Scratch Tutorial

Pixel Arts Game Education

Updated as of: November 6, 2018

1 Scratch: The Why & Wherefore

What is Scratch, why would you use it, and *how* do you use it? Those are the questions we'll be answering in this tutorial.

The first question is the simplest: Scratch is a programming environment built for teaching beginners, particularly children, how to code. I say programming *environment* not programming *language* because there's so much more too Scratch than the core language. There's a simple graphics editing programming, a music editor, a visual *stage* that displays the actions of entities you program on screen, as well as a robust social media component that allows Scratch users to share their projects with each other and even *build on* each other's work through *remixing*.

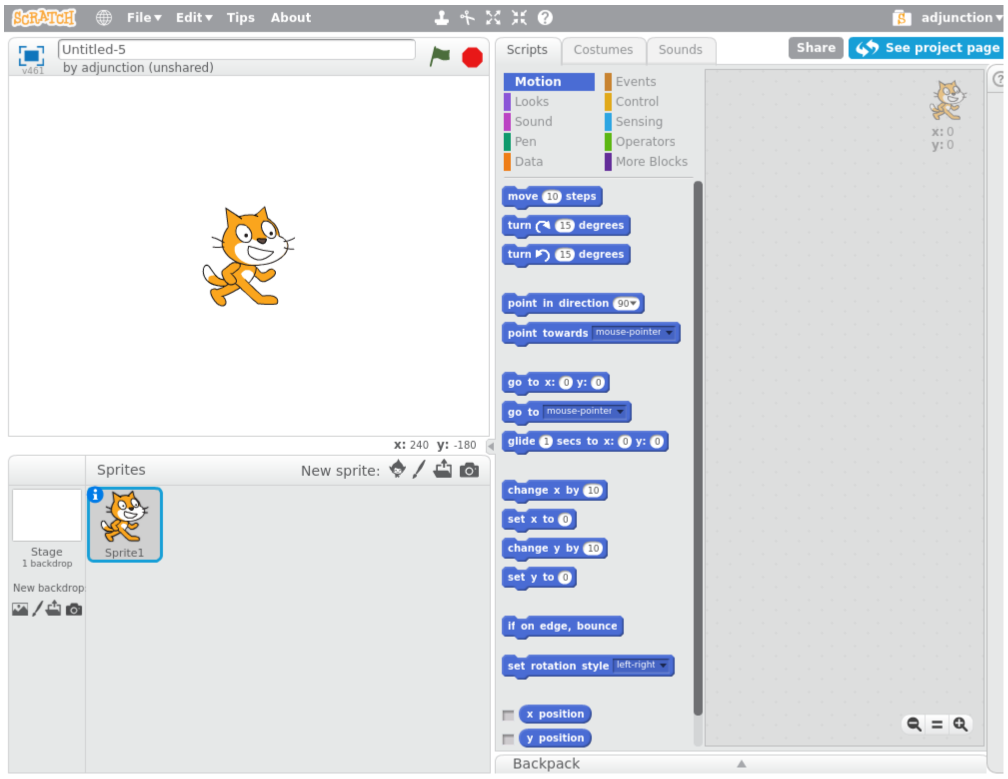


Figure 1: Scratch provides a rich programming environment

Scratch isn't the only programming environment meant for teaching, but it is one that exists at a sweet spot for a number of desireable qualities: it's free, can work on under-powered hardware[1], and has enough adoption already that there's no shortage of examples, tutorials, or help for any project imaginable.

Scratch is a blocks-based programming system, which means that rather than typing code you connect individual statements—*blocks*—together in order to build up the program piece by piece.

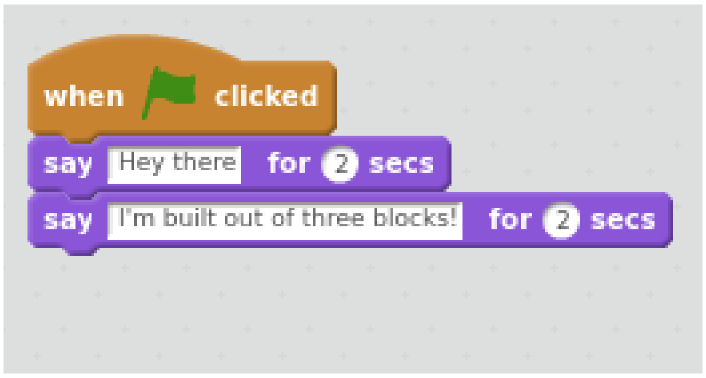


Figure 2: Three blocks connected together

Blocks-based programming languages have become popular in recent years because 1) they're very easy to start using 2) they eliminate the frustration of syntax errors 3) they make all the basic operations of the language immediately visible

Now that we've covered that prefatory material, let's start working through a guided example!

2 Making a sprite & making it move

To begin our tutorial,[2] go to the Scratch website. Near the top of the page you should see an option that says `Create`. Click on that to get started on your first project!

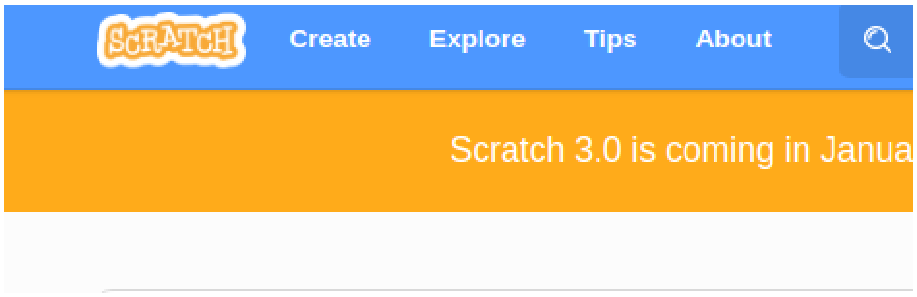


Figure 3: The `Create` button will start a new project

You should see something that looks like the following figure

[1]especially with the new version of Scratch debuting in January of 2019

[2]If you don't have a Scratch account, this is the time to make one!

