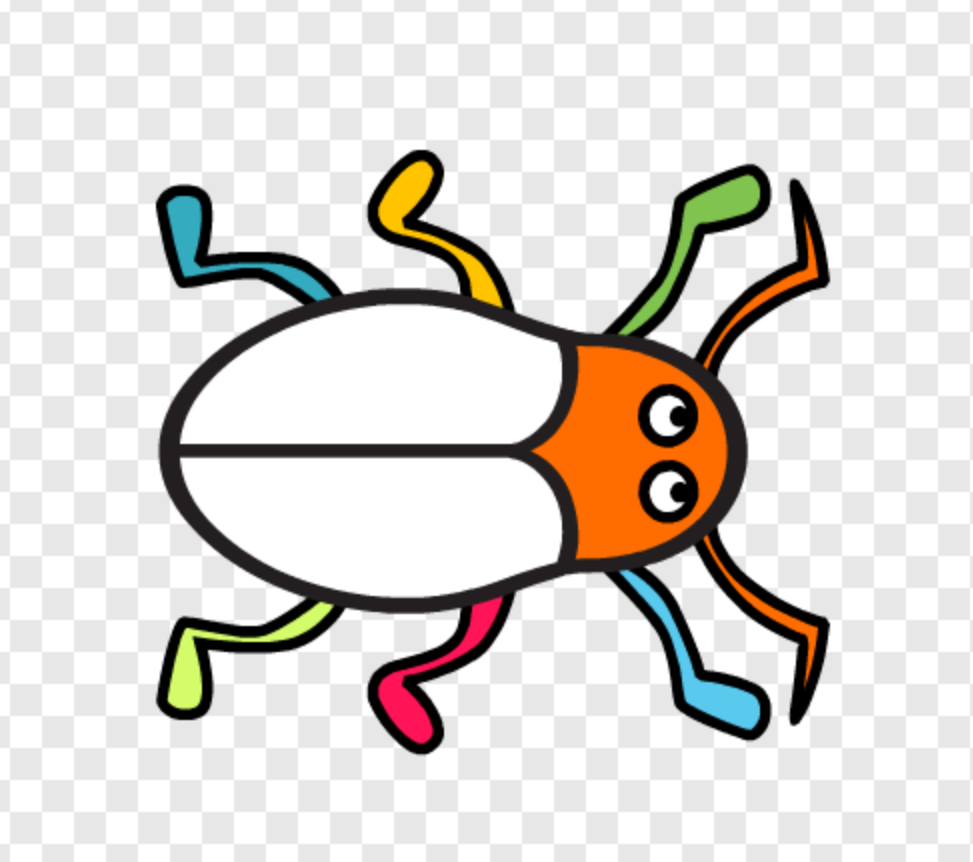


Figure 8.17

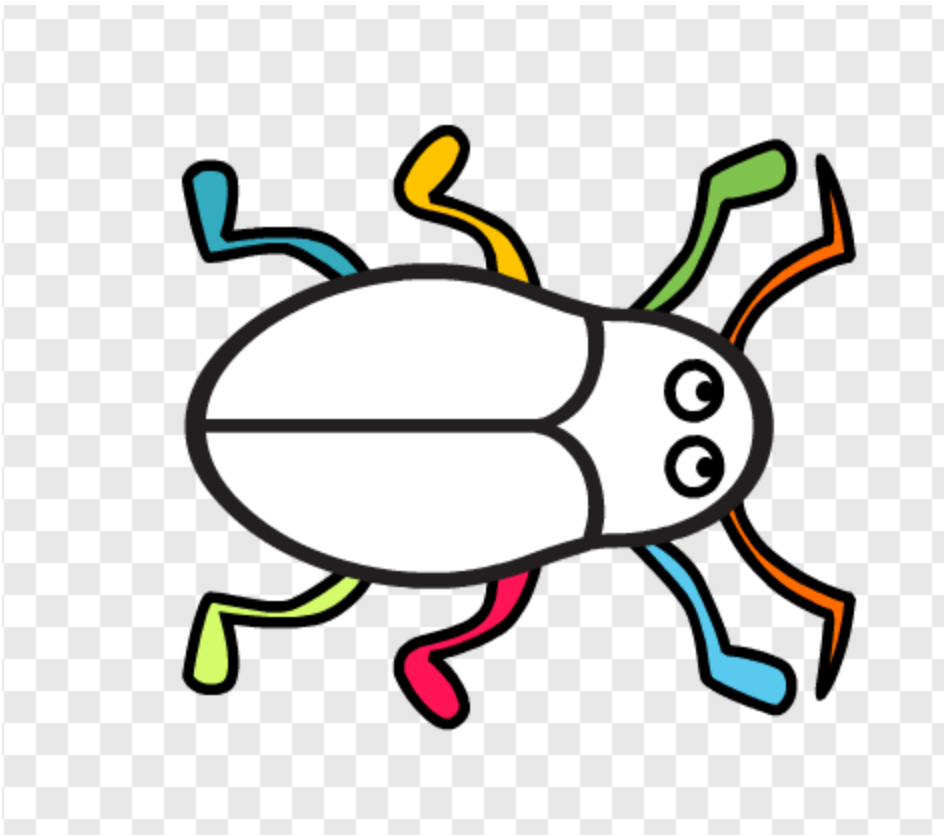


Figure 8.18

We are making these parts of the bee transparent so that the users can switch to this costume when they want to see what is beneath the bee at any time. This costume also makes it easy for the user to know the centre point of the bee, where the pen is located.

The last row on the Project Components table is for Sounds. Since all of the buttons except the one for clearing the stage are broadcasting their click events to the Drawing Bee component which then performs the required action, the sounds would be added to the Drawing Bee component. The sound for clearing the stage would be added to the *Clear Stage* button.

Add sound for the components

The first action on the components table that we need to add a sound for is moving the bee with the keyboard arrow and number keys. We will use the *pop* sound for this action. Since this sound is already present in the Drawing Bee sprite's Sounds Tab, simply rename it to *move*, in order to easily identify it when it is time to program the sprite to play the sound.

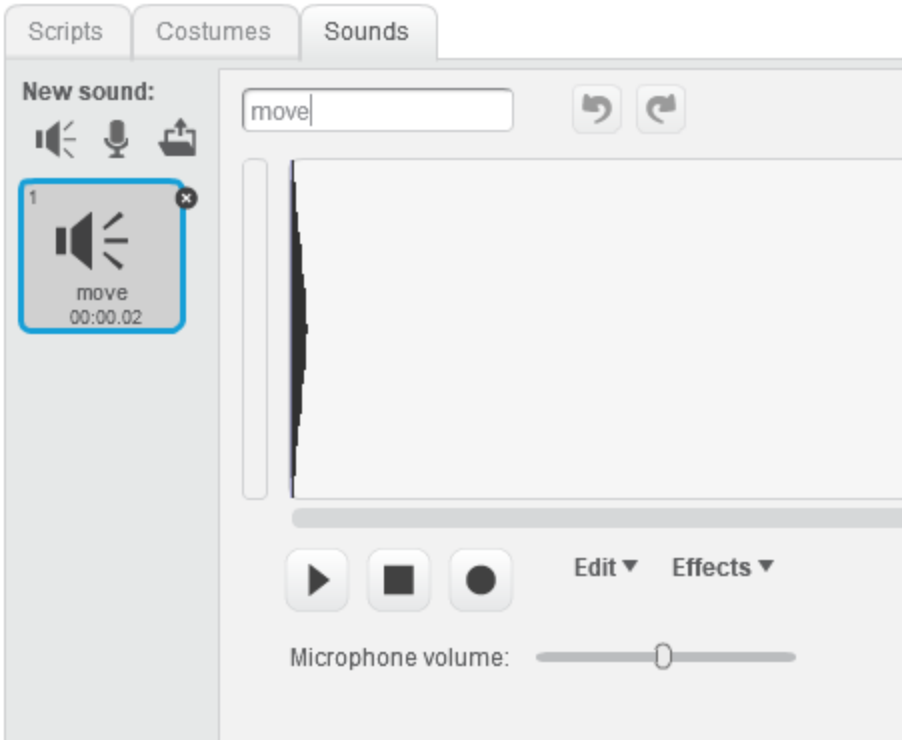


Figure 8.19

The next action is moving the bee to a point on the stage with the mouse. We will use the *zoop* sound for this action. Add the sound to the Drawing Bee sprite, and rename it to *move to point*.

