The Beauty and Joy of Computing

Lab Exercise 10: Shall we play a game?

[*Note: This lab isn't as complete as the others we have done in this class. There are no self-assessment questions and no post-lab reading/discussion. There are some very important concepts here, however, which will help you in your project. If you don't understand any of this lab, make sure you ask questions!*]

Objectives

By completing this lab exercise, you should learn to

- Understand and work with layering of multiple sprites (applied to make a side-scrolling background);
- Program self-cloning "projectiles" for use in a game; and
- Merge separately developed programs into one larger program to support development by teams.

Background (Pre-Lab Reading)

You should read this section before coming to the lab. It describes how various things work in BYOB and provides pictures to show what they look like in BYOB. You only need to read this material to get familiar with it. There is no need to actually do the actions described in this section, but you certainly can do them if it would help you follow along with the examples.

The purpose of this lab is to introduce a few new concepts and techniques that are important for developing larger projects, and games in particular, in BYOB. These are techniques that will be useful in many of your final projects, so are important to know and understand!

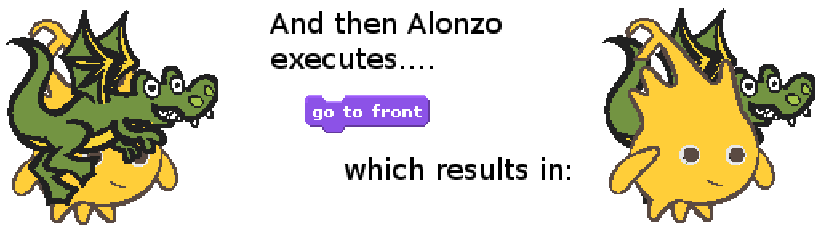
Image Layers

In the activities and examples we've done up until now, sprites have been separate and have not overlapped - what if that is not the case? For example, in the very first lab we added the dragon sprite to the initial Alonzo sprite, and both sprites started in the center of the screen and had to be dragged apart. But before we separated them, how did BYOB decide whether Alonzo or the Dragon should be displayed in those locations where they overlap? This uses the concept of **layers**, which is common now in graphics programs as well as in the drawing tools of programs like PowerPoint or Word. You can imagine each sprite being in a specific layer - with higher layers obscuring lower ones. In BYOB, each sprite is in a layer by itself that has a specific position relative to all other sprites, and when sprites are created they always are created in a new "top" layer - as a result, there is never any ambiguity about which sprite is visible. So the answer to the question of whether Alonzo or the dragon is shown is "whichever one is in the higher layer." Layers are not fixed, and can change based on program actions. The blocks used for this are the following two blocks in the "Looks" category:

go to front

go back 1 layers

If a sprite executes the "go to front" block, it will move to the very top layer and be displayed over any other sprites that overlap with it. For example, when the dragon is added to the base project with Alonzo it is put on top of Alonzo, and if Alonzo later executed the “go to front” block it would be put on top of the dragon. We can illustrate that as follows:

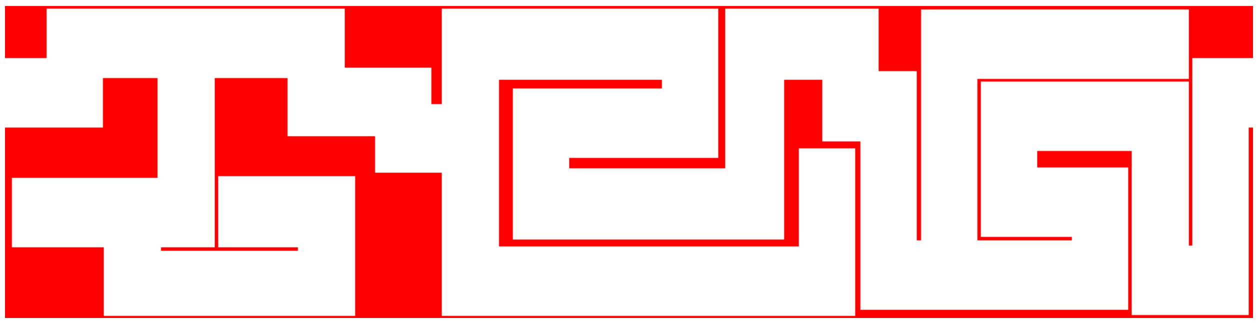


The “go back...” block does the opposite, but is more fine-grained since it can indicate a number of layers to send the sprite back, staying on top of some sprites while moving behind some others. To send a sprite to the lowest possible layer you can just use a large argument like 1000, which will max out the layer so that it is behind everything else (as long as the number of layers to go back is greater than or equal to the total number of sprites).

