The Beauty and Joy of Computing

Lab Exercise 10: Shall we play a game?

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);
- Be able to use object-oriented programming as supported by BYOB;
- 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)

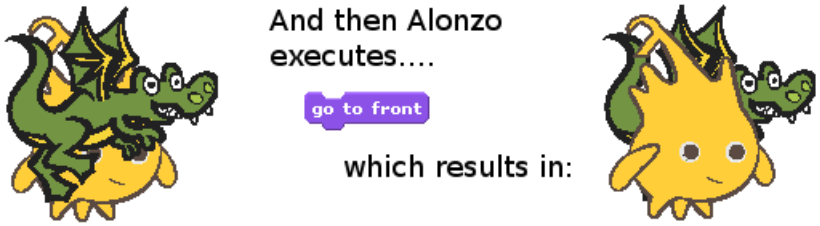
You should read this section before coming to the lab. It describes how various things work in BYOB and provides pictures to show what they look like in BYOB. You only need to read this material to get familiar with it. There is no need to actually do the actions described in this section, but you certainly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

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:

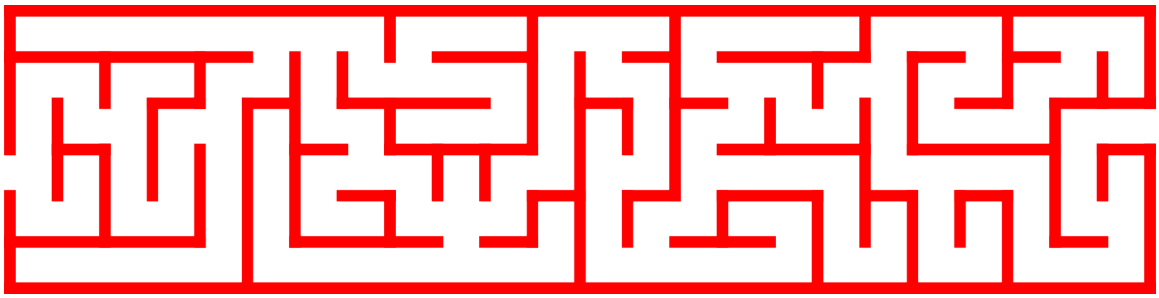


The “go back...” block does the opposite, but is more fine-grained since it can indicate a number of layers to send the sprite back, staying on top of some sprites while moving behind some others. To send a sprite to the lowest possible layer you can just use a large argument like 1000, which will max 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

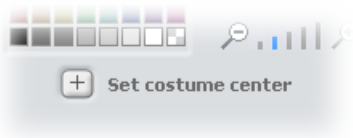
Most of you are probably familiar with side-scrolling video games, in which the background slides left and right behind a character as it progresses through the game. However, in BYOB backgrounds are fixed and cannot be moved, which creates a challenge. So while we think of this as a scrolling background 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:



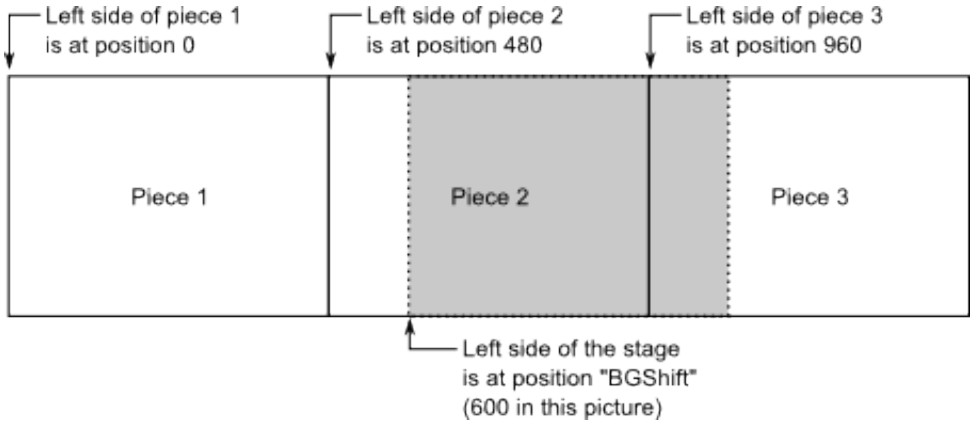
Next, we divide this up into three pieces, each 480 pixels wide, which will each be a different sprite. This brings up a small concept that we haven’t discussed: the unfortunately-named “**center**” of a sprite. When we say to put a sprite in a particular (x,y) location, what does that mean? Does the 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.