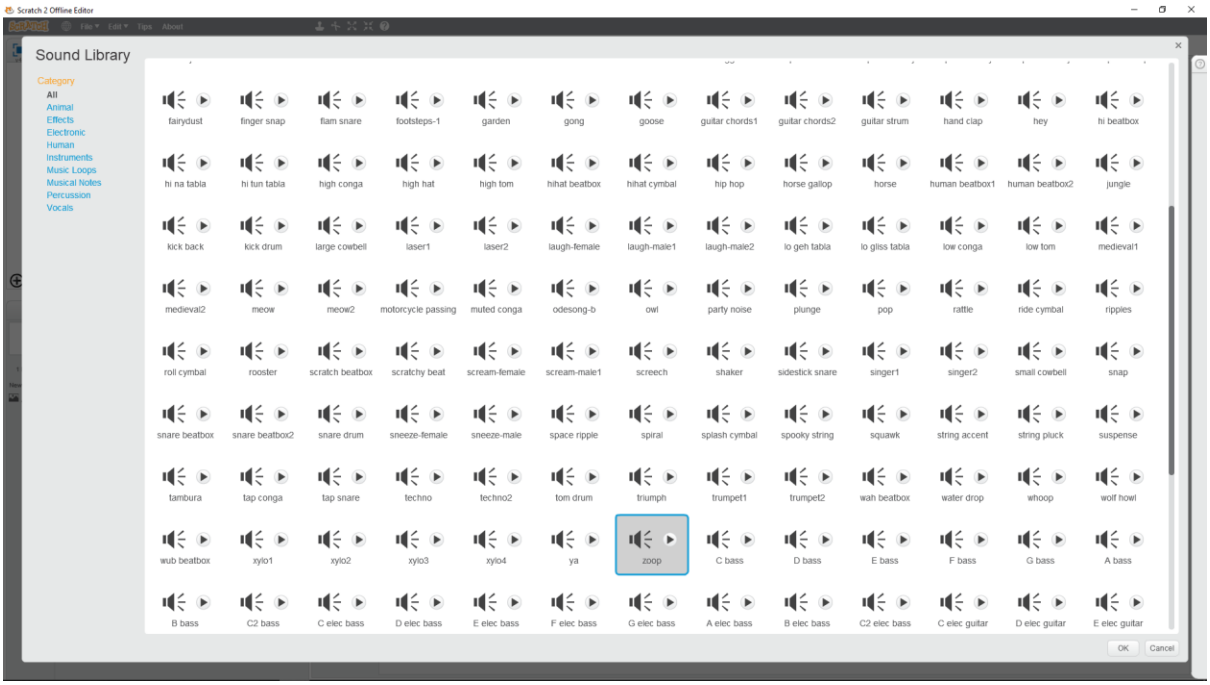


Figure 8.20

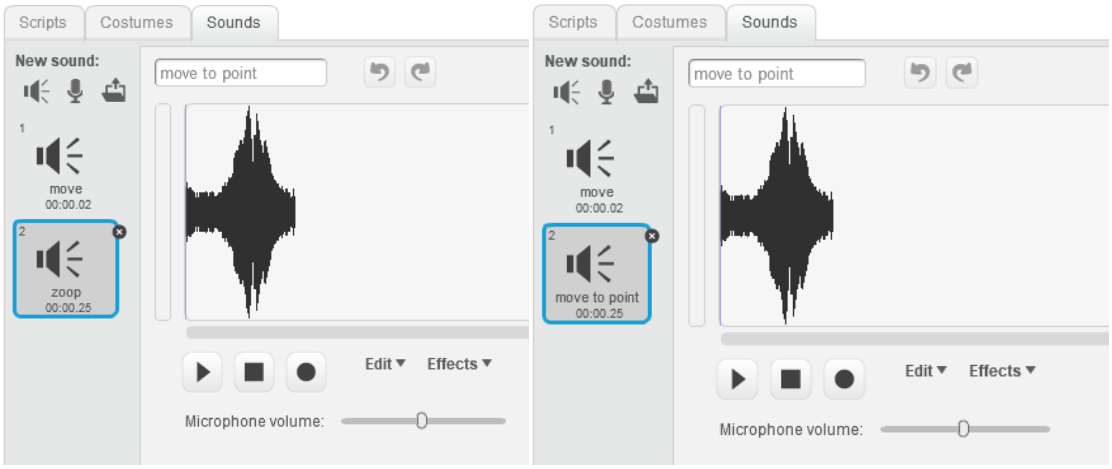


Figure 8.21

Do the same thing for the remaining actions and sounds listed below:

- Increase Pen Size - hihat beatbox sound
- Decrease Pen SIze - hihat cymbal sound
- Increase Bee Size - lo gliss tabla sound
- Decrease Bee Size - lo geh tabla sound
- Stamp Bee - rattle sound
- Switch Costume - water drop sound
- Show Bee - laser1 sound
- Hide Bee - laser2 sound
- Change Pen Colour - boing sound
- Pen Down - muted conga sound
- Pen Up - hi na tabla sound

For the Clear Stage action, add the crash beatbox sound to the Clear Stage sprite. Remember to rename each sound added to indicate the function it is used for in your application, like we did for the *move* and *move to point* sounds above.

Program the components (with blocks and scripts) to perform their functions

Here are the functions we derived from the first step of the process:

- Ability to hide and show the bee.
- Ability to switch the bee's costume.
- Ability to increase and decrease the bee's size.
- Play sounds when any function is performed on the bee, its pen, or the stage.
- Perform an animation when the bee's size is increased or decreased.

- Perform an animation when the bee is shown or hidden.
- Perform an animation when the bee is stamped.

We will tackle these functions one after the other.

1. Ability to hide and show the bee.

Like we did for the other buttons when creating the first version of the application, the Hide Bee and Show Bee buttons would broadcast messages when they are clicked, and the Drawing Bee would react to the broadcast by performing the required action.

For the Hide Bee button:

Figure 8.22

For the Show Bee button:

Figure 8.23

Do the same thing for the other new buttons: Switch Costume, Increase Bee Size, and Decrease Bee Size.

For Switch Costume:

Figure 8.24

For Increase Bee Size:

Figure 8.25

For Decrease Bee Size:

Figure 8.26

Now that all the buttons perform their main functions, let us program them to play sounds when they are clicked on. The block for playing of sounds should be placed just before the block for the main function, so that the sound will play while the function is being carried out. For this, we would use the *play sound* block.

Figure 8.27

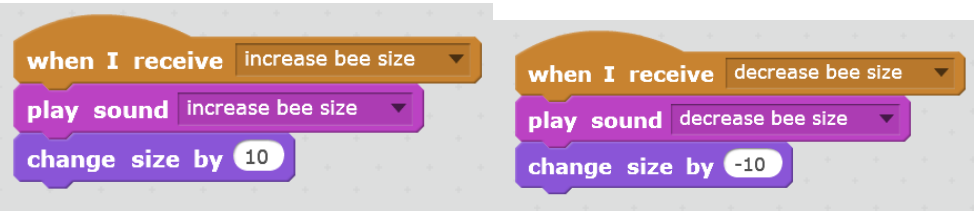


Figure 8.28

Figure 8.29

Figure 8.30

Figure 8.31

Figure 8.32

The bee should also play sounds when it is moved with the keyboard keys or the mouse

Figure 8.33

Figure 8.34

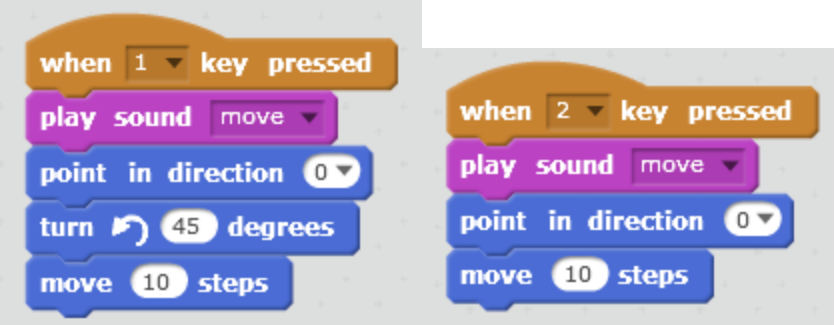


Figure 8.35

Figure 8.36

Figure 8.37

Figure 8.38

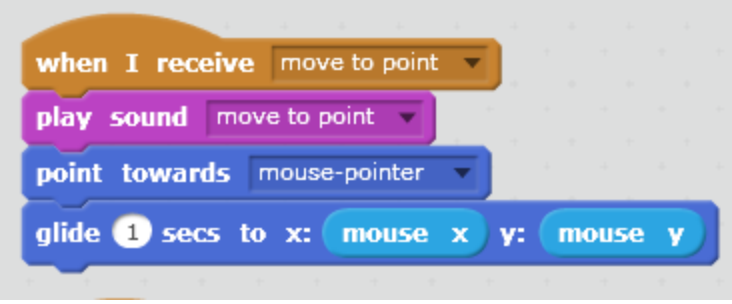


Figure 8.39

We now have all of the control buttons in the application reacting by playing sound and performing their main function. The application is great as it is, but we can make it even better by performing animations when some actions are carried out. For example, doing a whirl animation when hiding and showing the bee would make the application more exciting to use.

These are the animations we want to program into the Drawing Bee application:

- A colour animation when the bee's size is increased or decreased.
- A whirl and grow-shrink animation when the bee is shown or hidden.
- A grow-shrink animation when the bee is stamped.

We will use graphic effect blocks for the first two animations. The third animation can be accomplished with the size property setter block in the *Looks* category.

A colour animation when the bee's size is increased or decreased

For this animation, the bee would firstly play the appropriate sound and perform its main function of increasing or decreasing the bee's size. It would then change its color graphic effect by a value of 10 twice, and revert to its original colour by changing its color graphic effect by a value of -10 twice.

Add the appropriate blocks for this animation:

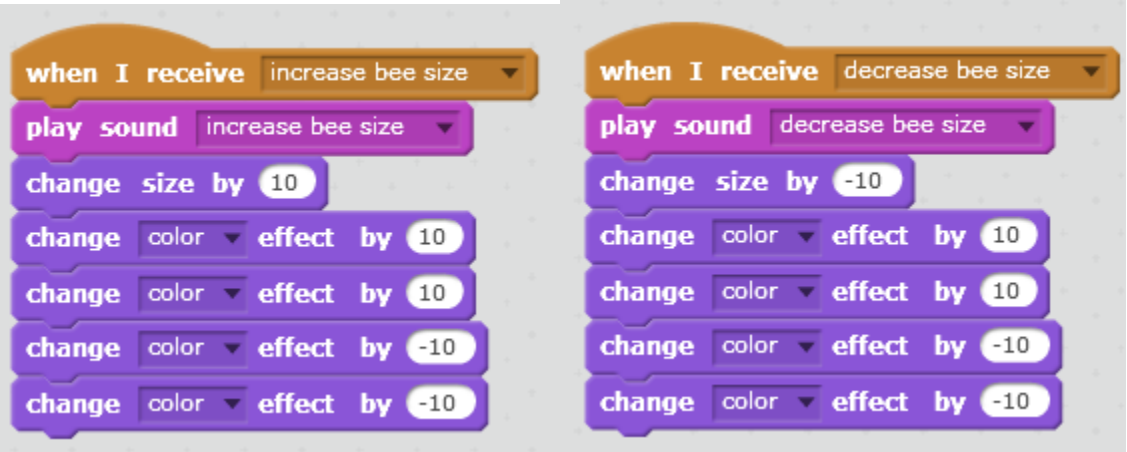


Figure 8.40

Now test this animation by clicking on the increase and decrease bee size buttons. Does the bee perform the colour animation while playing the action sound and performing the action?