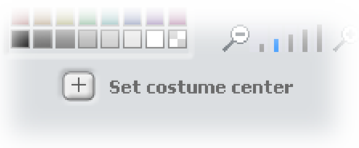
Side-Scrolling Background

Most of you are probably familiar with side-scrolling video games, in which the background slides left and right behind a character as it progresses through the game. However, in BYOB backgrounds are fixed and cannot be moved, which creates a challenge. So while we think of this as a scrolling background we actually use sprites as the background, making sure they are in the bottom layer so that characters will appear on top of the background sprites. Sprites can be moved around and even placed partially off the stage, which is exactly what we need to create a side-scrolling game.

The first step in creating a side-scrolling effect is to create the background. The stage is 480 pixels wide by 360 pixels high, so it is best to make a background that is 360 pixels high and some multiple of 480 pixels wide. You can use whatever drawing program you like, but make sure the dimensions of your finished background picture follow these guidelines. We’ll make a background that is three stages wide, and so create the following 1440 pixel wide image for a maze game:

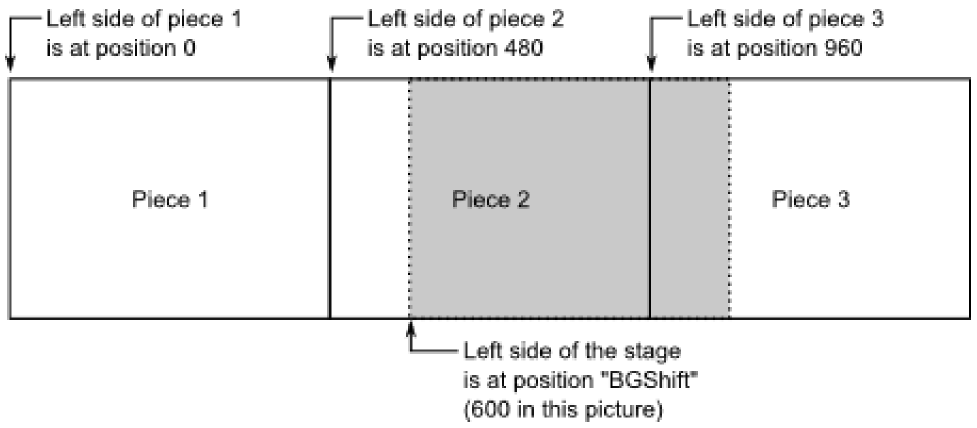


Next, we divide this up into three pieces, each 480 pixels wide, which will each be a different sprite. This brings up a small concept that we haven’t discussed: the unfortunately-named “**center**” of a sprite. When we say to put a sprite in a particular (x,y) location, what does that mean? Does the lower-left corner of the sprite get placed at (x,y)? The center of the sprite gets placed at (x,y)? The answer is that BYOB allows us to define any point we want as a reference point in a costume, and that point gets placed at (x,y). Unfortunately, BYOB calls this the “center” of the sprite, even if it isn’t anywhere near the center. You can set the costume “center” in the costume editor - in the lower-left of the editor window is the following button:



Clicking that button will bring up “cross-hairs” that you can position wherever you want, and that will become the reference point (the “center”) of the costume. For our background images, the calculations work out simplest if the “center” is the lower-left corner of the image (see what the “center” is such an unfortunate name?). You could, of course, adjust the discussion below to use whatever center you wanted, but for our discussion assume the “center” is in the lower-left corner.

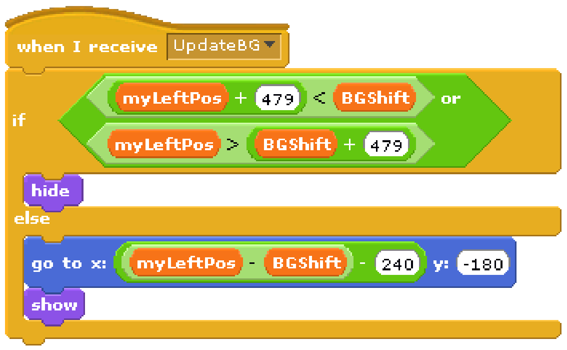
Here’s the idea for creating a scrolling background: the stage is a “window” into the larger image, and the part that we can see contains at most two of the 480 pixel wide pieces. This picture shows the basics: our picture is divided into three pieces, and when viewed as one big picture each piece starts at a multiple of 480. We use a variable named “BGShift” to indicate the position of the stage, so in this picture valid values of BGShift are 0 through 960, inclusive.