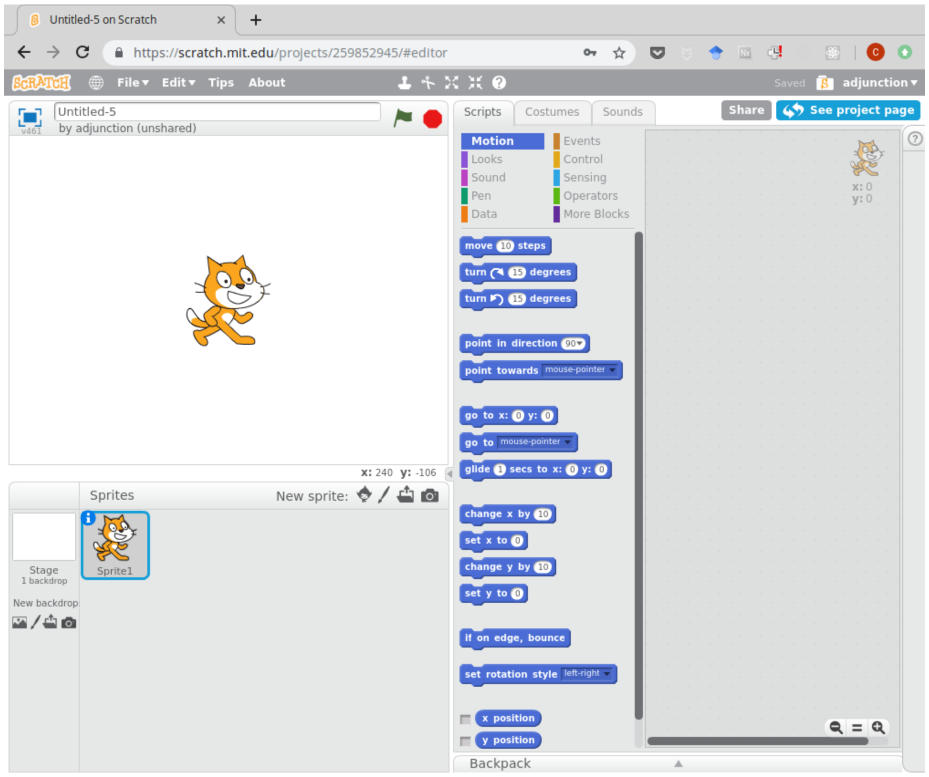


Figure 4: The Scratch equivalent of a blank canvas

What a nice, happy little cat! This cat is a *sprite* in Scratch's parlance. If you've heard of "sprites" in other digital art or gaming contexts, this is a little bit different. Sprites in Scratch are a bundle of the art *and* all the code and sound information that you add into the sprite. Effectively, they're little software robots that you program.

Now the next thing you should do is *right-click* on the little cat and select `delete`. We're going to all choose our *own* sprite art to start with instead of the default Scratch cat. To pick a new sprite you can click on the little elfin figure next to the the text that says `New sprite`. You'll see a menu with many possible choices, like in figure 5.

Pick whatever character you want, though we're choosing a main character for our game so a person or animal might work better than, say, a piece of fruit. As for me, I chose the elephant—which will appear as my choice for the rest of the screenshots in this tutorial

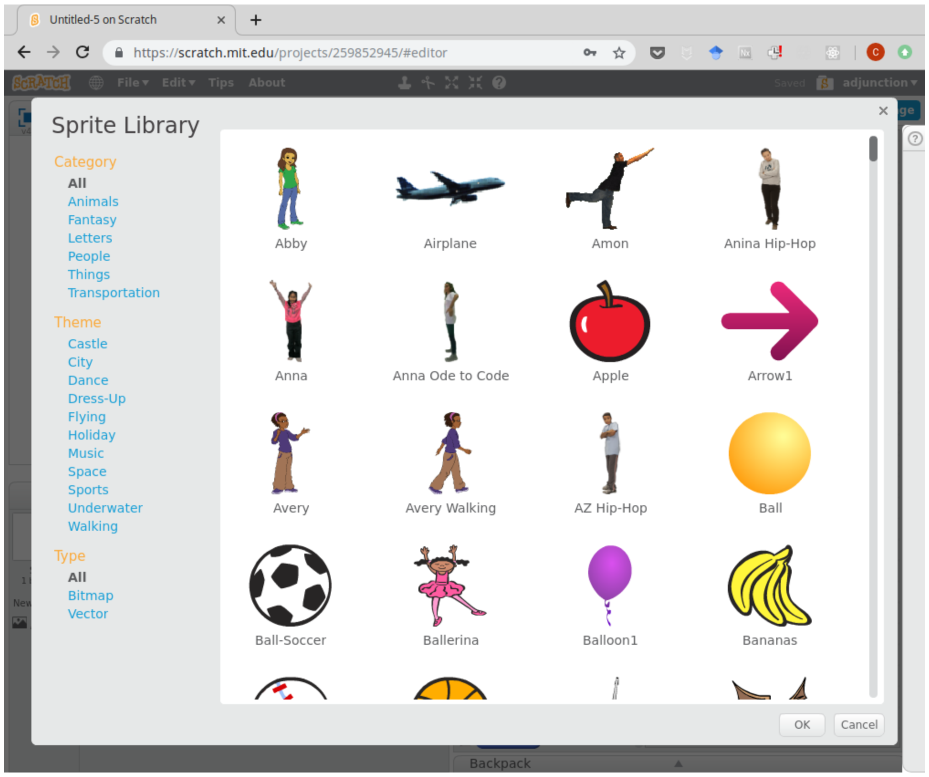


Figure 5: An embarrassment of pre-made art

We're finally at the point of adding our first code: a small loop to make our sprite move when we press the `wasd` keys. The first thing to notice is that there's a number of sections for Scratch blocks, shown in figure 6:

Motion All the code that makes your sprite move around. Scratch uses an x,y co-ordinate system: ranging from $(-240, -180)$ in the lower-left corner to $(240, 180)$ in the upper-right corner

Looks Code blocks that change the appearance of the sprite, including making the sprite talk and causing it to appear & disappear

Sound Code for playing and stopping sounds

Pen Code for drawing with the sprite as it moves

Data Where you declare variables and find the code to modify them once declared

Events Code for having a sprite respond the program starting, certain keyboard and mouse actions, or *messages*[3] sent by your code

Control Code for combining with other blocks to make things happen repeatedly or conditionally

Sensing Code that helps you make decisions based on 1. what the sprite is touching 2. how far the sprite is from other sprites 3. what keys are pressed among other things

Operators The code blocks for arithmetic and comparison operators as well as standard mathematical functions

[3]We'll cover messages later in the tutorial!

