The Beauty and Joy of Computing[1]

Lab Exercise 2: Making things interesting with interaction, variables, and more

Objectives

By completing this lab exercise, you should learn to

- control sprite movement through a variety of mechanisms;
- write scripts that respond to user input and user actions;
- define and modify variables to track values in a program;
- use the sprite pen and stamp capability to draw pictures; and
- work with division and mod operators.

Background

In lab exercise 1, you learned how to make simple animations using BYOB, including how to get multiple sprites to coordinate and synchronize their actions. The end-product was an animation which could be viewed like a movie. What makes computers a powerful medium though, is the ability to interact with the user - for actions to change based on actions of the user. This two-way interaction is one of the important things that we look at in this lab exercise. As programs respond to users and change behavior based on user interaction, a program typically needs to keep track of certain values that control its actions, remembering user choices and responses to user actions. This is the fundamental concept of a program variable, which we introduce and explore in this lab exercise.

Activities

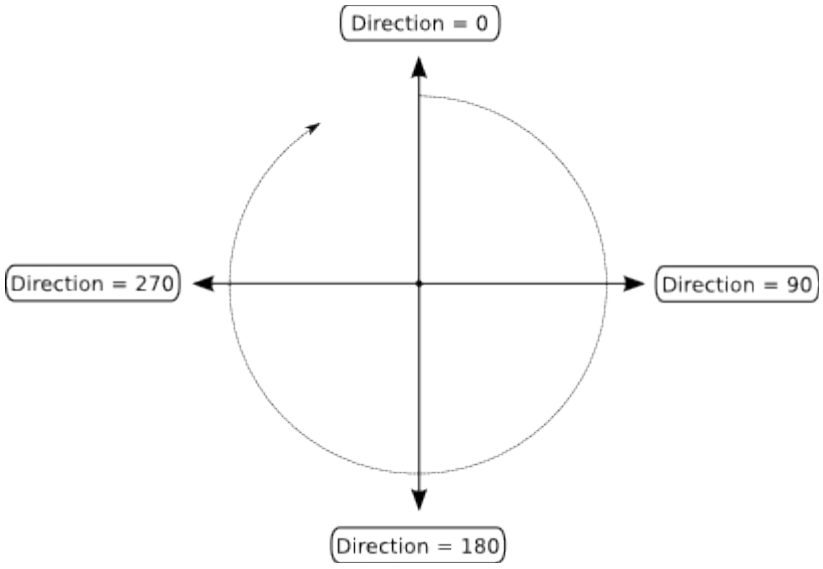
Since this is your second lab, the amount of things that you have to figure out for yourself is a little higher. The activity descriptions still lead you through things step-by-step, but there is more text and fewer pictures - you need to find the right pieces based on their descriptions, rather than just mimicking diagrams. This is the first step in "taking of the training wheels" - in future labs, there will be less of the step-by-step instructions, and you'll need to figure out even more on your own!

Activity 1: More on moving sprites

In Lab 1 we looked a little at sprite movement in order to make the dragon "hover." In this lab we will explore more fully how sprites move. Start a new project, so that Alonzo is the only sprite and is located in the center of the screen. Make sure the "Motion" category is selected in the blocks palette, and read through the different blocks to familiarize yourself with the possibilities. The first block ("Move x steps") is obviously a way to make a sprite move, but in what direction? You can click on this block in the palette, and see Alonzo move - which direction did he move?

Sprites have several properties, including x and y coordinates (which we used in the last lab) and a direction, which determines how the position changes when a “move” block is executed. Look down at the bottom of the “Motion” blocks in the blocks palette, and you’ll see three items with checkboxes, the last one saying “direction” - click on the checkbox to add a check and see what happens. In computing terminology, checking this box sets “direction” to be a **watch variable** - a variable that the programmer wants to observe while the program is running. Normally, watch variables are only enabled during **debugging**, when the programmer is testing the program and wants to gain insight into what is happening or possibly going wrong in the program.