There are several ways to implement a side-scrolling background. The approach we describe is to have a script associated with each “piece” sprite that runs when it receives a signal and decides if and how to display itself. Lets use a sprite-local variable named “myLeftPos” to indicate the position of the leftmost column in the piece (so the value of “myLeftPos” for the first piece is 0, and the value of “myLeftPos” for the second piece is 480, etc.). Under what conditions is the piece off the stage and hence not seen? First, it is off the screen if the rightmost pixel in the piece is strictly to the left of BGShift (in other words, if myLeftPos+479 < BGShift). Second, it’s off the screen if the leftmost pixel is greater than BGShift+479, the rightmost position of the stage (in other words, if myLeftPos > BGShift+479). Writing all of this out as a Boolean expression, a piece is not visible if (myLeftPos+479 < BGShift) or (myLeftPos > BGShift+479).

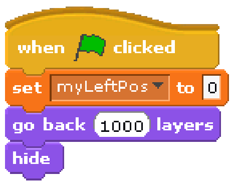
Using this Boolean expression, we can make each of the background pieces decide whether to display itself or not. If the expression is true, then the piece is invisible, so all it has to do is make sure it's not visible by using the "hide" block. What if it is visible? In that case we need to decide how to position the block, and make sure it is visible. Recall that we set the "center" of each piece to be the lower left hand corner of the piece, so we need to figure out what the coordinates of the lower left hand corner are, relative to the stage coordinates. The x coordinate of the lower left corner of the piece is offset by myLeftPos-BGShift from the stage lower-left corner, which is at position (-240,-180). Therefore, the coordinates of the lower-left corner of the piece are ((myLeftPos-BGShift)-240,-180), and we can use this in a "goto x y" block.

Let's put all these ideas together. When we receive a signal saying the background needs to be updated (we'll name this signal "UpdateBG"), we first test to see if the piece is visible. If it's not, we hide the piece; otherwise, we set the position as just described, and show the piece. Here's the entire script:



```
when I receive UpdateBG
if ((myLeftPos + 479) < BGShift) or (myLeftPos > (BGShift + 479))
    hide
else
    go to x: ((myLeftPos - BGShift) - 240) y: -180
    show
```

We also need a script that will initialize each piece - we'll set it up so that when the program starts (when the green flag is clicked), it sets its "myLeftPos" variable moves back in layers as far as possible (moving back 1000 layers should be far enough), and hides itself. Here is the initialization script for the first background piece - for the others, just change the "myLeftPos" variable to the correct value.



```
when clicked
set myLeftPos to 0
go back 1000 layers
hide
```

If these are the only scripts that start when the green flag is clicked, then nothing will be visible since all of the background pieces did a "hide." To get the initial background displayed, change the "hide" in the left-most background piece to "show".

Combining Projects

Almost all major software development is done by teams of developers, and not by a single person. Coordinating the work of a team is not easy, but it is a skill that's developed with lots of practice. Obviously, a team wouldn't work very well if there was only one copy of a program and one computer that could be used for development - while that was OK for the simple pairprogramming activities we've done in these labs, for large projects its best if everyone can work independently on a piece of the project and then combine their pieces into a single large program later. All "version control systems" (a concept mentioned Lab 1) provide support for merging programming done by different team members, and while BYOB doesn't support this to the same extent as a professional development environment there are certainly some things that can be done.

There are actually two main ways to merge work in BYOB. The most straightforward and most reliable way is to have each programmer's code isolated to one or a few sprites, and then those sprites can be imported into other projects. This is what we do below in Activity 4. The other way is to use the "Import Project" action in the "File" menu. This is a little trickier, and can cause problems if you import a project that duplicates global variable names or something similar. We don't explore the "Import Project" operation in this lab - if you are curious, try experimenting on your own and seeing how it works!