Surprisingly, it does not. This is not because the animation is not correct; it is, but the blocks are executed very quickly, and since the effect of the animation is reversed at the end, the bee appears to stay the same. We need to find a way to pause a bit after the execution of each block that comprises of the animation, so that the effect(s) of the animation would be more visible.

Remember the *switch backdrop and wait* block from the stage's *Looks* blocks palette, that waits for another script to be executed before continuing its own script? We need something like that for this animation. While the *Looks* block category does not have a "change color effect and wait" block, we can use the *wait n secs* block from the *Control* blocks category, which pauses the execution of a script for the specified number of seconds, *n*.

Figure 8.41

We need the animation blocks to pause for just a tiny bit, so we would pass in 0.1 as the input parameter to the *wait n secs* block.

Figure 8.42

Now test the animation and see that it works. You can pass in a higher value as the input parameter to the *wait n secs* blocks if you want the animation to last longer.

A whirl and grow-shrink animation when the bee is shown or hidden

For this animation, the bee would whirl in the clockwise (positive) direction and shrink before finally disappearing when the “Hide Bee” button is clicked, and the bee would whirl in the anti-clockwise (negative) direction and grow before finally disappearing when the “Show Bee” button is clicked.

Add the appropriate blocks for this animation:

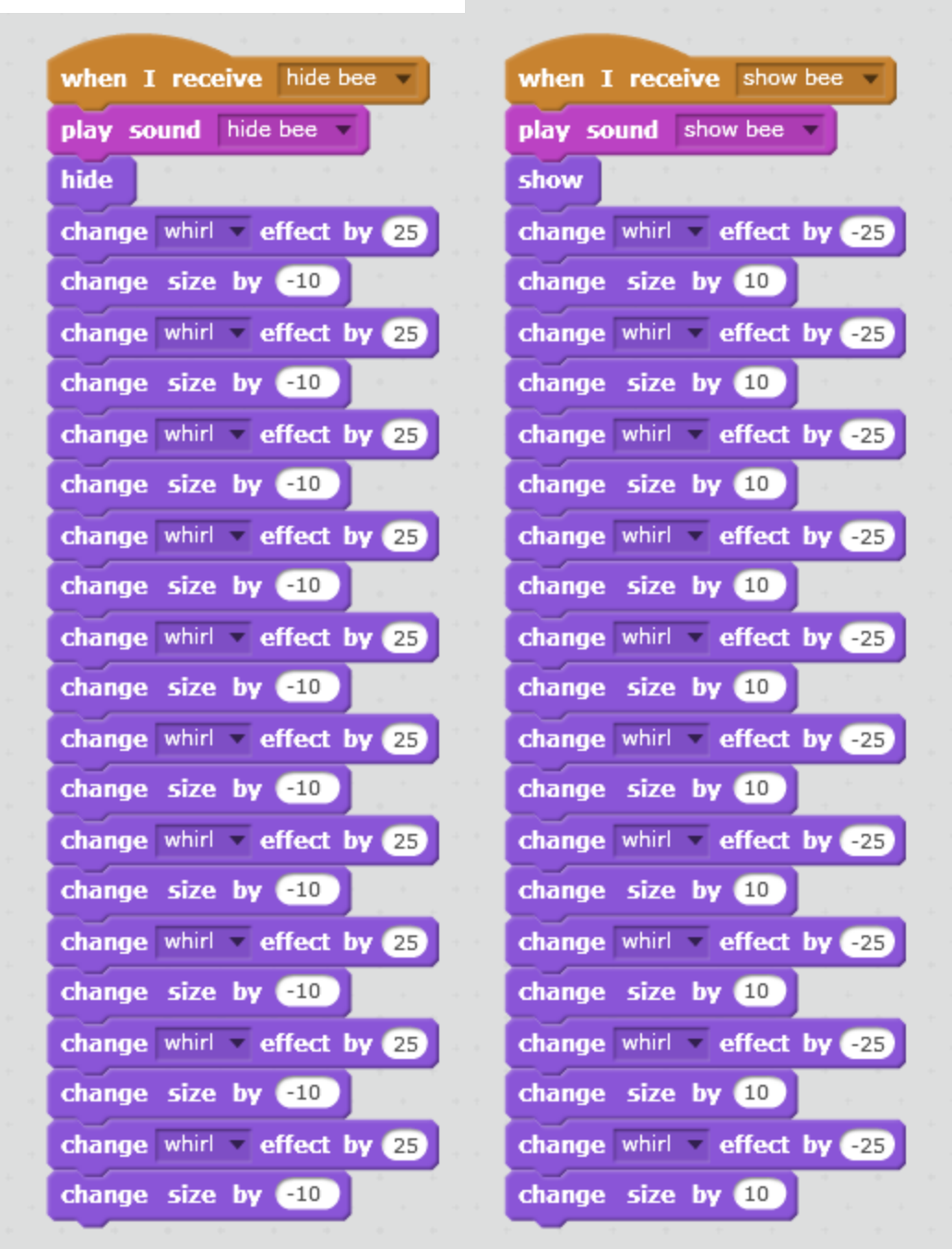


Figure 8.43

Test the block by clicking the *Hide Bee* and *Show Bee* buttons. Once again, the main functions of hiding and showing the bee happen without the animation happening visibly. Follow the steps we took above to fix the issue, this time passing 0.01 into the *wait n secs* block as the delay needs to be very short so that the whirling and grow/shrink animations can happen in quick successions.

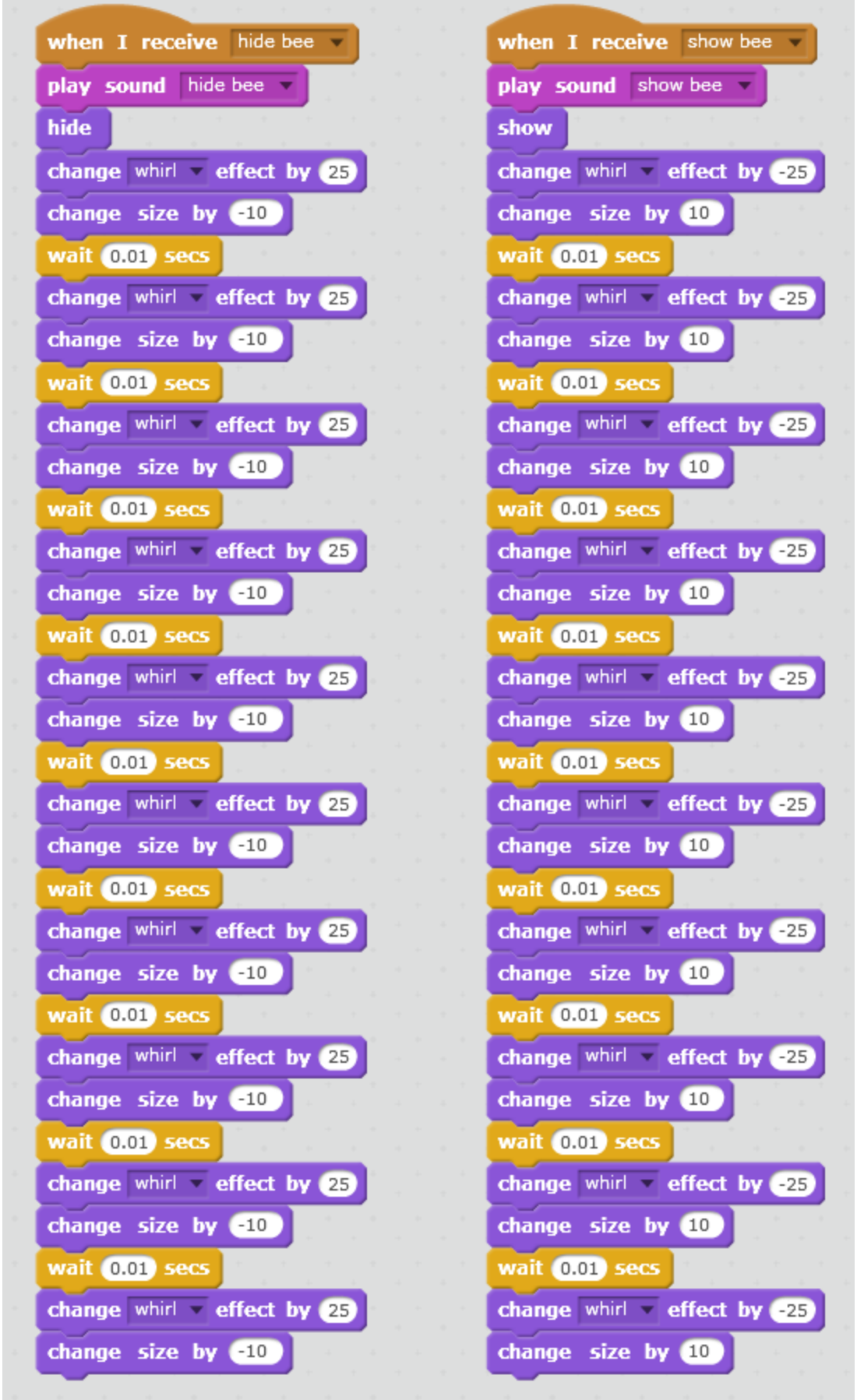

when I receive hide bee
play sound hide bee
hide
change whirl effect by 25
change size by -10
wait 0.01 secs
change whirl effect by 25
change size by -10
wait 0.01 secs
change whirl effect by 25
change size by -10
wait 0.01 secs
change whirl effect by 25
change size by -10
wait 0.01 secs
change whirl effect by 25
change size by -10
wait 0.01 secs
change whirl effect by 25
change size by -10
wait 0.01 secs
change whirl effect by 25
change size by -10
wait 0.01 secs
change whirl effect by 25
change size by -10
wait 0.01 secs
change whirl effect by 25
change size by -10
wait 0.01 secs
change whirl effect by 25
change size by -10
when I receive show bee
play sound show bee
show
change whirl effect by -25
change size by 10
wait 0.01 secs
change whirl effect by -25
change size by 10
wait 0.01 secs
change whirl effect by -25
change size by 10
wait 0.01 secs
change whirl effect by -25
change size by 10
wait 0.01 secs
change whirl effect by -25
change size by 10
wait 0.01 secs
change whirl effect by -25
change size by 10
wait 0.01 secs
change whirl effect by -25
change size by 10
wait 0.01 secs
change whirl effect by -25
change size by 10

Figure 8.44

The Show Bee animation works as expected, but clicking on the *Hide Bee* button hides the bee immediately without any visible animation. This is because the *hide* block is placed before the animation blocks, so the bee is hidden and the animation performed in the hidden state. Move the *hide* block below the animation blocks.