More blocks This section is for defining your own blocks and using special software integration not always available

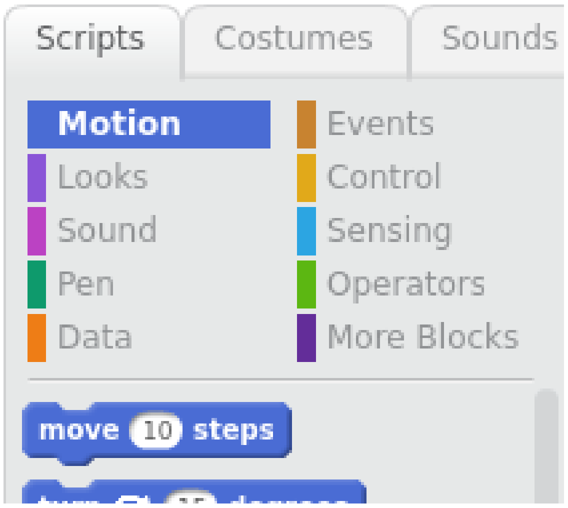
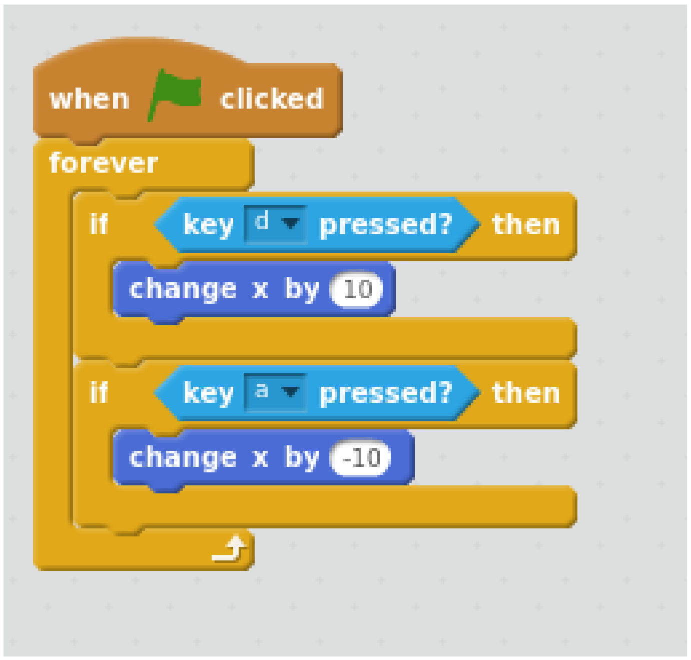


Figure 6: Scratch divides its code into categories

With these categories in hand, we can break down our proposed task down into what categories we're going to need: we're going to need *Motion* to make the sprite move, *Sensing* to tell which keys are held down, *Control* to keep our code running for the duration of the program, and *Events* to make the program start in the first place. The first example is shown below in figure 7.



```
when [green flag] clicked
forever
    if <key d pressed?> then
        change x by 10
    if <key a pressed?> then
        change x by -10
```

Figure 7: Our first movement code

We can verbally describe what this code does! When the program starts, i.e. when the green flag is clicked, we want to start checking for movement keys being pressed. We want to keep checking until the program ends, which as far as Scratch is concerned is *forever*. *If* the *d key is pressed* we want to move right, i.e. increase the x co-ordinate. *If* the *a key is pressed* we want to move left, i.e. decrease the x co-ordinate.

Note the way that we've nested blocks of code inside each other! Blocks like *forever* and *if* modify how the code inside them is run.

Exercise Now modify your code to add the ability to move up and down when you press `w` and `s` respectively

Exercise Modify your movement code so your sprite faces right when you move right and faces left when you move left. Hint: to make this look correct you might need to check out the sub-menu for the sprite that you can access by pressing the little "i"

3 Idle animations, costumes, and variables

Our next task is to create a cute little idle-animation loop for our sprite. First, let's look at the sprite you've made so far. So far you've been using the *Scripts* tab on your sprite. There's also the *Costumes* and *Sounds* tabs! Click on the *Costumes* tab and you'll see something like the following figure

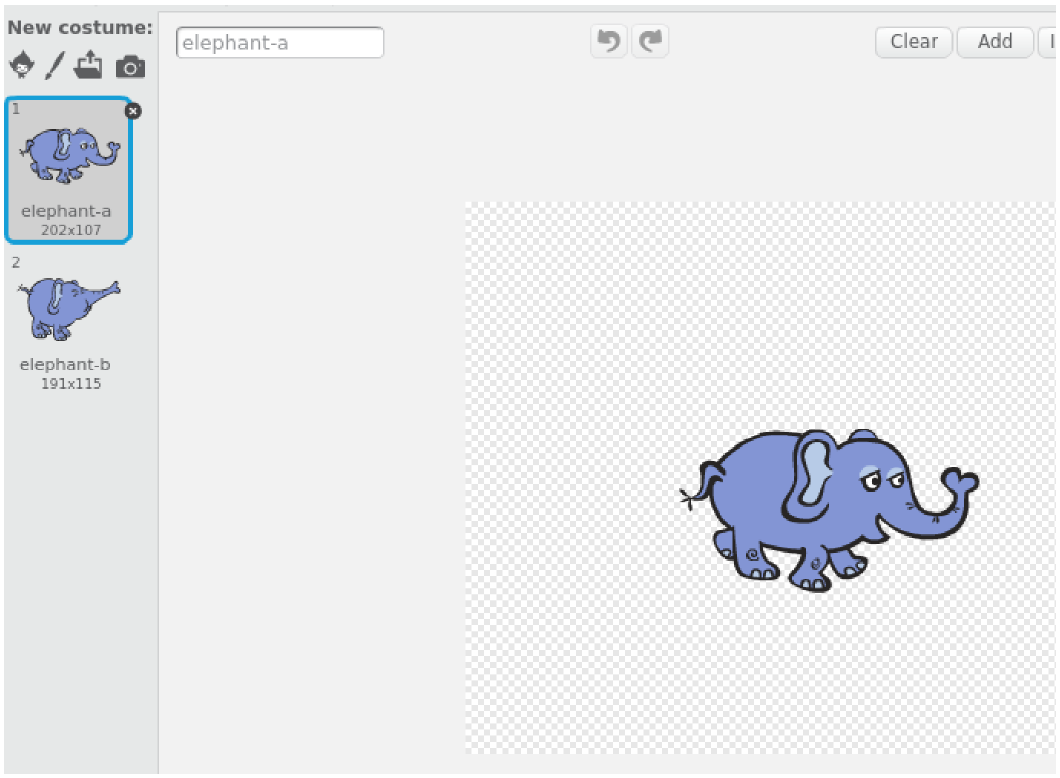
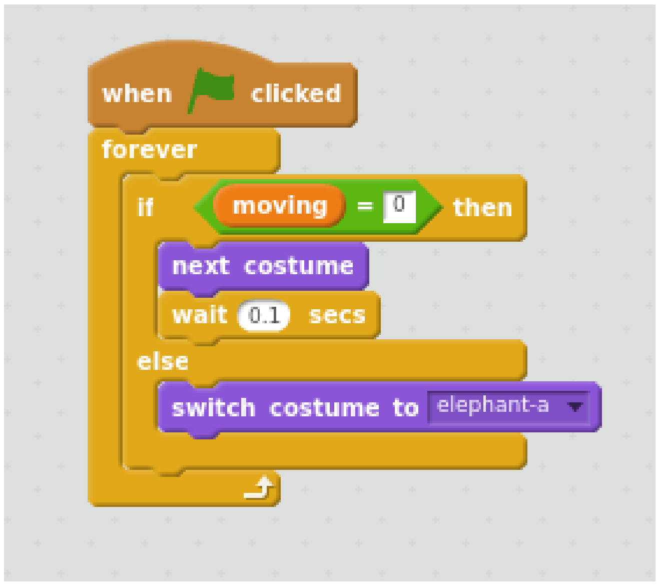


Figure 8: All the costumes for your sprite!