Activities (In-Lab Work)

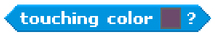
Activity 1: For the first activity, you are to implement the side-scrolling technique described in the pre-lab reading. The three maze pieces from that discussion are available on the class web site along with this lab write-up, so your first step is to download those to your computer. Next, you will need a sprite for each background piece, and each will need the scripts to initialize and update the display of that piece, as described in the pre-lab reading. Since these scripts are basically identical for all pieces, the easiest way to do this is to create a single sprite with all the necessary scripts, and then duplicate that sprite and replace the costume with the other maze background pieces. Specifically, first create a new sprite using the "maze1" piece as its costume, set the "center" of the costume to the lower left corner, create a sprite-local variable named "myLeftPos," a global variable named "BGShift," and finally build the two scripts from the pre-lab reading. Once you have one sprite created, you can right-click on it in the sprites pane and select "duplicate" to make a copy (do this twice). For the two new sprites, it is tempting to import the pieces as new costumes, but this is *not* to best way to do this since it will re-set the costume center. Instead, go into costumes, click "Edit," and then use the "Import" button within the costume editor. If you do this, it will replace the image used for the costume, but keep the "center" in the lower-left corner so you don't have to reset that!

Once you have the three background sprites set up, go back to the Alonzo sprite and add scripts that run when the left and right arrows are pressed that will shift the background in the appropriate direction. How do you do this? Think about the "BGShift" variable - update it when an arrow key is pressed and then broadcast the "UpdateBG" signal so that the background is drawn in the new, shifted position.

If you have done this correctly then you should be able to scroll the maze left and right with arrow keys. That's not very exciting right now, but hopefully you can see the potential! Once you've got this working, save it as Lab10-Activity1.

Activity 2: For this activity, you'll complete a basic maze game using the scrolling background created in Activity 1. You should still have the default Alonzo sprite in addition to your background sprites, but Alonzo is pretty big, so the first thing you should do is shrink Alonzo to 50% size. The maze was in fact created specifically for Alonzo at 50%, so make sure you use the correct size!

In Activity 1 you made it so the background scrolled in response to the left and right arrow keys. For this activity, you'll make it so Alonzo can only move in the white passages of the maze, and can move up and down as well as left and right. I'm not going to give you the scripts for these - you'll have to figure them out yourself! I'll give you a few tips though. Ideally, you'd like to ask "will this move collide with a maze wall?" before moving Alonzo, and then only move if there is no collision. Unfortunately, this is difficult to do, so instead we move Alonzo and then check if there was a collision - if so, then we immediately move Alonzo back to "undo" the collision. The computer will do this move/test/undo so quickly that you won't even see it happen, even though this seems like a strange way to do things. How do you tell if there was a collision? This is the block you want:



```
touching color ■ ?
```

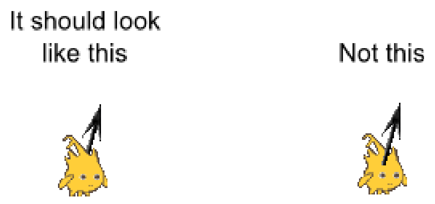
When you drag this out, you can click on the color square and the cursor will turn into an eyedropper. You can move the eyedropper to the stage, click on any red maze wall, and it will change the color for the test to red. Now you can tell if Alonzo is touching the maze wall!

Once you understand this, up and down movements are easy. Left and right are a little more difficult, since they must be coordinated with moving the background - and you don't want to move Alonzo to the right relative to the stage position, or he'll eventually run off the edge of the stage! Think this through, and consider this: move Alonzo, test for collision, shift the background appropriate, and then *always* move Alonzo back (collision or not). Do you see why that works?

Once you've written appropriate handlers for all four arrow keys, save your program with Alonzo appropriately positioned at the left-most maze background (in a clear spot) - then your program should move Alonzo through the maze, where walls are impassible! Once you have this working, save it as Lab10-Activity2.