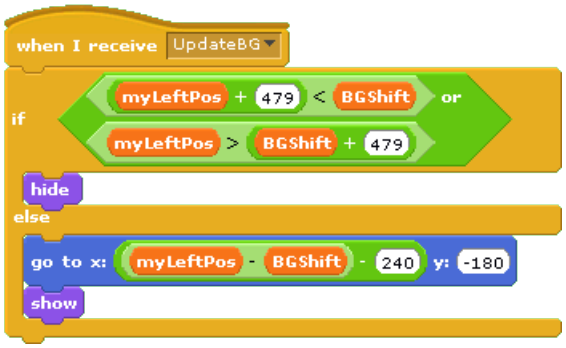


There are several ways to implement a side-scrolling background. The approach we describe is to have a script associated with each “piece” sprite that runs when it receives a signal and decides if and how to display itself. Lets use a sprite-local variable named “myLeftPos” to indicate the position of 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

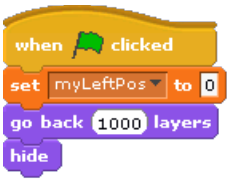
(myLeftPos+479 < BGShift) or (myLeftPos > BGShift+479).

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:



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.



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".

Object-Oriented Programming Concepts and Terminology

In the previous labs, we have worked with a small number of sprites, and every sprite was specifically created and programmed through scripts that operated on that sprite. Scripts can create interesting and complex actions for individual sprites, but we often want to have many sprites. Think about an interesting game: there are often tens or hundreds or even thousands of characters that you must keep track of. In addition to characters, we might also have other game items such as weapons, clothing, buildings, that we also need to track. We obviously

don't want to create a specific sprite for each entity in a game in which there are thousands of characters, and what saves us is the fact that many items share a similar structure and set of actions that they can perform. A style of program design that handles this very nicely is called **object-oriented programming**. While a full explanation of object-oriented design is beyond what we will do in this class, the basics are easy to understand.

Object-oriented programming is organized around **classes**, which are programming elements that share certain characteristics. Classes might represent things that you could potentially see, like characters in an animation, players or weapons in a game, or bigger program components such as backgrounds. However, classes can also represent more abstract things non-visible things like fractions or matrices or student lists. In a fighting game we might have a class for something as general as weapons, or for something slightly more specific like daggers which we might refer to as the "Dagger" class. **Objects** are specific instances of a class, so the dagger that your player is holding could be an instance of the Dagger class. If there is another player holding a different dagger, then that might be a different instance of the same general class. For another example, in BYOB consider that sprites are a class, and when you create a specific sprite (such as Alonzo) you have just created an instance of the sprite class. Note that you can create multiple sprites that look exactly alike - multiple Alonzos, for instance, but these are in fact separate objects.

It object-oriented programming, every object can contain attributes and methods:

- An **attribute** is a variable that is associated with one particular object. For example, an object from the "Dagger" class might have attributes such as weight, size, and sharpness. The collection of all attribute values of an object is referred to as the **state** of the object.
- A **method** is a function or a script that causes an object to perform some action. A method can access or change the attributes associate with this object, or it can perform some other action. For example, our Dagger class might have methods such as throw or stab, reflecting game play actions, or a draw method that draws it on the screen. A special kind of method called a **constructor** defines the actions required to initialize all of the attributes of an object - when an object is created, it creates variables for all of its attributes, and just like any variables these should be initialized!

BYOB sprites are good examples of objects: the attributes for a sprite object include things like its current position (x and y coordinates), direction, and whether it is visible, and the methods include blocks that operate on sprites such as the "move" and "glide" blocks. The correct way to think about these actions from an object-oriented standpoint is that they modify the state of the sprite object by changing the x and y coordinate attributes of the object.

