The Beauty and Joy of Computing

Lab Exercise 10: Putting the pieces together

Objectives

By completing this lab exercise, you should learn to

- Understand and work with "costume centers" in BYOB;
- Understand and work with layering of multiple sprites; and
- Combine multiple BYOB concepts and features to create a small but complete game.

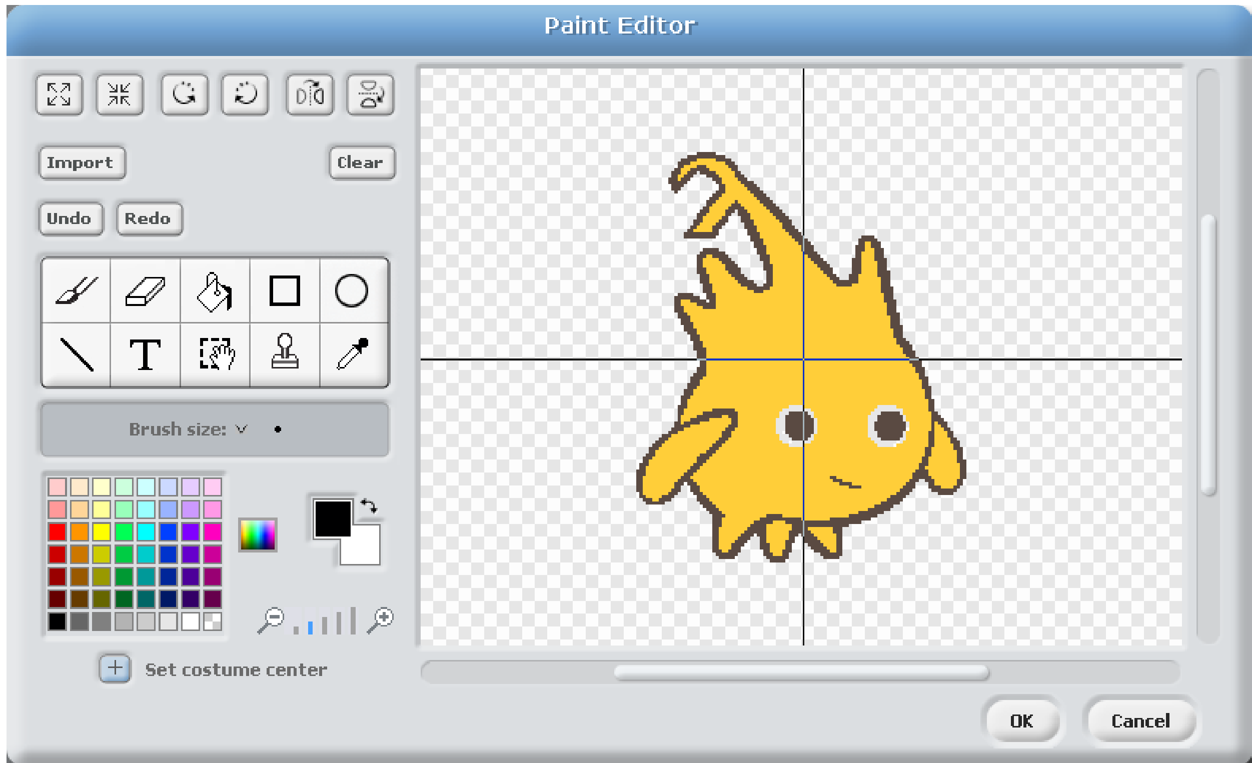
Background (Pre-Lab Reading)

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 wrap up the organized lab exercises by putting together several things you've learned about in previous labs to create a small game. While there are no major new concepts in this lab, we do look at a few new minor concepts with the way sprites are placed and rendered in this lab, which we describe next.

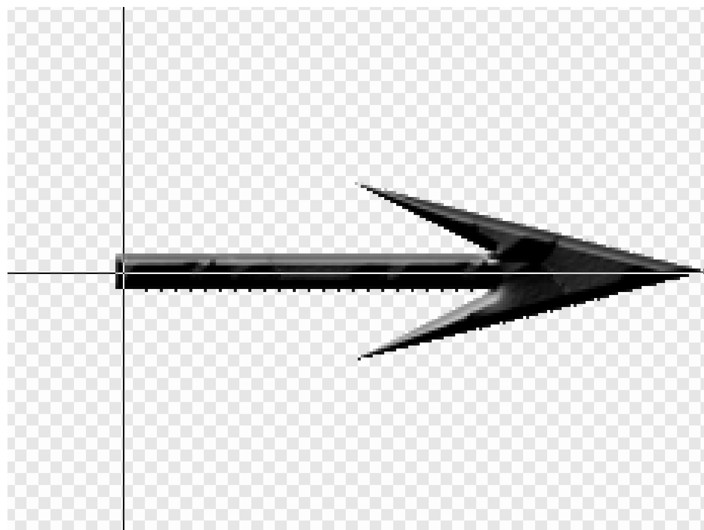
Sprites cover an area - they have a width and a height, and cover a range of x values and y values. For example, the basic Alonzo sprite is 104 pixels wide by 128 pixels high, so for a particular placement could cover x coordinates ranging from 16 to 119 and y coordinates from -24 to 103. So when we talk about "the location" of a sprite that is a single point, so where is it within this area? The center? The lower left corner? Something else? And related to this issue, when we rotate a sprite it rotates around a single point: which point?