So what does the “direction” mean? It’s an angle, that is based on the following coordinate system:



A direction set to 0 will result in the “move” block moving the sprite up, 90 will result in moving right, 180 will move down, etc. Increasing the direction value will adjust the angle in a clockwise direction, like the dashed line in the figure above. Only the main directions are shown in the diagram, but any value in between these can be chosen as well: a direction of 45 will go up and to the right, and 225 will go down and to the left. This also “wraps around” at 360 (because 360 degrees is a full rotation), so direction 450 is to the right just like direction 90 (since 90+360=450), and direction -90 goes to the left just like direction 270 (since -90+360=270). [*Note: If you have seen polar coordinates in a math class, these directions might unfortunately cause some confusion. In standard polar coordinates, direction 0 is to the right, and increasing this angle moves counter-clockwise. It’s unfortunate that BYOB uses non-standard directions, which were inherited from the Scratch system that BYOB is based on, but unfortunate or not it’s just something you have to learn to work with.*]

If you haven’t changed the direction from the initial setting when you started this project, when you set direction as a watch variable you should see that the direction is set to 90, and so when

you clicked the “move” block as instructed above, Alonzo should have moved a little to the right. Now you know why! Let’s make Alonzo move up instead: Drag out the “point in direction...” block, and then the “move … steps” block, which you should snap below the “point in direction...” block. Now click on the argument in the “point in direction...” block, and change the direction to 0 (i.e., up). Your blocks should look like this:

Now if you click on the two-block set you just created, you should see Alonzo move up (click more than once if you didn’t see him move the first time). Did anything else change on the stage other than the direction Alonzo moves?

To see what controls some other aspects of how sprites are drawn when they move, look at the top of the sprite info pane, and look specifically at the buttons to the left of Alonzo’s picture, which we call the **sprite orientation control buttons**:

Like control buttons in almost any computer application, you can hover the mouse over one of these buttons and a hint will pop up giving information about what the button does - try this with all three of these buttons. Once you see what all three descriptions are, try clicking them (these are called “**radio buttons**”, so only one can be enabled at a time). Try selecting different options, and clicking on your two-block move-up sequence for each one. You might not be able to tell the difference between the bottom two options now, but the difference will become clear in a few minutes.

If you want to take a little time to play around with the other movement blocks, such as “glide...” or “turn...” this would be a good time to do that. Just don’t get too distracted that you don’t have time to finish this lab!

Activity 2: Responding to the user

Making actions that respond to user actions is the key ingredient to making this animation more engaging than just watching a movie. Our first step along these lines is to make it so that sprite movements are controlled by the arrow keys. Starting from the two-block action defined in the previous activity, let’s trigger that movement by a keypress - in computing terminology, something that can trigger an action is called an **event**, and the code that responds to an event is called an **event handler**. In this activity, we are responding to a **keypress event**. There was an event in the previous lab as well - what was it?

To define an event handler, you find the start block associated with that event. In this case it is in the “Control” blocks, and is the one that is labeled “when … key pressed” - drag that piece out and snap it on top of your two blocks from the previous activity. By default, the key that you are responding to is the space bar - change that to “up arrow” to respond to the up arrow key. Next, we’d like to do the same things, but in different directions, for the other three arrow keys. Since the sequence of actions is the same in all cases, but with different arguments for the key and the direction parameters, we’ll speed up this task by duplicating the set of blocks you just made. If you right-click on the top block, you’ll get a pop-up menu in which you can select “duplicate” - do that, and you’ll see a new copy of that set of blocks that is ghosted out and moves with the mouse cursor so that you can place the new copy wherever you want. Do this three times so that you have a total of four event handlers, and set the keys and directions accordingly. After you do this, you should be able to press all four arrow keys and see Alonzo move around in the desired direction. This is an excellent time to experiment some more with the sprite orientation control buttons, and see how the sprite reacts in each of the three possible orientation control settings.

Now let’s add a new sprite that we can move independently, and we’ll make a little game of tag. Since we want another sprite that moves in a similar way to Alonzo, the easiest way is to duplicate Alonzo and then make the necessary changes. In the sprites pane, right click on the existing Alonzo sprite (“Sprite1” unless you changed the name - which would be a good idea!), and select “duplicate”. The next thing to do is to change the looks (costume) of the new sprite, and then change which keys control it. To change the looks, select the new sprite in the sprites pane, select the “Costumes” tab in the sprite info area, and then click “Import” to select a new costume - pick whichever one you like best! After the new costume is imported, scale it to an appropriate size, and delete the original costume (that one that looks like Alonzo). Finally, go back to the scripts pane and change the keys that trigger movement events so that ‘w’ moves up, ‘s’ moves right, ‘a’ moves left, and ‘z’ moves down. Once that is done try having two different people on the two sides of the keyboard, one using arrow keys to move Alonzo and one using w/s/a/z to move the new sprite. You’ve almost got a game!

The only thing left to do is to detect when the player who is “it” catches the other one. Let’s say that Alonzo is always “it”, to make things simple. To do this, we need to see another kind of event handler - one that iterates forever, but waits until a collision happens to do anything. Look in the “Control” category for a block that says “wait until …” - this will pause this script until some event occurs or condition is met, so we just need to find the condition. The one we want is under “Sensing” - it’s the one that indicates when the current sprite is touching another sprite. Drag that out and put it in the wait block, and select the appropriate argument for the “touching …” block. Finally, right after the wait block, let’s have Alonzo say “Got you!” for 1 second. Finish your game by placing this in a “forever” iteration block, capped off when a “when green flag clicked” which will act as a “Start Game” event.