There are several different styles of object-oriented programming - for example, while Java and JavaScript are common programming languages with similar names, they support distinctly different styles of object-oriented programming. BYOB supports a particular style of object-oriented programming called **prototype-based programming** in which new instances of objects are created by cloning an existing object (called the **prototype**), and this is the style of object-oriented programming that we will focus on in this class. There's obviously a lot more to object-oriented programming than has was just described - we'll talk about a few other issues in

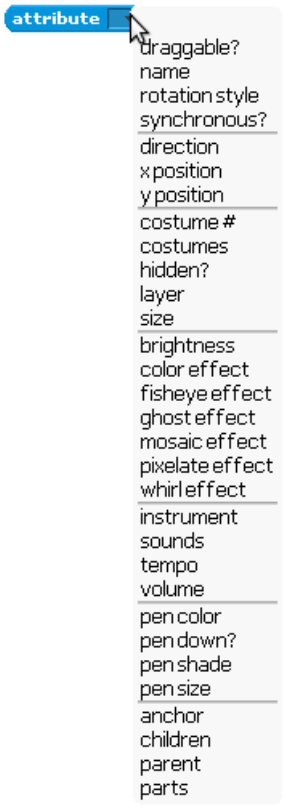
the Post-Lab Reading, but for now let's look at how the basics work by working through an example.

Working with Objects

In this section, we will go through a complete example showing how to set up a class with both attributes and methods, and create objects that are instances of that class by cloning a prototype object. For this example, let's make a silly animation: we want cars to drive around on the stage, bouncing against the edges. A new car is created every time a car runs over a magic hat that moves randomly around the stage, and there are two bombs on the stage that also move around randomly and any car that runs into a bomb is destroyed. Since the number of cars is controlled by luck (where the hat and bombs are randomly placed), we don't know how many cars we need, so we can't just create all the sprites manually at the beginning. Just because they are all cars and share some characteristics doesn't mean they will all look the same - there are a few different "styles" of cars, and the style of a car is randomly chosen when it is created.

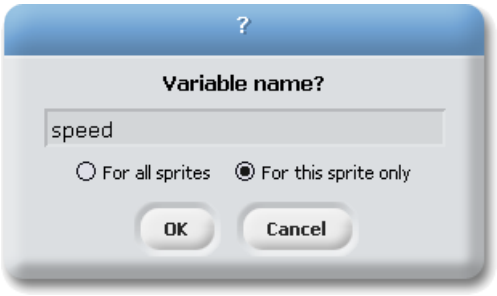
From an object-oriented programming standpoint, we want to define a class for cars, and then create a new car object when necessary. One car object is created when the animation starts, and a new car object is created every time an existing car hits the magic hat. Each car object will have all of the attributes that are common to all sprites (position, direction, etc.), and will have two attributes that are specific to our car definition: a speed and a style. As a first step, we create a new sprite with a car costume, and then import three other costumes in the costumes tab for the other styles. We'll change the name of this sprite to "CarProto" to indicate that it is the prototype for the car class. This will be our car prototype - we won't use the sprite at all other than to create new car objects. Since this particular sprite is never used in the animation, we click the "hide" button so that it's not shown on the stage.

Let's explore some BYOB operations that work on objects. In the "Sensing" category of the blocks palette, there is a reporter block at the bottom that is named "attribute". If you drag that out and click the drop down menu, you'll see this long list:



Those are all the “standard attributes” that are part of any sprite! You can see everything from the name of the sprite, to direction and x and y position coordinates, current costume number, and more. You’ve only seen some of these so far in this class - clearly there is still a lot to learn about how sprites work in BYOB!

Next, we create the attributes that are specific to the car prototype: Attributes are just variables, set to be “For this sprite only” as shown in this variable definition window:



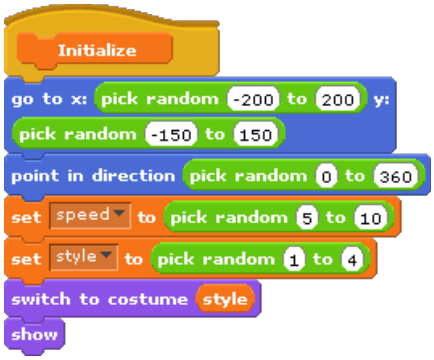
We define both “speed” and “style” in this way, and these variables show up in the blocks palette as follows:

Note that they are “under the line” since they are sprite local and not global variables.