These are all the *costumes* for your sprite. You can switch between them using code from the *Looks* section. Take note of what the name of the first costume is because you'll need it shortly.

Now you're going to do a few things:

1. Go to the *Data* tab and click on *Make a Variable*
2. Name the variable *moving* and choose *for this sprite only*
3. Put together the block of code in figure 9 below, substituting the name of the first costume for *elephant-a* in my example
4. Modify your movement code, as in figure 10 to set the *moving* variable to 0 if the sprite doesn't move and 1 if it does!



```
when clicked
forever
  if moving = 0 then
    next costume
    wait 0.1 secs
  else
    switch costume to elephant-a
```

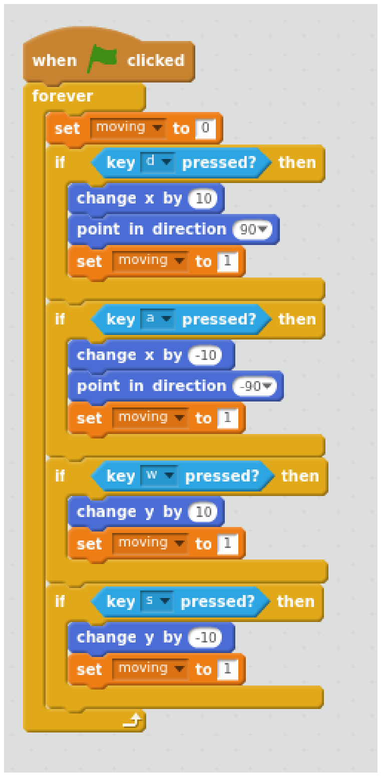
Figure 9: Idle animation code

How does this code actually work? It's worth discussing in some detail. The first new concept is that multiple sections of code can be running at the same time! Every block of code that starts with *when green flag clicked* is going to start running simultaneously! I'd argue that this is one of the core concepts of learning Scratch programming, particularly if you have past programming experience[4]. We take advantage of this feature of Scratch in order to add animations to our sprite while only *slightly* modifying our existent code!

The second new concept is the use of variables. In Scratch, as in most programming languages, variables are used to keep track of the state of the program. There's a couple of small details about variables that I want to mention. First, that the *name* of a variable doesn't really matter: whatever helps you or anyone else reading your code understand what the variable is used for is a perfectly fine name. You can also change the name of a variable at any time by right-clicking on variable's name in the *Data* section.

Finally, I want to describe the difference between variables being *For all sprites* versus *For this sprite only*. Variables that are *For all sprites* can, as the name might imply, be seen by all sprites. This is really useful if you're using a variable to keep track of something that multiple sprites need

[4]If you have programming experience you might be surprised to see that, unlike most programming languages, Scratch is innately & implicitly multi-threaded.



```
when [flag] clicked
forever
    set moving to 0
    if key d pressed? then
        change x by 10
        point in direction 90
        set moving to 1
    if key a pressed? then
        change x by -10
        point in direction -90
        set moving to 1
    if key w pressed? then
        change y by 10
        set moving to 1
    if key s pressed? then
        change y by -10
        set moving to 1
```

Figure 10: Modified movement code

to be able to access, such as "what level of the game is the player on" or "how many enemies are left in the wave", but has the disadvantage that there can only be a single *For all sprites* variable with the same name. Variables that are *For this sprite only* are good for when you want to have multiple sprites[5] all use a variable with the same name. When would this happen? Common examples are variables that keep track of the sprite's velocity or hit points.

Exercise Maybe your sprite has costumes that work better for a moving animation than an idle one? Try modifying the code so that the animation plays only when the sprite is moving!

4 Coding enemies

What we have so far isn't really *a game*, is it? This is true for a number of reasons but we'll tackle one of the more obvious ones: there's no other characters! In this section, we'll be creating a new sprite to be an enemy, learning how to make multiple copies of the sprite using "clones", and giving the enemy some really primitive AI.

To start, we're going to make a new sprite just like we picked our old sprite: clicking on the little "sprite" figure and picking a suitable character. I picked the little crab for no reason beyond finding crabs versus elephants inherently funny.

Click on your enemy sprite and you'll see something like the following

[5]or clones, as we'll cover later

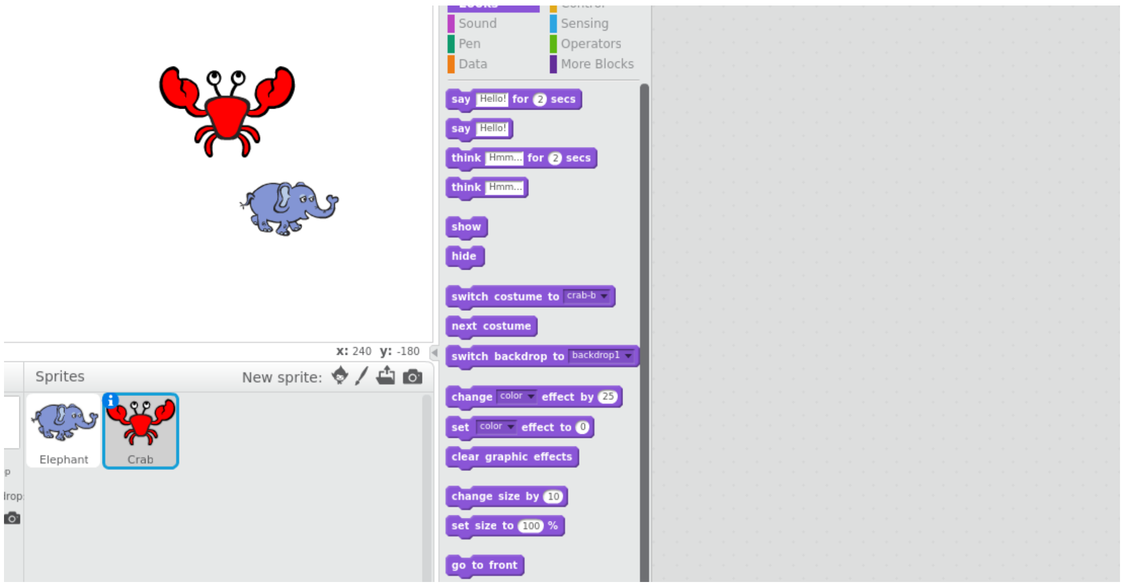


Figure 11: No code on this crab!