Activity 3: This activity does not build on the previous two, so after you are sure that Activity 2 is saved you should start a new project. For this activity, you will create a program that shoots arrows, and in Activity 4 you'll merge this with another program that will turn this into a complete game. For the first activity, you are to set up Alonzo and an arrow sprite so that you can rotate the arrow to aim shots. Start with a new project that contains the basic Alonzo sprite, and add a second sprite - in the standard sprites, look in the "Things" category, use the "Clock-hand" sprite. This is the sprite with the costume center located at the tail, so that rotating around this center points the arrow from a fixed tail position. There is a script that is imported with this sprite - you don't need the script (and the "reset timer" block in the scripts pane) so you can drag these out to the blocks palette to get rid of them. You should make a script for each sprite that is triggered when the green flag is clicked, and each script will set its sprite up as follows. First, the size of each sprite must be set - the default size is too large, so we want to set Alonzo's size to

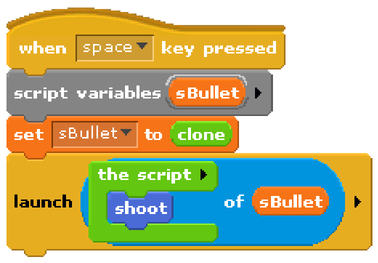
30% and the arrow's size should be set to 20%. The position for both sprites should be set at the center near the bottom of the stage, at location (0, -150). Set the initial direction of the arrow to some angle of your choosing, between -90 and 90. Finally, use the "go to front" block in the appropriate script so that the arrow is behind Alonzo where they overlap. In other words:



Now you need to add some scripts that rotate the arrow - you should rotate counterclockwise by 10 degrees when the left arrow is pressed and clockwise by 10 degrees when the right arrow is pressed. You should put in tests so that you never rotate beyond horizontal (in other words, the angle should always be between -90 and 90, inclusive).

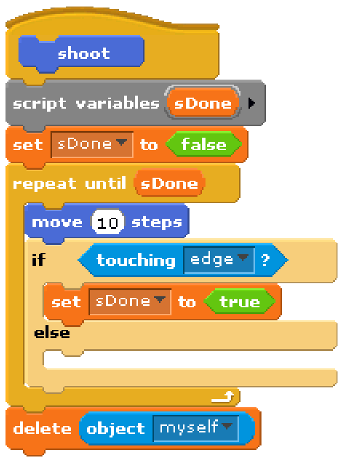
Next, you need to make it so Alonzo can shoot an arrow. In particular, when the spacebar is pressed your program should clone the arrow sprite and start it moving in the direction that the arrow is pointing. When the arrow reaches the edge of the stage it should destroy itself (in other words, the clone is deleted).

As a first step, create a script named "shoot" that will set up and operate the flying arrow after it is created - remember that we called this a "constructor" in Lab 7. The goal is to use a script something like this in the arrow sprite:



```
when space key pressed
script variables sBullet
set sBullet to clone
launch (the script (shoot) of sBullet)
```

So what should "shoot" do? This arrow should keep moving in the direction it is pointing until it reaches the edge of the stage, and then it should delete itself. Basically we want to move some number of steps (for example, 10 - this will control the speed of the arrow) inside a repeat loop. Looking ahead a little bit, there will eventually be several conditions that will cause the arrow to stop moving - eventually we'll have targets, and need to stop when the arrow hits a target. This can lead to a really complicated condition for the "repeat until …" loop, so we use another fairly common loop pattern: the **loop until done pattern**. This pattern uses a Boolean variable - a variable that only takes on values "true" or "false" - to indicate when we are done processing and should exit the loop. This is a script variable (so by our naming convention for script variables we will name this "sDone") which is initialized to "false", and inside the loop we can set the variable to "true" when we detect that we should exit the loop. Here is the basic pattern, including the "move 10 steps" block to move the arrow and a block at the end to delete this arrow after the loop exits (when we're done with it).



```
shoot
script variables sDone
set sDone to false
repeat until sDone
    move 10 steps
    if touching edge ?
        set sDone to true
    else
delete object myself
```

This *almost* works. To see what the problem is, build all these scripts - make sure you save your project before you start testing it, because there are some things that can get out of control very quickly if you're not careful, and you need a saved version to fall back to if necessary. Once you've saved everything, start your program by clicking the green flag. Then move all the way to the right, and start moving left shooting somewhere in the middle - but keep hitting the left arrow key after you shoot! What do you see?

You should see that the arrow curves - that's not what we want! Once an arrow is shot, it should stay on its course. The problem is that the arrow clone has the scripts that respond to left and right arrow keys just like the original arrow. We actually had a similar problem in the Pre-Lab reading example for Lab7 - if you don't remember how that worked, go back and look at the event handler for the "MakeNewCar" signal in the pre-lab reading. See how we checked the name of the sprite, and only executed that script if it was being run in the "CarProto" sprite? Do the same thing here, where your event handler for keypresses checks whether you are in the original arrow sprite (the "aiming sprite" that is located with Alonzo) and only changes the direction if it is that original sprite. Make this change and see if the arrow still curves when the arrow keys are pressed.