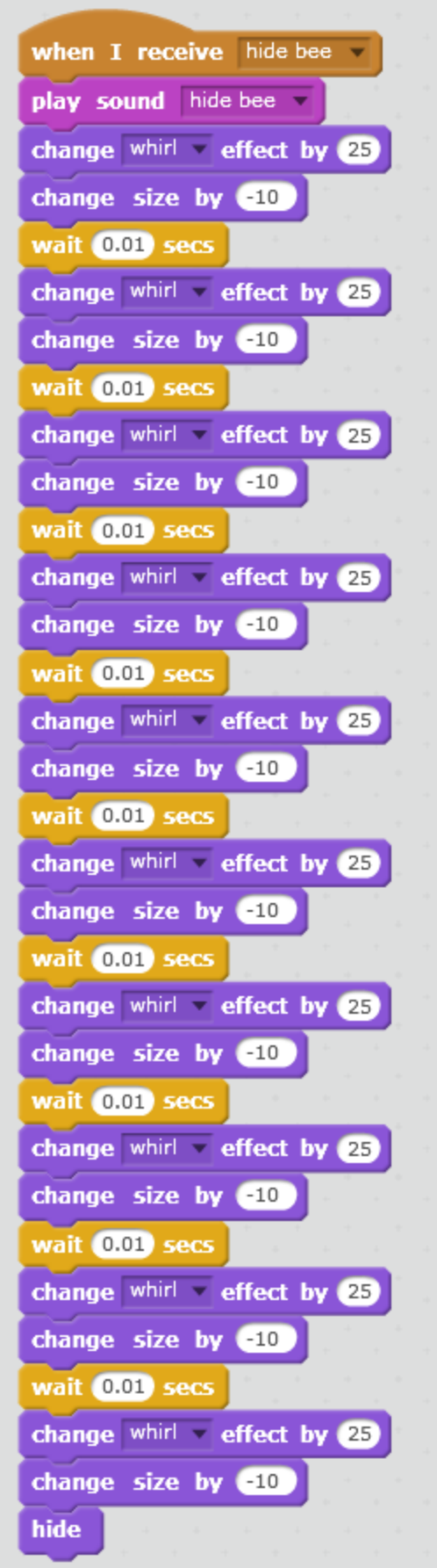


Figure 8.45

The animation now works as expected.

A grow-shrink animation when the bee is stamped

For this animation, we want the bee to play its sound, stamp itself on the stage, grow to a particular size and shrink to its original size when the *Stamp Bee* button is clicked. The first two actions have been defined; let us now add the appropriate blocks to perform the last two actions as animations.

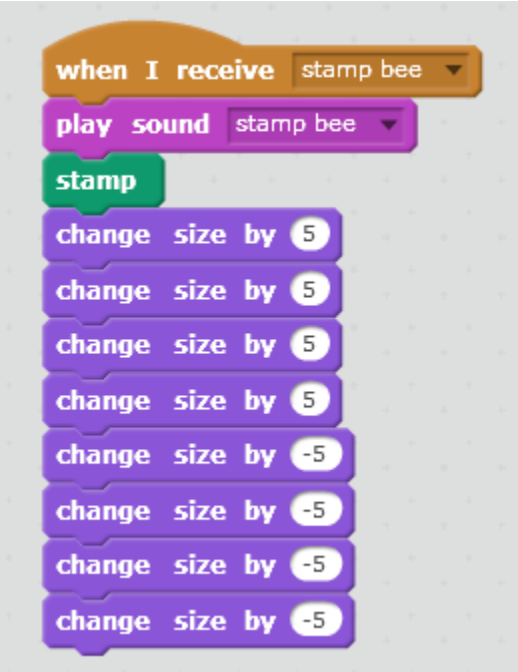


Figure 8.46

Since the effect of the animation is reversed at the end, the bee appears to stay the same. Use the wait blocks to resolve this issue.

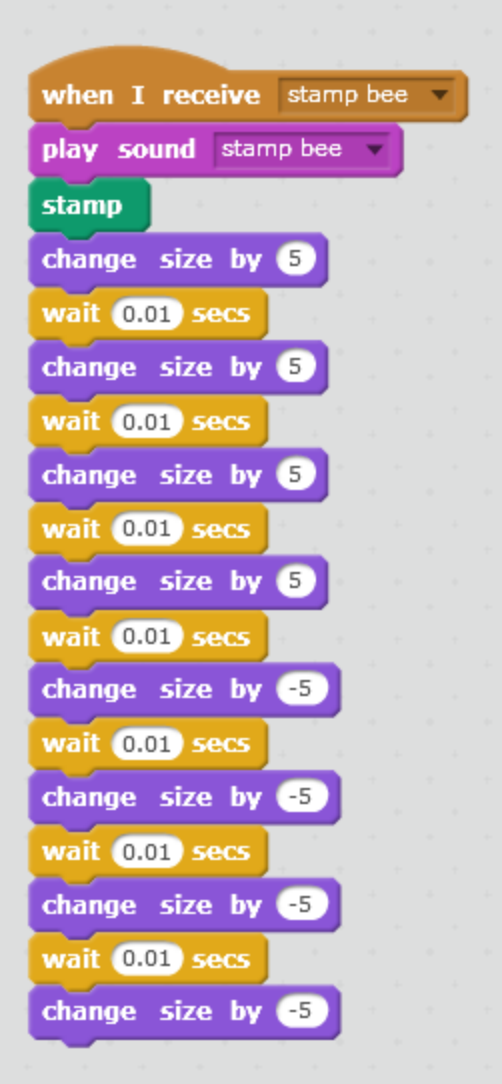


Figure 8.47

The animation now works perfectly.

Summary of Chapter Eight

In this chapter, we modified the Drawing Bee application built in the fifth chapter to enhance its user experience and functionality using sounds and animations. We added more control buttons to the app, and used blocks from the *Sounds* category to play sound effects when buttons are clicked. We also used blocks from the *Looks* category to perform animations when certain actions are performed. When we ran into an issue with very fast animations, we used the *wait for n secs* block from the *Control* category to insert pauses between fast animations so that they can happen visibly.

This app still has some hiccups. Clicking on the *Show Bee* button performs the whirling animation even when the bee is shown, resulting in an out-of-shape bee to perform our drawings. The same goes for the

Hide Bee button, and the colour animation performed by the *Increase Bee Size* and *Decrease Bee Size* buttons sometimes leaves the bee with a colour different from the original, when they are pressed many times in quick successions.

We will learn how to resolve issues like this, as well as introduce a whole new dimension to the apps we can build with Program Control and Decision Making in the next two chapters.

CHAPTER 9

Introduction to Data Processing

Table of Contents

Learning Objectives

By the end of this chapter, the student would:

- Understand what data is and how to represent data in different forms in Scratch
- Understand what Variables are and how to use variables to store information in Scratch
- How to manipulate and perform operations on data to produce new data.

Chapter Prerequisites

- Scratch 2 Offline Editor set up on the computer with access to local storage.

What is Data?

When it comes to programming components in an application with blocks, you would need to manipulate data in one form or another. Data is passed as input parameters to blocks like *move n steps* to control the distance moved by a sprite on the stage, *say for n secs* block to control both the text being spoken, and for how long the speech bubble appears on the stage.

Data can also be in the form of component properties like a sprite's *x position* property getter block in the *Motion* block category which holds the current x-coordinate of a sprite on the stage, or the *volume* property getter block in the *Sound* block category which holds the current volume of the sounds played by the stage or a sprite. All of these things are data.

As we have seen, data can be passed either literally as text - as we have done with *say* and *think* blocks - or numbers - as we have done with the *move to* and *glide to* blocks, or as property blocks that hold these data, as we did for the *move to point* event in the Drawing Bee application:

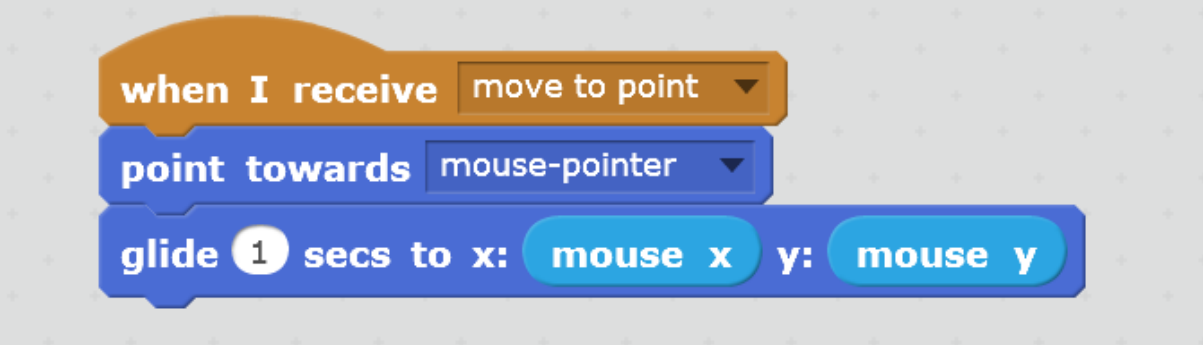


Figure 9.1

These different forms in which data can be presented and used are called **data types**.