The first thing you might notice is that there's no code! In Scratch you code each sprite separately[6]. The same way that multiple scripts on your player character all ran simultaneously, you *also* have all code on all *sprites* running simultaneously.

So what are we going to do with our enemy sprite now that we have them? This is where we have some game design choices, but I'm going to make a choice for a very simple AI and very simple enemy creation and I'll describe it in words before starting the code.

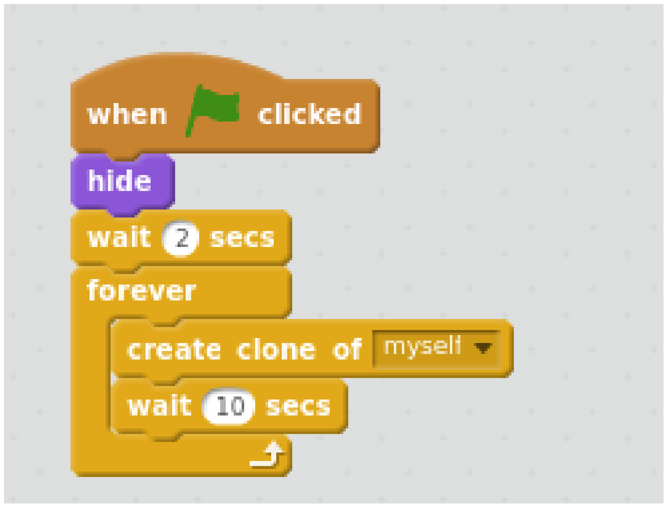
> **Enemy Behavior** After two seconds enemies should start spawning. One enemy should spawn every ten seconds. Once an enemy is spawned it should move between random locations on the screen every two seconds *unless* it's within 150 px of the player character, then it should point towards the player and start moving towards them until the player is beyond 150 px distance again.

Now we can *implement* this design!

To start, let's handle the initial spawning of enemies. This is our first example of using *clones*! Cloning allows you to use a single sprite and create many copies of it *while the game is running*. Unlike making an entirely new sprite, these clones will all have copies of the code from the original sprite and have *their own copies* of the original's *For this sprite only* variables. Making a clone is as simple as using the *create clone of...* block that can be found under *Control*. Any sprite can either make a copy of itself or of any other sprite. Here we'll be having the enemy sprite make a clone of itself. Later we'll have our player character make a clone of a projectile sprite.

Below in figure 12 the code to make the sprites spawn at the rate we described in our specification. You might be wondering why we have the *hide* at the beginning. That's because we're using the original, non-cloned, sprite almost like a factory—a template—for making the enemies. We don't want it to move around and attack the player character or anything like that. Hiding the original sprite helps keep it out of the way.

[6]And you can also put code on the stage, apart from any particular sprite, but we won't be doing that in this tutorial



```
when [flag] clicked
hide
wait (2) secs
forever
    create clone of [myself v]
    wait (10) secs
```

Figure 12: Making clones of your enemy sprite

Experiment Try making the original enemy sprite visible and see how it affects your game as you proceed. With the code to make clones done, how do we write code that will run on-the-clones-and-only-the-clones? You may have seen the relevant block when you were copying the code to create sprites from the *Control* section: *when I start as clone.*

Here, rather than give the actual code to make the enemy sprites attack the player character I'm going to give a partial solution while leaving holes for you to fill in!

Exercise Fill in the proper blocks to the shell in figure 13 to match the written specification for the enemy behavior!

Exercise Do the relative proportions of all the characters look the way you want? Try experimenting by putting a *set size to* block under a *when green flag clicked* and playing with the numbers.

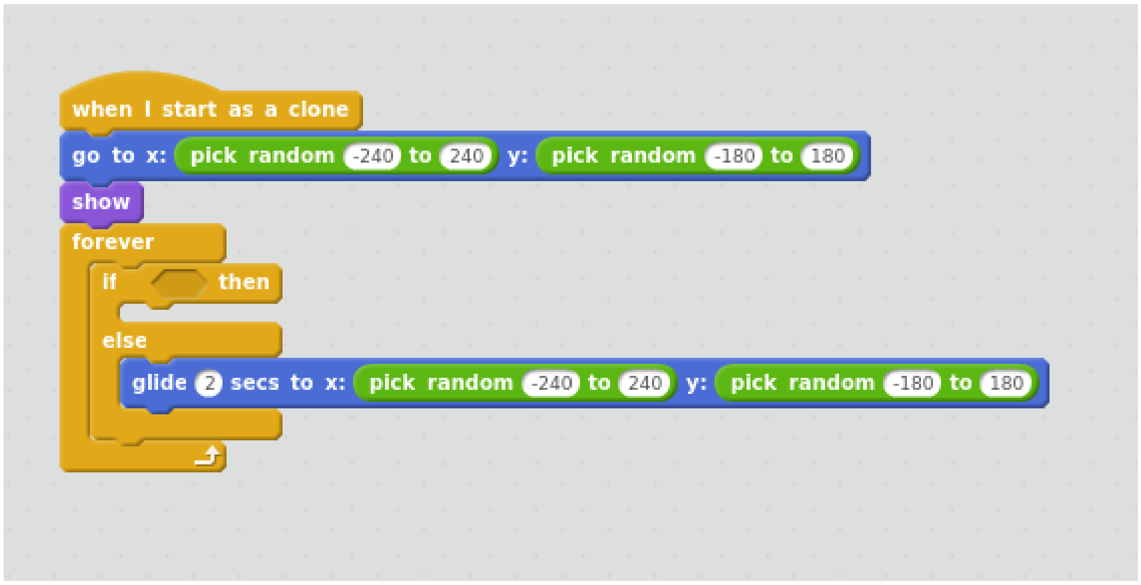
Exercise Add another enemy type and have it behave differently. It could move entirely randomly. It could teleport around the stage. It could hold still until the player crosses its line of site and then **slam** quickly to the other side of the screen.

5 Health and messages

Now we have enemies that can doggedly[7] pursue our player character. You may have noticed, though, that when they make contact with the player nothing happens! That's what we'll be fixing in this section. We're going to

1. Add a health variable for the player
2. Add a new loop that keeps track of if an enemy is touching the player
3. Decrease the health variable by one every time an enemy touches the player but *not* allowing more than one point of health drop per second.