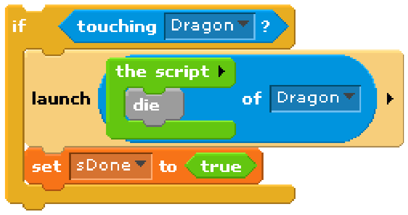
Finally, how does your script respond to rapidly firing? Start it, and press the space bar three times quickly while watching the sprites pane - how many arrows were created? You should have seen eight arrows! Why so many? It's the same problem as we just described with the curving arrows, and the way to correct this problem is exactly the same as before. Fix this so that if you hit spacebar three times quickly, only three new arrows are created.

Once all of this is working, save your work as Lab10-Activity3.

Activity 4: Finally, we want to finish out our game by having something to shoot at. The real point of this activity is to think about how a multi-person team might work on a game like this. Pretend that you have divided the tasks for creating this game: your job (which you did in Activity 3!) was to initialize the game and make it so that Alonzo could aim and shoot arrows. My job is to create a dragon that flies around randomly, and includes a "die" script that gets executed every time it gets hit by an arrow. I have created a fancy script that starts the dragon at a random edge of the stage (left, right, or top), and has it fly around randomly until it is hit and dies. After developing this and testing it enough where I'm confident that it works, I saved my project, then right-clicked on the dragon sprite and selected "export this sprite." This created a file "Dragon-Sprite.ysp" which is on the class web page - you should download this file to your computer.

After downloading my dragon sprite, you need to load it into your project. To do this, use the "Choose new sprite from file" button to load in my sprite. Now my sprite and my scripts are loaded along with yours! If you click the green flag, you'll see the dragon flying around randomly - you can still aim Alonzo's arrow and shoot arrows, but the arrows will fly right through the dragon because the arrow script doesn't know about the dragon yet.

Recall that the "shoot" script deletes a flying arrow when it hits the edge of the stage. We would now like to add the following code to the "shoot" block so that it tells the dragon to die, and deletes the arrow, when it hits the dragon:


```
if touching Dragon ?
    launch (the script [die] of Dragon)
    set sDone to true
```

What is the "die" script? It is a script that is local to the dragon, so was loaded in when you loaded the dragon sprite. However, there is something tricky here: we need to build the script above in the "shoot" block of the arrow, but the "die" block only exists for the dragon, so how can we pull it out and create this script? This is not obvious at all, but here's what you do: first select the dragon sprite, and the "Variables" category. See the "die" block at the bottom? Pull it out and drag it all the way over to the sprites pane, and drop it on the arrow sprite. Now select the arrow sprite, and you'll see the "die" block sitting there in the arrow's script area! Now you can complete the script construction above and put it in the "shoot" block of the arrow.

If you did that correctly, then you should have a complete game! Start it by clicking on the green arrow, and try shooting the dragon. When you hit the dragon 5 times, it will do a small animation (with the dragon breathing fire), and then then it will tell you how long it took you to kill the dragon. Note that you didn't have to do very much programming work for this part, because your partner (me!) wrote all of the dragon's scripts. When you work on team projects, try to plan out your work so that each person is responsible for just a few sprites, and figure out ahead of time how they will communicate with each other. Then each person can work on their own code, and all the parts can be combined together at the end! However, make sure you leave time for putting things together - most novice programmers seriously underestimate how long this will take. This is one of the challenges of working in a team!

Once you are finished with this, save it as Lab10-Activity4.

Submission

In this lab, you should have saved the following files: Lab10-Activity1, Lab10-Activity2, Lab10-Activity3, and Lab10-Activity4. Turn these in using whatever submission mechanism your school has set up for you.

Discussion (Post-Lab Follow-up)

There is no post-lab reading for this lab.

Terminology

The following new words and phrases were used in this lab:

- *costume center*: The one point on a sprite that specifies its placement (location) and serves as the point that it rotates around
- *layers*: A depth indication for sprites, so that the front/top sprite is shown in front of sprites in lower layers
- *loop until done pattern*: An iteration pattern in which a Boolean variable (almost always named “done”) controls when the loop should stop iterating