Data Types in Scratch

There are three data types in Scratch:

- Numbers
- Strings
- Booleans

Numbers are any type of data that can be used to represent numeric information. They can be either whole numbers or decimals, and positive numbers or negative numbers.

Strings are any type of data that represents text in an application. The words you speak, the text you type, the things you think are all strings.

Booleans are the simplest data types in Scratch. A Boolean holds either of two values: true or false. Data of this type is mostly used to test conditions and make decisions based on the result of these conditions in a program.

We have not used booleans so far in the applications we have built in this textbook, but we used something related to booleans when we used the *if on edge, bounce* block in the Patrol Wizard application. The block says *if on edge*, which causes the question *is the sprite on edge?* to be asked. The answer to this question would be *true* (yes) or *false* (no), and depending on the answer, the sprite would either bounce or keep moving. We will learn more about testing conditions and programming applications to make decisions in the next chapter.

Every operation we have performed so far on data has been about retrieving, modifying and making use of data, but we do not know how these data are stored. Storing of data is an important part of an application, and so far, Scratch applications have helped us with automatic storage of data for properties like *x position, volume, size*, etc, in its memory. To build more complex applications that perform sophisticated functions, you need to be able to manually program applications to store, retrieve, and modify data in their memory. This can be done with the aid of variables.

Variables

A variable is a portion of an application's memory that can store data. An app's memory can be likened to a room used to store boxes, and each box can be used to store different types of items. Each box that can be stored in this room is identical, so there needs to be a way to identify a box for easy reference when the item in the box is needed. Hence, each box has to be tagged with a unique name, which is used to refer to the box in the storage room when the item it stores needs to be retrieved or modified.

In the same way, variables are used to store data for easy reference when the data is needed. A variable can store data of any type, and a variable must have a name which is used to identify it when the data it stores needs to be retrieved or modified.

We have used variables in form of component property blocks; a component property block has a name - for example, the *volume* block in the Sound category has the name, *volume* - and stores data related to its name - the *volume* block stores the current volume level of the sounds played by a component, which has a number data type. We use the *set volume to* block to modify the data stored in the property, and the *volume* block to retrieve and use the data stored in the property.

Creating a Variable

The *Data* block category hold different blocks for manipulating variables in Scratch. However, there are no visible blocks the first time you switch open the *Data* blocks palette.

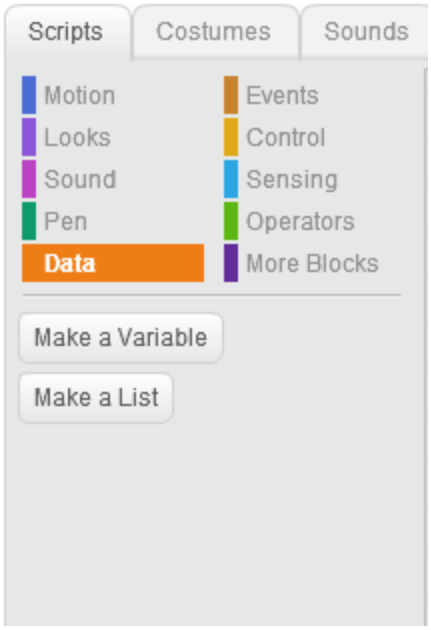


Figure 9.2

This is because you do not currently have any variables in your program. To create a variable, click on the *Make a Variable* button.

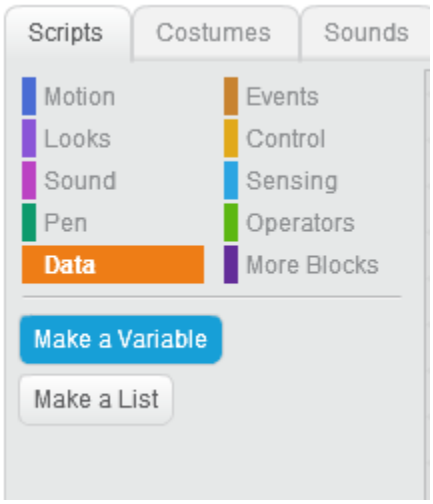


Figure 9.3

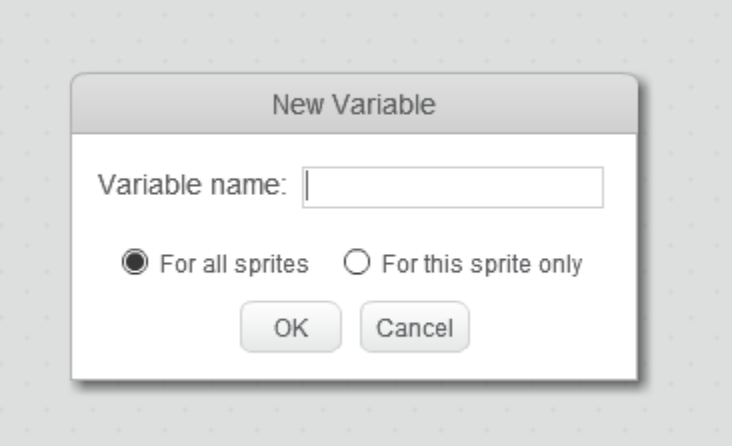


Figure 9.4

As noted earlier, a variable needs a name for identification and reference purposes, and the name has to be unique. Enter a desired name in the *Variable name:* box, and click on the *OK* button to create a new variable. For example, if we wanted to keep track of the player's scores in a game, we would create a variable named *score*.

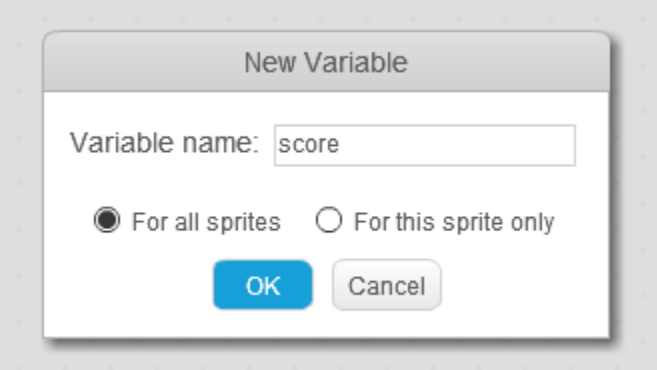


*Figure 9.5: We will learn what the **For all sprites** and **For this sprite only** options mean in the next section.*

Clicking on the *OK* button creates the variable and adds blocks for manipulating the variable to the *Data* blocks palette.

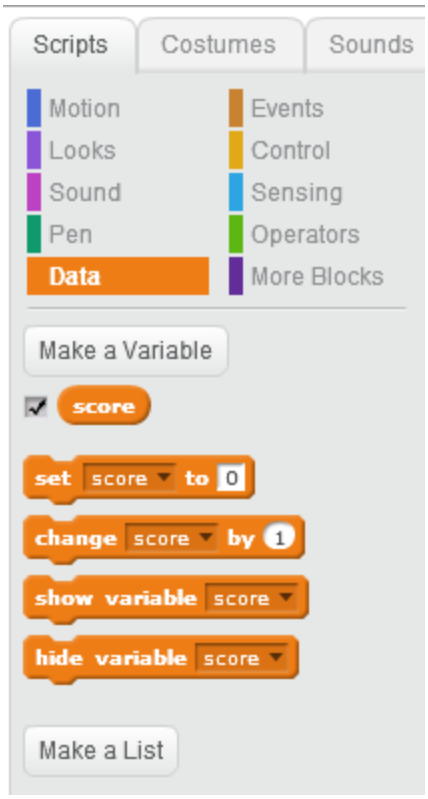


Figure 9.6

Modifying a Variable

Like most component properties that we have manipulated using blocks so far in this textbook, variables can be modified using two types of blocks: *change* and *set* blocks.

The *set variable to* block

Figure 9.7

This block is used to directly set the value of a variable to the data passed as an input parameter to the block. Using this block replaces the existing value of the variable with the new value passed as an input parameter.

The *change variable by n* block

Figure 9.8

This block is used to increase (for positive values of n) or decrease (for negative values of n) the value of a variable by a number, *n* passed as an input parameter. It is important to note that this block accepts only numbers as an input parameter and is suitable only for variables whose values are of the number data type, as it attempts to perform an addition or subtraction on the existing value of the variable with the new value.

For example, if the *score* variable had a value of 4 and the *change variable by n* block was executed with 5 passed as an input parameter, the new value of the *score* variable would be 9 (4 + 5 = 9). If -5 was passed as an input parameter instead, the new value of the variable would be -1 (4 + 5 = 4 - 5 = -1).

If, however, we had a *username* variable which is used to store the name of the user of an application and its value was "Alibaba", executing a *change username by 4* block would result in the variable having a new value "NaN", which means Not a Number. This is because the application would attempt to perform the operation "Alibaba" + 4, which is not a valid arithmetic operation and results in a value that is not a number.

Variable Scope

The box that appeared above when you clicked on the *Make a Variable* button in the *Data* blocks palette has two options: *For all sprites* and *For this sprite only*.