Creating a new car object then involves two steps:

- First, the car prototype sprite runs a script that clones itself - this creates a new object that is a copy of the prototype, with its own copies of each attribute and sprite-specific scripts. For example, the prototype has a “speed” attribute, so the newly-created clone has its own “speed” attribute. The clone’s speed is initialized from the prototype’s speed, but once it is set up it is then independent: changing the prototype’s speed will not affect the clone’s speed and vice-versa. Understanding that each object within a class has its own independent set of variables (defining its state) is one of the most important concepts to understand when working with objects.
- Second, we need to initialize the newly-created car object. When we create a new object we generally don’t want all attributes to be exactly the same as its prototype, so we need to initialize them to make this object unique. This is the job of the constructor, the special method that was mentioned above. One very important thing to understand about the constructor is that we must run the constructor from the newly-created object. If you just called the constructor from the prototype and set attributes for “speed” or “style”, you would be setting the prototype’s attributes, not the new object’s attributes!

Let’s build the constructor first. For this application, we want to set up the car object’s attributes as follows: the constructor picks random values for the location coordinates, sets a random direction, picks a random speed between 5 and 10, picks a random style which we use to set the costume, and then “shows” the new sprite. Here’s what this block definition looks like - just like the attributes, this block should be defined “*For this sprite only*”:

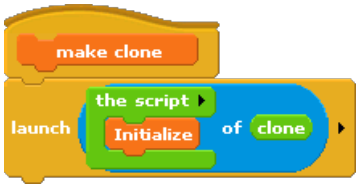


So now we have the constructor, which should be run in any new car object that is created, how do we actually create the new object and execute the constructor? At the bottom of the “Operators” category in the blocks palette is a very simple looking reporter block that’s simply named “clone”. Despite how plain this block looks, it is very powerful! It creates an complete

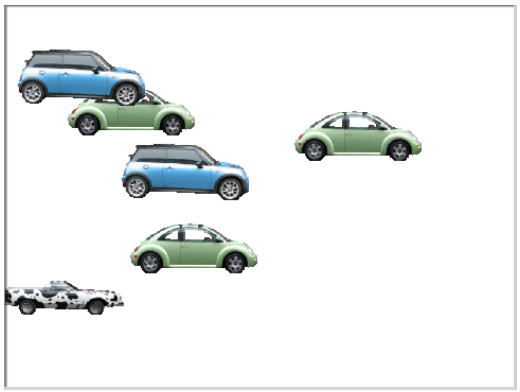
copy of the current sprite, and then reports the object that was created so that it can be used in further operations. The original object/sprite is called the **parent object**, and the newly created object is called the **child object**. What do we want to do with the new object? At this point, all we want to do is run its constructor, but remember that we need to run the constructor *from the new sprite*, while we are currently executing in the context of the prototype sprite. This is a little tricky, but we'll go through the reasoning, and then show you the code. To refer to either an attribute or a script that is in another sprite, we use the block that looks like this (it's in the sensing category):

The first parameter gives what we want to access, and the second one specifies the object that we want to use. In this case we want to run the "Initialize" script (that's the constructor) in an object that we create using "clone" - we use the special "the script" block in "Operators" to specify the script we want to run, so we can create a clone and refer to the Initialize script in that clone as follows:

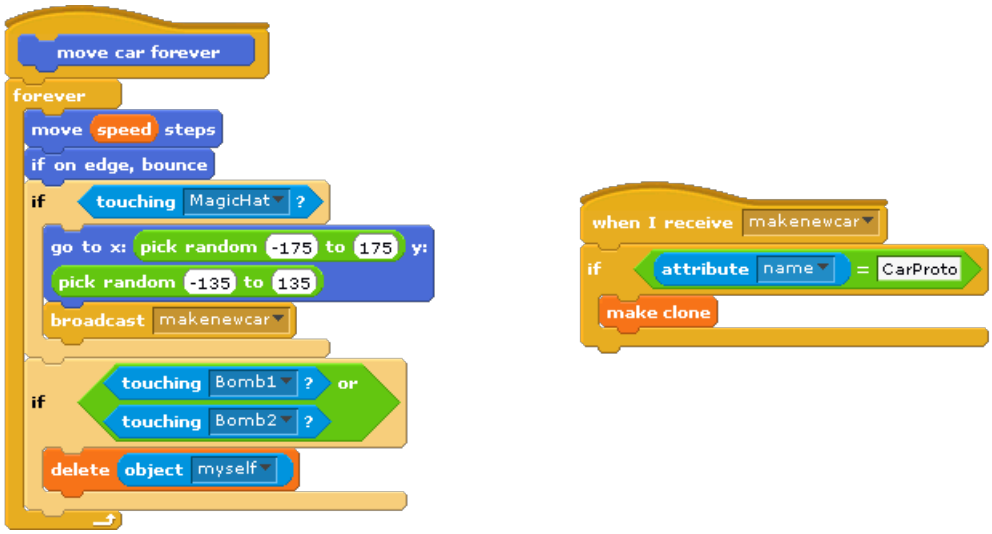
Now that we can refer to the script, we need to say what we want to do with this script, and the options are "launch" or "run", which are both in the "Control" category. The difference between these two options is whether we run the script and wait for it to complete (this is "run"), or whether we start the script in its own separate thread and let it run on its own, without waiting for it to finish (this is "launch"). This is a little messy, so let's put all these pieces together into a custom "make clone" block in the CarProto sprite that looks like this:



Once this is defined, we can drag the "make clone" block out in the "CarProto" sprite, and click it a few times - each click creates a new car object that is placed randomly on the stage, resulting in a stage that looks like this:



Finally, we need a car to start moving around when it is created. We first think about how movement works, and define a block (“For this sprite only”) named “move car forever” that does the correct movement based on the “speed” attribute of this sprite. It then checks for collisions - if it collides with the hat then it needs to create a new car, and if it collides with either of the bombs then it needs to delete itself. The first one of these is a little subtle - we want to make sure that it is the *prototype* object that clones itself (not the current sprite), and we need to ensure that only one car is created at a time. The solution is to broadcast a signal back to the CarProto sprite to tell it to create a new car. The resulting solution, showing both the “move car forever” definition and the event handler for CarProto, is shown below:



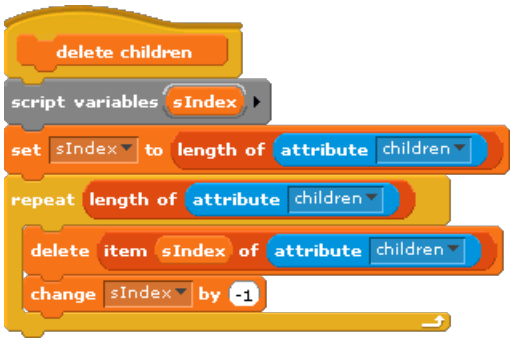
That is almost all we need! We need to launch the “move car forever” block from the car constructor, and then add some other game features such as random movement of the hat and bombs. These are fairly straightforward and don’t show anything new about object-oriented programming, so are not included here. You can download the complete solution from the class web site.

Cleaning Up After Yourself

If you start writing and experimenting with scripts that create new objects, you will reach a point where you have created several new objects, and you want to get rid of your objects to start your code over. Or worse yet, you run a script that starts creating objects in an out-of-control way, and you end up with 1000 new objects that you don’t want. How do you delete these objects? There’s no standard answer for this, but I would recommend that you create a global script called “delete children” that you can call from any prototype object to delete every object that was created from that prototype. To understand how to do this, look back at the long list of attributes that are displayed above in the “attribute …” reporter block - see the one that is named “children”? That attribute is actually a list of all children that were created by that prototype object (every time you executed the “clone” block from this prototype).