[7]literally, if you chose a dog sprite

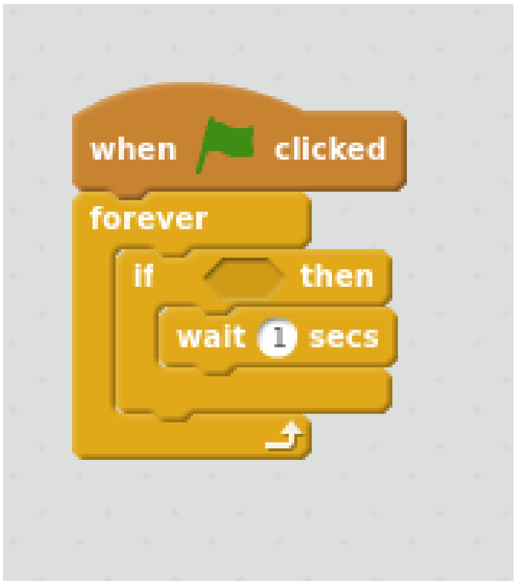


```
when I start as a clone
go to x: (pick random (-240) to (240)) y: (pick random (-180) to (180))
show
forever
    if <> then
    else
        glide (2) secs to x: (pick random (-240) to (240)) y: (pick random (-180) to (180))
```

Figure 13: The start of the enemy code

Make a new variable called *health* by going to the *Data* section. Add a block that sets the initial health to five when the game is started.

Next, we'll write the loop that keeps track of when the player is damaged. Again, I'm providing a partial loop in figure 14 below.



```
when [flag] clicked
forever
    if <> then
        wait (1) secs
```

Figure 14: Partial code for taking damage

Exercise Fill in the code to make the damage loop work correctly. Should the health reduction take place *before* or *after* the contact with the enemy happens?

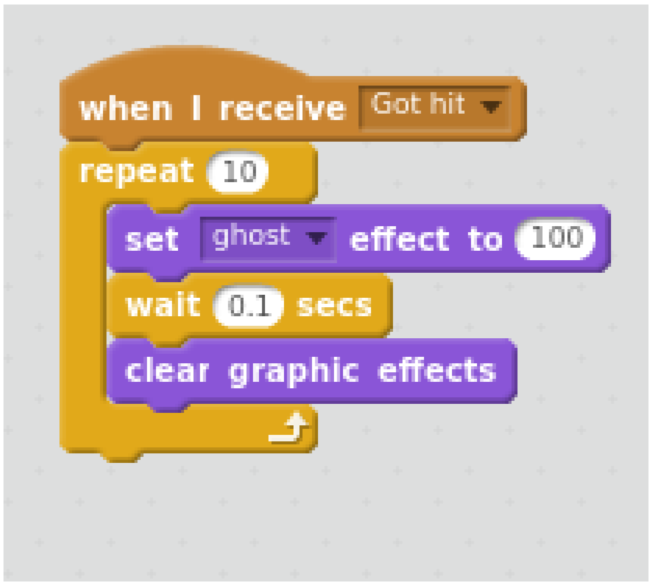
We've put in the one second delay to ensure the player has a chance to survive. Scratch is running all these loops as fast as it can. If two sprites are touching for even a tenth of a second then Scratch might count that tens or hundreds of times without some kind of delay to slow it down. This delay is actually an old video game concept called "invincibility frames", or *i-frames*.

Old video games usually gave some kind of visual signal that i-frames were occurring, often the player character would blink rapidly for the duration of the i-frames. Let's now be old school and do that ourselves!

This is going to be our first example of *messages* in Scratch. Messages are the tool that helps you *synchronize* between different parts of your code, ensuring that various loops or actions are happening exactly when you want them to. In our case, we'll be broadcasting a message called *Got hit* whenever the player takes damage. This *Got hit* message will then trigger the visual effect that represents the i-frames.

Now go to the *Events* section. You'll see a block called *broadcast ...*. Put that block in your damage loop just before the *wait* block. Click on the triangle that's on the *broadcast* block and choose *New message*. Type "Got hit" and you're done with this part.

For the other half, copy the loop from figure 15 below



```
when I receive Got hit
repeat 10
    set ghost effect to 100
    wait 0.1 secs
    clear graphic effects
```

Figure 15: Code for i-frames visual

In this code, we're using the *ghost* effect to implement our blinking. We use this instead of *hide* so that we have more flexibility in our visual effects: we could change the i-frames visual to something else just by changing the effect and its strength. I also have a personal preference to only use *hide* when I don't want the sprite to be "in play" at all.

In this loop we use the *repeat* block to turn the sprite invisible and back again ten times, with the duration of the visual effect lasting a tenth of a second each time: one second in total, the same as our i-frames length.

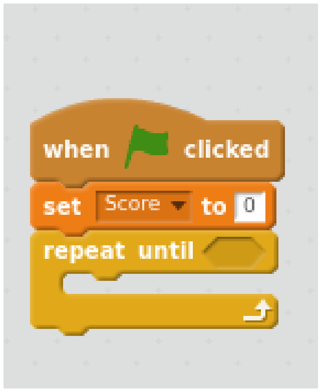
Experiment Try taking out the delay we built into the damage-dealing code and seeing what happens. Is there another way to implement i-frames without having to put in the simple *wait* that we used above?

6 Keeping score and ending the game

Now that you're keeping track of your character's health there needs to be a way to actually end the game when your health hits zero! This is going to also tie in with adding a very simple scoring system that adds points for every second the player has lived. This involves another loop, but a slightly different one! We're going to be using the *repeat until ...* block this time.

First things first, we need to make a new variable called *score*. Then you can copy and fill in the partial code below in order to create the scoring loop.

Exercise Fill in the code for the loop in figure . When done this code should 1. increment the score *after* every second 2. run *until* the player's health equals zero

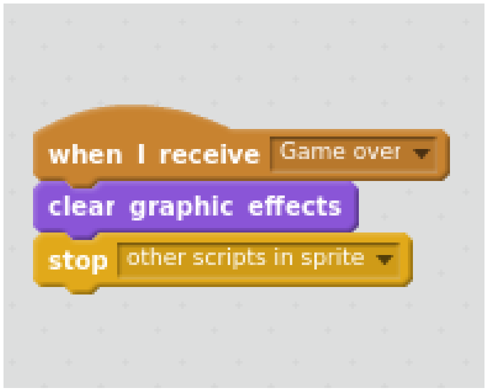


```
when [flag] clicked
set Score to 0
repeat until <>
```

Figure 16: Partial score-keeping code

Now, we add the code to end the game properly! Just this loop alone means that you'll stop accruing points when your health hits zero but if you try the game out you'll see that you can keep moving and the enemies will keep spawning. The game hasn't *really* stopped.