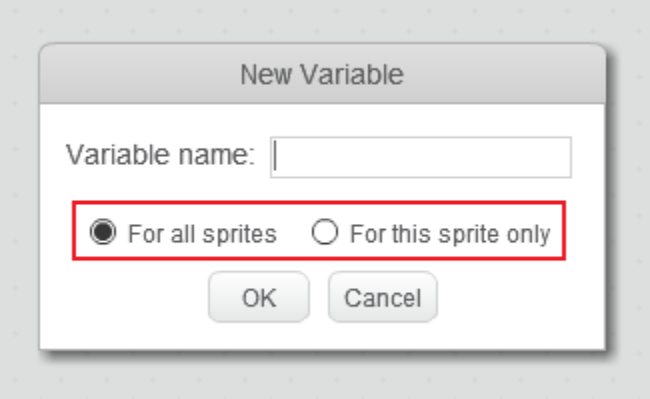


Figure 9.9

We have seen in past chapters that the stage and each sprite has to be programmed separately with blocks, and a particular component's blocks can not be accessed by other component's blocks. When it

comes to variables, the stage is seen as the master container for all other sprites, such that when you try to create a variable with the *Make a Variable* button when a sprite is selected in the Sprite List, you see two options: *For all sprites* and *For this sprite only*.

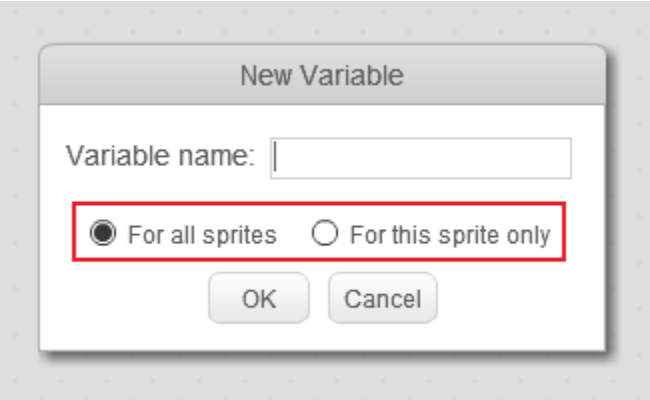


Figure 9.10

But when you try to create a variable when the stage is selected, you see just one option: *For all sprites*.

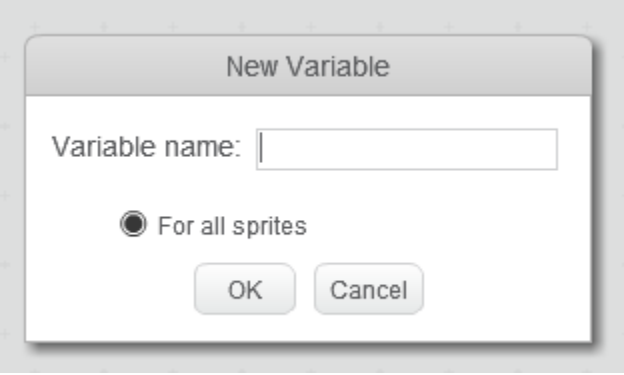


Figure 9.11

This is because a variable created on the stage is available to all sprites on the stage (which is, by extension, all sprites in an application) for retrieval and modification, while a variable created on a sprite can be limited to just that sprite, such that other sprites can retrieve and use the value of that variable, but only the sprite on which it is created can modify its value. This concept is known as *variable scope*.

There are two types of variable scope: *global* and *local*. When a variable is created with the *For all sprites* option, it is said to have a *global scope* and hence can be called a global variable, while creating a variable with the *For this sprite only* option gives it a *local scope,* and that variable can be called a local variable.

The difference between global scope and local scope is that the value of a variable with global scope can be retrieved and modified by every sprite and the stage in an application, but the value of a variable with local scope can also be retrieved by every sprite and the stage in an application, but can only be modified by the sprite on which the variable was created. We will see how this works in subsequent sections.

Variable Monitors

Like component property blocks, variables also have value getter blocks that can be used to retrieve the current value of the variable. This block is created alongside other blocks when a variable is created using the *Make a Variable* button in the *Data* block category.

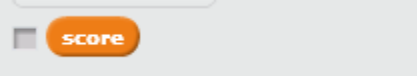


Figure 9.12

Clicking on this block right there in the blocks palette will show the current value of the variable.

Figure 9.13

This block can also be used to track the value of a variable on the stage in real-time, using variable monitors. To show a variable monitor, tick the checkbox beside the variable's value getter block in the *Data* blocks palette.

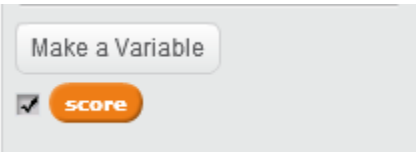


Figure 9.14

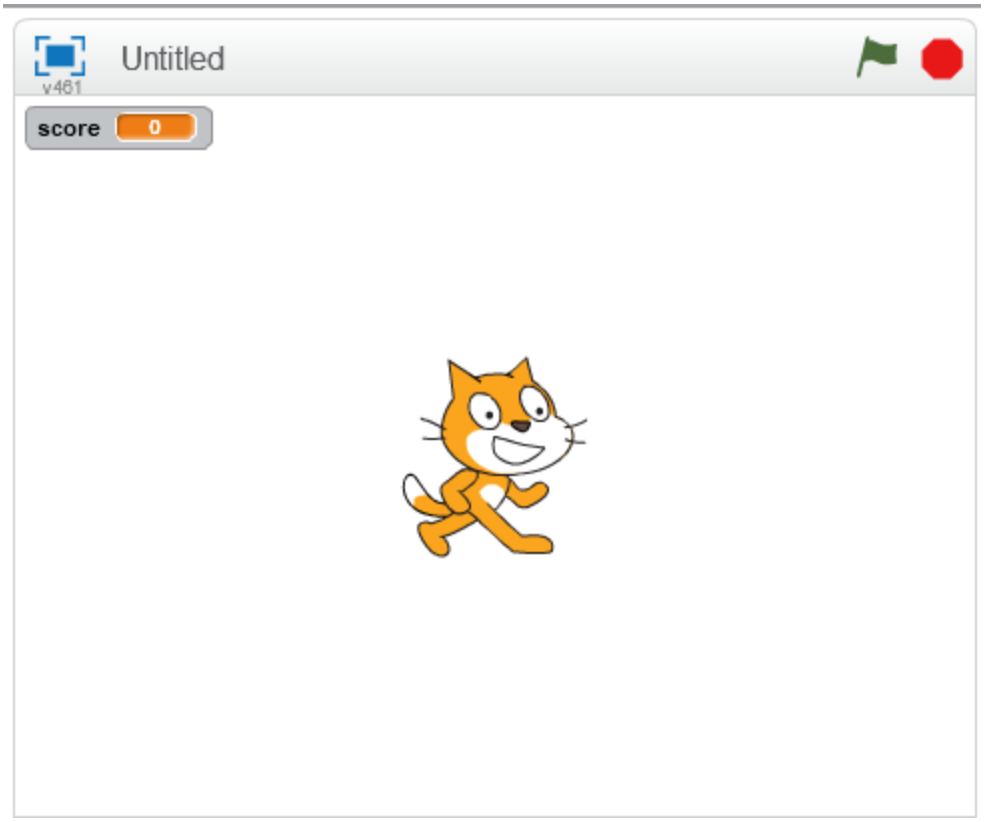


Figure 9.15

A variable monitor can also be shown or hidden programmatically on the stage using the *show variable* and *hide variable* blocks.

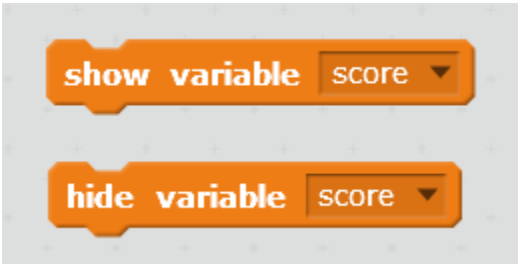


Figure 9.16

We have seen the different ways in which data about components (sprites and the stage) can be created automatically by an application through property blocks, and how to manually program an application to store data in its memory using variables. Scratch offers ways to work with data that is not directly tied to a particular component through blocks in the *Sensing* block category.

The *Sensing* block category

This block category provides blocks that allows us to program very useful functionality into our applications. Recall that when building the Drawing Bee application, we were able to program the bee to move to a point on the stage when that point is clicked, with the *mouse x* and *mouse y* blocks from the *Sensing* category passed as input parameters to the *glide* block from the *Motion* category.

One of the most useful applications of the *Sensing* block category, is getting input from users in an application.

Getting input from users

Many applications require the user to input one form of information or another. For example, your application might need to display the player's name along with the final score at the end of a game, as such: "Your score is [insert score], [insert name]". Therefore, you would need a way to ask the user to input their name at the beginning of the game, store it in a variable, and then retrieve the name from the variable when the time comes to display it. You can get input from the user with the *ask and wait* block in the *Sensing* category.

Figure 9.17

The *ask and wait* block takes in a string as an input parameter, and displays a text box, into which the user of an application can enter any text, at the bottom of the screen when it is executed. The block behaves differently when executed on the stage and on a sprite.

When it is executed on the stage, the string passed into the block is shown above the input box at the bottom of the stage.