Now you have a game of tag! I’m sure you’ll want to play for hours and hours, but for now save your game as “Lab2-Tag” and move on to the next activity!

Activity 3: Keeping track of changing values with variables

In the last activity, the position of sprites changed in response to the user pressing keys (keypress events). Sometimes we want sprites to move as time progresses, even if no obvious event occurs. Let's try something simple first: move Alonzo across the stage from left to right, under the control of our script. For this activity, it's best if Alonzo were a little smaller, and to do that we need to edit Alonzo's "costume." We edited a sprite's costume in the last Lab as well, when we flipped the dragon to face the other way - this time you should shrink the size of Alonzo, so hover over each of the buttons at the top of the sprite costume edit window to find the right one. Each time you click the "shrink" button the sprite drawing should get a little smaller. Three clicks should do it.[1]

We're going to be moving Alonzo around, so go ahead and select the "Motion" category in the blocks palette. Once you've made Alonzo the right size, move him to the lower left corner of the stage. You should make a note of Alonzo's position - this is his starting position before moving - so that you can move him back to the starting position while you experiment with different ways of moving. You can find his starting position at the top of the sprite info pane (where it gives x and y coordinates), but here's a simple and useful trick: Watch the "go to x: … y: ..." block while you double-click on Alonzo on the stage. What happened to the values in this block? Now you can drag this block out into the scripts area for Alonzo - just let it sit by itself for now, and think about how useful this is: You can move Alonzo all over the place now, and if you ever want to reset his position to this spot, just click on this block in the scripts pane and Alonzo will return to the location that was set when you dragged the block out! Let's call this the "position reset" block.

Now it's time to make Alonzo move across the screen. We want to repeatedly move Alonzo to the right, 10 steps at a time, but instead of using the "move … steps" block, use the "change x by …" block - drag it out to the scripts area, click it a few times to assure yourself that it does what you think it should, and then click on your position reset block to get Alonzo back to his starting position. To avoid clicking for each step, we need to repeat this action several times - or in terminology more commonly used in computing, we **iterate** this action (the general process of repeating actions is referred to as **iteration**). In the last lab we used the "forever" block to repeat operations, and in this lab we want to try a more refined approach than just doing something forever. The blocks to control iteration are in the "Control" category of the blocks palette - as a first try, use the "repeat …" block with the number inside (it should have the default value of 10 if you haven't changed it). When you drag this out, position it so that it "eats" your "change x by ..." block (so the change block will snap inside the repeat block). Now click the script that you just made - what happens?

Note: Right next to the "change x by …" block is a "set x to …" block. The difference between these blocks sometimes confuses beginning students, but the main thing to keep in mind is that "change … by …" always updates a value *relative to its current value* - for example, changing the x coordinate to be 10 greater than the current value. If you want to set a value to an absolute value (rather than relative), which doesn't depend on the current value, then you use

[1] This statement is a reference to a random movie/book quote - do you recognize it?

"set." There are also more involved situations where you will use "set", but for now just think of it as absolute vs. relative modifications of values.

You probably figured out that this repeat block repeats whatever is inside it 10 times, or however many times the number is set to. Try 20. Then 40. Experiment to find the right number of repetitions to move Alonzo all the way to the right without going off the stage. To make Alonzo start in the same position every time, snap your reset position block onto the top of this block by either moving the block or duplicating it and snapping in the duplicate. Alonzo moves pretty fast, right? There are two ways to slow him down - first, you could decrease the distance he moves in each step by decreasing 10 to 5. You can also put in a slight delay after each movement by dragging out the "wait … secs" block from the "Control" category, and snapping it directly under the "change x by …" block (make sure it's inside the "repeat" block!) - change the wait time from 1 second to 0.1 second unless you want Alonzo to move really, really slowly. Think about these two methods of slowing Alonzo down. Which one provides smoother motion? (Optional exploration: See if you can figure out how to use the "glide" block to make a really smooth motion.)