The important point for a sprite is called the **costume center** in BYOB, although you should know that this is *not* always in the actual center of the sprite! Yes, that's a little confusing, but let's see how you can explore this concept and look at some examples. First, let's look at Alonzo. If you look at the "Costumes" pane in the Alonzo sprite, and click the "Edit" button next to the Alonzo costume, BYOB will open up the Paint Editor. In the lower-left corner of this window is a button labeled "Set costume center" - if you click on the button with the plus sign, the Paint Editor will draw crosshairs on the sprite costume that indicates where the center of the sprite is. Here's what this window looks like for Alonzo, with the crosshairs shown:

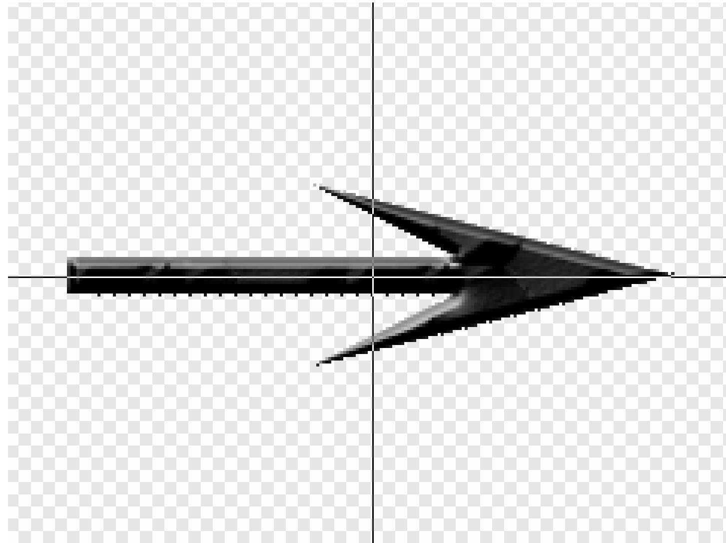


The point where the crosshairs meet is the center of the sprite. This is the point that is placed at specific (x,y) coordinates when the sprite is moved or placed somewhere, and it is the point around which rotations occur. For Alonzo this point is roughly in the middle of the sprite, so rotating Alonzo will make it look like he's spinning around his center.

For an example that illustrates how the costume center affects behavior of the sprite, consider a sprite that looks like an arrow. In some situations, we might want to rotate around the tail of the arrow, looking like a clock hand. In other cases, we might want to rotate such a sprite around its center, with rotation looking like a game spinner. In fact, there are two such sprites in the standard BYOB sprites: in the “Things” folder the two sprites are called “Clock-hand” and “Following Arrow”. The Clock-hand costume looks like this:



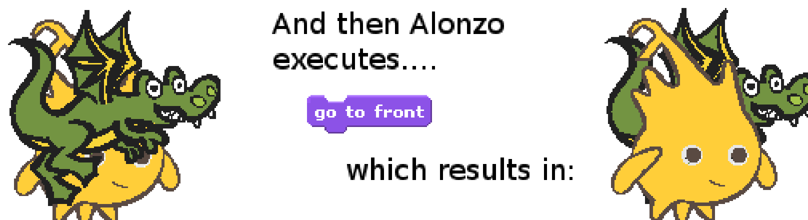
While the Following Arrow costume looks like this:



Note that the arrows are the same, but the marked “center” is different. Using these two different sprites will result in the two different behaviors described above.

The second minor concept to discuss also relates to the area on the stage that sprites cover: what if two sprites overlap in the same region? 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. In such a situation, how does 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 we use for this are the following two blocks in the “Looks” category:

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).

Activities (In-Lab Work)

The goal of this lab is to create a game in which Alonzo shoots arrows at sprites that pop up in random spots.