How do we syncronize between parts of our code? Messages! You should now add a *broadcast* message just underneath your *repeat until* loop. Make the new block broadcast a message called *Game over*. Now add the following code into your player character sprite:



```
when I receive Game over
clear graphic effects
stop other scripts in sprite
```

Figure 17: Stopping the player

Two things about this code. First, we're clearing the graphic effects because we can't predict how the i-frames code is running relative to when the hit is resolved. Without this block the player might be visible or might not be when the game ends. The second is that we're using a new block from the *Control* section: *stop ...*. The *stop ...* block can do several different things: it can stop the entire program with *stop all*, it can immediately end the current loop its running in with *stop this script*, or it can immediately end all the other code running in a sprite with *stop other scripts in this sprite*. It's that last one that's useful for us! It keeps the player from being able to still move the sprite when the game has ended.

Finally, we need to add the same code to the enemy sprite!

Exercise Add in code to the enemy sprite so that it will stop the production of new enemies and keep them from chasing the player. Extra credit: you can make all of the clones disappear by *delete*-ing them after you *stop other scripts in this sprite*. Look in the *Control* section for the block you need!

Exercise Add a "game over" sprite that is hidden at the start of the game and is shown when the *game over* message is sent

Exercise Add a new costume to your player that represents "dying" in the game. Switch to it when the player loses. This will require that you modify your idle loop so you're not cycling through the death costume!

7 Fighting back

This game is going to be incredibly unfair if the enemy sprites just keep spawning and spawning forever. What your character needs is a way to fight back! We're going to give them a way to shoot projectiles at the enemy sprites, destroying them on contact.

This is, as we alluded to earlier, another application of clones. We're also going to write another specification again.

> **Firing projectiles** When the player character hits the space bar a projectile should be fired in the direction the player is facing, but at a rate of no more than once per second. This projectile should travel horizontally in a straight line *until* it either *touches an enemy* or *touches the edge of the screen*. After it touches either of those, it should be deleted. When an enemy is hit by a projectile, it should be deleted from the game.

With this specification in hand, we can start working on this last feature! The steps we'll need to take are

1. Create a new sprite to be the projectile
2. Add a new loop to the player to check if the space bar is pressed
3. Add code to clone the projectile sprite and send it in the appropriate direction
4. Add code to the enemy sprites to handle being hit by the projectile

As this is the final section, I'm going to just be giving a rough outline of what to do with the "answers" given in the back of this tutorial.

We'll elide the act of making a new sprite as its the same as previous sprites. I picked a "sun" sprite as my projectile. You probably want to make the sprite fairly small, so you might want to put a *set size to...* block after a *when green flag clicked* for this sprite in order to play around with sizes. You'll also want to hide the original sprite just like we did with the original enemy earlier.

Next, we'll want to make a new loop to check whether the player is pressing the space bar. It's easiest to make this a new loop instead of grafting it onto our movement loop so we can easily put in a delay that prevents the player from firing a hundred projectiles a second by holding the button down.

The code for our projectile involves the most new concepts. You need to

1. Start your code with a *When I start as a clone*
2. Place the projectile clone at the player's location
3. Have the projectile point in the same direction the player is facing
4. At this point, make the sprite visible
5. Have the projectile move until it hits either an enemy or an edge. You can use a *repeat until* loop for this
6. Delete the sprite after it hits either of those two things, *after a very short delay*

Finally, you need to add code for the enemy sprite to respond to *being* hit. I think the easiest way to do this is with a *wait until* block. As another hint, you should add a tiny tenth of a second delay before deleting the enemy sprite. This helps ensure that both the projectile and enemy will properly delete themselves after they come into contact.

Once you're done with all these tasks, you've got a small yet complete game!

Exercise Does the game feel too easy now? Brainstorm some ways to make the game more complicated and try implementing them. Low-hanging fruit might be to increase the rate of enemy spawns the longer the game goes.

Exercise Maybe you should get points every time you kill an enemy in addition to staying alive for longer. Modify your code so that the score goes up by five every time an enemy sprite is destroyed by the player's attack.

8 Sharing your work!

As we wrap up, I want to point out the little *Share* button in the upper-right corner of the screen

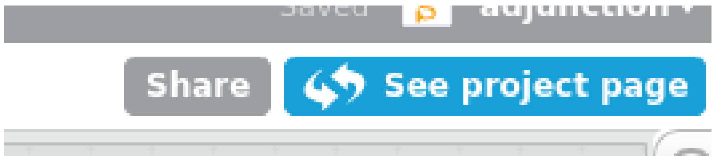
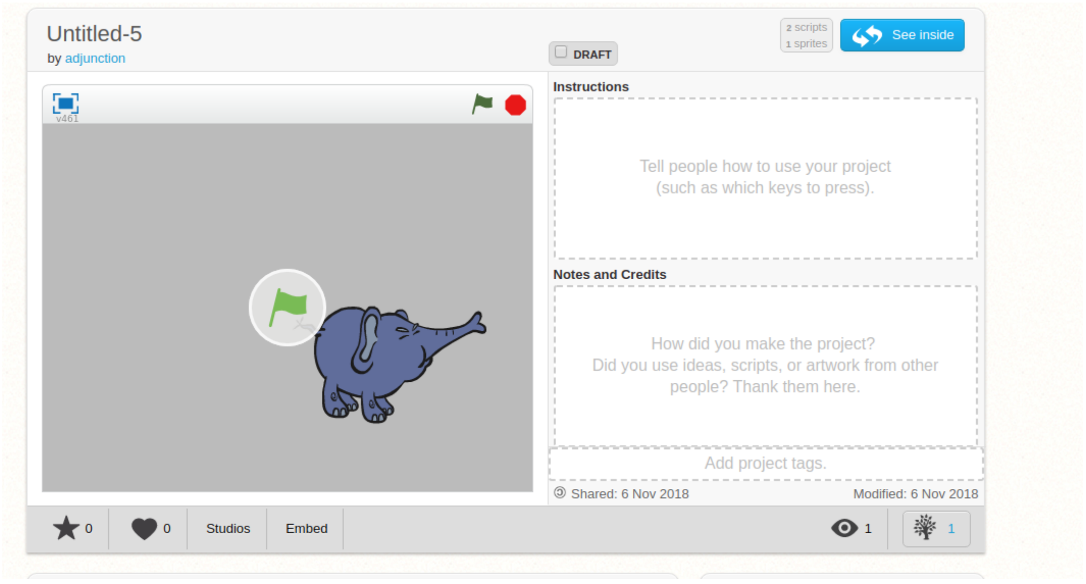


Figure 18: Share your work!

When you're done, or at least done enough!, with this tutorial you can press this button and you'll be taken to a screen where you can add instructions, credits, and other comments!



Tell people how to use your project (such as which keys to press).

How did you make the project? Did you use ideas, scripts, or artwork from other people? Thank them here.

Add project tags.

Shared: 6 Nov 2018

Modified: 6 Nov 2018

Figure 19: Write down instructions to help other players!