What we want to do is to walk through the children list and delete each child object that is on that list. There is one subtle problem here: if you delete the first object in the “children” list, it will delete the first element from the list, changing the position of everything that follows. To make

this even more confounding, this object does not disappear from the “children” list immediately - sometimes it will happen right away, and sometimes it will take a little time, so you don’t actually know if it has been removed from the list when you go to the next iteration of the loop. This is frustrating, but fortunately there is an easy fix: we delete objects starting at the end of the children list and working forward! Think about this: deleting an item in the list can only affect other items that follow it in the list, so by going from the end of the list to the beginning we avoid problems of objects changing positions in the list. If you didn’t understand that, take a chance to think it through and try to understand what the issue is. In the end, this is the block definition that we want:



Once this is defined, if you have accidentally (or on purpose!) created 10 car objects using the “make clone” script, if you click on the “delete children” block in the CarProto object you can watch all of the car objects (except the prototype!) in the sprites pane disappear one by one.

The concepts of object-oriented programming are difficult and challenging, and we have just scratched the surface. For this class if you follow the patterns given here carefully, you should be able to use these techniques reliably.

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 pair-programming 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!

Self-Assessment Questions

Use these questions to test your understanding of the Background reading. Answers can be found at the end of the lab exercise.

1. Why is it important that attributes for an object be created "For this sprite only"? For example, what would happen if, in our animation, the "speed" attribute for a car object was created "For all sprites"?
2. Consider defining a "Fraction" class, where each object is a fraction with an integer numerator and integer denominator. What are some attributes and methods that this class might contain? Note that there's no single right answer here - see what you can think of!
3. Describe a situation where a sprite is both a parent object and a child object.
4. Think back to the first lab with the "Alonzo and the Dragon" animation. Using the blocks introduced in the pre-lab reading, how could Alonzo make the Dragon disappear directly ("directly" means you can't have Alonzo broadcast a signal that the Dragon catches and then makes itself disappear).
5. We used the "children" attribute in the "delete children" definition, and the "children" attribute turned out to be a list of sprites created by this sprite. What do you suppose the "parent" attribute is? Is it a list?
6. (Warning: this is actually a somewhat tricky question - you should definitely think it through and try to answer it, but the correct answer might surprise you!) What if we start up BYOB and define the following event-handler for Alonzo:

   Next, we press the space bar 4 times. How many Alonzo sprite are there now?

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, and Alonzo just plows through walls, 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 28% size. The maze corridors ar just the right size for Alonzo at 28%, 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:

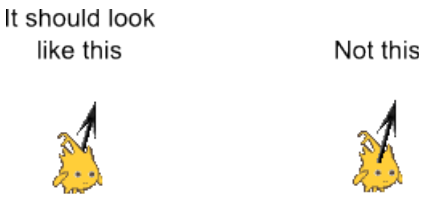
When you drag this out, you can click on the color square and the cursor will turn into an eyedropper. You can move the eyedropper to the stage, click on any red maze wall, and it will change the color for the test to red. Now you can tell if Alonzo is touching the maze wall!

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 if there is no collision, and then *always* move Alonzo back (collision or not). Do you see why that works? Be sure to use the algorithm described here - some other algorithms that seem like they

might do the same thing at first don't actually work. This one is tested and accurate, so just use it!

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.

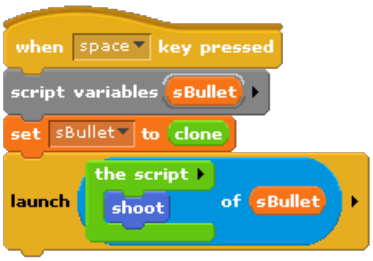
Activity 3: This activity does not build on the previous two, so after you are sure that Activity 2 is saved you should start a new project. For this activity, you will create a program that shoots arrows, and in Activity 4 you'll merge this with another program that will turn this into a complete game. 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:



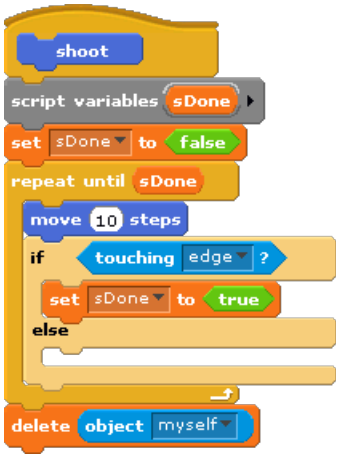
Now you need to add some scripts that rotate the arrow - you should rotate counterclockwise by 10 degrees when the left arrow is pressed and clockwise by 10 degrees when the right arrow is pressed. You should put in tests so that you never rotate beyond horizontal (in other words, the angle should always 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 - this is the "constructor" method that was describe in the Pre-Lab Reading. 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 - 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 dealt with a similar problem in the Pre-Lab Reading - 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.

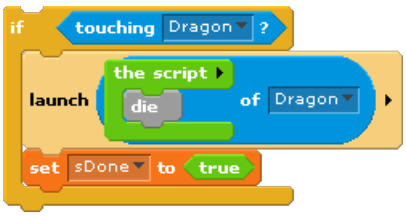
Finally, how does your script respond to rapidly firing? Start it, and press the space bar three times quickly while watching the sprites pane - how many arrows were created? You should have seen eight arrows! Why so many? It’s the same problem as we just described with the curving arrows, and the way 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:



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)

More on Object-Oriented Programming

One of the key features of object-oriented programming, which we didn't address above or in the lab activities, are the notions of sub-classes and inheritance. Classes are often specializations of other classes: for example, a game might have an "Enemy" class, but then there are times when you might want to be more specific and have a class for specific types of enemies, such as an "Alien" class, a "Zombie" class, or a "Dragon" class. All enemies might share certain attributes (like a health value), but there are also some attributes that are specific to a type of enemy - of these choices, only a dragon would have a "wingspan" attribute (OK, I just made an assumption that an alien wouldn't have wings, but seriously, has anyone ever seen an alien with wings?).

Using object-oriented programming, a situation such as this would be handled by defining an "Enemy" class, and then defining the others as what are called **sub-classes**, so for example the "Dragon" class would be a sub-class of the "Enemy" class. Going the other way, we say that the more general class (like "Enemy") is a **super-class** of the more specific class (like "Dragon"). A sub-class **inherits** all of the attributes of its super-class, which means that the sub-class will have attributes for everything defined in the super-class, without having to define them again. If we defined an "Enemy" class that has attributes "health" and "location", and then defined a "Dragon" sub-class where the definition only has a "wingspan" attribute, every dragon would still have its own health and location simply because it is a sub-class of "Enemy".

The terminology that is used in object-oriented programming is to refer to an "**IS-A**" relationship. To see if one class should be a sub-class of another, ask yourself if you can put the phrase "IS-A" between the two class names. For example, we can say "A Dragon IS-A Enemy" (not grammatically correct, but give me a little break here), so the Dragon class should be a sub-class of the Enemy class. This makes sense because if one thing "is" another, then it should have all the characteristics (attributes) and can perform the same actions (method) as the second thing. To consider another example, if you are creating an information system for a university, you might have classes for "Professor" and "Student"; however, since you can't put "IS-A" between those two class (in either order) it doesn't make sense for one to be a sub-class of the other. You could define a "Person" class and then both of these could be sub-classes of "Person" since a "Professor IS-A Person." We could even define another level of class, such as "Freshman" - then a "Freshman IS-A Student" and a "Student IS-A Person." This is another nice example of abstraction in computing: we can define a sequence of sub-classes that get more and more detailed and specific, and at each level we can (mostly) ignore the details that are hidden in the super-class.

We didn't explore this in the lab activities since BYOB does not provide a good mechanism for defining sub-classes - you can "fake it" in BYOB, but it's ugly and obscures what is really a very clean and elegant notion in more capable object-oriented languages. There is in fact one place where you see inheritance in BYOB: if you define your own class using a sprite, then it automatically includes all of the regular sprite attributes such as direction and position - these are really attributes that are inherited from the super-class "Sprite". Unfortunately, this is a special case in BYOB, and doesn't translate into a consistent method for defining sub-classes with inheritance.