Our next step is to make Alonzo move vertically as well as horizontally, where gravity pulls him down toward the bottom of the stage - we could throw several sprites and call it "Angry Alonzos." Think a little bit about how gravity works: if you throw something straight up in the air, it steadily slows down until it reverses direction and comes back down, speeding up as it falls. One way we can slow things down is to reduce the distance it moves in each step - if we start moving 10 steps, then in the next move we might only go 9 steps, then 8, then 7, etc. We will change the y coordinate of Alonzo in exactly this manner, but to do this we can't just put a number in the "change y by …" block. In algebra we use letters to stand in for values that can change - for example, 2x+10, where x is a variable that can take on different values. We do the same thing in programming with **variables** - to see how to define and use variables in BYOB, select the "Variables" category in the blocks palette. Let's define variables for the x speed and y speed: Click "Make a variable" and when it prompts for a variable name, use "x speed" and check that it is "For this sprite only" (note that this would allow you to define different speed variables for each sprite in a complicated program) - recall that we introduced a term for this in that last lab: the variable is <u>local</u> to the sprite. Do the same thing for y speed, and unless you just want to watch the values of these variables, uncheck them after they are created.

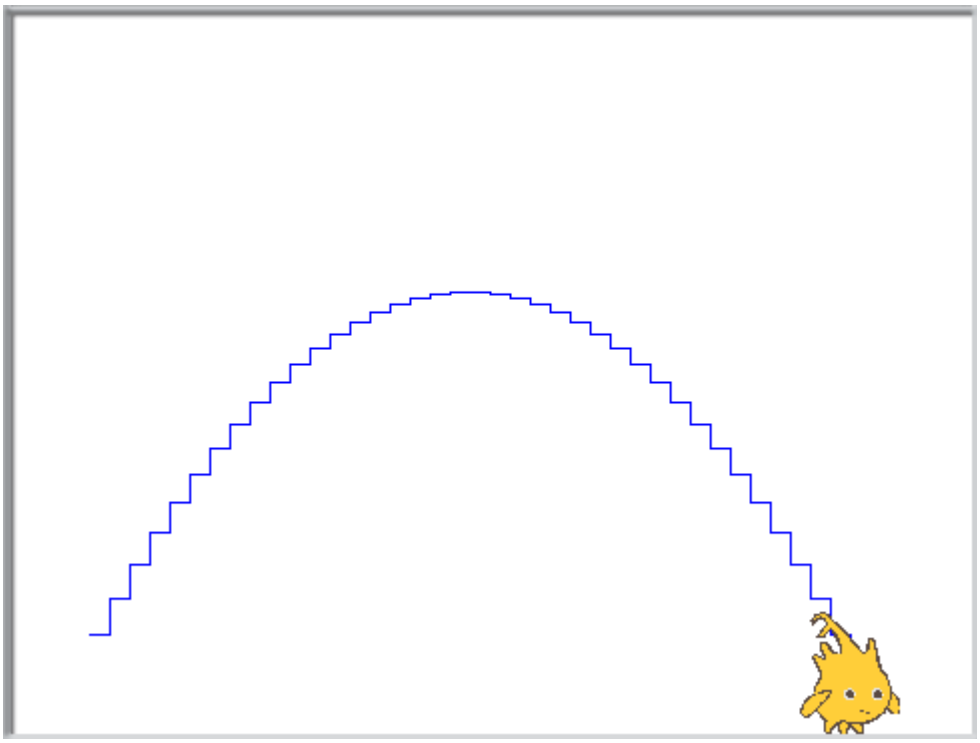
Now we'll use these variables in moving Alonzo - since they can change over time (the y speed will change in our example), we should start our code block by setting them to starting values - this is called **initializing** the variable. Going back to our code block for moving Alonzo, drag out a "set … to …" block, and put it at the top, either before your position reset block or after it. Click on the first argument and change it to "x speed" and make sure the second argument is 10, and then do the same thing to create a block that initializes the y speed variable. Now we will use those speed variables: You should still have a "change x by …" block, so drag out "x speed" from the Variables category, and snap it in place right over the number. Make a similar block to change y, and use the y speed. Finally, we want to change the y speed since gravity steadily slows things down - drag out a "change … by …" block and put it, inside the "repeat" loop but below the "change" blocks, and have it change the y speed by -1.

Your blocks should now look something like this:

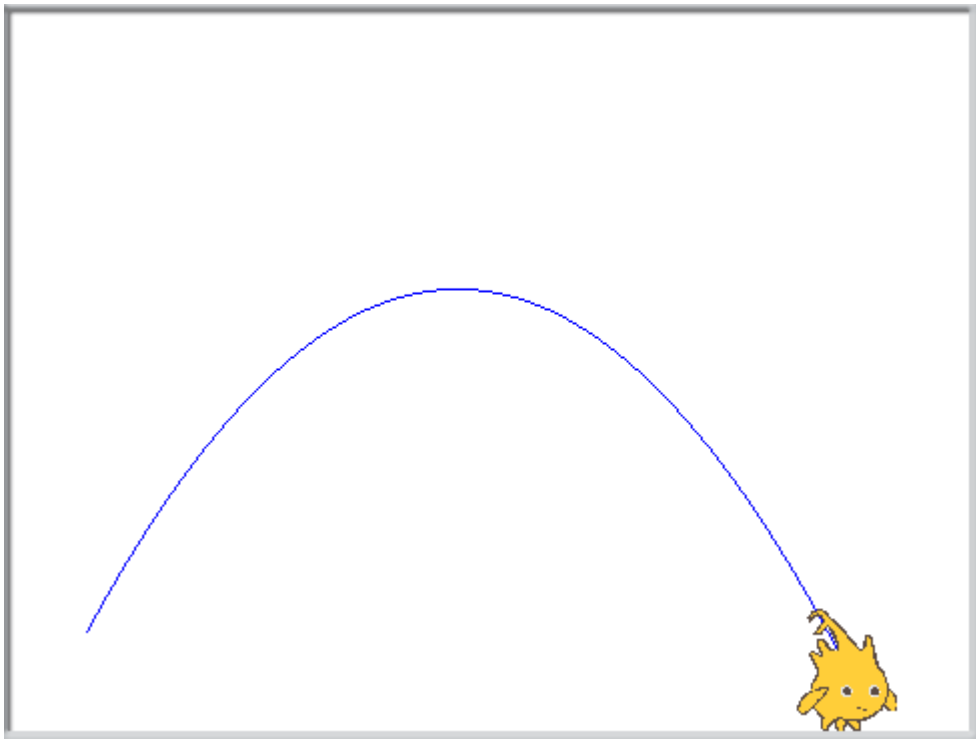
Try it! Does it look like a projectile flying through the air? Adjust the initial values for x speed and y speed to see how that changes things. Finally, the repetition count of 40 is just a value we put in by trial-and-error, trying to find something that worked out. In this situation, what's a better way to do this? Could we stop as soon as we hit the ground? For this, we repeat until the y coordinate gets below some level that we call the ground - since my Alonzo started out at y coordinate of -130, I'm going to call y coordinate of -135 the ground, and repeat until y is less than -135. Our first step is to replace the "repeat 40" block by the "repeat until …" block, and then we construct the following condtion for this repeat block:

Now play around with initial x and y speeds so that Alonzo makes a nice long arc from one side of the stage to the other. Alonzo is flying!

Finally, consider how smooth Alonzo's motion is - it looks a little jerky to me, but it would be good to see the exact path that Alonzo takes. Fortunately, BYOB makes this easy through the use of "pens." A pen can be associated with any sprite, so that when the sprite moves - if the pen is "down" - it will draw out a path of where it has been. To see how pens work, click on the "Pen" category in the block palette and think about what we want to do. First, we want to raise the pen ("pen up"), then clear any drawing it has made ("clear"), and then after moving to the starting position we want to put the pen down ("pen down"). Place these three pen control blocks in appropriate places in your Alonzo movement block, and shoot him across the screen again. Now you should see something roughly like this on the stage:



See how bumpy Alonzo's motion is? Think about why this might be, and how you could correct it. The final thing you will do in this activity is to change the way Alonzo moves to make it more smooth - look through the Motion blocks and see if you can figure out how to make the movement smoother. There is a very simple solution that will result in the following picture:



See if you can figure out how to do this (hint: in addition to a different motion block, you might also consider using other "Operators" blocks, such as + for addition).

Save your best solution as “Lab2-Flying”.

Activity 4: Displaying numbers

In this activity, we will explore some additional ways to draw, like the pen commands in the previous activity. Here’s the problem we want to solve: We want to display a number (for now, a two-digit number) using drawing commands - in other words, without using “say” or “think” or setting a watch variable. A warning ahead of time: you would think that, since computers are good with numbers, this would be easy to do - it’s not! At the end of this activity you’ll end up with the longest script that you’ve created so far, and it will be a bit ugly. To make you feel a little better about this, you should know that there are ways to clean this up and make it nicer and more concise - and in fact, we’ll continue with this example next week to introduce some of the concepts you need to make a more elegant solution.

Our first step is to figure out how to draw numbers on the screen. Bring up a new, blank project (with Alonzo sitting in the middle of the screen), and select the “Pen” category in the blocks palette. Click on the “stamp” block, and then move Alonzo around on the stage. What happened? Do another “stamp” in a new position for Alonzo, and keep moving around. So now we can draw not based on movements, but based on images that are in sprites! You click on the “clear” block in the blocks palette to clear off your “stamps” once you’ve got the idea. You can either delete the Alonzo sprite now, or just move him out of the way - down to the bottom of the stage, for instance.