Finally, you should check out the *Explore* link at the top of the Scratch page. You can click on games, try them out, and then remix them if you'd like! We haven't talked much about *remixing* yet, but it's a core part of Scratch-as-a-site. Remixing a project means that you make a copy of it for yourself[8] but while still keeping the history of who originally made it. It's an easy way to expand on work done by other people, get your feet wet by modifying complex projects, or just checking out neat tips and tricks.

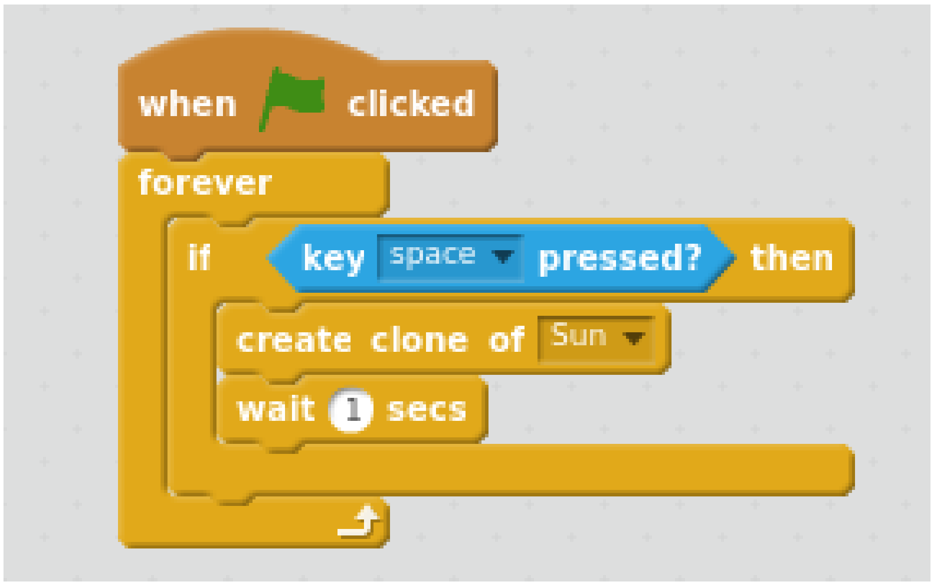
9 Topics we couldn't cover

Here's a short, but incomplete, list of topics that we haven't gotten to cover but are good to learn!

- Changing backgrounds
- Drawing sprites and backgrounds
- Using sounds
- Uploading animated gifs to make costumes
- Using cloud variables
- Using the *Pen* commands to draw on the screen
- Making solid walls
- Simple physics in Scratch

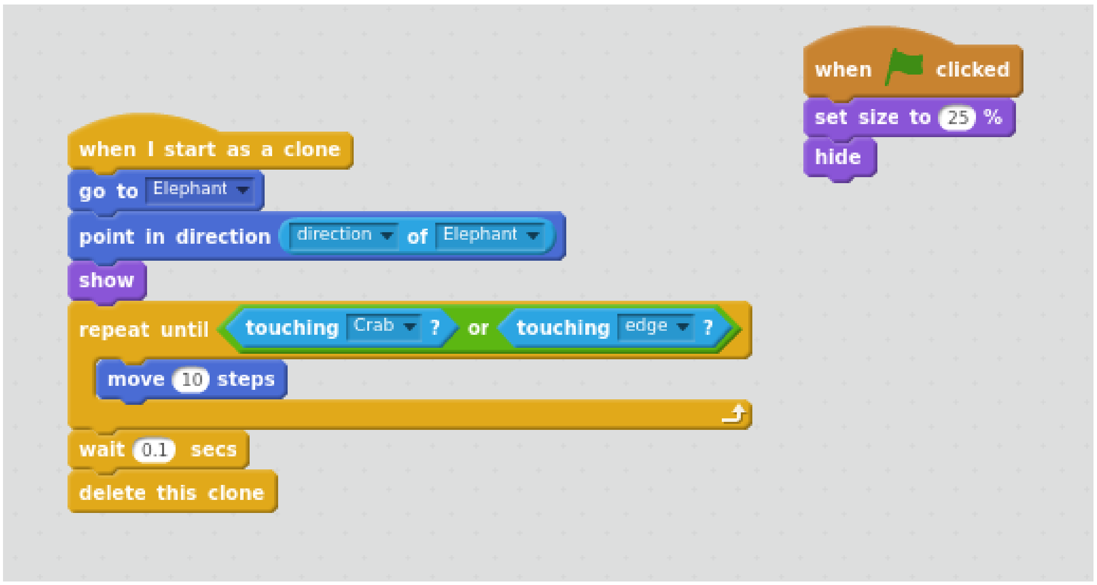
[8]It's like forking a repository on Github, if that's a useful metaphor

10 Solutions to projectiles section



```
when [flag] clicked
forever
  if key space pressed? then
    create clone of Sun
    wait 1 secs
```

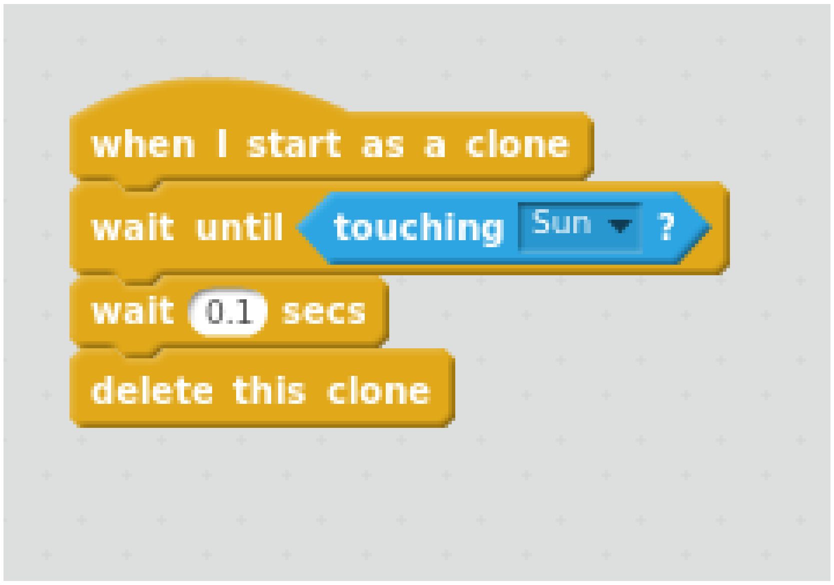
Figure 20: Firing projectiles



```
when I start as a clone
go to Elephant
point in direction (direction of Elephant)
show
repeat until <touching Crab ? or touching edge ?>
  move 10 steps
wait 0.1 secs
delete this clone
```

```
when [flag] clicked
set size to 25 %
hide
```

Figure 21: Projectile code



```
when I start as a clone
wait until <touching [Sun ▼] ?>
wait (0.1) secs
delete this clone
```

Figure 22: Code for enemy getting hit by projectile