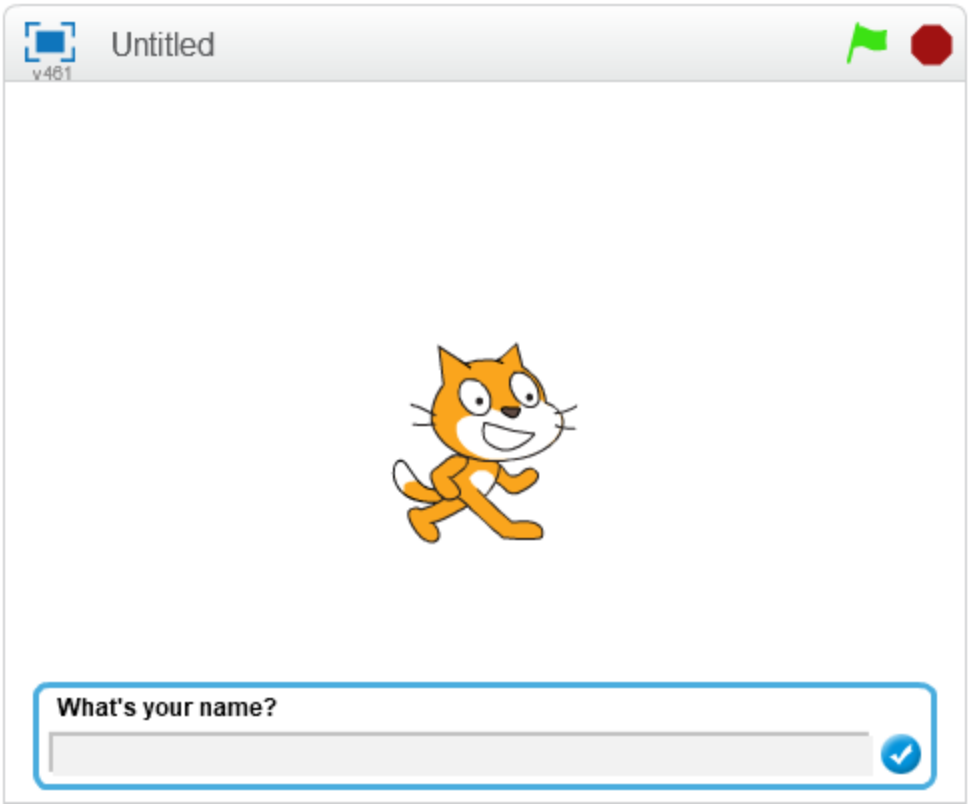


Figure 9.18

When the block is executed on a sprite, the sprite displays the string passed as an input parameter to the block in a speech bubble above it, and displays the input box at the bottom of the stage.

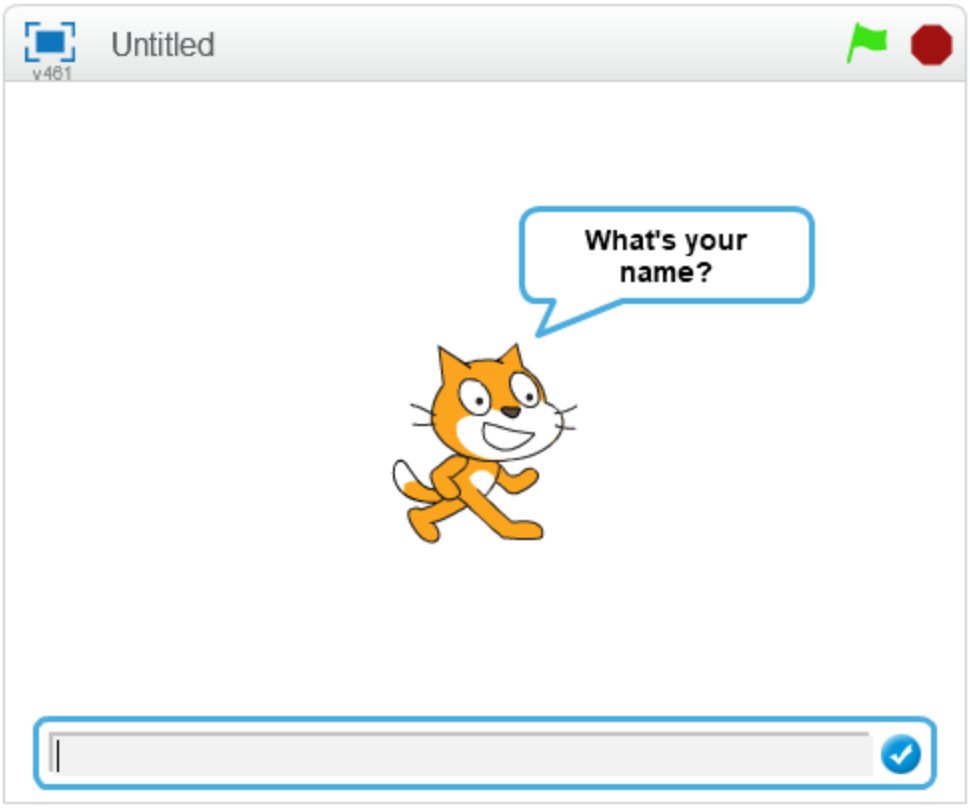


Figure 9.19

Notice that the input box is displayed alongside a done (✓) button. When the *ask and wait* block is executed, it pauses the script and waits for the user to manually click the *done* button before resuming the script. When the *done* button is clicked, the program saves the text entered into the input box in a special block called *answer*, in the *Sensing* category.

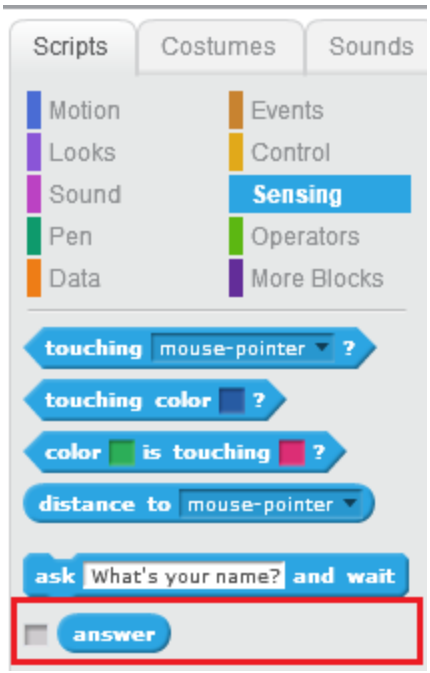


Figure 9.20

This block can be used to retrieve the input from the user, for use in the application. A monitor can be displayed for this block on the stage by ticking the checkbox next to it in the *Sensing* blocks palette.

Figure 9.21

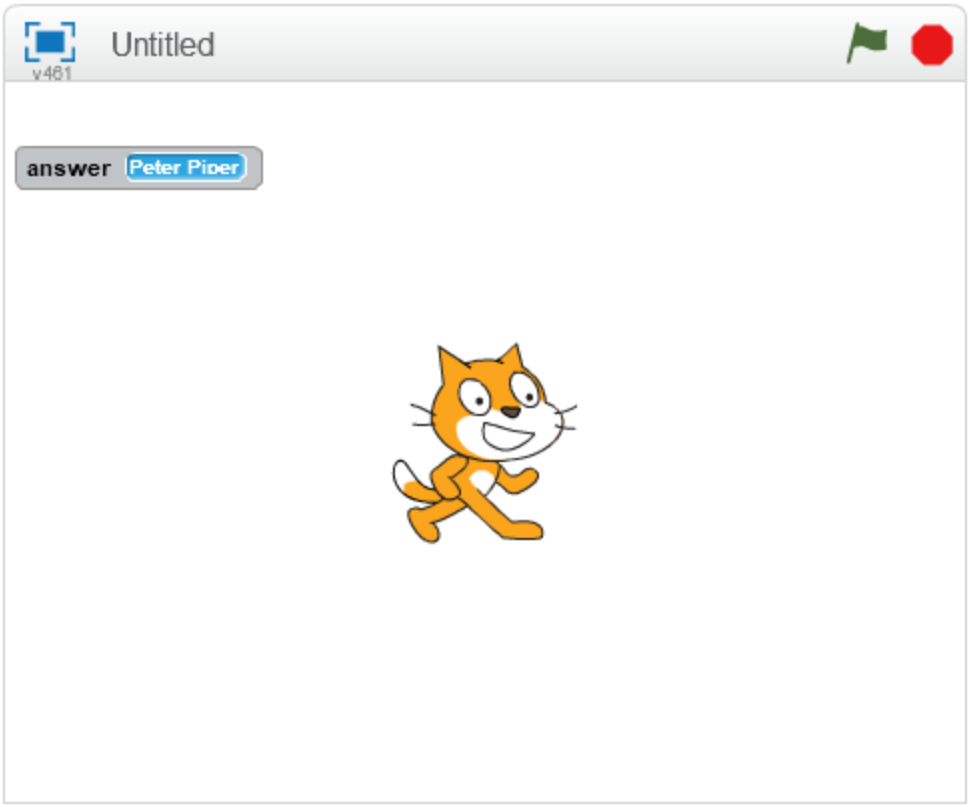


Figure 9.22

Conversion of Data Types in Scratch

Some blocks allow you to pass data of any type as input parameters to them, while some enforce a particular data type. For example, the *move 10 steps* block in the Motion category allows you to input only numbers, while the *say* blocks in the Looks category allow you to enter both strings and numbers.

However, data type enforcement works only when the data is being entered literally into the input boxes of a block; you can pass data blocks (property getter and variable value getter blocks) with values of any data type into any block's input parameter. What then happens, if you pass the *answer* block in *Figure 9.21* into the *move n steps* block in the Motion category?

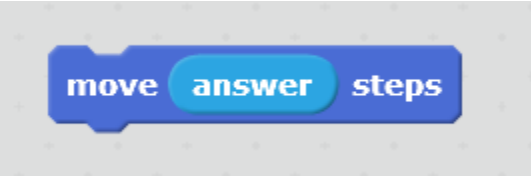


Figure 9.23

The value of the *answer* block is “Peter Piper”, which is of the string data type, and the *move n steps* block can only process the number data type. What happens in a situation like this is that Scratch tries to automatically convert the data from the string type to the number type.

In the example above, the conversion fails and the execution of the *move n steps* block does not move the sprite. If, however, the value of the *answer* block is a number string, say “150”, Scratch would be able to successfully convert this string to its number data type equivalent, 150, and use it to move the sprite 150 steps when the *move n steps* block is executed.