Given that we can stamp out pictures based on sprites, what if the sprite that we were using looked like a digit? Go to the sprites pane, and create a new sprite using a provided image - do you remember how to do this? You want the button where the “hint” that pops up says “Choose new sprite from file.” Look in the provided costumes under the “Letters” category (I know, I know, … we want numbers, not letters, but trust me on the category), and browse through the different letter styles to find one you like. For this lab, you can use any style you want - what looks good to you? Once you have decided on a style, click on the “0” number in that style to create the sprite. Now the boring part: we need to add “costumes” to this sprite for every other digit - select your new sprite (which should look like a zero!), go to the “Costumes” tab in the sprite info pane, and click “Import” to load in a second costume, the one that looks like a 1 (one). Repeat this for all the other digits - it’s important that you do them in order, so that in the end you have 10 costumes, looking like the digits 0 through 9, in that order. Play around a little bit with your new sprite - use “stamp” to leave an image, and use “switch to costume” to change which digit is displayed.

In the previous activity, you saw variables for values that were... well... variable. They changed (at least the “y speed” did!). Another use of variables is to isolate and give a name to a value that we want to use, even if that value doesn’t change. The purpose of doing this is to give the value a meaningful name, which will help someone looking at the program (including you!) understand what you are using certain values for. In most programming languages this would be called a **constant**, which is like a special kind of variable, but BYOB doesn’t have a

separate way for defining a constant - it's just a variable, and we'll be careful not to change it. In this case, let's use a variable for a digit that we want to display, which we set at the top of our script, and can change in order to test our script with different values. The real benefit for arranging our script this way will become more apparent in the next lab, but for now you'll just have to trust me. To get started, go to the "Variables" category in the block palette, and make a variable named "digit". Should this be "for all sprites" or "for this sprite only"? You may not have the experience to make a fully informed decision on this, but think through the possibilities, and make a choice. We'll discuss this further in class.

Now that you have a digit variable, drag out a block that will initialize it to some value that is 4 or greater, and our goal will be to "stamp" out this digit. The problem we have now is this: how can we select the right "costume" for our digit, based on the value of the digit? Take a look at costume changing blocks under the "Looks" category. Unless you're particularly brilliant, the solution to this problem isn't immediately clear - however, this is one of the things about computing that can be both very frustrating, since what you want to do isn't immediately available, and also very very cool, because you get to "think outside the box" and come up with creative solutions which aren't obvious. You can be a problem solving ninja! You have seen all the pieces you need to solve this problem in this lab: you have a block that can set the sprite to use a particular costume (like the first one), and you have a block that will advance to the "next costume", and you also have a block to repeat an action a certain number of times. See if you can put those ideas together so that you have a script that advances the costume to the correct digit - then try changing the value that "digit" is set to and see if it still works. (Hint: Try dragging out the oval representing the "digit" variable and putting it into various places, such as the parameter for the "repeat …" block - what do you think this does?) Test your script for several digits and make sure you consistently get the right answer - you could try all digit values, but at the very least make sure you test the extremes: Does it work for zero? Does it work for 9? **Software testing** is an important skill for developing high quality software, and entire courses can be taught on how to do software testing of complex systems - although for small scripts like we use in this class, common sense testing (as long as your "common sense" includes testing with both common and rare or extreme values) is good enough.

Let's move to two digits now: what if your "digit" value were actually a two digit number, like 47? What do you think your script would do? If you didn't try to guess at the answer, do that now - don't go on until you think you have a reasonable idea in your head of what should happen. And then try it! Remember - be curious - experiment - test things out. You can't break anything, so be bold! This process - guessing what should happen based on what you know, and then seeing if it agrees with what actually happens - is one of the most important critical reading skills you can develop. As you read and learn, you create a "mental model" of how things should work, and you should always be testing and refining that mental model as you read and learn more. If things don't work as you think they should, you need to reconsider your mental model - and this is a great learning experience!

What you should have noticed is that "next costume" wraps back to the first costume when you try to advance past the last one, and you keep cycling through all costumes. In this case it has a nice benefit for us, since it leaves the digit at the correct value for the right-most digit (if you used 47, you should have ended up with "7" being the current costume). However, that's a

wasteful way to find the last digit - math gives us the power to figure out that the last digit is 7 without having to advance through 47 costumes, and the math magic that we need is under "Operators" in the block called the "mod" operator. The mod operator is similar to a division, except instead of giving the quotient it gives the remainder. For example, dividing 23 by 9 we get a quotient of 2 and a remainder of 5, so the value of "23 mod 9" is 5. You can try out any block, including the "mod" block, directly in the blocks palette: enter 47 for the first argument and 10 for the second, then click the mod block and you'll see a quote bubble pop up with the answer. It should look like this:

Try this with other values in place of 47, including 3 or 4 digit values, and what do you see? The remainder (the "mod value") should always be the value of the rightmost digit. What if you used this mod operator in the "repeat" loop - can you make it display the correct "rightmost digit" without having to cycle through 47 values, like before? Modify your script so that even if a multi-digit value is provided as input, you will select the rightmost digit "costume" and then stamp it in the current position.