Terminology

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

- *attributes*: variables that are associated with a specific object or sprite
- *child object*: an object that is created by cloning a prototype object - in cloning, the original object is known as the "parent object" and the new object is a "child object"
- *class*: a general class of object – individual objects are instances of some class, which defines the common characteristics of the class (weapons, monsters, etc.)
- *constructor*: a script that is run to initialize various attributes in a new object
- *costume center*: The one point on a sprite that specifies its placement (location) and serves as the point that it rotates around
- *inherit*: the process by which a sub-class includes the attributes and methods of its super-class
- *IS-A*: the relationship between a sub-class and its super-class - for example, a "Dagger" IS-A "Weapon"
- *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

- *methods*: scripts that are specific to a particular object or sprite, performing actions on or for that object
- *object-oriented programming*: a style of program design that is oriented around defining programmatic objects in terms of the attributes and methods (note: object-oriented programming has many other aspects that are beyond the scope of this lab!)
- *parent object*: an object that creates another object by cloning - in cloning, the original object is known as the "parent object" and the new object is a "child object"
- *prototype*: an object that is defined so as to serve as a template or model of objects from a particular class
- *prototype-based programming*: a style of object-oriented programming in which new objects are created by cloning a prototype object
- *sub-class*: A class that is a specialization of another class - for example, a "Dagger" is a special type of "Weapon", so a "Dagger" class might be a sub-class of a "Weapon" class
- *super-class*: The more general class of two classes related by an IS-A relationship - in the example given in the sub-class definition, the "Weapon" class is a super-class of the "Dagger" class

Answers to Pre-Lab Self-Assessment Questions

1. When you set a variable as "For this sprite only", every time you clone that sprite it clones the variable, creating a new "For this sprite only" variable for the new sprite/object. Therefore, if you create 5 new car objects, you have also created 5 new "speed" variables, one for each new car. If you didn't do this, then cars could not keep track of their own speed separately from the other cars. If you had defined "speed" as a "For all sprites" variable, every new car object would share the *same* global speed variable, so changing the speed for one car would change the speed for *all* cars!
2. The two obvious attributes are the numerator and denominator of the fraction. There are lots of choices for methods: you could have a method that reduces a fraction to lowest terms, stamps out the fraction, multiplies a fraction by an integer, or adds two fractions together. Basically anything you might want to do with a fraction defines an action on that fraction, and any of these actions could be methods in the Fraction class.
3. Consider the following situation: you start off with a sprite named Sprite1, and this sprite executes "clone" to create a new sprite named "Sprite2". Then this new sprite executes "clone" to create a new sprite named "Sprite3". In this situation, Sprite2 is a child of Sprite1 (since it was created by Sprite1), and Sprite2 is also the parent of Sprite3. In other words, Sprite2 is both a parent sprite and a child sprite.
4. The way we launched the constructor of the new sprite from the parent sprite works for any two sprites - they don't have to be parent and child! The following script, run by Alonzo, would make the Dragon disappear:

5. The parent attribute is actually just a single sprite, not a list. A sprite can have many children, because it can call “clone” many times, and so it needs to use a list for children. However, a sprite can have only one parent (the sprite that created it), so no list is needed and just a single sprite can be stored.
6. There would be 16 Alonzo sprites in the end. The key observation here is that when a sprite is cloned, everything is cloned, including scripts. Therefore, if the original Alonzo is named “Sprite1”, then when the spacebar is pressed it executes “clone” to create a new Alonzo named “Sprite2”. The new sprite has an event-handler for spacebar presses, just like the original Alonzo, so when the spacebar is pressed again both Sprite1 and Sprite2 handle this event and clone themselves, creating two new objects. Therefore, after 2 spacebar presses we have 4 Alonzo sprites. On the third spacebar press, all 4 Alonzos clone themselves, resulting in 8 Alonzos. And finally, on the 4th spacebar press these 8 Alonzos clone themselves, resulting in a total of 16 Alonzos. This is why you generally only want the prototype object to clone itself, not any of the child objects, which is why we included the “if” block in the “makenewcar” event handler in our example.