This makes it possible to program some interesting interactions into our applications. For example, you can use the *ask and wait* block to ask the user how many steps they would like the sprite to move, and then use the *answer block* to program the sprite to move the number of steps the user enters. Scratch automatically handles the conversion of data from one type to another.

Other Useful Data Blocks in the Sensing Category

The *touching object?* Block

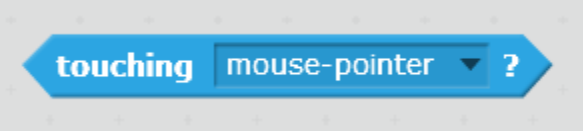


Figure 9.24

This is a reporter block that reports data of the boolean type. This block is used to detect when a sprite is touching an object or another sprite on the stage.

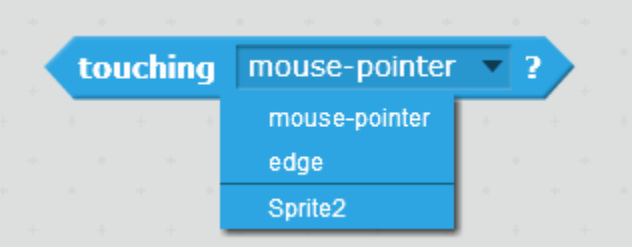


Figure 9.25

It can be used to detect when a sprite is touching the mouse pointer, or the edge of the stage. It reports either true or false depending on the validity of the statement at the point of its execution.

The *touching color?* Block

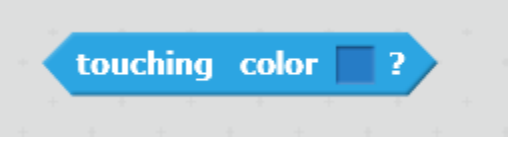


Figure 9.26

This block reports true if a sprite is touching the colour specified in the colour input parameter at the point of its execution, or false if otherwise.

The *color is touching* block

Figure 9.27

This block is used to detect when a colour is touching another colour on the stage.

The *distance to object* block

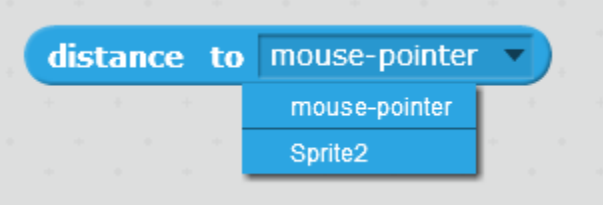


Figure 9.28

This block is used to report the distance of a sprite from another sprite or the mouse pointer on the stage. It reports data of the number data type.

The *key pressed* block

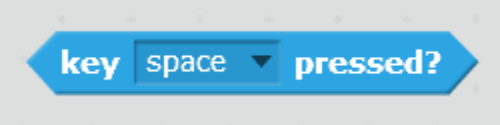


Figure 9.29

This block is used to detect whether a particular key on the keyboard is being pressed at the point of its execution.

The *mouse down* block

Figure 9.30

This block is used to detect whether the mouse button is pressed at the point of its execution.

The *mouse x/y* blocks

Figure 9.31

These blocks are used to report the x and y coordinates of the mouse pointer on the stage at any time.

The *timer* blocks

Figure 9.32

The Scratch Editor starts a timer whenever it is opened. The *timer* property getter block is used to retrieve the current value of the timer (in seconds), and the *reset timer* block is used to start the timer all over from zero.

The component property block

Figure 9.33

This block is used to retrieve the value of properties and variables belonging to the stage or a sprite in an application. It accepts two input parameters, both chosen from dropdowns; the first input parameter specifies the property or variable whose value is to be retrieved, while the second input parameter specifies the component whose property's value is to be retrieved.

Figure 9.34

When local variables are present in the component selected as the second input parameter, they are listed below the properties for that component in the dropdown for the first input parameter, separated by a black horizontal line.

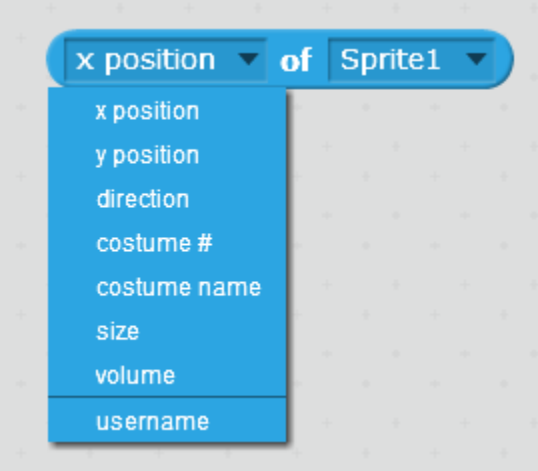


Figure 9.35

All global variables are listed under the Stage.

The *current time* block

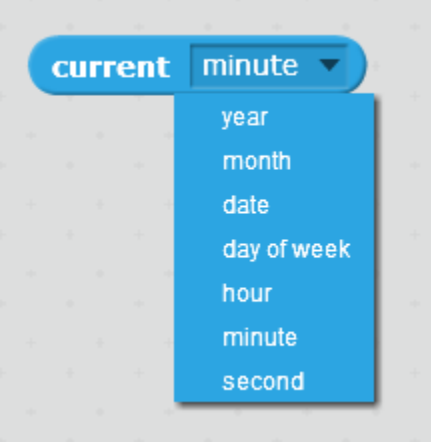


Figure 9.36

This block is used to get different values of time in an application. Depending on the input parameter selected, it can be used to get the current year, minute, second, and so on.

Scratch also provides ways to perform operations on two or more data of the same type to produce new data of the same type, with blocks from the *Operators* block category.

The *Operators* block category

Figure 9.37

Depending on the type of the data, there are three types of operations that can be performed on data in Scratch: arithmetic (number) operations, string operations, and comparison (boolean) operations.

Arithmetic Operators

These operators are used to perform arithmetic operations on data of the number data type. When these operators are used to perform operations on numbers, new numbers are produced.

The + operator block

Figure 9.38

This operator block is used to add two numbers together. When this operation is performed, a new number, which is the result of the addition of the two numbers passed as input parameters to the block, is produced. You can click on the block in the blocks palette or the scripts area to view the result of the operation.

Figure 9.39

The - operator block

Figure 9.40

This operator block is used to subtract a number from another one. When this operation is performed, a new number, which is the result of the subtraction of the second input parameter from the first input parameter, is produced.

Figure 9.41

The * operator block

Figure 9.42

This operator block is used to multiply two numbers together. When this operation is performed, a new number, which is the result of the multiplication of the two numbers passed as input parameters to the block, is produced.

Figure 9.43

The / operator block

Figure 9.44

This operator block is used to divide a number by another one. When this operation is performed, a new number, which is the result of the division of the first input parameter by the second input parameter, is produced.

Figure 9.45

The *pick random* operator block

Figure 9.46

This block is used to generate a random number between the number passed as the first input parameter, and the number passed as the second input parameter. Every time this block is executed, a new random number is generated.

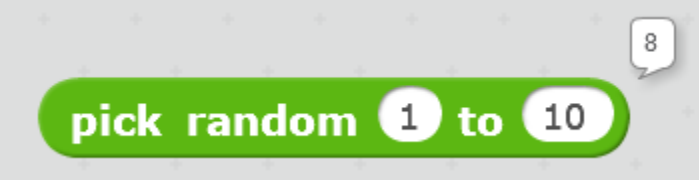


Figure 9.47

The *mod* operator block

Figure 9.48

This block is used to find the modulus of two numbers, that is the remainder of the division of the two numbers passed as input parameters to the block.

Figure 9.49

The *round* operator block

Figure 9.50

This block is used to round off a decimal number to the nearest whole number.

Figure 9.51

The *sqrt of* operator block

Figure 9.52

This block is used to find the square root of a number.

String Operators

These operators are used to manipulate strings to produce new strings.

The *join* operator block

Figure 9.53

This block joins the two strings passed as input parameters to it, to produce a single string.

The *letter n of string* operator block

Figure 9.54

This block is used to retrieve a single letter from a string. Each letter in a string has an *index*, which represents the numerical position of the letter in the string. For example, the letter “h” in the string “hello” has an index of 1, and the letter “o” in the same string has an index of 5.

The index of the letter to be retrieved is passed as the first input parameter *n* to the block, and the string from which the letter is to be retrieved is passed as the second input parameter to the block.

Figure 9.55

The *length of string* block

Figure 9.56

This block is used to get the length of a string, that is the number of letters that make up a string.

The third type of operation that can be performed on data is the comparison operation, which is performed on data of different types to produce Booleans. Booleans are usually compared in order to make decisions in an application; hence, we will learn how to use the two types of boolean operators, relational and logical operators, to compare data and make decisions in an application in the next chapter.

Summary of Chapter Nine

The following key concepts were covered in this chapter:

- Data needs to be manipulated in one form or another when programming an application. Data can be in the form of input parameters to blocks, component properties, and variables.
- The different forms in which data can be presented and used are called data types. There are three data types in Scratch: numbers, strings and booleans.
- Variables are portions of an application's memory that are used to store data. A variable must have a name, which is used to identify it, and a value.
- Variables can be created and manipulated using blocks from the *Data* block category.
- A variable's scope determines which components can modify its value. Global variables can be modified by all components in an application, while local variables can only be modified by the component that created the variable. All variables created by the Stage are global variables.
- Variable monitors are used to track the value of a variable in real time on the stage.
- The Sensing block category provides blocks that allows the manipulation of data that is not tied directly to a particular component to program useful functionalities into an application.

- The *ask and wait* and *answer* blocks are used to retrieve information from the user of an application.
- When data of a type other than the one required by a block is passed as an input parameter to the block, Scratch automatically attempts to convert the data to the required data type.
- The Operator block category is used to perform different types of operations on data to produce new data.