We're dealing with two-digit numbers now, and we want to output both digits. How do we get the left digit? Since "mod" provided the remainder after division by 10, you really would like the quotient part (which would be 4 when dividing 47 by 10) - however, there isn't an easy way to get the quotient in BYOB. What happens if you do the following (guess first, and then try it!)?

So now we have an opportunity for more creative problem solving! If only we had 40/10, the division would come out even and the result would be 4 - the leftmost digit. But to get 47 down to a multiple of 10, we'd need to know how much greater than a multiple of 10 this number was... But wait! We do know this - it's the remainder! So if we took our value, subtracted the remainder when divided by 10 (which would make it a multiple of 10) and then divided by 10, we'd have the leftmost digit! This is exactly what you can do, but requires putting formulas in as arguments to other formulas. See if you can make this work - it's more complex than the things you've worked on so far, but it's just building up pieces from things you've done before. It is really to your benefit to work this out on your own, but if you get really stuck, the answer is at the end of this lab handout. However, it really really (really!) is to your benefit not to "peek" and to use that *only* as a last resort.

Finally, now that you can compute the leftmost digit, let's stamp out a two-digit number. Here's a high-level description in words - you can put this into code from this description: Stamp out the rightmost digit (you already did that!), move the sprite left by the width of the sprite (for example, in the "Scratch" costumes, each digit has width 100), figure out the leftmost digit and stamp that out using the same code as before. Using the "duplicate" capability will allow you to duplicate your first loop for stamping out the leftmost digit, and save you some steps.

Hopefully you should have the basic **functionality** (how your script behaves) working correctly now, and you should spend a some of the time remaining in the lab session to refine your script so it works better. For example, do you really want to see the digit values scrolling by with each "next costume"? Maybe you could have them just appear with the right values - experiment with

the “show” and “hide” blocks to see if you can make this happen. What about the size of the digits? They’re pretty large right now, so you’ll need a way to scale this down if you want a more compact number being drawn - experiment with “set size to … %” and see how you can adjust the size.

Once you are happy with your script, save it as “Lab2-Digits”.

Discussion

Animation smoothness: In creating “Angry Alonzo,” there were several different ways to move the Alonzo sprite around the screen, and some were smoother than others. Since we simulate motion by cycling through a series of still images (or “frames”), there are two main things that control the smoothness of the motion: how consistent the motion is between frames, and how far things move between frames. Changing just the x coordinate between two frames and then just the y coordinate between the next two frames leads to jerky horizontal-then-vertical movement, giving a stairstep path (like the first path drawn by Alonzo in the activity you just did). Even with consistent motion, however, if a sprite moves too far between frames the motion will look jumpy, and to smooth this out you need to move a shorter distance between frames. For example, you could half the distance that the sprite moves so that it doesn’t “jump” so far between frames, but to do this while keeping the overall movement the same you will need to double the number of frames you display every second. This is called the “frame rate” and people who play video games can be a little fanatical about frame rate - they will spend lots of money on faster video cards, so that they will draw the frames at a higher frame rate, and hence produce a smoother game. When people benchmark (i.e., test the speed of) video cards, they will often do it by testing how high a frame rate can be achieved for certain games. For reference, movies in a theater typically run at 24 frames per second, while TV signals are around 30 frames per second in the United States. People who make movies also think about frame rate - according to the entertainment news, Peter Jackson is filming *The Hobbit* at 48 frames per second, and certain theaters (who buy new projection equipment) will play the movie at that frame rate - do you think you would notice the difference?