Activity 1: 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:

It should look like this

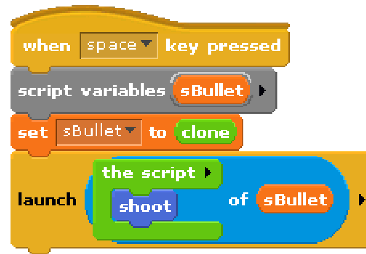
Not this

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.

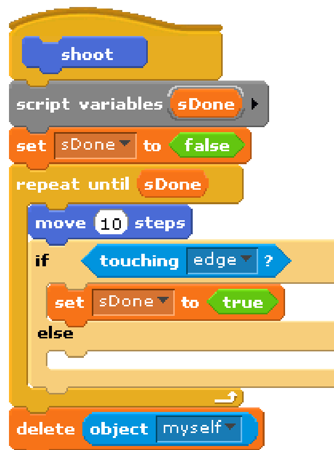
Once you’re happy that this works correctly, save it as Lab10-Activity1.

Activity 2: For this activity, you are to make it so Alonzo can shoot an arrow. In particular, when the space bar 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 8. The goal is to use a script something like this in the arrow sprite:



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).



This *almost* works. To see what the problem is, build all these scripts and start things 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?

Note that as you test this, if something stops the arrow before it reaches the edge of the stage and deletes itself, you’ll have to delete it yourself in the sprites pane. This isn’t a big deal, but make sure you keep the original sprite so that you don’t lose your code!

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. There are several ways we could fix this, but for this lab we’ll concentrate on one: the use of guard variables. A **guard variable** is a Boolean variable

that controls whether a set of operations should be executed or not. In our case, we will have a sprite local variable named “isOriginal” - remember that sprite local variables are actually separate variables for each clone instance. Start by creating this sprite-local variable for the arrow. Then initialize this variable to “true” in the script that is run when the green flag is clicked. The constructor for each clone (the “shoot” script) should set this variable to “false” so that even if there are 50 active clones, only the original will have the “isOriginal” variable set to true. Finally, each key-response script - for the left and right arrows as well as for space - should be “guarded” so that the the body of the script is run only if the “isOriginal” variable is true. You can just put the entire existing script inside an “if” block. After doing this, only the original arrow (the aiming arrow) will respond to keypress events. Your task in this activity is to add this code to your game script so that the flying arrows stay on the course that they start on.

Test your code carefully when you’ve got this finished. You should be able to shoot an entire stream of arrow by pressing space repeatedly. When you get this working, save it as Lab10-Activity2.

Activity 3: For this activity, we’ll add a target to shoot. For the initial target, use the dragon sprite and start by defining a block named “move to random place” that sets the x and y coordinate to random values where x is between -210 and 210 and y is between -110 and 140. It may not be clear why you want to define a block just to replace this one other block, but the reason should become clear soon. Next, give the dragon its own “green arrow clicked” script that sets it to 40% size, and then calls the “move to random place” block.

Next, we need to detect when the arrow hits the dragon, so select the arrow again - you need to modify the “shoot” script so by adding actions to the “else” part of the “shoot” script that had been left blank previously. This should check the following condition to trigger the “hit” actions:

Figure out how to make this work so that when the arrow hits the Dragon, the dragon very briefly says “Ouch” (say for 0.5 second), moves to a new random location, and finally sets sDone to “true” so that the loop will exit and delete the arrow clone. Moving the dragon to a new random location is perhaps a little tricky, but this should reveal why we defined the “move to random place” block. Think about this:

Once you get that working, save this version as Lab10-Activity3.

Activity 4: Finally, add a score and time limit. Specifically, create global variables named “Score” and “Time Remaining”, and create another script that is triggered by clicking the green arrow to initialize these variables: set the score to 0 and the time remaining to 15. The script should then enter a repeat loop that repeats until the time remaining reaches zero, and at each step it waits for 1 second and then changes the time remaining by -1. When this loop exits (the time remaining is 0) you should stop all scripts. Note that the arrow keys and space bar will still work even after time runs out, which isn’t a great idea. There’s a really simple way to disable

this - can you figure out what it is? In addition to the timing, you should increment the score in the “shoot” script every time the dragon is hit. Check the “Score” and “Time Remaining” variables so that they are watch variables, and position them on the stage in good locations. Now you’ve got a complete game! See how many times you can shoot the dragon before time runs out!

If you still have time after getting this done, consider enhancements that will count for extra credit. For example, you could have a different target (like the witch) that is worth more points. Or you could have “bonus” target that would increase the amount of time available to you. Finally (although this is really challenging!) you could make the targets move to new locations if they have been in the same location for some amount of time without being hit. Use your own creativity to figure out what enhancements you want to make.

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

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
- *guard variable*: A Boolean variable that controls whether a script (such as an event handler) is allowed to run
- *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

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.