Modeling Physics: “Angry Alonzo” flew across the screen in what was (hopefully) a fairly realistic looking arc. There are a lot of situations where we want computations performed by a program to reflect actions as they would happen in the real, physical world. The mathematics that gets turned into code is called a “model,” and is a way to simulate an action in the real world. What we did in the activity in this lab is to create and use a very crude model of gravity: When an object is subject to gravity (at least an object that is not on a small, atomic scale), the vertical velocity changes at a steady rate. Specifically, the downward velocity increases by 9.8 meters/second (or m/s) every second. Therefore, if an object is dropped from a great height, it is released from a standstill (velocity 0 m/s) - one second later it is traveling down at 9.8 m/s, and two seconds after release it is traveling down at 19.6 m/s (or 2*9.8 m/s). Lots of other factors affect the velocity of an object, including air resistance, but this is a reasonable approximation for now. Since we change the velocity steadily in our “Angry Alonzo” activity, we are roughly approximating the effect of gravity on Alonzo’s vertical velocity. Of course, even if we used real gravity values, this wouldn’t be a great approximation: we “jump” the velocity from 9.8 to 19.6 rather than smoothly varying it, and we have Alonzo travel at 9.8 m/s for an entire

"time step" until increasing the velocity. Similar to our discussion of animation smoothness, we can increase the accuracy of our physics model by making the time step smaller: change the velocity every half second, so Alonzo travels at 9.8 m/s for half a second, 14.7 m/s for the next half second, and then finally increases to 19.6 m/s. If we keep decreasing the timestep, we approach a situation in which the velocity is continuously changing - this is what Calculus is all about, and using Calculus we could make even more accurate models! Physics models can get very complicated, and most people who write games use a software library that provides a ready-made physics model. For example, the Angry Birds game (and many others) uses a physics library called "Box 2D" - using a library that someone else has written, so you can use it in your program without having to worry about the details, is an example of "abstraction." Abstraction is one of the "big ideas of computing" that we will discuss much more over the next few days in this class, and is probably the single most important idea that enables the construction of complex and useful software systems.

Elegant Solutions: Programmers often talk about a solution or program being "elegant." The solution to displaying two digits in this lab exercise was definitely not elegant, which motivates the need for more powerful techniques that enable a more elegant solution. There is definitely a sense of aesthetics when it comes to programs, and people who are fluent in the basic language of computing recognize this and talk about "beautiful" or "elegant" code. Code that is wasteful, or repeats things unnecessarily is not elegant. Code that jumps all over the place so that it is hard to follow (sometimes called "spaghetti code") is not elegant. Code that uses an inefficient algorithm so that it runs slowly is not elegant. It's harder to explain what *is* elegant, but it's a lot like art - you know it when you see it, at least once you speak the basic language. Elegance in code comes from being concise, efficient, clear in purpose, and organized on the page or screen in a way that reflects good design and aesthetics. Elegance is such a central notion in computing that David Galernter, a computer science professor at Yale University, wrote a book on the subject entitled *Machine Beauty - Elegance and the Heart of Technology*, in which he wrote "Beauty is more important in computing than anywhere else in technology because software is so complicated. Beauty is the ultimate defense against complexity."

Terminology

- *constant*: A value that doesn't change as the program is run, and typically has a symbolic name to make the use of the constant more clear. For example, a programmer might define a constant named "days per week" with the value 7, so whenever this is used in the code the meaning of the value is clear.
- *debugging*: The process of observing code as it is running, typically in order to track down some part of the code that is not working correctly.
- *event*: Something that happens in a computer system that could potentially have a "response action" defined. Examples of events include things like the user pressing a key, an internal timer running out, a message being broadcast, etc.
- *event handler*: The script, or code, that is designed to run when a particular event occurs.
- *functionality*: How a program operates. In computing we can evaluate a program by several criteria, including functionality, robustness (how it responds to unexpected or erroneous inputs or events), and elegance.

- *initialize*: The act of giving a new variable a starting (initial) value.
- *iterate*: To repeat some action.
- *iteration*: The general idea of repeating an action.
- *keypress event*: An event that is generated when the user presses a key.
- *radio buttons*: Input elements in a program that typically look like buttons, where only one of the buttons can be selected or active any any point in time.
- *software testing*: The process of trying code with different inputs and combinations of events and inputs to see if it responds as it should.
- *sprite orientation control buttons*: In BYOB, these buttons - specific to each sprite and located in the sprite info pane whtn that sprite is selected - determine how a sprite responds to changing its direction. Does the picture of the sprite rotate to point in the direction that it is moving?
- *variable*: A symbolic name that can take on different values as a program is running, and can be changed by the program.
- *watch variable*: A variable that has been designated as a "variable of interest," typically when a programmer is debugging a program. Watch variables have their current value displayed while the program is running, so that the programmer can watch the variable as it changes.

Submission

In this lab, you should have saved the following files: Lab2-Tag, Lab2-Flying, and Lab2-Digits. Turn these in using whatever submission mechanism your school has set up for you.

Solution to Computing Leftmost Digit

Hopefully you were able to put the pieces to this together yourself, but since we haven't nested operator blocks inside other operator blocks yet, the solution is provided here in case students get really stuck: